Aqueduct Documentation

Release 0.2.11

Tomasz Magdziarz

Aug 10, 2016

CONTENTS

Contents:

CHAPTER

ONE

AQUEDUCT INSTALLATION GUIDE

1.1 Overview

This package comprises of two elements:

1. aqueduct,
2. valve.

Aqueduct is a Python module. It is a collection of tools to trace residues in MD simulation. Valve is a driver Python script. It uses aqueduct to perform such a tracing.

1.2 Install

1.2.1 Aqueduct

Installation was tested on limited number of POSIX-like systems.

In some specific cases installation is very simple:

1. Download `aqueduct.tar.gz` bundle file..
2. Unpack aqueduct bundle file.
3. Go to src directory.
4. Type:

```
python setup.py install
```

Aqueduct requires several Python modules to work and in particular it requires MDAnalysis with AMBER support. This, on the other hand, requires netCDF4. Installation of this combination is sometimes cumbersome. General procedure is following:

1. Install libnetcdf4 and libhdf5 development libraries.
2. Install netCDF4:

```
pip install netCDF4
```

3. Try to install aqueduct.

If, by chance, you are on Ubuntu 14.04 you can try helper script `ubuntu_mdanalysis_install_helper.sh`.

1.2.2 Valve

Valve does not need installation *per se*. Once aqueduct is installed, valve can be run by a following command:

```
python valve.py --help
```

Valve script, ie valve.py, is located in apps directory.

1.2.3 Extras

Access to some visualization capabilities of Aqueduct requires additional Python modules:

1. matplotlib,
2. pymol.

These are usually easy to install.

VALVE MANUAL

Valve application is a driver that uses *aqueduct* module to perform analysis of trajectories of selected residues in MD simulation.

2.1 *Valve* invocation

Once *aqueduct* module is installed (see *Aqueduct installation guide*) properly on the machine *Valve* is available as `valve.py` command line tool.

2.1.1 Usage

Basic help of *Valve* usage can be displayed by following command:

```
valve.py --help
```

It should display following information:

```
usage: valve.py [-h] [--dump-template-config] [-t THREADS] [-c CONFIG_FILE]
                [--max-frame MAX_FRAME]

Valve, Aqueduct driver

optional arguments:
  -h, --help            show this help message and exit
  --dump-template-config
                        Dumps template config file. Suppress all other output
                        or actions. (default: False)
  -t THREADS            Limit Aqueduct calculations to given number of
                        threads. (default: None)
  -c CONFIG_FILE        Config file filename. (default: None)
  --max-frame MAX_FRAME
                        Limit number of frames. (default: None)
  --version             Prints versions and exits.. (default: False)
```

2.1.2 Configuration file template

Configuration file used by *Valve* is of moderate length and complexity. It can be easily prepared with a template file that can be printed by *Valve*. Use following command to print configuration file template on the screen:

```
valve.py --dump-template-config
```

Configuration file template can also be easily saved in to a file with:

```
valve.py --dump-template-config > config.txt
```

Where config.txt is a configuration file template.

For detailed description of configuration file and available options see *Configuration file options*

2.1.3 *Valve* calculation run

Once configuration file is ready *Valve* calculations can be run with a following simple command:

```
valve.py -c config.txt
```

Some of *Valve* calculations can be run in parallel. By default all available CPU cores are used. This is not always desired - limitation of used CPU cores can be done with `-t` option which limits number of concurrent threads used by *Valve*. If it equals 1 no parallelism is used.

Note: Specifying number of threads greater then available CPU cores is generally not optimal.

However, in order to maximize usage of available CPU power it is recommended to set it as number of cores + 1. The reason is that *Valve* uses one thread for the main process and the excess over one for processes for parallel calculations. When parallel calculations are executed the main threads waits for results.

Note: Option `--max-frame` can be used for testing or debugging purposes. It allows to limit number of frames processed by *Valve*. If it set, for example, to `1000` only first 1000 frames will be processed making all calculations very fast.

2.2 How does *Valve* work

Application starts with parsing input options. If `--help` or `--dump-template-config` options are provided appropriate messages are printed on the screen and *Valve* quits with signal `0`.

Note: In current version *Valve* does not check the validity of the config file.

