idpflex Documentation

Release 0.1.8

Jose Borreguero

Jan 17, 2019

Contents

Analysis of intrinsically disordered proteins by comparing MD simulations to Small Angle Scattering experiments.

CHAPTER 1

Contents:

1.1 Installation

1.1.1 Requirements

- Operative system: Linux or iOS

1.1.2 Stable release

To install idpflex, run this command in your terminal:

```
$ pip install idpflex
```

This is the preferred method to install idpflex, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.1.3 From sources

The sources for idpflex can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/jmborr/idpflex
```

Or download the tarball:

```
$ curl  -OL https://github.com/jmborr/idpflex/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

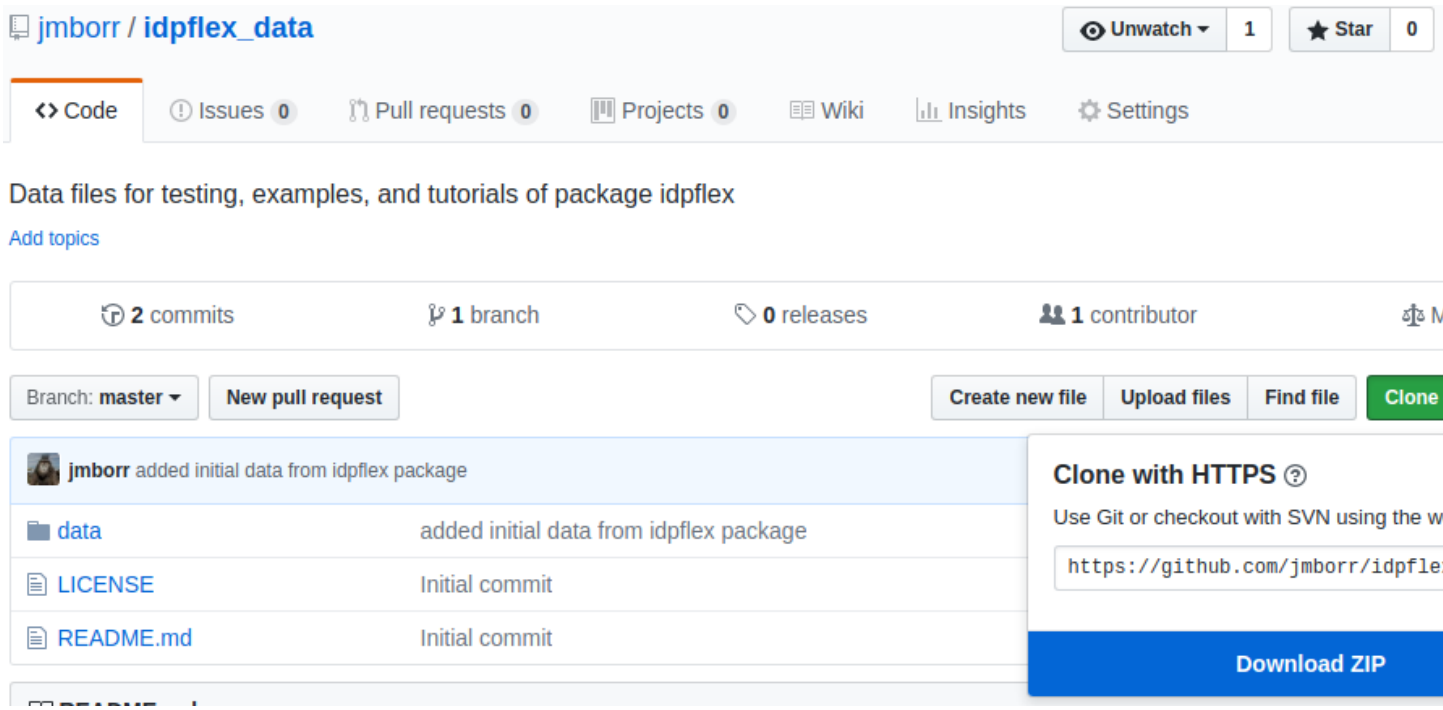
1.1.4 Testing & Tutorials Data

The external repository *idpflex_data <https://github.com/jmborr/idpflex_data>* contains all data files used in testing, examples, and tutorials. There are several ways to obtain this dataset:

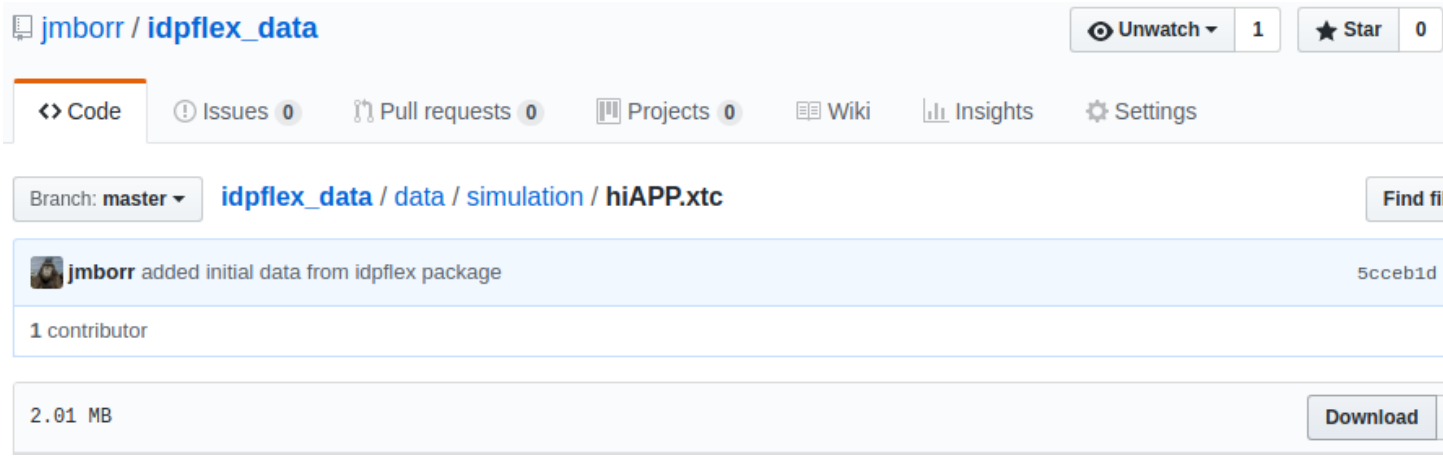
1. Clone the repository with a git command in a terminal:

```
cd some/directory/
git clone https://github.com/jmborr/idpflex_data.git
```

2. Download all data files as a zip file using GitHub's web interface:



3. Download individual files using GitHub's web interface by browsing to the file. If the file is in binary format, then click in Download button:



If the file is in ASCII format, you must first click in the *Raw* view. After this you can right-click and *Save as*.

