MDorado

Documentation of Version 0.3.0

Jan Neumann,
February 1, 2022, Rostock

Contents

1 Importing Testfiles

A number of files for unit testing and examples are shipped with the installation of MDorado. Their absolute paths can be obtained via a python import statement using the `mdorado.data.datafilenames` module.

1.1 File references

`water_topology`	Gromacs .tpr file (`water.tpr`) of a small SPC water simulation (125 molecules, 300 K, 10^5 Pa) used as topology reference by MDAnalysis. Is used in a number of unit tests and examples throughout all functions.
`water_trajectory`	Gromacs .xtc file (`water.xtc`) of a small SPC water simulation (125 molecules, 300 K, 10^5 Pa) used as trajectory reference by MDAnalysis. Is used in a number of unit tests and examples throughout all functions.
`test_gofr_ss`	Textfile (`gofr_ss.dat`) containing the reference output of the `mdorado.gofr.Gofr` calculation using the `"site-site"` mode. Is used in the unit test of the `mdorado.gofr` module.
`test_gofr_cc`	Textfile (`gofr_cc.dat`) containing the reference output of the `mdorado.gofr.Gofr` calculation using the `"cms-cms"` mode. Is used in the unit test of the `mdorado.gofr` module.
`test_gofr_sc`	Textfile (`gofr_sc.dat`) containing the reference output of the `mdorado.gofr.Gofr` calculation using the `"site-cms"` mode. Is used in the unit test of the `mdorado.gofr` module.
`test_hbanalyze`	Textfile (`hb_analyze.dat`) containing the reference output of the `mdorado.hb_analyze.hb_analyze` function. Is used in the unit test of the `mdorado.hb_analyze` module.

`test_lifetime`	Textfile (`lifetime_test.dat`) containing the reference output of the `mdorado.lifetime.calc_lifetime` function. Is used in the unit test of the `mdorado.lifetime` module.
`test_unwrap`	File (`msd_molpos.npy`) containing the reference output of the `mdorado.msd.unwrap` function Is used in the unit test of the `mdorado.msd` module.
`test_msd`	Textfile (`msd_h.dat`) containing the reference output of the `mdorado.msd.msd` function. Is used in the unit test of the `mdorado.msd` module.
`test_getvecarray`	File (`vecarray42.npy`) containing the reference output of the `mdorado.vectors.get_vecarray` function. Is used in the unit test of the `mdorado.vectors` module.
`test_getnormvecarray`	File (`normal_vecarray42.npy`) containing the reference output of the `mdorado.vectors.get_normal_vecarray` function. Is used in the unit test of the `mdorado.vectors` module.
`test_getvecmatrix`	File (`vecmatrix4247.npy`) containing the reference output of the `mdorado.vectors.get_vectormatrix` function. Is used in the unit test of the `mdorado.vectors` module.
`test_correlvec`	Textfile (`anisolg2.dat`) containing the reference output of the `mdorado.veccor.correlvec` function. Is used in the unit test of the `mdorado.veccor` module.
`test_vectors`	File (`vectors_isocorrel.npy`) containing input for the unittest of the `mdorado.veccor.isocorrelvec` function.
`test_isocorrelvec`	Textfile (`slow_isolg2.dat`) containing the reference output of the `mdorado.veccor.isocorrelvec` function. Is used in the unit test of the `mdorado.veccor` module.
`test_isocorrelveclg1`	Textfile (`fast_isolg1.dat`) containing the reference output of the `mdorado.veccor.isocorrelveclg1` function. Is used in the unit test of the `mdorado.veccor` module.
`test_isocorrelveclg2`	Textfile (`fast_isolg2.dat`) containing the reference output of the `mdorado.veccor.isocorrelveclg2` function. Is used in the unit test of the `mdorado.veccor` module.

1.2 Example

In particular the water simuation can be used as an example input to test the functionality of MDorado. It is used in that way in many examples throughout this documentation, so the functions can be tested on an user-independent trajectory. This code, for example, is an example for the `hb_analyze` module and imports the file paths of the `water.tpr` (`water_topology`) and `water.xtc` (`water_trajectory`):

```
1 import MDAnalysis
2 from mdorado.hb_analyze import hb_analyze
3 from mdorado.data.datafilenames import water_topology,
  ↪ water_trajectory
4
5 u = MDAnalysis.Universe(water_topology, water_trajectory)
6 xgrp = u.select_atoms("name ow")
7 hgrp = u.select_atoms("name hw")[::2]
8
9 hb_analyze(universe=u, xgrp=xgrp, hgrp=hgrp, rmin=1.5, rmax=5,
  ↪ cosalphamin=-1, cosalphamax=1, bins=50)
```

2 Correlate

This function uses `scipy.signal.correlate` to cross correlate two discrete functions $a(t)$ and $b(t)$. The function computes $\langle a(0)b(t)\rangle$ directly via sums or using a Fast Fourier Transform algorithm, depending on which is faster (see `scipy.signal.convolve`). In addition to the original `scipy` functionality, the function tailors the correlation function so that only non-negative time values ($\langle a(0)b(t)\rangle$ for $t \geq 0$) are returned. It may only be sensible to calculate such a function if $a(t)$ and $b(t)$ (and the corresponding array elements `a[t]` and `b[t]`) reference the same point in time and the arrays `a` and `b` are of equal length. The autocorrelation function $\langle a(0)a(t)\rangle$ is computed if `b=None` (default).

2.1 Function

`mdorado.correlations.correlate(a, b=None)`

Parameters:

`a`:	one-dimensional ndarray or list Discrete values of the function $a(t)$.
`b`:	one-dimensional ndarray or list, optional Discrete values of the function $b(t)$. If `None` the autocorrelation function $\langle a(0)a(t)\rangle$ is computed. Default is `None`.

Returns: ndarray
An ndarray containing the correlation function $\langle a(0)b(t)\rangle$ for $t \geq 0$ is returned.

3 Lifetimes

To analyze the lifetime of a hydrogen bond or any other impermanent interaction we can define a bonding operator $h(t)$ which is unity if the criteria for the interaction are fulfilled and zero otherwise:[1,2]

$$h(t) = \begin{cases} 1, \text{if criteria are fulfilled} \\ 0, \text{otherwise} \end{cases} . \tag{3.1}$$

The fluctuations of $h(t)$ can be described by the autocorrelation function $C(t)$

$$C(t) = \frac{\langle h(0)h(t)\rangle - \langle h\rangle^2}{\langle h\rangle}, \tag{3.2}$$

which describes the probability of the bond being intact at the time t if the bond was intact at $t = 0$. The so-called intermittent lifetime of the interaction can be estimated from $C(t)$.[3–6]

The reactive flux approach[4,6–9] is another approach to estimate the lifetime of such interactions. To follow that approach we require the function $k_{\text{in}}(t)$

$$k_{\text{in}}(t) = -\frac{\left\langle \dot{h}(0)[1 - h(t)]H(t)\right\rangle}{\langle h\rangle}, \tag{3.3}$$

where $\dot{h}$ denotes the time-derivative of $h(t)$. $H(t)$ is a vicinity operator closely related to $h(t)$. If the donor and acceptor of the interaction are "near" each other $H(t)$ equals unity otherwise it equals zero.

The purpose of `calc_lifetime` is to calculate the correlation functions $\langle h(0)h(t)\rangle$ from equation 3.2 and $-\left\langle \dot{h}(0)[1 - h(t)]H(t)\right\rangle$ from equation 3.3. Normalizing the correlation functions will be up to the user, since several approaches are viable.[10] To obtain both correlations we first need to determine $h(t)$ and $H(t)$ for every donor-acceptor pair. After that, we are able to compute both correlation functions and average them over all donor-acceptor pairs.

