MDAnalysis User Guide

Lily Wang, Irfan Alibay, Patricio Barletta, Cédric Bouysset, Jan Do

Dec 29, 2023

GETTING STARTED

MDAnalysis version: 2.8.0-dev0

Last updated: Dec 29, 2023

MDAnalysis (www.mdanalysis.org) is a Python toolkit to analyse molecular dynamics files and trajectories in *many popular formats*. MDAnalysis can write most of these formats, too, together with atom selections for use in *visualisation tools or other analysis programs*. It provides a fast framework for *complex analysis tasks*, as well as flexible tooling to construct your own analyses.

CHAPTER ONE

WHY MDANALYSIS?

The typical use case for MDAnalysis is to manipulate or analyse molecular dynamics trajectories. The library focuses on two key features:

- **Memory efficiency.** The size of trajectory data can quickly overwhelm the memory resources of your computer. MDAnalysis typically accesses your trajectory by only loading data for one frame at a time. This allows you to work with trajectories of any length without difficulty.
- **Flexibility.** MDAnalysis is constructed to be easily extensible. If an analysis method is not already available in MDAnalysis, you can write your own custom trajectory analysis with the building blocks provided. If you need to read in a custom file format, you can construct your own Reader or Parser that will automatically get picked up when MDAnalysis is constructing a Universe from files. You can create and add your own labels for atoms, residues, or segments (called *topology attributes*) and relationships between atoms (e.g. bonds, angles).

CHAPTER

TWO

PARTICIPATING

MDAnalysis welcomes all contributions from its users. There are many ways you can help improve MDAnalysis, from asking questions on the **`mdnalysis-discussion`_** mailing list, to raising issues on the Issue Tracker, to adding your own code. Please see *Contributing to MDAnalysis* for an introduction and guide to contributing to the code and documentation.

Important: Ground rules and expectations

The MDAnalysis community subscribes to a Code of Conduct. By participating in this project and community, you agree to abide by its terms. Please read it.

In general, we expect you to **be kind and thoughtful in your conversations around this project.** We all come from different backgrounds and projects, which means we will not always agree. Try to listen and understand why others hold their viewpoints in discussions. Rather than blaming each other, focus on helping to resolve issues and learning from mistakes.

2.1 Communications

Questions and discussions about MDAnalysis take place on GitHub Discussions and this repository's Issue Tracker. Anybody is welcome to join these conversations. Please ask questions about the usage of MDAnalysis on the **`mdnalysis-discussion`_** mailing list, and report problems on the Issue Tracker.

Wherever possible, do not take these conversations to private channels, including contacting the maintainers directly. Keeping communication public means everybody can benefit and learn from the conversation.

2.1.1 Installation

The latest versions of **MDAnalysis** can be installed using *conda* or *pip*. Currently, the conda releases only support serial calculations. If you plan to use the parallel OpenMP algorithms, you need to install MDAnalysis with pip and have a working OpenMP installation.

MDAnalysis has a separate *test suite* **MDAnalysisTests** that is required to run the test cases and examples. The test files change less frequently, take up around 90 MB of space, and are not needed for daily use of MDAnalysis. However, they are often used in examples, including many in this User Guide. If you are not interested in developing MDAnalysis or using the example files, you most likely don't need the tests. If you want to run examples in the User Guide, install the tests. The tests are distributed separately from the main package.

Note: If you are installing on Windows, you must have Microsoft Visual C++ 14.0 installed. If your installation fails with the error message:

> error: Microsoft Visual C++ 14.0 is required. Get it with "Build Tools for Visual Studio": https://visualstudio.microsoft.com/downloads/

Try installing Build Tools for Visual Studio from https://visualstudio.microsoft.com/downloads/ (scroll down to the Tools for Visual Studio section).

If you encounter any issues following these instructions, please ask for help on GitHub Discussions (Installation).

conda

If you use conda to manage your Python environment, we highly recommend creating a new environment for MDAnalysis. This will ensure that you have a clean installation of MDAnalysis and its dependencies, and will not interfere with other packages you may have installed. We further recommend that you install and use mamba, a faster drop-in replacement for conda.

```
conda create --name mdanalysis
conda activate mdanalysis
conda install -c conda-forge mamba
```

To install the latest stable version of MDAnalysis via conda, use the following command. This installs all dependencies needed for full analysis functionality (excluding external programs such as HOLE):

```
mamba install -c conda-forge mdanalysis
```

To upgrade:

```
mamba update mdanalysis
```

To install the tests:

```
mamba install -c conda-forge MDAnalysisTests
```

If you intend to use MDAnalysis in JupyterLab, you will have to install an extra package for the progress bar in analysis classes:

```
conda install -c conda-forge nodejs
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

pip

The following command will install or upgrade the latest stable version of MDAnalysis via pip, with core dependencies. This means that some packages required by specific analysis modules will not be installed.

```
pip install --upgrade MDAnalysis
```

If you need to install a fully-featured MDAnalysis, add the analysis tag. As with conda, this will not install external programs such as HOLE.

```
pip install --upgrade MDAnalysis[analysis]
```

To install/upgrade tests:

```
pip install --upgrade MDAnalysisTests
```

If you intend to use MDAnalysis in JupyterLab, you will have to install an extra package for the progress bar in analysis classes:

```
pip install nodejs
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

Development versions

To install development versions of MDAnalysis, you can compile it from source. In order to install from source, you will need `numpy` and `cython`. See *Creating a development environment* for instructions on how to create a full development environment.

```
git clone https://github.com/MDAnalysis/mdanalysis
cd mdanalysis
# assuming you have already installed required dependencies
pip install -e package/
```

And to install the test suite:

```
pip install -e testsuite/
```

Testing

The tests rely on the *pytest* and *numpy* packages, which must also be installed. Run tests with:

```
pytest --disable-pytest-warnings --pyargs MDAnalysisTests
```

All tests should pass (i.e. no FAIL, ERROR); SKIPPED or XFAIL are ok. If anything fails or gives an error, ask on GitHub Discussions or raise an issue.

Testing MDAnalysis can take a while, as there are quite a few tests. The plugin pytest-xdist can be used to run tests in parallel.

```
pip install pytest-xdist
pytest --disable-pytest-warnings --pyargs MDAnalysisTests --numprocesses 4
```

Custom compiler flags and optimised installations

You can pass any additional compiler flags for the C/C++ compiler using the `extra_cflags` variable in `setup.cfg`. This allows you to add any additional compiler options required for your architecture.

For example, `extra_cflags` can be used to tune your MDAnalysis installation for your current architecture using the *-march*, *-mtune*, *-mcpu* and related compiler flags. *Which* particular compiler flags to use depends on your CPU architecture. An example for an x86_64 machine would be to change the line in *setup.cfg* as follows:

```
- #extra_cflags =
+ extra_cflags = -march=native -mtune=native
```

Use of these flags can give a significant performance boost where the compiler can effectively autovectorise.

Be sure to use the recommended flags for your target architecture. For example, ARM platforms recommend using `-mcpu` *instead* of `-mcpu`, while PowerPC platforms prefer *both* `-mcpu` and `-mtune`.

Full dicussion of the these flags is available elsewhere (such as here in this wiki or in this ARM blog post) and a list of supported options should be provided by your compiler. The list for GCC is provided here.

> **Warning:** Use of these compiler options is considered **advanced** and may reduce the binary compatibility of MDAnalysis significantly, especially if using *-march*, making it usable only on a matching CPU architecture to the one it is compiled on. We **strongly** recommend that you run the test suite on your intended platform before proceeding with analysis.

In cases where you might encounter multiple CPU architectures (e.g. on a supercomputer where the login node and compute node have different architectures), you should avoid changing these options unless you are experienced with compiling software in these situations.

Additional datasets

MDAnalysisData is an additional package with datasets that can be used in example tutorials. You can install it with `conda` or `pip`:

```
# conda
conda install -c conda-forge mdanalysisdata
# pip
pip install --upgrade MDAnalysisData
```

This installation does not download all the datasets; instead, the datasets are cached when they are first downloaded using a Python command.

2.1.2 Quick start guide

MDAnalysis version: 0.18.0

Last updated: December 2022 with MDAnalysis 2.4.0

This guide is designed as a basic introduction to MDAnalysis to get you up and running. You can see more complex tasks in our Example notebooks. This page outlines how to:

- *load a molecular dynamics structure or trajectory*
- *work with AtomGroups, a central data structure in MDAnalysis*
- *work with a trajectory*
- *write out coordinates*
- *use the analysis algorithms in MDAnalysis*
- *correct and automated citation of MDAnalysis and algorithms*

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD, GRO, XTC

import warnings
# suppress some MDAnalysis warnings about PSF files
```

(continues on next page)

(continued from previous page)

```
warnings.filterwarnings('ignore')

from matplotlib import pyplot as plt

print(mda.Universe(PSF, DCD))
print("Using MDAnalysis version", mda.__version__)

%matplotlib inline
```

```
<Universe with 3341 atoms>
Using MDAnalysis version 2.6.0-dev0
```

This tutorial assumes that you already have MDAnalysis installed. Running the cell above should give something similar to:

```
<Universe with 3341 atoms>
2.6.0
```

If you get an error message, you need to install MDAnalysis. If your version is under `0.18.0`, you need to upgrade MDAnalysis. Instructions for both are here. After installing, restart this notebook.

Overview

MDAnalysis is a Python package that provides tools to access and analyse data in molecular dynamics trajectories. Several key data structures form the backbone of MDAnalysis.

- A molecular system consists of particles. A particle is represented as an `Atom` object, even if it is a coarse-grained bead.
- `Atom`s are grouped into AtomGroups. The `AtomGroup` is probably the most important class in MDAnalysis, as almost everything can be accessed through it. See *Working with atoms* below.
- A `Universe` contains all the particles in a molecular system in an `AtomGroup` accessible at the `.atoms` attribute, and combines it with a trajectory at `.trajectory`.

A fundamental concept in MDAnalysis is that at any one time, only one time frame of the trajectory is being accessed. The `trajectory` attribute of a `Universe` is usually a file reader. Think of the trajectory as a function $X(t)$ of the frame index t that only makes the data from this specific frame available. This structure is important because it allows MDAnalysis to work with trajectory files too large to fit into the computer's memory.

MDAnalysis stores trajectories using its *internal units*: **Å** (**ångström**) for **length** and **ps** (**picosecond**) for **time**, regardless of the original MD data format.

Loading a structure or trajectory

Working with MDAnalysis typically starts with loading data into a `Universe`, the central data structure in MDAnalysis. The user guide has a complete explanation of ways to create and manipulate a `Universe`.

The first arguments for creating a `Universe` are topology and trajectory files.

- A **topology file** is always required for loading data into a Universe. A topology file lists atoms, residues, and their connectivity. MDAnalysis accepts the PSF, PDB, CRD, and GRO formats.
- A topology file can then be followed by **any number of trajectory files**. A trajectory file contains a list of coordinates in the order defined in the topology. If no trajectory files are given, then only a structure is loaded.

If multiple trajectory files are given, the trajectories are concatenated in the given order. MDAnalysis accepts single frames (e.g. PDB, CRD, GRO) and timeseries data (e.g. DCD, XTC, TRR, XYZ).

[2]:
```
psf = mda.Universe(PSF)
print(psf)
print(hasattr(psf, 'trajectory'))
```

```
<Universe with 3341 atoms>
False
```

As PSF files don't contain any coordinate information and no trajectory file has been loaded, the `psf` universe does not contain a trajectory. If the topology file does contain coordinate information, a trajectory of 1 frame is created.

[3]:
```
gro = mda.Universe(GRO)
print(gro)
print(len(gro.trajectory))
```

```
<Universe with 47681 atoms>
1
```

For the remainder of this guide we will work with the universe `u`, created below. This is a simulation where the enzyme adenylate kinase samples a transition from a closed to an open conformation (Beckstein et al., 2009).

[4]:
```
u = mda.Universe(PSF, DCD)
print(u)
print(len(u.trajectory))
```

```
<Universe with 3341 atoms>
98
```

Note

The MDAnalysis test suite is packaged with a bunch of test files and trajectories, which are named after their file format. We are using these files throughout this guide for convenience. To analyse your own files, simply replace the `PSF` and `DCD` above with paths to your own files. For example:

```
structure_only = mda.Universe("my_pdb_file.pdb")
```

Working with groups of atoms

Most analysis requires creating and working with an `AtomGroup`, a collection of `Atoms`. For convenience, you can also work with chemically meaningful groups of `Atoms` such as a `Residue` or a `Segment`. These come with analogous containers to `AtomGroup`: `ResidueGroup` and `SegmentGroup`. For instance, the `.residues` attribute of a `Universe` returns a `ResidueGroup`.

[5]:
```
print(u.residues)
```

```
<ResidueGroup [<Residue MET, 1>, <Residue ARG, 2>, <Residue ILE, 3>, ..., <Residue ILE,␣
↪212>, <Residue LEU, 213>, <Residue GLY, 214>]>
```

Selecting atoms

The easiest way to access the particles of your `Universe` is with the `atoms` attribute:

```
[6]: u.atoms
```

```
[6]: <AtomGroup with 3341 atoms>
```

This returns an `AtomGroup`, which can be thought of as a list of `Atom` objects. Most analysis involves working with groups of atoms in `AtomGroups`. `AtomGroups` can easily be created by slicing another `AtomGroup`. For example, the below slice returns the last five atoms.

```
[7]: last_five = u.atoms[-5:]
     print(last_five)
```

```
<AtomGroup [<Atom 3337: HA1 of type 6 of resname GLY, resid 214 and segid 4AKE>, <Atom␣
↪3338: HA2 of type 6 of resname GLY, resid 214 and segid 4AKE>, <Atom 3339: C of type␣
↪32 of resname GLY, resid 214 and segid 4AKE>, <Atom 3340: OT1 of type 72 of resname␣
↪GLY, resid 214 and segid 4AKE>, <Atom 3341: OT2 of type 72 of resname GLY, resid 214␣
↪and segid 4AKE>]>
```

MDAnalysis supports fancy indexing: passing an array or list of indices to get a new AtomGroup with the atoms at those indices in the old AtomGroup.

```
[8]: print(last_five[[0, 3, -1, 1, 3, 0]])
```

```
<AtomGroup [<Atom 3337: HA1 of type 6 of resname GLY, resid 214 and segid 4AKE>, <Atom␣
↪3340: OT1 of type 72 of resname GLY, resid 214 and segid 4AKE>, <Atom 3341: OT2 of␣
↪type 72 of resname GLY, resid 214 and segid 4AKE>, <Atom 3338: HA2 of type 6 of␣
↪resname GLY, resid 214 and segid 4AKE>, <Atom 3340: OT1 of type 72 of resname GLY,␣
↪resid 214 and segid 4AKE>, <Atom 3337: HA1 of type 6 of resname GLY, resid 214 and␣
↪segid 4AKE>]>
```

MDAnalysis has also implemented a powerful atom selection language that is similar to existing languages in VMD, PyMol, and other packages. This is available with the `.select_atoms()` function of an `AtomGroup` or `Universe` instance:

```
[9]: print(u.select_atoms('resname ASP or resname GLU'))
```

```
<AtomGroup [<Atom 318: N of type 54 of resname GLU, resid 22 and segid 4AKE>, <Atom 319:␣
↪HN of type 1 of resname GLU, resid 22 and segid 4AKE>, <Atom 320: CA of type 22 of␣
↪resname GLU, resid 22 and segid 4AKE>, ..., <Atom 3271: OE2 of type 72 of resname GLU,␣
↪resid 210 and segid 4AKE>, <Atom 3272: C of type 20 of resname GLU, resid 210 and␣
↪segid 4AKE>, <Atom 3273: O of type 70 of resname GLU, resid 210 and segid 4AKE>]>
```

Numerical ranges can be written as `first-last` or `first:last` where the range is **inclusive**. Note that in slicing, the `last` index is not included.

```
[10]: print(u.select_atoms('resid 50-100').n_residues)
      print(u.residues[50:100].n_residues)
```

```
51
50
```

Selections can also be combined with boolean operators, and allow wildcards.

For example, the command below selects the C_α atoms of glutamic acid and histidine in the first 100 residues of the protein. Glutamic acid is typically named “GLU”, but histidine can be named “HIS”, “HSD”, or “HSE” depending on its protonation state and the force field used.

[11]:
```
u.select_atoms("(resname GLU or resname HS*) and name CA and (resid 1:100)")
```

[11]:
```
<AtomGroup with 6 atoms>
```

Note

An AtomGroup created from a selection is sorted and duplicate elements are removed. This is not true for an AtomGroup produced by slicing. Thus, slicing can be used when the order of atoms is crucial.

The user guide has a complete rundown of creating AtomGroups through indexing, selection language, and set methods.

Getting atom information from AtomGroups

An AtomGroup can tell you information about the atoms inside it with a number of convenient attributes.

[12]:
```
print(u.atoms[:20].names)
```

```
['N' 'HT1' 'HT2' 'HT3' 'CA' 'HA' 'CB' 'HB1' 'HB2' 'CG' 'HG1' 'HG2' 'SD'
 'CE' 'HE1' 'HE2' 'HE3' 'C' 'O' 'N']
```

[13]:
```
print(u.atoms[50:70].masses)
```

```
[ 1.008  1.008  1.008 12.011  1.008  1.008 12.011  1.008  1.008  1.008
 12.011 15.999 14.007  1.008 12.011  1.008 12.011  1.008 12.011  1.008]
```

It also knows which residues and segments the atoms belong to. The .residues and .segments return a ResidueGroup and SegmentGroup, respectively.

[14]:
```
print(u.atoms[:20].residues)
print(u.atoms[-20:].segments)
```

```
<ResidueGroup [<Residue MET, 1>, <Residue ARG, 2>]>
<SegmentGroup [<Segment 4AKE>]>
```

Note that there are no duplicates in the ResidueGroup and SegmentGroup above. To get residue attributes atom-wise, you can access them directly through AtomGroup.

[15]:
```
print(u.atoms[:20].resnames)
```

```
['MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'MET'
 'MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'MET' 'ARG']
```

You can group atoms together by topology attributes.

For example, to group atoms with the same residue name and atom name together:

[16]:
```
near_met = u.select_atoms('not resname MET and (around 2 resname MET)')
sorted(near_met.groupby(['resnames', 'names']))
```

[16]:
```
[('ALA', 'C'),
 ('ALA', 'HN'),
 ('ARG', 'N'),
 ('ASN', 'O'),
```

(continues on next page)

(continued from previous page)

```
 ('ASP', 'C'),
 ('ASP', 'N'),
 ('GLN', 'C'),
 ('GLU', 'N'),
 ('ILE', 'C'),
 ('LEU', 'N'),
 ('LYS', 'N'),
 ('THR', 'N')]
```

A complete list of topology attributes can be found in the user guide.

AtomGroup positions and methods

The `.positions` attribute is probably the most important information you can get from an `AtomGroup`: a `numpy.ndarray` of coordinates, with the shape `(n_atoms, 3)`.

[17]:

```
ca = u.select_atoms('resid 1-5 and name CA')
print(ca.positions)
print(ca.positions.shape)
```

```
[[11.664622    8.393473   -8.983231  ]
 [11.414839    5.4344215  -6.5134845 ]
 [ 8.959755    5.612923   -3.6132305 ]
 [ 8.290068    3.075991   -0.79665166]
 [ 5.011126    3.7638984   1.130355  ]]
(5, 3)
```

A number of other quantities have been defined for an AtomGroup, including:

- `.center_of_mass()`
- `.center_of_geometry()`
- `.total_mass()`
- `.total_charge()`
- `.radius_of_gyration()`
- `.bsphere()` (the bounding sphere of the selection)

See the user guide for a complete list and description of AtomGroup methods.

[18]:

```
print(ca.center_of_mass())
```

```
[ 9.06808195  5.25614133 -3.75524844]
```

Note

The `.center_of_mass()` function, like many of the analysis modules in MDAnalysis, relies on having accurate mass properties available. Particle masses may not always be available or accurate!

Currently, MDAnalysis assigns masses to particles based on their element or 'atom type', which is guessed from the particle name. If MDAnalysis guesses incorrectly (e.g. a calcium atom called CA is treated as a C_α), the mass of that atom will be inaccurate. If MDAnalysis has no idea what the particle is (e.g. coarse-grained beads), it will raise a warning, and give that particle a mass of 0.

To be certain that MDAnalysis is using the correct masses, you can set them manually.

MDAnalysis can also create topology geometries such as bonds, angles, dihedral angles, and improper angles from an `AtomGroup`. This `AtomGroup` has a special requirement: only the atoms involved in the geometry can be in the group. For example, an `AtomGroup` used to create a bond can only have 2 atoms in it; an `AtomGroup` used to create a dihedral or improper angle must have 4 atoms.

[19]:

```
nhh = u.atoms[:3]
print(nhh.names)
```

```
['N' 'HT1' 'HT2']
```

After a topology object such as an angle is created, the value of the angle (in degrees) can be calculated based on the positions of the atoms.

[20]:

```
angle_nhh = nhh.angle
print(f"N-H-H angle: {angle_nhh.value():.2f}")
```

```
N-H-H angle: 37.99
```

Note that the order of the atoms matters for angles, dihedrals, and impropers. The value returned for an angle is the angle between first and third atom, with the apex at the second. Fancy indexing is one way to get an ordered AtomGroup.

```
   3
  /
 /
2------1
```

[21]:

```
hnh = u.atoms[[1, 0, 2]]
print(hnh.names)
```

```
['HT1' 'N' 'HT2']
```

[22]:

```
angle_hnh = hnh.angle
print(f"N-H-H angle: {angle_hnh.value():.2f}")
```

```
N-H-H angle: 106.20
```

Working with trajectories

The trajectory of a Universe contains the changing coordinate information. The number of frames in a trajectory is its length:

[23]:

```
print(len(u.trajectory))
```

```
98
```

The standard way to assess the information of each frame in a trajectory is to iterate over it. When the timestep changes, the universe only contains information associated with that timestep.

[24]:

```
for ts in u.trajectory[:20]:
    time = u.trajectory.time
    rgyr = u.atoms.radius_of_gyration()
    print(f"Frame: {ts.frame:3d}, Time: {time:4.0f} ps, Rgyr: {rgyr:.4f} A")
```

```
Frame:   0, Time:    1 ps, Rgyr: 16.6690 A
Frame:   1, Time:    2 ps, Rgyr: 16.6732 A
Frame:   2, Time:    3 ps, Rgyr: 16.7315 A
Frame:   3, Time:    4 ps, Rgyr: 16.7223 A
Frame:   4, Time:    5 ps, Rgyr: 16.7440 A
Frame:   5, Time:    6 ps, Rgyr: 16.7185 A
Frame:   6, Time:    7 ps, Rgyr: 16.7741 A
Frame:   7, Time:    8 ps, Rgyr: 16.7764 A
Frame:   8, Time:    9 ps, Rgyr: 16.7894 A
Frame:   9, Time:   10 ps, Rgyr: 16.8289 A
Frame:  10, Time:   11 ps, Rgyr: 16.8521 A
Frame:  11, Time:   12 ps, Rgyr: 16.8549 A
Frame:  12, Time:   13 ps, Rgyr: 16.8723 A
Frame:  13, Time:   14 ps, Rgyr: 16.9108 A
Frame:  14, Time:   15 ps, Rgyr: 16.9494 A
Frame:  15, Time:   16 ps, Rgyr: 16.9810 A
Frame:  16, Time:   17 ps, Rgyr: 17.0033 A
Frame:  17, Time:   18 ps, Rgyr: 17.0196 A
Frame:  18, Time:   19 ps, Rgyr: 17.0784 A
Frame:  19, Time:   20 ps, Rgyr: 17.1265 A
```

After iteration, the trajectory 'resets' back to the first frame. Please see the user guide for more information.

[25]:

```
print(u.trajectory.frame)
```

```
0
```

You can set the timestep of the trajectory with the frame index:

[26]:

```
print(u.trajectory[10].frame)
```

```
10
```

This persists until the timestep is next changed.

[27]:

```
frame = u.trajectory.frame
time = u.trajectory.time
rgyr = u.atoms.radius_of_gyration()
print("Frame: {:3d}, Time: {:4.0f} ps, Rgyr: {:.4f} A".format(frame, time, rgyr))
```

```
Frame:  10, Time:   11 ps, Rgyr: 16.8521 A
```

Generally, trajectory analysis first collects frame-wise data in a list.

[28]:

```
rgyr = []
time = []
protein = u.select_atoms("protein")
for ts in u.trajectory:
    time.append(u.trajectory.time)
    rgyr.append(protein.radius_of_gyration())
```

This can then be converted into other data structures, such as a numpy array or a pandas DataFrame. It can be plotted (as below), or used for further analysis.

The following section requires the pandas package (installation: `conda install pandas` or `pip install pandas`) and matplotlib (installation: `conda install matplotlib` or `pip install matplotlib`)

[29]:

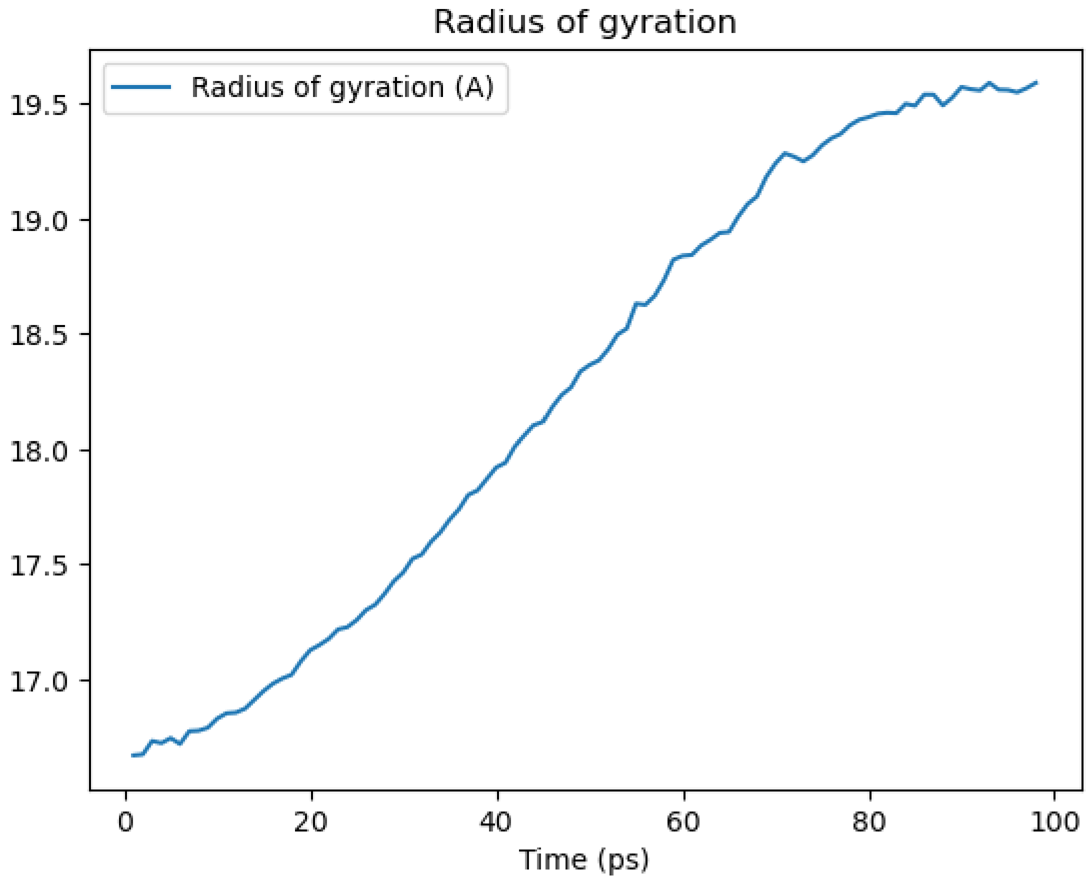
```
import pandas as pd
rgyr_df = pd.DataFrame(rgyr, columns=['Radius of gyration (A)'], index=time)
rgyr_df.index.name = 'Time (ps)'

rgyr_df.head()
```

[29]:

```
           Radius of gyration (A)
Time (ps)
1.0                     16.669018
2.0                     16.673217
3.0                     16.731454
4.0                     16.722283
5.0                     16.743961
```

[30]:

```
rgyr_df.plot(title='Radius of gyration')
plt.show()
```



Dynamic selection

As seen above, coordinates change while iterating over the trajectory. Therefore, properties calculated from the coordinates, such as the radius of gyration, also change.

Selections are often defined on static properties that do not change when moving through a trajectory. Above, the static selection is all the atoms that are in a protein. You can define the selection once and then recalculate the property of interest for each frame of the trajectory.

However, some selections contain distance-dependent queries (such as `around` or `point`, see selection keywords for more details). In this case, the selection should be updated for each time step using a dynamic selection by setting the keyword `updating=True`. This command gives an `UpdatingAtomGroup` rather than a static `AtomGroup`.

```
[31]: dynamic = u.select_atoms('around 2 resname ALA', updating=True)
      print(type(dynamic))
      dynamic
```

```
      <class 'MDAnalysis.core.groups.UpdatingAtomGroup'>
```

```
[31]: <AtomGroup with 54 atoms, with selection 'around 2 resname ALA' on the entire Universe.>
```

```
[32]: static = u.select_atoms('around 2 resname ALA')
      print(type(static))
      static
```

```
      <class 'MDAnalysis.core.groups.AtomGroup'>
```

```
[32]: <AtomGroup with 54 atoms>
```

When you call the next frame of the universe, the atoms in the dynamic selection are updated, whereas the atoms in the static selection remain the same.

```
[33]: u.trajectory.next()
      dynamic
```

```
[33]: <AtomGroup with 56 atoms, with selection 'around 2 resname ALA' on the entire Universe.>
```

```
[34]: static
```

```
[34]: <AtomGroup with 54 atoms>
```

Writing out coordinates

MDAnalysis supports writing data out into a range of file formats, including both single frame formats (e.g. PDB, GRO) and trajectory writers (e.g. XTC, DCD, and multi-frame PDB files). (You can also write selections out to other programs, such as VMD macros.)The user guide has a complete list of formats, each with their own reference pages.

Single frame

The most straightforward way to write to a file that can only hold a single frame is to use the `write()` method of any `AtomGroup`. MDAnalysis uses the file extension to determine the output file format. For instance, to only write out the C_α atoms to a file in GRO format:

```
ca = u.select_atoms('name CA')
ca.write('calphas.gro')
```

Trajectories

The standard way to write out trajectories is to:

1. Open a trajectory `Writer` and specify how many atoms a frame will contain
2. Iterate through the trajectory and write coordinates frame-by-frame with `Writer.write()`
3. If you do not use the context manager and the `with` statement below, you then need to close the trajectory with `.close()`.

For instance, to write out the C_α atoms to a trajectory in the XTC format:

```
ca = u.select_atoms('name CA')
with mda.Writer('calphas.xtc', ca.n_atoms) as w:
    for ts in u.trajectory:
        w.write(ca)
```

Analysis

MDAnalysis comes with a diverse set of analysis modules, and the building blocks to implement your own. (Please see a list of analysis tutorials in the user guide.)

The majority of these follow a common interface:

1. Initialise the analysis with a `Universe` and other required parameters.
2. Run the analysis with `.run()`. Optional arguments are the `start` frame index, `stop` frame index, `step` size, and toggling `verbose`. The default is to run analysis on the whole trajectory.
3. Results are stored within the class.
4. Often, a function is available to operate on single frames.

However, not all analysis uses this model. It is important to check the documentation for each analysis. You can also see examples in the Example gallery.

Below, simple RMSD analysis is shown. The `rms` module follows the interface above.

RMSD

Not all sub-modules of MDAnalysis are imported with `import MDAnalysis`. Most analysis modules have to be imported explicitly.

[35]:
```
from MDAnalysis.analysis import rms
```

MDAnalysis provides a `rmsd()` function for calculating the RMSD between two numpy arrays of coordinates.

[36]:
```
bb = u.select_atoms('backbone')

u.trajectory[0] # first frame
first = bb.positions

u.trajectory[-1] #last frame
last = bb.positions

rms.rmsd(first, last)
```

[36]:
```
6.852774844656239
```

An `RMSD` class is also provided to operate on trajectories.

Below, the `RMSD` class is created.

- The first argument is the `AtomGroup` or `Universe` for which the RMSD is calculated.
- As a reference `AtomGroup` or `Universe` is not given as the second argument, the default is to align to the current frame of the first argument. Here it is set to the first frame.
- We choose to align the trajectory over the backbone atoms, and then compute the RMSD for the backbone atoms (`select`).
- Then, without re-aligning the trajectory, the RMSD is also computed (`groupselections`) for the C_α atoms (`name CA`) and every protein atom (`protein`).

The RMSD is computed when we call `.run()`.

[37]:
```
u.trajectory[0] # set to first frame
rmsd_analysis = rms.RMSD(u, select='backbone', groupselections=['name CA', 'protein'])
rmsd_analysis.run()
```

[37]:
```
<MDAnalysis.analysis.rms.RMSD at 0x140451fd0>
```

The results are stored in the `results.rmsd` attribute. This is an array with the shape `(n_frames, 2 + n_selections)`.

[38]:
```
print(rmsd_analysis.results.rmsd.shape)
```

```
(98, 5)
```

We can interpret this as an array with 98 rows and 5 columns. Each row is the RMSD associated with a frame in the trajectory. The columns are as follows:

1. Frame number
2. Time (ps)
3. RMSD (backbone)
4. RMSD (C-alpha)

5. RMSD (protein)

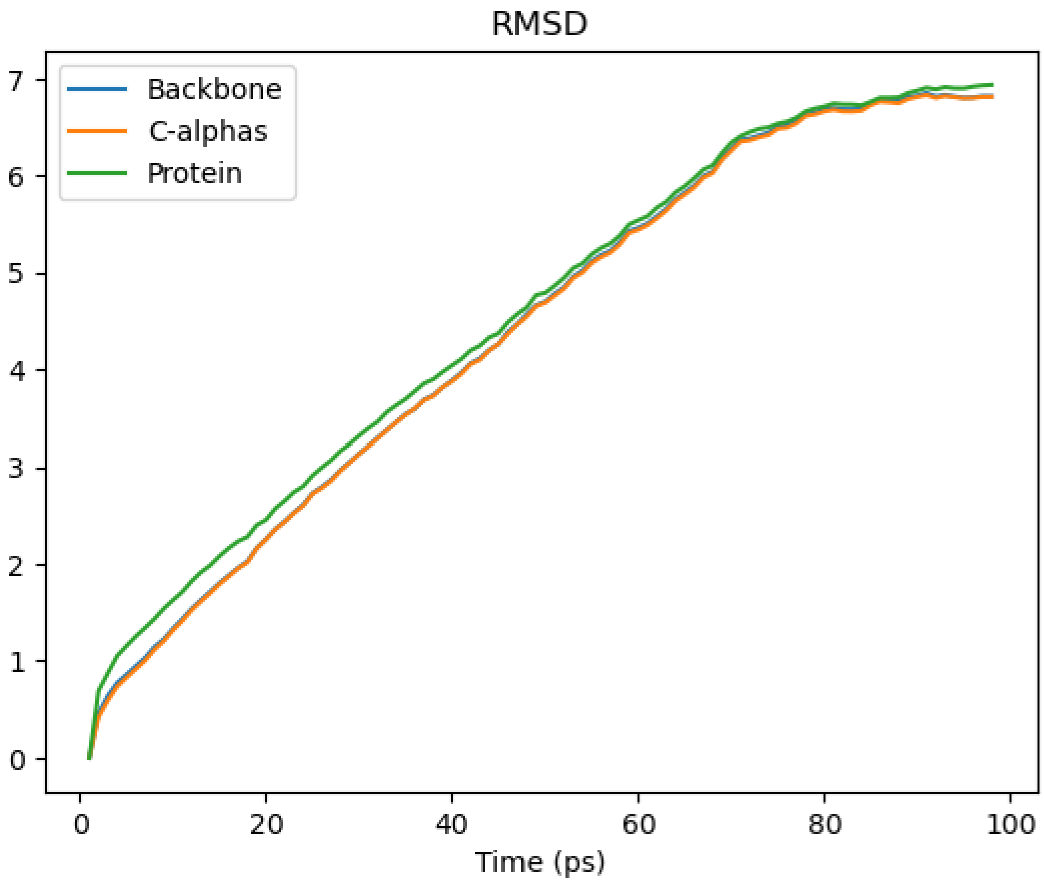
We can turn this into a pandas DataFrame and plot the results.

```
[39]: import pandas as pd

      rmsd_df = pd.DataFrame(rmsd_analysis.results.rmsd[:, 2:],
                             columns=['Backbone', 'C-alphas', 'Protein'],
                             index=rmsd_analysis.results.rmsd[:, 1])
      rmsd_df.index.name = 'Time (ps)'
      rmsd_df.head()
```

```
[39]:              Backbone      C-alphas       Protein
      Time (ps)
      1.0        5.834344e-07  4.263638e-08  5.443850e-08
      2.0        4.636592e-01  4.235205e-01  6.934167e-01
      3.0        6.419340e-01  5.939111e-01  8.748416e-01
      4.0        7.743983e-01  7.371346e-01  1.052780e+00
      5.0        8.588600e-01  8.279498e-01  1.154986e+00
```

```
[40]: rmsd_df.plot(title='RMSD')
```

```
[40]: <Axes: title={'center': 'RMSD'}, xlabel='Time (ps)'>
```



See the the RMSD and alignment notebooks for more information.

References

When using MDAnalysis in published work, please cite both these papers:

- N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein. MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations. *J. Comput. Chem.* **32** (2011), 2319–2327. doi:10.1002/jcc.21787
- R. J. Gowers, M. Linke, J. Barnoud, T. J. E. Reddy, M. N. Melo, S. L. Seyler, D. L. Dotson, J. Domanski, S. Buchoux, I. M. Kenney, and O. Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In S. Benthall and S. Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98-105, Austin, TX, 2016. SciPy. doi:10.25080/Majora-629e541a-00e

MDAnalysis includes many algorithms and modules that should also be individually cited. For example, if you use the `MDAnalysis.analysis.rms` or `MDAnalysis.analysis.align` modules, please cite:

- Douglas L. Theobald. Rapid calculation of RMSD using a quaternion-based characteristic polynomial. *Acta Crystallographica A* **61** (2005), 478-480.
- Pu Liu, Dmitris K. Agrafiotis, and Douglas L. Theobald. Fast determination of the optimal rotational matrix for macromolecular superpositions. *J. Comput. Chem.* **31** (2010), 1561–1563.

The primary sources of each module will be in their documentation.

Automatic citations with duecredit

Citations can be also be automatically generated using `duecredit`. Complete installation and usage instructions can be found in the user guide, but it is simple to generate a list of citations for your python script `my_script.py`.

```
$ python -m duecredit my_script.py
```

This extracts citations into a hidden file, which can then be exported to different formats. For example, to display them as BibTeX, use the command:

```
$ duecredit summary --format=bibtex
```

2.1.3 Frequently asked questions

Trajectories

Why do the atom positions change over trajectories?

A fundamental concept in MDAnalysis is that at any one time, only one time frame of the trajectory is being accessed. The `trajectory` attribute of a `Universe` is actually (usually) a file reader. Think of the trajectory as a function $X(t)$ of the frame index t that makes the data from this specific frame available. This structure is important because it allows MDAnalysis to work with trajectory files too large to fit into the computer's memory. See *Trajectories* for more information.

2.1.4 Examples

MDAnalysis maintains a collection of Jupyter notebooks as examples of what the code can do. Each notebook can be downloaded from GitHub to run on your own computer, or viewed as an online tutorial on the user guide. You can also interact with each notebook on Binder.

Constructing, modifying, and adding to a Universe

MDAnalysis version: 0.20.1

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Sometimes you may want to construct a Universe from scratch, or add attributes that are not read from a file. For example, you may want to group a Universe into chains, or create custom segments for protein domains.

In this tutorial we:

- create a Universe consisting of water molecules
- merge this with a protein Universe loaded from a file
- create custom segments labeling protein domains

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

[1]:

```python
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PDB_small
import numpy as np
from IPython.core.display import Image

import warnings
# suppress some MDAnalysis warnings when writing PDB files
warnings.filterwarnings('ignore')

print("Using MDAnalysis version", mda.__version__)

# Optionally, use NGLView to interactively view your trajectory
import nglview as nv
print("Using NGLView version", nv.__version__)
```

```
Using MDAnalysis version 2.6.0-dev0
```

```
Using NGLView version 3.0.3
```

Creating and populating a Universe with water

Creating a blank Universe

The `Universe.empty()` method creates a blank Universe. The `natoms` (int) argument must be included. Optional arguments are:

- n_residues (int): number of residues
- n_segments (int): number of segments
- atom_resindex (list): list of resindices for each atom
- residue_segindex (list): list of segindices for each residue
- trajectory (bool): whether to attach a MemoryReader trajectory (default False)
- velocities (bool): whether to include velocities in the trajectory (default False)
- forces (bool): whether to include forces in the trajectory (default False)

We will create a Universe with 1000 water molecules.

[2]:
```
n_residues = 1000
n_atoms = n_residues * 3

# create resindex list
resindices = np.repeat(range(n_residues), 3)
assert len(resindices) == n_atoms
print("resindices:", resindices[:10])

# all water molecules belong to 1 segment
segindices = [0] * n_residues
print("segindices:", segindices[:10])
```

```
resindices: [0 0 0 1 1 1 2 2 2 3]
segindices: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

[3]:
```
# create the Universe
sol = mda.Universe.empty(n_atoms,
                         n_residues=n_residues,
                         atom_resindex=resindices,
                         residue_segindex=segindices,
                         trajectory=True) # necessary for adding coordinates
sol
```

[3]:
```
<Universe with 3000 atoms>
```

Adding topology attributes

There isn't much we can do with our current Universe because MDAnalysis has no information on the particle elements, positions, etc. We can add relevant information manually using TopologyAttrs.

names

```
[4]: sol.add_TopologyAttr('name', ['O', 'H1', 'H2']*n_residues)
     sol.atoms.names
```

```
[4]: array(['O', 'H1', 'H2', ..., 'O', 'H1', 'H2'], dtype=object)
```

elements ("types")

Elements are typically contained in the `type` topology attribute.

```
[5]: sol.add_TopologyAttr('type', ['O', 'H', 'H']*n_residues)
     sol.atoms.types
```

```
[5]: array(['O', 'H', 'H', ..., 'O', 'H', 'H'], dtype=object)
```

residue names ("resnames")

```
[6]: sol.add_TopologyAttr('resname', ['SOL']*n_residues)
     sol.atoms.resnames
```

```
[6]: array(['SOL', 'SOL', 'SOL', ..., 'SOL', 'SOL', 'SOL'], dtype=object)
```

residue counter ("resids")

```
[7]: sol.add_TopologyAttr('resid', list(range(1, n_residues+1)))
     sol.atoms.resids
```

```
[7]: array([   1,    1,    1, ..., 1000, 1000, 1000])
```

segment/chain names ("segids")

```
[8]: sol.add_TopologyAttr('segid', ['SOL'])
     sol.atoms.segids
```

```
[8]: array(['SOL', 'SOL', 'SOL', ..., 'SOL', 'SOL', 'SOL'], dtype=object)
```

Adding positions

Positions can simply be assigned, without adding a topology attribute.

The O-H bond length in water is around 0.96 Angstrom, and the bond angle is 104.45°. We can first obtain a set of coordinates for one molecule, and then translate it for every water molecule.

```
[9]: # coordinates obtained by building a molecule in the program IQMol
     h2o = np.array([[ 0,        0,       0      ],  # oxygen
                     [ 0.95908, -0.02691, 0.03231],  # hydrogen
                     [-0.28004, -0.58767, 0.70556]]) # hydrogen
```

```
[10]: grid_size = 10
      spacing = 8
```

(continues on next page)

(continued from previous page)

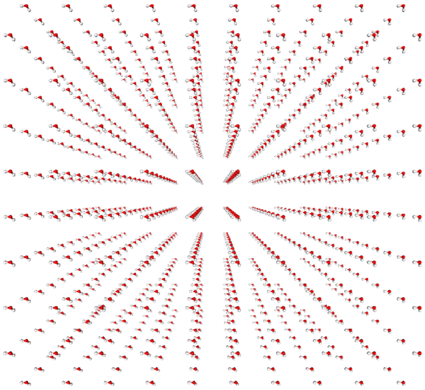
```
coordinates = []

# translating h2o coordinates around a grid
for i in range(n_residues):
    x = spacing * (i % grid_size)
    y = spacing * ((i // grid_size) % grid_size)
    z = spacing * (i // (grid_size * grid_size))

    xyz = np.array([x, y, z])

    coordinates.extend(h2o + xyz.T)

print(coordinates[:10])
```

```
[array([0., 0., 0.]), array([ 0.95908, -0.02691,  0.03231]), array([-0.28004, -0.58767, ␣
↪0.70556]), array([8., 0., 0.]), array([ 8.95908, -0.02691,  0.03231]), array([ 7.71996,
↪ -0.58767,  0.70556]), array([16.,  0.,  0.]), array([16.95908, -0.02691,  0.03231]),␣
↪array([15.71996, -0.58767,  0.70556]), array([24.,  0.,  0.])]
```

[11]:
```
coord_array = np.array(coordinates)
assert coord_array.shape == (n_atoms, 3)
sol.atoms.positions = coord_array
```

We can view the atoms with NGLView, a library for visualising molecules. It guesses bonds based on distance.

[12]:
```
# sol_view = nv.show_mdanalysis(sol)
# sol_view.add_representation('ball+stick', selection='all')
# sol_view.center()
# sol_view
```



Adding bonds

Currently, the sol universe doesn't contain any bonds.

```
[13]: assert not hasattr(sol, 'bonds')
```

They can be important for defining 'fragments', which are groups of atoms where every atom is connected by a bond to another atom in the group (i.e. what is commonly called a molecule). You can pass a list of tuples of atom indices to add bonds as a topology attribute.

```
[14]: bonds = []
      for o in range(0, n_atoms, 3):
          bonds.extend([(o, o+1), (o, o+2)])

      bonds[:10]
```

```
[14]: [(0, 1),
       (0, 2),
       (3, 4),
       (3, 5),
       (6, 7),
       (6, 8),
       (9, 10),
       (9, 11),
       (12, 13),
       (12, 14)]
```

```
[15]: sol.add_TopologyAttr('bonds', bonds)
      sol.bonds
```

```
[15]: <TopologyGroup containing 2000 bonds>
```

The bonds associated with each atom or the bonds within an AtomGroup can be accessed with the bonds attribute:

```
[16]: print(sol.atoms[0].bonds)
      print(sol.atoms[-10:].bonds)
```

```
<TopologyGroup containing 2 bonds>
<TopologyGroup containing 7 bonds>
```

Merging with a protein

Now we can merge the water with a protein to create a combined system by using `MDAnalysis.Merge` to combine AtomGroup instances.

The protein is adenylate kinase (AdK), a phosphotransferase enzyme. *[1]*

```
[17]: protein = mda.Universe(PDB_small)
```

```
[18]: # protein_view = nv.show_mdanalysis(protein)
      # protein_view
```

We will translate the centers of both systems to the origin, so they can overlap in space.

```
[19]: cog = sol.atoms.center_of_geometry()
      print('Original solvent center of geometry: ', cog)
      sol.atoms.positions -= cog
      cog2 = sol.atoms.center_of_geometry()
      print('New solvent center of geometry: ', cog2)
```

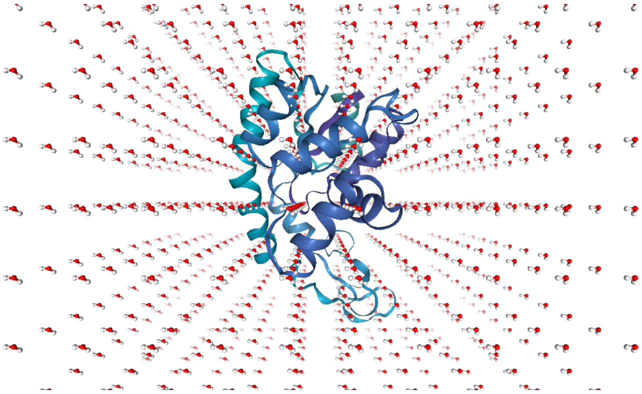
```
Original solvent center of geometry:  [36.22634681 35.79514029 36.24595657]
New solvent center of geometry:  [ 2.78155009e-07 -1.27156576e-07  3.97364299e-08]
```

```
[20]: cog = protein.atoms.center_of_geometry()
      print('Original solvent center of geometry: ', cog)
      protein.atoms.positions -= cog
      cog2 = protein.atoms.center_of_geometry()
      print('New solvent center of geometry: ', cog2)
```

```
Original solvent center of geometry:  [-3.66508082  9.60502842 14.33355791]
New solvent center of geometry:  [8.30580288e-08 3.49225059e-08 2.51332265e-08]
```

```
[21]: combined = mda.Merge(protein.atoms, sol.atoms)
```

```
[22]: # combined_view = nv.show_mdanalysis(combined)
      # combined_view.add_representation("ball+stick", selection="not protein")
      # combined_view
```



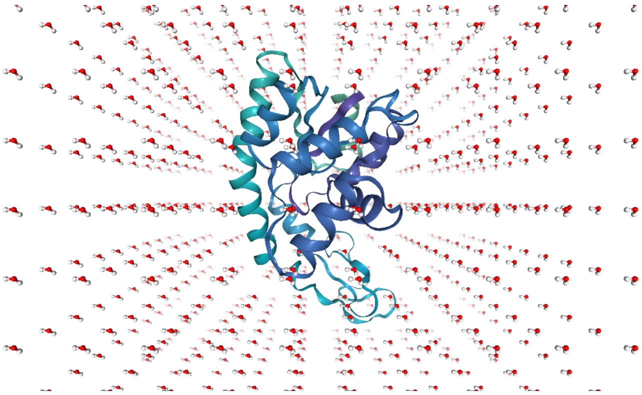
Unfortunately, some water molecules overlap with the protein. We can create a new AtomGroup containing only the molecules where every atom is further away than 6 angstroms from the protein.

```
[23]: no_overlap = combined.select_atoms("same resid as (not around 6 protein)")
```

With this AtomGroup, we can then construct a new Universe.

```
[24]: u = mda.Merge(no_overlap)
```

```
[25]: # no_overlap_view = nv.show_mdanalysis(u)
      # no_overlap_view.add_representation("ball+stick", selection="not protein")
      # no_overlap_view
```



Adding a new segment

Often you may want to assign atoms to a segment or chain – for example, adding segment IDs to a PDB file. This requires adding a new Segment with `Universe.add_Segment`.

Adenylate kinase has three domains: CORE, NMP, and LID. As shown in the picture below,*[1]* these have the residues:

- CORE: residues 1-29, 60-121, 160-214 (gray)
- NMP: residues 30-59 (blue)
- LID: residues 122-159 (yellow)

```
[26]: u.segments.segids
```

```
[26]: array(['4AKE', 'SOL'], dtype=object)
```

On examining the Universe, we can see that the protein and solvent are already divided into two segments: protein ('4AKE') and solvent ('SOL'). We will add three more segments (CORE, NMP, and LID) and assign atoms to them.

First, add a Segment to the Universe with a `segid`. It will be empty:

```
[27]: core_segment = u.add_Segment(segid='CORE')
      core_segment.atoms
```

```
[27]: <AtomGroup with 0 atoms>
```

Residues can't be broken across segments. To put atoms in a segment, assign the segments attribute of their residues:

```
[28]: core_atoms = u.select_atoms('resid 1:29 or resid 60:121 or resid 160-214')
      core_atoms.residues.segments = core_segment
      core_segment.atoms
```

[28]:

```
<AtomGroup with 2744 atoms>
```

[29]:

```
nmp_segment = u.add_Segment(segid='NMP')
lid_segment = u.add_Segment(segid='LID')

nmp_atoms = u.select_atoms('resid 30:59')
nmp_atoms.residues.segments = nmp_segment

lid_atoms = u.select_atoms('resid 122:159')
lid_atoms.residues.segments = lid_segment
```

As of MDAnalysis 2.1.0, PDBs use the `chainID` `TopologyAttr` for the `chainID` column. If it is missing, it uses a placeholder "X" value instead of the `segid`. We therefore must manually set that ourselves to visualise the protein in NGLView.

[30]:

```
# add the topologyattr to the universe
u.add_TopologyAttr("chainID")
core_segment.atoms.chainIDs = "C"
nmp_segment.atoms.chainIDs = "N"
lid_segment.atoms.chainIDs = "L"
```

We can check that we have the correct domains by visualising the protein.

[31]:

```
# domain_view = nv.show_mdanalysis(u)
# domain_view.add_representation("protein", color_scheme="chainID")
# domain_view
```

Tiling into a larger Universe

We can use MDAnalysis to tile out a smaller Universe into a bigger one, similarly to `editconf` in GROMACS. To start off, we need to figure out the box size. The default in MDAnalysis is a zero vector. The first three numbers represent the length of each axis, and the last three represent the alpha, beta, and gamma angles of a triclinic box.

[32]:

```
print(u.dimensions)
```

```
None
```

We know that our system is cubic in shape, so we can assume angles of 90°. The difference between the lowest and highest x-axis positions is roughly 73 Angstroms.

[33]:

```
max(u.atoms.positions[:, 0]) - min(u.atoms.positions[:, 0])
```

[33]:

```
73.23912
```

So we can set our dimensions.

[34]:

```
u.dimensions = [73, 73, 73, 90, 90, 90]
```

To tile out a Universe, we need to copy it and translate the atoms by the box dimensions. We can then merge the cells into one large Universe and assign new dimensions.

[35]:

```
def tile_universe(universe, n_x, n_y, n_z):
    box = universe.dimensions[:3]
    copied = []
    for x in range(n_x):
        for y in range(n_y):
            for z in range(n_z):
                u_ = universe.copy()
                move_by = box*(x, y, z)
                u_.atoms.translate(move_by)
                copied.append(u_.atoms)

    new_universe = mda.Merge(*copied)
    new_box = box*(n_x, n_y, n_z)
    new_universe.dimensions = list(new_box) + [90]*3
    return new_universe
```

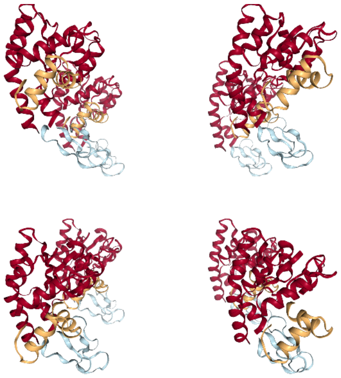
Here is a 2 x 2 x 2 version of our original unit cell:

[36]:

```
tiled = tile_universe(u, 2, 2, 2)
```

[37]:

```
# tiled_view = nv.show_mdanalysis(tiled)
# tiled_view
```



References

[1]: Beckstein O, Denning EJ, Perilla JR, Woolf TB. Zipping and unzipping of adenylate kinase: atomistic insights into the ensemble of open<–>closed transitions. J Mol Biol. 2009;394(1):160–176. doi:10.1016/j.jmb.2009.09.009

Acknowledgments

The Universe tiling code was modified from @richardjgowers's gist on the issue in 2016.

Transformations

Centering a trajectory in the box

Here we use MDAnalysis transformations to make a protein whole, center it in the box, and then wrap the water back into the box. We then look at how to do this on-the-fly.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation: * nglview ([NCR18])

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

See also:

- *On-the-fly transformations*
- On-the-fly transformations (blog post)

[1]:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import TPR, XTC
import numpy as np
# import nglview as nv
```

Loading files

The test files we will be working with here are trajectories of a adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

For the step-by-step transformations, we need to load the trajectory into memory so that our changes to the coordinates persist. If your trajectory is too large for that, see the *on-the-fly transformation* section for how to do this out-of-memory.

[2]:

```
u = mda.Universe(TPR, XTC, in_memory=True)
```

Before transformation

If you have NGLView installed, you can view the trajectory as it currently is.

[3]:

```
# view = nv.show_mdanalysis(u)
# view.add_representation('point', 'resname SOL')
# view.center()
# view
```

[4]:

```
# from nglview.contrib.movie import MovieMaker
# movie = MovieMaker(
#     view,
#     step=2,
#     render_params={"factor": 2},  # average quality render
#     output='original.gif',
# )
# movie.make()
```

Otherwise, we embed a gif of it below.

For easier analysis and nicer visualisation, we want to center this protein in the box.

Unwrapping the protein

The first step is to "unwrap" the protein from the border of the box, to make the protein whole. MDAnalysis provides the `AtomGroup.unwrap` function to do this easily. Note that this function requires your universe to have bonds in it.

We loop over the trajectory to unwrap for each frame.

[5]:

```
protein = u.select_atoms('protein')

for ts in u.trajectory:
    protein.unwrap(compound='fragments')
```

As you can see, the protein is now whole, but not centered.

[6]:

```
# unwrapped = nv.show_mdanalysis(u)
# unwrapped.add_representation('point', 'resname SOL')
# unwrapped.center()
# unwrapped
```

Over the course of the trajectory it leaves the box.

Centering in the box

The next step is to center the protein in the box. We calculate the center-of-mass of the protein and the center of the box for each timestep. We then move *all* the atoms so that the protein center-of-mass is in the center of the box.

If you don't have masses in your trajectory, try using the `center_of_geometry`.

[7]:
```
for ts in u.trajectory:
    protein_center = protein.center_of_mass(wrap=True)
    dim = ts.triclinic_dimensions
    box_center = np.sum(dim, axis=0) / 2
    u.atoms.translate(box_center - protein_center)
```

The protein is now in the center of the box, but the solvent is likely outside it, as we have just moved all the atoms.

[8]:
```
# centered = nv.show_mdanalysis(u)
# centered.add_representation('point', 'resname SOL')
# centered.center()
# centered
```

Wrapping the solvent back into the box

Luckily, MDAnalysis also has `AtomGroup.wrap` to wrap molecules back into the box. Our trajectory has dimensions defined, which the function will find automatically. If your trajectory does not, or you wish to use a differently sized box, you can pass in a `box` with dimensions in the form `[lx, ly, lz, alpha, beta, gamma]`.

[9]:
```
not_protein = u.select_atoms('not protein')

for ts in u.trajectory:
    not_protein.wrap(compound='residues')
```

And now it is centered!

[10]:
```
# wrapped = nv.show_mdanalysis(u)
# wrapped.add_representation('point', 'resname SOL')
# wrapped.center()
# wrapped
```

Doing all this on-the-fly

Running all the transformations above can be difficult if your trajectory is large, or your computational resources are limited. Use on-the-fly transformations to keep your data out-of-memory.

Some common transformations are defined in `MDAnalysis.transformations`.

[11]:
```
import MDAnalysis.transformations as trans
```

We re-load our universe.

```
[12]: u2 = mda.Universe(TPR, XTC)

      protein2 = u2.select_atoms('protein')
      not_protein2 = u2.select_atoms('not protein')
```

From version 1.0.0 onwards, the `MDAnalysis.transformations` module contains `wrap` and `unwrap` functions that correspond with the `AtomGroup` versions above. You can only use `add_transformations` *once*, so pass them all at the same time.

```
[13]: transforms = [trans.unwrap(protein2),
                    trans.center_in_box(protein2, wrap=True),
                    trans.wrap(not_protein2)]

      u2.trajectory.add_transformations(*transforms)
```

```
[14]: # otf = nv.show_mdanalysis(u2)
      # otf.add_representation('point', 'resname SOL')
      # otf.center()
      # otf
```

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Hai Nguyen, David A Case, and Alexander S Rose. NGLview–interactive molecular graphics for Jupyter notebooks. Bioinformatics, 34(7):1241–1242, April 2018. 00024. URL: https://academic.oup.com/bioinformatics/article/34/7/1241/4721781, doi:10.1093/bioinformatics/btx789.

Other

Using ParmEd with MDAnalysis and OpenMM to simulate a selection of atoms

Here we use MDAnalysis to convert a ParmEd structure to an MDAnalysis Universe, select a subset of atoms, and convert it back to ParmEd to simulate with OpenMM.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Last updated: December 2022

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis *[1, 2]*
- MDAnalysisTests
- ParmEd
- OpenMM *[3]*

[1]:
```
import parmed as pmd
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PRM7_ala2, RST7_ala2

import warnings
# suppress some MDAnalysis warnings when writing PDB files
warnings.filterwarnings('ignore')
```

Loading files: the difference between ParmEd and MDAnalysis

Both ParmEd and MDAnalysis read a number of file formats. However, while MDAnalysis is typically used to analyse simulations, ParmEd is often used to set them up. This requires ParmEd to read topology parameter information that MDAnalysis typically ignores, such as the equilibrium length and force constants of bonds in the system. For example, the ParmEd structure below.

[2]:
```
pprm = pmd.load_file(PRM7_ala2, RST7_ala2)
pprm
```

[2]:
```
<AmberParm 3026 atoms; 1003 residues; 3025 bonds; PBC (orthogonal); parameterized>
```

[3]:
```
pprm.bonds[0]
```

[3]:
```
<Bond <Atom C [10]; In ALA 0>--<Atom O [11]; In ALA 0>; type=<BondType; k=570.000, req=1.
↪229>>
```

When MDAnalysis reads these files in, it does not include that information.

[4]:
```
mprm = mda.Universe(PRM7_ala2, RST7_ala2, format='RESTRT')
mprm
```

[4]:
```
<Universe with 3026 atoms>
```

The bond type simply shows the atom types involved in the connection.

[5]:
```
mprm.atoms.bonds[0].type
```

[5]:
```
('N3', 'H')
```

If you then convert this Universe to ParmEd, you can see that the resulting Structure is not `parametrized`.

[6]:
```
mprm_converted = mprm.atoms.convert_to('PARMED')
mprm_converted
```

[6]:
```
<Structure 3026 atoms; 1003 residues; 3025 bonds; parameterized>
```

While the bonds are present, there is no `type` information associated.

[7]:
```
mprm_converted.bonds[0]
```

[7]:
```
<Bond <Atom N [0]; In ALA 0>--<Atom H1 [1]; In ALA 0>; type=None>
```

Therefore, if you wish to use ParmEd functionality that requires parametrization on a MDAnalysis Universe, you need to create that Universe *from* a ParmEd structure in order to convert it *back to* something useable in ParmEd.

[8]:
```
mprm_from_parmed = mda.Universe(pprm)
mprm_from_parmed
```

[8]:
```
<Universe with 3026 atoms>
```

Now the bond type is actually a ParmEd Bond object.

[9]:
```
mprm_from_parmed.bonds[0].type
```

[9]:
```
<Bond <Atom N [0]; In ALA 0>--<Atom H1 [1]; In ALA 0>; type=<BondType; k=434.000, req=1.
↪010>>
```

Using MDAnalysis to select atoms

One reason we might want to convert a ParmEd structure into MDAnalysis is to use its sophisticated atom selection syntax. While ParmEd has its own ways to select atoms, MDAnalysis allows you to select atoms based on geometric distance.

[10]:
```
water = mprm_from_parmed.select_atoms('around 5 protein').residues.atoms
protein_shell = mprm_from_parmed.select_atoms('protein') + water
prm_protein_shell = protein_shell.convert_to('PARMED')
```

[11]:
```
prm_protein_shell
```

[11]:
```
<Structure 155 atoms; 46 residues; 154 bonds; PBC (orthogonal); parameterized>
```

Using ParmEd and OpenMM to create a simulation system

[12]:
```
import sys
import openmm as mm
import openmm.app as app
from parmed import unit as u
from parmed.openmm import StateDataReporter, MdcrdReporter
```

You can create an OpenMM simulation system directly from a ParmEd structure, providing that it is parametrized.

[13]:
```
system = prm_protein_shell.createSystem(nonbondedMethod=app.NoCutoff,
                                        constraints=app.HBonds,
                                        implicitSolvent=app.GBn2)
```

Here we set the integrator to do Langevin dynamics.

[14]:
```
integrator = mm.LangevinIntegrator(
                        300*u.kelvin,       # Temperature of heat bath
                        1.0/u.picoseconds,  # Friction coefficient
```

(continues on next page)

(continued from previous page)

```
                                    2.0*u.femtoseconds, # Time step
                                    )
```

We create the Simulation object and set particle positions.

[15]:

```
sim = app.Simulation(prm_protein_shell.topology, system, integrator)
sim.context.setPositions(prm_protein_shell.positions)
```

We now minimise the energy.

[16]:

```
sim.minimizeEnergy(maxIterations=500)
```

The reporter below reports energies and coordinates every 100 steps to stdout, but every 10 steps to the ala2_shell.nc file.

[17]:

```
sim.reporters.append(
        StateDataReporter(sys.stdout, 100, step=True, potentialEnergy=True,
                          kineticEnergy=True, temperature=True, volume=True,
                          density=True)
)
sim.reporters.append(MdcrdReporter('ala2_shell.trj', 10, crds=True))
```

We can run dynamics for 500 steps (1 picosecond).

[18]:

```
sim.step(500)
```

```
#"Step","Time (ps)","Potential Energy (kilocalorie/mole)","Kinetic Energy (kilocalorie/
↪mole)","Total Energy (kilocalorie/mole)","Temperature (K)","Box Volume (angstrom**3)",
↪"Density (gram/(item*milliliter))"
100,0.20000000000000015,-623.6779995219885,20.140631869613383,-603.5373676523751,63.
↪74314071570579,45325.8064191062,0.034909350700361955
200,0.4000000000000003,-614.1849904397706,38.40737137186695,-575.7776190679035,121.
↪55559436896063,45325.8064191062,0.034909350700361955
300,0.6000000000000004,-606.5526783580306,48.61919832973248,-557.933480028298,153.
↪87503334950912,45325.8064191062,0.034909350700361955
400,0.8000000000000006,-600.0374380078872,57.988937528818184,-542.0485004790689,183.
↪52934648642113,45325.8064191062,0.034909350700361955
500,1.0000000000000007,-603.2854886173518,79.46589388029852,-523.8195947370533,251.
↪50182419815255,45325.8064191062,0.034909350700361955
```

If we write a topology file out from our former protein_shell atomgroup, we can load the trajectory in for further analysis.

[19]:

```
protein_shell.write('ala2_shell.pdb')
```

[20]:

```
u = mda.Universe('ala2_shell.pdb', 'ala2_shell.trj')
```

[21]:

```
u.trajectory
```

[21]:

```
<TRJReader ala2_shell.trj with 50 frames of 155 atoms>
```

References

[1] R. J. Gowers, M. Linke, J. Barnoud, T. J. E. Reddy, M. N. Melo, S. L. Seyler, D. L. Dotson, J. Domanski, S. Buchoux, I. M. Kenney, and O. Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In S. Benthall and S. Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98-105, Austin, TX, 2016. SciPy, doi: 10.25080/majora-629e541a-00e.

[2] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein. MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations. *J. Comput. Chem.* 32 (2011), 2319-2327, doi:10.1002/jcc.21787. PMCID:PMC3144279

[3] Peter Eastman, Jason Swails, John D. Chodera, Robert T. McGibbon, Yutong Zhao, Kyle A. Beauchamp, Lee-Ping Wang, Andrew C. Simmonett, Matthew P. Harrigan, Chaya D. Stern, Rafal P. Wiewiora, Bernard R. Brooks, Vijay S. Pande. OpenMM 7: Rapid Development of High Performance Algorithms for Molecular Dynamics. *PLoS Comput. Biol.* 13:e1005659, 2017.

[4] Hai Nguyen, David A Case, Alexander S Rose. NGLview - Interactive molecular graphics for Jupyter notebooks. *Bioinformatics*. 34 (2018), 1241–1242, doi:10.1093/bioinformatics/btx789

Alignments and RMS fitting

The `MDAnalysis.analysis.align` and `MDAnalysis.analysis.rms` modules contain the functions used for aligning structures, aligning trajectories, and calculating root mean squared quantities.

Demonstrations of alignment are in align_structure, align_trajectory_first, and align_trajectory. Another example of generating an average structure from an alignment is demonstrated in rmsf. Typically, trajectories need to be aligned for RMSD and RMSF values to make sense.

Note: These modules use the fast QCP algorithm to calculate the root mean square distance (RMSD) between two coordinate sets [The05] and the rotation matrix R that minimizes the RMSD [LAT09]. Please cite these references when using these modules.

Aligning a structure to another

We use `align.alignto` to align a structure to another.

Last updated: December 2022

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for molecular visualisation:

- nglview ([NCR18])

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

See also

- *Aligning a trajectory to a frame from another*
- *Aligning a trajectory to a frame from itself*

- *RMSD*

Note

MDAnalysis implements RMSD calculation using the fast QCP algorithm ([The05]) and a rotation matrix *R* that minimises the RMSD ([LAT09]). Please cite ([The05]) and ([LAT09]) when using the `MDAnalysis.analysis.align` module in published work.

[1]:
```
import MDAnalysis as mda
from MDAnalysis.analysis import align
from MDAnalysis.tests.datafiles import CRD, PSF, DCD, DCD2
# import nglview as nv

import warnings
# suppress some MDAnalysis warnings about writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

The test files we will be working with here are trajectories of a adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectories sample a transition from a closed to an open conformation.

[2]:
```
adk_open = mda.Universe(CRD, DCD2)
adk_closed = mda.Universe(PSF, DCD)
```

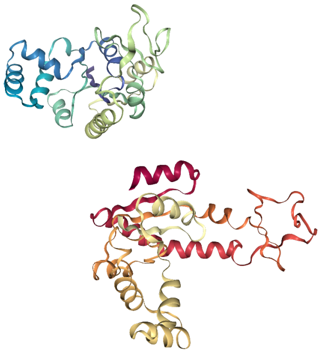
[3]:
```
# adk_open_view = nv.show_mdanalysis(adk_open)
# adk_open_view
```

[4]:
```
# adk_closed_view = nv.show_mdanalysis(adk_closed)
# adk_closed_view
```

Currently, the proteins are not aligned to each other. The difference becomes even more obvious when the closed conformation is compared to the open. Below, we set `adk_open` to the last frame and see the relative positions of each protein in a merged Universe.

[5]:
```
adk_open.trajectory[-1] # last frame
merged = mda.Merge(adk_open.atoms, adk_closed.atoms)
```

[6]:
```
# merged_view = nv.show_mdanalysis(merged)
# merged_view
```



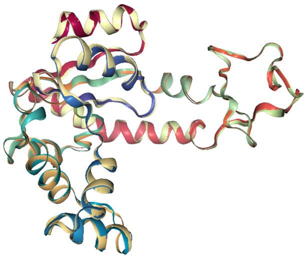
Aligning a structure with align.alignto

alignto (API docs) aligns the mobile AtomGroup to the target AtomGroup by minimising the *root mean square deviation (RMSD) between particle positions* (please see the linked notebook for an explanation of RMSD). It returns *(old_rmsd, new_rmsd)*. By default (match_atoms=True), it will attempt to match the atoms between the mobile and reference structures by mass.

```
[7]: rmsds = align.alignto(adk_open,  # mobile
                           adk_closed,  # reference
                           select='name CA', # selection to operate on
                           match_atoms=True) # whether to match atoms
     print(rmsds)
```

```
(21.712154435976014, 6.817293751703919)
```

```
[8]: # aligned_view = nv.show_mdanalysis(mda.Merge(adk_open.atoms, adk_closed.atoms))
     # aligned_view
```



However, you may want to align to a structure that where there is not a clear match in particle mass. For example, you could be aligning the alpha-carbons of an atomistic protein to the backbone beads of a coarse-grained structure. Below, we use the somewhat contrived example of aligning 214 alpha-carbons to the first 214 atoms of the reference structure. In this case, we need to switch match_atoms=False or the alignment will error.

```
[9]: rmsds = align.alignto(adk_open.select_atoms('name CA'),  # mobile
                           adk_closed.atoms[:214],  # reference
                           select='all', # selection to operate on
```

(continues on next page)

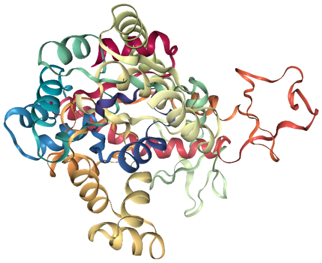
(continued from previous page)

```
                            match_atoms=False) # whether to match atoms
print(rmsds)
```

```
(18.991465038265208, 16.603704620787127)
```

[10]:

```
# shifted_aligned_view = nv.show_mdanalysis(mda.Merge(adk_open.atoms, adk_closed.atoms))
# shifted_aligned_view
```



When we align structures, positions are set temporarily. If we flip to the first frame of adk_open and back to the last frame, we can see that it has returned to its original location.

[11]:

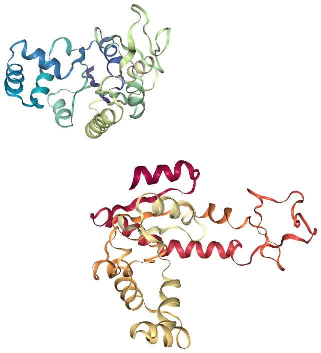
```
adk_open.trajectory[0] # set to first frame
adk_open.trajectory[-1] # set to last frame
```

[11]:

```
< Timestep 101 >
```

[12]:

```
# reset_view = nv.show_mdanalysis(mda.Merge(adk_open.atoms, adk_closed.atoms))
# reset_view
```



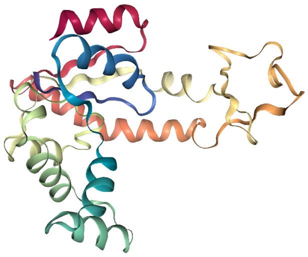
You can save the aligned positions by writing them out to a PDB file and creating a new Universe.

[13]:

```
align.alignto(adk_open, adk_closed, select='name CA')
adk_open.atoms.write('aligned.pdb')
```

[14]:

```
# from_file_view = nv.show_mdanalysis(mda.Universe('aligned.pdb'))
# from_file_view
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Pu Liu, Dimitris K. Agrafiotis, and Douglas L. Theobald. Fast determination of the optimal rotational matrix for macromolecular superpositions. Journal of Computational Chemistry, pages n/a–n/a, 2009. URL: http://doi.wiley.com/10.1002/jcc.21439, doi:10.1002/jcc.21439.

[4] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[5] Hai Nguyen, David A Case, and Alexander S Rose. NGLview–interactive molecular graphics for Jupyter notebooks. Bioinformatics, 34(7):1241–1242, April 2018. 00024. URL: https://academic.oup.com/bioinformatics/article/34/7/1241/4721781, doi:10.1093/bioinformatics/btx789.

[6] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallographica Section A Foundations of Crystallography, 61(4):478–480, July 2005. 00127. URL: http://scripts.iucr.org/cgi-bin/paper?S0108767305015266, doi:10.1107/S0108767305015266.

Aligning a trajectory to a reference

We use `align.AlignTraj` to align a trajectory to a frame in a reference trajectory and write it to a file.

Last updated: December 2022

Minimum version of MDAnalysis: 2.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for molecular visualisation:

- nglview ([NCR18])

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

See also

- *Aligning a trajectory to a frame from itself*
- *Aligning a structure to another*
- *RMSD*

Note

MDAnalysis implements RMSD calculation using the fast QCP algorithm ([The05]) and a rotation matrix R that minimises the RMSD ([LAT09]). Please cite ([The05]) and ([LAT09]) when using the `MDAnalysis.analysis.align` module in published work.

[1]:
```
import MDAnalysis as mda
from MDAnalysis.analysis import align
from MDAnalysis.tests.datafiles import CRD, PSF, DCD, DCD2
# import nglview as nv

import warnings
# suppress some MDAnalysis warnings when writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

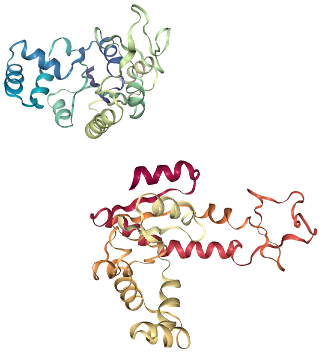
The test files we will be working with here are trajectories of a adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectories sample a transition from a closed to an open conformation.

[2]:
```
adk_open = mda.Universe(CRD, DCD2)
adk_closed = mda.Universe(PSF, DCD)
```

Currently, the proteins are not aligned to each other. The difference becomes obvious when the closed conformation is compared to the open. Below, we set `adk_open` to the last frame and see the relative positions of each protein in a merged Universe.

[3]:
```
adk_open.trajectory[-1] # last frame
merged = mda.Merge(adk_open.atoms, adk_closed.atoms)
```

[4]:
```
# merged_view = nv.show_mdanalysis(merged)
# merged_view
```

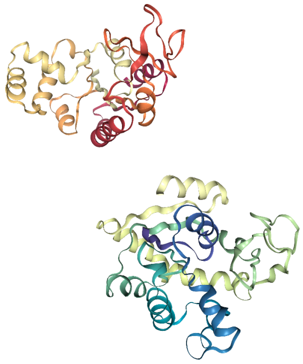


Aligning a trajectory with AlignTraj

While align.alignto aligns structures, or a frame of a trajectory, `align.AlignTraj` (API docs) efficiently aligns an entire trajectory to a reference. Unlike most other analysis modules, `AlignTraj` allows you to write the output of the analysis to a file. This is because when `Universe`s are created by loading from a file, changes to frame-by-frame (dynamic) information do not persist when the frame is changed. If the trajectory is not written to a file, or pulled into memory (below), `AlignTraj` will have no effect.

[5]:
```
align.AlignTraj(adk_closed,  # trajectory to align
                adk_open,  # reference
                select='name CA',  # selection of atoms to align
                filename='aligned.dcd',  # file to write the trajectory to
                match_atoms=True,  # whether to match atoms based on mass
               ).run()
# merge adk_closed and adk_open into the same universe
merged1 = mda.Merge(adk_closed.atoms, adk_open.atoms)
```

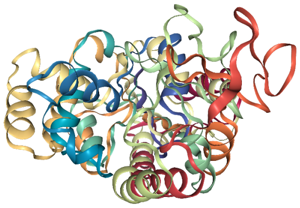
[6]:
```
# merged1_view = nv.show_mdanalysis(merged1)
# merged1_view
```



As you can see, the `adk_closed` and `adk_open` trajectories still look the same. However, when we load our aligned trajectory from `aligned.dcd`, we can see them superposed:

[7]:
```
aligned = mda.Universe(PSF, 'aligned.dcd')
aligned.segments.segids = ['Aligned']  # rename our segments
adk_open.segments.segids = ['Open']  # so they're coloured differently
merged2 = mda.Merge(aligned.atoms, adk_open.atoms)
```

[8]:
```
# merged2_view = nv.show_mdanalysis(merged2)
# merged2_view
```



If you don't want to write a file, you can also choose to load the entire trajectory into memory. (This is not always feasible depending on how large your trajectory is, and how much memory your device has, in which case it is much more efficient to write an aligned trajectory to a file as above). You can accomplish this in one of two ways:

1. Load the trajectory into memory in the first place

```
adk_closed = mda.Universe(PSF, DCD, in_memory=True)
```

2. Select `in_memory=True` during `AlignTraj` (below)

[9]:
```
align.AlignTraj(adk_closed,  # trajectory to align
                adk_open,  # reference
                select='name CA',  # selection of atoms to align
                filename='aligned.dcd',  # file to write the trajectory to
                match_atoms=True,  # whether to match atoms based on mass
                in_memory=True
               ).run()
# merge adk_closed and adk_open into the same universe
merged3 = mda.Merge(adk_closed.atoms, adk_open.atoms)
```

Copying coordinates into a new Universe

`MDAnalysis.Merge` does not automatically load coordinates for a trajectory. We can do this ourselves. Below, we copy the coordinates of the 98 frames in the `aligned` universe.

[10]:
```
from MDAnalysis.analysis.base import AnalysisFromFunction
import numpy as np
from MDAnalysis.coordinates.memory import MemoryReader

def copy_coords(ag):
    return ag.positions.copy()

aligned_coords = AnalysisFromFunction(copy_coords,
                                      aligned.atoms).run().results

print(aligned_coords['timeseries'].shape)
```

```
(98, 3341, 3)
```

To contrast, we will keep the open conformation static.

```
[11]: adk_coords = adk_open.atoms.positions.copy()
      adk_coords.shape
```

```
[11]: (3341, 3)
```

Because there are 98 frames of the `aligned` Universe, we copy the coordinates of the `adk_open` positions and stack them.

```
[12]: adk_traj_coords = np.stack([adk_coords] * 98)
      adk_traj_coords.shape
```

```
[12]: (98, 3341, 3)
```

We join `aligned_coords` and `adk_traj_coords` on the second axis with `np.hstack` and load the coordinates into memory into the `merged2` Universe.

```
[13]: merged_coords = np.hstack([aligned_coords['timeseries'],
                                 adk_traj_coords])
      merged2.load_new(merged_coords, format=MemoryReader)
```

```
[13]: <Universe with 6682 atoms>
```

```
[14]: # m2_view = nv.show_mdanalysis(merged2)
      # m2_view
```

Online notebooks do not show the molecule trajectory, but here you can use `nglview.contrib.movie.MovieMaker` to make a gif of the trajectory. This requires you to install `moviepy`.

```
[15]: # from nglview.contrib.movie import MovieMaker
      # movie = MovieMaker(
      #     m2_view,
      #     step=4,  # only render every 4th frame
      #     output='merged.gif',
      #     render_params={"factor": 3},  # set to 4 for higher quality
      # )
      # movie.make()
```

Writing trajectories to a file

Finally, we can also save this new trajectory to a file.

```
[16]: with mda.Writer('aligned.xyz', merged2.atoms.n_atoms) as w:
          for ts in merged2.trajectory:
              w.write(merged2.atoms)
```

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Pu Liu, Dimitris K. Agrafiotis, and Douglas L. Theobald. Fast determination of the optimal rotational matrix for macromolecular superpositions. Journal of Computational Chemistry, pages n/a–n/a, 2009. URL: http://doi.wiley.com/10.1002/jcc.21439, doi:10.1002/jcc.21439.

[4] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[5] Hai Nguyen, David A Case, and Alexander S Rose. NGLview–interactive molecular graphics for Jupyter notebooks. Bioinformatics, 34(7):1241–1242, April 2018. 00024. URL: https://academic.oup.com/bioinformatics/article/34/7/1241/4721781, doi:10.1093/bioinformatics/btx789.

[6] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallographica Section A Foundations of Crystallography, 61(4):478–480, July 2005. 00127. URL: http://scripts.iucr.org/cgi-bin/paper?S0108767305015266, doi:10.1107/S0108767305015266.

Aligning a trajectory to itself

We use `align.AlignTraj` to align a trajectory to a reference frame and write it to a file.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

See also

- *Aligning a trajectory to a frame from another*
- *Aligning a structure to another*
- *RMSD*

Note

MDAnalysis implements RMSD calculation using the fast QCP algorithm ([The05]) and a rotation matrix R that minimises the RMSD ([LAT09]). Please cite ([The05]) and ([LAT09]) when using the `MDAnalysis.analysis.align` module in published work.

[1]:

```
import MDAnalysis as mda
from MDAnalysis.analysis import align, rms
from MDAnalysis.tests.datafiles import PSF, DCD
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectory samples a transition from a closed to an open conformation.

[2]:

```
mobile = mda.Universe(PSF, DCD)
ref = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Aligning a trajectory to the first frame

While align.alignto aligns structures, or a frame of a trajectory, `align.AlignTraj` (API docs) efficiently aligns an entire trajectory to a reference.

We first check the *root mean square deviation (RMSD) values* of our unaligned trajectory, so we can compare results (please see the linked notebook for an explanation of RMSD). The code below sets the `mobile` trajectory to the last frame by indexing the last timestep, `ref` to the first frame by indexing the first timestep, and computes the root mean squared deviation between the alpha-carbon positions.

[3]:

```
mobile.trajectory[-1]  # set mobile trajectory to last frame
ref.trajectory[0]  # set reference trajectory to first frame

mobile_ca = mobile.select_atoms('name CA')
ref_ca = ref.select_atoms('name CA')
unaligned_rmsd = rms.rmsd(mobile_ca.positions, ref_ca.positions, superposition=False)
print(f"Unaligned RMSD: {unaligned_rmsd:.2f}")
```

```
Unaligned RMSD: 6.84
```

Now we can align the trajectory. We have already set `ref` to the first frame. In the cell below, we load the positions of the trajectory into memory so we can modify the trajectory in Python.

[4]:

```
aligner = align.AlignTraj(mobile, ref, select='name CA', in_memory=True).run()
```

If you don't have enough memory to do that, write the trajectory out to a file and reload it into MDAnalysis (uncomment the cell below). Otherwise, you don't have to run it.

[5]:

```
# aligner = align.AlignTraj(mobile, ref, select='backbone',
#                           filename='aligned_to_first_frame.dcd').run()
# mobile = mda.Universe(PSF, 'aligned_to_first_frame.dcd')
```

Now we can see that the RMSD has reduced (minorly).

[6]:
```
mobile.trajectory[-1]  # set mobile trajectory to last frame
ref.trajectory[0]  # set reference trajectory to first frame

mobile_ca = mobile.select_atoms('name CA')
ref_ca = ref.select_atoms('name CA')
aligned_rmsd = rms.rmsd(mobile_ca.positions, ref_ca.positions, superposition=False)

print(f"Aligned RMSD: {aligned_rmsd:.2f}")
```

```
Aligned RMSD: 6.81
```

Aligning a trajectory to the third frame

We can align the trajectory to any frame: for example, the third one. The procedure is much the same, except that we must set `ref` to the third frame by indexing the third timestep.

[7]:
```
mobile.trajectory[-1]  # set mobile trajectory to last frame
ref.trajectory[2]  # set reference trajectory to third frame

aligned_rmsd_3 = rms.rmsd(mobile.atoms.positions, ref.atoms.positions,␣
↪superposition=False)

print(f"Aligned RMSD: {aligned_rmsd_3:.2f}")
```

```
Aligned RMSD: 6.73
```

[8]:
```
aligner = align.AlignTraj(mobile, ref, select='all', in_memory=True).run()
```

[9]:
```
mobile.trajectory[-1]  # set mobile trajectory to last frame
ref.trajectory[2]  # set reference trajectory to third frame

aligned_rmsd_3 = rms.rmsd(mobile.atoms.positions, ref.atoms.positions,␣
↪superposition=False)
print(f"Aligned RMSD, all-atom: {aligned_rmsd_3:.2f}")
```

```
Aligned RMSD, all-atom: 6.72
```

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Pu Liu, Dimitris K. Agrafiotis, and Douglas L. Theobald. Fast determination of the optimal rotational matrix for macromolecular superpositions. Journal of Computational Chemistry, pages n/a–n/a, 2009. URL: http://doi.wiley.com/10.1002/jcc.21439, doi:10.1002/jcc.21439.

[4] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[5] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallographica Section A Foundations of Crystallography, 61(4):478–480, July 2005. 00127. URL: http://scripts.iucr.org/cgi-bin/paper?S0108767305015266, doi:10.1107/S0108767305015266.

Calculating the root mean square deviation of atomic structures

We calculate the RMSD of domains in adenylate kinase as it transitions from an open to closed structure, and look at calculating weighted RMSDs.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- pandas

See also

- *Pairwise (2D) RMSD*
- *RMSF*

Note

MDAnalysis implements RMSD calculation using the fast QCP algorithm ([The05]). Please cite ([The05]) when using the MDAnalysis.analysis.align module in published work.

[1]:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD, CRD
from MDAnalysis.analysis import rms

import pandas as pd
# the next line is necessary to display plots in Jupyter
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectory DCD samples a transition from a closed to an open conformation. AdK has three domains:

- CORE
- LID: an ATP-binding domain
- NMP: an AMP-binding domain

The LID and NMP domains move around the stable CORE as the enzyme transitions between the opened and closed conformations. One way to quantify this movement is by calculating the root mean square deviation (RMSD) of atomic positions.

```
[2]: u = mda.Universe(PSF, DCD)  # closed AdK (PDB ID: 1AKE)
     ref = mda.Universe(PSF, CRD)  # open AdK (PDB ID: 4AKE)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Background

The root mean square deviation (RMSD) of particle coordinates is one measure of distance, or dissimilarity, between molecular conformations. Each structure should have matching elementwise atoms i in the same order, as the distance between them is calculated and summed for the final result. It is calculated between coordinate arrays $\mathbf{x}$ and $\mathbf{x}^{\text{ref}}$ according to the equation below:

$$\text{RMSD}(\mathbf{x}, \mathbf{x}^{\text{ref}}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}|\mathbf{x}_i - \mathbf{x}_i^{\text{ref}}|^2}$$

As molecules can move around, the structure $\mathbf{x}$ is usually translated by a vector $\mathbf{t}$ and rotated by a matrix R to align with the reference $\mathbf{x}^{\text{ref}}$ such that the RMSD is minimised. The RMSD after this optimal superposition can be expressed as follows:

$$\text{RMSD}(\mathbf{x}, \mathbf{x}^{\text{ref}}) = \min_{\mathsf{R},\mathbf{t}} \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left[(\mathsf{R}\cdot\mathbf{x}_i(t) + \mathbf{t}) - \mathbf{x}_i^{\text{ref}}\right]^2}$$

The RMSD between one reference state and a trajectory of structures is often calculated as a way to measure the dissimilarity of the trajectory conformational ensemble to the reference. This reference is frequently the first frame of the trajectory (the default in MDAnalysis), in which case it can provide insight into the overall movement from the initial starting point. While stable RMSD values from a reference structure are frequently used as a measure of conformational convergence, this metric suffers from the problem of *degeneracy*: many different structures can have the same RMSD from the same reference. For an alternative measure, you could use *pairwise or 2D RMSD*.

Typically not all coordinates in a structures are included in an RMSD analysis. With proteins, the fluctuation of the residue side-chains is not representative of overall conformational change. Therefore when RMSD analyses are performed to investigate large-scale movements in proteins, the atoms are usually restricted only to the backbone atoms (forming the amide-bond chain) or the alpha-carbon atoms.

MDAnalysis provides functions and classes to calculate the *RMSD between coordinate arrays*, and *Universes* or *AtomGroups*.

The contribution of each particle i to the final RMSD value can also *be weighted* by w_i:

$$\text{RMSD}(\mathbf{x}, \mathbf{x}^{\text{ref}}) = \sqrt{\frac{\sum_{i=1}^{n} w_i|\mathbf{x}_i - \mathbf{x}_i^{\text{ref}}|^2}{\sum_{i-1}^{n} w_i}}$$

RMSD analyses are frequently weighted by mass. The MDAnalysis RMSD class (API docs) allows you to both *select mass-weighting* with `weights='mass'` or `weights_groupselections='mass'`, or to *pass custom arrays* into either keyword.

RMSD between two sets of coordinates

The MDAnalysis.analysis.rms.rmsd function returns the root mean square deviation (in Angstrom) between two sets of coordinates. Here, we calculate the RMSD between the backbone atoms of the open and closed conformations of AdK. Only considering the backbone atoms is often more helpful than calculating the RMSD for all the atoms, as movement in amino acid side-chains isn't indicative of overall conformational change.

```
[3]: rms.rmsd(u.select_atoms('backbone').positions,  # coordinates to align
              ref.select_atoms('backbone').positions,  # reference coordinates
              center=True,  # subtract the center of geometry
              superposition=True)  # superimpose coordinates
```

```
[3]: 6.823686867261616
```

RMSD of a Universe with multiple selections

It is more efficient to use the MDAnalysis.analysis.rms.RMSD class to calculate the RMSD of an entire trajectory to a single reference point, than to use the the MDAnalysis.analysis.rms.rmsd function.

The rms.RMSD class first performs a rotational and translational alignment of the target trajectory to the reference universe at ref_frame, using the atoms in select to determine the transformation. The RMSD of the select selection is calculated. Then, *without further alignment*, the RMSD of each group in groupselections is calculated.

```
[4]: CORE = 'backbone and (resid 1-29 or resid 60-121 or resid 160-214)'
     LID = 'backbone and resid 122-159'
     NMP = 'backbone and resid 30-59'
```

```
[5]: R = rms.RMSD(u,  # universe to align
                  u,  # reference universe or atomgroup
                  select='backbone',  # group to superimpose and calculate RMSD
                  groupselections=[CORE, LID, NMP],  # groups for RMSD
                  ref_frame=0)  # frame index of the reference
     R.run()
```

```
[5]: <MDAnalysis.analysis.rms.RMSD at 0x7f2e830164f0>
```

The data is saved in R.rmsd as an array.

```
[6]: R.results.rmsd.shape
```

```
[6]: (98, 6)
```

R.rmsd has a row for each timestep. The first two columns of each row are the frame index of the time step, and the time (which is guessed in trajectory formats without timesteps). The third column is RMSD of select. The last few columns are the RMSD of the groups in groupselections.

Plotting the data

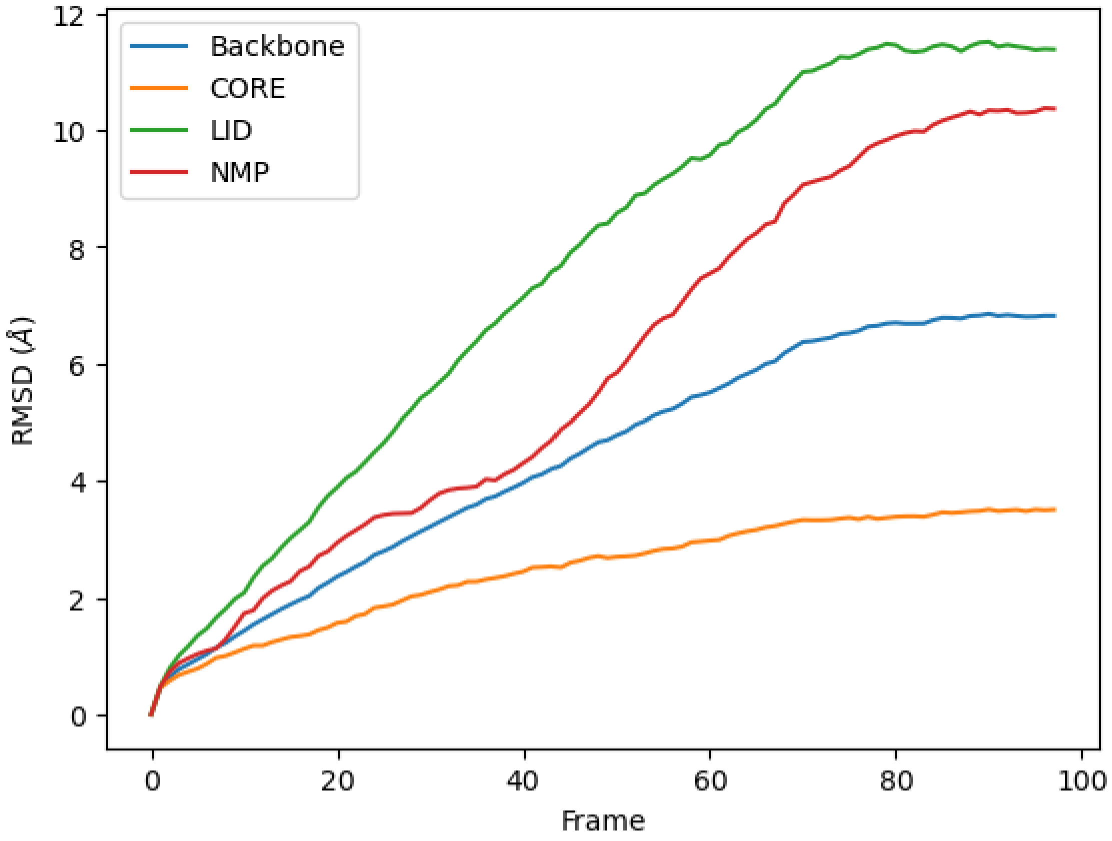
We can easily plot this data using the common data analysis package pandas. We turn the `R.rmsd` array into a DataFrame and label each column below.

```
[7]: df = pd.DataFrame(R.results.rmsd,
                       columns=['Frame', 'Time (ns)',
                                'Backbone', 'CORE',
                                'LID', 'NMP'])

     df
```

```
[7]:     Frame  Time (ns)      Backbone          CORE           LID           NMP
     0     0.0   1.000000  5.834344e-07  3.921486e-08  1.197000e-07  6.276497e-08
     1     1.0   2.000000  4.636592e-01  4.550181e-01  4.871915e-01  4.745572e-01
     2     2.0   3.000000  6.419340e-01  5.754418e-01  7.940994e-01  7.270191e-01
     3     3.0   4.000000  7.743983e-01  6.739184e-01  1.010261e+00  8.795031e-01
     4     4.0   5.000000  8.588600e-01  7.318859e-01  1.168397e+00  9.612989e-01
     ..    ...        ...           ...           ...           ...           ...
     93   93.0  93.999992  6.817898e+00  3.504430e+00  1.143376e+01  1.029266e+01
     94   94.0  94.999992  6.804211e+00  3.480681e+00  1.141134e+01  1.029879e+01
     95   95.0  95.999992  6.807987e+00  3.508946e+00  1.137593e+01  1.031958e+01
     96   96.0  96.999991  6.821205e+00  3.498081e+00  1.139156e+01  1.037768e+01
     97   97.0  97.999991  6.820322e+00  3.507119e+00  1.138474e+01  1.036821e+01

     [98 rows x 6 columns]
```

```
[8]: ax = df.plot(x='Frame', y=['Backbone', 'CORE', 'LID', 'NMP'],
                  kind='line')
     ax.set_ylabel(r'RMSD ($\AA$)')
```

```
[8]: Text(0, 0.5, 'RMSD ($\\AA$)')
```



RMSD of an AtomGroup with multiple selections

The `RMSD` class accepts both `AtomGroups` and `Universes` as arguments. Restricting the atoms considered to an AtomGroup can be very helpful, as the `select` and `groupselections` arguments are applied only to the atoms in the original `AtomGroup`. In the example below, for example, only the alpha-carbons of the CORE domain are incorporated in the analysis.

Note

This feature does not currently support `groupselections`.

```
[9]: ca = u.select_atoms('name CA')

     R = rms.RMSD(ca, ca, select=CORE, ref_frame=0)
     R.run()
```

```
[9]: <MDAnalysis.analysis.rms.RMSD at 0x7f2e80561610>
```

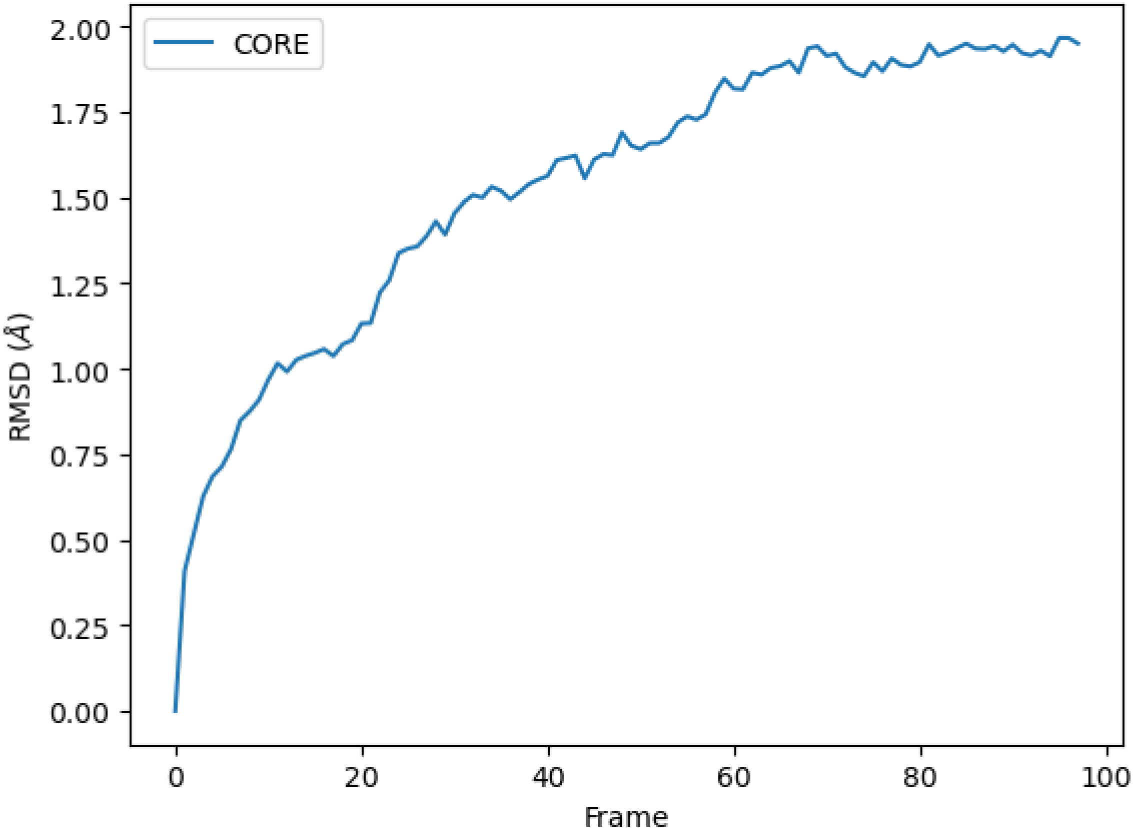
```
[10]: df = pd.DataFrame(R.results.rmsd,
                        columns=['Frame', 'Time (ns)',
                                 'CORE'])
```

(continues on next page)

(continued from previous page)

```
ax = df.plot(x='Frame', y='CORE', kind='line')
ax.set_ylabel('RMSD ($\AA$)')
```

```
[10]: Text(0, 0.5, 'RMSD ($\\AA$)')
```



Weighted RMSD of a trajectory

You can also calculate the weighted RMSD of a trajectory using the `weights` and `weights_groupselections` keywords. The former only applies weights to the group in `select`, while the latter must be a list of the *same length and order* as `groupselections`. If you would like to only weight certain groups in `groupselections`, use `None` for the unweighted groups. Both `weights` and `weights_groupselections` accept `None` (for unweighted), `'mass'` (to weight by mass), and custom arrays. A custom array should have the same number of values as there are particles in the corresponding AtomGroup.

Mass

It is common to weight RMSD analyses by particle weight, so the RMSD class accepts 'mass' as an argument for weights. Note that the weights keyword only applies to the group in select, and does not apply to the groupselections; these remain unweighted below.

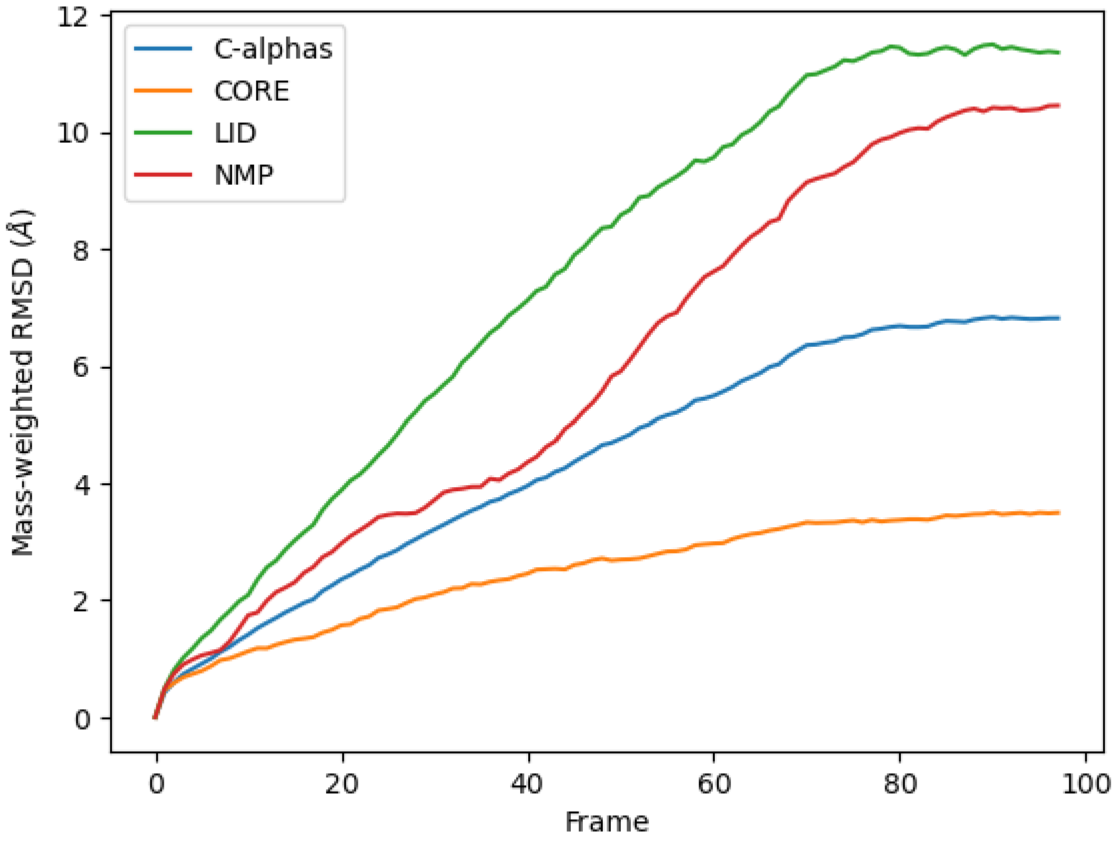
```
[11]: R_mass = rms.RMSD(u, u,
                        select='protein and name CA',
                        weights='mass',
                        groupselections=[CORE, LID, NMP])
      R_mass.run()
```

```
[11]: <MDAnalysis.analysis.rms.RMSD at 0x7f2e804833d0>
```

The plot looks largely the same as above because all the alpha-carbons and individual backbone atoms have the same mass. Below we show how you can pass in custom weights.

```
[12]: df_mass = pd.DataFrame(R_mass.results.rmsd,
                             columns=['Frame',
                                      'Time (ns)',
                                      'C-alphas', 'CORE',
                                      'LID', 'NMP'])
      ax_mass = df_mass.plot(x='Frame',
                             y=['C-alphas', 'CORE', 'LID', 'NMP'])
      ax_mass.set_ylabel('Mass-weighted RMSD ($\AA$)')
```

```
[12]: Text(0, 0.5, 'Mass-weighted RMSD ($\\AA$)')
```



Custom weights

You can also pass in an array of values for the weights. This must have the same length as the number of atoms in your selection groups. In this example, we pass in arrays of residue numbers. This **is not a typical weighting you might choose**, but we use it here to show how you can get different results to the previous graphs for mass-weighted and non-weighted RMSD. You could choose to pass in an array of charges, or your own custom value.

First we select the atom groups to make this easier.

[13]:

```
ag = u.select_atoms('protein and name CA')
print('Shape of C-alpha charges:', ag.charges.shape)
core = u.select_atoms(CORE)
lid = u.select_atoms(LID)
nmp = u.select_atoms(NMP)
```

```
Shape of C-alpha charges: (214,)
```

Below, we pass in weights for `ag` with the `weights` keyword, and weights for the `groupselections` with the `weights_groupselections` keyword.

[14]:

```
R_charge = rms.RMSD(u, u,
                    select='protein and name CA',
                    groupselections=[CORE, LID, NMP],
                    weights=ag.resids,
                    weights_groupselections=[core.resids,
                                             lid.resids,
                                             nmp.resids])
R_charge.run()
```

[14]:

```
<MDAnalysis.analysis.rms.RMSD at 0x7f2e80483490>
```

You can see in the below graph that the NMP domain has much higher RMSD than before, as each particle is weighted by its residue number. Conversely, the CORE domain doesn't really change much. We could potentially infer from this that residues later in the NMP domain (with a higher residue number) are mobile during the length of the trajectory, whereas residues later in the CORE domain do not seem to contribute significantly to the RMSD.

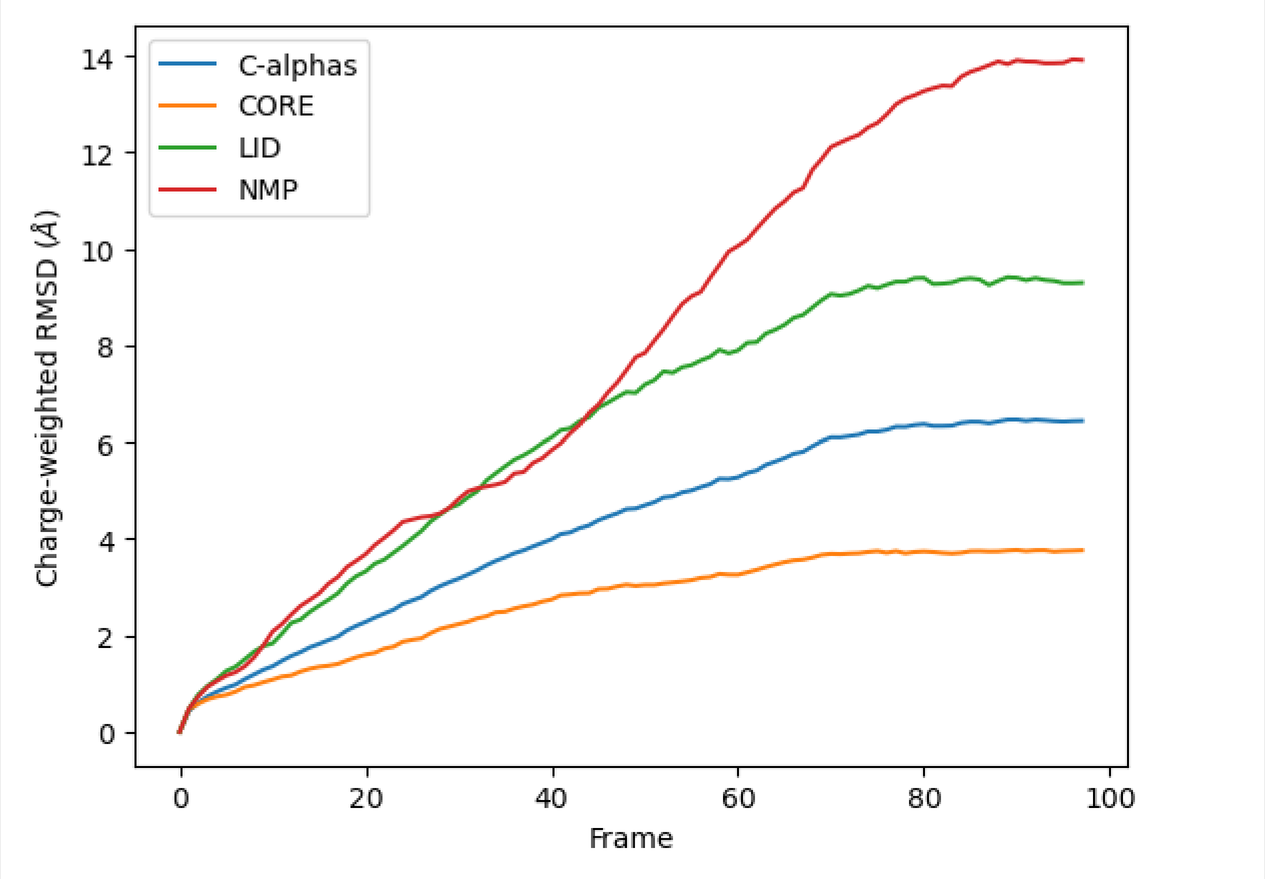
However, again, this is a **very non-conventional metric** and is shown here simply to demonstrate how to use the code.

[15]:

```
df_charge = pd.DataFrame(R_charge.results.rmsd,
                         columns=['Frame',
                                  'Time (ns)',
                                  'C-alphas', 'CORE',
                                  'LID', 'NMP'])
ax_charge = df_charge.plot(x='Frame',
                           y=['C-alphas', 'CORE', 'LID', 'NMP'])
ax_charge.set_ylabel('Charge-weighted RMSD ($\AA$)')
```

[15]:

```
Text(0, 0.5, 'Charge-weighted RMSD ($\\AA$)')
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallographica Section A Foundations of Crystallography, 61(4):478–480, July 2005. 00127. URL: http://scripts.iucr.org/cgi-bin/paper?S0108767305015266, doi:10.1107/S0108767305015266.

Calculating the pairwise RMSD of a trajectory

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.17.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- matplotlib

See also

- *1D RMSD*
- *RMSF*

Note

MDAnalysis implements RMSD calculation using the fast QCP algorithm ([The05]). Please cite ([The05]) when using the `MDAnalysis.analysis.align` module in published work.

```
[1]: import MDAnalysis as mda
     from MDAnalysis.tests.datafiles import PSF, DCD, CRD, DCD2
     from MDAnalysis.analysis import diffusionmap, align, rms
     import numpy as np
     import matplotlib.pyplot as plt
     %matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectories sample a transition from a closed to an open conformation.

```
[2]: adk_open = mda.Universe(CRD, DCD2)
     adk_closed = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Background

While *1-dimensional RMSD* is a quick way to estimate how much a structure changes over time, it can be a misleading measure. It is easy to think that two structures with the same RMSD from a reference frame are also similar; but in fact, they can be very different. Instead, calculating the RMSD of each frame in the trajectory to all other frames in the other trajectory can contain much more information. This measure is often called the pairwise, all-to-all, or 2D RMSD.

The other, or reference, trajectory in pairwise RMSD can either be the first trajectory, or another one. If the pairwise RMSD of a trajectory is calculated to itself, it can be used to gain insight into the conformational convergence of the simulation. The diagonal of the plot will be zero in this case (as this represents the RMSD of a structure to itself). Blocks of low RMSD values *along* the diagonal indicate similar structures, suggesting the occupation of a given state. Blocks of low RMSD values *off* the diagonal indicate that the trajectory is revisiting an earlier state. Please see the living guide Best Practices for Quantification of Uncertainty and Sampling Quality in Molecular Simulations by Grossfield et al. for more on using 2D RMSD as a measure of convergence. MDAnalysis provides a *DistanceMatrix* class for easy calculation of the pairwise RMSD of a trajectory to itself.

When the other trajectory in pairwise RMSD is a different trajectory, the pairwise RMSD can be used to compare the similarity of the two conformational ensembles. There is no requirement that the two trajectories be the same length. In this case, the diagonal is no longer necessarily zero. Blocks of low RMSD values anywhere indicate that the two trajectories are sampling similar states. *Pairwise RMSDs with different trajectories must be manually calculated in MDAnalysis*.

Pairwise RMSD of a trajectory to itself

In order to calculate the pairwise RMSD of a trajectory to itself, you should begin by aligning the trajectory in order to minimise the resulting RMSD. You may not have enough memory to do this `in_memory`, in which case you can write out the aligned trajectory to a file (please see *the aligning tutorials* for more).

```
[3]: aligner = align.AlignTraj(adk_open, adk_open, select='name CA',
                               in_memory=True).run()
```

We can then calculate a pairwise RMSD matrix with the diffusionmap.DistanceMatrix class, by using the default the rms.rmsd metric.

```
[4]: matrix = diffusionmap.DistanceMatrix(adk_open, select='name CA').run()
```

The results array is in `results.matrix.dist_matrix` as a square array with the shape (#n_frames, #n_frame).