1.2 Usage

To use idpflex in a project:

```
import idpflex
```

1.2.1 Tutorials and Examples

Clustering of disordered hiAPP peptide 

1.3 Modules

1.3.1 bayes : Fit simulated profiles against experiment

class idpflex.bayes.**TabulatedFunctionModel**(*xdata*, *ydata*, *interpolator_kind='linear'*, *prefix=''*, *missing=None*, *name=None*, ***kwargs*)

Bases: `lmfit.model.Model`

A fit model that uses a table of (x, y) values to interpolate

Uses `interp1d`

Fitting parameters:

- integrated intensity `amplitude` A
- position of the peak `center` E_0
- nominal relaxation time `tau` τ
- stretching exponent `beta` β

Parameters

- **xdata** (ndarray) – X-values to construct the interpolator
- **ydata** (ndarray) – Y-values to construct the interpolator
- **interpolator_kind** (*str*) – Interpolator that interp1d should use

guess (*y*, *x=None*, ***kwargs*)
Estimate fitting parameters from input data

Parameters

- **y** (ndarray) – Values to fit to, e.g., SANS or SAXS intensity values
- **x** (ndarray) – independent variable, e.g., momentum transfer

Returns Parameters with estimated initial values.

Return type Parameters

idpflex.bayes.**fit_at_depth** (*tree*, *experiment*, *property_name*, *depth*)
Fit at a particular tree depth from the root node

Fit experiment against the property stored in the nodes. The fit model is generated by *model_at_depth()*

Parameters

- **tree** (*Tree*) – Hierarchical tree
- **experiment** (*ProfileProperty*) – A property containing the experimental info.
- **property_name** (*str*) – The name of the simulated property to compare against experiment
- **max_depth** (*int*) – Fit at each depth up to (and including) max_depth

Returns Results of the fit

Return type ModelResult

idpflex.bayes.**fit_to_depth** (*tree*, *experiment*, *property_name*, *max_depth=5*)
Fit at each tree depth from the root node up to a maximum depth

Fit experiment against the property stored in the nodes. The fit model is generated by *model_at_depth()*

Parameters

- **tree** (*Tree*) – Hierarchical tree
- **experiment** (*ProfileProperty*) – A property containing the experimental info.
- **property_name** (*str*) – The name of the simulated property to compare against experiment
- **max_depth** (*int*) – Fit at each depth up to (and including) max_depth

Returns A list of ModelResult items containing the fit at each level of the tree up to and including *max_depth*

Return type list

idpflex.bayes.**model_at_depth** (*tree*, *depth*, *property_name*)
Generate a fit model at a particular tree depth

Parameters

- **tree** (*Tree*) – Hierarchical tree
- **depth** (*int*) – depth level, starting from the tree's root (depth=0)

- **property_name** (*str*) – Name of the property to create the model for

Returns A model composed of a *TabulatedFunctionModel* for each node plus a ConstantModel accounting for a flat background

Return type CompositeModel

idpflex.bayes.**model_at_node**(*node*, *property_name*)

Generate fit model as a tabulated function with a scaling parameter, plus a flat background

Parameters

- **node** (*ClusterNodeX*) – One node of the hierarchical *Tree*
- **property_name** (*str*) – Name of the property to create the model for

Returns A model composed of a *TabulatedFunctionModel* and a ConstantModel

Return type CompositeModel

1.3.2 cluster : group trajectory frames by structural similarity

class idpflex.cluster.**ClusterTrove**

Bases: *idpflex.cluster.ClusterTrove*

A namedtuple with a *keys()* method for easy access of fields, which are described below under header *Parameters*

Parameters

- **idx** (list) – Frame indexes for the representative structures (indexes start at zero)
- **rmsd** (ndarray) – distance matrix between representative structures.
- **tree** (*Tree*) – Clustering of representative structures. Leaf nodes associated with each centroid contain property *iframe*, which is the frame index in the trajectory pointing to the atomic structure corresponding to the centroid.

keys()

Return the list of field names

save(*filename*)

Serialize the cluster trove and save to file

Parameters **filename** (*str*) – File name

idpflex.cluster.**cluster_trajectory**(*a_universe*, *selection='not name H*'*, *segment_length=1000*, *n_representatives=1000*)

Cluster a set of representative structures by structural similarity (RMSD)

The simulated trajectory is divided into segments, and hierarchical clustering is performed on each segment to yield a limited number of representative structures. These are then clustered into the final hierachical tree.

Parameters

- **a_universe** (Universe) – Topology and trajectory.
- **selection** (*str*) – atoms for which to calculate RMSD. See the selections page for atom selection syntax.
- **segment_length** (*int*) – divide trajectory into segments of this length
- **n_representatives** (*int*) – Desired total number of representative structures. The final number may be close but not equal to the desired number.
- **distance_matrix** (ndarray)

Returns clustering results for the representatives

Return type *ClusterTrove*

idpflex.cluster.**cluster_with_properties**(*a_universe*, *pcls*, *p_names=None*, *selection='not name H*'*, *segment_length=1000*, *n_representatives=1000*)

Cluster a set of representative structures by structural similarity (RMSD) and by a set of properties

The simulated trajectory is divided into segments, and hierarchical clustering is performed on each segment to yield a limited number of representative structures (the centroids). Properties are calculated for each centroid, thus each centroid is described by a property vector. The dimensionality of the vector is related to the number of properties and the dimensionality of each property. The distances between any two centroids is calculated as the Euclidean distance between their respective vector properties. The distance matrix containing distances between all possible centroid pairs is employed as the similarity measure to generate the hierarchical tree of centroids.

The properties calculated for the centroids are stored in the leaf nodes of the hierarchical tree. Properties are then propagated up to the tree's root node.

