Distance calculations in MDAnalysis

The lib.distances module

Particle positions are given as numpy arrays, so most work can be done using numpy (and numpy derived) libraries.

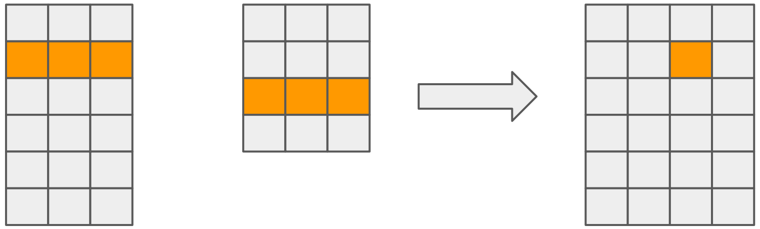
One import exception to this is distance calculations, where:

- periodic boundaries must be considered
- domain specific algorithms can be used

distance_array

To calculate all pairwise distances between two arrays of coordinates.

Takes 2 arrays of size (n, 3) & (m, 3) and returns array of shape (n, m).



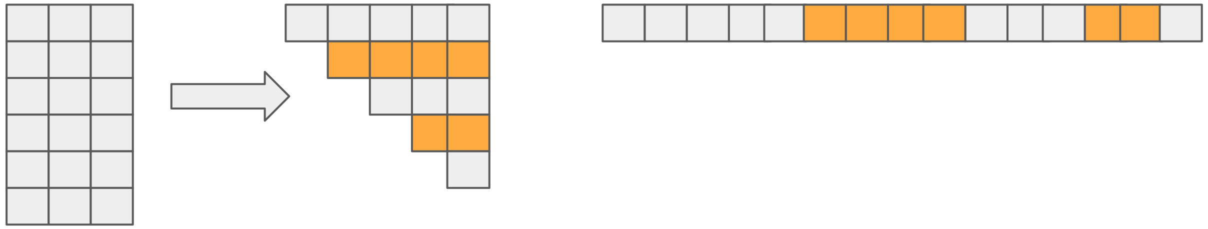
```
from MDAnalysis.lib import distances
import MDAnalysis as mda

u = mda.Universe(...)
ag1 = u.select_atoms(...)
ag2 = u.select_atoms(...)

reference = ag1.positions
configuration = ag2.positions

distances.distance_array(
      reference, configuration,
      box=None, result=None)
```

self_distance_array

For calculating distances between all combinations of coordinates.

Takes a single array of coordinates and calculates all pairwise distances (½ n(n-1) results).

```
from MDAnalysis.lib import distances

distances.self_distance_array(
      reference,
      box=None, result=None)
```

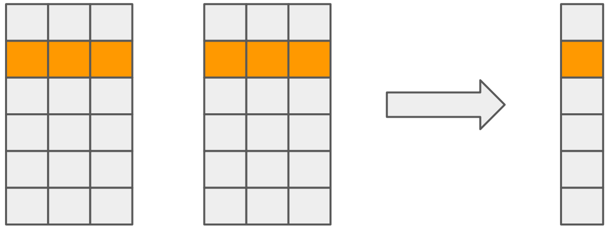


calc_bonds

For calculating a series of distances between points.

Takes two arrays of coordinates, of equal length, and returns the distances between coordinates in each row.

```
from MDAnalysis.lib import distances

distances.calc_bonds(
      coords1, coords2,
      box=None, result=None)
```

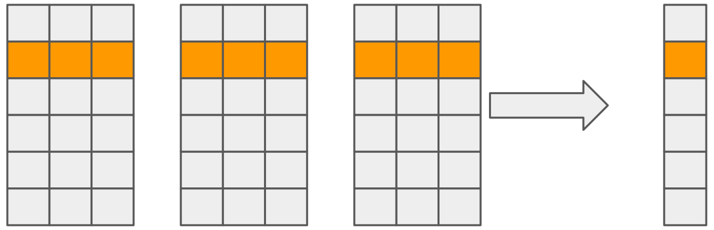


calc_angles & calc_dihedrals

For calculating either the angle or dihedral angle between 3 or 4 arrays of coordinates.