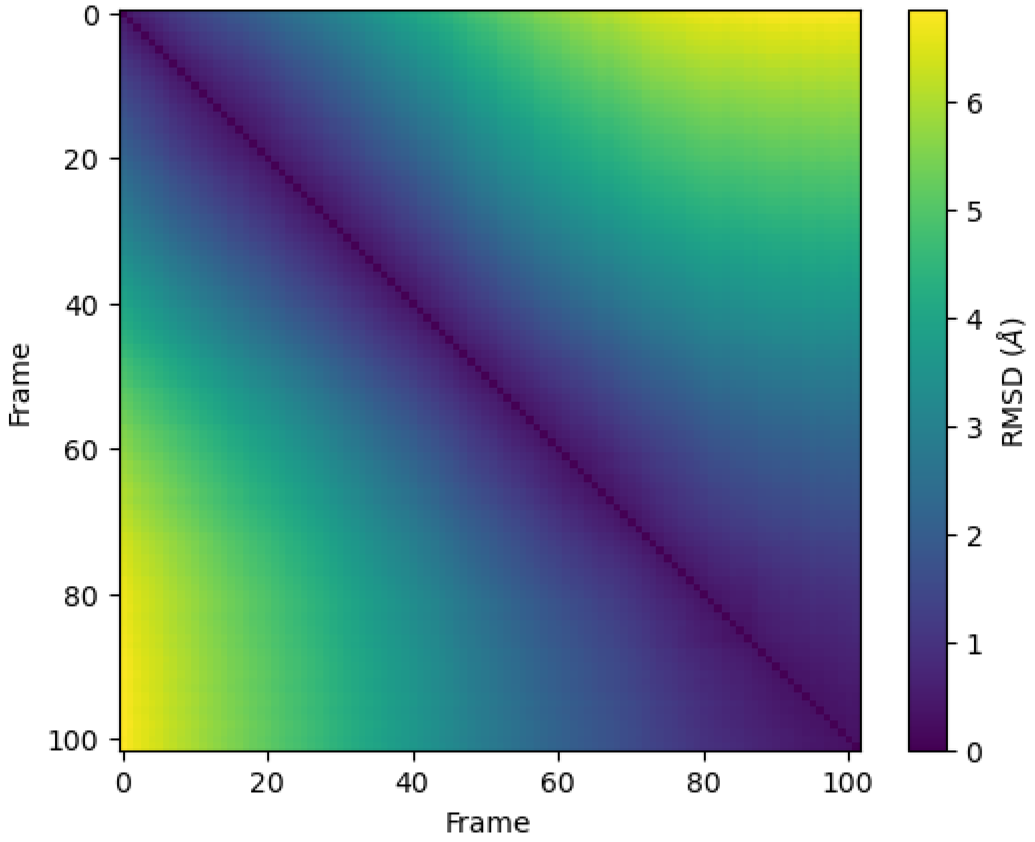
```
[5]: matrix.results.dist_matrix.shape
```

```
[5]: (102, 102)
```

We can use the common plotting package matplotlib to create a heatmap from this array. For other ways to plot heat maps, you can look at the seaborn, plotly (for interactive images), or holoviews (also interactive) packages.

```
[6]: plt.imshow(matrix.results.dist_matrix, cmap='viridis')
     plt.xlabel('Frame')
     plt.ylabel('Frame')
     plt.colorbar(label=r'RMSD ($\AA$)')
```

```
[6]: <matplotlib.colorbar.Colorbar at 0x7fd0e5ddcc40>
```



Pairwise RMSD between two trajectories

Calculating the 2D RMSD between two trajectories is a bit more finicky; `DistanceMatrix` can only calculate the RMSD of a trajectory to itself. Instead, we do it the long way by simply calculating the RMSD of each frame in the second trajectory, to each frame in the first trajectory.

First we set up a 2D numpy array a shape corresponding to the length of each of our trajectories to store our results. To start off, it is populated with zeros.

[7]:

```
prmsd = np.zeros((len(adk_open.trajectory),  # y-axis
                  len(adk_closed.trajectory)))  # x-axis
```

Then we iterate through each frame of the `adk_open` trajectory (our y-axis), and calculate the RMSD of `adk_closed` to each frame, storing it in the `prmsd` array.

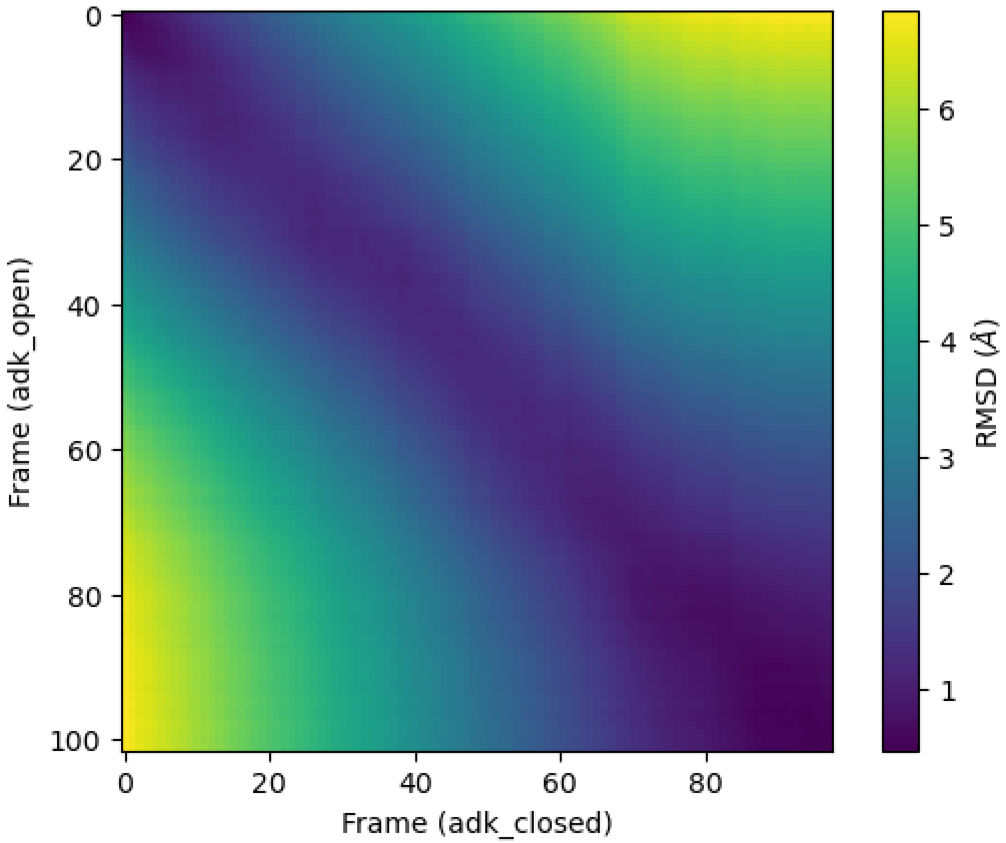
[8]:

```
for i, frame_open in enumerate(adk_open.trajectory):
    r = rms.RMSD(adk_closed, adk_open, select='name CA',
                 ref_frame=i).run()
    prmsd[i] = r.results.rmsd[:, -1]  # select 3rd column with RMSD values
```

We plot it below. As you can see, there is no longer a line of zero values across the diagonal. Here, the frames of `adk_closed` and `adk_open` are similar but not identical.

```
[9]: plt.imshow(prmsd, cmap='viridis')
     plt.xlabel('Frame (adk_closed)')
     plt.ylabel('Frame (adk_open)')
     plt.colorbar(label=r'RMSD ($\AA$)')
```

```
[9]: <matplotlib.colorbar.Colorbar at 0x7fd0e5c7a850>
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallographica Section A Foundations of Crystallography, 61(4):478–480, July 2005. 00127. URL: http://scripts.

iucr.org/cgi-bin/paper?S0108767305015266, doi:10.1107/S0108767305015266.

Calculating the root mean square fluctuation over a trajectory

We calculate the RMSF of the alpha-carbons in adenylate kinase (AdK) as it transitions from an open to closed structure, with reference to the average conformation of AdK.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisData
- matplotlib

Optional packages for visualisation: * nglview

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

See also

- *RMSD*
- *Pairwise (2D) RMSD*

Note

MDAnalysis implements RMSD calculation using the fast QCP algorithm ([The05]) and a rotation matrix *R* that minimises the RMSD ([LAT09]). Please cite ([The05]) and ([LAT09]) when using the `MDAnalysis.analysis.align` and `MDAnalysis.analysis.rms` modules in published work.

[1]:

```
import MDAnalysis as mda
from MDAnalysisData import datasets
from MDAnalysis.analysis import rms, align
# import nglview as nv

import warnings
# suppress some MDAnalysis warnings about writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

The test files we will be working with here are an equilibrium trajectory of adenylate kinase (AdK), a phosophotransferase enzyme. ([SB17]) AdK has three domains:

- CORE
- LID: an ATP-binding domain
- NMP: an AMP-binding domain

The LID and NMP domains move around the stable CORE as the enzyme transitions between the opened and closed conformations. We therefore might wonder whether the LID and NMP residues are more mobile than the CORE residues. One way to quantify this flexibility is by calculating the root mean square fluctuation (RMSF) of atomic positions.

Note: downloading these datasets from MDAnalysisData may take some time.

```
[2]: adk = datasets.fetch_adk_equilibrium()
```

```
[3]: u = mda.Universe(adk.topology, adk.trajectory)
```

Background

The root-mean-square-fluctuation (RMSF) of a structure is the time average of the *RMSD*. It is calculated according to the below equation, where $\mathbf{x}_i$ is the coordinates of particle i, and $\langle \mathbf{x}_i \rangle$ is the ensemble average position of i.

$$\rho_i = \sqrt{\langle (\mathbf{x}_i - \langle \mathbf{x}_i \rangle)^2 \rangle}$$

Where the RMSD quantifies how much a structure diverges from a reference over time, the RSMF can reveal which areas of the system are the most mobile. While RMSD is frequently calculated to an initial state, the RMSF should be calculated to an average structure of the simulation. An area of the structure with high RMSF values frequently diverges from the average, indicating high mobility. When RMSF analysis is carried out on proteins, it is typically restricted to backbone or alpha-carbon atoms; these are more characteristic of conformational changes than the more flexible side-chains.

Creating an average structure

We can generate an average structure to align to with the `align.AverageStructure` class. Here we first align to the first frame (`ref_frame=0`), and then average the coordinates.

```
[4]: average = align.AverageStructure(u, u, select='protein and name CA',
                                      ref_frame=0).run()
     ref = average.results.universe
```

Aligning the trajectory to a reference

`rms.RMSF` does not allow on-the-fly alignment to a reference, and presumes that you have already aligned the trajectory. Therefore we need to first align our trajectory to the average conformation.

```
[5]: aligner = align.AlignTraj(u, ref,
                               select='protein and name CA',
                               in_memory=True).run()
```

You may not have enough memory to load the trajectory into memory. In that case, save this aligned trajectory to a file and re-load it into MDAnalysis by uncommenting the code below.

```
[6]: # aligner = align.AlignTraj(u, ref,
     #                           select='protein and name CA',
     #                           filename='aligned_traj.dcd',
     #                           in_memory=False).run()
     # u = mda.Universe(PSF, 'aligned_traj.dcd')
```

Calculating RMSF

The trajectory is now fitted to the reference, and the RMSF (API docs) can be calculated.

Note

MDAnalysis implements an algorithm that computes sums of squares and avoids underflows or overflows. Please cite ([Wel62]) when using the `MDAnalysis.analysis.rms.RMSF` class in published work.

[7]:

```
c_alphas = u.select_atoms('protein and name CA')
R = rms.RMSF(c_alphas).run()
```

Plotting RMSF

We can now plot the RMSF using the common plotting package matplotlib.

[8]:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

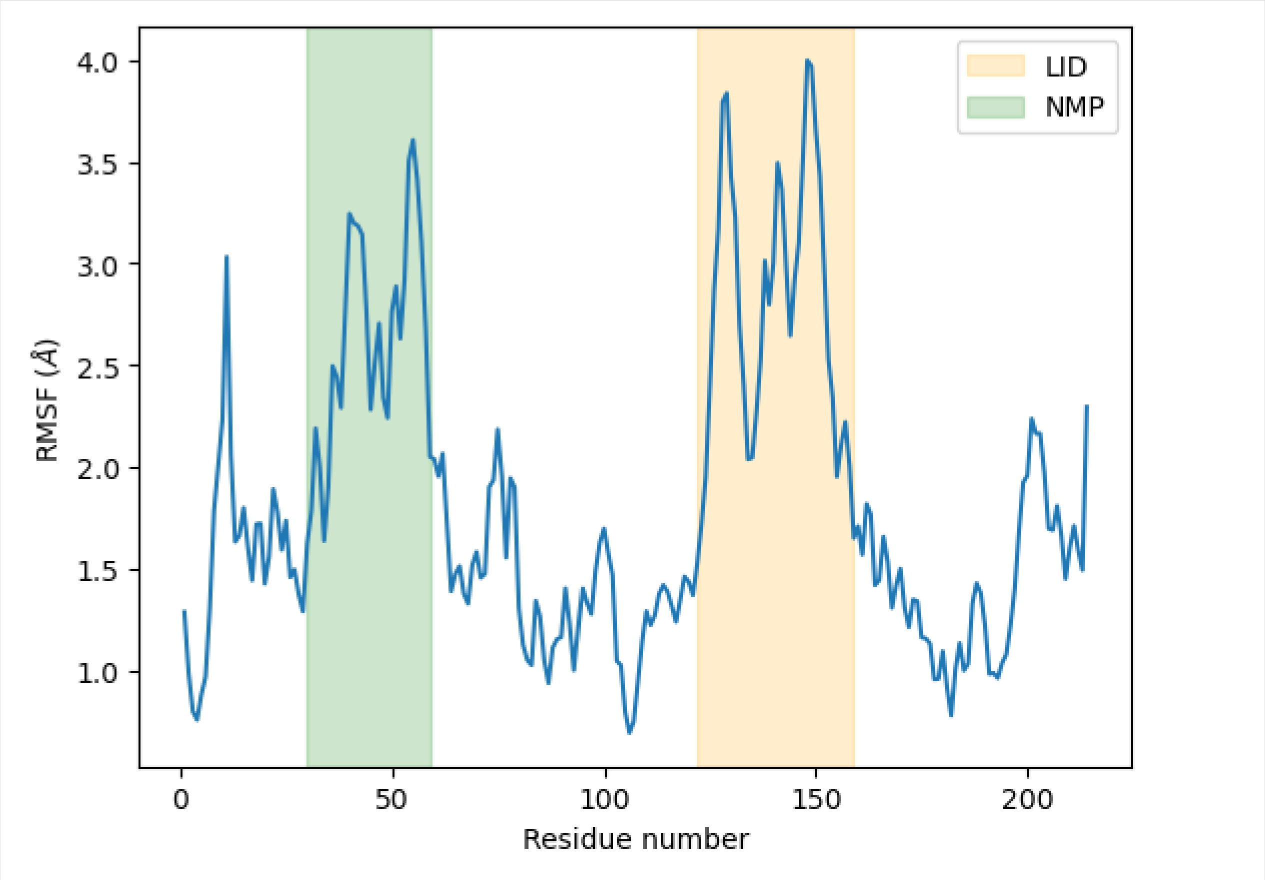
As we can see, the LID and NMP residues indeed move much more compared to the rest of the enzyme.

[9]:

```
plt.plot(c_alphas.resids, R.results.rmsf)
plt.xlabel('Residue number')
plt.ylabel('RMSF ($\AA$)')
plt.axvspan(122, 159, zorder=0, alpha=0.2, color='orange', label='LID')
plt.axvspan(30, 59, zorder=0, alpha=0.2, color='green', label='NMP')
plt.legend()
```

[9]:

```
<matplotlib.legend.Legend at 0x13a476d30>
```



Visualising RMSF as B-factors

Colouring a protein by RMSF allows you to visually identify regions of high fluctuation. This is commonly done by setting temperature factor (also known as b-factor) values, writing out to a format with B-factor specification (e.g. PDB), and visualising the file in a program such as VMD or nglview.

MDAnalysis uses the `tempfactor` topology attribute for this information. Below, we iterate through each residue of the protein and set the `tempfactor` attribute for *every* atom in the residue to the alpha-carbon RMSF value; this is necessary so every atom in the residue is coloured with the alpha-carbon RMSF.

[10]:
```
u.add_TopologyAttr('tempfactors') # add empty attribute for all atoms
protein = u.select_atoms('protein') # select protein atoms
for residue, r_value in zip(protein.residues, R.results.rmsf):
    residue.atoms.tempfactors = r_value
```

Below we visualise these values with a rainbow colour scheme. Purple values correspond to low RMSF values, whereas red values correspond to high RMSFs.

[11]:
```
# view = nv.show_mdanalysis(u)
# view.update_representation(color_scheme='bfactor')
# view
```

[12]:
```
# from nglview.contrib.movie import MovieMaker
# movie = MovieMaker(
```

(continues on next page)

(continued from previous page)

```
#     view,
#     step=100,  # keep every 100th frame
#     output='images/rmsf-view.gif',
#     render_params={"factor": 3},  # set to 4 for highest quality
# )
# movie.make()
```

You can also write the atoms to a file and visualise it in another program. As the original Universe did not contain `altLocs`, `icodes` or `occupancies` for each atom, some warnings will be printed (which are not visible here).

```
[13]: u.atoms.write('rmsf_tempfactors.pdb')
```

References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Pu Liu, Dimitris K. Agrafiotis, and Douglas L. Theobald. Fast determination of the optimal rotational matrix for macromolecular superpositions. Journal of Computational Chemistry, pages n/a–n/a, 2009. URL: http://doi.wiley.com/10.1002/jcc.21439, doi:10.1002/jcc.21439.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Sean Seyler and Oliver Beckstein. Molecular dynamics trajectory for benchmarking MDAnalysis. June 2017. 00002. URL: https://figshare.com/articles/Molecular_dynamics_trajectory_for_benchmarking_MDAnalysis/5108170, doi:10.6084/m9.figshare.5108170.v1.

[5] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallographica Section A Foundations of Crystallography, 61(4):478–480, July 2005. 00127. URL: http://scripts.iucr.org/cgi-bin/paper?S0108767305015266, doi:10.1107/S0108767305015266.

[6] B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. Technometrics, 4(3):419–420, August 1962. URL: https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022, doi:10.1080/00401706.1962.10490022.

Distances and contacts

The `MDAnalysis.analysis.distances` module provides functions to rapidly compute distances. These largely take in coordinate arrays.

Atom-wise distances between matching AtomGroups

Here we compare the distances between alpha-carbons of the enzyme adenylate kinase in its open and closed conformations. `distances.dist` can be used to calculate distances between atom groups with the *same number of atoms* within them.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.19.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

[1]:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PDB_small, PDB_closed
from MDAnalysis.analysis import distances

import matplotlib.pyplot as plt
%matplotlib inline

import warnings
# suppress some MDAnalysis warnings when writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) AdK has three domains:

- CORE
- LID: an ATP-binding domain (residues 122-159)
- NMP: an AMP-binding domain (residues 30-59)

The LID and NMP domains move around the stable CORE as the enzyme transitions between the opened and closed conformations.

[2]:

```
u1 = mda.Universe(PDB_small)   # open AdK
u2 = mda.Universe(PDB_closed)  # closed AdK
```

Calculating the distance between CA atoms

We select the atoms named 'CA' of each Universe.

```
[3]: ca1 = u1.select_atoms('name CA')
     ca2 = u2.select_atoms('name CA')
```

distances.dist(API docs) returns the residue numbers of both selections given. The offset keyword adds an offset to these residue numbers to help with comparison to each other and other file formats. Here we are happy with our residue numbers, so we use the default offset of 0. (See the documentation of distances.dist for more information.)

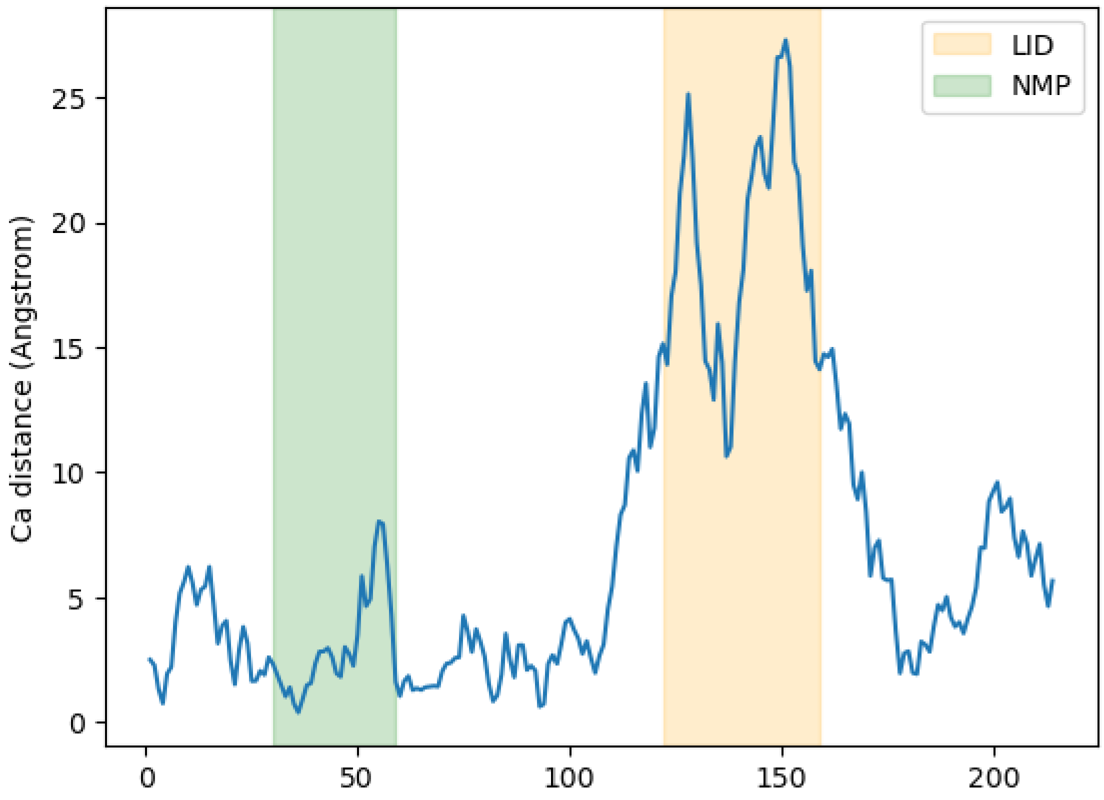
```
[4]: resids1, resids2, dist = distances.dist(ca1, ca2,
                                             offset=0)  # for residue numbers
```

Plotting

Below, we plot the distance over the residue numbers and highlight the LID and NMP domains of the protein. The LID domain in particular moves a significant distance between its opened and closed conformations.

```
[8]: plt.plot(resids1, dist)
     plt.ylabel('Ca distance (Angstrom)')
     plt.axvspan(122, 159, zorder=0, alpha=0.2, color='orange', label='LID')
     plt.axvspan(30, 59, zorder=0, alpha=0.2, color='green', label='NMP')
     plt.legend()
```

```
[8]: <matplotlib.legend.Legend at 0x7f06d2a3b7c0>
```



Calculating the distance with periodic boundary conditions

It is common to want to calculate distances with the minimum image convention. To do this, you *must* pass the unitcell dimensions of the system to the box keyword, **even if your Universe has dimensions defined**.

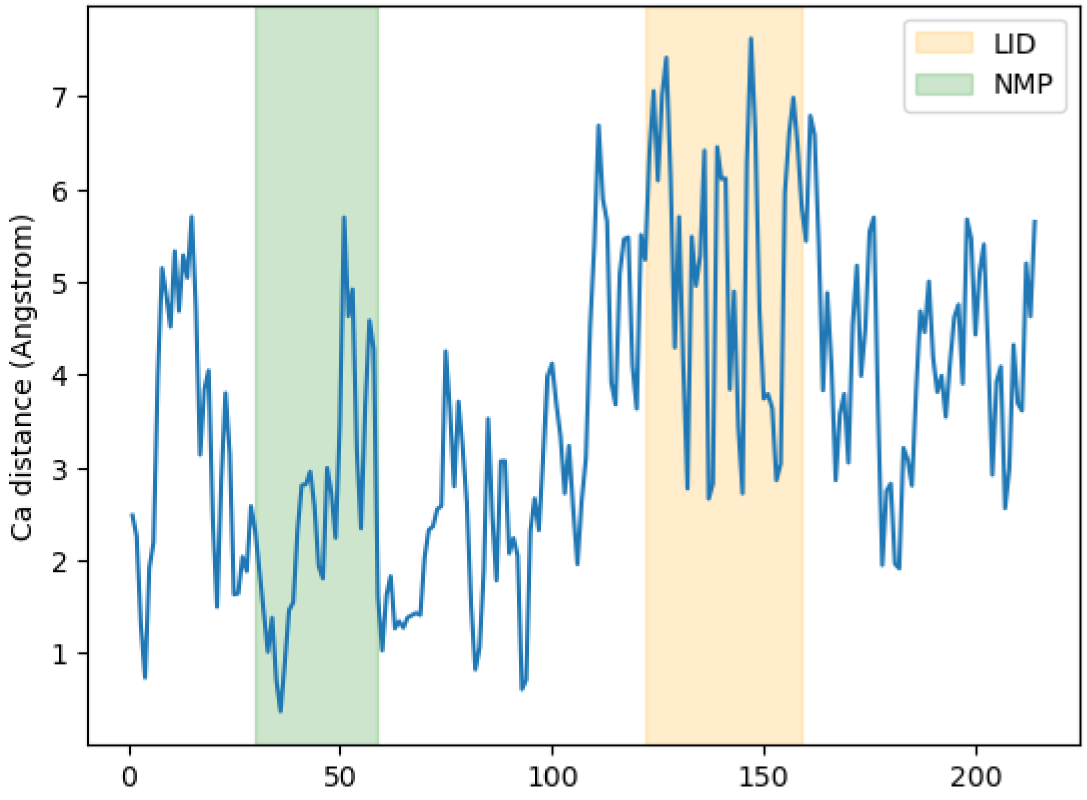
This should have the format: `[lx, ly, lz, alpha, beta, gamma]`, where the first three numbers are the box lengths along each axis and the last three are the angles of the box.

```
[6]: resids1_box, resids2_box, dist_box = distances.dist(ca1, ca2,
                                                          box=[10, 10, 10, 90, 90, 90])
```

Plotting

```
[7]: plt.plot(resids1_box, dist_box)
     plt.ylabel('Ca distance (Angstrom)')
     plt.axvspan(122, 159, zorder=0, alpha=0.2, color='orange', label='LID')
     plt.axvspan(30, 59, zorder=0, alpha=0.2, color='green', label='NMP')
     plt.legend()
```

```
[7]: <matplotlib.legend.Legend at 0x7f06d09bc700>
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

All distances between two selections

Here we use `distances.distance_array` to quantify the distances between each atom of a target set to each atom in a reference set, and show how we can extend that to calculating the distances between the centers-of-mass of residues.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.19.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PDB_small
from MDAnalysis.analysis import distances

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
# suppress some MDAnalysis warnings when writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) AdK has three domains:

- CORE
- LID: an ATP-binding domain (residues 122-159)
- NMP: an AMP-binding domain (residues 30-59)

[2]:
```
u = mda.Universe(PDB_small)   # open AdK
```

Calculating atom-to-atom distances between non-matching coordinate arrays

We select the alpha-carbon atoms of each domain.

[3]:
```
LID_ca = u.select_atoms('name CA and resid 122-159')
NMP_ca = u.select_atoms('name CA and resid 30-59')

n_LID = len(LID_ca)
n_NMP = len(NMP_ca)
print('LID has {} residues and NMP has {} residues'.format(n_LID, n_NMP))
```

```
LID has 38 residues and NMP has 30 residues
```

`distances.distance_array`(API docs) will produce an array with shape `(n, m)` distances if there are `n` positions in the reference array and `m` positions in the other configuration. If you want to calculate distances following the minimum image convention, you *must* pass the universe dimensions into the `box` keyword.

[4]:
```
dist_arr = distances.distance_array(LID_ca.positions, # reference
                                    NMP_ca.positions, # configuration
                                    box=u.dimensions)
dist_arr.shape
```

[4]:
```
(38, 30)
```

Plotting distance as a heatmap

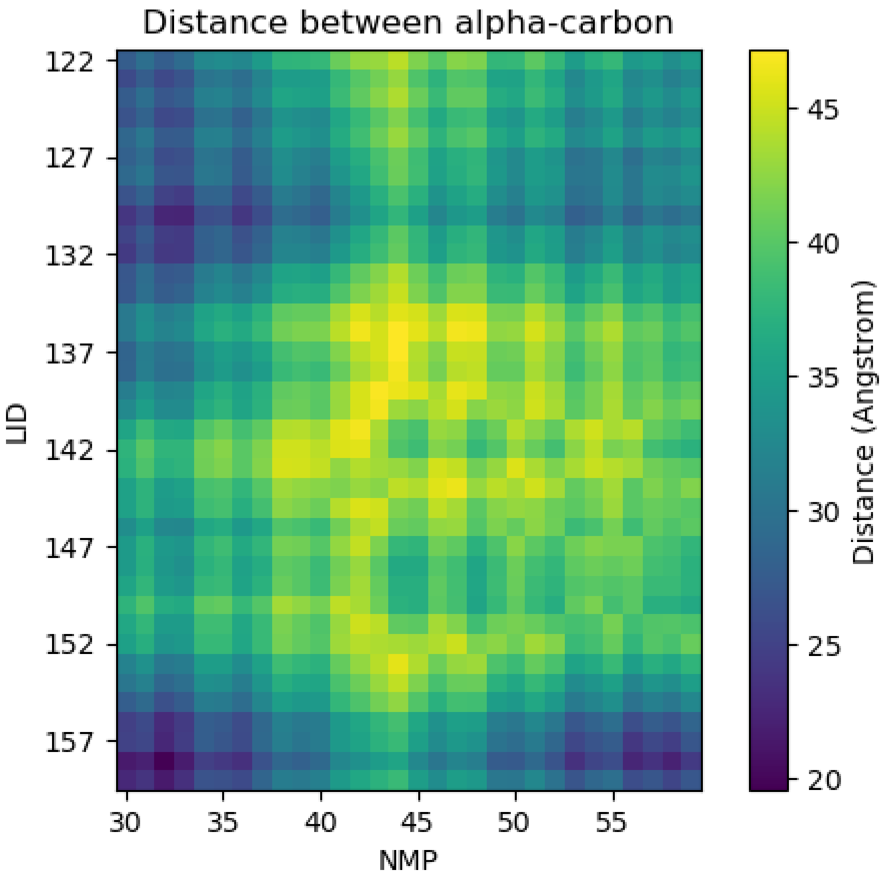
[5]:
```
fig, ax = plt.subplots()
im = ax.imshow(dist_arr, origin='upper')

# add residue ID labels to axes
tick_interval = 5
ax.set_yticks(np.arange(n_LID)[::tick_interval])
ax.set_xticks(np.arange(n_NMP)[::tick_interval])
ax.set_yticklabels(LID_ca.resids[::tick_interval])
ax.set_xticklabels(NMP_ca.resids[::tick_interval])

# add figure labels and titles
plt.ylabel('LID')
plt.xlabel('NMP')
plt.title('Distance between alpha-carbon')
```

(continues on next page)

(continued from previous page)

```
# colorbar
cbar = fig.colorbar(im)
cbar.ax.set_ylabel('Distance (Angstrom)')
```

[5]:
```
Text(0, 0.5, 'Distance (Angstrom)')
```



Calculating residue-to-residue distances

As `distances.distance_array` just takes coordinate arrays as input, it is very flexible in calculating distances between each atom, or centers-of-masses, centers-of-geometries, and so on.

Instead of calculating the distance between the alpha-carbon of each residue, we could look at the distances between the centers-of-mass instead. The process is very similar to the atom-wise distances above, but we give `distances.distance_array` an array of residue center-of-mass coordinates instead.

[6]:
```
LID = u.select_atoms('resid 122-159')
NMP = u.select_atoms('resid 30-59')

LID_com = LID.center_of_mass(compound='residues')
NMP_com = NMP.center_of_mass(compound='residues')

n_LID = len(LID_com)
```

(continues on next page)

(continued from previous page)

```
n_NMP = len(NMP_com)

print('LID has {} residues and NMP has {} residues'.format(n_LID, n_NMP))
```

```
LID has 38 residues and NMP has 30 residues
```

We can pass these center-of-mass arrays directly into `distances.distance_array`.

[7]:

```
res_dist = distances.distance_array(LID_com, NMP_com,
                                    box=u.dimensions)
```

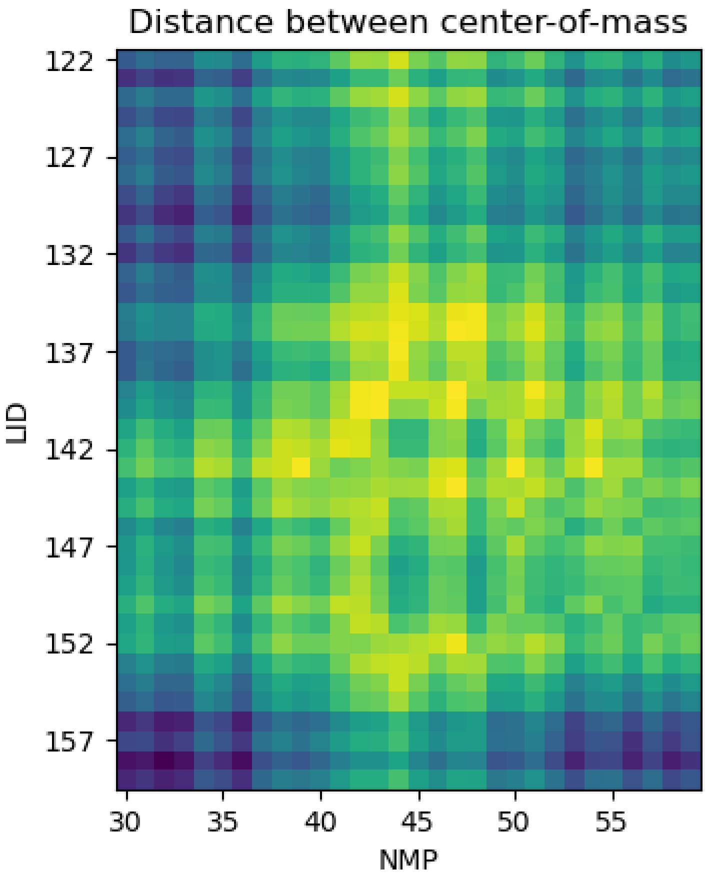
Plotting

[8]:

```
fig2, ax2 = plt.subplots()
im2 = ax2.imshow(res_dist, origin='upper')

# add residue ID labels to axes
tick_interval = 5
ax2.set_yticks(np.arange(n_LID)[::tick_interval])
ax2.set_xticks(np.arange(n_NMP)[::tick_interval])
ax2.set_yticklabels(LID.residues.resids[::tick_interval])
ax2.set_xticklabels(NMP.residues.resids[::tick_interval])

# add figure labels and titles
plt.ylabel('LID')
plt.xlabel('NMP')
plt.title('Distance between center-of-mass')

# colorbar
cbar2 = fig2.colorbar(im)
cbar2.ax.set_ylabel('Distance (Angstrom)')
```

[8]:

```
Text(0, 0.5, 'Distance (Angstrom)')
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

All distances within a selection

Here we calculate the distance of every atom to every other atom in a selection, and show how we can extend this to residues.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.19.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PDB_small
from MDAnalysis.analysis import distances

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
# suppress some MDAnalysis warnings when writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

[2]:
```
u = mda.Universe(PDB_small)
```

Calculating atom-wise distances

We begin by selecting the alpha-carbons of the protein.

[3]:
```
ca = u.select_atoms('name CA')
n_ca = len(ca)
n_ca
```

[3]:
```
214
```

When given an array with n positions, `distances.self_distance_array` (API docs) returns the distances in a flat vector with length $\frac{n(n-1)}{2}$. These correspond to the flattened upper triangular values of a square distance matrix.

[4]:
```
self_distances = distances.self_distance_array(ca.positions)
self_distances.shape
```

[4]:
```
(22791,)
```

We can convert this into a more easily interpreted square distance array. First we create an all-zero square array and get the indices of the upper and lower triangular matrices.

[5]:
```
sq_dist_arr = np.zeros((n_ca, n_ca))
triu = np.triu_indices_from(sq_dist_arr, k=1)
```

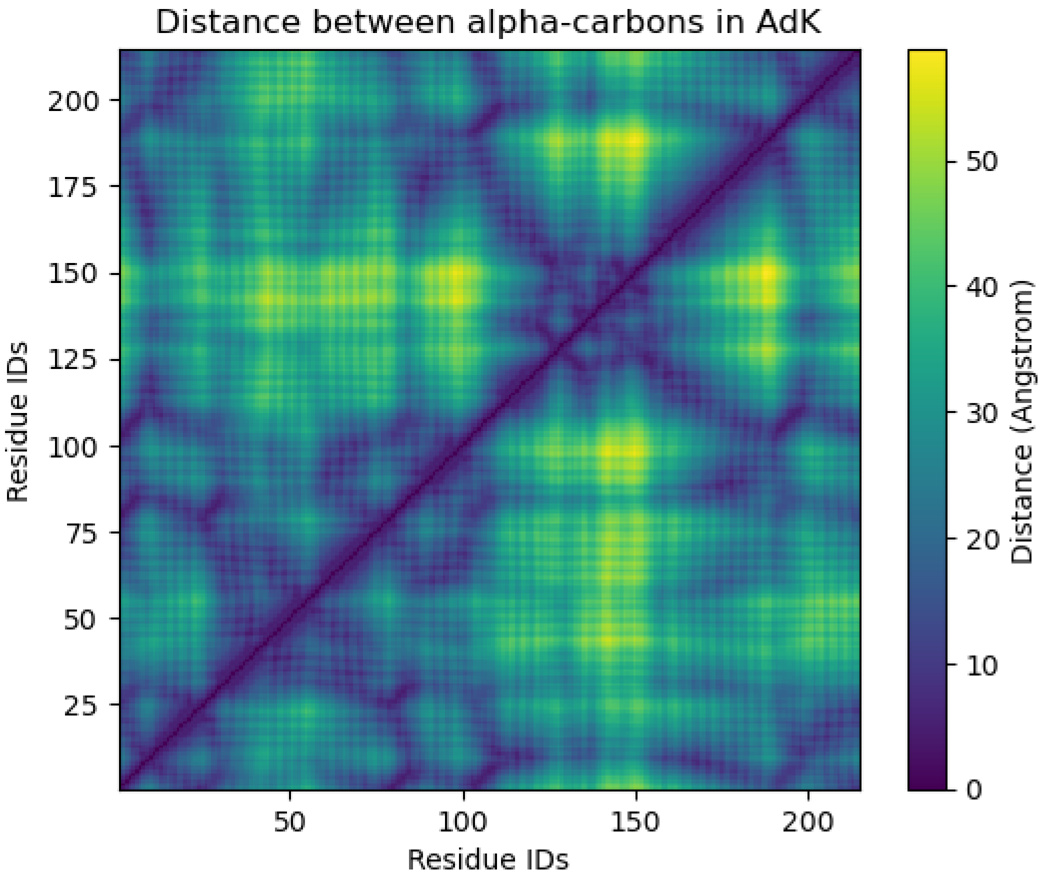
Then we simply assign the calculated distances to the upper and lower triangular positions.

[6]:
```
sq_dist_arr[triu] = self_distances
sq_dist_arr.T[triu] = self_distances
```

Plotting

[7]:
```
fig, ax = plt.subplots()
im = ax.pcolor(ca.resids, ca.resids, sq_dist_arr)

# plt.pcolor gives a rectangular grid by default
# so we need to make our heatmap square
ax.set_aspect('equal')

# add figure labels and titles
plt.ylabel('Residue IDs')
plt.xlabel('Residue IDs')
plt.title('Distance between alpha-carbons in AdK')

# colorbar
cbar = fig.colorbar(im)
cbar.ax.set_ylabel('Distance (Angstrom)')
```

[7]:
```
Text(0, 0.5, 'Distance (Angstrom)')
```



Calculating distances for each residue

Instead of calculating the distance between the alpha-carbon of each residue, we could look at the distances between the centers-of-mass instead. The process is very similar to the atom-wise distances above, but we have to pass `distances.self_distance_array` an array of residue center-of-mass coordinates instead.

[8]:

```
res_com = u.atoms.center_of_mass(compound='residues')
n_res = len(res_com)
n_res
```

[8]:

```
214
```

As the number of residues remains the same, the resulting distances array has the same length.

[9]:

```
res_dist = distances.self_distance_array(res_com)
res_dist.shape
```

[9]:

```
(22791,)
```

This means we don't need to re-define `triu`.

[10]:

```
sq_dist_res = np.zeros((n_res, n_res))
sq_dist_res[triu] = res_dist
sq_dist_res.T[triu] = res_dist
```

Plotting

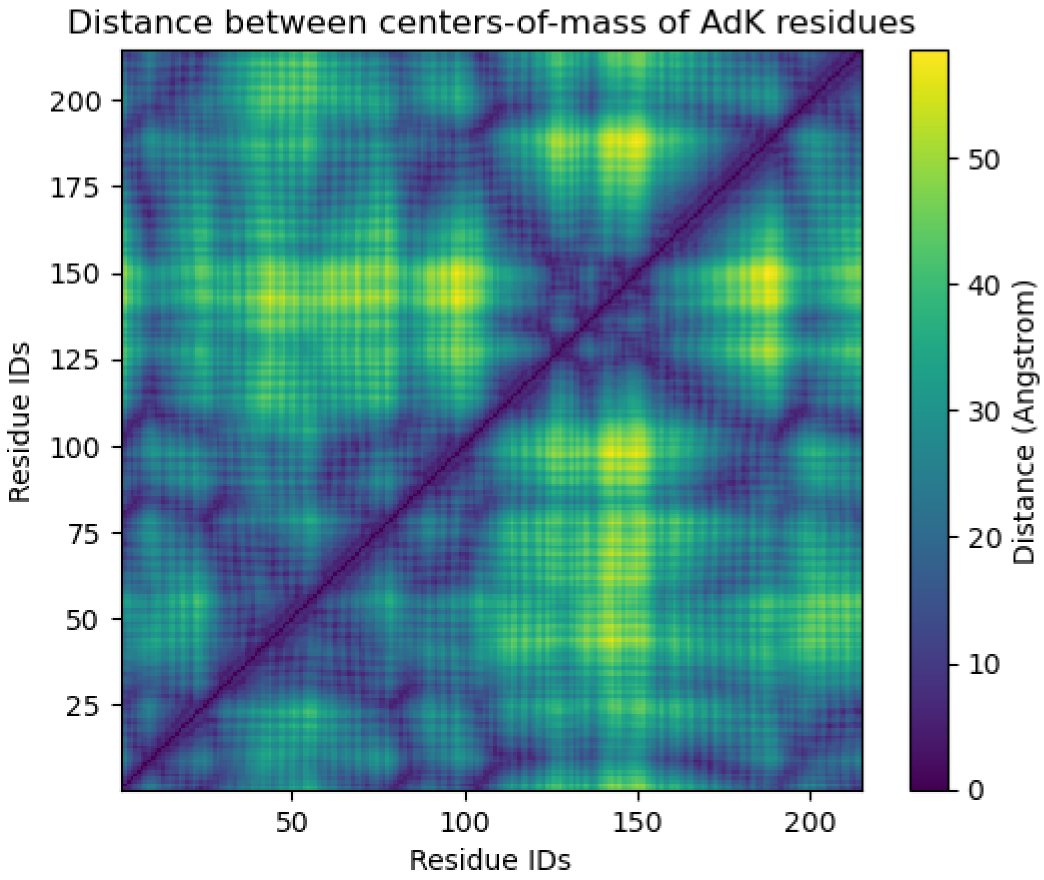
The resulting plot looks pretty similar.

[11]:

```
fig2, ax2 = plt.subplots()
im2 = ax2.pcolor(u.residues.resids, u.residues.resids, sq_dist_res)

# plt.pcolor gives a rectangular grid by default
# so we need to make our heatmap square
ax2.set_aspect('equal')

# add figure labels and titles
plt.ylabel('Residue IDs')
plt.xlabel('Residue IDs')
plt.title('Distance between centers-of-mass of AdK residues')

# colorbar
cbar2 = fig2.colorbar(im2)
cbar2.ax.set_ylabel('Distance (Angstrom)')
```

[11]:

```
Text(0, 0.5, 'Distance (Angstrom)')
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Residues can be determined to be in contact if atoms from the two residues are within a certain distance. Analysing the fraction of contacts retained by a protein over at trajectory, as compared to the number of contacts in a reference frame or structure, can give insight into folding processes and domain movements.

`MDAnalysis.analysis.contacts` contains functions and a class to analyse the fraction of native contacts over a trajectory.

Fraction of native contacts over a trajectory

Here, we calculate the native contacts of a trajectory as a fraction of the native contacts in a given reference.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- matplotlib
- pandas

Optional packages for molecular visualisation: * nglview

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

See also

- *Contact analysis: number of contacts within a cutoff* (all contacts within a cutoff)
- *Write your own contacts analysis method*
- *Q1 vs Q2 contact analysis*

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis import contacts

# import nglview as nv
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectory DCD samples a transition from a closed to an open conformation.

[2]:

```
u = mda.Universe(PSF, DCD)
```

```
/Users/lily/pydev/mdanalysis/package/MDAnalysis/coordinates/DCD.py:165:␣
↪DeprecationWarning: DCDReader currently makes independent timesteps by copying self.ts␣
↪while other readers update self.ts inplace. This behavior will be changed in 3.0 to be␣
↪the same as other readers. Read more at https://github.com/MDAnalysis/mdanalysis/
↪issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Background

Residues can be determined to be in contact if atoms from the two residues are within a certain distance. *Native* contacts are those contacts that exist within a native state, as opposed to *non-native* contacts, which are formed along the path to a folded state or during the transition between two conformational states. MDAnalysis defines native contacts as those present in the reference structure (`refgroup`) given to the analysis.

Proteins often have more than one native state. Calculating the fraction of native contacts within a protein over a simulation can give insight into transitions between states, or into folding and unfolding processes. MDAnalysis supports three metrics for determining contacts:

- *Hard distance cutoff (hard_cut_q)*
- *Radius cutoff (radius_cut_q)* ([FKDD07])
- *Soft potential-based cutoff (soft_cut_q)* ([BHE13])

Please see the API documentation for the Contacts class for more information.

Defining the groups for contact analysis

For the purposes of this tutorial, we define pseudo-salt bridges as contacts. A more appropriate quantity for studying the transition between two protein conformations may be the contacts formed by alpha-carbon atoms, as this will give us insight into the movements of the protein in terms of the secondary and tertiary structure. The *Q1 vs Q2 contact analysis* demonstrates an example using the alpha-carbon atoms.

[3]:

```
sel_basic = "(resname ARG LYS) and (name NH* NZ)"
sel_acidic = "(resname ASP GLU) and (name OE* OD*)"
acidic = u.select_atoms(sel_acidic)
basic = u.select_atoms(sel_basic)
```

Hard cutoff with a single reference

The 'hard_cut' or hard_cut_q() method uses a hard cutoff for determining native contacts. Two residues are in contact if the distance between them is lower than or equal to the distance in the reference structure.

Below, we use the atomgroups in the universe at the current frame as a reference.

```
[4]: ca1 = contacts.Contacts(u,
                             select=(sel_acidic, sel_basic),
                             refgroup=(acidic, basic),
                             radius=4.5,
                             method='hard_cut').run()
```

The results are available as a numpy array at ca1.timeseries. The first column is the frame, and the second is the fraction of contacts present in that frame.

```
[5]: ca1_df = pd.DataFrame(ca1.results.timeseries,
                           columns=['Frame',
                                    'Contacts from first frame'])
     ca1_df.head()
```

```
[5]:    Frame  Contacts from first frame
     0    0.0                   1.000000
     1    1.0                   0.492754
     2    2.0                   0.449275
     3    3.0                   0.507246
     4    4.0                   0.463768
```

Note that the data is presented as fractions of the native contacts present in the reference configuration. In order to find the number of contacts present, multiply the data with the number of contacts in the reference configuration. Initial contact matrices are saved as pairwise arrays in ca1.initial_contacts.

```
[6]: ca1.initial_contacts[0].shape
```

```
[6]: (70, 44)
```

You can sum this to work out the number of contacts in your reference, and apply that to the fractions of references in your timeseries data.

```
[7]: n_ref = ca1.initial_contacts[0].sum()
     print('There are {} contacts in the reference.'.format(n_ref))
```

```
There are 69 contacts in the reference.
```

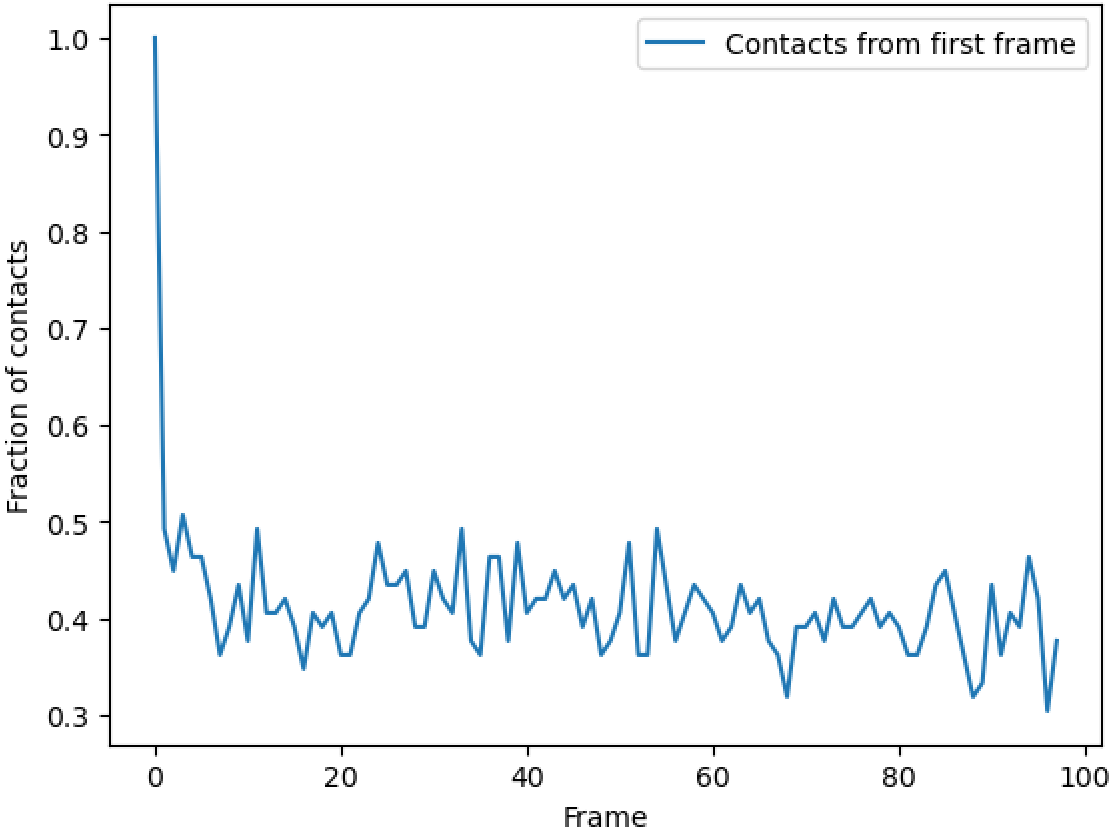
```
[8]: n_contacts = ca1.results.timeseries[:, 1] * n_ref
     print(n_contacts[:5])
```

```
[69. 34. 31. 35. 32.]
```

Plotting

You can plot directly from the dataframe, or use other tools such as seaborn. In this trajectory, the fraction of native contats drops immediately to under 50%, and fluctuates around 40% for the rest of the simulation. This means that the protein retains a structure where around 40% salt bridges in the reference remain within the distance of the reference. However, it is difficult to infer information on domain rearrangements and other large-scale movement, other than that the the protein never returns to a similar state as the initial frame.

[9]:
```
ca1_df.plot(x='Frame')
plt.ylabel('Fraction of contacts')
plt.show()
```



Radius cutoff

Another metric that MDAnalysis supports is determining residues to be in contact if they are within a certain radius. This is similar to the hard cutoff metric, in that there is no potential. The difference is that a single radius is used as the cutoff for all contacts, rather than the distance between the residues in the reference. For a tutorial on similar contact analysis of residues within a cutoff, see *Number of contacts within cutoff*. That tutorial is for calculating the overall number or fraction of contacts, instead of the fraction of native contacts.

You can choose this method by passing in the method name `'radius_cut'`, which uses the radius_cut_q(). The `radius` keyword specifies the distance used in ångström. No other arguments need to passed into `kwargs`.

[10]:
```
ca2 = contacts.Contacts(u, select=(sel_acidic, sel_basic),
                        refgroup=(acidic, basic),
```

(continues on next page)

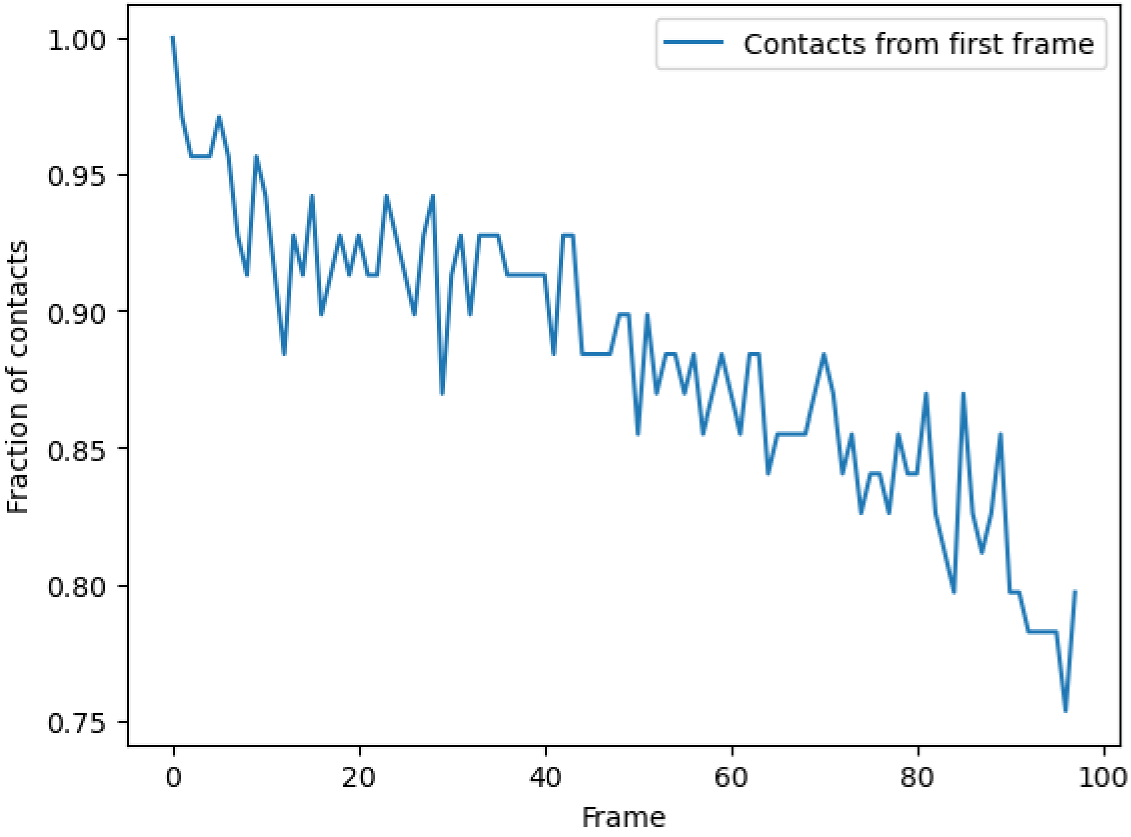
(continued from previous page)

```
                               radius=4.5,
                               method='radius_cut').run()
```

Plotting

Again, we can plot over time. We can see that the fraction of native contacts from the first frame has a very different shape for the `radius_cut` method vs the `hard_cut` method. While the `hard_cut` metric tells us that >50% the native contacts never have equal or lower distance during the trajectory, as compared to the reference, the `radius_cut` analysis shows us that the fraction of contacts within 4.5 Å decreases gradually to 75% over the trajectory. We can infer that almost half the native contacts in the reference frame were closer than 4.5 Å. Moreover, the continuous decrease suggests that the protein may be unfolding, or a large-scale changes in conformation are occurring in such a way that the native salt bridges are not preserved or re-formed.

```
[11]: ca2_df = pd.DataFrame(ca2.results.timeseries,
                             columns=['Frame', 'Contacts from first frame'])
      ca2_df.plot(x='Frame')
      plt.ylabel('Fraction of contacts')
      plt.show()
```



Soft cutoff and multiple references

Multiple references

`refgroup` can either be two contacting groups in a reference configuration, or a list of tuples of two contacting groups.

Below we want to look at native contacts from the first frame, and the last frame. To do this, we create a new universe called `ref` with the same files (and therefore same data) as `u`. We need to do this so that the `(acidic, basic)` selections from `u`, which are assigned from the first frame, remain unchanged. `ref` is a different Universe so when we set it to its last frame (with index `-1`), it does not affect `u` or the previous selections. Now, when we re-select the atomgroups from `ref` with the selection string used in the *hard-cutoff section*, different contacts are selected to the contacts found in the first frame of `u`.

```
[12]: ref = mda.Universe(PSF, DCD)

      ref.trajectory[-1]
      acidic_2 = ref.select_atoms(sel_acidic)
      basic_2 = ref.select_atoms(sel_basic)
```

```
/Users/lily/pydev/mdanalysis/package/MDAnalysis/coordinates/DCD.py:165:␣
↪DeprecationWarning: DCDReader currently makes independent timesteps by copying self.ts␣
↪while other readers update self.ts inplace. This behavior will be changed in 3.0 to be␣
↪the same as other readers. Read more at https://github.com/MDAnalysis/mdanalysis/
↪issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Soft cutoff

This time we will use the soft_cut_q algorithm to calculate contacts by setting `method='soft_cut'`. This method uses the soft potential below to determine if atoms are in contact:

$$Q(r, r_0) = \frac{1}{1 + e^{\beta (r - \lambda r_0)}}$$

r is a distance array and $r0$ are the distances in the reference group. β controls the softness of the switching function and λ is the tolerance of the reference distance.

Suggested values for λ is 1.8 for all-atom simulations and 1.5 for coarse-grained simulations. The default value of β is 5.0. To change these, pass `kwargs` to contacts.Contacts. We also pass in the contacts from the first frame (`(acidic, basic)`) and the last frame (`(acidic_2, basic_2)`) as two separate reference groups. This allows us to calculate the fraction of native contacts in the first frame and the fraction of native contacts in the last frame simultaneously.

```
[13]: ca3 = contacts.Contacts(u, select=(sel_acidic, sel_basic),
                              refgroup=[(acidic, basic), (acidic_2, basic_2)],
                              radius=4.5,
                              method='soft_cut',
                              kwargs={'beta': 5.0,
                                      'lambda_constant': 1.5}).run()
```

Again, the first column of the data array in `ca2.timeseries` is the frame. The next columns of the array are fractions of native contacts with reference to the `refgroups` passed, in order.

```
[14]: ca3_df = pd.DataFrame(ca3.results.timeseries,
                            columns=['Frame',
```

(continues on next page)

(continued from previous page)

```
                                'Contacts from first frame',
                                'Contacts from last frame'])
ca3_df.head()
```

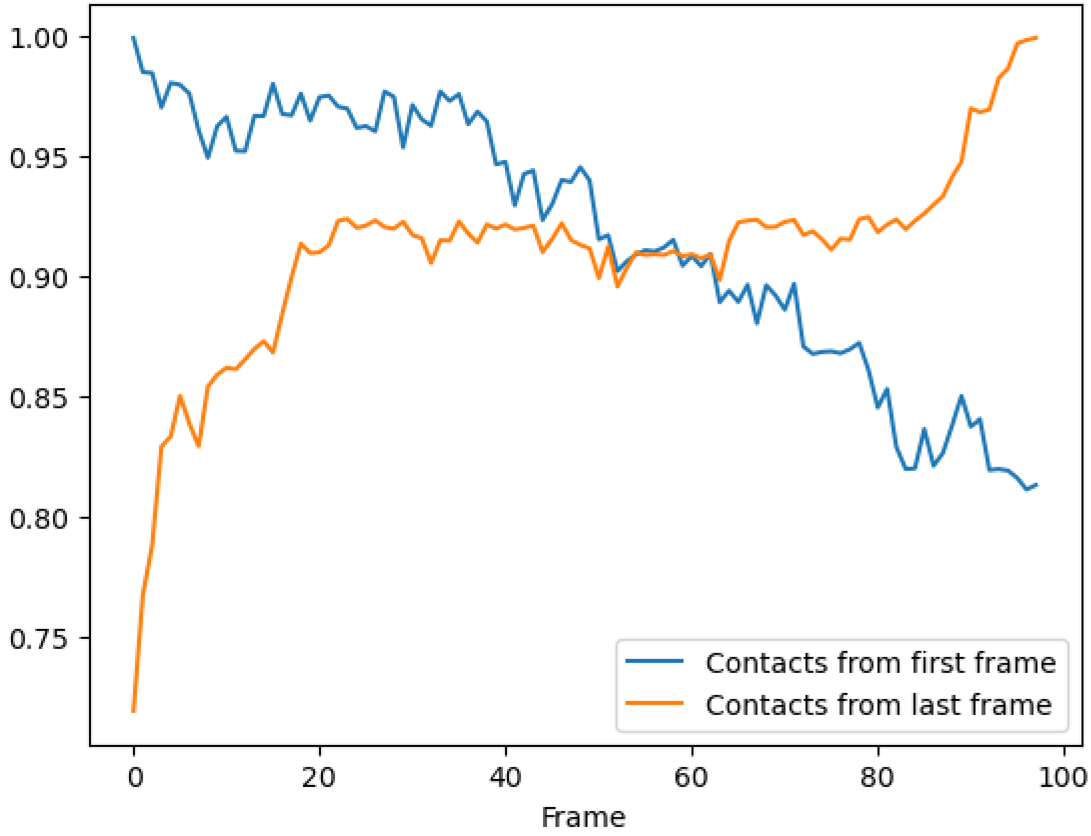
```
[14]:    Frame  Contacts from first frame  Contacts from last frame
      0    0.0                   0.999094                  0.719242
      1    1.0                   0.984928                  0.767501
      2    2.0                   0.984544                  0.788027
      3    3.0                   0.970184                  0.829219
      4    4.0                   0.980425                  0.833500
```

Plotting

Again, we can see that the fraction of native contacts from the first frame has a very different shape for the `soft_cut` method vs the other methods. Like the `radius_cut` method, a gradual decrease in salt bridges is visible; unlike that plot, however, more than 80% native contacts are counted by 100 frames using this metric. By itself, this analysis might suggest that the protein is unfolding.

More interesting is the fraction of native contacts from the *last* frame, which rises from ~70% to 100% over the simulation. This rise indicates that the protein is not *unfolding*, per se (where contacts from the last frame would be expected to rise much less); but instead, a rearrangement of the domains is occurring, where new contacts are formed in the final state.

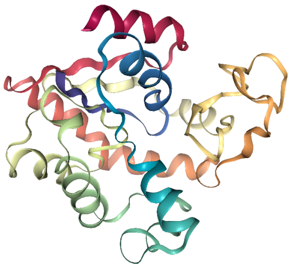
```
[15]: ca3_df.plot(x='Frame')
      plt.show()
```



Indeed, viewing the trajectory shows us that the enzyme transitions from a closed to open state.

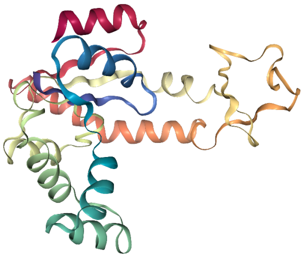
[16]:
```
u.trajectory[0]  # set trajectory to first frame (closed)
# make a new Universe with coordinates of first frame
adk_closed = mda.Merge(u.atoms).load_new(u.atoms.positions)
```

[17]:
```
# adk_closed_view = nv.show_mdanalysis(adk_closed)
# adk_closed_view
```



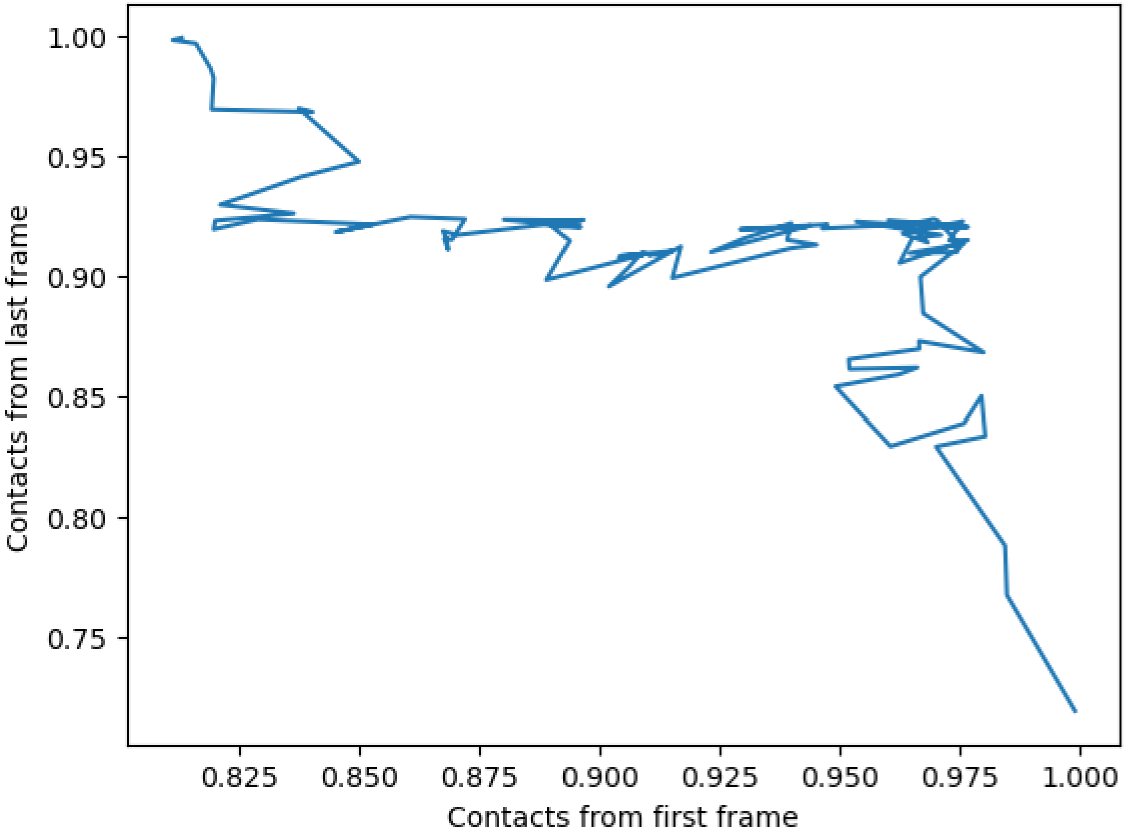
[18]:
```
u.trajectory[-1]  # set trajectory to last frame (open)
# make a new Universe with coordinates of last frame
adk_open = mda.Merge(u.atoms).load_new(u.atoms.positions)
```

[19]:
```
# adk_open_view = nv.show_mdanalysis(adk_open)
# adk_open_view
```



We can also plot the fraction of salt bridges from the first frame, over the fraction from the last frame, as a way to characterise the transition of the protein from closed to open.

[20]:
```
ca3_df.plot(x='Contacts from first frame',
            y='Contacts from last frame',
            legend=False)
plt.ylabel('Contacts from last frame')
plt.show()
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] R. B. Best, G. Hummer, and W. A. Eaton. Native contacts determine protein folding mechanisms in atomistic simulations. Proceedings of the National Academy of Sciences, 110(44):17874–17879, October 2013. 00259. URL: http://www.pnas.org/cgi/doi/10.1073/pnas.1311599110, doi:10.1073/pnas.1311599110.

[3] Joel Franklin, Patrice Koehl, Sebastian Doniach, and Marc Delarue. MinActionPath: maximum likelihood trajectory for large-scale structural transitions in a coarse-grained locally harmonic energy landscape. Nucleic Acids Research, 35(suppl_2):W477–W482, July 2007. 00083. URL: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkm342, doi:10.1093/nar/gkm342.

[4] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[5] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Q1 vs Q2 contact analysis

Here we calculate a Q1 vs Q2 plot, where Q1 refers to fraction of native contacts along a trajectory with reference to the first frame, and Q2 represents the fraction of native contacts with reference to the last.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.17.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- matplotlib
- pandas

See also

- *Fraction of native contacts over a trajectory*
- *Write your own contacts analysis method*
- *Contact analysis: number of contacts within a cutoff*

Note

The `contacts.q1q2` function uses the `radius_cut_q` method to calculate the fraction of native contacts for a conformation by determining that atoms *i* and *j* are in contact if they are within a given radius ([FKDD07], [BHE13])

```
[1]: import MDAnalysis as mda
     from MDAnalysis.tests.datafiles import PSF, DCD
     from MDAnalysis.analysis import contacts

     import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     %matplotlib inline
```

Background

Please see the *Fraction of native contacts* for an introduction to general native contacts analysis.

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectory DCD samples a transition from a closed to an open conformation.

```
[2]: u = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Calculating Q1 vs Q2

We choose to calculate contacts for all the alpha-carbons in the protein, and define the contact radius cutoff at 8 Angstrom. contacts.q1q2 returns a `contacts.Contacts` object, which we can run directly.

```
[3]: q1q2 = contacts.q1q2(u, 'name CA', radius=8).run()
```

The data is in `q1q2.timeseries`. The first column of the data is always the frame number.

```
[4]: q1q2_df = pd.DataFrame(q1q2.results.timeseries,
                            columns=['Frame',
                                     'Q1',
                                     'Q2'])
     q1q2_df.head()
```

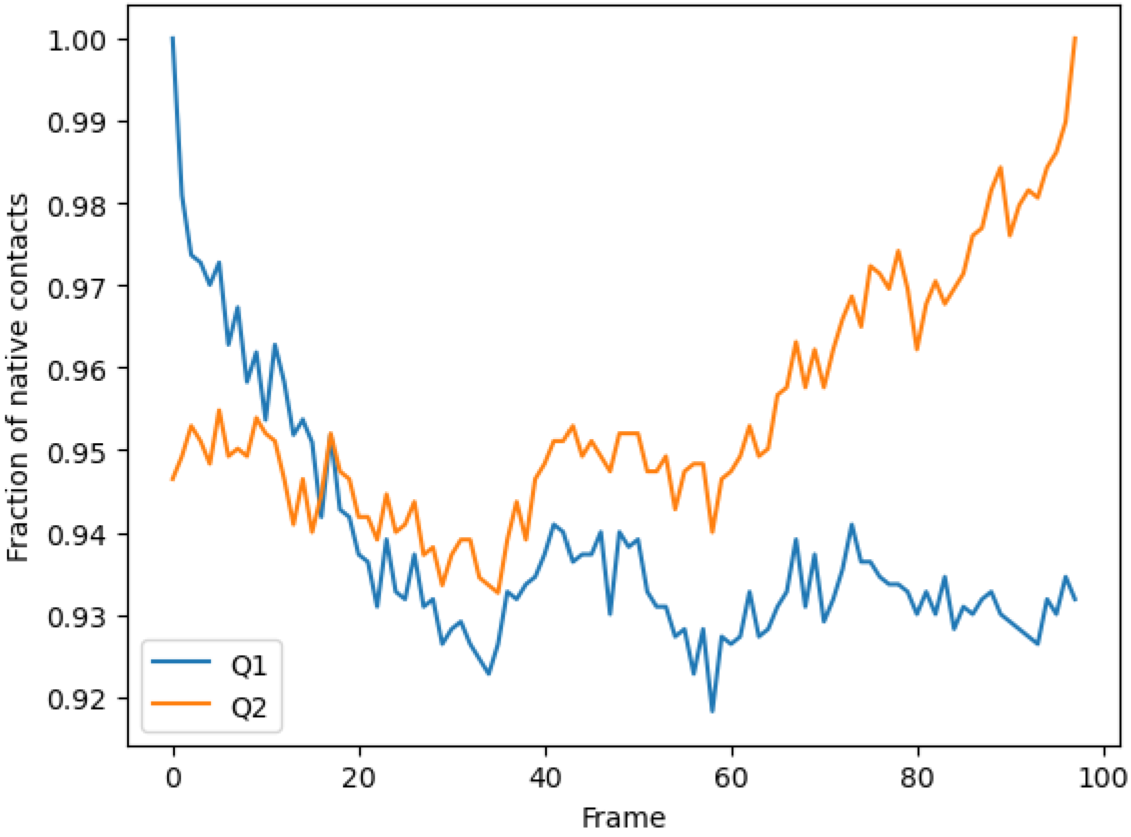
```
[4]:    Frame        Q1        Q2
     0    0.0  1.000000  0.946494
     1    1.0  0.980926  0.949262
     2    2.0  0.973660  0.952952
     3    3.0  0.972752  0.951107
     4    4.0  0.970027  0.948339
```

Plotting

We can plot the fraction of native contacts over time.

[5]:
```
q1q2_df.plot(x='Frame')
plt.ylabel('Fraction of native contacts')
```

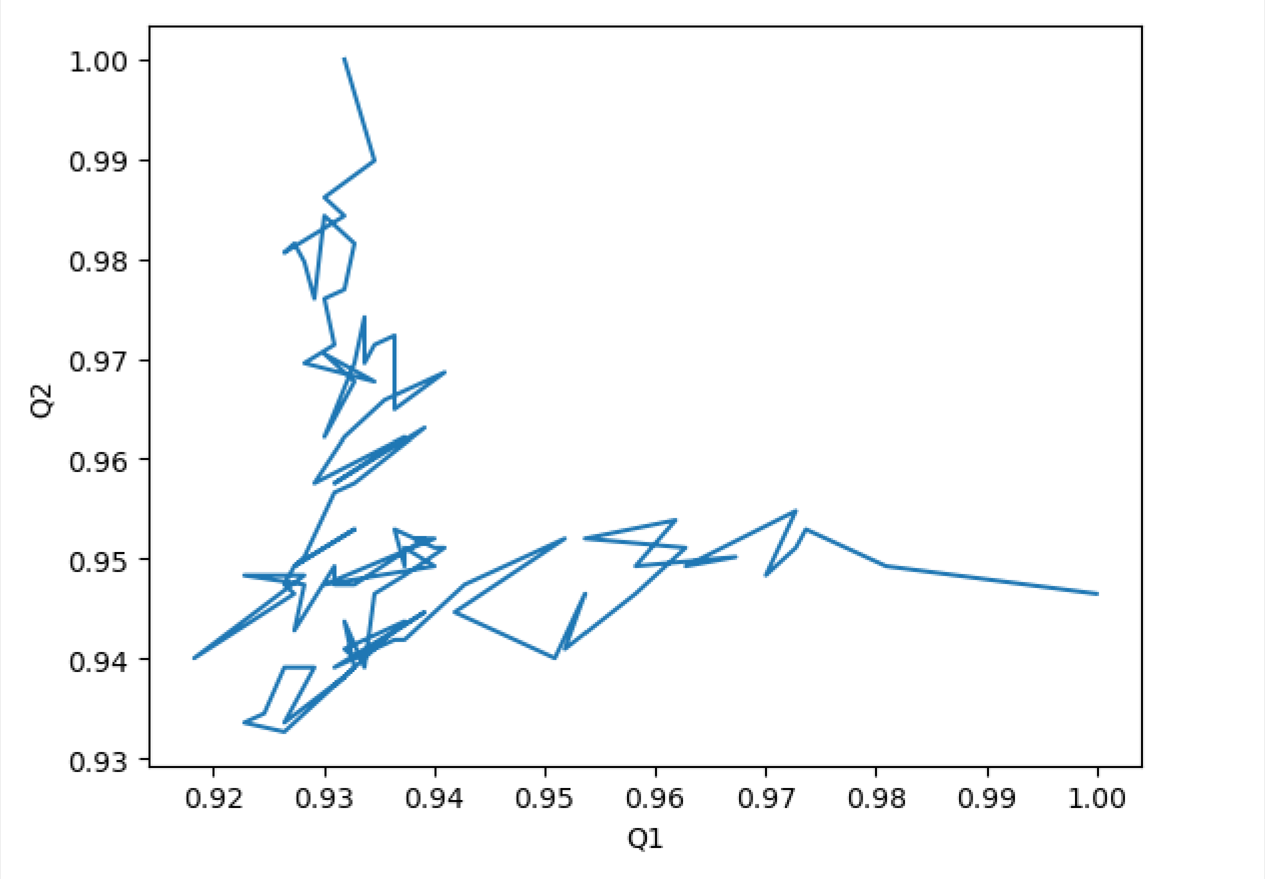
[5]:
```
Text(0, 0.5, 'Fraction of native contacts')
```



Alternatively, we can create a Q1 vs Q2 plot to characterise the transition of AdK from its opened to closed position.

[6]:
```
q1q2_df.plot(x='Q1', y='Q2', legend=False)
plt.ylabel('Q2')
```

[6]:
```
Text(0, 0.5, 'Q2')
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] R. B. Best, G. Hummer, and W. A. Eaton. Native contacts determine protein folding mechanisms in atomistic simulations. Proceedings of the National Academy of Sciences, 110(44):17874–17879, October 2013. 00259. URL: http://www.pnas.org/cgi/doi/10.1073/pnas.1311599110, doi:10.1073/pnas.1311599110.

[3] Joel Franklin, Patrice Koehl, Sebastian Doniach, and Marc Delarue. MinActionPath: maximum likelihood trajectory for large-scale structural transitions in a coarse-grained locally harmonic energy landscape. Nucleic Acids Research, 35(suppl_2):W477–W482, July 2007. 00083. URL: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkm342, doi:10.1093/nar/gkm342.

[4] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[5] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Contact analysis: number of contacts within a cutoff

We calculate the number of salt bridges in an enzyme as it transitions from a closed to an open conformation.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.17.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- matplotlib
- pandas

See also

- *Write your own contacts analysis method*
- *Q1 vs Q2 contact analysis*
- *Fraction of native contacts over a trajectory*

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis import contacts

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Background

Quantifying the number of contacts over a trajectory can give insight into the formation and rearrangements of secondary and tertiary structure. This is closely related to native contacts analysis; where the fraction of native contacts refers to the fraction of contacts retained by a protein from the contacts in a reference frame, the number of contacts simply counts how many residues are within a certain cutoff for each frame. No reference is necessary. Please see the *Fraction of native contacts* for an introduction to native contacts analysis.

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectory DCD samples a transition from a closed to an open conformation.

[2]:
```
u = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Defining the groups for contact analysis

We define salt bridges as contacts between NH/NZ in ARG/LYS and OE*/OD* in ASP/GLU. It is not recommend to use this overly simplistic definition for real work that you want to publish.

```
[3]: sel_basic = "(resname ARG LYS) and (name NH* NZ)"
     sel_acidic = "(resname ASP GLU) and (name OE* OD*)"
     acidic = u.select_atoms(sel_acidic)
     basic = u.select_atoms(sel_basic)
```

Calculating number of contacts within a cutoff

Below, we define a function that calculates the number of contacts between `group_a` and `group_b` within the `radius` cutoff, for each frame in a trajectory.

```
[4]: def contacts_within_cutoff(u, group_a, group_b, radius=4.5):
         timeseries = []
         for ts in u.trajectory:
             # calculate distances between group_a and group_b
             dist = contacts.distance_array(group_a.positions, group_b.positions)
             # determine which distances <= radius
             n_contacts = contacts.contact_matrix(dist, radius).sum()
             timeseries.append([ts.frame, n_contacts])
         return np.array(timeseries)
```

The results are returned as a numpy array. The first column is the frame, and the second is the number of contacts present in that frame.

```
[5]: ca = contacts_within_cutoff(u, acidic, basic, radius=4.5)
     ca.shape
```

```
[5]: (98, 2)
```

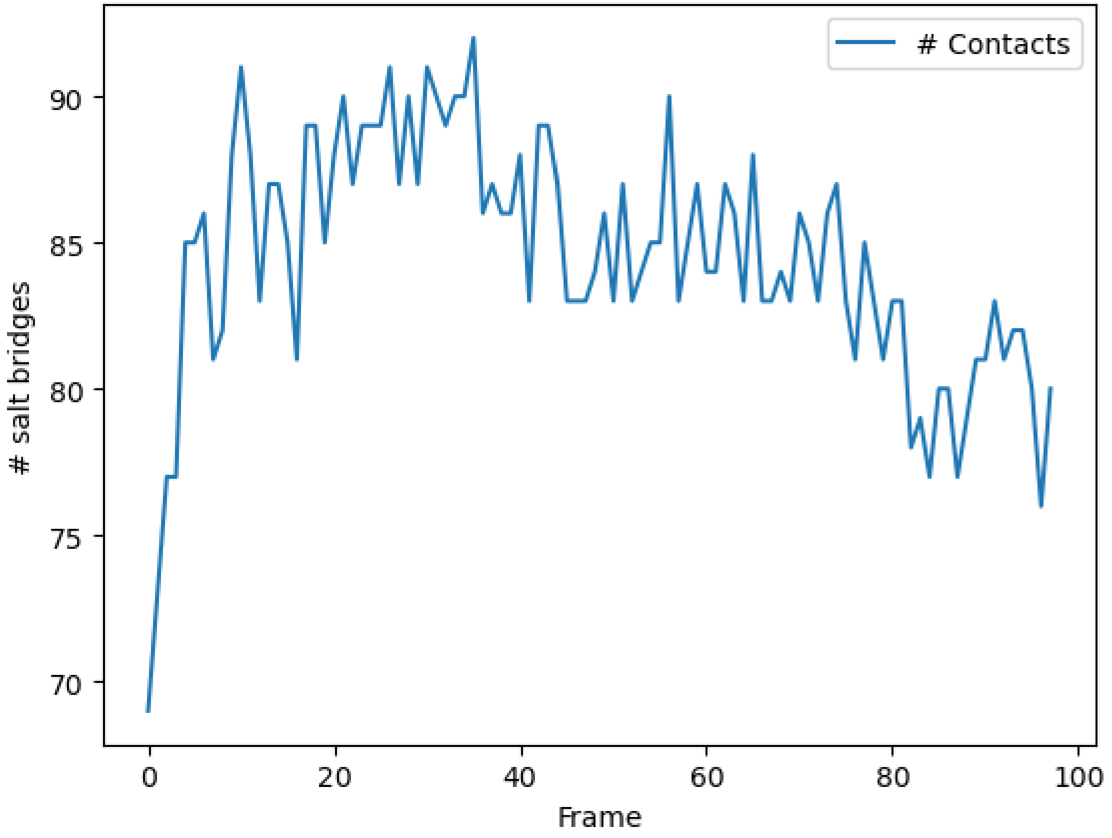
```
[6]: ca_df = pd.DataFrame(ca, columns=['Frame',
                                       '# Contacts'])
     ca_df.head()
```

```
[6]:    Frame  # Contacts
     0      0          69
     1      1          73
     2      2          77
     3      3          77
     4      4          85
```

Plotting

```
[7]: ca_df.plot(x='Frame')
     plt.ylabel('# salt bridges')
```

```
[7]: Text(0, 0.5, '# salt bridges')
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Write your own native contacts analysis method

The `contacts.Contacts` class has been designed to be extensible for your own analysis. Here we demonstrate how to define a new method to use to determine the fraction of native contacts.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- matplotlib
- pandas

See also

- *Fraction of native contacts over a trajectory* (pre-defined metrics and a general introduction to native contacts analysis)
- *Q1 vs Q2 contact analysis*
- *Contact analysis: number of contacts within a cutoff*

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis import contacts

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectory DCD samples a transition from a closed to an open conformation.

[2]:
```
u = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Background

Please see the *Fraction of native contacts* for an introduction to general native contacts analysis.

Defining salt bridges

We define salt bridges as contacts between NH/NZ in ARG/LYS and OE*/OD* in ASP/GLU. You may not want to use this definition for real work.

[3]:

```
sel_basic = "(resname ARG LYS) and (name NH* NZ)"
sel_acidic = "(resname ASP GLU) and (name OE* OD*)"
acidic = u.select_atoms(sel_acidic)
basic = u.select_atoms(sel_basic)
```

Define your own function

Any function you define *must* have `r` and `r0` as its first and second arguments respectively, even if you don't necessarily use them:

- `r`: an array of distances between atoms at the current time
- `r0`: an array of distances between atoms in the reference

You can then define following arguments as keyword arguments.

In the function below, we calculate the fraction of native contacts that are less than `radius`, but greater than `min_radius`.

[4]:

```
def fraction_contacts_between(r, r0, radius=3.4, min_radius=2.5):
    is_in_contact = (r < radius) & (r > min_radius)  # array of bools
    fraction = is_in_contact.sum()/r.size
    return fraction
```

Then we pass `fraction_contacts_between` to the `contacts.Contacts` class. Keyword arguments for our custom function must be in the `kwargs` dictionary. Even though we define a `radius` keyword in my custom analysis function, it is *not* automatically passed from `contacts.Contacts`. We have to make sure that it is in `kwargs`.

[5]:

```
ca = contacts.Contacts(u,
                       select=(sel_acidic, sel_basic),
                       refgroup=(acidic, basic),
                       method=fraction_contacts_between,
                       radius=5.0,
                       kwargs={'radius': 5.0,
                               'min_radius': 2.4}).run()
```

One easy way to post-process results is to turn them into a dataframe.

[6]:

```
ca_df = pd.DataFrame(ca.results.timeseries,
                    columns=['Frame',
                             'Contacts from first frame'])
ca_df.head()
```

[6]:

```
   Frame  Contacts from first frame
0    0.0                   1.000000
1    1.0                   0.988764
2    2.0                   0.943820
3    3.0                   0.943820
4    4.0                   0.943820
```

Plotting

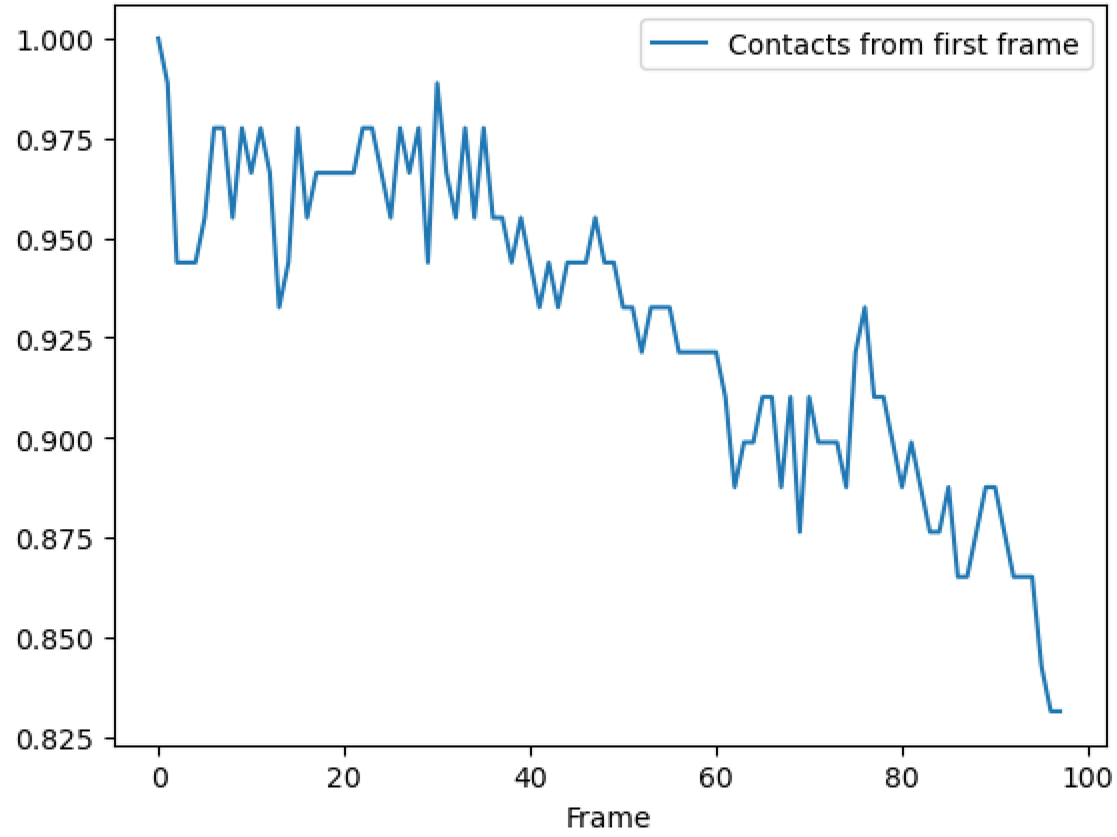
We can plot directly from a dataframe (below), or you could use it with other plotting packags such as seaborn.

[7]:

```
ca_df.plot(x='Frame')
```

[7]:

```
<AxesSubplot: xlabel='Frame'>
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Trajectory similarity

A molecular dynamics trajectory with N atoms can be considered through a path through $3N$-dimensional molecular configuration space. MDAnalysis contains a number of algorithms to compare the conformational ensembles of different trajectories. Most of these are in the `MDAnalysis.analysis.encore` module ([TPB+15]) and compare estimated probability distributions to measure similarity. The path similarity analysis compares the RMSD between pairs of structures in conformation transition paths. `MDAnalysis.analysis.encore` also contains functions for evaluating the conformational convergence of a trajectory using the similarity over conformation clusters or similarity in a reduced dimensional space.

Comparing the geometric similarity of trajectories

Here we compare the geometric similarity of trajectories using the following path metrics:

- the Hausdorff distance
- the discrete Fréchet

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Last updated: December 2022

Minimum version of MDAnalysis: 0.18.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib
- seaborn

Note

The metrics and methods in the `psa` path similarity analysis module are from ([SKTB15]). Please cite them when using the `MDAnalysis.analysis.psa` module in published work.

[1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import MDAnalysis as mda
from MDAnalysis.tests.datafiles import (PSF, DCD, DCD2, GRO, XTC,
                                        PSF_NAMD_GBIS, DCD_NAMD_GBIS,
                                        PDB_small, CRD)
from MDAnalysis.analysis import psa
import warnings
# suppress some MDAnalysis warnings about writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

[2]:

```
u1 = mda.Universe(PSF, DCD)
u2 = mda.Universe(PSF, DCD2)
u3 = mda.Universe(GRO, XTC)
u4 = mda.Universe(PSF_NAMD_GBIS, DCD_NAMD_GBIS)
u5 = mda.Universe(PDB_small, CRD, PDB_small,
                  CRD, PDB_small, CRD, PDB_small)

ref = mda.Universe(PDB_small)


labels = ['DCD', 'DCD2', 'XTC', 'NAMD', 'mixed']
```

The trajectories can have different lengths, as seen below.

[3]:

```
print(len(u1.trajectory), len(u2.trajectory), len(u3.trajectory))
```

```
98 102 10
```

Aligning trajectories

We set up the PSAnalysis (API docs) with our list of Universes and labels. While `path_select` determines which atoms to calculate the path similarities for, `select` determines which atoms to use to align each Universe to `reference`.

[4]:

```
CORE_sel = 'name CA and (resid 1:29 or resid 60:121 or resid 160:214)'

ps = psa.PSAnalysis([u1, u2, u3, u4, u5],
                    labels=labels,
                    reference=ref,
                    select=CORE_sel,
                    path_select='name CA')
```

Generating paths

For each Universe, we will generate a transition path containing each conformation from a trajectory using `generate_paths` (API docs).

First, we will do a mass-weighted alignment of each trajectory to the reference structure `reference`, along the atoms in `select`. To turn off the mass weighting, set `weights=None`. If your trajectories are already aligned, you can skip the alignment with `align=False`.

```
[5]: ps.generate_paths(align=True, save=False, weights='mass')
```

Hausdorff method

Now we can compute the similarity of each path. The default metric is to use the Hausdorff method. *[5]* The Hausdorff distance between two conformation transition paths P and Q is:

$$\delta_H(P,Q) = \max\left(\delta_h(P|Q), \delta_h(Q|P)\right)$$

$\delta_h(P|Q)$ is the directed Hausdorff distance from P to Q, and is defined as:

$$\delta_h(P|Q) = \max_{p \in P} \min_{q \in Q} d(p,q)$$

The directed Hausdorff distance of P to Q is the distance between the two points, $p \in P$ and its structural nearest neighbour $q \in Q$, for the point p where the distance is greatest. This is not commutative, i.e. the directed Hausdorff distance from Q to P is not the same. (See scipy.spatial.distance.directed_hausdorff for more information).

In MDAnalysis, the Hausdorff distance is the RMSD between a pair of conformations in P and Q, where the one of the conformations in the pair has the least similar nearest neighbour.

```
[6]: ps.run(metric='hausdorff')
```

The distance matrix is saved in `ps.D`.

```
[7]: ps.D
```

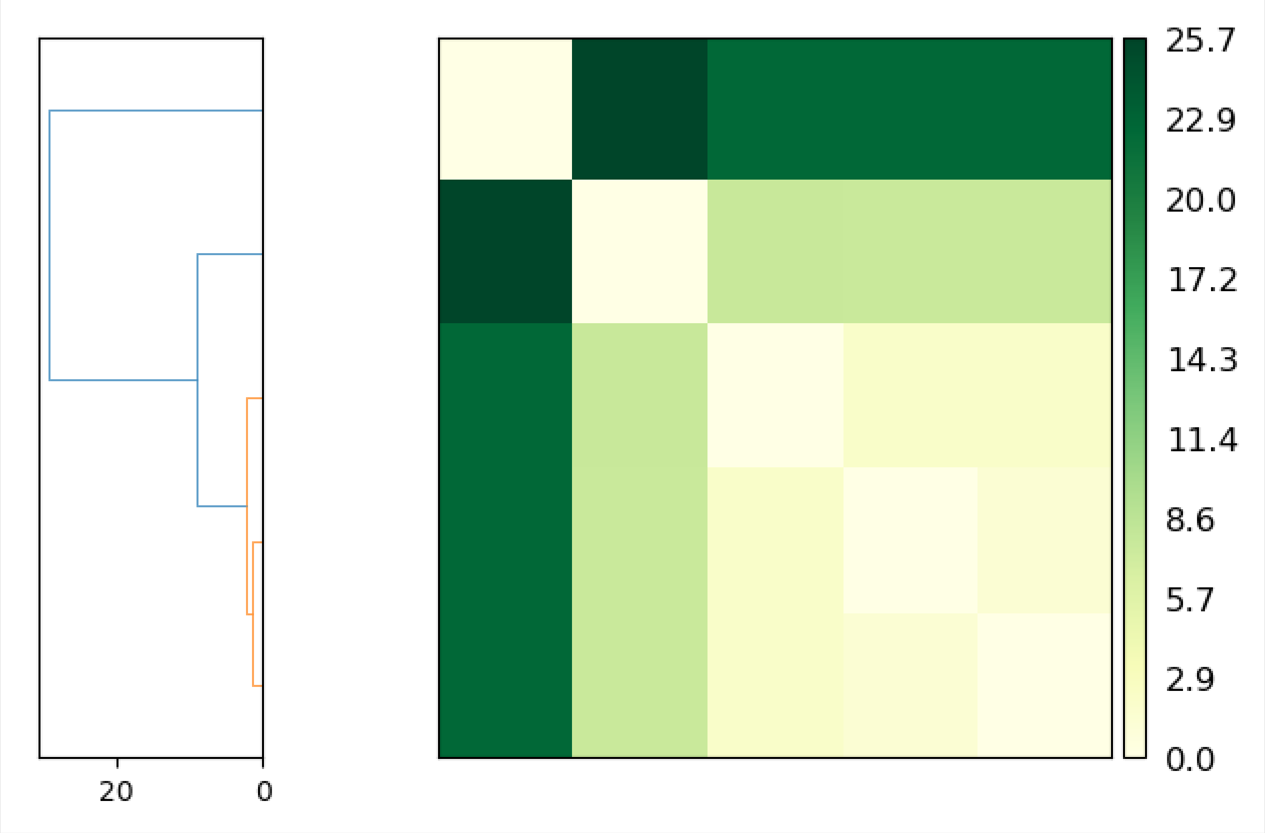
```
[7]: array([[ 0.        ,  1.33312648, 22.37206002,  2.04737477,  7.55204678],
            [ 1.33312648,  0.        , 22.3991666 ,  2.07957562,  7.55032598],
            [22.37206002, 22.3991666 ,  0.        , 22.42282661, 25.74534554],
            [ 2.04737477,  2.07957562, 22.42282661,  0.        ,  7.67052252],
            [ 7.55204678,  7.55032598, 25.74534554,  7.67052252,  0.        ]])
```

Plotting

`psa.PSAnalysis` provides two convenience methods for plotting this data. The first is to plot a heat-map dendrogram from clustering the trajectories based on their path similarity. You can use any clustering method supported by scipy.cluster.hierarchy.linkage; the default is 'ward'.

```
[8]: fig = ps.plot(linkage='ward')
```

```
<Figure size 640x480 with 0 Axes>
```



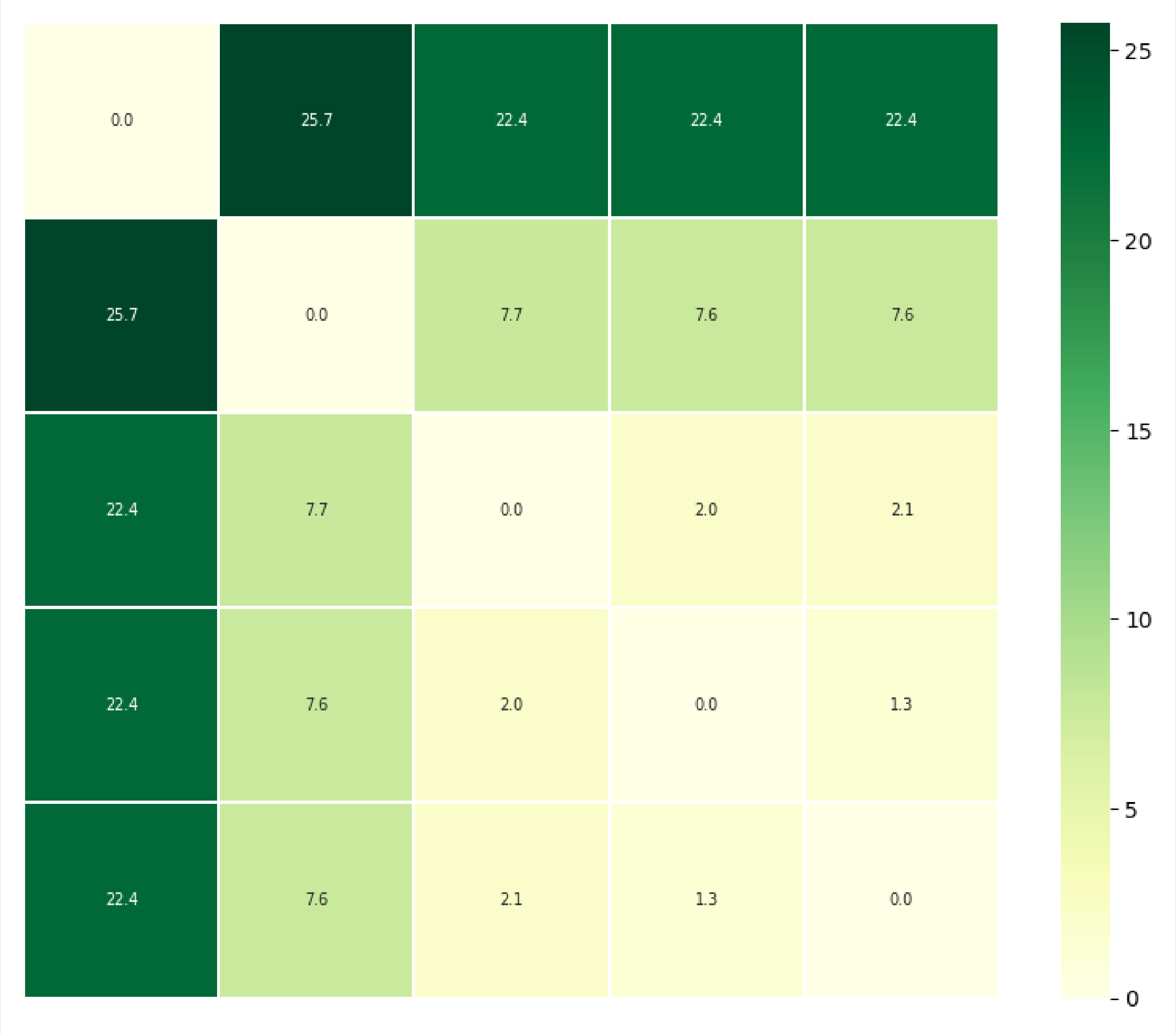
The other is to plot a heatmap annotated with the distance values. Again, the trajectories are displayed in an arrangement that fits the clustering method.

Note

You will need to install the data visualisation library Seaborn for this function.

```
[9]: fig = ps.plot_annotated_heatmap(linkage='single')
```

```
<Figure size 640x480 with 0 Axes>
```



Discrete Fréchet distances

The discrete Fréchet distance between two conformation transition paths P and Q is:

$$\delta_{dF}(P,Q) = \min_{C\in\Gamma_{P,Q}} \|C\|$$

where C is a coupling in the set of all couplings $\Gamma_{P,Q}$ between P and Q. A coupling $C(P,Q)$ is a sequence of pairs of conformations in P and Q, where the first/last pairs are the first/last points of the respective paths, and for each successive pair, at least one point in P or Q must advance to the next frame.

$$C(P,Q) \equiv (p_{a_1}, q_{b_1}), (p_{a_2}, q_{b_2}), ..., (p_{a_L}, q_{b_L})$$

The coupling distance $\|C\|$ is the largest distance between a pair of points in such a sequence.

$$\|C\| \equiv \max_{i=1,...,L} d(p_{a_i}, q_{b_i})$$

In MDAnalysis, the discrete Fréchet distance is the lowest possible RMSD between a conformation from P and a conformation from Q, where the two frames are at similar points along the trajectory, and they are the least structurally similar in that particular coupling sequence. [6-9]

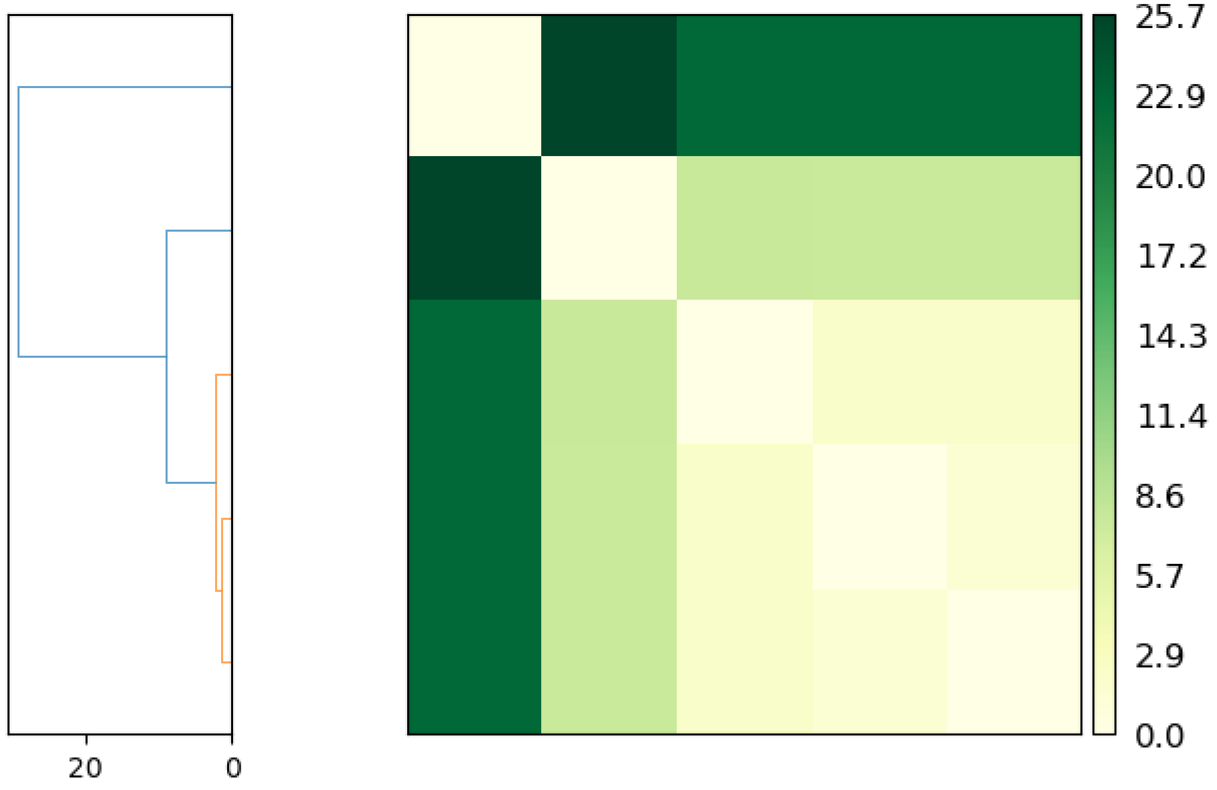
```
[10]: ps.run(metric='discrete_frechet')
      ps.D
```

```
[10]: array([[ 0.        ,  1.33312649, 22.37205967,  2.04737475,  7.55204694],
             [ 1.33312649,  0.        , 22.39916723,  2.07957565,  7.55032613],
             [22.37205967, 22.39916723,  0.        , 22.42282569, 25.74534511],
             [ 2.04737475,  2.07957565, 22.42282569,  0.        ,  7.67052241],
             [ 7.55204694,  7.55032613, 25.74534511,  7.67052241,  0.        ]])
```

Plotting

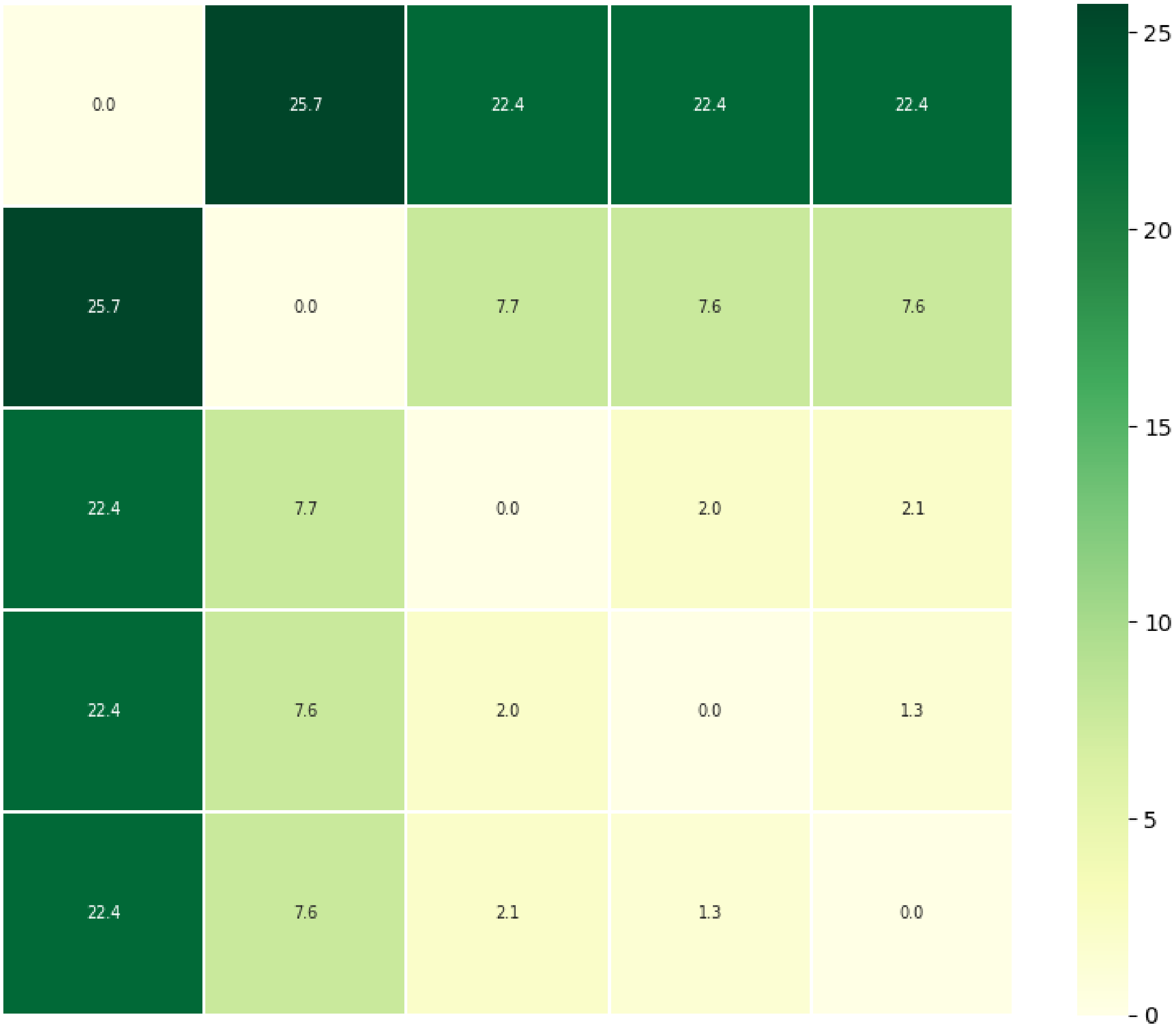
```
[11]: fig = ps.plot(linkage='ward')
```

```
<Figure size 640x480 with 0 Axes>
```



```
[12]: fig = ps.plot_annotated_heatmap(linkage='single')
```

```
<Figure size 640x480 with 0 Axes>
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Sean L. Seyler, Avishek Kumar, M. F. Thorpe, and Oliver Beckstein. Path Similarity Analysis: A Method for

Quantifying Macromolecular Pathways. PLOS Computational Biology, 11(10):e1004568, October 2015. URL: https://dx.plos.org/10.1371/journal.pcbi.1004568, doi:10.1371/journal.pcbi.1004568.

Calculating the Harmonic Ensemble Similarity between ensembles

Here we compare the conformational ensembles of proteins in four trajectories, using the harmonic ensemble similarity method.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Last updated: December 2022

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

Note

The metrics and methods in the `encore` module are from ([TPB+15]). Please cite them when using the `MDAnalysis.analysis.encore` module in published work.

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     %matplotlib inline

     import MDAnalysis as mda
     from MDAnalysis.tests.datafiles import (PSF, DCD, DCD2, GRO, XTC,
                                             PSF_NAMD_GBIS, DCD_NAMD_GBIS,
                                             PDB_small, CRD)
     from MDAnalysis.analysis import encore
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

```
[2]: u1 = mda.Universe(PSF, DCD)
     u2 = mda.Universe(PSF, DCD2)
     u3 = mda.Universe(GRO, XTC)
     u4 = mda.Universe(PSF_NAMD_GBIS, DCD_NAMD_GBIS)

     labels = ['DCD', 'DCD2', 'XTC', 'NAMD']
```

```
/home/pbarletta/mambaforge/envs/mda-user-guide/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/DCD.py:165: DeprecationWarning: DCDReader currently makes independent␣
↪timesteps by copying self.ts while other readers update self.ts inplace. This␣
↪behaviour will be changed in 3.0 to be the same as other readers
  warnings.warn("DCDReader currently makes independent timesteps"
```

The trajectories can have different lengths, as seen below.

```
[3]: print(len(u1.trajectory), len(u2.trajectory), len(u3.trajectory))
```

```
98 102 10
```

Calculating harmonic similarity

The harmonic ensemble similarity method treats the conformational ensemble within each trajectory as a high-dimensional Gaussian distribution $N(\mu, \Sigma)$. The mean μ is estimated as the average over the ensemble. The covariance matrix Σ is calculated either using a shrinkage estimator (`cov_estimator='shrinkage'`) or a maximum-likelihood method (`cov_estimator='ml'`).

The harmonic ensemble similarity is then calculated using the symmetrised version of the Kullback-Leibler divergence. This has no upper bound, so you can get some very high values for very different ensembles.

The function we will use is `encore.hes` (API docs here). It is recommended that you align your trajectories before computing the harmonic similarity. You can either do this yourself with `align.AlignTraj`, or pass `align=True` into `encore.hes`. The latter option will align each of your Universes to the current timestep of the first Universe. Note that since `encore.hes` will pull your trajectories into memory, this changes the positions of your Universes.