Parameters

- **a_universe** (*Universe*) – Topology and trajectory.
- **pcls** (*list*) – Property classes, such as *Asphericity* of *SaSa*
- **p_names** (*list*) – Property names. If None, then default property names are used
- **selection** (*str*) – atoms for which to calculate RMSD. See the selections page for atom selection syntax.
- **segment_length** (*int*) – divide trajectory into segments of this length
- **n_representatives** (*int*) – Desired total number of representative structures. The final number may be close but not equal to the desired number.

Returns Hierarchical clustering tree of the centroids

Return type *ClusterTrove*

idpflex.cluster.**load_cluster_trove**(*filename*)

Load a previously saved *ClusterTrove* instance

Parameters **filename** (*str*) – File name containing the serialized *ClusterTrove*

Returns Cluster trove instance stored in file

Return type *ClusterTrove*

idpflex.cluster.**propagator_size_weighted_sum**(*values*, *tree*)

Calculate a property of the node as the sum of its siblings' property values, weighted by the relative cluster sizes of the siblings.

Parameters

- **values** (*list*) – List of property values (of same type), one item for each leaf node.
- **node_tree** (*Tree*) – Tree of *ClusterNodeX* nodes

idpflex.cluster.**trajectory_centroids**(*a_universe*, *selection='not name H*'*, *segment_length=1000*, *n_representatives=1000*)

Cluster a set of consecutive trajectory segments into a set of representative structures via structural similarity (RMSD)

The simulated trajectory is divided into consecutive segments, and hierarchical clustering is performed on each segment to yield a limited number of representative structures (centroids) per segment.

Parameters

- **a_universe** (Universe) – Topology and trajectory.
- **selection** (*str*) – atoms for which to calculate RMSD. See the selections page for atom selection syntax.
- **segment_length** (*int*) – divide trajectory into segments of this length
- **n_representatives** (*int*) – Desired total number of representative structures. The final number may be close but not equal to the desired number.

Returns **rep_ifr** – Frame indexes of representative structures (centroids)

Return type list

1.3.3 cnextend : Functionality for the hiearchical tree

class idpflex.cnextend.**ClusterNodeX**(**args*, ***kwargs*)

Bases: scipy.cluster.hierarchy.ClusterNode

Extension of ClusterNode to accommodate a parent reference and a protected dictionary of properties.

add_property(*a_property*)

Insert or update a property in the set of properties

Parameters **a_property** (*ProfileProperty*) – a property instance

distance_submatrix(*dist_mat*)

Extract matrix of distances between leafs under the node.

Parameters **dist_mat** (*numpy.ndarray*) – Distance matrix (square or in condensed form) among all N leaves of the tree to which the node belongs to. The row indexes of *dist_mat* must correspond to the node IDs of the leaves.

Returns square distance matrix MxM between the M leafs under the node

Return type ndarray

leaf_ids

ID's of the leafs under the tree, ordered by increasing ID.

Returns

Return type list

leafs

Find the leaf nodes under this cluster node.

Returns node leafs ordered by increasing ID

Return type list

representative(*dist_mat*, *similarity=<function mean>*)

Find leaf under node that is most similar to all other leaves under the node

Find the leaf that minimizes the similarity between itself and all the other leaves under the node. For instance, the average of all distances between one leaf and all the other leaves results in a similarity scalar for the leaf.

Parameters

- **dist_mat** (ndarray) – condensed or square distance matrix MxM or NxN among all N leaves in the tree or among all M leaves under the node. If dealing with the distance matrix among all leaves in the tree, self.distance_submatrix is first applied.
- **similarity** (*function object*) – reduction operation on a the list of distances between one leaf and the other (M-1) leaves.

Returns representative leaf node

Return type *ClusterNodeX*

tree
Tree object owning the node

Returns

Return type *Tree*

class idpflex.cnextend.**Tree**(*z=None*)
Bases: object

Hierarchical binary tree.

Parameters **z** (ndarray) – linkage matrix from which to create the tree. See linkage()

from_linkage_matrix(*z*, *node_class=<class idpflex.cnextend.ClusterNodeX>*)
Refactored to_tree() converts a hierarchical clustering encoded in matrix *z* (by linkage) into a convenient tree object.

Each *node_class* instance has a *left*, *right*, *dist*, *id*, and *count* attribute. The *left* and *right* attributes point to *node_class* instances that were combined to generate the cluster. If both are *None* then *node_class* is a leaf node, its count must be 1, and its distance is meaningless but set to 0.

Parameters

- **z** (ndarray) – linkage matrix. See linkage()
- **node_class** (*ClusterNodeX*) – the type of nodes composing the tree. Now supports *ClusterNodeX* and parent class ClusterNode

leafs

Returns leaf nodes ordered by increasing ID

Return type list

nodes_above_depth(*depth=0*)
Nodes at or above depth from the root node

Parameters **depth** (*int*) – Depth level starting from the root level (depth=0)

Returns List of nodes ordered by increasing ID. Last one is the root node

Return type list

nodes_at_depth(*depth=0*)
Nodes at a given depth from the root node

Parameters **depth** (*int*) – Depth level starting from the root level (depth=0)

Returns List of nodes corresponding to that particular level

Return type list

save(*filename*)
Serialize the tree and save to file

Parameters **filename** (*str*) – File name

idpflex.cnextend.**load_tree**(*filename*)
Load a previously saved tree

Parameters **filename** (*str*) – File name containing the serialized tree

Returns Tree instance stored in file

Return type *Tree*

idpflex.cnextend.**random_distance_tree**(**args*, ***kwargs*)
Instantiate a tree where leafs and nodes have random distances to each other.

Distances randomly retrieved from a flat distribution of numbers between 0 and 1

Parameters **n_leafs** (*int*) – Number of tree leaves

Returns

Elements of the named tuple: - tree: *Tree*

Tree instance

- **distance_matrix:** **ndarray** square distance matrix in between pair of tree leafs

Return type namedtuple

1.3.4 distances : Utility functions to calculate structural similarity

idpflex.distances.**distance_submatrix**(*dist_mat*, *indexes*)
Extract matrix of distances for a subset of indexes

If matrix is in condensed format, then the submatrix is returned in condensed format too.

Parameters

- **dist_mat** (ndarray) – NxN distance matrix
- **indexes** (*sequence of int*) – sequence of indexes from which a submatrix is extracted.

