Hop Documentation

Release 0.4.0

Thomas B. Woolf, Oliver Beckstein

Sep 20, 2018

Contents

Release 0.4.0

Date Sep 20, 2018

This is a collection of python modules to analyze (primarily) water behaviour in MD simulations. The idea is to find regions with a density above a given threshold (hydration sites) and catalogue those sites. Once this is done, one can analyze water movement in terms of hops between those sites. The complicated solvation dynamics is thus represented as a graph in which hydration sites are the nodes (or vertices) and movements between sites are the edges.

Of course, it is also possible to look at the movement of other particles such as ions or small molecules — one simply selects a different species.

The package is called *Hop* (no clever acronym, just quick to type, and reflecting the fact that a "hopping analysis" is performed).

Hop is built with MDAnalysis.

Warning: This is legacy software that is provided "AS IS". In particular, there are currently *no tests* and it is not guaranteed to work or produce correct results. Help and contributions are welcome!

CHAPTER 1

License

hop is released under the GNU General Public License, v3 (because it links to MDAnalysis, which is GPL licensed).

CHAPTER 2

Documentation

The primary documentation consists of the online docs (which you are reading).

There is also some content in the `doc/` directory, in particular `doc/overview.txt`.

CHAPTER 3

Bug reporting

Almost invariably, things will not work right away or it will be unclear how to accomplish a certain task. In order to keep track of feedback I ask you to use the Issue tracker at https://github.com/Becksteinlab/hop/issues

CHAPTER 4

Citing

If you use Hop in published work please cite (for the time being) the abstract *[Hop2009]* and the MDAnalysis paper (because Hop is built on top of MDAnalysis) *[MDAnalysis2011]*. Thanks!

CHAPTER 5

Contents

5.1 Installation

Warning: This is legacy research code. It might not work at all. Use at your own risk. Feedback is very welcome through the issue tracker.

Note: Only Python 2.7 is currently supported.

5.1.1 Source installation

At the moment, only source installation is supported. Use pip. Download the tarball from https://github.com/Becksteinlab/hop/releases or do a web-install (choose the appropriate URL!):

```
pip install https://github.com/Becksteinlab/hop/archive/release-0.4.0.tar.gz
```

(This will also install all dependencies.)

5.1.2 Conda

At the moment, we do not have a conda package. However, it is easy to set up a working environment for *hop* and then do the *install from source* described above.

```
conda create -c conda-forge -n hop python=2.7 numpy scipy networkx MDAnalysis␣
↪matplotlib pygraphviz GridDataFormats
source activate hop
pip install https://github.com/Becksteinlab/hop/archive/release-0.4.0.tar.gz
```

You can then run all the `hop-*` scripts in this environment. (Exit the environment as usual with `source deactivate`.)

5.2 Background

`hop` is a collection of Python modules to analyze solvent dynamics in molecular dynamics (MD) simulations. It generates a spatially and temporally coarse grained representation of the dynamics in terms of a **hopping graph**.

The idea is to first find regions with a density above a given threshold and catalogue those **sites** (for water, these would be hydration sites, for other solvent molecules simply high density locations) . Once this is done, one can analyze water movement in terms of **hops between those sites**. The complicated solvation dynamics is thus represented as a **hopping graph** in which hydration sites are the nodes (or vertices) and movements between sites are the edges.

However, in principle one is not restricted to using high density sites. Any geometric partition of space can be used (such as "inside" and "outside" of a binding site to measure exchange with a binding site and derive on/off rate constants or "periplasmic" and "cytosolic" side of a membrane to derive permeation rates through a channel).

5.3 Quickstart: using the hop package — `hop.interactive`

A typical session starts with a trajectory (which should have been RMS-fitted to a reference structure). Any topology and trajectory file suitable for MDAnalysis can be used such as PSF+DCD, PDB+XTC or a single PDB. In the following Charmm/NAMD psf and dcd files are used as examples.

We will use the high-level wrapper functions in *`hop.interactive`*:

```
>>> import hop
>>> from hop.interactive import (make_density, analyze_density,
...                              make_hoppingtraj, build_hoppinggraph)
```

5.3.1 Hydration sites

Hydration sites are sites of water density higher than the bulk density but one special site is the bulk. The hydration sites and the bulk site are computed in two separate steps.

High density sites

First build the density of the water oxygens.

```
>>> density = make_density(psf,dcd,filename,delta=1.0)
```

The density is also saved as a pickled python object so that one can easily reload it. The density is also exported as a dx file for visualization (e.g. use *`hop.interactive.visualize_density()`*, which calls **VMD**).

From the density one creates the *site map* for a given threshold (by default this is a multiple of the water bulk density):

```
>>> density.map_sites(threshold=2.72)
```

Experiment with the threshold; `hop.analysis.DensityAnalysis` can help to systematically explore the parameter space, and it is also helpful to load the density into a visualization software such as VMD and interactively explore contour levels. Values between 1.65 and 3 have given decent results in the past but this is system-dependent.)

Bulk site

For a full analysis of hopping events one also needs to define a bulk site. This is currently accomplished by calculating a second *bulk density* (all water not within 3.5 Å of the protein) and manually inserting the bulk site into the site map for the first density.

```
>>> density_bulk = make_density(psf,dcd,'bulk',delta=1.0,
...          atomselection='name OH2',
...          soluteselection='protein and not name H*',
...          cutoff=3.5
...          )
```

Using VMD's VolMap can be potentially be faster — try it if the default seems too slow to you:

```
>>> density_bulk = make_density(psf,dcd,'bulk',delta=1.0,
...          atomselection='name OH2 and not within 3.5 of (protein and name not␣
↪hydrogen)',
...          backend='VMD',load_new=False)
```

The bulk density should be a big, well defined volume so we choose a fairly low threshold:

```
>>> density_bulk.map_sites(0.6)
```

Add the biggest bulk site:

```
>>> density.site_insert_bulk(density_bulk)
>>> density.save()
>>> del density_bulk
```

Note: Behind the scenes, the bulk is simply prepended to the list of all sites (`hop.sitemap.Density.sites`), found so far. By convention the site at position 1 in the list of all sites is treated specially in many parts of hop (it has the so-called sitelabel "1", which is simply the position in the list of sites) and hence you might encounter unexpected behaviour later if you do not insert a bulk site.

Statistics about the sites can be produced with

```
>>> analyze_density(density,figname)
```

The results figures will be named <*figname*>*.pdf*.

Remapping for comparing site maps

This section is only relevant if you plan on comparing site maps. Then you *must* compare the density to your reference density *now* before proceeding. You will

1. remap this density to be defined on the same grid as the reference density (for this to work, this density must have been generated from a trajectory that has been RMS-fitted to the same reference structure as; see hop.trajectory.rms_fit_trj() and hop.trajectory.fasta2select())

   ```
   >>> ref_density = hop.sitemap.Density(filename='my_reference_density')
   >>> remapped_density = hop.sitemap.remap_density(density,ref_density)
   ```

2. find the equivalence sites in the two densities and add those sites to **both** densities:

```
>>> remapped_density.find_equivalence_sites_with(ref_density,verbosity=3)
>>> remapped_density.save(<filename>)
>>> ref_density.save()
```

(You must also recalculate the reference densities hopping trajectory (see below) because some sites may have been merged into 'equivalence sites'. See docs for hop.sitemap.find_equivalence_sites_with() and hop.graph.CombinedGraph()).

From now on, work with the remapped density: >>> density = remapped_density

5.3.2 Hopping trajectory

Next we translate the dcd into a 'hopping trajectory' (saved in dcd format) in which coordinates for a given water oxygen are replaced by the site it visits at each time step.

```
>>> hops = make_hoppingtraj(density,'hop_water+bulk')
```

All further analysis should use this hopping trajectory (from disk) as it is computationally much cheaper to read the trajectory than to re-translate the coordinate trajectory (which is done behind the scences if the hopping trajectory is not available).

5.3.3 Hopping graph

The final step is to map out the graph of transitions between sites (using the hopping trajectory):

```
>>> tn = build_hoppinggraph(hops,density)
```

tn.hopgraph holds this graph (tn.graph just contains all jumps including the interstitial and off-sites). The edges of hopgraph are the rate constants k_ji (in 1/ps) for hops i –> j. They are computed from an exponential fit to the site survival function S_ji(t) for particles waiting to hop from i to j.

The density is provided to attach data to the nodes of the hopgraph. It is required for visualization and analysis (although not strictly necessary for the hopgraph itself).

Further analysis uses tn.hopgraph:

```
>>> h = tn.hopgraph            # main result is the 'hopgraph'
>>> h.save('hopgraph')         # save the hopping graph (necessary for cg part)
>>> h.filter(exclude={'outliers':True, 'Nmin':2, 'unconnected':True})
>>> h.show_rates()             # show all calculated rate constants (filtered graph)
>>> h.plot_fits(xrange(301))   # plot rate constant fits for t=0ps to 300ps
>>> h.plot_fits()
>>> h.export('water')          # write dot file to visualize (filtered) graph
```

To compare the water network based on density with another hop graph (based on ref_density), construct the CombinedGraph:

```
>>> h_ref = hop.graph.HoppingGraph(filename=<filename>) --- basically repeat steps␣
↪from
###                                                    --- ref_density only with␣
↪different labels
>>> cg = hop.graph.CombinedGraph(g0=h,g1=h_ref)
>>> cg.plot(0,'cg_h',linewidths=(0.01,))
>>> cg.plot(1,'cg_h_ref',linewidths=(0.01,))
```

5.3.4 Other topics

The following topics are not fully documented but the individual functions and classes contain some hints on how to make them work for the purposes outlined below:

- Remapping densities to a reference density (see *hop.sitemap.remap_density()*).
- Comparing densities and finding equivalence sites (see *hop.sitemap.find_common_sites()* and *hop.sitemap.Density.find_equivalence_sites_with()*).
- Comparing hopgraphs across different simulations: requires equivalence sites in both densities; then build the *hop.graph.CombinedGraph*.

5.3.5 Functions

hop.interactive.**analyze_density**(*density*, *figure='sitestats'*)
Site statistics based on the density alone.

Plots site volumes, average densities and occupancy, and writes it to the pdf file <figure>.pdf

hop.interactive.**build_hoppinggraph**(*hoppingtrajectory*, *density*)
Compute the graph of all site hops and calculate the rate constants.

tgraph = build_hoppinggraph(hops,density)

Arguments

hops hop.trajectory.HoppingTrajectory object

density hop.sitemap.Density object

Returns tgraph, a *hop.graph.TransportNetwork* object

hop.interactive.**build_hoppinggraph_fromfiles**(*hoppingtrajectory_filename*, *density_filename*)
Compute the TransportNetwork including HoppingGraph from files.

tn = build_hoppinggraph_fromfiles('hoptraj','water_density')

Input: hoppingtrajectory_filename filename for HoppingTrajectory psf and dcd density_filename filename for pickled Density

Output: tn hop.graph.TransportNetwork object (qv)

hop.interactive.**generate_densities**(**args*, ***kwargs*)
Analyze the trajectory and generate solvent and bulk density.

generate_densities(topol, traj, atomselection='name OW') –> densities

This function can take a long time because it has to read the whole trajectory. Progress is printed to the screen. It saves results to pickle files. These files are *hop.sitemap.Density* objects and can be used to instantiate such a density object.

Arguments

filename name of the solvent density with bulk site

bulkname bulk density

density_unit unit of measurement for densities and thresholds (Molar, nm^{-3}, Angstrom^{-3}, water, SPC, TIP3P, TIP4P)

solvent_threshold [exp(1) = 2.7182818284590451] hydration sites when density > this threshold

bulk_threshold [exp(-0.5) = 0.60653065971263342] bulk site are regions with density > this threshold (and water farther away from the protein heavy atoms than *cutoff*)

delta [1.0] cubic grid size in Angstrom

cutoff bulk-water is assumed to start at this distance from the soluteselection

soluteselection ["protein and not name H*"] how to select the solute (for bulk density)

Returns a dict containing *hop.sitemap.Density* instances for the the "solvent" and the "bulk" density; the "solvent" has the bulk site (largest site in "bulk") inserted as site 1.

Note: The "solvent" density is going to be used throughout the rest of the protocol. Should you ever remap the sites (i.e. run *map_sites()* with a different threshold) then **you must insert the bulk site again** (because the bulk site is removed for technical reasons whenever the sites change); use the saved bulk site and the *hop.sitemap.Density.site_insert_bulk()* method.

See also:

Keyword arguments are passed on to `hop.density.DensityCreator` where all possible keywords are documented; the site mapping is done with *hop.sitemap.Density.map_sites()*.

`hop.interactive.`**`hopgraph_basic_analysis`**(*h*, *density*, *filename*)
Do some simple analysis tasks on the hopgraph.

hopgraph_basic_analysis(h, density, filename)

Arguments

h hopgraph, a *hop.graph.HoppingGraph*

density density, a *hop.sitemap.Density*

filename default filename for generated files; all files and new directories are written in the directory pointed to by the path component

`hop.interactive.`**`make_density`**(*psf*, *dcd*, *filename*, *delta=1.0*, *atomselection='name OH2'*, ***kwargs*)
Build the density by histogramming all the water oxygens in a dcd.

density = make_density(psf,dcd,filename,delta=1.0)

The function builds the density object, writes it to disk, and also exports it as a dx file for visualization (use vizualize_density(density)).