If config file is provided *Valve* parse it quickly and regular calculations starts according to its content. Calculations performed by *Valve* are done in several stages described in the next sections.

2.2.1 Traceable residues

The first stage finds all residues that should be traced and appends them to the list of *traceable residues*. It is done in a loop over all frames. In each frame residues of interest are searched and appended to the list but only if they are not already present on the list.

The search of the residues is done according to user provided definitions.. Two requirements have to be met to append residue to the list:

1. The residue have to be found according to the *Object* definition.
2. The residues have to be within the *Scope* of interest.

The *Object* definition encompasses usually the active site of the protein. The *Scope* of interest defines, on the other hand, the boundaries in which residues are traced and is usually defined as protein.

Since *`aqueduct`* in its current version uses MDAnalysis Python module for reading, parsing and searching of MD trajectory data, definitions of *Object* and *Scope* have to be given as its *Selection Commands*.

Object definition

Object definition have to comprise of two elements:

1. It have to define residues to trace.
2. It have to define spatial boundaries of the *Object* site.

For example, proper Object definition could be following:

```
(resname WAT) and (sphzone 6.0 (resnum 99 or resnum 147))
```

It defines `WAT` as residues that should be traced and defines spatial constrains of the *Object* site as spherical zone within 6 Angstroms of the center of masses of residues with number 99 and 147.

Scope definition

Scope can be defined in two ways: as *Object* but with broader boundaries or as the convex hull of selected molecular object.

In the first case definition is very similar to *Object* and it have to follow the same limitations. For example, proper *Scope* definition could be following:

```
resname WAT around 2.0 protein
```

It consequently have to define `WAT` as residues of interest and defines spatial constrains as all `WAT` residues that are within 2 Angstroms of the protein.

If the *Scope* is defined as the convex hull of selected molecular object (which is recommended), the definition itself have to comprise of this molecular object only, for example `protein`. In that case the scope is interpreted as the interior of the convex hull of atoms from the definition. Therefore, *traceable residues* would be in the scope only if they are within the convex hull of atoms of `protein`.

2.2.2 Raw paths

The second stage of calculations uses the list of all traceable residues from the first stage and finds coordinates of center of masses for each residue in each frame. As in the first stage, it is done in a loop over all frames. For each residue in each frame *Valve* calculates or checks two things:

1. Is the residue in the *Scope* (this is always calculated according to the Scope definition).
2. Is the residue in the *Object*. This information is calculated in the first stage and can be reused in the second. However, it is also possible to recalculate this data according to the new *Object* definition.

For each of the *traceable residues* a special *Path* object is created. If the residue is in the *Scope* its center of mass is added to the appropriate *Path* object together with the information if it is in the *Object* or not.

2.2.3 Separate paths

The third stage uses collection of *Path* objects to create *Separate Path* objects. Each *Path* comprise data for one residue. It may happen that the residue enters and leaves the *Scope* and the *Object* many times over the entire MD. Each such an event is considered by *Valve* as a separate path.

Each *separate path* comprises of three parts:

1. *Incoming* - Defined as a path that leads from the point in which residue enters the *Scope* and enters the object for the firs time.
2. *Object* - Defined as a path that leads from the point in which residue enters the *Object* for the first time and leaves it for the last time.
3. *Outgoing* - Defined as a path that leads from the point in which residue leaves the *Object* for the last lime and leaves the *Scope*.

2.2.4 Clusterization of inlets

Each of the separate paths has beginning and end. If either of them are at the boundaries of the *Scope* they are considered as *Inlets*, i.e. points that mark where the *traceable residues* enters or leaves the *Scope*. Clusters of inlets, on the other hand, mark endings of tunnels or ways in the system which was simulated in the MD.

Clusterization of inlets is performed in following steps:

1. Initial clusterization. Depending on the method, some of the inlets might not be arranged to any cluster and are considered as outliers.
2. [Optional] Outliers detection. Arrangement of inlets to clusters is sometimes far from optimal. In this step, *inlets* that do not fit to cluster are detected and annotated as outliers. This step can be executed in two modes:
   (a) Automatic mode. Inlet is considered to be an outlier if its distance from the centroid is greater then mean distance + 4 * standard deviation of all distances within the cluster.
   (b) Defined threshold. Inlet is considered to be an outlier if its minimal distance from any other point in the cluster is greater then the threshold.
3. [Optional] Reclusterization of outliers. It may happen that the outliers form actually clusters but it was not recognized in initial clusterization. In this step clusterization is executed for outliers only and found clusters are appended to the clusters identified in the first step. Rest of the inlets are marked as outliers.