```
[4]: hes, details = encore.hes([u1, u2, u3, u4],
                              select='backbone',
                              align=True,
                              cov_estimator='shrinkage',
                              weights='mass')
```

```
[5]: for row in hes:
         for h in row:
             print("{:>10.1f}".format(h), end = ' ')
         print("")
```

```
       0.0     24955.7   1879874.5    145622.3
   24955.7         0.0   1659867.5    161102.3
 1879874.5   1659867.5         0.0   9900092.7
  145622.3    161102.3   9900092.7         0.0
```

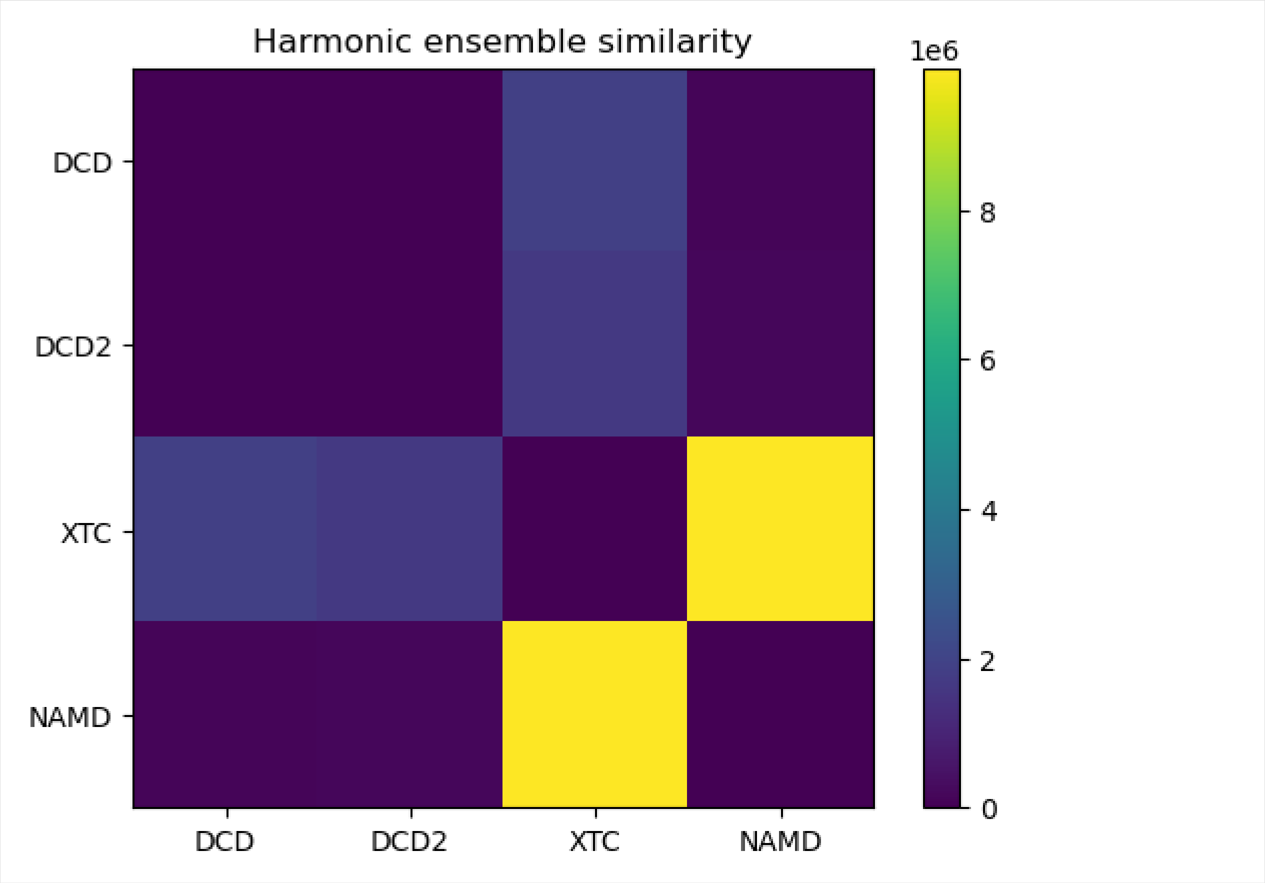
The mean and covariance matrices for each Universe are saved in `details`.

```
[6]: details["ensemble1_mean"].shape
```

```
[6]: (2565,)
```

Plotting

```
[7]: fig, ax = plt.subplots()
     im = plt.imshow(hes)
     plt.xticks(np.arange(4), labels)
     plt.yticks(np.arange(4), labels)
     plt.title('Harmonic ensemble similarity')
     cbar = fig.colorbar(im)
```



References

[1] R. J. Gowers, M. Linke, J. Barnoud, T. J. E. Reddy, M. N. Melo, S. L. Seyler, D. L. Dotson, J. Domanski, S. Buchoux, I. M. Kenney, and O. Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In S. Benthall and S. Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98-105, Austin, TX, 2016. SciPy, doi: 10.25080/majora-629e541a-00e.

[2] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein. MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations. *J. Comput. Chem.* 32 (2011), 2319-2327, doi:10.1002/jcc.21787. PMCID:PMC3144279

[3] ENCORE: Software for Quantitative Ensemble Comparison. Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, Kresten Lindorff-Larsen. *PLoS Comput Biol.* 2015, 11, e1004415.

[4] Beckstein O, Denning EJ, Perilla JR, Woolf TB. Zipping and unzipping of adenylate kinase: atomistic insights into the ensemble of open<–>closed transitions. *J Mol Biol*. 2009;394(1):160–176. doi:10.1016/j.jmb.2009.09.009

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, and Kresten Lindorff-Larsen. ENCORE: Software for Quantitative Ensemble Comparison. PLOS Computational Biology, 11(10):e1004415, October 2015. 00031. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004415, doi:10.1371/journal.pcbi.1004415.

Calculating the Clustering Ensemble Similarity between ensembles

Here we compare the conformational ensembles of proteins in three trajectories, using the clustering ensemble similarity method.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- scikit-learn

Optional packages for visualisation:

- matplotlib

Note

The metrics and methods in the `encore` module are from ([TPB+15]). Please cite them when using the `MDAnalysis.analysis.encore` module in published work.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import MDAnalysis as mda
from MDAnalysis.tests.datafiles import (PSF, DCD, DCD2, GRO, XTC,
                                        PSF_NAMD_GBIS, DCD_NAMD_GBIS)
from MDAnalysis.analysis import encore
from MDAnalysis.analysis.encore.clustering import ClusteringMethod as clm
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

```
[2]: u1 = mda.Universe(PSF, DCD)
     u2 = mda.Universe(PSF, DCD2)
     u3 = mda.Universe(PSF_NAMD_GBIS, DCD_NAMD_GBIS)

     labels = ['DCD', 'DCD2', 'NAMD']
```

```
/home/pbarletta/mambaforge/envs/mda-user-guide/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/DCD.py:165: DeprecationWarning: DCDReader currently makes independent␣
↪timesteps by copying self.ts while other readers update self.ts inplace. This␣
↪behaviour will be changed in 3.0 to be the same as other readers
  warnings.warn("DCDReader currently makes independent timesteps"
```

The trajectories can have different lengths, as seen below.

```
[3]: print(len(u1.trajectory), len(u2.trajectory), len(u3.trajectory))
```

```
98 102 100
```

Calculating clustering similarity with default settings

The clustering ensemble similarity method (`ces`, API docs here) combines every trajectory into a whole space of conformations, and then uses a user-specified `clustering_method` to partition this into clusters. The population of each trajectory ensemble within each cluster is taken as a probability density function.

The similarity of each probability density function is compared using the Jensen-Shannon divergence. This divergence has an upper bound of $\ln(2)$, representing no similarity between the ensembles, and a lower bound of 0.0, representing identical conformational ensembles.

You do not need to align your trajectories, as the function will align it for you (along your `selection` atoms, which are `select='name CA'` by default).

```
[4]: ces0, details0 = encore.ces([u1, u2, u3])
```

`encore.ces` returns two outputs. `ces0` is the similarity matrix for the ensemble of trajectories.

```
[5]: ces0
```

```
[5]: array([[0.        , 0.68070702, 0.69314718],
            [0.68070702, 0.        , 0.69314718],
            [0.69314718, 0.69314718, 0.        ]])
```

`details0` contains the calculated clusters as a `encore.clustering.ClusterCollection.ClusterCollection`.

```
[6]: cluster_collection = details0['clustering'][0]
     print(type(cluster_collection))
     print('We have found {} clusters'.format(len(cluster_collection)))
```

```
<class 'MDAnalysis.analysis.encore.clustering.ClusterCollection.ClusterCollection'>
We have found 49 clusters
```

We can access each Cluster at `cluster_collection.clusters`. For example, the first one has these elements:

[7]:

```
first_cluster = cluster_collection.clusters[0]
first_cluster
```

[7]:

```
<Cluster with 5 elements, centroid=1, id=0>
```

[8]:

```
first_cluster.elements
```

[8]:

```
array([ 0,  1,  2,  3, 98])
```

Each cluster has an ID number and a centroid conformation.

[9]:

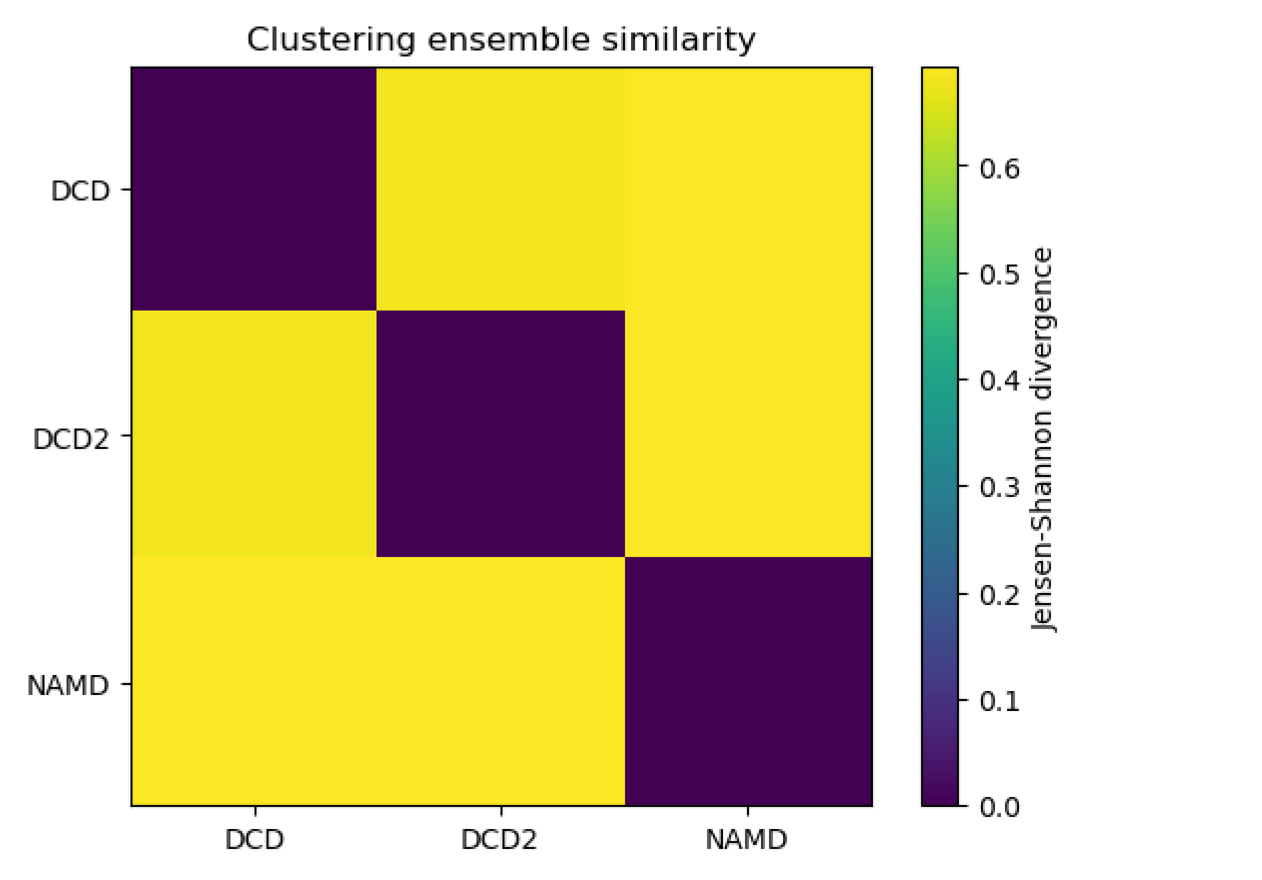
```
print('The ID of this cluster is:', first_cluster.id)
print('The centroid is', first_cluster.centroid)
```

```
The ID of this cluster is: 0
The centroid is 1
```

Plotting

[10]:

```
fig0, ax0 = plt.subplots()
im0 = plt.imshow(ces0, vmax=np.log(2), vmin=0)
plt.xticks(np.arange(3), labels)
plt.yticks(np.arange(3), labels)
plt.title('Clustering ensemble similarity')
cbar0 = fig0.colorbar(im0)
cbar0.set_label('Jensen-Shannon divergence');
```



Calculating clustering similarity with one method

Clustering methods should be subclasses of `analysis.encore.clustering.ClusteringMethod`, initialised with your chosen parameters. Below, we set up an affinity progragation scheme, which uses message-passing to choose a number of 'exemplar' points to represent the data and updates these points until they converge. The `preference` parameter controls how many exemplars are used – a higher value results in more clusters, while a lower value results in fewer clusters. The `damping` factor damps the message passing to avoid numerical oscillations. (See the scikit-learn user guide for more information.)

The other keyword arguments control when to stop clustering. Adding noise to the data can also avoid numerical oscillations.

[11]:
```
clustering_method = clm.AffinityPropagationNative(preference=-1.0,
                                                  damping=0.9,
                                                  max_iter=200,
                                                  convergence_iter=30,
                                                  add_noise=True)
```

By default, MDAnalysis will run the job on one core. If it is taking too long and you have the resources, you can increase the number of cores used.

[12]:
```
ces1, details1 = encore.ces([u1, u2, u3],
                         select='name CA',
```

(continues on next page)

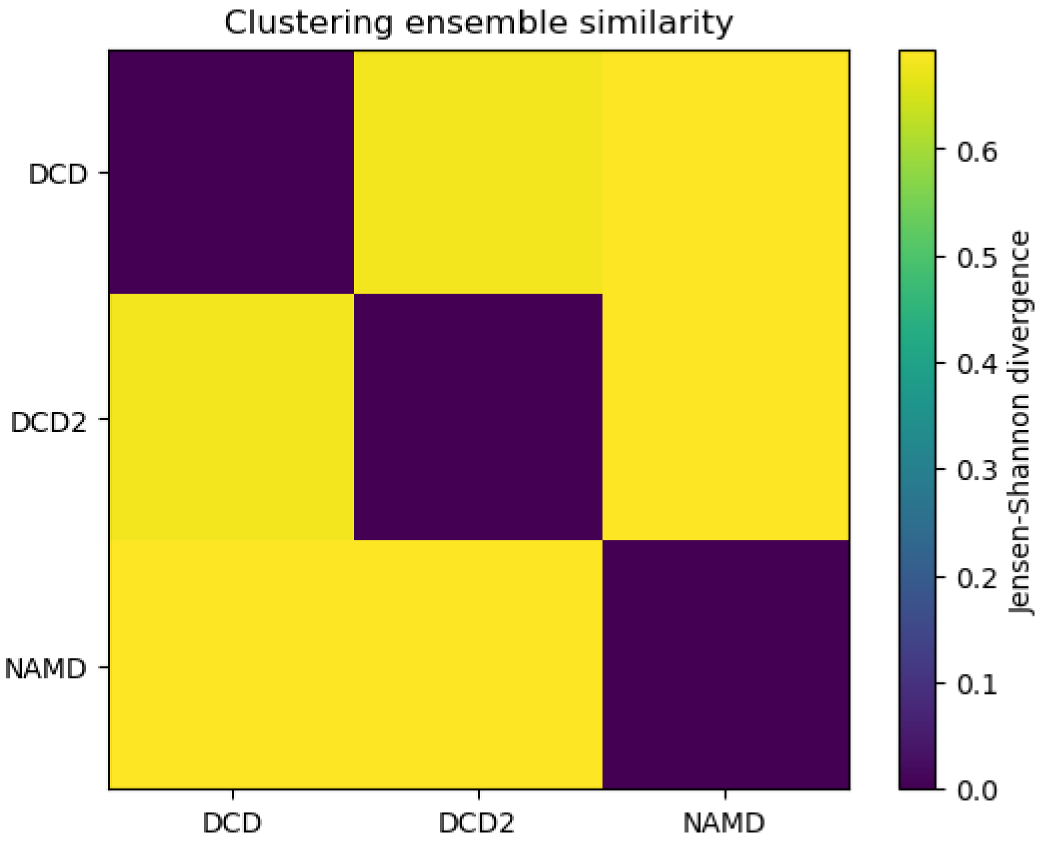
(continued from previous page)

```
                         clustering_method=clustering_method,
                         ncores=4)
```

Plotting

[13]:

```
fig1, ax1 = plt.subplots()
im1 = plt.imshow(ces1, vmax=np.log(2), vmin=0)
plt.xticks(np.arange(3), labels)
plt.yticks(np.arange(3), labels)
plt.title('Clustering ensemble similarity')
cbar1 = fig1.colorbar(im1)
cbar1.set_label('Jensen-Shannon divergence');
```



Calculating clustering similarity with multiple methods

You may want to try different clustering methods, or use different parameters within the methods. encore.ces allows you to pass a list of clustering_methods to be applied.

Note

To use the other ENCORE methods available, you need to install scikit-learn.

Trying out different clustering parameters

The KMeans clustering algorithm separates samples into n groups of equal variance, with centroids that minimise the inertia. You must choose how many clusters to partition. (See the scikit-learn user guide for more information.)

[14]:
```
km1 = clm.KMeans(12,  # no. clusters
                 init = 'k-means++',  # default
                 algorithm="auto")    # default

km2 = clm.KMeans(6,  # no. clusters
                 init = 'k-means++',  # default
                 algorithm="auto")    # default
```

The DBSCAN algorithm is a density-based clustering method that defines clusters as 'high density' areas, separated by low density areas. The parameters min_samples and eps define how dense an area should be to form a cluster. Clusters are defined around core points which have at least min_samples neighbours within a distance of eps. Points that are at least eps in distance from any core point are considered outliers. (See the scikit-learn user guide for more information.)

A higher min_samples or lower eps mean that data points must be more dense to form a cluster. You should consider your eps carefully. In MDAnalysis, eps can be interpreted as the distance between two points in Angstrom.

Note

DBSCAN is an algorithm that can identify outliers, or data points that don't fit into any cluster. dres() and dres_convergence() treat the outliers as their own cluster. This means that the Jensen-Shannon divergence will be lower than it should be for trajectories that have outliers. Do not use this clustering method unless you are certain that your trajectories will not have outliers.

[15]:
```
db1 = clm.DBSCAN(eps=0.5,
                 min_samples=5,
                 algorithm='auto',
                 leaf_size=30)

db2 = clm.DBSCAN(eps=1,
                 min_samples=5,
                 algorithm='auto',
                 leaf_size=30)
```

When we pass a list of clustering methods to encore.ces, the results get saved in ces2 and details2 in order.

[16]:

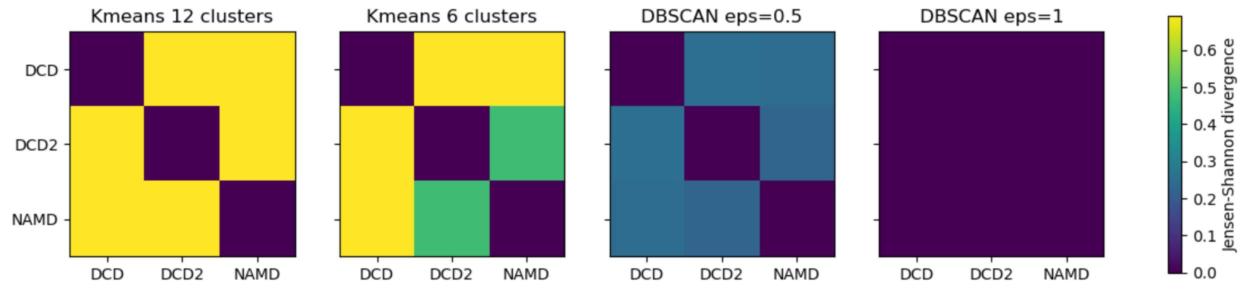
```
ces2, details2 = encore.ces([u1, u2, u3],
                            select='name CA',
                            clustering_method=[km1, km2, db1, db2],
                            ncores=4)
print(len(ces2), len(details2['clustering']))
```

```
4 4
```

Plotting

[17]:

```
titles = ['Kmeans 12 clusters', 'Kmeans 6 clusters', 'DBSCAN eps=0.5', 'DBSCAN eps=1']
fig2, axes = plt.subplots(1, 4, sharey=True, figsize=(15, 3))
for i, (data, title) in enumerate(zip(ces2, titles)):
    imi = axes[i].imshow(data, vmax=np.log(2), vmin=0)
    axes[i].set_xticks(np.arange(3))
    axes[i].set_xticklabels(labels)
    axes[i].set_title(title)
plt.yticks(np.arange(3), labels)
cbar2 = fig2.colorbar(imi, ax=axes.ravel().tolist())
cbar2.set_label('Jensen-Shannon divergence');
```



As can be seen, reducing the number of clusters in the K-means method emphasises that DCD2 is more similar to the NAMD trajectory than DCD. Meanwhile, increasing `eps` in DBSCAN clearly lowered the density required to form a cluster so much that every trajectory is in the same cluster, and therefore they have identical probability distributions.

[18]:

```
n_db = len(details2['clustering'][-1])

print('Number of clusters in DBSCAN eps=1: {}'.format(n_db))
```

```
Number of clusters in DBSCAN eps=1: 1
```

Estimating the error in a clustering ensemble similarity analysis

`encore.ces` also allows for error estimation using a bootstrapping method. This returns the average Jensen-Shannon divergence, and standard deviation over the samples.

[19]:

```
avgs, stds = encore.ces([u1, u2, u3],
                        select='name CA',
                        clustering_method=clustering_method,
                        estimate_error=True,
                        ncores=4)
```

```
[20]: avgs
```

```
[20]: array([[0.        , 0.68682809, 0.69314718],
             [0.68682809, 0.        , 0.69314718],
             [0.69314718, 0.69314718, 0.        ]])
```

```
[21]: stds
```

```
[21]: array([[0.00000000e+00, 5.26432545e-03, 7.02166694e-17],
             [5.26432545e-03, 0.00000000e+00, 8.59975057e-17],
             [7.02166694e-17, 8.59975057e-17, 0.00000000e+00]])
```

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, and Kresten Lindorff-Larsen. ENCORE: Software for Quantitative Ensemble Comparison. PLOS Computational Biology, 11(10):e1004415, October 2015. 00031. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004415, doi:10.1371/journal.pcbi.1004415.

Calculating the Dimension Reduction Ensemble Similarity between ensembles

Here we compare the conformational ensembles of proteins in four trajectories, using the dimension reduction ensemble similarity method.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Last updated: December 2022

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- scikit-learn

Optional packages for visualisation:

- matplotlib

Note

The metrics and methods in the `encore` module are from ([TPB+15]). Please cite them when using the `MDAnalysis.analysis.encore` module in published work.

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     # This import registers a 3D projection, but is otherwise unused.
     from mpl_toolkits.mplot3d import Axes3D
     %matplotlib inline

     import MDAnalysis as mda
     from MDAnalysis.tests.datafiles import (PSF, DCD, DCD2, GRO, XTC,
                                             PSF_NAMD_GBIS, DCD_NAMD_GBIS)
     from MDAnalysis.analysis import encore
     from MDAnalysis.analysis.encore.dimensionality_reduction import␣
     ↪DimensionalityReductionMethod as drm
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

```
[2]: u1 = mda.Universe(PSF, DCD)
     u2 = mda.Universe(PSF, DCD2)
     u3 = mda.Universe(PSF_NAMD_GBIS, DCD_NAMD_GBIS)

     labels = ['DCD', 'DCD2', 'NAMD']
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

The trajectories can have different lengths, as seen below.

```
[3]: print(len(u1.trajectory), len(u2.trajectory), len(u3.trajectory))
```

```
98 102 100
```

Calculating dimension reduction similarity with default settings

The dimension reduction similarity method projects ensembles onto a lower-dimensional space using your chosen dimension reduction algorithm (by default: stochastic proximity embedding). A probability density function is estimated with Gaussian-based kernel-density estimation, using Scott's rule to select the bandwidth.

The similarity of each probability density function is compared using the Jensen-Shannon divergence. This divergence has an upper bound of $\ln(2)$ and a lower bound of 0.0. Normally, $\ln(2)$ represents no similarity between the ensembles, and 0.0 represents identical conformational ensembles. However, due to the stochastic nature of the dimension reduction, two identical symbols will not necessarily result in an exact divergence of 0.0. In addition, calculating the similarity with `dres()` twice will result in similar but not identical numbers.

You do not need to align your trajectories, as the function will align it for you (along your `selection` atoms, which are `select='name CA'` by default). The function we use is `dres` (API docs).

```
[4]: dres0, details0 = encore.dres([u1, u2, u3])
```

`encore.dres` returns two outputs. `dres0` is the similarity matrix for the ensemble of trajectories.

```
[5]: dres0
```

```
[5]: array([[0.        , 0.68134177, 0.68452079],
            [0.68134177, 0.        , 0.66369356],
            [0.68452079, 0.66369356, 0.        ]])
```

`details0` contains information on the dimensionality reduction, as well as the associated reduced coordinates. Each frame is in the conformational ensemble is reduced to 3 dimensions.

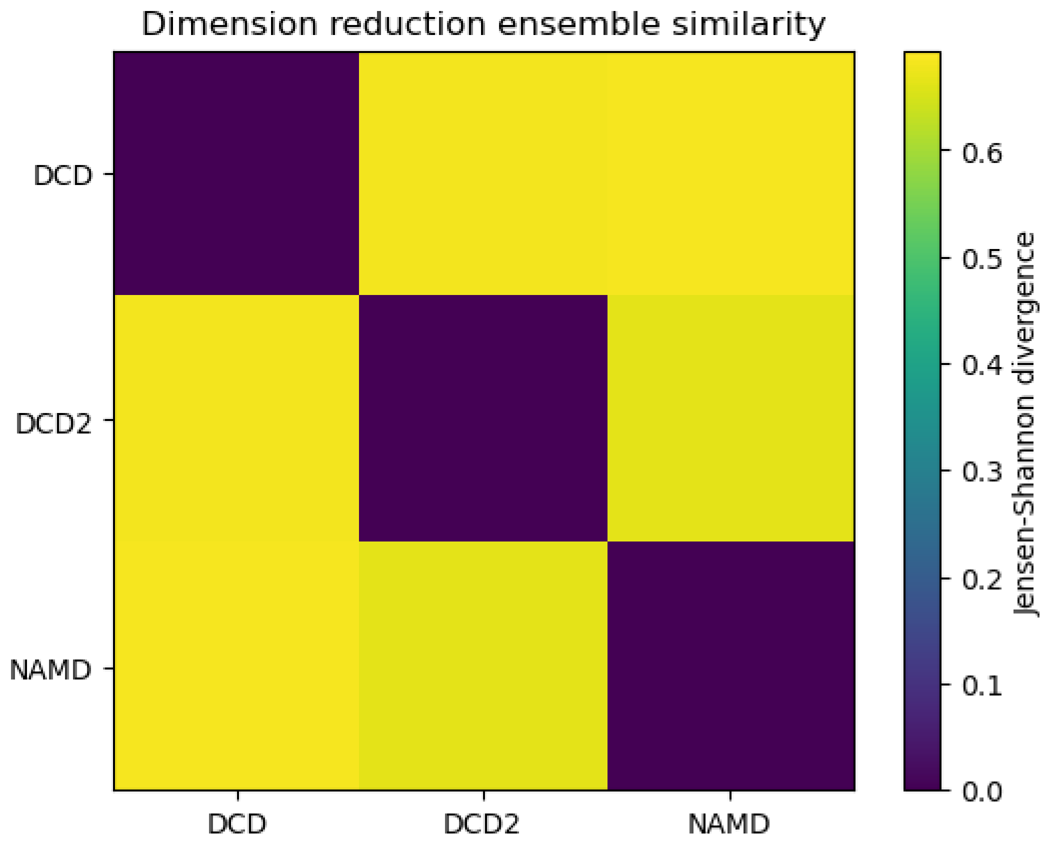
```
[6]: reduced = details0['reduced_coordinates'][0]
     reduced.shape
```

```
[6]: (3, 300)
```

Plotting

As with the other ensemble similarity methods, we can plot a flat matrix of similarity values.

```
[7]: fig0, ax0 = plt.subplots()
     im0 = plt.imshow(dres0, vmax=np.log(2), vmin=0)
     plt.xticks(np.arange(3), labels)
     plt.yticks(np.arange(3), labels)
     plt.title('Dimension reduction ensemble similarity')
     cbar0 = fig0.colorbar(im0)
     cbar0.set_label('Jensen-Shannon divergence')
```



We can also plot the reduced coordinates to directly visualise where each trajectory lies in the lower-dimensional space.

For the plotting of the reduced dimensions, we define a helper function to make it easier to partition the data.

[8]:

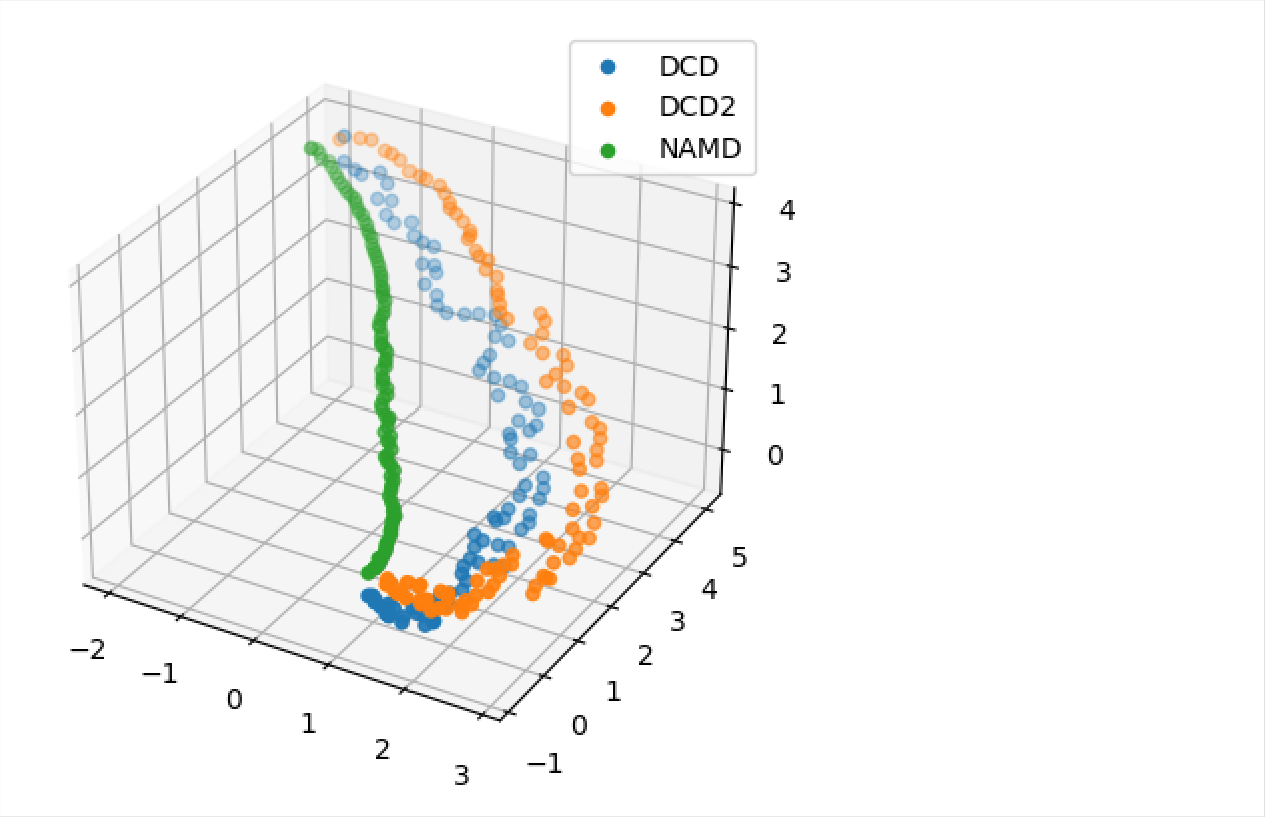
```
def zip_data_with_labels(reduced):
    rd_dcd = reduced[:, :98]  # first 98 frames
    rd_dcd2 = reduced[:, 98:(98+102)]  # next 102 frames
    rd_namd = reduced[:,(98+102):]  # last 100 frames
    return zip([rd_dcd, rd_dcd2, rd_namd], labels)
```

[24]:

```
rdfig0 = plt.figure()
rdax0 = rdfig0.add_subplot(111, projection='3d')
for data, label in zip_data_with_labels(reduced):
    rdax0.scatter(*data, label=label)
plt.legend()
```

[24]:

```
<matplotlib.legend.Legend at 0x7fd2165443a0>
```



Calculating dimension reduction similarity with one method

Dimension reduction methods should be subclasses of `analysis.encore.dimensionality_reduction.DimensionalityReductionMethod`, initialised with your chosen parameters.

Below, we set up stochastic proximity embedding scheme, which maps data to lower dimensions by iteratively adjusting the distance between a pair of points on the lower-dimensional map to match their full-dimensional proximity. The learning rate controls the magnitude of these adjustments, and decreases over the mapping from `max_lam` (default: 2.0) to `min_lam` (default: 0.1) to avoid numerical oscillation. The learning rate is updated every cycle for `ncycles`, over which `nstep` adjustments are performed.

The number of dimensions to map to is controlled by the keyword `dimension` (default: 2).

```
[10]: dim_red_method = drm.StochasticProximityEmbeddingNative(dimension=3,
                                                            min_lam=0.2,
                                                            max_lam=1.0,
                                                            ncycle=50,
                                                            nstep=1000)
```

You can also control the number of samples `nsamples` drawn from the ensembles used to calculate the Jensen-Shannon divergence.

By default, MDAnalysis will run the job on one core. If it is taking too long and you have the resources, you can increase the number of cores used.

```
[11]: dres1, details1 = encore.dres([u1, u2, u3],
                                    select='name CA',
```

(continues on next page)

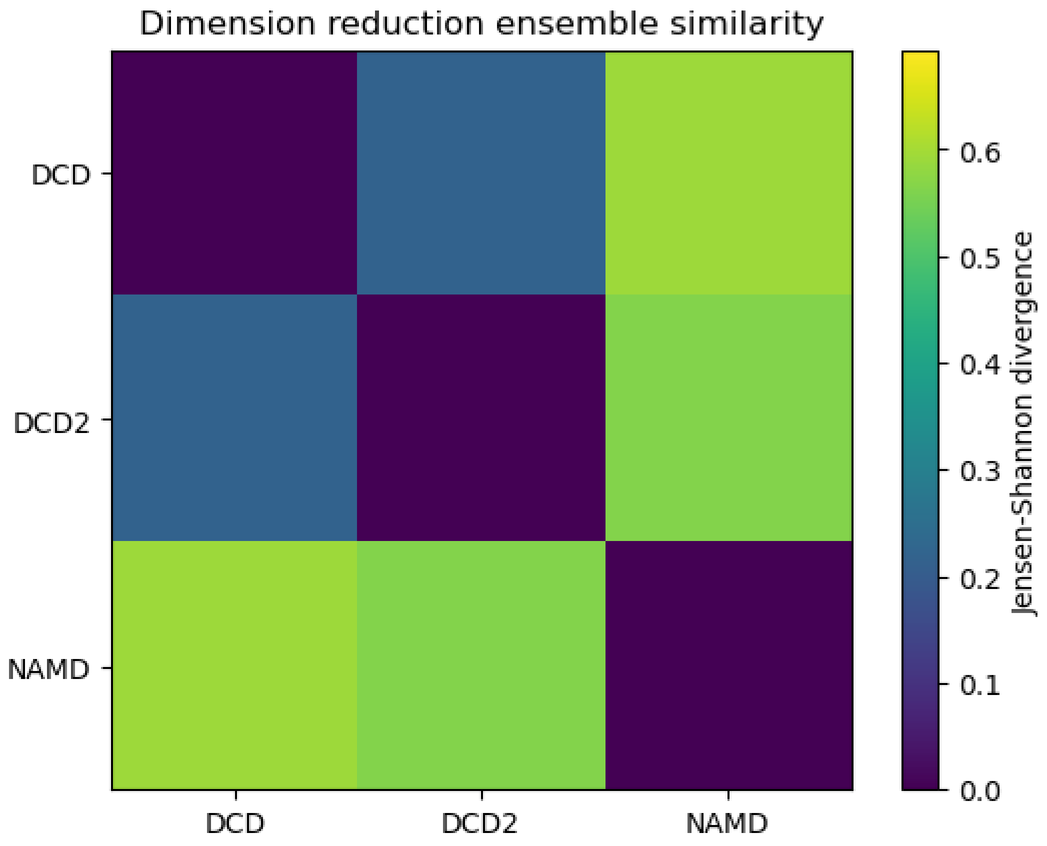
(continued from previous page)

```
                                dimensionality_reduction_method=dim_red_method,
                                nsamples=1000,
                                ncores=4)
```

Plotting

Reducing the learning rate, number of cycles, and number of steps for the stochastic proximity embedding seems to have left our trajectories closer on the lower-dimensional map.

[12]:
```
fig1, ax1 = plt.subplots()
im1 = plt.imshow(dres1, vmax=np.log(2), vmin=0)
plt.xticks(np.arange(3), labels)
plt.yticks(np.arange(3), labels)
plt.title('Dimension reduction ensemble similarity')
cbar1 = fig1.colorbar(im1)
cbar1.set_label('Jensen-Shannon divergence')
```



[13]:
```
reduced1 = details1['reduced_coordinates'][0]

rdfig1 = plt.figure()
rdax1 = rdfig1.add_subplot(111, projection='3d')
for data, label in zip_data_with_labels(reduced1):
    rdax1.scatter(*data, label=label)
```

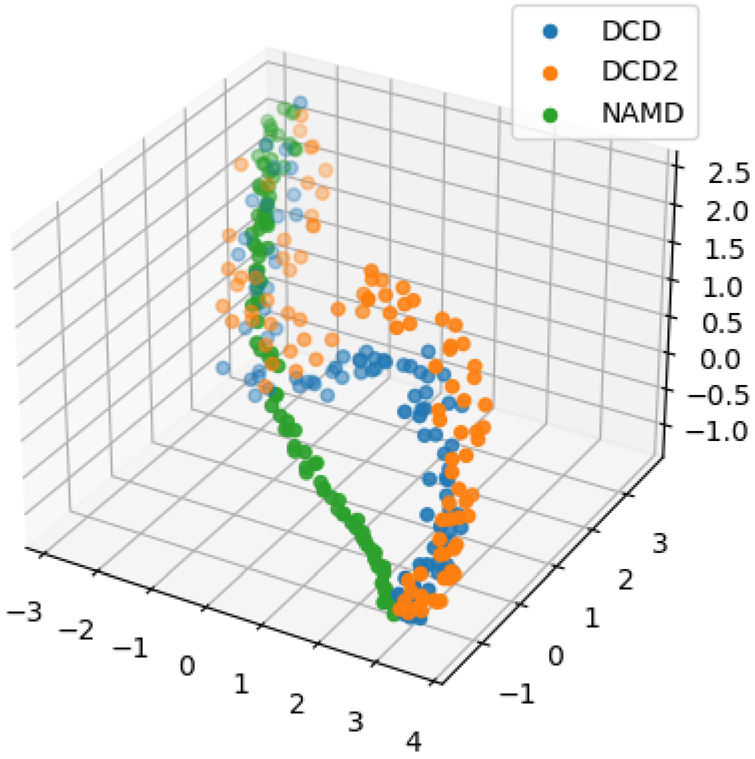
(continues on next page)

(continued from previous page)

```
plt.legend()
```

[13]:

```
<matplotlib.legend.Legend at 0x7fd1d0c7ad00>
```



Calculating dimension reduction similarity with multiple methods

You may want to try different dimension reduction methods, or use different parameters within the methods. `encore.dres` allows you to pass a list of `dimensionality_reduction_methods` to be applied.

Note

To use the other ENCORE methods available, you need to install scikit-learn.

Trying out different dimension reduction parameters

Principal component analysis uses singular value decomposition to project data onto a lower dimensional space. (See the scikit-learn user guide for more information.)

The method provided by MDAnalysis.encore accepts any of the keyword arguments of sklearn.decomposition.PCA *except* `n_components`. Instead, use `dimension` to specify how many components to keep.

[14]:

```
pc1 = drm.PrincipalComponentAnalysis(dimension=1,
                                     svd_solver='auto')
pc2 = drm.PrincipalComponentAnalysis(dimension=2,
```

(continues on next page)

(continued from previous page)

```
                                     svd_solver='auto')
pc3 = drm.PrincipalComponentAnalysis(dimension=3,
                                     svd_solver='auto')
pc4 = drm.PrincipalComponentAnalysis(dimension=4,
                                     svd_solver='auto')
```

When we pass a list of clustering methods to `encore.dres`, the results get saved in `dres2` and `details2` in order.

[15]:

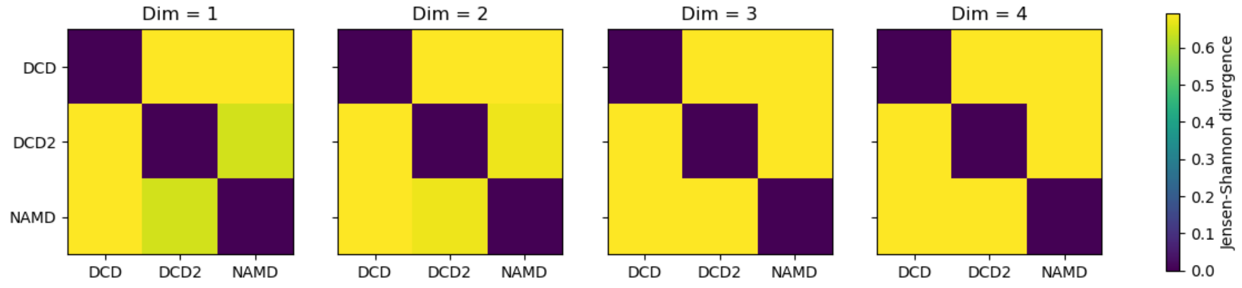
```
dres2, details2 = encore.dres([u1, u2, u3],
                         select='name CA',
                         dimensionality_reduction_method=[pc1, pc2, pc3, pc4],
                         ncores=4)
print(len(dres2), len(details2['reduced_coordinates']))
```

```
4 4
```

Plotting

[16]:

```
titles = ['Dim = {}'.format(n) for n in range(1, 5)]
fig2, axes = plt.subplots(1, 4, sharey=True, figsize=(15, 3))
for i, (data, title) in enumerate(zip(dres2, titles)):
    imi = axes[i].imshow(data, vmax=np.log(2), vmin=0)
    axes[i].set_xticks(np.arange(3))
    axes[i].set_xticklabels(labels)
    axes[i].set_title(title)
plt.yticks(np.arange(3), labels)
cbar2 = fig2.colorbar(imi, ax=axes.ravel().tolist())
cbar2.set_label('Jensen-Shannon divergence')
```



In this case, adding more dimensions to the principal component analysis has little difference in how similar each ensemble is over its resulting probability distribution (i.e. not similar at all!)

[17]:

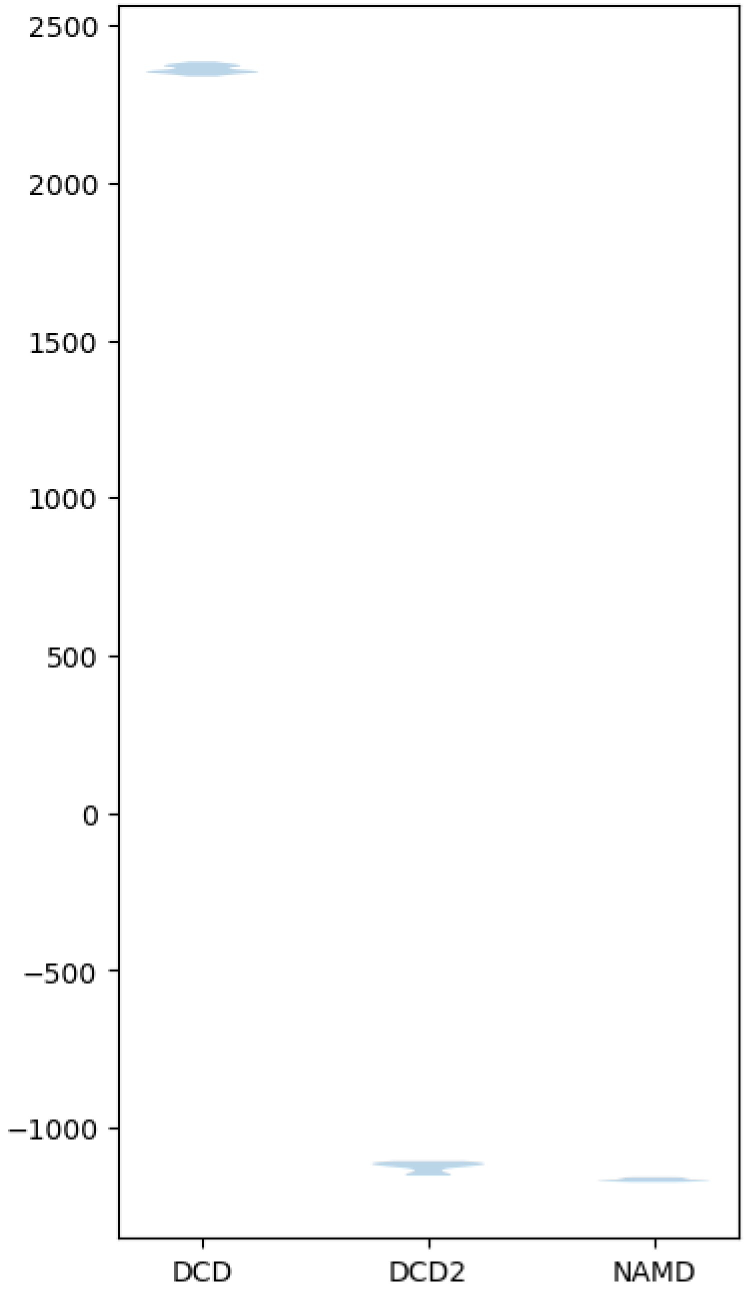
```
rd_p1, rd_p2, rd_p3, _ = details2['reduced_coordinates']
```

If we plot how the trajectories vary on one dimension with a violin plot, we can see that DCD is indeed very distant from DCD2 and NAMD on the first principal component.

[18]:

```
rd_p1_fig, rd_p1_ax = plt.subplots(figsize=(4, 8))
split_data = [x[0].reshape((-1,)) for x in zip_data_with_labels(rd_p1)]
rd_p1_ax.violinplot(split_data, showextrema=False)
rd_p1_ax.set_xticks(np.arange(1, 4))
rd_p1_ax.set_xticklabels(labels)
```

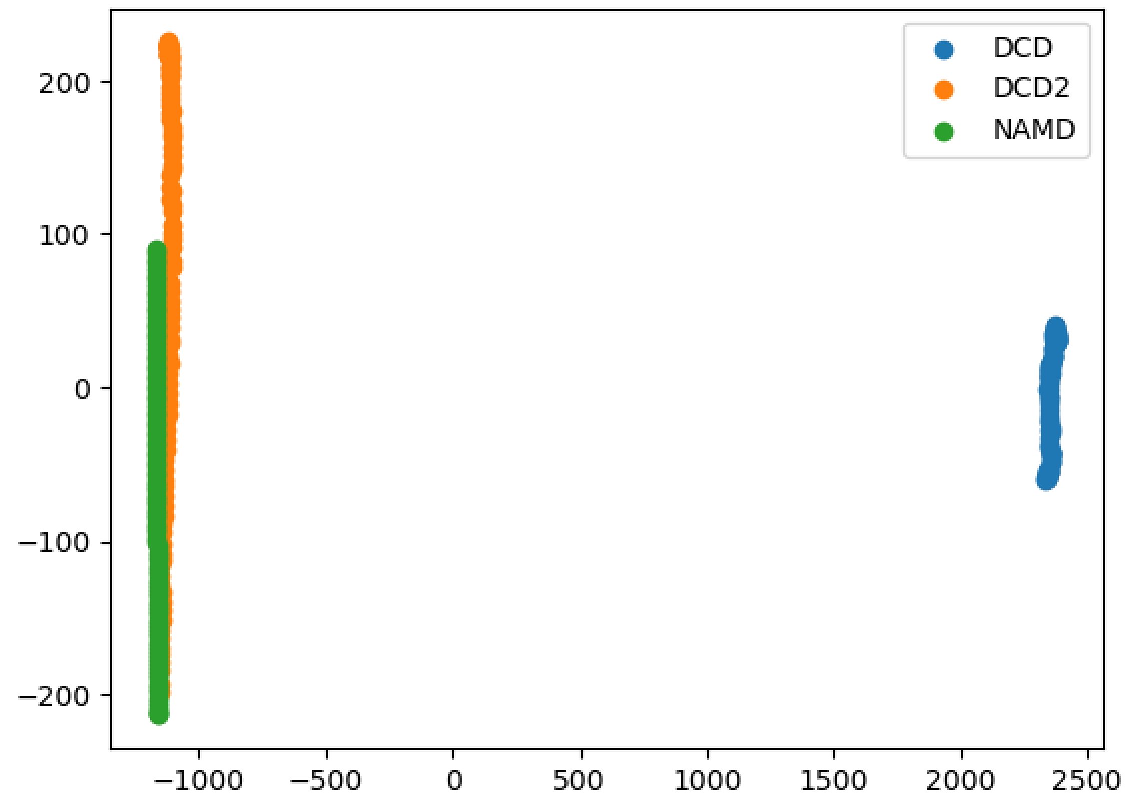
```
[18]: [Text(1, 0, 'DCD'), Text(2, 0, 'DCD2'), Text(3, 0, 'NAMD')]
```



Expanding out to the second principal component shows that DCD2 and NAMD mainly vary on the second axis.

```
[19]: rd_p2_fig, rd_p2_ax = plt.subplots()
      for data, label in zip_data_with_labels(rd_p2):
          rd_p2_ax.scatter(*data, label=label)
      plt.legend()
```

```
[19]: <matplotlib.legend.Legend at 0x7fd21662ec10>
```



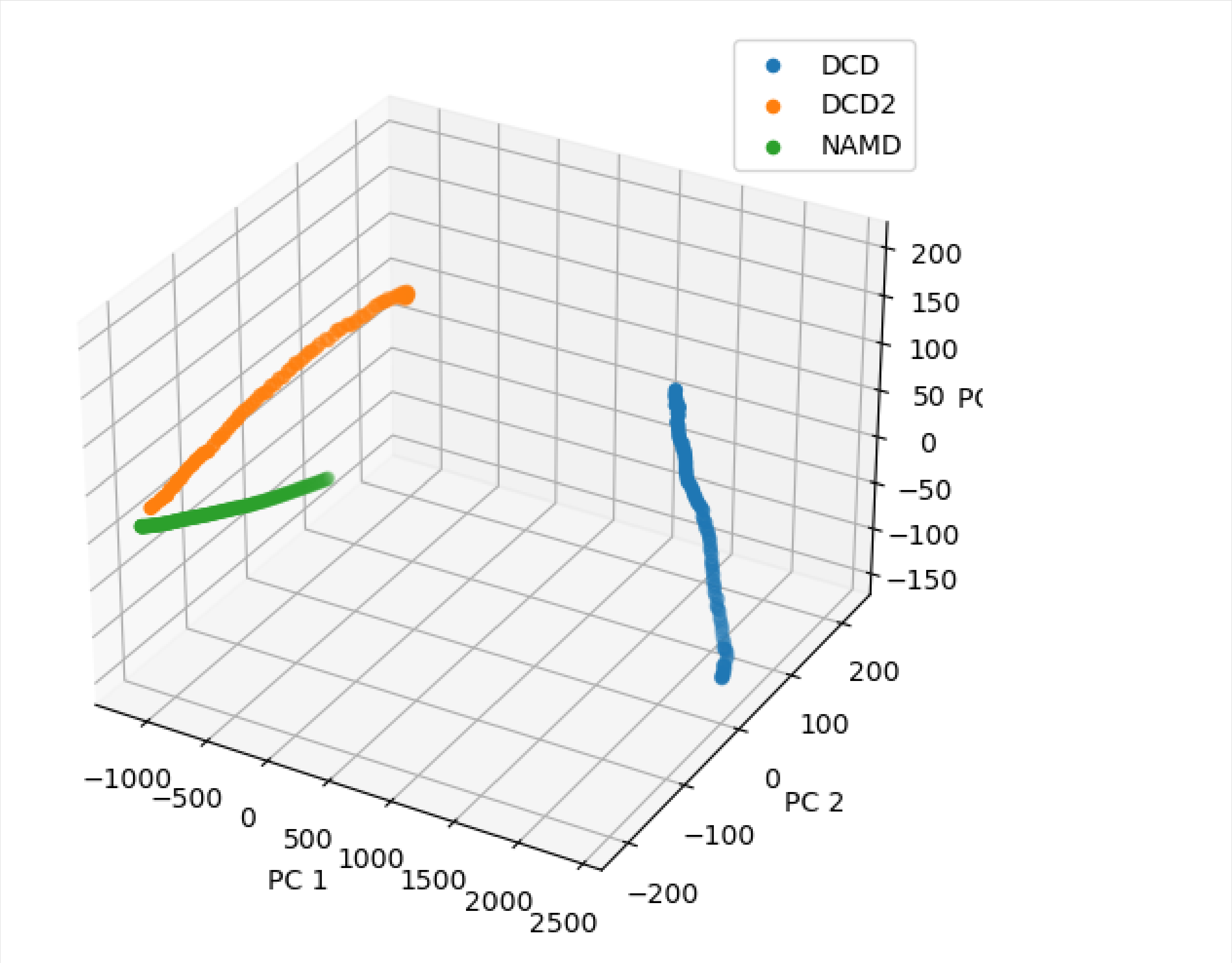
Plotting over the top three principal components gives quite a different result to the reduced coordinates given by stochastic proximity embedding.

[20]:

```
rd_p3_fig = plt.figure(figsize=(8, 6))
rd_p3_ax = rd_p3_fig.add_subplot(111, projection='3d')
for data, label in zip_data_with_labels(rd_p3):
    rd_p3_ax.scatter(*data, label=label)
rd_p3_ax.set_xlabel('PC 1')
rd_p3_ax.set_ylabel('PC 2')
rd_p3_ax.set_zlabel('PC 3')
plt.legend()
```

[20]:

```
<matplotlib.legend.Legend at 0x7fd216582df0>
```



Estimating the error in a dimension reduction ensemble similarity analysis

`encore.dres` also allows for error estimation using a bootstrapping method. This returns the average Jensen-Shannon divergence, and standard deviation over the samples.

```
[21]: avgs, stds = encore.dres([u1, u2, u3],
                             select='name CA',
                             dimensionality_reduction_method=dim_red_method,
                             estimate_error=True,
                             ncores=4)
```

```
[22]: avgs
```

```
[22]: array([[0.        , 0.24545978, 0.60069985],
             [0.24545978, 0.        , 0.59556372],
             [0.60069985, 0.59556372, 0.        ]])
```

```
[23]: stds
```

```
[23]: array([[0.        , 0.06153911, 0.05076614],
             [0.06153911, 0.        , 0.03881675],
             [0.05076614, 0.03881675, 0.        ]])
```

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, and Kresten Lindorff-Larsen. ENCORE: Software for Quantitative Ensemble Comparison. PLOS Computational Biology, 11(10):e1004415, October 2015. 00031. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004415, doi:10.1371/journal.pcbi.1004415.

Evaluating convergence

Here we evaluate the convergence of a trajectory using the clustering ensemble similarity method and the dimensionality reduction ensemble similarity methods.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Last updated: December 2022

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- scikit-learn

Optional packages for visualisation:

- matplotlib

Note

The metrics and methods in the `encore` module are from ([TPB+15]). Please cite them when using the `MDAnalysis.analysis.encore` module in published work.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis import encore
from MDAnalysis.analysis.encore.clustering import ClusteringMethod as clm
from MDAnalysis.analysis.encore.dimensionality_reduction import␣
↪DimensionalityReductionMethod as drm
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

[2]:
```
u = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Evaluating convergence with similarity measures

The convergence of the trajectory is evaluated by the similarity of the conformation ensembles in windows of the trajectory. The trajectory is divided into windows that increase by `window_size` frames. For example, if your trajectory had 13 frames and you specified a `window_size=3`, your windows would be:

```
- Window 1: ---
- Window 2: ------
- Window 3: ---------
- Window 4: -------------
```

Where - represents 1 frame.

These are compared using either the similarity of their clusters (`ces_convergence`) or their reduced dimension coordinates (`dres_convergence`). The rate at which the similarity values drop to 0 is indicative of how much the trajectory keeps on resampling the same regions of the conformational space, and therefore is the rate of convergence.

Using default arguments with clustering ensemble similarity

See *clustering_ensemble_similarity.ipynb* for an introduction to comparing trajectories via clustering. See the API documentation for ces_convergence for more information.

[3]:
```
ces_conv = encore.ces_convergence(u,  # universe
                                  10,  # window size
                                  select='name CA')  # default
```

The output is an array of similarity values, with the shape (number_of_windows, number_of_clustering_methods).

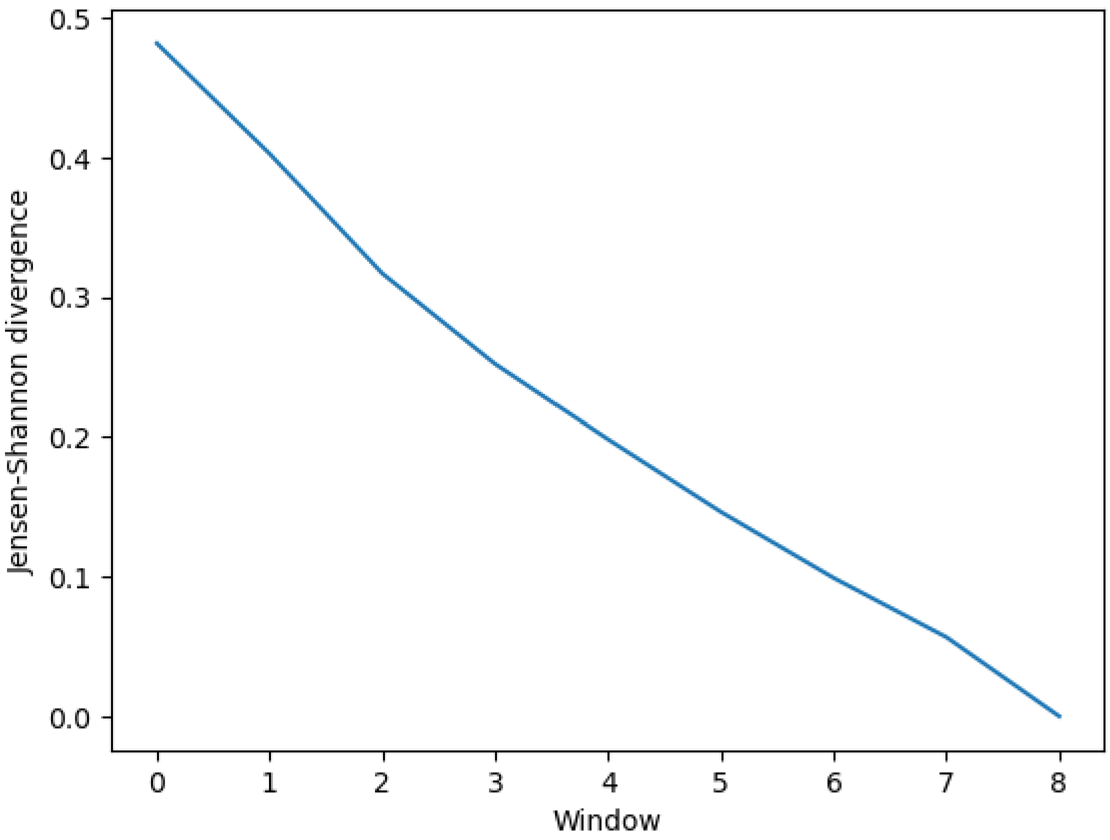
```
[4]: for row in ces_conv:
         for sim in row:
             print("{:>7.4f}".format(sim))
```

```
 0.4819
 0.4028
 0.3170
 0.2522
 0.1983
 0.1464
 0.0991
 0.0567
 0.0000
```

This can be easily plotted as a line.

```
[15]: ces_fig, ces_ax = plt.subplots()
      plt.plot(ces_conv)
      ces_ax.set_xlabel('Window')
      ces_ax.set_ylabel('Jensen-Shannon divergence')
```

```
[15]: Text(0, 0.5, 'Jensen-Shannon divergence')
```



Comparing different clustering methods

You may want to try different clustering methods, or use different parameters within the methods. `encore.ces_convergence` allows you to pass a list of `clustering_methods` to be applied, much like *normal clustering ensemble similarity methods*.

Note

To use the other ENCORE methods available, you need to install scikit-learn.

The KMeans clustering algorithm separates samples into n groups of equal variance, with centroids that minimise the inertia. You must choose how many clusters to partition. (See the scikit-learn user guide for more information.)

[6]:

```
km1 = clm.KMeans(12,  # no. clusters
                 init = 'k-means++',  # default
                 algorithm="auto")    # default

km2 = clm.KMeans(6,  # no. clusters
                 init = 'k-means++',  # default
                 algorithm="auto")    # default

km3 = clm.KMeans(3,  # no. clusters
                 init = 'k-means++',  # default
                 algorithm="auto")    # default
```

When we pass a list of clustering methods to `encore.ces_convergence`, the similarity values get saved in `ces_conv2` in order.

[7]:

```
ces_conv2 = encore.ces_convergence(u,  # universe
                                  10,  # window size
                                  select='name CA',
                                  clustering_method=[km1, km2, km3]
                                 )
ces_conv2.shape
```

[7]:

```
(9, 3)
```

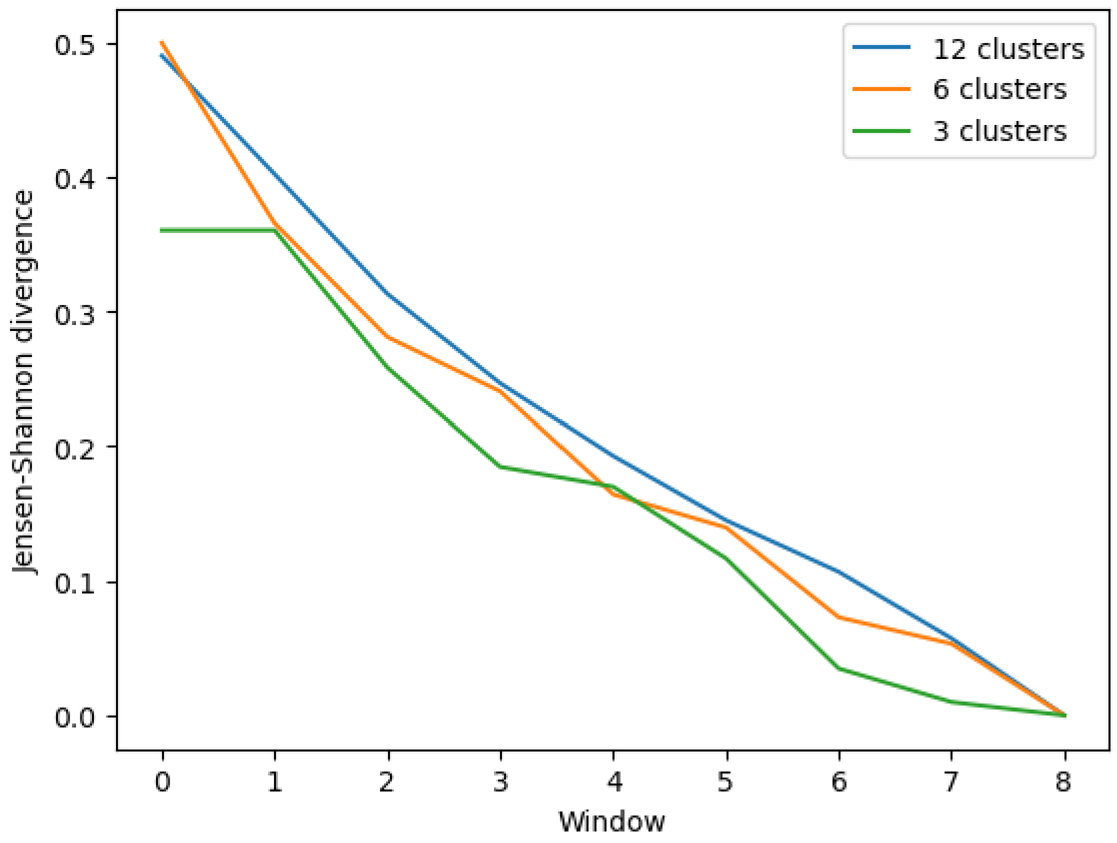
As you can see, the number of clusters partitioned by KMeans has an effect on the resulting rate of convergence.

[8]:

```
labels = ['12 clusters', '6 clusters', '3 clusters']

ces_fig2, ces_ax2 = plt.subplots()
for data, label in zip(ces_conv2.T, labels):
    plt.plot(data, label=label)
ces_ax2.set_xlabel('Window')
ces_ax2.set_ylabel('Jensen-Shannon divergence')
plt.legend()
```

[8]:

```
<matplotlib.legend.Legend at 0x7f9eb2146160>
```



Using default arguments with dimension reduction ensemble similarity

See *dimension_reduction_ensemble_similarity.ipynb* for an introduction on comparing trajectories via dimensionality reduction. We now use the `dres_convergence` function (API docs).

```
[9]: dres_conv = encore.dres_convergence(u,  # universe
                                         10,  # window size
                                         select='name CA')  # default
```

Much like `ces_convergence`, the output is an array of similarity values.

```
[10]: dres_conv
```

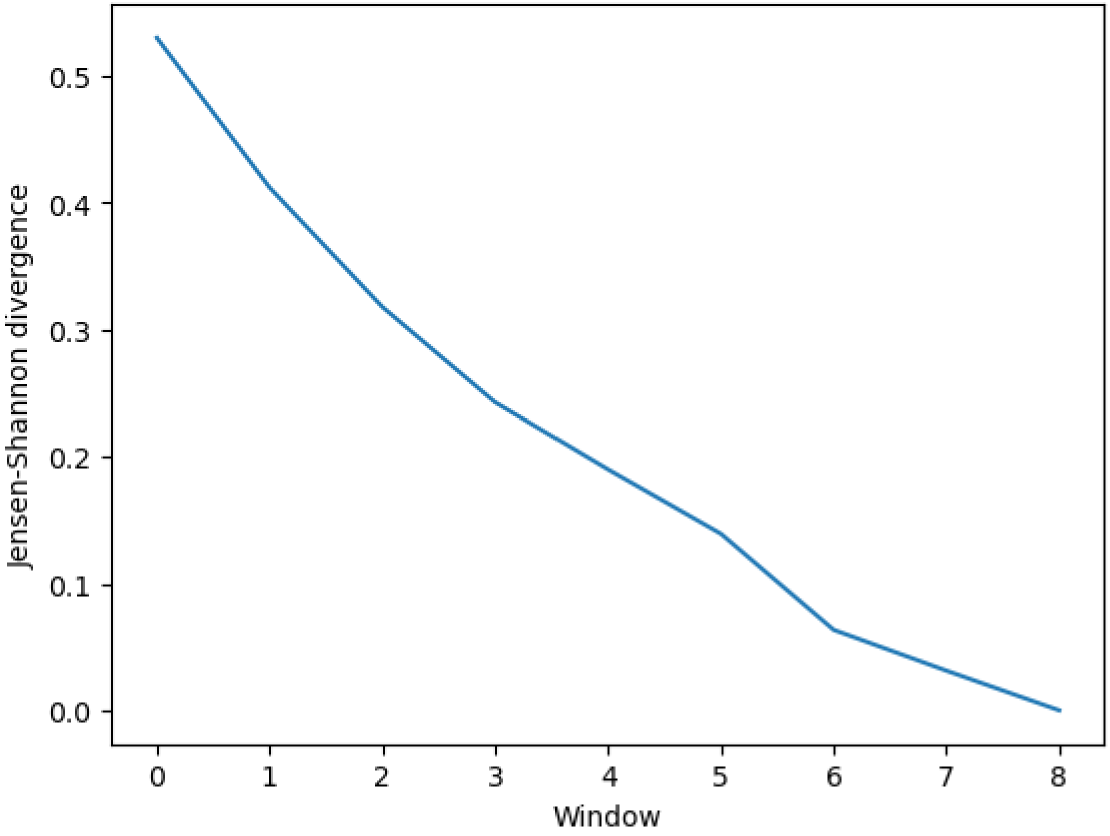
```
[10]: array([[0.52983036],
             [0.41177493],
             [0.31770319],
             [0.24269804],
             [0.18980852],
             [0.13913721],
             [0.06342056],
             [0.03125632],
             [0.        ]])
```

[11]:

```
dres_fig, dres_ax = plt.subplots()
plt.plot(dres_conv)
dres_ax.set_xlabel('Window')
dres_ax.set_ylabel('Jensen-Shannon divergence')
```

[11]:

```
Text(0, 0.5, 'Jensen-Shannon divergence')
```



Comparing different dimensionality reduction methods

Again, you may want to compare the performance of different methods.

Principal component analysis uses singular value decomposition to project data onto a lower dimensional space. (See the scikit-learn user guide for more information.)

The method provided by MDAnalysis.encore accepts any of the keyword arguments of sklearn.decomposition.PCA *except* `n_components`. Instead, use `dimension` to specify how many components to keep.

[12]:

```
pc1 = drm.PrincipalComponentAnalysis(dimension=1,
                                     svd_solver='auto')
pc2 = drm.PrincipalComponentAnalysis(dimension=2,
                                     svd_solver='auto')
pc3 = drm.PrincipalComponentAnalysis(dimension=3,
                                     svd_solver='auto')
```

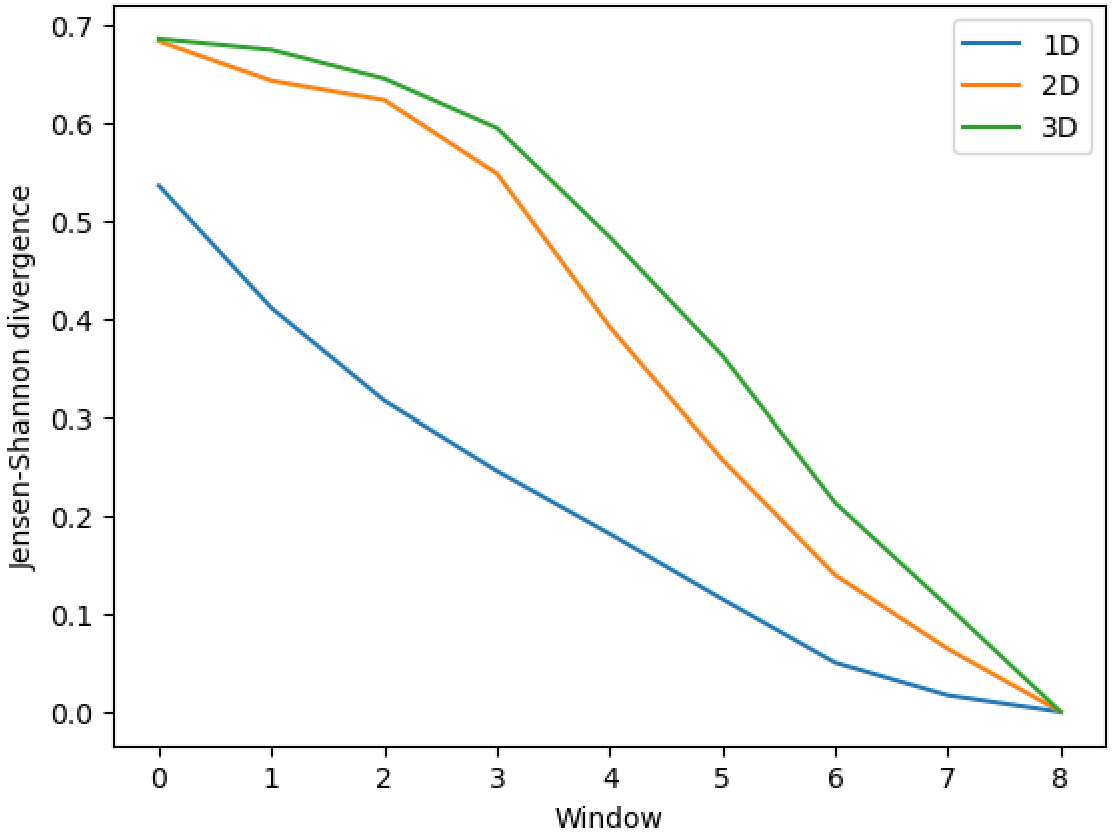
[13]:
```
dres_conv2 = encore.dres_convergence(u,  # universe
                                     10,  # window size
                                     select='name CA',
                                     dimensionality_reduction_method=[pc1, pc2, pc3]
                                    )
dres_conv2.shape
```

[13]:
```
(9, 3)
```

Again, the size of the subspace you choose to include in your similarity comparison, affects the apparent rate of convergence over the trajectory.

[14]:
```
labels = ['1D', '2D', '3D']

dres_fig2, dres_ax2 = plt.subplots()
for data, label in zip(dres_conv2.T, labels):
    plt.plot(data, label=label)
dres_ax2.set_xlabel('Window')
dres_ax2.set_ylabel('Jensen-Shannon divergence')
plt.legend()
```

[14]:
```
<matplotlib.legend.Legend at 0x7f9e98499ee0>
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, and Kresten Lindorff-Larsen. ENCORE: Software for Quantitative Ensemble Comparison. PLOS Computational Biology, 11(10):e1004415, October 2015. 00031. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004415, doi:10.1371/journal.pcbi.1004415.

Structure

Elastic network analysis

Here we use a Gaussian network model to characterise conformational states of a trajectory.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

Note

The elastic network analysis follows the approach of ([HKP+07]). Please cite them when using the `MDAnalysis.analysis.gnm` module in published work.

```
[1]: import MDAnalysis as mda
     from MDAnalysis.tests.datafiles import PSF, DCD, DCD2
     from MDAnalysis.analysis import gnm
     import matplotlib.pyplot as plt
     %matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

```
[2]: u1 = mda.Universe(PSF, DCD)
     u2 = mda.Universe(PSF, DCD2)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Using a Gaussian network model

Using a Gaussian network model to represent a molecule as an elastic network, we can characterise the concerted motions of a protein, and the dominance of these motions, over a trajectory. The analysis is applied to the atoms in the `selection`. If two atoms are within the `cutoff` distance (default: 7 ångström), they are considered to be bound by a spring. This analysis is reasonably robust to the choice of cutoff (between 5-9 Å), but the singular value decomposition may not converge with a lower cutoff.

We use the GNMAnalysis class (API docs) for the analysis.

```
[3]: nma1 = gnm.GNMAnalysis(u1,
                            select='name CA',
                            cutoff=7.0)
     nma1.run()
```

```
[3]: <MDAnalysis.analysis.gnm.GNMAnalysis at 0x7f05569e96d0>
```

The output is saved in `nma1.results`: the time in picoseconds, the first eigenvalue, and the first eigenvector, associated with each frame.

```
[4]: list(nma1.results.keys())
```

```
[4]: ['eigenvalues', 'eigenvectors', 'times']
```

```
[5]: (len(nma1.results['eigenvalues']), len(nma1.results['eigenvectors']),
         len(nma1.results['times']))
```

```
[5]: (98, 98, 98)
```

```
[6]: nma2 = gnm.GNMAnalysis(u2,
                            select='name CA',
                            cutoff=7.0)
     nma2.run()
```

```
[6]: <MDAnalysis.analysis.gnm.GNMAnalysis at 0x7f0556a1f8e0>
```

Unlike normal mode analysis, Gaussian network model analysis uses only a single eigenvalue to represent the rotation and translation of each frame. The motion with the lowest positive eigenvalue represents the dominant motion of a structure. The frequency of this motion is the square root of the eigenvalue.

Plotting the probability distribution of the frequency for the first eigenvector can highlight variation in the probability distribution, which can indicate trajectories in different states.

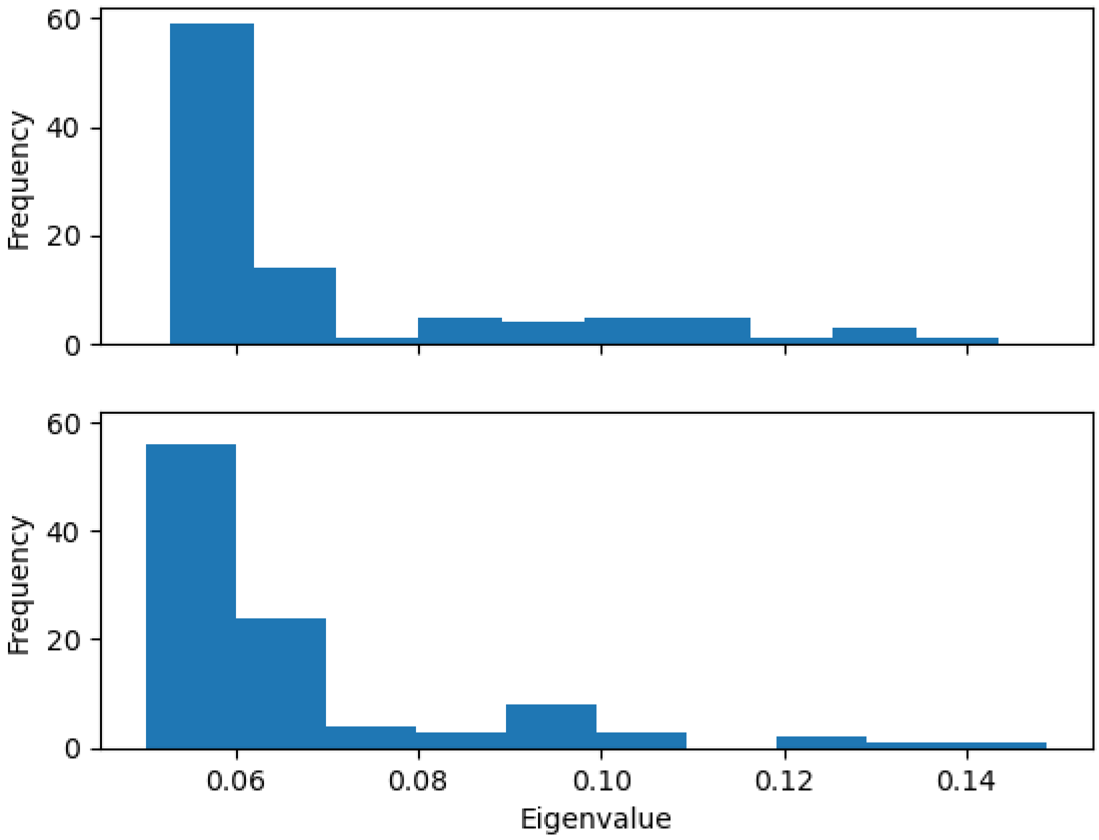
Below, we plot the distribution of eigenvalues. The dominant conformation state is represented by the peak at 0.06.

[7]:

```
histfig, histax = plt.subplots(nrows=2, sharex=True, sharey=True)
histax[0].hist(nma1.results['eigenvalues'])
histax[1].hist(nma2.results['eigenvalues'])

histax[1].set_xlabel('Eigenvalue')
histax[0].set_ylabel('Frequency')
histax[1].set_ylabel('Frequency')
```

[7]:

```
Text(0, 0.5, 'Frequency')
```



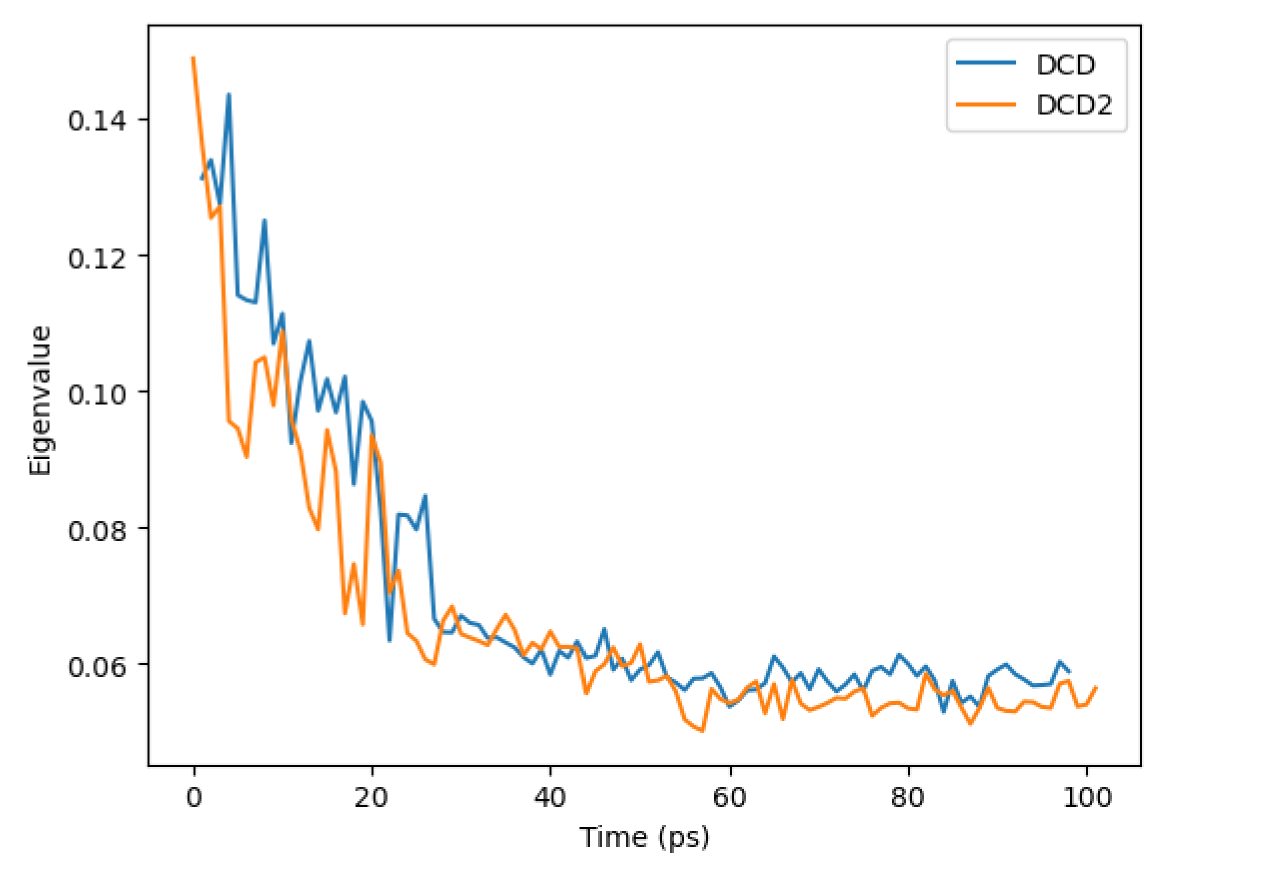
When we plot how the eigenvalue varies with time, we can see that the simulation transitions into the dominant conformation and stays there in both trajectories.

[8]:

```
linefig, lineax = plt.subplots()
plt.plot(nma1.results['times'], nma1.results['eigenvalues'], label='DCD')
plt.plot(nma2.results['times'], nma2.results['eigenvalues'], label='DCD2')
lineax.set_xlabel('Time (ps)')
lineax.set_ylabel('Eigenvalue')
plt.legend()
```

[8]:

```
<matplotlib.legend.Legend at 0x7f05540e1ac0>
```



DCD and DCD2 appear to be in similar conformation states.

Using a Gaussian network model with only close contacts

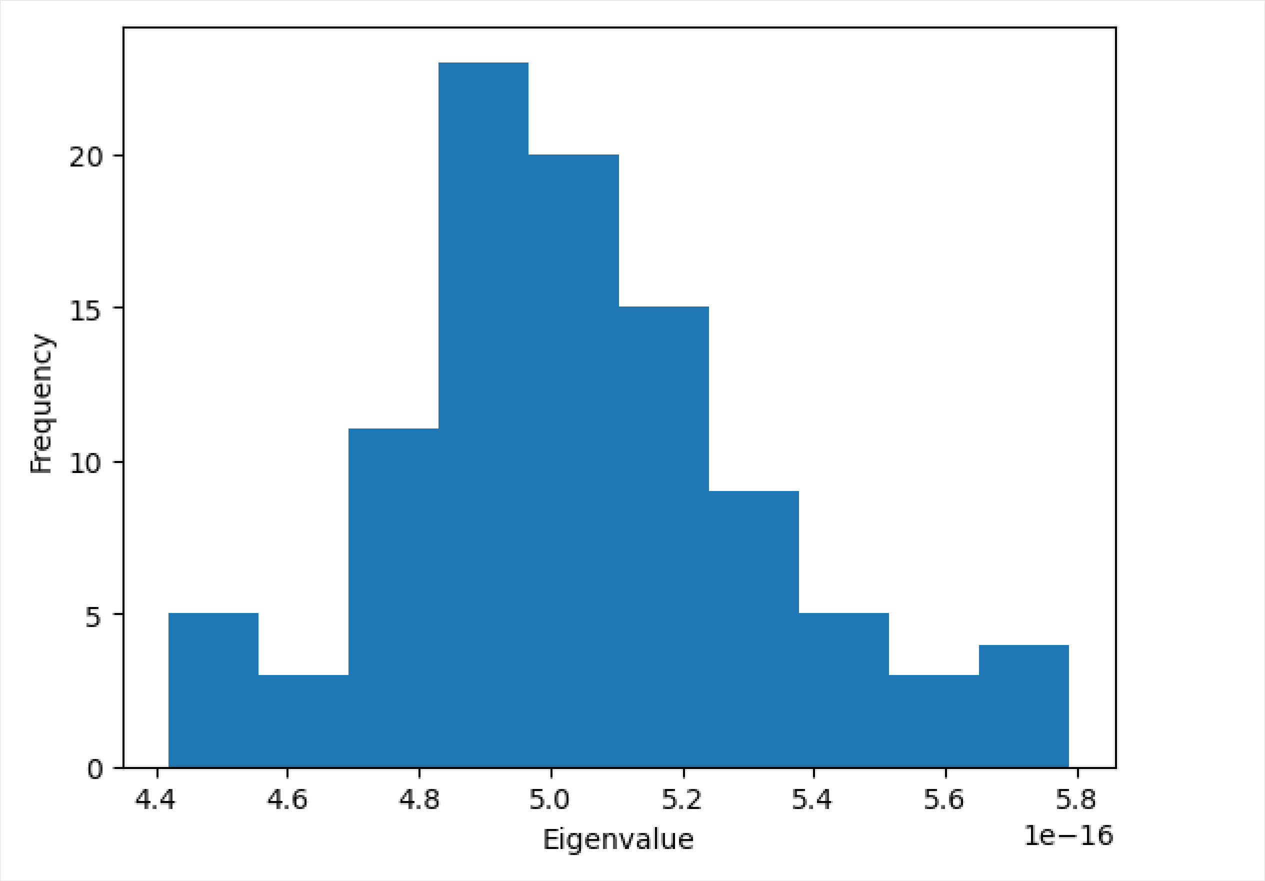
The `MDAnalysis.analysis.gnm.closeContactGNMAnalysis` class provides a version of the analysis where the Kirchhoff contact matrix is generated from close contacts between individual atoms in different residues, whereas the `GNMAnalysis` class generates it directly from all the atoms. In this close contacts class, you can weight the contact matrix by the number of atoms in the residues.

```
[9]: nma_close = gnm.closeContactGNMAnalysis(u1,
                                         select='name CA',
                                         cutoff=7.0,
                                         weights='size')
     nma_close.run()
```

```
[9]: <MDAnalysis.analysis.gnm.closeContactGNMAnalysis at 0x7f0554052c40>
```

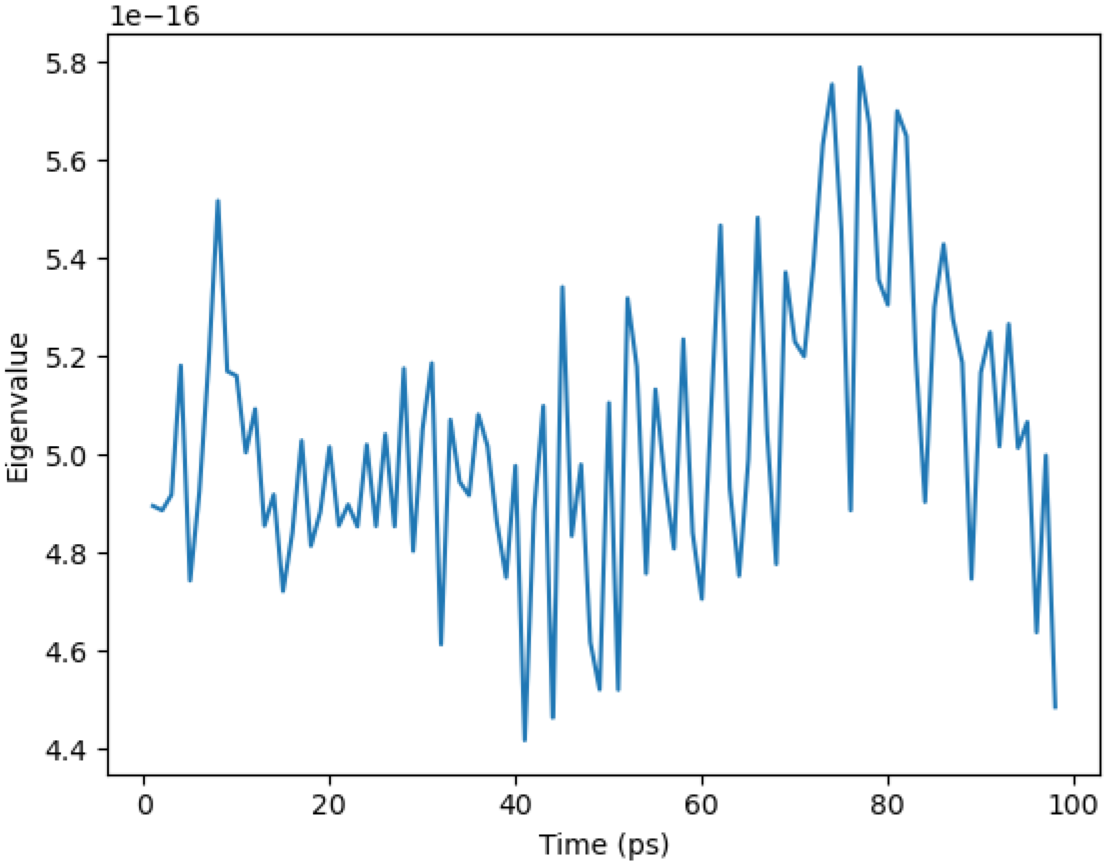
```
[10]: plt.hist(nma_close.results['eigenvalues'])
      plt.xlabel('Eigenvalue')
      plt.ylabel('Frequency')
```

```
[10]: Text(0, 0.5, 'Frequency')
```



```
[11]: ax = plt.plot(nma_close.results['times'], nma_close.results['eigenvalues'])
      plt.xlabel('Time (ps)')
      plt.ylabel('Eigenvalue')
```

```
[11]: Text(0, 0.5, 'Eigenvalue')
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Benjamin A. Hall, Samantha L. Kaye, Andy Pang, Rafael Perera, and Philip C. Biggin. Characterization of Protein Conformational States by Normal-Mode Frequencies. Journal of the American Chemical Society, 129(37):11394–11401, September 2007. 00020. URL: https://doi.org/10.1021/ja071797y, doi:10.1021/ja071797y.

[4] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Average radial distribution functions

Here we calculate the average radial cumulative distribution functions between two groups of atoms.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.17.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import TPR, XTC
from MDAnalysis.analysis import rdf
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. *[3]*

[2]:
```
u = mda.Universe(TPR, XTC)
```

Calculating the average radial distribution function for two groups of atoms

A radial distribution function $g_{ab}(r)$ describes the time-averaged density of particles in b from the reference group a at distance r. It is normalised so that it becomes 1 for large separations in a homogenous system.

$$g_{ab}(r) = (N_a N_b)^{-1} \sum_{i=1}^{N_a} \sum_{j=1}^{N_b} \langle \delta(|\mathbf{r}_i - \mathbf{r}_j| - r) \rangle$$

The radial cumulative distribution function is

$$G_{ab}(r) = \int_0^r dr' 4\pi r'^2 g_{ab}(r')$$

The average number of b particles within radius r at density ρ is:

$$N_{ab}(r) = \rho G_{ab}(r)$$

The average number of particles can be used to compute coordination numbers, such as the number of neighbours in the first solvation shell.

Below, I calculate the average RDF between each atom of residue 60 to each atom of water to look at the distribution of water over the trajectory. The RDF is limited to a spherical shell around each atom in residue 60 by `range`. Note that the range is defined around *each atom*, rather than the center-of-mass of the entire group.

If you are after non-averaged radial distribution functions, have a look at the *site-specific RDF class*. The API docs for the InterRDF class are here.

```
[3]: res60 = u.select_atoms('resid 60')
     water = u.select_atoms('resname SOL')

     irdf = rdf.InterRDF(res60, water,
                         nbins=75,  # default
                         range=(0.0, 15.0),  # distance in angstroms
                        )
     irdf.run()
```

```
[3]: <MDAnalysis.analysis.rdf.InterRDF at 0x7f0442f64370>
```

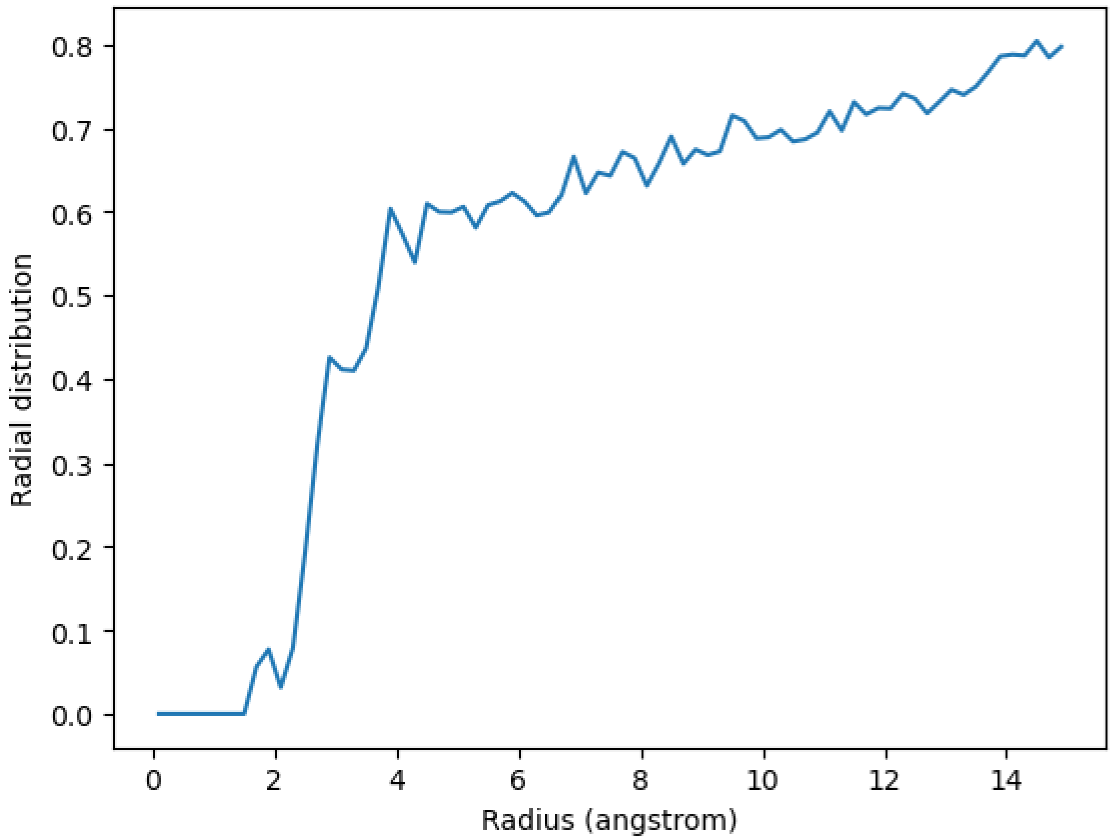
The distance bins are available at `irdf.bins` and the radial distribution function is at `irdf.rdf`.

```
[4]: irdf.results.bins
```

```
[4]: array([ 0.1,  0.3,  0.5,  0.7,  0.9,  1.1,  1.3,  1.5,  1.7,  1.9,  2.1,
             2.3,  2.5,  2.7,  2.9,  3.1,  3.3,  3.5,  3.7,  3.9,  4.1,  4.3,
             4.5,  4.7,  4.9,  5.1,  5.3,  5.5,  5.7,  5.9,  6.1,  6.3,  6.5,
             6.7,  6.9,  7.1,  7.3,  7.5,  7.7,  7.9,  8.1,  8.3,  8.5,  8.7,
             8.9,  9.1,  9.3,  9.5,  9.7,  9.9, 10.1, 10.3, 10.5, 10.7, 10.9,
            11.1, 11.3, 11.5, 11.7, 11.9, 12.1, 12.3, 12.5, 12.7, 12.9, 13.1,
            13.3, 13.5, 13.7, 13.9, 14.1, 14.3, 14.5, 14.7, 14.9])
```

```
[15]: plt.plot(irdf.results.bins, irdf.results.rdf)
      plt.xlabel('Radius (angstrom)')
      plt.ylabel('Radial distribution')
```

```
[15]: Text(0, 0.5, 'Radial distribution')
```



The total number of atom pairs in each distance bin over the trajectory, before it gets normalised over the density, number of frames, and volume of each radial shell, is at `irdf.count`.

[6]:

```
irdf.results.count
```

[6]:

```
array([0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
       0.000e+00, 0.000e+00, 7.000e+00, 1.200e+01, 6.000e+00, 1.800e+01,
       5.200e+01, 1.010e+02, 1.540e+02, 1.700e+02, 1.920e+02, 2.300e+02,
       3.000e+02, 3.950e+02, 4.140e+02, 4.290e+02, 5.310e+02, 5.700e+02,
       6.190e+02, 6.780e+02, 7.020e+02, 7.910e+02, 8.560e+02, 9.320e+02,
       9.800e+02, 1.017e+03, 1.089e+03, 1.197e+03, 1.364e+03, 1.349e+03,
       1.483e+03, 1.556e+03, 1.713e+03, 1.783e+03, 1.781e+03, 1.950e+03,
       2.145e+03, 2.140e+03, 2.298e+03, 2.379e+03, 2.501e+03, 2.777e+03,
       2.868e+03, 2.900e+03, 3.024e+03, 3.186e+03, 3.244e+03, 3.382e+03,
       3.551e+03, 3.817e+03, 3.829e+03, 4.160e+03, 4.219e+03, 4.411e+03,
       4.557e+03, 4.824e+03, 4.943e+03, 4.980e+03, 5.237e+03, 5.507e+03,
       5.630e+03, 5.878e+03, 6.193e+03, 6.533e+03, 6.740e+03, 6.922e+03,
       7.276e+03, 7.293e+03, 7.616e+03])
```

Calculating the average radial distribution function for a group of atoms to itself

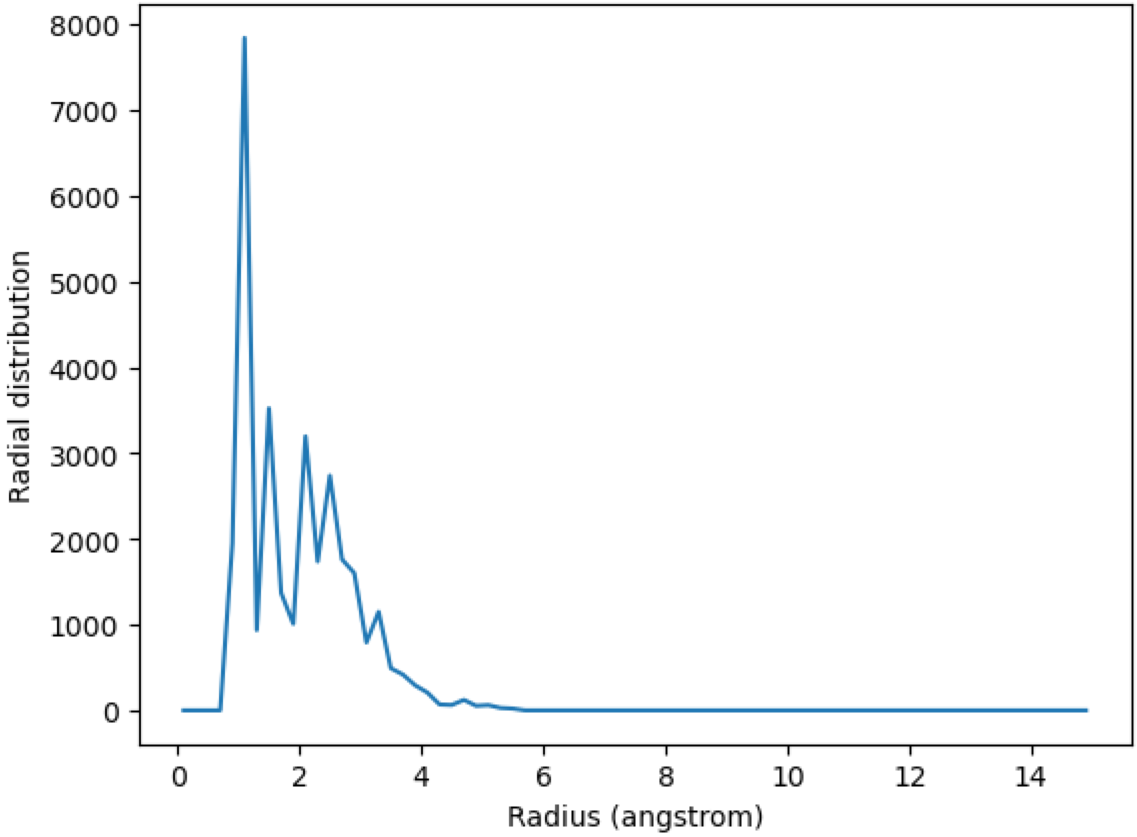
You may want to calculate the average RDF for a group of atoms where atoms overlap; for instance, looking at residue 60 around itself. In this case you should avoid including contributions from atoms interacting with themselves. The `exclusion_block` keyword allows you to mask pairs within the same chunk of atoms. Here you can pass `exclusion_block=(1, 1)` to create chunks of size 1 and avoid computing the RDF to itself.

[7]:
```
irdf2 = rdf.InterRDF(res60, res60,
                     exclusion_block=(1, 1))
irdf2.run()
```

[7]:
```
<MDAnalysis.analysis.rdf.InterRDF at 0x7f044305ffa0>
```

[8]:
```
plt.plot(irdf2.results.bins, irdf2.results.rdf)
plt.xlabel('Radius (angstrom)')
plt.ylabel('Radial distribution')
```

[8]:
```
Text(0, 0.5, 'Radial distribution')
```



Similarly, you can apply this to residues.

[9]:
```
thr = u.select_atoms('resname THR')
print('There are {} THR residues'.format(len(thr.residues)))
print('THR has {} atoms'.format(len(thr.residues[0].atoms)))
```

```
There are 11 THR residues
THR has 14 atoms
```

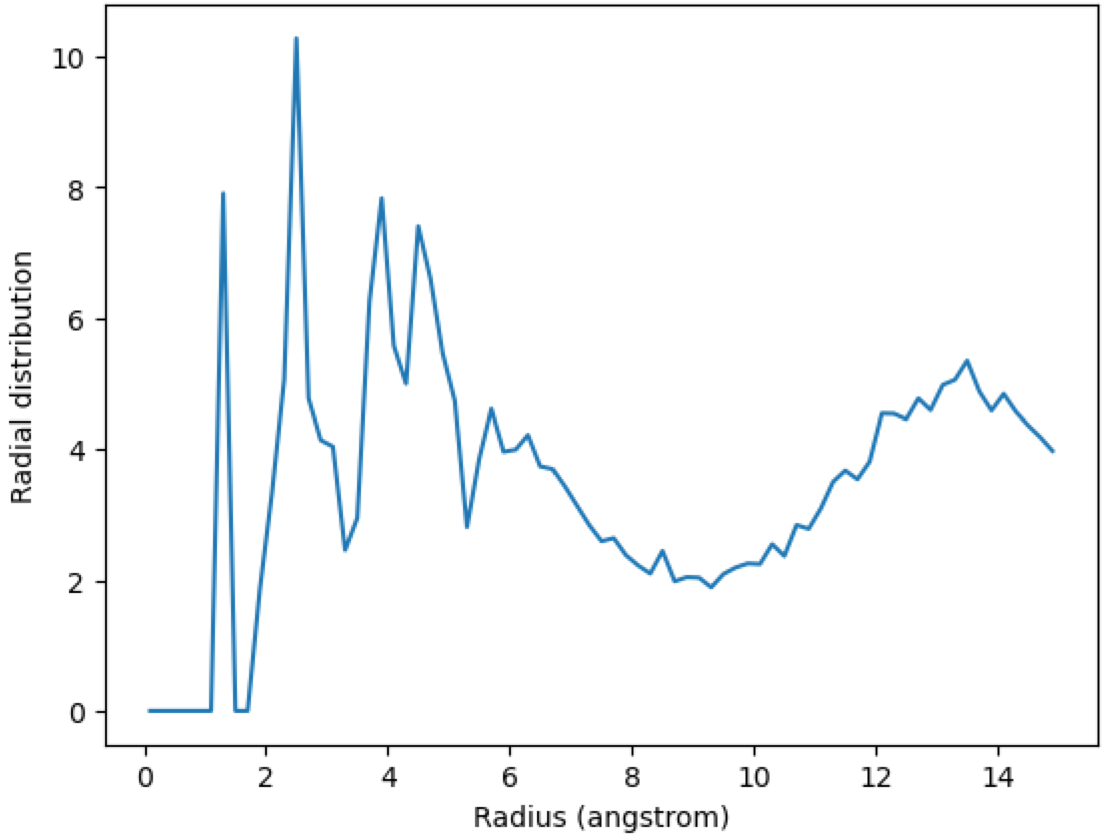
The code below calculates the RDF only using contributions from pairs of atoms where the two atoms are *not* in the same threonine residue.

```
[10]: irdf3 = rdf.InterRDF(thr, thr,
                          exclusion_block=(14, 14))
      irdf3.run()
```

```
[10]: <MDAnalysis.analysis.rdf.InterRDF at 0x7f0443041df0>
```

```
[11]: plt.plot(irdf3.results.bins, irdf3.results.rdf)
      plt.xlabel('Radius (angstrom)')
      plt.ylabel('Radial distribution')
```

```
[11]: Text(0, 0.5, 'Radial distribution')
```



If you are splitting a residue over your two selections, you can discount pairs from the same residue by choosing appropriately sized exclusion blocks.

```
[12]: first = thr.residues[0]
      print('THR has these atoms: ', ', '.join(first.atoms.names))
      thr_c1 = first.atoms.select_atoms('name C*')
      print('THR has {} carbons'.format(len(thr_c1)))
      thr_other1 = first.atoms.select_atoms('not name C*')
      print('THR has {} non carbons'.format(len(thr_other1)))
```

```
THR has these atoms:  N, H, CA, HA, CB, HB, OG1, HG1, CG2, HG21, HG22, HG23, C, O
THR has 4 carbons
THR has 10 non carbons
```

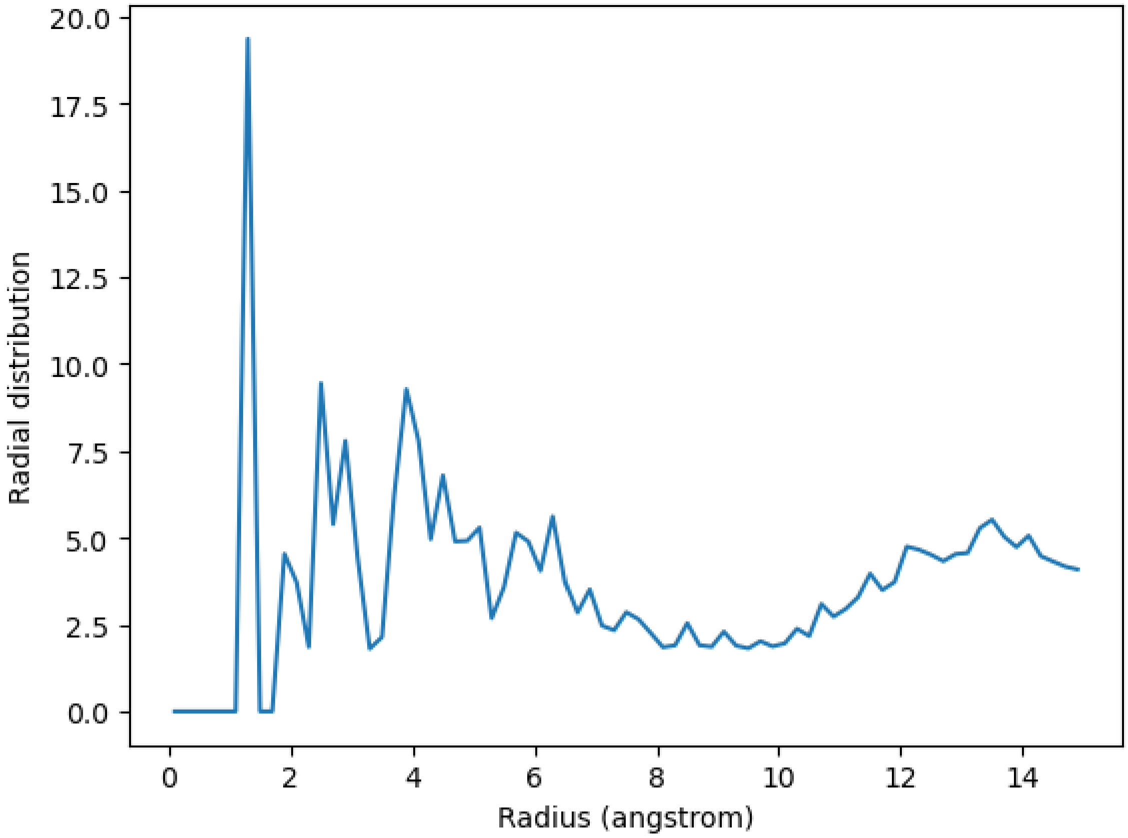
The exclusion_block here ensures that the RDF is only computed from threonine carbons to atoms in different threonine residues.

[13]:
```
thr_c = thr.select_atoms('name C*')
thr_other = thr.select_atoms('not name C*')

irdf4 = rdf.InterRDF(thr_c, thr_other,
                     exclusion_block=(4, 10))
irdf4.run()
```

[13]:
```
<MDAnalysis.analysis.rdf.InterRDF at 0x7f0442bc0af0>
```

[14]:
```
plt.plot(irdf4.results.bins, irdf4.results.rdf)
plt.xlabel('Radius (angstrom)')
plt.ylabel('Radial distribution')
```

[14]:
```
Text(0, 0.5, 'Radial distribution')
```



References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Calculating the RDF atom-to-atom

We calculate the site-specific radial distribution functions of solvent around certain atoms.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.19.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

[1]:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import TPR, XTC
from MDAnalysis.analysis import rdf
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

[2]:

```
u = mda.Universe(TPR, XTC)
```

Calculating the site-specific radial distribution function

A radial distribution function $g_{ab}(r)$ describes the time-averaged density of particles in b from the reference group a at distance r. It is normalised so that it becomes 1 for large separations in a homogenous system. See *the tutorial on averaged RDFs* for more information. The `InterRDF_s` class (API docs) allows you to compute RDFs on an atom-to-atom basis, rather than simply giving the averaged RDF as in `InterRDF`.

Below, I calculate the RDF between selected alpha-carbons and the water atoms within 15 angstroms of CA60, *in the first frame of the trajectory*. The water group does not update over the trajectory as the water moves towards and away from the alpha-carbon.

The RDF is limited to a spherical shell around each atom by `range`. Note that the range is defined around *each atom*, rather than the center-of-mass of the entire group.

If `density=True`, the final RDF is over the average density of the selected atoms in the trajectory box, making it comparable to the output of `rdf.InterRDF`. If `density=False`, the density is not taken into account. This can make it difficult to compare RDFs between AtomGroups that contain different numbers of atoms.

[3]:

```
ca60 = u.select_atoms('resid 61 and name CA')
ca61 = u.select_atoms('resid 62 and name CA')
ca62 = u.select_atoms('resid 63 and name CA')
water = u.select_atoms('resname SOL and sphzone 15 group sel_a', sel_a=ca60)

ags = [[ca60+ca61, water], [ca62, water]]

ss_rdf = rdf.InterRDF_s(u, ags,
                        nbins=75,  # default
                        range=(0.0, 15.0),  # distance
                        norm='density',
                       )
ss_rdf.run()
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/analysis/
↪rdf.py:531: DeprecationWarning: The `u` attribute is superflous and will be removed in␣
↪MDAnalysis 3.0.0.
  warnings.warn("The `u` attribute is superflous and will be removed "
```

[3]:

```
<MDAnalysis.analysis.rdf.InterRDF_s at 0x7faa885ab280>
```

Like `rdf.InterRDF`, the distance bins are available at `ss_rdf.bins`.

[4]:

```
ss_rdf.results.bins
```

[4]:

```
array([ 0.1,  0.3,  0.5,  0.7,  0.9,  1.1,  1.3,  1.5,  1.7,  1.9,  2.1,
        2.3,  2.5,  2.7,  2.9,  3.1,  3.3,  3.5,  3.7,  3.9,  4.1,  4.3,
        4.5,  4.7,  4.9,  5.1,  5.3,  5.5,  5.7,  5.9,  6.1,  6.3,  6.5,
        6.7,  6.9,  7.1,  7.3,  7.5,  7.7,  7.9,  8.1,  8.3,  8.5,  8.7,
        8.9,  9.1,  9.3,  9.5,  9.7,  9.9, 10.1, 10.3, 10.5, 10.7, 10.9,
       11.1, 11.3, 11.5, 11.7, 11.9, 12.1, 12.3, 12.5, 12.7, 12.9, 13.1,
       13.3, 13.5, 13.7, 13.9, 14.1, 14.3, 14.5, 14.7, 14.9])
```

`ss_rdf.rdf` contains the atom-pairwise RDF for each of your pairs of AtomGroups. It is a list with the same length as your list of pairs `ags`. A result array has the shape `(len(ag1), len(ag2), nbins)` for the AtomGroup pair `(ag1, ag2)`.

[5]:

```
print('There are {} water atoms'.format(len(water)))
print('The first result array has shape: {}'.format(ss_rdf.results.rdf[0].shape))
print('The second result array has shape: {}'.format(ss_rdf.results.rdf[1].shape))
```

```
There are 1041 water atoms
The first result array has shape: (2, 1041, 75)
The second result array has shape: (1, 1041, 75)
```

Index the results array to get the RDF for a particular pair of atoms. `ss_rdf.rdf[i][j][k]` will return the RDF between atoms j and k in the i-th pair of atom groups. For example, below we get the RDF between the alpha-carbon in residue 61 (i.e. the second atom of the first atom group) and the 571st atom of water.

[6]:

```
ca61_h2o_571 = ss_rdf.results.rdf[0][1][570]
ca61_h2o_571
```

[6]:

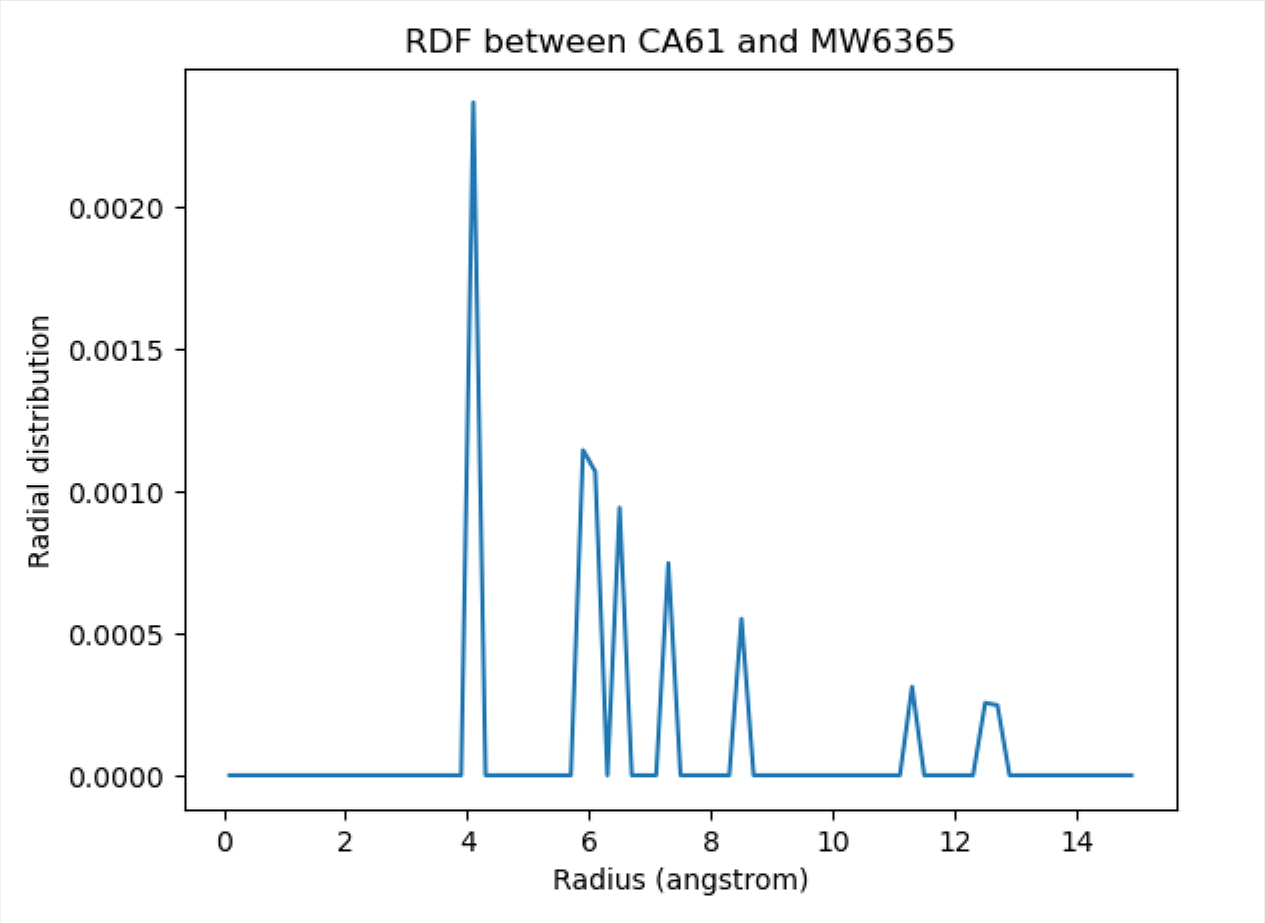
```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.0023665 , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.00114292,
       0.00106921, 0.        , 0.00094167, 0.        , 0.        ,
       0.        , 0.0007466 , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.00055068, 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.0003116 , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.00025464, 0.00024669, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

[7]:

```
plt.plot(ss_rdf.results.bins, ca61_h2o_571)
w570 = water[570]
plt.xlabel('Radius (angstrom)')
plt.ylabel('Radial distribution')
plt.title('RDF between CA61 and {}{}'.format(w570.name, w570.resid))
```

[7]:

```
Text(0.5, 1.0, 'RDF between CA61 and MW6365')
```



If you are having trouble finding pairs of atoms where the results are not simply 0, you can use Numpy functions to find the indices of the nonzero values. Below we count the nonzero entries in the first `rdf` array.