Arguments

***psf** topology

dcd trajectory (should be RMS fitted to a reference frame)

filename default filename for the density

delta grid spacing Angstrom

***kwargs*:**

padding increase box dimensions for 3D histogramming by padding

soluteselection cutoff

for bulk density: setting both *soluteselection='protein and not name H*'* and *cutoff=3.5* A selects *'<atomsel> NOT WITHIN <cutoff> OF <solutesel>'*

Returns *density*, `hop.sitemap.Density` object; the density is converted to a fraction of the density of bulk TIP3P water

`hop.interactive.`**`make_hoppingtraj`**(*density*, *filename*, ***hopargs*)
Create the hopping trajectory from a density with a site map.

hops = make_hoptraj(density,filename)

Arguments

density density object with a site map

filename prefix for the hop trajectory files (psf and dcd)

hopargs keyword args to add to `HoppingTrajectory` such as *fixtrajectory = {'delta':10.22741474887299}*

This function relies on the density's metadata. In particular it uses *density.metadata['psf']* and *metadata['dcd']* to find its input data and *metadata['atomselection']* to define the atoms to track.

`hop.interactive.`**`make_xstal_density`**(*pdb*, *filename*, ***kwargs*)
Generate a density from the crystalwaters in a PDB.

For arguments see `make_density()`.

(These water are typically named HOH.)

See also:

Water molecules are counted as point-like particles. One can also use `hop.sitemap.BfactorDensityCreator` to broaden water molecules according to their B-factor.

`hop.interactive.`**`visualize_density`**(*density*)
Visualize the trajectory with the density in VMD.

visualize_density(density)

Arguments

density hop.sitemap.Density object

5.4 Hop package — `hop`

5.4.1 Generating a hopping graph

Defining solvation sites — `hop.sitemap`

Histogram positions of particles from a MD trajectory on a grid. Calculate the density, change units (both of the grid and of the density), save the density, export into 3D visualization formats, manipulate the density as a numpy array.

`class` `hop.sitemap.`**`Density`**(*grid=None*, *edges=None*, *filename=None*, *dxfile=None*, *parameters=None*, *unit=None*, *metadata=None*)
Class with an annotated density, i.e. additional information for each grid cell. Adds information about sites to the grid. A 'site' consists of all connected grid cells with a density >= threshold.

A site is defined as a set of at least 'MINsite' grid cells with density >= threshold that are located in each others' first and second nearest neighbour shell (of 26 cells, on the cubic lattice). A site is labelled by an integer 1..N.

The interstitial is labelled '0'. By default, a site may consist of a single grid cell (MINsite == 1) but this can be changed by setting the parameter MINsite to another number >1.

When neither grid nor edges are given then the density object can also be read from a pickled file (filename) or a OpenDX file (dxfile). In the latter case, care should be taken to properly set up the units and the isDensity parameter:

```
>>> g = Density(dxfile='bulk.dx',parameters={'isDensity':True,'MINsite':1},
                unit={'length':'Angstrom','density':'Angstrom^{-3}'}, ....)
```

Attributes:

grid density on a grid edges the lower and upper edges of the grid cells along the

three dimensions of the grid

map grid with cells labeled as sites (after label_sites()) sites list of sites: site 0 is the interstitial, then follows

the largest site, and then sites in decreasing order. Each site is a list of tuples. Each tuple is the index (i,j,k) into the map or grid.

graph NetworkX graph of the cells

unit physical units of various components P (default) values of parameters

Methods:

map_sites(threshold) label all sites, defined by the threshold. The threshold value is stored with the object as the default. The default can be explicitly set as P['threshold']

save(filename) save object.pickle load(filename) restore object.pickle (or use d=Density(filename=<filename>)) export() write density to a file for visualization export_map() write individual sites

Adds information about sites to the grid. Sites are all cells with a density >= threshold.

density = Density(kargs**)

Sets up a Grid with additional data, namely the site map The threshold is given as key-value pair in the parameters dictionary and is assumed to be in the same units as the density.

If the input grid is a histogram then it is transformed into a density.

When neither grid nor edges are given then the density object can also be read from a pickled file (filename) or a OpenDX file (dxfile). In the latter case, care should be taken to properly set up the units and the isDensity parameter if the dx file is a density:

```
>>> g = Density(dxfile='bulk.dx',parameters={'isDensity':True},
            unit={'length':'Angstrom','density':'Angstrom^{-3}'}, ....)
```

export3D (*filename=None*, *site_labels='default'*)

Export pdb and psf file of site centres for interactive visualization.

```
>>> density.export3D()
```

Arguments

filename prefix for output files: <filename>.psf, <filename>.pdb, and <filename>.vmd

site_labels selects sites (See site_labels())

The method writes a psf and a pdb file from the site map, suitable for visualization in, for instance, VMD. In addition, a VMD tcl file is produced. When it is sourced in VMD then the psf and pdb are loaded and site labels are shown next to the sites.

Sites are represented as residues of resname 'NOD'; each site is marked by one 'ATOM' (of type CA) at the center of geometry of the site.

Bulk and interstitial are always filtered from the list of sites because they do not have a well defined center.

export_map(*labels='default'*, *format='dx'*, *directory=None*, *value='density'*, *combined=False*, *verbosity=3*)

Write sites as a density file for visualization.

export_map(**kwargs)

labels='default' Select the sites that should be exported. Can be a list of numbers (site labels) or one of the keywords recognized by site_labels() (qv). The interstitial is always excluded.

combined=False True: write one file. False: write one file for each site.

format='dx' Only dx format supported directory='site_maps'

Files are created in new directory, 'site_maps' by default. File names are generated and indexed with the label of the site. By default, 'site_maps' is located in the same directory as the default filename.

value= 'density' Writes the actual density in the site.

'threshold' The densities have the threshold value wherever the site is defined. Note that the interstitial (label = 0) is also written.

<float> Writes the value <float> into the site.

verbosity=3 Set to 0 to disable status messages.

Quick hack to write out sites. Each site can be written as a separate density file (combined=False) so that one can distinguish them easily in say VMD. Display with

vmd site_maps/*.dx

find_equivalence_sites_with(*reference*, *fmt='%d*'*, *update_reference=True*, *use_ref_equivalencesites=False*, *verbosity=0*, *equivalence_graph='equivalence_graph.png'*)

Find overlapping sites with a reference density and update site descriptions.

Density.find_equivalence_sites_with(ref)

Arguments

ref a Density object defined on the same grid fmt python format string used for the equivalent_name, which should

contain %d for the reference label number (max 10 chars) (but see below for magical use of xray water names)

update_reference True (default): Also update the site_properties in the *reference* so that one can make graphs that highlight the common sites. (This is recommended.) False: don't change the reference

use_ref_equivalencesites True: use sites + equivalence sites from the reference density False*: remove all equivalence sites als from the ref density

verbosity For verbosity >= 3 output some statistics; verbosity >=5 also returns the equivalence graph for analysis; verbosity >= 7 displays the graph (and saves to equivalence_graph.png).

An 'equivalence site' is a site that contains all sites that overlap in real space with another site in the reference density. This also means that two or more sites in one density can become considered equivalent if they both overlap with a larger site in the other density, and it is also possible that one creates 'equivalence' chains (0,a) <-> (1,b) <-> (0,c) <-> (1,d) (although (0,a) ~<-> (1,d), and by construction (0,a) ~<-> (0,c) and (1,b) ~<-> (1,d)), leading to extensive equivalence sites.

When hopping properties are computed, an equivalence site is used instead of the individual sub sites.

The equivalence sites themselves are constructed as new sites and added to the list of sites; their site numbers are constructed by adding to the total number of existing sites. Sub-sites are marked up by an entry of the equivalence site's site number in site_properties.equivalence_site.

The common sites are consecutively numbered, starting at 2, from the one containing most sites to the one with fewest.

The method updates Density.site_properties.equivalent_name with the new descriptor of the equivalent site. Equivalent site names are consecutively numbered, starting at 2, and can be optionally formatted with the fmt argument.

However, if the reference density was built from an X-ray density AND if each site corresponds to single X-ray water molecule then the equivalence names contain the water identifiers eg 'W136' or 'W20_W34_W36'.

See the hop.sitemap.find_common_sites() function for more details.

has_bulk()
Returns `True` if a bulk site has been inserted and `False` otherwise.

map_hilo(*lomin=0.0*, *lomax=0.5*, *himin=2.72*)
Experimental mapping of low density sites together with high density ones.

Keywords

lomin low-density sites must have a density > *lomin* [0.0]

lomax low-density sites must have a density < *lomax* [0.5]

himin high-density sites must have a density > *himin* [2.72]

map_sites(*threshold=None*)
Find regions of connected density and label them consecutively

map_sites([threshold=<threshold>])

threshold Use the given threshold to generate the graph; the threshold is assumed to be in the same units as the density. (This updates the Density object's threshold value as well.)

The interstitial has label '0', the largest connected subgraph has '1' etc. The sites (i.e.the list of indices into map/grid) can be accesed as Density.sites[label].

masked_density(*density*, *site_labels*)
Returns only that portion of density that corresponds to sites; everything else is zeroed.

masked = masked_density(density,sites)

Arguments:

density a array commensurate with the map site_labels label or list of site labels

Results:

Returns numpy array of same shape as input with non-site cells zeroed.

remove_equivalence_sites()
Delete equivalence sites and recompute site map.

site_insert_bulk(*bulkdensity*, *bulklabel=1*, *force=False*)
Insert a bulk site from a different density map as bulk site into this density.

site_insert_bulk(bulkdensity)

This is a bit of a hack. The idea is that one can use a site from a different map (computed from the same trajectory with the same grid!) and insert it into the current site map to define a different functional region. Typically, the bulk site is the largest site in bulkdensity (and has site label 1) but if this is not the case manually choose the appropriate bulklabel.

The site is always inserted as the bulk site in the current density.

Example: >>> bulkdensity = hop.interactive.make_density(psf,dcd,'bulk',delta=1.0,

atomselection='name OH2 and not within 4.0 of protein')

```
>>> bulkdensity.map_sites(threshold=0.6)
>>> density.site_insert_bulk(bulkdensity)
>>> density.save()
>>> del bulkdensity
```

site_insert_nobulk()
Insert an empty bulk site for cases when this is convenient.

site_labels(*include='default'*, *exclude='default'*)
Return a list of site labels, possibly filtered.

L = site_labels(include=<inclusions>,exclude=<exclusions>)

<inclusions> and <exclusions> consist of a list of site labels (integers) and/or keywords that describe a site selection. All entries in one list are logically ORed. All exclusions are then removed from the inclusions and the final list of site labels is returned as a numpy array. (As a special case, the argument need not be a list but can be a single keyword or site label).

For convenience, some inclusions such as 'subsites' and 'equivalencesites' automatically remove themselves from the exclusions.

For standard use the defaults should do what you expect, i.e. only see the sites that are relevant or that have been mapped in a hopping trajectory.

Set verbosity to 10 in order to see the parsed selection.

<inclusions>

'all' all mapped sites, including bulk and subsites of equivalent sites (but read the NOTE below: set exclude=None)

'default' all mapped sites, including bulk but excluding subsites and interstitial

'sites' all mapped sites, excluding bulk and interstitial (removes 'subsites' and 'equivalencesites' from exclusions)

'subsites' all sites that have been renamed or aggreated into equivalence sites 'equivalencesites'

only the equivalence sites

int, list site label(s)

<exclusions>

'default' equivalent to ['interstitial','subsites']; always applied unless exludsions=None is set!

None do not apply any exclusions 'interstitial'

exclude interstitial (almost no reason to ever include it)

'subsites' exclude sites that have been aggregated or simply renamed as equivalence sites

'equivalencesites' exclude equivalence sites (and possibly include subsites)

'bulk' exclude the bulk site

Provides the ordered list L of site labels, excluding sites listed in the exclude list. Site labels are integers, starting from '0' (the interstitial). These labels are the index into the site_properties[] and sites[] arrays.

NOTE that by default the standard exclusions are already being applied to any 'include'; if one really wants all sites one has to set exclude=None.

Exclusions are applied _after_ inclusions.

'site' discards the bulk site, self.P['bulk_site']; this parameter is automatically set when adding the bulk site with site_insert_bulk().

See find_equivalence_sites_with() for more on equivalence sites and subsites.

site_occupancy (***labelargs*)
Returns the labels and the average/stdev occupancy of the labeled site(s).

labels, <N>, std(N) = site_coocupancy(include='all' | <int> | <list>)

Average occupancy is the average number of water molecules on the site i:

<N_i> = <n_i> * V_i

where n_i is the average density of the site and V_i its volume.

The label selection arguments are directly passed to site_labels() (see doc string).

If the interstitial is included then 0,0 is returned for the interstitial site (so ignore those numbers).

site_remove_bulk (*force=False*)
Cleanup bulk site.

site_volume (***labelargs*)
Returns the label(s) and volume(s) of the selected sites.

labels, volumes = site_volume('all')

The volume is calculated in the unit set in unit['length']. The label selection arguments are directly passed to site_labels() (see doc string).

