lipyphilic

Release 0.9.0

Paul Smith

Sep 16, 2021

CONTENTS

A Python toolkit for the analysis of lipid membrane simulations

lipyphilic is free software licensed under the GNU General Public License v2 or later (GPLv2+)

CHAPTER

ONE

OVERVIEW

lipyphilic is a set of tools for analysing MD simulations of lipid bilayers. It is an object-oriented Python package built directly on top of MDAnalysis, and makes use of NumPy and SciPy for efficient computation. The analysis classes are designed with the same interface as those of MDAnalysis - so if you know how to use analysis modules in MDAnalysis then learning **lipyphilic** will be a breeze.

Analysis tools in **lipyphilic** include: identifying sterol flip-flop events, calculating domain registration over time, and calculating local lipid compositions. **lipyphilic** also has three on-the-fly trajectory transformations to i) fix membranes split across periodic boundaries and ii) perform nojump coordinate unwrapping and iii) convert triclinic coordinates to their orthorhombic representation.

These tools position **lipyphilic** as complementary to, rather than competing against, existing membrane analysis software such as MemSurfer and FatSlim.

CHAPTER

TWO

INTERACTIVE TUTORIALS

We recommend new users take a look out our interactive tutorials. These will show you how to get the most out of **lipyphilic**

CHAPTER

THREE

BASIC USAGE

Alternatively, check out the Basic Usage example to see how to use **lipyphilic**, and see the Analysis tools section for detailed information and examples on each tool.

CHAPTER

FOUR

INSTALLATION

The easiest way to install **lipyphilic** along with its dependencies is through Conda:

```
conda config --add channels conda-forge
conda install lipyphilic
```

See the installation guide for futher information.

CHAPTER

FIVE

CITING

If you use **lipyphilic** in your research, please cite our paper:

```
@article{LiPyphilic2021,
    author = {Smith, Paul and Lorenz, Christian D.},
    title = {LiPyphilic: A Python Toolkit for the Analysis of Lipid Membrane Simulations}
,
    journal = {Journal of Chemical Theory and Computation},
    year = {2021},
    volume = {17},
    number = {9},
    pages = {5907-5919},
    doi = {10.1021/acs.jctc.1c00447}
}
```

Please also cite MDAnalysis, on which **lipyphilic** is built. If you use the Area Per Lipid tool please also cite Freud.

CHAPTER

SIX

CONTENTS

6.1 Installation

6.1.1 Conda

The easiest way to install **lipyphilic** is through the conda-forge channel of Conda:

```
conda config --add channels conda-forge
conda install lipyphilic
```

This will install **lipyphilic** along with all of its dependencies.

If you do not already have Conda installed on your machine, we recommend downloading and installing Miniconda — a lightweight version of Conda.

6.1.2 PyPI

It's also possible to install **lipyphilic** from the Python Package Index. If you already have the necessary dependencies installed, you can use pip to install **lipyphilic**:

```
pip install lipyphilic
```

Alternatively, you can also install the in-development version with:

```
pip install https://github.com/p-j-smith/lipyphilic/archive/master.zip
```

6.1.3 Dependencies

lipyphilic uses MDAnalysis to carry out all analysis calculations, and Freud for performing Voronoi tessellations.

As mentioned above, the simplest way to install these packages, along with **lipyphilic**, is with Conda. However, it is also possible to install MDAnalysis and Freud using pip, or from source. See the MDAnalysis and Freud installation instructions for further information.

6.2 Basic Usage

The analysis tools in **lipyphilic** all require an MDAnalysis Universe as input, so to use **lipyphilic** you will also need to import MDAnalysis. The analyses are then performed in the same way as the majority of those in MDAnalysis. For example, to assign each lipid to the upper or lower leaflet at each frame in a trajectory:

```
import MDAnalysis as mda
from lipyphilic.lib.assign_leaflets import AssignLeaflets

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

# Find which leaflet each lipid is in at each frame
leaflets = AssignLeaflets(
    universe=u,
    lipid_sel="name PO4 ROH"  # Select headgroup beads in the MARTINI forcefield
)

# Select which frames to use and perform the analysis
leaflets.run(start=None, stop=None, step=None)  # this will use every frame in the␣
↪trajectory
```

And the results will be available as a NumPy array stored in the *leaflets.leaflets* attribute.

For more details on how to use **lipyphilic**, check out our *interactive tutorials*.

6.3 Interactive tutorials

To help you get the most out of **lipyphilic**, we have created a set of interactive tutorials in the form of Jupyter Notebooks. There is no need to download or install anything, simply click the link below:

We currently have tutorials on the following topics:

1. **Basic usage**: Illustrates basic usage of **lipyphilic**, including how to store results for later usage. Also shows how to assign lipids to leaflets, which is required for many other analyses.
2. **Flip-flop rate**: Shows how to use **lipyphilic** to calculate the rate of cholesterol flip-flop, as well as identify the frames at which each flip-flop event begins and ends.
3. **Local lipid environments**: Illustrates how to determine the local lipid environment of each lipid over time, as well as the enrichment/depletion index.
4. **Lipid domains**: Shows how to calculate the largest cluster of specific lipids over time. Examples include finding the largest ganglioside cluster in a neuronal plasma membrane and identifying the largest domain of L_o lipids in a phase separated membrane.
5. **Interleaflet registration**: This notebook shows how to calculate the interleaflet registration over time. The example shows how to calculate the registration of L_o lipids across leaflets.
6. **Lateral diffusion**: Illustrates how to perform "nojump" trajectory unwrapping with LiPyphilic, then use the unwrapped coordinates to calculate the mean-squared displacement and lateral diffusion coefficient of lipids in a membrane.

7. **Coarse-grained lipid order parameter**: Shows how to calculate the coarse-grained order parameter, and how to create a two-dimensional projection of these values onto the membrane plane.
8. **Projection plots**: Shows how to create two-dimensional projections of arbitrary lipid properties onto the membrane plane. Examples include projecting local membrane thicknesses calculated using FATSLiM onto the membrane plane, and projecting the ordered state (L_o and L_d) of lipids onto the membrane plane.
9. **Potential of mean force (PMF)**: This notebook illustrates how to use **lipyphilic** to calculate the height and orientation of sterols in a membrane, and subsequently plot the two-dimensional PMF of sterol height and orientation.
10. **Hidden Markov Models (HMM)**: Learn how to use the output of **lipyphilic** to construct HMMs with HMMLearn. We will create a HMM based on lipid thicknesses to detect L_o and L_d lipids in a phase separated membrane. The output from this is can be used as input to other analyses in **lipyphilic**, such as calculating interleaflet registration or local lipid environments.

6.4 Overview of analysis tools

Here we provide a brief description of the analysis tools currently available in **lipyphilic**. For more information on each analysis tool, including details of all optional input parameters see the *API*. To learn more about how to use **lipyphilic**, check out our *interactive tutorials*.

6.4.1 Assign leaflets: `lipyphilic.lib.assign_leaflets`

This module provides methods for assigning lipids to leaflets in a bilayer. Leaflet assignment is based on the distance in z from a lipid the midpoint of the bilayer. Lipids may be assigned to the upper leaflet (indicated by *1*), the lower leaflet (*-1*) or the bilayer midplane (*0*).

Below we see how to assign lipids to the upper or lower leaflet of a MARTINI bilayer:

```
import MDAnalysis as mda
from lipyphilic.lib.assign_leaflets import AssignLeaflets

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

# Find which leaflet each lipid is in at each frame
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name PO4 ROH"
)

# Select which frames to use and perform the analysis
leaflets.run(start=None, stop=None, step=None)  # this will use every frame in the␣
↪trajectory
```

The results are stored as a NumPy array of shape (n_lipids, n_frames) in the `leaflets.leaflets` attribute.

If you have used a different force field, you simply need to change the `lipid_sel` to select the relevant headgroup atoms of your lipids. See the MDAnalysis selection language for more info on how to select atoms.

By default, lipids are only allowed to be in the upper (*1*) or lower (*-1*) leaflet. See *lipyphilic.lib.assign_leaflets* for more information on selecting which molecules are allowed in the midplane.

Note: Assignment of lipids to leaflets is not in itself useful, but it is required in order to calculate, for example, area per lipid, interleaflet correlations, and flip-flop rates.

6.4.2 Flip-flop: `lipyphilic.lib.flip_flop`

This module provides methods for detecting the flip-flop of molecules in a lipid bilayer. A flip-flop occurs when a molecule - typically a sterol - moves from one leaflet of a bilayer into the opposing leaflet.

To find all flip-flop events, we first should assign lipids to leaflets as seen in the above example, then:

```
import MDAnalysis as mda
from lipyphilic.lib.flip_flop import FlipFlop

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

flip_flops = FlipFlop(
  universe=u,
  lipid_sel="name ROH",  # select molecules that may flip-flop
  leaflets=leaflets.filter_leaflets("name ROH")
)

flip_flops.run(start=None, stop=None, step=None)
```

The results are stored as a NumPy array of shape (n_flip_flops, 4) in the `flip_flops.flip_flops` attribute. Each row is a single flip-flop event, and the four columns correspond to: the residue index of the flip-flopping molecule; the frame at which the molecule left its original leaflet; the frame at which it entered its new leaflet; the leaflet ID to which it moves.

See *lipyphilic.lib.flip_flop* for more information on how flip-flop is detected and options such as specifying how long a molecule must residue in the new leaflet for the flip-flop to be considered successful.

6.4.3 Interlealet registration: `lipyphilic.lib.registration`

This module provides methods for determining registration of leaflets in a bilayer. Registration is defined by the pearson correlation coefficient of molecular densities in the two leaflets. This is an implementation of the method described by Thallmair et al. (2018).

To calculate the interleaflet correlation of cholesterol, we first need to calculate which leaflet each lipid is in at each frame using *lipyphilic.lib.assign_leaflets.AssignLeaflets*. Then we pass atom selections for which density correlations will be calculated, along with the relevant leaflet membership data, to `Registration`:

```
import MDAnalysis as mda
from lipyphilic.lib.registration import Registration

# Load an MDAnalysis Universe
```

(continues on next page)

(continued from previous page)

```
u = mda.Universe('production.tpr','production.xtc')

registration = Registration(
  upper_sel="resname CHOL and name ROH",
  lower_sel="resname CHOL and name ROH",
  leaflets=leaflets.filter_leaflets("name ROH")
)

registration.run(start=None, stop=None, step=None)
```

The results are stored in a NumPy array of shape (n_frames), containing the pearson correlation coefficient of cholesterol densities in the two leaflets. The data are accessible via the `registration.registration` attribute.

As well as calculating registration of lipid species across the two leaflets, it is also possible to calculate the registration of arbitrary user-defined values across the two leaflets. For example, if you have created a Hidden Markov Model to assign lipids to the Ld or Lo phase, you can calculate the registration of Lo lipids across the two leaflets. See *lipyphilic.lib.registration* for more details.

6.4.4 Neighbours: `lipyphilic.lib.neighbours`

This module provides methods for finding neighbouring lipids in a bilayer. Lipids are neighbours if they are within a user-defined cutoff of one another.

Below we see how to find all neighbours in a MARTINI bilayer based on the 'GL1' and 'GL2' beads of phospholipids and the 'ROH' bead of sterols, using a cutoff of *12* Å:

```
import MDAnalysis as mda
from lipyphilic.lib.neighbours import Neighbours

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

# Find neighbouring lipids
neighbours = Neighbours(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  cutoff=12.0
)

neighbours.run(start=None, stop=None, step=None)
```

The results are stored in the `neighbours.neighbours` attribute as a NumPy array of SciPy sparse matrices (of type `scipy.sparse.csc_matrix`). Each sparse matrix contains the lipid neighbours at a given frame.

Tip: Once the neighbour matrices has been generated, the local lipid compositions or the largest lipids cluster at each frame can be readily.

See *lipyphilic.lib.neighbours* for more information on this module, including how to calculate local lipid compositions or the lipid enrichment/depletion index, and how to find the largest cluster of a given lipid species over time.

6.4.5 Area per lipid: `lipyphilic.lib.area_per_lipid`

This module provides methods for calculating the area per lipid. Areas are calculated via a 2D Voronoi tessellation, using the *locality* module of Freud to perform the tessellation of atomic positions. See Lukat et al. (2013) a thorough description of calculating the area per lipid via Voronoi tessellations.

Once lipids have been assigned to leaflets, the area per lipid can be calculated as follows:

```
import MDAnalysis as mda
from lipyphilic.lib.area_per_lipid import AreaPerLipid

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

areas = AreaPerLipid(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",  # assuming we're using the MARTINI forcefield
  leaflets=leaflets.leaflets
)

areas.run(start=None, stop=None, step=None)
```

The above will use GL1 and GL2 beads to calculate the area of each phospholipid, and the ROH bead to calculate the area of each sterol.

For a more complete description of calculating the area per lipid, and the API of the analysis class, see *lipyphilic.lib.area_per_lipid*.

6.4.6 Lipid order parameter — `lipyphilic.lib.order_parameter`

This module provides methods for calculating the coarse-grained orientational order parameter of acyl tails in a lipid bilayer. The coarse-grained order parameter, S_{CC}, is a measure of the degree of ordering of an acyl tail, based on the extent to which the vector connecting two consecutive tail beads is aligned with the membrane normal.

See Seo et al. (2020) for a definition of S_{CC} and Piggot et al. (2017) for an excellent discussion on acyl tail order parameters in molecular dynamics simulations.

To calculate S_{CC}, we need to provide an atom selection for the beads in a **single** tail of lipids in the bilayer — that is, **either** the *sn1* or *sn2* tails, not both. If we have performed a MARTINI simulation, we can calculate the S_{CC} of all *sn1* tails of phospholipids as follows:

```
import MDAnalysis as mda
from lipyphilic.lib.order_parameter import SCC

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

scc = SCC(
  universe=u,
  tail_sel="name ??A"
)
```

The above makes use of the powerful MDAnalysis selection language. It will select beads such as *C1A*, *C2A*, *D2A* etc. This makes it simple to quickly calculate S_{CC} for the *sn1* tails of all species in a bilayer.