[8]:
```
j, k, nbin = np.nonzero(ss_rdf.results.rdf[0])
print(len(j), len(k), len(nbin))
```

```
4374 4374 4374
```

Each triplet of `[j, k, nbin]` indices is a nonzero value, corresponding to the `nbin`th bin between atoms j and k. For example:

[9]:
```
print(f"{ss_rdf.results.rdf[0][j[0], k[0], nbin[0]]: .5f}")
```

```
 0.00028
```

Right now, we don't care which particular bin has a nonzero value. Let's find which water atom is the most present around the alpha-carbon of residue 60, i.e. the first atom.

[10]:
```
# where j == 0, representing the first atom
water_for_ca60 = k[j==0]
# count how many of each atom index are in array
k_values, k_counts = np.unique(water_for_ca60,
                               return_counts=True)
# get the first k value with the most counts
k_max = k_values[np.argmax(k_counts)]
```

(continues on next page)

(continued from previous page)

```
print('The water atom with the highest distribution '
      'around CA60 has index {}'.format(k_max))
```

```
The water atom with the highest distribution around CA60 has index 568
```

You can also calculate a cumulative distribution function for each of your results with `ss_rdf.get_cdf()`. This is the actual count of atoms within the given range, averaged over the trajectory; the volume of each radial shell is not taken into account. The result then gets saved into `ss_rdf.cdf`. The CDF has the same shape as the corresponding RDF array.

[11]:

```
cdf = ss_rdf.get_cdf()
print(cdf[0].shape)
```

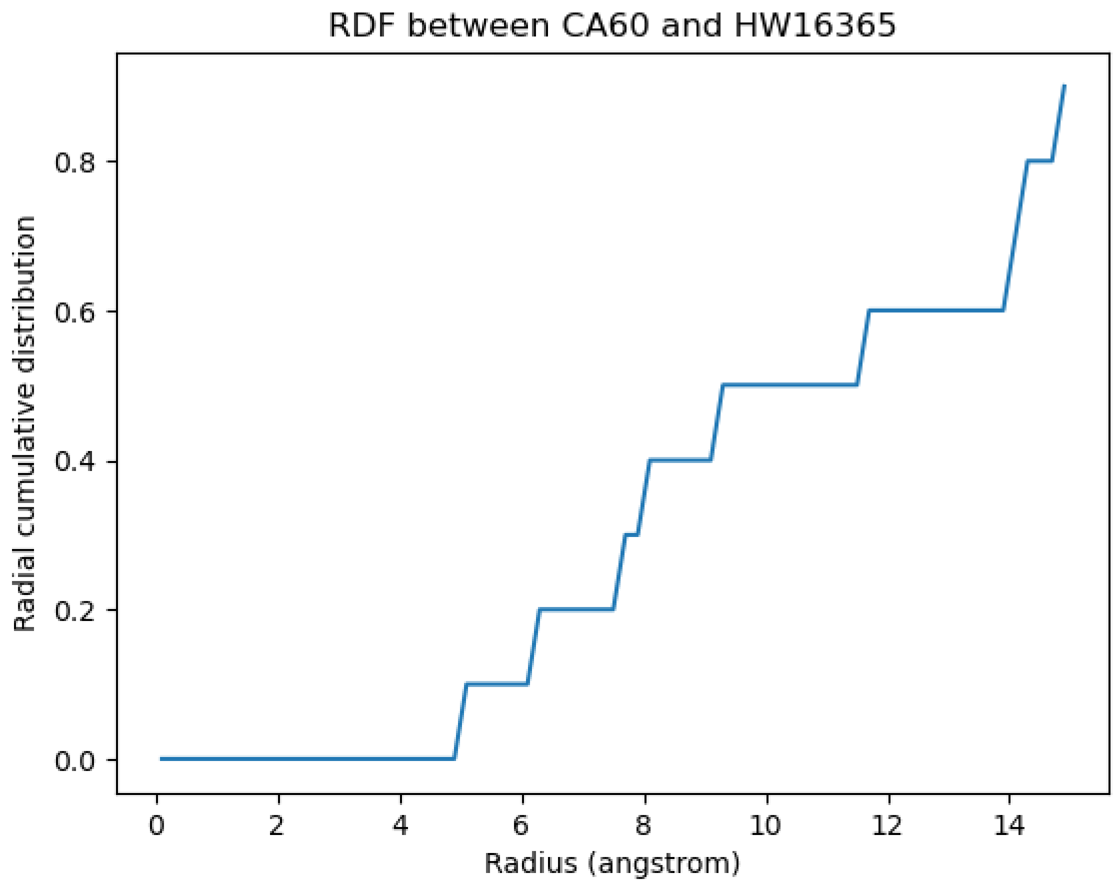
```
(2, 1041, 75)
```

[12]:

```
plt.plot(ss_rdf.results.bins, ss_rdf.results.cdf[0][0][568])
w568 = water[568]
plt.xlabel('Radius (angstrom)')
plt.ylabel('Radial cumulative distribution')
plt.title('RDF between CA60 and {}{}'.format(w568.name, w568.resid))
```

[12]:

```
Text(0.5, 1.0, 'RDF between CA60 and HW16365')
```



The site-specific RDF without densities

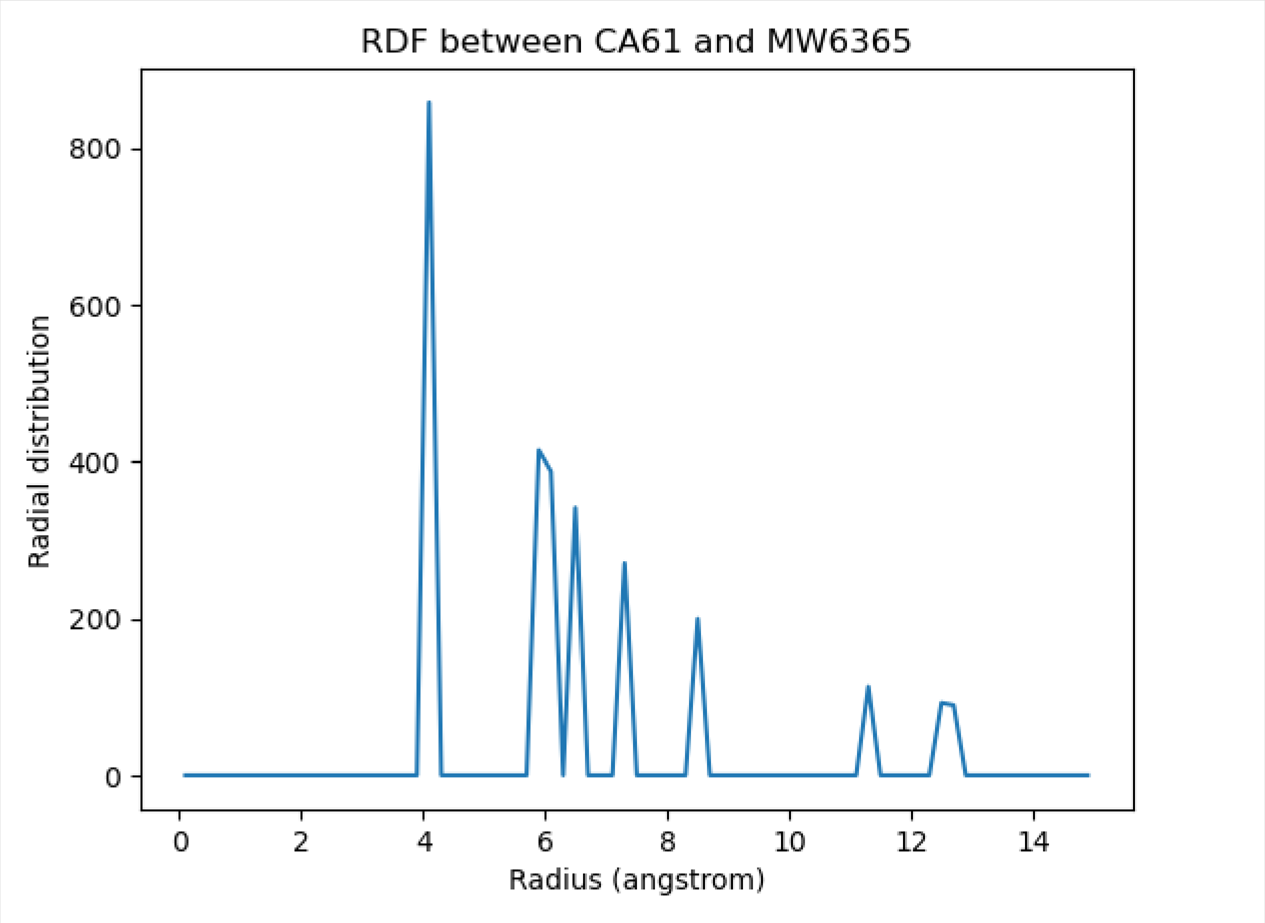
When the density of the selected atom groups over the box volume is not accounted for, your distribution values will be proportionally lower.

[13]:
```
ss_rdf_nodensity = rdf.InterRDF_s(u, ags,
                                  nbins=75,  # default
                                  range=(0.0, 15.0),  # distance
                                  density=False,
                                 )
ss_rdf_nodensity.run()
ss_rdf_nodensity.get_cdf()
```

[13]:
```
[array([[[0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         ...,
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1]],

        [[0. , 0. , 0. , ..., 0. , 0.1, 0.1],
         [0. , 0. , 0. , ..., 0. , 0. , 0. ],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         ...,
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1]]]),
 array([[[0. , 0. , 0. , ..., 0. , 0.1, 0.1],
         [0. , 0. , 0. , ..., 0. , 0. , 0. ],
         [0. , 0. , 0. , ..., 0.1, 0.1, 0.1],
         ...,
         [0. , 0. , 0. , ..., 0. , 0. , 0.1],
         [0. , 0. , 0. , ..., 0. , 0. , 0. ],
         [0. , 0. , 0. , ..., 0. , 0. , 0. ]]])]
```

[14]:
```
plt.plot(ss_rdf_nodensity.results.bins,
         ss_rdf_nodensity.results.rdf[0][1][570])
plt.xlabel('Radius (angstrom)')
plt.ylabel('Radial distribution')
plt.title('RDF between CA61 and {}{}'.format(w570.name, w570.resid))
```

[14]:
```
Text(0.5, 1.0, 'RDF between CA61 and MW6365')
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Protein dihedral angle analysis

We look at backbone dihedral angles and generate Ramachandran and Janin plots.

The methods and examples shown here are only applicable to Universes where protein residue names have standard names, i.e. the backbone is comprised of –N–CA–C–N–CA– atoms.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.19.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import GRO, XTC
from MDAnalysis.analysis import dihedrals
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

Loading files

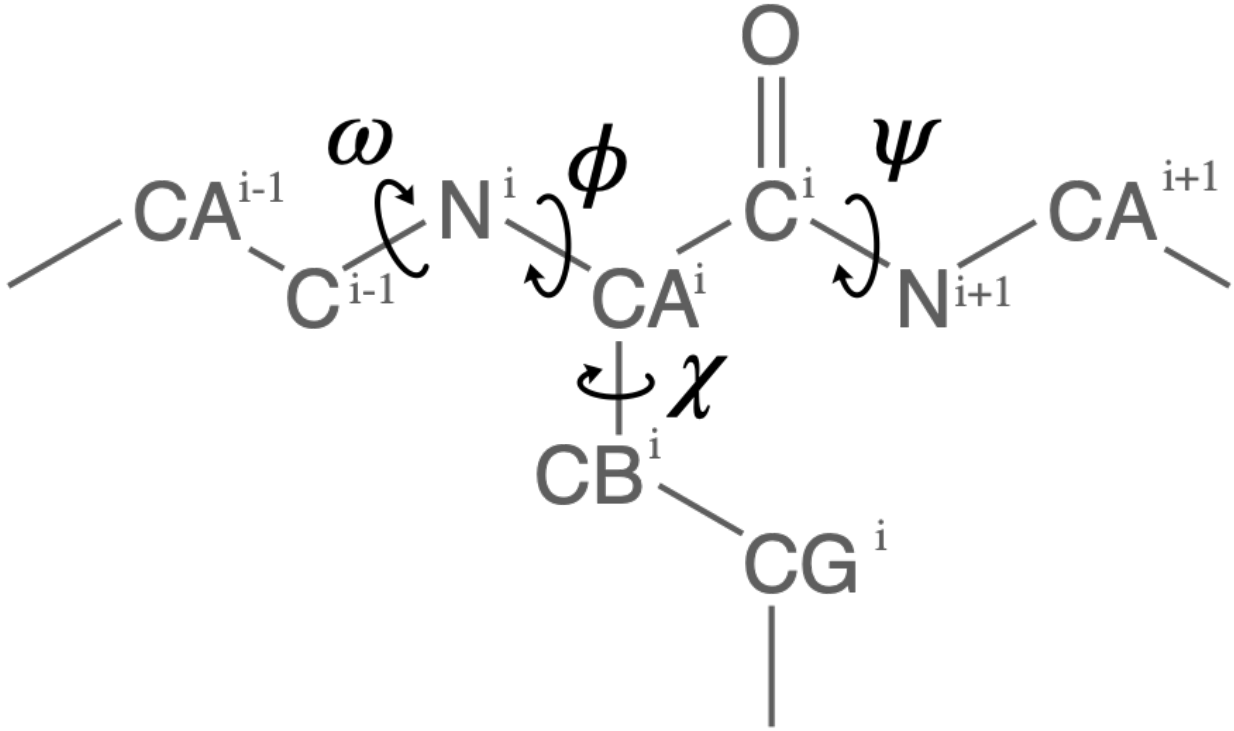
The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

[2]:
```
u = mda.Universe(GRO, XTC)
protein = u.select_atoms('protein')
print('There are {} residues in the protein'.format(len(protein.residues)))
```

```
There are 214 residues in the protein
```

Selecting dihedral atom groups

Proteins have canonical dihedral angles defined on the backbone atoms. ϕ (phi), ψ (psi) and ω (omega) are backbone angles. The side-chain dihedral angles are called χ_n (chi-n), and can vary in number.



MDAnalysis allows you to directly select the atoms involved in the ϕ, ψ, ω, and χ_1 angles, provided that your protein atoms have standard names. If MDAnalysis cannot find atoms matching the names that it expects, it will return `None`. You can see below that `phi_selection()` returns an ordered AtomGroup of the atoms in the ϕ angle of a residue if they can be found, and `None` if not.

```
[3]: for res in u.residues[210:220]:
         phi = res.phi_selection()
         if phi is None:
             names = None
         else:
             names = phi.names
         print('{}: {} '.format(res.resname, names))
```

```
LYS: ['C' 'N' 'CA' 'C']
ILE: ['C' 'N' 'CA' 'C']
LEU: ['C' 'N' 'CA' 'C']
GLY: ['C' 'N' 'CA' 'C']
SOL: None
SOL: None
SOL: None
SOL: None
SOL: None
SOL: None
```

Similar functions exist for the other angles:

- ψ angle (`Residue.psi_selection`)
- ω angle (`Residue.omega_selection`)
- χ_1 angle (`Residue.chi1_selection`)

Calculating dihedral angles

Dihedral angles can be calculated directly from the AtomGroup, by converting it to a `Dihedral` object.

```
[4]: omegas = [res.omega_selection() for res in protein.residues[5:10]]
     omegas[0].dihedral.value()
```

```
[4]: -169.78220560918737
```

The `analysis.dihedrals.Dihedral` class (API docs) can be used to rapidly calculate dihedrals for AtomGroups over the entire trajectory.

```
[5]: dihs = dihedrals.Dihedral(omegas).run()
```

The angles are saved in `dihs.angles`, in an array with the shape (`n_frames`, `n_atomgroups`).

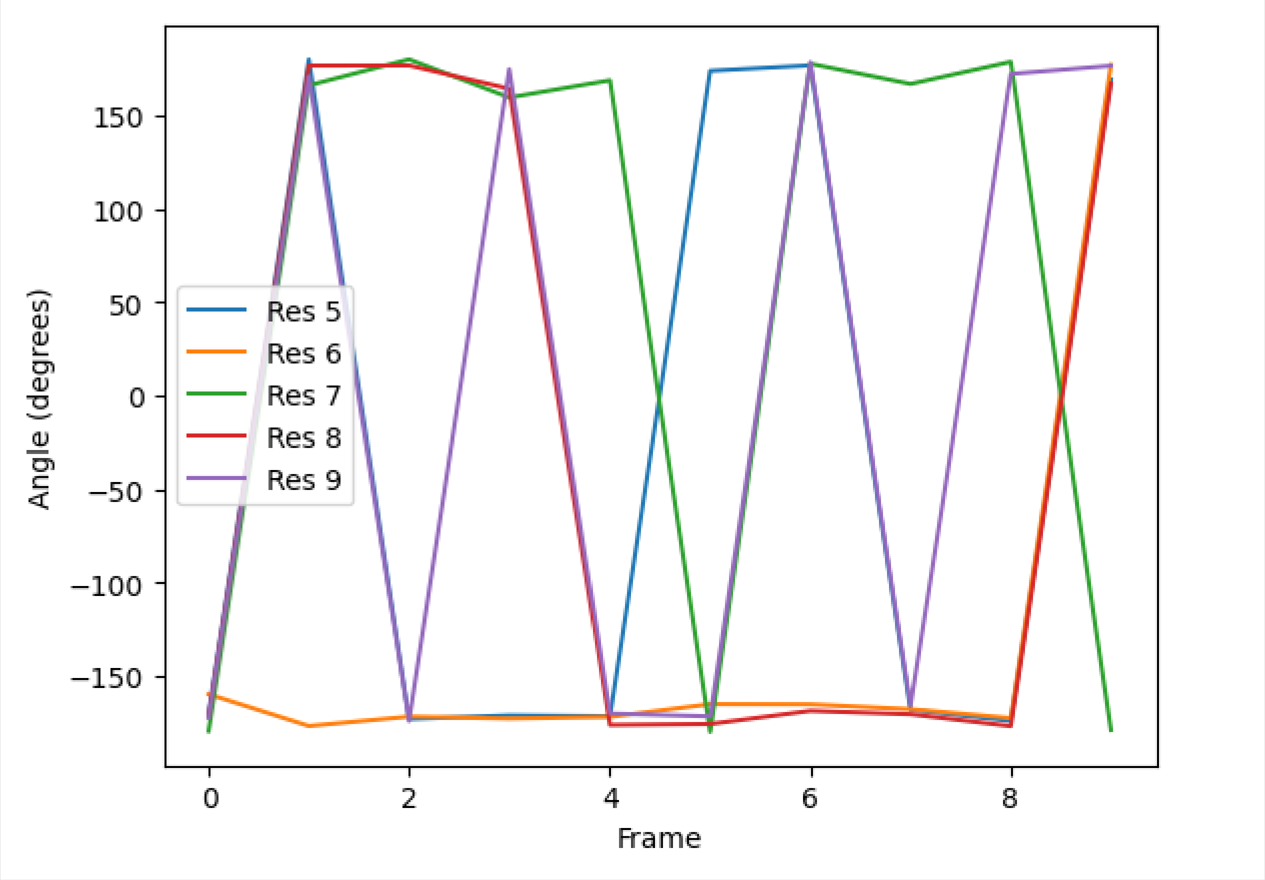
```
[6]: dihs.results.angles.shape
```

```
[6]: (10, 5)
```

Plotting

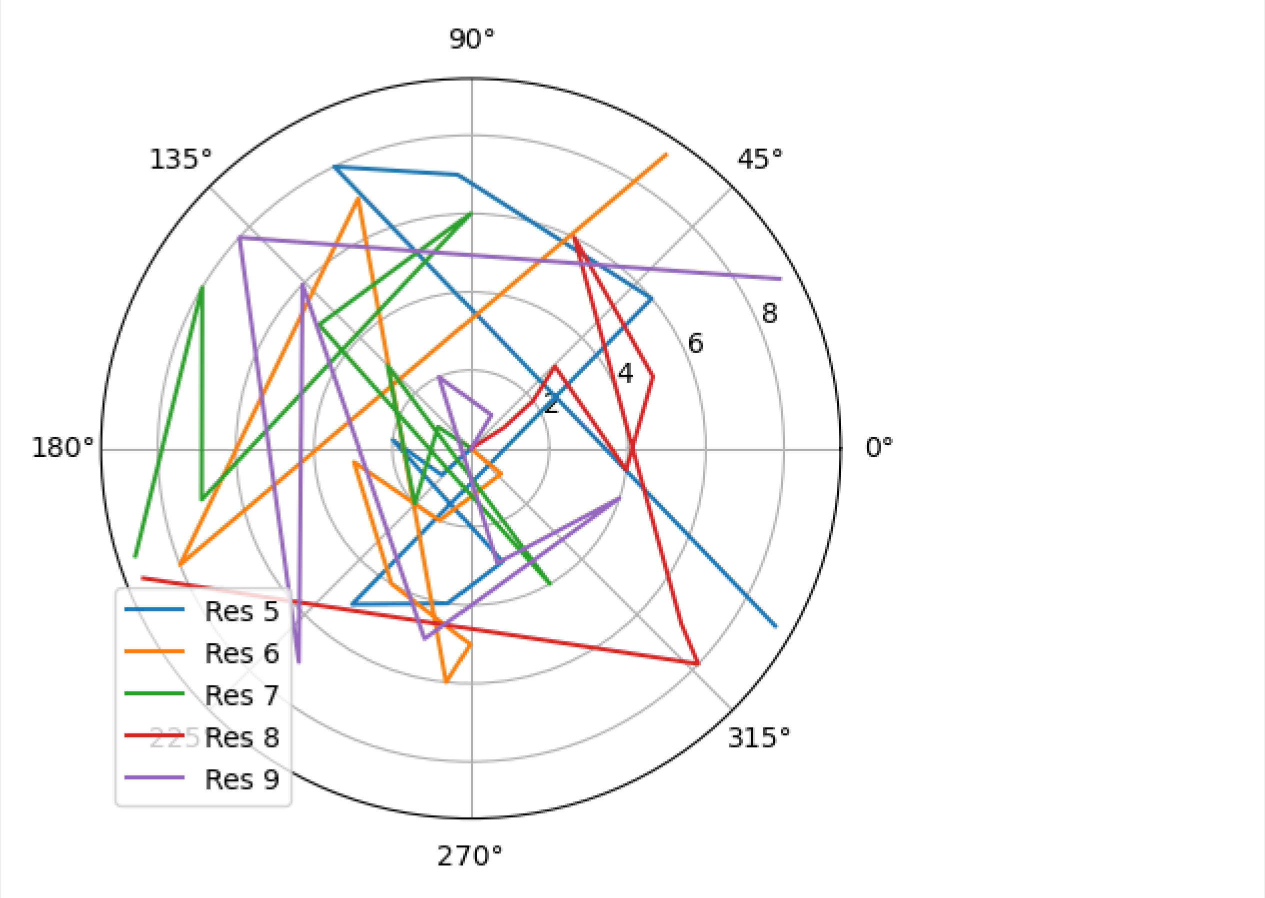
```
[7]: labels = ['Res {}'.format(n) for n in np.arange(5, 10)]
     for ang, label in zip(dihs.results.angles.T, labels):
         plt.plot(ang, label=label)
     plt.xlabel('Frame')
     plt.ylabel('Angle (degrees)')
     plt.legend()
```

```
[7]: <matplotlib.legend.Legend at 0x7fcd2ba11580>
```



```
[8]: fig_polar = plt.figure()
     ax_polar = fig_polar.add_subplot(111, projection='polar')
     frames = np.arange(10)
     for res, label in zip(dihs.results.angles.T, labels):
         c = ax_polar.plot(res, frames, label=label)
     plt.legend()
```

```
[8]: <matplotlib.legend.Legend at 0x7fcd2b8cb430>
```



Ramachandran analysis

The `Ramachandran` class (API docs) calculates the ϕ and ψ angles of the selected residues over the course of the trajectory, again saving it into `.angles`. If residues are given that do not contain a ϕ and ψ angle, they are omitted from the results. For example, the angles returned are from every residue in the protein *except* the first and last, for which a ϕ angle and a ψ angle do not exist, respectively.

The returned `angles` are in the shape (`n_frames`, `n_residues`, 2) where the last dimension holds the ϕ and ψ angle.

[9]:
```
rama = dihedrals.Ramachandran(protein).run()
print(rama.results.angles.shape)
```

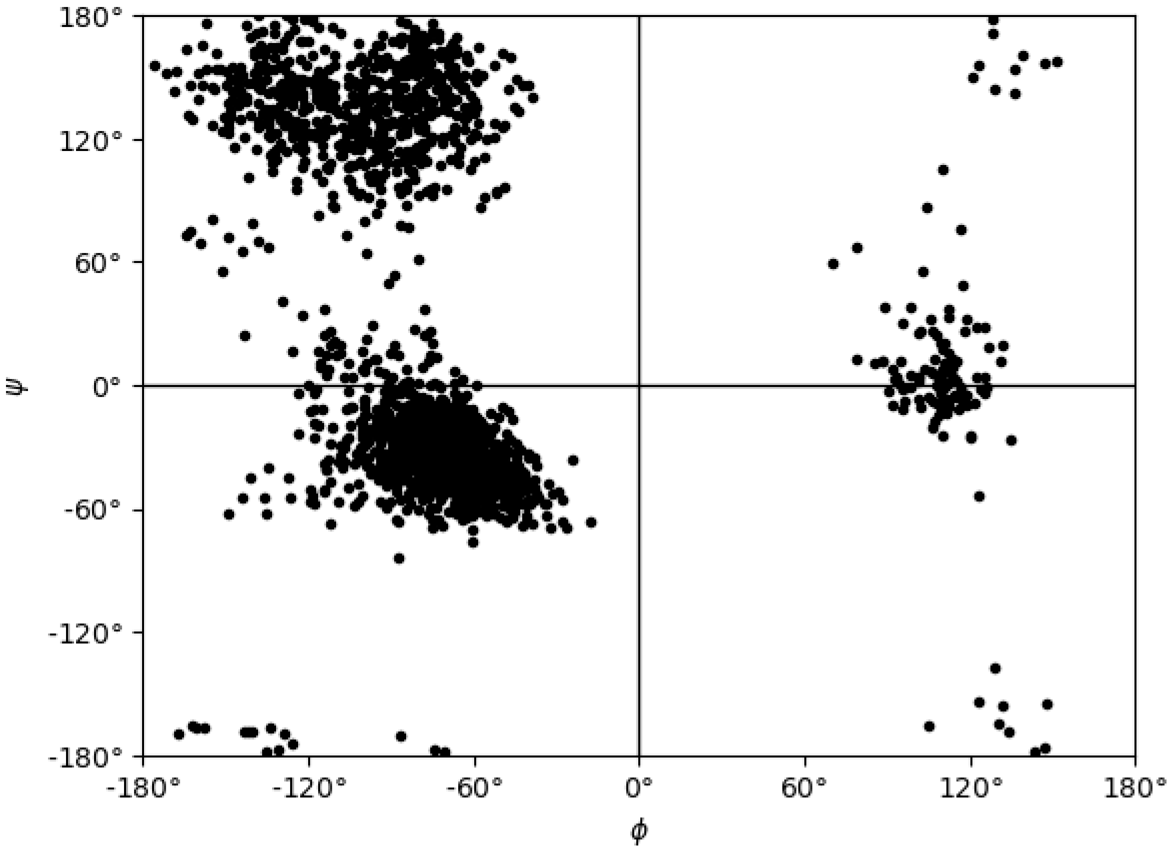
```
(10, 212, 2)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/analysis/
↪dihedrals.py:407: UserWarning: Cannot determine phi and psi angles for the first or␣
↪last residues
  warnings.warn("Cannot determine phi and psi angles for the first "
```

You can plot this yourself, but `Ramachandran.plot()` is a convenience method that plots the data from each time step onto a standard Ramachandran plot. You can call it with no arguments; any keyword arguments that you give (except `ax` and `ref`) will be passed to `matplotlib.axes.Axes.scatter` to modify your plot.

[10]:
```
rama.plot(color='black', marker='.')
```

[10]:
```
<AxesSubplot: xlabel='$\\phi$', ylabel='$\\psi$'>
```



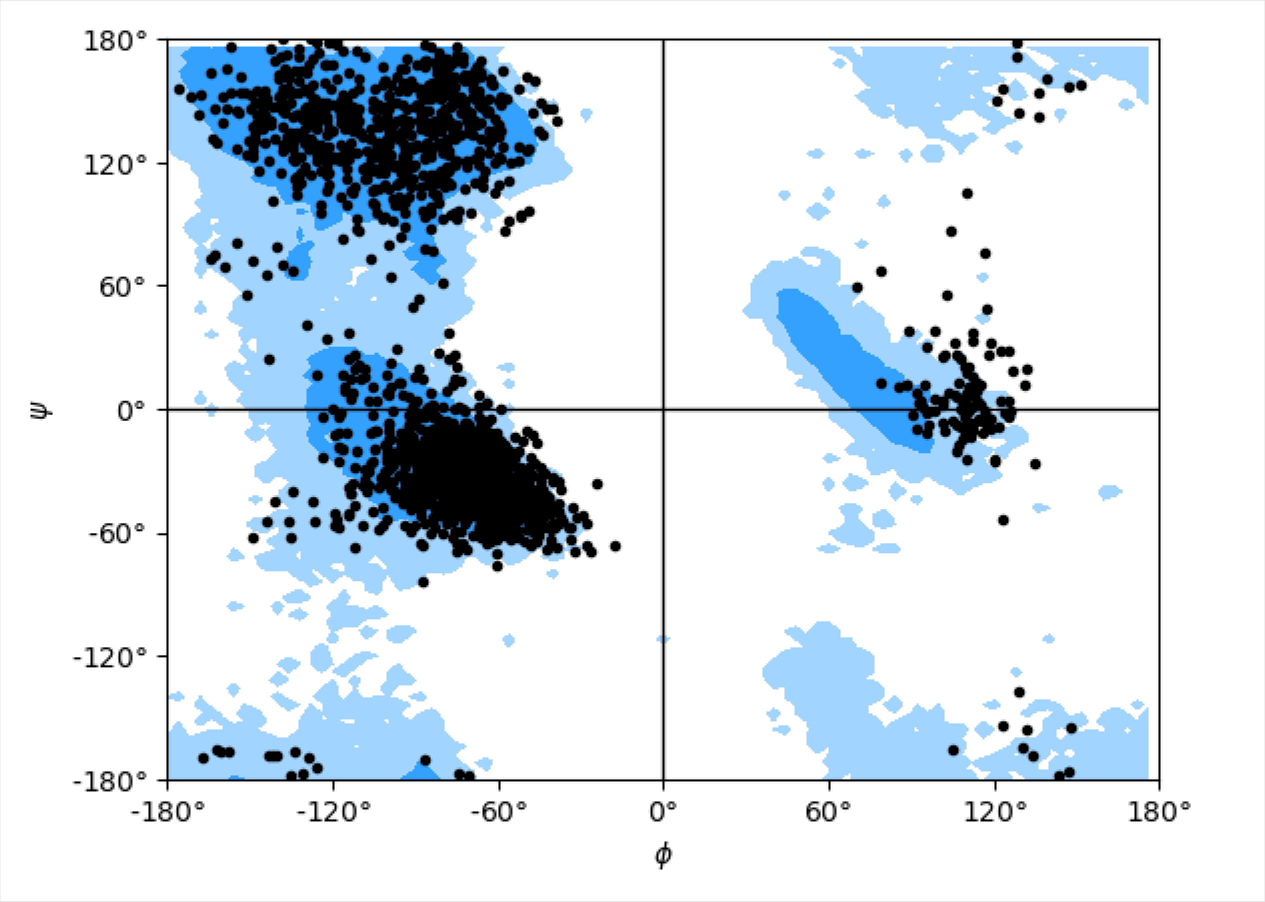
If you set `ref=True`, your data will be plotted with areas that show the allowed (dark blue) and marginally allowed (lighter blue) regions.

Note

These regions are computed from a reference set of 500 PDB files from ([LDA+03]). The allowed region includes 90% data points, while the marginally allowed region includes 99% data points.

[11]:
```
rama.plot(color='black', marker='.', ref=True)
```

[11]:
```
<AxesSubplot: xlabel='$\\phi$', ylabel='$\\psi$'>
```



Janin analysis

The Janin class (API docs) works similarly to the Ramachandran analysis, but looks at the χ_1 and χ_2 angles instead. It therefore ignores all residues without a long enough side-chain, such as alanine, cysteine, and so on.

Again, the returned angles are in the shape (n_frames, n_residues, 2) where the last dimension holds the χ_1 and χ_2 angle. We can see that only about half of the residues in AdK have side-chains long enough for this analysis.

```
[12]: janin = dihedrals.Janin(protein).run()
      print(janin.results.angles.shape)
```

```
(10, 129, 2)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/analysis/
↪dihedrals.py:589: UserWarning: All residues selected with 'resname ALA CYS* GLY PRO␣
↪SER THR VAL' have been removed from the selection.
  warnings.warn(f"All residues selected with '{select_remove}' "
```

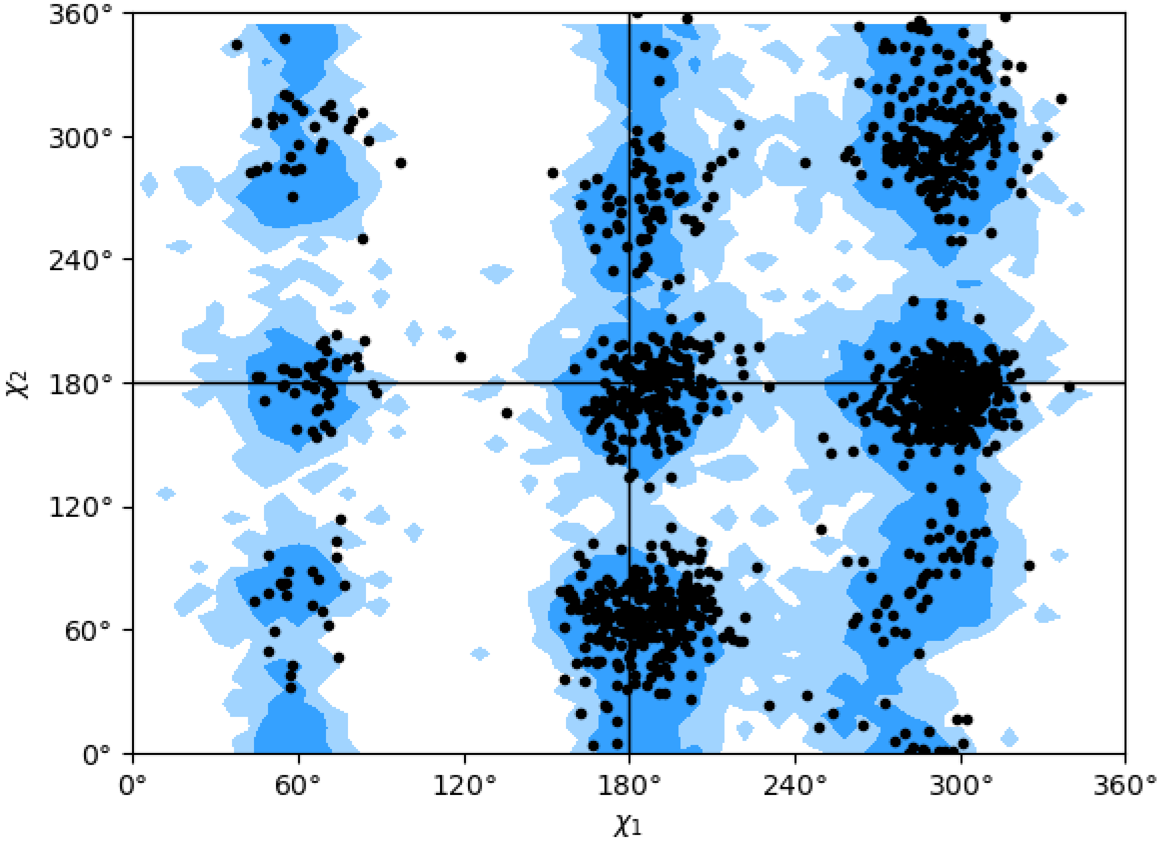
The Janin class also contains a plot() method.

Note

The reference regions here are also computed from the reference set of 500 PDB files from ([LDA+03]) (the allowed region includes 90% data points, while the marginally allowed region includes 98% data points). Information about general Janin regions is from ([JWLM78]).

```
[13]: janin.plot(ref=True, marker='.', color='black')
```

```
[13]: <AxesSubplot: xlabel='$\\chi_1$', ylabel='$\\chi_2$'>
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Joël Janin, Shoshanna Wodak, Michael Levitt, and Bernard Maigret. Conformation of amino acid side-chains in proteins. Journal of Molecular Biology, 125(3):357 – 386, 1978. 00874. URL: http://www.sciencedirect.com/science/article/pii/0022283678904084, doi:10.1016/0022-2836(78)90408-4.

[4] Simon C. Lovell, Ian W. Davis, W. Bryan Arendall, Paul I. W. de Bakker, J. Michael Word, Michael G. Prisant, Jane S. Richardson, and David C. Richardson. Structure validation by C geometry: , and C deviation. Proteins:

Structure, Function, and Bioinformatics, 50(3):437–450, January 2003. 03997. URL: http://doi.wiley.com/10.1002/prot.10286, doi:10.1002/prot.10286.

[5] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Helix analysis

We look at protein helices with HELANAL.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 2.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- matplotlib
- nglview

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

Note

`MDAnalysis.analysis.helix_analysis.HELANAL` implements the HELANAL algorithm from [BKV00], which itself uses the method of [SM67] to characterise each local axis. Please cite them when using this module in published work.

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis import helix_analysis as hel
import matplotlib.pyplot as plt
# import nglview as nv
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

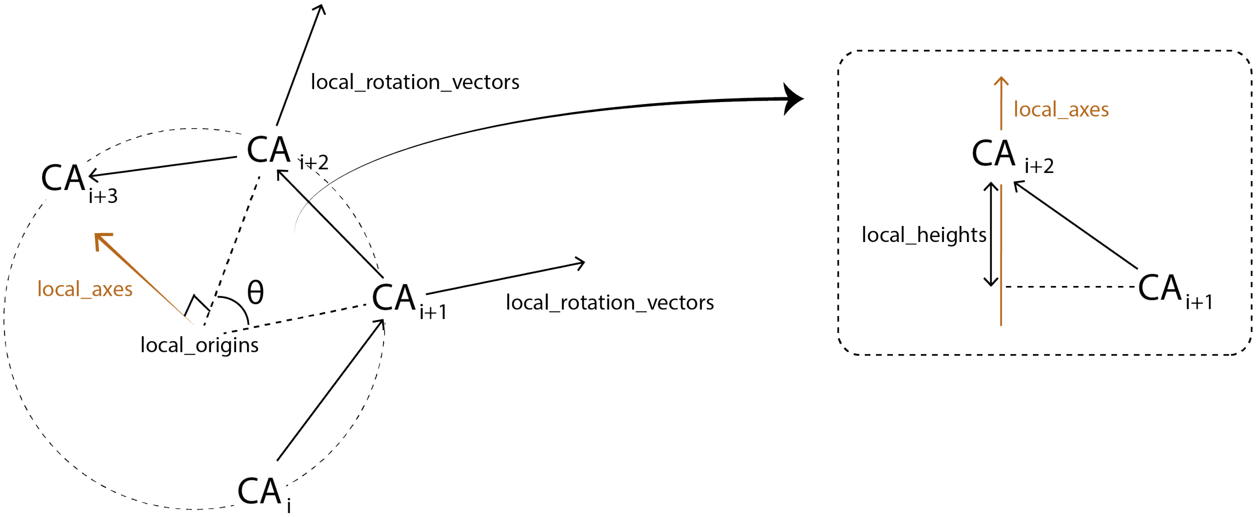
[2]:
```
u = mda.Universe(PSF, DCD)
```

```
/Users/lily/pydev/mdanalysis/package/MDAnalysis/coordinates/DCD.py:165:␣
↪DeprecationWarning: DCDReader currently makes independent timesteps by copying self.ts␣
↪while other readers update self.ts inplace. This behavior will be changed in 3.0 to be␣
↪the same as other readers. Read more at https://github.com/MDAnalysis/mdanalysis/
↪issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Helix analysis

HELANAL can be used to characterize the geometry of helices with at least 9 residues. The geometry of an alpha helix is characterized by computing local helix axes and local helix origins for four contiguous C-alpha atoms, using the procedure of Sugeta and Miyazawa ([SM67]) and sliding this window over the length of the helix in steps of one C-alpha atom.

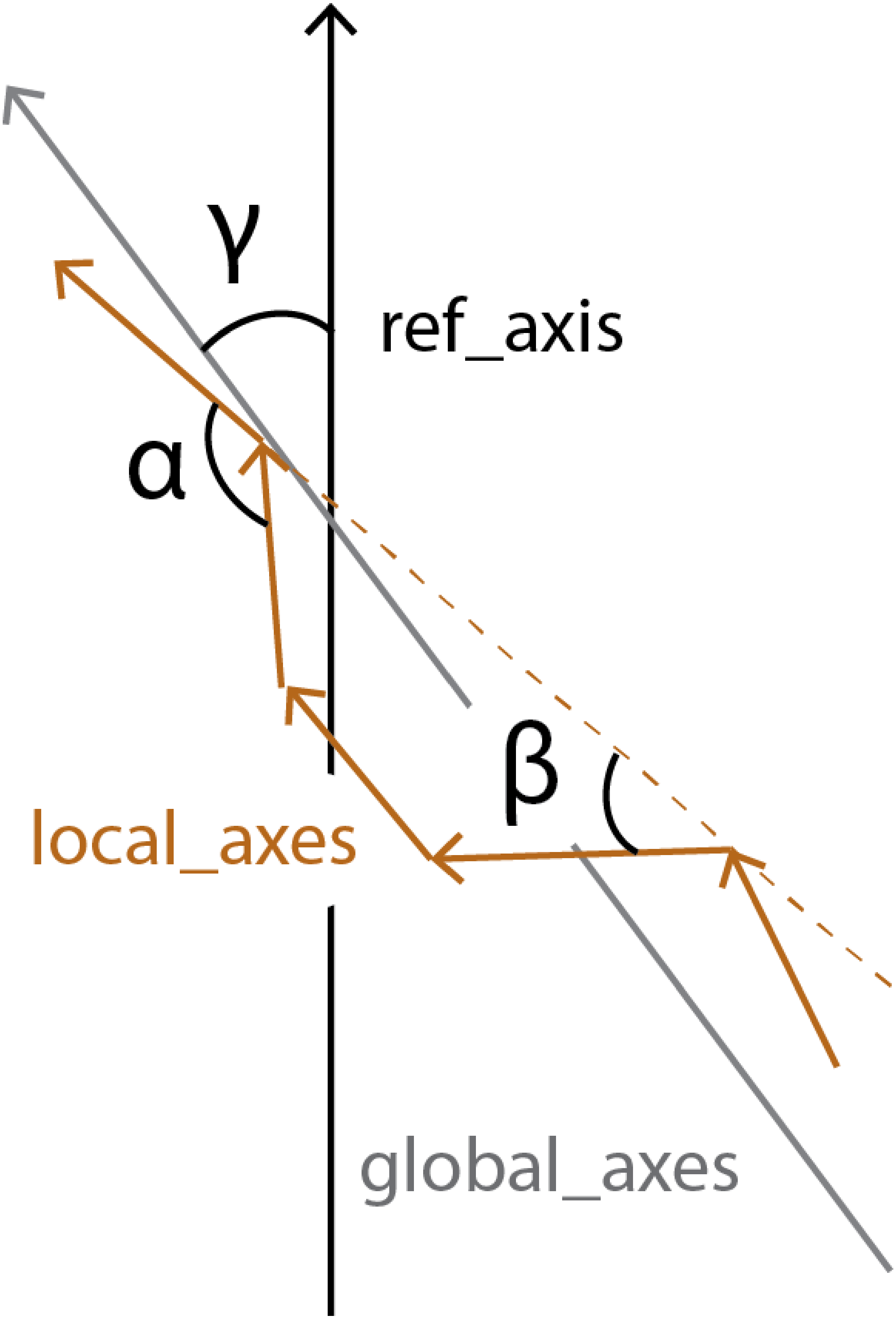
`HELANAL` computes a number of properties.



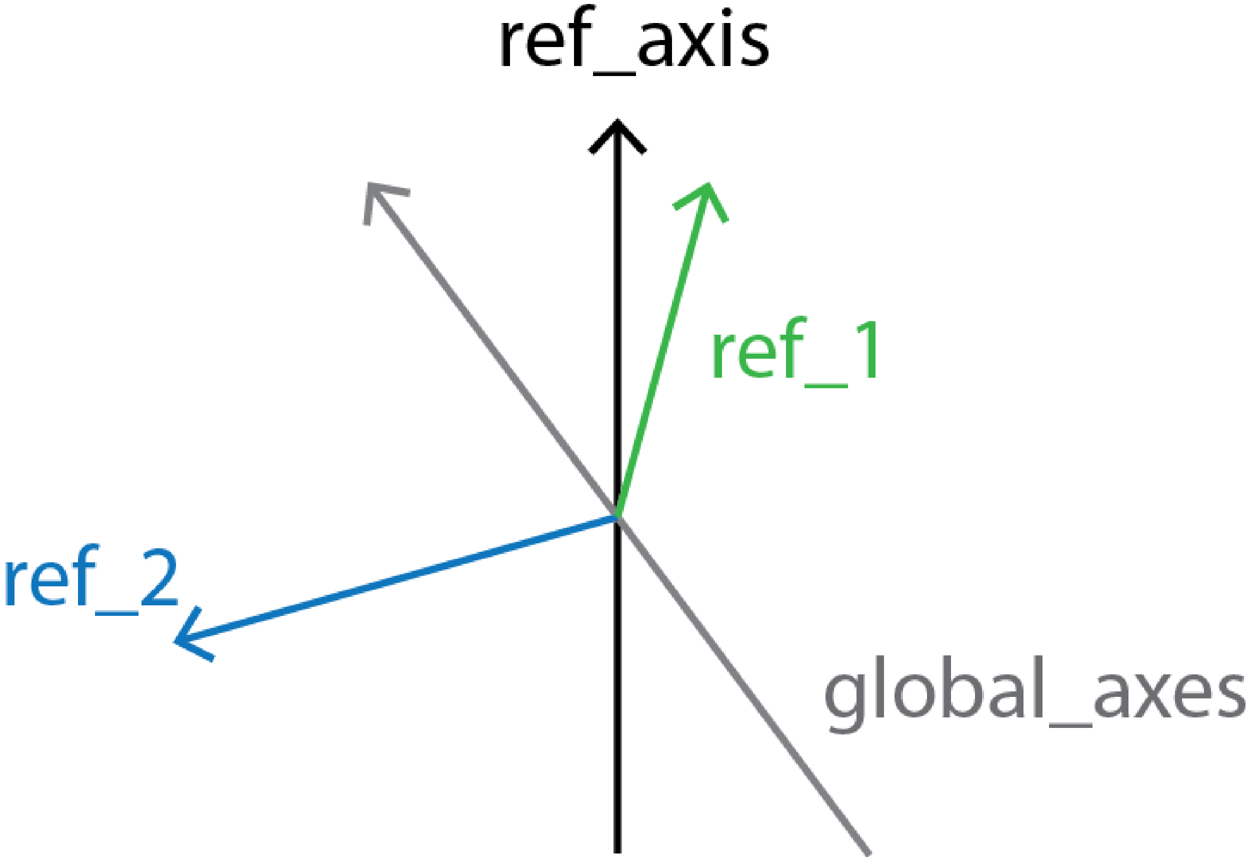
For each sliding window, it calculates:

- `local_rotation_vectors`: the vectors bisecting the angles of the middle 2 atoms
- `local_origins`: the projected origins of the helix
- `local_twists`: the twist of each window (θ)
- `residues_per_turn`: how many residues would fit in a turn, based on `local_twist`
- `local_axes`: the axis of each local helix
- `local_heights`: the rise of each helix

HELANAL calculates the bends between each `local_axes` and fits the vector `global_axes` to the `local_origins`.


γ
ref_axis
α
β
local_axes
global_axes

`all_bends` contains the angles between every `local_axes` (α) in a pairwise matrix, whereas `local_bends` contains the angles between `local_axes` that are calculated 3 windows apart (β). The `global_tilts` (γ) are calculated as the angle between the `global_axes` and the user-given reference `ref_axis`.



Finally, `local_screw` angles are computed between the `local_rotation_vectors` and the normal plane of the `global_axes`.

Running the analysis

As with most other analysis classes in MDAnalysis, pass in the universe and selection that you would to like to operate on. The default reference axis is the z-axis. You can also pass in a list of selection strings to run HELANAL on multiple helices at once.

```
[3]: h = hel.HELANAL(u, select='name CA and resnum 161-187',
                     ref_axis=[0, 0, 1]).run()
```

The properties described above are stored as attributes in `h.results`. For example, the `all_bends` matrix contains the bends in a `(n_frames, n_residues-3, n_residues-3)` array.

```
[4]: h.results.all_bends.shape
```

```
[4]: (98, 24, 24)
```

Each property is also summarised with a mean value, the sample standard deviation, and the average deviation from the mean.

```
[5]: h.results.summary.keys()
```

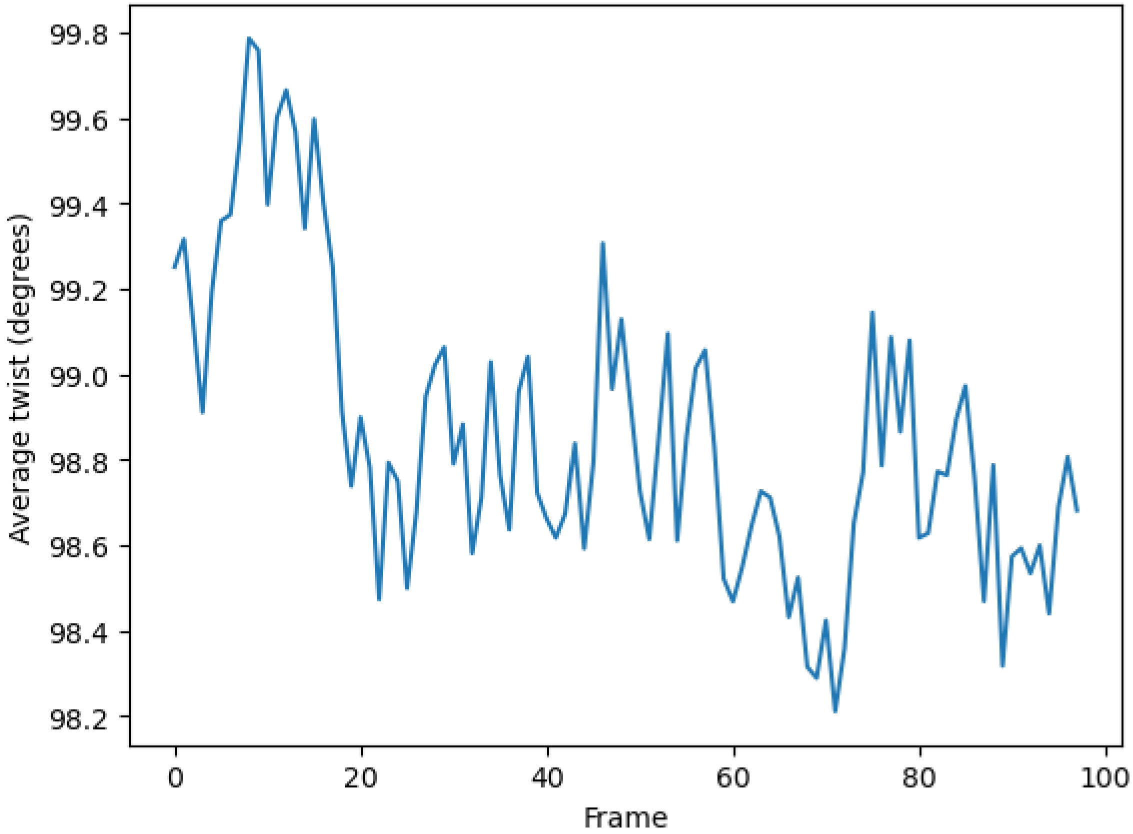
```
[5]: dict_keys(['local_twists', 'local_bends', 'local_heights', 'local_nres_per_turn', 'local_
↪origins', 'local_axes', 'local_helix_directions', 'local_screw_angles', 'global_axis',
↪'global_tilts', 'all_bends'])
```

```
[6]: for key, val in h.results.summary['global_tilts'].items():
         print(f"{key}: {val:.3f}")
```

```
mean: 86.121
sample_sd: 2.011
abs_dev: 1.715
```

As the data is stored as arrays, it can easily be plotted.

```
[7]: plt.plot(h.results.local_twists.mean(axis=1))
     plt.xlabel('Frame')
     plt.ylabel('Average twist (degrees)')
     plt.show()
```



You can also create a Universe from the `local_origins` if you would like to save it as a file and visualise it in programs such as VMD.

```
[8]: origins = h.universe_from_origins()
```

```
[9]: # view = nv.show_mdanalysis(h.atomgroups[0])
     # view.add_trajectory(origins)
     # view
```

Below we use NGLView to create a representative GIF.

```
[10]: # from nglview.contrib.movie import MovieMaker
      # movie = MovieMaker(
      #     view,
      #     step=4,  # keep every 4th step
      #     render_params={"factor": 3},  # controls quality
      #     output='helanal_images/helanal-view.gif',
      # )
      # movie.make()
```

References

[1] M. Bansal, S. Kumar, and R. Velavan. HELANAL: a program to characterize helix geometry in proteins. Journal of Biomolecular Structure & Dynamics, 17(5):811–819, April 2000. 00175. doi:10.1080/07391102.2000.10506570.

[2] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[3] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[4] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Dimension reduction

A molecular dynamics trajectory with N atoms can be considered a path through $3N$-dimensional molecular configuration space. It remains difficult to extract important dynamics or compare trajectory similarity from such a high-dimensional space. However, collective motions and physically relevant states can often be effectively described with low-dimensional representations of the conformational space explored over the trajectory. MDAnalysis implements two methods for dimensionality reduction.

Principal component analysis is a common linear dimensionality reduction technique that maps the coordinates in each frame of your trajectory to a linear combination of orthogonal vectors. The vectors are called *principal components*, and they are ordered such that the first principal component accounts for the most variance in the original data (i.e. the largest uncorrelated motion in your trajectory), and each successive component accounts for less and less variance. Trajectory coordinates can be transformed onto a lower-dimensional space (*essential subspace*) constructed from these principal components in order to compare conformations. Your trajectory can also be projected onto each principal component in order to visualise the motion described by that component.

Diffusion maps are a non-linear dimensionality reduction technique that embeds the coordinates of each frame onto a lower-dimensional space, such that the distance between each frame in the lower-dimensional space represents their "diffusion distance", or similarity. It integrates local information about the similarity of each point to its neighbours, into a global geometry of the intrinsic manifold. This means that this technique is not suitable for trajectories where the transitions between conformational states is not well-sampled (e.g. replica exchange simulations), as the regions may become disconnected and a meaningful global geometry cannot be approximated. Unlike PCA, there is no explicit mapping between the components of the lower-dimensional space and the original atomic coordinates; no physical interpretation of the eigenvectors is immediately available.

For computing similarity, see the tutorials in *Trajectory similarity*.

Principal component analysis of a trajectory

Here we compute the principal component analysis of a trajectory.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- nglview

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

[1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis import pca, align
# import nglview as nv

import warnings
# suppress some MDAnalysis warnings about writing PDB files
warnings.filterwarnings('ignore')
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]) The trajectory DCD samples a transition from a closed to an open conformation.

[2]:

```
u = mda.Universe(PSF, DCD)
```

Principal component analysis

Principal component analysis is a common linear dimensionality reduction technique that maps the coordinates in each frame of your trajectory to a linear combination of orthogonal vectors. The vectors are called **principal components**, and they are ordered such that the first principal component accounts for the most variance in the original data (i.e. the largest uncorrelated motion in your trajectory), and each successive component accounts for less and less variance. The frame-by-frame conformational fluctuation can be considered a linear combination of the essential dynamics yielded by the PCA. Please see [ALB93], [Jol02], [SJS14], or [SS18] for a more in-depth introduction to PCA.

Trajectory coordinates can be transformed onto a lower-dimensional space (*essential subspace*) constructed from these principal components in order to compare conformations. You can thereby visualise the motion described by that component.

In MDAnalysis, the method implemented in the PCA class (API docs) is as follows:

1. Optionally align each frame in your trajectory to the first frame.
2. Construct a 3N x 3N covariance for the N atoms in your trajectory. Optionally, you can provide a mean; otherwise the covariance is to the averaged structure over the trajectory.
3. Diagonalise the covariance matrix. The eigenvectors are the principal components, and their eigenvalues are the associated variance.
4. Sort the eigenvalues so that the principal components are ordered by variance.

Note

Principal component analysis algorithms are deterministic, but the solutions are not unique. For example, you could easily change the sign of an eigenvector without altering the PCA. Different algorithms are likely to produce different answers, due to variations in implementation. `MDAnalysis` may not return the same values as another package.

[3]:

```
aligner = align.AlignTraj(u, u, select='backbone',
                          in_memory=True).run()
```

You can choose how many principal components to save from the analysis with `n_components`. The default value is `None`, which saves all of them. You can also pass a `mean` reference structure to be used in calculating the covariance matrix. With the default value of `None`, the covariance uses the mean coordinates of the trajectory.

[4]:

```
pc = pca.PCA(u, select='backbone',
             align=True, mean=None,
             n_components=None).run()
```

The principal components are saved in `pc.p_components`. If you kept all the components, you should have an array of shape $(n_{atoms} \times 3, n_{atoms} \times 3)$.

[5]:

```
backbone = u.select_atoms('backbone')
n_bb = len(backbone)
print('There are {} backbone atoms in the analysis'.format(n_bb))
print(pc.p_components.shape)
```

```
There are 855 backbone atoms in the analysis
(2565, 2565)
```

The variance of each principal component is in `pc.variance`. For example, to get the variance explained by the first principal component to 5 decimal places:

```
[6]: print(f"PC1: {pc.variance[0]:.5f}")
```

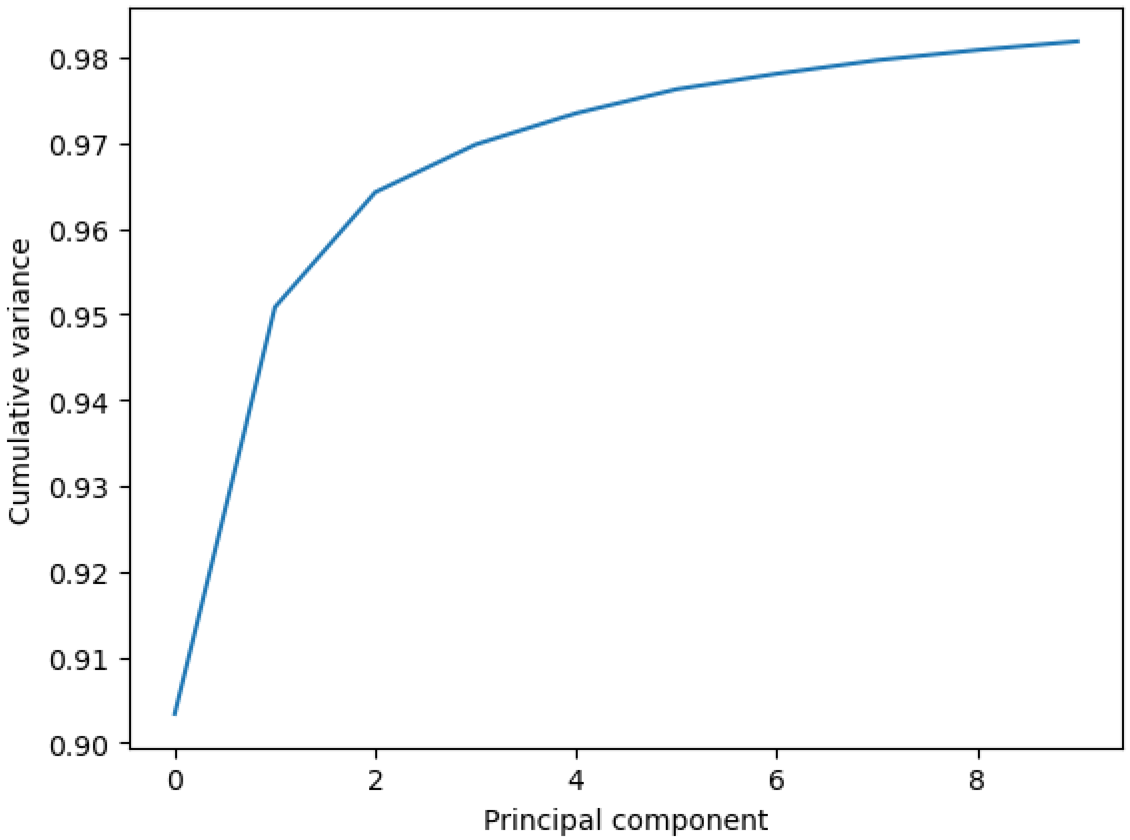
```
PC1: 4203.19053
```

This variance is somewhat meaningless by itself. It is much more intuitive to consider the variance of a principal component as a percentage of the total variance in the data. MDAnalysis also tracks the percentage cumulative variance in `pc.cumulated_variance`. As shown below, the first principal component contains 90.3% the total trajectory variance. The first three components combined account for 96.4% of the total variance.

```
[7]: for i in range(3):
         print(f"Cumulated variance: {pc.cumulated_variance[i]:.3f}")
```

```
Cumulated variance: 0.903
Cumulated variance: 0.951
Cumulated variance: 0.964
```

```
[8]: plt.plot(pc.cumulated_variance[:10])
     plt.xlabel('Principal component')
     plt.ylabel('Cumulative variance')
     plt.show()
```



Visualising projections into a reduced dimensional space

The `pc.transform()` method transforms a given atom group into weights $\mathbf{w}_i$ over each principal component i.

$$\mathbf{w}_i(t) = (\mathbf{r}(t) - \overline{\mathbf{r}}) \cdot \mathbf{u}_i$$

$\mathbf{r}(t)$ are the atom group coordinates at time t, $\overline{\mathbf{r}}$ are the mean coordinates used in the PCA, and $\mathbf{u}_i$ is the ith principal component eigenvector $\mathbf{u}$.

While the given atom group must have the same number of atoms that the principal components were calculated over, it does not have to be the same group.

Again, passing `n_components=None` will tranform your atom group over every component. Below, we limit the output to projections over 3 principal components only.

[9]:
```
transformed = pc.transform(backbone, n_components=3)
transformed.shape
```

[9]:
```
(98, 3)
```

The output has the shape (n_frames, n_components). For easier analysis and plotting we can turn the array into a DataFrame.

[10]:
```
df = pd.DataFrame(transformed,
                  columns=['PC{}'.format(i+1) for i in range(3)])
df['Time (ps)'] = df.index * u.trajectory.dt
df.head()
```

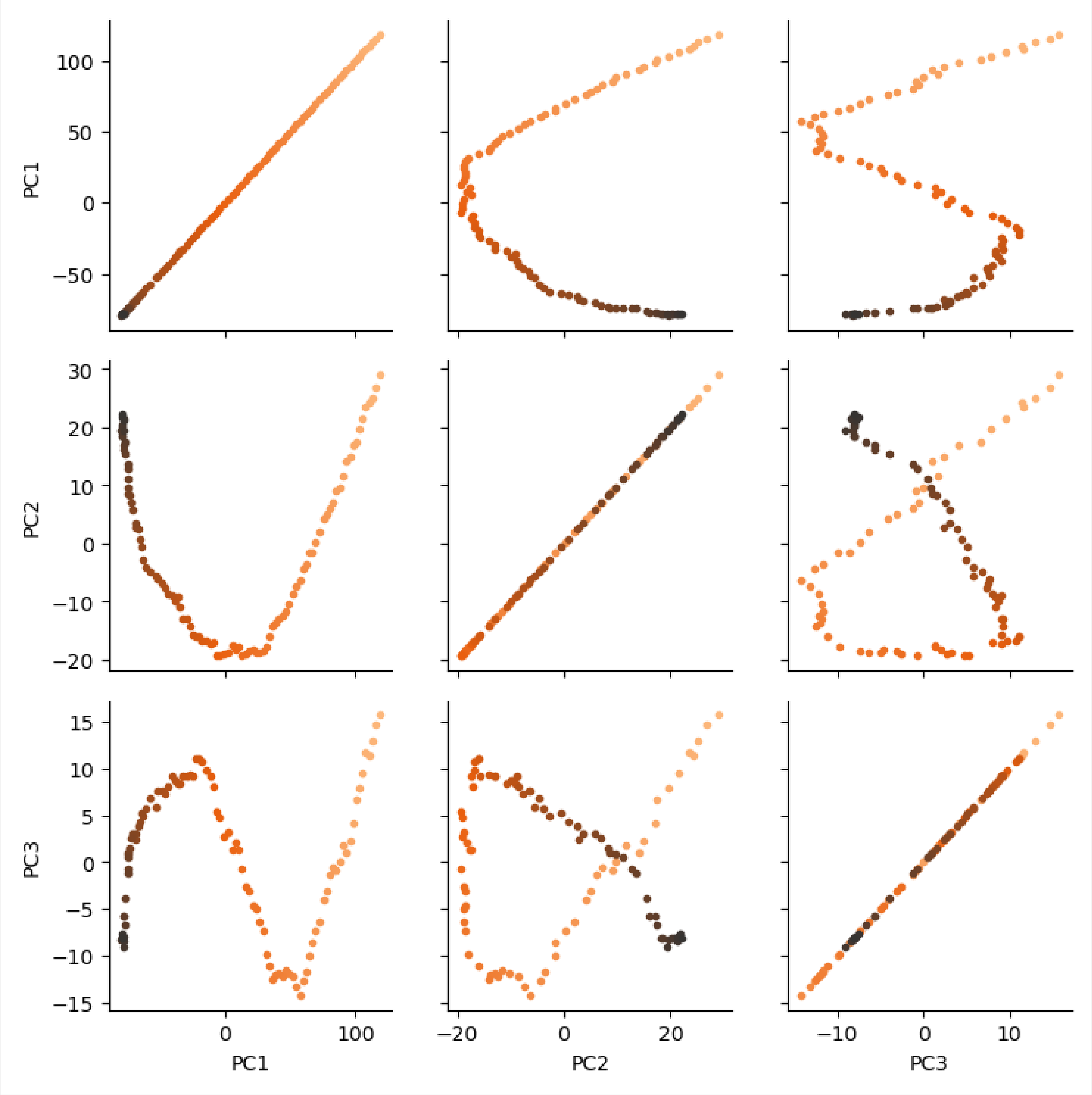
[10]:
```
          PC1        PC2        PC3  Time (ps)
0  118.408413  29.088241  15.746624        0.0
1  115.561879  26.786797  14.652498        1.0
2  112.675616  25.038766  12.920274        2.0
3  110.341467  24.306984  11.427098        3.0
4  107.584302  23.464154  11.612104        4.0
```

There are several ways we can visualise the data. Using the Seaborn's `PairGrid` tool is the quickest and easiest way, if you have seaborn already installed.

Note

You will need to install the data visualisation library Seaborn for this function.

[11]:
```
import seaborn as sns

g = sns.PairGrid(df, hue='Time (ps)',
                 palette=sns.color_palette('Oranges_d',
                                           n_colors=len(df)))
g.map(plt.scatter, marker='.')
plt.show()
```



Another way to investigate the essential motions of the trajectory is to project the original trajectory onto each of the principal components, to visualise the motion of the principal component. The outer product $\otimes$ of the weights $\mathbf{w}_i(t)$ for principal component i with the eigenvector $\mathbf{u}_i$ describes fluctuations around the mean on that axis, so the projected trajectory $\mathbf{r}_i(t)$ is simply the fluctuations added onto the mean positions $\bar{\mathbf{r}}$.

$$\mathbf{r}_i(t) = \mathbf{w}_i(t) \otimes \mathbf{u}_i + \bar{\mathbf{r}}$$

Below, we generate the projected coordinates of the first principal component. The mean positions are stored at `pc.mean`.

[12]:
```
pc1 = pc.p_components[:, 0]
trans1 = transformed[:, 0]
projected = np.outer(trans1, pc1) + pc.mean.flatten()
```

(continues on next page)

(continued from previous page)

```
coordinates = projected.reshape(len(trans1), -1, 3)
```

We can create a new universe from this to visualise the movement over the first principal component.

[13]:
```
proj1 = mda.Merge(backbone)
proj1.load_new(coordinates, order="fac")
```

[13]:
```
<Universe with 855 atoms>
```

[14]:
```
# view = nv.show_mdanalysis(proj1.atoms)
# view
```

If you have `nglview` installed, you can view the trajectory in the notebook. Otherwise, you can write the trajectory out to a file and use another program such as VMD. Below, we create a movie of the component.

[15]:
```
# from nglview.contrib.movie import MovieMaker
# movie = MovieMaker(
#     view,
#     step=4,  # keep every 4th frame
#     output='pc1.gif',
#     render_params={"factor": 3},  # set to 4 for highest quality
# )
# movie.make()
```

Measuring convergence with cosine content

The essential modes of a trajectory usually describe global, collective motions. The cosine content of a principal component can be interpreted to determine whether proteins are transitioning between conformational states. However, random diffusion can also appear to produce collective motion. The cosine content can measure the convergence of a trajectory and indicate poor sampling.

The cosine content of a principal component measures how similar it is to a cosine shape. Values range from 0 (no similarity to a cosine) and 1 (a perfect cosine shape). If the values of the first few principal components are close to 1, this can indicate poor sampling, as the motion of the particles may not be distinguished from random diffusion. Values below 0.7 do not indicate poor sampling.

For more information, please see [MLS09].

Note

[Hes02] first published the usage of cosine content to evaluate sampling. Please cite this paper when using the `MDAnalysis.analysis.pca.cosine_content` method in published work.

Below we calculate the cosine content of the first five principal components in the transformed subspace. Note that this is an example only, to dmonstrate how to use the method; the first few principal components of short simulations always represent random diffusion ([Hes02]).

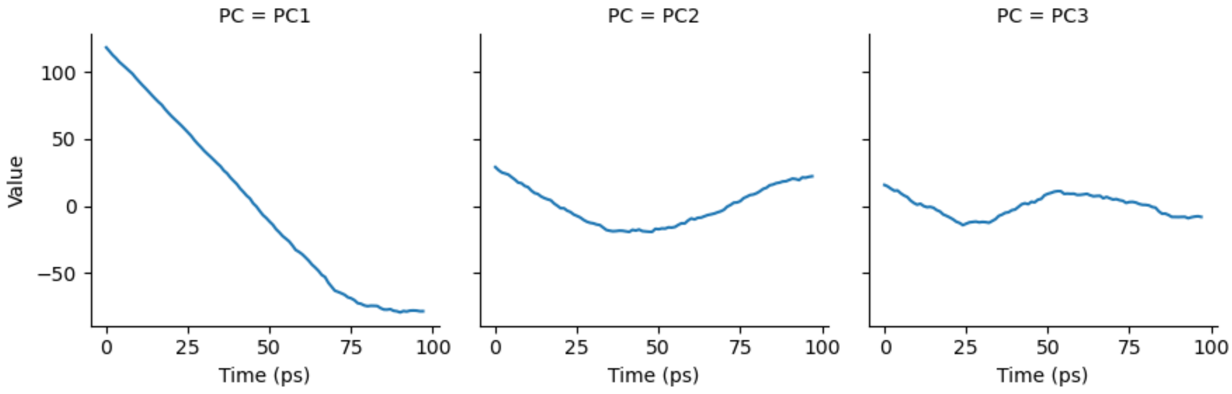
[16]:
```
for i in range(3):
    cc = pca.cosine_content(transformed, i)
    print(f"Cosine content for PC {i+1} = {cc:.3f}")
```

```
Cosine content for PC 1 = 0.960
Cosine content for PC 2 = 0.906
Cosine content for PC 3 = 0.723
```

As can be seen, the cosine content of each component is quite high. If we plot the transformed components over time, we can see that each component does resemble a cosine curve.

[17]:

```
# melt the dataframe into a tidy format
melted = pd.melt(df, id_vars=["Time (ps)"],
                 var_name="PC",
                 value_name="Value")
g = sns.FacetGrid(melted, col="PC")
g.map(sns.lineplot,
      "Time (ps)", # x-axis
      "Value", # y-axis
      ci=None) # no confidence interval
plt.show()
```



References

[1] Andrea Amadei, Antonius B. M. Linssen, and Herman J. C. Berendsen. Essential dynamics of proteins. Proteins: Structure, Function, and Bioinformatics, 17(4):412–425, 1993. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/prot.340170408. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/prot.340170408, doi:https://doi.org/10.1002/prot.340170408.

[2] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[3] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[4] I. T. Jolliffe. Principal Component Analysis. Springer Series in Statistics. Springer-Verlag, New York, 2 edition, 2002. ISBN 978-0-387-95442-4. URL: http://www.springer.com/gp/book/9780387954424, doi:10.1007/b98835.

[5] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July

2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[6] Florian Sittel, Abhinav Jain, and Gerhard Stock. Principal component analysis of molecular dynamics: on the use of Cartesian vs. internal coordinates. The Journal of Chemical Physics, 141(1):014111, July 2014. doi:10.1063/1.4885338.

[7] Florian Sittel and Gerhard Stock. Perspective: Identification of collective variables and metastable states of protein dynamics. The Journal of Chemical Physics, 149(15):150901, October 2018. Publisher: American Institute of Physics. URL: http://aip.scitation.org/doi/10.1063/1.5049637, doi:10.1063/1.5049637.

Non-linear dimension reduction to diffusion maps

Here we reduce the dimensions of a trajectory into a diffusion map.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.17.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Note

Please cite [CL06] if you use the `MDAnalysis.analysis.diffusionmap.DiffusionMap` in published work.

[1]:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis import diffusionmap
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme ([BDPW09]). The trajectory DCD samples a transition from a closed to an open conformation.

[2]:

```
u = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/mda-user-guide/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/DCD.py:165: DeprecationWarning: DCDReader currently makes independent␣
↪timesteps by copying self.ts while other readers update self.ts inplace. This behavior␣
↪will be changed in 3.0 to be the same as other readers. Read more at https://github.
↪com/MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect␣
↪you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Diffusion maps

Diffusion maps are a non-linear dimensionality reduction technique that embeds the coordinates of each frame onto a lower-dimensional space, such that the distance between each frame in the lower-dimensional space represents their "diffusion distance", or similarity. It integrates local information about the similarity of each point to its neighbours, into a global geometry of the intrinsic manifold. This means that this technique is not suitable for trajectories where the transitions between conformational states are not well-sampled (e.g. replica exchange simulations), as the regions may become disconnected and a meaningful global geometry cannot be approximated. Unlike *principal component analysis*, there is no explicit mapping between the components of the lower-dimensional space and the original atomic coordinates; no physical interpretation of the eigenvectors is immediately available. Please see [CL06], [dlPHHvdW08], [RZMC11] and [FPKD11] for more information.

The default distance metric implemented in MDAnalysis' DiffusionMap class is RMSD.

Note

MDAnalysis implements RMSD calculation using the fast QCP algorithm ([The05]). Please cite [The05] if you use the default distance metric in published work.

```
[3]: dmap = diffusionmap.DiffusionMap(u, select='backbone', epsilon=2)
     dmap.run()
```

```
[3]: <MDAnalysis.analysis.diffusionmap.DiffusionMap at 0x7f75f1291520>
```

The first eigenvector in a diffusion map is always essentially all ones (when divided by a constant):

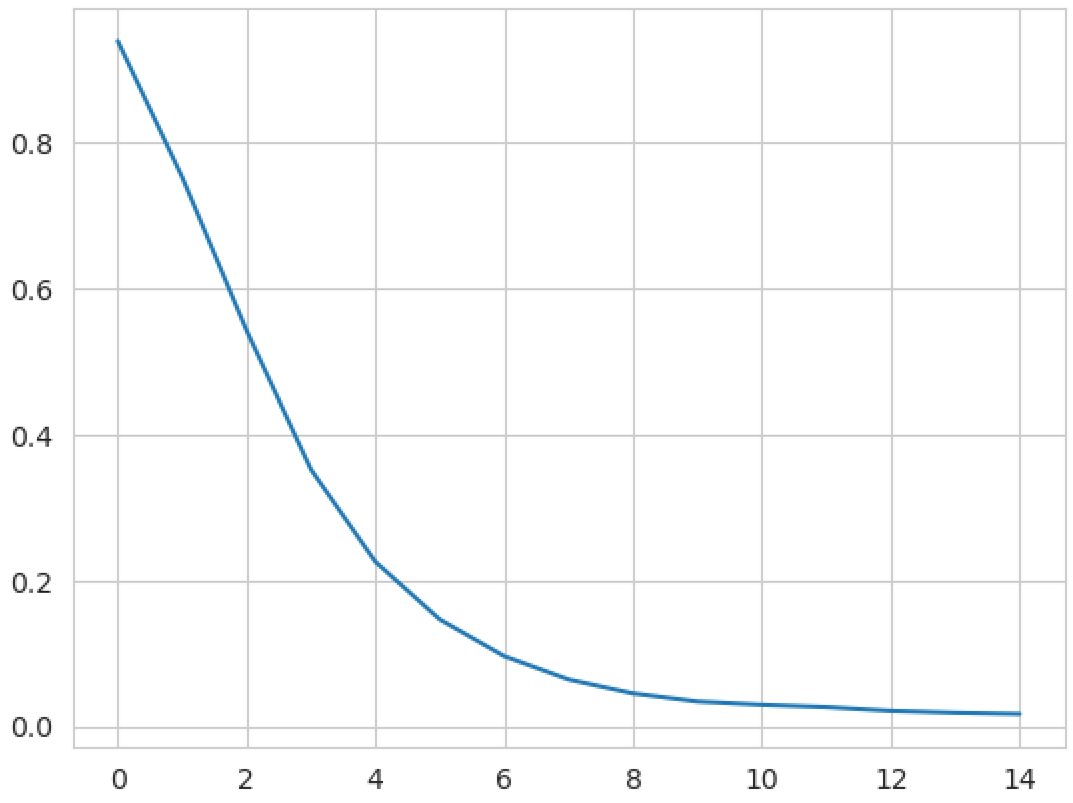
```
[4]: dmap._eigenvectors[:, 0]
```

```
[4]: array([0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525, 0.10101525, 0.10101525,
            0.10101525, 0.10101525, 0.10101525])
```

Therefore, when we embed the trajectory onto the dominant eigenvectors, we ignore the first eigenvector. In order to determine which vectors are dominant, we can examine the eigenvalues for a **spectral gap**: where the eigenvalues stop decreasing constantly in value.

```
[5]: fig, ax = plt.subplots()
     ax.plot(dmap.eigenvalues[1:16])
```

```
[5]: [<matplotlib.lines.Line2D at 0x7f75f1291970>]
```



From this plot, we take the first *k* dominant eigenvectors to be the first five. Below, we transform the trajectory onto these eigenvectors. The `time` argument is the exponent that the eigenvalues are raised to for embedding. As values increase for `time`, more dominant eigenvectors (with lower eigenvalues) dominate the diffusion distance more. The `transform` method returns an array of shape (# frames, # eigenvectors).

```
[6]: transformed = dmap.transform(5,   # number of eigenvectors
                                 time=1)
     transformed.shape
```

```
[6]: (98, 5)
```

For easier analysis and plotting we can turn the array into a DataFrame.

```
[7]: df = pd.DataFrame(transformed,
                       columns=['Mode{}'.format(i+2) for i in range(5)])
     df['Time (ps)'] = df.index * u.trajectory.dt
     df.head()
```

```
[7]:       Mode2     Mode3     Mode4     Mode5     Mode6  Time (ps)
     0  0.094795  0.075950  0.054708  0.035526  0.022757        0.0
     1  0.166068  0.132017  0.094409  0.060914  0.038667        1.0
     2  0.199960  0.154475  0.107425  0.067632  0.041445        2.0
     3  0.228815  0.168694  0.111460  0.067112  0.038469        3.0
```

(continues on next page)

(continued from previous page)

```
4  0.250384  0.171873  0.103407  0.057143  0.028398        4.0
```

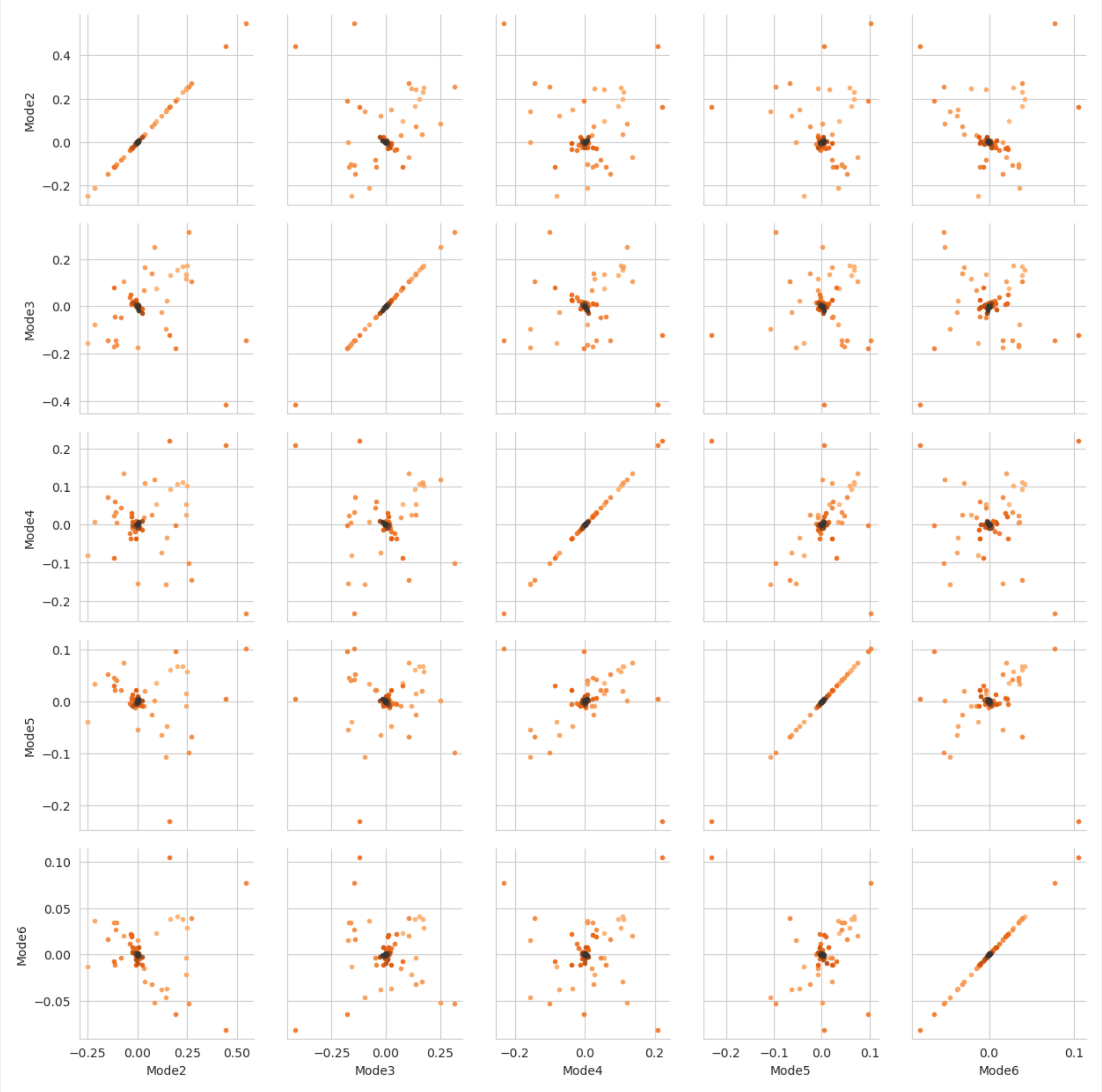
There are several ways we can visualise the data. Using the Seaborn's `PairGrid` tool is the quickest and easiest way, if you have seaborn already installed. Each of the subplots below illustrates axes of the lower-dimensional embedding of the higher-dimensional data, such that dots (frames) that are close are kinetically close (connected by a large number of short pathways), whereas greater distance indicates states that are connected by a smaller number of long pathways. Please see [FPKD11] for more information.

Note

You will need to install the data visualisation library Seaborn for this function.

[8]:
```
import seaborn as sns

g = sns.PairGrid(df, hue='Time (ps)',
                 palette=sns.color_palette('Oranges_d',
                                           n_colors=len(df)))
g.map(plt.scatter, marker='.')
```

[8]:
```
<seaborn.axisgrid.PairGrid at 0x7f75f0149520>
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Ronald R. Coifman and Stéphane Lafon. Diffusion maps. Applied and Computational Harmonic Analysis, 21(1):5–30, July 2006. 02271. doi:10.1016/j.acha.2006.04.006.

[3] J. de la Porte, B. M. Herbst, W. Hereman, and S. J. van der Walt. An introduction to diffusion maps. In In The 19th Symposium of the Pattern Recognition Association of South Africa. 2008. 00038.

[4] Andrew Ferguson, Athanassios Z. Panagiotopoulos, Ioannis G. Kevrekidis, and Pablo G. Debenedetti. Nonlinear

dimensionality reduction in molecular simulation: The diffusion map approach. Chemical Physics Letters, 509(1-3):1–11, June 2011. 00085. doi:10.1016/j.cplett.2011.04.066.

[5] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[6] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[7] Mary A. Rohrdanz, Wenwei Zheng, Mauro Maggioni, and Cecilia Clementi. Determination of reaction coordinates via locally scaled diffusion map. The Journal of Chemical Physics, 134(12):124116, March 2011. 00220. doi:10.1063/1.3569857.

[8] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallographica Section A Foundations of Crystallography, 61(4):478–480, July 2005. 00127. URL: http://scripts.iucr.org/cgi-bin/paper?S0108767305015266, doi:10.1107/S0108767305015266.

Polymers and membranes

MDAnalysis has several analyses specifically for polymers, membranes, and membrane proteins.

Determining the persistence length of a polymer

Here we determine the persistence length of a polymer.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import TRZ_psf, TRZ
from MDAnalysis.analysis import polymer
%matplotlib inline
```

Loading files

The test files we will be working with here feature a pure polymer melt of a polyamide.

[2]:
```
u = mda.Universe(TRZ_psf, TRZ)
```

Choosing the chains and backbone atoms

We can define the chains of polyamide to be the common definition of a molecule: where each atom is bonded to at least one other in the group, and not bonded to any atom outside the group. MDAnalysis provides these as `fragments`.

[3]:
```
chains = u.atoms.fragments
```

We then want to select only the backbone atoms for each chain, i.e. only the carbons and nitrogens.

[4]:
```
backbones = [ch.select_atoms('not name O* H*') for ch in chains]
```

This should give us AtomGroups where the spatial arrangement is linear. However, the atoms are in index order. We can use `sort_backbone` to arrange our atom groups into their linear arrangement order.

[5]:
```
sorted_bb = [polymer.sort_backbone(bb) for bb in backbones]
```

Calculating the persistence length

The persistence length is the length at which two points on the polymer chain become decorrelated. This is determined by first measuring the autocorrelation $C(n)$ of two bond vectors $(\mathbf{a}_i, \mathbf{a}_{i+n})$ separated by n bonds, where

$$C(n) = \langle \cos \theta_{i,i+n} \rangle = \langle \mathbf{a_i} \cdot \mathbf{a_{i+n}} \rangle$$

An exponential decay is then fitted to this, which yields the persistence length l_P from the average bond length $\bar{l_B}$.

$$C(n) \approx \exp\left(-\frac{n\bar{l_B}}{l_P}\right)$$

We set up our `PersistenceLength` class (API docs). Note that every chain we pass into it must have the same length.

[6]:
```
plen = polymer.PersistenceLength(sorted_bb)
plen.run()
```

[6]:
```
<MDAnalysis.analysis.polymer.PersistenceLength at 0x7f4b7ddfd160>
```

The average bond length is found at `plen.results.lb`, the calculated persistence length at `plen.results.lp`, the measured autocorrelation at `plen.results` and the modelled decorrelation fit at `plen.results.fit`.

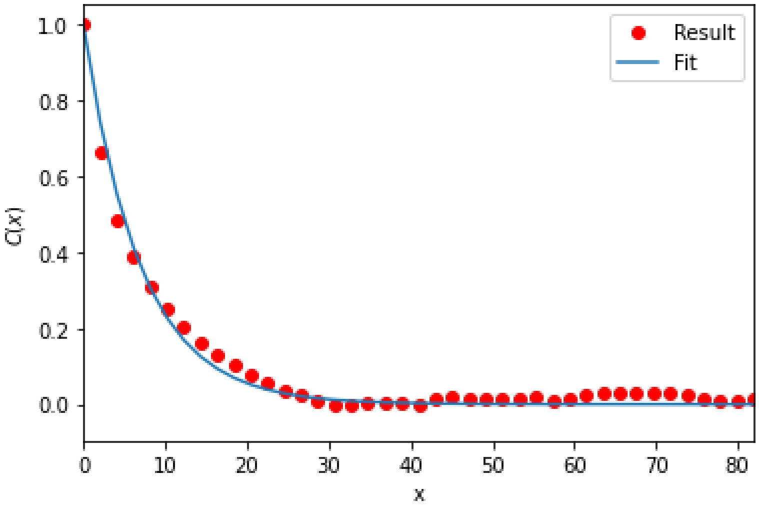
[20]:
```
print(plen.results.fit.shape)
print('The persistence length is {:.3f}'.format(plen.results.lp))
```

```
(179,)
The persistence length is 6.917
```

`MDAnalysis.analysis.polymer.PersistenceLength` provides a convenience method to plot the results.

[21]:
```
plen.plot()
```

[21]:
```
<AxesSubplot:xlabel='x', ylabel='$C(x)$'>
```



References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Analysing pore dimensions with HOLE2

Here we use `HOLE` to analyse pore dimensions in a membrane.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- HOLE
- matplotlib
- numpy

Note

The classes in `MDAnalysis.analysis.hole2` are wrappers around the HOLE program. Please cite ([SGW93], [SNW+96]) when using this module in published work.

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PDB_HOLE
from MDAnalysis.analysis import hole2
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

import warnings
# suppress some MDAnalysis warnings when writing PDB files
warnings.filterwarnings('ignore')
```

Background

The `MDAnalysis.analysis.hole2` module (API docs) provides wrapper classes and functions that call the HOLE program. This means you must installed the program yourself before you can use the class. You then have 2 options, you either pass the path to your `hole` executable to the class; in this example, `hole` is installed at `~/hole2/exe/hole`. Or, set your binary path variable (`$PATH` in Unix systems) to point to the executable's folder so you don't have to point to the binary explicitly every time you call `hole` or any of its helper tools. This is what we have done here, so we don't have to set the `executable` argument.

HOLE defines a series of points throughout the pore from which a sphere can be generated that does not overlap any atom (as defined by its van der Waals radius). (Please see ([SGW93], [SNW+96]) for a complete explanation). By default, it ignores residues with the following names: "SOL", "WAT", "TIP", "HOH", "K ", "NA ", "CL ". You can change these with the `ignore_residues` keyword. Note that the residue names must have 3 characters. Wildcards *do not* work.

This tutorial first demonstrates how to use the `MDAnalysis.analysis.hole2.hole` function similarly to *the HOLE binary on a PDB file*. We then demonstrate how to use the `MDAnalysis.analysis.hole2.HoleAnalysis` class on a *trajectory of data*. You may prefer to use the more fully-featured `HoleAnalysis` class for the extra functionality we provide, such as creating an animation in VMD of the pore.

Using HOLE with a PDB file

The `hole` function allows you to specify points to begin searching at (`cpoint`) and a search direction (`cvect`), the sampling resolution (`sample`), and more. Please see the documentation for full details.

The PDB file here is the experimental structure of the Gramicidin A channel. Note that we pass `HOLE` a PDB file directly, without creating a `MDAnalysis.Universe`.

We are setting a `random_seed` here so that the results in the tutorial can be reproducible. This is normally not advised.

[2]:
```
profiles = hole2.hole(PDB_HOLE,
                      outfile='hole1.out',
                      sphpdb_file='hole1.sph',
                      vdwradii_file=None,
                      random_seed=31415,
                      #   executable='~/hole2/exe/hole',
                      )
```

`outfile` and `sphpdb_file` are the names of the files that HOLE will write out. `vdwradii_file` is a file of necessary van der Waals' radii in a HOLE-readable format. If set to `None`, MDAnalysis will create a `simple2.rad` file with the built-in radii from the HOLE distribution.

This will create several outputs in your directory:

- **hole1.out**: the log file for HOLE.
- **hole1.sph**: a PDB-like file containing the coordinates of the pore centers.
- **simple2.rad**: file of Van der Waals' radii
- **tmp/pdb_name.pdb**: the short name of a PDB file with your structure. As `hole` is a FORTRAN77 program, it is limited in how long of a filename that it can read. Symlinking the file to the current directory can shorten the path.

If you do not want to keep the files, set `keep_files=False`. Keep in mind that you will not be able to create a VMD surface without the sphpdb file.

The pore profile itself is in the `profiles1` dictionary, indexed by frame. There is only one frame in this PDB file, so it is at `profiles1[0]`.

```
[3]: profiles[0].shape
```

```
[3]: (425,)
```

Each profile is a `numpy.recarray` with the fields below as an entry for each `rxncoord`:

- **rxn_coord**: the distance along the pore axis in angstrom
- **radius**: the pore radius in angstrom
- **cen_line_D**: distance measured along the pore centre line - the first point found is set to zero.

```
[4]: profiles[0].dtype.names
```

```
[4]: ('rxn_coord', 'radius', 'cen_line_D')
```

You can then proceed with your own analysis of the profiles.

```
[5]: rxn_coords = profiles[0].rxn_coord
     pore_length = rxn_coords[-1] - rxn_coords[0]
     print('The pore is {} angstroms long'.format(pore_length))
```

```
The pore is 42.4 angstroms long
```

You can create a VMD surface from the `hole1.sph` output file, using the `create_vmd_surface` function.

```
[6]: hole2.create_vmd_surface(filename='hole1.vmd',
                              sphpdb='hole1.sph',
                            #  sph_process='~/hole2/exe/sph_process',
                             )
```

```
[6]: 'hole1.vmd'
```

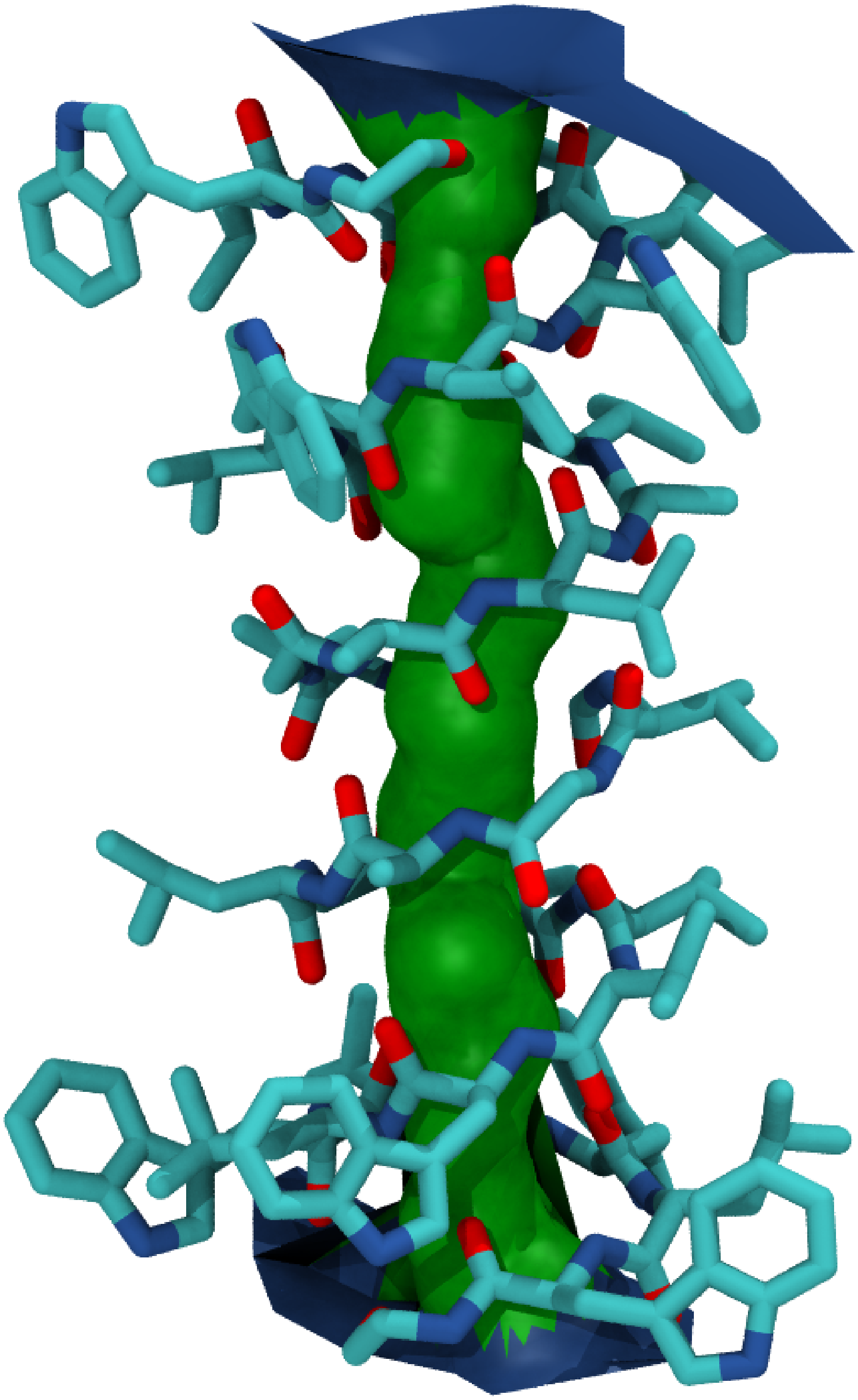
To view this, open your PDB file in VMD.

```
vmd tmp*/*.pdb
```

Load the output file in Extensions > Tk Console:

```
source hole1.vmd
```

Your pore surface will be drawn as below.



MDAnalysis supports many of the options that can be customised in HOLE. For example, you can specify a starting point for the pore search within the pore with `cpoint`, and a `sample` distance (default: 0.2 angstrom) for the distance between the planes used in HOLE. Please see the MDAnalysis.analysis.hole2 for more information.

Using HOLE with a trajectory

One of the limitations of the `hole` program is that it can only accept PDB files. In order to use other formats with `hole`, or to run `hole` on trajectories, we can use the `hole2.HoleAnalysis` class with an `MDAnalysis.Universe`. While the example file below is a PDB, you can use any files to create your Universe. You can also specify that the HOLE analysis is only run on a particular group of atoms with the `select` keyword (default value: 'protein').

As with `hole()`, `HoleAnalysis` allows you to select a starting point for the search (`cpoint`). You can pass in a coordinate array; alternatively, you can use the center-of-geometry of your atom selection in each frame as the start.

[7]:

```
from MDAnalysis.tests.datafiles import MULTIPDB_HOLE

u = mda.Universe(MULTIPDB_HOLE)

ha = hole2.HoleAnalysis(u, select='protein',
                        cpoint='center_of_geometry',
                        # executable='~/hole2/exe/hole',
                        )
ha.run(random_seed=31415)
```

[7]:

```
<MDAnalysis.analysis.hole2.hole.HoleAnalysis at 0x7f1a146af610>
```

Working with the data

Again, the data is stored in `ha.results.profiles` as a dictionary of `numpy.recarrays`. The dictionary is indexed by frame; we can see the HOLE profile for the fourth frame below (truncated to the first 10 values).

[8]:

```
for rxn_coord, radius, cen_line_D in (ha.results.profiles[3][:10]):
    print(f"{rxn_coord:.2f}, {radius:.2f}, {cen_line_D:.2f}")
```

```
-21.01, 15.35, -39.24
-20.91, 12.63, -34.65
-20.81, 10.64, -30.05
-20.71, 9.58, -27.73
-20.61, 8.87, -25.40
-20.51, 8.57, -23.62
-20.41, 8.56, -21.84
-20.31, 8.48, -21.74
-20.21, 8.39, -21.64
-20.11, 8.30, -21.54
```

If you want to collect each individual property, use `gather()`. Setting `flat=True` flattens the lists of `rxn_coord`, `radius`, and `cen_line_D`, in order. You can select which frames you want by passing an iterable of frame indices to `frames`. `frames=None` returns all frames.

[9]:

```
gathered = ha.gather()
print(gathered.keys())
```

```
dict_keys(['rxn_coord', 'radius', 'cen_line_D'])
```

[10]:

```
print(len(gathered['rxn_coord']))
```

```
11
```

[11]:

```
flat = ha.gather(flat=True)
print(len(flat['rxn_coord']))
```

```
3967
```

You may also want to collect the radii in bins of `rxn_coord` for the entire trajectory with the `bin_radii()` function. `range` should be a tuple of the lower and upper edges of the first and last bins, respectively. If `range=None`, the minimum and maximum values of `rxn_coord` are used.

`bins` can be either an iterable of (lower, upper) edges (in which case `range` is ignored), or a number specifying how many bins to create with `range`.

[12]:

```
radii, edges = ha.bin_radii(bins=100, range=None)
```

The closely related `histogram_radii()` function takes the same arguments as `bin_radii()` to group the pore radii, but allows you to specify an aggregating function with `aggregator` (default: `np.mean`) that will be applied to each array of radii. The arguments for this function, and returned values, are analogous to those for `np.histogram`.

[13]:

```
means, edges = ha.histogram_radii(bins=100, range=None,
                                  aggregator=np.mean)
```

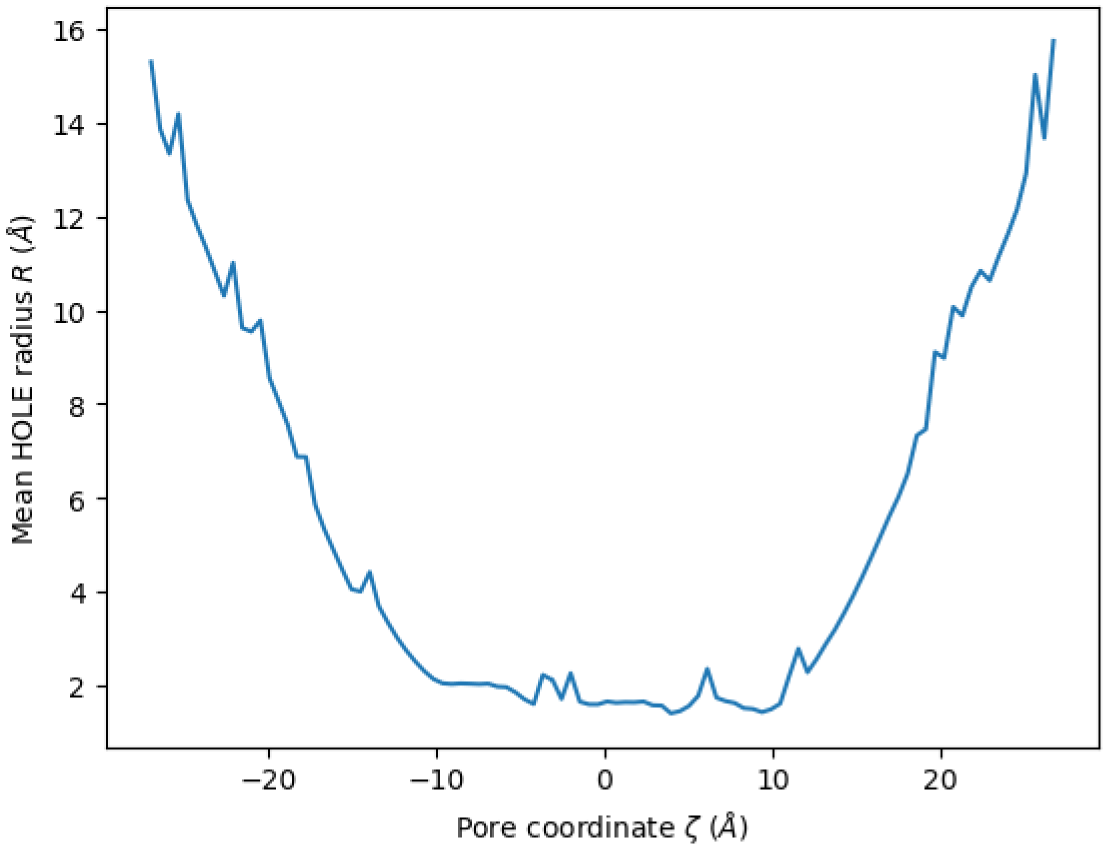
We can use this to plot the mean radii of the pore over the trajectory. (You can also accomplish this with the `plot_mean_profile()` function shown below, by setting `n_std=0`.)

[14]:

```
midpoints = 0.5*(edges[1:]+edges[:-1])
plt.plot(midpoints, means)
plt.ylabel(r"Mean HOLE radius $R$ ($\AA$)")
plt.xlabel(r"Pore coordinate $\zeta$ ($\AA$)")
```

[14]:

```
Text(0.5, 0, 'Pore coordinate $\\zeta$ ($\\AA$)')
```



`HoleAnalysis` also has the `min_radius()` function, which will return the minimum radius in angstrom for each frame. The resulting array has the shape (#n_frames, 2).

[15]:

```
min_radii = ha.min_radius()
for frame, min_radius in min_radii:
    print(f"Frame {int(frame)}: {min_radius:.3f}")
```

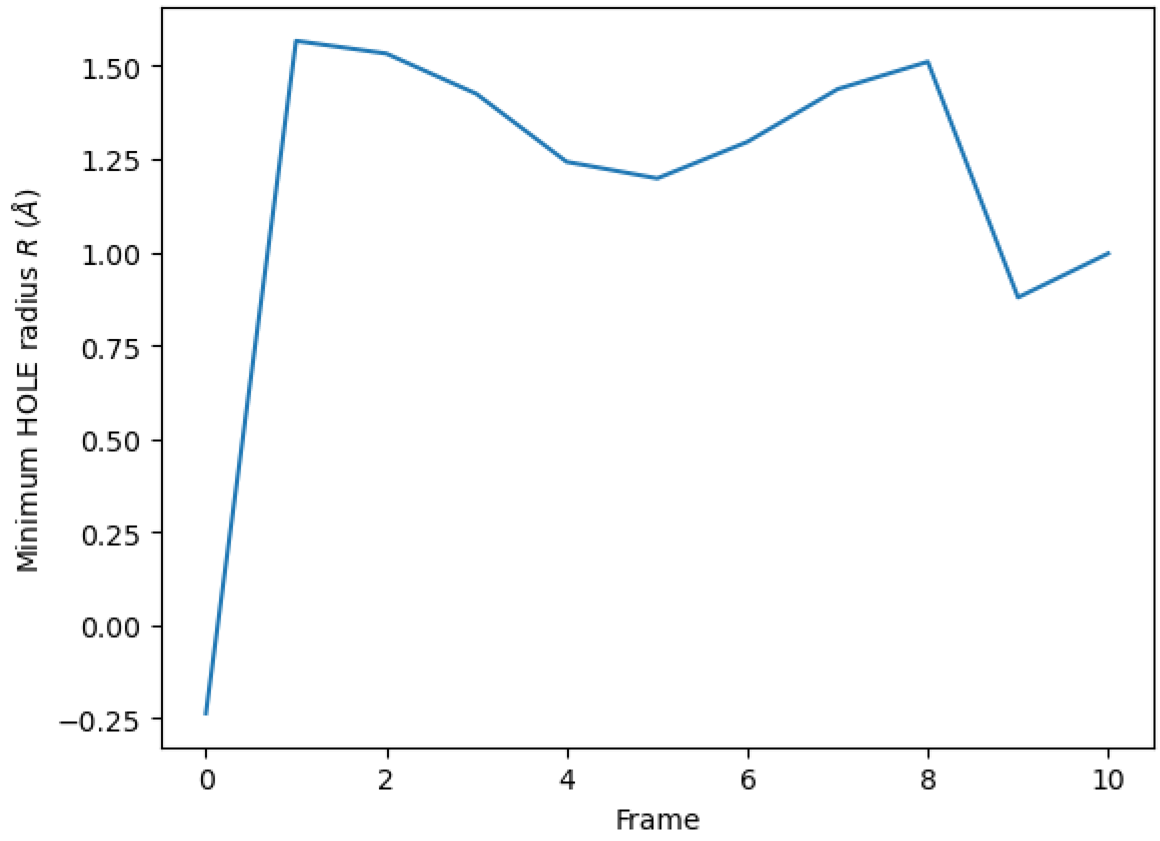
```
Frame 0: -0.237
Frame 1: 1.567
Frame 2: 1.533
Frame 3: 1.425
Frame 4: 1.243
Frame 5: 1.198
Frame 6: 1.296
Frame 7: 1.438
Frame 8: 1.511
Frame 9: 0.879
Frame 10: 0.997
```

[16]:

```
plt.plot(min_radii[:, 0], min_radii[:, 1])
plt.ylabel('Minimum HOLE radius $R$ ($\AA$)')
plt.xlabel('Frame')
```

[16]:

```
Text(0.5, 0, 'Frame')
```



Visualising the VMD surface

The `create_vmd_surface()` method is built into the `HoleAnalysis` class. It writes a VMD file that changes the pore surface for each frame in VMD. Again, load your file and source the file in the Tk Console:

```
source holeanalysis.vmd
```

[17]:
```
ha.create_vmd_surface(filename='holeanalysis.vmd')
```

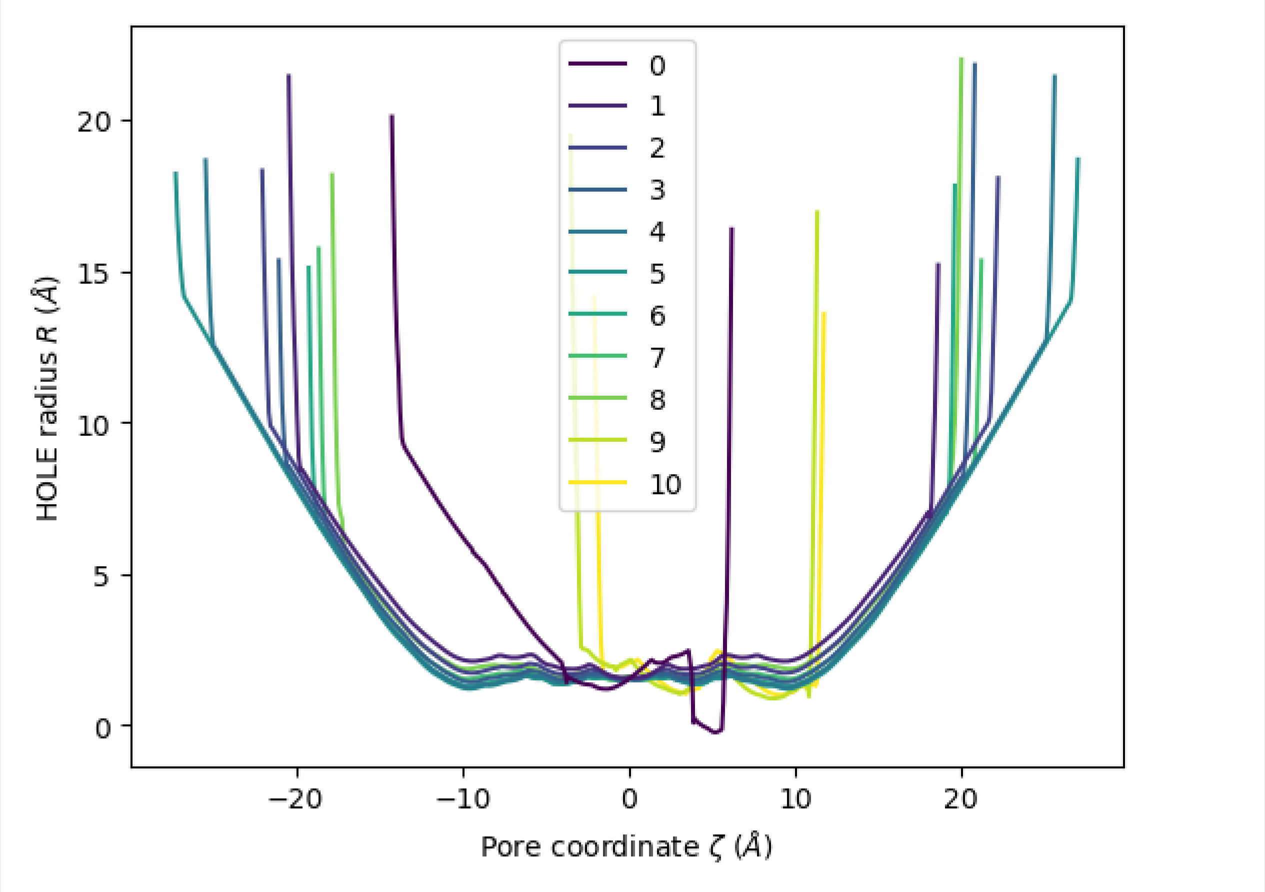
[17]:
```
'holeanalysis.vmd'
```

Plotting

`HoleAnalysis` has several convenience methods for plotting. `plot()` plots the HOLE radius over each pore coordinate, differentiating each frame with colour.

[18]:
```
ha.plot()
```

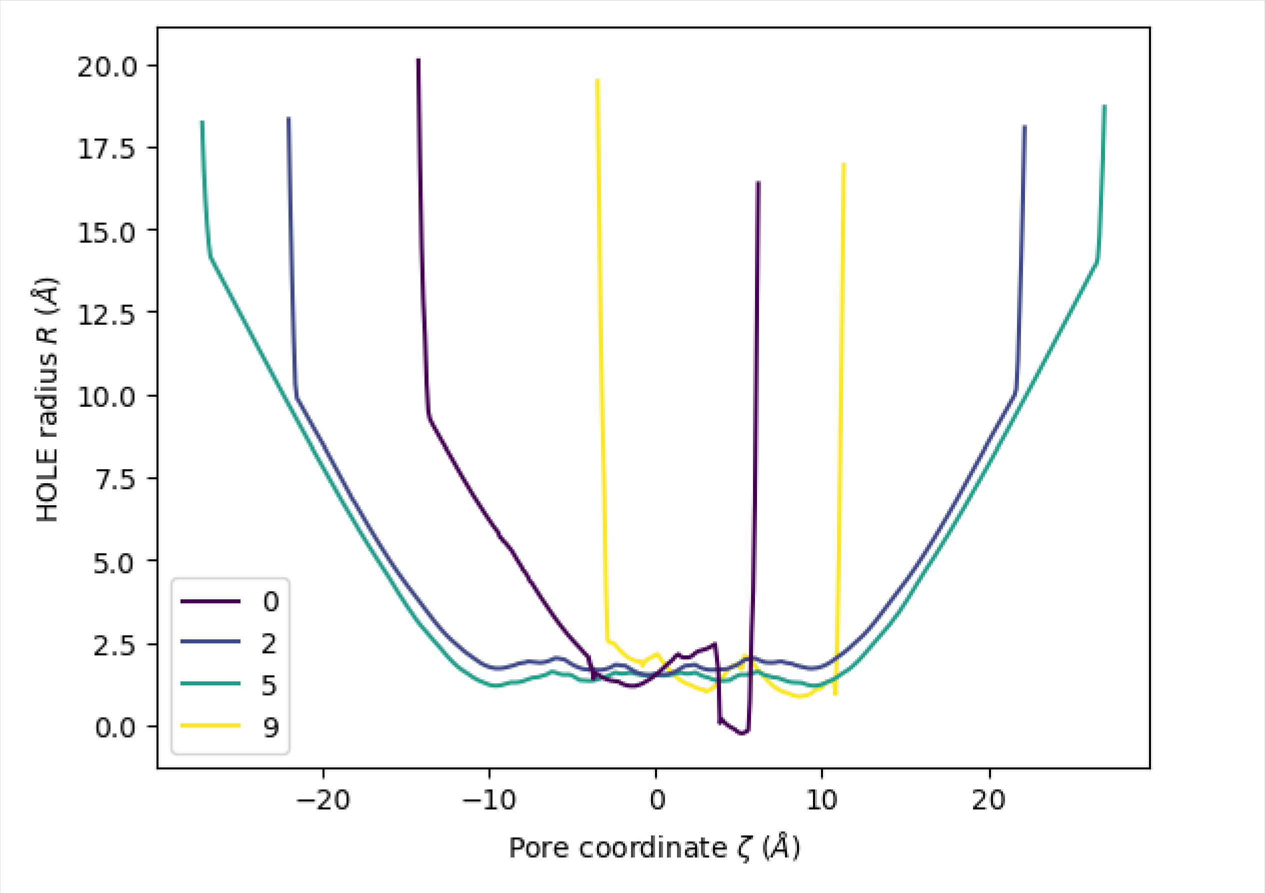
[18]:
```
<AxesSubplot: xlabel='Pore coordinate $\\zeta$ ($\\AA$)', ylabel='HOLE radius $R$ ($\\AA
↪$)'>
```



You can choose to plot specific frames, or specify colours or a colour map. Please see the documentation for a full description of arguments.