The volume of the interstitial (if included) is returned as 0 (which is not correct but for technical reasons more convenient).

stats (*data=None*)
Statistics for the density (excludes bulk, interstitial, subsites).

d = stats([data=dict])

subsites_of (*equivsites*, *kind='sitelabel'*)
Return subsites of given equivalence sites as a dict.

dict <– subsites_of(equivsites,kind='sitelabel')

The dict is indexed by equivsite label. There is one list of subsites for each equivsitelabel.

kind 'sitelabel': equivsites are the sitelabels as uses internally; this is

the default because site_labels() returns these numbers and so one can directly use the output from site_labels() as input (see example)

'equivlabel': equivsites are treated as labels of equivalence sites; these are integers N that typically start at 2

'name': equivsites are treated as strings that are given as names to sites; the default settings produce something like 'N*'

EXAMPLES:

dens.subsites_of(dens.site_labels('equivalencesites')) dens.subsites_of([2,5,10], kind='equivsites') dens.subsites_of('10*', kind='name')

NOTE: * equivlabel == 0 is silently filtered (it is used as a merker for NO equivalence

site)

- empty equivalence sites show up as empty entries in the output dict; typically this means that one gave the wrong input or kind

class hop.sitemap.**Grid**(*grid=None*, *edges=None*, *filename=None*, *dxfile=None*, *parameters=None*, *unit=None*, *metadata=None*)

Class to manage a multidimensional grid object.

The grid (Grid.grid) can be manipulated as a standard numpy array. Changes can be saved to a file using the save() method. The grid can be restored using the load() method or by supplying the filename to the constructor.

The attribute Grid.metadata holds a user-defined dictionary that can be used to annotate the data. It is saved with save().

The export(format='dx') method always exports a 3D object, the rest should work for an array of any dimension.

Create a Grid object from data.

From a numpy.histogramdd(): g = Grid(grid,edges)

From files (created with Grid.save(<filename>): g = Grid(filename=<filename>)

From a dx file: g = Grid(dxfile=<dxfile>)

Arguments:

grid histogram or density and ... edges list of arrays, the lower and upper bin edges along the axes

(both are output by numpy.histogramdd())

filename file name of a pickled Grid instance (created with Grid.save(filename))

dxfile OpenDX file parameters dictionary of class parameters; saved with save()

isDensity False: grid is a histogram with counts, True: a density. Applying Grid.make_density() sets it to True.

unit dict(length='Angstrom', density=None) length: physical unit of grid edges (Angstrom or nm) density: unit of the density if isDensity == True or None

metadata a user defined dictionary of arbitrary values associated with the density; the class does not touch metadata[] but stores it with save()

Returns: g a Grid object

If the input histogram consists of counts per cell then the make_density() method converts the grid to a physical density. For a probability density, divide it by grid.sum() or use normed=True right away in histogramdd().

If grid, edges, AND filename are given then the extension-stripped filename is stored as the default filename.

NOTE:

- It is suggested to construct the Grid object from a histogram, to supply the appropriate length unit, and to use make_density() to obtain a density. This ensures that the length- and the density unit correspond to each other.

TODO: * arg list is still messy * probability density not supported as a unit

centers()
Returns the coordinates of the centers of all grid cells as an iterator.

convert_density(*unit='Angstrom^{-3}'*)
Convert the density to the physical units given by unit

unit can be one of the following:

name	description of the unit
Angstrom^{-3}	particles/A**3
nm^{-3}	particles/nm**3
SPC	density of SPC water at standard conditions
TIP3P	... see MDAnalysis.units.water
TIP4P	... see MDAnalysis.units.water
water	density of real water at standard conditions (0.997 g/cm**3)
Molar	mol/l

Note: (1) This only works if the initial length unit is provided.

2. Conversions always go back to unity so there can be rounding and floating point artifacts for multiple conversions.

There may be some undesirable cross-interactions with convert_length...

convert_length(*unit='Angstrom'*)
Convert Grid object to the new unit:

unit Angstrom, nm

This changes the edges but will not change the density; it is the user's responsibility to supply the appropriate unit if the Grid object is constructed from a density. It is suggested to start from a histogram and a length unit and use make_density().

export(*filename=None*, *format='dx'*)
export density to file using the given format; use 'dx' for visualization.

export(filename=<filename>,format=<format>)

The <filename> can be omitted if a default file name already exists for the object (e.g. if it was loaded from a file or it was saved before.) Do not supply the filename extension. The correct one will be added by the method.

The default format for export() is 'dx'.

Only implemented formats:

dx OpenDX (WRITE ONLY) python pickle (use Grid.load(filename) to restore); Grid.save()

is simpler than export(format='python').

importdx(*dxfile*)
Initializes Grid from a OpenDX file.

make_density()
Convert the grid (a histogram, counts in a cell) to a density (counts/volume).

make_density()

Note: (1) This changes the grid irrevocably.

2. For a probability density, manually divide by grid.sum().

hop.sitemap.**find_common_sites**(*a*, *b*, *use_equivalencesites=None*)
Find sites that overlap in space in Density a and b.

m = find_common_sites(a,b)

Arguments

a Density instance b Density instance

Returns

array of mappings between sites in a and b that overlap m[:,0] site labels in a m[:,1] site labels in b dict(m) translates labels in a to labels in b dict(m[:,[1,0]])

translates labels in b to labels in a

hop.sitemap.**find_overlap_coeff**(*a*, *b*)
Find sites that overlap in space in Density a and b.

m = find_overlap_coeff(a,b)

Arguments

a Density instance b Density instance

Returns

array sites in a and b that overlap and array of probability of overlap for overlapped sites

m[:,0] site labels in a m[:,1] site labels in b oc amount of overlap

hop.sitemap.**remap_density**(*density*, *ref*, *verbosity=0*)
Transform a Density object to a grid given by a reference Density.

```
>>> newdensity = remap_density(old,ref)
```

The user is repsonsible to guarantee that: * the grid spacing is the same in both densities * the grids only differ by a translation, not a rotation

Arguments

old Density object with site map ref reference Density object that provides the new grid shape verbosity=0 increase to up to 3 for status and diagnostic messages

Returns

newdensity Density object with old's density and site map transformed to ref's coordinate system. It is now possible to manipulate newdensity's and ref's arrays (grid and map) together, e.g.

```
>>> common = (newdensity.map > 1 & ref.map > 1)
>>> pairs = newdensity.map[common], ref.map[common]
```

Note that this function is not well implemented at the moment and can take a considerable amount of time on bigger grids (100x100x100 take about 3 Min).

An implicit assumption is that the two coordinate systems for the two grids are parallel and are only offset by a translation. This cannot be checked based on the available data and must be guaranteed by the user. RMS-fitting the trajectories is sufficient for this to hold.

BUGS:

- This is not a good way to do the remapping: It requires parallel coordinate systems and the exact same delta.
- It is slow.
- It would be much better to interpolate density on the reference grid,

`hop.sitemap.`**`unique_tuplelist`**`(`*x*`)`
Sort a list of tuples and remove all values None

Generating densities from trajectories — `hop.density`

As an input a trajectory is required that

1. Has been centered on the protein of interest.
2. Has all molecules made whole that have been broken across periodic boundaries.
3. Has the solvent molecules remap so that they are closest to the solute (this is important when using funky unit cells such as dodechedra or truncated octahedra).

Classes and functions

`class` `hop.density.`**`BfactorDensityCreator`**`(`*pdb*, *delta=1.0*, *atomselection='resname HOH and name O'*, *metadata=None*, *padding=1.0*, *sigma=None*`)`
Create a density grid from a pdb file using MDAnalysis.

dens = BfactorDensityCreator(psf,pdb,...).PDBDensity()

The main purpose of this function is to convert crystal waters in an X-ray structure into a density so that one can compare the experimental density with the one from molecular dynamics trajectories. Because a pdb is a single snapshot, the density is estimated by placing Gaussians of width sigma at the position of all selected atoms.

Sigma can be fixed or taken from the B-factor field, in which case sigma is taken as sqrt(3.*B/8.)/pi.

TODO:

- Make Gaussian convolution more efficient (at least for same sigma) because right now it is VERY slow (which may be acceptable if one only runs this once)
- Using a temporary Creator class with the PDBDensity() helper method is clumsy (but was chosen as to keep the PDBDensity class clean and __init__ compatible with Density).

See also:

- `MDAnalysis.analysis.density`
- *`PDBDensity`*

Construct the density from psf and pdb and the atomselection.

pdb [str] PDB file or `MDAnalysis.Universe`;

atomselection [str] selection string (MDAnalysis syntax) for the species to be analyzed

delta [float] bin size for the density grid in Angstroem (same in x,y,z) [1.0]

metadata [dict] dictionary of additional data to be saved with the object

padding [float] increase histogram dimensions by padding (on top of initial box size)

sigma [float] width (in Angstrom) of the gaussians that are used to build up the density; if `None` (the default) then uses B-factors from pdb

For assigning X-ray waters to MD densities one might have to use a sigma of about 0.5 A to obtain a well-defined and resolved x-ray water density that can be easily matched to a broader density distribution.

The following creates the density with the B-factors from the pdb file:

```
DC = BfactorDensityCreator(pdb, delta=1.0, atomselection="name HOH",
                           padding=2, sigma=None)
density = DC.Density()
```

`density_from_PDB()` for a convenience function

PDBDensity(*threshold=None*)
Returns a PDBDensity object.

The PDBDensity is a Density with a xray2psf translation table; it has also got an empty bulk site inserted (so that any further analysis which assumes that site number 1 is the bulk) does not discard a valid site.

threshold Use the given threshold to generate the graph; the threshold is assumed to be in the same units as the density. None: choose defaults (1.0 if bfactors were used, 1.3 otherwise)

class hop.density.**DensityCollector**(*name*, *universe*, ***kwargs*)
Collect subsequent coordinate frames to build up a `Density`.

class hop.density.**PDBDensity**(*grid=None*, *edges=None*, *filename=None*, *dxfile=None*, *parameters=None*, *unit=None*, *metadata=None*)
Density with additional information about original crystal structure.

This is simply the Density class (see below) enhanced by the add_xray2psf(), W(), and Wequiv() methods.

Note that later analysis often ignores the site with the bulknumber by default so one should (after computing a site map) also insert an empty bulk site:

canonical way to build a PDBDensity # (builds the sitepa at threshold and inserts a pseudo bulk site) xray = BfactorDensityCreator(. . .).PDBDensity(threshold)

rebuild site map xray.map_sites(threshold) # map sites at density cutoff threshold xray.site_insert_nobulk() # insert 'fake' bulk site at position SITELABEL['bulk']

find X-ray waters that correspond to a site in another density Y: # (1) build the list of equivalence sites, using the x-ray density as reference Y.find_equivalence_sites(xray) # also updates equiv-sites in xray! # (2) look at the matches in xray xray.Wequiv() TODO: not working yet

Density Class

Class with an annotated density, i.e. additional information for each grid cell. Adds information about sites to the grid. A 'site' consists of all connected grid cells with a density >= threshold.

A site is defined as a set of at least 'MINsite' grid cells with density >= threshold that are located in each others' first and second nearest neighbour shell (of 26 cells, on the cubic lattice). A site is labelled by an integer 1..N. The interstitial is labelled '0'. By default, a site may consist of a single grid cell (MINsite == 1) but this can be changed by setting the parameter MINsite to another number >1.

When neither grid nor edges are given then the density object can also be read from a pickled file (filename) or a OpenDX file (dxfile). In the latter case, care should be taken to properly set up the units and the isDensity parameter:

```
>>> g = Density(dxfile='bulk.dx',parameters={'isDensity':True,'MINsite':1},
                unit={'length':'Angstrom','density':'Angstrom^{-3}'}, ....)
```

Attributes:

grid density on a grid edges the lower and upper edges of the grid cells along the

three dimensions of the grid

map grid with cells labeled as sites (after label_sites()) sites list of sites: site 0 is the interstitial, then follows

the largest site, and then sites in decreasing order. Each site is a list of tuples. Each tuple is the index (i,j,k) into the map or grid.

graph NetworkX graph of the cells

unit physical units of various components P (default) values of parameters

Methods:

map_sites(threshold) label all sites, defined by the threshold. The threshold value is stored with the object as the default. The default can be explicitly set as P['threshold']

save(filename) save object.pickle load(filename) restore object.pickle (or use d=Density(filename=<filename>)) export() write density to a file for visualization export_map() write individual sites

Adds information about sites to the grid. Sites are all cells with a density >= threshold.

density = Density(kargs**)

Sets up a Grid with additional data, namely the site map The threshold is given as key-value pair in the parameters dictionary and is assumed to be in the same units as the density.

If the input grid is a histogram then it is transformed into a density.

When neither grid nor edges are given then the density object can also be read from a pickled file (filename) or a OpenDX file (dxfile). In the latter case, care should be taken to properly set up the units and the isDensity parameter if the dx file is a density:

```
>>> g = Density(dxfile='bulk.dx',parameters={'isDensity':True},
            unit={'length':'Angstrom','density':'Angstrom^{-3}'}, ....)
```

W (*N*, *returntype='auto'*, *format=False*)

Returns the resid of water N.

If returntype == 'psf' then N is interpreted as the resid in the x-ray crystal structure (or original pdb file) and a resid N' in the psf is returned.

If returntype == 'xray' then N is a resid in the psf and the corresponding crystal structure water is returned. This is useful to label water molecules by their published identifier, eg 'W128'.

If the returntype is set to 'auto' and N starts with a W (eg 'W128') then it is assumed to be a crystal water and the returntype is automatically set to psf, otherwise it acts like 'xray'.