3.1 Function

```
mdorado.lifetime.calc_lifetime(universe, timestep, xgrp, hgrp,
    cutoff_hy, cutoff_xy, angle_cutoff, ygrp=None, nproc=1,
    check_memory=True)
```

Parameters:

`universe`:	MD.Analysis.Universe Universe containing the trajectory.
`timestep`:	int or float Timestep between configurations in the `universe`. The unit is freely selectable and will influence the units of the output.
`xgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms X involved in the interaction X–H$\cdots$Y. Has to be the same size as `hgrp`.
`hgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms H involved in the interaction X–H$\cdots$Y. Has to be the same size as `xgrp`.
`ygrp`:	AtomGroup from MDAnalysis or None, optional MDAnalysis AtomGroup containing all atoms Y involved in the interaction X–H$\cdots$Y. If `None` is given, it is assumed that Y=X (interaction X–H$\cdots$X) and `xgrp` is taken as acceptor group. The default is `None`.
`cutoff_hy`	int or float Criterion for the H$\cdots$Y distance in Å to define $h(t)$. The criterion is fulfilled if the distance between a HY-pair is smaller than the value specified.
`cutoff_xy`	int or float Criterion for the X$\cdots$Y distance in Å to define $H(t)$. The criterion is fulfilled if the distance between a XY-pair is smaller than the value specified.
`angle_cutoff`	int or float Criterion for the angle $\alpha\angle$XHY in in radian to define $h(t)$. The cutoff is set so that if $\alpha >$ angle_cutoff the criterion is fulfilled.

`nproc`	int, optional Number of processors available to parallelize the execution of the script. The default is 1.
`check_memory`	bool, optional Perform an approximate check if the amout of memory is sufficient. The default is `True`.

Output:
For every donor i in `xgrp` a file `ct_i.dat` will be created. The file contains the results in three columns. The first column contains the timestep t in the same unit given in the option `timestep`. The second column contains $\langle h(0)h(t)\rangle$ for that donor. The third column contains $-\left\langle \dot{h}(0)[1-h(t)]H(t)\right\rangle$ for that donor in inverse units of `timestep`. As long as the amount of acceptors (`ygrp`) is constant the data of multiple donors i can be averaged by computing the arithmetic mean of the desired quantity.

3.2 Example

Using a water simulation from the files of the module, we take the first 20 water molecules and calculate correlation functions using one of the hydrogen atoms as donor but all water oxygen atoms as donor. The timestep of this example trajectory is 0.2 ps. The hydrogen bond was here defined by a distance cutoff H$\cdots$O of 2.5 Å and and angle cutoff of $\alpha > 2.27$ rad. For $H(t)$ the distance criterion X$\cdots$Y was set to 3.5 Å.

```
import MDAnalysis
from mdorado.lifetime import calc_lifetime
from mdorado.data.datafilenames import water_topology,
    water_trajectory

universe = MDAnalysis.Universe(water_topology, water_trajectory)

xgrp = universe.select_atoms("name ow")[:20]
hgrp = universe.select_atoms("name hw")[:40:2]
ygrp = universe.select_atoms("name ow")

```

```
calc_lifetime(universe=universe, timestep=0.2, xgrp=xgrp, hgrp=hgrp,
    ygrp=ygrp, cutoff_hy=2.5, angle_cutoff=2.27, cutoff_xy=3.5)
```

4 gofr

The radial distribution function $g_{\mathrm{AB}}(r)$ describes the density of particle B in a spherical shell of width dr at distance r around particle A in relation to the average number density of B $\langle \rho_{\mathrm{B}} \rangle$ in the system

$$g_{\mathrm{AB}}(r) = \frac{\langle \rho_{\mathrm{B}}(r) \rangle}{\langle \rho_{\mathrm{B}} \rangle} = \frac{1}{\langle \rho_{\mathrm{B}} \rangle \cdot N_{\mathrm{A}}} \left\langle \sum_{i \in \mathrm{A}}^{N_{\mathrm{A}}} \sum_{j \in \mathrm{B}}^{N_{\mathrm{B}}} \frac{\delta(r_{ij} - r)}{4\pi r^2} \right\rangle . \tag{4.1}$$

Here, N_{A} and N_{B} references the number of particles A and B in the system, respectively. It should be noted that $g_{\mathrm{AB}}(r) = g_{\mathrm{BA}}(r)$.

From $g_{\mathrm{AB}}(r)$ and $\langle \rho_{\mathrm{B}} \rangle$ the average cumulative number of neighbors $N_{\mathrm{B}}(R)$ of paricles B in a sphere of radius R around a particle A is obtainable via

$$N_{\mathrm{B}}(R) = \rho_{\mathrm{B}} \cdot 4\pi \int_0^R g_{\mathrm{AB}}(r)\, r^2 \mathrm{d}r \tag{4.2}$$

Similarily, $N_{\mathrm{A}}(R)$ can be computed using $\langle \rho_{\mathrm{A}} \rangle$.

Three modes (`mode`) are implemeted at the moment: `"site-site"`, `"cms-cms"`, and `"site-cms"`. The mode `"site-site"` computes the average $g_{\mathrm{AB}}(r)$ between all atoms A in `agrp` and all atoms B in `bgrp`. For example, if `agrp` contains atoms A0 and A1 while `bgrp` contains atoms B0 and B1, the pairs A0B0, A0B1, A1B0, and A1B1 will contribute to $g_{\mathrm{AB}}(r)$.

The mode `"cms-cms"` can be used to compute center-of-mass radial distribution functions. It will calculate the center-of-mass of atoms belonging to the same molecule in `agrp` and `bgrp` and procede to calculate the radial distribution function of these centers-of-mass. For example, given an `agrp` containing four atoms belonging to two different molecules (A0 and A1 belonging to molecule M0, A2 and A3 belonging to M1) and the same for `bgrp` (B0 and B1 belonging to M2, B2 and B3 belonging to M3) it will first calculate the centers-of-mass $\mathrm{cms}_{\mathrm{M0}}$(A0,A1), $\mathrm{cms}_{\mathrm{M1}}$(A2,A3), $\mathrm{cms}_{\mathrm{M2}}$(B0,B1), and $\mathrm{cms}_{\mathrm{M3}}$(B2,B3). The radial distribution function will then contain contributions from the pairs $\mathrm{cms}_{\mathrm{M0}}\mathrm{cms}_{\mathrm{M2}}$, $\mathrm{cms}_{\mathrm{M0}}\mathrm{cms}_{\mathrm{M3}}$, $\mathrm{cms}_{\mathrm{M1}}\mathrm{cms}_{\mathrm{M2}}$, and $\mathrm{cms}_{\mathrm{M1}}\mathrm{cms}_{\mathrm{M3}}$.

The mode `"site-cms"` is a mix of both of the functions described above, where `agrp` is taken atom-wise as in `gofr` and for `bgrp` the center-of-mass of atoms belonging to the same molecule is calculated as in `gofr_cms`.

4.1 Function

```
mdorado.gofr.Gofr(universe, agrp, bgrp, rmax, rmin=0, bins=100,
    mode="site-site", outfilename="gofr.dat")
```

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms A.
`bgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms B.
`rmax`:	int or float The upper boundary of the A$\cdots$B distance used for the $g(r)$ in units of Å.
`rmin`:	int or float, optional The lower boundary of the A$\cdots$B distance used for the $g(r)$ in units of Å. The default is 0.
`bins`:	int or sequence of scalars or str, optional Specifies the number of points between `rmin` (inclueded) and `rmax` (excluded). Will be used directly by numpy.histogram. From the numpy documentation: "If bins is an int, it defines the number of equal-width bins in the given range. If bins is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge, allowing for non-uniform bin widths. If bins is a string, it defines the method used to calculate the optimal bin width, as defined by histogram_bin_edges." The default is 100.
`mode`:	str, optional

Sets the mode for calculating different radial distribution functions: `"site-site"`, `"cms-cms"`, `"site-cms"`. If mode is set to `"site-site"`, the average radial distribution function of all sites in `agrp` to all sites in `bgrp` will be computed. The mode `"cms-cms"` will first compute the center-of-mass of sites belonging to the same molecule in `agrp` and `bgrp`, respectively, and then determin the radial distribution function between those centers of mass. The mode `"site-cms"` is a mix between the two, where every site in `agrp` is taken individualy but for `bgrp` the center-of-mass of sites belonging to the same molecule is computed firs. The default is `"site-site"`.