```
[19]: ha.plot(frames=[0, 2, 5, 9])
```

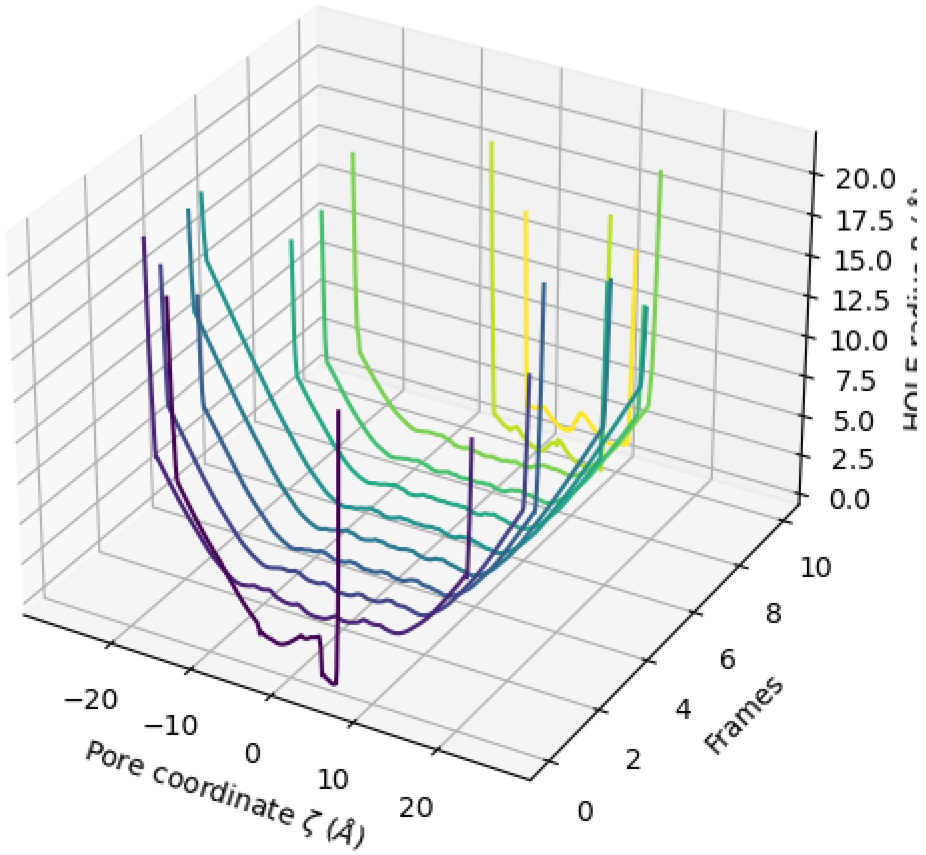
```
[19]: <AxesSubplot: xlabel='Pore coordinate $\\zeta$ ($\\AA$)', ylabel='HOLE radius $R$ ($\\AA
↪$)'>
```



The `plot3D()` function separates each frame onto its own axis in a 3D plot.

```
[20]: ha.plot3D()
```

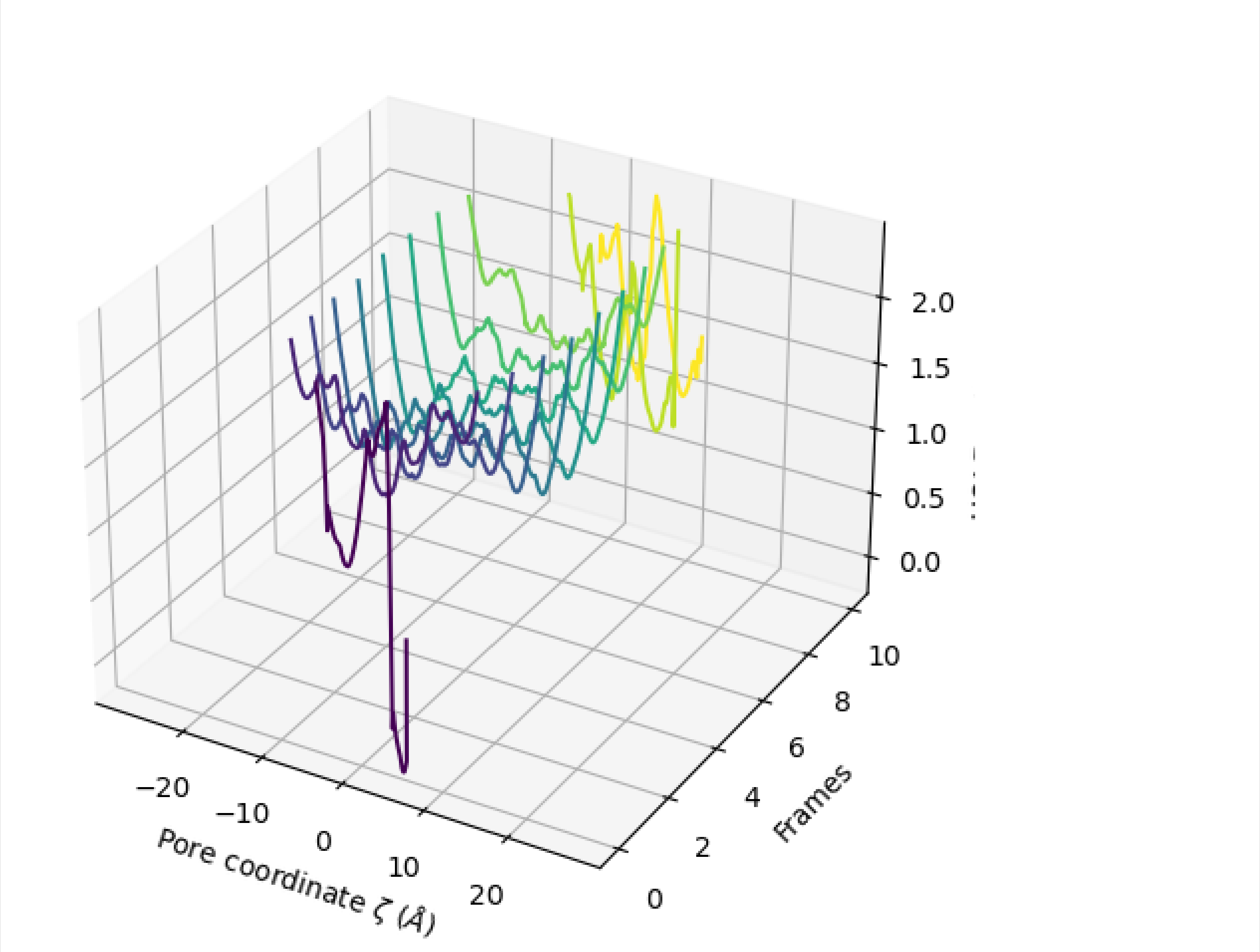
```
[20]: <Axes3DSubplot: xlabel='Pore coordinate $\\zeta$ ($\\AA$)', ylabel='Frames', zlabel=
      ↪'HOLE radius $R$ ($\\AA$)'>
```



You can choose to plot only the part of each pore lower than a certain radius by setting `r_max`.

```
[21]: ha.plot3D(r_max=2.5)
```

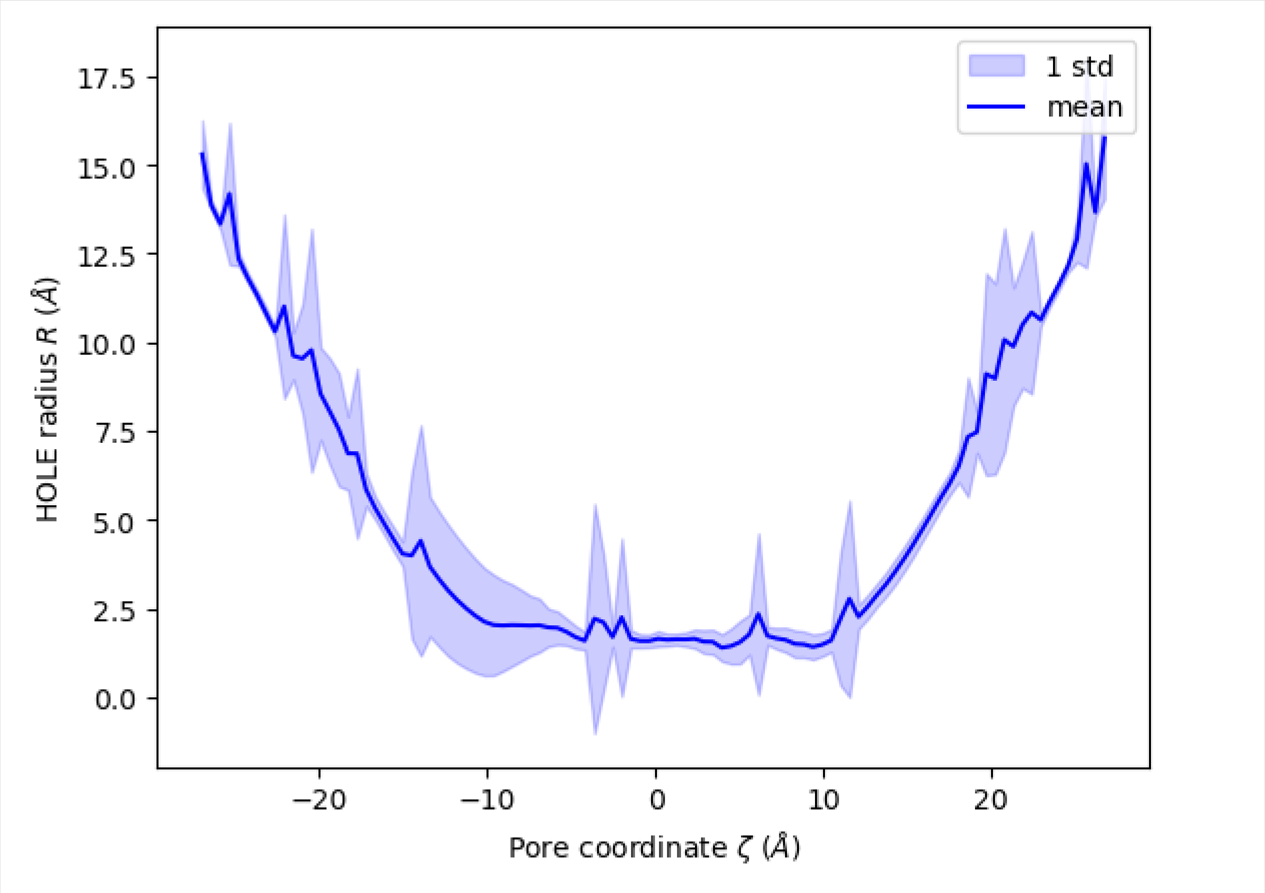
```
[21]: <Axes3DSubplot: xlabel='Pore coordinate $\\zeta$ ($\\AA$)', ylabel='Frames', zlabel=
      ↪'HOLE radius $R$ ($\\AA$)'>
```



You can also plot the mean and standard deviation of the pore radius over the pore coordinate.

```
[22]: ha.plot_mean_profile(bins=100,  # how much to chunk rxn_coord
                         n_std=1,  # how many standard deviations from mean
                         color='blue',  # color of plot
                         fill_alpha=0.2,  # opacity of standard deviation
                         legend=True)
```

```
[22]: <AxesSubplot: xlabel='Pore coordinate $\\zeta$ ($\\AA$)', ylabel='HOLE radius $R$ ($\\AA
↪$)'>
```



Ordering HOLE profiles with an order parameter

If you are interested in the HOLE profiles over an order parameter, you can directly pass that into the analysis after it is run. Below, we use an order parameter of RMSD from a reference structure.

Note

Please cite ([SFSB14]) when using the `over_order_parameters` functionality.

[23]:
```
from MDAnalysis.analysis import rms

ref = mda.Universe(PDB_HOLE)
rmsd = rms.RMSD(u, ref, select='protein', weights='mass').run()
rmsd_values = rmsd.rmsd[:, 2]
for i, rmsd in enumerate(rmsd_values):
    print(f"Frame {i}:  {rmsd:.2f}")
```

```
Frame 0:  6.11
Frame 1:  4.88
Frame 2:  3.66
Frame 3:  2.44
Frame 4:  1.22
```

(continues on next page)

(continued from previous page)

```
Frame 5:  0.00
Frame 6:  1.22
Frame 7:  2.44
Frame 8:  3.66
Frame 9:  4.88
Frame 10:  6.11
```

You can pass this in as `order_parameter`. The resulting `profiles` dictionary will have your order parameters as keys. **You should be careful with this if your order parameter has repeated values, as duplicate keys are not possible; each duplicate key just overwrites the previous value.**

[24]:

```
op_profiles = ha.over_order_parameters(rmsd_values)
```

You can see here that the dictionary does not order the entries by the order parameter. If you iterate over the dictionary, it will return each (key, value) pair in sorted key order.

[25]:

```
for order_parameter, profile in op_profiles.items():
    print(f"{order_parameter:.3f}, {len(profile)}")
```

```
0.000, 543
1.221, 389
1.221, 511
2.442, 419
2.442, 399
3.663, 379
3.663, 443
4.884, 391
4.884, 149
6.105, 205
6.105, 139
```

You can also select specific frames for the new profiles.

[26]:

```
op_profiles = ha.over_order_parameters(rmsd_values, frames=[0, 4, 9])
for order_parameter, profile in op_profiles.items():
    print(f"{order_parameter:.3f}, {len(profile)}")
```

```
1.221, 511
4.884, 149
6.105, 205
```

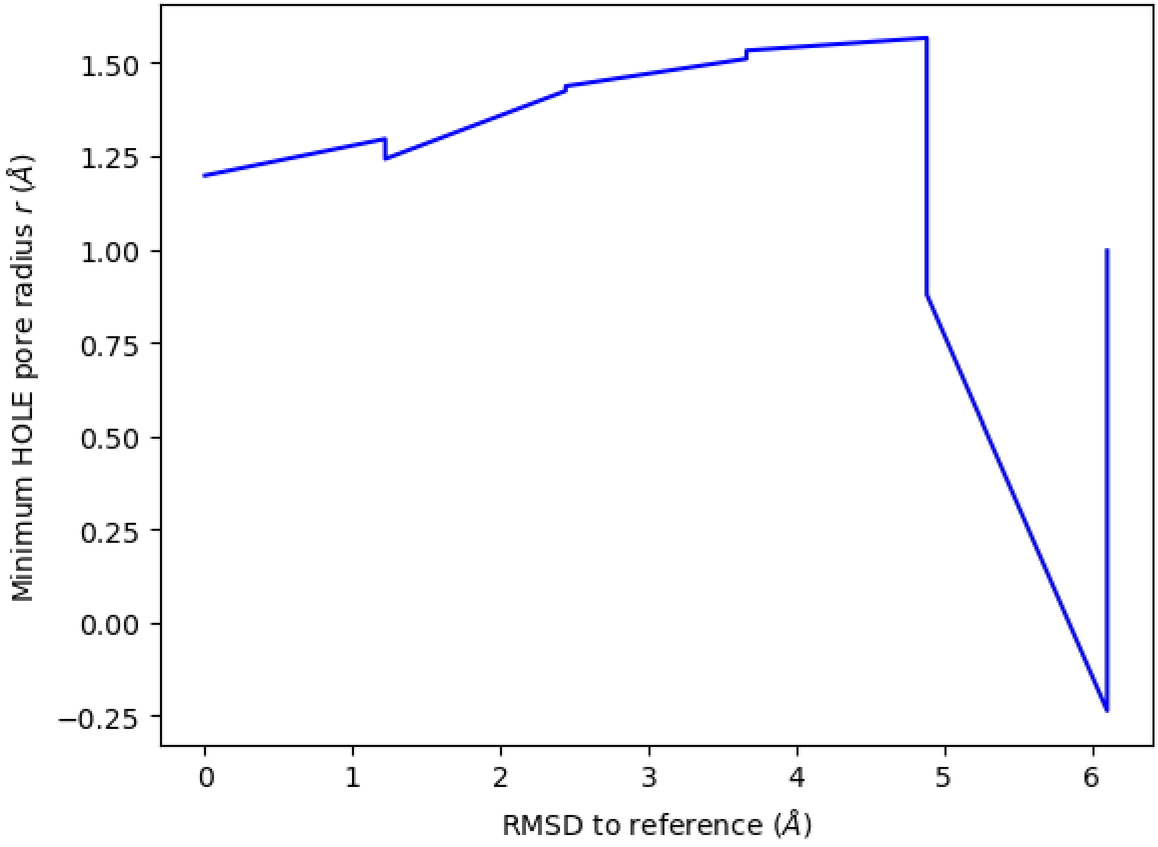
Plotting

`HoleAnalysis` also provides convenience functions for plotting over order parameters. Unlike `plot()`, `plot_order_parameters()` requires an `aggregator` function that reduces an array of radii to a singular value. The default function is `min()`. You can also pass in functions such as `max()` or `np.mean()`, or define your own function to operate on an array and return a vlue.

[27]:

```
ha.plot_order_parameters(rmsd_values,
                         aggregator=min,
                         xlabel='RMSD to reference ($\AA$)',
                         ylabel='Minimum HOLE pore radius $r$ ($\AA$)')
```

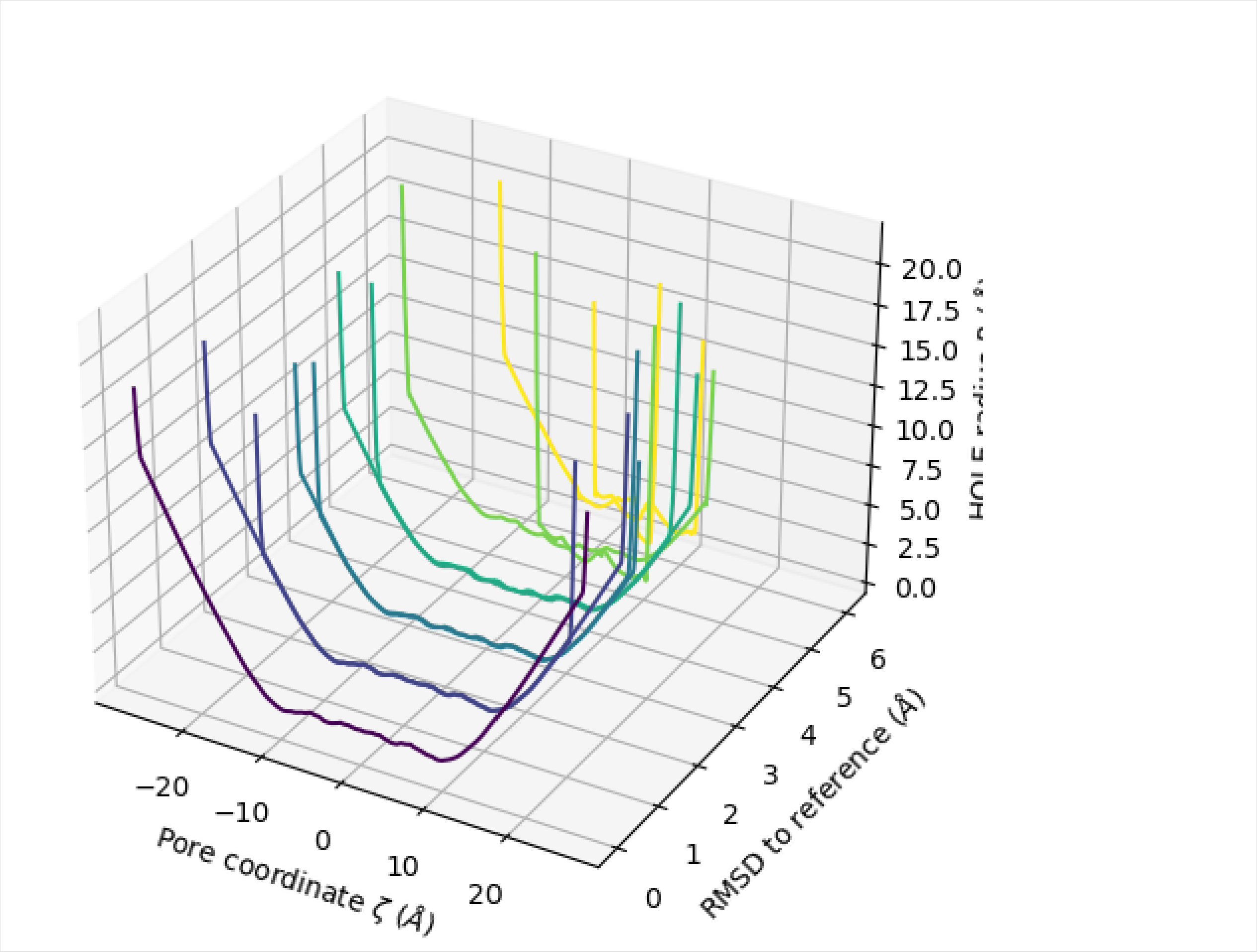
```
[27]: <AxesSubplot: xlabel='RMSD to reference ($\\AA$)', ylabel='Minimum HOLE pore radius $r$ (
↪$\\AA$)'>
```



plot3D_order_parameters() functions in a similar way to plot3D(), although you need to pass in the order parameters.

```
[28]: ha.plot3D_order_parameters(rmsd_values,
                                 ylabel='RMSD to reference ($\AA$)')
```

```
[28]: <Axes3DSubplot: xlabel='Pore coordinate $\\zeta$ ($\\AA$)', ylabel='RMSD to reference ($\
↪\AA$)', zlabel='HOLE radius $R$ ($\\AA$)'>
```



Deleting HOLE files

The HOLE program and related MDAnalysis code write a number of files out. Both the `hole()` function and `HoleAnalysis` class contain ways to easily remove these files.

For `hole()`, pass in `keep_files=False` to delete HOLE files as soon as the analysis is done. However, this will also remove the sphpdb file required to create a VMD surface from the analysis. If you need to write a VMD surface file, use the `HoleAnalysis` class instead.

You can track the created files at the `tmp_files` attribute.

[29]:
```
ha.tmp_files
```

[29]:
```
['simple2.rad',
 'hole000.out',
 'hole000.sph',
 'hole001.out',
 'hole001.sph',
 'hole002.out',
 'hole002.sph',
 'hole003.out',
 'hole003.sph',
```

(continues on next page)

(continued from previous page)

```
 'hole004.out',
 'hole004.sph',
 'hole005.out',
 'hole005.sph',
 'hole006.out',
 'hole006.sph',
 'hole007.out',
 'hole007.sph',
 'hole008.out',
 'hole008.sph',
 'hole009.out',
 'hole009.sph',
 'hole010.out',
 'hole010.sph']
```

The built-in method `delete_temporary_files()` will remove these.

[30]:

```
ha.delete_temporary_files()
ha.tmp_files
```

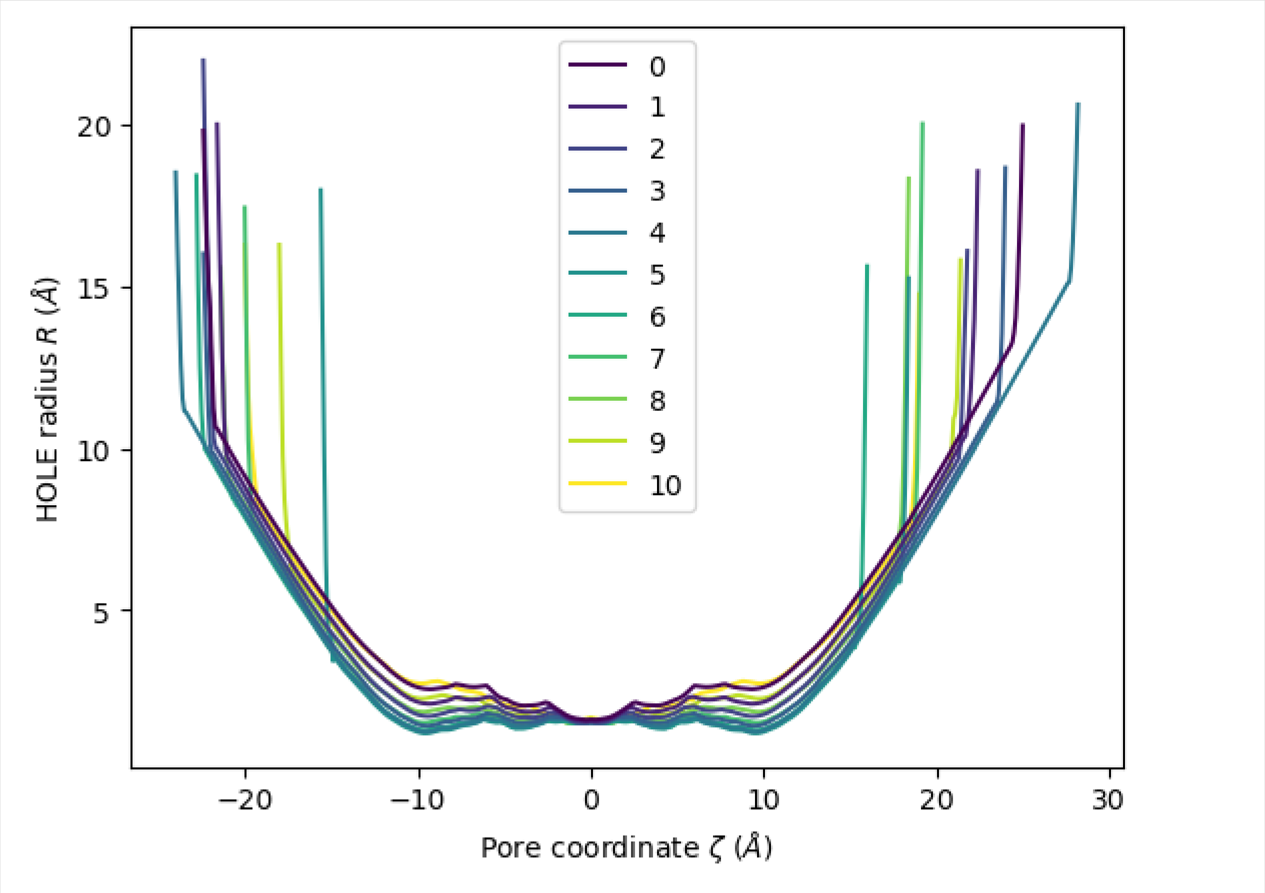
[30]:

```
[]
```

Alternatively, you can use `HoleAnalysis` as a context manager. When you exit the block, the temporary files will be deleted automatically.

[31]:

```
with hole2.HoleAnalysis(u,
    # executable='~/hole2/exe/hole',
    ) as ha2:
    ha2.run()
    ha2.create_vmd_surface(filename='holeanalysis.vmd')
    ha2.plot()
```



```
[32]: ha.profiles[0][0].radius
```

```
[32]: 20.0962
```

References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[3] O S Smart, J M Goodfellow, and B A Wallace. The pore dimensions of gramicidin A. Biophysical Journal, 65(6):2455–2460, December 1993. 00522. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1225986/, doi:10.1016/S0006-3495(93)81293-1.

[4] O. S. Smart, J. G. Neduvelil, X. Wang, B. A. Wallace, and M. S. Sansom. HOLE: a program for the analysis of the pore dimensions of ion channel structural models. Journal of Molecular Graphics, 14(6):354–360, 376, December 1996. 00935. doi:10.1016/s0263-7855(97)00009-x.

[5] Lukas S. Stelzl, Philip W. Fowler, Mark S. P. Sansom, and Oliver Beckstein. Flexible gates generate occluded intermediates in the transport cycle of LacY. Journal of Molecular Biology, 426(3):735–751, February 2014. 00000. URL:

https://asu.pure.elsevier.com/en/publications/flexible-gates-generate-occluded-intermediates-in-the-transport-c, doi:10.1016/j.jmb.2013.10.024.

Volumetric analyses

Computing mass and charge density on each axis

Here we compute the mass and charge density of water along the three cartesian axes of a fixed-volume unit cell (i.e. from a simulation in the NVT ensemble).

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 0.17.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

[1]:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import waterPSF, waterDCD
from MDAnalysis.analysis import lineardensity as lin

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading files

The test files we are working with are a cube of water.

[2]:

```
u = mda.Universe(waterPSF, waterDCD)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

`MDAnalysis.analysis.lineardensity.LinearDensity` (API docs) will partition each of your axes into bins of user-specified `binsize` (in angstrom), and give the average mass density and average charge density of your atom group selection.

This analysis is only suitable for a trajectory with a fixed box size. While passing a trajectory with a variable box size will not raise an error, `LinearDensity` will not account for changing dimensions. It will only evaluate the density of your atoms in the bins created from the trajectory frame when the class is first initialised.

Below, we iterate through the trajectory to verify that its box dimensions remain constant.

[3]:

```
for ts in u.trajectory:
    print(ts.dimensions)
```

```
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
[50. 50. 50. 90. 90. 90.]
```

You can choose to compute the density of individual atoms, residues, segments, or fragments (groups of bonded atoms with no bonds to any atom outside the group). By default, the grouping is for `atoms`.

[4]:

```
density = lin.LinearDensity(u.atoms,
                            grouping='atoms').run()
```

The results of the analysis are in `density.results`.

[5]:

```
density.nbins
```

[5]:

```
200
```

[6]:

```
density.results['x']['mass_density']
```

[6]:

```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.00053562, 0.00080344, 0.00876945, 0.03507781, 0.00107125,
       0.00348155, 0.00241031, 0.02791523, 0.04277601, 0.0175389 ,
       0.00160687, 0.00133906, 0.00026781, 0.        , 0.00107125,
       0.00107125, 0.00053562, 0.        , 0.03400656, 0.0196814 ,
       0.02339659, 0.0135559 , 0.00026781, 0.00107125, 0.00107125,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
```

(continues on next page)

(continued from previous page)

```
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

[7]:

```
density.results['x'].keys()
```

[7]:

```
KeysView({'dim': 0, 'slice_volume': 625.0, 'mass_density': array([0.        , 0.        ,
↪ 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.00053562, 0.00080344, 0.00876945, 0.03507781, 0.00107125,
       0.00348155, 0.00241031, 0.02791523, 0.04277601, 0.0175389 ,
       0.00160687, 0.00133906, 0.00026781, 0.        , 0.00107125,
       0.00107125, 0.00053562, 0.        , 0.03400656, 0.0196814 ,
       0.02339659, 0.0135559 , 0.00026781, 0.00107125, 0.00107125,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
```

(continues on next page)

(continued from previous page)

```
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 'mass_density_stddev
↪': array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.00107125, 0.00122727, 0.01688797, 0.01691979, 0.00177646,
       0.00241031, 0.00279604, 0.02179554, 0.02689655, 0.02096112,
       0.001312  , 0.00133906, 0.00080344, 0.        , 0.001312  ,
       0.001312  , 0.00107125, 0.        , 0.01700328, 0.03402765,
       0.02131476, 0.01957657, 0.00080344, 0.001312  , 0.001312  ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 'charge_density':␣
↪array([ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
```

(continues on next page)

(continued from previous page)

```
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.00022158,  0.00033237, -0.00033237, -0.00132949,  0.00044316,
        0.00144029,  0.00099712, -0.00033237, -0.00210503, -0.00066475,
        0.00066475,  0.00055396,  0.00011079,  0.        ,  0.00044316,
        0.00044316,  0.00022158,  0.        , -0.00177266,  0.00022158,
       -0.00022158, -0.00033237,  0.00011079,  0.00044316,  0.00044316,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ]), 'charge_
↪density_stddev': array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.00044316, 0.00050771, 0.00099712, 0.00108553, 0.00073491,
       0.00099712, 0.00115669, 0.00111344, 0.00144029, 0.00112985,
       0.00054276, 0.00055396, 0.00033237, 0.        , 0.00054276,
       0.00054276, 0.00044316, 0.        , 0.00088633, 0.0018406 ,
       0.00129204, 0.00111344, 0.00033237, 0.00054276, 0.00054276,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
```

(continues on next page)

(continued from previous page)

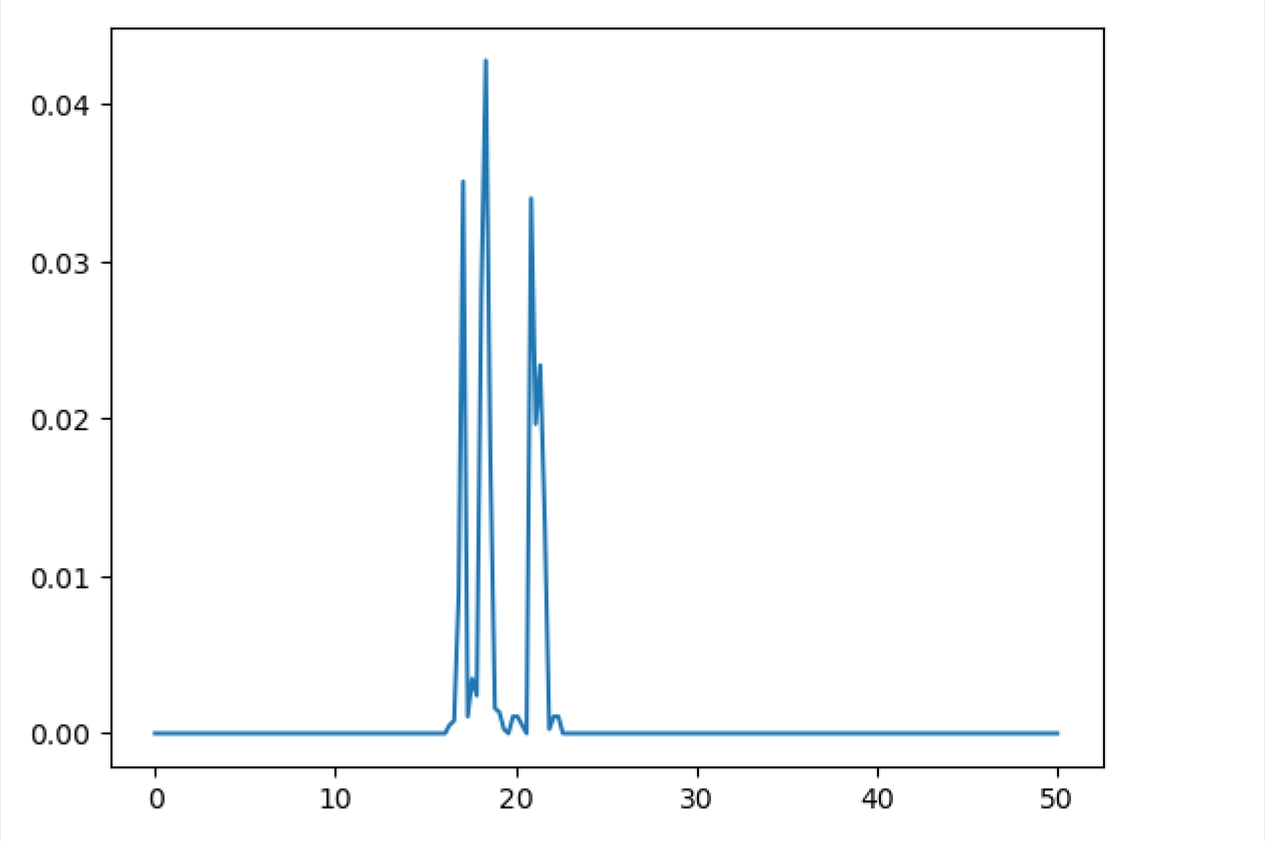
```
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ]), 'hist_bin_edges':
↪array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ,
        2.25,  2.5 ,  2.75,  3.  ,  3.25,  3.5 ,  3.75,  4.  ,  4.25,
        4.5 ,  4.75,  5.  ,  5.25,  5.5 ,  5.75,  6.  ,  6.25,  6.5 ,
        6.75,  7.  ,  7.25,  7.5 ,  7.75,  8.  ,  8.25,  8.5 ,  8.75,
        9.  ,  9.25,  9.5 ,  9.75, 10.  , 10.25, 10.5 , 10.75, 11.  ,
       11.25, 11.5 , 11.75, 12.  , 12.25, 12.5 , 12.75, 13.  , 13.25,
       13.5 , 13.75, 14.  , 14.25, 14.5 , 14.75, 15.  , 15.25, 15.5 ,
       15.75, 16.  , 16.25, 16.5 , 16.75, 17.  , 17.25, 17.5 , 17.75,
       18.  , 18.25, 18.5 , 18.75, 19.  , 19.25, 19.5 , 19.75, 20.  ,
       20.25, 20.5 , 20.75, 21.  , 21.25, 21.5 , 21.75, 22.  , 22.25,
       22.5 , 22.75, 23.  , 23.25, 23.5 , 23.75, 24.  , 24.25, 24.5 ,
       24.75, 25.  , 25.25, 25.5 , 25.75, 26.  , 26.25, 26.5 , 26.75,
       27.  , 27.25, 27.5 , 27.75, 28.  , 28.25, 28.5 , 28.75, 29.  ,
       29.25, 29.5 , 29.75, 30.  , 30.25, 30.5 , 30.75, 31.  , 31.25,
       31.5 , 31.75, 32.  , 32.25, 32.5 , 32.75, 33.  , 33.25, 33.5 ,
       33.75, 34.  , 34.25, 34.5 , 34.75, 35.  , 35.25, 35.5 , 35.75,
       36.  , 36.25, 36.5 , 36.75, 37.  , 37.25, 37.5 , 37.75, 38.  ,
       38.25, 38.5 , 38.75, 39.  , 39.25, 39.5 , 39.75, 40.  , 40.25,
       40.5 , 40.75, 41.  , 41.25, 41.5 , 41.75, 42.  , 42.25, 42.5 ,
       42.75, 43.  , 43.25, 43.5 , 43.75, 44.  , 44.25, 44.5 , 44.75,
       45.  , 45.25, 45.5 , 45.75, 46.  , 46.25, 46.5 , 46.75, 47.  ,
       47.25, 47.5 , 47.75, 48.  , 48.25, 48.5 , 48.75, 49.  , 49.25,
       49.5 , 49.75, 50.  ], dtype=float32)}})
```

```
[8]: density.results['y']['dim']
```

```
[8]: 1
```

```
[9]: plt.plot(np.linspace(0, 50, 200), density.results['x']['mass_density'])
```

```
[9]: [<matplotlib.lines.Line2D at 0x7f5d682b3b20>]
```



References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

Calculating the solvent density around a protein

Here we use `density.DensityAnalysis` to analyse the solvent density around an enzyme.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 1.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

Optional packages for visualisation:

- nglview
- matplotlib
- scikit-image
- pyvista
- ipygany

Throughout this tutorial we will include cells for visualising Universes with the NGLView library. However, these will be commented out, and we will show the expected images generated instead of the interactive widgets.

[1]:
```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import TPR, XTC
from MDAnalysis.analysis import density

import numpy as np
import matplotlib.pyplot as plt
# import nglview as nv
%matplotlib inline
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09]). It is solvated in TIP4P water and broken across periodic boundaries.

[2]:
```
u = mda.Universe(TPR, XTC)
```

[3]:
```
# view1 = nv.show_mdanalysis(u)
# view1.add_representation(
#     'licorice',
#     selection='resname SOL',
# )
# view1
```

[4]:
```
# from nglview.contrib.movie import MovieMaker
# movie = MovieMaker(
#     view1,
#     step=4,  # keep every 4th step
#     render_params={"factor": 3},  # average quality render
#     output='density_analysis_images/density_analysis-view1.gif',
# )
# movie.make()
```

Centering, aligning, and making molecules whole with on-the-fly transformations

`DensityAnalysis` uses a fixed grid to analyse the density of molecules. As it is likely that this grid may cross the unit cell wall, this means that molecules that have broken across the periodic boundary must be made whole. Because we want to analyse the density of water *around a protein*, this means:

- that the solvent must be mapped so they are closest to the protein, and
- we need to align the trajectory on the protein for a fixed frame of reference

In practice, the transformations that we need (in order) are shown in the table below. GROMACS's trjconv is often used to perform these transformations; the equivalent command is also given. MDAnalysis offers on-the-fly transformations to accomplish much the same task; however, where `trjconv` saves the transformed trajectory into a file, MDAnalysis does not alter the initial trajectory. Instead, it transforms each frame "on-the-fly" as it is loaded into MDAnalysis.

Transformation	On-the-fly transformation	GROMACS trjconv argument
Making molecules whole	`wrap.unwrap()`	`-pbc whole`
Moving the protein to the center of the box for more symmetric density	`translate.center_in_box()`	`-center`
Wrapping water back into the box	`wrap.wrap()`	
Aligning the trajectory onto the protein	`fit.fit_rot_trans()`	`-fit rot+trans`

We want wrap water back into the box before we fit the trajectory, in order to avoid odd placements from the rotation in the alignment.

You can do this yourself with external tools such as `gmx trjconv`, using the arguments above. Here, we use on-the-fly transformations so we can avoid writing out new trajectories.

[5]:
```python
from MDAnalysis import transformations as trans

protein = u.select_atoms('protein')
water = u.select_atoms('resname SOL')

workflow = [trans.unwrap(u.atoms),  # unwrap all fragments
            trans.center_in_box(protein, # move atoms so protein
                                center='geometry'), # is centered
            trans.wrap(water, # wrap water back into box
                       compound='residues'), # keep each water whole
            trans.fit_rot_trans(protein, # align protein to first frame
                                protein,
                                weights='mass'),
           ]

u.trajectory.add_transformations(*workflow)
```

When we visualise the transformed trajectory, we can see that it is now centered in the box and whole.

[6]:
```python
# view2 = nv.show_mdanalysis(u)
# view2.add_representation(
#     'licorice',
#     selection='resname SOL',
# )
# view2
```

[7]:

```
# from nglview.contrib.movie import MovieMaker
# movie = MovieMaker(
#     view2,
#     step=4,  # keep every 4th step
#     render_params={"factor": 3},  # average quality render
#     output='density_analysis_images/density_analysis-view2.gif',
# )
# movie.make()
```

Analysing the density of water around the protein

Now that the input trajectory has been pre-processed, we can carry out our analysis. We only want to look at the density of numbers of water molecules, so we choose the oxygen atoms only (see *LinearDensity* for mass and charge density analysis).

[8]:

```
ow = u.select_atoms('name OW')
dens = density.DensityAnalysis(ow,
                               delta=4.0,
                               padding=2)
dens.run()
```

[8]:

```
<MDAnalysis.analysis.density.DensityAnalysis at 0x1444979d0>
```

The results are stored in `dens.density`, a `Density` object. `dens.density.grid` is a numpy array with the average density of the water oxygen atoms, histogrammed onto a grid with 1 spacing on each axis.

[9]:

```
grid = dens.results.density.grid
grid.shape
```

[9]:

```
(31, 42, 20)
```

When first calculated, these are in the default units of $^{-3}$.

[10]:

```
dens.results.density.units
```

[10]:

```
{'length': 'Angstrom', 'density': 'Angstrom^{-3}'}
```

You can convert the units both for the length (`convert_length`) and for the density (`convert_density`). MDAnalysis stores a number of precomputed ways to convert units. Densities can be converted to nm^{-3}, or converted to a density relative to the bulk density. After executing the code below, the array at `dens.density.grid` now contains the density of water relative to bulk TIP4P water at ambient positions.

[11]:

```
dens.results.density.convert_density('TIP4P')
dens.results.density.units
```

[11]:

```
{'length': 'Angstrom', 'density': 'TIP4P'}
```

Visualisation

A number of 3D and 2D visualization methods are illustrated below.

matplotlib (3D static plot)

You may want to visualise your densities as part of your analysis. One trivial way is to plot the density of water around the protein as a 3D scatter plot.

First we need to obtain the x, y, and z axes for the plot by taking the midpoints of the histogram bins. These are stored as `dens.density.midpoints`.

[12]:

```
mx, my, mz = dens.results.density.midpoints
```

In the plot below we represent the density of water in a particular bin with the opacity of the scatter point. To do that, we need to first normalise the density values. In the `flat` vector below before, the highest opacity (i.e. the point with the highest density of water oxygen atoms) is 0.1. The array is also flattened so we can treat it as a list of values.

[13]:

```
grid = dens.results.density.grid
flat = grid.ravel() / (grid.max()*10)
```

We set the colour to an RGBA array representing the colour blue. The last number in an RGBA array represents the alpha channel, which controls the opacity of the point.

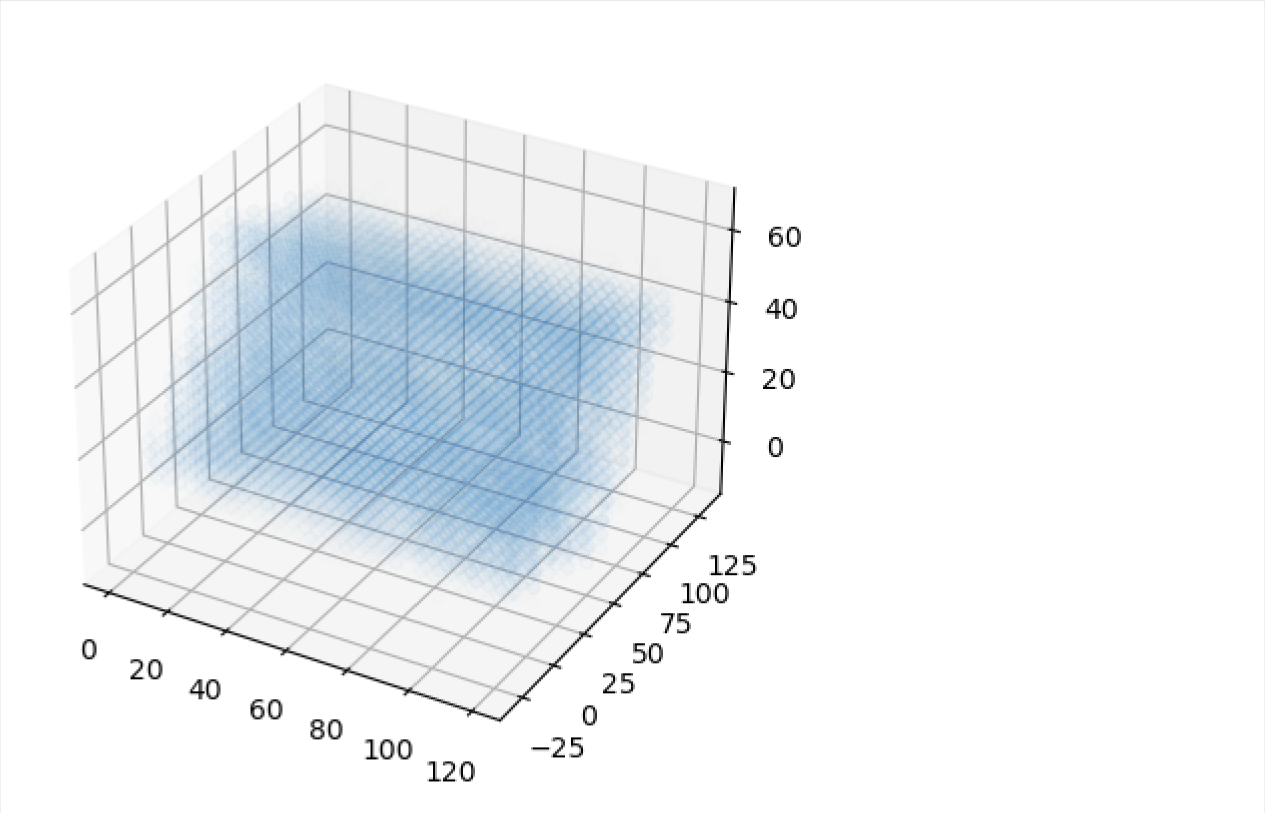
[14]:

```
blue = [44, 130, 201, 1]
colors = [blue] * len(mx) * len(my) * len(mz)
colors = np.array(colors, dtype=float)
colors[:, -1] *= flat
colors[:, :3] /= 255
```

Finally we can plot the points on a 3D plot. `Axes3D` must be imported for a 3D plot, even though we do not directly use it. In this case, the plot is not very interesting; it just looks like a box of water.

[15]:

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x, y, z = np.meshgrid(mx, my, mz)

ax.scatter(x, y, z, c=colors)
plt.show()
```



nglview (interactive)

You could also view the density in NGLView by exporting it to a DX format:

```
[16]: dens.results.density.export("water.dx")
```

Use the surface representation in NGLViewer to visualize the loaded density at different isolevels: - `contour = False` shows a continuous surface (the default); `True` shows a wirefram - `isolevel = float` sets the contour level and with `isolevel_type="value"` is in the units of the density - `isolevel_type="value"` for densities (the default is "sigma") and then `isolevel` has a different meanin - One can use multiple surfaces at different isolevels (although the current example trajectory has too few frames to generate a well resolved density - `smooth = float` controls the surface smoothing of the representation

```
[17]: # view3 = nv.show_mdanalysis(u)
      # d = view3.add_component("water.dx")
      # d.clear_representations()
      # d.add_surface(isolevel=0.5, isolevel_type="value", opacity=0.1, contour=False,␣
      ↪smooth=1, color="blue")
      # d.add_surface(isolevel=1.2, isolevel_type="value", opacity=1, contour=True, smooth=1,␣
      ↪color="cyan")

      # view3
```

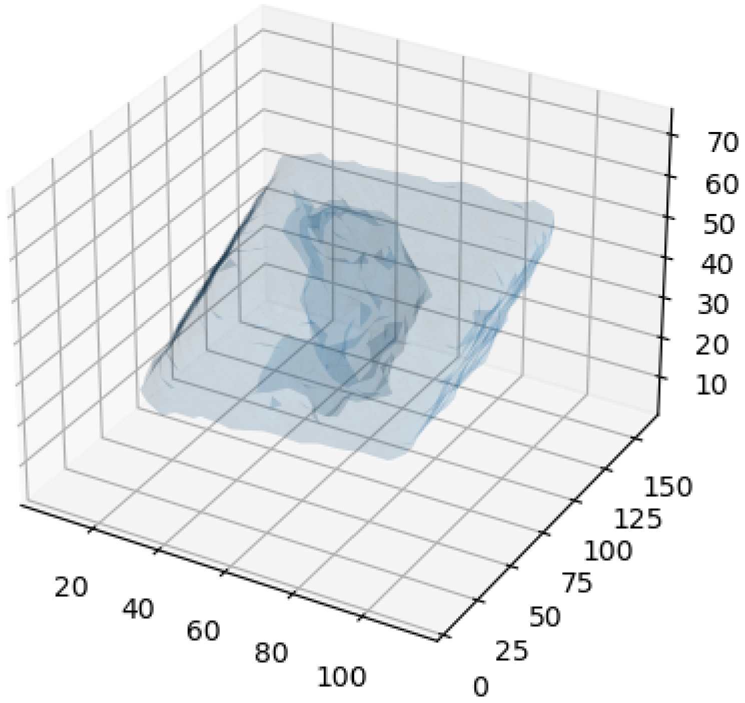
scikit-image (triangulated surface)

You could use the Marching Cube (Lewiner) algorithm to triangulate the surface (following this tutorial and StackOverflow post.

This will require the skimage library.

[18]:

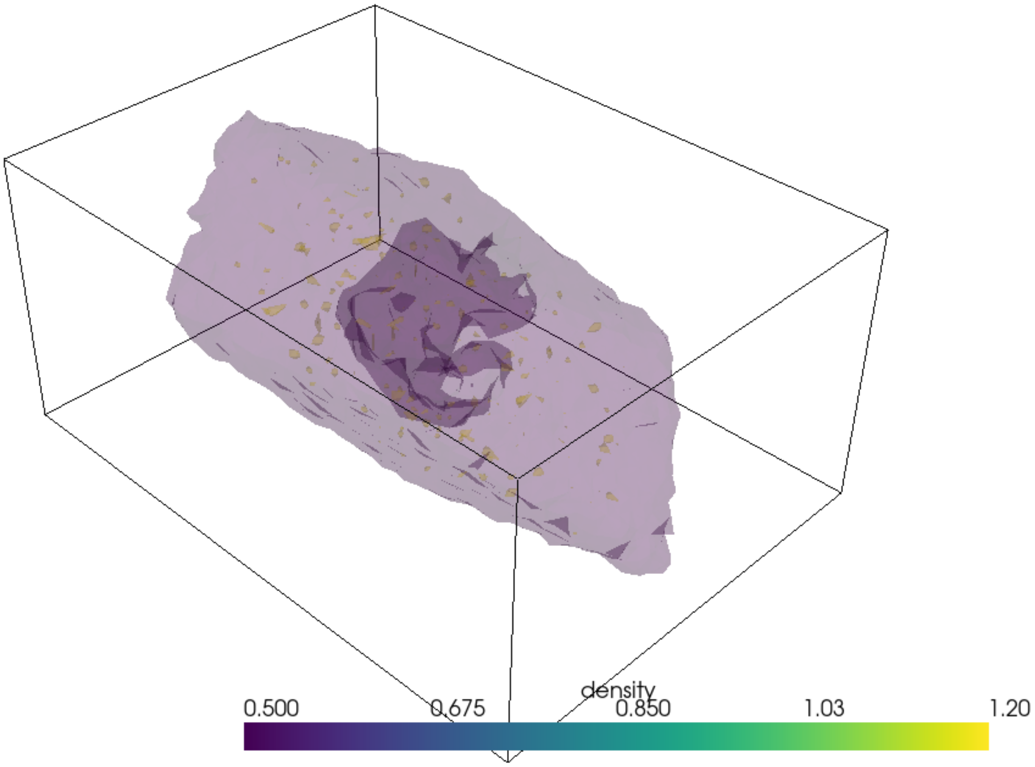
```
from skimage import measure
from mpl_toolkits.mplot3d import Axes3D

iso_val = 0.5
verts, faces, _, _ = measure.marching_cubes(dens.results.density.grid, iso_val,
                                            spacing=dens.results.density.delta)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_trisurf(verts[:, 0], verts[:,1], faces, verts[:, 2],
                lw=1, alpha=0.1)
plt.show()
```



pyvista (3D surface)

Similarly, using PyVista, you can plot both static and interactive visualizations of the surface at different iso levels (following this StackOverflow post). Uncomment the last lines to show the plot in your local machine.

[19]:
```
import pyvista as pv

pv.set_plot_theme("document")

x, y, z = np.meshgrid(mx, my, mz, indexing="ij")
mesh = pv.StructuredGrid(x, y, z)
mesh["density"] = dens.results.density.grid.T.flatten() # note transpose
contours = mesh.contour([0.5, 1.2])
p = pv.Plotter(notebook=True)
p.background_color = 'white'
p.add_mesh(mesh.outline(), color="k")  # box lines
p.add_mesh(contours, opacity=0.2);  # surfaces
# p.show()
# p.screenshot("./density_analysis_images/surface.png");
```



Unfortunately plotting interactively appears to render everything with opaque surfaces. Note that this code snippet requires ipygany to be installed.

[20]:

```
p = pv.Plotter(notebook=True)
p.background_color = 'white'
p.add_mesh(mesh.outline(), color="k")  # box lines
p.add_mesh(contours, opacity=0.2);  # surfaces
# uncomment the below for interactivity
# p.show(jupyter_backend="ipygany")
# p.screenshot("./density_analysis_images/interactive-surface.png");
```

2D averaging

Alternatively, you could plot the average density of water on the xy-plane. We get the average x-y positions by averaging over the z-axis.

[21]:

```
avg = grid.mean(axis=-1)
avg.shape
```

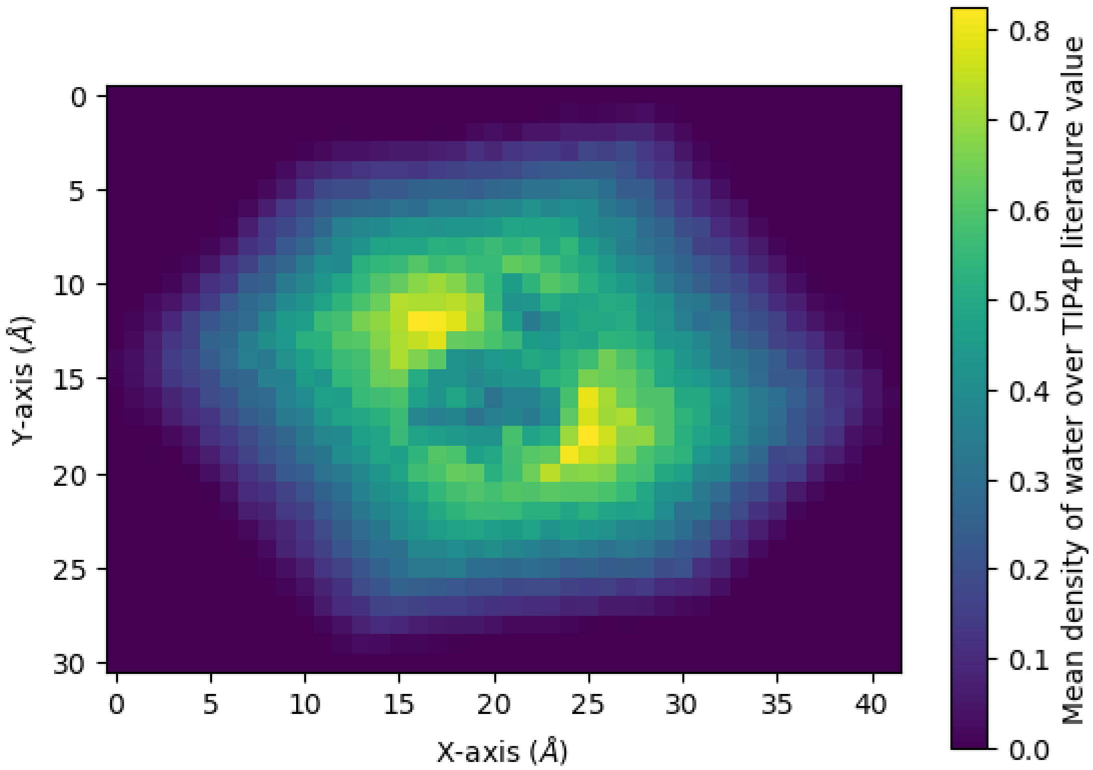
[21]:

```
(31, 42)
```

Below, it is plotted as a heat map.

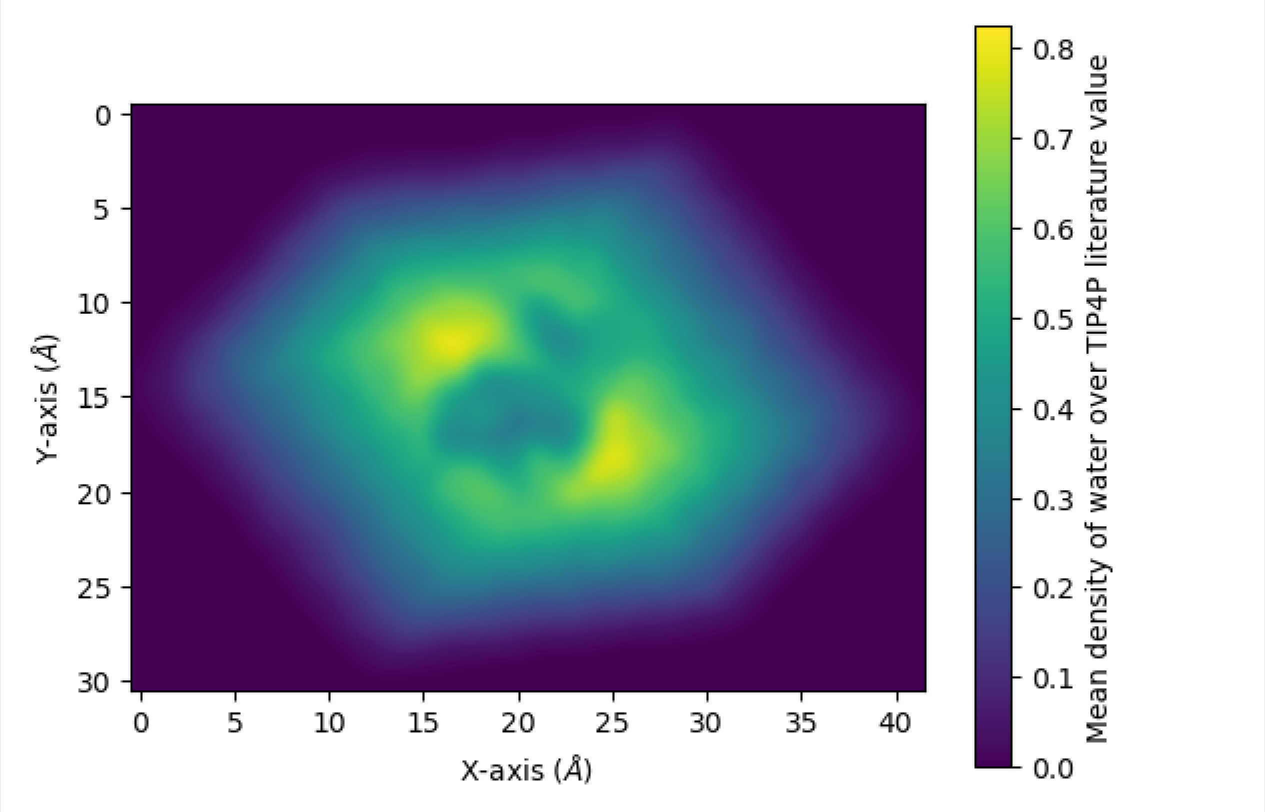
[22]:

```
fig, ax = plt.subplots()

im = ax.imshow(avg)
cbar = plt.colorbar(im)
cbar.set_label('Mean density of water over TIP4P literature value')
plt.xlabel('X-axis ($\AA$)')
plt.ylabel('Y-axis ($\AA$)')
plt.show()
```



You can interpolate values for a smoother average:

```
[23]: fig, ax = plt.subplots()

im = ax.imshow(avg, interpolation="bicubic")
cbar = plt.colorbar(im)
cbar.set_label('Mean density of water over TIP4P literature value')
plt.xlabel('X-axis ($\AA$)')
plt.ylabel('Y-axis ($\AA$)')
plt.show()
```



References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

2.1.5 MDAnalysis Release Notes

Release 2.6.1 of MDAnalysis

This is a bugfix release of the 2.6.x version branch of MDAnalysis, it serves as an amendment to the earlier released version 2.6.0.

See the CHANGELOG for more details.

Bug fixes and changes

- Reverting the v2.6.0 behaviour, builds are now again made using the oldest supported NumPy version (NumPy 1.22.3 for Python 3.9-3.10, and 1.22.3 for Python 3.11) [PR #4261]
- Uses of numpy `in1d` have been replaced with `isin` in prepartion for NumPy 2.0 [PR #4255]
- Cython DEF statements have been replaced with compile time integer constants as DEF statements are now deprecated in Cython 3.0 [Issue #4237, PR #4246]
- Fix to element guessing code to more accurately interpret atom names split by numbers (i.e. N0A is now recognised as N rather than NA) [Issue #4167, PR #4168]
- Clarification of SurvivalProbability function documentation [Issue #4247, PR #4248]1

New Contributors

- @pillose made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4168

Release 2.6.0 of MDAnalysis

This a minor release of MDAnalysis.

This release of MDAnalysis is packaged under a GPLv3+ license, additionally all contributions made from commit 44733fc214dcfdcc2b7cb3e3705258781bb491bd onwards are made under the LGPLv2.1+ license. More details about these license changes will be provided in an upcoming blog post.

The minimum supported NumPy version has been raised to 1.22.3 as per NEP29. Please note that package builds are now made with NumPy 1.25+ which offer backwards runtime compatibility with NEP29 supported versions of NumPy.

Supported Python versions:

- 3.9, 3.10, 3.11

Major changes:

See the CHANGELOG and our release blog post for more details.

Fixes:

- The -ffast-math compiler flag is no longer used by default at build time, avoiding inconsistent (although still scientifically correct) results seen in Intel MacOS systems when calling `AtomGroup.center_of_charge(…, unwrap=True). This also avoids potentially incorrect floating point results as detailed here. (https://github.com/MDAnalysis/mdanalysis/pull/4220)
- DATAWriter, CRD, PQR, and PDBQT files can now be correctly written to compressed files. Before this change, any attempt to write to a compressed format (gzip or bzip2) would lead to writing out an uncompressed file. (https://github.com/MDAnalysis/mdanalysis/pull/4163)
- Prevent accidental merging of bond/angle/dihedral types when they are defined as LAMMPS style string integers instead of tuples. This was leading to an incorrect number of bond/angle/dihedral types being written to lammps data files. (https://github.com/MDAnalysis/mdanalysis/pull/4003)

Enhancements:

- An `exclude_same` argument has been added to `InterRDF` allowing pairs of atoms that share the same residue, segment or chain to be excluded from the calculation. (https://github.com/MDAnalysis/mdanalysis/pull/4161)
- LAMMPS reader now supports the `continuous` ChainReader option. (https://github.com/MDAnalysis/mdanalysis/pull/4170)
- AtomGroup representation now returns atom indices in the same order as they are stored in the AtomGroup. (https://github.com/MDAnalysis/mdanalysis/pull/4191)

Changes:

- Package builds now use NumPy 1.25 or higher instead of the lowest supported NumPy version. (https://github.com/MDAnalysis/mdanalysis/pull/4198)
- As per NEP29, the minimum supported runtime version of NumPy has been increased to 1.22.3. (https://github.com/MDAnalysis/mdanalysis/pull/4160)
- The GSD package is now an optional dependency. (https://github.com/MDAnalysis/mdanalysis/pull/4174)
- The MDAnalysis package now only supports GSD versions 3.0.1 or above. (https://github.com/MDAnalysis/mdanalysis/pull/4153)
- MDAnalysis no longer officially supports 32 bit installations. (they are no longer tested in our continuous integration pipelines). Note: no code changes have been made to disable 32 bit, although it is known that new versions of most MDAnalysis core dependencies no longer release 32 bit compatible versions. (https://github.com/MDAnalysis/mdanalysis/pull/4176)
- The package license has been updated to GPLv3+ to better reflect the compatibility of GPLv2+ with Apache and GPLv3 licensed codes. Additionally all new contributions from commit 44733fc214dcfdcc2b7cb3e3705258781bb491bd onwards are made under the LGPLv2.1+ license. (https://github.com/MDAnalysis/mdanalysis/pull/4219)

Deprecations:

- The misspelt `Boltzman_constant` entry in `MDAnalysis.units` is now deprecated in favour the correctly spelt `Boltzmann_constant`. (https://github.com/MDAnalysis/mdanalysis/pull/4230 and https://github.com/MDAnalysis/mdanalysis/pull/4214)
- `MDAnalysis.analysis.hole2` is now deprecated in favour of a new HOLE2 MDAKit. (https://github.com/MDAnalysis/mdanalysis/pull/4200)

New Contributors

- @MohitKumar020291 made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4182
- @Shubx10 made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4184
- @ztimol made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4191

Release 2.5.0 of MDAnalysis

This a minor release of MDAnalysis.

The minimum Python version has been raised to 3.9 and NumPy to 1.21.0 as per NEP29. We also now package wheels for both linux and osx arm64 machines on PyPi.

Supported Python versions:

- 3.9, 3.10, 3.11

Major changes:

See the CHANGELOG and our release blog post for more details.

Fixes:

- Fixed an issue where transformations were not being properly applied to Universes with multiple trajectories (i.e. using the ChainReader) (Issue #3657 #4008 PR #3906)
- Fixed an issue with the the `heavy` distance_type for `WaterBridgeAnalysis` where distance was not correctly assigned when more than one hydrogen was bonded to a heavy atom (Issue #4040, PR #4066).
- PDB topology parser no longer fails when encountering unknown formal charges and instead simply does not populate attribute (Issue #4027)
- Fixed an issue where using the `between` keyword of `HydrogenBondAnalysis` led to incorrect donor-atom distances being returned (PR #4092, Issue #4091)
- Fixed an issue where chi1_selections() ignored atom names CG1 OG OG1 SG and incorrectly returned `None` for amino acids CYS, ILE, SER, THR, VAL (Issue #4108, PR #4109)
- Fix H5MD reader to read box vectors rather than returning `None` as the dimensions (Issue #4075, PR #4076)
- Fix to allow reading NetCDF trajectories which do not have the `time` variable defined (Issue #4073, PR #4074)
- Allows shape_parameter and asphericity to yield per residue quantities (Issue #3002, PR #3905)
- Fix EDRReader failing when parsing single-frame EDR files (Issue #3999)

- Add 'PairIJ Coeffs' to the list of sections in LAMMPSParser.py (Issue #3336)
- PDBReader now defaults atom values for ts.data['occupancy'] to 0.0, rather than the previous default of 1.0. This now matches the default used when setting Universe Topology attributes using the first frame's information (PR #3988)

Enchancements:

- ARM64 (osx and linux) wheels are now provided via PyPi (Issue #4054)
- Addition of a new analysis class `analysis.atomicdistances.AtomicDistances` to provide distances between two atom groups over a trajectory. (Issue #3654, PR #4105)
- Add kwarg `n_frames` to class method `empty()` in `MDAnalysis.core.universe`, enabling creation of a blank `Universe` with multiple frames (PR #4140)
- PDBReader now populates ts.data['tempfactor'] with the tempfactor for each atom *for each frame*. If an entry is missing for a given atom, this will default to a `1.0` value. Note, this does not affect the topology, i.e. `AtomGroup.tempfactors` is not dynamically updated. (Issue #3825, PR #3988)
- Add writing u.trajectory.ts.data['molecule_tag'] as molecule tags to LAMMPS data file (Issue #3548)
- Add `progressbar_kwargs` parameter to `AnalysisBase.run` method, allowing to modify description, position etc of tqdm progressbars. (PR #4085)
- Add a nojump transformation, which unwraps trajectories so that particle paths are continuous. (Issue #3703, PR #4031)
- Added AtomGroup TopologyAttr to calculate gyration moments (Issue #3904, PR #3905)
- Add support for TPR files produced by Gromacs 2023 (Issue #4047)
- Add distopia distance calculation library bindings as a selectable backend for `calc_bonds` in `MDA.lib.distances`. (Issue #3783, PR #3914)
- AuxReaders are now pickle-able and copy-able (Issue #1785, PR #3887)
- Add pickling support for Atom, Residue, Segment, ResidueGroup and SegmentGroup. (PR #3953)

Changes:

- As per NEP29 the minimum supported Python version has been raised to 3.9 and NumPy has been raised to 1.21 (note: in practice later versions of NumPy may be required depending on your architecture, operating system, or Python version) (PRs #4115 and #3983).
- Add progress bars to track the progress of `mds.EinsteinMSD` _conclude() methods (_conclude_simple() and _conclude_fft()) (Issue #4070, PR #4072)
- The deprecated direct indexing and `times` from the `results` attribute of analysis.nucleicacids' NucPairDist and WatsonCrickDist classes has been removed. Please use the `results.pair_distances` and `times` attributes instead (Issue #3744)
- RDKitConverter changes (part of Issue #3996):
  - moved some variables (`MONATOMIC_CATION_CHARGES` and `STANDARDIZATION_REACTIONS`) out of the related functions to allow users fine tuning them if necessary.
  - changed the sorting of heavy atoms when inferring bond orders and charges: previously only based on the number of unpaired electrons, now based on this and the number of heavy atom neighbors.

  - use RDKit's RunReactantInPlace for the standardization reactions, which should result in a significant speed improvement as we don't need to use bespoke code to transfer atomic properties from the non-standardized mol to the standardized one.

New Contributors

- @mglagolev made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/3959
- @chrispfae made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4009
- @ooprathamm made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4010
- @MeetB7 made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4022
- @v-parmar made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4032
- @MoSchaeffler made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4049
- @jandom made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4043
- @xhgchen made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4037
- @DrDomenicoMarson made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4074
- @AHMED-salah00 made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4059
- @schlaicha made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4076
- @jvermaas made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4031
- @SophiaRuan made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4072
- @marinegor made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4085
- @g2707 made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4089
- @DanielJamesEvans made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/4109

Release 2.4.3 of MDAnalysis

This is a bugfix release of the 2.4.x version of MDAnalysis, it serves as an amendment to the earlier released version 2.4.2.

Bug fixes

- Fixed DCD reading for large (>2Gb) files (Issue #4039). This was broken for versions 2.4.0, 2.4.1 and 2.4.2.
- Fix element parsing from PSF files tests read via Parmed (Issue #4015)

Release 2.4.2 of MDAnalysis

This is a bugfix release of the 2.4.x version of MDAnalysis, it serves as an amendment to the earlier released version 2.4.1.

Bug fixes

- Fixed an issue where the arguments passed to np.histogramdd in `MDAnalysis.analysis.DensityAnalysis` were not compatible with the 1.24 release of NumPy (PR #3976)
- Fixed upcoming incompatibilities with NumPy 1.25 in `MDAnalysis.visualization.streamlines_3D` and `MDAnalysis.visualization.streamlines` where incorrect comparison of the truth of arrays would have led to failures (PR #3977)

Release 2.4.1 of MDAnalysis

This is a bugfix release of the 2.4.x version of MDAnalysis, it serves as an amendment to the earlier released version 2.4.0.

Bug fixes

- The minimum version of biopython has been raised to 1.80 for pip installs
- pytng has been added as an optional dependency

Release 2.4.0 of MDAnalysis

This a minor release of MDAnalysis, as per our once-every-three-months schedule.

The minimum NumPy and Python versions remain largely unchanged, however the minimum version of `biopython` has been raised to 1.80. This is also the first release to officially support Python 3.11.

Supported Python versions:

- 3.8, 3.9, 3.10, 3.11

Major changes:

See the CHANGELOG and our release blog post for more details.

Fixes:

Enchancements:

- As part of their outreachy project @umak has started adding type annotations throughout the MDAnalysis codebase
- As part of their GSoC project @BFedder has added an auxialliary reader for EDR files (PR #3749)
- As part of their GSoC project @aya9aladdin has fixed various issues with guessing and and attribute reading. This will be followed by the introduction of a new guesser system in a future release.

- A reader for TNG files has been added by @hmacdope, follow up on his previous GSoC 2020 work on creating a python library for reading TNG files (PR 3765)
- Addition of a new isolayer selection method (PR #3846)
- Various enchancements and fixes to the LAMMPS DUMP Parser (allowing box translation on reading, allowing coordinates to be unwrapped based on dump image flags, and importing of forces and velocities) (PR #3844)
- All readers now have a timeseries attribute (PR #3890)
- ReaderBase file formats now accept pathlib inputs (PR #3935)
- Added ability for hbond analysis to use types when resnames are not present (PR #3848)

Changes:

- The deprecated setup.py `extra_requires AMBER` entry has been removed in favor of `extra_formats` (PR #3810)
- Various issues with the auxilliary reader, this should not be much more robust (PR #3749)
- The Cython headers have been moved to MDAnalysis.lib.libmdanalysis (PR #3913)
- The `MDAnalysis.analysis.align.sequence_alignment` now uses Bio.Align.PairwiseAligner instead of the deprecated Bio.pairwise2 (PR #3951)

Deprecations:

- The MemoryReader's timeseries inclusive indexing will be changed to exclusive in version 3.0.0 (PR #3894)
- The `sequence_aligment()` method has been deprecated and will be removed in version 3.0.0 (PR #3951)
- MDAnalysis.analysis.nucleicacids' direct indexing of selection indices to obtain pair distances results has been deprecated in favor of accessing `results.pair_distances` (PR #3958)

New Contributors

- @jaclark5 made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/3846
- @pgbarletta made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/3876
- @jfennick made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/3832
- @Hakarishirenai made their first contribution in https://github.com/MDAnalysis/mdanalysis/pull/3956

Release 2.3.0 of MDAnalysis

This a minor release of MDAnalysis, as per our once-every-three-months schedule.

The minimum NumPy version has been raised to 1.20.0 (1.21 for macosx-arm64) in line with NEP29.

Supported python versions:

- 3.8, 3.9, 3.10

Major changes:

See the CHANGELOG and our release blog post for more details.

Fixes:

- Fixed reading error when dealing with corrupt PDB CONECT records, and an issue where MDAnalysis would write out unusable CONECT records with index>100000 (Issue #988).

Enhancements:

- Formal charges are now read from PDB files and stored in a `formalcharge` attribute (PR #3755).
- A new normalizing `norm` parameter for the `InterRDF` and `InterRDF_s` analysis methods (Issue #3687).
- Improved Universe serialization performance (Issue #3721, PR #3710).

Changes:

- To install optional packages for different file formats supported by MDAnalysis, use `pip install ./package[extra_formats]` (Issue #3701, PR #3711).

Deprecations:

- The `extra_requires` target `AMBER` for `pip install ./package[AMBER]` will be removed in 2.4.0. Use `extra_formats` (Issue #3701, PR #3711).

CZI EOSS Performance Improvements:

A series of performance improvements to the MDAnalysis library's backend have been made as per planned work under MDAnalysis' CZI EOSS4 grant. Further details about these will be provided in a future blog post.

- `MDAnalysis.lib.distances` now accepts `AtomGroups` as well as NumPy arrays (PR #3730).
- Timestep has been converted to a Cython Extension type (PR #3683).

Release 2.2.0 of MDAnalysis

In line with NEP29, this version of MDAnalysis drops support for Python 3.7 and raises the minimum NumPy version to 1.19.0. Minimum version support has also been changed for the following packages; `networkx>=2.0`, `scipy>=1.5.0`, `gsd>=1.9.3`. Further details on MDAnalysis future support strategy and NEP29 will be released shortly.

Supported python versions:

- 3.8, 3.9, 3.10

Major changes:

See the CHANGELOG and our release blog post for more changes and details.

Enhancements:

- The `frames` argument was added to AnalysisBase-derived classes (i.e. modern analysis classes) allowing for specific frames to be defined when running an analysis. (PR #3415)
- DL_POLY classic HISTORY files are now supported (Issue #3678)
- Python wheels are now made available through PyPI for x86_64 architectures (Issue #1300, PR #3680)
- Added a `center_of_charge` attribute for AtomGroups (PR #3671)
- LinearDensity now work with UpdatingAtomGroups (Issue #2508, PR #3617)
- Addition of a PCA transformation and an associated inverse-PCA transformation was added to the PCA analysis class (PR #3596, Issue #2703)
- Major improvements to the RDKitConverter's accuracy (PR #3044)
  - Accuracy of 99.14% when benchmarked against ChEMBL30
  - AtomGroups containing monatomic ion charges and edge cases with nitrogen, sulfur, phosphorus and conjugated systems should now have correctly assigned bond orders and charges.
- Addition of a new AnalysisBase derived Watson-Crick distance analysis class (PR #3611)

Fixes:

- Fixed issues where calling the `copy` method of Readers did not preserve optional arguments (Issue #3664, PR #3685)
- Fixed several issues where iterating trajectories had undefined behaviour
  - Iterating (not in memory) SingleFrame readers now reset modified trajectory attributes (Issue #3423)
  - Iterating using defined indices did not rewind the trajectory (Issue #3416)
- Fixed issues with competing processes writing to an XTC offset file leading to offset corruption (Issue #1988, PR #3375)
- Fixed issue preventing OpenMMTopologyParsers from parsing systems with missing elements (Issue #3317, PR #3511)
- Fixed issue with encore.covariance.covariance_matrix not working when providing an external reference (Issue #3539, PR #3621)
- Fixed issue with broken code paths for "residues" and "segment" groupings for LinearDensity (Issue #3571, PR #3572)
- Improved the flexibility of MOL2 reading, allowing for optional columns (`subst_id`, `subst_name` and `charge`) not to be provided (Issue #3385, PR #3598)
- Fixed several issues related to converting AtomGroups to RDKit molecules (PR #3044):
  - Atoms are now in the same order
  - `atom.GetMonomerInfor().GetName()` now follows the guidelines for PDB files
  - Using `NoImplicit=False` no longer throws a `SanitizationError`

- Fixed issues with incorrect reading of triclinic boxes from DUMP files (Issue #3386, PR #3403)
- Fixed issue with the BAT method modifying input coordinate data (Issue #3501)

Changes:

- The number of matches allowed when doing a smarts selection has been increased from the default 1000 to max(1000, n_atoms * 10), an additional set of `smarts_kwargs` can now also be passed to override this behaviour (Issue #3469, PR #3470)
- The `fasteners` package is now a core dependency (PR #3375)
- LinearDensity now saves the histogram bin edges for easier plotting as `hist_bin_edges for each dimension in the results dictionary (Issue #2508, PR #3617)
- ContactAnalysis now accepts AtomGroups (Issue #2666, PR #3565)

Deprecations:

- The following results attribute for LinearDensity are now deprecated: (Issue #2508, PR #3617)
  - `pos` is now `mass_density`
  - `char` is now `charge_density`
  - `std` entries are now `stddev`

Known test failures:

- Windows builds
  - In some cases users may get permission errors with tests involving symlinks. This should not impact the behaviour of MDAnalysis but may impact the creation of temporary files when using HOLE2 (see: https://github.com/MDAnalysis/mdanalysis/issues/3556).

Release 2.1.0 of MDAnalysis

In line with ongoing attempts to align with NEP29, this version of MDAnalysis drops support for Python 3.6 and raises the minimum NumPy version to 1.18.0.

Please note that at time of release whilst all the MDAnalysis core functionality supports Python 3.10, some optional modules do not due to a lack of support by dependencies which they require. We hope that this support will gradually be added as more of these dependencies release new versions compatible with Python 3.10.

Supported python versions:

- 3.7, 3.8, 3.9, 3.10

Major changes:

See the CHANGELOG and our release blog post for more changes and details.

Enhancements:

- Addition of a new dielectric analysis module (PR #2118)
- The TPR parser now supports reading files from GROMACS 2022 (PR #3514)
- The H5MDReader can now load trajectories without a topology (PR #3466)
- Custom compiler flags can be used when building MDAnalysis from source (PR #3429)
- The RDKit reader now supports parsing R/S chirality (PR #3445)
- A new method to apply the minimum image convention to a collection of vectors, `minimize_vectors`, has been introduced (PR #3472)

Fixes:

- Fixed various integer overflow issues in the distance calculation backend of MDAnalysis which would prevent calculations on large systems (Issues #3183, #3512).
- Fixed issues with the creation of VMD surfaces in HOLE2 when using a non-contiguous start/stop/step.
- Fixes reading of charges with the ITPParser (Issue #3419).
- Fixed issue with the creation of a Universe from a custom object which only provides a topology (Issue #3443).
- Fixed issue with accessing newly created values added via `add_Segment` or `add_Residue` (Issue #3437).

Changes:

- `packaging` is now a core dependency of MDAnalysis.
- Indexing a Group (AtomGroup, ResidueGroup, SegmentGroup) with `None` now raises a `TypeError`. Prior to this indexing by `None` would incorrectly return the whole Group but claim to have a length of 1 atom (Issue #3092).
- The TRZReader now defaults to a `dt` value of 1.0 ps instead of the previous 0.0 ps (Issue #3257).

Deprecations:

- The `pbc` keyword argument for various Group methods has been deprecated in favor of `wrap`. The deprecated keyword will be removed in version 3.0.0 (Issue #1760).

Known test failures:

- `pytest-xdist` and more than 4 workers
  - Under these conditions a test related to logging for HydrogenBondAnalysis can fail. This is not thought to impact the validity of MDAnalysis. See here for more details: https://github.com/MDAnalysis/mdanalysis/issues/3543
- Windows builds
  - In some cases users may get permission errors with tests involving symlinks. This should not impact the behaviour of MDAnalysis but may impact the creation of temporary files when using HOLE2 (see: https://github.com/MDAnalysis/mdanalysis/issues/3556).

Release 2.0.0 of MDAnalysis

This is the first version of MDAnalysis to solely support python 3.6+

Supported python versions:

- 3.6, 3.7, 3.8, 3.9

Please note that starting with the next minor version, MDAnalysis will be following NEP29.

Notes:

- This is a major release and introduces major advertised API breaks. Caution is advised when upgrading to 2.0.0.

Major changes:

Enhancements:

- LAMMPSDumpReader can now read coordinates in all different LAMMPS coordinate conventions (Issue #3358)
- New `Results` class for storing analysis results (Issue #3115)
- New OpenMM coordinate and topology converters (Issue #2863, PR #2917)
- New `intra_bonds`,`intra_angles`, `intra_dihedrals`, etc… methods to return connections involve atoms within AtomGroups instead of including atoms outside of it (Issue #1264, #2821, PR #3200)
- Support for Groamcs 2021 TPR files (Issue #3180)
- Adds preliminary support for ppc64le and aarch64 [ARM] (Issue #3127, PR #2956 #3149)
- New selection operators (Issue #3054, PR #2927)
- New refactor of helix analysis class as `analysis.helix_analysis` (Issue #2452)
- New converter between RDKit molecules and MDAnalysis AtomGroup objects (Issue #2468). Also includes `from_smiles` Universe generator method, and the aromatic and smarts selection.
- New analysis method for calculating Mean Squared Dsiplacements (Issue #2438)
- New converter between Cartesian and Bond-Angle-Torsion coordinates (PR #2668)
- Universes and readers can now be `pickled` paving the way to easier parallel analyses (Issue #2723)
- New H5MDReader and H5MDWriter (Issue #762, #2866)

Fixes:

- Fixes an issue where `select_atom`, AtomGroup.unique, ResidueGroup.unique, and SegmentGroup.unique did not sort the output atoms (Issues #3364 #2977)
- GRO files now only support unit cells defined with 3 or 9 entries (Issue #3305)
- Fixes the sometimes wrong sorting of atoms into fragments when unwrapping (Issue #3352)
- Fixes issue when atttempting to use/pass mean positions to PCA analysis (Issue #2728)
- Fixes support for DL_POLY HISTORY files that contain cell information even if there are no periodic boundary conditions (Issue #3314)
- Fixes issue with WaterBridgeAnalysis double counting waters (Issue #3119)
- PDBWriter will use chainID instead of segID (Issue #3144)
- PDBParser and PDBWriter now assign and use the element attribute (Issues #3030 #2422)
- AtomGroup.center now works correctly for compounds + unwrapping (Issue #2984)
- Documents and fixes the `density` keyword for `rdf.InterRDF_s` (Isuse #2811)
- Fixed Janin analysis residue filtering, including CYSH (Issue #2898)

Changes:

- New converter API for all MDAnalysis converters under MDAnalysis.converters
- Timestep now stores information in ‘C’ memory layout instead of the previous ‘F’ default (PR #1738)
- `hbonds.hbond_analysis` has been remove din favour of `hydrogenbonds.hbond_analysis` (Issues #2739, #2746)
- TPRParser now loads TPR files with `tpr_resid_from_one=True` by deafult, which starts TPR resid indexing from 1 (instead of 0 as in previous MDAnalysis versions) (Issue #2364, PR #3152)
- `analysis.hole` has now been removed in favour of `analysis.hole2.hole` (Issue #2739)
- `Writer.write(Timestep)` and `Writer.write_next_timestep` have been removed. Please use `write()` instead (Issue #2739)
- Removes deprecated density_from_Universe, density_from_PDB, Bfactor2RMSF, and notwithin_coordinates_factory from MDAnalysis.analysis.density (Issue #2739)
- Changes the minimum numpy supported version to 1.16.0 (Issue #2827)
- Removes deprecated `waterdynamics.HydrogenBondLifetimes` (PR #2842)
- `hbonds.WaterBridgeAnalysis` has been moved to `hydrogenbonds.WaterBridgeAnalysis` (Issue #2739 PR #2913)

Deprecations:

- The bfactors attribute is now aliased to tempfactors and will be removed in 3.0.0 (Issue #1901)
- WaterBridgeAnalysis.generate_table() now returns table information, with the table attribute being deprecated
- Various analysis result attributes which are now stored in Results will be deprecated in 3.0.0 (Issue #3261)
- In 3.0.0 the ParmEd classes will only be accessible from the MDAnalysis.converters module
- In 2.1.0 the TRZReader will default to a dt of 1.0 ps when failing to read it from the input TRZ trajectory

See the CHANGELOG for more changes and details.

Known issues:

- Windows builds
  - For some compilers (seen on MVC v.19xx), differences in floating point precision leads to PBC wrapping differing from expected outcomes. This leads to failures in the MDAnalysisTests.lib.test_augment tests. To our knowledge this does not significantly affect results (as all other tests pass). We will aim to fix this in version 2.1.0.

2.1.6 Universe

> *If you wish to make an apple pie from scratch, you must first invent the universe.*
>
> —Carl Sagan, Cosmos

MDAnalysis is structured around two fundamental classes: the Universe and the AtomGroup. Almost all code in MDAnalysis begins with Universe, which contains all the information describing a molecular dynamics system.

It has two key properties:

- atoms: an AtomGroup of the system's atoms, providing access to important analysis methods (described below)
- trajectory: the currently loaded trajectory reader

A Universe ties the static information from the "topology" (e.g. atom identities) to dynamically updating information from the "trajectory" (e.g. coordinates). A key feature of MDAnalysis is that an entire trajectory is not loaded into memory (unless the user explicitly does so with MemoryReader). Instead, the trajectory attribute provides a view on a specific frame of the trajectory. This allows the analysis of arbitrarily long trajectories without a significant impact on memory.

Creating a Universe

Loading from files

A Universe is typically created from a "topology" file, with optional "trajectory" file/s. Trajectory files must have the coordinates in the same order as atoms in the topology. See *Formats* for the topology and trajectory formats supported by MDAnalysis, and how to load each specific format.

```
u = Universe(topology, trajectory)
u = Universe(pdbfile)                       # read atoms and coordinates from PDB or GRO
u = Universe(topology, [traj1, traj2, ...]) # read from a list of trajectories
u = Universe(topology, traj1, traj2, ...)   # read from multiple trajectories
```

The line between topology and trajectory files is quite blurry. For example, a PDB or GRO file is considered both a topology and a trajectory file. The difference is that a **topology file** provides static information, such as atom identities (name, mass, etc.), charges, and bond connectivity. A **trajectory file** provides dynamic information, such as coordinates, velocities, forces, and box dimensions.

If only a single file is provided, MDAnalysis tries to read both topology and trajectory information from it. When multiple trajectory files are provided, coordinates are loaded in the order given.

The default arguments should create a Universe suited for most analysis applications. However, the `Universe` constructor also takes optional arguments.

The following options specify how to treat the input:

- `format`: the file format of the trajectory file/s. (default: None, formats are guessed)
- `topology_format`: the file format of the topology file. (default: None, formats are guessed)
- `all_coordinates`: whether to read coordinate information from the first file (default: False. Ignored when only one file is provided)
- `continuous`: whether to give multiple trajectory files continuous time steps. This is currently only supported for XTC/TRR trajectories with a GRO/TPR topology, following the behaviour of gmx trjcat (default: False.)

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PDB, GRO, XTC

In [3]: u1 = mda.Universe(GRO, XTC, XTC, all_coordinates=True)

In [4]: u1.trajectory
Out[4]: <ChainReader containing adk_oplsaa.gro, adk_oplsaa.xtc, adk_oplsaa.xtc with 21␣
↪frames of 47681 atoms>

In [5]: u2 = mda.Universe(GRO, XTC, XTC, all_coordinates=False, continuous=False)

In [6]: print([int(ts.time) for ts in u2.trajectory])
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 0, 100, 200, 300, 400, 500, 600, 700,␣
↪800, 900]
```

The following options modify the created Universe:

- `guess_bonds`: whether to guess connectivity between atoms. (default: False)
- `vdwradii`: a dictionary of `{element: radius}` of van der Waals' radii for use in guessing bonds.
- `transformations`: a function or list of functions for on-the-fly trajectory transformation.
- `in_memory`: whether to load coordinates into memory (default: False)
- `in_memory_step`: only read every nth frame into an in-memory representation. (default: 1)
- `is_anchor`: whether to consider this Universe when unpickling `AtomGroup`s (default: True)
- `anchor_name`: the name of this Universe when unpickling `AtomGroup`s (default: None, automatically generated)

You can also pass in keywords for parsing the topology or coordinates. For example, many file formats do not specify the timestep for their trajectory. In these cases, MDAnalysis assumes that the default timestep is 1 ps. If this is incorrect, you can pass in a `dt` argument to modify the timestep. **This does not modify timesteps for formats that include time information.**

```
In [7]: from MDAnalysis.tests.datafiles import PRM, TRJ

In [8]: default_timestep = mda.Universe(PRM, TRJ)

In [9]: default_timestep.trajectory.dt
Out[9]: 1.0

In [10]: user_timestep = mda.Universe(PRM, TRJ, dt=5)  # ps

In [11]: user_timestep.trajectory.dt
Out[11]: 5
```

Constructing from AtomGroups

A new Universe can be created from one or more `AtomGroup` instances with `Merge()`. The `AtomGroup` instances can come from different Universes, meaning that this is one way to concatenate selections from different datasets.

For example, to combine a protein, ligand, and solvent from separate PDB files:

```
u1 = mda.Universe("protein.pdb")
u2 = mda.Universe("ligand.pdb")
u3 = mda.Universe("solvent.pdb")
u = Merge(u1.select_atoms("protein"), u2.atoms, u3.atoms)
u.atoms.write("system.pdb")
```

Constructing from scratch

A Universe can be constructed from scratch with `Universe.empty`. There are three stages to this process:

1. Create the blank Universe with specified number of atoms. If coordinates, set `trajectory=True`.
2. Add topology attributes such as atom names.
3. (Optional) Load coordinates.

For example, to construct a universe with 6 atoms in 2 residues:

```
In [12]: u = mda.Universe.empty(6, 2, atom_resindex=[0, 0, 0, 1, 1, 1], trajectory=True)

In [13]: u.add_TopologyAttr('masses')

In [14]: coordinates = np.empty((1000,  # number of frames
   ....:                         u.atoms.n_atoms,
   ....:                         3))
   ....:

In [15]: u.load_new(coordinates, order='fac')
Out[15]: <Universe with 6 atoms>
```

See this notebook tutorial for more information.

Guessing topology attributes

MDAnalysis can guess two kinds of information. Sometimes MDAnalysis guesses information instead of reading it from certain file formats, which can lead to mistakes such as assigning atoms the wrong element or charge. See *the available topology parsers* for a case-by-case breakdown of which atom properties MDAnalysis guesses for each format. See *Guessing* for how attributes are guessed, and *Default values and attribute levels* for which attributes have default values.

Universe properties and methods

A Universe holds master groups of atoms and topology objects:

- `atoms`: all Atoms in the system, in an *AtomGroup*.
- `residues`: all Residues in the system
- `segments`: all Segments in the system
- `bonds`: all bond TopologyObjects in the system
- `angles`: all angle TopologyObjects in the system
- `dihedrals`: all dihedral TopologyObjects in the system
- `impropers`: all improper TopologyObjects in the system

Residues and Segments are chemically meaningful groups of Atoms.

Modifying a topology is typically done through the `Universe`, which contains several methods for adding properties:

- `add_TopologyAttr()`
- `add_Residue()`
- `add_Segment()`

See *Topology attributes* for more information on which topology attributes can be added, and *examples/constructing_universe.ipynb* for examples on adding attributes and Segments.

2.1.7 AtomGroup

A `Universe` contains all particles in the molecular system. MDAnalysis calls a particle an `Atom`, regardless of whether it really is (e.g. it may be a united-atom particle or coarse-grained bead). `Atom`s are grouped with an `AtomGroup`; the 'master' `AtomGroup` of a Universe is accessible at `Universe.atoms`.

Note: The `AtomGroup` is probably the most important object in MDAnalysis. Virtually everything can be accessed through an `AtomGroup`.

Creating an AtomGroup

Atom selection language

AtomGroup instances are typically created with Universe.select_atoms or by manipulating another AtomGroup, e.g. by slicing.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PDB

In [3]: u = mda.Universe(PDB)

In [4]: u.select_atoms('resname ARG')
Out[4]: <AtomGroup with 312 atoms>
```

See *Atom selection language* for more information.

Indexing and slicing

An AtomGroup can be indexed and sliced like a list:

```
In [5]: print(u.atoms[0])
<Atom 1: N of type N of resname MET, resid 1 and segid SYSTEM and altLoc >
```

Slicing returns another AtomGroup. The below code returns an AtomGroup of every second element from the first to the 6th element, corresponding to indices 0, 2, and 4.

```
In [6]: ag = u.atoms[0:6:2]

In [7]: ag.indices
Out[7]: array([0, 2, 4])
```

MDAnalysis also supports fancy indexing: passing a ndarray or a list.

```
In [8]: indices = [0, 3, -1, 10, 3]

In [9]: u.atoms[indices].indices
Out[9]: array([    0,     3, 47680,    10,     3])
```

Boolean indexing allows you to pass in an array of True or False values to create a new AtomGroup from another. The array must be the same length as the original AtomGroup. This allows you to select atoms on conditions.

```
In [10]: arr = u.atoms.resnames == 'ARG'

In [11]: len(arr) == len(u.atoms)

In [12]: arr
Out[12]: Out[11]: array([False, False, False, ..., False, False, False])

In [13]: u.atoms[arr]
```

Group operators and set methods

MDAnalysis supports a number of ways to compare AtomGroups or construct a new one: group operators (e.g. concatenate(), subtract()) and set methods (e.g. union(), difference()). Group operators achieve a similar outcome to set methods. However, a key difference is that concatenate() and subtract() preserve the order of the atoms and any duplicates. union() and difference() return an AtomGroup where each atom is unique, and ordered by its topology index.

```
In [14]: ag1 = u.atoms[1:6]

In [15]: ag2 = u.atoms[8:3:-1]

In [16]: concat = ag1 + ag2

In [17]: concat.indices
Out[17]: array([1, 2, 3, 4, 5, 8, 7, 6, 5, 4])

In [18]: union = ag1 | ag2

In [19]: union.indices
Out[19]: array([1, 2, 3, 4, 5, 6, 7, 8])
```

Available operators

Unlike set methods and atom selection language, concatenation and subtraction keep the order of the atoms as well as duplicates.

Operation	Equivalent	Result
`len(s)`		number of atoms in the group
`s == t`		test if `s` and `t` contain the same elements in the same order
`s.concatenate(t)`	`s + t`	new Group with elements from `s` and from `t`
`s.subtract(t)`		new Group with elements from `s` that are not in `t`

Available set methods

Each of these methods create groups that are sorted sets of unique Atoms.

Operation	Equivalent	Result
`s.isdisjoint(t)`		`True` if `s` and `t` do not share elements
`s.issubset(t)`		test if all elements of `s` are part of `t`
`s.is_strict_subset(t)`		test if all elements of `s` are part of `t`, and `s != t`
`s.issuperset(t)`		test if all elements of `t` are part of `s`
`s.is_strict_superset(t)`		test if all elements of `t` are part of `s`, and `s != t`
`s.union(t)`	`s | t`	new Group with elements from both `s` and `t`
`s.intersection(t)`	`s & t`	new Group with elements common to `s` and `t`
`s.difference(t)`	`s - t`	new Group with elements of `s` that are not in `t`
`s.symmetric_difference(t)`	`s ^ t`	new Group with elements that are part of `s` or `t` but not both

Groupby and split

An AtomGroup can be constructed from another by separating atoms by properties.

AtomGroup.split can create a list of AtomGroups by splitting another AtomGroup by the 'level' of connectivity: one of *atom*, *residue*, *molecule*, or *segment*.

```
In [20]: ag1 = u.atoms[:100]

In [21]: ag1.split('residue')
Out[21]:
[<AtomGroup with 19 atoms>,
 <AtomGroup with 24 atoms>,
 <AtomGroup with 19 atoms>,
 <AtomGroup with 19 atoms>,
 <AtomGroup with 19 atoms>]
```

An AtomGroup can also be separated according to values of *topology attributes* to produce a dictionary of {value: AtomGroup}.

```
In [22]: u.atoms.groupby('masses')
Out[22]:
{32.06: <AtomGroup with 7 atoms>,
 1.008: <AtomGroup with 23853 atoms>,
 0.0: <AtomGroup with 11084 atoms>,
 12.011: <AtomGroup with 1040 atoms>,
 14.007: <AtomGroup with 289 atoms>,
 15.999: <AtomGroup with 11404 atoms>,
 22.98977: <AtomGroup with 4 atoms>}
```

Passing in multiple attributes groups them in order:

```
In [23]: u.select_atoms('resname SOL NA+').groupby(['masses', 'resnames'])
Out[23]:
{(0.0, 'SOL'): <AtomGroup with 11084 atoms>,
 (1.008, 'SOL'): <AtomGroup with 22168 atoms>,
 (22.98977, 'NA+'): <AtomGroup with 4 atoms>,
 (15.999, 'SOL'): <AtomGroup with 11084 atoms>}
```

Constructing from Atoms

An AtomGroup can be created from an iterable of Atom instances:

```
In [24]: atom1 = u.atoms[4]

In [25]: atom2 = u.atoms[6]

In [26]: atom3 = u.atoms[2]

In [27]: ag = mda.AtomGroup([atom1, atom2, atom3])

In [28]: print(ag)
<AtomGroup [<Atom 5: CA of type C of resname MET, resid 1 and segid SYSTEM and altLoc >,
```

(continues on next page)

(continued from previous page)

```
↪<Atom 7: CB of type C of resname MET, resid 1 and segid SYSTEM and altLoc >, <Atom 3:␣
↪H2 of type H of resname MET, resid 1 and segid SYSTEM and altLoc >]>
```

A neat shortcut for this is to simply add an Atom to another Atom or AtomGroup:

```
In [29]: ag = atom1 + atom2

In [30]: print(ag)
<AtomGroup [<Atom 5: CA of type C of resname MET, resid 1 and segid SYSTEM and altLoc >,
↪<Atom 7: CB of type C of resname MET, resid 1 and segid SYSTEM and altLoc >]>

In [31]: ag += atom3

In [32]: print(ag)
<AtomGroup [<Atom 5: CA of type C of resname MET, resid 1 and segid SYSTEM and altLoc >,
↪<Atom 7: CB of type C of resname MET, resid 1 and segid SYSTEM and altLoc >, <Atom 3:␣
↪H2 of type H of resname MET, resid 1 and segid SYSTEM and altLoc >]>
```

An alternative method is to provide a list of indices and the Universe that the Atoms belong to:

```
In [33]: ag = mda.AtomGroup([4, 6, 2], u)

In [34]: print(ag)
<AtomGroup [<Atom 5: CA of type C of resname MET, resid 1 and segid SYSTEM and altLoc >,
↪<Atom 7: CB of type C of resname MET, resid 1 and segid SYSTEM and altLoc >, <Atom 3:␣
↪H2 of type H of resname MET, resid 1 and segid SYSTEM and altLoc >]>
```

Order and uniqueness

These methods of creating an AtomGroup result in a sorted, unique list of atoms:

- Atom selection language
- Slicing
- Boolean indexing
- Set methods
- AtomGroup.split and AtomGroup.groupby

These methods return a user-ordered AtomGroup that can contain duplicates:

- Fancy indexing (with arrays or lists)
- Group operations (AtomGroup.concatenate and AtomGroup.subtract)
- Constructing directly from Atoms

Empty AtomGroups

MDAnalysis can also work with empty AtomGroups:

```
In [35]: null = u.atoms[[]]

In [36]: null
Out[36]: <AtomGroup with 0 atoms>
```

The above is the same as creating an `AtomGroup` from an empty list and a `Universe`.

```
In [37]: mda.AtomGroup([], u)
Out[37]: <AtomGroup with 0 atoms>
```

Each method of creating an AtomGroup can also be used to create an empty one. For example, using selection language:

```
In [38]: u.select_atoms("resname DOES_NOT_EXIST")
Out[38]: <AtomGroup with 0 atoms>
```

and indexing:

```
In [39]: u.atoms[6:6]
Out[39]: <AtomGroup with 0 atoms>
```

or set methods:

```
In [40]: u.atoms - u.atoms
Out[40]: <AtomGroup with 0 atoms>
```

Empty AtomGroups have a length of 0 and evaluate to `False` in a boolean context.

```
In [41]: bool(null)
Out[41]: False
```

This allows analysis methods to skip over empty AtomGroups instead of raising an error, which is helpful as occasionally empty AtomGroups can arise from selection logic that is too restrictive (e.g. *geometric selections*).

Dynamically updating AtomGroups

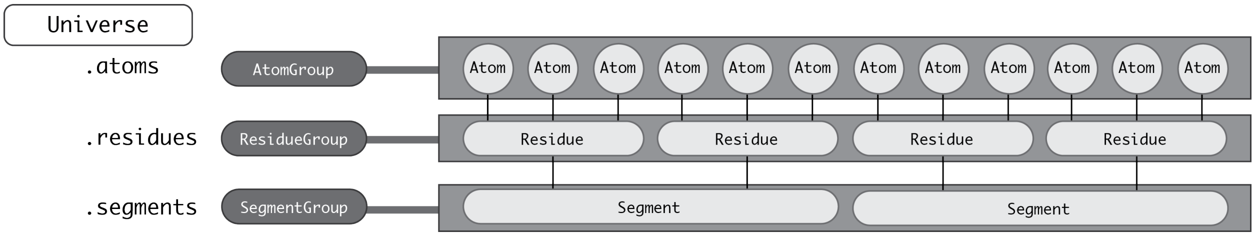
A normal AtomGroup is static, and the atoms within it do not change as the trajectory frame changes. Several methods require dynamically updating AtomGroups. These are typically created using atom selection language. See *Dynamic selections* for more information.

Methods

Most of the analysis functionality in MDAnalysis is implemented in *the analysis module*, but many interesting methods can be accessed from an `AtomGroup` directly. For example, Bonds, Angles, Dihedrals and ImproperDihedrals *can be created from AtomGroups*. Providing that required topology attributes are present, *a number of analysis methods are also available* to a `AtomGroup`, `ResidueGroup`, and `SegmentGroup`.

2.1.8 Groups of atoms

MDAnalysis has a hierarchy of `Atom` containers that are used throughout the code.



First and foremost is the `AtomGroup`. An `AtomGroup` is the primary `Atom` container; virtually everything can be accessed through it, as detailed in *AtomGroup*. This includes chemically meaningful groups of `Atom`s such as a `Residue` or a `Segment`.

Residues and Segments

A `Residue` is composed of `Atom`s, and a `Segment` is composed of `Residue`s.

The corresponding container groups are `ResidueGroup` and `SegmentGroup`. These have similar properties and available methods as `AtomGroup`.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import TPR, XTC

In [3]: u = mda.Universe(TPR, XTC)

In [4]: ag = u.atoms.select_atoms('resname ARG and name CA')

In [5]: ag
Out[5]: <AtomGroup with 13 atoms>
```

Each of these container groups can be accessed through another. The behaviour of this differs by level. For example, the residues of the `ag` are the residues that the atoms of `ag` belong to.

```
In [6]: ag.residues
Out[6]: <ResidueGroup with 13 residues>
```

Accessing the atoms of *those* residues, however, returns *all* the atoms in the residues, not just those originally in `ag`.

```
In [7]: ag.residues.atoms
Out[7]: <AtomGroup with 312 atoms>
```

The same applies to segments.

```
In [8]: ag[:3].segments.atoms
Out[8]: <AtomGroup with 3341 atoms>
```

Similarly, an `Atom` has direct knowledge of the `Residue` and `Segment` it belongs to. Note that an `Atom` belongs to *one* `Residue` and the residue belongs to *one* `Segment`, but a `Segment` has multiple residues.

```
In [9]: a = u.atoms[0]

In [10]: a.residue
Out[10]: <Residue LYSH, 0>

In [11]: a.residue.segment
Out[11]: <Segment seg_0_Protein_A>

In [12]: a.residue.segment.residues
Out[12]: <ResidueGroup with 129 residues>
```

For information on adding custom Residues or Segments, have a look at *Adding a Residue or Segment to a Universe*.