Input similar to calc_bonds, takes 3 or 4 arrays of identical length coordinates.

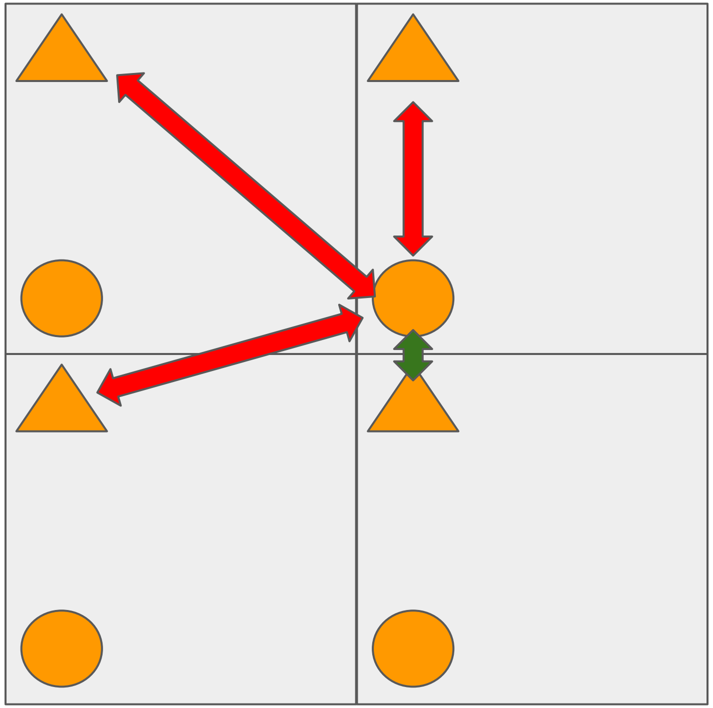
```
from MDAnalysis.lib import distances

distances.calc_angles(
      coords1, coords2, coords3,
      box=None, result=None)
```



Minimum image convention

For molecular simulation coordinates it is important that we consider periodic boundary conditions and that the smallest (correct) distance is considered.

This can be achieved by supplying the box information as "`box=ag.dimensions`" to any distance function.

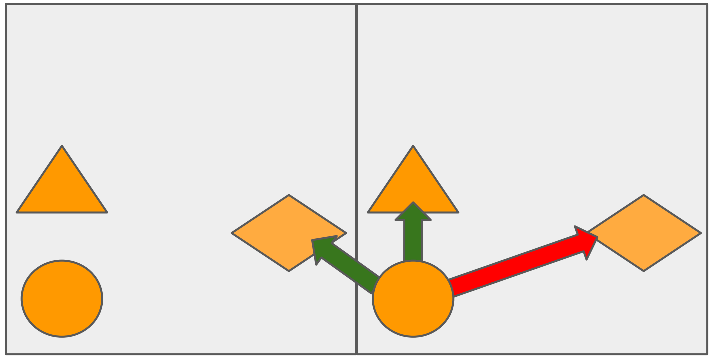


Minimum image convention for angles

For angle calculations, minimum image convention is still important!

The vectors between coordinates must obey minimum image convention.

Again, pass "`box=ag.dimensions`" to the function and this will be taken care of for you.



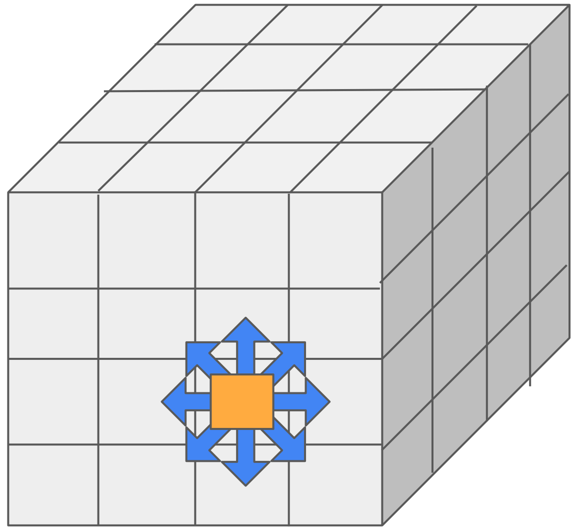
Capped distances

The problem: Quadratic scaling!