To see how to calculate S_{CC} using local membrane normals to define the molecular axes, as well as the full API of the class, see *lipyphilic.lib.order_parameter*.

6.4.7 Lipid z angles: `lipyphilic.lib.z_angles`

This module provides methods for calculating the angle lipids make with the positive z axis. If we define the orientation of MARTINI cholesterol as the angle between the z-axis and the vector from the the 'R5' bead to the 'ROH' bead, we can calculate the orientation of each cholesterol molecule as follows:

```
import MDAnalysis as mda
from lipyphilic.lib.z_angles import ZAngles

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

z_angles = ZAngles(
  universe=u,
  atom_A_sel="name R5",
  atom_B_sel="name ROH"
)

z_angles.run(start=None, stop=None, step=None)
```

The results are stored in a `numpy.ndarray` of shape (n_residues, n_lipids) in the `z_angles.z_angles` attribute.

For more information on this module, including how to return the angles in radians rather than degrees, see *lipyphilic.lib.z_angles*.

6.4.8 Lipid z positions: `lipyphilic.lib.z_positions`

This module provides methods for calculating the height in z of lipids from the bilayer center.

If we have used the MARTINI forcefield to study a phospholipid/cholesterol mixture, we can calculate the height of cholesterol in the bilayer as follows:

```
import MDAnalysis as mda
from lipyphilic.lib.z_positions import ZPositions

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

z_positions = ZPositions(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  height_sel="name ROH",
  n_bins=10
)

z_positions.run(start=None, stop=None, step=None)
```

`lipid_sel` is an atom selection that covers all lipids in the bilayer. This is used for calculating the membrane midpoint. `height_sel` selects which atoms to use for calculating the height of each lipid.

Local membrane midpoints are calculated by creating a grid of membrane patches, with the number of grid points controlled with the `n_bins` parameter. The distance in z of each lipid to its local midpoint is then calculated.

Data are returned in a `numpy.ndarray` of shape (n_residues, n_frames). See *lipyphilic.lib.z_positions* for more information on this module including the full API of the class.

6.4.9 Lipid z thickness: `lipyphilic.lib.z_thickness`

This module provides methods for calculating the thickness, in z, of lipid tails. This is defined as the maximum distance in z between to atoms in a tail.

If we have used the MARTINI forcefield to study a DPPC/DOPC/cholesterol mixture, we can calculate the thickness of DPPC and DOPC *sn1* tails, as well as the thickness of cholesterol, as follows:

```
import MDAnalysis as mda
from lipyphilic.lib.z_positions import ZThickness

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

z_thickness = ZThickness(
  universe=u,
  lipid_sel="(name ??1 ??A) or (resname CHOL and not name ROH)"
)

z_thickness.run()
```

The above makes use of the powerful MDAnalysis atom selection language to select the DPPC and DOPC sn1 tails along with cholesterol.

The thickness data are stored in a `numpy.ndarray` of shape (n_residues, n_frames) in the `z_thickness.z_thickness` attribute. See *lipyphilic.lib.z_thickness* for the full API of the class.

6.4.10 Membrane z thickness: `lipyphilic.lib.memb_thickness`

This module provides methods for calculating the bilayer thickness. It is defined as the peak-to-peak distance of lipid headgroup density in z.

Lipids must first be assigned to the upper and lower leaflets. This can be done with the class *lipyphilic.lib.assign_leaflets.AssignLeaflets*. Then, to calculate the membrane thickness we need to define which atoms to treat as headgroup atoms and pass the leaflet membership information to `MembThickness`. If we have studied a DPPC/DOPC/cholesterol mixture with MARTINI, we could calculate the membrane thickness as follows:

```
import MDAnalysis as mda
from lipyphilic.lib.z_positions import ZThickness

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

memb_thickness = MembThickness(
  universe=u,
  leaflets=leaflets.filter_leaflets("resname DOPC and DPPC"),  # exclude cholesterol␣
↪from thickness calculation
  lipid_sel="resname DPPC DOPC and name PO4"
```

(continues on next page)

(continued from previous page)

```
)

memb_thickness.run()
```

The results are then available in the `memb_thickness.memb_thickness` attribute as a `numpy.ndarray`.

For more information on calculating membrane thickness, including options to calculating local membrane thicknesses rather than a single global thickness, see *lipyphilic.lib.memb_thickness*.

6.4.11 Lateral diffusion `lipyphilic.lib.lateral_diffusion`

This module contains methods for calculating the mean squared displacement (MSD) and lateral diffusion coefficient, D_{xy},of lipids in a bilayer.

The MSD of all lipids in a DPPC/DOPC/cholesterol MARTINI bilayer can be calculated using *lipyphilic.lib.lateral_diffusion.MSD*:

```
import MDAnalysis as mda
from lipyphilic.lib.lateral_diffusion import MSD

# Load an MDAnalysis Universe
u = mda.Universe('production.tpr','production.xtc')

msd = MSD(
  universe=u,
  lipid_sel="name PO4 ROH"
)

 msd.run()
```

The MSD of each lipid is then available in the `msd.msd` attribute as a `numpy.ndarray`, and the lagtimes are stored in the `msd.lagtimes` attribute.

For more information on this module, including how to calculate the lateral diffusion coefficient, see *lipyphilic.lib.lateral_diffusion*.

6.4.12 Plotting utilities: `lipyphilic.lib.plotting`

lipyphilic can produce joint probability density plots (or PMFs if a temperature is provided), as well as density maps of membrane properties projected onto the membrane plane. The former may be used to plot, for example, the PMF of cholesterol orientation and height in a bilayer. The latter may be used to generate plots of, for example, the area per lipid as a function of xy in the membrane plane.

See *lipyphilic.lib.plotting* for the full API of *lipyphilic.lib.plotting.JointDensity* and *lipyphilic.lib.plotting.ProjectionPlot*.

6.4.13 On-the-fly transformations `lipyphilic.transformations`

lipyphilic contains a module for applying on-the-fly transformation to atomic coordinates while iterating over a trajectory. These are available in the module *lipyphilic.transformations*.

There are three transformations available in *lipyphilic*:

1. *lipyphilic.transformations.nojump*, which prevents atoms from jumping across periodic boundaries. This is useful when calculating the lateral diffusion of lipids.
2. *lipyphilic.transformations.center_membrane*, which can take a membrane that is split across periodic boundaries, make it whole and center it in the box.
3. *lipyphilic.transformations.triclinic_to_orthorhombic*, which transforms triclinic coordinates into their orthorhombic representation.

See *lipyphilic.transformations* for full details on these transformations including how to apply them to your trajectory.

6.5 API

See the following pages for the full API of each tool:

6.5.1 Assign leaflets — `lipyphilic.lib.assign_leaflets`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for assigning lipids to leaflets in a bilayer.

Assigning leaflets in planar bilayers

The class *lipyphilic.lib.assign_leaflets.AssignLeaflets* assigns each lipid to a leaflet based on the distance in z to the midpoint of the bilayer. Lipids may be assigned to the upper leaflet (indicated by *1*), the lower leaflet (*-1*) or the bilayer midplane (*0*).

Input

Required:

- *universe* : an MDAnalysis Universe object
- *lipid_sel* : atom selection for *all* lipids in the bilayer, including e.g. sterols

Options:

- *midplane_sel* : atom selection for lipid that may occupy the midplane
- *midplane_cutoff* : atoms within this distance from the midpoint are considered to be the midplane
- *n_bins* : split the membrane into *n_bins* * *n_bins* patches, and calculate local membrane midpoints for each patch

Output

- *leaflets* : leaflet to which each lipid is assigned at each frame

Leaflet data are returned in a `numpy.ndarray`, where each row corresponds to an individual lipid and each column corresponds to an individual frame, i.e. leaflets[i, j] refers to the leaflet of lipid *i* at frame *j*. The results are accessible via the *AssignLeaflets.leaflets* attribute.

Example usage of `AssignLeaflets`

An MDAnalysis Universe must first be created before using AssignLeaflets:

```
import MDAnalysis as mda
from lipyphilic.lib.assign_leaflets import AssignLeaflets

u = mda.Universe(tpr, trajectory)
```

If we have used the MARTINI forcefield to study a phospholipid/cholesterol mixture, we can assign lipids and cholesterol to the upper and lower as follows:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH"
)
```

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
leaflets.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `leaflets.leaflets` attribute as a `numpy.ndarray`. Each row corresponds to an individual lipid and each column to an individual frame, i.e *leaflets.leaflets[i, j]* contains the leaflet membership of lipid *i* at frame *j*. Lipid *i*, at frame *j*, is in the upper leaflet if *leaflets.leaflets[i, j]==1* and in the lower leaflet if *leaflets.leaflets[i, j]==-1*.

Allowing lipids in the midplane

The above example will assign every lipid (including sterols) to either the upper or lower leaflet. To allow cholesterol to be in the midplane, we can provide a `midplane_sel` and `midplane_cutoff` to *AssignLeaflets*:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  midplane_sel="resname CHOL and name ROH C2",
  midplane_cutoff=12.0
)
```

A cholesterol molecule that has both its *ROH* and *C2* atoms within *12* Å of membrane midpoint will be assigned to the midplane, i.e. for cholesterol *i* at frame *j* that is in the midplane, *leaflets.leaflets[i, j]==0*.

Changing the resolution of the membrane grid

The first two examples compute a global membrane midpoint based on all the atoms of the lipids in the membrane. Lipids are then assigned a leaflet based on their distance in z to this midpoint. This is okay for planar bilayers, but can lead to incorrect leaflet classification in membranes with undulations. If your bilayer has undulations, *AssignLeaflets* can account for this by creating a grid in xy of your membrane, calculating the local membrane midpoint in each patch, then assigning leaflet membership based on distance in z to the local membrane midpoint. This is done through use of *n_bins*:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  midplane_sel="resname CHOL and name ROH C2",
  midplane_cutoff=12.0,
  n_bins=10
)
```

In this example, the membrane will be split into a *10 x 10* grid and a lipid assigned a leaflet based on the distance to the midpoint of the patch the lipid is in.

Assigning leaflets in membranes with high curvature

If your membrane is a vesicle or bilayer with *very* large undulations, such as in a buckled membrane, *lipyphilic.lib.assign_leaflets.AssignLeaflets* will assign lipids to the wrong leaflet

The class *lipyphilic.lib.assign_leaflets.AssignCurvedLeaflets* can be used in these scenaries to assign each lipid to a leaflet using MDAnalysis' Leaflet Finder. Lipids may still be assigned to the upper/outer leaflet (indicated by *1*), the lower/inner leaflet (*-1*) or the membrane midplane (*0*).

Input

Required:

- *universe* : an MDAnalysis Universe object
- *lipid_sel* : atom selection for *all* lipids in the bilayer, including e.g. sterols
- lf_cutoff : distance cutoff below which two neighbouring atoms will be considered to be in the same leaflet.

Options:

- *midplane_sel* : atom selection for lipid that may occupy the midplane
- *midplane_cutoff* : atoms further than this distance from the either leaflet are considered to be the midplane
- pbc : bool, specifying whether or not to take periodic boundaries into account

Output

- *leaflets* : leaflet to which each lipid is assigned at each frame

Leaflet data are returned in a `numpy.ndarray`, where each row corresponds to an individual lipid and each column corresponds to an individual frame, i.e. leaflets[i, j] refers to the leaflet of lipid *i* at frame *j*. The results are accessible via the *AssignLeaflets.leaflets* attribute.

Example usage of `AssignCurvedLeaflets`

An MDAnalysis Universe must first be created before using AssignCurvedLeaflets:

```
import MDAnalysis as mda
from lipyphilic.lib.assign_leaflets import AssignLeaflets

u = mda.Universe(tpr, trajectory)
```

If we have used the MARTINI forcefield to study a phospholipid/cholesterol mixture, we can assign lipids and cholesterol to the upper and lower as follows:

```
leaflets = AssignCurvedLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  lf_cutoff=12.0,
  midplane_sel="name ROH",
  midplane_cutoff=10.0
)
```

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
leaflets.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

This will first use MDAnalysis' Leaflet Finder to assign all lipids, excluding those in `midplane_sel`, to either the upper or lower leaflet. The `LeafletFinder` will consider two lipids to be in the same leaflet if they have *GL1* or *GL2* atoms within 12 Å of one another. From this, we find the two largest leaflets, then assign the remaining phospholipids to a leaflet based on whichever leaflet they are closest to.

The phospholipids do not change leaflets throughtout the trajectory, only cholesterol — as specified with `midplane_sel` and `midplane_cutoff`. Thus, at each frame, each cholesterol is assinged a leaflet based on it's minimum distance to the leaflet. In the above example, if a cholesterol is within 10 Å of one leaflet it is assigned to that leaflet. If it is within 10 Å of *neither* or *both* leaflets then it is assigned to the midplane.

The results are then available in the `leaflets.leaflets` attribute as a `numpy.ndarray`. Each row corresponds to an individual lipid and each column to an individual frame, i.e *leaflets.leaflets[i, j]* contains the leaflet membership of lipid *i* at frame *j*. Lipid *i*, at frame *j*, is in the upper leaflet if *leaflets.leaflets[i, j]==1* and in the lower leaflet if *leaflets.leaflets[i, j]==-1*.

The classes and their methods

class lipyphilic.lib.assign_leaflets.**AssignLeaflets**(*universe*, *lipid_sel*, *midplane_sel=None*, *midplane_cutoff=None*, *n_bins=1*)

Assign lipids in a bilayer to the upper leaflet, lower leaflet, or midplane.

Set up parameters for assigning lipids to a leaflet.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for the lipids in a membrane. The selection should cover **all** residues in the membrane, including cholesterol.
- **midplane_sel** (*str, optional*) – Selection string for residues that may be midplane. Any residues not in this selection will be assigned to a leaflet regardless of its proximity to the midplane. The default is *None*, in which case all lipids will be assigned to either the upper or lower leaflet.
- **midplane_cutoff** (*float, optional*) – Minimum distance in z an atom must be from the midplane to be assigned to a leaflet rather than the midplane. The default is *0*, in which case all lipids will be assigned to either the upper or lower leaflet. Must be non-negative.
- **n_bins** (*int, optional*) – Number of bins in x and y to use to create a grid of membrane patches. Local membrane midpoints are computed for each patch, and lipids assigned a leaflet based on the distance to their local membrane midpoint. The default is *1*, which is equivalent to computing a single global midpoint.

Note: Typically, `midplane_sel` should select only sterols. Other lipids have flip-flop rates that are currently unaccessible with MD simulations, and thus should always occupy either the upper or lower leaflet.

---

class lipyphilic.lib.assign_leaflets.**AssignCurvedLeaflets**(*universe*, *lipid_sel*, *lf_cutoff=15*, *midplane_sel=None*, *midplane_cutoff=None*, *pbc=True*)

Assign lipids in a membrane to the upper leaflet, lower leaflet, or midplane.

Set up parameters for assigning lipids to a leaflet.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for the lipids in a membrane. The selection should cover **all** residues in the membrane, including cholesterol.
- **lf_cutoff** (*float, optional*) – Cutoff to pass to *MDAnalysis.analysis.leaflet.LeafletFinder*. Lipids closer than this cutoff distance apart will be considered to be in the same leaflet. The default is 15.0
- **midplane_sel** (*str, optional*) – Selection string for residues that may be midplane. Any residues not in this selection will be assigned to a leaflet at ever frame. The default is *None*, in which case no molecules will be considered to be in the midplane.
- **midplane_cutoff** (*float, optional*) – Lipids with atoms selected in *midplane_sel* that are within this distance of a leaflet will be to that leaflet. If a molecule is within this distance of *neither* or *both* leaflets, it will be assigned to the midplane. The default is *None*.
- **pbc** (*bool, optional*) – Take periodic boundary conditions into account. The default is *True*.

Note: Typically, `midplane_sel` should select only sterols. Other lipids have flip-flop rates that are currently unaccessible with MD simulations, and thus should always occupy either the upper or lower leaflet.

`filter_leaflets`(*lipid_sel=None*, *start=None*, *stop=None*, *step=None*)

Create a subset of the leaflets results array.

Filter either by lipid species or by the trajectory frames, or both.

Parameters

- **lipid_sel** (*str, optional*) – MDAnalysis selection string that will be used to select a subset of lipids present in the leaflets results array. The default is *None*, in which case data for all lipids will be returned.
- **start** (*int, optional*) – Start frame for filtering. The default is *None*, in which case the first frame is used as the start.
- **stop** (*int, optional*) – Stop frame for filtering. The default is *None*, in which case the final frame is used as the stop.
- **step** (*int, optional*) – Number of frames to skip when filtering frames. The deafult is *None*, in which case all frames between *start* and *stop* are used.

6.5.2 Flip-flop — `lipyphilic.lib.flip_flop`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for finding flip-flop events in a lipid bilayer.

A flip-flop event occurs when a molecule - typically a sterol - moves from one leaflet of a bilayer into the opposing leaflet.

The class *lipyphilic.lib.flip_flop.FlipFlop* finds the frames at which a flip-flop event begins and ends, as well as the direction of travel (upper-to-lower or lower-to-upper). *FlipFlop* can also determine whether each event was successful (the molecule resides in the opposing leaflet for at least a given length of time), or not (the molecule went to the midplane but returned to its original leaflet).

See Baral et al. (2020) for further discussion on flip-flop in lipid bilayers, including the affect on the flip-flop rate of the buffer size used to assign molecules to the midplane of the bilayer.

Input

Required:

- *universe* : an MDAnalysis Universe object.
- *lipid_sel* : atom selection for atoms to use in detecting flip-flop
- *leaflets* : leaflet membership (-1: lower leaflet, 0: midplane, 1: upper leaflet) of each lipid in the membrane at each frame

Output

- *resindex* : residue index of a flip-flopping molecule
- *flip_flop_start_frame* : final frame at which the molecule was present in its original leaflet
- *flip_flop_end_frame* : first frame at which the molecule is present in the new leaflet
- *moves_to* : direction of travel of the molecule: equal to 1 if the upper leaflet is the new lealet, equal to -1 if the lower leaflet is the new leaflet

Flip-flop data area returned in a `numpy.ndarray`, on a "one line, one observation" basis and can be accessed via `FlipFlop.flip_flops`:

```
flip_flops = [
    [
        <resindex (0-based)>,
        <end_frame (0-based)>,
        <start_frame (0-based)>,
        <moves_to>
    ],
    ...
]
```

moves_to is equal to 1 or -1 if the molecule flip-flops into the upper or the lower leaflet, respectively.

Additionaly, the success or failure of each flip-flop event is stored in the attribute `FlipFlop.flip_flop_success`.

Example usage of `FlipFlop`

An MDAnalysis Universe must first be created before using *FlipFlop*:

```
import MDAnalysis as mda
from lipyphilic.lib.assign_leaflets import AssignLeaflets
from lipyphilic.lib.flip_flop import FlipFlop

u = mda.Universe(tpr, trajectory)
```

Then we need to know which leaflet each lipid is in at each frame. This may be done using *lipyphilic.lib.assign_leaflets.AssignLeaflets*:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH"  # assuming we are using the MARTINI forcefield
  midplane_sel="name ROH",       # only cholesterol is allowed to flip-flop
  midplane_cutoff=8.0,          # buffer size for assigning molecules to the midplane
)
leaflets.run()
```

The leaflet data are stored in the `leaflets.leaflets` attribute. We can now create our *FlipFlop* object:

```
flip_flop = FlipFlop(
  universe=u,
  lipid_sel="name ROH",
  leaflets=leaflets.filter_leaflets("name ROH")  # pass only the relevant leaflet data
)
```

We then select which frames of the trajectory to analyse (*None* will use every frame):

```
flip_flop.run(
  start=None,
  stop=None,
  step=None
)
```

The results are then available in the `flipflop.flip_flop` attribute as a `numpy.ndarray`. Each row corresponds to an individual flip-flop event, and the four columns correspond, respectively, to the molecule resindex, flip-flop start frame, flip-flop end frame, and the leaflet in which the molecule resides after the flip-flop.

Specify minimum residence time for successful flip-flops

We can also specify the minumum number of frames a molecule must reside in its new leaflet for the flip-flop to be considered successful. We do this using the `frame_cutoff` parameter:

```
flip_flop = FlipFlop(
  universe=u,
  lipid_sel="name ROH",
  leaflets=leaflets.filter_leaflets("name ROH")
  frame_cuotff=10,
)
```

With *frame_cutoff=10*, a molecule must remain in its new leaflet for at least 10 consecutive frames for the flip-flop to be considered successful. If this condition is not met, the flip-flop event is recorded as failing.

Calculating the flip-flop rate

The flip-flop rate can be calculatd directly from the number of successfull flip-flop evetns, which itself can be calculated as:

```
n_successful = sum(flip_flop.flip_flop_success == "Success")
```

The rate is then given by the total number of successful flip-flops divided by the total simulations time and the number of molecules of the translocating species.

The class and its methods

class lipyphilic.lib.flip_flop.**FlipFlop**(*universe*, *lipid_sel*, *leaflets*, *frame_cutoff=1*)

Find flip-flop events in a lipid bilayer.

Set up parameters for finding flip-flop events.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for atoms to use in detecting flip-flop.
- **leaflets** (*numpy.ndarray (n_lipids,, n_frames)*) – An array of leaflet membership for each lipid as each frame, in which: -1 corresponds to the lower leaflet; 1 corresponds to the upper leaflet; and 0 corresponds to the midplane.

- **frame_cutoff** (*int, optional*) – To be counted as a successful flip-flop, a molecule must reside in its new leaflet for at least 'frame_cutoff' consecutive frames. The default is *1*, in which case the molecule only needs to move to the opposing leaflet for a single frame for the flip-flop to be successful.

Tip: Leaflet membership can be determined using `lipyphilic.lib.assign_leaflets.AssignLeaflets`.

6.5.3 Registration — `lipyphilic.lib.registration`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for determining registration of leaflets in a bilayer.

The degree of registration is calculated as the pearson correlation coefficient of densities in the upper and lower leaflets. First, the 2D density of each leaflet, L, is calculated:

$$\rho(x,y)_L = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{x'-x}{\sigma}\right)^2\right) dx'dy'$$

where the (x, y) positions of lipid atoms in leaflet L are binned into two-dimensional histograms, then convolved with a circular Gaussian density of standard deviation σ. L is either the upper (u) or lower (l) leaflet.

The correlation between the two leaflets, $r_{u/l}$, is then calculated as the pearson correlation coefficient between $\rho(x,y)_u$ and $\rho(x,y)_l$, where values of:

- 1 correspond to perfectly registered
- -1 correspond to perfectly anti-registered

For more information on interleaflet registration in bilayers see Thallmair et al. (2018).

Input

Required:

- *universe* : an MDAnalysis Universe object.
- *upper_sel* : atom selection for lipids in the upper leaflet to use in the registration calculation
- *lower_sel* : atom selection for lipids in the lower leaflet to use in the registration calculation
- *leaflets* : leaflet membership (-1: lower leaflet, 0: midplane, 1: upper leaflet) of each lipid in the membrane.

Optional:

- *filter_by* : boolean mask for determining which lipids to include in the registration calculation
- *n_bins* : the number of bins to use in *x* and *y* for the 2D histogram
- *gaussian_sd* : the standard deviation of the circular Gaussian to convole with the grid densities

Output

- *registration* : the degree of interleaglet registration at each frame

The data are stored in the `registration.registration` attribute, containing the pearson correlation coefficient of the two-dimensional leaflet densities at each frame.

Example usage of `Registration`

An MDAnalysis Universe must first be created before using *Registration*:

```
import MDAnalysis as mda
from lipyphilic.lib.registration import Registration

u = mda.Universe(tpr, trajectory)
```

Then we need to know which leaflet each lipid is in at each frame. This may be done using *lipyphilic.lib.assign_leaflets.AssignLeaflets*:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH" # assuming we are using the MARTINI forcefield
)
leaflets.run()
```

The leaflets data are stored in the `leaflets.leaflets` attribute. We can now create our Registration object by passing our *lipyphilic.lib.assign_leaflets.AssignLeaflets* object to *Registration* along with atom selections for the lipids:

```
registration = Registration(
  upper_sel="resname CHOL and name ROH",
  lower_sel="resname CHOL and name ROH",
  leaflets=leaflets.filter_leaflets("resname CHOL and name ROH")
)
```

To calculate the interleaflet correlation of cholesterol molecules using their ROH beads we then need to use the `run()` method. We select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
registration.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `registration.registration` attribute as a `numpy.ndarray`. Again, 1 corresponds to the leaflets being perfectly in register and -1 corresponds to the leaflets being perfectly anti-registered.

Selecting a subset of lipids for the registration analysis

The previous example will compute the registration of cholesterol across the upper and lower leaflets. In, for example, simulations of phase-separation domains, it is useful to know the registration of liquid-ordered domains (regardless of the species in the domain) rather than the registrtion of specific lipid species.

If we have a 2D array, 'lipid_order_data', that contains information on whether each lipid is in the liquid-disordered phase or the liquid-ordered phase at each frame, we can used this to calculate the registration of ordered domains. The array must take the shape '(n_residues, n_frames)', and in the below example 'lipid_order_data[i, j]' would be equal to *-1* if lipid 'i' is liquid-disordered at frame 'j' and equal to *1* if it is liquid-ordered:

```
registration = Registration(
  upper_sel="name PO4 ROH",
  lower_sel="name PO4 ROH",
  leaflets=leaflets.leaflets,
  filter_by=lipid_order_data == 1
)
```

If we have a ternary mixture of DPPC/DOPC/Cholesterol, we can also specifcy that we wish to consider only DPPC and cholesterol in the liquid-ordered phase:

```
registration = Registration(
  upper_sel="(resname CHOL and name ROH) or (resname DPPC and name PO4)",
  lower_sel="(resname CHOL and name ROH) or (resname DPPC and name PO4)",
  leaflets=leaflets.filter_leaflets("resname CHOL DPPC"),
  filter_by=lipid_order_data == 1
)
```

Changing the resolution of the 2D grid

By default, the lipid positions of each leaflet are binned into a two-dimensional histogram using $n\_bins_x = \lceil x \rceil$, where n_bins_x is the numer of bins in x and $\lceil x \rceil$ is the size of system in x rounded up to the nearest integer. This gives a grid resolution of *1* Å.

It is also possible to specify the number of bins to use for binning the data:

```
registration = Registration(
  upper_sel="resname CHOL and name ROH",
  lower_sel="resname CHOL and name ROH",
  leaflets=leaflets.filter_leaflets("resname CHOL"),
  n_bins=100
)
```

This will use *100* bins for creating the two-dimensional histogram. Fewer bins will result in a performance increase but at the cost of spatial resolution. For all but the largest systems, the default of *1* Å is appropriate. If your system is larger than a few hundred nm in one dimension, you will likely want to set `n_bins` to 2000 or less.

Changing the standard deviation of the circular Gaussian density

The defualt value of σ is *15*, which is the value used by Thallmair et al. (2018) for determining interleaflet cholesterol correlations. This deault value can be changed using the `gaussian_sd` parameter:

```
registration = Registration(
  upper_sel="resname CHOL and name ROH",
  lower_sel="resname CHOL and name ROH",
  leaflets=leaflets.filter_leaflets("resname CHOL"),
  gaussian_sd=12
)
```

Figure 2d of Thallmair et al. (2018) shows how correlation tends to increase with increasing `gaussian_sd`. This is because the density of atomic positions is more diffuse and thus more likely to overlap between the two leaflets. Increasing `gaussian_sd` also incurs a performance cost.

The class and its methods

class lipyphilic.lib.registration.**Registration**(*universe*, *upper_sel*, *lower_sel*, *leaflets*, *filter_by=None*, *n_bins=None*, *gaussian_sd=15*)

Calculate interleaflet registration in a bilayer.

Set up parameters for the registration calculation.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **upper_sel** (*str*) – Selection string for lipids in the upper leaflet of the bilayer to be used for determining registration.
- **lower_sel** (*str*) – Selection string for lipids in the lower leaflet of the bilayer to be used for determining registration.
- **leaflets** (*numpy.ndarray*) – An array of leaflet membership in which: -1 corresponds to the lower leaflet; 1 corresponds to the upper leaflet; and 0 corresponds to the midplane. If the array is 1D and of shape (n_lipids), each lipid is taken to remain in the same leaflet over the trajectory. If the array is 2D and of shape (n_lipids, n_frames), the leaflet to which each lipid is assisgned at each frame will be taken into account when calculating the area per lipid.
- **filter_by** (*numpy.ndarray, optional*) – A boolean array indicating whether or not to include each lipid in the registration analysis. If the array is 1D and of shape (n_lipids), the same lipids will be used in the registration analysis at every frame. If the array is 2D and of shape (n_lipids, n_frames), the boolean value of each lipid at each frame will be taken into account. The default is *None*, in which case no filtering is performed.
- **n_bins** (*int, optional*) – The number of bins to use in each dimension for the two-dimensional density calculations. The default is *None*, in which case the number of bins will be given by the size of the system in the 'x' dimension rounded up to the nearest integer.
- **gaussian_sd** (*float, optional*) – The standard deviation of the circular Gaussian density to convolve with the two-dimensional densities. The spreads out the data to better represent the size of the lipids. The default is 15.

6.5.4 Neighbours — `lipyphilic.lib.neighbours`

Author Paul Smith

Year 2021

This module provides methods for finding neighbouring lipids in a bilayer, calculating local lipid compositions and lipid enrichment, and finding the largest cluster of specific species of lipids over time.

Two lipids are considered neighbours if they have any atoms within a given cutoff distance of one another.

Input

Required:

- *universe* : an MDAnalysis Universe object.
- *lipid_sel* : atom selection for lipids in the bilayer

Optional:

- *cutoff* : lipids are considered to be neighbouring if they have at least one pair of atoms less than this distance apart (in Å)

Output

- *neighbours* : a sparse matrix of binary variables, equal to 1 if two lipids are in contact, and 0 otherwise

For efficient use of memory, an adjacency matrix of neighbouring lipids is stored in a `scipy.sparse.csr_matrix` sparse matrix for each frame of the analysis. The data are stored in the `neighbours.neighbours` attribute as a NumPy array of sparse matrices. Each matrix has shape (n_residues, n_residues)

Tip: The resultant sparse matrix can be used to calculate the local lipid composition of each individual lipid at each frame using `lipyphilic.lib.neighbours.count_neighbours()`, or to find the largest cluster of lipids at each frame using `lipyphilic.lib.neighbours.largest_cluster()`.

Example usage of `Neighbours`

An MDAnalysis Universe must first be created before using *`Neighbours`*:

```
import MDAnalysis as mda
from lipyphilic.lib.neighbours import Neighbours

u = mda.Universe(tpr, trajectory)
```

We can now create our *`Neighbours`* object:

```
neighbours = Neighbours(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",  # assuming we're using the MARTINI forcefield
  cutoff=12.0
)
```

A lipid will be considered to be neighbouring a cholesterol molecule if either its *GL1* or *GL2* bead is within *12* Å of the ROH bead of the cholesterol. For neighbouring lipids, the distances between there respective *GL1* and "GL2* beads will be considered.

We then select which frames of the trajectory to analyse (*None* will use every frame) and select to display a progress bar (*verbose=True*):

```
neighbours.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `neighbours.Neighbours` attribute as a `numpy.ndarray` of Compressed Sparse Row matrices.

Counting the number of neighbours: by lipid species

In order to compute the number of each lipid species around each lipid at each frame, after generating the neighbour matrix we can use the `count_neighbours()` method:

```
counts = neighbours.count_neighbours()
```

Counts is a `pandas.DataFrame` in which each row contains the following information (if there are N distinct species in the membrane):

```
[
    <lipid identifier>,  # by default, the lipid resname
    <lipid resindex>,
    <frame>,
    <num species_1 neighbours>,
    ...
    <num species_N neighbours>,
    <total num neighbours>
]
```

Counting the number of neighbours: by user-defined labels

Instead of using the lipid resname to identify neighbouring lipids, any ordinal data may be used for counting lipid neighbours. This is done through use of the `count_by` and `count_by_labels` parameters:

```
counts = neighbours.count_neighbours(
  count_by=lipid_order_data,
  count_by_labels={'Ld': 0, 'Lo': 1}
)
```

Here we assume that 'lipid_order_data' contains information on whether each lipid is in the liquid-disordered phase or the liquid-ordered phase at each frame. It must take the shape '(n_residues, n_frames)', and in this example 'lipid_order_data[i, j]' would be equal to '0' if lipid 'i' is liquid-disordered at frame 'j' and equal to '1' if it is liquid-ordered. 'count_by_labels' is used to signify that the value '0' corresponds to the liquid-disordered (Ld) phase and the value '1' to the liquid-ordered (Lo) phase. In this example, the returned `pandas.DataFrame` would contain the following information in each row:

```
[
    <Ld or Lo>,
    <lipid resindex>,
    <frame>,
    <num Ld neighbours>,
    <num Lo neighbours>,
    <total num neighbours>
]
```

Calculate the enrichment index of lipid species

The `count_neighbours()` method will, by default, return the number of neighbouring lipids around each individual lipid.

However, a clearer picture of aggregation of certain lipid species can be gained by instead considering the enrichment/depletion index of each lipid species, defined in Ingólfsson et al. (2014). In this instance, the number of each neighbour species B around a given reference species A is normalized by the average number of species B around any lipid.

To calculate the enrichment/depletion index of each species at each frame, as well as the raw neighbour counts, we can set the `return_enrichment` keyword to true:

```
counts, enrichment = neighbours.count_neighbours(return_enrichment=True)
```

This will return two `pandas` `DataFrames`, one containing the neighbour counts and the other the enrichment/depletion index of each species at each frame. The benefit of having the enrichment index at each frame is that you can plot its time-evolution to see whether particular species form aggregates over time.

Find the largest cluster

To find the largest cluster of a set of lipid species we can use the `largest_cluster()` method:

```
largest_cluster = neighbours.largest_cluster(
  cluster_sel="resname CHOL DPPC"
)
```

The results are returned in a `numpy.ndarray` and contain the number of lipids in the largest cluster at each frame.

Find the largest cluster in a given leaflet

The previous example will compute the largest cluster formed by cholesterol and DPPC molecules at each frame. In large coarse-grained systems where there is substantial flip-flop of sterols, this cluster may span both leaflets. In order to find the largest cluster at each frame within a given leaflet, we can tell `largest_cluster()` to consider only lipids in the upper leaflet by using the *filter_by* parameter.

First, though, we need to know which leaflet each lipid is in at each frame. This may be done using *lipyphilic.lib.assign_leaflets.AssignLeaflets*:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH"  # pass the same selection that was passed to Neighbours
```

(continues on next page)

(continued from previous page)

```
)
leaflets.run()  # run the analysis on the same frames as Neighbours.run()
```

The leaflets data are stored in the `leaflets.leaflets` attribute, will be equal to '1' if the lipid is in the upper leaflet at a given frame and equal to '-1' if it is in the lower leaflet. See *lipyphilic.lib.assign_leaflets.AssignLeaflets* for more information. We can now find the largest cluster over time in the upper (1) leaflet.

The `filter_by` parameter takes as input a 2D `numpy.ndarray` of shape (n_residues, n_frames). The array should be a boolean mask, where *True* indicates that we should include this lipid in the neighbour calculation:

```
upper_leaflet_mask = leaflet.leaflets == 1

largest_cluster_upper_leaflet = neighbours.largest_cluster(
  cluster_sel="resname CHOL DPPC",
  filter_by=upper_leaflet_mask
)
```

Now, lipids either in the lower leaflet (-1) or the midplane (0) will not be included when determining the largest cluster.

Get residue indices of lipids in the largest cluster

If we want to know not just the cluster size but also which lipids are in the largest cluster at each frame, we can set the `return_indices` parameter to *True*:

```
largest_cluster, largest_cluster_indices = neighbours.largest_cluster(
  cluster_sel="resname CHOL DPPC",
  return_indices=True
)
```

The residue indices will be returned as list of *numpy.ndarray* arrays - one per frame of the analysis. Each array contains the residue indices of the lipids in the largest cluster at that frame

The class and its methods

class lipyphilic.lib.neighbours.**Neighbours**(*universe*, *lipid_sel*, *cutoff=10.0*)

Find neighbouring lipids in a bilayer.

Set up parameters for finding neighbouring lipids.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for lipids in the bilayer.
- **cutoff** (*float, optional*) – To be considered neighbours, two lipids must have at least one pair of atoms within this cutoff distance (in Å). The default is *10.0*.

count_neighbours(*count_by=None*, *count_by_labels=None*, *return_enrichment=False*)

Count the number of each neighbour type at each frame.

Parameters

- **count_by** (*numpy.ndarray, optional*) – An array containing ordinal data describing each lipid at each frame. For example, it may be an array containing information on the ordered

state or each lipid. Defaults to None, in which case the lipid species (resnames) are used for counting neighbours.

- **count_by_labels** (*dict, optional*) – A dictionary of labels describing what each unique value in *count_by* refers to, e.g if *count_by* contains information on the ordered state of each lipid at each frame, whereby 0 corresponds to disordered and 1 corresponds to ordered, then *count_by_labels = {'Ld': 0, 'Lo': 1}*. There **must** be precisely one label for each unique value in 'count_by'. If *count_by* is given but *count_by_labels* is left as *None*, the values in *count_by* will be used as the labels.
- **return_enrichment** (*bool, optional*) – If *True*, a second DataFrame containing the fractional enrichment of each lipid species at each frame is also returned. The default is *False*, in which case the fractional enrichment if not returned.

Returns

- **counts** (*pandas.DataFrame*) – A DataFrame containing the following data for each lipid at each frame: lipid identifier (default is resname), lipid residue index, frame number, number of neighbours of each species (or of each type in 'count_by' if this is provided), as well as the total number of neighbours.
- **enrichment** (*pandas.DataFrame*) – A DataFrame containing the following data enrichment/depletion data for each lipid species at each frame.

`largest_cluster`(*cluster_sel=None*, *filter_by=None*, *return_indices=False*)

Find the largest cluster of lipids at each frame.

Parameters

- **cluster_sel** (*str, optional*) – Selection string for lipids to include in the cluster analysis. The default is *None*, in which case all lipid used in identiying neighbouring lipids will be used for finding the largest cluster.
- **filter_by** (*numpy.ndarray, optional*) – A boolean array indicating whether or not to include each lipid in the cluster analysis. If the array is 1D and of shape (n_lipids), the same lipids will be used in the cluster analysis at every frame. If the array is 2D and of shape (n_lipids, n_frames), the boolean value of each lipid at each frame will be taken into account. The default is *None*, in which case all lipids used in identiying neighbours will be used for finding the largest cluster.
- **return_indices** (*bool, optional*) – If *True*, a list of NumPy arrays will also be returned, on for each frame. Each NumPy array will contain the residue indices of the lipids in the largest cluster at that frame. Note, if there are two largest clusters of equal size, only the residue indices of lipids in one cluster will be returned (the cluster that has the lipid with the smallest residue index). The default is *False*, in which case no reidue indices are returned.

Returns

- **largest_cluster** (*numpy.ndarray*) – An array containing the number of lipids in the largest cluster at each frame.
- **indices** (*list*) – A list of 1D NumPy arrays, where each array corresponds to a single frame and contains the residue indices of lipids in the largest cluster at that frame.

Note: Neighbours must be found by using *Neighbours.run()* before calling either *Neighbours.count_neighbours()* or *Neighbours.largest_cluster()*.

6.5.5 Area per lipid — `lipyphilic.lib.area_per_lipid`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for calculating the area per lipid in a bilayer.

The class *lipyphilic.lib.area_per_lipid.AreaPerLipid* calculates the area of each lipid via a 2D Voronoi tessellation of atomic positions. See Lukat et al. (2013) for a description of calculating the area per lipid via Voronoi tessellations.

This class uses Freud for performing the Voronoi tessellations from which the area per lipid is calculated.

Input

Required:

- *universe* : an MDAnalysis Universe object.
- *lipid_sel* : atom selection for lipids in the bilayer. These atoms will be used to perform the Voronoi tessellation.
- *leaflets* : leaflet membership (-1: lower leaflet, 0: midplane, 1: upper leaflet) of each lipid in the membrane.

Output

- *area* : area per lipid of each lipid as each frame

Area data are returned in a `numpy.ndarray`, where each row corresponds to an individual lipid and each column corresponds to an individual frame, i.e. *areas[i, j]* refers to the area of lipid *i* at frame *j*. The results are accessible via the `AreaPerLipid.areas` attribute.

Note: No area can be calculated for molecules that are in the midplane, i.e. those for which *leaflets==0*. These molecules will have *NaN* values in the results array for the frames at which they are in the midplane.

Example usage of `AreaPerLipid`

An MDAnalysis Universe must first be created before using AreaPerLipid:

```
import MDAnalysis as mda
from lipyphilic.lib.assign_leaflets import AssignLeaflets

u = mda.Universe(tpr, trajectory)
```

Then we need to know which leaflet each lipid is in at each frame. This may be done using *lipyphilic.lib.assign_leaflets.AssignLeaflets*:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH" # assuming we are using the MARTINI forcefield
)
leaflets.run()
```

The leaflet data are stored in the `leaflets.leaflets` attribute. We can now create our AreaPerLipid object:

```
areas = AreaPerLipid(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  leaflets=leaflets.leaflets
)
```

The above will use GL1 and GL2 beads to calculate the area of each phospholipid, and the ROH bead to calculate the area of each sterol. Two Voronoi tessellations will be performed at each frame — one for the upper leaflet and one for the lower leaflet.

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
areas.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `areas.areas` attribute as a `numpy.ndarray`. Each row corresponds to an individual lipid and each column to an individual frame, i.e *areas.areas[i, j]* contains the area of lipid *i* at frame *j*.

Warning: If your membrane is highly curved the calculated area per lipid will be inaccurate. In this case we recommend you use either FATSlim, MemSurfer or ML-LPA.