Arguments

N resid of molecule (can be an iterable) returntype 'auto' | 'psf' | 'xray' format False: return a integer number

True: default string (either "WN'" for x-ray or "#N'" for psf) python format string: if the string contains %(resid)d then the string

will be used as a format, otherwise the bare number is returned without raising an error

Wequiv (*format=True*)
Return a list of the PDB resids of the equivalent sites.

array = Wequiv(format=True)

format True: array of identifiers 'Wnn' False: array of integers string: python format string; %(resid)d is replaced

add_xray2psf (*pdbfile*, *regex='\\s*W\\s*|HOH|WAT|.*TIP.*|.*SPC.*'*)
Add translation table between sequential psf numbering and original pdb numbering for water.

D.add_xray2psf(pdbfilename)

The original pdb is read and all water molecules are sequentially mapped to the water molecules in the psf (without any checks). The pdb is read and analyzed using Bio.PDB.

pdbfilename Original crystallographic pdb file regex extended regular expression to detect water residues

equivalence_sites (*format=True*)
All equivalence sites (if defined) together with crystallographic water labels.

recarray <– equivalence_sites(self,format=True)

The numpy.recarray has columns equivalence_label the integer label of the equivalence site equivalence_name the name, a string xray the identifier of the X-ray water

equivalence_label and equivalence_name are identical between the densities from which the equivalence sites were computed. The xray identifier is specific for the structure; by default it is a string such as 'W135'.

format True: print 'W<N>' identifier False: integer <N> (see W() for more possibilities)

BUG: THIS IS NOT WORKING AS THOUGHT BECAUSE THERE IS NO 1-1 MAPPING BETWEEN WATER MOLECULES AND SITES AND BECAUSE SITES ARE NOT NUMBERED IN THE SAME ORDER AS THE WATER MOLECULES

TODO: The proper way to do this is to find all water molecules within a cutoff of each grid cell that belongs to a site and then store all the waters as the string name of the site.

site2resid (*sitelabel*)
Returns the resid of the particle that provided the density for the site.

site_insert_nobulk ()
Insert an empty bulk site for cases when this is convenient.

hop.density.**density_from_Universe** (**args*, ***kwargs*)
Create a `hop.sitemap.Density from a :class:`Universe`.

See also:

`MDAnalysis.analysis.density.density_from_Universe()` for all parameters and *`density_from_trajectory()`* for a convenience wrapper.

hop.density.**density_from_trajectory** (**args*, ***kwargs*)
Create a density grid from a trajectory.

density_from_trajectory(PSF, DCD, delta=1.0, atomselection='name OH2', ...) –> density

or

density_from_trajectory(PDB, XTC, delta=1.0, atomselection='name OH2', ...) –> density

Arguments

psf/pdb/gro topology file

dcd/xtc/trr/pdb trajectory; if reading a single PDB file it is sufficient to just provide it once as a single argument

Keywords

atomselection selection string (MDAnalysis syntax) for the species to be analyzed ["name OH2"]

delta approximate bin size for the density grid in Angstroem (same in x,y,z) (It is slightly adjusted when the box length is not an integer multiple of delta.) [1.0]

metadata dictionary of additional data to be saved with the object

padding increase histogram dimensions by padding (on top of initial box size) in Angstroem [2.0]

soluteselection MDAnalysis selection for the solute, e.g. "protein" [`None`]

cutoff With *cutoff*, select '<atomsel> NOT WITHIN <cutoff> OF <soluteselection>' (Special routines that are faster than the standard AROUND selection) [0]

verbosity: int level of chattiness; 0 is silent, 3 is verbose [3]

Returns *hop.sitemap.Density*

TODO

- Should be able to also set skip and start/stop for data collection.

Note:

- In order to calculate the bulk density, use

  atomselection='name OH2',soluteselection='protein and not name H*',cutoff=3.5

  This will select water oxygens not within 3.5 A of the protein heavy atoms. Alternatively, use the VMD-based `density_from_volmap()` function.
- The histogramming grid is determined by the initial frames min and max.
- metadata will be populated with psf, dcd, and a few other items. This allows more compact downstream processing.

See also:

docs for `MDAnalysis.analysis.density.density_from_Universe()` (defaults for kwargs are defined there).

`hop.density.`**`print_combined_equivalence_sites`**(*target*, *reference*)
Tabulate equivalence sites of target against the reference.

BUG: THIS IS NOT WORKING (because the assignment sites <–> waters is broken)

Using qhull to define regions for hopping analysis — `hop.qhull`

Interface to some functions of the **'qhull'**_ (or rather the qconvex) program. **'qhull'**_ must be installed separately (see links).

The main functionality is to define a region in space within the convex hull of a protein. The hull is typically defined by a selection of atoms and written as a "density" file for use in `hop`.

Example

In this example the convex hull of the C-alpha atoms is computed. Initially, points must be extracted from the structure to a file:

```
hop.qhull.points_from_selection(psf='protein.psf', pdb='protein.pdb', filename='ca_100
↪%.dat')
```

and saved to file `ca_100%.dat`.

This is usually too large and also entails regions of the hydration shell outside of interal cavities. A relatively robust workaround for roughly globular proteins is to shrink the convex hull, using the `scale` argument of `hop.qhull.make_ca_points()`. Shrinking to 70% appears to be a good starting point:

```
hop.qhull.points_from_selection(psf='protein.psf', pdb='protein.pdb', filename='ca_70
↪%.dat', scale=0.7)
```

The convex hull itself is generated from the datafile of the points:

```
Q70 = hop.qhull.ConvexHull('ca_70%.dat', workdir='cavity70%')
```

Another density grid `b` (such as a real water density for the bulk) is currently required to generate a pseudo density based on the convex hull. The real density provides the grid on which the convex hull is mapped:

```
b = hop.sitemap.Density(filename='bulk')
QD70 = Q70.Density(b)
```

(This maps out sites at the threshold level set in `b`; change it with the *hop.sitemap.Density.map_sites()* method if required.)

Insert a bulk density:: QD70.site_insert_bulk(b)

class hop.qhull.**ConvexHull**(*coordinates*, *workdir=None*, *prefix=None*)

The convex hull of a set of points.

The convex hull is calculated with the **'qhull'_** program.

Compute convex hull and populate data structures.

Arguments

- coordinates: input suitable for qconvex
- workdir: store intermediate files in workdir (tmp dir by default)
- prefix: filename prefix for intermediate output files

Density(*density*, *fillvalue=None*)

Create a Density object of the interior of the convex hall.

Uses another Density object *density* as a template for the grid.

Note: This is rather slow and should be optimized.

---

point_inside(*point*)

Check if point [x,y,z] is inside the polyhedron defined by planes.

Iff for all i: plane[i]([x,y,z]) = n*[x,y,z] + p < 0 <==> [x,y,z] inside

(i.e. [x,y,z] is under *all* planes and the planes completely define the enclosed space

points_inside(*points*)
Return bool array for all points:

True: inside False: outside

Arguments

- points = [[x1,y1,z1], . . .] or an iterator that supplies points
- planes: normal forms of planes

Returns

Array with truth values such as [True, False, True, . . .]

read_planes()
Read planes from qconvex n file.

Numpy array [[n1,n2,n3,-p], . . .] for planes n*x = -p.

Planes are oriented and point outwards.

read_vertices()
Read vertices from qconvex p file.

Numpy array of points [[x,y,z], . . .]

wd(**args*)
Return path in workdir.

class hop.qhull.**VertexPDBWriter**(*filename*)
PDB writer that implements a subset of the PDB 3.2 standard. http://www.wwpdb.org/documentation/format32/v3.2.html

ATOM(*serial=None*, *name=None*, *altLoc=None*, *resName=None*, *chainID=None*, *resSeq=None*, *iCode=None*, *x=None*, *y=None*, *z=None*, *occupancy=1.0*, *tempFactor=0.0*, *element=None*, *charge=0*)
Write ATOM record. http://www.wwpdb.org/documentation/format32/sect9.html Only some keword args are optional (altLoc, iCode, chainID), for some defaults are set.

All inputs are cut to the maximum allowed length. For integer numbers the highest-value digits are chopped (so that the serial and reSeq wrap); for strings the trailing characters are chopped.

Note: Floats are not checked and can potentially screw up the format.

REMARK(**remark*)
Write generic REMARK record (without number). http://www.wwpdb.org/documentation/format32/remarks1.html http://www.wwpdb.org/documentation/format32/remarks2.html

TITLE(**title*)
Write TITLE record. http://www.wwpdb.org/documentation/format32/sect2.html

write(*coordinates*, *name='CA'*, *resname='VRT'*, *resid=1*)
Write coordinates as CA.

hop.qhull.**points_from_selection**(**args*, ***kwargs*)
Create a list of points from selected atoms in a format suitable for `qhull`.

points_from_selection(topology, structure, selection="name CA", filename="points.dat", scale=None)

Arguments

- psf: Charmm topology file

- pdb: coordinates
- selection: MDAnalysis select_atoms() selection string [C-alpha atoms]
- filename: name of the output file; used as input for *ConvexHull*
- scale: scale points around the centre of geometry; values of 0.5 - 0.7 typically ensure that the convex hull is inside the protein; default is to not to scale, i.e. scale = 1.

hop.qhull.**write_coordinates**(*filename*, *points*, *scale=None*)
Write an array of points to a file suitable for qhull.

Generating the hopping trajectory — hop.trajectory

Based on a definition of grid sites, convert a molecular dynamics trajectory into a trajectory of site hops.

You will also need the following modules to create the input for HoppingTraj: *hop.sitemap*.

Classes

class hop.trajectory.**HoppingTrajectory**(*trajectory=None*, *group=None*, *density=None*, *filename=None*, *hopdcd=None*, *hoppsf=None*, *fixtrajectory=None*, *verbosity=3*)
Provides a time-sequence of sites visited by individual molecules, called a 'hopping trajectory' because the molecules hop between sites. Their coordinates are mapped to site labels, which have been defined on a grid previously (using hop.sitemap).

Output format

For simplicity and code reusal this is again a dcd with the site as the x-coordinate; the y coordinate is set to the 'orbit site', i.e. it records the site the particle was last at for as long as it does not enter a new site. It describes the site in whose 'basin of attraction' the particle orbits. Note, however, that the transition to a new site is still counted as belonging to the previous site (which is arguably incorrect); the hop.graph module, however, does a proper analysis, which is cannot be done here for efficieny reasons. The z field is unused at the moment and set to 0.

Attributes

ts MDAnalysis.Timestep object n_frames number of frames in hopping trajectory group AtomGroup of atoms that are tracked

Methods

[start:stop] object can be used as an iterator over the ## hopping trajectory (disabled du to problems when doing random ## access on large dcds; either a bug in DCDReader or python) next() advances time step in the hopping trajectory map_dcd() iterator that updates the ts and maps the trajectory

coordinates to site labels

_map_next_timestep() map next coordinate trajectory step to hopping time step _read_next_timestep() read next timestep from hopping trajectory

write() write the hopping trajectory to a dcd file + psf write_psf() write a dummy psf for visualization

Converts a trajectory into a hopping trajectory, using a sitemap as an index for sites.

```
>>> h = HoppingTrajectory(trajectory=DCDReader,group=AtomGroup,density=Density,
                          fixtrajectory=<dict>,verbosity=3)
>>> h = HoppingTrajectory(filename=<name>)
```

Create from a coordinate trajectory of a group of atoms and a site map:

u = MDAnalysis.Universe(psf,dcd) water = u.select_atoms('name OH2') h = HoppingTrajectory(trajectory=u.trajectory,group=water,density=water_density)

Load from a saved hopping trajectory (in dcd format with dummy psf)

h = HoppingTrajectory(hopdcd='hops.trajectory',hoppsf='hops.psf')

Arguments

trajectory MDAnalysis.trajectory trajectory instance group MDAnalysis.group instance density grid3Dc.Grid instance with sitemap set

hopdcd dcd written by write() hoppsf psf written by write() (or write_psf()) filename or simply provide one filename prefix for psf and dcd

fixtrajectory dictionary with attributes of a dcd object and new values; used to provide correct values after using a catdcd-generated trajectory (hack!), e.g. fixtrajectory = {'delta':10.22741474887299}

verbosity show status messages for >= 3

filename (*filename=None*, *ext=None*, *set_default=False*, *use_my_ext=False*)

Supply a file name for the object.

fn = filename() —> <default_filename> fn = filename('name.ext') —> 'name' fn = filename(ext='pickle') —> <default_filename>'.pickle' fn = filename('name.inp','pdf') –> 'name.pdf' fn = filename('foo.pdf',ext='png',use_my_ext=True) –> 'foo.pdf'

The returned filename is stripped of the extension (use_my_ext=False) and if provided, another extension is appended. Chooses a default if no filename is given. Raises a ValueError exception if no default file name is known.

If set_default=True then the default filename is also set.

use_my_ext=True lets the suffix of a provided filename take priority over a default ext(tension).

map_dcd (*start=None*, *stop=None*, *step=None*)

Generator to read the trajectory from start to stop and map positions to sites.

ts = map_dcd(**kwargs)

Arguments: start starting frame number (None means first) stop last frame to read (exclusive) (None means last)

(Those are arguments to dcd[start:stop].)