`outfilename`: str, optional
The name of the output file. The default is `"gofr.dat"`.

Output:
The program creates a file named `outfilename` with the distance r in Å (first column), the radial distribution function $g_{\mathrm{AB}}(r)$ (second column), the cumulative number of neighbors A in a sphere of radius r around particle B $N_{\mathrm{A}}(r)$ (third column), and the cumulative number of neighbors B in a sphere of radius r around particle A $N_{\mathrm{B}}(r)$ (fourth column).

Class Methods:

`rdat`: Distance r (center of bins).

`edges`: Edges of the bins.

`hist`: Radial distribution function $g_{\mathrm{AB}}(r)$.

`annn`: Average number of neighbors A in a sphere of radius r around particle B $N_{\mathrm{A}}(r)$.

`bnnn`: Average number of neighbors B in a sphere of radius r around particle A $N_{\mathrm{B}}(r)$.

`avvol`: Average volume of the universe.

`na`: Number of particles A in `agrp`. If mode is `"site-site"` or `"site-cms"`, `na` is the number of sites in `agrp`. If mode is `"cms-cms"`, `na` is the number of molecules (centers-of-mass) in `agrp`.

`nb`: Number of particles B in `bgrp`. If mode is `"site-site"`, `na` is the number of sites in `agrp`. If mode is `"site-cms"` or `"cms-cms"`, `nb` is the number of molecules (centers-of-mass) in `bgrp`.

4.2 Example

We start with a simulation of water from the data files shipped with the module, where all oxygen atoms are named `"ow"` and all hydrogen atoms `"hw"`. The resname of the water molecules is `"sol"` for solvent. We will compute three different radial distribution functions to show reveal the differences in `"site-site"`, `"cms-cms"`, and `"site-cms"`: Firstly, we will use `"site-site"` to calculate the radial distribution function between all hydrogen and oxygen (H$\cdots$O) atoms, which could for example be used to define the hydrogen bond O–H$\cdots$O. Secondly, we will calculate the center-of-mass radial distribution function of all water molecules (cms$\cdots$cms) using `"cms-cms"`. Thirdly, we use `"site-cms"` to compute the radial distribution function of all hydrogen atoms to the centers-of-mass of all water molecules (H$\cdots$cms).

We first have to create a universe and select different AtomGroups to achieve the goals described above.

```
1 import MDAnalysis
2 from mdorado.gofr import Gofr
3 from mdorado.data.datafilenames import water_topology,
  ↪ water_trajectory
4
5 u = MDAnalysis.Universe(water_topology, water_trajectory)
6 hgrp = u.select_atoms("name hw")
7 ogrp = u.select_atoms("name ow")
8 watergrp = u.select_atoms("resname sol")
9
10 sitesite = Gofr(universe=u, agrp=hgrp, bgrp=ogrp, rmin=1.1, rmax=6,
  ↪ bins=200, mode="site-site", outfilename="h_o.dat")
11 cmscms = Gofr(universe=u, agrp=watergrp, bgrp=watergrp, rmin=1.1,
  ↪ rmax=6, bins=200, mode="cms-cms", outfilename="cms_cms.dat")
12 sitecms = Gofr(universe=u, agrp=hgrp, bgrp=watergrp, rmin=1.1,
  ↪ rmax=6, bins=200, mode="site-cms", outfilename="h_cms.dat")
```

In Fig. 4.1 the three different $g(r)$ are plotted. Additionaly, we obtain the neighbour numbers $N_\mathrm{A}(r)$ $N_\mathrm{B}(r)$ for each pair. In case of the H$\cdots$O distribution $N_\mathrm{A}(r)$ would be the average number of hydrogen atoms in a sphere of radius r around an oxygen atom. In case of the cms$\cdots$cms distribution $N_\mathrm{A}(r) = N_\mathrm{B}(r)$ denotes the number of water molecules in a sphere of radius r around a water molecule. In case of the H$\cdots$cms distribution $N_\mathrm{A}(r)$ is the number of water molecules around in a sphere of radius r around a hydrogen atom.

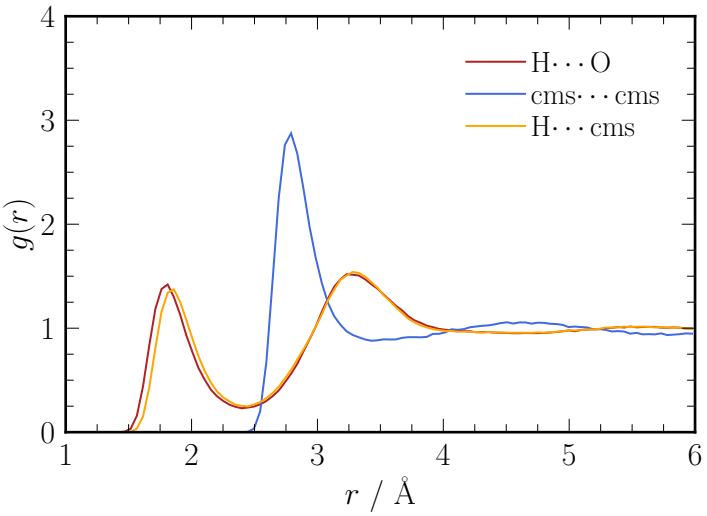

Fig. 4.1: Radial distribution functions obtained from the example above.

5 hb_analyze

Geometric criteria can be used to define a hydrogen bond. Distance criteria can often be derived from pair-correlation functions but it may be required to include angular restrictions on the interaction. Twodimensional potentials of mean force (PMFs) can be used to obtain such criteria.[11,12] The PMF is calculated using the probability density of finding a donor-acceptor pair with the respective donor-acceptor ($\mathrm{H}\cdots\mathrm{Y}$) distance r and angle α ($\alpha\angle\mathrm{XHY}$).

This density can be derived from populations from equilibrium molecular dynamics trajectories. Therefore, donor-acceptor pairs with a distance r between `rmin` and `rmax` and an angle $\cos(\alpha)$ between `cosalphamin` and `cosalphamax` will be counted in a twodimensional histogram according to the number of bins specified with the option `bins`. Each count is weighted with the respective r^{-2} to account for the growth of the spherical volume element with increasing r. The histogram is then normalized to the respective probabilty density function $P(r, \cos(\alpha))$ using the area of each bin $\mathrm{d}r \cdot \mathrm{d}\cos(\alpha)$ and the sum of all counts so that the integral over P is unity. At the end, the natural logarithm of P in each bin is calculated, due to the connection between P and the PMF via

$$F = -k_\mathrm{B} T \ln(P) + c, \tag{5.1}$$

with the Boltzmann constant k_B, the temperature T, and an unknown constant c. The output is a twodimensional grid where each bin contains $\ln(P)$ of the respective bin.

5.1 Function

```
mdorado.hb_analyze.hb_analyze(universe, xgrp, hgrp, rmax, ygrp=None,
    rmin=0, cosalphamin=-1, cosalphamax=1, bins=50,
    outfilename="hb_analyze.dat", ralphalist=False)
```

Parameters:

`universe`: MDAnalysis.Universe
Universe containing the trajectory.