Access to other classes via the `AtomGroup` object can be pretty powerful, but also needs to be used with caution to avoid accessing data outside the intended selection. Therefore, we present two use cases showing commonly used applications, for which we define `Universe` on a simple extract from the PDB file:

```
In [9]: import MDAnalysis as mda

In [10]: import io

In [11]: pdb = io.StringIO("""
   ....: ATOM    414  N   GLY A 402     -51.919   9.578 -14.287  1.00 68.46           N
   ....: ATOM    415  CA  GLY A 402     -52.405  10.954 -14.168  1.00 68.41           C
   ....: ATOM    416  C   GLY A 402     -51.821  11.946 -15.164  1.00 69.71           C
   ....: ATOM    417  O   GLY A 402     -51.958  13.159 -14.968  1.00 69.61           O
   ....: ATOM    418  H   GLY A 402     -52.551   8.935 -14.743  1.00  0.00           H
   ....: ATOM    419  HA3 GLY A 402     -52.225  11.313 -13.155  1.00  0.00           H
   ....: ATOM    420  HA2 GLY A 402     -53.492  10.960 -14.249  1.00  0.00           H
   ....: TER
   ....: HETATM 1929  N1  XYZ A 900     -40.275  19.399 -28.239  1.00  0.00           N1+
   ....: TER
   ....: ATOM   1029  N   ALA B 122     -25.408  19.612 -13.814  1.00 37.52           N
   ....: ATOM   1030  CA  ALA B 122     -26.529  20.537 -14.038  1.00 37.70           C
   ....: ATOM   1031  C   ALA B 122     -26.386  21.914 -13.374  1.00 45.35           C
   ....: ATOM   1032  O   ALA B 122     -26.885  22.904 -13.918  1.00 48.34           O
   ....: ATOM   1033  CB  ALA B 122     -27.835  19.889 -13.613  1.00 37.94           C
   ....: ATOM   1034  H   ALA B 122     -25.636  18.727 -13.385  1.00  0.00           H
   ....: ATOM   1035  HA  ALA B 122     -26.592  20.707 -15.113  1.00  0.00           H
   ....: ATOM   1036  HB1 ALA B 122     -28.658  20.583 -13.783  1.00  0.00           H
   ....: ATOM   1037  HB2 ALA B 122     -27.998  18.983 -14.196  1.00  0.00           H
   ....: ATOM   1038  HB3 ALA B 122     -27.788  19.635 -12.554  1.00  0.00           H
   ....: ATOM   1039  N   GLY B 123     -25.713  21.969 -12.223  1.00 41.18           N
   ....: ATOM   1040  CA  GLY B 123     -25.550  23.204 -11.460  1.00 41.40           C
   ....: ATOM   1041  C   GLY B 123     -24.309  24.018 -11.745  1.00 45.74           C
   ....: ATOM   1042  O   GLY B 123     -24.349  25.234 -11.601  1.00 46.81           O
   ....: ATOM   1043  H   GLY B 123     -25.290  21.133 -11.845  1.00  0.00           H
   ....: ATOM   1044  HA3 GLY B 123     -25.593  22.976 -10.395  1.00  0.00           H
   ....: ATOM   1045  HA2 GLY B 123     -26.430  23.831 -11.600  1.00  0.00           H
   ....: TER
   ....: """)
   ....:

In [12]: u = mda.Universe(pdb, format="PDB")
```

Use case: Sequence of residues by segment

In order to select only ATOM record types and get a list of residues by segment, one needs to call:

```
In [13]: residues_by_seg = list()

In [14]: for seg in u.segments:
   ....:     p_seg = seg.atoms.select_atoms("record_type ATOM")
   ....:     residues_by_seg.append(p_seg.residues)
   ....:
```

Residue names can be extracted using Python's list comprehensions. As required, HETATM record type lines are not considered:

```
In [15]: [rg.resnames for rg in residues_by_seg]
Out[15]: [array(['GLY'], dtype=object), array(['ALA', 'GLY'], dtype=object)]
```

Note that accessing residues by first selecting the segments of an `AtomGroup` returns all the residues in that segment for both the ATOM and HETATM record types (no memory of the original selection). The meaning of this is: "give me all residue names from segments in which there is at least one of the selected atoms".

```
In [16]: selected_atoms = u.select_atoms("record_type ATOM")

In [17]: all_residues = selected_atoms.segments.residues

In [18]: all_residues.resnames
Out[18]: array(['GLY', 'XYZ', 'ALA', 'GLY'], dtype=object)
```

Use case: Atoms list grouped by residues

In order to list all the heavy protein backbone and sidechain atoms in every residue, one needs to call:

```
In [19]: atoms_in_residues = list()

In [20]: for seg in u.segments:
   ....:     p_seg = seg.atoms.select_atoms("record_type ATOM and not name H*")
   ....:     for p_res in p_seg.residues:
   ....:         atoms_in_residues.append(p_seg.select_atoms(f"resid {p_res.resid} and␣
↪resname {p_res.resname}"))
   ....:
```

Atom names can be extracted using Python's list comprehensions. As required, HETATM record type lines and hydrogen atoms are not considered:

```
In [21]: [ag.names for ag in atoms_in_residues]
Out[21]: 
[array(['N', 'CA', 'C', 'O'], dtype=object),
 array(['N', 'CA', 'C', 'O', 'CB'], dtype=object),
 array(['N', 'CA', 'C', 'O'], dtype=object)]
```

The Python syntax can be further simplified by using `split()` function:

```
In [22]: rds = u.select_atoms("record_type ATOM and not name H*").split("residue")

In [23]: [ag.names for ag in rds]
Out[23]: 
[array(['N', 'CA', 'C', 'O'], dtype=object),
 array(['N', 'CA', 'C', 'O', 'CB'], dtype=object),
 array(['N', 'CA', 'C', 'O'], dtype=object)]
```

Note that accessing atoms by first selecting the residues of an `AtomGroup` also returns hydrogen atoms (no memory of the original selection). The meaning of this is "give me all atom names from residues in which there is at least one of the selected atoms". However, it doesn't contain a nitrogen atom from XYZ residue as no atoms from this residue were in the `AtomGroup`.

```
In [24]: all_atoms_in_residues = list()

In [25]: for seg in u.segments:
   ....:     p_seg = seg.atoms.select_atoms("record_type ATOM and not name H*")
   ....:     all_atoms_in_residues.append(p_seg.residues)
   ....: 

In [26]: [atom.names for atom in all_atoms_in_residues]
Out[26]: 
[[array(['N', 'CA', 'C', 'O', 'H', 'HA3', 'HA2'], dtype=object)],
 [array(['N', 'CA', 'C', 'O', 'CB', 'H', 'HA', 'HB1', 'HB2', 'HB3'],
        dtype=object),
  array(['N', 'CA', 'C', 'O', 'H', 'HA3', 'HA2'], dtype=object)]]
```

Fragments

Certain analysis methods in MDAnalysis also make use of additional ways to group atoms. A key concept is a fragment. A fragment is what is typically considered a molecule: an AtomGroup where any atom is reachable from any other atom in the AtomGroup by traversing bonds, and none of its atoms is bonded to any atoms outside the AtomGroup. (A 'molecule' in MDAnalysis methods *refers to a GROMACS-specific concept*). The fragments of a Universe are determined by MDAnalysis as a derived quantity. They can only be determined if bond information is available.

The fragments of an `AtomGroup` are accessible via the `fragments` property. Below is a Universe from a GROMACS TPR file of lysozyme (PDB ID: 2LYZ) with 101 water molecules. While it has 230 residues, there are only 102 fragments: 1 protein and 101 water fragments.

```
In [27]: from MDAnalysis.tests.datafiles import TPR2021

In [28]: u = mda.Universe(TPR2021)

In [29]: len(u.residues)
Out[29]: 230

In [30]: len(u.atoms.fragments)
Out[30]: 102
```

See *Topology objects* for more on bonds and which file formats give MDAnalysis bond information.

You can also look at which fragment a particular `Atom` belongs to:

```
In [31]: u.atoms[0].fragment  # first atom of lysozyme
Out[31]: <AtomGroup with 1960 atoms>
```

and see which fragments are associated with atoms in a smaller `AtomGroup`:

```
In [32]: u.atoms[1959:1961].fragments
Out[32]: (<AtomGroup with 1960 atoms>, <AtomGroup with 3 atoms>)
```

Note: `AtomGroup.fragments` returns a tuple of fragments with at least one `Atom` in the `AtomGroup`, not a tuple of fragments where *all* Atoms are in the `AtomGroup`.

2.1.9 Atom selection language

AtomGroups can be created by selecting atoms using the MDAnalysis atom selection language:

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PSF, DCD

In [3]: u = mda.Universe(PSF, DCD)

In [4]: ala = u.select_atoms('resname ALA')

In [5]: ala
Out[5]: <AtomGroup with 190 atoms>
```

The `select_atoms()` method of a `AtomGroup` or a `Universe` returns an `AtomGroup`. These two methods have different behaviour: while `Universe.select_atoms` operates on all the atoms in the universe, `AtomGroup.select_atoms` only operates on the atoms within the original AtomGroup. A single selection phrase always returns an `AtomGroup` with atoms sorted according to their index in the topology. This is to ensure that there are not any duplicates, which can happen with complicated selections. When order matters, *you can pass in multiple phrases*.

This page documents selection keywords and their arguments. `select_atoms()` also accepts keywords that modify the behaviour of the selection string and the resulting `AtomGroup` (documented further down this page). For example, you can:

- Pass in *named AtomGroups as arguments*:

```
In [6]: sph_6 = u.select_atoms("sphzone 6 protein")

In [7]: u.select_atoms("around 3 group sph_6", sph_6=sph_6)
Out[7]: <AtomGroup with 81 atoms>
```

- Turn off *periodic boundary conditions for geometric keywords* with `periodic=False`:

```
In [8]: u.select_atoms("around 6 protein", periodic=False)
Out[8]: <AtomGroup with 0 atoms>
```

- Create *dynamic UpdatingAtomGroups* with `updating=True`:

```
In [9]: u.select_atoms("prop x < 5 and prop y < 5 and prop z < 5", updating=True)
Out[9]: <AtomGroup with 917 atoms, with selection 'prop x < 5 and prop y < 5 and prop z
↪< 5' on the entire Universe.>
```

It is possible to export selections for external software packages with the help of *Selection exporters*.

Selection Keywords

The following describes all selection keywords currently understood by the selection parser. The following applies to all selections:

- Keywords are case sensitive.
- Atoms are automatically sequentially ordered in a resulting selection (see notes below on *Ordered selections* for how to circumvent this if necessary).
- Selections are parsed left to right and parentheses can be used for grouping. For example:

```
In [10]: u.select_atoms("segid DMPC and not (name H* or type OW)")
Out[10]: <AtomGroup with 0 atoms>
```

- String selections such as names and residue names can be matched with Unix shell-style wildcards. These rules include:
  - Using * in a string matches any number of any characters
  - ? matches any single character
  - `[seq]` matches any character in *seq*;
  - `[!seq]` matches any character not in *seq*
  - `[!?]` selects empty strings

  For example, the string `GL*` selects all strings that start with "GL", such as "GLU", "GLY", "GLX29", "GLN". `GL[YN]` will select all "GLY" and "GLN" strings. Any number of patterns can be included in the search. For more information on pattern matching, see the `fnmatch` documentation.

Simple selections

protein

Selects atoms that belong to a *hard-coded set of standard protein residue names*.

backbone

Selects the backbone atoms of a hard-coded set of protein residues. These atoms have the names: CA, C, O, N.

nucleic

Selects atoms that belong to a *hard-coded set of standard nucleic residue names*.

nucleicbackbone

Selects the backbone atoms of a hard-coded set of nucleic residues. These atoms have the names: P, O5', C5', C3', O3'

nucleicbase

Selects the atoms in *nucleobases*.

nucleicsugar

Selects the atoms in nucleic sugars. These have the names: C1', C2', C3', C4', O2', O4', O3'

segid ***seg-name***
select by segid (as given in the topology), e.g. `segid 4AKE` or `segid DMPC`

resid ***residue-number-range***
`resid` can take a single residue number or a range of numbers, followed by insertion codes. A range consists of two selections separated by a colon (inclusive) such as `resid 1A:1C`. This selects all residues with `resid==1` and `icode in ('A', 'B', 'C')`. A residue number ("resid") and icode is taken directly from the topology. Unlike `resnum`, `resid` is sensitive to insertion codes.

resnum ***residue-number-range***
`resnum` can take a single residue number or a range of numbers. A range consists of two numbers separated by a colon (inclusive) such as `resnum 1:5`. A residue number ("resnum") is taken directly from the topology. Unlike `resid`, `resnum` is insensitive to insertion codes.

resname ***residue-name***
select by residue name, e.g. `resname LYS`

name ***atom-name***
select by atom name (as given in the topology). Often, this is force field dependent. Example: `name CA` (for C-alpha atoms) or `name OW` (for SPC water oxygen)

type ***atom-type***
select by atom type; this is either a string or a number and depends on the force field; it is read from the topology file (e.g. the CHARMM PSF file contains numeric atom types). This uses the `Atom.type` *topology attribute*.

atom ***seg-name residue-number atom-name***
a selector for a single atom consisting of segid resid atomname, e.g. `DMPC 1 C2` selects the C2 carbon of the first residue of the DMPC segment

altloc ***alternative-location***
a selection for atoms where alternative locations are available, which is often the case with high-resolution crystal structures e.g. `resid 4 and resname ALA and altloc B` selects only the atoms of ALA-4 that have an altloc B record.

icode ***icode***
a selector for atoms where insertion codes are available. This can be combined with residue numbers using the `resid` selector above. e.g. `icode [!?]` selects atoms *without* insertion codes.

moltype ***molecule-type***
select by the `moltype` *topology attribute*, e.g. `moltype Protein_A`. At the moment, only the TPR format defines the `moltype`.

Boolean

not
all atoms not in the selection, e.g. `not protein` selects all atoms that aren't part of a protein

and
the intersection of two selections, i.e. the boolean and. e.g. `protein and not resname ALA` selects all atoms that belong to a protein but are not in an alanine residue

or
the union of two selections, i.e. the boolean or. e.g. `protein and not (resname ALA or resname LYS)` selects all atoms that belong to a protein, but are not in a lysine or alanine residue

Geometric

The geometric keywords below all implement periodic boundary conditions by default when valid cell dimensions are accessible from the Universe. This can be turned off by passing in the keyword `periodic=False`:

```
In [11]: u.select_atoms("around 6 protein", periodic=False)
Out[11]: <AtomGroup with 0 atoms>
```

around ***distance selection***

selects all atoms a certain cutoff away from another selection, e.g. `around 3.5 protein` selects all atoms not belonging to protein that are within 3.5 Angstroms from the protein

sphzone ***externalRadius selection***

selects all atoms within a spherical zone centered in the center of geometry (COG) of a given selection, e.g. `sphzone 6.0 ( protein and ( resid 130 or resid 80 ) )` selects the center of geometry of protein, resid 130, resid 80 and creates a sphere of radius 6.0 around the COG.

sphlayer ***innerRadius externalRadius selection***

selects all atoms within a spherical layer centered in the center of geometry (COG) of a given selection, e.g., `sphlayer 2.4 6.0 ( protein and ( resid 130 or resid 80 ) )` selects the center of geometry of protein, resid 130, resid 80 and creates a spherical layer of inner radius 2.4 and external radius 6.0 around the COG.

cyzone ***externalRadius zMax zMin selection***

selects all atoms within a cylindric zone centered in the center of geometry (COG) of a given selection, e.g. `cyzone 15 4 -8 protein and resid 42` selects the center of geometry of protein and resid 42, and creates a cylinder of external radius 15 centered on the COG. In z, the cylinder extends from 4 above the COG to 8 below. Positive values for *zMin*, or negative ones for *zMax*, are allowed.

cylayer ***innerRadius externalRadius zMax zMin selection***

selects all atoms within a cylindric layer centered in the center of geometry (COG) of a given selection, e.g. `cylayer 5 10 10 -8 protein` selects the center of geometry of protein, and creates a cylindrical layer of inner radius 5, external radius 10 centered on the COG. In z, the cylinder extends from 10 above the COG to 8 below. Positive values for *zMin*, or negative ones for *zMax*, are allowed.

point ***x y z distance***

selects all atoms within a cutoff of a point in space, make sure coordinate is separated by spaces, e.g. `point 5.0 5.0 5.0 3.5` selects all atoms within 3.5 Angstroms of the coordinate (5.0, 5.0, 5.0)

prop ***[abs] property operator value***

selects atoms based on position, using *property* **x**, **y**, or **z** coordinate. Supports the **abs** keyword (for absolute value) and the following *operators*: **<, >, <=, >=, ==, !=**. For example, `prop z >= 5.0` selects all atoms with z coordinate greater than 5.0; `prop abs z <= 5.0` selects all atoms within -5.0 <= z <= 5.0.

Similarity and connectivity

same ***subkeyword*** **as** ***selection***

selects all atoms that have the same *subkeyword* value as any atom in *selection*. Allowed *subkeyword* values are the atom properties: `name, type, resname, resid, resnum, segid, mass, charge, radius, bfactor`, the groups an atom belong to: `residue, segment, fragment`, and the atom coordinates `x, y, z`. (Note that `bfactor` currently only works for MMTF formats.) e.g. `same charge as protein` selects all atoms that have the same charge as any atom in protein.

byres ***selection***

selects all atoms that are in the same segment and residue as selection, e.g. specify the subselection after the byres keyword. `byres` is a shortcut to `same residue as`

bonded *selection*

selects all atoms that are bonded to selection e.g.: `name H and bonded name N` selects only hydrogens bonded to nitrogens

Index

index *index-range*

selects all atoms within a range of (0-based) inclusive indices, e.g. `index 0` selects the first atom in the universe; `index 5:10` selects the 6th through 11th atoms, inclusive. This uses the `Atom.index` *topology attribute*.

bynum *number-range*

selects all atoms within a range of (1-based) inclusive indices, e.g. `bynum 1` selects the first atom in the universe; `bynum 5:10` selects 5th through 10th atoms, inclusive.

> **Note:** These are **not** the same as the 1-indexed `Atom.id` *topology attribute*. `bynum` simply adds 1 to the 0-indexed `Atom.index`.

Preexisting selections and modifiers

group *group-name*

selects the atoms in the `AtomGroup` passed to the function as an argument named *group-name*. Only the atoms common to *group-name* and the instance `select_atoms()` was called from will be considered, unless `group` is preceded by the `global` keyword. *group-name* will be included in the parsing just by comparison of atom indices. This means that it is up to the user to make sure the *group-name* group was defined in an appropriate `Universe`.

global *selection*

by default, when issuing `select_atoms()` from an `AtomGroup`, selections and subselections are returned intersected with the atoms of that instance. Prefixing a selection term with `global` causes its selection to be returned in its entirety. As an example, the `global` keyword allows for `lipids.select_atoms("around 10 global protein")` — where `lipids` is a group that does not contain any proteins. Were `global` absent, the result would be an empty selection since the `protein` subselection would itself be empty. When calling `select_atoms()` from a `Universe`, `global` is ignored.

Dynamic selections

By default `select_atoms()` returns an `AtomGroup`, in which the list of atoms is constant across trajectory frame changes. If `select_atoms()` is invoked with named argument `updating` set to `True`, an `UpdatingAtomGroup` instance will be returned instead.

```
# A dynamic selection of corner atoms:
In [12]: ag_updating = u.select_atoms("prop x < 5 and prop y < 5 and prop z < 5",␣
↪updating=True)

In [13]: ag_updating
Out[13]: <AtomGroup with 917 atoms, with selection 'prop x < 5 and prop y < 5 and prop z
↪< 5' on the entire Universe.>
```

It behaves just like an `AtomGroup` object, with the difference that the selection expressions are re-evaluated every time the trajectory frame changes (this happens lazily, only when the `UpdatingAtomGroup` object is accessed so that there is no redundant updating going on):

```
In [14]: u.trajectory.next()
Out[14]: < Timestep 1 with unit cell dimensions [ 0.  0.  0. 90. 90. 90.] >

In [15]: ag_updating
Out[15]: <AtomGroup with 923 atoms, with selection 'prop x < 5 and prop y < 5 and prop z
↪< 5' on the entire Universe.>
```

Using the `group` selection keyword for *Preexisting selections and modifiers*, one can make updating selections depend on `AtomGroup`, or even other `UpdatingAtomGroup`, instances. Likewise, making an updating selection from an already updating group will cause later updates to also reflect the updating of the base group:

```
In [16]: chained_ag_updating = ag_updating.select_atoms("resid 1:1000", updating=True)

In [17]: chained_ag_updating
Out[17]: <AtomGroup with 923 atoms, with selection 'resid 1:1000' on another AtomGroup.>

In [18]: u.trajectory.next()
Out[18]: < Timestep 2 with unit cell dimensions [ 0.  0.  0. 90. 90. 90.] >

In [19]: chained_ag_updating
Out[19]: <AtomGroup with 921 atoms, with selection 'resid 1:1000' on another AtomGroup.>
```

Finally, a non-updating selection or a slicing/addition operation made on an `UpdatingAtomGroup` will return a static `AtomGroup`, which will no longer update across frames:

```
In [20]: static_ag = ag_updating.select_atoms("resid 1:1000")

In [21]: static_ag
Out[21]: <AtomGroup with 921 atoms>

In [22]: u.trajectory.next()
Out[22]: < Timestep 3 with unit cell dimensions [ 0.  0.  0. 90. 90. 90.] >

In [23]: static_ag
Out[23]: <AtomGroup with 921 atoms>
```

Ordered selections

`select_atoms()` sorts the atoms in the `AtomGroup` by atom index before returning them (this is to eliminate possible duplicates in the selection). If the ordering of atoms is crucial (for instance when describing angles or dihedrals) or if duplicate atoms are required then one has to concatenate multiple AtomGroups, which does not sort them.

The most straightforward way to concatenate two AtomGroups is by using the + operator:

```
In [14]: ordered = u.select_atoms("resid 3 and name CA") + u.select_atoms("resid 2 and␣
↪name CA")

In [15]: list(ordered)
Out[15]: 
[<Atom 46: CA of type 22 of resname ILE, resid 3 and segid 4AKE>,
 <Atom 22: CA of type 22 of resname ARG, resid 2 and segid 4AKE>]
```

A shortcut is to provide *two or more* selections to `select_atoms()`, which then does the concatenation automatically:

```
In [16]: list(u.select_atoms("resid 3 and name CA", "resid 2 and name CA"))
Out[16]:
[<Atom 46: CA of type 22 of resname ILE, resid 3 and segid 4AKE>,
 <Atom 22: CA of type 22 of resname ARG, resid 2 and segid 4AKE>]
```

Just for comparison to show that a single selection string does not work as one might expect:

```
In [17]: list(u.select_atoms("(resid 3 or resid 2) and name CA"))
Out[17]:
[<Atom 22: CA of type 22 of resname ARG, resid 2 and segid 4AKE>,
 <Atom 46: CA of type 22 of resname ILE, resid 3 and segid 4AKE>]
```

2.1.10 The topology system

MDAnalysis groups static data about a `Universe` into its topology. This is typically loaded from a topology file. Topology information falls into 3 categories:

- *Atom containers (Residues and Segments)*
- *Atom attributes (e.g. name, mass, tempfactor)*
- *Topology objects: bonds, angles, dihedrals, impropers*

Users will almost never interact directly with a `Topology`. Modifying atom containers or topology attributes is typically done through `Universe`. Methods for viewing containers or topology attributes, or for calculating topology object values, are accessed through `AtomGroup`.

Topology attributes

MDAnalysis supports a range of topology attributes for each `Atom` and `AtomGroup`. If an attribute is defined for an Atom, it will be for an AtomGroup, and vice versa – however, they are accessed with singular and plural versions of the attribute specifically.

Canonical attributes

These attributes are derived for every `Universe`, including Universes created with `empty()`. They encode the MDAnalysis order of each object.

Atom	AtomGroup	Description
index	indices	MDAnalysis canonical atom index (from 0)
resindex	resindices	MDAnalysis canonical residue index (from 0)
segindex	segindices	MDAnalysis segment index (from 0)

The following attributes are read or guessed from every format supported by MDAnalysis.

Atom	AtomGroup	Description
id	ids	atom serial (from 1, except PSF/DMS/TPR formats)
mass	masses	atom mass (guessed, default: 0.0)
resid	resids	residue number (from 1, except for TPR)
resnum	resnums	alias of resid
segid	segids	names of segments (default: ‘SYSTEM’)
type	types	atom name, atom element, or force field atom type

Format-specific attributes

The table below lists attributes that are read from supported formats. These can also be *added to a Universe* created from a file that does not support them.

Connectivity information

MDAnalysis can also read connectivity information, if the file provides it. These become available as *Topology objects*, which have additional functionality.

Adding TopologyAttrs

Each of the attributes above can be added to a Universe if it was not available in the file with `add_TopologyAttr()`.

`add_TopologyAttr()` takes two arguments:

- `topologyattr` : the singular or plural name of a TopologyAttr, *or* a MDAnalysis TopologyAttr object. This must already have been defined as a `TopologyAttr` (see *Adding custom TopologyAttrs* for an example of adding a custom topology attribute).
- `values` (optional) : if `topologyattr` is a string, the values for that attribute. This can be `None` if the attribute has default values defined, e.g. `tempfactors`.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PSF

In [3]: psf = mda.Universe(PSF)

In [4]: hasattr(psf.atoms, 'tempfactors')
Out[4]: False

In [5]: psf.add_TopologyAttr('tempfactors')

In [6]: psf.atoms.tempfactors
Out[6]: array([0., 0., 0., ..., 0., 0., 0.])
```

One way to modify topology attributes is to simply replace them with `add_TopologyAttr()`:

```
In [7]: psf.add_TopologyAttr('tempfactors', range(len(psf.atoms)))

In [8]: psf.atoms.tempfactors
Out[8]:
```

(continues on next page)

(continued from previous page)

```
array([0.000e+00, 1.000e+00, 2.000e+00, ..., 3.338e+03, 3.339e+03,
       3.340e+03])
```

The number of values provided should correspond with the "level" of the attribute. For example, B-factors are atomic-level information. However, residue names and residue ids apply to residues. See a *table of attribute levels and default values* for more information.

Modifying TopologyAttrs

Existing topology attributes can be directly modified by assigning new values.

```
In [9]: import MDAnalysis as mda

In [10]: from MDAnalysis.tests.datafiles import PDB

In [11]: pdb = mda.Universe(PDB)

In [12]: pdb.residues[:3].resnames
Out[12]: array(['MET', 'ARG', 'ILE'], dtype=object)

In [13]: pdb.residues[:3].resnames = ['RES1', 'RES2', 'RES3']

In [14]: pdb.residues[:3].atoms.resnames
Out[14]: 
array(['RES1', 'RES1', 'RES1', 'RES1', 'RES1', 'RES1', 'RES1', 'RES1',
       'RES1', 'RES1', 'RES1', 'RES1', 'RES1', 'RES1', 'RES1', 'RES1',
       'RES1', 'RES1', 'RES1', 'RES2', 'RES2', 'RES2', 'RES2', 'RES2',
       'RES2', 'RES2', 'RES2', 'RES2', 'RES2', 'RES2', 'RES2', 'RES2',
       'RES2', 'RES2', 'RES2', 'RES2', 'RES2', 'RES2', 'RES2', 'RES2',
       'RES2', 'RES2', 'RES2', 'RES3', 'RES3', 'RES3', 'RES3', 'RES3',
       'RES3', 'RES3', 'RES3', 'RES3', 'RES3', 'RES3', 'RES3', 'RES3',
       'RES3', 'RES3', 'RES3', 'RES3', 'RES3', 'RES3'], dtype=object)
```

Note: This method cannot be used with connectivity attributes, i.e. bonds, angles, dihedrals, and impropers.

Similarly to adding topology attributes with `add_TopologyAttr()`, the "level" of the attribute matters. Residue attributes can only be assigned to attributes at the Residue or ResidueGroup level. The same applies to attributes for Atoms and Segments. For example, we would get a NotImplementedError if we tried to assign resnames to an AtomGroup.

```
In [15]: pdb.residues[0].atoms.resnames = ['new_name']

NotImplementedErrorTraceback (most recent call last)
<ipython-input-15-0f99b0dc5f49> in <module>
----> 1 pdb.residues[0].atoms.resnames = ['new_name']
...
NotImplementedError: Cannot set resnames from AtomGroup. Use 'AtomGroup.residues.
↪resnames = '
```

Default values and attribute levels

Topology information in MDAnalysis is always associated with a level: one of atom, residue, or segment. For example, `indices` is Atom information, `resindices` is Residue information, and `segindices` is Segment information. Many topology attributes also have default values, so that they can be *added to a Universe without providing explicit values*, and expected types. The table below lists which attributes have default values, what they are, and the information level.

Topology objects

MDAnalysis defines four types of `TopologyObject` by connectivity:

- `Bond`
- `Angle`
- `Dihedral`
- `ImproperDihedral`

The value of each topology object can be calculated with `value()`.

Each `TopologyObject` also contains the following attributes:

- `atoms` : the ordered atoms in the object
- `indices` : the ordered atom indices in the object
- `type` : this is either the 'type' of the bond/angle/dihedral/improper, or a tuple of the atom types.
- `is_guessed` : MDAnalysis can guess bonds. This property records if the object was read from a file or guessed.

Groups of these are held in `TopologyGroup`s. The master groups of TopologyObjects are *accessible as properties of a Universe*. TopologyObjects are typically read from a file with connectivity information (*see the supported formats here*). However, they can be created in two ways: by adding them to a Universe, or by creating them from an `AtomGroup`. Bonds can be guessed based on distance and Van der Waals' radii with `AtomGroup.guess_bonds`.

Adding to a Universe

As of version 0.21.0, there are specific methods for adding `TopologyObject`s to a `Universe`:

- `add_Bonds()`
- `add_Angles()`
- `add_Dihedrals()`
- `add_Impropers()`

These accept the following values:

- a `TopologyGroup`
- an iterable of atom indices
- an iterable of `TopologyObject`s

Prior to version 0.21.0, objects could be added to a Universe with `add_TopologyAttr()`.

```
In [15]: hasattr(pdb, 'angles')
Out[15]: False
```

(continues on next page)

(continued from previous page)

```
In [16]: pdb.add_TopologyAttr('angles', [(0, 1, 2), (2, 3, 4)])

In [17]: pdb.angles
Out[17]: <TopologyGroup containing 2 angles>
```

Both of these methods add the new objects to the associated master `TopologyGroup` in the `Universe`.

Creating with an AtomGroup

An `AtomGroup` can be represented as a bond, angle, dihedral angle, or improper angle `TopologyObject` through the respective properties:

- `bond`
- `angle`
- `dihedral`
- `improper`

The `AtomGroup` must contain the corresponding number of atoms, in the desired order. For example, a bond cannot be created from three atoms.

```
In [18]: pdb.atoms[[3, 4, 2]].bond

ValueErrorTraceback (most recent call last)
<ipython-input-21-e59c36ab66f4> in <module>
----> 1 pdb.atoms[[3, 4, 2]].bond
...
ValueError: bond only makes sense for a group with exactly 2 atoms
```

However, the angle Atom 2 —— Atom 4 —— Atom 3 can be calculated, even if the atoms are not connected with bonds.

```
In [18]: a = pdb.atoms[[3, 4, 2]].angle

In [19]: print(a.value())
47.770653826924175
```

These AtomGroup `TopologyObject`s are not added to the associated master `TopologyGroup` in the `Universe`.

Deleting from a Universe

As of version 0.21.0, there are specific methods for deleting `TopologyObject`s from a `Universe`:

- `delete_Bonds()`
- `delete_Angles()`
- `delete_Dihedrals()`
- `delete_Impropers()`

Topology-specific methods

A number of analysis and transformation methods are defined for AtomGroup, ResidueGroup, and SegmentGroup that require specific properties to be available. The primary requirement is the *positions* attribute. With positions, you can easily compute a center of geometry:

```
>>> u.atoms.center_of_geometry()
array([-0.04223882,  0.01418196, -0.03504874])
```

The following methods all require coordinates.

- bbox()
- bsphere()
- center()
- center_of_geometry()
- centroid()
- pack_into_box()
- rotate()
- rotate_by()
- transform()
- translate()
- unwrap()
- wrap()

Other methods are made available when certain topology attributes are defined in the Universe. These are listed below.

2.1.11 Trajectories

In MDAnalysis, static data is contained in your universe Topology, while dynamic data is drawn from its trajectory at `Universe.trajectory`. This is typically loaded from a trajectory file and includes information such as:

- atom coordinates (`Universe.atoms.positions`)
- box size (`Universe.dimensions`)
- velocities and forces (if your file format contains the data) (`Universe.atoms.velocities`)

Although these properties look static, they are actually dynamic, and the data contained within can change. In order to remain memory-efficient, MDAnalysis does not load every frame of your trajectory into memory at once. Instead, a Universe has a state: the particular timestep that it is currently associated with in the trajectory. When the timestep changes, the data in the properties above shifts accordingly.

The typical way to change a timestep is to index it. `Universe.trajectory` can be thought of as a list of `Timesteps`, a data structure that holds information for the current time frame. For example, you can query its length.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PSF, DCD

In [3]: u = mda.Universe(PSF, DCD)

```

(continues on next page)

(continued from previous page)

```
In [4]: len(u.trajectory)
Out[4]: 98
```

When a trajectory is first loaded from a file, it is set to the first frame (with index 0), by default.

```
In [5]: print(u.trajectory.ts, u.trajectory.time)
< Timestep 0 > 0.9999999119200186
```

Indexing the trajectory returns the timestep for that frame, and sets the Universe to point to that frame until the timestep next changes.

```
In [6]: u.trajectory[3]
Out[6]: < Timestep 3 >
```

```
In [7]: print('Time of fourth frame', u.trajectory.time)
Time of fourth frame 3.9999996476800743
```

Many tasks involve applying a function to each frame of a trajectory. For these, you need to iterate through the frames, *even if you don't directly use the timestep*. This is because the act of iterating moves the Universe onto the next frame, changing the dynamic atom coordinates.

Trajectories can also be *sliced* if you only want to work on a subset of frames.

```
In [8]: protein = u.select_atoms('protein')

In [9]: for ts in u.trajectory[:20:4]:
   ...:     rad = protein.radius_of_gyration()
   ...:     print('frame={}: radgyr={}'.format(ts.frame, rad))
   ...:
frame=0: radgyr=16.669018368649777
frame=4: radgyr=16.743960893217544
frame=8: radgyr=16.78938645874581
frame=12: radgyr=16.87231363208217
frame=16: radgyr=17.003316543310998
```

Note that after iterating over the trajectory, the frame is always set back to the first frame, even if your loop stopped before the trajectory end.

```
In [10]: u.trajectory.frame
Out[10]: 0
```

Because MDAnalysis will pull trajectory data directly from the file it is reading from, changes to atom coordinates and box dimensions will not persist once the frame is changed. The only way to make these changes permanent is to load the trajectory into memory, or to write a new trajectory to file for every frame. For example, to set a cubic box size for every frame and write it out to a file:

```
with mda.Writer('with_box.trr', 'w', n_atoms=u.atoms.n_atoms) as w:
    for ts in u.trajectory:
        ts.dimensions = [10, 10, 10, 90, 90, 90]
        w.write(u.atoms)

u_with_box = mda.Universe(PSF, 'with_box.trr')
```

Sometimes you may wish to only transform part of the trajectory, or to not write a file out. In these cases, MDAnalysis supports *"on-the-fly" transformations* that are performed on a frame when it is read.

2.1.12 Slicing trajectories

MDAnalysis trajectories can be indexed to return a `Timestep`, or sliced to give a `FrameIterator`.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PSF, DCD

In [3]: u = mda.Universe(PSF, DCD)

In [4]: u.trajectory[4]
Out[4]: < Timestep 4 >
```

Indexing a trajectory shifts the `Universe` to point towards that particular frame, updating dynamic data such as `Universe.atoms.positions`.

Note: The trajectory frame is not read from the MD data. It is the internal index assigned by MDAnalysis.

```
In [5]: u.trajectory.frame
Out[5]: 4
```

Creating a `FrameIterator` by slicing a trajectory does not shift the `Universe` to a new frame, but *iterating* over the sliced trajectory will rewind the trajectory back to the first frame.

```
In [6]: fiter = u.trajectory[10::10]

In [7]: frames = [ts.frame for ts in fiter]

In [8]: print(frames, u.trajectory.frame)
[10, 20, 30, 40, 50, 60, 70, 80, 90] 0
```

You can also create a sliced trajectory with boolean indexing and fancy indexing. Boolean indexing allows you to select only frames that meet a certain condition, by passing a `ndarray` with the same length as the original trajectory. Only frames that have a boolean value of `True` will be in the resulting `FrameIterator`. For example, to select only the frames of the trajectory with an RMSD under 2 angstrom:

```
In [9]: from MDAnalysis.analysis import rms

In [10]: protein = u.select_atoms('protein')

In [11]: rmsd = rms.RMSD(protein, protein).run()

In [12]: bools = rmsd.results.rmsd.T[-1] < 2

In [13]: print(bools)
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
```

(continues on next page)

(continued from previous page)

```
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False]
```

```
In [14]: fiter = u.trajectory[bools]

In [15]: print([ts.frame for ts in fiter])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

You can also use fancy indexing to control the order of specific frames.

```
In [16]: indices = [10, 2, 3, 9, 4, 55, 2]

In [17]: print([ts.frame for ts in u.trajectory[indices]])
[10, 2, 3, 9, 4, 55, 2]
```

You can even slice a `FrameIterator` to create a new `FrameIterator`.

```
In [18]: print([ts.frame for ts in fiter[::3]])
[0, 3, 6, 9, 12]
```

2.1.13 On-the-fly transformations

An on-the-fly transformation is a function that silently modifies the dynamic data contained in a trajectory `Timestep` (typically coordinates) as it is loaded into memory. It is called for each current time step to transform data into your desired representation. A transformation function must also return the current `Timestep`, as transformations are often chained together.

The `MDAnalysis.transformations` module contains a collection of transformations. For example, `fit_rot_trans()` can perform a mass-weighted alignment on an `AtomGroup` to a reference.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import TPR, XTC

In [3]: from MDAnalysis import transformations as trans

In [4]: u = mda.Universe(TPR, XTC)

In [5]: protein = u.select_atoms('protein')

In [6]: align_transform = trans.fit_rot_trans(protein, protein, weights=protein.masses)

In [7]: u.trajectory.add_transformations(align_transform)
```

Other implemented transformations include functions to `translate`, `rotate`, `fit` an `AtomGroup` to a reference, and `wrap` or unwrap groups in the unit cell. (Please see the MDAnalysis on-the-fly transformations blog post contains a more complete introduction to these fitting and wrapping functions.)

Although you can only call `add_transformations()` *once*, you can pass in multiple transformations in a list, which will be executed in order.

There is a transformations tutorial that shows in more detail how to use transformations. A few simple examples are given below.

Example workflows

The workflow below

- makes all molecules whole (unwraps them over periodic boundary conditions),
- centers the protein in the center of the box,
- wraps water back into the box.

```
# create new Universe for new transformations
In [8]: u = mda.Universe(TPR, XTC)

In [9]: protein = u.select_atoms('protein')

In [10]: water = u.select_atoms('resname SOL')

In [11]: workflow = [trans.unwrap(u.atoms),
   ....:             trans.center_in_box(protein, center='geometry'),
   ....:             trans.wrap(water, compound='residues')]
   ....:

In [12]: u.trajectory.add_transformations(*workflow)
```

Please see the full tutorial for more information.

If your transformation does not depend on something within the Universe (e.g. a chosen AtomGroup), you can also create a Universe directly with transformations. The code below translates coordinates 1 angstrom up on the z-axis:

```
In [13]: u = mda.Universe(TPR, XTC, transformations=[trans.translate([0, 0, 1])])
```

If you need a different transformation, it is easy to implement your own.

Custom transformations

At its core, a transformation function must only take a Timestep as its input and return the Timestep as the output.

```
In [14]: import numpy as np

In [15]: def up_by_2(ts):
   ....:     """Translates atoms up by 2 angstrom"""
   ....:     ts.positions += np.array([0.0, 0.0, 0.2])
   ....:     return ts
   ....:

In [16]: u = mda.Universe(TPR, XTC, transformations=[up_by_2])
```

If your transformation needs other arguments, you will need to wrap your core transformation with a wrapper function that can accept the other arguments.

```
In [17]: def up_by_x(x):
   ....:     """Translates atoms up by x angstrom"""
   ....:     def wrapped(ts):
   ....:         """Handles the actual Timestep"""
   ....:         ts.positions += np.array([0.0, 0.0, float(x)])
   ....:         return ts
   ....:     return wrapped
   ....:

# load Universe with transformations that move it up by 7 angstrom
In [18]: u = mda.Universe(TPR, XTC, transformations=[up_by_x(5), up_by_x(2)])
```

Alternatively, you can use `functools.partial()` to substitute the other arguments.

```
In [19]: import functools

In [20]: def up_by_x(ts, x):
   ....:     ts.positions += np.array([0.0, 0.0, float(x)])
   ....:     return ts
   ....:

In [21]: up_by_5 = functools.partial(up_by_x, x=5)

In [22]: u = mda.Universe(TPR, XTC, transformations=[up_by_5])
```

On-the-fly transformation functions can be applied to any property of a Timestep, not just the atom positions. For example, to give each frame of a trajectory a box:

```
In [23]: def set_box(ts):
   ....:     ts.dimensions = [10, 20, 30, 90, 90, 90]
   ....:     return ts
   ....:

In [24]: u = mda.Universe(TPR, XTC, transformations=[set_box])
```

2.1.14 Units and constants

The units of MDAnalysis trajectories are the **Å** (**ångström**) for **length** and **ps** (**picosecond**) for **time**. Regardless of how the original MD format stored the trajectory data, MDAnalysis converts it to MDAnalysis units when reading the data in, and converts back when writing the data out. Analysis classes generally also use these default units. Exceptions to the default units are always noted in the documentation; for example, mass densities can be given in g/cm^3.

Other base units are listed in the table *Base units in MDAnalysis*.

Table 1: Base units in MDAnalysis

Quantity	Unit	SI units
length	Å	10^{-10} m
time	ps	10^{-12} s
energy	kJ/mol	$1.66053892103219 \times 10^{-21}$ J
charge	e	$1.602176565 \times 10^{-19}$ As
force	kJ/(mol·Å)	$1.66053892103219 \times 10^{-11}$ J/m
speed	Å/ps	100 m/s
mass	u	$1.66053906660(50) \times 10^{-27}$ kg
angle	degrees	$\frac{\pi}{180}$ rad

Unit conversion

Quantities can be converted from units with `convert()`. `convert()` simply multiplies the initial quantity with a precomputed conversion factor, as obtained from `get_conversion_factor()`.

The computed conversion factors for each quantity type are stored in `MDAnalysis.units` and shown below.

Constants

2.1.15 Reading and writing files

Input

Read information from topology and coordinate files to create a *Universe*:

```
import MDAnalysis as mda
u = mda.Universe('topology.gro', 'trajectory.xtc')
```

A topology file is always required for a Universe, whereas coordinate files are optional. *Some file formats provide both topology and coordinate information*. MDAnalysis supports a number of *formats*, which are automatically detected based on the file extension. For example, the above loads a *GROMACS XTC trajectory*. Multiple coordinate files can be loaded, as *described below*; the following code loads two *CHARMM/NAMD DCD files* and concatenates them:

```
u = mda.Universe('topology.psf', 'trajectory1.dcd', 'trajectory2.dcd')
```

Some formats can be loaded with format-specific keyword arguments, such as the *LAMMPS DATA* `atom_style` specification.

See also:

See *Loading from files* for more information on loading data into a Universe from files.

Reading multiple trajectories

A Universe can load multiple trajectories, which are concatenated in the order given. One exception to this is with *XTC* and *TRR* files. If the `continuous=True` flag is *passed to Universe*, MDAnalysis will try to stitch them together so that the trajectory is as time-continuous as possible. This means that there will be no duplicate time-frames, or jumps back in time.

As an example, the following depicted trajectory is split into four parts. The column represents the time. As you can see, some segments overlap. With `continuous=True`, only the frames marked with a + will be read.

```
part01:  ++++--
part02:      ++++++-
part03:            ++++++++
part04:                         ++++
```

However, there can be gaps in time (i.e. frames appear to be missing). Ultimately it is the user's responsiblity to ensure that the trajectories can be stitched together meaningfully.

Note: While you can set `continuous=True` for either XTC or TRR files, you cannot mix different formats.

More information can be found at the API reference for `ChainReader`.

Trajectory formats

If no `format` keyword is provided, `ChainReader` will try to guess the format for each file from its extension. You can force `ChainReader` to use the same format for every file by using the `format` keyword. You can also specify which format to use by file, by passing in a sequence of `(filename, format)` tuples.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PDB, GRO

In [3]: u = mda.Universe(PDB, [(GRO, 'gro'), (PDB, 'pdb'), (GRO, 'gro')])

In [4]: u.trajectory
Out[4]: <ChainReader containing adk_oplsaa.gro, adk_oplsaa.pdb, adk_oplsaa.gro with 3␣
↪frames of 47681 atoms>
```

In-memory trajectories

Reading trajectories into memory

If your device has sufficient memory to load an entire trajectory into memory, then analysis can be sped up substantially by transferring the trajectory to memory. This makes it possible to operate on raw coordinates using existing MDAnalysis tools. In addition, it allows the user to make changes to the coordinates in a trajectory (e.g. through `AtomGroup.positions`) without having to write the entire state to file.

The most straightforward way to do this is to pass `in_memory=True` to `Universe`, which automatically transfers a trajectory to memory:

```
In [5]: from MDAnalysis.tests.datafiles import TPR, XTC

In [6]: universe = mda.Universe(TPR, XTC, in_memory=True)
```

MDAnalysis uses the MemoryReader class to load this data in.

Transferring trajectories into memory

The decision to transfer the trajectory to memory can be made at any time with the transfer_to_memory() method of a Universe:

```
In [7]: universe = mda.Universe(TPR, XTC)

In [8]: universe.transfer_to_memory()
```

This operation may take a while (passing verbose=True to transfer_to_memory() will display a progress bar). However, subsequent operations on the trajectory will be very fast.

Building trajectories in memory

MemoryReader can also be used to directly generate a trajectory as a numpy array.

```
In [9]: from MDAnalysisTests.datafiles import PDB

In [10]: from MDAnalysis.coordinates.memory import MemoryReader

In [11]: import numpy as np

In [12]: universe = mda.Universe(PDB)

In [13]: universe.atoms.positions
Out[13]: 
array([[ 52.017,  43.56 ,  31.555],
       [ 51.188,  44.112,  31.722],
       [ 51.551,  42.828,  31.039],
       ...,
       [105.342,  74.072,  40.988],
       [ 57.684,  35.324,  14.804],
       [ 62.961,  47.239,   3.753]], dtype=float32)
```

The load_new() method can be used to load coordinates into a Universe, replacing the old coordinates:

```
In [14]: coordinates = np.random.rand(len(universe.atoms), 3)

In [15]: universe.load_new(coordinates, format=MemoryReader);

In [16]: universe.atoms.positions
Out[16]: 
array([[0.84312797, 0.9728311 , 0.85561925],
       [0.8775258 , 0.25540632, 0.7579896 ],
       [0.8522356 , 0.89522445, 0.01509629],
```

(continues on next page)

(continued from previous page)

```
       ...,
       [0.70939714, 0.86493427, 0.263896  ],
       [0.8345135 , 0.4645135 , 0.08588151],
       [0.83520323, 0.24146114, 0.20034026]], dtype=float32)
```

or they can be directly passed in when creating a Universe.

```
In [17]: universe2 = mda.Universe(PDB, coordinates, format=MemoryReader)

In [18]: universe2.atoms.positions
Out[18]: 
array([[0.84312797, 0.9728311 , 0.85561925],
       [0.8775258 , 0.25540632, 0.7579896 ],
       [0.8522356 , 0.89522445, 0.01509629],
       ...,
       [0.70939714, 0.86493427, 0.263896  ],
       [0.8345135 , 0.4645135 , 0.08588151],
       [0.83520323, 0.24146114, 0.20034026]], dtype=float32)
```

In-memory trajectories of an atom selection

Creating a trajectory of an atom selection requires transferring the appropriate units. This is often needed when using `Merge()` to create a new Universe, as coordinates are not automatically loaded in.

Output

Frames and trajectories

MDAnalysis `Universe`s can be written out to a *number of formats* with `write()`. For example, to write the current frame as a PDB:

```
from MDAnalysis.tests.datafiles import PDB, TRR
u = mda.Universe(PDB, TRR)
ag = u.select_atoms("name CA")
ag.write("c-alpha.pdb")
```

Pass in the `frames` keyword to write out trajectories.

```
ag.write('c-alpha_all.xtc', frames='all')
```

Slice or index the trajectory to choose which frames to write:

```
ag.write('c-alpha_skip2.trr', frames=u.trajectory[::2])
ag.write('c-alpha_some.dcd', frames=u.trajectory[[0,2,3]])
```

Alternatively, iterate over the trajectory frame-by-frame with `Writer()`. This requires you to pass in the number of atoms to write.

```
with mda.Writer('c-alpha.xyz', ag.n_atoms) as w:
    for ts in u.trajectory:
        w.write(ag)
```

You can pass keyword arguments to some format writers. For example, the *LAMMPS DATA* format allows the `lengthunit` and `timeunit` keywords to specify the output units.

Pickling

MDAnalysis supports pickling of most of its data structures and trajectory formats. Unsupported attributes can be found in PR #2887.

```
In [19]: import pickle

In [20]: from MDAnalysis.tests.datafiles import PSF, DCD

In [21]: psf = mda.Universe(PSF, DCD)

In [22]: pickle.loads(pickle.dumps(psf))
Out[22]: <Universe with 3341 atoms>
```

As for `MDAnalysis.core.groups.AtomGroup`, during serialization, it will be pickled with its bound `MDAnalysis.core.universe.Universe`. This means that after unpickling, a new `MDAnalysis.core.universe.Universe` will be created and be attached to the new `MDAnalysis.core.groups.AtomGroup`. If the Universe is serialized with its `MDAnalysis.core.groups.AtomGroup`, they will still be bound together afterwards:

```
In [23]: import pickle

In [24]: from MDAnalysis.tests.datafiles import PSF, DCD

In [25]: u = mda.Universe(PSF, DCD)

In [26]: g = u.atoms

In [27]: g_pickled = pickle.loads(pickle.dumps(g))

In [28]: print("g_pickled.universe is u: ", u is g_pickled.universe)
g_pickled.universe is u:  False

In [29]: g_pickled, u_pickled = pickle.loads(pickle.dumps((g, u)))

In [30]: print("g_pickled.universe is u_pickled: ",
   ....:       u_pickled is g_pickled.universe)
   ....:
g_pickled.universe is u_pickled:  True
```

If multiple `MDAnalysis.core.groups.AtomGroups` are bound to the same `MDAnalysis.core.universe.Universe`, they will also be bound to the same one after serialization:

```
In [1]: u = mda.Universe(PSF, DCD)

In [2]: g = u.atoms[:2]

In [3]: h = u.atoms[2:4]

In [4]: g_pickled = pickle.loads(pickle.dumps(g))
```

(continues on next page)

(continued from previous page)

```
In [5]: h_pickled = pickle.loads(pickle.dumps(h))

In [6]: print("g_pickled.universe is h_pickled.universe : ",
   ...:        g_pickled.universe is h_pickled.universe)
   ...:
g_pickled.universe is h_pickled.universe :  False

In [7]: g_pickled, h_pickled = pickle.loads(pickle.dumps((g, h)))

In [8]: print("g_pickled.universe is h_pickled.universe: ",
   ...:        g_pickled.universe is h_pickled.universe)
   ...:
g_pickled.universe is h_pickled.universe:  True
```

2.1.16 Format overview

MDAnalysis can read topology or coordinate information from a wide variety of file formats. The emphasis is on formats used in popular simulation packages. By default, MDAnalysis figures out formats by looking at the extension, unless the format is *explicitly specified* with the `format` or `topology_format` keywords.

Below is *a table of formats in MDAnalysis*, and which information can be read from them. A topology file supplies the list of atoms in the system, their connectivity and possibly additional information such as B-factors, partial charges, etc. The details depend on the file format and not every topology file provides all (or even any) additional data.

Important: File formats are complicated and not always well defined. MDAnalysis tries to follow published standards but this can sometimes surprise users. It is *highly* recommended that you read the page for your data file format instead of assuming certain behaviour. If you encounter problems with a file format, please *get in touch with us*.

As a minimum, all topology parsers will provide atom `ids`, atom `types`, `masses`, `resids`, `resnums`, and `segids`. They will also assign all Atoms to Residues and all Residues to Segments. For systems without residues and segments, this results in there being a single Residue and Segment to which all Atoms belong. See *Topology attributes* for more topology attributes.

Often when data is not provided by a file, it will be guessed based on other data in the file. In this scenario, MDAnalysis will issue a warning. See *Guessing* for more information.

If a trajectory is loaded without time information, MDAnalysis will set a default timestep of 1.0 ps, where the first frame starts at 0.0 ps. In order to change these, *pass the following optional arguments to Universe*:

- `dt`: the timestep
- `time_offset`: the starting time from which to calculate the time of each frame

Topology

Coordinates

2.1.17 Guessing

When a Universe is created from a Universe, MDAnalysis guesses properties that have not been read from the file. Sometimes these properties are available in the file, but are simply not read by MDAnalysis. For example, *masses are always guessed*.

Masses

Atom masses are always guessed for every file format. They are guessed from the `Atom.atom_type`. This attribute represents a number of different values in MDAnalysis, depending on which file format you used to create your Universe. `Atom.atom_type` can be force-field specific atom types, from files that provide this information; or it can be an element, guessed from the atom name. See further discussion here.

Important: When an atom mass cannot be guessed from the atom `atom_type` or `name`, the atom is assigned a mass of 0.0. Masses are guessed atom-by-atom, so even if most atoms have been guessed correctly, it is possible that some have been given masses of 0. It is important to check for non-zero masses before using methods that rely on them, such as `AtomGroup.center_of_mass()`.

Types

When atom `atom_types` are guessed, they represent the atom element. Atom types are always guessed from the atom name. MDAnalysis follows biological naming conventions, where atoms named "CA" are much more likely to represent an alpha-carbon than a calcium atom. This guesser is still relatively fragile for non-traditionally biological atom names.

Bonds, Angles, Dihedrals, Impropers

MDAnalysis can guess if bonds exist between two atoms, based on the distance between them. A bond is created if the 2 atoms are within

$$d < f \cdot (R_1 + R_2)$$

of each other, where R_1 and R_2 are the VdW radii of the atoms and f is an ad-hoc *fudge_factor*. This is the same algorithm that VMD uses.

Angles can be guessed from the bond connectivity. MDAnalysis assumes that if atoms 1 & 2 are bonded, and 2 & 3 are bonded, then (1,2,3) must be an angle.

```
1
 \
  2 -- 3
```

Dihedral angles and improper dihedrals can both be guessed from angles. Proper dihedrals are guessed by assuming that if (1,2,3) is an angle, and 3 & 4 are bonded, then (1,2,3,4) must be a dihedral.

```
1         4
 \       /
  2 -- 3
```

Likewise, if (1,2,3) is an angle, and 2 & 4 are bonded, then (2, 1, 3, 4) must be an improper dihedral (i.e. the improper dihedral is the angle between the planes formed by (1, 2, 3) and (1, 3, 4))

```
1
 \
  2 -- 3
 /
4
```

The method available to users is `AtomGroup.guess_bonds`, which allows users to pass in a dictionary of van der Waals' radii for atom types. This guesses bonds, angles, and dihedrals (but not impropers) for the specified AtomGroup and adds it to the underlying Universe.

2.1.18 Auxiliary files

Auxiliary readers allow you to read in timeseries data accompanying a trajectory that is not stored in the regular trajectory file.

Supported formats

Reader	Format	Extension (if file)	Remarks
`XVGReade`	XVG	xvg (default)	Produced by Gromacs during simulation or analysis. Reads full file on initialisation.
`XVGFileR`	XVG-F	xvg	Alternate xvg file reader, reading each step from the file in turn for a lower memory footprint. `XVGReader` is the default reader for .xvg files.
`EDRReade`	EDR	edr	Produced by Gromacs during simulation or analysis. Reads full file on initialisation.

XVG Files

Reading data directly

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import XVG_BZ2  # cobrotoxin protein forces

In [3]: aux = mda.auxiliary.core.auxreader(XVG_BZ2)

In [4]: aux
Out[4]: <MDAnalysis.auxiliary.XVG.XVGReader at 0x7f4c916108e0>
```

In stand-alone use, an auxiliary reader allows you to iterate over each step in a set of auxiliary data.

```
In [5]: for step in aux:
   ...:     print(step.data)
   ...:
```

(continues on next page)

(continued from previous page)

```
[    0.        200.71288 -1552.2849  ...   128.4072   1386.0378
 -2699.3118 ]
[   50.      -1082.6454   -658.32166 ...  -493.02954   589.8844
  -739.2124 ]
[  100.       -246.27269   146.52911 ...   484.32501  2332.3767
 -1801.6234 ]
```

Use slicing to skip steps.

```
In [6]: for step in aux[1:2]:
   ...:     print(step.time)
   ...:
50.0
```

The `auxreader()` function uses the `get_auxreader_for()` to return an appropriate class. This can guess the format either from a filename, ‘

```
In [7]: mda.auxiliary.core.get_auxreader_for(XVG_BZ2)
Out[7]: MDAnalysis.auxiliary.XVG.XVGReader
```

or return the reader for a specified format.

```
In [8]: mda.auxiliary.core.get_auxreader_for(format='XVG-F')
Out[8]: MDAnalysis.auxiliary.XVG.XVGFileReader
```

Loading data into a Universe

Auxiliary data may be added to a trajectory Reader through the `add_auxiliary()` method. Auxiliary data may be passed in as a AuxReader instance, or directly as e.g. a filename, in which case `get_auxreader_for()` is used to guess an appropriate reader.

```
In [9]: from MDAnalysis.tests.datafiles import PDB_xvf, TRR_xvf

In [10]: u = mda.Universe(PDB_xvf, TRR_xvf)

In [11]: u.trajectory.add_auxiliary('protein_force', XVG_BZ2)

In [12]: for ts in u.trajectory:
   ....:     print(ts.aux.protein_force)
   ....:
[    0.        200.71288 -1552.2849  ...   128.4072   1386.0378
 -2699.3118 ]
[   50.      -1082.6454   -658.32166 ...  -493.02954   589.8844
  -739.2124 ]
[  100.       -246.27269   146.52911 ...   484.32501  2332.3767
 -1801.6234 ]
```

Passing arguments to auxiliary data

For alignment with trajectory data, auxiliary readers provide methods to assign each auxiliary step to the nearest trajectory timestep, read all steps assigned to a trajectory timestep and calculate 'representative' value(s) of the auxiliary data for that timestep.

To set a timestep or ??

'Assignment' of auxiliary steps to trajectory timesteps is determined from the time of the auxiliary step, `dt` of the trajectory and time at the first frame of the trajectory. If there are no auxiliary steps assigned to a given timestep (or none within `cutoff`, if set), the representative value(s) are set to `np.nan`.

Iterating over auxiliary data

Auxiliary data may not perfectly line up with the trajectory, or have missing data.

```
In [13]: from MDAnalysis.tests.datafiles import PDB, TRR

In [14]: u_long = mda.Universe(PDB, TRR)

In [15]: u_long.trajectory.add_auxiliary('protein_force', XVG_BZ2, dt=200)

In [16]: for ts in u_long.trajectory:
   ....:     print(ts.time, ts.aux.protein_force[:4])
   ....:
0.0 [    0.        200.71288 -1552.2849   -967.21124]
100.00000762939453 [  100.       -246.27269   146.52911 -1084.2484 ]
200.00001525878906 [nan nan nan nan]
300.0 [nan nan nan nan]
400.0000305175781 [nan nan nan nan]
500.0000305175781 [nan nan nan nan]
600.0 [nan nan nan nan]
700.0000610351562 [nan nan nan nan]
800.0000610351562 [nan nan nan nan]
900.0000610351562 [nan nan nan nan]
```

The trajectory `ProtoReader` methods `next_as_aux()` and `iter_as_aux()` allow for movement through only trajectory timesteps for which auxiliary data is available.

```
In [17]: for ts in u_long.trajectory.iter_as_aux('protein_force'):
   ....:     print(ts.time, ts.aux.protein_force[:4])
   ....:
0.0 [    0.        200.71288 -1552.2849   -967.21124]
100.00000762939453 [  100.       -246.27269   146.52911 -1084.2484 ]
```

This may be used to avoid representative values set to `np.nan`, particularly when auxiliary data is less frequent.

Sometimes the auxiliary data is longer than the trajectory.

```
In [18]: u_short = mda.Universe(PDB)

In [19]: u_short.trajectory.add_auxiliary('protein_force', XVG_BZ2)

In [20]: for ts in u_short.trajectory:
```

(continues on next page)

(continued from previous page)

```
   ....:     print(ts.time, ts.aux.protein_force)
   ....:
0.0 [    0.         200.71288 -1552.2849  ...   128.4072   1386.0378
 -2699.3118 ]
```

In order to acess auxiliary values at every individual step, including those outside the time range of the trajectory, `iter_auxiliary()` allows iteration over the auxiliary independent of the trajectory.

```
In [21]: for step in u_short.trajectory.iter_auxiliary('protein_force'):
   ....:     print(step.data)
   ....:
[    0.         200.71288 -1552.2849  ...   128.4072   1386.0378
 -2699.3118 ]
[   50.       -1082.6454   -658.32166 ...  -493.02954   589.8844
  -739.2124 ]
[  100.        -246.27269   146.52911 ...   484.32501  2332.3767
 -1801.6234 ]
```

To iterate over only a certain section of the auxiliary:

```
In [22]: for step in u_short.trajectory.iter_auxiliary('protein_force', start=1, step=2):
   ....:     print(step.time)
   ....:
50.0
```

The trajectory remains unchanged, and the auxiliary will be returned to the current timestep after iteration is complete.

Accessing auxiliary attributes

To check the values of attributes of an added auxiliary, use `get_aux_attribute()`.

```
In [23]: u.trajectory.get_aux_attribute('protein_force', 'dt')
Out[23]: 50.0
```

If attributes are settable, they can be changed using `set_aux_attribute()`.

```
In [24]: u.trajectory.set_aux_attribute('protein_force', 'data_selector', [1])
```

The auxiliary may be renamed using `set_aux_attribute` with 'auxname', or more directly by using `rename_aux()`.

```
In [25]: u.trajectory.ts.aux.protein_force
Out[25]:
array([    0.      ,   200.71288, -1552.2849 , ...,   128.4072 ,
        1386.0378 , -2699.3118 ])

In [26]: u.trajectory.rename_aux('protein_force', 'f')

In [27]: u.trajectory.ts.aux.f
Out[27]:
array([    0.      ,   200.71288, -1552.2849 , ...,   128.4072 ,
        1386.0378 , -2699.3118 ])
```

Recreating auxiliaries

To recreate an auxiliary, the set of attributes necessary to replicate it can first be obtained with `get_description()`. The returned dictionary can then be passed to `auxreader()` to load a new copy of the original auxiliary reader.

```
In [28]: description = aux.get_description()

In [29]: list(description.keys())
Out[29]:
['represent_ts_as',
 'cutoff',
 'dt',
 'initial_time',
 'time_selector',
 'data_selector',
 'constant_dt',
 'auxname',
 'format',
 'auxdata']

In [30]: del aux

In [31]: mda.auxiliary.core.auxreader(**description)
Out[31]: <MDAnalysis.auxiliary.XVG.XVGReader at 0x7f4c90793250>
```

The 'description' of any or all the auxiliaries added to a trajectory can be obtained using `get_aux_descriptions()`.

```
In [32]: descriptions = u.trajectory.get_aux_descriptions(['f'])
```

To reload, pass the dictionary into `add_auxiliary()`.

```
In [33]: u2 = mda.Universe(PDB, TRR)

In [34]: for desc in descriptions:
   ....:     u2.trajectory.add_auxiliary(**desc)
   ....:
```

EDR Files

EDR files are created by GROMACS during simulations and contain additional non-trajectory time-series data of the system, such as energies, temperature, or pressure. The EDRReader allows direct reading of these binary files and associating of the data with trajectory time steps just like the XVGReader does. As all functionality of the base AuxReaders (see XVGReader above) also works with the EDRReader, this section will highlight functionality unique to the EDRReader.

Standalone Usage

The EDRReader is initialised by passing the path to an EDR file as an argument.

```
In [35]: import MDAnalysis as mda

In [36]: from MDAnalysisTests.datafiles import AUX_EDR

In [37]: aux = mda.auxiliary.EDR.EDRReader(AUX_EDR)

 # Or, for example
 # aux = mda.auxiliary.EDR.EDRReader("path/to/file/ener.edr")
```

Dozens of terms can be defined in EDR files. A list of available data is conveniently available unter the *terms* attribute.

```
In [38]: aux.terms
Out[38]:
['Time',
 'Bond',
 'Angle',
 'Proper Dih.',
 'Ryckaert-Bell.',
 'LJ-14',
 'Coulomb-14',
 'LJ (SR)',
 'Disper. corr.',
 'Coulomb (SR)',
 'Coul. recip.',
 'Potential',
 'Kinetic En.',
 'Total Energy',
 'Conserved En.',
 'Temperature',
 'Pres. DC',
 'Pressure',
 'Constr. rmsd',
 'Box-X',
 'Box-Y',
 'Box-Z',
 'Volume',
 'Density',
 'pV',
 'Enthalpy',
 'Vir-XX',
 'Vir-XY',
 'Vir-XZ',
 'Vir-YX',
 'Vir-YY',
 'Vir-YZ',
 'Vir-ZX',
 'Vir-ZY',
 'Vir-ZZ',
 'Pres-XX',
 'Pres-XY',
```

(continues on next page)

(continued from previous page)

```
 'Pres-XZ',
 'Pres-YX',
 'Pres-YY',
 'Pres-YZ',
 'Pres-ZX',
 'Pres-ZY',
 'Pres-ZZ',
 '#Surf*SurfTen',
 'Box-Vel-XX',
 'Box-Vel-YY',
 'Box-Vel-ZZ',
 'T-Protein',
 'T-non-Protein',
 'Lamb-Protein',
 'Lamb-non-Protein']
```

To extract data for plotting, the *get_data* method can be used. It can be used to extract a single, multiple, or all terms as follows:

```
In [39]: temp = aux.get_data("Temperature")

In [40]: print(temp.keys())
dict_keys(['Time', 'Temperature'])

In [41]: energies = aux.get_data(["Potential", "Kinetic En."])

In [42]: print(energies.keys())
dict_keys(['Time', 'Potential', 'Kinetic En.'])

In [43]: all_data = aux.get_data()

In [44]: print(all_data.keys())
dict_keys(['Time', 'Bond', 'Angle', 'Proper Dih.', 'Ryckaert-Bell.', 'LJ-14', 'Coulomb-14
↪', 'LJ (SR)', 'Disper. corr.', 'Coulomb (SR)', 'Coul. recip.', 'Potential', 'Kinetic␣
↪En.', 'Total Energy', 'Conserved En.', 'Temperature', 'Pres. DC', 'Pressure', 'Constr.␣
↪rmsd', 'Box-X', 'Box-Y', 'Box-Z', 'Volume', 'Density', 'pV', 'Enthalpy', 'Vir-XX',
↪'Vir-XY', 'Vir-XZ', 'Vir-YX', 'Vir-YY', 'Vir-YZ', 'Vir-ZX', 'Vir-ZY', 'Vir-ZZ', 'Pres-
↪XX', 'Pres-XY', 'Pres-XZ', 'Pres-YX', 'Pres-YY', 'Pres-YZ', 'Pres-ZX', 'Pres-ZY',
↪'Pres-ZZ', '#Surf*SurfTen', 'Box-Vel-XX', 'Box-Vel-YY', 'Box-Vel-ZZ', 'T-Protein', 'T-
↪non-Protein', 'Lamb-Protein', 'Lamb-non-Protein'])
```

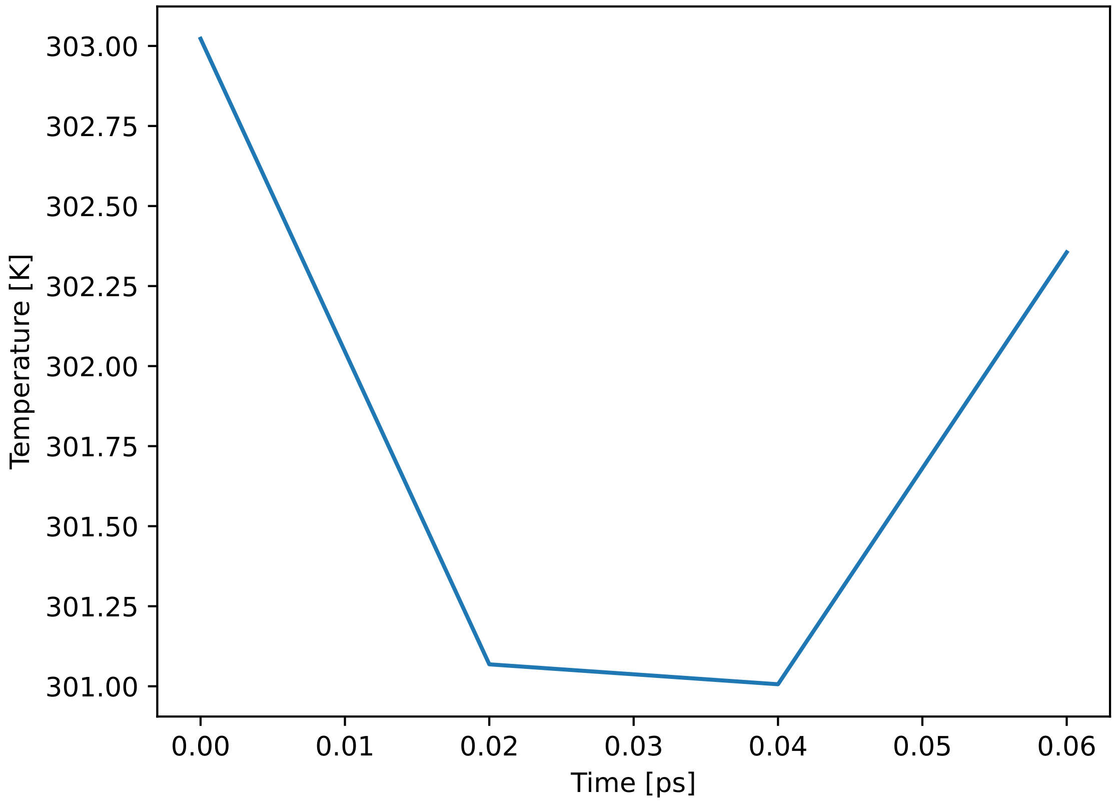
The times of each data point are always part of the returned dictionary to facilicate plotting.

```
In [45]: import matplotlib.pyplot as plt

In [46]: plt.plot(temp["Time"], temp["Temperature"])
Out[46]: [<matplotlib.lines.Line2D at 0x7f4c901991c0>]

In [47]: plt.ylabel("Temperature [K]")
Out[47]: Text(0, 0.5, 'Temperature [K]')

In [48]: plt.xlabel("Time [ps]")
```

(continues on next page)

(continued from previous page)

```
Out[48]: Text(0.5, 0, 'Time [ps]')

In [49]: plt.show()
```



Unit Handling

On object creation, the EDRReader creates a `unit_dict` attribute which contains information on the units of the data stored within. These units are read from the EDR file automatically and by default converted to MDAnalysis base units where such units are defined. The automatic unit conversion can be disabled by setting the `convert_units` kwarg to False.

```
In [50]: aux.unit_dict["Box-X"]
Out[50]: 'A'

In [51]: aux_native = mda.auxiliary.EDR.EDRReader(AUX_EDR, convert_units=False)

In [52]: aux_native.unit_dict["Box-X"]
Out[52]: 'nm'
```

Use with Trajectories

An arbitrary number of terms to be associated with a trajectory can be specified as a dictionary. The dictionary is chosen so that the name to be used in MDAnalysis to access the data is mapped to the name in *aux.terms*.

```
In [53]: from MDAnalysisTests.datafiles import AUX_EDR_TPR, AUX_EDR_XTC

In [54]: u = mda.Universe(AUX_EDR_TPR, AUX_EDR_XTC)

In [55]: term_dict = {"temp": "Temperature", "epot": "Potential"}

In [56]: u.trajectory.add_auxiliary(term_dict, aux)

In [57]: u.trajectory.ts.aux.epot
Out[57]: -525164.0625
```

Adding all data is possible by omitting the dictionary as follows. It is then necessary to specify that the EDRReader should be passed as the *auxdata* argument as such:

```
In [58]: u = mda.Universe(AUX_EDR_TPR, AUX_EDR_XTC)

In [59]: u.trajectory.add_auxiliary(auxdata=aux)

In [60]: u.trajectory.aux_list
Out[60]: dict_keys(['Time', 'Bond', 'Angle', 'Proper Dih.', 'Ryckaert-Bell.', 'LJ-14',
↪'Coulomb-14', 'LJ (SR)', 'Disper. corr.', 'Coulomb (SR)', 'Coul. recip.', 'Potential',
↪'Kinetic En.', 'Total Energy', 'Conserved En.', 'Temperature', 'Pres. DC', 'Pressure',
↪'Constr. rmsd', 'Box-X', 'Box-Y', 'Box-Z', 'Volume', 'Density', 'pV', 'Enthalpy', 'Vir-
↪XX', 'Vir-XY', 'Vir-XZ', 'Vir-YX', 'Vir-YY', 'Vir-YZ', 'Vir-ZX', 'Vir-ZY', 'Vir-ZZ',
↪'Pres-XX', 'Pres-XY', 'Pres-XZ', 'Pres-YX', 'Pres-YY', 'Pres-YZ', 'Pres-ZX', 'Pres-ZY',
↪ 'Pres-ZZ', '#Surf*SurfTen', 'Box-Vel-XX', 'Box-Vel-YY', 'Box-Vel-ZZ', 'T-Protein', 'T-
↪non-Protein', 'Lamb-Protein', 'Lamb-non-Protein'])
```

Selecting Trajectory Frames Based on Auxiliary Data

One use case for the new auxiliary readers is the selection of frames based on auxiliary data. To select only those frames with a potential energy below a certain threshold, the following can be used:

```
In [61]: u = mda.Universe(AUX_EDR_TPR, AUX_EDR_XTC)

In [62]: term_dict = {"epot": "Potential"}

In [63]: u.trajectory.add_auxiliary(term_dict, aux)

In [64]: selected_frames = np.array([ts.frame for ts in u.trajectory if ts.aux.epot < -
↪524600])
```

A slice of the trajectory can then be obtained from the list of frames as such:

```
In [65]: trajectory_slice = u.trajectory[selected_frames]

In [66]: print(len(u.trajectory))
4
```

(continues on next page)

(continued from previous page)

```
In [67]: print(len(trajectory_slice))
2
```

Memory Usage

It is assumed that the EDR file is small enough to be read in full. However, since one EDRReader instance is created for each term added to a trajectory, memory usage monitoring was implemented. A warning will be issued if 1 GB of memory is used by auxiliary data. This warning limit can optionally be changed by defining the *memory_limit* (in bytes) when adding data to a trajectory. Below, the memory limit is set to 200 MB.

```
In [68]: u = mda.Universe(AUX_EDR_TPR, AUX_EDR_XTC)

In [69]: term_dict = {"temp": "Temperature", "epot": "Potential"}

In [70]: u.trajectory.add_auxiliary(term_dict, aux, memory_limit=2e+08)
```

2.1.19 Selection exporters

Selection exporters allow you to write a selection of atoms to a file that can be read by another program.

Writing selections

Single AtomGroup

The typical situation is that one has an AtomGroup and wants to work with the same selection of atoms in a different package, for example, to visualize the atoms in VMD.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PDB

In [3]: u = mda.Universe(PDB)

In [4]: ag = u.select_atoms('resname ALA')
```

As with a normal structure file, use AtomGroup.write method with the appropriate file extension.

```
ag.write("ala_selection.vmd", name="alanine")
```

In VMD, sourcing the file ala_selection.vmd (written in Tcl) defines the “macro” alanine that contains the atom indices to select.

```
source ala_selection.vmd
set sel [atomselect top alanine]
```

and in the GUI the macro appears in the *Graphics* → *Representations* window in the list *Selections: Singlewords* as “alanine”.

Names are not always required; if name is not passed to AtomGroup.write, MDAnalysis defaults to “mdanalysis001”, “mdanalysis002”, and so on.

Multiple selections

AtomGroup.write can take additional keyword arguments, including mode. The default is mode='w', which will overwrite the provided filename. If mode='a', the selection is appended to the file.

```
u.select_atoms('resname T*').write('residues.ndx',
                                   name='TYR_THR',
                                   mode='a')
u.select_atoms('resname GLY').write('residues.ndx',
                                    name='GLY',
                                    mode='a')
u.select_atoms('resname PRO').write('residues.ndx',
                                    name='PRO',
                                    mode='a')
```

Looking at this GROMACS index file, we see:

```
$ gmx make_ndx -n residues.ndx

Command line:
gmx make_ndx -n residues.ndx

Going to read 1 old index file(s)
Counted atom numbers up to 3341 in index file

0 TYR_THR             :   301 atoms
1 GLY                 :   141 atoms
2 PRO                 :   140 atoms

nr : group      '!': not  'name' nr name   'splitch' nr    Enter: list groups
'a': atom       '&': and  'del' nr         'splitres' nr   'l': list residues
't': atom type  '|': or   'keep' nr        'splitat' nr    'h': help
'r': residue              'res' nr         'chain' char
"name": group             'case': case sensitive           'q': save and quit
'ri': residue index
```

Alternatively, you can direcly use the selection writer itself as a context manager and write each AtomGroup inside the context. For example:

```
with mda.selections.gromacs.SelectionWriter('residues.ndx', mode='w') as ndx:
    ndx.write(u.select_atoms('resname T*'),
              name='TYR_THR')
    ndx.write(u.select_atoms('resname GLY'),
              name='GLY')
```

And again, you can append to the file with mode='a':

```
with mda.selections.gromacs.SelectionWriter('residues.ndx', mode='a') as ndx:
    ndx.write(u.select_atoms('resname PRO'),
              name='PRO')
```

Reading in selections

Currently, MDAnalysis doesn't support reading in atom selections. However, there are other tools that can read files from other programs, such as GromacsWrapper.

2.1.20 Format reference

chemfiles (chemfiles Trajectory or file)

The chemfiles library provides a C++ implementation of readers and writers for multiple formats. Pass in either a `chemfiles.Trajectory` to be converted into an MDAnalysis Universe, or pass in files to this format with the keyword `format='CHEMFILES'` to read the information with the chemfiles implementation. You can also write the MDAnalysis Universe back into a chemfiles object for further work.

CONFIG (DL_Poly Config)

HISTORY (DL_Poly Config)

MDAnalysis can read information both from DL Poly config and DL Poly history files. Although DL Poly input file units can be flexible, output files appear to have the following units:

- Time: ps
- Length: Angstrom
- Mass: amu (Dalton)
- Velocity: Angstrom/ps
- Force: Angstrom Dalton / ps ^{2}

MDAnalysis currently does not convert these into the native kJ/(mol A) force units when reading files in. See Issue 2393 for discussion on units.

COOR, NAMBDIN (NAMD binary restart files)

You can read or write coordinates from the NAMD double-precision binary format .

CRD (CHARMM CARD files)

Reading in

Read a list of atoms from a CHARMM standard or extended CARD coordinate file (CRD) to build a basic topology. Reads atom ids (ATOMNO), atom names (TYPES), resids (RESID), residue numbers (RESNO), residue names (RESNames), segment ids (SEGID) and tempfactor (Weighting). Atom element and mass are guessed based on the name of the atom.

Writing out

MDAnalysis automatically writes the CHARMM EXT extended format if there are more than 99,999 atoms.

Writing a CRD file format requires the following attributes to be present:

- resids
- resnames
- names
- chainIDs
- tempfactors

If these are not present, then *default values* are provided and a warning is raised.

DATA (LAMMPS)

Important: Lennard-Jones units are not implemented. See *Units and constants* for other recognized values and the documentation for the LAMMPS units command.

Reading in

Lammps atoms can have lots of different formats, and even custom formats. By default, MDAnalysis checks:

- "full" : atoms with 7 fields (reading id, resid, type, and charge)
- "molecular": atoms with 6 fields (reading id, resid, and type)

Users can pass in their own `atom_style` specifications.

- Required fields: id, type, x, y, z
- Optional fields: resid, charge

For example:

```
u = mda.Universe(LAMMPSDATA, atom_style="id resid type charge element bfactor occupancy␣
↪x y z")
```

Only id, resid, charge, type, and coordinate information will be read from the file, even if other topology attributes are specified in the `atom_style` argument.

Writing out

MDAnalysis supports writing out the header and applicable sections from Atoms, Masses, Velocities, Bonds, Angles, Dihedrals, and Impropers. The Atoms section is written in the "full" sub-style if charges are available or "molecular" sub-style if they are not. The molecule id is set to 0 for all atoms.

This writer assumes "conventional" or "real" LAMMPS units where length is measured in Angstroms and velocity is measured in Angstroms per femtosecond. To write in different units, specify `lengthunit` or `timeunit`.

For example, to write a certain frame with nanometer units:

```
>>> for ts in u.trajectory:
...     # analyze frame
...     if take_this_frame == True:
...         with mda.Writer('frame.data') as W:
...             W.write(u.atoms, lengthunit="nm")
...         break
```

If atom types are not already positive integers, the user must set them to be positive integers, because the writer will not automatically assign new types.

To preserve numerical atom types when writing a selection, the Masses section will have entries for each atom type up to the maximum atom type. If the universe does not contain atoms of some type in `{1, ... max(atom_types)}`, then the mass for that type will be set to 1.

In order to write bonds, each selected bond type must be explicitly set to an integer >= 1.

DCD (CHARMM, NAMD, or LAMMPS trajectory)

DCD is used by NAMD, CHARMM and LAMMPS as the default trajectory format.

Reading in

Unitcell dimensions

Generally, DCD trajectories produced by any code can be read (with the `DCDReader`) although there can be issues with the unitcell dimensions (simulation box). Currently, MDAnalysis tries to guess the correct **format for the unitcell representation** but it can be wrong. **Check the unitcell dimensions**, especially for triclinic unitcells (see Issue 187).

MDAnalysis always uses `(*A*, *B*, *C*, *alpha*, *beta*, *gamma*)` to represent the unit cell. Lengths A, B, C are in the MDAnalysis length unit (Å), and angles are in degrees.

The ordering of the angles in the unitcell is the same as in recent versions of VMD's DCDplugin (2013), namely the X-PLOR DCD format: The original unitcell is read as `[A, gamma, B, beta, alpha, C]` from the DCD file. If any of these values are < 0 or if any of the angles are > 180 degrees then it is assumed it is a new-style CHARMM unitcell (at least since c36b2) in which box vectors were recorded.

Important: Check your unit cell dimensions carefully, especially when using triclinic boxes. Old CHARMM trajectories might give wrong unitcell values.

Units

The DCD file format is not well defined. In particular, NAMD and CHARMM use it differently. DCD trajectories produced by CHARMM and NAMD(>2.5) record time in AKMA units. If other units have been recorded (e.g., ps) then employ the configurable *LAMMPS DCD format* and set the time unit as an optional argument. You can find a list of units used in the DCD formats on the MDAnalysis wiki.

Writing out

The writer follows recent NAMD/VMD convention for the unitcell (box lengths in Å and angle-cosines, `[A, cos(gamma), B, cos(beta), cos(alpha), C]`). It writes positions in Å and time in AKMA time units.

Reading and writing these trajectories within MDAnalysis will work seamlessly. However, if you process those trajectories with other tools, you need to watch out that time and unitcell dimensions are correctly interpreted.

DCD (Flexible LAMMPS trajectory)

LAMMPS can write DCD trajectories but unlike a CHARMM trajectory (which is often called a DCD, even though CHARMM itself calls them "trj") the time unit is not fixed to be the AKMA time unit but can depend on settings in LAMMPS. The most common case for biomolecular simulations appears to be that the time step is recorded in femtoseconds (command units real in the input file) and lengths in ångströms. Other cases are unit-less Lennard-Jones time units.

This presents a problem for MDAnalysis, because it cannot autodetect the unit from the file. By default, we assume that the unit for length is the ångström and the unit for time is the femtosecond. If this is not true, then the user *should supply the appropriate units* in the keywords `timeunit` and/or `lengthunit` to `DCDWriter` and `Universe` (which then calls `DCDReader`).

DMS (Desmond Molecular Structure files)

The DESRES Molecular Structure (DMS) file is an SQLite-format database for storing coordinate and topology information. See the Desmond Users Guide (chapter 6 and chapter 17) for more information.

Important: Atom ids

Unlike most other file formats, Desmond starts atom numbers at 0. This means the first atom in a DMS file will have an `Atom.id` of 0. However, residues are not necessarily numbered from 0. `Residue.resid` numbering can start from 1.

GMS (Gamess trajectory)

The GMS output format is a common output format for different GAMESS distributions: GAMESS-US, Firefly (PC-GAMESS) and GAMESS-UK. The current version has been tested with US GAMESS and Firefly only.

Reading in

MDAnalysis can read a GAMESS output file and pull out atom information. Atom names are their elements. Information about residues and segments is not read.

GRO (GROMACS structure file)

GRO files provide topology, coordinate, and sometimes velocity information.

Reading in

Prior to MDAnalysis version 0.21.0 and GROMACS 2019.5, MDAnalysis failed to parse GRO files with box sizes where an axis length was longer than 10 characters.

Important: A Universe created with a GRO file and a Universe created with a corresponding TPR file will have *different atom and residue numbering*, due to how a TPR file is parsed. **This behaviour will change in 2.0.0 where TPR parsing will be made consistent with the other file formats.**

Writing out

AtomGroups can be written out to a GRO file. However, this format does not support multi-frame trajectories.

GSD (HOOMD GSD file)

The HOOMD schema GSD file format can contain both topology and trajectory information (output of HOOMD-blue).

Reading in

Important: The GSD format was developed to support changing numbers of particles, particle types, particle identities and topologies. However, MDAnalysis currently does not support changing topologies. Therefore, the MDAnalysis reader should only be used for trajectories that keep the particles and topologies fixed.

A user will only get an error if the number of particles changes from the first time step. MDAnalysis does not currently check for changes in the particle identity or topology, and it does not update these over the trajectory.

Note: Residue resnames

Unlike other formats, MDAnalysis treats residue `resnames` from GSD files as integers. These are identical to `resids` and `resnums`.

IN, FHIAIMS (FHI-aims input files)

FHI-aims input files are similar to *xyz* in format. The specification is publicly available.

INPCRD, RESTRT (AMBER restart files)

MDAnalysis can read coordinates in Amber coordinate/restart files (suffix "inpcrd").

ITP (GROMACS portable topology files)

A ITP file is a portable topology file.

Important: Unlike *TPR* files, atom `ids` and residues `resids` in ITP files are indexed from 1. This means that a TPR file created from your ITP files will have *different* numbering in MDAnalysis than the ITP file.

LAMMPSDUMP (LAMMPS ascii dump file)

Reading in

MDAnalysis expects ascii dump files to be written with the default LAMMPS dump format of 'atom'.It will automatically convert positions from their scaled/fractional representation to their real values.

Important: Lennard-Jones units are not implemented. See *Units and constants* for other recognized values and the documentation for the LAMMPS units command.

MMTF (Macromolecular Transmission Format)

The Macromolecular Transmission Format format (MMTF) should generally be a quicker alternative to *PDB*.

Individual models within the MMTF file are available via the `models` attribute of Universe.

MOL2 (Tripos structure)

The Tripos molecule structure format (MOL2) is a commonly used format. It is used, for instance, by the DOCK docking code.

Warning: `MOL2Writer` can only be used to write out previously loaded MOL2 files. For example, if you're trying to convert a PDB file to MOL2, you should use other tools such as rdkit.

Here is an example how to use rdkit to convert a PDB to MOL:

```
from rdkit import Chem
mol = Chem.MolFromPDBFile("molecule.pdb", removeHs=False)
Chem.MolToMolFile(mol, "molecule.mol" )
```

MOL2 specification

- Example file:

```
 #    Name: benzene
 #    Creating user name: tom
 #    Creation time: Wed Dec 28 00:18:30 1988

 #    Modifying user name: tom
 #    Modification time: Wed Dec 28 00:18:30 1988

 @<TRIPOS>MOLECULE
 benzene
 12 12 1  0   0
 SMALL
 NO_CHARGES


 @<TRIPOS>ATOM
 1   C1  1.207   2.091   0.000   C.ar    1   BENZENE 0.000
 2   C2  2.414   1.394   0.000   C.ar    1   BENZENE 0.000
 3   C3  2.414   0.000   0.000   C.ar    1   BENZENE 0.000
 4   C4  1.207   -0.697  0.000   C.ar    1   BENZENE 0.000
 5   C5  0.000   0.000   0.000   C.ar    1   BENZENE 0.000
 6   C6  0.000   1.394   0.000   C.ar    1   BENZENE 0.000
 7   H1  1.207   3.175   0.000   H   1   BENZENE 0.000
 8   H2  3.353   1.936   0.000   H   1   BENZENE 0.000
 9   H3  3.353   -0.542  0.000   H   1   BENZENE 0.000
 10  H4  1.207   -1.781  0.000   H   1   BENZENE 0.000
 11  H5  -0.939  -0.542  0.000   H   1   BENZENE 0.000
 12  H6  -0.939  1.936   0.000   H   1   BENZENE 0.000
 @<TRIPOS>BOND
 1   1   2   ar
 2   1   6   ar
 3   2   3   ar
 4   3   4   ar
 5   4   5   ar
 6   5   6   ar
 7   1   7   1
 8   2   8   1
 9   3   9   1
 10  4   10  1
 11  5   11  1
 12  6   12  1
@<TRIPOS>SUBSTRUCTURE
 1   BENZENE 1   PERM    0   ****    ****    0   ROOT
```

NCDF, NC (AMBER NetCDF trajectory)

AMBER binary trajectories are automatically recognised by the file extension ".ncdf". The NCDF module uses `scipy.io.netcdf` and therefore `scipy` must be installed.

Reading in

Units are assumed to be the following default AMBER units:

- length: Angstrom
- time: ps

Currently, if other units are detected, MDAnalysis will raise a `NotImplementedError`.

Writing out

NCDF files are always written out in ångström and picoseconds.

Although scale_factors can be read from NCDF files, they are not kept or used when writing NCDF files out.

Writing with the netCDF4 module and potential issues

> Although `scipy.io.netcdf` is very fast at reading NetCDF files, it is slow at writing them out. The netCDF4 package is fast at writing (but slow at reading). This requires the compiled netcdf library to be installed. MDAnalysis tries to use `netCDF4` for writing if it is available, but will fall back to `scipy.io.netcdf` if it is not.
>
> **AMBER users** might have a hard time getting netCDF4 to work with a conda-based installation (as discussed in Issue #506) because of the way that AMBER itself handles netcdf. In this scenario, MDAnalysis will simply switch to the scipy package. If you encounter this error and wish to use the the faster `netCDF4` writer, the only solution is to unload the AMBER environment.

ParmEd (ParmEd Structure)

The ParmEd library is a general tool for molecular modelling, often used to manipulate system topologies or convert between file formats. You can pass in a `parmed.Structure` to be converted into an MDAnalysis Universe, and convert it back using a `ParmEdConverter`. While you can convert Universes created by other means (e.g. by reading files) into a ParmEd structure, MDAnalysis does not read or generate details such the "type" of a bond (i.e. bondtype), or information such as `ureybradley` terms.

PDB, ENT (Standard PDB file)

Reading in

MDAnalysis parses the following *PDB records* (see PDB coordinate section for details):

- **CRYST1** for unit cell dimensions A,B,C, alpha,beta,gamma
- **ATOM** or **HETATM** for serial, name, resName, chainID, resSeq, x, y, z, occupancy, tempFactor, segID
- **CONECT** records for bonds
- **HEADER** (`Universe.trajectory.header`)
- **TITLE** (`Universe.trajectory.title`)

- **COMPND** (`Universe.trajectory.compound`)
- **REMARK** (`Universe.trajectory.remarks`)

All other lines are ignored. Multi-MODEL PDB files are read as trajectories with a default timestep of 1 ps *(pass in the dt argument to change this)*. Currently, MDAnalysis cannot read multi-model PDB files written by VMD, as VMD uses the keyword "END" to separate models instead of "MODEL"/"ENDMDL" keywords.

Important: Previously, MDAnalysis did not read elements from a file. Now, if valid elements are provided, MDAnalysis will read them in and will *not* guess them from atom names.

MDAnalysis attempts to read `segid` attributes from the *segID* column. If this column does not contain information, segments are instead created from chainIDs. If chainIDs are also not present, then `segid`s are set to the default `'SYSTEM'` value.

Writing out

MDAnalysis can write both single-frame PDBs and convert trajectories to multi-model PDBs. If the Universe is missing fields that are *required in a PDB file*, MDAnalysis provides default values and raises a warning. There are 2 exceptions to this:

- `chainIDs`: if a Universe does not have `chainIDs`, MDAnalysis uses the first character of the segment `segid` instead.
- `elements`: Elements are *always* guessed from the atom name.

These are the default values:

- names: 'X'
- altLocs: ''
- resnames: 'UNK'
- icodes: ''
- segids: ''
- resids: 1
- occupancies: 1.0
- tempfactors: 0.0

PDB specification

Table 2: CRYST1 fields

COLUMNS	DATA TYPE	FIELD	DEFINITION
1 - 6	Record name	"CRYST1"	
7 - 15	Real(9.3)	a	a (Angstroms).
16 - 24	Real(9.3)	b	b (Angstroms).
25 - 33	Real(9.3)	c	c (Angstroms).
34 - 40	Real(7.2)	alpha	alpha (degrees).
41 - 47	Real(7.2)	beta	beta (degrees).
48 - 54	Real(7.2)	gamma	gamma (degrees).

Table 3: ATOM/HETATM fields

COLUMNS	DATA TYPE	FIELD	DEFINITION
1 - 6	Record name	“ATOM “	
7 - 11	Integer	serial	Atom serial number.
13 - 16	Atom	name	Atom name.
17	Character	altLoc	Alternate location indicator.
18 - 21	Residue name	resName	Residue name.
22	Character	chainID	Chain identifier.
23 - 26	Integer	resSeq	Residue sequence number.
27	AChar	iCode	Code for insertion of residues.
31 - 38	Real(8.3)	x	Orthogonal coordinates for X in Angstroms.
39 - 46	Real(8.3)	y	Orthogonal coordinates for Y in Angstroms.
47 - 54	Real(8.3)	z	Orthogonal coordinates for Z in Angstroms.
55 - 60	Real(6.2)	occupancy	Occupancy.
61 - 66	Real(6.2)	tempFactor	Temperature factor.
67 - 76	String	segID	(unofficial CHARMM extension ?)
77 - 78	LString(2)	element	Element symbol, right-justified.
79 - 80	LString(2)	charge	Charge on the atom.

PDBQT (Autodock structure)

Reading in

MDAnalysis reads coordinates from PDBQT files and additional optional data such as B-factors, partial charge and AutoDock atom types. It is also possible to substitute a PDBQT file for a PSF file in order to define the list of atoms (but no connectivity information will be available in this case).

Although PDBQT is a similar file format to PDB, MDAnalysis treats them with several differences:

- Multi-model PDBQT files are not supported
- Connectivity is not supported (i.e. bonds are not read)

Writing out

MDAnalysis implements a subset of the PDB 3.2 standard and the PDBQT spec. Unlike the *PDB, ENT (Standard PDB file)* writer, MDAnalysis cannot write multi-frame trajectories to a PDBQT file.

If the Universe is missing fields that are *required in a PDBQT file*, MDAnalysis provides default values and raises a warning. There are 2 exceptions to this:

- `chainIDs`: if a Universe does not have `chainIDs`, MDAnalysis will use the `segids` instead.
- `elements`: MDAnalysis uses the atom type as the element.

These are the default values:

- names: ‘X’
- altLocs: ‘’
- resnames: ‘UNK’
- icodes: ‘’
- segids: ‘’

- resids: 1
- occupancies: 1.0
- tempfactors: 0.0
- types (elements): ‘’
- charges: 0.0

PDBQT specification

Records read:

- **CRYST1** for unit cell dimensions A,B,C, alpha,beta,gamma
- **ATOM** or **HETATM** for serial, name, resName, chainID, resSeq, x, y, z, occupancy, tempFactor, segID

Table 4: PDB format with AutoDOCK extensions for the PDBQT format.

COLUMNS	DATA TYPE	FIELD	DEFINITION
1 - 6	Record name	“CRYST1”	
7 - 15	Real(9.3)	a	a (Angstroms).
16 - 24	Real(9.3)	b	b (Angstroms).
25 - 33	Real(9.3)	c	c (Angstroms).
34 - 40	Real(7.2)	alpha	alpha (degrees).
41 - 47	Real(7.2)	beta	beta (degrees).
48 - 54	Real(7.2)	gamma	gamma (degrees).
1 - 6	Record name	“ATOM “	
7 - 11	Integer	serial	Atom serial number.
13 - 16	Atom	name	Atom name.
17	Character	altLoc	Alternate location indicator. IGNORED
18 - 21	Residue name	resName	Residue name.
22	Character	chainID	Chain identifier.
23 - 26	Integer	resSeq	Residue sequence number.
27	AChar	iCode	Code for insertion of residues. IGNORED
31 - 38	Real(8.3)	x	Orthogonal coordinates for X in Angstroms.
39 - 46	Real(8.3)	y	Orthogonal coordinates for Y in Angstroms.
47 - 54	Real(8.3)	z	Orthogonal coordinates for Z in Angstroms.
55 - 60	Real(6.2)	occupancy	Occupancy.
61 - 66	Real(6.2)	tempFactor	Temperature factor.
67 - 70	LString(4)	footnote	Usually blank. IGNORED.
71 - 76	Real(6.4)	partialChrg	Gasteiger PEOE partial charge *q*.
79 - 80	LString(2)	atomType	AutoDOCK atom type *t*.

We ignore torsion notation and just pull the partial charge and atom type columns:

```
COMPND    NSC7810
REMARK  3 active torsions:
REMARK  status: ('A' for Active; 'I' for Inactive)
REMARK    1  A    between atoms: A7_7  and  C22_23
REMARK    2  A    between atoms: A9_9  and  A11_11
REMARK    3  A    between atoms: A17_17  and  C21_21
ROOT
123456789.123456789.123456789.123456789.123456789.123456789.123456789.123456789. (column
```

(continues on next page)

(continued from previous page)

```
↪reference)
ATOM      1  A1  INH I           1.054   3.021   1.101  0.00  0.00     0.002 A
ATOM      2  A2  INH I           1.150   1.704   0.764  0.00  0.00     0.012 A
ATOM      3  A3  INH I          -0.006   0.975   0.431  0.00  0.00    -0.024 A
ATOM      4  A4  INH I           0.070  -0.385   0.081  0.00  0.00     0.012 A
ATOM      5  A5  INH I          -1.062  -1.073  -0.238  0.00  0.00     0.002 A
ATOM      6  A6  INH I          -2.306  -0.456  -0.226  0.00  0.00     0.019 A
ATOM      7  A7  INH I          -2.426   0.885   0.114  0.00  0.00     0.052 A
ATOM      8  A8  INH I          -1.265   1.621   0.449  0.00  0.00     0.002 A
ATOM      9  A9  INH I          -1.339   2.986   0.801  0.00  0.00    -0.013 A
ATOM     10  A10 INH I          -0.176   3.667   1.128  0.00  0.00     0.013 A
ENDROOT
BRANCH   9  11
ATOM     11  A11 INH I          -2.644   3.682   0.827  0.00  0.00    -0.013 A
ATOM     12  A16 INH I          -3.007   4.557  -0.220  0.00  0.00     0.002 A
ATOM     13  A12 INH I          -3.522   3.485   1.882  0.00  0.00     0.013 A
ATOM     14  A15 INH I          -4.262   5.209  -0.177  0.00  0.00    -0.024 A
ATOM     15  A17 INH I          -2.144   4.784  -1.319  0.00  0.00     0.052 A
ATOM     16  A14 INH I          -5.122   4.981   0.910  0.00  0.00     0.012 A
ATOM     17  A20 INH I          -4.627   6.077  -1.222  0.00  0.00     0.012 A
ATOM     18  A13 INH I          -4.749   4.135   1.912  0.00  0.00     0.002 A
ATOM     19  A19 INH I          -3.777   6.285  -2.267  0.00  0.00     0.002 A
ATOM     20  A18 INH I          -2.543   5.650  -2.328  0.00  0.00     0.019 A
BRANCH  15  21
ATOM     21  C21 INH I          -0.834   4.113  -1.388  0.00  0.00     0.210 C
ATOM     22  O1  INH I          -0.774   2.915  -1.581  0.00  0.00    -0.644 OA
ATOM     23  O3  INH I           0.298   4.828  -1.237  0.00  0.00    -0.644 OA
ENDBRANCH  15  21
ENDBRANCH   9  11
BRANCH   7  24
ATOM     24  C22 INH I          -3.749   1.535   0.125  0.00  0.00     0.210 C
ATOM     25  O2  INH I          -4.019   2.378  -0.708  0.00  0.00    -0.644 OA
ATOM     26  O4  INH I          -4.659   1.196   1.059  0.00  0.00    -0.644 OA
ENDBRANCH   7  24
TORSDOF 3
123456789.123456789.123456789.123456789.123456789.123456789.123456789.123456789. (column␣
↪reference)
```

PQR file (PDB2PQR / APBS)

MDAnalysis can read classes from a PQR file (as written by PDB2PQR). Parsing is adopted from the description of the PQR format as used by APBS.

> **Warning:** Fields *must be white-space separated* or data are read incorrectly. PDB formatted files are *not* guaranteed to be white-space separated so extra care should be taken when quickly converting PDB files into PQR files using simple scripts.

For example, PQR files created with PDB2PQR and the *–whitespace* option are guaranteed to conform to the above format:

```
pdb2pqr --ff=charmm --whitespace 4ake.pdb 4ake.pqr
```

Reading in

MDAnalysis reads data on a per-line basis from PQR files using the following format:

```
recordName serial atomName residueName chainID residueNumber X Y Z charge radius
```

If this fails it is assumed that the *chainID* was omitted and the shorter format is read:

```
recordName serial atomName residueName residueNumber X Y Z charge radius
```

Anything else will raise a `ValueError`.

The whitespace is the most important feature of this format: fields *must* be separated by at least one space or tab character.

Writing out

Charges ("Q") are taken from the `MDAnalysis.core.groups.Atom.charge` attribute while radii are obtained from the `MDAnalysis.core.groups.Atom.radius` attribute.

- If the segid is 'SYSTEM' then it will be set to the empty string. Otherwise the first letter will be used as the chain ID.
- The serial number always starts at 1 and increments sequentially for the atoms.

The output format is similar to `pdb2pqr --whitespace`.

Output should look like this (although the only real requirement is *whitespace* separation between *all* entries). The chainID is optional and can be omitted:

```
ATOM      1  N    MET     1      -11.921   26.307   10.410 -0.3000 1.8500
ATOM     36  NH1  ARG     2       -6.545   25.499    3.854 -0.8000 1.8500
ATOM     37 HH11  ARG     2       -6.042   25.480    4.723  0.4600 0.2245
```

PQR specification

The PQR fields read are:

recordName
A string which specifies the type of PQR entry and should either be ATOM or HETATM.

serial
An integer which provides the atom index (but note that MDAnalysis renumbers atoms so one cannot rely on the *serial*)

atomName
A string which provides the atom name.

residueName
A string which provides the residue name.

chainID
An optional string which provides the chain ID of the atom.

residueNumber
An integer which provides the residue index.

X Y Z
Three floats which provide the atomic coordiantes.

charge
A float which provides the atomic charge (in electrons).

radius
A float which provides the atomic radius (in Å).

Clearly, this format can deviate wildly from PDB due to the use of whitespaces rather than specific column widths and alignments. This deviation can be particularly significant when large coordinate values are used.

PSF (CHARMM, NAMD, or XPLOR protein structure file)

A protein structure file (PSF) contains topology information for CHARMM, NAMD, and XPLOR. The MDAnalysis PSFParser only reads information about atoms, bonds, angles, dihedrals, and impropers. While PSF files can include information on hydrogen-bond donor and acceptor groups, MDAnalysis does not read these in.

Important: Atom ids

Although PSF files index atoms from 1 in the file, the MDAnalysis PSFParser subtracts 1 to create atom `ids`. This means that if your atom is numbered 3 in your PSF file, it will have an `Atom.id` of 2 in MDAnalysis.

Atom `indices` are MDAnalysis derived and always index from 0, no matter the file type.

Reading in

PSF files can come in a number of "flavours": STANDARD, EXTENDED, and NAMD. If your file is not a standard file, it must have a NAMD or EXT flag to tell MDAnalysis to how to *parse the atom section*.

As a NAMD file is space-separated, files with missing columns can cause MDAnalysis to read information incorrectly. This can cause issues for PSF files written from VMD.

PSF files can encode insertion codes. However, MDAnalysis does not currently support reading PSF files with insertion codes.

PSF specification

CHARMM

Normal (standard) and extended (EXT) PSF format are supported. CHEQ is supported in the sense that CHEQ data is simply ignored.

CHARMM Format from `source/psffres.src`:

CHEQ:

```
II,LSEGID,LRESID,LRES,TYPE(I),IAC(I),CG(I),AMASS(I),IMOVE(I),ECH(I),EHA(I)

standard format:
(I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,I4,1X,2G14.6,I8,2G14.6)
```

(continues on next page)

(continued from previous page)

```
(I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,2G14.6,I8,2G14.6)  XPLOR
expanded format EXT:
(I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,I4,1X,2G14.6,I8,2G14.6)
(I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,A4,1X,2G14.6,I8,2G14.6) XPLOR
```

no CHEQ:

```
II,LSEGID,LRESID,LRES,TYPE(I),IAC(I),CG(I),AMASS(I),IMOVE(I)

standard format:
(I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,I4,1X,2G14.6,I8)
(I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,2G14.6,I8)  XPLOR
expanded format EXT:
(I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,I4,1X,2G14.6,I8)
(I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,A4,1X,2G14.6,I8) XPLOR
```

NAMD

This format is space separated (see the release notes for VMD 1.9.1, psfplugin).

TNG (Trajectory Next Generation)

TNG (The Next Generation) is a highly flexible and high performance trajectory file format for molecular simulations, particularly designed for GROMACS. It includes support for storing both reduced precision coordinates and a lossless, full precision version simultaneously. This enables high quality scientific analysis alongside long term storage in a format that retains full scientific integrity. TNG supports an arbitrary number of data blocks of various types to be added to the file.

Reading in

MDAnalysis supports reading of TNG files. The TNGReader uses the GROMACS tng library for reading the binary TNG format.

The TNG format includes high level C-style API functions for increased ease of use.

TOP, PRMTOP, PARM7 (AMBER topology)

AMBER specification

Note: The Amber charge is converted to electron charges as used in MDAnalysis and other packages. To get back Amber charges, multiply by 18.2223.

Table 5: Attributes parsed from AMBER keywords

AMBER flag	MDAnalysis attribute
ATOM_NAME	names
CHARGE	charges
ATOMIC_NUMBER	elements
MASS	masses
BONDS_INC_HYDROGEN BONDS_WITHOUT_HYDROGEN	bonds
ANGLES_INC_HYDROGEN ANGLES_WITHOUT_HYDROGEN	angles
DIHEDRALS_INC_HYDROGEN DIHEDRALS_WITHOUT_HYDROGEN	dihedrals / improper
ATOM_TYPE_INDEX	type_indices
AMBER_ATOM_TYPE	types
RESIDUE_LABEL	resnames
RESIDUE_POINTER	residues

Developer notes

The format is defined in PARM parameter/topology file specification. The reader tries to detect if it is a newer (AMBER 12?) file format by looking for the flag “ATOMIC_NUMBER”.

TPR (GROMACS run topology files)

A GROMACS TPR file is a portable binary run input file. It contains both topology and coordinate information. However, MDAnalysis currently only reads topology information about atoms, bonds, dihedrals, and impropers; it does not read the coordinate information.

Important: Atom ids, residue resids, and molnums

GROMACS indexes atom numbers and residue numbers from 1 in user-readable files. However, the MDAnalysis TPRParser creates atom `ids` and residue `resids` *from 0*. This means that if your atom is numbered 3 in your GRO, ITP, or TOP file, it will have an `Atom.id` of 2 in MDAnalysis. Likewise, if your residue ALA has a residue number of 4 in your GRO file, it will have a `Residue.resid` number of 3 in MDAnalysis. Finally, molecules are also numbered from 0, in the attribute `molnums`.

This will change in version 2.0.0. The TPRParser will number resids, ids, etc. from 1 to be consistent with other formats.

Atom `indices` and residue `resindices` are MDAnalysis derived and always index from 0, no matter the file type.

Supported versions

Table 6: TPR format versions and generations read by `MDAnalysis.topology.TPRParser.parse()`.

TPX format	TPX generation	Gromacs release	read
??	??	3.3, 3.3.1	no
58	17	4.0, 4.0.2, 4.0.3, 4.0.4, 4.0.5, 4.0.6, 4.0.7	yes
73	23	4.5.0, 4.5.1, 4.5.2, 4.5.3, 4.5.4, 4.5.5	yes
83	24	4.6, 4.6.1	yes
100	26	5.0, 5.0.1, 5.0.2, 5.0.3,5.0.4, 5.0.5	yes
103	26	5.1	yes
110	26	2016	yes
112	26	2018	yes
116	26	2019	yes

For further discussion and notes see Issue 2. Please *open a new issue* in the Issue Tracker when a new or different TPR file format version should be supported.

TPR specification

The TPR reader is a pure-python implementation of a basic TPR parser. Currently the following topology attributes are parsed:

- Atoms: number, name, type, resname, resid, segid, chainID, mass, charge, [residue, segment, radius, bfactor, resnum, moltype]
- Bonds
- Angles
- Dihedrals
- Impropers

`segid` and `chainID`

MDAnalysis gets the `segment` and `chainID` attributes from the TPR `molblock` field. Since TPR files are built from GROMACS topology files, `molblock` fields get their names from the compounds listed under the `[ molecules ]` header. For example:

```
[ molecules ]
; Compound        #mols
Protein_chain_A     1
Protein_chain_B     1
SOL             40210
```

So, the TPR will get 3 `molblock` s: `Protein_chain_A`, `Protein_chain_B` and `SOL`, while MDAnalysis will set the `segid` s to `seg_{segment_index}_{molblock}`, thus: `seg_0_Protein_chain_A`, `seg_1_Protein_chain_B` and `seg_2_SOL`. On the other hand, `chainID` will be identical to `molblock` unless `molblock` is named "Protein_chain_XXX", in which case `chainID` will be set to `XXX`. Thus in this case the `chainID` s will be: `A`, `B` and `SOL`.

Bonds

Bonded interactions available in Gromacs are described in the Gromacs manual. The following ones are used to build the topology (see Issue 463):

Table 7: GROMACS entries used to create bonds.

Directive	Type	Description
bonds	1	regular bond
bonds	2	G96 bond
bonds	3	Morse bond
bonds	4	cubic bond
bonds	5	connections
bonds	6	harmonic potentials
bonds	7	FENE bonds
bonds	8	tabulated potential with exclusion/connection
bonds	9	tabulated potential without exclusion/connection
bonds	10	restraint potentials
constraints	1	constraints with exclusion/connection
constraints	2	constraints without exclusion/connection
settles	1	SETTLE constraints

Table 8: GROMACS entries used to create angles.

Directive	Type	Description
angles	1	regular angle
angles	2	G96 angle
angles	3	Bond-bond cross term
angles	4	Bond-angle cross term
angles	5	Urey-Bradley
angles	6	Quartic angles
angles	8	Tabulated angles
angles	10	restricted bending potential

Table 9: GROMACS entries used to create dihedrals.

Directive	Type	Description
dihedrals	1	proper dihedral
dihedrals	3	Ryckaert-Bellemans dihedral
dihedrals	5	Fourier dihedral
dihedrals	8	Tabulated dihedral
dihedrals	9	Periodic proper dihedral
dihedrals	10	Restricted dihedral
dihedrals	11	Combined bending-torsion potential

Table 10: GROMACS entries used to create improper dihedrals.

Directive	Type	Description
dihedrals	2	improper dihedral
dihedrals	4	periodic improper dihedral

Developer notes

This tpr parser is written according to the following files

- *gromacs_dir*/src/kernel/gmxdump.c
- *gromacs_dir*/src/gmxlib/tpxio.c (the most important one)
- *gromacs_dir*/src/gmxlib/gmxfio_rw.c
- *gromacs_dir*/src/gmxlib/gmxfio_xdr.c
- *gromacs_dir*/include/gmxfiofio.h

or their equivalent in more recent versions of Gromacs.

The function read_tpxheader() is based on the TPRReaderDevelopment notes. Functions with names starting with read_ or do_ are trying to be similar to those in gmxdump.c or tpxio.c, those with extract_ are new.

Wherever fver_err(fver) is used, it means the tpx version problem has not been solved. Versions prior to Gromacs 4.0.x are not supported.

TRJ, MDCRD, CRDBOX (AMBER ASCII trajectory)

MDAnalysis supports reading of AMBER ASCII trajectories ("traj") and *binary trajectories* ("netcdf").

Important: In the AMBER community, these trajectories are often saved with the suffix '.crd'. This extension conflicts with the CHARMM CRD format and MDAnalysis *will not correctly autodetect AMBER ".crd" trajectories*. Instead, explicitly provide the format="TRJ" argument to Universe:

```
u = MDAnalysis.Universe("top.prmtop", "traj.crd", format="TRJ")
```

Reading in

Units are assumed to be the following default AMBER units:

- length: Angstrom
- time: ps

Limitations:

- Periodic boxes are only stored as box lengths A, B, C in an AMBER trajectory; the reader always assumes that these are orthorhombic boxes.
- The trajectory does not contain time information so we simply set the time step to 1 ps *(or the user could provide it with the dt argument)*
- Trajectories with fewer than 4 atoms probably fail to be read (BUG).
- If the trajectory contains exactly *one* atom then it is always assumed to be non-periodic (for technical reasons).
- Velocities are currently *not supported* as ASCII trajectories.

TRR (GROMACS lossless trajectory file)

The GROMACS TRR trajectory is a lossless format. This file format can store coordinates, velocities, and forces.

Important: MDAnalysis currently treats trajectories with damaged frames by truncating them at the frame before. Check that you are loading a valid file with gmx check.

Reading in

MDAnalysis uses XDR based readers for GROMACS formats, which store offsets on the disk. The offsets are used to enable access to random frames efficiently. These offsets will be generated automatically the first time the trajectory is opened, and offsets are generally stored in hidden `*_offsets.npz` files.[1]

Trajectories split across multiple files can be *read continuously into MDAnalysis* with `continuous=True`, in the style of gmx trjcat.

Writing out

If the `data` dictionary of a `Timestep` contains a `lambda` value, this will be used for the written TRR file. Otherwise, lambda is set to 0.

Developer notes

It sometimes can happen that the stored offsets get out off sync with the trajectory they refer to. For this the offsets also store the number of atoms, size of the file and last modification time. If any of them change, the offsets are recalculated. Writing of the offset file can fail when the directory where the trajectory file resides is not writable or if the disk is full. In this case a warning message will be shown but the offsets will nevertheless be used during the lifetime of the trajectory Reader. However, the next time the trajectory is opened, the offsets will have to be rebuilt again.

TRZ (IBIsCO and YASP trajectory)

MDAnalysis reads and writes IBIsCO / YASP TRZ binary trajectories in *little-endian* byte order.

TXYZ, ARC (Tinker)

MDAnalysis can read Tinker xyz files .txyz and trajectory .arc files.

[1] Occasionally, MDAnalysis fails to read XDR offsets, resulting in an error. The workaround for this is to create the Universe with regenerated offsets by using the keyword argument `refresh_offsets=True`, as documented in Issue 1893.

Developer notes

Differences between Tinker format and normal xyz files:

- there is only one header line containing both the number of atoms and a comment
- column 1 contains atom numbers (starting from 1)
- column 6 contains atoms types
- the following columns indicate connectivity (atoms to which that particular atom is bonded, according to numbering in column 1)

XML (HOOMD)

MDAnalysis can read topology informatin from a HOOMD XML file. Masses and charges are set to zero if not present in the XML file. Hoomd XML does not identify molecules or residues, so placeholder values are used for residue numbers. Bonds and angles are read if present.

Hoomd XML format does not contain a node for names. The parser will look for a name node anyway, and if it doesn't find one, it will use the atom types as names. If the Hoomd XML file doesn't contain a type node (it should), then all atom types will be 'none'.

Similar to the names, the parser will try to read atom type, mass, and charge from the XML file. Therefore, they are not present, masses and charges will not be guessed. Instead, they will be set to zero, as Hoomd uses unitless mass, charge, etc.

XPDB (Extended PDB file)

The extended PDB reader acts virtually the same as the *PDB, ENT (Standard PDB file)* reader. The difference is that extended PDB files (MDAnalysis format specifier *XPDB*) may contain residue sequence numbers up to 99,999 by utilizing the insertion character field of the PDB standard. Five-digit residue numbers may take up columns 23 to 27 (inclusive) instead of being confined to 23-26 (with column 27 being reserved for the insertion code in the PDB standard).

PDB files in this format are written by popular programs such as VMD.

As extended PDB files are very similar to PDB files, tell MDAnalysis to use the Extended PDB parser by passing in the `topology_format` keyword.