Iterator Returns: ts hopping trajectory timestep object (iterator)

next ()

Provides the next time step of a hopping trajectory.

ts = next()

If a hopping trajectory file exists then this is used. Otherwise, the coordinate trajectory is mapped on the fly (which is computationally more expensive).

ts

Timestep of the hoptraj

write (*filename*, *start=None*, *step=None*, *delta=None*, *load=True*)

Write hopping trajectory as standard dcd file, together with a minimal psf.

write('hop')

Arguments:

load = True Immediately loads the trajectory so that further calls to next() will use the computed trajectory and don't use expensive mapping.

Ignore the other options and leave them at the defaults. Currently, only the whole trajectory is written. For visualization one also needs the dummy psf of the group.

Results:

filename.trajectory and filename.psf

Note that it is your responsibility to load the hopping trajectory and the appropriate psf together as there is very limited information stored in the dcd itself.

write_psf(*filename*)

Write a dummy psf just for the atoms in the selected group so that one can visualize the hopping trajectory.

write_psf(filename)

The psf is NOT a fully functional psf. It only contains the header and the ATOMS section. It is sufficient to display the hopping trajectory in VMD and can be read in by the MDAnalysis tools in order to store the atom numbers for the hopping trajectory.

Format from psffres.src

CHEQ: II,LSEGID,LRESID,LRES,TYPE(I),IAC(I),CG(I),AMASS(I),IMOVE(I),ECH(I),EHA(I)

standard format: (I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,I4,1X,2G14.6,I8,2G14.6)
(I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,2G14.6,I8,2G14.6) XPLOR

expanded format EXT: (I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,A8,1X,I4,1X,2G14.6,I8,2G14.6)
(I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,A8,1X,A4,1X,2G14.6,I8,2G14.6) XPLOR

no CHEQ: II,LSEGID,LRESID,LRES,TYPE(I),IAC(I),CG(I),AMASS(I),IMOVE(I)

standard format: (I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,I4,1X,2G14.6,I8)
(I8,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,A4,1X,2G14.6,I8) XPLOR

expanded format EXT: (I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,A8,1X,I4,1X,2G14.6,I8)
(I10,1X,A8,1X,A8,1X,A8,1X,A8,1X,A8,1X,A4,1X,2G14.6,I8) XPLOR

class hop.trajectory.**TAPtrajectory**(*trajectory=None*, *group=None*, *TAPradius=2.8*, *TAPsteps=3*, *filename=None*, *dcd=None*, *psf=None*, *fixtrajectory=None*, *verbosity=3*)

Provides a Time-Averaged Position (TAP) version of the input trajectory.

The method is described in Henchman and McCammon, J Comp Chem 23 (2002), 861 doi:10.1002/jcc.10074

Attributes

ts MDAnalysis.Timestep object n_frames number of frames in TAP trajectory group AtomGroup of atoms that are tracked

Methods

[start:stop] object can be used as an iterator over the ## hopping trajectory (disabled due to dcdreader bug) next() advances time step in the hopping trajectory map_dcd() iterator that updates the ts and maps the trajectory

coordinates to site labels

_map_next_timestep() map next coordinate trajectory step to hopping time step _read_next_timestep() read next timestep from hopping trajectory

write() write the hopping trajectory to a dcd file + psf

A TAP trajectory object converts a trajectory into a TAP trajectory.

Create from a coordinate trajectory of a group of water residues:

u = MDAnalysis.Universe(psf,dcd) water = u.select_atoms('resname TIP*') # see NOTE below!! water = u.select_atoms('name OH2') # better, see NOTE below!! h = TAPtrajectory(trajectory=u.trajectory,group=water)

Load from a saved hopping trajectory (in dcd format with dummy psf)

h = TAPtrajectory(dcd='TAP.trajectory',psf='TAP.psf')

The given atom group is filtered according to the Time-Averaged Positon algorithm (Henchman and McCammon, J Comp Chem 23 (2002), 861). Original positions are replaced by their TAPs: A particles last position (TAP) is retained unless it has moved farther than TAPradius from its TAP measured by its root mean square distance over the last TAPsteps frames.

One can use a TAP filtered trajectory 'on-the-fly' to build the density:

u = Universe(psf,dcd) oxy = u.select_atoms('name OH2') TAP = TAPtrajectory(u.trajectory,oxy) u.trajectory = TAP.trajectory # <— replace orig dcd with TAP !! dens = hop.density.density_from_Universe(u,atomselection='name OH2')

NOTE: In the current implementation residues are often ripped apart because all coordinates are processed independently. It is recommended to only do TAP on the water oxygens (for speed). This will create a trajectory in which hydrogens are always ripped from the oxygen but this trajectory is ONLY being used for creating a density from those oxygen using hop.sitemap.build_density().

(This could be fixed at the cost of speed; in this case TAP would be done on the centre of mass and the whole residue would be translated.)

Arguments

trajectory MDAnalysis.trajectory trajectory instance group MDAnalysis.group instance (from the same Universe as trajectory) TAPradius particles are considered to be on the TAP as long as they

haven't moved farther than TAPradius over the last TAPsteps frames

TAPsteps RMS distance of particle from TAP over TAPsteps is compared to TAPradius

dcd dcd written by write() psf psf written by write() (or write_psf()) filename or simply provide one filename prefix for psf and dcd

fixtrajectory dictionary with attributes of a dcd object and new values; used to provide correct values after using a catdcd-generated trajectory (hack!), e.g. fixtrajectory = {'delta':10.22741474887299}

verbosity show status messages for >= 3

filename(*filename=None*, *ext=None*, *set_default=False*, *use_my_ext=False*)

Supply a file name for the object.

fn = filename() —> <default_filename> fn = filename('name.ext') —> 'name' fn = filename(ext='pickle') —> <default_filename>'.pickle' fn = filename('name.inp','pdf') –> 'name.pdf' fn = filename('foo.pdf',ext='png',use_my_ext=True) –> 'foo.pdf'

The returned filename is stripped of the extension (use_my_ext=False) and if provided, another extension is appended. Chooses a default if no filename is given. Raises a ValueError exception if no default file name is known.

If set_default=True then the default filename is also set.

use_my_ext=True lets the suffix of a provided filename take priority over a default ext(tension).

map_dcd(*start=None*, *stop=None*, *skip=1*)

Generator to read the trajectory from start to stop and map positions to TAP sites.

ts = map_dcd(**kwargs)

Arguments: start starting frame number (None means first) stop last frame to read (exclusive) (None means last)

(Those are arguments to dcd[start:stop].)

Iterator Returns: ts hopping trajectory timestep object (iterator)

next()
Provides the next time step of a TAP trajectory.

ts = next()

If a TAP trajectory file exists then this is used. Otherwise, the coordinate trajectory is mapped on the fly (which is computationally more expensive).

write(*filename*, *start=None*, *step=None*, *delta=None*, *load=True*)
Write hopping trajectory as standard dcd file.

write('TAP')

Arguments

load = True Immediately loads the trajectory so that further calls to next() will use the computed trajectory and don't use expensive mapping.

Ignore the other options and leave them at the defaults. Currently, only the whole trajectory is written. All atoms in the original trajectory are written to the output so you should be able to use your original psf file.

NOTE: Fixed atoms are possibly not accounted for properly.

Note that it is your responsibility to load the TAP trajectory and the appropriate psf together as there is very limited information stored in the dcd itself.

class hop.trajectory.**ThinDCDReader**(*datafeeder*)
DCD-like object that supports a subsection of the DCDReader interface such as iteration over frames and most attributes. The important part is that the __iter__() method is overriden to provide data from another source. This allows a filter architecture for trajectories.

Generating and analyzing a hopping graph — hop.graph

Interprete the high density sites as graph ('transport graph'), with the sites as vertices and transitions (sampled by the simulation) as edges. The graph is directed.

Each edge (transition) is decorated with the dominant transition rate, the number of events seen, and an instance of fit_func, which represents the fitted function to the survival times.

Each vertex (site) is decorated with the average residency time (and stdev, N).

Typical use of the module:

```
TN = TransportNetwork(hoppingTrajectory,density)
hopgraph = TN.HoppingGraph()
hopgraph.save('hopgraph')
```

The basic object is the *hop.graph.HoppingGraph*; see its documentation for further analysis methods.

Classes and functions

class hop.graph.**CombinedGraph**(*g0=None*, *g1=None*, *filename=None*)
Hybrid graph between hop graphs that share common nodes.

equivalent_sites_stats(*graphnumber*, *elabels*, *equivalence=True*)
Print statistics about one or a list of equivalence sites for the numbered graph.

CombinedGraph.equivalent_sites_stats(graphnumber,elabels)

Arguments

graphnumber index into CombinedGraph.graphs (typically, 0 or 1) elabels single label or list of labels of equivalence sites

(without a '*' if the default identifier is used)

equivalence True: interprete elabels as equivalence labels

False: elabels are labels local to the graph (as used in the output of this method)

export(*igraph*, *filename=None*, *format='XGMML'*, *imageformat=None*, *use_filtered_graph=True*)
Layout the combined graph and highlight the chosen graph.

h.export(igraph=0)

Arguments

graph 0 or 1, selects which graph is to be highlighted filename name for the output files; appropriate suffixes are added

automatically

format XGMML or dot imageformat graphics output format (png, jpg, ps, svg, ... see below) use_filtered_graph

By default, the filtered graph (see the filter() method) is plotted. If set to False then the original HoppingGraph is used instead.

Common nodes are always highlighted in red and shown with the common label. Nodes and edges belonging to the selected graph are shown in black; the other graph is only shown in light gray.

The graph is only written to an image file if an image format is supplied. See https://networkx.lanl.gov/reference/pygraphviz/pygraphviz.agraph.AGraph-class.html#draw for possible output formats but png, jpg, ps are safe bets.

Format

XGMML http://www.cs.rpi.edu/~puninj/XGMML/draft-xgmml.html#Intro and GML http://www.infosun.fim.uni-passau.de/Graphlet/GML/

dot See http://graphviz.org/doc/info/attrs.html for attributes.

Note: On Mac OS X 10.3.9+fink the pygraphviz rendering is buggy and does not include node labels. Simply use the exported .dot file and use Mac OS X graphviz from http://www.pixelglow.com/graphviz/

export3D(***kwargs*)
Export pdb and psf file for visualization in 3D.

```
>>> h.export3D()
Uses h.site_properties if it exists.
```

```
>>> h.export3D(density)
Uses a (hopefully matching) Density object to pull in site_properties.
```

Arguments

density hop.sitemap.Density with full site_properties filename prefix for output files: <filename>.psf and <filename>.pdb use_filtered_graph

define a filtered graph with h.filter() first

The method writes a psf and a pdb file from the graph, suitable for visualization in, for instance, VMD.

Sites are represented as residues of resname 'NOD'; each site is marked by one 'ATOM' (of type CA) at the center of geometry of the site. Edges are bonds between those pseudo atoms.

#Currently: B-factor 1 if common site label exist, 0 otherwis # occupancy: avg site occupancy # (but this should become customizable)

One should use a filtered graph with the bulk site removed for visualization.

Bugs: * with a filtered graph, the degree is the one of the filtered

graph and not of the real underlying graph

- cannot yet select what to display in B-factor and occupancy field: choose from: ['identity','occupancy','degree','volume']

is_connected(*igraph*, *n1*, *n2*)
Return True if nodes n1 and n2 in graph igraph are connected.

load(*filename=None*)
Reinstantiate CombinedGraph from a pickled CombinedGraph (from save()).

plot(*igraph*, *filename=None*, *format='png'*, *use_filtered_graph=True*, *label_sites=None*, *prog='neato'*, *cmap=None*, *max_node_size=500*, *interactive=True*, ***drawargs*)
Plot filtered graph using matplotlib.

Arguments

igraph number of the graph (0 or 1) filename file to write to format any format that matplotlib allows and pdf use_filtered_graph

use a previously defined filtered graph (should be True)

label_sites {'all':False, 'common':True, 'none':False} switches that determine which labels to add to the nodes

prog layout program, can be any of the graphviz programs 'dot','neato','twopi','circo','fdp','nop'

cmap matplotlib color map: nodes are colored by distance of the site from the geometric center of all sites (excluding bulk)

max_node_size maximum node size (in point**2, q.v. matplotlib.scatter()) interactive True: display graph. False: only save to file (eg if no X11) **drawargs additional keyword arguments to networkx.draw() (q.v.)

eg 'linewidths=(0.01,)' for vanishing outlines.

plot_fits(***kwargs*)
Plot survival time fit against data.

plot_fits(ncol=2)

The time values are taken to cover all measured tau.

ncol number of columns nrow number of rows per page plottype 'linear' or 'log' dt time step in ps; use value in self.trjdata['dt'] or 1ps use_filtered_graph

True: use the filtered graph (see filter()), False: use raw data.

directory save all pdf files under this directory format file format for plot (png,eps,pdf... depends on matplotlib) interactive False: do not display graphs on scren (default)

True: show graphs on screen, can be slow and probably requires ipython as your python shell

verbosity chattiness level

All N graphs are laid out in nrow x ncol grids on as many pages/figures as necessary.

The pages are written as eps/pdf files using a fixed filename in the given directory ('survival_times' by default).

site_properties
site_properties of the combined graph, indexed by node label.

stats(*igraph*, *data=None*)
Statistics for the hopping graph.

d = stats(igraph,[data=dict])

Without the data argument, the method just returns some interesting values gathered from the graph igraph and the density. If a data dictionary is given, then the raw data are loaded into the dict and can be processed further by histogramming etc.