Returns

Return type ndarray

idpflex.distances.**extract_coordinates**(*a_universe*, *group*, *indexes=None*)
Obtain XYZ coordinates for an atom group and for a subset of frames

Parameters

- **a_universe** (Universe) – Topology and trajectory.
- **group** (AtomGroup) – Atom selection.
- **indexes** (list) – sequence of frame indexes

Returns XYZ coordinates shape=(M, N, 3) with M number of indexes and N number of atoms in group.

Return type ndarray

idpflex.distances.**rmsd_matrix**(*xyz*, *condensed=False*)
RMSD matrix between coordinate frames.

Parameters

- **xyz** (ndarray) – Bare coordinates shape=(N, M, 3) with N: number of frames, M: number of atoms
- **condensed** (*bool*) – Flag return matrix as square or condensed

Returns Square NxN matrix, or condensed N*(N+1)/2 matrix

Return type ndarray

1.3.5 properties : Injection of properties in a tree's node

class idpflex.properties.**Asphericity**(**args*, ***kwargs*)

Bases: *idpflex.properties.ScalarProperty*, *idpflex.properties.AsphericityMixin*

Implementation of a node property to store the asphericity from the gyration radius tensor

$\frac{(L_1-L_2)^2+(L_1-L_3)^2+L_2-L_3)^2}{2(L_1+L_2+L_3)^2}$

where L_i are the eigenvalues of the gyration tensor. Units are same as units of a_universe.

Reference: https://pubs.acs.org/doi/pdf/10.1021/ja206839u

Does not apply periodic boundary conditions

See *ScalarProperty* for initialization

asphericity

Property to read and set the asphericity

default_name = 'asphericity'

class idpflex.properties.**AsphericityMixin**

Bases: object

Mixin class providing a set of methods to calculate the asphericity from the gyration radius tensor

from_pdb(*filename*, *selection=None*)

Calculate asphericity from a PDB file

$\frac{(L_1-L_2)^2+(L_1-L_3)^2+L_2-L_3)^2}{2(L_1+L_2+L_3)^2}$

where L_i are the eigenvalues of the gyration tensor. Units are same as units of a_universe.

Does not apply periodic boundary conditions

Parameters

- **filename** (*str*) – path to the PDB file
- **selection** (*str*) – Atomic selection. All atoms are considered if None is passed. See the selections page for atom selection syntax.

Returns **self** – Instantiated Asphericity object

Return type *Asphericity*

from_universe(*a_universe*, *selection=None*, *index=0*)

Calculate asphericity from an MDAnalysis universe instance

$\frac{(L_1-L_2)^2+(L_1-L_3)^2+L_2-L_3)^2}{2(L_1+L_2+L_3)^2}$

where L_i are the eigenvalues of the gyration tensor. Units are same as units of a_universe.

Does not apply periodic boundary conditions

Parameters

- **a_universe** (Universe) – Trajectory or single-conformation instance
- **selection** (*str*) – Atomic selection. All atoms considered if None is passed. See the selections page for atom selection syntax.

Returns **self** – Instantiated Asphericity object

Return type *Asphericity*

class idpflex.properties.**EndToEnd**(**args*, ***kwargs*)
Bases: *idpflex.properties.ScalarProperty*, *idpflex.properties.EndToEndMixin*

Implementation of a node property to store the end-to-end distance

See *ScalarProperty* for initialization

default_name = 'end_to_end'

end_to_end
Property to read and set the end-to-end distance

class idpflex.properties.**EndToEndMixin**
Bases: object

Mixin class providing a set of methods to load and calculate the end-to-end distance for a protein

from_pdb(*filename*, *selection='name CA'*)
Calculate end-to-end distance from a PDB file

Does not apply periodic boundary conditions

Parameters

- **filename** (*str*) – path to the PDB file
- **selection** (*str*) – Atomic selection. The first and last atoms of the selection are considered for the calculation of the end-to-end distance. See the selections page for atom selection syntax.

Returns **self** – Instantiated EndToEnd object

Return type *EndToEnd*

from_universe(*a_universe*, *selection='name CA'*, *index=0*)
Calculate radius of gyration from an MDAnalysis Universe instance

Does not apply periodic boundary conditions

Parameters

- **a_universe** (Universe) – Trajectory or single-conformation instance
- **selection** (*str*) – Atomic selection. The first and last atoms of the selection are considered for the calculation of the end-to-end distance. See the selections page for atom selection syntax.

Returns **self** – Instantiated EndToEnd object

Return type *EndToEnd*

class idpflex.properties.**ProfileProperty**(*name=None*, *qvalues=None*, *profile=None*, *errors=None*)
Bases: object

Implementation of a node property valid for SANS or X-Ray data.

Parameters

- **name** (*str*) – Property name.
- **qvalues** (ndarray) – Momentun transfer domain
- **profile** (ndarray) – Intensity values
- **errors** (ndarray) – Errors in the intensity values

default_name = 'profile'

e
(ndarray) intensity errors

Type property *errors*

name
(str) name of the profile

Type property *name*

x
(ndarray) momentum transfer values

Type property *qvalues*

y
(ndarray) profile intensities

Type property *profile*

class idpflex.properties.**RadiusOfGyration**(**args*, ***kwargs*)
Bases: *idpflex.properties.ScalarProperty*, *idpflex.properties.RadiusOfGyrationMixin*

Implementation of a node property to store the radius of gyration.

See *ScalarProperty* for initialization

default_name = 'rg'

rg
Property to read and write the radius of gyration value

class idpflex.properties.**RadiusOfGyrationMixin**
Bases: object

Mixin class providing a set of methods to load the Radius of Gyration data into a Scalar property

from_pdb(*filename*, *selection=None*)
Calculate Rg from a PDB file

Parameters

- **filename** (*str*) – path to the PDB file
- **selection** (*str*) – Atomic selection for calculating Rg. All atoms considered if default None is passed. See the selections page for atom selection syntax.

Returns **self** – Instantiated RadiusOfGyration property object

Return type *RadiusOfGyration*

from_universe(*a_universe*, *selection=None*, *index=0*)
Calculate radius of gyration from an MDAnalysis Universe instance

Parameters

- **a_universe** (Universe) – Trajectory, or single-conformation instance.

- **selection** (*str*) – Atomic selection. All atoms considered if None is passed. See the selections page for atom selection syntax.