`xgrp`: AtomGroup from MDAnalysis
AtomGroup containing all atoms X involved in the interaction X–H$\cdots$Y. Has to be the same size as `hgrp`.

`hgrp`: AtomGroup from MDAnalysis
AtomGroup containing all atoms H involved in the interaction X–H$\cdots$Y. Has to be the same size as `xgrp`.

`ygrp`: AtomGroup from MDAnalysis or None, optional
MDAnalysis AtomGroup containing all atoms Y involved in the interaction X–H$\cdots$Y. If `None` is given, it is assumed that Y=X (interaction X–H$\cdots$X) and `xgrp` is taken as acceptor group. The default is `None`.

`rmax`: int or float
The upper boundary of the H$\cdots$Y distance in units of Å.

`rmin`: int or float, optional
The lower boundary of the H$\cdots$Y distance in units of Å. The default is 0.

`cosalphamin`: int or float, optional
The lower boundary of $\cos(\alpha)$ ($\alpha\angle$XHY). The default is -1.

`cosalphamax`: int or float, optional
The upper boundary of $\cos(\alpha)$ ($\alpha\angle$XHY). The default is 1.

`bins`: int or array_like or [int, int] or [array, array], optional

Bins used for the 2D-histogram. Will be used directly by numpy.histogram2d. For two numbers the first will specify the bins of the H$\cdots$Y distance (x_edges) and the second will specify the bins of cos(α) (y_edges) The default is 50. Specifications:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
- If [int, int], the number of bins in each dimension (nx, ny = bins).
- If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).
- A combination [int, array] or [array, int], where int is the number of bins and array is the bin edges.

`outfilename`: str, optional
The name of the outputfile. The default is `hb_analyze.dat`.

`ralphalist`: bool, optional
Changes the output from the weighted probability density matrix to the list containing all the H$\cdots$Y distances and corresponding cos(α) from which the probability density is calculated. The default is `False`.

Output:
The program creates a file named `outfilename` with the weighted twodimensional histogram. The first axis represents the H$\cdots$Y distance and the second axis represents cos(α) ($\alpha\angle$XHY).

If `ralphalist=True` the file contains the distances and corresponding angles of HY-pairs as a list: in the first column the distances are written in units of Å and the second column indicates the cosine of the corresponding angle cos(α), both in the respective range `rmin` to `rmax` and `cosalphamin` to `cosalphamax`.

5.2 Example and Visualization

To use `hb_analyze` we first have to create a universe, define `xgrp` and `hgrp`, the range of the histogram, and the amount of bins in each dimension. If no `ygrp` is given the

program will use `xgrp` as acceptor group and analyze the interaction X–H $\cdots$ X instead. Here an example for a water simulation from the datafiles of mdorado where the oxygen atoms are named `"ow"` and the hydrogen atoms `"hw"`:

```
import MDAnalysis
from mdorado.hb_analyze import hb_analyze
from mdorado.data.datafilenames import water_topology,
    water_trajectory

u = MDAnalysis.Universe(water_topology, water_trajectory)
xgrp = u.select_atoms("name ow")
hgrp = u.select_atoms("name hw")[::2]

hb_analyze(universe=u, xgrp=xgrp, hgrp=hgrp, rmin=1.5, rmax=5,
    cosalphamin=-1, cosalphamax=1, bins=50)
```

After excecution a file `hb_analyze.dat` (changable by the option `outfilename`) can be found in the current folder. It contains the 50×50 (`bins`) matrix of the weighted probability function. This matrix can be plotted by matplotlibs `contour` and similar programs. Here an example using matplotlibs `contourf`:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import ticker
import matplotlib.colors as col

lowcolor = '#ffffff'
midcolor1 = '#6090f0'
midcolor2 = '#30f050'
midcolor3 = '#f0f000'
midcolor4 = '#f06000'
highcolor = '#b02000'
cmapown = col.LinearSegmentedColormap.from_list('own',
    [lowcolor,midcolor1,midcolor2,midcolor3,midcolor4,highcolor])
cmapown.set_over('#9e1c00')

```

```
15 rmin=1.5
16 rmax=5
17 cosalphamin=-1
18 cosalphamax=1
19
20 fig, ax = plt.subplots()
21 histo_matrix = np.loadtxt("hb_analyze.dat")
22 levels = ticker.MaxNLocator(nbins=60).tick_values(-3, 2)
23 cax = ax.contourf(histo_matrix, extent=(cosalphamin, cosalphamax,
   ↪  rmin, rmax), levels=levels, extend='both', cmap=cmapown)
24 plt.xlabel('$\\cos(\\alpha)$')
25 plt.ylabel('$r$ / \\AA')
26 plt.axis([cosalphamin, cosalphamax, rmin, rmax])
27 cbar = fig.colorbar(cax, ticks=[-3, -2, -1, 0, 1, 2])
28 cbar.ax.set_ylabel('$\\log[W(\\cos(\\alpha), r)]$')
29 plt.tight_layout()
30 plt.savefig("histo.pdf")
31 plt.clf()
```

After importing the necessary modules, we first define or own colormap `cmapown` (line 6 to 13). Standard colormaps can be found here. The output of `hb_analyze` only contains the weighted probabilty densities for each bin and not their position, so we have to tell the program in line 15–18 in which range the histogram is plotted (option `extent` of `contourf` line 23 and x- and y-axis limits line 26).

The actual plotting happens onwards from line 20. Using numpys `loadtxt` we load the histogram matrix into the array `histo_matrix` (line 21). In line 22 we define the amount of bins (`nbis=60`) and the range `(-3, 2)` of the coloraxis. The array can directly be processed by `contourf` where we also input the range, levels, and colormap. The option `extend='both'` enables the colors beyond the levels definded before (arrows above and below the coloraxis). The lines 24 to 28 are defining the axis-ticks and -labels. After that the plot is already finished and can be saved or shown directly. An example plot for a small watersimulation is shown in Fig. 5.1.

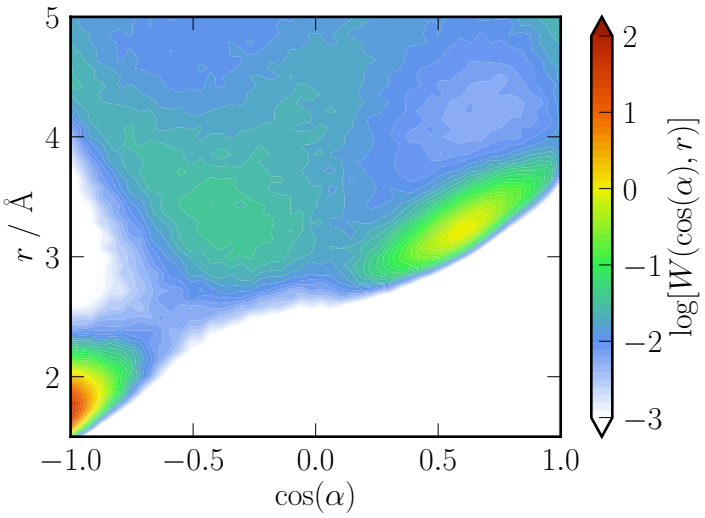

Fig. 5.1: Example plot of a twodimensional histogram computed with `hb_analyze`.

6 Mean Square Displacement

First, to compute the mean square displacement (MSD) of a particle the trajectory has to be "unwrapped", so that the effects of the periodic boudary conditions (PBCs) are reversed. The function `mdorado.msd.unwrap` is able to accomplish this task for cuboid boxes (all box angles are 90°).

Given a trajectory (`universe`) AtomGroup (`agrp`) the positions of the atoms of interest are recomputed so that each atom starts in the origin of the coordinate system at $t = 0$. The displacement in each time step is then added incrementally to obtain a trajectory relative to this starting position. If the magnitude of the displacement in one dimension (x, y or z) is larger than half a box length in that dimension, indicating a jump of the atom due to PBCs, the displacement is adjusted by adding or subtracting the box length to ensure "jump-free" movement.

From these coordinates `mdorado.msd.msd` is able to compute the MSD using an efficient algorithm[13]

$$\mathrm{MSD}(t) = \left\langle |\boldsymbol{r}(t) - \boldsymbol{r}(0)|^2 \right\rangle , \tag{6.1}$$

where $\boldsymbol{r}(t)$ denotes a positional vector of the unwrapped trajectory.

6.1 Functions