Arguments

igraph number of the graph data optional dictionary to hold raw data for

processing; modified by method

Returns

d dictionary with expressive keys, holding the results

tabulate_k(***kwargs*)
List of tuples (from, to, rate (in 1/ns), number of transitions).

class hop.graph.**HoppingGraph**(*graph=None*, *properties=None*, *filename=None*, *trjdata=None*, *site_properties=None*)
A directed graph that describes the average movement of molecules between different well-defined sites by treating the sites as nodes and transitions as edges.

Attributes

graph graph with edges; edges contain rates, fit functions, etc

properties raw data for edges

trjdata metadata of the original trajectory

site_properties density-derived node properties, imported from hop.sitemap.Density

theta dict of nodes with residence times (see compute_site_times())

occupancy_avg average occupancy with standard deviation (see compute_site_occupancy())

occupancy_std (numpy array)

Methods

compute_site_occupancy() Computes occupancies from the residency times theta and updates self.occupancy_avg and self.occupancy_std.

compute_site_times() Computes residency time theta.

save() save graph as a pickled file

load() reinstantiate graph from saved file; typically just use the constructor with the filename argument

filter() make a filtered graph for further analysis and visualization; most plot/export functions require a filtered graph

plot_fits() plot fits of the survival time against the data

tabulate_k() table of rate constants

export() export graph as a dot file that can be used with graphviz

export3D() export graph as a psf/pdb file combination for visualization in VMD

Properties for nodes are always stored as numpy arrays so that one can directly index with the node label (==site label), which is an integer from 0 to the number of nodes. Note that 0 is the interstitial (and only contains bogus data or None), and 1 is the bulk. The bulk site is often excluded from analysis because it is different in nature from the 'real' sites defined as high density regions.

Directed graph with edges containing the rate k_ji,number of observations and S(t) fit.

h = HoppingGraph(graph,properties) h = HoppingGraph(filename='HoppingGraph.pickle')

Arguments

graph networkx graph with nodes (i) and edges (i,j)

properties dictionary of edges: For each edge e, properties contains a dictionary, which contains under the key 'tau' a list of observed waiting times tau_ji. nodes are also listed if they do not participate in a transition

trjdata dictionary describing properties of the trajectory such as time step 'dt' or name of 'dcd' and 'psf'.

Attributes that are in use:

dt time between saved snapshots in ps

hoppsf hopping trajectory psf file name

hopdcd hopping trajectory dcd file name

density pickle file of the density with the sites

totaltime length of trajectory in ps[*]_

Not used:

time_unit 'ps'

site_properties list of site properties: `hop.sitemap.Density.site_properties` (add if you want graphs with mapped labels) (**Really required for most things...!**)

When the graph is built from edges and properties then the rate constants are calculated. For graphs with many hopping events this can take a long time (hours...).

The decorated and directed graph is accessible as `HoppingGraph.graph`

BUGS

- *trjdata* is required for full functionality but it is currently the user's responsibility to fill it appropriately (although `TransportNetwork.compute_residency_times()` already adds some data)
- *site_properties* **are** required and must be added with the constructor

compute_site_occupancy()
Computes occupancies from the residency times theta and updates self.occupancy.

compute_site_occupancy()

occupancy::

$$o[i] = 1/T \sum_{k=1}^{N_i} \theta[i,k]$$

where T is the total trajectory time and the sum runs over all residency times that were recorded for the site i.

attributes:

self.occupancy_avg numpy array with occupancies, site label == index

self.occupancy_std numpy array with error estimates for occupancies (Delta = Delta(theta)/T; this is a biased estimate because Delta(theta) is calculated with N instead of N-1)

compute_site_times(*verbosity=3*)
Compute the 'residency' time of each water molecule on each site.

compute_site_times()

The 'life time' of a site i is computed as

theta[i] = <t[*,i]>

where t[*,i] stands for all waiting times t[j,i] for hops from site i to all other sites AND the waiting times t[i] of molecules that are not observed to leave site i.

- The function updates self.theta[site] for each site with an

array of residency times (in ps).

- The life times are stored in self.lifetime_avg[site] and self.lifetime_std[site]

TODO: * Maybe use the barrier time as well (or a portion thereof,

perhaps proportional to the barrier height (related to the kji) — rate theory??)

connectedness(*n*)
Return values that measure connectedness (can be used in occupancy field)

equivalent_sites_stats(*elabels*, *equivalence=True*)
Statistics about one or a list of equivalence sites.

g.equivalent_sites_stats(elabels,equivalence=True)

Arguments

elabels a single label or a list of node labels equivalence True: interprete elabels as 'equivalence labels', i.e. the label

attached to a site common to two densities False: elabels are labels local to the graph

export (*filename=None*, *format='XGMML'*, *use_filtered_graph=True*, *use_mapped_labels=True*)
Export the graph to a graph format or an image.

export('hopgraph',format='XGMML',use_filtered_graph=True)

Arguments

filename name for the output files; appropriate suffixes are added automatically

format output format: graphs (XGMML or DOT) or image (png, jpg, ps, svg) use_filtered_graph

By default, the filtered graph (see the filter() method) is plotted. If set to False then the original HoppingGraph is used instead.

use_mapped_labels If site_properties is provided then each node that has been identified to exist in a reference network is coloured black and the mapped label is printed instead of the graph label.

export3D (*density=None*, *filename=None*, *use_filtered_graph=True*)
Export pdb and psf file for visualization in 3D.

```
>>> h.export3D()
Uses h.site_properties if it exists.
```

```
>>> h.export3D(density)
Uses a (hopefully matching) Density object to pull in site_properties.
```

Arguments

density hop.sitemap.Density with full site_properties filename prefix for output files: <filename>.psf and <filename>.pdb use_filtered_graph

define a filtered graph with h.filter() first

The method writes a psf and a pdb file from the graph, suitable for visualization in, for instance, VMD.

Sites are represented as residues of resname 'NOD'; each site is marked by one 'ATOM' (of type CA) at the center of geometry of the site. Edges are bonds between those pseudo atoms.

#Currently: B-factor 1 if common site label exist, 0 otherwis # occupancy: avg site occupancy # (but this should become customizable)

One should use a filtered graph with the bulk site removed for visualization.

Bugs: * with a filtered graph, the degree is the one of the filtered

graph and not of the real underlying graph

- cannot yet select what to display in B-factor and occupancy field: choose from: ['identity','occupancy','degree','volume']

filename (*filename=None*, *ext=None*, *set_default=False*, *use_my_ext=False*)
Supply a file name for the object.

fn = filename() —> <default_filename> fn = filename('name.ext') —> 'name' fn = filename(ext='pickle') —> <default_filename>'.pickle' fn = filename('name.inp','pdf') –> 'name.pdf' fn = filename('foo.pdf',ext='png',use_my_ext=True) –> 'foo.pdf'

The returned filename is stripped of the extension (use_my_ext=False) and if provided, another extension is appended. Chooses a default if no filename is given. Raises a ValueError exception if no default file name is known.

If set_default=True then the default filename is also set.

use_my_ext=True lets the suffix of a provided filename take priority over a default ext(tension).

filter(*exclude=None*)
Create a filtered version of the graph.

For looking at most things: >>> h.filter(exclude={'outliers':True})

For looking at exchange rates and plotting: >>> h.filter(exclude={'outliers':True, 'Nmin':5, 'unconnected':True})

For export3D do not use the bulk site: >>> h.filter(exclude={'outliers':True,'bulk':True})

This method makes a copy of the hopping graph and applies the filter rules to the copy. Other output functions use this copy if it exists.

exclude dict of components to exclude. May contain

{'outliers':True, 'Nmin':integer, 'bulk': True, 'unconnected':True}

If outliers == True then all edges from the 'outlier' node are deleted previous to displaying the graph. Those edges correspond to particles starting in a region not covered by the intial histogram boundaries and enter a mapped site at a later point in time.

With Nmin, any node that has fewer than Nmin transition is discarded.

unconnected == True finaly filters all nodes that have no edges left

from_site(*edge*)
Returns the originating site of hop.

internal_sites()
Returns list of sites that have no connection to the bulk.

is_connected(*n1*, *n2*)
True if node n1 has any connection to the site n2.

is_from_bulk(*edge*)
True if the edge originated in the bulk.

is_internal(*n*)
True if site n has no connection to the bulk.

is_isolated(*n*)
True if site n has no connections to other sites (ie its degree equals 0).

isolated_sites()
Returns list of sites that have no other connections.

load(*filename=None*)
Reinstantiate HoppingGraph from a pickled HoppingGraph (from save()).

number_of_hops(*edge*)
Number of transitions recorded.

plot_fits(*ncol=2*, *nrow=3*, *dt=None*, *plottype='log'*, *use_filtered_graph=True*, *directory='survival_times'*, *format='png'*, *interactive=False*, *verbosity=3*)
Plot survival time fit against data.

plot_fits(ncol=2)

The time values are taken to cover all measured tau.

ncol number of columns nrow number of rows per page plottype 'linear' or 'log' dt time step in ps; use value in self.trjdata['dt'] or 1ps use_filtered_graph

True: use the filtered graph (see filter()), False: use raw data.

directory save all pdf files under this directory format file format for plot (png,eps,pdf... depends on matplotlib) interactive False: do not display graphs on scren (default)

True: show graphs on screen, can be slow and probably requires ipython as your python shell

verbosity chattiness level

All N graphs are laid out in nrow x ncol grids on as many pages/figures as necessary.

The pages are written as eps/pdf files using a fixed filename in the given directory ('survival_times' by default).

rate (*edge*)
Returns the fastest rate on an edge, in ns^-1

rates (*n*, *use_filtered_graph=True*)
Returns k_tot, k_in, k_out (and N_*) for site n (bulk rates omitted from k).

dictionary = rates(n,use_filtered=True)

k_in = sum_j k_nj (> 0) j<>bulk k_out = sum_j k_jn (< 0) j<>bulk k_tot = k_in + k_out

Note that k_tot should be ~0 if a bulk rate is included because the graph should obey detailed balance.

save (*filename=None*)
Save HoppingGraph as a pickled python object.

select_graph (*use_filtered_graph*)
Returns filtered graph for True argument, or the raw graph otherwise)

show_rates (*filename=None*)
Print the rates (in 1/ns) between sites, and the total number of observations.

show_rates(file=filename)

By default, prints to stdout but if *file* = filename then filename is opened and data are written to the file.

A description of the fit function used to obtain the rate is also printed in the last column.

Only the "dominant" rate is shown; see the fit_func description for cases when two rates were computed.

See also:

HoppingGraph.tabulate_k().

show_site (*sites*, *use_filtered_graph=True*)
Display data about sites (list of site labels or single site).

show_total_rates (*use_filtered_graph=True*)
Display total rates for all nodes (excluding bulk –> site contributions).

site_properties
Site_properties, indexed by node label. Setting this attribut also updates self.equivalent_sites_index.

stats (*data=None*)
Statistics for the hopping graph.

stats([data=dict]) –> dict

Without the data argument, the method just returns some interesting values gathered from the graph and the density. If a data dictionary is given, then the raw data are loaded into the dict and can be processed further by histogramming etc.

Arguments

data optional dictionary to hold raw data for processing; modified by method

Returns dictionary with expressive keys, holding the results

tabulate_k()
List of tuples (from, to, rate (in 1/ns), number of transitions).

to_site(*edge*)
Returns the site to which a hop is directed.

waitingtime_fit(*edge*)
Returns the fit function for the edge's waiting time distribution.

write_psf(*graph*, *props*, *filename=None*)
Pseudo psf with nodes as atoms and edges as bonds

class hop.graph.**TransportNetwork**(*traj*, *density=None*, *sitelabels=None*)
A framework for computing graphs from hopping trajectories.

The unit of time is ps.

The *TransportNetwork* is an intermediate data structure that is mainly used in order to build a *HoppingGraph* with the *TransportNetwork.HoppingGraph()* method.

Setup a transport graph from a hopping trajectory instance.

:: hops = hop.trajectory.HoppingTrajectory(hopdcd='whop.dcd',hoppsf='whop.psf') tn = TransportNetwork(hops)

HoppingGraph(*verbosity=3*)
Compute the HoppingGraph from the data and return it.

compute_site_occupancy()
Computes occupancies from the residency times theta and updates self.occupancy.

occupancy::

N_i

o[i] = 1/T Sum theta[i,k] k=1

where T is the total trajectory time and the sum runs over all residency times that were recorded for the site i.

Attributes

self.occupancy numpy array with occupancies, site label == index

self.occupancy_error numpy array with error estimates for occupancies (Delta = Delta(theta)/T; this is a biased estimate because Delta(theta) is calculated with N instead of N-1)

compute_site_times(*verbosity=3*)
Compute the 'residency' time of each water molecule on each site.

The 'site time' of a site i is computed as:

```
theta[i] = 1/T_sim ( Sum_j tau[j,i] + Sum tau[i] )
```

tau[j,i] is the waiting time for hops from i to j. tau[i] is the waiting time for molecules that are not observed to leave site i.

The function updates self.theta[site] for each site with an array of residency times (in ps).

It uses the residency times and thus requires compute_residency_times() was run previously.

TODO Maybe use the barrier time as well (or a portion thereof, perhaps proportional to the barrier height (related to the kji) — rate theory??)

export (*filename=None*, *format='png'*, *exclude_outliers=False*)
Export the graph as dot file and as an image.

export(filename)

See https://networkx.lanl.gov/reference/pygraphviz/pygraphviz.agraph.AGraph-class.html#draw for possible output formats.