The class and its methods

class lipyphilic.lib.area_per_lipid.**AreaPerLipid**(*universe*, *lipid_sel*, *leaflets*)

Calculate the area of lipids in each leaflet of a bilayer.

Set up parameters for calculating areas.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for lipids in the bilayer. Typically, in all-atom simulations, one atom per sterol and three atoms per non-sterol lipid would be used. In coarse-grained simulations, one bead per sterol and two beads per non-sterol lipid would typically be used.
- **leaflets** (*numpy.ndarray (n_lipids,)*) – An array of leaflet membership in which: -1 corresponds to the lower leaflet; 1 corresponds to the upper leaflet; and 0 corresponds to the midplane. If the array is 1D and of shape (n_lipids), each lipid is taken to remain in the same leaflet over the trajectory. If the array is 2D and of shape (n_lipids, n_frames), the leaflet to which each lipid is assisgned at each frame will be taken into account when calculating the area per lipid.

Tip: Leaflet membership can be determined using *lipyphilic.lib.assign_leaflets.AssignLeaflets*.

project_area(*lipid_sel=None*, *start=None*, *stop=None*, *step=None*, *filter_by=None*, *bins=None*, *ax=None*, *cmap=None*, *vmin=None*, *vmax=None*, *cbar=True*, *cbar_kws={}*, *imshow_kws={}*)

Project the area per lipid onto the xy plane of the membrane.

The areas per lipid, averaged over a selected range of frames, are projected onto the xy plane based on the center of mass of each lipid. The atoms to be used in calculating the center of mass of the lipids can be specified using the *lipid_sel* arugment.

This method creates an instance of *lipyphilic.lib.plotting.ProjectionPlot* with the projected areas interpolated across periodic boundaries. The plot is returned so further modification can be performed if needed.

Note: The lipid positions are taken from the middle frame of the selected range.

Parameters

- **lipid_sel** (*MDAnalysis atom selection, optional*) – The center of mass of each lipid will be determined via this selection. The default is *None*, in which case every atom of a lipid is used to determine its center of mass.
- **start** (*int, optional*) – Start frame for averaging the area per lipid results.
- **stop** (*int, optional*) – Final frame for averaging the area per lipid results.
- **step** (*int, optional*) – Number of frames to skip
- **filter_by** (*array-like, optional*) – A Boolean mask for selecting a subset of lipids. It may take the following shapes:

  `(n_lipids)` The mask is used to select a subset of lipids for projecting the areas onto the membrane plane.

  `(n_lipids, n_frames)` This is the same shape as the NumPy array created by the *lipyphilic.lib.AreaPerLipid.run()* method. Boolean values are used only from the column corresponding to the middle frame of the range selected by *start*, *stop*, and *step*.

  The default is *None*, in which case no filtering is applied.

- **bins** (*int or array_like or [int, int] or [array, array]*) – The bin specification:

  `int` If int, the number of bins for the two dimensions (nx=ny=bins).

  `array-like` If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).

  `[int, int]` If [int, int], the number of bins in each dimension (nx, ny = bins).

  `[array, array]` If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

  `combination` A combination [int, array] or [array, int], where int is the number of bins and array is the bin edges.

  The default is *None*, in which case a grid with 1 x 1 Angstrom resolution is created.

- **ax** (*Axes, optional*) – Matplotlib Axes on which to plot the projection. The default is *None*, in which case a new figure and axes will be created.
- **cmap** (str or ~*matplotlib.colors.Colormap*, optional) – The Colormap instance or registered colormap name used to map scalar data to colors.
- **vmin, vmax** (*float, optional*) – Define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data.
- **cbar** (*bool, optional*) – Whether or not to add a colorbar to the plot.

- **cbar_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.colorbar.
- **imshow_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.imshow, which is used to plot the 2D density map.

Returns **area_projection** (*ProjectionPlot*) – The ProjectionPlot object containing the area per lipid data and the matplotlob.pyplot.imshow plot of the projection.

6.5.6 Lipid order parameter — `lipyphilic.lib.order_parameter`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for calculating the orientational order parameter of lipid tails in a bilayer.

Coarse-grained order parameter

The class `liyphilic.lib.order_parameter.SCC` calculates the coarse-grained order parameter, as defined in Seo et al. (2020). The coarse-grained order parameter, S_{CC}, is defined as:

$$S_{CC} = \frac{\left\langle 3\cos^2\theta - 1 \right\rangle}{2}$$

where θ is the angle between the membrane normal and the vector connecting two consecutive tail beads. Angular brackets denote averages over all beads in an acyl tail.

See Piggot et al. (2017) for an excellent discussion on calculating acyl tali order parameters in molecular dynamics simulations.

Input

Required:

- *universe* : an MDAnalysis Universe object
- *tail_sel* : atom selection for beads in the acyl tail

Options:

- *normals* : local membrane normals for each tail at each frame

Output

- *SCC* : order parameter of each tail at each frame

The order parameter data are returned in a `numpy.ndarray`, where each row corresponds to an individual lipid and each column corresponds to an individual frame.

Warning: *tail_sel* should select beads in **either** the *sn1* **or** *sn2* tails, not both tails.

Example usage of Scc

An MDAnalysis Universe must first be created before using *SCC*:

```
import MDAnalysis as mda
from lipyphilic.lib.order_parameter import SCC

u = mda.Universe(tpr, trajectory)
```

If we have used the MARTINI forcefield to study a DPPC/DOPC/cholesterol mixture, we can calculate the order parameter of the *sn1* of tails of DPPC and DOPC as follows:

```
scc_sn1 = SCC(
  universe=u,
  tail_sel="name ??A"  # selects C1A, C2A, D2A, C3A, and C4A
)
```

This will calculate S_{CC} of each DOPC and DPPC sn1 tail.

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
scc_sn1.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `scc_sn1.SCC` attribute as a `numpy.ndarray`. The array has the shape (n_residues, n_frames). Each row corresponds to an individual lipid and each column to an individual frame.

Likewise, to calculate the S_{CC} of the *sn2* tails, we can do:

```
scc_sn2 = SCC(
  universe=u,
  tail_sel="name ??B"  # selects C1B, C2B, D2B, C3B, and C4B
)
scc_sn2.run(verbose=True)
```

And then get a weighted-average S_{CC} we can do:

```
SCC.weighted_average(scc_sn1, scc_sn2)
```

which will take into account the number of beads in each tail and return a new *SCC* object whose SCC attribute contains the weighted-average S_{CC} for each lipid at each frame.

Local membrane normals

By default, the S_{CC} is calculated as the angle between the positive z axis and the vector between two consecutive beads in an acyl tail. However, it is also possible to pass to *SCC* local membrane normals to use instead of the positive z axis.

You can also calculate local membrane normals using, for example, MemSurfer. If you store the local membrane normals in a `numpy.ndarray` called *normals*, with shape (n_residues, n_frames, 3), then you can simply pass these normals to *SCC*:

```
scc_sn1 = SCC(
  universe=u,
  tail_sel="name ??A",
  normals=normals
)
scc_sn1.run(verbose=True)
```

S_{CC} projected onto the membrane plane

Once the S_{CC} has been calculated, it is possible to create a 2D plot of time-averaged S_{CC} values projected onto the membrane plane. This can be done using the `liypphilic.lib.SCC.project_SCC()` method, which is a wrapper around the more general `liypphilic.lib.plotting.ProjectionPlot` class.

If the lipids have been assigned to leaflets, and the weighted average of the sn1 and sn2 tails stored in an *SCC* object named *scc*, we can plot the projection of the coarse-grained order parameter onto the membrane plane as follows:

```
scc_projection = scc.project_SCC(
  lipid_sel="name ??A ??B",
  start=-100,
  stop=None,
  step=None,
  filter_by=leaflets.filter_by("name ??A ??B") == -1
)
```

The order parameter of each lipid parameter will be averaged over the final 100 frames, as specified by the `start` argument. The frame in the middle of the selected frames will be used for determining lipid positions. In the above case, the lipid positions at frame -50 will be used. The `lipid_sel` specifies that the center of mass of the sn1 ("??A") and sn2 ("??B") atoms will be used for projecting lipid positions onto the membrane plane. And the *filter_by* argument is used here to specificy that only lipids in the lower (-1) leaflet should be used for plotting the projected S_{CC} values.

The class and its methods

class lipyphilic.lib.order_parameter.**SCC**(*universe*, *tail_sel*, *normals=None*)

Calculate coarse-grained acyl tail order parameter.

Set up parameters for calculating the SCC.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **tail_sel** (*str*) – Selection string for atoms in either the sn1 **or** sn2 tail of lipids in the membrane
- **normals** (*numpy.ndarray, optional*) – Local membrane normals, a 3D array of shape (n_residues, n_frames, 3), containing x, y and z vector components of the local membrane normals.

project_SCC(*lipid_sel=None*, *start=None*, *stop=None*, *step=None*, *filter_by=None*, *bins=None*, *ax=None*, *cmap=None*, *vmin=None*, *vmax=None*, *cbar=True*, *cbar_kws={}*, *imshow_kws={}*)

Project the SCC values onto the xy plane of the membrane.

The SCC values, averaged over a selected range of frames, are projected onto the xy plane based on the center of mass of each lipid. The atoms to be used in calculating the center of mass of the lipids can be specified using the *lipid_sel* arugment.

This method creates an instance of *lipyphilic.lib.plotting.ProjectionPlot* with the projected S_{CC} interpolated across periodic boundaries. The plot is returned so further modification can be performed if needed.

Note: The lipid positions are taken from the middle frame of the selected range.

Parameters

- **lipid_sel** (*MDAnalysis atom selection, optional*) – The center of mass of each lipid will be determined via this selection. The default is *None*, in which case every atom of a lipid is used to determine its center of mass.
- **start** (*int, optional*) – Start frame for averaging the SCC results.
- **stop** (*int, optional*) – Final frame for averaging the SCC results.
- **step** (*int, optional*) – Number of frames to skip
- **filter_by** (*array-like, optional*) – A Boolean mask for selecting a subset of lipids. It may take the following shapes:

  `(n_lipids)` The mask is used to select a subset of lipids for projecting the SCC onto the membrane plane.

  `(n_lipids, n_frames)` This is the same shape as the NumPy array created by the *lipyphilic.lib.SCC.run()* method. Boolean values are used only from the column corresponding to the middle frame of the range selected by *start*, *stop*, and *step*.

  The default is *None*, in which case no filtering is applied.
- **bins** (*int or array_like or [int, int] or [array, array]*) – The bin specification:

  `int` If int, the number of bins for the two dimensions (nx=ny=bins).

  `array-like` If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).

  `[int, int]` If [int, int], the number of bins in each dimension (nx, ny = bins).

  `[array, array]` If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

  `combination` A combination [int, array] or [array, int], where int is the number of bins and array is the bin edges.

  The default is *None*, in which case a grid with 1 x 1 Angstrom resolution is created.
- **ax** (*Axes, optional*) – Matplotlib Axes on which to plot the projection. The default is *None*, in which case a new figure and axes will be created.
- **cmap** (str or ~*matplotlib.colors.Colormap*, optional) – The Colormap instance or registered colormap name used to map scalar data to colors.
- **vmin, vmax** (*float, optional*) – Define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data.
- **cbar** (*bool, optional*) – Whether or not to add a colorbar to the plot.

- **cbar_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.colorbar.
- **imshow_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.imshow, which is used to plot the 2D density map.

Returns **scc_projection** (*ProjectionPlot*) – The ProjectionPlot object containing the SCC data and the matplotlob.pyplot.imshow plot of the projection.

`static weighted_average`(*sn1_scc*, *sn2_scc*)

Calculate the weighted average Scc of two tails.

Given two SCC objects, a weighted average of the Scc of each lipid is calculated.

Parameters

- **sn1_scc** (*SCC*) – An SCC object for which the order parameters have been calculated.
- **sn2_scc** (*SCC*) – An SCC object for which the order parameters have been calculated.

Returns **scc** (*SCC*) – An SCC object with the weighted average Scc of each lipid at each frame stored in the scc.SCC attirbute

Warning: The frames used in analysing 'sn1_scc' and 'sn2_scc' must be the same - i.e. the 'start', 'stop', and 'step' parameters passed to the '.run()' methods must be identical.

6.5.7 Lipid *z* angles — `lipyphilic.lib.z_angles`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for calculating the angle lipids make with the positive z axis.

Two atoms must be selected per lipid, and the angle between the z axis and the vector joining the two atoms will be calculated for each lipid. The vector will always point from atom B to atom A, even for lipids in the lower leaflet. This means the angle θ_{ABz} will be in the range $-180 < \theta < 180$.

Input

Required:

- *universe* : an MDAnalysis Universe object
- *atom_A_sel* : atom selection for atom A in each lipid
- *atom_B_sel* : atom selection for atom B in each lipid

Options:

- rad : boolean variable specifying whether to return the angle in radians

Output

- *z_angles* : angle made between the z-axis and the vector from B to A

The z angles data are returned in a `numpy.ndarray`, where each row corresponds to an individual lipid and each column corresponds to an individual frame.

Example usage of `ZAngles`

An MDAnalysis Universe must first be created before using ZAngles:

```
import MDAnalysis as mda
from lipyphilic.lib.z_angles import ZAngles

u = mda.Universe(tpr, trajectory)
```

If we have used the MARTINI forcefield to study a phospholipid/cholesterol mixture, we can calculate the orientation of cholesterol in the bilayer as follows:

```
z_angles = ZAngles(
  universe=u,
  atom_A_sel="name ROH",
  atom_B_sel="name R5"
)
```

This will calculate the angle between the z-axis and the vector from the *R5* bead to the *ROH* bead of cholesterol.

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
z_angles.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `z_angles.z_angles` attribute as a `numpy.ndarray`. The array has the shape (n_residues, n_frames). Each row corresponds to an individual lipid and each column to an individual frame.

Calculate the angle in radians

By default, the results are returned in degrees. We can also specify that the results should be returned in radians:

```
z_angles = ZAngles(
  universe=u,
  atom_A_sel="name ROH",
  atom_B_sel="name R5",
  rad=True
)
```

The class and its methods

class lipyphilic.lib.z_angles.**ZAngles**(*universe*, *atom_A_sel*, *atom_B_sel*, *rad=False*)
Calculate the orientation of lipids in a bilayer.