```
mdorado.msd.unwrap(universe, agrp, dimensionskey="xyz", cms=False)
```

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms for which the trajectory should be unwrapped.
`dimensionskey`:	str, optional

Dimensions in which the trajectory is unwrapped. The keywords are:

- `"xyz"` for all dimensions
- `"x"`, `"y"`, and `"z"` for one of the three principal box axes
- `"xy"`, `"xz"`, and `"yz"` for a combination of two of the three box axes

Changes the shape of the output array. The default value is `"xyz"`.

`cms`: bool, optional
If `cms=True` the program computes the movement of the center-of-mass of atoms belonging to the same residue in `agrp`. If, for example, `agrp` would contain all atoms of water molecules in the simulation this option allows for the calculation of the average center-of-mass MSD of these water molecules. If for the same case `cms=False` is chosen, the program calculates the MSD of all the various atoms individually, averaging over hydrogen as well as oxygen atoms. The default value is `False`.

Returns: ndarray
An ndarray containing the unwrapped positions. The shape of the array is $(N_\mathrm{A}, N_\mathrm{dim}, N_\mathrm{steps})$, where N_A denotes the number of atoms in `agrp` (or the number of residues if `cms=True`), N_dim denotes the number of dimensions according to the option `dimensionskey`, and N_steps is the number of timesteps in the `universe`.

```
mdorado.msd.msd(positions, dt, outfilename="msd.dat")
```

Parameters:

`positions`: ndarray
ndarray of d

`dt`: int or float
Difference in time between two configurations in `positions`.

`outfilename`: str, optional
The name of the outputfile. The default is `"msd.dat"`.

Output:
The function writes a file with two columns, where the first column is the time delay t and the second column is the average MSD for that time delay $\mathrm{MSD}(t) = \langle|(\boldsymbol{r}(t) - \boldsymbol{r}(0)|^2\rangle$

6.2 Example

We are going to compute two slightly different MSDs from the example water trajectory. First, we are interested in the movement of the center of mass of the water molecules. Therefore, we create an atom group with all oxygen and hydrogen atoms (`solgrp`). The `msd.unwrap` function repairs the jumps due to the periodic boundary conditions ("unwrapping"). Because we are interested in the 3D movement of the center-of-mass, we set the option `dimensionskey` to `"xyz"` and the option `cms` to `"True"` (line 10). The function atomatically computes the center-of-mass of atoms belonging to the same molecule and returns the trajectory of these centers in the ndarray `cmspos` of shape (128, 3, 2501) $[(N_\mathrm{A}, N_\mathrm{dim}, N_\mathrm{steps})]$. This array can be used as input for the `msd.msd` function along with the timestep (`dt`) to calculate the average MSD of these centers-of-mass.

The second example computes the average MSD of all hydrogen atoms in the trajectory. Therefore, we create an atom group containing the hydrogen atoms (`hgrp`) and unwrap their trajectory with `cms` set to `False` (default) (line 13). The array `hpos` is of shape (256, 3, 2501) (two hydrogen atoms per water molecule) and can again be used as an input for the `msd.msd` function to compute the MSD.

```
1  import MDAnalysis
2  from mdorado import msd
3  from mdorado.data.datafilenames import water_topology,
   ↪  water_trajectory
4
5  u = MDAnalysis.Universe(water_topology, water_trajectory)
6  solgrp = u.select_atoms("resname sol")
7  hgrp = u.select_atoms("name hw")
8  dt = 0.2
9
10 cmspos = msd.unwrap(universe=u, agrp=solgrp, dimensionskey="xyz",
   ↪  cms=True)
11 msd.msd(positions=cmspos, dt=dt, outfilename="msd_cms.dat")
12
```

```
13 hpos = msd.unwrap(universe=u, agrp=hgrp, dimensionskey="xyz",
   ↪  cms=False)
14 msd.msd(positions=hpos, dt=dt, outfilename="msd_h.dat")
```

7 vectors

7.1 norm_vecarray

Given an array of vectors, normalizes each vector and returns the array of normalized vectors as well as the length of each vector.

`mdorado.vectors.norm_vecarray(vecarray)`

Parameters:

`vecarray`:	ndarray Array of shape N_{vec} (number of vectors), N_{dim} (dimensionality of the vectors) containing the vectors that will be normalized.

Returns: `unitvecarray`, `norm`: ndarray, ndarray
Returns two arrays, the first (unitvecarray) is of the same shape as the input array and contains the normalized vectors and the second (norm) is of the shape (N_{vec},) and contains the length of the corresponding original vector.

Example:

```
1 # example_norm_vecarray.py
2 import numpy as np
3 from mdorado.vectors import norm_vecarray
4
5 vectorarray = np.array([[11,4,-4], [4,1,8], [-6,-7, 2], [3,0, -1]])
6 unitvectors, lengths = norm_vecarray(vecarray=vectorarray)
7 print(unitvectors)
8 print(lengths)
```

The function requires an array of input vectors (line 5). These vectors will each be

normalized and returned as unit vectors. Additionally, the length of the original vectors is returned:

```
$ python python example_norm_vecarray.py
[[ 0.88929729  0.32338083 -0.32338083]
 [ 0.44444444  0.11111111  0.88888889]
 [-0.63599873 -0.74199852  0.21199958]
 [ 0.9486833   0.         -0.31622777]]
[12.36931688  9.          9.43398113  3.16227766]
```

7.2 pbc_vecarray

Applies periodic boundary condition to an array of vectors so that the resulting array satisfies the minimum image convention for cuboid simulation boxes (all angles are 90°).

`mdorado.vectors.pbc_vecarray(vecarray, box)`

Parameters:

`vecarray`:	ndarray Array of shape N_{vec} (number of vectors), N_{dim} (dimensionality of the vectors) on which the periodic boundary condition will be applied.
`box`:	array-like One-dimensional array-like where the elements contain the length of the simulation box in the particular dimension.

Returns: ndarray
Array of the same shape as `vecarray` containing the vectors corrected for the minimum image convention.

Example:

```
1 # example_pbc_vecarray.py
2 import numpy as np
3 from mdorado.vectors import pbc_vecarray
```

```
4
5 vectorarray = np.array([[11,4,-4], [4,1,8], [-6,-7, 2], [3,0, -1]])
6 box = [10,5,7]
7 pbc_vectorarray = pbc_vecarray(vecarray=vectorarray, box=box)
8 print(pbc_vectorarray)
```

The function requires an array containing vectors (line 5) as well as the length of each box dimension (line 6) as an input. At the moment, this only works for cuboid simulation boxes (all angles 90°). The printed array contains the corrected vectors:

```
$ python example_pbc_vecarray.py
[[ 1 -1  3]
 [ 4  1  1]
 [ 4 -2  2]
 [ 3  0 -1]]
```

7.3 vectormatrix

Given two sets of particle positions $\boldsymbol{A} = [\boldsymbol{A_1}, \boldsymbol{A_2}, \ldots, \boldsymbol{A_n}]$ and $\boldsymbol{B} = [\boldsymbol{B_1}, \boldsymbol{B_2}, \ldots, \boldsymbol{B_m}]$ computes a matrix $\boldsymbol{C}$ containing vectors for all combinations $\boldsymbol{C_{i,j}} = \boldsymbol{B_j} - \boldsymbol{A_i}$.

`mdorado.vectors.vectormatrix(apos, bpos)`

Parameters:

apos:	ndarray Array of shape (n, 3) containing the position vectors of all particles A.
bpos:	ndarray Array of shape (m, 3) containing the position vectors of all particles B.

Returns: ndarray
Returns an array (`abmat`) of shape (n, m, 3) containing all vectors $\overrightarrow{A_i B_j}$:
`abmat[i, j] = bpos[j] - apos[i]`.

Example:

```
# example_vectormatrix.py
import numpy as np
from mdorado.vectors import vectormatrix

vectorarray = np.array([[11,4,-4], [4,1,8], [-6,-7, 2], [3,0, -1]])
vecmat = vectormatrix(apos=vectorarray, bpos=vectorarray)
print(vecmat)
```

Here, we use the vectors in `vectorarray` as positional vectors of four different particles $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$. The resulting matrix then contains all vectors connecting the particles. The comments (everything after #) are of course not part of the output but should provide clarity over the structure of the resulting matrix:

```
$ python example_vectormatrix.py
[[[  0.   0.   0.]      #\vec{AA}
  [ -7.  -3.  12.]      #\vec{AB}
  [-17. -11.   6.]      #\vec{AC}
  [ -8.  -4.   3.]]     #\vec{AD}

 [[  7.   3. -12.]      #\vec{BA}
  [  0.   0.   0.]      #\vec{BB}
  [-10.  -8.  -6.]      #\vec{BC}
  [ -1.  -1.  -9.]]     #\vec{BD}

 [[ 17.  11.  -6.]      #\vec{CA}
  [ 10.   8.   6.]      #\vec{CB}
  [  0.   0.   0.]      #\vec{CC}
  [  9.   7.  -3.]]     #\vec{CD}

 [[  8.   4.  -3.]      #\vec{DA}
  [  1.   1.   9.]      #\vec{DB}
  [ -9.  -7.   3.]      #\vec{DC}
  [  0.   0.   0.]]]    #\vec{DD}
```

7.4 get_vectormatrix

Uses `mdorado.vectors.vectormatrix` to compute a vector matrix containing all vectors between all combinations of particles A and B for every timestep in a universe.

```
mdorado.vectors.get_vectormatrix(universe, agrp, bgrp, pbc=True)
```

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of A.
`bgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of B.
`pbc`:	bool, optional Specifies whether periodic boundary conditions should be applied to find the shortest vector from A to an image of B. Calls the `mdorado.vectors.pbc_vecarray` function. Only works for cuboid boxes (all angles are 90°). Default is `True`.

Returns: ndarray
Array of the shape (N_A, N_B, N_steps, 3), where N_A the number of particles A, N_B the number of particles B, N_steps is the number of timesteps in the universe, and the last axis refers to the three directions in space x, y and z. For example, the array element AB[i, j, k, 2] refers to the z-component of the vector pointing from A_i to B_j at the k-th timestep of the simulation.

Example:

```
# example_get_vectormatrix.py
import MDAnalysis
import mdorado.vectors as mvec
from mdorado.data.datafilenames import water_topology,
    water_trajectory

u = MDAnalysis.Universe(water_topology, water_trajectory)
```

```
7 ogrp = u.select_atoms("name ow")
8
9 vecmat = mvec.get_vectormatrix(universe=u, agrp=ogrp, bgrp=ogrp,
  ↪  pbc=True)
10 print(vecmat.shape)
```

Using the SPC water simulation (`water_topology`, `water_trajectory`) included in MDorado, we create an `AtomGroup` containing all 125 oxygen atoms (line 7). The function `get_vectormatrix` then computes a timeseries for every oxygen-oxygen vector in the trajectory (line 9). With `pbc=True` (default), we make sure that we always consider the vector from an oxygen atom to the closest image of the other oxygen atom. Due to the size of the resulting array, we only print its shape here:

```
$ python example_get_vectormatrix.py
(125, 125, 1001, 3)
```

The first axis refers to the oxygen atom used as a starting point for the vector. The second axis refers to the second oxygen atom, the end point of the vector. The thrid axis refers to the timestep in the trajectory and the last axis to the three coordinates x, y, and z. The element `vecmat[6, 108, 515, 0]` refers to the x-component of the vector pointing from the 6th oxygen atom to the 108th oxygen atom at the 515th timestep. Due to the symmetry of the matrix in our example, it is the opposite of the element `vecmat[108, 6, 515, 0]`.

7.5 get_vecarray

Given to AtomGroups agrp and bgrp, it computes the time evolution of every vector bgrp[i]-agrp[i] for the whole trajectory. Can account for periodic boundary conditions for cuboid boxes.

`mdorado.vectors.get_vecarray(universe, agrp, bgrp, pbc=True)`

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis

	AtomGroup containing all atoms of type A for which a vector $\overrightarrow{AB}$ should be computed.
`bgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of type B for which a vector $\overrightarrow{AB}$ should be computed.
`pbc`:	bool, optional Specifies whether periodic boundary conditions should be applied to find the shortest vector from A to an image of B. Calls the `mdorado.vectors.pbc_vecarray` function. Only works for cuboid boxes (all angles are 90°). Default is `True`.

Returns: ndarray
Array containing the trajectory of every vector bgrp[i]-agrp[i] of the shape N_{vec} (number of vectors), N_{ts} (number of timesteps in the universe), N_{dim} (number of dimensions).

Example:

```
# example_get_vecarray.py
import MDAnalysis
import mdorado.vectors as mvec
from mdorado.data.datafilenames import water_topology,
    water_trajectory

u = MDAnalysis.Universe(water_topology, water_trajectory)
ogrp = u.select_atoms("name ow")
hgrp = u.select_atoms("name hw")[::2]

vectorarray = mvec.get_vecarray(universe=u, agrp=ogrp, bgrp=hgrp,
    pbc=False)
print(vectorarray.shape)
```

From the example SPC water simulation, we create two `AtomGroups`, one containing all oxygen atoms and the other containing every second hydrogen atom (one per water molecule). Using these and the `get_vecarray` function, we can compute a timeseries of one OH-vector per water molecule in our simulation.

The trajectories of our example simulation are "repaired" in such a way, that atoms belonging to the same molecule are an the same side of the simulation box as the center-of-mass of the molecule. This has the advantage, that intramolecular vectors are not broken up by the periodic boundary condition and we can set `pbc=False`.

The resulting array is of shape (125, 1001, 3):

```
$ python example_get_vecarray.py
(125, 1001, 3)
```

The first axis references the 125 different OH-vectors in our simulation, one per water molecule. The second axis references the timestep of the trajectory and the third axis references the three coordinates x, y, and z. The array element `vectorarray[49][957][2]` references the z-component of one OH-vectors of the 49th water molecule at the timestep number 957.

7.6 get_normal_vecarray

Computes given to AtomGroups agrp, bgrp and cgrp, it computes the normal vectors

$$\boldsymbol{n}_i(t) = [\boldsymbol{b}_i(t) - \boldsymbol{a}_i(t)] \times [\boldsymbol{c}_i(t) - \boldsymbol{a}_i(t)] \tag{7.1}$$

for every triple of atoms positions $(\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i)$ and for every timestep t of the trajectory.