Returns **self** – Instantiated RadiusOfGyration object

Return type *RadiusOfGyration*

class idpflex.properties.**ResidueContactMap**(*name=None*, *selection=None*, *cmap=None*, *errors=None*, *cutoff=None*)

Bases: object

Contact map between residues of the conformation using different definitions of contact.

Parameters

- **name** (*str*) – Name of the contact map
- **selection** (AtomGroup) – Atomic selection for calculation of the contact map, which is then projected to a residue based map. See the selections page for atom selection syntax.
- **cmap** (ndarray) – Contact map between residues of the atomic selection
- **errors** (ndarray) – Underterminacies for every contact of cmap
- **cutoff** (*float*) – Cut-off distance defining a contact between two atoms

default_name = 'cm'

e

(ndarray) undeterminacies in the contact map

Type property *errors*

from_pdb(*filename*, *cutoff*, *selection=None*)

Calculate residue contact map from a PDB file

Parameters

- **filename** (*str*) – Path to the file in PDB format
- **cutoff** (*float*) – Cut-off distance defining a contact between two atoms
- **selection** (*str*) – Atomic selection for calculating interatomic contacts. All atoms are used if None is passed. See the selections page for atom selection syntax.

Returns **self** – Instantiated ResidueContactMap object

Return type *ResidueContactMap*

from_universe(*a_universe*, *cutoff*, *selection=None*, *index=0*)

Calculate residue contact map from an MDAnalysis Universe instance

Parameters

- **a_universe** (Universe) – Trajectory or single-conformation instance
- **cutoff** (*float*) – Cut-off distance defining a contact between two atoms
- **selection** (*str*) – Atomic selection for calculating interatomic contacts. All atoms are used if None is passed. See the selections page for atom selection syntax.

Returns **self** – Instantiated ResidueContactMap object

Return type *ResidueContactMap*

name

(str) name of the contact map

Type property *name*

plot()
Plot the residue contact map of the node

x
(AtomGroup) atom selection

Type property *selection*

y
(ndarray) contact map between residues

Type property *cmap*

class idpflex.properties.**SaSa**(**args*, ***kwargs*)
Bases: *idpflex.properties.ScalarProperty*, *idpflex.properties.SaSaMixin*

Implementation of a node property to calculate the Solvent Accessible Surface Area.

See *ScalarProperty* for initialization

default_name = 'sasa'

sasa
Property to read and write the SASA value

class idpflex.properties.**SaSaMixin**
Bases: object

Mixin class providing a set of methods to load and calculate the solvent accessible surface area

from_mdtraj(*a_traj*, *probe_radius=1.4*, ***kwargs*)
Calculate solvent accessible surface for frames in a trajectory

SASA units are Angstroms squared

Parameters

- **a_traj** (Trajectory) – mdtraj trajectory instance
- **probe_radius** (*float*) – The radius of the probe, in Angstroms
- **kwargs** (*dict*) – Optional arguments for the underlying mdtraj.shrake_rupley algorithm doing the actual SaSa calculation

Returns **self** – Instantiated SaSa property object

Return type *SaSa*

from_pdb(*filename*, *selection=None*, *probe_radius=1.4*, ***kwargs*)
Calculate solvent accessible surface area (SASA) from a PDB file

If the PBD contains more than one structure, calculation is performed only for the first one.

SASA units are Angstroms squared

Parameters

- **filename** (*str*) – Path to the PDB file
- **selection** (*str*) – Atomic selection for calculating SASA. All atoms considered if default None is passed. See the
- **'selections page <https** (//www.mdanalysis.org/docs/documentation_pages/selections.html>'_)
- **for atom selection syntax.**
- **probe_radius** (*float*) – The radius of the probe, in Angstroms

- **kwargs** (*dict*) –

  Optional arguments for the underlying mdtraj.shrake_rupley algorithm doing the actual SaSa calculation

Returns **self** – Instantiated SaSa property object

Return type *SaSa*

from_universe(*a_universe*, *selection=None*, *probe_radius=1.4*, *index=0*, ***kwargs*)

Calculate solvent accessible surface area (SASA) from an MDAnalysis universe instance.

This method is a thin wrapper around method *from_pdb()*

Parameters

- **a_universe** (Universe) – Trajectory or single-conformation instance
- **selection** (*str*) – Atomic selection for calculating SASA. All atoms considered if default None is passed. See the
- **'selections page <https** (//www.mdanalysis.org/docs/documentation_pages/selections.html>'_)
- **for atom selection syntax.**
- **probe_radius** (*float*) – The radius of the probe, in Angstroms
- **kwargs** (*dict*) – Optional arguments for underlying mdtraj.shrake_rupley doing the actual SASA calculation.

Returns **self** – Instantiated SaSa property object

Return type *SaSa*

class idpflex.properties.**SansLoaderMixin**

Bases: object

Mixin class providing a set of methods to load SANS data into a profile property

from_sassena(*handle*, *profile_key='fqt'*, *index=0*)

Load SANS profile from sassena output.

It is assumed that Q-values are stored under item *qvalues* and listed under the *X* column.

Parameters

- **handle** (*h5py.File*) – h5py reading handle to HDF5 file
- **profile_key** (*str*) – item key where profiles are stored in the HDF5 file
- **param index** (*int*) – profile index, if data contains more than one profile

Returns **self**

Return type *SansProperty*

class idpflex.properties.**SansProperty**(**args*, ***kwargs*)

Bases: *idpflex.properties.ProfileProperty*, *idpflex.properties.SansLoaderMixin*

Implementation of a node property for SANS data

default_name = 'sans'

class idpflex.properties.**SaxsLoaderMixin**

Bases: object

Mixin class providing a set of methods to load X-ray data into a profile property

from_ascii (*file_name*)
Load profile from an ascii file.

Expected file format:
Rows have three items separated by a blank space:
- *col1* momentum transfer
- *col2* profile
- *col3* errors of the profile

Parameters **file_name** (*str*) – File path

Returns self

Return type *SaxsProperty*

from_crysol_fit (*file_name*)
Load profile from a crysol *.fit file.