Calculating all pairwise distances between two sets of coordinates will scale badly for larger inputs.

Interactions between particles are quite short ranged and we're often only interested in relatively short distances.

We can therefore perform a search for all pairwise distances which are below a given limit - capped distance search.



capped_distance & capped_self_distance

These work similarly to distance_array and self_distance array but only find distances up to a limit.

Instead of returning a full n x m matrix of distances, it returns:

- an array of indices
- an array of distances

The indices refer to the row and column it would be in the full matrix, the distance is the value for that cell.

If a given row and column is not in the indices, the distance between these points is larger than the max cutoff.

```
from MDAnalysis.lib import distances

distances.capped_distance(
    reference, configuration, max_cutoff,
    box=None)

distances.self_capped_distance(
    reference, max_cutoff,
    box=None)
```

Capped distance example

```
idx, dists = capped_distance(acceptors.positions,
                             hydrogens.positions,
                             max_cutoff=3.0,
                             box=u.dimensions)
idx[:5], dists[:5]
(array([[    0, 13456],
        [    0, 19217],
        [    0, 19216],
        [    0,  8949],
        [    0, 14295]]),
 array([2.94881834, 2.91110877, 1.58488013, 2.24118461,
1.79654237]))
```

Distance calculation recap

Calculating pairwise distances:

- `calc_bonds`
- `distance_array`
- `self_distance_array`

Faster, sparse pairwise distances:

- `capped_distance`
- `self_capped_distance`

Calculating angles:

- `calc_angles`
- `calc_dihedrals`

Distance questions?

The Analysis framework

The unified analysis framework

MDAnalysis contains many prepackaged “turn-key” analysis methods.

These are defined as classes.

The analysis is then performed using the “`.run()`” method, which also controls which frames are considered.

Results are then made available from the “`.results`” attribute of the class, which provides dictionary-like access to the data produced.

```
import MDAnalysis as mda
from MDAnalysis.analysis.hydrogenbonds import HydrogenBondAnalysis

u = mda.Universe(...)

hbonds = HydrogenBondAnalysis(u,
                              acceptors_sel='resname ASP and element O',
                              hydrogens_sel='resname SOL and element H')

hbonds.run(start=5, stop=10)

hbonds.results
```

MDAnalysis.analysis

There is currently a lot of premade analysis modules ready to go, check the documentation for what exists!

There is a new project which will make these available from the command line, check out:

https://github.com/MDAnalysis/mdacli

There is also a lot of other projects that aren’t inside our package, check out:

https://www.mdanalysis.org/pages/used-by/

`MDAnalysis.analysis.hydrogenbonds`

- hydrogen bonds, hydrogen bond lifetimes and more

`MDAnalysis.analysis.psa`

- path similarity analysis

`MDanalysis.analysis.pca`

- principal component analysis

`MDAnalysis.analysis.rdf`

- radial distribution functions

And lots more!

Divide and conquer.

Analysis within a statistical mechanics framework allows us to consider snapshots of our results independently, and then use the average properties.

We can therefore define how a single frame is analysed, and how this is averaged, then let users choose what in their system is analysed and which frames are considered.

Writing an Analysis class

Internally, analysis classes in MDAnalysis must define three methods:

- `_prepare`
  - sets up all necessary data structures for collecting data
- `_single_frame`
  - performs analysis on a single frame
- `_conclude`
  - performs the necessary averaging of results

The inherited "`run`" method then automatically calls these functions to perform the analysis.

```
from MDAnalysis.analysis import base


class MyAnalysis(base.AnalysisBase):
    def __init__(self):
        pass

    def _prepare(self):
        pass

    def _single_frame(self):
        pass

    def _conclude(self):
        pass
```

Analysis questions?