```
mdorado.vectors.get_normal_vecarray(universe, agrp, bgrp, cgrp,
    pbc=True)
```

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of type A used to define the plane containing the atoms A_i, B_i and C_i.
`bgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of type B used to define the plane containing the atoms A_i, B_i and C_i.
`cgrp`:	AtomGroup from MDAnalysis

AtomGroup containing all atoms of type C used to define the plane containing the atoms A_i, B_i and C_i.

`pbc`: bool, optional
Specifies whether periodic boundary conditions should be applied to find the shortest vector from A to an image of B and C. Calls the `mdorado.vectors.pbc_vecarray` function. Only works for cuboid boxes (all angles are 90°). Default is `True`.

Returns: ndarray
An ndarray of shape N_{vec} (number of normal vectors $\overrightarrow{n}$), N_{steps} (number of timesteps in `universe`), N_{dim} (number of dimensions) containing the time evolution of all normal vectors $\overrightarrow{n} = \overrightarrow{\text{AB}} \times \overrightarrow{\text{AC}}$ in the trajectory.

Example:

```
1  # example_get_normal_vecarray.py
2  import MDAnalysis
3  import mdorado.vectors as mvec
4  from mdorado.data.datafilenames import water_topology,
   ↪  water_trajectory
5
6  u = MDAnalysis.Universe(water_topology, water_trajectory)
7  ogrp = u.select_atoms("name ow")
8  h1grp = u.select_atoms("name hw")[::2]
9  h2grp = u.select_atoms("name hw")[1::2]
10
11 normalvectorarray = mvec.get_normal_vecarray(universe=u, agrp=ogrp,
   ↪  bgrp=h2grp, cgrp=h2grp, pbc=False)
12 print(normalvectorarray.shape)
```

To compute a vector perpendicular to the molecular plane for every water molecule (normal vector) from our example SPC water simulation at every timestep, we first have to create three `AtomGroups`. The first (line 7) contains all oxygen atoms, the second (line 8) contains every second hydrogen atom starting with the 0th ("the first" of each water molecule), an the second (line 9) also contains every second hydrogen atom but starting with the 1st ("the second" of each water molecule).

The trajectories of our example simulation are "repaired" in such a way, that atoms belonging to the same molecule are an the same side of the simulation box as the center-of-mass of the molecule. This has the advantage, that intramolecular vectors are not broken up by the periodic boundary condition and we can set `pbc=False`.

The shape of the resulting array is (125, 1001, 3):

```
$ python example_get_normal_vecarray.py
(125, 1001, 3)
```

The first axis references the 125 different water molecules in our simulation. The second axis references the timestep of the trajectory and the third axis references the three coordinates x, y, and z. The array element `normalvectorarray[120][531][0]` references the x-component of a vector perpendicular to the plane containing the three atoms of the 120th water molecule at the timestep number 531.

8 Reorientational Correlation

The dynamic of the loss of reorientational correlation can be determined via reorientational correlation functions $R_i(t)$. In general, it is defined via

$$R_i(t) = \langle P_i\{\cos[\theta(0)]\} P_i\{\cos[\theta(t)]\} \rangle \,, \tag{8.1}$$

where $P_i\{\cos[\theta(t)]\}$ denotes the i-th Legendre polynominal of the cosine of the angle θ between a vector and a fixed external reference vector at time t. The brackets $\langle \ldots \rangle$ indicate averaging vectors of the same kind as well as all times "0".

In an isotropic medium like unordered liquids, the choice of reference vector does not influence $R_i(t)$. By choosing the orientation of the vector at time "0" as the reference vector equation 8.1 simplifies to

$$R_i(t) = \langle P_i\{\cos[\theta(t)]\} \rangle \,, \tag{8.2}$$

as $\cos[\theta(0)] = 1$ and therefore $P_i\{\cos[\theta(0)]\} = 1$ in that case. Compared to using a fixed reference vector this simplification also improves the statistics of $R_i(t)$ immensly. Both, the anisotropic as well as the isotropic case, can be investigated with MDorado.

To obtain $P_i\{\cos[\theta(t)]\}$, we first need to compute $\cos[\theta(t)]$ via

$$\cos[\theta(t)] = \boldsymbol{u}(t) \cdot \boldsymbol{u}_{\text{ref}}, \tag{8.3}$$

where $\boldsymbol{u}(t)$ is the unit vector of interest and $\boldsymbol{u}_{\text{ref}}$ is a fixed external reference unit vector. Thus, $\boldsymbol{u}_{\text{ref}}$ for all timesteps is needed. The `get_vecarray` and `norm_vecarray` functions of the `mdorado.vectors` module can be used to obtain a trajectory of unit vectors. For the computation of $R_i(t)$, one should also make sure that the vector of interest does not suddenly "flip" due to atoms crossing the boundary of the simulation box, as this reorientation would be an artifact of the periodic boundary condition.

Given a trajectory of vectors, the functions `correlvec` and `isocorrelvec` can be used to obtain $R_i(t)$. The `correlvec` function follows equation 8.1 and needs a fixed external vector to obtain $R_i(t)$. The computation makes use of correlations via a Fast-Fourier-transform algorithm and is therefore fast even for long trajectories. But compared to the isotropic simplification via `isocorrelvec`, the statistics of $R_i(t)$ is worse for `correlvec`.

Vice versa, the `isocorrelvec` function can be very slow for long trajectories, as we are not able to to use a Fast-Fourier-transform algorithm for an arbitrary Legendre polyomial. For this reason, fast algorithms for the correlation functions of the first and second Legendre polynomials using the isotropic simplification are included in MDORADO via the functions `isocorrelveclg1` ($R_1(t)$) and `isocorrelveclg2` ($R_2(t)$). These functions combine the superior statistics of the isotropic simplification and the computational speed of a Fast-Fourier-transform algorithm.

8.1 Functions