Parameters **file_name** (*str*) – File path

Returns self

Return type *SaxsProperty*

from_crysol_int (*file_name*)
Load profile from a crysol *.int file

Parameters **file_name** (*str*) – File path

Returns self

Return type *SaxsProperty*

from_crysol_pdb (*file_name*, *command='crysol'*, *args='-lm 20 -sm 0.6 -ns 500 -un 1 -eh -dro 0.075'*, *silent=True*)
Calculate profile with crysol from a PDB file

Parameters

- **file_name** (*str*) – Path to PDB file
- **command** (*str*) – Command to invoke crysol
- **args** (*str*) – Arguments to pass to crysol
- **silent** (*bool*) – Suppress crysol standard output and standard error

Returns self

Return type *SaxsProperty*

to_ascii (*file_name*)
Save profile as a three-column ascii file.

Rows have three items separated by a blank space
- *col1* momentum transfer
- *col2* profile
- *col3* errors of the profile

class idpflex.properties.**SaxsProperty**(**args*, ***kwargs*)
Bases: *idpflex.properties.ProfileProperty*, *idpflex.properties.SaxsLoaderMixin*

Implementation of a node property for SAXS data

default_name = 'saxs'

class idpflex.properties.**ScalarProperty**(*name=None*, *x=0.0*, *y=0.0*, *e=0.0*)
Bases: object

Implementation of a node property for a number plus an error.

Instances have *name*, *x*, *y*, and *e* attributes, so they will follow the property node protocol.

Parameters

- **name** (*str*) – Name associated to this type of property
- **x** (*float*) – Domain of the property
- **y** (*float*) – value of the property
- **e** (*float*) – error of the property's value

histogram(*bins=10*, *errors=False*, ***kwargs*)
Histogram of values for the leaf nodes

Parameters

- **nbins** (*int*) – number of histogram bins
- **errors** (*bool*) – estimate error from histogram counts
- **kwargs** (*dict*) – Additional arguments to underlying histogram()

Returns

- ndarray – histogram bin edges
- ndarray – histogram values
- ndarray – Errors for histogram counts, if *error=True*. Otherwise None.

plot(*kind='histogram'*, *errors=False*, ***kwargs*)

Parameters

- **kind** (*str*) – 'histogram': Gather Rg for the leafs under the node associated to this property, then make a histogram.
- **errors** (*bool*) – Estimate error from histogram counts
- **kwargs** (*dict*) – Additional arguments to underlying hist()

Returns Axes object holding the plot

Return type Axes

set_scalar(*y*)

class idpflex.properties.**SecondaryStructureProperty**(*name=None*, *aa=None*, *profile=None*, *errors=None*)
Bases: object

Node property for secondary structure determined by DSSP

Every residue is assigned a vector of length 8. Indexes corresponds to different secondary structure assignment:

Index__|__DSSP code__|__ Color__|__Structure__|
==========
__0__|__H__|__yellow__|__Alpha helix (4-12)
__1__|__B__|__pink__|__Isolated beta-bridge residue
__2__|__E__|__red__|__Strand
__3__|__G__|__orange__|__3-10 helix
__4__|__I__|__green__|__Pi helix
__5__|__T__|__magenta__|__Turn
__6__|__S__|__cyan__|__Bend
__7__|_____|__white__|__Unstructured (coil)

We follow here Bio.PDB.DSSP ordering

For a leaf node (single structure), the vector for any given residue will be all zeroes except a value of one for the corresponding assigned secondary structure. For all other nodes, the vector will correspond to a probability distribution among the different DSSP codes.

Parameters

- **name** (*str*) – Property name
- **aa** (*str*) – One-letter amino acid sequence encoded in a single string
- **profile** (ndarray) – N x 8 matrix with N number of residues and 8 types of secondary structure
- **errors** (ndarray) – N x 8 matrix denoting undeterminacies for each type of assigned secondary residue in every residue

classmethod code2profile(*code*)

Generate a secondary structure profile vector for a particular DSSP code

Parameters **code** (*str*) – one-letter code denoting secondary structure assignment

Returns profile vector

Return type ndarray

collapsed

For every residue, collapse the secondary structure profile onto the component with the highest probability

Returns List of indexes corresponding to collapsed secondary structure states

Return type ndarray

colors = ('yellow', 'pink', 'red', 'orange', 'green', 'magenta', 'cyan', 'white')

associated colors to each element of secondary structure

default_name = 'ss'

disparity(*other*)

Secondary Structure disparity of other profile to self, akin to χ^2

$\frac{1}{N(n-1)} \sum_{i=1}^{N} \sum_{j=1}^{n} (\frac{p_{ij}-q_{ij}}{e})^2$

with N number of residues and n number of DSSP codes. Errors e are those of *self*, and are set to one if they have not been initialized. We divide by $n-1$ because it is implied a normalized distribution of secondary structure elements for each residue.

Parameters **other** (*SecondaryStructureProperty*) – Secondary structure property to compare to

Returns disparity measure

Return type float

dssp_codes = 'HBEGITS '
list of single-letter codes for secondary structure. Last code is a blank space denoting no secondary structure (Unstructured)

e
(ndarray) assignment undeterminacy

Type property *errors*

elements = {' ': 'Unstructured', 'B': 'Isolated beta-bridge', 'E': 'Strand', 'G': '3-
Description of single-letter codes for secondary structure

fractions
Output fraction of each element of secondary structure.

Fractions are computed summing over all residues.

Returns Elements of the form {single-letter-code: fraction}

Return type dict

from_dssp(*file_name*)
Load secondary structure profile from a dssp file

Parameters **file_name** (*str*) – File path

Returns self

Return type *SecondaryStructureProperty*

from_dssp_pdb(*file_name*, *command='mkdssp'*, *silent=True*)
Calculate secondary structure with DSSP

Parameters

- **file_name** (*str*) – Path to PDB file
- **command** (*str*) – Command to invoke dssp. You need to have DSSP installed in your machine
- **silent** (*bool*) – Suppress DSSP standard output and error

Returns self

Return type *SecondaryStructureProperty*

from_dssp_sequence(*codes*)
Load secondary structure profile from a single string of DSSP codes

Attributes *aa* and *errors* are not modified, only **profile**.