2.2.5 Analysis

Fifth stage of *Valve* calculations analyses results calculated in stages 1 to 4. Results of the analysis is displayed on the screen or can be save to text file and comprise of following parts:

- Tile and data stamp.
- [Optional] Dump of configuration options.
- **Basic information on traceable residues and separate paths.**
    - Number of traceable residues.
    - Number of separate paths.
- **Basic information on inlets.**
    - Number of inlets.
    - Number of clusters.
    - Are outliers detected.
- **Summary of inlets clusters. Table with 5 columns:**
    1. **Nr**: Row number, starting from 0.
    2. **Cluster**: ID of the cluster. Outliers have 0.
    3. **Size**: Size of the cluster.
    4. **INCOMING**: Number of inlets corresponding to separate paths that enter the scope.
    5. **OUTGOING**: Number of inlets corresponding to separate paths that leave the scope.
- **Summary of separate paths clusters types. Table with 9 columns.**
    1. **Nr**: Row number, starting from 0.
    2. **CType**: Separate path Cluster Type.
    3. **Size**: Number of separate paths belonging to Cluster type.
    4. **Inp**: Average length of incoming part of the path. If no incoming part is available it is nan.
    5. **InpStd**: Standard deviation of length Inp.

6. **Obj**: Average length of object part of the path. If no incoming part is available it is nan.
7. **ObjStd**: Standard deviation of length Inp.
8. **Out**: Average length of outgoing part of the path. If no incoming part is available it is nan.
9. **OutStd**: Standard deviation of length Inp.

- **List of separate paths and their properties. Table with 17 columns.**
    1. **Nr**: - Row number, starting from 0.
    2. **ID**: - Separate path ID.
    3. **BeginF**: Number of frame in which the path begins.
    4. **InpF**: Number of frame in which path begins Incoming part.
    5. **ObjF**: Number of frame in which path begins Object part.
    6. **OutF**: Number of frame in which path begins Outgoing part.
    7. **EndF**: Number of frame in which the path ends.
    8. **InpL**: Length of Incoming part. If no incoming part nan is given.
    9. **ObjL**: Length of Object part.
    10. **OutL**: Length of Outgoing part. If no outgoing part nan is given.
    11. **InpS**: Average step of Incoming part. If no incoming part nan is given.
    12. **InpStdS**: Standard deviation of InpS.
    13. **ObjS**: Average step of Object part.
    14. **ObjStdS**: Standard deviation of ObjS.
    15. **OutS**: Average step of Outgoing part. If no outgoing part nan is given.
    16. **OutStdS**: Standard deviation of OutS.
    17. **CType**: Cluster type of separate path.

Separate path ID

Separate Paths IDs are composed of two numbers separated by colon. First number is the residue number. Second number is consecutive number of the separate path made by the residue. Numeration starts with 0.

Cluster Type of separate path

Each separate paths has two ends: beginning and end. Both of them either belong to one of the inlets clusters, or are among outliers, or are inside the scope. If an end belongs to one of the clusters (including outliers) it has ID of the cluster. If it is inside the scope it has special ID of `N`. Cluster type is an ID composed of IDs of both ends of separate path separated by colon charter.

2.2.6 Visualization

Sixth stage of *Valve* calculations visualizes results calculated in stages 1 to 4. Visualization is done with PyMOL. *Valve* automatically starts PyMOL and loads visualizations in to it. Molecule is loaded as PDB file. Other objects like Inlets clusters or paths are loaded as CGO objects.

Following is a list of objects created in PyMOL (all of them are optional). PyMOL object names given in **bold** text or short explanation is given.

- Selected frame of the simulated system. Object name: *molecule*.