```
mdorado.veccor.correlvec(vecarray, refvec, dt, nlegendre,
    outfilename=False, normed=True)
```

Parameters:

`vecarray`: ndarray
Array of shape N_{vec} (number of vectors), N_{steps} (number of timesteps), N_{dim} (number of dimensions) containing the time evolution of all unit vectors of interest. See functions `get_vecarray`, `get_normal_vecarray` and `norm_vecarray` of the `mdorado.vectors` module for ways to obtain such an array.

`refvec`: ndarray or array-like
Fixed external reference vector. Will be normalized internally.

`dt`: int or float
Timestep used in `vecarray`.

`nlegendre`: int
Specifies which Legendre polynomial should be computed. For example `nlegendre` $= 2$ references the second legendre polynomial $P_2\{\cos[\theta(t)]\} = \frac{3}{2}\cos^2[\theta(t)] - \frac{1}{2}$ which will be used to computed $R_2(t)$ according to equation 8.1.

`outfilename`: str
If specified an xy-file with the name `str(outfilename)` containing t and $R_i(t)$ will be written. If `False` no file will be written. Default is `False`

`normed`: boolean

Specifies whether the function $R_i(t)$ should be normalized or not. Default is `True`

Returns: `timesteps, allcorrel`: ndarray, ndarray
Returns two arrays, the first containing information about the timestep t and the second containing the averaged function $R_i(t)$.

`mdorado.veccor.isocorrelvec(vecarray, dt, nlegendre, outfilename=False)`

Parameters:

`vecarray`: ndarray
Array of shape N_{vec} (number of vectors), N_{steps} (number of timesteps), N_{dim} (number of dimensions) containing the time evolution of all unit vectors of interest. See functions `get_vecarray`, `get_normal_vecarray` and `norm_vecarray` of the `mdorado.vectors` module for ways to obtain such an array.

`dt`: int or float
Timestep used in `vecarray`.

`nlegendre`: int
Specifies which Legendre polynomial should be computed. For example `nlegendre` $= 2$ references the second legendre polynomial $P_2\{\cos[\theta(t)]\} = \frac{3}{2}\cos^2[\theta(t)] - \frac{1}{2}$ which will be used to computed $R_2(t)$ according to equation 8.2.

`outfilename`: str
If specified an xy-file with the name `str(outfilename)` containing t and $R_1(t)$ will be written. If `False` no file will be written. Default is `False`

Returns: `timesteps, allcorrel`: ndarray, ndarray
Returns two arrays, the first containing information about the timestep t and the second containing the averaged function $R_i(t)$ using the isotropic simplification in equation 8.2.

`mdorado.veccor.isocorrelveclg1(vecarray, dt, outfilename=False)`

Parameters:

`vecarray`: ndarray
Array of shape N_{vec} (number of vectors), N_{steps} (number of timesteps), N_{dim} (number of dimensions) containing the time evolution of all unit vectors of interest. See functions `get_vecarray`, `get_normal_vecarray` and `norm_vecarray` of the `mdorado.vectors` module for ways to obtain such an array.

`dt`: int or float
Timestep used in `vecarray`.

`outfilename`: str
If specified an xy-file with the name `str(outfilename)` containing t and $R_i(t)$ will be written. If `False` no file will be written. Default is `False`

Returns: `timesteps, allcorrel`: ndarray, ndarray
Returns two arrays, the first containing information about the timestep t and the second containing the averaged function $R_1(t)$ using the isotropic simplification in equation 8.2.

`mdorado.veccor.isocorrelveclg2(vecarray, dt, outfilename=False)`

Parameters:

`vecarray`: ndarray
Array of shape N_{vec} (number of vectors), N_{steps} (number of timesteps), N_{dim} (number of dimensions) containing the time evolution of all unit vectors of interest. See functions `get_vecarray`, `get_normal_vecarray` and `norm_vecarray` of the `mdorado.vectors` module for ways to obtain such an array.

`dt`: int or float
Timestep used in `vecarray`.

`outfilename`: str
If specified an xy-file with the name `str(outfilename)` containing t and $R_2(t)$ will be written. If `False` no file will be written. Default is `False`

Returns: `timesteps, allcorrel`: ndarray, ndarray
Returns two arrays, the first containing information about the timestep t and the second

containing the averaged function $R_2(t)$ using the isotropic simplification in equation 8.2.

8.2 Example

First, we need the time evolution of the vector of interest in all cases. The function `get_vecarray` of the `mdorado.vectors` module can be used to compute a time-series of molecular vectors $\overrightarrow{\mathrm{AB}}$ given the `AtomGroups agrp` and `bgrp`. The function `mdorado.vectors.get_normal_vecarray` can be used to compute normal vectors of the plane given by the particles in the `AtomGroups agrp`, `bgrp` and `cgrp`. For both functions the `AtomGroups` have to be of equal length, so that for every element i a vector $\overrightarrow{\mathrm{A}_i\mathrm{B}_i} = \texttt{bgrp}[i] - \texttt{agrp}[i]$ (or normal vector $\overrightarrow{n} = \overrightarrow{\mathrm{A}_i\mathrm{B}_i} \times \overrightarrow{\mathrm{A}_i\mathrm{C}_i}$) can be computed.

```
1  import MDAnalysis
2  from mdorado import vectors as mvec
3  from mdorado import veccor
4  from mdorado.data.datafilenames import water_topology,
   ↪  water_trajectory
5
6  u = MDAnalysis.Universe(water_topology, water_trajectory)
7  ogrp = u.select_atoms("name ow")
8  hgrp = u.select_atoms("name hw")[::2]
9  dt = 0.2
10
11 vectors = mvec.get_vecarray(universe=u, agrp=ogrp, bgrp=hgrp)
12 vectors = mvec.norm_vecarray(vectors)[0]
13
14 ts_aniso, correl_anisolg2 = veccor.correlvec(vectors, refvec=[1,1,1],
   ↪  dt=dt, nlegendre=2, outfilename="111_vec.dat", normed=True)
15 ts_iso, correl_iso = veccor.isocorrelvec(vectors, dt=dt, nlegendre=2,
   ↪  outfilename="slow_isolg2.dat")
16 ts_isolg2, correl_isolg2 = veccor.isocorrelveclg2(vectors, dt=dt,
   ↪  outfilename="fast_isolg2.dat")
```

In our example, we want to investigate the reorientational behavior of the O–H bond vector in water. After initializing the universe (line 5), we create one AtomGroup with all oxygen atoms (line 6, `ogrp`) and another AtomGroup where we choose every second

hydrogen atom in our system, meaning one hydrogen atom per water molecule (line 7, `hgrp`). Using the `get_vecarray` function, we are able to compute an array containing one O–H vector per water molecule for every timestep (line 11). The normalized (line 12) vector array can then be used to compute the correlation function according to equations 8.1 or 8.2.

Without assuming an isotropic phase or when it is desirable to investigate the reorientation in comparison to a fixed external vector, we can use equation 8.1 and employ the `correlvec` function to compute the autocorrelation function of the second Legendre polynomial (nlegendre=2, line 14).The reference vector, here [1,1,1], can be of arbitrary length, as it will be normalized internally. The `normed` keyword ensures, that the resulting correlation function starts at 1 for $t = 0$.

In line 15, we compute the reorientation correlation function using the isotropic simplification (equation 8.2) employing the `isocorrelvec`. The `refvec` keyword is omitted here, as the orientation of the vectors at every time "0" will be used as reference at time t. This also makes normalization unnecessary, as $\cos[\theta(t = 0)] = 1$. As mentioned, although `isocorrelvec` yields a better statistic for $R_i(t)$, it is significantly slower than the `correlvec` function as we can not easily use FFT correlation for every arbitrary Legendre polynomial.

To circumvent this problem, faster algorithms to compute the first and second Legendre polynomials using the isotropic simplification in equation 8.2 are available in MDoRADO via the functions `isocorrelveclg1` and `isocorrelveclg2`, respectively. Line 16 exemplifies the usage of these functions. The `nlegendre` keyword is omitted in both functions, as the type of Legendre polynomial is hard-coded in the respective function.

Bibliography

[1] D. Chandler, *J. Chem. Phys.* **1978**, *68*, 2959.

[2] C. H. Bennett in *Algorithms for Chemical Computations*, Chapter 4, p. 63.

[3] S. Gehrke, M. von Domaros, R. Clark, O. Hollóczki, M. Brehm, T. Welton, A. Luzar, B. Kirchner, *Faraday Discuss.* **2018**, *206*, 219.

[4] S. Gehrke, B. Kirchner, *J. Chem. Eng. Data* **2019**, *65*, 1146.

[5] J. Neumann, D. Paschek, A. Strate, R. Ludwig, *J. Phys. Chem. B* **2021**, *125*, 281.

[6] J. Neumann, R. Ludwig, D. Paschek, *J. Phys. Chem. B* **2021**, *125*, 5132.

[7] A. Luzar, D. Chandler, *Phys. Rev. Lett.* **1996**, *76*, 928.

[8] A. Luzar, D. Chandler, *Nature* **1996**, *379*, 55.

[9] A. Luzar, *J. Chem. Phys.* **2000**, *113*, 10663.

[10] J. Busch, J. Neumann, D. Paschek, *J. Chem. Phys.* **2021**, *154*, 214501.

[11] R. Kumar, J. R. Schmidt, J. L. Skinner, *J. Chem. Phys.* **2007**, *126*, 204107.

[12] D. Trzesniak, A.-P. E. Kunz, W. F. van Gunsteren, *ChemPhysChem* **2007**, *8*, 162.

[13] Calandrini, V., Pellegrini, E., Calligari, P., Hinsen, K., Kneller, G.R., *JDN* **2011**, *12*, 201.