Parameters **codes** (*str*) – Sequence of one-letter DSSP codes

Returns self

Return type *SecondaryStructureProperty*

n_codes = 8
number of distinctive elements of secondary structure

name
(str) name of the profile

Type property *name*

plot (*kind='percents'*)
Plot the secondary structure of the node holding the property

Parameters **kind** (*str*) – 'percents': bar chart with each bar denoting the percent of a particular secondary structure in all the protein; — 'node': gray plot of secondary structure element probabilities for each residue; — 'leafs': color plot of secondary structure for each leaf under the node. Leafs are sorted by increasing disparity to the secondary structure of the node.

x
(str) amino-acid sequence

Type property *aa*

y
(ndarray) secondary structure assignment

Type property *profile*

idpflex.properties.**decorate_as_node_property**(*nxye*)
Decorator that endows a class with the node property protocol

For details, see *register_as_node_property()*

Parameters **nxye** (*list*) – list of (name, description) pairs denoting the property components

idpflex.properties.**propagator_size_weighted_sum**(*values*, *tree*)

Calculate a property of the node as the sum of its siblings' property values, weighted by the relative cluster sizes of the siblings.

Parameters

- **values** (*list*) – List of property values (of same type), one item for each leaf node.
- **node_tree** (*Tree*) – Tree of *ClusterNodeX* nodes

idpflex.properties.**propagator_weighted_sum**(*values*, *tree*, *weights=<function <lambda>>*)
Calculate the property of a node as the sum of its two siblings' property values. Propagation applies only to non-leaf nodes.

Parameters

- **values** (*list*) – List of property values (of same type), one item for each leaf node.
- **tree** (*Tree*) – Tree of *ClusterNodeX* nodes
- **weights** (*tuple*) – Callable of two arguments (left-node and right-node) returning a tuple of left and right weights. Default callable returns (1.0, 1.0) always.

idpflex.properties.**register_as_node_property**(*cls*, *nxye*)
Endows a class with the node property protocol.

The node property assumes the existence of these attributes
- *name* name of the property
- *x* property domain
- *y* property values
- *e* errors of the property values

This function will endow class *cls* with these attributes, implemented through the python property pattern. Names for the corresponding storage attributes must be supplied when registering the class.

Parameters

- **cls** (*class type*) – The class type
- **nxye** (*tuple (len==4)*) – nxye is a four element tuple. Its elements are in this order:

  (property name, 'stores the name of the property'), (domain_storage_attribute_name, description of the domain), (values_storage_attribute_name, description of the values), (errors_storage_attribute_name, description of the errors)

  Example:

  (('name', 'stores the name of the property'), ('qvalues', 'momentum transfer values'), ('profile', 'profile intensities'), ('errors', 'intensity errors'))

idpflex.properties.**weights_by_size**(*left_node*, *right_node*)

Calculate the relative size of two nodes

Parameters

- **left_node** (*ClusterNodeX*) – One of the two sibling nodes
- **right_node** (*ClusterNodeX*) – One of the two sibling nodes

Returns Weights representing the relative populations of two nodes

Return type tuple

1.3.6 utils : Miscellanea boiler plate

idpflex.utils.**namedtuplefy**(*func*)

Decorator to transform the return dictionary of a function into a namedtuple

Parameters

- **func** (*Function*) – Function to be decorated
- **name** (*str*) – Class name for the namedtuple. If None, the name of the function will be used

Returns

Return type Function

idpflex.utils.**temporary_file**(**args*, ***kwds*)

Creates a temporary file

Parameters **kwargs** (*dict*) – optional arguments to tempfile.mkstemp

Yields *str* – Absolute path name to file

idpflex.utils.**write_frame**(*a_universe*, *iframe*, *file_name*)

Write a single trajectory frame to file.

Format is guessed from the file's extension.

Parameters

- **a_universe** (*Universe*) – Universe describing the simulation
- **iframe** (*int*) – Trajectory frame index (indexes begin with zero)
- **file_name** (*str*) – Name of the file to create

1.4 Contributing

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

You can contribute in many ways:

1.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/jmborr/idpflex/issues.

If you are reporting a bug, please include:

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

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/jmborr/idpflex/issues.

If you are proposing a feature:

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

1.4.2 Get Started!

Ready to contribute? Here's how to set up *idpflex* for local development.

1. Fork the *idpflex* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/idpflex.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv idpflex
$ cd idpflex/
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 idpflex tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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

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

7. Submit a pull request through the GitHub website.

1.4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/jmborr/idpflex/pull_requests and make sure that the tests pass for all supported Python versions.

1.4.4 Tips

To run a subset of tests:

```
$ py.test tests.test_idpflex
```

1.5 Credits

1.5.1 Development Lead

- Jose Borreguero <borreguero@gmail.com>

1.5.2 Contributors

- Fahima Islam

1.5.3 Collaborators

- Utsab Shrestha
- Loukas Petridis

1.6 History

1.6.1 0.1.8 (2019-01-12)

- cluster random distance (PR #87)
- decorator to create a namedtuple out of a dictionary (PR #86)
- drop support for python 2.x (PR #83)
- Add function fit_at_dept (PR #81)

1.6.2 0.1.7 (2018-12-07)

- Check for executable dssp (PR #79)
- Conda support to build readthedocs (PR #76)
- Check for executable crysol (PR #77)
- Added a statement-of-need (PR #73)

1.6.3 0.1.6 (2018-08-17)

- implement K-means clustering (PR #67)
- Added Asphericity property (PR #65)
- Added SASA property (PR #64)
- Added end-to-end property (PR #59)
- plot histogram for ScalarProperty (PR #56)
- Added Radius of Gyration property (PR #53)

1.6.4 0.1.5 (2018-02-15)

- Update notebook and docs with data repo (PR #51)

1.6.5 0.1.4 (2018-02-11)

- Fix secondary structure plots

1.6.6 0.1.3 (2018-02-10)

- Bug in add_property

1.6.7 0.1.2 (2018-02-10)

- Clustering Jupyter notebook
- Secondary structure property

1.6.8 0.1.1.0 (2018-01-11)

- Parallel calculation of RMSD

1.6.9 0.1.0.2 (2018-01-10)

- Integrate travis, github, and readthedocs

1.6.10 0.1.0.1 (2018-01-09)

- readthedocs fixed

1.6.11 0.1.0.0 (2017-12-15)

- First release on PyPI.