Set up parameters for calculating the orientations.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **atom_A_sel** (*str*) – Selection string for atom *A* of lipids in the membrane.
- **atom_B_sel** (*str*) – Selection string for atom *B* of lipids in the membrane.
- **rad** (*bool, optional*) – Whether to return the angles in radians. The default is *False*, in which case the results are returned in degrees.

Note: The orientation is defined as the angle between z and the vector from atom *B* to atom *A*.

6.5.8 Lipid *z* positions — lipyphilic.lib.z_positions

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for calculating the distance in z of lipids to the bilayer center.

The class `lipyphilic.lib.z_position.ZPositions` assigns the membrane midpoint to be at $z = 0$ Lipids in the upper leaflet will have positive z values and those in the lower leaflet will have negative z values.

Input

Required:

- *universe* : an MDAnalysis Universe object
- *lipid_sel* : atom selection for all lipids in the bilayer
- *height_sel* : atom selection for the molecules for which the z position will be calculated

Options:

- *n_bins* : split the membrane into *n_bins* * *n_bins* patches, and calculate local membrane midpoints for each patch

Output

- *z_position* : height in z of each selected molecule in the bilayer

The z positions data are returned in a `numpy.ndarray`, where each row corresponds to an individual molecule and each column corresponds to an individual frame.

Example usage of `ZPositions`

An MDAnalysis Universe must first be created before using ZPositions:

```
import MDAnalysis as mda
from lipyphilic.lib.z_positions import ZPositions

u = mda.Universe(tpr, trajectory)
```

If we have used the MARTINI forcefield to study a phospholipid/cholesterol mixture, we can calculate the height of cholesterol in the bilayer as follows:

```
z_positions = ZPositions(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  height_sel="name ROH"
)
```

`lipid_sel` is an atom selection that covers all lipids in the bilayer. This is used for calculating the membrane midpoint. `height_sel` selects which atoms to use for caclulating the height of each each molecule.

Note: In the above example we are calculating the height of cholesterol in the bilayer, although the height of any molecule - even those not in the bilayer, such as peptides - can be calculated instead.

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
z_positions.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `z_positions.z_positions` attribute as a `numpy.ndarray`. The array has the shape (n_residues, n_frames). Each row corresponds to an individual molecule and each column to an individual frame. The height is signed (not absolute) — positive and negative values correspond to the molecule being in the upper of lower leaflet respecitvely.

z positions based on local membrane midpoints

The first example computes a global membrane midpoint based on all the atoms of the lipids in the membrane. z positions are then calculated as the distance to this midpoint. This is okay for planar bilayers, but can lead to inaccurate results in membranes with undulations. If your bilayer has undulations, *ZPositions* can account for this by creating a grid in xy of your membrane, calculating the local membrane midpoint in each patch, then find the distance of each molecule to its local midpoint. This is done through use of *n_bins*:

```
z_positions = ZPositions(
  universe=u,
  lipid_sel="name GL1 GL2 ROH",
  height_sel="name ROH"
  n_bins=10
)
```

In this example, the membrane will be split into a *10 x 10* grid and a lipid z positions calculated based on the distance to the midpoint of the patch the molecule is in.

Warning: Using *n_bins* can account for small undulations. However, if you have large unulations in your bilayer the calculated height will be inaccurate.

The class and its methods

class lipyphilic.lib.z_positions.**ZPositions**(*universe*, *lipid_sel*, *height_sel*, *n_bins=1*)

Calculate the z position of molecules in a bilayer.

Set up parameters for calculating z positions.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for the lipids in a membrane. Atoms in this selection are used for calculating membrane midpoints.
- **height_sel** (*str*) – Selection string for molecules for which the height in z will be calculated. Any residues not in this selection will not have their z positions calculated.
- **n_bins** (*int, optional*) – Number of bins in x and y to use to create a grid of membrane patches. Local membrane midpoints are computed for each patch, and lipid z positions calculated based on the distance to their local membrane midpoint. The default is *1*, which is equivalent to computing a single global midpoint.

Note: `height_sel` must be a subset of `lipid_sel`

6.5.9 Lipid *z* thickness — `lipyphilic.lib.z_thickness`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for calculating the thickness in z of lipids or lipid tails.

The thickness of lipid tails is a useful input feature for creating Hidden Markov Models (HMM) to detect phase separation in lipid bilayers. See Park and Im (2019) for a description of using HMMs in lipid membrane analysis.

Input

Required:

- *universe* : an MDAnalysis Universe object
- *lipid_sel* : atom selection for the atoms to be used in calculating the thickness of a lipid

Output

- *z_thickness* : thickness in z of each lipid in the bilayer

The z thickness data are returned in a `numpy.ndarray`, where each row corresponds to an individual lipid and each column corresponds to an individual frame.

Example usage of `ZThickness`

An MDAnalysis Universe must first be created before using ZThickness:

```
import MDAnalysis as mda
from lipyphilic.lib.z_thickness import ZThickness

u = mda.Universe(tpr, trajectory)
```

If we have used the MARTINI forcefield to study a phospholipid/cholesterol mixture, we can calculate the thickness of cholesterol and *sn1* tails in the bilayer as follows:

```
z_thickness_sn1 = ZThickness(
  universe=u,
  lipid_sel="(name ??1 ??A) or (resname CHOL and not name ROH)"
)
```

Above, our *lipid_sel* selection will select sn1 beads and cholesterol beads in the MARTINI forcefield, making use of the powerful MDAnalysis atom selection language.

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
z_thickness_sn1.run(
  start=None,
  stop=None,
  step=None,
```

(continues on next page)

(continued from previous page)

```
  verbose=True
)
```

The results are then available in the `z_thickness_sn1.z_thickness` attribute as a `numpy.ndarray`. The array has the shape (n_residues, n_frames). Each row corresponds to an individual lipid and each column to an individual frame.

Averaging the thickness of two tails

Above we saw how to calculate the thickness of the *sn1* tail of lipids along with cholesterol. Similarly, we can calculate the thickness of the *sn2* tails:

```
z_thickness_sn2 = ZThickness(
  universe=u,
  lipid_sel="(name ??1 ??A) or (resname CHOL and not name ROH)"
)
z_thickness_sn2.run(verbose=True)
```

Now, if we would like to know the mean thickness of acyl tails across both *sn1* and *sn2* tails, we can use the `average()` method of *ZThickness*:

```
z_thickness = ZThickness.average(
    z_thickness_sn1,
    z_thickness_sn2
)
```

This will average the thickness of the two tails, leaving the cholesterol thicknesses (from *z_thickness_sn1*) unchanged, and return a new *ZThickness* object containing the averaged data in its `z_thickness` attribute.

The class and its methods

class lipyphilic.lib.z_thickness.**ZThickness**(*universe*, *lipid_sel*)

Calculate the thickness in z of lipids in a bilayer.

Set up parameters for calculating lipid thicknesses.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for atoms to use in calculating lipid thickneses

static average(*sn1_thickness*, *sn2_thickness*)

Calculate the average thickness of two tails.

Given two ZThickness objects, typically each representing either the sn1 or sn2 tails of the lipids, an averagte thickness of each lipid is calculated.

Parameters

- **sn1_thickness** (*ZThickness*) – A ZThickness object for which the thicknesses have been calculated.
- **sn2_thickness** (*ZThickness*) – A ZThickness object for which the thicknesses have been calculated.

Returns **z_thickness** (*ZThickness*) – A new *ZThickness* object containing the averaged data in its *z_thickness* attribute.

> **Warning:** The frames used in analysing 'sn1_thickness' and 'sn2_thickness' must be the same - i.e. the 'start', 'stop', and 'step' parameters passed to the '.run()' methods must be identical.

6.5.10 Membrane thickness — `lipyphilic.lib.memb_thickness`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

This module provides methods for calculating the membrane thickness over time.

The membrane thickness is defined as the mean distance in z between lipids in the upper and lower leaflets. A discrete intrinsic surface is constructed for each leaflet based on user-defined lipid headgroup atoms, and the mean distance in z between the two surfaces defines the membrane thickness.

Input

Required:

- *universe* : an MDAnalysis Universe object
- *leaflets* : a NumPy array containing the leaflet membership of each lipid at each frame
- *lipid_sel* : atom selection for lipids in the upper leaflet to used in the thickness calculation

Options: - *n_bins* : a discrete intrinsic surface of each leaflet is created with *n_bins* * *n_bins* patches

Output

- *thickness* : the mean membrane thickness at each frame

Thickness data are returned in a `numpy.ndarray`, containing the mean membrane thickness at each frame.

Example usage of `MembThickness`

An MDAnalysis Universe must first be created before using *MembThickness*:

```
import MDAnalysis as mda
from lipyphilic.lib.memb_thickness import MembThickness

u = mda.Universe(tpr, trajectory)
```

Then we need to know which leaflet each lipid is in at each frame. This may be done using *lipyphilic.lib.assign_leaflets.AssignLeaflets*:

```
leaflets = AssignLeaflets(
  universe=u,
  lipid_sel="name GL1 GL2 ROH" # assuming we are using the MARTINI forcefield
)
leaflets.run()
```

The leaflets data are stored in the `leaflets.leaflets` attribute. We can now create our MembThickness object by passing the results of *lipyphilic.lib.assign_leaflets.AssignLeaflets* *MembThickness* along with an atom selection for the lipids:

```
memb_thickness = MembThickness(
  universe=u,
  leaflets=leaflets.filter_leaflets("resname DOPC and DPPC"),  # exclude cholesterol␣
↪from thickness calculation
  lipid_sel="resname DPPC DOPC and name PO4"
)
```

To calculate the membrane thickness, based on interleaflet PO4 to PO4 distances, we need to use the `run()` method. We select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
memb_thickness.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `memb_thickness.memb_thickness` attribute as a `numpy.ndarray`.

Changing the resolution of the 2D grid

By default, the lipid positions of each leaflet are binned into a two-dimensional histogram using 1 bins in each dimension. This is equivalent to calculating the mean height of all headgroup atoms in the bilayer, without discretising the surface.

It is also possible to specify the bins to use for binning the data:

```
memb_thickness = MembThickness(
  universe=u,
  leaflets=leaflets.filter_leaflets("resname DOPC and DPPC"),  # exclude cholesterol␣
↪from thickness calculation
  lipid_sel="resname DPPC DOPC and name PO4",
  n_bins=10
)
```

This will use *10* bins in each dimension for creating the two-dimensional histogram.

Interpolate missing values in a grid with many bins

This is useful only if you would like a very high resolution grid. Having a higher resolution grid may be useful if you would like to later calculate, for example, the correlation between local membrane thicknesses and the local membrane area per lipid. The area per lipid can be projected onto the membrane plane using the class `ProjectionPlot`, and the height of the bilayer as a function of xy can be obtained from *MembThickness* by setting the `return_surface` keyword to *True*.

A grid with a small bin size (large `n_bins`) will lead to bins with no atom, and thus no height value. In this instance, the `interpolate` keyword should be set to True. However, interpolation substantially decreases performance and should be left as *False* unless it is strictly necessary.

The class and its methods

class lipyphilic.lib.memb_thickness.**MembThickness**(*universe*, *lipid_sel*, *leaflets*, *n_bins=1*, *interpolate=False*, *return_surface=False*)

Calculate the bilayer thickness.

Set up parameters for calculating membrane thickness.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **leaflets** (*numpy.ndarray (n_lipids,)*) – An array of leaflet membership in which: -1 corresponds to the lower leaflet; 1 corresponds to the upper leaflet; and 0 corresponds to the midplane. If the array is 1D and of shape (n_lipids), each lipid is taken to remain in the same leaflet over the trajectory. If the array is 2D and of shape (n_lipids, n_frames), the leaflet to which each lipid is assisgned at each frame will be taken into account when calculating the area per lipid.
- **lipid_sel** (*str, optional*) – Selection string for lipid atoms to be used in the thickness calculation. The default is *None*, in which case all atoms of the lipids will be used.
- **n_bins** (*int, optional*) – Number of bins in x and y to use to create a grid of membrane patches. The intrinsic surface of a leaflet is constructed via the height in z of each patch. The default is *1*, which is equivalent to computing a single global leaflet height.
- **interpolate** (*bool, optional*) – If True, interpolate the two intrinsic surfaces to fill missing values. This substantially decreases performance but allows for the construction of higher-resolution grids. The default is False.
- **return_surface** (*bool, optional*) – If True, the height of the bilayer at grid point at each frame is returned as numpy ndarray. The default is False.

Tip: Leaflet membership can be determined using *lipyphilic.lib.assign_leaflets.AssignLeaflets*.

6.5.11 Lateral diffusion — `lipyphilic.lib.lateral_diffusion`

This module contains methods for calculating the lateral diffusion coefficient of lipids in a bilayer.

The class *lipyphilic.lib.lateral_diffusion.MSD* calculates the two-dimensional mean squared displacent (MSD) of lipids in a bilayer. The Fast Correlation Algorithm, implemented in tidynamics is used to calculate the MSD of each lipid, with optional removal of the center of mass motion of the bilayer.

lipyphilic.lib.lateral_diffusion.MSD also contains a method for calculating the lateral diffusion coefficient, D_{xy}, via the Einstein relation:

$$D_{xy} = \frac{1}{4} \lim_{t \to \infty} \frac{d}{dt} \left\langle \frac{1}{N} \sum_{i=1}^{N} |r_i(t_0 + \Delta t) - r_i(t_0)|^2 \right\rangle_{t_0}$$

where N is the number of lipids, $r_i(t0)$ is the center of mass in xy of lipids i at a time origin *t_0*, $r_i(t0 + \Delta t)$ is the same lipid's center of mass at a lagtime Δt, and the angular brackets denote an average over all time origins, t_0.

Typically, the MSD is averaged over all molecules. However, *lipyphilic.lib.lateral_diffusion.MSD* will return the MSD for each individual lipid. This makes it simple to later calculate the diffusion coefficient using a subset of the lipids, such as a specific lipid species or lipids near a protein.

Input

Required:

- *universe* : an MDAnalysis Universe object
- *lipid_sel* : atom selection for calculating the MSD

Optional:

- *com_removal_sel* : atom selection for center of mass removal from the MSD
- *dt* : time period betwen consecutive frames in the MSD analysis

Output

- *msd* : the mean squared displacement of each lipid at each lagtime, Δt, in nm^2:
- *lagtimes* : a NumPy array of lagtimes (in ns)

The data are stored in the `MSD.msd` and `MSD.lagtimes` attributes.

Warning: Before using *lipyphilic.lib.lateral_diffusion.MSD* you *must* ensure that the coordinates have been unwrapped using, for example, *lipyphilic.transformations.nojump*.

Example usage of `MSD`

To calculate the MSD of each lipid in a bilayer we must first load a trajectory using MDAnalysis:

```
import MDAnalysis as mda
from lipyphilic.lib.lateral_diffusion import MSD

u = mda.Universe(tpr, trajectory)
```

If we have used the MARTINI forcefield to study a phospholipid/cholesterol mixture, we can calculate the MSD of each lipid as follows:

```
msd = MSD(
  universe=u,
  lipid_sel="resname DPPC DOPC CHOL"
)
```

We then select which frames of the trajectory to analyse (*None* will use every frame) and choose to display a progress bar (*verbose=True*):

```
msd.run(
  start=None,
  stop=None,
  step=None,
  verbose=True
)
```

The results are then available in the `msd.MSD` attribute as a `numpy.ndarray`. Each row corresponds to an individual lipid and each column to a different lagtime`.

Center of mass removal

During your simulation, it is likely that you removed the center of mass motion of your bilayer in the z direction. However, it is not possible to remove the x and y center of mass motions until you have unwrapped your lipid positions.

You may select which lipids to use for the center of mass motion removal using the `com_removal_sel` keyword:

```
msd = MSD(
  universe=u,
  lipid_sel="resname DPPC",
  com_removla_sel="resname DPPC DOPC CHOL"
)
```

In this case, the MSD of DPPC will be calculated with the center of mass motion of the bilayer will be subtracted from it.

Plotting the MSD of each species

If you have calculated the MSD of DPPC, DOPC and cholesterol as in the first example, you can plot the MSD of each species as follows:

```
for species in ["DPPC", "DOPC", "CHOL"]:

  plt.loglog(
    msd.lagtimes,
    np.mean(msd.msd[msd.membrane.residues.resnames == species], axis=0),
    label=species
  )

plt.legend()
```

The linear part of the log-log plot can be used for fitting a line and calculating the diffusion coefficient.

Calculating the lateral diffusion coefficient

After calculating the MSD and identifying the linear portion of the plot, the :func: *diffusion_coefficient* method of *lipyphilic.lib.lateral_diffusion.MSD* can be used to calculate D_{xy}. We need to pass the time at which to start and stop the linear fit:

```
d, sem = msd.diffusion_coefficient(
  start_fit=400,
  end_fit=600
)
```

This will calculate a diffusion coefficient for each individual lipid and return the mean and standard error of the distribution of coefficients.

To calculate the diffusion coefficient of a subset of lipids we can use the *:attr:lipid_sel* keyword:

```
d, sem = msd.diffusion_coefficient(
  start_fit=400,
  end_fit=600,
  lipid_sel="resname CHOL"
)
```

which will calculate the lateral diffusion coefficient for cholesterol, using a fit to the MSD curve from lagtime $\Delta t = 400$ to lagtime $\Delta t = 600$.

class lipyphilic.lib.lateral_diffusion.**MSD**(*universe*, *lipid_sel*, *com_removal_sel=None*, *dt=None*)

Calculate the mean-squared lateral displacement of lipids in a bilayer.

The MSD is returned in units of nm^2/ns.

Parameters

- **universe** (*Universe*) – MDAnalysis Universe object
- **lipid_sel** (*str*) – Selection string for calculating the mean-squared displacemnt. IF multiple atoms per lipid are selected, the center-of-mass of these atoms will be used for calculating the MSD.
- **com_removal_sel** (*str, optional*) – The MSD of the center of mass of atoms in this selection will be subtracted from all individual lipid MSDs. The default is *None*, in which case no center of mass motion removal is performed.
- **dt** (*float, optional*) – The time, in nanoseconds, between consecutive frames in *universe.trajectory*. The defualt is *None*, in which case *dt* is taken to be *universe.trajectory.dt* divided by 1000.

diffusion_coefficient(*start_fit=None*, *stop_fit=None*, *lipid_sel=None*)

Calculate the lateral diffusion coefficient via the Einstein relation.

A diffusion is calculated for each lipid through a linear fit to its MSD curve. The mean and standard error of the diffusion coefficient is returned.

Parameters

- **start_fit** (*float, optional*) – The time at which to start the linear fit to the MSD curve. The default is *None*, in which case the fit will exclude the first 20% of the MSD data.
- **stop_fit** (*float, optional*) – The time at which to stop the linear fit to the MSD curve. The default is *None*, in which case the fit will exclude the final 20% of the MSD data.

- **lipid_sel** (*str, optional*) – Selection string for lipids to include in calculating the diffusion coefficient.

Returns

- **d** (*float*) – The mean lateral diffusion coefficient, in cm^2/s., averaged over all lipids in *lipid_sel*.
- **sem** (*float*) – The standard error of the diffusion coefficients.

6.5.12 Plotting utilities — `lipyphilic.lib.plotting`

Author Paul Smith

Year 2021

Copyright GNU Public License v2

Generally, **lipyphilic** is not a plotting library — everyone has their favourite plotting tool and aesthetics and so plotting is generally left up to the user. However, some plots are complex to make, requiring further processing of results or lots of boilerplate code to get the end result.

This module provides methods for plotting joint probability densities and lateral distribution maps of lipid properties projected onto the membrane plane.

The class *lipyphilic.lib.plotting.ProjectionPlot* can be used, for example, to plot the area per lipid projected onto the membrane plane, i.e. plot the area per lipid as a function of xy. See Gu et al. (2020) for examples of these projection plots.

The class *lipyphilic.lib.plotting.JointDensity* can be used, for example, to plot a 2D PMF of cholesterol orientation and height in a lipid membrane. See Gu et al. (2019) for an example of the this 2D PMF.

The classes and their methods

class lipyphilic.lib.plotting.**ProjectionPlot**(*x_pos*, *y_pos*, *values*)

Plot membrane properties as a function of *xy*. See

This class can be used for plotting membrane properties projected onto the xy plane. This is useful, for example, for detecting phase separation in lipid membranes.

The plotted data are stored in the *.statistic* attribute. This means, if you plot separately the projection of a membrane property of lipids in the upper and lower leaflets, you can easily calculate the correlation coefficient of this property across the leaflets.

interpolate(*tile=True*, *method='linear'*, *fill_value=numpy.NaN*)

Interpolate NaN values in the projection array.

Uses scipy.interpolate.griddata to interpolate missing values and optionally remove NaN values.

Parameters

- **tile** (*bool, optional*) – If *True*, the xy values will be tiled on a (3, 3) grid to reproduce the effect of periodic boundary conditions. If *False*, no periodic boundary conditions are taken into account when interpolating.
- **method** (*{'linear', 'nearest', 'cubic'}, optional*) – Method of interpolation. One of:

  `nearest` return the value at the data point closest to the point of interpolation. See SciPy's *NearestNDInterpolator* for more details.

`linear` tessellate the input point set to N-D simplices, and interpolate linearly on each simplex. See SciPy's *LinearNDInterpolator* for more details.

`cubic` return the value determined from a piecewise cubic, continuously differentiable (C1), and approximately curvature-minimizing polynomial surface. See SciPy's *CloughTocher2DInterpolator* for more details.

- **fill_value** (*float, optional*) – Value used to fill in for requested points outside of the convex hull of the input points. This option has no effect for the 'nearest' method. If not provided, then the these points will have NaN values.
- **rescale** (*bool, optional*) – Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.

`plot_projection`(*ax=None*, *title=None*, *xlabel=None*, *ylabel=None*, *cmap=None*, *vmin=None*, *vmax=None*, *cbar=True*, *cbar_kws={}*, *imshow_kws={}*)

Plot the 2D projection of a membrane property.

Use matplotlib.pyplot.imshow to plot a heatmap of the values.

Parameters

- **ax** (*Axes, optional*) – Matplotlib Axes on which to plot the projection. The default is *None*, in which case a new figure and axes will be created.
- **title** (*str, optional*) – Title for the plot. By default, there is no title.
- **xlabel** (*str, optional*) – Label for the x-axis. By default, there is no label on the x-axis.
- **ylabel** (*str, optional*) – Label for the y-axis. By default, there is no label on the y-axis.
- **cmap** (str or *~matplotlib.colors.Colormap*, optional) – The Colormap instance or registered colormap name used to map scalar data to colors.
- **vmin, vmax** (*float, optional*) – Define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data.
- **cbar** (*bool, optional*) – Whether or not to add a colorbar to the plot.
- **cbar_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.colorbar.
- **imshow_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.imshow, which is used to plot the 2D density map.

Returns

- *ProjectionPlot.fig* – Matplotlib Figure on which the plot was drawn.
- *ProjectionPlot.ax* – Matplotlib Axes on which the plot was drawn.
- *ProjectionPlot.cbar* – If a colorbar was added to the plot, this is the Matplotlib colorbar instance for ProjectionPlot.ax. Otherwise it is *None*.

`project_values`(*bins*, *statistic='mean'*)

Discretise the membrane and project values onto the xy-plane

Parameters

- **bins** (*int or array_like or [int, int] or [array, array]*) – The bin specification:

  `int` If int, the number of bins for the two dimensions (nx=ny=bins).

  `array-like` If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).

[int, int] If [int, int], the number of bins in each dimension (nx, ny = bins).

[array, array] If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

combination A combination [int, array] or [array, int], where int is the number of bins and array is the bin edges.

- **statistic** (*string or callable, optional*) – The statistic to project onto the membrae plane (the default is 'mean'). The following statistics are available:

  mean compute the mean of values for points within each bin. Empty bins will be represented by NaN.

  std` compute the standard deviation within each bin.

  median compute the median of values for points within each bin. Empty bins will be represented by NaN.

  count compute the count of points within each bin. This is identical to an unweighted histogram. The value of the membrane property is not referenced.

  sum compute the sum of values for points within each bin. This is identical to a weighted histogram.

  min compute the minimum of values for points within each bin. Empty bins will be represented by NaN.

  max compute the maximum of values for point within each bin. Empty bins will be represented by NaN.

  function a user-defined function which takes a 1D array of values, and outputs a single numerical statistic. This function will be called on the values in each bin. Empty bins will be represented by function([]), or NaN if this returns an error.

class lipyphilic.lib.plotting.**JointDensity**(*ob1*, *ob2*)

Calculate and plot the joint probability density of two observables.

Set up parameters for calculating joint densities.

Parameters

- **ob1** (*array_like*) – An array containing values of the first observable.
- **ob2** (*array_like*) – An array containing values of the second observable. It **must** have the same shape as *ob1*

calc_density_2D(*bins*, *filter_by=None*, *temperature=None*)

Calculate the joint probability density of two observables.

If a tempearutre is provided, the PMF is calculated directly from the probability distribution.

Parameters

- **bins** (*int or array_like or [int, int] or [array, array]*) – The bin specification:

  int If int, the number of bins for the two dimensions (nx=ny=bins).

  array-like If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).

  [int, int] If [int, int], the number of bins in each dimension (nx, ny = bins).

  [array, array] If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

`combination` A combination [int, array] or [array, int], where int is the number of bins and array is the bin edges.

- **filter_by** (*2D numpy array of shape (n_residues, n_frames), optional*) – A boolean mask for filtering lipids or frames. The default is *None*, in which case no filtering is performed.
- **temperature** (*float, optional*) – Temperature of the system, which will be used to convert the 2D density into a PMF. The default is *None*, in which case the density is returned rather than the PMF.

`interpolate`(*method='linear'*, *fill_value=None*, *rescale=True*)

Interpolate NaN values in the joint probability density or PMF.

Uses scipy.interpolate.griddata to interpolate the joint density and optionally remove NaN values.

Parameters

- **method** (*{'linear', 'nearest', 'cubic'}, optional*) – Method of interpolation. One of:

  `nearest` return the value at the data point closest to the point of interpolation. See SciPy's *NearestNDInterpolator* for more details.

  `linear` tessellate the input point set to N-D simplices, and interpolate linearly on each simplex. See SciPy's *LinearNDInterpolator* for more details.

  `cubic` return the value determined from a piecewise cubic, continuously differentiable (C1), and approximately curvature-minimizing polynomial surface. See SciPy's *CloughTocher2DInterpolator* for more details.

- **fill_value** (*float, optional*) – Value used to fill in for requested points outside of the convex hull of the input points. This option has no effect for the 'nearest' method. If not provided, then the default is to use the maximum free energy value if a PMF was calculated, or 0 otherwise.
- **rescale** (*bool, optional*) – Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.

`plot_density`(*difference=None*, *ax=None*, *title=None*, *xlabel=None*, *ylabel=None*, *cmap=None*, *vmin=None*, *vmax=None*, *n_contours=4*, *contour_labels=None*, *cbar=True*, *cbar_kws={}*, *imshow_kws={}*, *contour_kws={}*, *clabel_kws={}*)

Plot the 2D density or PMF.

Use matplotlib.pyplot.imshow to plot a heatmap of the density.

Optionally, add contour lines using matplotlib.pyplot.contour and label the contours with their values.

Parameters

- **difference** (*JointDensity, optional*) – A JointDensity object for which the probability density or PMF has been calculated. Before ploting, the density or PMF of *difference* will be subtracted from the density of PMF of this object. This is useful for plotting difference in PMFs due to e.g a change in membrane lipid composition.
- **ax** (*Axes, optional*) – Matplotlib Axes on which to plot the 2D denstiy. The default is *None*, in which case a new figure and axes will be created.
- **title** (*str, optional*) – Title for the plot. By default, there is no title.
- **xlabel** (*str, optional*) – Label for the x-axis. By default, there is no label on the x-axis.
- **ylabel** (*str, optional*) – Label for the y-axis. By default, there is no label on the y-axis.

- **cmap** (str or *~matplotlib.colors.Colormap*, optional) – The Colormap instance or registered colormap name used to map scalar data to colors.
- **vmin, vmax** (*float, optional*) – Define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data.
- **n_contours** (*int or array-like, optional*) – Determines the number and positions of the contour lines / regions plotted with matplotlib.pyplot.contour:

  int If an int *n*, use *~matplotlib.ticker.MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between *vmin* and *vmax*.

  array-like If array-like, draw contour lines at the specified levels. The values must be in increasing order.

  0 If 0, no contour lines are drawn.
- **contour_labels** (*array-like, optional*) – A list of contour level indices specifyig which levles should be labeled. The default is *None*, in which case no contours are labeled.
- **cbar** (*bool, optional*) – Whether or not to add a colorbar to the plot.
- **cbar_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.colorbar.
- **imshow_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.imshow, which is used to plot the 2D density map.
- **contour_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.contour, which is used to plot the contour lines.
- **clabel_kws** (*dict, optional*) – A dictionary of keyword options to pass to matplotlib.pyplot.contour, which is used to add labels to the contour lines.

Returns

- *JointDensity.fig* – Matplotlib Figure on which the plot was drawn.
- *JointDensity.ax* – Matplotlib Axes on which the plot was drawn.
- *JointDensity.cbar* – If a colorbar was added to the plot, this is the Matplotlib colorbar instance for JointDensity.ax. Otherwise it is *None*.

6.5.13 Trajectory transformations — `lipyphilic.transformations`

This module contains methods for applying on-the-fly trajectory transformations with MDAnalysis.

Prevent atoms from jumping across periodic boundaries

lipyphilic.transformations.nojump can be used to prevent atoms from jumping across periodic boundaries. It is equivalent to using the GROMACS command trjconv with the flag *-pbc nojump*.

The on-the-fly transformation can be added to your trajectory after loading it with MDAnalysis:

```
import MDAnalysis as mda
from lipyphilic.transformations import nojump

u = mda.Universe("production.tpr", "production.xtc")

ag = u.select_atoms("name GL1 GL2 ROH")
```

(continues on next page)

(continued from previous page)

```
u.trajectory.add_transformations(nojump(ag))
```

Upon adding this transformation to your trajectory, *lipyphilic* will determine at which frames each atom crosses a boundary, keeping a record of the net movement across each boundary. Then, every time a new frame is loaded into memory by *MDAnalysis* — such as when you iterate over the trajectory — the transformation is applied.

This transformation is required when calculating the lateral diffusion of lipids in a membrane using, for example, *lipyphilic.lib.lateral_diffusion.MSD*. It can be used to remove the need to create an unwrapped trajectory using *GROMACS*.

Fix membranes broken across periodic boundaries

The callable class *lipyphilic.transformations.center_membrane* can be used to fix a membrane split across periodic boundaries and then center it in the unit cell. The membrane is iteratively shifted along a dimension until it is no longer split across periodic boundaries. It is then moved it to the center of the box in this dimension.

The on-the-fly transformation can be added to your trajectory after loading it with MDAnalysis:

```
import MDAnalysis as mda
from lipyphilic.transformations import center_membrane

u = mda.Universe("production.tpr", "production.xtc")

ag = u.select_atoms("resname DPPC DOPC CHOL")

u.trajectory.add_transformations(center_membrane(ag))
```

This will center a DPPC/DOPC/cholesterol membrane in z every time a new frame is loaded into memory by MDAnalysis, such as when you iterate over the trajectory:

```
for ts in u.trajectory:

    # do some nice analysis with your centered membrane
```

Note: *ag* should be an AtomGroup that contains *all* atoms in the membrane.

Transform triclinic coordinates to their orthorhombic representation

lipyphilic.transformations.triclinic_to_orthorhombic can be used to transform triclinic coordinates to their orthorhombic representation. It is equivalent to using the GROMACS command trjconv with the flag *-ur rect*.

The on-the-fly transformation can be added to your trajectory after loading it with MDAnalysis:

```
import MDAnalysis as mda
from lipyphilic.transformations import triclinic_to_orthorhombic

u = mda.Universe("production.tpr", "production.xtc")

ag = u.select_atoms("resname DPPC DOPC CHOL")
u.trajectory.add_transformations(triclinic_to_orthorhombic(ag=ag))
```

After adding this transformation, upon load a new frame into memory the coordinates of the selected atoms will be transformed, and the dimensions of your system will be modified so that the angles are all 90°. Further analysis may then be performed using the orthorhombic coordinate system.

Some analyses in *lipyphilic* create a surface of the membrane plane using a two-dimensional rectangular grid. This includes

- `lipyphilic.lib.assign_leaflet.AssignLeaflets`
- `lipyphilic.lib.memb_thickness.MembThicnkess`
- *lipyphilic.lib.registration.Registration*

These analyses will fail with triclinic boxes - the *triclinic_to_orthorhombic* transformation *must* be applied to triclinic systems before these tools can be used.

Another case that will fail with triclinic systems is the *lipyphilic.transformations.nojump* transformation - this transformation can currently only unwrap coordinates for orthorhombic systems.

See *lipyphilic.transformations.triclinic_to_orthorhombic* for the full list.

class lipyphilic.transformations.**nojump**(*ag*, *nojump_x=True*, *nojump_y=True*, *nojump_z=False*, *filename=None*)

Prevent atoms jumping across periodic boundaries.

This is useful if you would like to calculate the diffusion coefficient of lipids in your membrane.

This transformation does an initial pass over the trajectory to determine at which frames each atom crosses a boundary, keeping a record of the net movement across each boundary. Then, as a frame is loaded into memory, atom positions are translated according to their total displacement, taking into account crossing of boundaries as well box fluctuations in the box volume.

By default, atoms are only unwrapped in xy, as it is assumed the membrane is a bilayer. To unwrap in all dimensions, `center_z` must also be set to *True*.

Parameters

- **ag** (*AtomGroup*) – MDAnalysis AtomGroup to which to apply the transformation
- **nojump_x** (*bool, optional*) – If true, atoms will be prevented from jumping across periodic boundaries in the x dimension.
- **nojump_y** (*bool, optional*) – If true, atoms will be prevented from jumping across periodic boundaries in the y dimension.
- **nojump_z** (*bool, optional*) – If true, atoms will be prevented from jumping across periodic boundaries in the z dimension.
- **filename** (*str, optional*) – File in which to write the unwrapped, nojump trajectory. The default is *None*, in which case the transformation will be applied on-the-fly.py

Returns `MDAnalysis.coordinates.base.Timestep` object, or *None* if a filename is provided.

Note: The *nojump* transformation is memory intensive to perform on-the-fly. If you have a long trajectory or a large number of atoms to be unwrapped, you can write the unwrapped coordinates to a new file by providing a `filename` to *nojump*.

Warning: The current implementation of *nojump* can only unwrap coordinates in orthorhombic systems.

class lipyphilic.transformations.**center_membrane**(*ag*, *shift=20*, *center_x=False*, *center_y=False*, *center_z=True*, *min_diff=10*)

Fix a membrane split across periodic boundaries and center it in the primary unit cell.

If, for example, the bilayer is split across z, it will be iteratively translated in z until it is no longer broken. Then it will be moved to the center of the box.

A membrane with a maximum extent almost the same size as the box length in a given dimension will be considered to be split across that dimension.

By default, the membrane is only centered in z, as it is assumed the membrane is a bilayer. To center a micelle, `center_x` and `center_y` must also be set to *True*.

Parameters

- **ag** (*AtomGroup*) – MDAnalysis AtomGroup containing *all* atoms in the membrane.
- **shift** (*float, optional*) – The distance by which a bilayer will be iteratively translated. This *must* be smaller than the thickness of your bilayer or the diameter of your micelle.
- **min_diff** (*float, optional*) – Minimum difference between the box size and the maximum extent of the membrane in order for the membrane to be considered unwrapped.
- **center_x** (*bool, optional*) – If true, the membrane will be iteratively shifted in x until it is not longer split across periodic boundaries.
- **center_y** (*bool, optional*) – If true, the membrane will be iteratively shifted in y until it is not longer split across periodic boundaries.
- **center_z** (*bool, optional*) – If true, the membrane will be iteratively shifted in z until it is not longer split across periodic boundaries.

Returns `MDAnalysis.coordinates.base.Timestep` object

class lipyphilic.transformations.**triclinic_to_orthorhombic**(*ag*)

Transform triclinic coordinates to their orthorhombic representation.

If you have a triclinic system, it is *essential* to apply this transformation before using the following analyses:

- *lipyphilic.lib.assign_leaflet.AssignLeaflets*
- *lipyphilic.lib.area_per_lipid.AreaPerLipid*
- *lipyphilic.lib.memb_thickness.MembThicnkess*
- *lipyphilic.lib.registration.Registration*

as well as before the following on-the-fly transformations:

- *lipyphilic.transformations.nojump*
- *lipyphilic.transformations.center_membrane*

The above tools will fail unless provided with an orthorhombic system.

This transformation is equivalent to using the GROMACS command trjconv with the flag *-ur rect*.

Note: *triclinic_to_rectangular* will put all selected atoms into the primary (orthorhombic) unit cell - molecules will **not** be kept whole or unwrapped.

Warning: If you wish to apply the *triclinic_to_orthorhombic* transformation along with other on-the-fly transformations, *triclinic_to_orthorhombic* **must** be the first one applied.

Parameters **ag** (*AtomGroup*) – MDAnalysis AtomGroup to which to apply the transformation

6.6 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

6.6.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.6.2 Documentation improvements

lipyphilic could always use more documentation, whether as part of the official lipyphilic docs, in docstrings, or even on the web in blog posts, articles, and such.

6.6.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/p-j-smith/lipyphilic/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

6.6.4 Development

To set up *lipyphilic* for local development:

1. Create and activate your isolated development environment:

```
curl https://raw.githubusercontent.com/p-j-smith/lipyphilic/master/requirements-dev.
↪yml -o lipyphilic-dev.yml
conda env create -f lipyphilic-dev.yml
conda activate lipyphilic-dev
```

2. Fork lipyphilic (look for the "Fork" button).
3. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/lipyphilic.git
```

4. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

6. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

To check that the docs build:

```
tox -e docs
```

And to check the build and test coverage (using python 3.8):

```
tox -e py38-cover
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though …

6.7 Authors

LiPyphilic was created by Paul Smith.

6.7.1 Authors (chronological)

- Paul Smith
- Raquel Lopez-Rios

6.8 LiPyphilic CHANGELOG

6.8.1 0.9.0 (2021-09-02)

- PR#78 Min MDAnalysis version increased to 2.0

6.8.2 0.8.0 (2021-07-31)

- PR#74 Add the triclinic_to_orthorhombic transformation in order to support analysis of triclinic systems

6.8.3 0.7.0 (2021-07-03)

- PR#70 Remove support for Python 3.6
- PR#69 Change MSD lagtimes to be in ns rather than ps. Fix nojump unwrapping for the first frame.

6.8.4 0.6.3 (2021-05-09)

- PR#60 AssignLeaflets and AssignCurvedLeaflets inherit from shared leaflet analysis base class
- PR#59 Ensure SCC.weighted_average can handle different sized sn1 and sn2 residue groups.
- PR#56 Update docs

6.8.5 0.6.2 (2021-04-18)

- PR#54 Fixed typos in docs
- PR#53 Improved performance of lipyphilic.lib.flip_flop.FlipFlop
- PR#52 Improved performance of lipyphilic.lib.neighbours.Neighbours (Fixes #51)

6.8.6 0.6.1 (2021-04-16)

- PR#49 Add min_diff argument to transformations.center_membrane
- PR#48 Add MDAnalysis badge to README and fix typos in the docs
- PR#47 Fixed typos in docs

6.8.7 0.6.0 (2021-03-26)

- PR#44 Refactor the Registration analysis to have a more useful API
- PR#43 Add a method for calculating the lipid enrichment/depletion index
- PR#42 Add a MSD and lateral diffusion analysis, as well as a transformation to perform "nojump" unwrapping.
- PR#39 Add support for assigning lipids to leaflets of highly curved membranes

6.8.8 0.5.0 (2021-03-16)

- PR#38 Add a trajectory transformation for unwrapping broken membranes (Fixes #37)
- PR#36 Add method for projecting areas onto the membrane plane (Fixes #33)
- PR#35 Added a tool for calculating membrane thickness (Fixes #34)
- PR#32 ZThickness.average() now returns a new ZThickness object rather than a NumPy array
- PR#31 SCC.weighted_average() now returns a new SCC object rather than a NumPy array
- PR#30 Add class for plotting projections of membrane properties onto the xy plane.
- PR#29 Added plotting of joint probability distributions or PMFs (Fixed #28).

6.8.9 0.4.0 (2021-03-05)

- PR#26 Added a tool to calculate the thickness of lipids or their tails (Fixes #25)
- PR#24 Added a tool to calculate the coarse-grained order parameter (Fixes #23)
- PR#22 Added a tool to calculate orientation of lipids in a bilayer (Fixes #20)
- PR#21 Added a tool to calculate lipid height in a bilayer (Fixes #19)
- Better description of analysis tools in the docs
- Updated installation instructions, including installing via conda-forge

6.8.10 0.3.2 (2021-02-27)

- Fix typo in requirements

6.8.11 0.3.1 (2021-02-27)

- Add support for numpy 1.20

6.8.12 0.3.0 (2021-02-26)

- Fix neighbour calculation for non-sequential residue indices Fixes #11
- Added a tool to calculate interleaflet registration

6.8.13 0.2.0 (2021-02-23)

- Improved documentation
- Add method to count number of each neighbour type
- Add functionality to find neighbouring lipids

6.8.14 0.1.0 (2021-02-17)

- Add functionality to find flip-flop events in bilayers
- Add functionality to calculate area per lipid
- Add functionality to find assign lipids to leaflets in a bilayer

6.8.15 0.0.0 (2021-02-08)

- First release on PyPI.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

N

P

R

S

T

W

Z