```
In [1]: import MDAnalysis as mda

In [2]: from MDAnalysis.tests.datafiles import PDB

In [3]: pdb = mda.Universe(PDB)

In [4]: pdb.trajectory.format
Out[4]: ['PDB', 'ENT']

In [5]: xpdb = mda.Universe(PDB, topology_format='XPDB')

In [6]: xpdb.trajectory.format
Out[6]: 'XPDB'
```

XTC (GROMACS compressed trajectory file)

The GROMACS XTC trajectory compresses data with reduced precision (3 decimal places by default). MDAnalysis can only read coordinates from these files. See *TRR (GROMACS lossless trajectory file)* for uncompressed files that provide velocity and force information.

Reading in

MDAnalysis uses XDR based readers for GROMACS formats, which store offsets on the disk. The offsets are used to enable access to random frames efficiently. These offsets will be generated automatically the first time the trajectory is opened, and offsets are generally stored in hidden `*_offsets.npz` files.[1]

Trajectories split across multiple files can be *read continuously into MDAnalysis* with `continuous=True`, in the style of gmx trjcat.

XYZ trajectory

The *XYZ format* is a loosely defined, simple coordinate trajectory format. The implemented format definition was taken from the VMD xyzplugin and is therefore compatible with VMD.

Reading in

As XYZ files only have atom name information, the atoms are all assigned to the same residue and segment.

The default timestep in MDAnalysis is 1 ps. A different timestep can be defined *by passing in the dt argument to Universe*.

XYZ specification

Definiton used by the `XYZReader` and `XYZWriter` (and the VMD xyzplugin from whence the definition was taken):

```
[ comment line            ] !! NOT IMPLEMENTED !! DO NOT INCLUDE
[ N                       ] # of atoms, required by this xyz reader plugin  line 1
[ molecule name           ] name of molecule (can be blank)                 line 2
atom1 x y z [optional data] atom name followed by xyz coords                line 3
atom2 x y z [ ...         ] and (optionally) other data.
...
atomN x y z [ ...         ]                                                 line N+2
```

[1] Occasionally, MDAnalysis fails to read XDR offsets, resulting in an error. The workaround for this is to create the Universe with regenerated offsets by using the keyword argument `refresh_offsets=True`, as documented in Issue 1893.

Note

- comment lines not implemented (do not include them)
- molecule name: the line is required but the content is ignored at the moment
- optional data (after the coordinates) are presently ignored

2.1.21 Analysis

The `analysis` module of MDAnalysis provides the tools needed to analyse your data. Several analyses are included with the package. These range from standard algorithms (e.g. *calculating root mean squared quantities*) to unique algorithms such as the *path similarity analysis*.

Generally these bundled analyses are contributed by various researchers who use the code for their own work. Please refer to the individual module documentation or relevant user guide tutorials for additional references and citation information.

If you need functionality that is not already provided in MDAnalysis, there are *several ways to write your own analysis*.

If you want to run your own script in parallel in MDAnalysis, here is a *tutorial on make your code parallelizable*.

Imports and dependencies

Analysis modules are not imported by default. In order to use them, you will need to import them separately, e.g.:

```
from MDAnalysis.analysis import align
```

Note: Several modules in `MDAnalysis.analysis` require additional Python packages. For example, `encore` makes use of scikit-learn. The Python packages are not automatically installed with *pip*, although they are with *conda*.

Other modules require external programs. For example, `hole` requires the HOLE programs. You will need to install these yourself.

2.1.22 Alignments and RMS fitting

The `MDAnalysis.analysis.align` and `MDAnalysis.analysis.rms` modules contain the functions used for aligning structures, aligning trajectories, and calculating root mean squared quantities.

Demonstrations of alignment are in align_structure, align_trajectory_first, and align_trajectory. Another example of generating an average structure from an alignment is demonstrated in rmsf. Typically, trajectories need to be aligned for RMSD and RMSF values to make sense.

Note: These modules use the fast QCP algorithm to calculate the root mean square distance (RMSD) between two coordinate sets [The05] and the rotation matrix R that minimizes the RMSD [LAT09]. Please cite these references when using these modules.

2.1.23 Distances and contacts

The `MDAnalysis.analysis.distances` module provides functions to rapidly compute distances. These largely take in coordinate arrays.

Residues can be determined to be in contact if atoms from the two residues are within a certain distance. Analysing the fraction of contacts retained by a protein over at trajectory, as compared to the number of contacts in a reference frame or structure, can give insight into folding processes and domain movements.

`MDAnalysis.analysis.contacts` contains functions and a class to analyse the fraction of native contacts over a trajectory.

2.1.24 Trajectory similarity

A molecular dynamics trajectory with N atoms can be considered through a path through $3N$-dimensional molecular configuration space. MDAnalysis contains a number of algorithms to compare the conformational ensembles of different trajectories. Most of these are in the `MDAnalysis.analysis.encore` module ([TPB+15]) and compare estimated probability distributions to measure similarity. The path similarity analysis compares the RMSD between pairs of structures in conformation transition paths. `MDAnalysis.analysis.encore` also contains functions for evaluating the conformational convergence of a trajectory using the similarity over conformation clusters or similarity in a reduced dimensional space.

2.1.25 Structure

2.1.26 Volumetric analyses

2.1.27 Dimension reduction

A molecular dynamics trajectory with N atoms can be considered a path through $3N$-dimensional molecular configuration space. It remains difficult to extract important dynamics or compare trajectory similarity from such a high-dimensional space. However, collective motions and physically relevant states can often be effectively described with low-dimensional representations of the conformational space explored over the trajectory. MDAnalysis implements two methods for dimensionality reduction.

Principal component analysis is a common linear dimensionality reduction technique that maps the coordinates in each frame of your trajectory to a linear combination of orthogonal vectors. The vectors are called *principal components*, and they are ordered such that the first principal component accounts for the most variance in the original data (i.e. the largest uncorrelated motion in your trajectory), and each successive component accounts for less and less variance. Trajectory coordinates can be transformed onto a lower-dimensional space (*essential subspace*) constructed from these principal components in order to compare conformations. Your trajectory can also be projected onto each principal component in order to visualise the motion described by that component.

Diffusion maps are a non-linear dimensionality reduction technique that embeds the coordinates of each frame onto a lower-dimensional space, such that the distance between each frame in the lower-dimensional space represents their "diffusion distance", or similarity. It integrates local information about the similarity of each point to its neighours, into a global geometry of the intrinsic manifold. This means that this technique is not suitable for trajectories where the transitions between conformational states is not well-sampled (e.g. replica exchange simulations), as the regions may become disconnected and a meaningful global geometry cannot be approximated. Unlike PCA, there is no explicit mapping between the components of the lower-dimensional space and the original atomic coordinates; no physical interpretation of the eigenvectors is immediately available.

For computing similarity, see the tutorials in *Trajectory similarity*.

2.1.28 Polymers and membranes

MDAnalysis has several analyses specifically for polymers, membranes, and membrane proteins.

2.1.29 Hydrogen Bond Analysis

The `MDAnalysis.analysis.hydrogen_bonds` module provides methods to find and analyse hydrogen bonds in a Universe.

Calculating hydrogen bonds: the basics

We will find the hydrogen bonds in a box of water in order to demonstrate the basic usage of HydrogenBondAnaysis.

Last updated: December 2022

Minimum version of MDAnalysis: 2.0.0-dev0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- numpy
- matplotlib

See also

- *Calculating hydrogen bonds: advanced selections*
- *Calculating hydrogen bond lifetimes*

Note

Please cite Smith et al. (2018) when using HydrogenBondAnaysis in published work.

[1]:
```
import pickle
import numpy as np
np.set_printoptions(linewidth=100)
import pandas as pd

import matplotlib.pyplot as plt

import MDAnalysis as mda
from MDAnalysis.tests.datafiles import waterPSF, waterDCD
from MDAnalysis.analysis.hydrogenbonds import HydrogenBondAnalysis

# the next line is necessary to display plots in Jupyter
%matplotlib inline
```

Loading files

We will load a small water-only system containing 5 water molecules and 10 frames then find the hydrogen bonds present at each frame.

```
[2]: u = mda.Universe(waterPSF, waterDCD)
```

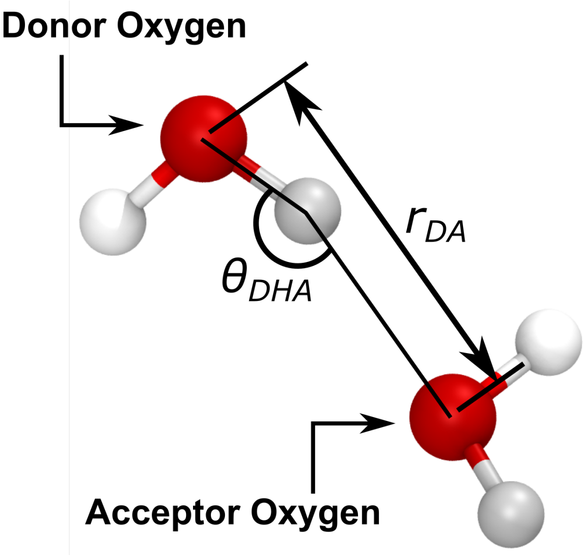
```
/Users/lily/pydev/mdanalysis/package/MDAnalysis/coordinates/DCD.py:165:␣
↪DeprecationWarning: DCDReader currently makes independent timesteps by copying self.ts␣
↪while other readers update self.ts inplace. This behavior will be changed in 3.0 to be␣
↪the same as other readers. Read more at https://github.com/MDAnalysis/mdanalysis/
↪issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

> **Warning**
>
> It is **highly recommended** that a topology with bond information is used (e.g. PSF, TPR, PRMTOP), as this is the only way to guarantee the correct identification of donor-hydrogen pairs.

Hydrogen bonds

In molecular dynamics simulations, hydrogen bonds are typically identified via the following geometric criteria:

1. the donor-acceptor distance (r_{DA}) must be less than a specified cutoff, typically 3
2. the donor-hydrogen-acceptor angle (θ_{DHA}) must be greater than a specified cutoff, typically 150°



Find water-water hydrogen bonds

Basic usage of HydrogenBondAnalysis involvees passing an acceptor atom selection (`acceptors_sel`) and a hydrogen atom selection (`hydrogens_sel`) to `HydrogenBondAnalysis`. By not providing a donor atom selection (`donor_sel`) we will use the topology bond information to find donor-hydrogen pairs.

In the following cell, `acceptors_sel="name OH2"` and `hydrogens_sel="name H1 H2"` will select the oxygen and hydrogen atoms, respectively, of water. The names correspond to those for the CHARMM TIP3P water model and may be different for other force fields.

[3]:

```
hbonds = HydrogenBondAnalysis(
    universe=u,
    donors_sel=None,
    hydrogens_sel="name H1 H2",
    acceptors_sel="name OH2",
    d_a_cutoff=3.0,
    d_h_a_angle_cutoff=150,
    update_selections=False
)
```

Note

For a performance boost, `update_selections` can be set to `False`.

However, always set `update_selections` to `True` if you think that your selection will update over time. For example, the number of oxygen atoms within 3.0 may change over time if you choose `acceptors_sel="name OH2 and around 3.0 protein`.

See also the MDAnalysis documentation on updating AtomGroups.

We then use the `run()` method to perform the analysis. If we do not set the `start`, `stop`, and `step` for frames to analyse, all frames will be used.

[4]:

```
hbonds.run(
    start=None,
    stop=None,
    step=None,
    verbose=True
)
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

[4]:

```
<MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at 0x14115fdf0>
```

Accessing the results

The results are stored as a numpy array in the `hbonds.results.hbonds` attribute. The array is of shape $(N_{\text{hbonds}}, 6)$.

[5]:
```
# We see there are 27 hydrogen bonds in total
print(hbonds.results.hbonds.shape)
```

```
(27, 6)
```

Each row of the results array contains the follwing information on a single hydrogen bond:

`[frame, donor_index, hydrogen_index, acceptor_index, DA_distance, DHA_angle]`

Let's take a look at the first hydrogen bond found:

[6]:
```
print(hbonds.results.hbonds[0])
```

```
[  0.           9.          10.           3.           2.82744082 150.48955173]
```

This hydrogen bond was found at frame 0. The donor, hydrogen, and acceptor atoms have indices 9, 10, and 3, respectively. There is a distance of 2.83 between the donor and acceptor atoms, and an angle of 150° made by the donor-hydrogen-acceptor atoms.

The frame number and atom indices can be used to select the atoms involved in the above hydrogen bond. However, as the results array is of type `float64`, these values must first be cast to integers:

[7]:
```
hbonds.results.hbonds.dtype
```

[7]:
```
dtype('float64')
```

[8]:
```
first_hbond = hbonds.results.hbonds[0]
```

[9]:
```
frame, donor_ix, hydrogen_ix, acceptor_ix = first_hbond[:4].astype(int)
```

[10]:
```
# select the correct frame and the atoms involved in the hydrogen bond
u.trajectory[frame]
atoms = u.atoms[[donor_ix, hydrogen_ix, acceptor_ix]]
```

[11]:
```
atoms
```

[11]:
```
<AtomGroup with 3 atoms>
```

For clarity, below we define constants that will be used throughout the rest of the notebook to access the relevant column of each hbond row

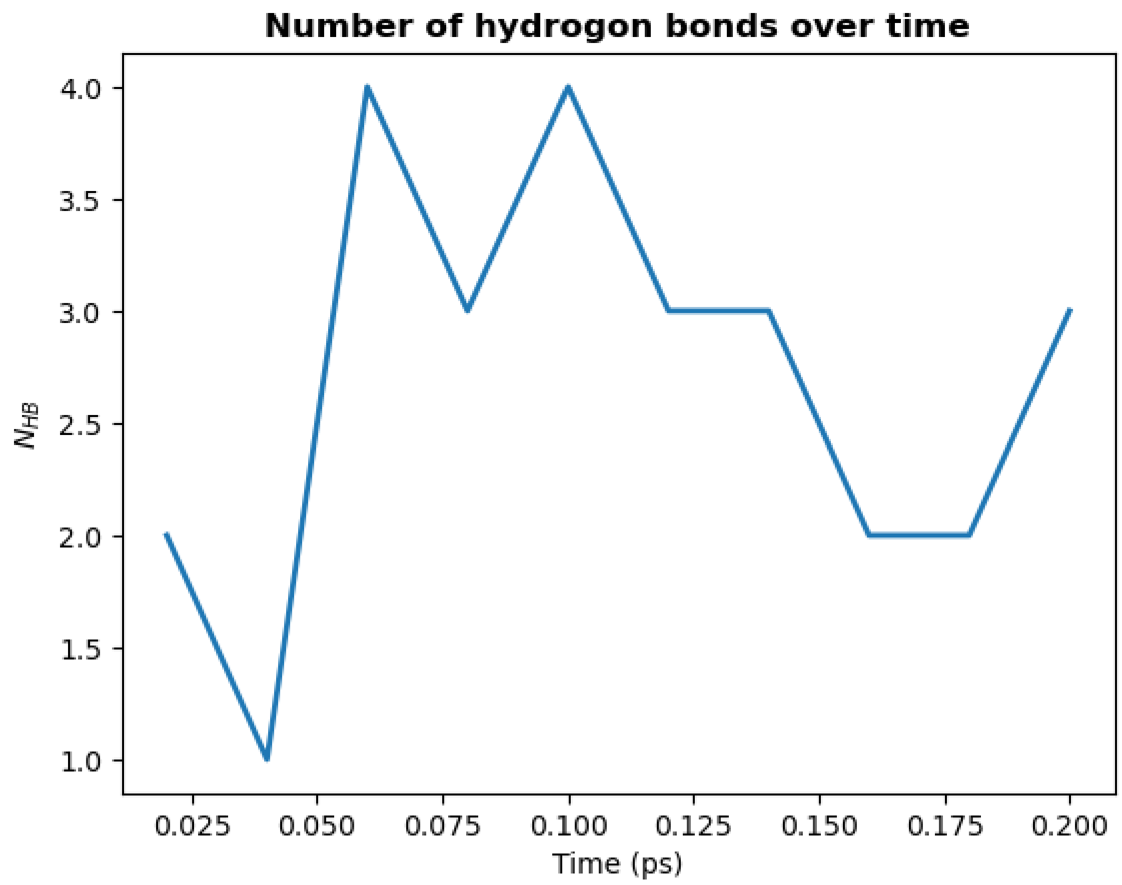
[12]:
```
FRAME = 0
DONOR = 1
HYDROGEN = 2
ACCEPTOR = 3
DISTANCE = 4
ANGLE = 5
```

Helper functions

The are three helper functions that can be used to quickly post-process the results.

1. Count by time. Counts the total number of hydrogen bonds at each frame

[13]:
```
plt.plot(hbonds.times, hbonds.count_by_time(), lw=2)

plt.title("Number of hydrogon bonds over time", weight="bold")
plt.xlabel("Time (ps)")
plt.ylabel(r"$N_{HB}$")

plt.show()
```



2. Count by type. Counts the total number of each type of hydrogen bond.

A type is a unique combination of donor residue name, donor atom type, acceptor residue name, and acceptor atom type.

[14]:
```
hbonds.count_by_type()
```

[14]:
```
array([['TIP3:OT', 'TIP3:OT', '27']], dtype='<U21')
```

Each row contains a unique hydrogen bond type. The three columns correspond to:

1. the donor resname and atom name (here, the OT atom of the TIP3 water residue)

2. the acceptor resname and atom name (here, the OT atom of the TIP3 water residue)
3. the total count

The average number of each type of hydrogen bond formed at each frame is likely more informative than the total number over the trajectory. This can be calculated for each hydrogen bond type as follows:

[15]:
```
for donor, acceptor, count in hbonds.count_by_type():

    donor_resname, donor_type = donor.split(":")
    n_donors = u.select_atoms(f"resname {donor_resname} and type {donor_type}").n_atoms

    # average number of hbonds per donor molecule per frame
    mean_count = 2 * int(count) / (hbonds.n_frames * n_donors)  # multiply by two as␣
↪each hydrogen bond involves two water molecules
    print(f"{donor} to {acceptor}: {mean_count:.2f}")
```

```
TIP3:OT to TIP3:OT: 1.08
```

Note

In a water-only system the average number of hydrogen bonds per water molecule should be around 3.3, depending on temperature and forcefield. We don't see this due to the small system size (15 water molecules).

3. Count by ids. Counts the total number of each hydrogen bond formed between specific atoms.

[16]:
```
hbonds.count_by_ids()
```

[16]:
```
array([[12, 14,  9, 10],
       [ 9, 10,  3,  9],
       [ 3,  5,  0,  4],
       [ 0,  1,  6,  3],
       [ 6,  7, 12,  1]])
```

Each row contains a unique hydrogen bond. The four columns correspond to:

1. the donor atom index
2. the hydrogen atom index
3. the acceptor atom index
4. the total count

The array is sorted in descending total count. You can check which atoms are involved in the most frequently observed hydrogen bond:

[17]:
```
counts = hbonds.count_by_ids()
most_common = counts[0]

print(f"Most common donor: {u.atoms[most_common[0]]}")
print(f"Most common hydrogen: {u.atoms[most_common[1]]}")
print(f"Most common acceptor: {u.atoms[most_common[2]]}")
```

```
Most common donor: <Atom 13: OH2 of type OT of resname TIP3, resid 21 and segid WAT>
Most common hydrogen: <Atom 15: H2 of type HT of resname TIP3, resid 21 and segid WAT>
Most common acceptor: <Atom 10: OH2 of type OT of resname TIP3, resid 15 and segid WAT>
```

Further analysis

There are many different analyses you may want to perform after finding hydrogen bonds. Below we will calculate the mean number of hydrogen bonds as a function of $z$ position of the donor atom.

[18]:

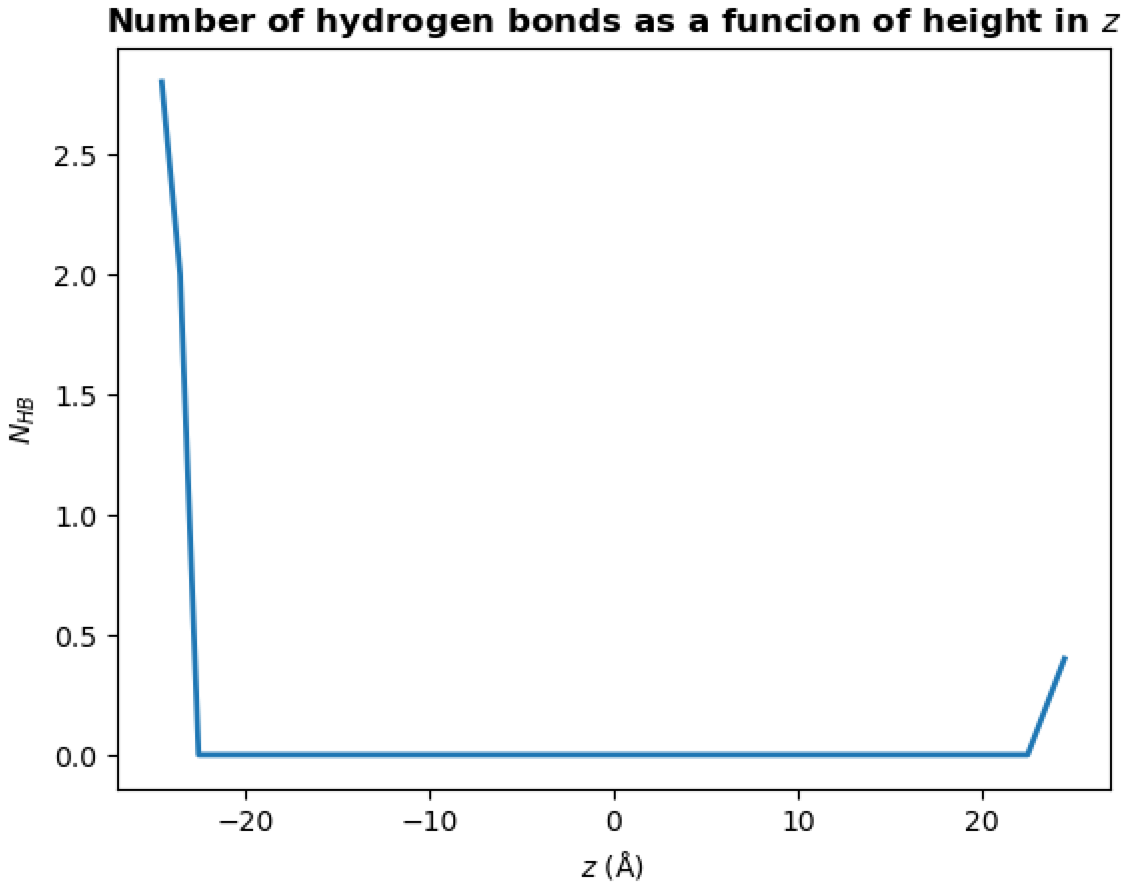
```
# bins in z for the histogram
bin_edges = np.linspace(-25, 25, 51)
bin_centers = bin_edges[:-1] + 0.5

# results array (this is faster and more memory efficient than appending to a list)
counts = np.full(bin_centers.size, fill_value=0.0)
```

[19]:

```
for frame, donor_ix, *_ in hbonds.results.hbonds:

    u.trajectory[frame.astype(int)]
    donor = u.atoms[donor_ix.astype(int)]

    zpos = donor.position[2]
    hist, *_ = np.histogram(zpos, bins=bin_edges)
    counts += hist * 2  # multiply by two as each hydrogen bond involves two water␣
↪molecules

counts /= hbonds.n_frames
```

[20]:

```
plt.plot(bin_centers, counts, lw=2)

plt.title(r"Number of hydrogen bonds as a funcion of height in $z$", weight="bold")
plt.xlabel(r"$z\ \rm (\AA)$")
plt.ylabel(r"$N_{HB}$")

plt.show()
```



You may wish to find the number of hydrogen bonds in z per unit area in xy, rather than the total number a each z position. To do this, simply divide `counts` by the mean area in xy:

[21]:
```
mean_xy_area = np.mean(
    [np.prod(ts.dimensions[:2]) for ts in u.trajectory[hbonds.frames]]
)
counts /= mean_xy_area
```

Note

It is likely you will want to calculate the distance from an interface, such as a lipid membrane, rather than the absolute position in z. For example, if you have a bilayer that is centered at 0, you might define your interface as the mean position in z of the phosphorous atoms in the upper leaflet of the bilayer:

headgroup_atoms = u.select_atoms("name P") interface_zpos = np.mean(headgroup_atoms.positions[:, 2])

zpos = donors.positions[:,2] - interface_zpos

Store data

There are various ways of storing the results from the analysis.

1. You can persist the HydrogenBondsAnalysis object using pickle (or joblib)

[22]:
```
with open("hbonds.pkl", 'wb') as f:
    pickle.dump(hbonds, f)
```

[23]:
```
with open("hbonds.pkl", 'rb') as f:
    hbonds = pickle.load(f)
```

2. You can extract the results and store the numpy array

[24]:
```
np.save("hbonds.npy", hbonds.results.hbonds)
```

3. You can create a Pandas DataFrame with more information about hydrogen bonding atoms, which may make some further analysis easier.

[25]:
```
df = pd.DataFrame(hbonds.results.hbonds[:, :DISTANCE].astype(int),
                  columns=["Frame",
                           "Donor_ix",
                           "Hydrogen_ix",
                           "Acceptor_ix",])

df["Distances"] = hbonds.results.hbonds[:, DISTANCE]
df["Angles"] = hbonds.results.hbonds[:, ANGLE]

df["Donor resname"] = u.atoms[df.Donor_ix].resnames
df["Acceptor resname"] = u.atoms[df.Acceptor_ix].resnames
df["Donor resid"] = u.atoms[df.Donor_ix].resids
df["Acceptor resid"] = u.atoms[df.Acceptor_ix].resids
df["Donor name"] = u.atoms[df.Donor_ix].names
df["Acceptor name"] = u.atoms[df.Acceptor_ix].names
```

[26]:
```
df.to_csv("hbonds.csv", index=False)
```

References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[3] Paul Smith, Robert M. Ziolek, Elena Gazzarrini, Dylan M. Owen, and Christian D. Lorenz. On the interaction of hyaluronic acid with synovial fluid lipid membranes. Phys. Chem. Chem. Phys., 21(19):9845-9857, 2018. URL: http://dx.doi.org/10.1039/C9CP01532A

Calculating hydrogen bonds: advanced selections

We will find all intramolcular hydrogen bonds of a protein without passing any atom selections. We will also look at how to use more advanced atom selections than we saw in *Calculating hydrogen bonds: the basiscs*.

Last updated: December 2022

Minimum version of MDAnalysis: 2.0.0-dev0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- numpy

See also

- *Calculating hydrogen bonds: the basics*
- *Calculating hydrogen bond lifetimes*

Note

Please cite Smith et al. (2018) when using HydrogenBondAnaysis in published work.

[1]:

```
import numpy as np

import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis.hydrogenbonds import HydrogenBondAnalysis
```

Loading files

[2]:

```
u = mda.Universe(PSF, DCD)
```

```
/home/pbarletta/mambaforge/envs/mda-user-guide/lib/python3.9/site-packages/MDAnalysis/
↪coordinates/DCD.py:165: DeprecationWarning: DCDReader currently makes independent␣
↪timesteps by copying self.ts while other readers update self.ts inplace. This␣
↪behaviour will be changed in 3.0 to be the same as other readers
  warnings.warn("DCDReader currently makes independent timesteps"
```

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

Find all hydrogen bonds

The simplest use case is to allow `HydrogenBondAnalysis` to guess the acceptor and hydrogen atoms, then to identify donor-hydrogen pairs via the bond information in the topology.

Accetptor and hydrogen atoms are guessed based on the mass and partial charge of the atoms.

```
[3]: hbonds = HydrogenBondAnalysis(universe=u)
```

```
[4]: hbonds.run(verbose=True)
```

```
  0%|          | 0/98 [00:00<?, ?it/s]
```

```
[4]: <MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at 0x7fce51309190>
```

Use guess_acceptors and guess_hydrogens to create atom selections

It is also possible to use generated `hydrogens_sel` and `acceptors_sel` via the `guess_hydrogens` and `guess_acceptors` class methods.

This selection strings may then be modified prior to calling `run`, or a subset of the universe may be used to guess the atoms. For example, below we will find hydrogens and acceptors belonging to specific residues.

```
[5]: hbonds = HydrogenBondAnalysis(universe=u)
hbonds.hydrogens_sel = hbonds.guess_hydrogens("resname ARG HIS LYS")
hbonds.acceptors_sel = hbonds.guess_acceptors("resname ARG HIS LYS")
```

```
[6]: hbonds.run(verbose=True)
```

```
  0%|          | 0/98 [00:00<?, ?it/s]
```

```
/home/pbarletta/mambaforge/envs/mda-user-guide/lib/python3.9/site-packages/MDAnalysis/
↪analysis/hydrogenbonds/hbond_analysis.py:751: UserWarning: No hydrogen bonds were␣
↪found given angle of 150 between Donor, None, and Acceptor, (resname ARG and name NE)␣
↪or (resname ARG and name NH1) or (resname ARG and name NH2) or (resname ARG and name␣
↪O) or (resname LYS and name O).
  warnings.warn(
```

```
[6]: <MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at 0x7fce50983790>
```

We can check which atoms were used in the analysis:

```
[7]: print(f"hydrogen_sel = {hbonds.hydrogens_sel}")
print(f"acceptors_sel = {hbonds.acceptors_sel}")
```

```
hydrogen_sel = (resname ARG and name HE) or (resname ARG and name HH11) or (resname ARG␣
↪and name HH12) or (resname ARG and name HH21) or (resname ARG and name HH22) or␣
↪(resname ARG and name HN) or (resname LYS and name HN) or (resname LYS and name HZ1)␣
↪or (resname LYS and name HZ2) or (resname LYS and name HZ3)
acceptors_sel = (resname ARG and name NE) or (resname ARG and name NH1) or (resname ARG␣
↪and name NH2) or (resname ARG and name O) or (resname LYS and name O)
```

More advanced selections

Slightly more complex selection strings are also possible. For example, to find hydrogen bonds involving the protein and water within 10 Å of the protein:

[8]:

```
hbonds = HydrogenBondAnalysis(universe=u)

protein_hydrogens_sel = hbonds.guess_hydrogens("protein")
protein_acceptors_sel = hbonds.guess_acceptors("protein")

water_hydrogens_sel = "resname TIP3 and name H1 H2"
water_acceptors_sel = "resname TIP3 and name OH2"

hbonds.hydrogens_sel = f"({protein_hydrogens_sel}) or ({water_hydrogens_sel} and around␣
↪10 not resname TIP3)"
hbonds.acceptors_sel = f"({protein_acceptors_sel}) or ({water_acceptors_sel} and around␣
↪10 not resname TIP3)"
```

[9]:

```
hbonds.run(verbose=True)
```

```
  0%|          | 0/98 [00:00<?, ?it/s]
```

[9]:

```
<MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at 0x7fce5098a250>
```

Note

The Universe we are analysing has the water removed. The above example is for illustrative purposes only.

Hydrogen bonds between specific groups

To calculate the hydrogen bonds between different groups, for example protein and water, one can use the `between` keyword. Below we will find protein-water and protein-protein hydrogen bonds, but not water-water hydrogen bonds. We do this by passing atom selection for the groups we wish to find hydrogen bonds between.

[10]:

```
hbonds = HydrogenBondAnalysis(
    universe=u,
    between=[
        ["protein", "resname TIP3"], # for protein-water hbonds
        ["protein", "protein"]       # for protein-protein hbonds
    ]
)
```

[11]:

```
hbonds.run(verbose=True)
```

```
  0%|          | 0/98 [00:00<?, ?it/s]
```

[11]:

```
<MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at 0x7fce5096fd60>
```

Note

The Universe we are analysing has the water removed. The above example is for illustrative purposes only.

References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[3] Paul Smith, Robert M. Ziolek, Elena Gazzarrini, Dylan M. Owen, and Christian D. Lorenz. On the interaction of hyaluronic acid with synovial fluid lipid membranes. Phys. Chem. Chem. Phys., 21(19):9845-9857, 2018. URL: http://dx.doi.org/10.1039/C9CP01532A

[4] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

Calculating hydrogen bond lifetimes

We will calculate the lifetime of intramolecular hydrogen bonds in a protein.

Last updated: December 2022

Minimum version of MDAnalysis: 2.0.0-dev0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests
- numpy
- matplotlib

See also

- *Calculating hydrogen bonds: the basics*
- *Calculating hydrogen bonds: advanced selections*

Note

Please cite Smith et al. (2018) when using HydrogenBondAnaysis in published work.

[1]:

```
from tqdm.auto import tqdm
import numpy as np
import matplotlib.pyplot as plt

import MDAnalysis as mda
```

(continues on next page)

(continued from previous page)

```
from MDAnalysis.tests.datafiles import PSF, DCD
from MDAnalysis.analysis.hydrogenbonds import HydrogenBondAnalysis

import warnings
# suppress some MDAnalysis warnings
warnings.filterwarnings('ignore')
```

Loading files

```
[2]: u = mda.Universe(PSF, DCD)
```

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

Find all hydrogen bonds

First, find the hydrogen bonds.

```
[3]: hbonds = HydrogenBondAnalysis(universe=u)
```

```
[4]: hbonds.run(verbose=True)
```

```
  0%|          | 0/98 [00:00<?, ?it/s]
```

```
[4]: <MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at 0x7f2cca7db1c0>
```

Calculate hydrogen bond lifetimes

The hydrogen bond lifetime is calculated via the time autocorrelation function of the presence of a hydrogen bond:

$$C(\tau) = \left\langle \frac{h_{ij}(t_0)h_{ij}(t_0+\tau)}{h_{ij}(t_0)^2} \right\rangle$$

where h_{ij} indicates the presence of a hydrogen bond between atoms i and j:

- $h_{ij} = 1$ if there is a hydrogen bond
- $h_{ij} = 0$ if there is no hydrogen bond

$h_{ij}(t_0) = 1$ indicates there is a hydrogen bond between atoms i and j at a time origin t_0, and $h_{ij}(t_0 + \tau) = 1$ indicates these atoms remain hydrogen bonded throughout the period t_0 to $t_0 + \tau$. To improve statistics, multiple time origins, t_0, are used in the calculation and the average is taken over all time origins.

See Gowers and Carbonne (2015) for further discussion on hydrogen bonds lifetimes.

Note

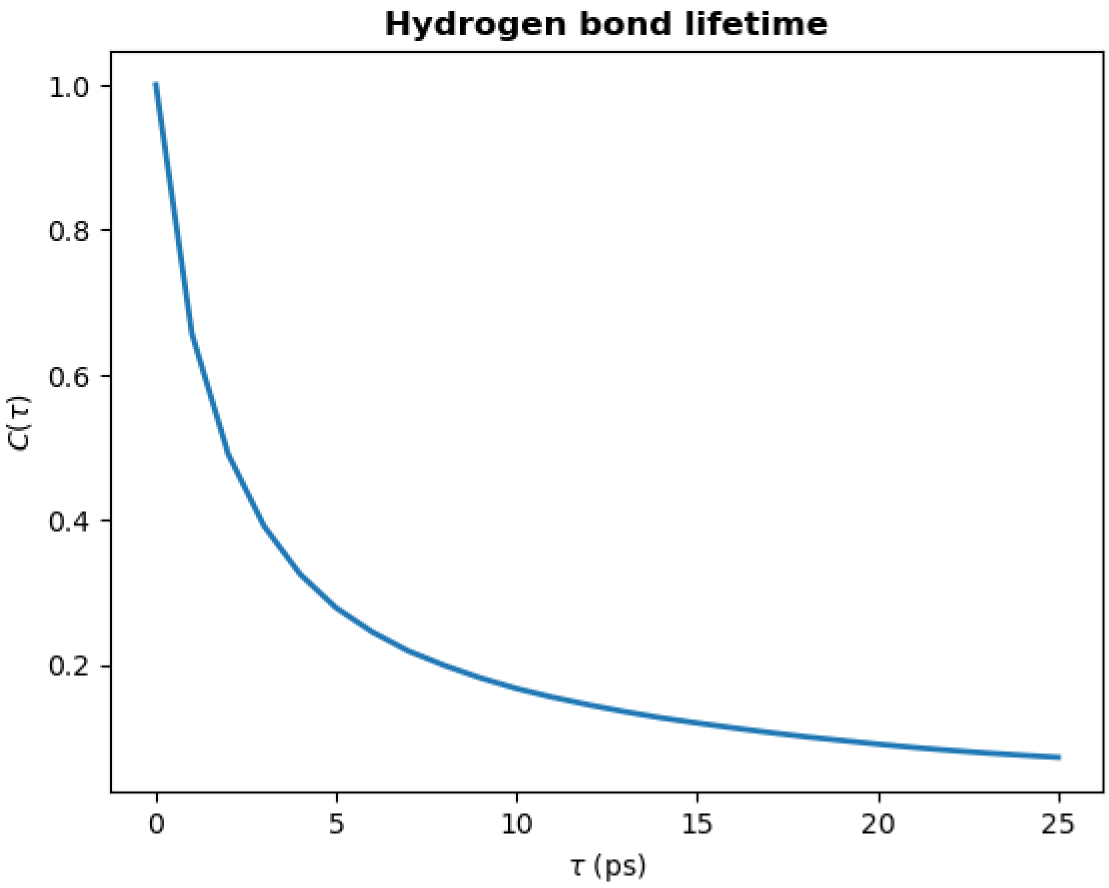
The period between time origins, t_0, should be chosen such that consecutive t_0 are uncorrelated.

The `hbonds.lifetime` method calculates the above time autocorrelation function. The period between time origins is set using `window_step`, and the maximum value of τ (in frames) is set using `tau_max`.

```
[5]: tau_max = 25
     window_step = 1
```

```
[6]: tau_frames, hbond_lifetime = hbonds.lifetime(
         tau_max=tau_max,
         window_step=window_step
     )
```

```
[7]: tau_times = tau_frames * u.trajectory.dt
     plt.plot(tau_times, hbond_lifetime, lw=2)

     plt.title(r"Hydrogen bond lifetime", weight="bold")
     plt.xlabel(r"$\tau\ \rm (ps)$")
     plt.ylabel(r"$C(\tau)$")

     plt.show()
```



Calculating the time constant

To obtain the hydrogen bond lifetime, you can fit a biexponential to the time autocorrelation curve. We will fit the following biexponential:

$$A\exp(-t/\tau_1) + B\exp(-t/\tau_2)$$

where τ_1 and τ_2 represent two time constants - one corresponding to a short-timescale process and the other to a longer timescale process. A and B will sum to 1, and they represent the relative importance of the short- and longer-timescale processes in the overall autocorrelation curve.

[8]:

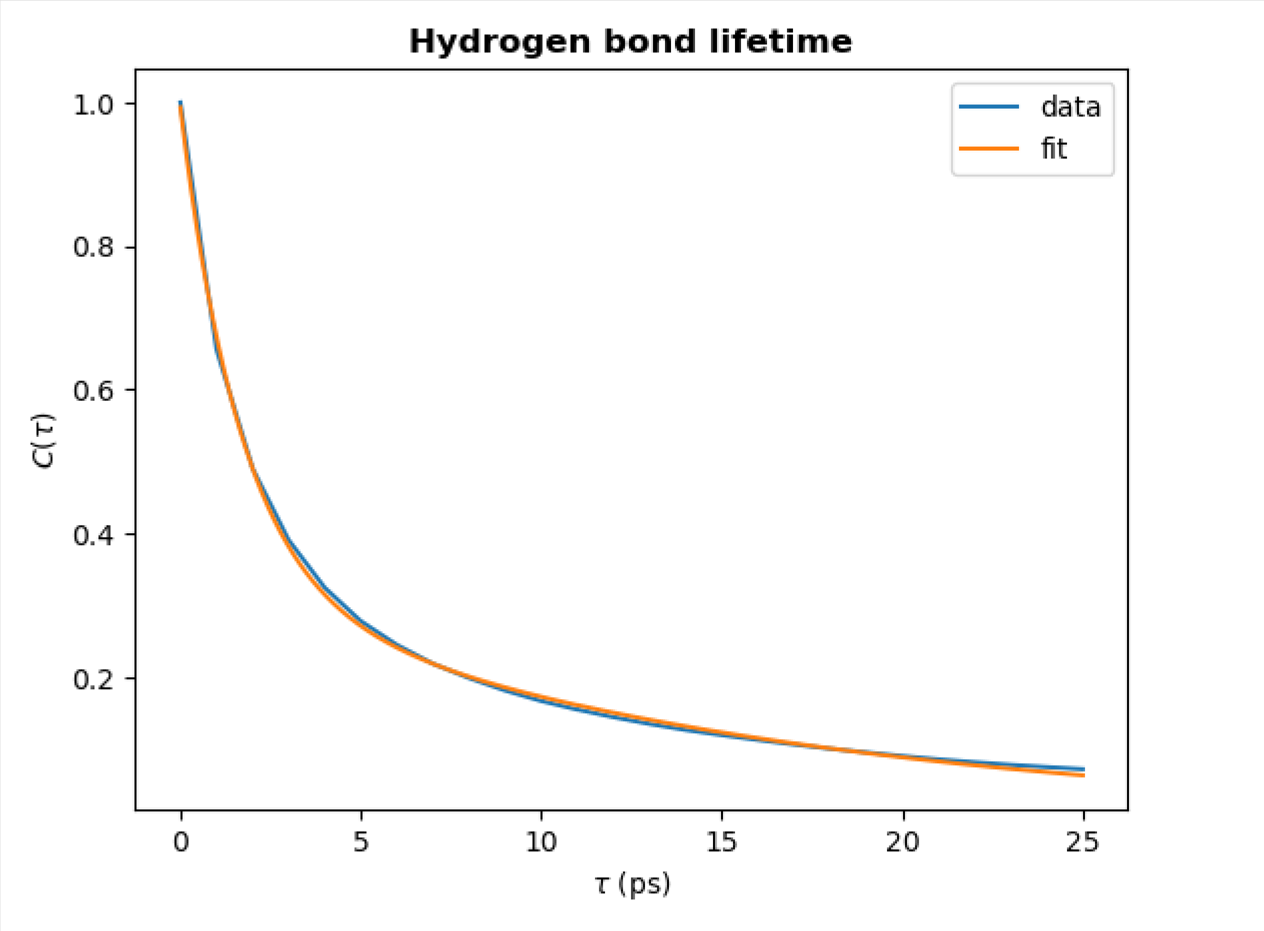
```
def fit_biexponential(tau_timeseries, ac_timeseries):
    """Fit a biexponential function to a hydrogen bond time autocorrelation function

    Return the two time constants
    """
    from scipy.optimize import curve_fit

    def model(t, A, tau1, B, tau2):
        """Fit data to a biexponential function.
        """
        return A * np.exp(-t / tau1) + B * np.exp(-t / tau2)

    params, params_covariance = curve_fit(model, tau_timeseries, ac_timeseries, [1, 0.5,␣
↪1, 2])

    fit_t = np.linspace(tau_timeseries[0], tau_timeseries[-1], 1000)
    fit_ac = model(fit_t, *params)

    return params, fit_t, fit_ac
```

[9]:

```
params, fit_t, fit_ac = fit_biexponential(tau_times, hbond_lifetime)
```

[10]:

```
# Plot the fit
plt.plot(tau_times, hbond_lifetime, label="data")
plt.plot(fit_t, fit_ac, label="fit")

plt.title(r"Hydrogen bond lifetime", weight="bold")
plt.xlabel(r"$\tau\ \rm (ps)$")
plt.ylabel(r"$C(\tau)$")

plt.legend()
plt.show()
```



[11]:

```
# Check the decay time constant
A, tau1, B, tau2 = params
time_constant = A * tau1 + B * tau2
print(f"time_constant = {time_constant:.2f} ps")
```

```
time_constant = 6.14 ps
```

Intermittent lifetime

The above example shows you how to calculate the continuous hydrogen bond lifetime. This means that the hydrogen bond must be present at every frame from t_0 to $t_0+\tau$. To allow for small fluctuations in the DA distance or DHA angle, the intermittent hydrogen bond lifetime may be calculated. This allows a hydrogen bond to break for up to a specified number of frames and still be considered present.

In the `lifetime` method, the `intermittency` argument is used to set the maxium number of frames for which a hydrogen bond is allowed to break. The default is `intermittency=0`, which means that if a hydrogen bond is missing at any frame between t_0 and $t_0 + \tau$, it will not be considered present at $t_0 + \tau$. This is equivalent to the continuous lifetime. However, with a value of `intermittency=2`, all hydrogen bonds are allowed to break for up to a maximum of consecutive two frames.

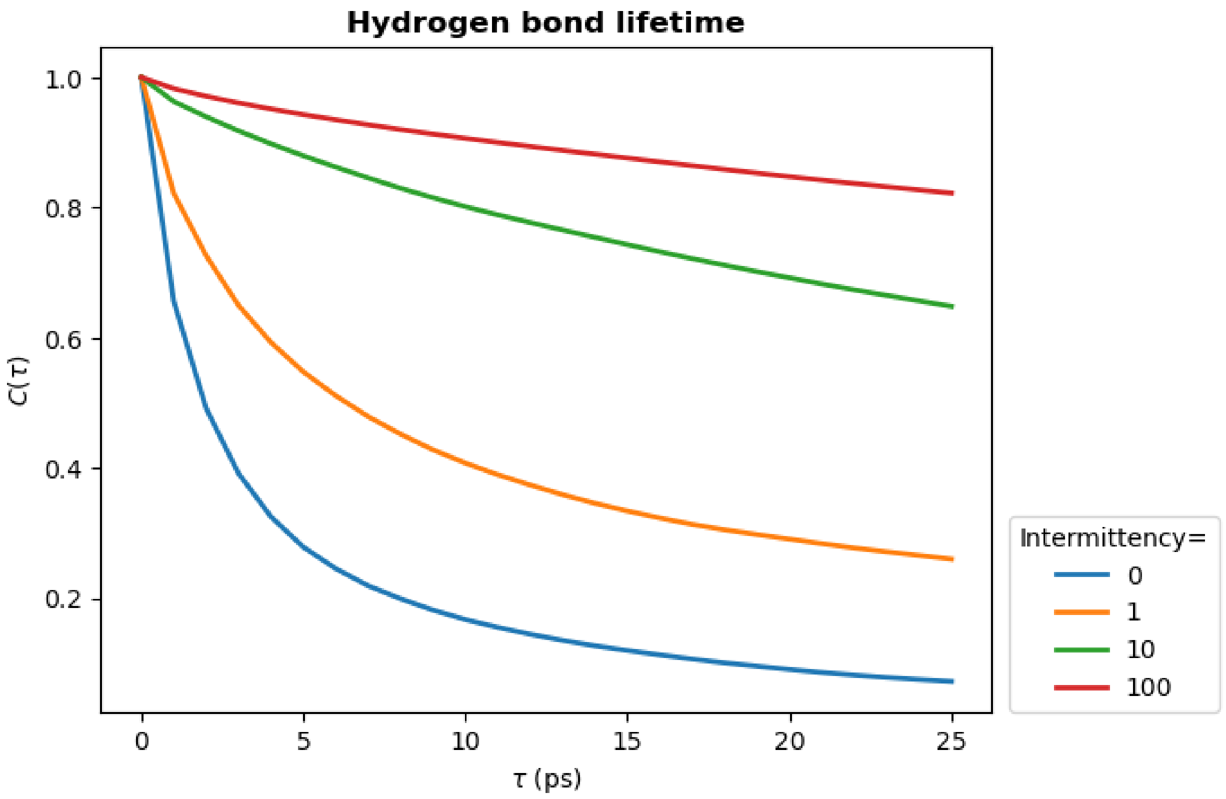
Below we see how changing the intermittency affects the hydrogen bond lifetime.

[12]:

```
tau_max = 25
window_step = 1
intermittencies = [0, 1, 10, 100]
```

[13]:

```
for intermittency in intermittencies:

    tau_frames, hbond_lifetime = hbonds.lifetime(
        tau_max=tau_max,
        window_step=window_step,
        intermittency=intermittency
    )

    times_times = tau_frames * u.trajectory.dt
    plt.plot(times_times, hbond_lifetime, lw=2, label=intermittency)

plt.title(r"Hydrogen bond lifetime", weight="bold")
plt.xlabel(r"$\tau\ \rm (ps)$")
plt.ylabel(r"$C(\tau)$")

plt.legend(title="Intermittency=", loc=(1.02, 0.0))
plt.show()
```



Hydrogen bond lifetime of individual hydrogen bonds

Let's first find the 3 most prevalent hydrogen bonds.

```
[14]: hbonds = HydrogenBondAnalysis(universe=u)
```

```
[15]: hbonds.run(verbose=True)
```

```
  0%|          | 0/98 [00:00<?, ?it/s]
```

```
[15]: <MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at 0x7f2cca7d3a60>
```

```
[16]: # Print donor, hydrogen, acceptor and count info for these hbonds
counts = hbonds.count_by_ids()
lines = []
for donor, hydrogen, acceptor, count in counts[:10]:
    d, h, a = u.atoms[donor], u.atoms[hydrogen], u.atoms[acceptor]
    lines.append(f"{d.resname}-{d.resid}-{d.name}\t{h.name}\t{a.resname}-{a.resid}-{a.
↪name}\tcount={count}")
for line in sorted(lines):
    print(line)
```

```
ARG-124-NH2     HH22    GLU-143-OE1     count=93
ARG-2-NH1       HH11    ASP-104-OD1     count=96
ARG-206-NE      HE      GLU-210-OE1     count=93
ARG-71-NH2      HH22    ASP-76-OD1      count=98
LYS-200-NZ      HZ2     ASP-208-OD2     count=93
LYS-211-NZ      HZ3     GLU-204-OE1     count=92
THR-199-OG1     HG1     ASP-197-OD1     count=95
TYR-133-OH      HH      ASP-146-OD1     count=95
TYR-193-OH      HH      GLU-108-OE1     count=97
TYR-24-OH       HH      GLY-214-OT2     count=95
```

Now we'll calculate the lifetime of these hydrogen bonds. To do this, the simplest way is to run `HydrogenBondAnalysis` for each hydrogen bond then use the `lifetime` method. It is very efficient to find hydrogen bonds between two specific atoms, especially with `update_selections=False`.

```
[17]: tau_max = 25
window_step = 1
intermittency = 0
```

```
[18]: hbond_lifetimes = []
labels = [] # for plotting

for hbond in counts[:3]:

        # find hbonds between specific atoms
        d_ix, h_ix, a_ix = hbond[:3]
        tmp_hbonds = HydrogenBondAnalysis(
            universe=u,
            hydrogens_sel=f"index {h_ix}",
            acceptors_sel=f"index {a_ix}",
            update_selections=False
        )
```

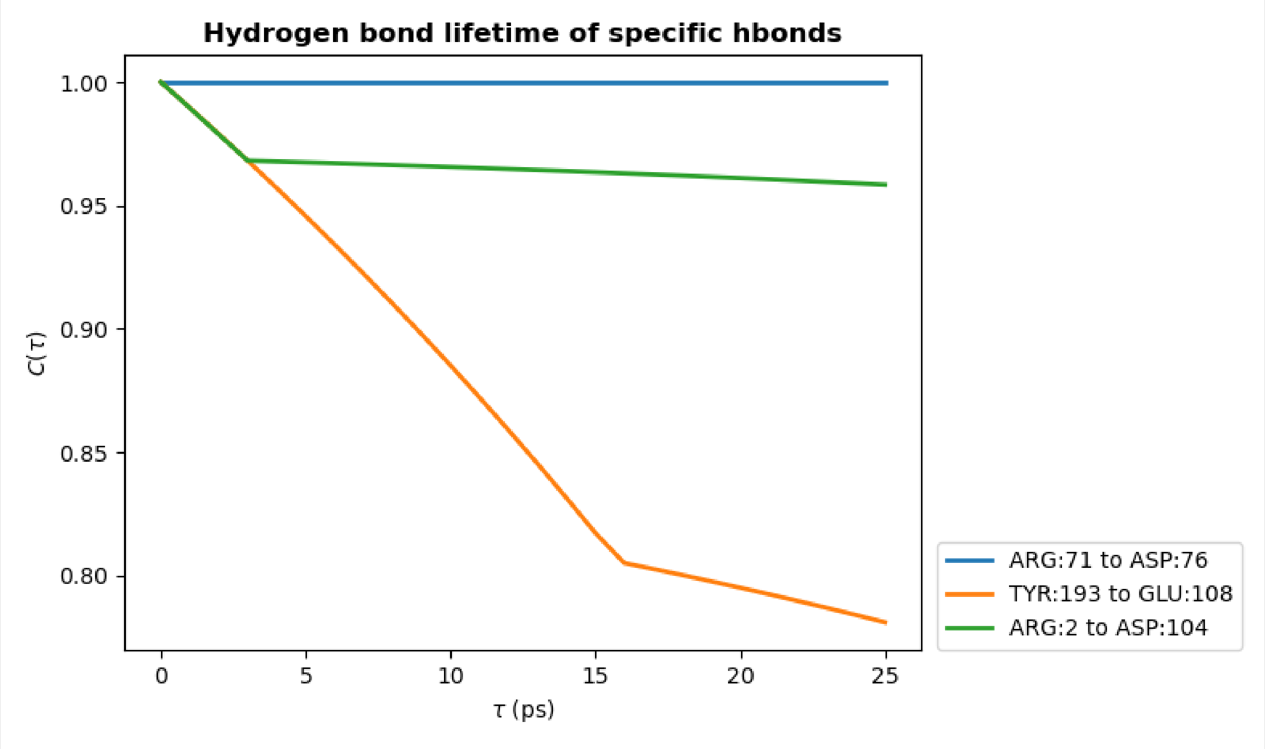
(continues on next page)

(continued from previous page)

```
        tmp_hbonds.run()

        # calculate lifetime
        taus, hbl, = tmp_hbonds.lifetime(
            tau_max=tau_max,
            intermittency=intermittency
        )
        hbond_lifetimes.append(hbl)

        # label for plotting
        donor, acceptor = u.atoms[d_ix], u.atoms[a_ix]
        label = f"{donor.resname}:{donor.resid} to {acceptor.resname}:{acceptor.resid}"
        labels.append(label)

hbond_lifetimes = np.array(hbond_lifetimes)
labels = np.array(labels)
```

[19]:

```
# Plot the lifetimes
times = taus * u.trajectory.dt
for hbl, label in zip(hbond_lifetimes, labels):
    plt.plot(times, hbl, label=label, lw=2)

plt.title(r"Hydrogen bond lifetime of specific hbonds", weight="bold")
plt.xlabel(r"$\tau\ \rm (ps)$")
plt.ylabel(r"$C(\tau)$")

plt.legend(ncol=1, loc=(1.02, 0))
plt.show()
```



Note

The shape of these curves indicates we have poor statistics in our lifetime calculations - we used only 100 frames and consider a single hydrogen bond.

The curve should decay smoothly toward 0, as seen in the first hydrogen bond lifetime plot we produced in this notebook. If the curve does not decay smoothly, more statistics are required either by increasing the value of `tau_max`, using a greater number of time origins, or increasing the length of your simulation.

See Gowers and Carbonne (2015) for further discussion on hydrogen bonds lifetimes.

References

[1] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[2] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[3] Paul Smith, Robert M. Ziolek, Elena Gazzarrini, Dylan M. Owen, and Christian D. Lorenz. On the interaction of hyaluronic acid with synovial fluid lipid membranes. Phys. Chem. Chem. Phys., 21(19):9845-9857, 2018. URL: http://dx.doi.org/10.1039/C9CP01532A

[4] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology,

394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[5] Richard J. Gowers and Paola Carbonne. A multiscale approach to model hydrogen bonding: The case of polyamide. J. Chem. Phys., 142:224907, June 2015. URL: https://doi.org/10.1063/1.4922445

2.1.30 Writing your own trajectory analysis

We create our own analysis methods for calculating the radius of gyration of a selection of atoms.

This can be done three ways, from least to most flexible:

1. *Running the analysis directly from a function*
2. *Turning a function into a class*
3. *Writing your own class*

The building blocks and methods shown here are only suitable for analyses that involve iterating over the trajectory once.

If you implement your own analysis method, please consider contributing it to the MDAnalysis codebase!

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 2.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisTests

[1]:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD, DCD2
from MDAnalysis.analysis.base import (AnalysisBase,
                                      AnalysisFromFunction,
                                      analysis_class)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Radius of gyration

Let's start off by defining a standalone analysis function.

The radius of gyration of a structure measures how compact it is. In GROMACS, it is calculated as follows:

$$R_g = \sqrt{\frac{\sum_i m_i \mathbf{r}_i^2}{\sum_i m_i}}$$

where m_i is the mass of atom i and $\mathbf{r}_i$ is the position of atom i, relative to the center-of-mass of the selection.

The radius of gyration around each axis can also be determined separately. For example, the radius of gyration around the x-axis:

$$R_{i,x} = \sqrt{\frac{\sum_i m_i [r_{i,y}^2 + r_{i,z}^2]}{\sum_i m_i}}$$

Below, we define a function that takes an AtomGroup and calculates the radii of gyration. We could write this function to only need the AtomGroup. However, we also add in a `masses` argument and a `total_mass` keyword to avoid recomputing the mass and total mass for each frame.

[2]:
```
def radgyr(atomgroup, masses, total_mass=None):
    # coordinates change for each frame
    coordinates = atomgroup.positions
    center_of_mass = atomgroup.center_of_mass()

    # get squared distance from center
    ri_sq = (coordinates-center_of_mass)**2
    # sum the unweighted positions
    sq = np.sum(ri_sq, axis=1)
    sq_x = np.sum(ri_sq[:,[1,2]], axis=1) # sum over y and z
    sq_y = np.sum(ri_sq[:,[0,2]], axis=1) # sum over x and z
    sq_z = np.sum(ri_sq[:,[0,1]], axis=1) # sum over x and y

    # make into array
    sq_rs = np.array([sq, sq_x, sq_y, sq_z])

    # weight positions
    rog_sq = np.sum(masses*sq_rs, axis=1)/total_mass
    # square root and return
    return np.sqrt(rog_sq)
```

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosophotransferase enzyme. ([BDPW09])

[3]:
```
u = mda.Universe(PSF, DCD)
protein = u.select_atoms('protein')

u2 = mda.Universe(PSF, DCD2)
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Creating an analysis from a function

`MDAnalysis.analysis.base.AnalysisFromFunction` can create an analysis from a function that works on AtomGroups. It requires the function itself, the trajectory to operate on, and then the arguments / keyword arguments necessary for the function.

[4]:
```
rog = AnalysisFromFunction(radgyr, u.trajectory,
                           protein, protein.masses,
                           total_mass=np.sum(protein.masses))
rog.run()
```

```
[4]: <MDAnalysis.analysis.base.AnalysisFromFunction at 0x7f901a31bca0>
```

Running the analysis iterates over the trajectory. The output is saved in `rog.results.timeseries`, which has the same number of rows, as frames in the trajectory. You can access the results both at `rog.results.timeseries` and `rog.results['timeseries']`:

```
[5]: rog.results['timeseries'].shape
```

```
[5]: (98, 4)
```

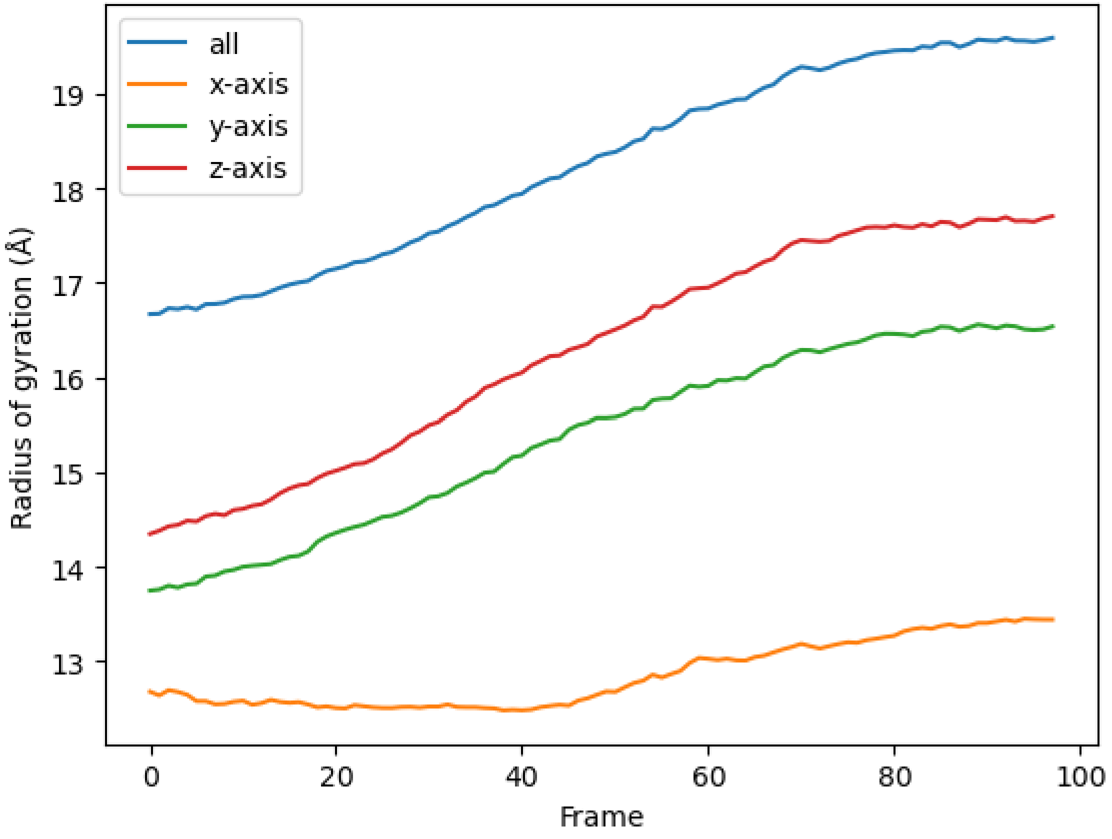
gives the same outputs as:

```
[6]: rog.results.timeseries.shape
```

```
[6]: (98, 4)
```

```
[7]: labels = ['all', 'x-axis', 'y-axis', 'z-axis']
     for col, label in zip(rog.results['timeseries'].T, labels):
         plt.plot(col, label=label)
     plt.legend()
     plt.ylabel('Radius of gyration (Å)')
     plt.xlabel('Frame')
```

```
[7]: Text(0.5, 0, 'Frame')
```



You can also re-run the analysis with different frame selections.

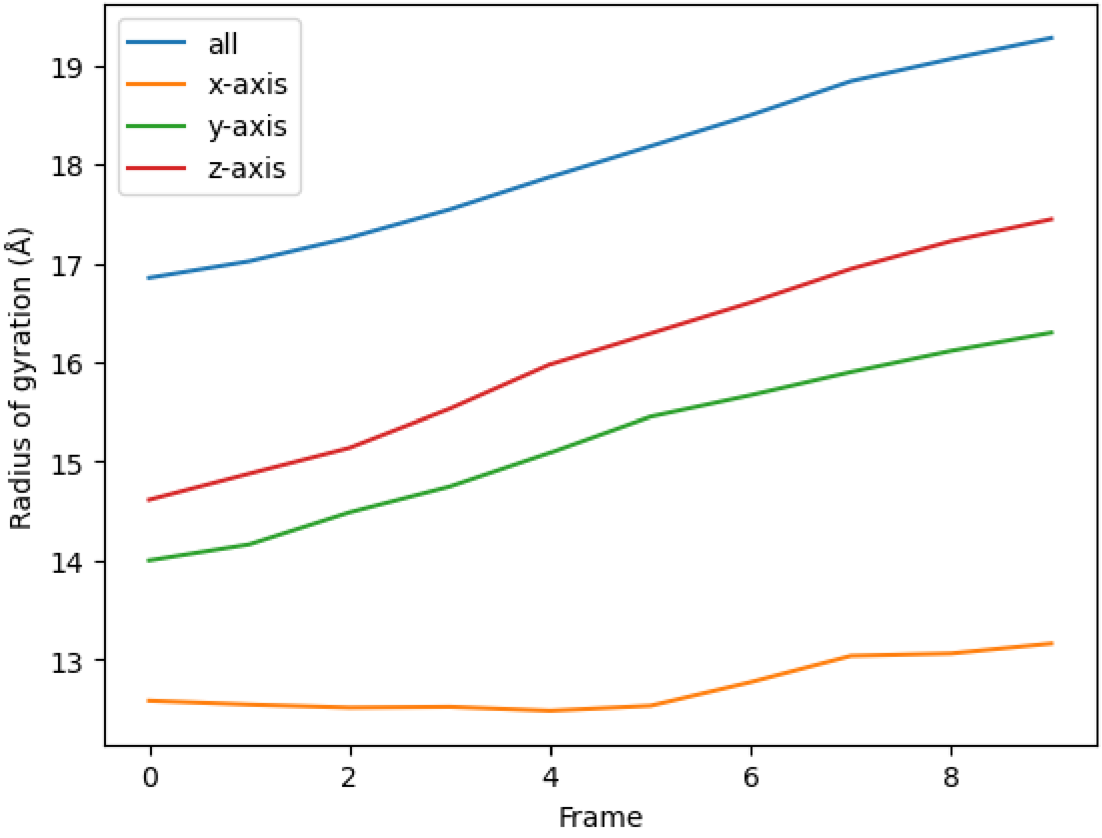
Below, we start from the 10th frame and take every 8th frame until the 80th. Note that the slice includes the `start` frame, but does not include the stop frame index (much like the actual range() function).

[8]:
```
rog_10 = AnalysisFromFunction(radgyr, u.trajectory,
                              protein, protein.masses,
                              total_mass=np.sum(protein.masses))

rog_10.run(start=10, stop=80, step=7)
rog_10.results['timeseries'].shape
```

[8]:
```
(10, 4)
```

[9]:
```
for col, label in zip(rog_10.results['timeseries'].T, labels):
    plt.plot(col, label=label)
plt.legend()
plt.ylabel('Radius of gyration (Å)')
plt.xlabel('Frame')
```

[9]:
```
Text(0.5, 0, 'Frame')
```



Transforming a function into a class

While the `AnalysisFromFunction` is convenient for quick analyses, you may want to turn your function into a class that can be applied to many different trajectories, much like other MDAnalysis analyses.

You can apply `analysis_class` to any function that you can run with `AnalysisFromFunction` to get a class.

```
[10]: RadiusOfGyration = analysis_class(radgyr)
```

To run the analysis, pass exactly the same arguments as you would for `AnalysisFromFunction`.

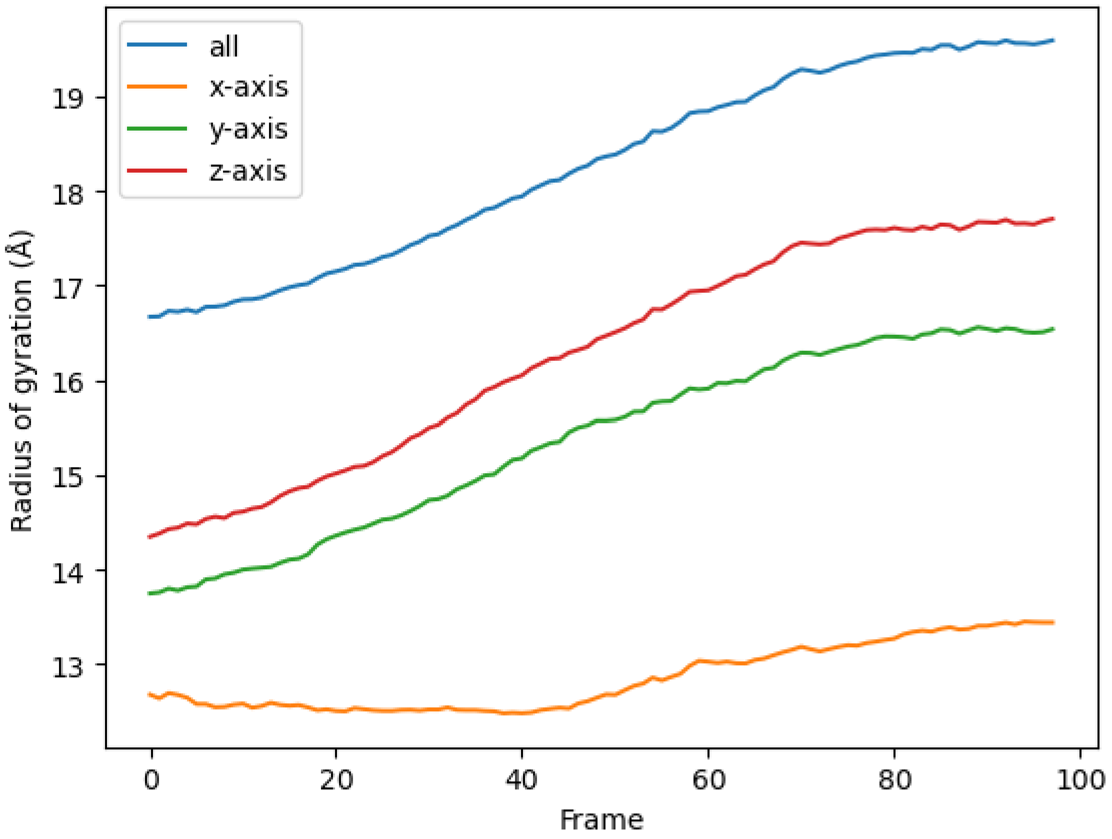
```
[11]: rog_u1 = RadiusOfGyration(u.trajectory, protein,
                                 protein.masses,
                                 total_mass=np.sum(protein.masses))
      rog_u1.run()
```

```
[11]: <MDAnalysis.analysis.base.analysis_class.<locals>.WrapperClass at 0x7f9000500820>
```

As with `AnalysisFromFunction`, the results are in `results`.

```
[12]: for col, label in zip(rog_u1.results['timeseries'].T, labels):
          plt.plot(col, label=label)
      plt.legend()
      plt.ylabel('Radius of gyration (Å)')
      plt.xlabel('Frame')
```

```
[12]: Text(0.5, 0, 'Frame')
```



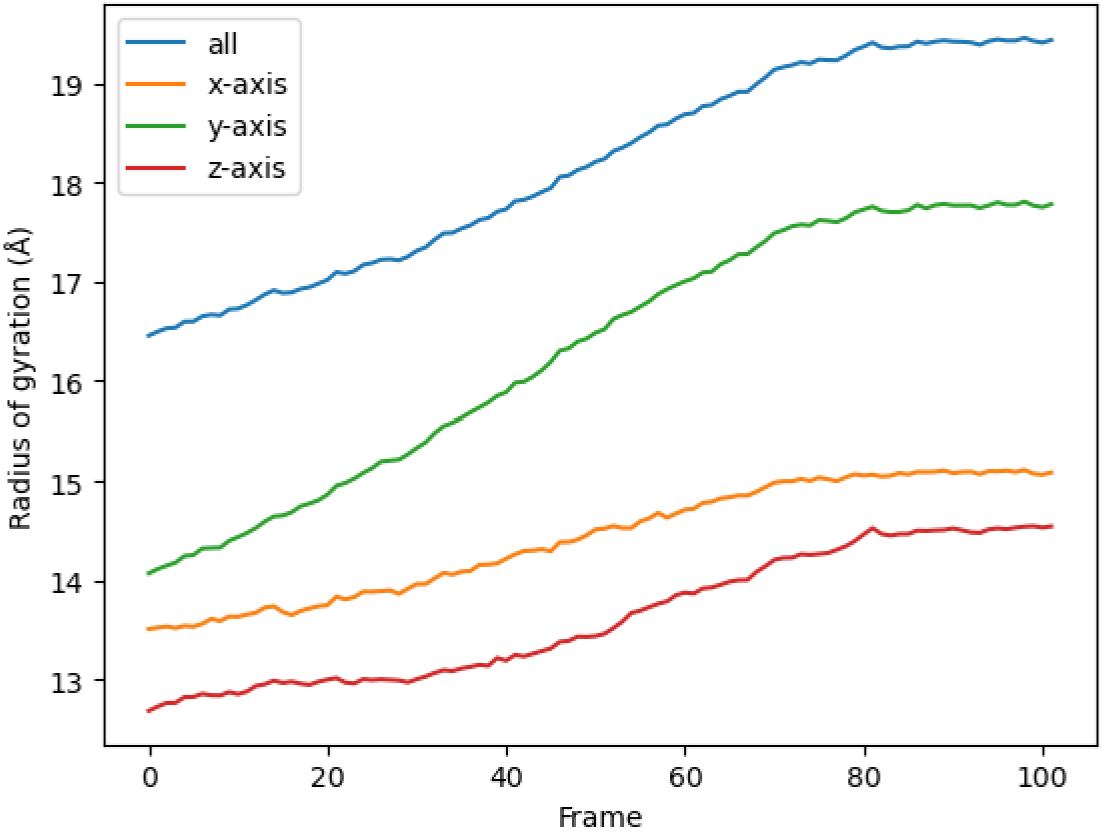
You can reuse the class for other trajectories and selections.

```
[13]: ca = u2.select_atoms('name CA')

      rog_u2 = RadiusOfGyration(u2.trajectory, ca,
                                ca.masses,
                                total_mass=np.sum(ca.masses))
      rog_u2.run()
```

```
[13]: <MDAnalysis.analysis.base.analysis_class.<locals>.WrapperClass at 0x7f9000474df0>
```

```
[14]: for col, label in zip(rog_u2.results['timeseries'].T, labels):
          plt.plot(col, label=label)
      plt.legend()
      plt.ylabel('Radius of gyration (Å)')
      plt.xlabel('Frame')
```

```
[14]: Text(0.5, 0, 'Frame')
```



Creating your own class

Although AnalysisFromFunction and analysis_class are convenient, they can be too limited for complex algorithms. You may need to write your own class.

MDAnalysis provides the MDAnalysis.analysis.base.AnalysisBase class as a template for creating multiframe analyses. This class automatically sets up your trajectory reader for iterating, and includes an optional progress meter.

The analysis is always run by calling run(). AnalysisFromFunction actually subclasses AnalysisBase, and analysis_class returns a subclass of AnalysisFromFunction, so the behaviour of run() remains identical.

1. Define __init__

You can define a new analysis by subclassing AnalysisBase. Initialise the analysis with the __init__ method, where you *must* pass the trajectory that you are working with to AnalysisBase.__init__(). You can also pass in the verbose keyword. If verbose=True, the class will set up a progress meter for you.

2. Define your analysis in _single_frame() and other methods

Implement your functionality as a function over each frame of the trajectory by defining _single_frame(). This function gets called for each frame of your trajectory.

You can also define _prepare() and _conclude() to set your analysis up before looping over the trajectory, and to finalise the results that you have prepared. In order, run() calls:

- _prepare()
- _single_frame() (for each frame of the trajectory that you are iterating over)
- _conclude()

Class subclassed from AnalysisBase can make use of several properties when defining the methods above:

- self.start: frame index to start analysing from. Defined in run()
- self.stop: frame index to stop analysis. Defined in run()
- self.step: number of frames to skip in between. Defined in run()
- self.n_frames: number of frames to analyse over. This can be helpful in initialising result arrays.
- self._verbose: whether to be verbose.
- self._trajectory: the actual trajectory
- self._ts: the current timestep object
- self._frame_index: the index of the currently analysed frame. This is *not* the absolute index of the frame in the trajectory overall, but rather the relative index of the frame within the list of frames to be analysed. You can think of it as the number of times that self._single_frame() has already been called.

Below, we create the class RadiusOfGyration2 to run the analysis function that we have defined above, and add extra information such as the time of the corresponding frame.

[15]:

```
class RadiusOfGyration2(AnalysisBase):  # subclass AnalysisBase

    def __init__(self, atomgroup, verbose=True):
        """
        Set up the initial analysis parameters.
```

(continues on next page)

(continued from previous page)

```
        """
        # must first run AnalysisBase.__init__ and pass the trajectory
        trajectory = atomgroup.universe.trajectory
        super(RadiusOfGyration2, self).__init__(trajectory,
                                               verbose=verbose)
        # set atomgroup as a property for access in other methods
        self.atomgroup = atomgroup
        # we can calculate masses now because they do not depend
        # on the trajectory frame.
        self.masses = self.atomgroup.masses
        self.total_mass = np.sum(self.masses)

    def _prepare(self):
        """
        Create array of zeroes as a placeholder for results.
        This is run before we begin looping over the trajectory.
        """
        # This must go here, instead of __init__, because
        # it depends on the number of frames specified in run().
        self.results = np.zeros((self.n_frames, 6))
        # We put in 6 columns: 1 for the frame index,
        # 1 for the time, 4 for the radii of gyration

    def _single_frame(self):
        """
        This function is called for every frame that we choose
        in run().
        """
        # call our earlier function
        rogs = radgyr(self.atomgroup, self.masses,
                      total_mass=self.total_mass)
        # save it into self.results
        self.results[self._frame_index, 2:] = rogs
        # the current timestep of the trajectory is self._ts
        self.results[self._frame_index, 0] = self._ts.frame
        # the actual trajectory is at self._trajectory
        self.results[self._frame_index, 1] = self._trajectory.time

    def _conclude(self):
        """
        Finish up by calculating an average and transforming our
        results into a DataFrame.
        """
        # by now self.result is fully populated
        self.average = np.mean(self.results[:, 2:], axis=0)
        columns = ['Frame', 'Time (ps)', 'Radius of Gyration',
                   'Radius of Gyration (x-axis)',
                   'Radius of Gyration (y-axis)',
                   'Radius of Gyration (z-axis)',]
        self.df = pd.DataFrame(self.results, columns=columns)
```

Because RadiusOfGyration2 calculates the masses of the selected AtomGroup itself, we do not need to pass it in ourselves.

```
[16]: rog_base = RadiusOfGyration2(protein, verbose=True).run()

        0%|          | 0/98 [00:00<?, ?it/s]
```

As calculated in `_conclude()`, the average radii of gyrations are at `rog.average`.

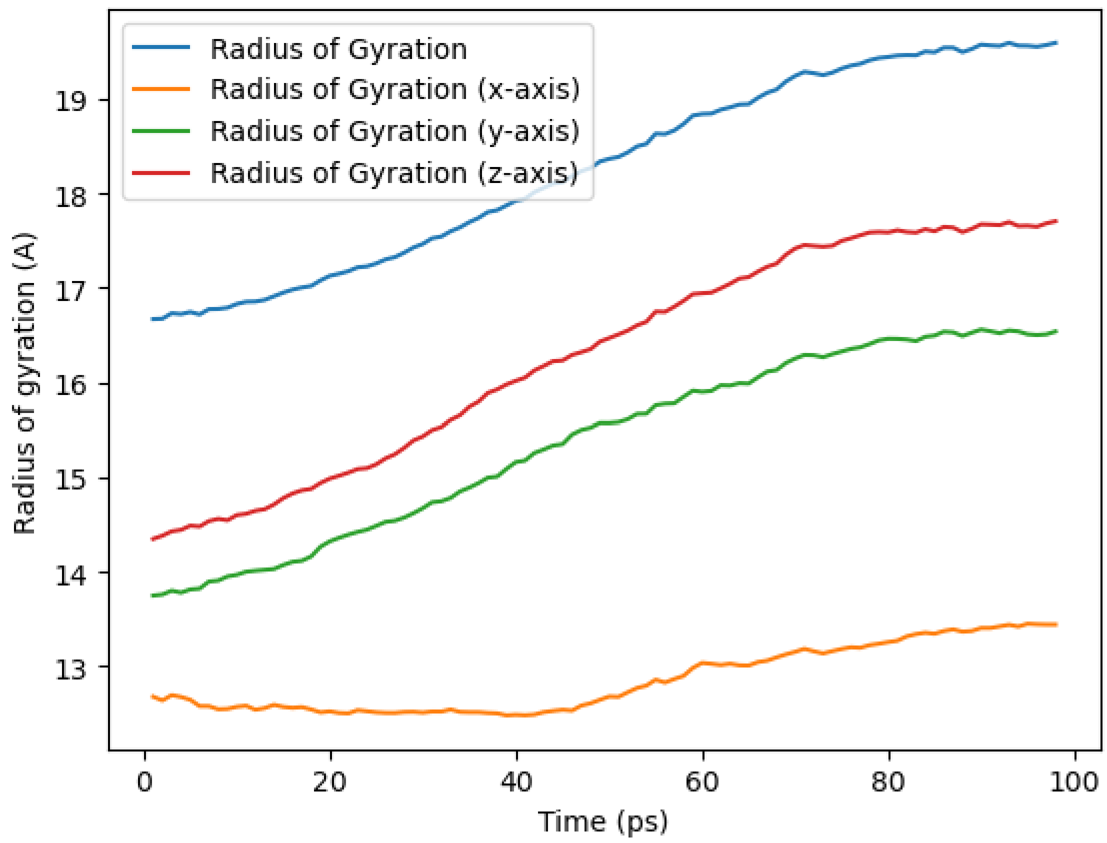
```
[17]: rog_base.average

[17]: array([18.26549552, 12.85342131, 15.37359575, 16.29185734])
```

The results are available at `rog.results` as an array or `rog.df` as a DataFrame.

```
[18]: rog_base.df

[18]:     Frame  Time (ps)  Radius of Gyration  Radius of Gyration (x-axis)  \
      0     0.0   1.000000           16.669018                    12.679625
      1     1.0   2.000000           16.673217                    12.640025
      2     2.0   3.000000           16.731454                    12.696454
      3     3.0   4.000000           16.722283                    12.677194
      4     4.0   5.000000           16.743961                    12.646981
      ..    ...        ...                 ...                          ...
      93   93.0  93.999992           19.562034                    13.421683
      94   94.0  94.999992           19.560575                    13.451335
      95   95.0  95.999992           19.550571                    13.445914
      96   96.0  96.999991           19.568381                    13.443243
      97   97.0  97.999991           19.591575                    13.442750

          Radius of Gyration (y-axis)  Radius of Gyration (z-axis)
      0                     13.749343                    14.349043
      1                     13.760545                    14.382960
      2                     13.801342                    14.429350
      3                     13.780732                    14.444711
      4                     13.814553                    14.489046
      ..                          ...                          ...
      93                    16.539112                    17.653968
      94                    16.508649                    17.656678
      95                    16.500640                    17.646130
      96                    16.507396                    17.681294
      97                    16.537926                    17.704494

      [98 rows x 6 columns]
```

Using this DataFrame we can easily plot our results.

```
[19]: ax = rog_base.df.plot(x='Time (ps)', y=rog_base.df.columns[2:])
      ax.set_ylabel('Radius of gyration (A)')

[19]: Text(0, 0.5, 'Radius of gyration (A)')
```



We can also run the analysis over a subset of frames, the same as the output from `AnalysisFromFunction` and `analysis_class`.

```
[20]: rog_base_10 = RadiusOfGyration2(protein, verbose=True)
      rog_base_10.run(start=10, stop=80, step=7)
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

```
[20]: <__main__.RadiusOfGyration2 at 0x7f90006ca9d0>
```

```
[21]: rog_base_10.results.shape
```

```
[21]: (10, 6)
```

```
[22]: rog_base_10.df
```

```
[22]:    Frame  Time (ps)  Radius of Gyration  Radius of Gyration (x-axis)  \
      0   10.0  10.999999           16.852127                    12.584163
      1   17.0  17.999998           17.019587                    12.544784
      2   24.0  24.999998           17.257429                    12.514341
      3   31.0  31.999997           17.542565                    12.522147
      4   38.0  38.999997           17.871241                    12.482385
      5   45.0  45.999996           18.182243                    12.533023
      6   52.0  52.999995           18.496493                    12.771949
      7   59.0  59.999995           18.839346                    13.037335
      8   66.0  66.999994           19.064333                    13.061491
```

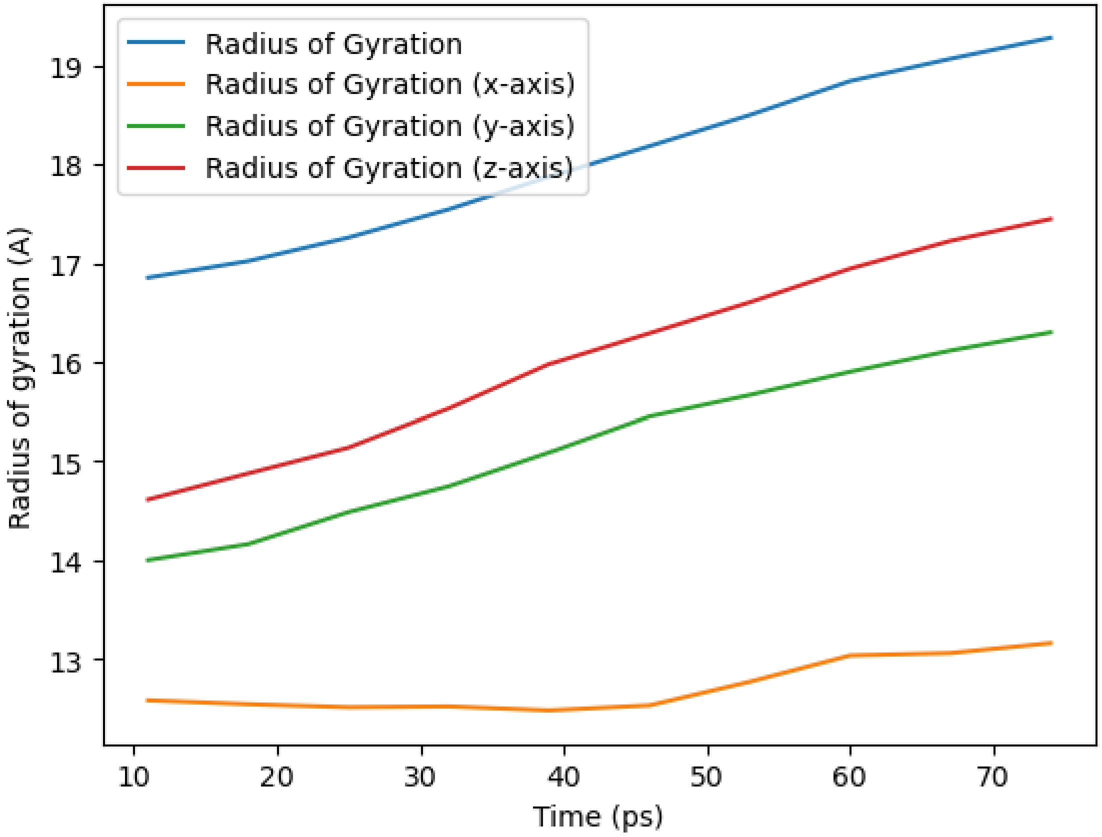
(continues on next page)

(continued from previous page)

```
9   73.0  73.999993          19.276639                    13.161863

   Radius of Gyration (y-axis)  Radius of Gyration (z-axis)
0                    14.001589                    14.614469
1                    14.163276                    14.878262
2                    14.487021                    15.137873
3                    14.747461                    15.530339
4                    15.088865                    15.977444
5                    15.451285                    16.290153
6                    15.667003                    16.603098
7                    15.900327                    16.942533
8                    16.114195                    17.222884
9                    16.298539                    17.444213
```

```
[23]: ax_10 = rog_base_10.df.plot(x='Time (ps)',
                                  y=rog_base_10.df.columns[2:])
      ax_10.set_ylabel('Radius of gyration (A)')
```

```
[23]: Text(0, 0.5, 'Radius of gyration (A)')
```



Contributing to MDAnalysis

If you think that you will want to reuse your new analysis, or that others might find it helpful, please consider contributing it to the MDAnalysis codebase. Making your code open-source can have many benefits; others may notice an unexpected bug or suggest ways to optimise your code. If you write your analysis for a specific publication, please let us know; we will ask those who use your code to cite your reference in published work.

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

2.1.31 Parallelizing analysis

As we approach the exascale barrier, researchers are handling increasingly large volumes of molecular dynamics (MD) data. Whilst MDAnalysis is a flexible and relatively fast framework for complex analysis tasks in MD simulations, implementing a parallel computing framework would play a pivotal role in accelerating the time to solution for such large datasets.

This document illustrates how you can run your own analysis scripts in parallel with MDAnalysis.

Last updated: December 2022 with MDAnalysis 2.4.0-dev0

Minimum version of MDAnalysis: 2.0.0

Packages required:

- MDAnalysis ([MADWB11], [GLB+16])
- MDAnalysisData
- dask (https://dask.org/)
- dask.distributed (https://distributed.dask.org/en/latest/)
- joblib (https://joblib.readthedocs.io/en/latest/)

[1]:

```
import MDAnalysis as mda
from MDAnalysisData.adk_equilibrium import fetch_adk_equilibrium

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

```

(continues on next page)

(continued from previous page)

```
n_jobs = 2

# You can also set `n_jobs` to the number of threads available:
# from multiprocessing import cpu_count
# n_jobs = cpu_count()
```

Background

In MDAnalysis, most implemented analysis methods are based on `AnalysisBase`, which provides a generic API for users to write their own trajectory analysis. However, this framework only takes single-core power of the PC by iterating through the trajectory and running a frame-wise analysis. Below we aim to first explore some possible simple implementations of parallelism, including using multiprocessing and dask. We will also discuss the acceleration approaches that should be considered, ranging from your own multiple-core laptops/desktops to distributed clusters. “

Loading files

The test files we will be working with here feature adenylate kinase (AdK), a phosopho-transferase enzyme. ([BDPW09]). The trajectory has 4187 frames, which will take quite some time to run the analysis on with the conventional serial (single-core) approach.

Note: downloading these datasets from MDAnalysisData may take some time.

[2]:
```
adk = fetch_adk_equilibrium()
```

[3]:
```
u = mda.Universe(adk.topology, adk.trajectory)
protein = u.select_atoms('protein')
print(f"Number of frames: {u.trajectory.n_frames}")
print(f"Number of atoms: {u.atoms.n_atoms}")
```

```
Number of frames: 4187
Number of atoms: 3341
```

```
/home/pbarletta/mambaforge/envs/guide/lib/python3.9/site-packages/MDAnalysis/coordinates/
↪DCD.py:165: DeprecationWarning: DCDReader currently makes independent timesteps by␣
↪copying self.ts while other readers update self.ts inplace. This behavior will be␣
↪changed in 3.0 to be the same as other readers. Read more at https://github.com/
↪MDAnalysis/mdanalysis/issues/3889 to learn if this change in behavior might affect you.
  warnings.warn("DCDReader currently makes independent timesteps"
```

Radius of gyration

For a detail description of this analysis, read Writing your own trajectory.

Here is a common form of single-frame method that we can normally see inside `AnalysisBase`. It may contain both some dynamic parts that changes along time either implicitly or explicitly (e.g. `AtomGroup`) and some static parts (e.g. a reference frame).

[4]:
```
def radgyr(atomgroup, masses, total_mass=None):
    # coordinates change for each frame
    coordinates = atomgroup.positions
```

(continues on next page)

(continued from previous page)

```
    center_of_mass = atomgroup.center_of_mass()

    # get squared distance from center
    ri_sq = (coordinates-center_of_mass)**2
    # sum the unweighted positions
    sq = np.sum(ri_sq, axis=1)
    sq_x = np.sum(ri_sq[:,[1,2]], axis=1) # sum over y and z
    sq_y = np.sum(ri_sq[:,[0,2]], axis=1) # sum over x and z
    sq_z = np.sum(ri_sq[:,[0,1]], axis=1) # sum over x and y

    # make into array
    sq_rs = np.array([sq, sq_x, sq_y, sq_z])

    # weight positions
    rog_sq = np.sum(masses*sq_rs, axis=1)/total_mass
    # square root and return
    return np.sqrt(rog_sq)
```

Serial Analysis

Below is the serial version of the analysis that we normally use.

[5]:

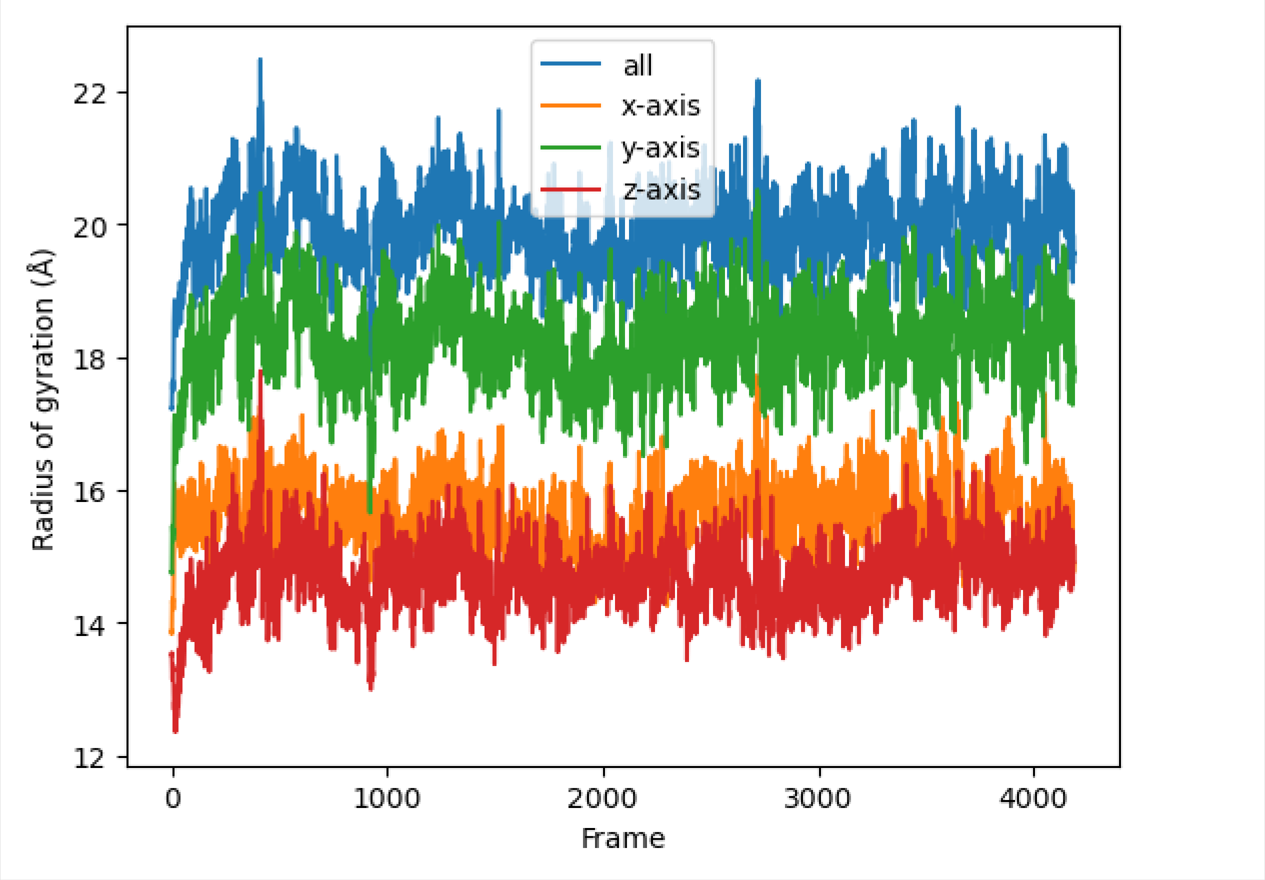
```
result = []
for frame in u.trajectory:
    result.append(radgyr(atomgroup=protein,
                         masses=protein.masses,
                         total_mass=np.sum(protein.masses)))
```

[6]:

```
result = np.asarray(result).T

labels = ['all', 'x-axis', 'y-axis', 'z-axis']
for col, label in zip(result, labels):
    plt.plot(col, label=label)
plt.legend()
plt.ylabel('Radius of gyration (Å)')
plt.xlabel('Frame')
```

[6]:

```
Text(0.5, 0, 'Frame')
```



Parallelization in a simple per-frame fashion

Frame-wise form of the function

The coordinates of the atomgroup analysed change with each frame of the trajectory. We need to explicitly point the analysis function to the frame that needs to be analysed with a frame_index: atomgroup.universe.trajectory[frame_index] in order to update the positions (and any other dynamic per-frame information) appropriately. Therefore, the first step to making the radgyr function parallelisable is to add a frame_index argument.

[7]:
```
def radgyr_per_frame(frame_index, atomgroup, masses, total_mass=None):

    # index the trajectory to set it to the frame_index frame
    atomgroup.universe.trajectory[frame_index]

    # coordinates change for each frame
    coordinates = atomgroup.positions
    center_of_mass = atomgroup.center_of_mass()

    # get squared distance from center
    ri_sq = (coordinates-center_of_mass)**2
    # sum the unweighted positions
    sq = np.sum(ri_sq, axis=1)
    sq_x = np.sum(ri_sq[:,[1,2]], axis=1) # sum over y and z
```

(continues on next page)

(continued from previous page)

```
    sq_y = np.sum(ri_sq[:,[0,2]], axis=1) # sum over x and z
    sq_z = np.sum(ri_sq[:,[0,1]], axis=1) # sum over x and y

    # make into array
    sq_rs = np.array([sq, sq_x, sq_y, sq_z])

    # weight positions
    rog_sq = np.sum(masses*sq_rs, axis=1)/total_mass
    # square root and return
    return np.sqrt(rog_sq)
```

Parallelization with multiprocessing

The native parallelisation module in Python is called multiprocessing. It contains useful tools to build a pool of working cores, map the function into different workers, and gather and order the results from all the workers.

Below we use `Pool` from `multiprocessing` as a context manager. We can define how many cores (or workers) we want to use with `Pool(n_jobs)`.

[8]:

```
import multiprocessing
from multiprocessing import Pool
from functools import partial
```

We use `functools.partial` to create a new method by supplying every argument needed for `radgyr_per_frame` *except* `frame_index`. We can do this because the `atomgroup`, `masses` etc. will not change when we iterate the function over each frame, but the `frame_index` will. We create a list of jobs where we use the `worker_pool` to map each `frame_index` to each job.

[9]:

```
run_per_frame = partial(radgyr_per_frame,
                        atomgroup=protein,
                        masses=protein.masses,
                        total_mass=np.sum(protein.masses))

frame_values = np.arange(u.trajectory.n_frames)
```

[10]:

```
with Pool(n_jobs) as worker_pool:
    result = worker_pool.map(run_per_frame, frame_values)
```

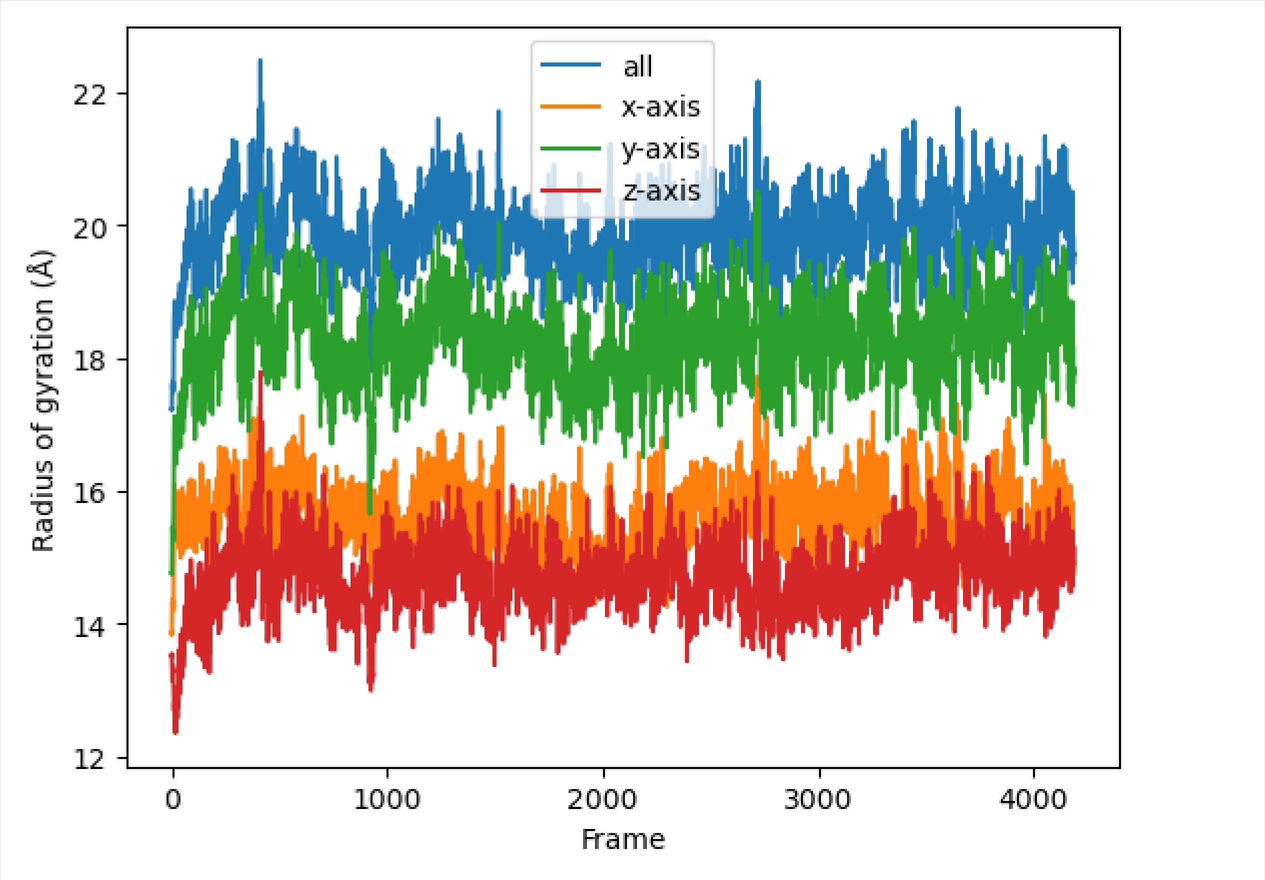
The `result` will be a list of arrays containing the result for each frame. Finally the results can be plotted along time.

[11]:

```
result = np.asarray(result).T

labels = ['all', 'x-axis', 'y-axis', 'z-axis']
for col, label in zip(result, labels):
    plt.plot(col, label=label)
plt.legend()
plt.ylabel('Radius of gyration (Å)')
plt.xlabel('Frame')
```

[11]:

```
Text(0.5, 0, 'Frame')
```



Parallelization with dask

Dask is a flexible library for parallel computing in Python. It provides advanced parallelism for analytics and has been integrated or utilized in many scientific softwares. It can be scaled from one single computer to a cluster of computers inside a HPC center.

Dask has a dynamic task scheduling system with synchronous (single-threaded), threaded, multiprocessing and distributed schedulers. The wrapping function in dask, `dask.delayed`, mimics for loops and wraps Python code into a Dask graph. This code can then be easily run in parallel, and visualized with `dask.visualize()` to examine if the task is well distributed. The code inside `dask.delayed` is not run immediately on execution, but pushed into a job queue waiting for submission. You can read more on dask website.

Comaring to `multiprocssing`, the downside of `multiprocessing` is that it is mostly focused on single-machine multicore parallelism (without extra manager). It is hard to operate on multimachine conditions. Below are two simple examples to use Dask to achieve the same task as `multiprocessing` does.

The API of `dask` is similar to `multiprocessing`. It also creates a pool of workers for your single machine with the given resources.

Note: The threaded scheduler in Dask (similar to `threading` in Python) should not be used as it will mess up with the state (timestep) of the trajectory.

[12]:
```
import dask
import dask.multiprocessing
dask.config.set(scheduler='processes')
```

```
[12]: <dask.config.set at 0x7f8bf93611c0>
```

Below is how you can utilize `dask.distributed` module to build a local cluster.

Note: this is not really needed for your laptop/desktop. Using dask.distributed may even slow down the performance, but it provides a diagnostic dashboard that can provide valuable insight on performance and progress.

See limitations here: https://distributed.dask.org/en/latest/limitations.html

```
[13]: from dask.distributed import Client

      client = Client(n_workers=n_jobs)
      client
```

```
[13]: <Client: 'tcp://127.0.0.1:35433' processes=2 threads=8, memory=33.18 GB>
```

First we have to create a list of jobs and transform them with `dask.delayed()` so they can be processed by Dask.

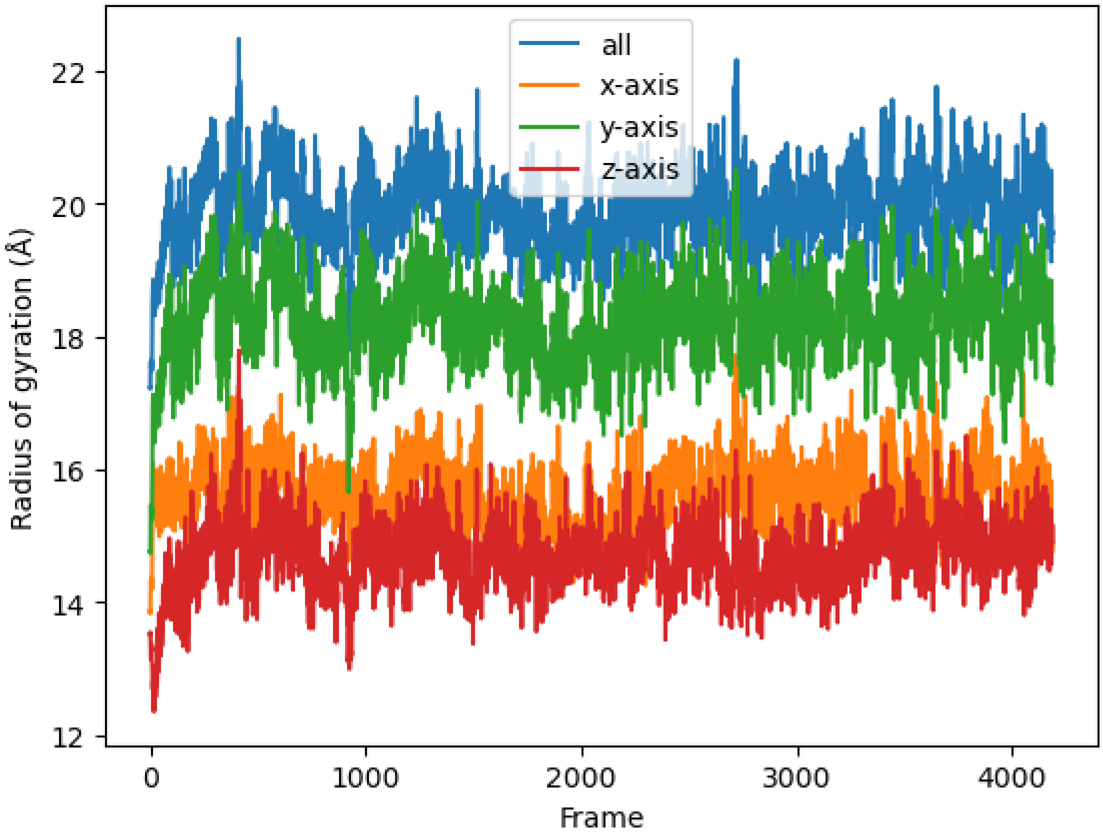
```
[14]: job_list = []
      for frame_index in range(u.trajectory.n_frames):
          job_list.append(dask.delayed(radgyr_per_frame(frame_index,
                                                        atomgroup=protein,
                                                        masses=protein.masses,
                                                        total_mass=np.sum(protein.masses))))
```

Then we simply use `dask.compute()` to get a list of ordered results.