See http://graphviz.org/doc/info/attrs.html for attributes.

Note: On Mac OS X 10.3.9+fink the pygraphviz rendering is buggy and does not include node labels. Simply use the exported .dot file and use Mac OS X graphviz from http://www.pixelglow.com/graphviz/

graph_alltransitions ()
Constructs the graph that contains all transitions in the trajectory.

Populates TransportGraph.graph with a graph that contains all sites and one edge for each transition that was observed in the trajectory. Useful for an initial appraisal of the complexity of the problem.

Warning: Erases any previous contents of graph.

plot_residency_times (*filename*, *bins=None*, *exclude_outliers=True*)
Plot histograms of all sites

plot_residency_times('sitetime.eps')

pylab always writes the figure to the named file. If pylab is already running, display the graph with pylab.show().

The histograms are normalized and the time values are the left edges of the bins.

If bins=None then the number of bins is determined heuristically.

plot_site_occupancy (*filename, bins=10, exclude_sites=[0, 1]*)
Plot site occupancy (from compute_site_occupancy().

plot_site_occupancy(filename,exclude_sites=[0,1])

filename name of output file bins bins for histogram (see numpy.histogram) exclude_site list of site labels which are NOT plotted. Typically,

exclude interstitial and bulk.

hop.graph.**Unitstep** (*x*, *x0*)

Heaviside step function / 1 if x >= x0

Unitstep(x,x0) == Theta(x - x0) = { 0.5 if x == x0 0 if x < x0

This is a numpy ufunc.

CAVEAT If both x and x0 are arrays of length > 1 then weird things are going to happen because of broadcasting. Using nD arrays can also lead to surprising results.

See also http://mathworld.wolfram.com/HeavisideStepFunction.html

class hop.graph.**fitExp** (*x*, *y*)
y = f(x) = exp(-p[0]*x)

f_factory ()
Stub for fit function factory, which returns the fit function. Override for derived classes.

initial_values()
List of initital guesses for all parameters p[]

class hop.graph.**fitExp2**(*x*, *y*)
y = f(x) = p[0]*exp(-p[1]*x) + (1-p[0])*exp(-p[2]*x)

f_factory()
Stub for fit function factory, which returns the fit function. Override for derived classes.

initial_values()
List of initital guesses for all parameters p[]

class hop.graph.**fit_func**(*x*, *y*)
Fit a function f to data (x,y) using the method of least squares.

Attributes:

parameters list of parameters of the fit

f_factory()
Stub for fit function factory, which returns the fit function. Override for derived classes.

fit(*x*)
Applies the fit to all x values

initial_values()
List of initital guesses for all parameters p[]

class hop.graph.**fitlin**(*x*, *y*)
y = f(x) = p[0]*x + p[1]

f_factory()
Stub for fit function factory, which returns the fit function. Override for derived classes.

initial_values()
List of initital guesses for all parameters p[]

hop.graph.**survivalfunction**(*waitingtimes*, *block_w=200*, *block_t=1000*)
Returns the survival function S(t), defined by a list of waiting times.

survival([t0, t1, . . . , tN]) –> S(t)

S(t) is a function that gives the fractional number of particles that have not yet left the site after time t. It is 1 at t=0 and decays to 0.

Arguments

waitingtimes sequence of the waiting times from the simulations

block_w reduce memory consumption by working on chunks of the waiting times of size <block_w>; reduce block_w if the code crashes with **:Exception:'MemoryError'**.

block_t chunk input function arguments into blocks of size block_t

TODO Make S(t) an interpolation function: massive speedup and fewer memory problems

5.4.2 Analyzing hopping graphs and densities

Extracting information from densities and hop graphs — hop.analysis

A collection of functions and classes to extract statistics and plot histograms. Use this as examples how to write your own.

Classes and functions

class hop.analysis.**DensityScanner**(*densityAnalysis*, *with_densities=True*)

load(*fn*, *merge=True*)
Reinstantiate class from a pickled file (produced with save()).

plot(*fn=None*, *idens=0*, *functions='all'*, *properties=None*, *fignumber=1*)
Plot density statistics against rho_cut for reference (black) and density 0 (red).

plot(filename,properties=<dict of dicts>)

Plot various functions of the density cut-off rho_cut. Current functions are 'sites', 'volume', 'occupancy', or 'all'.

Plots can be customized by using the properties dict. To change the ylim and add an title to the sites graph, use

properties = {'sites': {ylim':(0,220),'title':'number of sites'}}

Arguments

fn file name for the file; default is scan.pdf. Suffix determines file type. idens number of density plot; the first one is 0 in self.scanarrays[]. functions list of function names or 'all' properties dict1 of dicts; keys1: sites, volume, occupancy;

keys2: any matplotlib settable property, values2: appropriate values

fignumber pylab figure number

class hop.analysis.**HeatmapAnalysis**(*hoppinggraphs*, *normalization='maxabs'*, *verbosity=1*, *prune='default'*)

Combine Hopgraph statistics for a number of simulations into a grid, normalize each observable, and color. Clustering is performed if the R package is installed in the system. The idea is to quickly compare a number simulations based on a combination of observables.

Create a 'heatmap' for the Hopgraph statistics from a dictionary of CombinedGraphs.

```
>>> hm = HeatmapAnalysis(hg,normalize="maxabs")
```

Arguments

HoppingGraphs Dictionary of HoppingGraph instances. The key is used to label the simulation in the heat map and thus should be expressive.

normalization Method to normalize the data across observables. Can be None (not recommended), 'maxabs', or 'zscore'. See the normalize() method for documentation. NOTE that the normalization strongly influences the clustering in the heat map.

verbosity Chattiness; use at least 1 in order to be notified if you should install additional packages. Otherwise a less powerful alternative is chosen silently,

prune dict with keys that are removed from the heat map; see prune_default class attribute.

Methods

plot plot the heat map normalize normalize using the 'normalize' method labels dictionary of row, column names (and the normalization constants

as strings)

annotation 'enumerate' dictionaries of labels but not stringified

filename (*filename=None*, *ext=None*, *set_default=False*, *use_my_ext=False*)
Supply a file name for the object.

fn = filename() —> <default_filename> fn = filename('name.ext') —> 'name' fn = filename(ext='pickle') —> <default_filename>'.pickle' fn = filename('name.inp','pdf') –> 'name.pdf' fn = filename('foo.pdf',ext='png',use_my_ext=True) –> 'foo.pdf'

The returned filename is stripped of the extension (use_my_ext=False) and if provided, another extension is appended. Chooses a default if no filename is given. Raises a ValueError exception if no default file name is known.

If set_default=True then the default filename is also set.

use_my_ext=True lets the suffix of a provided filename take priority over a default ext(tension).

labels (*precision=2*)
labels of the columns (simulations) and rows (observables)

normalize (*method=None*)
Normalize the data by row.

normalize(method=None|'zscore'|'maxabs')

method can be None Return the unchanged data array. 'maxabs' Take the largest absolute value in each row/column and

divide each entry in the row/column by it. This results in values between -1 and +1.

'zscore' (X-<X>)/sd(X)

Sets self.heatmap, self.normalizations, self.normalization_method

normalizations only makes sense for 'maxabs'; in all other cases it only contains zeroes.

plot (*filename=None*, *format='pdf'*, ***kwargs*)
Plot the heatmap and save to an image file.

plot() # display using windowing system plot('hm') # –> hm.pdf plot('hm.png') # –> hm.png plot('hm','png') # –> hm.png

By default a clustered heat map is constructed using R's heatmap.2 function. If R cannot be found, an unclustered heat map is plotted. **kwargs can be used to customize the output.

Arguments

filename name of the image file; may contain extension If empty use the windowing system.

format eps,pdf,png... whatever matplotlib understands

**kwargs for R: scale Determines the coloring. Choose between 'none' (the

actual values in the heat map (possibly already normalized)), 'row' or 'column' (z-score across the dimension)

N_colors Number of color levels; default is 32.

****kwargs for matplotlib:** The kwargs are applied to the matplotlib.text() method and are typically used to set font properties. See the pylab/matplotlib documentation.

class hop.analysis.**HopgraphAnalysis** (*hopgraph*, *dir='.'*, *verbosity=3*)
Comprehensive analysis of an annotated hop graph.

Analyse hopgraph.

a = HopgraphAnalysis(hopgraph)

The show() method prints statistics on the HoppingGraph and histograms() produces a number of plots as pdf files in the current directory.

Arguments

hopgraph can be the name of a pickled file or a HoppingGraph instance

dir save figures in this directory verbosity=3 chattiness

Attributes

S statistics dictionary (see keys for explanation) D raw data dictionary

Methods

all() show() and histograms() show() print stats histograms() produce histograms

class hop.analysis.**LegendContainer**

For each bar plot, record first lines instance and the label with >>> Legend = LegendContainer() >>> lines = pylab.bar(...) >>> Legend.append(lines[0],'plotlabel') Once all legends have been collected, build the legend with >>> pylab.legend(*Legend.args())

args()

Use as pylab.legend(**Legend.args()).

hop.analysis.**kill_R**()

Manual last resort to kill the R quartz() window.

5.4.3 Markov Chain Monte Carlo sampling on a hop graph

The hop graph encodes the dynamic information of the system. Using a Markov Chain Monte Carlo method one can propagate the dynamics to much longer time scales than accessible by the underlying MD simulations alone and calculate fluxes across the network.

Markov Chain Monte Carlo on hopping graph — hop.MCMC

The *hop.MCMC* module uses the information encoded in a hopping graph to set up a Markov Chain Monte Carlo sampling procedure that allows one to rapidly generate site occupancy distributions that are distributed in the same way as the one sampled from MD.

The most convenient entry point is the *hop.MCMC.run()* function

```
M = MCMC.run(filename='hopgraph.pickle',Ntotal=<int>)
```

It takes as input a stored hopgraph and immediately runs an MCMC run of *Ntotal* steps. The output is a *MCMCsampler* object. It contains the 'trajectory' and useful analysis functions. (Use interactive introspection in ipython to explore the possibilities of the object.)

Notes on the algorithm:

- some sort of dynamic lattice Monte Carlo with very simple acceptance probabilities (0 or 1, if there's no space on the site, and 1 if there is)

  ... is 'MCMC' actually the proper description?

- Extension to multiply occupied sites: use the site occupancy distributions from siteanalysis, and replace the unconditional move by an acceptance probability == s_i(n)

- I am currently using time-forward (out-of) and time-backward (into) site moves (the latter inspired by coupling from the past).

Classes and functions

class hop.MCMC.**MCMCsampler**(*h=None*, *min_hops_observed=1*, *filename=None*)
Generate an equilibrium distribution of states from a hop graph.

Initialize the Markov Chain Monte Carlo sample with a HoppingGraph.

M = MCMCsampler(HoppingGraph)

Build a Markov Chain model from a <HoppingGraph> (with edges deleted that have less than <min_hops_observed> hops).

autocorrelation(*start=None*, *stop=None*, *step=None*, ***kwargs*)
Calculates the auto correlation function for all site trajectories.

averaged_autocorrelation(*step=None*, ***kwargs*)
Calculates the ACF or each site by resampling from the whole trajectory.

mean(acf), standardev(acf) = averaged_autocorrelation(**kwargs)

Arguments

step only take every <step> from the trajectory (None == 1) ??? step > 1 seems to take LONGER ???

length length (in frames) of the ACF (default: 1/2*len(series)) sliding_window repeat ACF calculation every N frames (default: len(series)/100)

Returns

mean_acf average over all resampled acfs per site, shape = (Nsites,length) std_acf standard deviation or the resampled acfs, shape = (Nsites,length)

See also for kwargs:

firstsiteindex
State array index of the first site after bulk.

index2node
Translates sequential array index to node label (in graph).

init_state(*Nbulk=10000.0*)
Initialize state with 1 particle per standard site and Nbulk for the bulk site.

mean()
Mean for each site (excl bulk).

mean_std()
Returns site labels, mean, and standard deviation for each site (excl bulk).

node2index
Translates node label (in graph) to the sequential array index.

occupancy()
Ensemble averaged occupancy (over ALL states incl bulk) and fluctuation.

occupancy_mean_correl()
Calculates the correlation coefficient between simulation and MCMC occupancies.

occupancy_std_correl()
Calculates the correlation coefficient between simulation and MCMC occupancy fluctuations.

plot(*filename=None*, *plot_skip=None*)
Plot density plot of the saved configurations in states[].

plot_correl(*legend=True*, ***kwargs*)
Plot the occupancy from the MD simulation vs the MCMC one.

plot_occupancy(*legend=True*, ***kwargs*)
Plot site label vs <N> +/- std(N).

legend True: add legend, False: return (line,description) **kwargs additional arguments for errbar plot such as color='k', fmt='o'

run(*Ntotal=500000*, *Nskip=1000*, *verbosity=None*)
MCMC run multiple cycles of lebgth <Nskip> scans for a total of <Ntotal>.

run(Ntotal=500000,Nskip=1000)

Starts from the current configuration in state. Creates the collection of configurations states: one state every Nskip steps

sample(*max_iter=10000*, *record_iterations=True*)
Run <max_iter> Monte Carlo site moves.

sample(max_iter=10000)

Runs a batch of MCMC moves.

sites
Translates sequential array index to node label (in graph).

statevector
State as a numpy array; the corresponding nodes are state.keys()

std()
Standard deviation for each site.

class hop.MCMC.**MultiPscan**(*repeat=10*, ***pscanargs*)
Run Pscan(**pscanargs) <repeat> times and collect all Pscan objects in list.

pscans = MultiPscan(repeat=10, parameter='Ntotal', pvalues=[1e4,2.5e4,....], ...) See Pscan() for description of pscanargs.

class hop.MCMC.**Pscan**(*parameter*, *pvalues=None*, *filename='hopgraph.pickle'*, *Ntotal=1000000.0*, ***kwargs*)
Run a MCMC sampler for a number of parameter values.