It is estimated that about 30% of the eucariotic proteome consists of intrinsically disordered proteins (IDP's), yet their presence in public structural databases is severely underrepresented. IDP's adopt heterogeneous inter-converting conformations with similar probabilities, preventing resolution of structures with X-Ray diffraction techniques. An alternative technique with wide application on IDP systems is small angle scattering (SAS). SAS can measure average structural features of IDP's when in vitro solution, or even at conditions mimicking protein concentrations found in the cell's cytoplasm.

Despite these advantages, the averaging nature of SAS measurements will prove unsatisfactory if one aims to differentiate among the different conformations that a particular IDP can adopt. Different distributions of conformations can yield the same average therefore it is not possible to retrace the true distribution if all that SAS provides is the average conformation.

To address this shortcoming, atomistic molecular dynamics (MD) simulations of IDP systems combined with enhanced sampling methods such as the Hamiltonian replica exchange method are specially suitable [ea06]. These simulations can probe extensive regions of the IDP's conformational space and have the potential to offer a full-featured description of the conformational landscape of IDP's. The results of these simulations should not be taken at faith value, however.

First, a proper comparison against available experimental SAS data is a must. This validation step is the requirement that prompted the development of *idpflex*.

The python package *idpflex* clusters the 3D conformations resulting from an MD simulation into a hierarchical tree by means of structural similarity among pairs of conformations. The conformations produced by the simulation take the role of Leafs in the hierarchichal tree. Nodes in the tree take the rol of IDP substates, with conformations under a particular Node making up one substate. Strictly speaking, *idfplex* does not require the IDP conformations to be produced by an MD simulation. Alternative conformation generators can be used, such as torsional sampling of the protein backbone [eal12]. In contrast to other methods [eal11], *idpflex* does not initially discard any conformation by labelling it as incompatible with the experimental data. This data is an average over all conformations, and using this average as the criterion by which to discard any specific conformation can lead to erroneous discarding decisions by the reasons stated above.

Default clustering is performed according to structural similarity between pairs of conformations, defined by the root mean square deviation algorithm [Kab76]. Alternatively, *idpflex* can cluster conformations according to an Euclidean distance in an abstract space spanned by a set of structural properties, such as radius of gyration and end-to-end distance. Comparison to experimental SAS data is carried out first by calculating the SAS intensities [eal95] for each conformation produced by the MD simulation. This result in SAS intensities for each Leaf in the hierarchical tree. Intensities are then propagated up the hierarchical tree, yielding a SAS intensity for each Node. Because each Node takes the role of a conformational substate, we obtain SAS intensities for each substate. *idpflex* can compare the SAS intensity of each substate against the experimental SAS data. Also, it can average intensities from different substates and compare against experimental SAS data. The fitting functionality included in *idpflex* allows for selection of the set of substates that will yield maximal similarity between computed and experimental SAS intensities. Thus, arranging tens of thousands of conformations into (typically) less than ten substates provides the researcher with a manageable number of conformations from which to derive meaningful conclusions regarding the conformational variability of IDP's.

idpflex also provides a set of convenience functions to compute structural features of IDP's for each of the conformations produced by the MD simulation. These properties can then be propagated up the hierarchical tree much in the same way as SAS intensities are propagated. Thus, one can compute for each substate properties such as radius of gyration, end-to-end distance, asphericity, solvent exposed surface area, contact maps, and secondary structure content. All these structural properties require atomistic detail, thus *idpflex* is more apt for the study of IDP's than for the study of quaternary protein arrangements, where clustering of coarse-grain simulations becomes a better option [eal11]. *idpflex* wraps other python packages (MDAnalysis [eal1}], [eal6}], mdtraj [eal5}]) and third party applications (CRYSOL [eal95], DSSP [WKS3}]) that actually carry out the calculation of said properties. Additional properties can be incorporated by inheriting from the base Property classes.

To summarize, *idpflex* integrates MD simulations with SAS experiments in order to obtain a manageable representation of the rich conformational diversity of IDP's, a pertinent problem in structural biology.

CHAPTER 2

Contact

Post your questions and comments on the gitter lobby

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Source: https://github.com/jmborr/idpflex

CHAPTER 4

Acknowledgements

This work is sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle LLC, for DOE. Part of this research is supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, User Facilities under contract number DE-AC05-00OR22725.

CHAPTER 5

References

Bibliography

[ea06] R. Affentranger et al. A novel Hamiltonian replica exchange MD protocol to enhance protein conformational space sampling. *Journal of Chemical Theory and Computation*, 2(2):217-228, MAR-APR 2006. doi:10.1021/ct050250b.

[eal11] Bartosz Rozycki et al. SAXS Ensemble Refinement of ESCRT-III CHMP3 Conformational Transitions. *Structure*, 19(1):109-116, JAN 12 2011. doi:10.1016/j.str.2010.10.006.

[eal95] D. Svergun et al. CRYSOL - A program to evaluate X-ray solution scattering of biological macromolecules from atomic coordinates. *Journal of Applied Crystallography*, 28(6):768-773, DEC 1 1995. doi:10.1107/S0021889895007047.

[eal12] Joseph E. Curtis et al. SASSIE: A program to study intrinsically disordered biological molecules and macromolecular ensembles using experimental scattering restraints. *Computer Physics Communications*, 183(2):382-389, FEB 2012. doi:10.1016/j.cpc.2011.09.010.

[eal1}] N. Michaud-Agrawal et al. MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations. *Journal of Computational Chemistry*, 32:2319–2327, 2011. doi:{10.1002/jcc.21787}.

[eal6}] R. J. Gowers et al. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. *Proceedings of the 15th Python in Science Conference*, pages 98-105, 2016.

[eal5}] Robert T. McGibbon et al. MDTraj: A Modern Open Library for the Analysis of Molecular Dynamics Trajectories. *BIOPHYSICAL JOURNAL*, 109(8):1528-1532, OCT 20 2015. doi:{10.1016/j.bpj.2015.08.015}.

[Kab76] W. Kabsch. Solution for best rotatation to relate 2 sets of vectors. *Acta Crystallograpica Section A*, 32(SEP1):922-923, 1976. doi:10.1107/S0567739476001873.

[WKS3}] Wolfgang Kabsch and Christian Sander. Dictionary of protein secondary structure: Pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers*, 22(12):2577-2637, 1983. doi:{10.1002/bip.360221211}.

Python Module Index

i

Index

G

H

I

K

L

M

N

P

R

S

T

W

X

Y