```
[15]: result = dask.compute(job_list)
```

```
[16]: result = np.asarray(result).T

      labels = ['all', 'x-axis', 'y-axis', 'z-axis']
      for col, label in zip(result, labels):
          plt.plot(col, label=label)
      plt.legend()
      plt.ylabel('Radius of gyration (Å)')
      plt.xlabel('Frame')
```

```
[16]: Text(0.5, 0, 'Frame')
```



We can also use the old `radgyr` function because `dask` is more flexible on the input arguments.

Note: the associated timestamp of `protein` will change during the trajectory iteration, so the processes are always aware which timestep the trajectory is in and change the `protein` (e.g. its coordinates) accordingly.

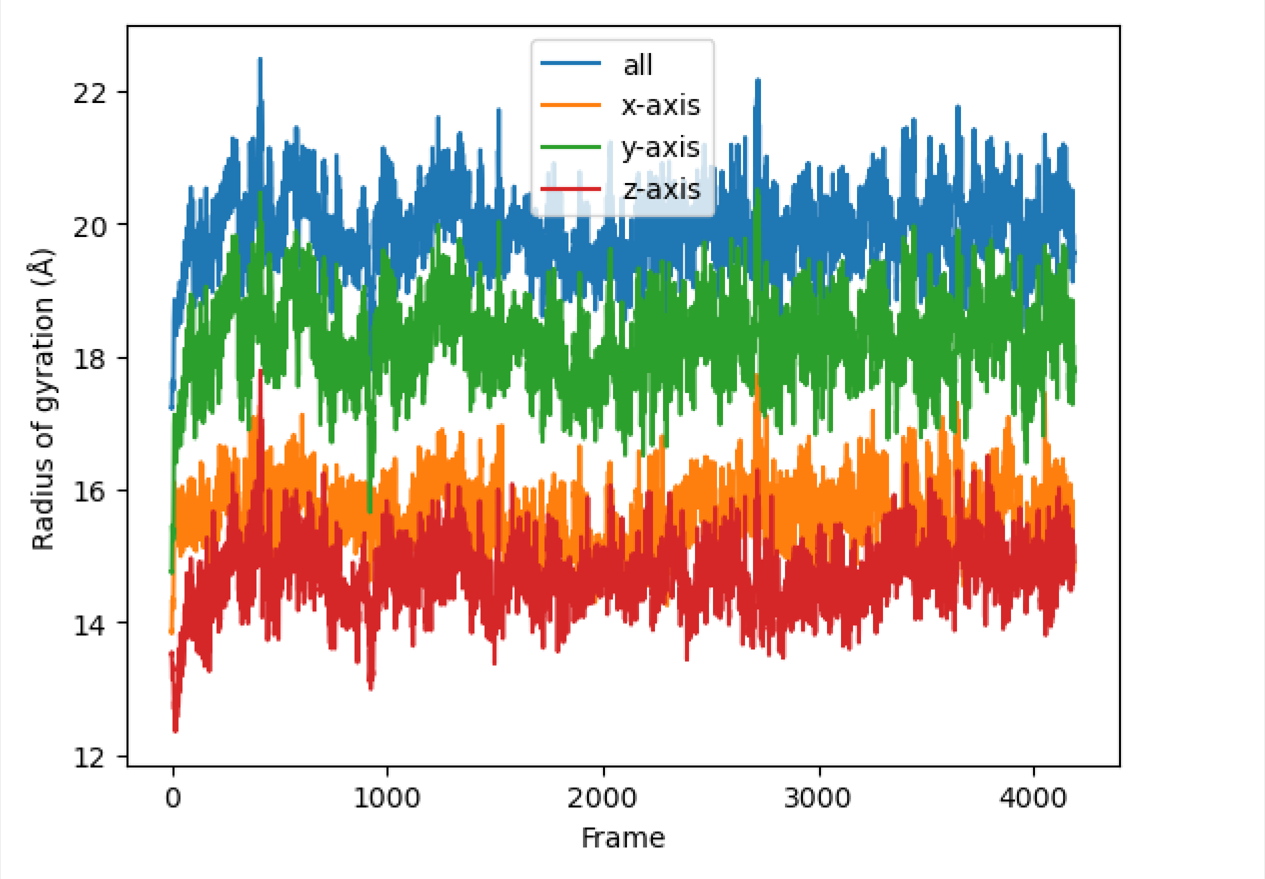
[17]:
```
job_list = []
for frame in u.trajectory:
    job_list.append(dask.delayed(radgyr(atomgroup=protein,
                                        masses=protein.masses,
                                        total_mass=np.sum(protein.masses))))
```

Then we simply use `dask.compute()` to get a list of ordered results.

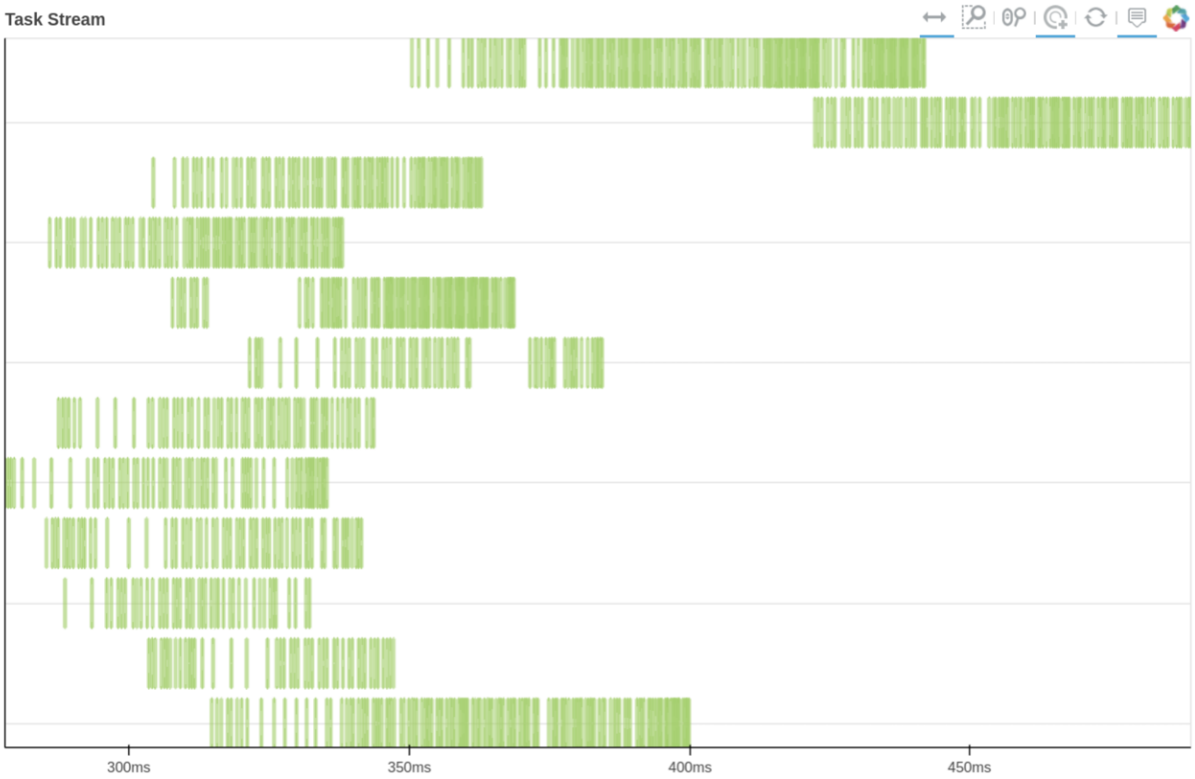
[18]:
```
result = dask.compute(job_list)
```

[19]:
```
result = np.asarray(result).T

labels = ['all', 'x-axis', 'y-axis', 'z-axis']
for col, label in zip(result, labels):
    plt.plot(col, label=label)
plt.legend()
plt.ylabel('Radius of gyration (Å)')
plt.xlabel('Frame')
```

[19]:
```
Text(0.5, 0, 'Frame')
```



We can also use Dask dashboard (with dask.distributed.Client) to examine how jobs are distributed along all the workers. Each green bar below represents one job, i.e. running `radgyr` on one frame of the trajectory. Be aware though, that your Task Stream will probably look different.



Parallelization in a split-apply-combine fashion

The aforementioned per-frame approach should normally be **avoided** because in **each** task, all the attributes (`AtomGroup`, `Universe`, and etc) need to be pickled. This pickling may take even more time than your lightweight analysis! Besides, in Dask, a significant amount of overhead time is needed to build a comprehensive Dask graph with thousands of tasks.

Therefore, we should normally use a split-apply-combine scheme for parallel trajectory analysis. Here, the trajectory is **split** into blocks, analysis is performed separately and in parallel on each block ("apply"), and then results from each block are gathered and **combined**.

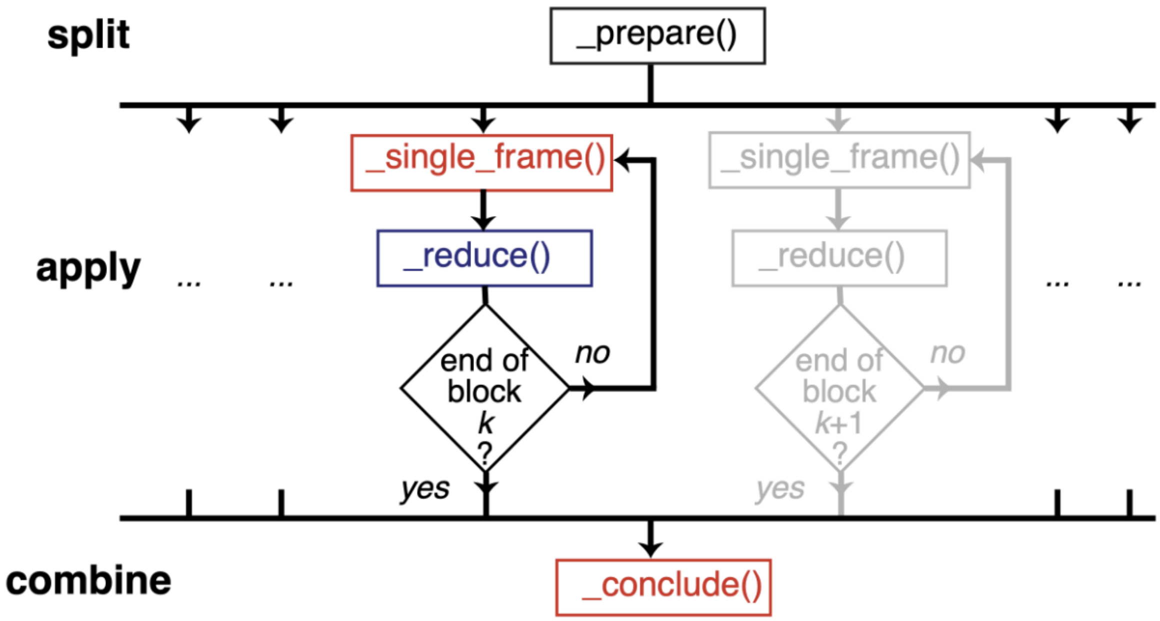


Image from ([SFPG+19]) used under CC-BY license.

We will show a simple illustration of split-apply-combine approach with dask below:

Block analysis function

`@dask.delayed` is a common syntax to decorate a function into delayed-enabled. It is the same as delaying the function by `dask.delayed(analyze_block)(bs, radgyr, ...)` later on.

[20]:
```
@dask.delayed
def analyze_block(blockslice, func, *args, **kwargs):
    result = []
    for ts in u.trajectory[blockslice.start:blockslice.stop]:
        A = func(*args, **kwargs)
        result.append(A)
    return result
```

Split the trajectory

This is a very simple way to split the trajectory. It splits the trajectory into defined `n_blocks` which is normally the same as the number of cores you want to use.

Since it is achieved by evenly dividing the `n_frames` by `n_blocks`, and setting the last block to end at the last frame, sometime it is not really balanced (e.g. the last block).

[21]:
```
n_frames = u.trajectory.n_frames
n_blocks = n_jobs   #  it can be any realistic value (0 < n_blocks <= n_jobs)

n_frames_per_block = n_frames // n_blocks
blocks = [range(i * n_frames_per_block, (i + 1) * n_frames_per_block) for i in range(n_
↪blocks-1)]
blocks.append(range((n_blocks - 1) * n_frames_per_block, n_frames))
```

[22]:

```
blocks
```

[22]:

```
[range(0, 2093), range(2093, 4187)]
```

Apply the analysis per block

[23]:

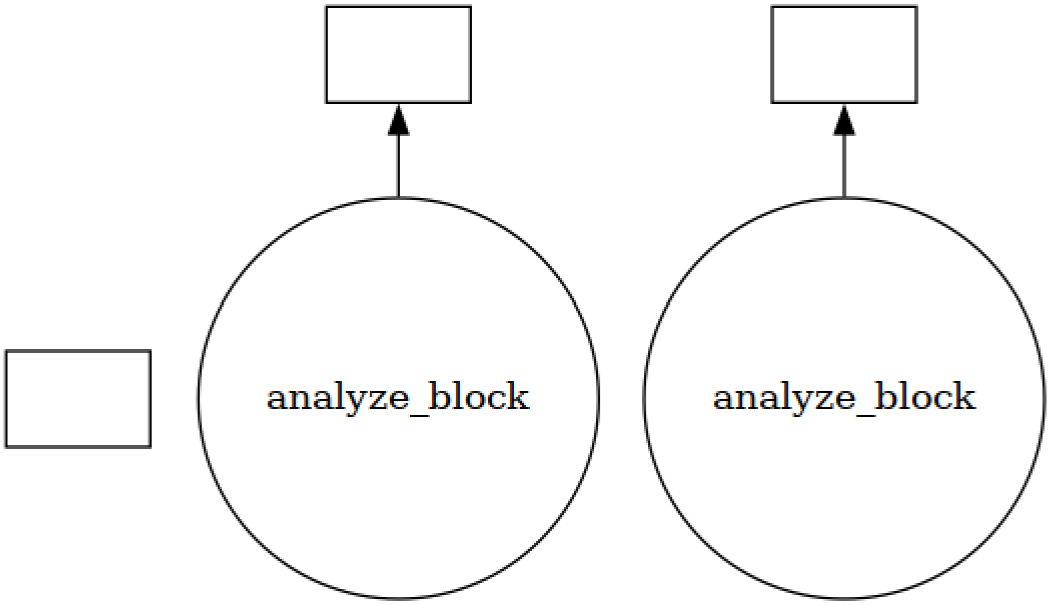
```
jobs = []
for bs in blocks:
    jobs.append(analyze_block(bs,
                              radgyr,
                              protein,
                              protein.masses,
                              total_mass=np.sum(protein.masses)))
jobs = dask.delayed(jobs)
```

Using `visualize()` we can see that the trajectory is split into a few blocks instead of ~4000 jobs.

[24]:

```
jobs.visualize()
```

[24]:



[25]:

```
results = jobs.compute()
```

Combine the results

[26]:

```
result = np.concatenate(results)
```

[27]:

```
result = np.asarray(result).T

labels = ['all', 'x-axis', 'y-axis', 'z-axis']
for col, label in zip(result, labels):
```

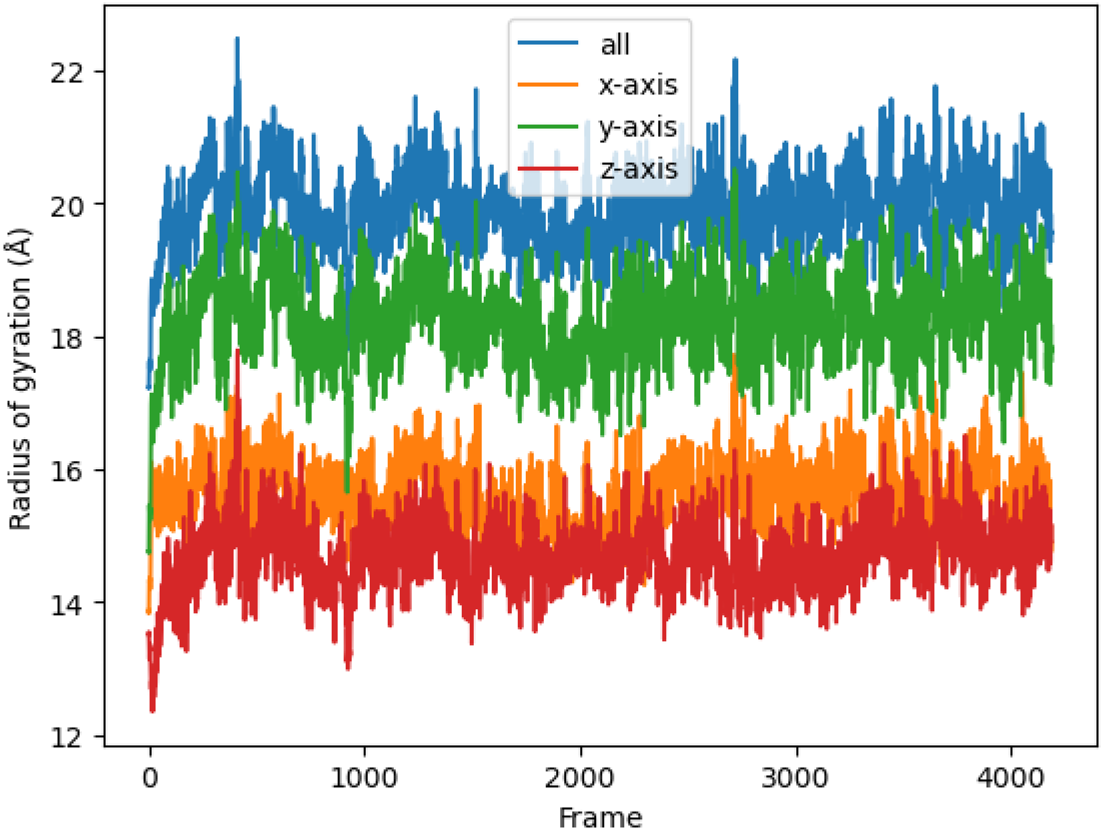
(continues on next page)

(continued from previous page)

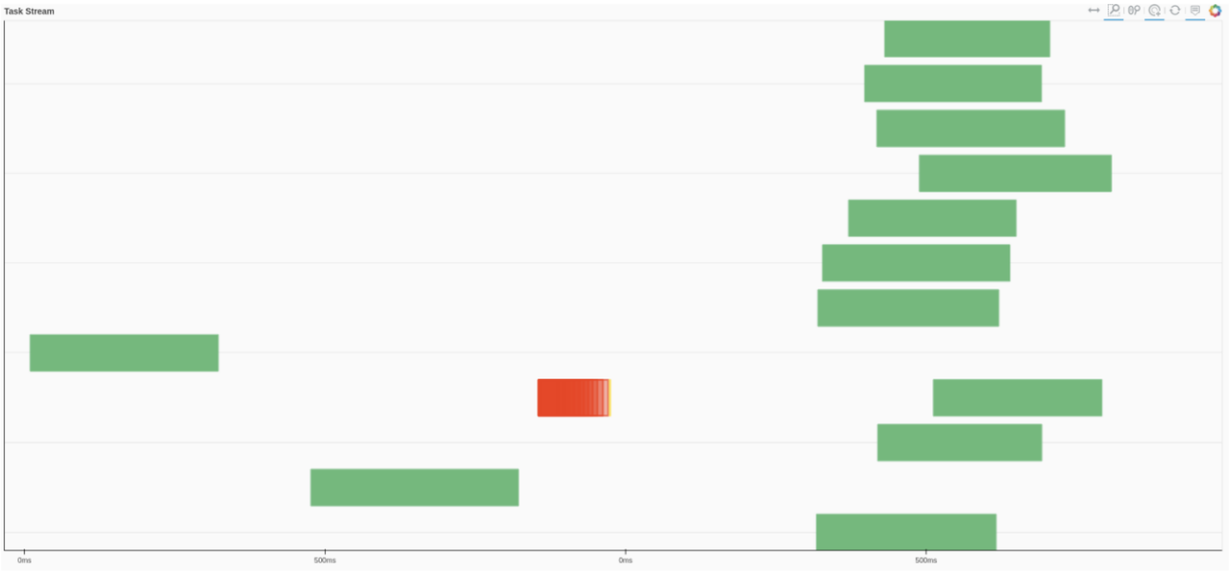
```
    plt.plot(col, label=label)
plt.legend()
plt.ylabel('Radius of gyration (Å)')
plt.xlabel('Frame')
```

[27]:

```
Text(0.5, 0, 'Frame')
```



If you look at the Dask dashboard (with dask.distributed.Client) this time, you will see each green bar below represents a per-block analysis for `radgyr`.



Other possible parallelism approaches for multiple analyses

You may want to perform multiple analyses (or analyze multiple trajectories). In this case, you can use some high-level parallelism, i.e. running all the serial analyses in parallel.

Here we use joblib. It is implemented on `multiprocessing` and provides lightweight pipelining in Python. Compared to `multiprocessing`, it has a simple API and convenient persistence of cached results."

[28]:
```python
from joblib import Parallel, delayed
import multiprocessing
num_cores = multiprocessing.cpu_count()
```

Here we leverage the power of `AnalysisFromFunction` to fast construct a class that will iterate through the trajectory and save the analysis results.

If you want to know more about how `AnalysisFromFunction` works, you can read it from Writing your own trajectory.

[29]:
```python
from MDAnalysis.analysis.base import AnalysisFromFunction
```

[30]:
```python
rog_1 = AnalysisFromFunction(radgyr, u.trajectory,
                             protein, protein.masses,
                             total_mass=np.sum(protein.masses))

rog_2 = AnalysisFromFunction(radgyr, u.trajectory,
                             protein, protein.masses,
                             total_mass=np.sum(protein.masses))

rog_3 = AnalysisFromFunction(radgyr, u.trajectory,
                             protein, protein.masses,
                             total_mass=np.sum(protein.masses))

rog_4 = AnalysisFromFunction(radgyr, u.trajectory,
                             protein, protein.masses,
                             total_mass=np.sum(protein.masses))
```

(continues on next page)

(continued from previous page)

```
analysis_ensemble = [rog_1, rog_2, rog_3, rog_4]
```

`run_analysis` is a simple way to run the analysis and retrieve the results.

```
[31]: def run_analysis(analysis):
          analysis.run()
          return analysis.results
```

The `joblib.delayed` is different from `dask.delayed`; it cannot be used as a "pie" syntax (`@joblib.delayed`), so you have to use it as below.

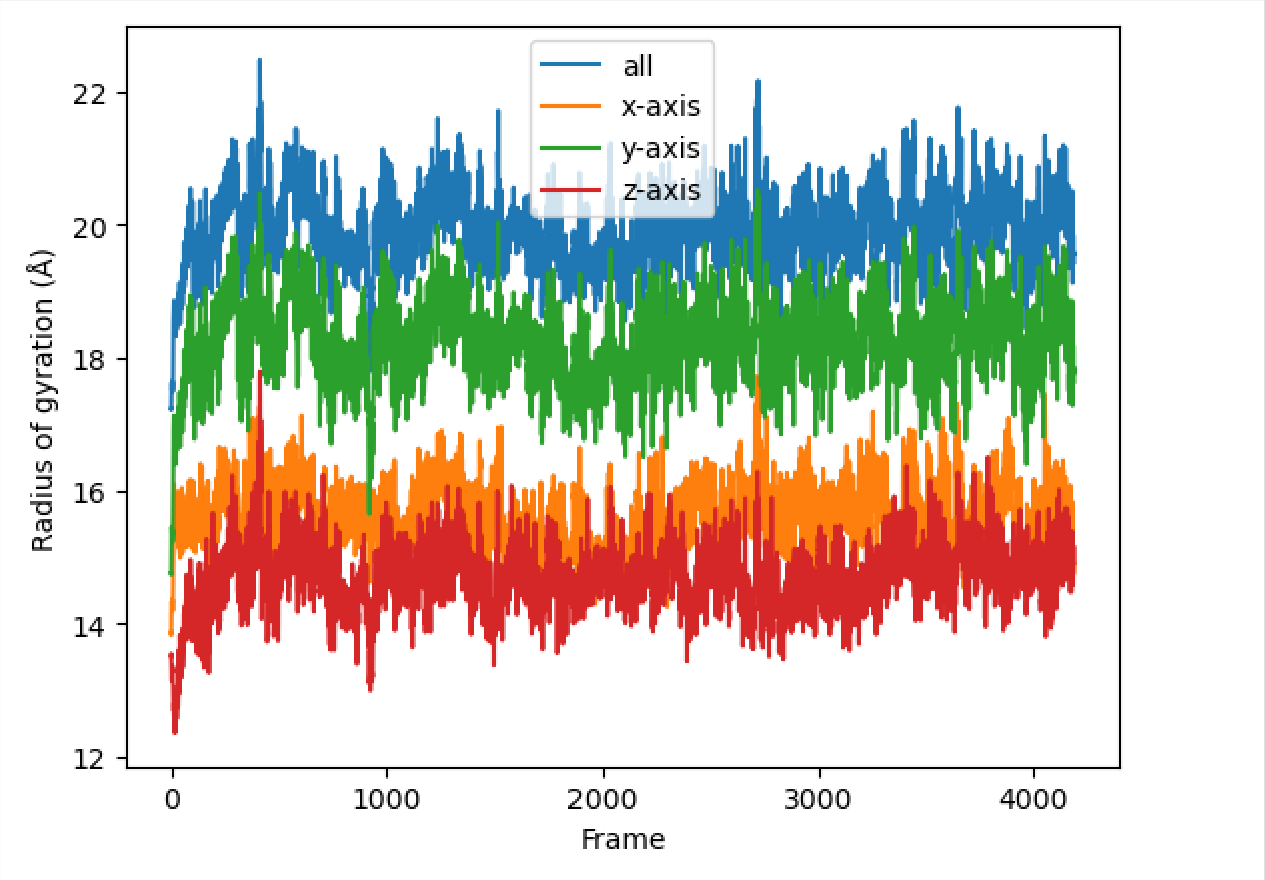
Similiar to `dask.delayed`, the code inside `joblib.delayed` will not run immediately but be pushed into a job queue waiting for processing. In this case, `run_anlaysis()` is processed by `Parallel` with defined number of workers–`n_jobs`.

```
[32]: results_ensemble = Parallel(n_jobs=num_cores)(delayed(run_analysis)(analysis)
                                                    for analysis in analysis_ensemble)
```

The `results_ensemble` will be a list of `results` that is returned from `run_analysis`. You can further split and process each analysis.

```
[33]: result_1 = np.asarray(results_ensemble[0]['timeseries']).T

      labels = ['all', 'x-axis', 'y-axis', 'z-axis']
      for col, label in zip(result_1, labels):
          plt.plot(col, label=label)
      plt.legend()
      plt.ylabel('Radius of gyration (Å)')
      plt.xlabel('Frame')
```

```
[33]: Text(0.5, 0, 'Frame')
```



See Also

The parallel version of MDAnalysis is still under development. For existing solutions and some implementations of parallel analysis, go to PMDA. PMDA ([SFPG+19]) applies the aforementioned split-apply-combine scheme with Dask. In the future, it may provide a framework that consolidates all the parallelisation schemes described in this tutorial."

References

[1] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. Journal of Molecular Biology, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164, doi:10.1016/j.jmb.2009.09.009.

[2] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. Proceedings of the 15th Python in Science Conference, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[3] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[4] Max Linke Shujie Fan, Ioannis Paraskevakos, Richard J. Gowers, Michael Gecht, and Oliver Beckstein. PMDA - Parallel Molecular Dynamics Analysis. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors,

Proceedings of the 18th Python in Science Conference, 134 – 142. 2019. doi:10.25080/Majora-7ddc1dd1-013.

2.1.32 Standard residues in MDAnalysis selections

Proteins

The residue names listed here are accessible via the “protein” keyword in the *Atom selection language*.

The below names are drawn from the CHARMM 27, OPLS-AA, GROMOS 53A6, AMBER 03, and AMBER 99sb*-ILDN force fields.

Protein backbone

Protein backbone atoms in MDAnalysis belong to a recognised protein residue and have the atom names:

Nucleic acids

The residue names listed here are accessible via the “nucleic” keyword in the *Atom selection language*.

The below names are drawn from largely from the CHARMM force field.

Nucleic backbone

Nucleic backbone atoms in MDAnalysis belong to a recognised nucleic acid residue and have the atom names:

Nucleobases

Nucleobase atoms from nucleic acid residues are recognised based on their names in CHARMM.

Nucleic sugars

Nucleic sugar atoms from nucleic acid residues are recognised by MDAnalysis if they have the atom names:

2.1.33 Advanced topology concepts

Adding a Residue or Segment to a Universe

To add a `Residue` or `Segment` to a topology, use the `Universe.add_Residue` or `Universe.add_Segment` methods.

```
>>> u = mda.Universe(PSF, DCD)
>>> u.segments
<SegmentGroup with 1 segment>
>>> u.segments.segids
array(['4AKE'], dtype=object)
>>> newseg = u.add_Segment(segid='X')
>>> u.segments.segids
array(['4AKE', 'X'], dtype=object)
```

(continues on next page)

(continued from previous page)

```
>>> newseg.atoms
<AtomGroup with 0 atoms>
```

To assign the last 100 residues from the `Universe` to this new Segment:

```
>>> u.residues[-100:].segments = newseg
>>> newseg.atoms
<AtomGroup with 1600 atoms>
```

Another example is *creating custom segments for protein domains*.

Molecules

In MDAnalysis, a molecule is a GROMACS-only concept that is relevant in some analysis methods. A group of atoms is considered a "molecule" if it is defined by the `[ moleculetype ]` section in a GROMACS topology. Molecules are only defined if a Universe is created from a GROMACS topology file (i.e. with a .tpr extension). Unlike fragments, they are not accessible directly from atoms.

```
>>> tpr = mda.Universe(TPR)
>>> tpr.atoms.molecules
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "MDAnalysis/core/groups.py", line 2278, in __getattr__
    cls=self.__class__.__name__, attr=attr))
AttributeError: AtomGroup has no attribute molecules
```

However, the order (`molnum`) and name (`moltype`) of each molecule is accessible as *topology attributes*:

```
>>> tpr.atoms.molnums
array([    0,     0,     0, ..., 11086, 11087, 11088])
>>> tpr.atoms.moltypes
array(['AKeco', 'AKeco', 'AKeco', ..., 'NA+', 'NA+', 'NA+'], dtype=object)
```

Adding custom TopologyAttrs

2.1.34 Example data

MDAnalysis offers a collection of example data files and datasets to run tests and tutorials. These are split into two packages:

- MDAnalysisTests: primarily for unit tests of the code
- MDAnalysisData: datasets for workshops and tutorials

MDAnalysisTests

While this is installed as a separate package, you should import files like so:

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD

u = mda.Universe(PSF, DCD)
```

A complete list of files and descriptions is in the `mdanalysis/testsuite/MDAnalysisTests/datafiles.py` file. The actual files are stored in the `mdanalysis/testsuite/MDAnalysisTests/data/` directory.

MDAnalysisData

The MDAnalysisData package is an interface to download, cache, and access certain datasets hosted on external repositories (e.g. figshare, zenodo, DataDryad). Data is not downloaded upon installation, so the package itself is small; but the directory where the datasets are cached can grow significantly.

You can access datasets like so:

```
import MDAnalysis as mda
from MDAnalysisData import datasets
adk = datasets.fetch_adk_equilibrium()
u = mda.Universe(adk.topology, adk.trajectory)

# to see the description of the dataset
print(adk.DESCR)
```

The cached files are stored by default in the `~/MDAnalysis_data` directory. This can be changed by setting the environment variable `MDANALYSIS_DATA`. You can change this for a Terminal session with the below command:

```
export MDANALYSIS_DATA=/my/chosen/path/MDAnalysis_data
```

Add it to your `.bashrc` for a permanent change.

```
echo 'export MDANALYSIS_DATA=/my/chosen/path/MDAnalysis_data' >> ~/.bashrc
```

In Python, you can check the location of your caching directory:

```
MDAnalysisData.base.get_data_home()
```

And clear the directory with:

```
MDAnalysisData.base.clear_data_home()
```

A list of datasets can be found at the MDAnalysisData documentation.

2.1.35 Contributing to MDAnalysis

MDAnalysis is a free and open source project. It evolves and grows with the demands of its user base. The development team very much welcomes contributions from its users. Contributions can take many forms, such as:

- **bug reports** or **enhancement requests** filed through the Issue Tracker
- **bug fixes**
- **improvements** to the code (speed, clarity, modernised code)
- **new features** in the code
- improvements and additions to **documentation** (including typo fixes)
- improvements to the **build systems**
- **questions** or **discussions** on **`GitHub Discussions`_**

The MDAnalysis community subscribes to a Code of Conduct that all community members agree and adhere to — please read it.

Important: The **MDAnalysis** and **MDAnalysisTests** packages are distributed under the GNU General Public License, version 2 (or any later version). This is a copyleft license: not only is MDAnalysis open-source, but all derivative work must be as well. **Any code, documentation, or files that you contribute to MDAnalysis will also be made available under this license.** Be sure that you are comfortable with that *before* you push additions to GitHub.

Parts of this page came from the Contributing to pandas guide.

Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements, and ideas are welcome.

If you are new to Git and version control, have a look at *Version control, Git, and GitHub*.

If you are looking to contribute a code or documentation fix, or core feature to MDAnalysis, and you are brand new to MDAnalysis or open-source development – we recommend going through the guides for *contributing to the main codebase* or the *user guide*.

If you are looking to contribute your own project or new addition to MDAnalysis, we encourage you to open an issue at the Issue Tracker or consider creating an MDAnalysis toolkit (MDAKit) . MDAKits are standalone packages that build on MDAnalysis to enhance its functionality. or solve specific scientific or technical problems. MDAKits can be optionally registered at the MDAKits registry to advertise to the broader MDAnalysis community. All packages on the registry are continuously tested against the latest and development versions of MDAnalysis.

For more on creating an MDAKit, please see the documentation on Making an MDAKit.

Version control, Git, and GitHub

Git is a version control system that allows many people to work together on a project. Working with Git can be one of the more daunting aspects of contributing to MDAnalysis. Sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account.

Some great resources for learning Git:

- the GitHub help pages.

- the NumPy's documentation.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

2.1.36 Contributing to the main codebase

If you would like to contribute, start by searching through the issues and pull requests to see whether someone else has raised a similar idea or question.

If you don't see your idea or problem listed, do one of the following:

- If your contribution is **minor**, such as a typo fix, go ahead and fix it by following the guide below and *open a pull request*.
- If your contribution is **major**, such as a bug fix or a new feature, start by opening an issue first. That way, other people can weigh in on the discussion before you do any work. If you also create a pull request, you should link it to the issue by including the issue number in the pull request's description.

Here is an overview of the development workflow for code or inline code documentation, as expanded on throughout the rest of the page.

1. *Fork the MDAnalysis repository* from the mdanalysis account into your own account
2. *Set up an isolated virtual environment* for code development
3. *Build development versions* of MDAnalysis and MDAnalysisTests on your computer into the virtual environment
4. *Create a new branch off the develop branch*
5. *Add your new feature or bug fix* or *add your new documentation*
6. *Add and run tests* (if adding to the code)
7. *Build and view the documentation* (if adding to the docs)
8. *Ensure PEP8 compliance (mandatory)* and format your code with Darker (optional)
9. *Commit and push your changes, and open a pull request.*

Working with the code

Forking

You will need your own fork to work on the code. Go to the MDAnalysis project page and hit the *Fork* button. You will want to clone your fork to your machine:

```
git clone https://github.com/your-user-name/mdanalysis.git
cd mdanalysis
git remote add upstream https://github.com/MDAnalysis/mdanalysis
```

This creates the directory *mdanalysis* and connects your repository to the upstream (main project) MDAnalysis repository.

Creating a development environment

To change code and test changes, you'll need to build both **MDAnalysis** and **MDAnalysisTests** from source. This requires a Python environment. We highly recommend that you use virtual environments. This allows you to have multiple experimental development versions of MDAnalysis that do not interfere with each other, or your own stable version. Since MDAnalysis is split into the actual package and a test suite, you need to install both modules in development mode.

You can do this either with *conda* or *pip*.

Note: If you are a first time contributor and/or don't have a lot of experience managing your own Python virtual environments, we **strongly** suggest using `conda`. You only need to follow the sections corresponding to the installation method you choose.

With conda

Install either Anaconda or miniconda. Make sure your conda is up to date:

```
conda update conda
```

Create a new environment with `conda create`. This will allow you to change code in an isolated environment without touching your base Python installation, and without touching existing environments that may have stable versions of MDAnalysis. :

```
conda create --name mdanalysis-dev "python>=3.9"
```

Use a recent version of Python that is supported by MDAnalysis for this environment.

Activate the environment to build MDAnalysis into it:

```
conda activate mdanalysis-dev
```

Warning: Make sure the `mdanalysis-dev` environment is active when developing MDAnalysis.

To view your environments:

```
conda info -e
```

To list the packages installed in your current environment:

```
conda list
```

Note: When you finish developing MDAnalysis you can deactivate the environment with `conda deactivate`, in order to return to your root environment.

See the full conda documentation for more details.

With pip and virtualenv

Like conda, virtual environments managed with virtualenv allow you to use different versions of Python and Python packages for your different project. Unlike conda, virtualenv is not a general-purpose package manager. Instead, it leverages what is available on your system, and lets you install Python packages using **pip**.

To use virtual environments you have to install the virtualenv package first. This can be done with pip:

```
python -m pip install virtualenv
```

Virtual environments can be created for each project directory.

```
cd my-project/
virtualenv my-project-env
```

This will create a new folder `my-project-env`. This folder contains the virtual environment and all packages you have installed in it. To activate it in the current terminal run:

```
source myproject-env/bin/activate
```

Now you can install packages via pip without affecting your global environment. The packages that you install when the environment is activated will be available in terminal sessions that have the environment activated.

Note: When you finish developing MDAnalysis you can deactivate the environment with `deactivate`, in order to return to your root environment.

The virtualenvwrapper package makes virtual environments easier to use. It provides some very useful features:

- it organises the virtual environment into a single user-defined directory, so they are not scattered throughout the file system;
- it defines commands for the easy creation, deletion, and copying of virtual environments;
- it defines a command to activate a virtual environment using its name;
- all commands defined by `virtualenvwrapper` have tab-completion for virtual environment names.

You first need to install `virtualenvwrapper` *outside* of a virtual environment:

```
python -m pip install virtualenvwrapper
```

Then, you need to load it into your terminal session. Add the following lines in `~/.bashrc`. They will be executed every time you open a new terminal session:

```
# Decide where to store the virtual environments
export WORKON_HOME=~/Envs
# Make sure the directory exists
mkdir -p ${WORKON_HOME}
# Load virtualenvwrapper
source /usr/local/bin/virtualenvwrapper.sh
```

Open a new terminal or run `source ~/.bashrc` to update your session. You can now create a virtual environment with:

```
mkvirtualenv my-project
```

Regardless of your current working directory, the environment is created in `~/Envs/` and it is now loaded in our terminal session.

You can load your virtual environments by running `workon my-project`, and exit them by running `deactivate`.

Virtual environments, especially with `virtualenvwrapper`, can do much more. For example, you can create virtual environments with different python interpreters with the `-p` flag. The Hitchhiker's Guide to Python has a good tutorial that gives a more in-depth explanation of virtual environments. The virtualenvwrapper documentation is also a good resource to read.

On a Mac

One more step is often required on macOS, because of the default number of files that a process can open simultaneously is quite low (256). To increase the number of files that can be accessed, run the following command:

```
ulimit -n 4096
```

This sets the number of files to 4096. However, this command only applies to your currently open terminal session. To keep this high limit, add the above line to your `~/.profile`.

Building MDAnalysis

With conda

Note: Make sure that you have *cloned the repository* and activated your virtual environment with `conda activate mdanalysis-dev`.

First we need to install dependencies. You'll need a mix of conda and pip installations:

```
conda install -c conda-forge \
  'Cython>=0.28' \
  'numpy>=1.21.0' \
  'biopython>=1.80' \
  'networkx>=2.0' \
  'GridDataFormats>=0.4.0' \
  'mmtf-python>=1.0.0' \
  'joblib>=0.12' \
  'scipy>=1.5.0' \
  'matplotlib>=1.5.1' \
  'tqdm>=4.43.0' \
  'threadpoolctl'\
  'packaging' \
  'fasteners' \
  'netCDF4>=1.0' \
  'h5py>=2.10' \
  'chemfiles>=0.10' \
  'pyedr>=0.7.0' \
  'pytng>=0.2.3' \
  'gsd>3.0.0' \
  'rdkit>=2020.03.1' \
  'parmed' \
```

(continues on next page)

(continued from previous page)

```
  'seaborn' \
  'scikit-learn' \
  'tidynamics>=1.0.0' \
  'mda-xdrlib'

# documentation dependencies
conda install -c conda-forge 'mdanalysis-sphinx-theme>=1.3.0' docutils sphinx-
↪sitemap sphinxcontrib-bibtex pybtex pybtex-docutils
python -m pip install docutils sphinx-sitemap sphinxcontrib-bibtex pybtex␣
↪pybtex-docutils
```

Ensure that you have a working C/C++ compiler (e.g. gcc or clang). You will also need Python 3.9. We will now install MDAnalysis.

```
# go to the mdanalysis source directory
cd mdanalysis/

# Build and install the MDAnalysis package
cd package/
python -m pip install -e .

# Build and install the test suite
cd ../testsuite/
python -m pip install -e .
```

At this point you should be able to import MDAnalysis from your locally built version. If you are running the development version, this is visible from the version number ending in `-dev0`. For example:

```
$ python  # start an interpreter
>>> import MDAnalysis as mda
>>> mda.__version__
'2.6.0-dev0'
```

With pip and virtualenv

Note: Make sure that you have *cloned the repository* and activated your virtual environment with `source myproject-env/bin/activate` (or `workon my-project` if you used the virtualenvwrapper package)

Install the dependencies:

```
python -m pip install \
  'Cython>=0.28' \
  'numpy>=1.21.0' \
  'biopython>=1.80' \
  'networkx>=2.0' \
  'GridDataFormats>=0.4.0' \
  'mmtf-python>=1.0.0' \
  'joblib>=0.12' \
  'scipy>=1.5.0' \
  'matplotlib>=1.5.1' \
```

(continues on next page)

(continued from previous page)

```
  'tqdm>=4.43.0' \
  'threadpoolctl'\
  'packaging' \
  'fasteners' \
  'netCDF4>=1.0' \
  'h5py>=2.10' \
  'chemfiles>=0.10' \
  'pyedr>=0.7.0' \
  'pytng>=0.2.3' \
  'gsd>3.0.0' \
  'rdkit>=2020.03.1' \
  'parmed' \
  'seaborn' \
  'scikit-learn' \
  'tidynamics>=1.0.0' \
  'mda-xdrlib'

# for building documentation
python -m pip install \
  sphinx 'mdanalysis-sphinx-theme>=1.3.0' sphinx-sitemap \
  pybtex docutils pybtex-docutils sphinxcontrib-bibtex
```

Some packages, such as `clustalw`, are not available via pip.

Ensure that you have a working C/C++ compiler (e.g. gcc or clang). You will also need Python 3.9. We will now install MDAnalysis.

```
# go to the mdanalysis source directory
cd mdanalysis/

# Build and install the MDAnalysis package
cd package/
python -m pip install -e .

# Build and install the test suite
cd ../testsuite/
python -m pip install -e .
```

At this point you should be able to import MDAnalysis from your locally built version. If you are running the development version, this is visible from the version number ending in "-dev0". For example:

```
$ python  # start an interpreter
>>> import MDAnalysis as mda
>>> mda.__version__
'2.7.0-dev0'
```

Branches in MDAnalysis

The most important branch of MDAnalysis is the `develop` branch, to which all development code for the next release is pushed.

The `develop` branch can be considered an "integration" branch for including your code into the next release. Only working, tested code should be committed to this branch. All code contributions ("features") should branch off `develop`. At each release, a snapshot of the `develop` branch is taken, packaged and uploaded to PyPi and conda-forge.

Creating a branch

The develop branch should only contain approved, tested code, so create a feature branch for making your changes. For example, to create a branch called `shiny-new-feature` from `develop`:

```
git checkout -b shiny-new-feature develop
```

This changes your working directory to the `shiny-new-feature` branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to MDAnalysis. You can have many branches with different names and switch in between them using the `git checkout my-branch-name` command.

There are several special branch names that you should not use for your feature branches:

- `master`
- `develop`
- `package-*`
- `gh-pages`

`package` branches are used to *prepare a new production release* and should be handled by the release manager only.

`master` is the old stable code branch and is kept protected for historical reasons.

`gh-pages` is where built documentation to be uploaded to github pages is held.

Writing new code

Code formatting in Python

MDAnalysis is a project with a long history and many contributors; it hasn't used a consistent coding style. Since version 0.11.0, we are trying to update all the code to conform with PEP8. Our strategy is to update the style every time we touch an old function and thus switch to PEP8 continuously.

Important requirements (from PEP8):

- keep line length to **79 characters or less**; break long lines sensibly
- indent with **spaces** and use **4 spaces per level**
- naming:
  - classes: *CapitalClasses* (i.e. capitalized nouns without spaces)
  - methods and functions: *underscore_methods* (lower case, with underscores for spaces)

We recommend that you use a Python Integrated Development Environment (IDE) (PyCharm and others) or external tools like flake8 for code linting. For integration of external tools with emacs and vim, check out elpy (emacs) and python-mode (vim).

To apply the code formatting in an automated way, you can also use code formatters. External tools include autopep8 and yapf. Most IDEs either have their own code formatter or will work with one of the above through plugins.

Modules and dependencies

MDAnalysis strives to keep dependencies small and lightweight. Code outside the `MDAnalysis.analysis` and `MDAnalysis.visualization` modules should only rely on the *core dependencies*, which are always installed. Analysis and visualization modules can use any *any package, but the package is treated as optional*.

Imports in the code should follow the *General rules for importing*.

See also:

See *Module imports in MDAnalysis* for more information.

Developing in Cython

The `setup.py` script first looks for the *.c* files included in the standard MDAnalysis distribution. These are not in the GitHub repository, so `setup.py` will use Cython to compile extensions. *.pyx* source files are used instead of *.c* files. From there, *.pyx* files are converted to *.c* files if they are newer than the already present *.c* files or if the `--force` flag is set (i.e. `python setup.py build --force`). End users (or developers) should not trigger the *.pyx* to *.c* conversion, since *.c* files delivered with source packages are always up-to-date. However, developers who work on the *.pyx* files will automatically trigger the conversion since *.c* files will then be outdated.

Place all source files for compiled shared object files into the same directory as the final shared object file.

.pyx files and cython-generated *.c* files should be in the same directory as the *.so* files. External dependent C/C++/Fortran libraries should be in dedicated `src/` and `include/` folders. See the following tree as an example:

```
MDAnalysis
    |--lib
    |   |-- _distances.so
    |   |-- distances.pyx
    |   |-- distances.c
    |-- coordinates
        |-- _dcdmodule.so
        |-- src
            |-- dcd.c
        |-- include
            |-- dcd.h
```

Testing your code

MDAnalysis takes testing seriously. All code added to MDAnalysis should have tests to ensure that it works as expected; we aim for 90% coverage. See *Tests in MDAnalysis* for more on *writing*, *running*, and interpreting tests.

Documenting your code

Changes to the code should be reflected in the ongoing CHANGELOG. Add an entry here to document your fix, enhancement, or change. In addition, add your name to the author list. If you are addressing an issue, make sure to include the issue number.

Ensure PEP8 compliance (mandatory) and format your code with Darker (optional)

Darker is a *partial formatting* tool that helps to reformat new or modified code lines so the codebase progressively adapts a code style instead of doing a full reformat, which would be a big commitment. It was designed with the black formatter in mind, hence the name.

In MDAnalysis **we only require PEP8 compliance**, so if you want to make sure that your PR passes the darker bot, you'll need both darker and flake8:

```
pip install darker flake8
```

You'll also need the original codebase so darker can first get a diff between the current develop branch and your code. After making your changes to your local copy of the **MDAnalysis** repo, add the remote repo (here we're naming it upstream), and fetch the content:

```
git remote add upstream https://github.com/MDAnalysis/mdanalysis.git
git fetch upstream
```

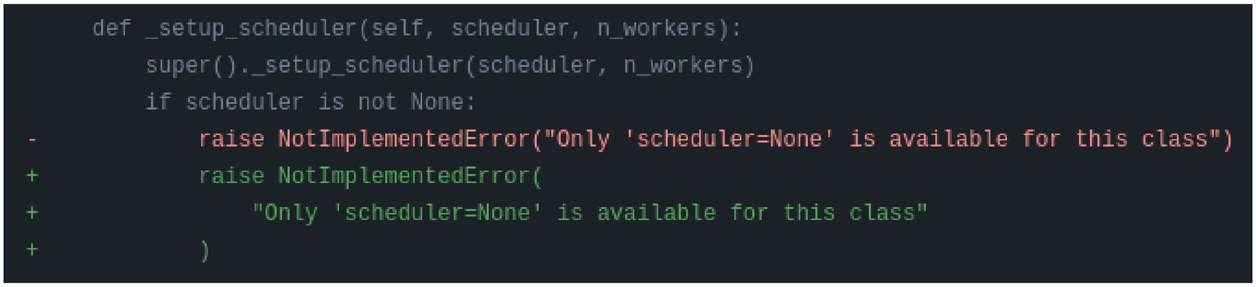
Now you can check your modifications on the package:

```
darker --diff -r upstream/develop package/MDAnalysis -L flake8
```

and the test suite:

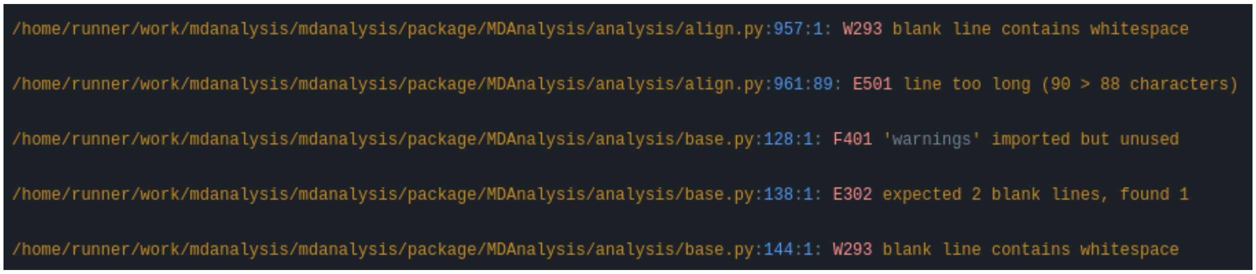
```
darker --diff -r upstream/develop testsuite/MDAnalysisTests -L flake8
```

Darker will first suggest changes so that the new code lines comply with black's rules, like this:



```
     def _setup_scheduler(self, scheduler, n_workers):
         super()._setup_scheduler(scheduler, n_workers)
         if scheduler is not None:
-            raise NotImplementedError("Only 'scheduler=None' is available for this class")
+            raise NotImplementedError(
+                "Only 'scheduler=None' is available for this class"
+            )
```

and then show flake8 errors and warnings. These look like this:



```
/home/runner/work/mdanalysis/mdanalysis/package/MDAnalysis/analysis/align.py:957:1: W293 blank line contains whitespace

/home/runner/work/mdanalysis/mdanalysis/package/MDAnalysis/analysis/align.py:961:89: E501 line too long (90 > 88 characters)

/home/runner/work/mdanalysis/mdanalysis/package/MDAnalysis/analysis/base.py:128:1: F401 'warnings' imported but unused

/home/runner/work/mdanalysis/mdanalysis/package/MDAnalysis/analysis/base.py:138:1: E302 expected 2 blank lines, found 1

/home/runner/work/mdanalysis/mdanalysis/package/MDAnalysis/analysis/base.py:144:1: W293 blank line contains whitespace
```

You are free to skip the diffs and then manually fix the PEP8 faults. Or if you're ok with the suggested formatting changes, just apply the suggested fixes:

```
darker -r upstream/develop package/MDAnalysis -L flake8
darker -r upstream/develop testsuite/MDAnalysisTests -L flake8
```

Adding your code to MDAnalysis

Committing your code

When you are happy with a set of changes and *all the tests pass*, it is time to commit. All changes in one revision should have a common theme. If you implemented two rather different things (say, one bug fix and one new feature), then split them into two commits with different messages.

Once you've made changes to files in your local repository, you can see them by typing:

```
git status
```

Tell git to track files by typing:

```
git add path/to/file-to-be-added.py
```

Doing `git status` again should give something like:

```
# On branch shiny-new-feature
#
#       modified:   /relative/path/to/file-you-added.py
#
```

Then commit with:

```
git commit -m
```

This opens up a message editor.

Always add a descriptive comment for your commit message (feel free to be verbose!):

- use a short (<50 characters) subject line that summarizes the change
- leave a blank line
- optionally, add additional more verbose descriptions; paragraphs or bullet lists (with - or *) are good
- manually break lines at 80 characters
- manually indent bullet lists

See also:

See Tim Pope's A Note About Git Commit Messages for a rationale for these rules.

Pushing your code to GitHub

When you want your changes to appear publicly on your GitHub page, push your forked feature branch's commits:

```
git push origin shiny-new-feature
```

Here *origin* is the default name given to your remote repository on GitHub. You can see the remote repositories:

```
git remote -v
```

If you added the upstream repository as described above you will see something like:

```
origin  git@github.com:your-username/mdanalysis.git (fetch)
origin  git@github.com:your-username/mdanalysis.git (push)
upstream        git@github.com:MDAnalysis/mdanalysis.git (fetch)
upstream        git@github.com:MDAnalysis/mdanalysis.git (push)
```

Now your code is on GitHub, but it is not yet a part of the MDAnalysis project. For that to happen, a pull request needs to be submitted on GitHub.

Rebasing your code

Often the upstream MDAnalysis develop branch will be updated while you are working on your own code. You will then need to update your own branch with the new code to avoid merge conflicts. You need to first retrieve it and then rebase your branch so that your changes apply to the new code:

```
git fetch upstream
git rebase upstream/develop
```

This will replay your commits on top of the latest development code from MDAnalysis. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to `git stash` them prior to updating. This will effectively store your changes and they can be reapplied after updating with `git stash apply`.

Once rebased, push your changes:

```
git push -f origin shiny-new-feature
```

and create a pull request.

Creating a pull request

The typical approach to adding your code to MDAnalysis is to make a pull request on GitHub. Please make sure that your contribution *passes all tests*. If there are test failures, you will need to address them before we can review your contribution and eventually merge them. If you have problems with making the tests pass, please ask for help! (You can do this in the comments of the pull request).

1. Navigate to your repository on GitHub
2. Click on the *Pull Request* button
3. You can then click on *Commits* and *Files Changed* to make sure everything looks okay one last time
4. Write a description of your changes and follow the PR checklist
    - check that docs are updated

   - check that tests run
   - check that you've updated CHANGELOG
   - reference the issue that you address, if any

5. Click *Send Pull Request*.

Your pull request is then sent to the repository maintainers. After this, the following happens:

1. A *suite of tests are run on your code* with the tools travis, appveyor and *Codecov*. If they fail, please fix your pull request by pushing updates to it.
2. Developers will ask questions and comment in the pull request. You may be asked to make changes.
3. When everything looks good, a core developer will merge your code into the `develop` branch of MDAnalysis. Your code will be in the next release.

If you need to make changes to your code, you can do so on your local repository as you did before. Committing and pushing the changes will update your pull request and restart the automated tests.

Working with the code documentation

MDAnalysis maintains two kinds of documentation:

1. This user guide: a map of how MDAnalysis works, combined with tutorial-like overviews of specific topics (such as the analyses)
2. The documentation generated from the code itself. Largely built from code docstrings, these are meant to provide a clear explanation of the usage of individual classes and functions. They often include technical or historical information such as in which version the function was added, or deprecation notices.

This guide is for the documentation generated from the code. If you are looking to contribute to the user guide, please see *Contributing to the user guide*.

MDAnalysis has a lot of documentation in the Python doc strings. The docstrings follow the Numpy Docstring Standard, which is used widely in the Scientific Python community. They are nice to read as normal text and are converted by sphinx to normal ReST through napoleon.

This standard specifies the format of the different sections of the docstring. See this document for a detailed explanation, or look at some of the existing functions to extend it in a similar manner.

Note that each page of the online documentation has a link to the *Source* of the page. You can look at it in order to find out how a particular page has been written in reST and copy the approach for your own documentation.

Building the documentation

The online documentation is generated from the pages in `mdanalysis/package/doc/sphinx/source/documentation_pages`. The documentation for the current release are hosted at www.mdanalysis.org/docs, while the development version is at www.mdanalysis.org/mdanalysis/.

In order to build the documentation, you must first *clone the main MDAnalysis repo*. *Set up a virtual environment* in the same way as you would for the code (you should typically use the same environment as you do for the code). Build the development version of MDAnalysis.

Then, generate the docs with:

```
cd doc/sphinx && make html
```

This generates and updates the files in `doc/html`. If the above command fails with an `ImportError`, run

```
python setup.py build_ext --inplace
```

and retry.

You will then be able to open the home page, `doc/html/index.html`, and look through the docs. In particular, have a look at any pages that you tinkered with. It is typical to go through multiple cycles of fix, rebuild the docs, check and fix again.

If rebuilding the documentation becomes tedious after a while, install the *sphinx-autobuild* extension.

Where to write docstrings

When writing Python code, you should always add a docstring to each public (visible to users):

- module
- function
- class
- method

When you add a new module, you should include a docstring with a short sentence describing what the module does, and/or a long document including examples and references.

Guidelines for writing docstrings

A typical function docstring looks like the following:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of `arg1`
    arg2 : str
        Description of `arg2`


    Returns
    -------
    bool
        Description of return value

    """
    return True
```

See also:

The napoleon documentation has further breakdowns of docstrings at the module, function, class, method, variable, and other levels.

- When writing reST markup, make sure that there are **at least two blank lines above** the reST after a numpy heading. Otherwise, the Sphinx/napoleon parser does not render correctly.

  ```
  some more docs bla bla

  Notes
  -----
  THE NEXT TWO BLANK LINES ARE IMPORTANT.


  .. versionadded:: 0.16.0
  ```

- Do not use "Example" or "Examples" as a normal section heading (e.g. in module level docs): *only* use it as a NumPy doc Section. It will not be rendered properly, and will mess up sectioning.
- When writing multiple common names in one line, Sphinx sometimes tries to reference the first name. In that case, you have to split the names across multiple lines. See below for an example:

  ```
  Parameters
  ----------
  n_atoms, n_residues : int
      numbers of atoms/residues
  ```

- We are using MathJax with sphinx so you can write LaTeX code in math tags.

  In blocks, the code below

  ```
  #<SPACE if there is text above equation>
  .. math::
      e^{i\pi} = -1
  ```

  renders like so:

  $$e^{i\pi} = -1$$

  Math directives can also be used inline.

  ```
  We make use of the identity :math:`e^{i\pi} = -1` to show...
  ```

  Note that you should *always* make doc strings with math code **raw** python strings **by prefixing them with the letter "r"**, or else you will get problems with backslashes in unexpected places.

  ```
  def rotate(self, R):
      r"""Apply a rotation matrix *R* to the selection's coordinates.

      :math:`\mathsf{R}` is a 3x3 orthogonal matrix that transforms a␣
  ↪vector
      :math:`\mathbf{x} \rightarrow \mathbf{x}'`:

      .. math::

      \mathbf{x}' = \mathsf{R}\mathbf{x}
      """
  ```

See also:

See Stackoverflow: Mathjax expression in sphinx python not rendering correctly for further discussion.

Documenting changes

We use reST constructs to annotate *additions*, *changes*, and *deprecations* to the code so that users can quickly learn from the documentation in which version of MDAnalysis the feature is available.

A **newly added module/class/method/attribute/function** gets a versionadded directive entry in its primary doc section, as below.

```
.. versionadded:: X.Y.Z
```

For parameters and attributes, we typically mention the new entity in a versionchanged section of the function or class (although a versionadded would also be acceptable).

Changes are indicated with a versionchanged directive

```
.. versionchanged:: X.Y.Z
   Description of the change. Can contain multiple descriptions.
   Don't assume that you get nice line breaks or formatting, write your text in
   full sentences that can be read as a paragraph.
```

Deprecations (features that are not any longer recommended for use and that will be removed in future releases) are indicated by the deprecated directive:

```
.. deprecated:: X.Y.Z
   Describe (1) alternatives (what should users rather use) and
   (2) in which future release the feature will be removed.
```

When a feature is removed, we remove the deprecation notice and add a versionchanged to the docs of the enclosing scope. For example, when a parameter of a function is removed, we update the docs of the function. Function/class removal are indicated in the module docs. When we remove a whole module, we typically indicate it in the top-level reST docs that contain the TOC tree that originally included the module.

Writing docs for abstract base classes

MDAnalysis contains a number of abstract base classes, such as `AnalysisBase`. Developers who define new base classes, or modify existing ones, should follow these rules:

- The *class docstring* needs to contain a list of methods that can be overwritten by inheritance from the base class. Distinguish and document methods as required or optional.
- The class docstring should contain a minimal example for how to derive this class. This demonstrates best practices, documents ideas and intentions behind the specific choices in the API, helps to promote a unified code base, and is useful for developers as a concise summary of the API.
- A more detailed description of methods should come in the *method docstring*, with a note specifying if the method is required or optional to overwrite.

See the documentation of `MDAnalysis.analysis.base.AnalysisBase` for an example of this documentation.

Adding your documentation to MDAnalysis

As with any contribution to an MDAnalysis repository, *commit and push* your documentation contributions to GitHub. If *any fixes in the restructured text* are needed, *put them in their own commit* (and do not include any generated files under *docs/html*). Try to keep all reST fixes in the one commit. `git add FILE` and `git commit --amend` is your friend when piling more and more small reST fixes onto a single "fixed reST" commit.

We recommend *building the docs locally first* to preview your changes. Then, *create a pull request*. All the tests in the MDAnalysis test suite will run, but only one checks that the documents compile correctly.

Viewing the documentation interactively

In the Python interpreter one can simply say:

```
import MDAnalysis
help(MDAnalysis)
help(MDAnalysis.Universe)
```

In `ipython` one can use the question mark operator:

```
In [1]: MDAnalysis.Universe?
```

2.1.37 Contributing to the user guide

MDAnalysis maintains two kinds of documentation:

1. This user guide: a map of how MDAnalysis works, combined with tutorial-like overviews of specific topics (such as the analyses)
2. The documentation generated from the code itself. Largely built from code docstrings, these are meant to provide a clear explanation of the usage of individual classes and functions. They often include technical or historical information such as in which version the function was added, or deprecation notices.

This guide is about how to contribute to the user guide. If you are looking to add to documentation of the main code base, please see *Working with the code documentation*.

The user guide makes use of a number of Sphinx extensions to ensure that the code examples are always up-to-date. These include nbsphinx and the ipython directive.

The `ipython` directive lets you put code in the documentation which will be run during the doc build. For example:

```
.. ipython:: python

    x = 2
    x**3
```

will be rendered as:

```
In [1]: x = 2

In [2]: x**3
Out[2]: 8
```

Many code examples in the docs are run during the doc build. This approach means that code examples will always be up to date, but it does make the doc building a bit more complex.

Here is an overview of the development workflow for the user guide, as expanded on throughout the rest of the page.

1. *Fork the MDAnalysis repository* from the mdanalysis account into your own account
2. *Set up an isolated virtual environment* for your documentation
3. *Create a new branch off the develop branch*
4. Add your new documentation.
5. *Build and view the documentation*.
6. *Test your notebook cells, if applicable*.
7. *Commit and push your changes, and open a pull request*.

Forking and cloning the User Guide

Go to the MDAnalysis project page and hit the *Fork* button. You will want to clone your fork to your machine:

```
git clone https://github.com/your-user-name/UserGuide.git
cd UserGuide
git remote add upstream https://github.com/MDAnalysis/UserGuide
```

This creates the directory *UserGuide* and connects your repository to the upstream (main project) MDAnalysis repository.

Creating a development environment

Create a new virtual environment for the user guide. Install the required dependencies, and activate the `nglview` extension. We use `nglview` for visualizing molecules in Jupyter notebook tutorials.

Using `conda` or similar (`miniconda`, `mamba`, `micromamba`), create a new environment with all the dependencies:

```
cd UserGuide/
conda env create --file environment.yml --quiet
conda activate mda-user-guide
jupyter-nbextension enable nglview --py --sys-prefix
```

Building the user guide

Navigate to the `doc/` directory and run `make html`:

```
cd doc/
make html
```

The HTML output will be in `doc/build/`, which you can open in your browser of choice. The homepage is `doc/build/index.html`.

If rebuilding the documentation becomes tedious after a while, install the *sphinx-autobuild* extension.

Saving state in Jupyter notebooks

One of the neat things about `nglview` is the ability to interact with molecules via the viewer. This ability can be preserved for the HTML pages generated from Jupyer notebooks by `nbsphinx`, if you save the notebook widget state after execution.

Test with pytest and nbval

Whenever you add or modify notebook cells, you should make sure they run without errors and that their outputs are consistent, since they are part of the documentation as well.

We use a pytest plugin for this called nbval, it takes advantage of the saved notebook state and re-runs the notebook to determine if its output is still identical to the saved state. Thus, cells not only have to work (no errors), but also must give the same output they gave when they were saved.

To test all notebooks, just cd into `UserGuide/tests` and run `pytest`. If you want to test a particular notebook, check the the contents of *pytest.ini*, this file defines flags that you can also pass directly to pytest. For example, if you wanted to test the *hole2* notebook:

```
pytest --nbval --nbval-current-env --nbval-sanitize-with ./sanitize_output.cfg ../doc/
↪source/examples/analysis/polymers_and_membranes/hole2.ipynb
```

Where `--nbval` tells pytest to use nbval to test Jupyter notebooks, `--nbval-current-env` to use the currently loaded python environment (make sure you actually loaded your environment) instead of trying to use the one that was used when the notebook was saved and `--nbval-sanitize-with` to point pytest to a file full of replacement rules like this one for example:

```
regex: (.*B \[0.*B/s\])
replace: DOWNLOAD
```

This tells pytest to scan the outputs of all cells and replace the matching string with the word *DOWNLOAD*. This is called *sanitization*.

Sanitization

Exactly matching cell outputs between runs is a high bar for testing and tends to give false errors – otherwise correct cells may give different outputs each time they are run (e.g. cells with code that outputs memory locations). To alleviate this, before testing each cell pytest will match its output against the regular expressions from `sanitize_output.cfg`. This file contains replacements for strings that we know will vary. Pytest will replace the dynamic output with these constant strings, which won't change between runs and hence prevent spurious failures.

If your code correctly outputs variable strings each time its run, you should add a replacement rule to the `sanitize_output.cfg` file and try to make it as specific as possible.

On the hole2 notebook

The *hole2* notebook is special in that it requires installation of extra software to run, namely the hole2 program. If you test all the notebooks you may therefore run into errors if hole2 is not installed. These errors can be generally ignored unless you do specifically want to test the hole2 notebook. Of course, you should take note of other errors that occur if hole2 is installed! To run the hole2 notebook you'll have to download hole2, compile it, and make sure your system can find the hole2 executable. In UNIX-based systems this implies adding its path to the $PATH environmental variable like this:

```
export PATH=$PATH:"<PATH_TO_HOLE2>/exe"
```

Adding changes to the UserGuide

As with the code, *commit and push* your code to GitHub. Then *create a pull request*. The only test run for the User Guide is: that your file compile into HTML documents without errors. As usual, a developer will review your PR and merge the code into the User Guide when it looks good.

It is often difficult to review Jupyter notebooks on GitHub, especially if you embed widgets and images. One way to make it easier on the developers who review your changes is to build the changes on your forked repository and link the relevant sections in your pull request. To do this, create a gh-pages branch and merge your new branch into it.

```
# the first time
git checkout -b gh-pages
git merge origin/my-new-branch
```

Fix any merge conflicts that arise. Then edit UserGuide/doc/source/conf.py and change the URL of the site, which is set to site_url = "https://www.mdanalysis.org/UserGuide". Change it to your personal site, e.g.

```
site_url = "https://www.my_user_name.github.io/UserGuide"
```

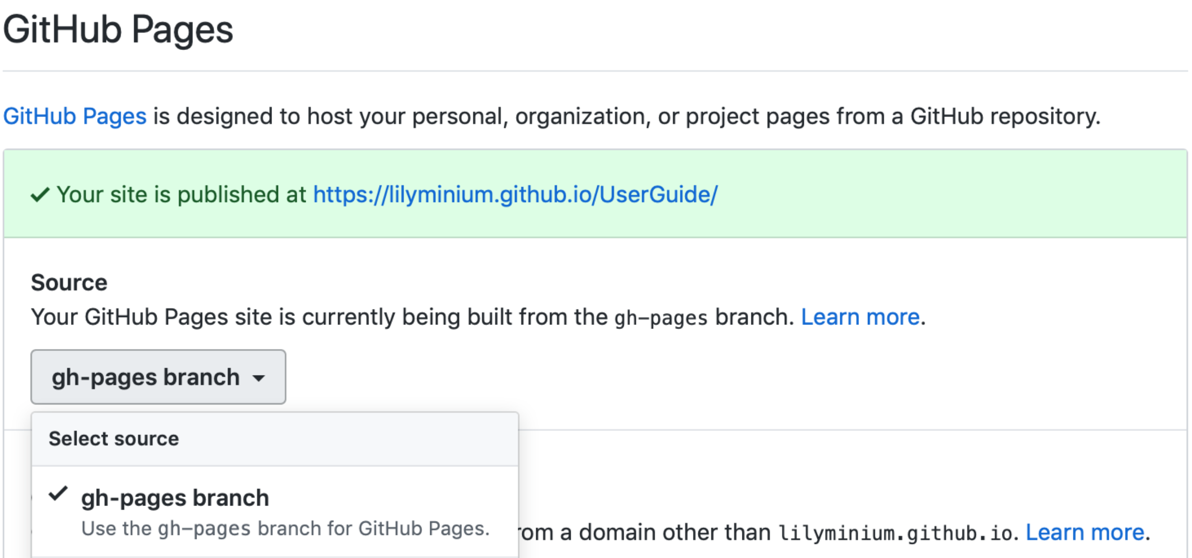
Now you can build your pages with the make github macro in the UserGuide/doc/ directory, which builds the files and copies them to the top level of your directory.

```
make github
```

You should be able to open one of these new HTML files (e.g. UserGuide/index.html) in a browser and navigate your new documentation. Check that your changes look right. If they are, push to your *gh-pages* branch from the UserGuide/ directory.

```
git add .
git commit -m 'built my-new-branch'
git push -f origin gh-pages
```

On GitHub, navigate to your fork of the repository and go to **Settings**. In the **GitHub Pages** section, select the "gh-pages branch" from the **Source** dropdown. Check that your website is published at the given URL.



GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

✓ Your site is published at https://lilyminium.github.io/UserGuide/

Source

Your GitHub Pages site is currently being built from the gh-pages branch. Learn more.

gh-pages branch ▾

Select source

✓ gh-pages branch

Use the gh-pages branch for GitHub Pages.

om a domain other than lilyminium.github.io. Learn more.

For each time you add changes to another branch later, just merge into gh-pages and rebuild.

```
git checkout gh-pages
git merge origin/my_branch
cd doc/
make github
```

Automatically building documentation

Constantly rebuilding documentation can become tedious when you have many changes to make. Use sphinx-autobuild to rebuild documentation every time you make changes to any document, including Jupyter notebooks. Install `sphinx-autobuild`:

```
pip install sphinx-autobuild
```

Then, run the following command in the `doc/` directory:

```
python -m sphinx_autobuild source build
```

This will start a local webserver at http://localhost:8000/, which will refresh every time you save changes to a file in the documentation. This is helpful for both the user guide (first navigate to `UserGuide/doc`) and the main repository documentation (navigate to `package/doc/sphinx`).

Using pre-commit hooks

Manually editing files can often lead to small inconsistencies: a whitespace here, a missing carriage return there. A tool called pre-commit can be used to automatically fix these problems, before a git commit is made. To enable the pre-commit hooks, run the following:

```
pre-commit install
```

To perform the pre-commit checks on all the files, run the following:

```
pre-commit run --all-files
```

To remove the pre-commit hooks from your .git directory, run the following:

```
pre-commit uninstall
```

2.1.38 Preparing a release

Rules for release branches:

- Branch from `develop`
- Naming convention: `package-*` where `*` is a version number

Release policy and release numbering

We use a **MAJOR.MINOR.PATCH** scheme to label releases. We adhere to the idea of semantic versioning (semantic versioning was introduced with release 0.9, see Issue 200): Given a version number **MAJOR.MINOR.PATCH**, we increment the:

- **MAJOR** version when we make **incompatible API changes**,
- **MINOR** version when we **add functionality** in a backwards-compatible manner, and
- **PATCH** version when we make backwards-compatible **bug fixes**.

However, as long as the **MAJOR** number is **0** (i.e. the API has not stabilized), even **MINOR** increases *may* introduce incompatible API changes. As soon as we have a 1.0.0 release, the public API can only be changed in a backward-incompatible manner with an increase in MAJOR version.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

The CHANGELOG lists important changes for each release.

MAJOR, *MINOR* and *PATCH* numbers are integers that increase monotonically.

The **release number** is set in setup.py *and* in `MDAnalysis.__version__` (MDAnalysis/version.py), e.g.

```
RELEASE = '0.7.5'
```

While the code is in development (i.e. whenever we are not preparing a release!) the release number gets the suffix **-dev0**, e.g.

```
RELEASE = '0.7.6-dev0'
```

so that people using the *develop branch* from the source repository can immediately see that it is not a final release. For example, “0.7.6-dev0” is the state *before* the 0.7.6 release.

Typical workflow for preparing a release

Summary of tasks

- Declare a *feature freeze* on *develop* via discord and/or GitHub Discussions (Announcement)
- Finalize the `CHANGELOG` file with the date of release
- Increment the version across `MDAnalysis` and `MDAnalysisTests` (4 places)
- *Merge* changes into *develop*

- Create a tag from *develop* named *release-<version_number>*
- Create a *package-<version_number>* branch from *develop*
- Add rebuilt *C / C++* files to *package-<version_number>*
- Create a tag from *package-<version_number>*
- Check automated testing of source distribution and wheel generation and upload
- Create a new release from newly created tag
- Check that deployment actions have adequately pushed dist and wheels to PyPi
- Manually upload Cirrus CI wheels (temporary)
- Update *conda-forge* packages
- Create a blog post outlining the release
- Increment develop branch files ready for the next version
- Clean up old dev docs builds

Getting the develop branch ready for a release

1. Declare feature freeze on `develop` via *discord* and GitHub Discussions (Announcement)
2. Create a pre-release feature branch from `develop`
3. Finalise the `CHANGELOG` with the release number and date. Ensure that the `CHANGELOG` summarizes important changes and includes all authors that contributed to this release.
4. Make sure the version number matches the release version. The following files need to be updated: `package/MDAnalysis/version.py`, `package/setup.py`, `testsuite/MDAnalysisTests/__init__.py`, and `testsuite/setup.py`.
5. Create a pull request against `develop` from this branch.

Packaging the release

1. Create a new tag from `develop` named `release-<version_number>` where `<version_number>` is the release version number (this tag contains a snapshot of the Python source files as they were when the release was created):

   ```
   git tag -m "release 0.7.5 of MDAnalysis and MDAnalysisTests" release-0.7.5
   git push --tags origin
   ```

2. Create a `package-<version_number>` branch from `develop`. This branch is automatically protected, so you will also need to create a separate branch to create commits via PR against `package-<version_number>`.
3. Generate new C/C++ files and commit them to the `package-<version_number>` branch via PR. We recommend generate the C/C++ files by building a *source distribution* tarball so you can check at the same time that there are no issues with the distribution creation process:

   ```
   # MDAnalysis
   cd package/
   pipx run build --sdist
   ```

4. Once committed, create a new tag based on `package-<version_number>` (this tag will contain a record of all the files as they were deployed to users for that version):

```
git tag -m "package 0.7.5 of MDAnalysis and MDAnalysisTests" package-0.7.5
git push --tags origin
```

5. Upon creation of the new `package-*` tag, the deploy github action workflow will be automatically triggered to create source/wheels, upload them to testpypi, re-download them and run tests.
6. If all the tests come back green, you are good to go for a full release.
    1. If tests fail you will need to work out the cause of the failure.
        1. A temporary github actions failure

            Re-run the action and wait for the tests to complete
        2. An issue with the source code.
            1. Delete the current `package-*` branch, and the newly created tags
            2. Add the new changes to `develop` and restart the release process.
            3. If the code had successfully uploaded to testpypi and failed later, you will need to create a test `package-*` tag which contains a different release number of in the source code (bumpy by a minor release or add a `-beta` modifier). Note: if the code had not successfully uploaded you can just continue the release process as normal.
            4. If CI comes back green then delete the test tag, and create a normal `package-*` tag with the correct version number.
            5. The github action will fail, but this is ok since we tested it with the test tag above.

Completing the release

If everything works, you can now complete the release by:

1. Creating a release on GitHub based on the newly created `package-<version_number>` tag.
2. Make sure you include relevant release notes, including any known issues and highlights for the release.
3. Once published, the deploy github action will be triggered which will upload the source distributions and wheels to PyPI.
    1. If the deploy github action fails and no files have been uploaded, then restart the action.
    2. If the action fails and some files have been uploaded, then you will not be able to re-upload to PyPI. At this point you will need to yank the release from PyPI and create a new minor version and re-deploy it.

Manually upload Cirrus CI wheels (temporary)

Unfortunately the deployment of Cirrus CI generated wheels (for *osx-arm64* and *linux-aarch64*) does not get properly triggered by a release. However, they are properly uploaded to TestPyPi

1. Go to the recently updated TestPyPi release and download all the *.whl* files which have the tags *arm64* and *aarch64*.
2. From a local directory upload these wheels using `twine`.

```
twine upload -r pypi *.whl --verbose
```

Update *conda-forge* packages

On push to PyPI, the conda-forge bot should automatically pick up the presense of a new version and create a pull request on the MDAnalysis feedstock and the MDAnalysisTests feedstock. You will need to merge the MDAnalysis feedstock followed by the MDAnalysisTests feedstock in order for the new package to appear on conda-forge.

To do this you will need to:

1. Update the `meta.yaml` files as necessary, especially bumping up the python and dependency minimum versions as necessary.
2. If NumPy pins differ from those conda-forge uses, you will need to update the `conda_build_config.yaml` accordingly.
3. Ask the conda-forge bot to re-render, check that CI returns green, approve and merge the pull request.

Create a release of the UserGuide

For now, the UserGuide is released at the same time as the core library. If it's failing please fix *before* you do the tag / release. Here is how to update the snapshots

1. Update the version of MDA used by the UserGuide to the release version.
2. Re-generate the Syrupy test snapshots, and commit those a to git and confirm the build passes.
3. Make a Pull Request with a re-generated `releases.md` which contains a copy of the GitHub release notes. This can be generated by doing:

   ```
   cd doc/source/scripts
   python gen_release_notes.py
   ```

4. Create a new release tag and upload them for the UserGuide repository.

   ```
   git tag -m 'release 2.6.1 of the MDAnalysis UserGuide' release-2.6.1
   git push --tags origin
   ```

5. This will automatically trigger a Github Action to build a new set of docs for that release and upload them. Due to the large size of the `gh-pages` branch on the UserGuide, this can be both slow and flaky, make sure to keep an eye out for any potential failures.

Create a blog post outlining the release

Create a blog post outlining the release notes and publicize it on GitHub Discussions / discord / twitter/ etc…!

Increment develop branch files ready for the next version

Once the release is completed you can go ahead and update the develop branch so that it is ready for the next round of development.

1. Update the 4 version file locations with the -dev0 appended version of the next release.
2. Update the CHANGELOG with a new entry for the next release.
3. Once these changes are merged into the develop branch, message the developers on discord and GitHub Discussions letting them know that the feature freeze is over.

Clean up old developer builds of the documentation

Whilst new docs are automatically deployed on a release, old developer builds (appended with -dev) are not automatically cleaned up. To avoid causing large amounts of files being uploaded to GitHub Pages, we need to delete these old developer builds manually. To do this switch to the gh-pages branch, delete these old files, and push the change directly. You should do this for both the core library and the UserGuide.

While this is still a manual procedure, you should also edit *versions.json* to remove the old dev links.

2.1.39 Module imports in MDAnalysis

We are striving to keep module dependencies small and lightweight (i.e., easily installable with pip).

General rules for importing

- Imports should all happen at the start of a module (not inside classes or functions).
- Modules must be imported in the following order:
  - future (from __future__ import absolute_import, print_function, division)
  - Compatibility imports (e.g. six)
  - global imports (installed packages)
  - local imports (MDAnalysis modules)
- use **absolute imports** in the library (i.e. relative imports must be explicitly indicated)

For example:

```
from __future__ import absolute_import
from six.moves import range

import numpy as np

import .core
import ..units
```

Module imports in `MDAnalysis.analysis`

1. In `MDAnalysis.analysis`, all imports must be at the top level (as in the General Rule) — see Issue 666 for more information.
2. *Optional modules* can be imported
3. No analysis module is imported automatically at the `MDAnalysis.analysis` level to avoid breaking the installation when optional dependencies are missing.

Module imports in the test suite

- Use the module import order in *General rules for importing*, but import `MDAnalysis` modules before `MDAnalysisTests` imports
- Do not use *relative imports* (e.g. `import .datafiles`) in the test suite. This breaks running the tests from inside the test directory (see Issue 189 for more information)
- Never import the `MDAnalysis` module from the `__init__.py` of `MDAnalysisTests` or from any of its plugins (it's ok to import from test files). Importing `MDAnalysis` from the test setup code will cause severe coverage underestimation.

Module dependencies in the code

List of core module dependencies

Any module from the standard library can be used, as well as the following nonstandard libraries:

- `six`
- `numpy`
- `biopython`
- `gridDataFormats`
- `mmtf-python`
- `scipy`
- `matplotlib`

because these packages are always installed.

If you must depend on a new external package, first discuss its use on GitHub Discussions (Development) or as part of the issue/pull request.

Modules in the "core"

The core of MDAnalysis contains all packages that are not in `MDAnalysis.analysis` or `MDAnalysis.visualization`. Only packages in the *List of core module dependencies* can be imported.

Optional modules in `MDAnalysis.analysis` and `MDAnalysis.visualization`

Modules under `MDAnalysis.analysis` are considered independent from the core package. Each analysis module can have its own set of dependencies. We strive to keep them small, but module authors are, in principle, free to import what they need. When analysis modules import packages outside of *List of core module dependencies*, the dependencies are considered **optional** (and should be listed in `setup.py` under *analysis*). (See also Issue 1159 for more discussion.)

A user who does not have a specific optional package installed must still be able to import everything else in MDAnalysis. An analysis module *may* simply raise an `ImportError` if a package is missing. However, it is recommended that the module should print and log an *error message* notifying the user that a specific additional package needs to be installed to use it.

If a large portion of the code in the module does not depend on a specific optional module then you should:

- guard the import at the top level with a `try/except`
- print and log a *warning*
- only raise an `ImportError` in the specific function or method that would depend on the missing module.

2.1.40 Tests in MDAnalysis

Note: Parts of this page came from the Contributing to pandas guide.

Whenever you add new code, you should create an appropriate test case that checks that your code is working as it should. This is very important because:

1. Firstly, it ensures that your code works as expected, i.e.
   - it succeeds in your test cases *and*
   - it fails predictably
2. More importantly, in the future we can always test that it is still working correctly. Unit tests are a crucial component of proper software engineering (see e.g. Software Carpentry on Testing) and a large (and growing) test suite is one of the strengths of MDAnalysis.

Adding tests is one of the most common requests after code is pushed to MDAnalysis. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue. We strive for 90% our code to be covered by tests.

We strongly encourage contributors to embrace test-driven development. This development process "relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test." So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Like many packages, MDAnalysis uses pytest and some of the numpy.testing framework.

Running the test suite

It is recommended that you run the tests from the `mdanalysis/testsuite/MDAnalysisTests/` directory.

```
cd testsuite/MDAnalysisTests
pytest  --disable-pytest-warnings
```

All tests should pass: no **FAIL** or **ERROR** cases should be triggered; *SKIPPED* or *XFAIL* are ok. For anything that fails or gives an error, ask on GitHub Discussions or raise an issue on the Issue Tracker.

We use the `--disable-pytest-warnings` when the whole testsuite is running, as pytest raises a lot of false positives when we warn users about missing topology attributes. When running single tests or only single modules, consider running the tests *with* warnings enabled (i.e. without `--disable-pytest-warnings`). This allows you to see if you trigger any un-caught deprecation warnings or other warnings in libraries we use.

To run specific tests just specify the path to the test file:

```
pytest testsuite/MDAnalysisTests/analysis/test_align.py
```

Specific test classes inside test files, and even specific test methods, can also be specified:

```
# Test the entire TestContactMatrix class
pytest testsuite/MDAnalysisTests/analysis/test_analysis.py::TestContactMatrix

# Test only test_sparse in the TestContactMatrix class
pytest testsuite/MDAnalysisTests/analysis/test_analysis.py::TestContactMatrix::test_
↪sparse
```

This is very useful when you add a new test and want to check if it passes. However, before you push your code to GitHub, make sure that your test case runs and that *all other test cases are still passing*.

Testing in parallel

Running the tests serially can take some time, depending on the performance of your computer. You can speed this up by using the plugin pytest-xdist to run tests in parallel by specifying the `--numprocesses` option:

```
pip install pytest-xdist
pytest --disable-pytest-warnings --numprocesses 4
```

You can try increasing the number of processes to speed up the test run. The number of processes you can use depends on your machine.

Test coverage

The tool pytest-cov can be used to generate the coverage report locally:

```
pip install pytest-cov
pytest --cov=MDAnalysis
```

Note: You can use the `--numprocesses` flag to run tests in parallel with the above command too. This will print the coverage statistic for every module in MDAnalysis at the end of a run. To get detailed line by line statistics you can add the `--cov-report=html` flag. This will create a `htmlcov` folder (in the directory you run the command from) and there will be an `index.html` file in this folder. Open this file in your browser and you will be able to see overall statistics and detailed line coverage for each file.

Continuous Integration tools

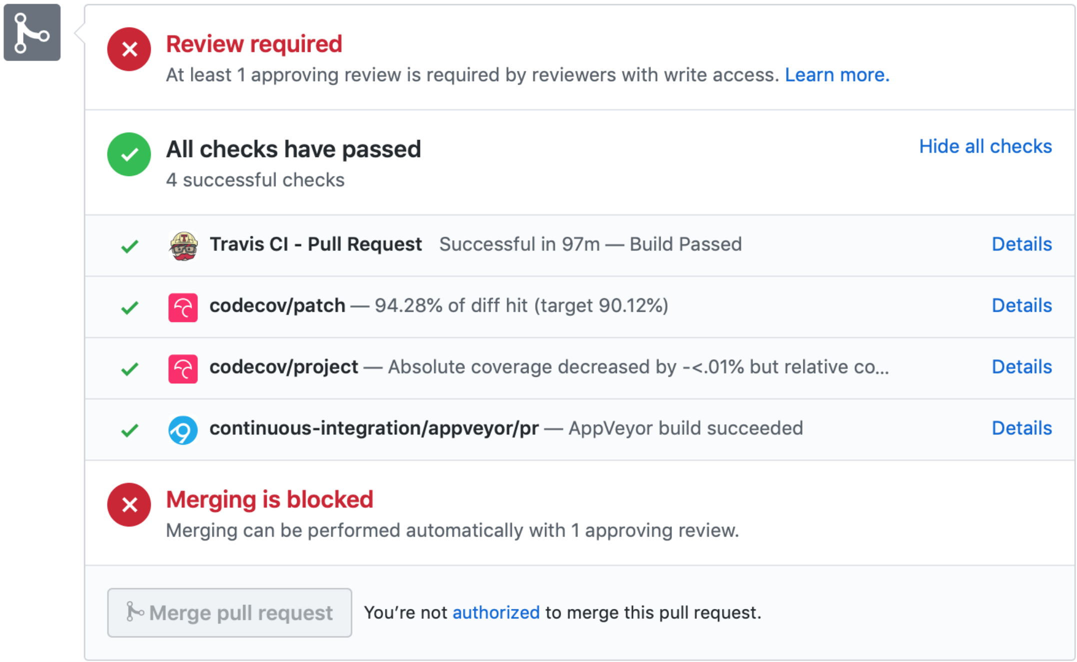
When you submit your pull request, several continuous integration tools will run a suite of tests. These should all pass before your code can be merged into MDAnalysis. You can check tests locally by *running the test suite*.

If your pull request fails immediately with an appveyor error, it is likely that you have merge conflicts with the latest code in the `develop` branch. *Rebase your code* and update your branch by pushing your changes.

If you get an error with travis, it is likely that you've failed a particular test. You should update your code and push again.

If you get *Codecov* errors, this means that your changes have not been adequately tested. Add new tests that address the "missed" lines, and push again.

Ideally, you want all tests to pass. This will look like:



✕ **Review required**
At least 1 approving review is required by reviewers with write access. Learn more.

✓ **All checks have passed** — Hide all checks
4 successful checks

✓ **Travis CI - Pull Request** Successful in 97m — Build Passed — Details

✓ **codecov/patch** — 94.28% of diff hit (target 90.12%) — Details

✓ **codecov/project** — Absolute coverage decreased by -<.01% but relative co... — Details

✓ **continuous-integration/appveyor/pr** — AppVeyor build succeeded — Details

✕ **Merging is blocked**
Merging can be performed automatically with 1 approving review.

You're not authorized to merge this pull request.

GitHub Actions

Configured in *.github/* directory. Uses YAML syntax to define both workflows and actions. See GitHub Actions documentation for more information.

Azure

Configured in *azure-pipelines.yml* file. Uses YAML syntax to define Azure Pipelines tasks. See Azure Pipelines documentation for more information.

Cirrus CI

The file *.cirrus.star* tells the provider what to do, it uses the Starlark syntax in YAML to define tasks. See Cirrus CI documentation for more information.

The actual files defining the workflows are in:

- maintainer/ci/cirrus-ci.yml
- maintainer/ci/cirrus-deploy.yml

Codecov

Code coverage measures how many lines, and which lines, of code are executed by a test suite. Codecov is a service that turns code coverage reports into a single visual report. Each line is described as one of three categories:

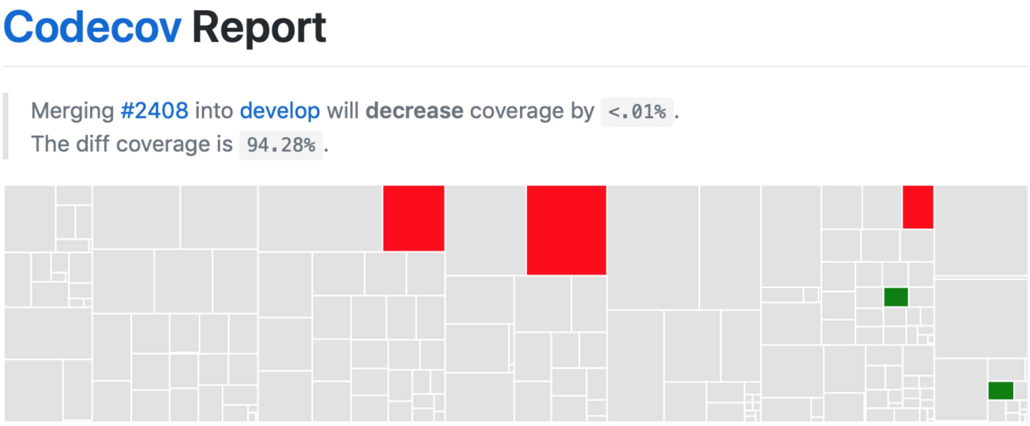
- a **hit** indicates that the source code was executed by the test suite.
- a **partial** indicates that the source code was not fully executed by the test suite; there are remaining branches that were not executed.
- a **miss** indicates that the source code was not executed by the test suite.

Coverage is the ratio of `hits / (sum of hit + partial + miss)`. See the Codecov documentation for more information.

MDAnalysis aims for 90% code coverage; your pull request will fail the Codecov check if the coverage falls below 85%. You can increase coverage by writing futher tests.

On your pull request, Codecov will leave a comment with three sections:

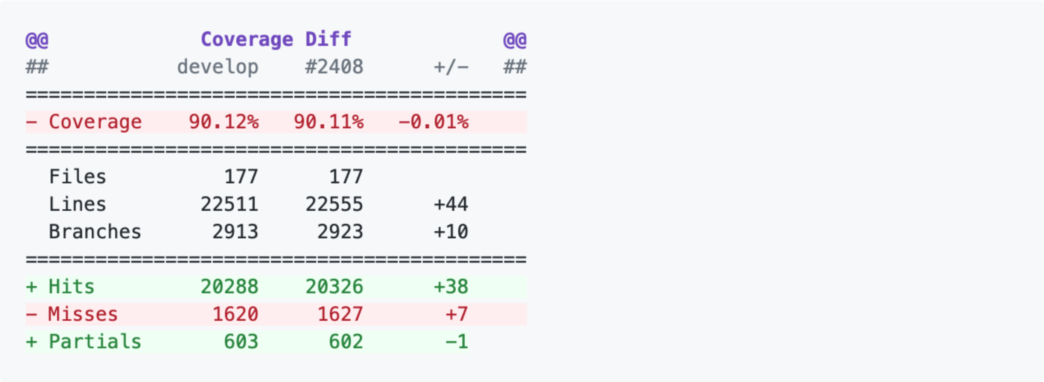
- a visual map of the areas with coverage changes



Codecov Report

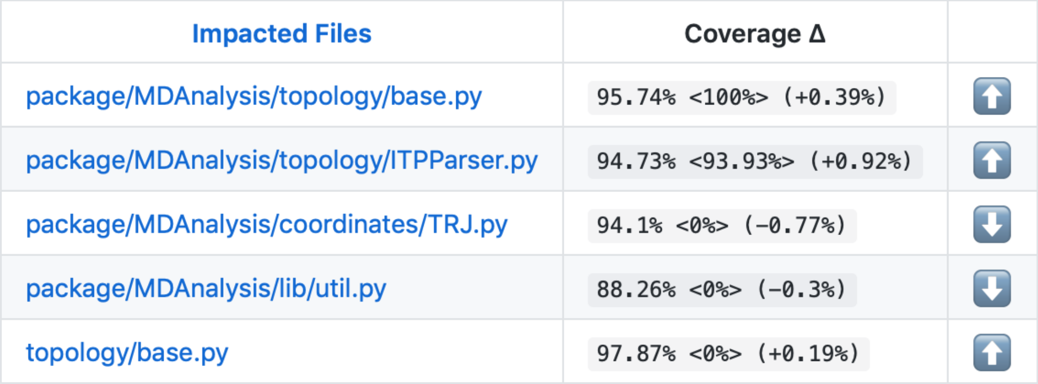
> Merging #2408 into develop will **decrease** coverage by `<.01%`.
> The diff coverage is `94.28%`.

- a summary of changes in coverage



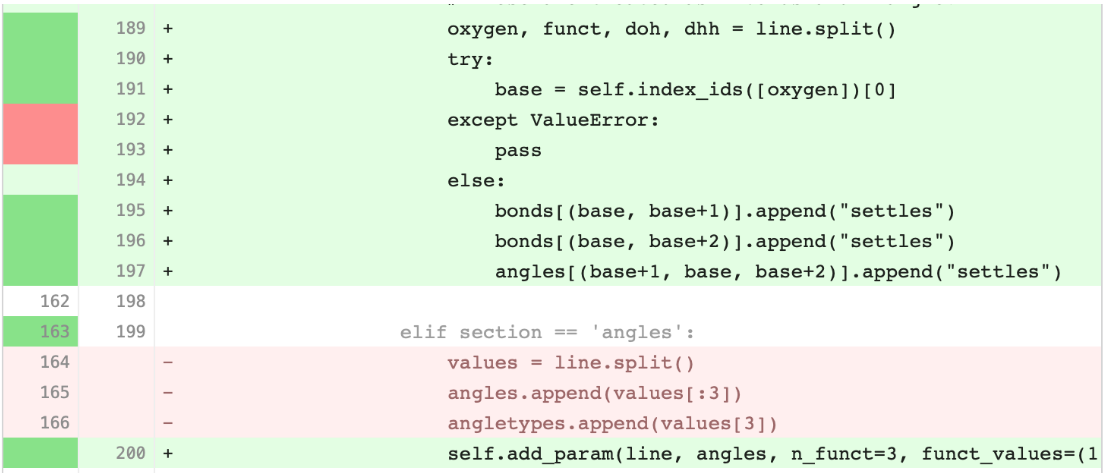
```
@@             Coverage Diff             @@
##           develop    #2408      +/-   ##
===========================================
- Coverage    90.12%   90.11%   -0.01%
===========================================
  Files          177      177
  Lines        22511    22555      +44
  Branches      2913     2923      +10
===========================================
+ Hits         20288    20326      +38
- Misses        1620     1627       +7
+ Partials       603      602       -1
```

- a list of files with changes



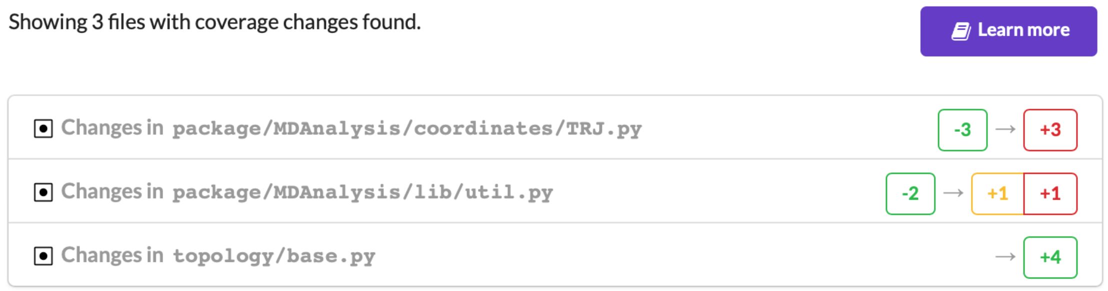
Impacted Files	Coverage Δ	
package/MDAnalysis/topology/base.py	95.74% <100%> (+0.39%)	⬆
package/MDAnalysis/topology/ITPParser.py	94.73% <93.93%> (+0.92%)	⬆
package/MDAnalysis/coordinates/TRJ.py	94.1% <0%> (-0.77%)	⬇
package/MDAnalysis/lib/util.py	88.26% <0%> (-0.3%)	⬇
topology/base.py	97.87% <0%> (+0.19%)	⬆

Clicking on one of those files will show the Codecov *Diff* view, highlighting the lines of code that have been missed by tests. In the image below, the column on the left hand side shows hits (green) and misses (red); the lighter colours highlighting the code show lines added (light green) or removed (light red).



```
+                   oxygen, funct, doh, dhh = line.split()
+                   try:
+                       base = self.index_ids([oxygen])[0]
+                   except ValueError:
+                       pass
+                   else:
+                       bonds[(base, base+1)].append("settles")
+                       bonds[(base, base+2)].append("settles")
+                       angles[(base+1, base, base+2)].append("settles")

                elif section == 'angles':
-                   values = line.split()
-                   angles.append(values[:3])
-                   angletypes.append(values[3])
+                   self.add_param(line, angles, n_funct=3, funct_values=(1
```

Changing to the *Coverage Changes* view highlights how your additions have changed the test coverage. See the documentation for viewing source code for more information.



Showing 3 files with coverage changes found.

▣ Changes in package/MDAnalysis/coordinates/TRJ.py -3 → +3

▣ Changes in package/MDAnalysis/lib/util.py -2 → +1 +1

▣ Changes in topology/base.py → +4

Writing new tests

Tests are organised by top-level module. Each file containing tests must start with `test_`. The tests themselves also have to follow the appropriate naming and organisational conventions.

Use classes to group tests if it makes sense (e.g., if the test class will be inherited by another test class and the code can be reused). We prefer subclassing over parametrizing classes (for examples, have a look at the `MDAnalysisTests/topology` module, where each class often tests a different file). For tests that are standalone, leave them as plain functions.

General conventions

Assertions

Use plain `assert` statements for comparing single values, e.g.

```python
def test_foo_is_length_3(foo):
    assert len(foo) == 3
```

To check equality up to a certain precision (e.g. floating point numbers and iterables of floats), use `assert_almost_equal()` from `numpy.testing`. Do not manually round off the value and use plain `assert` statements. Do not use `pytest.approx`.

```python
from numpy.testing import assert_almost_equal

def test_equal_coordinates():
    ref = mda.Universe(PSF, PDB_small)
    u = mda.Universe(PDB_small)
    assert_almost_equal(u.atoms.positions, ref.atoms.positions)
```

To test for exact equality (e.g. integers, booleans, strings), use `assert_equal()` from `numpy.testing`. As with `assert_almost_equal()`, this should be used for iterables of exact values as well. Do not iterate over and compare every single value.

```python
from numpy.testing import assert_equal

def test_equal_arrays(array1, array2):
    assert_equal(array1, array2)
```

Do not use `assert_array_equal` or `assert_array_almost_equal` from `numpy.testing` to compare array/array-like data structures. Instead, use `assert_equal()` or `assert_almost_equal()`. The former set of functions equate arrays and scalars, while the latter do not:

```
In [1]: from numpy.testing import assert_equal, assert_array_equal

In [2]: assert_array_equal([1], 1)

In [3]: assert_equal([1], 1)
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[3], line 1
----> 1 assert_equal([1], 1)

File ~/checkouts/readthedocs.org/user_builds/mdanalysisuserguide/conda/develop/lib/
↪python3.9/site-packages/numpy/testing/_private/utils.py:314, in assert_equal(actual,␣
↪desired, err_msg, verbose)
    312 # isscalar test to check cases such as [np.nan] != np.nan
    313 if isscalar(desired) != isscalar(actual):
--> 314     raise AssertionError(msg)
    316 try:
    317     isdesnat = isnat(desired)

AssertionError:
Items are not equal:
 ACTUAL: [1]
 DESIRED: 1
```

Do not use anything from `numpy.testing` that depends on `nose`, such as `assert_raises`.

Testing exceptions and warnings

Do not use `assert_raises` from `numpy.testing` or the `pytest.mark.raises` decorator to test for particular exceptions. Instead, use context managers:

```
def test_for_error():
    a = [1, 2, 3]
    with pytest.raises(IndexError):
        b = a[4]

def test_for_warning():
    with pytest.warns(DeprecationWarning):
        deprecated_function.run()
```

Failing tests

To mark an expected failure, use `pytest.mark.xfail()` decorator:

```
@pytest.mark.xfail
def tested_expected_failure():
    assert 1 == 2
```

To manually fail a test, make a call to `pytest.fail()`:

```
def test_open(self, tmpdir):
    outfile = str(tmpdir.join('lammps-writer-test.dcd'))
    try:
        with mda.coordinates.LAMMPS.DCDWriter(outfile, n_atoms=10):
            pass
    except Exception:
        pytest.fail()
```

Skipping tests

To skip tests based on a condition, use `pytest.mark.skipif(condition)` decorator:

```
import numpy as np
try:
    from numpy import shares_memory
except ImportError:
    shares_memory = False

@pytest.mark.skipif(shares_memory == False,
                    reason='old numpy lacked shares_memory')
def test_positions_share_memory(original_and_copy):
    # check that the memory in Timestep objects is unique
    original, copy = original_and_copy
    assert not np.shares_memory(original.ts.positions, copy.ts.positions)
```

To skip a test if a module is not available for importing, use `pytest.importorskip('module_name')`

```
def test_write_trajectory_netCDF4(self, universe, outfile):
    pytest.importorskip("netCDF4")
    return self._test_write_trajectory(universe, outfile)
```

Fixtures

Use fixtures as much as possible to reuse "resources" between test methods/functions. Pytest fixtures are functions that run before each test function that uses that fixture. A fixture is typically set up with the `pytest.fixture()` decorator, over a function that returns the object you need:

```
@pytest.fixture
def universe(self):
    return mda.Universe(self.ref_filename)
```

A function can use a fixture by including its name in its arguments:

```
def test_pdb_write(self, universe):
    universe.atoms.write('outfile.pdb')
```

The rule of thumb is to use the largest possible scope for the fixture to save time. A fixture declared with a class scope will run once per class; a fixture declared with a module scope will only run once per module. The default scope is `"function"`.

```
@pytest.fixture(scope='class')
def universe(self):
    return mda.Universe(self.ref_filename)
```

Testing the same function with different inputs

Use the `pytest.mark.parametrize()` decorator to test the same function for different inputs rather than looping. These can be stacked:

```
@pytest.mark.parametrize('pbc', (True, False))
@pytest.mark.parametrize('name, compound', (('molnums', 'molecules'),
                                            ('fragindices', 'fragments')))
# fragment is a fixture defined earlier
def test_center_of_mass_compounds_special(self, fragment,
                                          pbc, name, compound):
    ref = [a.center_of_mass() for a in fragment.groupby(name).values()]
    com = fragment.center_of_mass(pbc=pbc, compound=compound)
    assert_almost_equal(com, ref, decimal=5)
```

The code above runs `test_center_of_mass_compounds_special` 4 times with the following parameters:

- pbc = `True`, name = `'molnums'`, compound = `'molecules'`
- pbc = `True`, name = `'fragindices'`, compound = `'fragments'`
- pbc = `False`, name = `'molnums'`, compound = `'molecules'`
- pbc = `False`, name = `'fragindices'`, compound = `'fragments'`

Temporary files and directories

Do not use `os.chdir()` to change directories in tests, because it can break the tests in really weird ways (see Issue 556). To use a temporary directory as the working directory, use the `tmpdir.as_cwd()` context manager instead:

```
def test_write_no_args(self, u, tmpdir): # tmpdir is an in-built fixture
    with tmpdir.as_cwd():
        u.atoms.write()
```

To create a temporary file:

```
def outfile(tmpdir):
    temp_file = str(tmpdir.join('test.pdb'))
```

Module imports

Do not use relative imports in test files, as it means that tests can no longer be run from inside the test directory. Instead, use absolute imports.

```
from .datafiles import PDB  # this is relative and will break!
from MDAnalysisTests.datafiles import PDB  # use this instead
```

Tests for analysis and visualization modules

Tests for analysis classes and functions should at a minimum perform regression tests, i.e., run on input and compare to values generated when the code was added so that we know when the output changes in the future. (Even better are tests that test for absolute correctness of results, but regression tests are the minimum requirement.)

Any code in `MDAnalysis.analysis` that does not have substantial testing (at least 70% coverage) will be moved to a special `MDAnalysis.analysis.legacy` module by release 1.0.0. This legacy module will come with its own warning that this is essentially unmaintained functionality, that is still provided because there is no alternative. Legacy packages that receive sufficient upgrades in testing can come back to the normal `MDAnalysis.analysis` name space.

No consensus has emerged yet how to best test visualization code. At least minimal tests that run the code are typically requested.

Using test data files

If possible, re-use the existing data files in MDAnalysis for tests; this helps to keep the (separate) MDAnalysisTests package small. If new files are required (e.g. for a new coordinate Reader/Writer) then:

1. Use small files (e.g. trajectories with only a few frames and a small system).
2. Make sure that the data are *not confidential* (they will be available to everyone downloading MDAnalysis) and also be aware that by adding them to MDAnalysis *you license these files* under the GNU Public Licence v2 (or a compatible licence of your choice — otherwise we cannot include the files into MDAnalysis).
3. Add the files to the `testsuite/MDAnalysisTests/data` directory and appropriate file names and descriptions to `testsuite/MDAnalysisTests/datafiles.py`.
4. Make sure your new files are picked up by the pattern-matching in `testsuite/setup.py` (in the `package_data` dictionary).

2.1.41 References

MDAnalysis and the included algorithms are scientific software that are described in academic publications. **Please cite these papers when you use MDAnalysis in published work.**

It is possible to automatically generate a list of references for any program that uses MDAnalysis. This list (in common reference manager formats) contains the citations associated with the specific algorithms and libraries that were used in the program.

Citations using Duecredit

Citations can be automatically generated using duecredit, depending on the packages used. Duecredit is easy to install via pip. Simply type:

```
pip install duecredit
```

duecredit will remain an optional dependency, i.e. any code using MDAnalysis will work correctly even without duecredit installed.

A list of citations for yourscript.py can be obtained using simple commands.

```
cd /path/to/yourmodule
python -m duecredit yourscript.py
```

or set the environment variable DUECREDIT_ENABLE

```
DUECREDIT-ENABLE=yes python yourscript.py
```

Once the citations have been extracted (to a hidden file in the current directory), you can use the **duecredit** program to export them to different formats. For example, one can display them in BibTeX format, using:

```
duecredit summary --format=bibtex
```

Please cite your use of MDAnalysis and the packages and algorithms that it uses. Thanks!

BIBLIOGRAPHY

[ALB93] Andrea Amadei, Antonius B. M. Linssen, and Herman J. C. Berendsen. Essential dynamics of proteins. *Proteins: Structure, Function, and Bioinformatics*, 17(4):412–425, 1993. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/prot.340170408. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/prot.340170408 (visited on 2021-01-05), doi:https://doi.org/10.1002/prot.340170408.

[BKV00] M. Bansal, S. Kumar, and R. Velavan. HELANAL: a program to characterize helix geometry in proteins. *Journal of Biomolecular Structure & Dynamics*, 17(5):811–819, April 2000. 00175. doi:10.1080/07391102.2000.10506570.

[BDPW09] Oliver Beckstein, Elizabeth J. Denning, Juan R. Perilla, and Thomas B. Woolf. Zipping and Unzipping of Adenylate Kinase: Atomistic Insights into the Ensemble of OpenClosed Transitions. *Journal of Molecular Biology*, 394(1):160–176, November 2009. 00107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022283609011164 (visited on 2020-02-05), doi:10.1016/j.jmb.2009.09.009.

[BHE13] R. B. Best, G. Hummer, and W. A. Eaton. Native contacts determine protein folding mechanisms in atomistic simulations. *Proceedings of the National Academy of Sciences*, 110(44):17874–17879, October 2013. 00259. URL: http://www.pnas.org/cgi/doi/10.1073/pnas.1311599110 (visited on 2020-02-05), doi:10.1073/pnas.1311599110.

[CL06] Ronald R. Coifman and Stéphane Lafon. Diffusion maps. *Applied and Computational Harmonic Analysis*, 21(1):5–30, July 2006. 02271. doi:10.1016/j.acha.2006.04.006.

[dlPHHvdW08] J. de la Porte, B. M. Herbst, W. Hereman, and S. J. van der Walt. An introduction to diffusion maps. In *In The 19th Symposium of the Pattern Recognition Association of South Africa*. 2008. 00038.

[FPKD11] Andrew Ferguson, Athanassios Z. Panagiotopoulos, Ioannis G. Kevrekidis, and Pablo G. Debenedetti. Nonlinear dimensionality reduction in molecular simulation: The diffusion map approach. *Chemical Physics Letters*, 509(1-3):1–11, June 2011. 00085. doi:10.1016/j.cplett.2011.04.066.

[FKDD07] Joel Franklin, Patrice Koehl, Sebastian Doniach, and Marc Delarue. MinActionPath: maximum likelihood trajectory for large-scale structural transitions in a coarse-grained locally harmonic energy landscape. *Nucleic Acids Research*, 35(suppl_2):W477–W482, July 2007. 00083. URL: https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkm342 (visited on 2020-02-05), doi:10.1093/nar/gkm342.

[GLB+16] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. *Proceedings of the 15th Python in Science Conference*, pages 98–105, 2016. 00152. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html (visited on 2020-02-05), doi:10.25080/Majora-629e541a-00e.

[HKP+07] Benjamin A. Hall, Samantha L. Kaye, Andy Pang, Rafael Perera, and Philip C. Biggin. Characterization of Protein Conformational States by Normal-Mode Frequencies. *Journal of the American Chemical So-*

ciety, 129(37):11394–11401, September 2007. 00020. URL: https://doi.org/10.1021/ja071797y (visited on 2020-02-05), doi:10.1021/ja071797y.

[Hes02] Berk Hess. Convergence of sampling in protein simulations. *Physical Review E*, 65(3):031910, March 2002. 00348. URL: https://link.aps.org/doi/10.1103/PhysRevE.65.031910 (visited on 2020-03-07), doi:10.1103/PhysRevE.65.031910.

[JWLM78] Joël Janin, Shoshanna Wodak, Michael Levitt, and Bernard Maigret. Conformation of amino acid sidechains in proteins. *Journal of Molecular Biology*, 125(3):357 – 386, 1978. 00874. URL: http://www.sciencedirect.com/science/article/pii/0022283678904084, doi:10.1016/0022-2836(78)90408-4.

[Jol02] I. T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer-Verlag, New York, 2 edition, 2002. ISBN 978-0-387-95442-4. URL: http://www.springer.com/gp/book/9780387954424 (visited on 2021-01-05), doi:10.1007/b98835.

[LAT09] Pu Liu, Dimitris K. Agrafiotis, and Douglas L. Theobald. Fast determination of the optimal rotational matrix for macromolecular superpositions. *Journal of Computational Chemistry*, pages n/a–n/a, 2009. URL: http://doi.wiley.com/10.1002/jcc.21439 (visited on 2020-02-05), doi:10.1002/jcc.21439.

[LDA+03] Simon C. Lovell, Ian W. Davis, W. Bryan Arendall, Paul I. W. de Bakker, J. Michael Word, Michael G. Prisant, Jane S. Richardson, and David C. Richardson. Structure validation by C geometry: , and C deviation. *Proteins: Structure, Function, and Bioinformatics*, 50(3):437–450, January 2003. 03997. URL: http://doi.wiley.com/10.1002/prot.10286 (visited on 2020-02-06), doi:10.1002/prot.10286.

[MLS09] Gia G. Maisuradze, Adam Liwo, and Harold A. Scheraga. Principal component analysis for protein folding dynamics. *Journal of molecular biology*, 385(1):312–329, January 2009. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2652707/ (visited on 2020-12-27), doi:10.1016/j.jmb.2008.10.018.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry*, 32(10):2319–2327, July 2011. 00778. URL: http://doi.wiley.com/10.1002/jcc.21787 (visited on 2020-02-05), doi:10.1002/jcc.21787.

[NCR18] Hai Nguyen, David A Case, and Alexander S Rose. NGLview–interactive molecular graphics for Jupyter notebooks. *Bioinformatics*, 34(7):1241–1242, April 2018. 00024. URL: https://academic.oup.com/bioinformatics/article/34/7/1241/4721781 (visited on 2020-02-05), doi:10.1093/bioinformatics/btx789.

[RZMC11] Mary A. Rohrdanz, Wenwei Zheng, Mauro Maggioni, and Cecilia Clementi. Determination of reaction coordinates via locally scaled diffusion map. *The Journal of Chemical Physics*, 134(12):124116, March 2011. 00220. doi:10.1063/1.3569857.

[SB17] Sean Seyler and Oliver Beckstein. Molecular dynamics trajectory for benchmarking MDAnalysis. June 2017. 00002. URL: https://figshare.com/articles/Molecular_dynamics_trajectory_for_benchmarking_MDAnalysis/5108170 (visited on 2020-02-09), doi:10.6084/m9.figshare.5108170.v1.

[SKTB15] Sean L. Seyler, Avishek Kumar, M. F. Thorpe, and Oliver Beckstein. Path Similarity Analysis: A Method for Quantifying Macromolecular Pathways. *PLOS Computational Biology*, 11(10):e1004568, October 2015. URL: https://dx.plos.org/10.1371/journal.pcbi.1004568 (visited on 2020-02-05), doi:10.1371/journal.pcbi.1004568.

[SFPG+19] Max Linke Shujie Fan, Ioannis Paraskevakos, Richard J. Gowers, Michael Gecht, and Oliver Beckstein. PMDA - Parallel Molecular Dynamics Analysis. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, 134 – 142. 2019. doi:10.25080/Majora-7ddc1dd1-013.

[SJS14] Florian Sittel, Abhinav Jain, and Gerhard Stock. Principal component analysis of molecular dynamics: on the use of Cartesian vs. internal coordinates. *The Journal of Chemical Physics*, 141(1):014111, July 2014. doi:10.1063/1.4885338.

[SS18] Florian Sittel and Gerhard Stock. Perspective: Identification of collective variables and metastable states of protein dynamics. *The Journal of Chemical Physics*, 149(15):150901, October 2018. Publisher: American Institute of Physics. URL: http://aip.scitation.org/doi/10.1063/1.5049637 (visited on 2021-01-05), doi:10.1063/1.5049637.

[SGW93] O S Smart, J M Goodfellow, and B A Wallace. The pore dimensions of gramicidin A. *Biophysical Journal*, 65(6):2455–2460, December 1993. 00522. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1225986/ (visited on 2020-02-10), doi:10.1016/S0006-3495(93)81293-1.

[SNW+96] O. S. Smart, J. G. Neduvelil, X. Wang, B. A. Wallace, and M. S. Sansom. HOLE: a program for the analysis of the pore dimensions of ion channel structural models. *Journal of Molecular Graphics*, 14(6):354–360, 376, December 1996. 00935. doi:10.1016/s0263-7855(97)00009-x.

[SFSB14] Lukas S. Stelzl, Philip W. Fowler, Mark S. P. Sansom, and Oliver Beckstein. Flexible gates generate occluded intermediates in the transport cycle of LacY. *Journal of Molecular Biology*, 426(3):735–751, February 2014. 00000. URL: https://asu.pure.elsevier.com/en/publications/flexible-gates-generate-occluded-intermediates-in-the-transport-c (visited on 2020-02-10), doi:10.1016/j.jmb.2013.10.024.

[SM67] Hiromu Sugeta and Tatsuo Miyazawa. General method for calculating helical parameters of polymer chains from bond lengths, bond angles, and internal-rotation angles. *Biopolymers*, 5(7):673–679, 1967. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/bip.1967.360050708, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/bip.1967.360050708, doi:https://doi.org/10.1002/bip.1967.360050708.

[The05] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. *Acta Crystallographica Section A Foundations of Crystallography*, 61(4):478–480, July 2005. 00127. URL: http://scripts.iucr.org/cgi-bin/paper?S0108767305015266 (visited on 2020-02-05), doi:10.1107/S0108767305015266.

[TPB+15] Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, and Kresten Lindorff-Larsen. ENCORE: Software for Quantitative Ensemble Comparison. *PLOS Computational Biology*, 11(10):e1004415, October 2015. 00031. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004415 (visited on 2020-02-05), doi:10.1371/journal.pcbi.1004415.

[Wel62] B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, August 1962. URL: https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022 (visited on 2020-02-10), doi:10.1080/00401706.1962.10490022.

INDEX