Sample on hopping graph for different values of <parameter> = p.

P = Pscan(parameter=<string>,pvalues=<sequence>,filename=<filename>,**kwargs)

<parameter> must be a keyword argument to hop.MCMC.run(); the parameter overrides any default values that may have been set. For instance, <parameter> can be 'Ntotal' or 'filename'.

kwargs: all other kwargs are directly passed on to MCMC.run().

occupancy_mean_correl()
Returns X=pvalues, Y=occupancy_mean_correlations.

plot_occupancy(***kwargs*)
Plot <n_i> (site occupancy from MCMC) for all parameter values.

(See _plotter())

plot_occupancy_mean_correl (***kwargs*)
Plot MD occupancy vs MCMC occupancy.

plot_correl(colorscale='linear'|'log')

(See _plotter())

plot_states (*maxcolumns=2*)
Plot all state 'trajectories' as a tiled plot.

save (*filename='pscan.pickle'*)
Save pscan object to pickle file.

save(pscan.pickle)

Load with

import cPickle myPscan = cPickle.load(open('pscan.pickle'))

hop.MCMC.**multi_plot** (*plist*, *plottype='whisker'*, *Nequil=10000*, *funcname='occupancy_mean_correl'*, ***kwargs*)
Display a collection of functions.

multi_plot(plist,plottype='whisker',Nequil=10000,funcname='occupancy_mean_correl',**kwargs)

The function is obtained from a method call on the objects in plist. The assumption is that these are functions of Ntotal (if not, set Nequil=0; Nequil is added to x). Each object is a different realization, e.g. multiple MCMC runs.

plottype 'whisker' (whisker plot), 'standard' (average and standard deviations) Nequil correction, added to x funcname string; a method of the objects in plist that does EXACTLY the following:

x,y = obj.funcname() where x and y are numpy arrays of equal length

**kwargs color, boxcolor, mediancolor, capsize

hop.MCMC.**run** (*filename='hopgraph.pickle'*, *Ntotal=500000*, *Nskip=1000*, *Nequil=10000*)
Perform Markov Chain Monte Carlo on a model derived from the hopping graph.

5.4.4 Auxiliary modules

Constants — hop.constants

Constants that are being used throughout the hop module.

Conversions:

The conversion factor f to a unit b' for a quantity X (whose numeric value relative to the base unit b is stored in the program) is a quantity with unit b'/b. In the dictionaries below only the numeric value f(b->b') is stored:

```
X/b' = f(b->b') * X/b
```

See also:

MDAnalysis.units

Constants and functions

Utility functions — hop.utilities

Random mix of convenience functions that don't fit anywhere else.

For messages I should probably use python's logger module but this is working so far (even though it's pretty crappy).

class hop.utilities.**CustomProgressMeter**(*numsteps*, *format=None*, *interval=10*, *offset=1*, *verbose=None*, *dynamic=True*, *format_handling='auto'*, *quiet=None*)

ProgressMeter that uses addition '%(other)s' in format string.

See also:

MDAnalysis.lib.log.ProgressMeter

echo(*step*, *other*)

Output status for *step* with additional information *other*.

class hop.utilities.**DefaultDict**(*defaultdict*, *userdict=None*, ***kwargs*)

Dictionary based on defaults and updated with keys/values from user.

class hop.utilities.**Fifo**

pop()

Remove and return the leftmost element.

class hop.utilities.**IntrospectiveDict**(**args*, ***kwargs*)

A dictionary that contains its keys as attributes for easier introspection.

Keys that collide with dict methods or attributes are _not_ added as attributes.

The implementation is simple and certainly not optimized for larger dictionaries or ones which are often accessed. Only use it for 'final results' collections that you are likely to investigate interactively.

ARGH: This cannot be pickled safely.

hop.utilities.**Pearson_r**(*x*, *y*)

Pearson's r (correlation coefficient)

r = Pearson(x,y)

x and y are arrays of same length

Historical note: Naive implementation of Pearson's r:

Ex = scipy.stats.mean(x) Ey = scipy.stats.mean(y)

covxy = numpy.sum((x-Ex)*(y-Ey)) r = covxy/math.sqrt(numpy.sum((x-Ex)**2)*numpy.sum((y-Ey)**2)) return r

class hop.utilities.**Ringbuffer**(*capacity*, *iterable=None*)

Ring buffer of size capacity; 'pushes' data from left and discards on the right.

append(*x*)

Add an element to the right side of the deque.

class hop.utilities.**Saveable**(**args*, ***kwargs*)

Baseclass that supports save()ing and load()ing.

Override the class variables

_saved_attributes = [] # list attributes to be pickled _merge_attributes = [] # list dicts to be UPDATED from the pickled file with load(merge=True) _excluded_attributes = [] # list attributes that should never be pickled

Note:

_saved_attributes = 'all' # pickles ALL attributes, equivalent to self.__dict__.keys() # (use _excluded_attributes with 'all'!)

Use _excluded_attributes to filter out some attributes such as type('method-wrapper') objects that cannot be pickled (e.g. when using properties).

filename(*filename=None*, *ext=None*, *set_default=False*, *use_my_ext=False*)
Supply a file name for the object.

fn = filename() —> <default_filename> fn = filename('name.ext') —> 'name' fn = filename(ext='pickle') —> <default_filename>'.pickle' fn = filename('name.inp','pdf') –> 'name.pdf' fn = filename('foo.pdf',ext='png',use_my_ext=True) –> 'foo.pdf'

The returned filename is stripped of the extension (use_my_ext=False) and if provided, another extension is appended. Chooses a default if no filename is given. Raises a ValueError exception if no default file name is known.

If set_default=True then the default filename is also set.

use_my_ext=True lets the suffix of a provided filename take priority over a default ext(tension).

load(*filename=None*, *merge=False*)
Reinstantiate class from a pickled file (produced with save()).

save(*filename=None*)
Save class to a pickled file.

hop.utilities.**asiterable**(*obj*)
Return an object that is an iterable: object itself or wrapepd in a list.

iterable <– asiterable(something)

Treats strings as NOT-iterable.

hop.utilities.**autocorrelation_fft**(*series*, *include_mean=False*, *periodic=False*, *start=None*, *stop=None*, ***kwargs*)
Calculate the auto correlation function.

acf = autocorrelation_fft(series,include_mean=False,**kwargs)

The time series is correlated with itself across its whole length. It is 0-padded and the ACF is corrected for the 0-padding (the values for larger lags are increased) unless mode='valid' (see below). Only the [0,len(series)[interval is returned. The series is normalized to ots 0-th element.

Note that the series for mode='same'|'full' is inaccurate for long times and should probably be truncated at 1/2*len(series). Alternatively, only sample a subseries with the stop keyword.

Arguments

series (time) series, a 1D numpy array include_mean False: subtract mean(series) from series periodic False: corrected for 0-padding

True: return as is

start,stop If set, calculate the ACF of series[start:stop] with series; in this case mode='valid' is enforced

kwargs keyword arguments for scipy.signal.fftconvolve mode = 'full' | 'same' | 'valid' (see there)

hop.utilities.**averaged_autocorrelation**(*series*, *length=None*, *sliding_window=None*, ***kwargs*)

Calculates the averaged ACF of a series.

mean(acf), std(acf) = averaged_autocorrelation(series,length=None,sliding_window=None):

Calculate the ACF of a series for only a fraction of the total length, <length> but repeat the calculation by setting the origin progressively every <sliding_window> steps and average over all the ACFs.

Arguments

series time series (by default, mean will be removed) length length (in frames) of the ACF (default: 1/2*len(series)) sliding_window repeat ACF calculation every N frames (default: len(series)/100) kwargs additional arguments to autocorrelation_fft()

hop.utilities.**close_log**()

Close open logfile; must be done manually.

hop.utilities.**easy_load**(*names*, *baseclass*, *keymethod*)

Instantiate a class either from an existing instance or a pickled file.

instance_list = easy_load(names,baseclass,keymethod)

```
>>> x = easy_load(<filename>,Xclass,'my_method_name')
>>> [x1,x2,...] = easy_load([<filename1>, <fn2>,...], Xclass,'my_method_name')
>>> [x1,x2,...] = easy_load([x1, x2, ..], Xclass,'my_method_name')
```

If the argument does not implement the keymethod then try loading from a file.

API:

For this to work, the baseclass (eg Saveable) must be able to instantiate itself using

x = baseclass(filename=name)

If a single name is given, a singlet is returned, otherwise a list of instances.

(The docs are longer than the code...)

hop.utilities.**fileextension**(*filename*, *default=None*)

Return the file extension without the leading dot or the default.

hop.utilities.**filename_function**(*self*, *filename=None*, *ext=None*, *set_default=False*, *use_my_ext=False*)

Supply a file name for the object.

fn = filename() —> <default_filename> fn = filename('name.ext') —> 'name' fn = filename(ext='pickle') —> <default_filename>'.pickle' fn = filename('name.inp','pdf') –> 'name.pdf' fn = filename('foo.pdf',ext='png',use_my_ext=True) –> 'foo.pdf'

The returned filename is stripped of the extension (use_my_ext=False) and if provided, another extension is appended. Chooses a default if no filename is given. Raises a ValueError exception if no default file name is known.

If set_default=True then the default filename is also set.

use_my_ext=True lets the suffix of a provided filename take priority over a default ext(tension).

hop.utilities.**fixedwidth_bins**(*delta*, *xmin*, *xmax*)

Return bins of width delta that cover xmin,xmax (or a larger range).

dict = fixedwidth_bins(delta,xmin,xmax)

The dict contains 'Nbins', 'delta', 'min', and 'max'.

hop.utilities.**flatiter**(*seq*)
Returns an iterator that flattens a sequence of sequences of sequences... (c) 2005 Peter Otten, at http://www.thescripts.com/forum/thread23631.html

hop.utilities.**flatten**(*sequence*) → list
Returns a single, flat list which contains all elements retrieved from the sequence and all recursively contained sub-sequences (iterables).

Examples: >>> [1, 2, [3,4], (5,6)] [1, 2, [3, 4], (5, 6)] >>> flatten([[[1,2,3], (42,None)], [4,5], [6], 7, MyVector(8,9,10)]) [1, 2, 3, 42, None, 4, 5, 6, 7, 8, 9, 10]

From http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks

hop.utilities.**iterable**(*obj*)
Returns True if obj can be iterated over and is NOT a string.

hop.utilities.**linfit**(*x*, *y*, *dy=[]*)
Fit a straight line y = a + bx to the data in x and y; errors on y should be provided in dy in order to assess the goodness of the fit and derive errors on the parameters.

result_dict = linfit(x,y[,dy])

Fit y = a + bx to the data in x and y by analytically minimizing chi^2. dy holds the standard deviations of the individual y_i. If dy is not given, they are assumed to be constant (note that in this case Q is set to 1 and it is meaningless and chi2 is normalised to unit standard deviation on all points!).

Returns the parameters a and b, their uncertainties sigma_a and sigma_b, and their correlation coefficient r_ab; it also returns the chi-squared statistic and the goodness-of-fit probability Q (that the fit would have chi^2 this large or larger; Q < 10^-2 indicates that the model is bad — Q is the probability that a value of chi-square as _poor_ as the calculated statistic chi2 should occur by chance.)

result_dict = intercept, sigma_intercept a +/- sigma_a slope, sigma_slope b +/- sigma_b parameter_correlation correlation coefficient r_ab

between a and b

chi_square chi^2 test statistic Q_fit goodness-of-fit probability

Based on 'Numerical Recipes in C', Ch 15.2.

hop.utilities.**mkdir_p**(*path*)
Create a directory *path* with subdirs but do not complain if it exists.

This is like GNU `mkdir -p path`.

hop.utilities.**msg**(*level*[, *m*])
1) Print message string if the level <= verbose. level describes the priority with lower = more important.

Terminate string with n if a newline is desired or r to overwrite the current line (eg for output progress indication)

Note that if the global verbosity level is < 0 then the message is also written to the logfile.

2) If called without a string then msg(level) returns True if it would print a message, and False otherwise.

hop.utilities.**set_verbosity**([*level*], *logfilename=<filename>*)
Set the verbosity level level < 0 : level <- abs(level) but output is also appended to logfile level == 0: minimum level == 3: verbose level > 3 : debugging

hop.utilities.**unlink_f**(*path*)
Unlink path but do not complain if file does not exist.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[Hop2009] Oliver Beckstein, Naveen Michaud-Agrawal and Thomas B. Woolf. Quantitative Analysis of Water Dynamics in and near Proteins. Biophysical Journal 96 (2009), 601a. doi:10.1016/j.bpj.2008.12.3147

[MDAnalysis2011] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein. MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations. J. Comput. Chem. 32 (2011), 2319–2327, doi:10.1002/jcc.21787

Python Module Index

h

Index

G

H

I

K

L

M

N

O