- Inlets clusters, each cluster is a separate object. Object name: **cluster_** followed by cluster annotation: otliers are annotated as Out; regular clusters by ID.
- List of cluster types, raw paths. Each cluster type is a separate object. Object name composed of cluster type (colon replaced by underline) plus **_raw**.
- List of cluster types, smooth paths. Each cluster type is a separate object. Object name composed of cluster type (colon replaced by underline) plus **_smooth**.
- All raw paths. They can be displayed as one object or separated in to Incoming, Object and Outgoing part. Object name: **all_raw**, or **all_raw_in**, **all_raw_obj**, and **all_raw_out**.
- All raw paths inlets arrows. Object name: **all_raw_paths_io**.
- All smooth paths. They can be displayed as one object or separated in to Incoming, Object and Outgoing part. Object name: **all_smooth**, or **all_smooth_in**, **all_smooth_obj**, and **all_smooth_out**.
- All raw paths inlets arrows. Object name: **all_raw_paths_io**.
- Raw paths displayed as separate objects or as one object with several states. Object name: **raw_paths_** plus number of path or **raw_paths** if displayed as one object.
- Smooth paths displayed as separate objects or as one object with several states. Object name: **smooth_paths_** plus number of path or **smooth_paths** if displayed as one object.
- Raw paths arrows displayed as separate objects or as one object with several states. Object name: **raw_paths_io_** plus number of path or **raw_paths_io** if displayed as one object.
- Smooth paths arrows displayed as separate objects or as one object with several states. Object name: **smooth_paths_io_** plus number of path or **smooth_paths_io** if displayed as one object.

Color schemes

Inlets clusters are colored automatically. Outliers are gray.

Incoming parts of paths are red, Outgoing parts are blue. Object parts in case of smooth paths are green and in case of raw paths are green if residue is precisely in the object area or yellow if is leaved object area but it is not in the Outgoing part yet.

Arrows are colored in accordance to paths colors.

CHAPTER

THREE

CONFIGURATION FILE OPTIONS

Valve Configuration file is a simple and plain text file. It is similar to INI files commonly used in one of the popular operating systems and is compliant with Python module `ConfigParser`.

Configuration file comprises of several *sections*. They can be grouped in to three categories. Names of sections given in **bold** text.

1. **Global settings:**
    - **global**
2. **Stages options:**
    - **traceable_residues**
    - **raw_paths**
    - **separate_paths**
    - **inlets_clusterization**
    - **analysis**
    - **visualize**
3. **Methods options:**
    - **smooth**
    - **clusterization**
    - **reclusteriation**

3.1 Section global

This section allows settings of trajectory data and progress bar type.

3.1.1 Available options

- `top` - Path to topology file.
- `trj` - Path to trajectory file.

3.1.2 Example

```
[global]
top = path/to/topology/file.prmtop
trj = path/to/trajectory/file.nc
```

3.2 Common settings of stage sections

3.2.1 Option execute

All stage sections have `execute` option which decides if the stage is executed or skipped. There are two possible values of `execute` option: `run`, and `skip`.

If `execute` is set to `run` the stage is executed and results of calculations can be optionally saved.

If `execute` is set to `skip` execution of the stage is skipped. Results of calculations saved previously can be optionally loaded.

3.2.2 Option save

This options allows to save a dump of calculated data on the disk. If `execute` is set to `run` and `save` is set to file name results are saved as gziped pickled dump.

In case of **analysis** and **visualize** sections this setting has slightly different function. Results of **analysis** section as saved to the file pointed by **save** as plain text comprising of tables and summaries. Stage **visualize**, on the other hand, uses **save** option to save PyMOL session.

3.2.3 Option load

This options allows to read previously saved dump of results. It is used only if `ececute` is set to `skip` and is valid only for **traceable_residues**, **raw_paths**, **separate_paths**, and **inlets_clusterization** stages.

3.3 Stage traceable_residues

3.3.1 Option scope

Definition of *Scope* of interest. See also *Scope definition*.

Note: This option is mandatory.

3.3.2 Option scope_convexhull

Flag to set if the *Scope* is direct or convex hull definition.

3.3.3 Option object

Definition of *Object* of interest. See also *Object definition*.

Note: This option is mandatory.

3.4 Stage raw_paths

This stage also requires definition of the *Scope* and *Object*. If appropriate settings are not given, settings from the previous stage are used.

3.4.1 Option clear_in_object_info

If it is set to `True` information on occupation of *Object* site by traceable residues calculated in the previous stage are cleared and have to be recalculated. This is useful if definition of *Object* is changed.

3.5 Stage separate_paths

3.5.1 Option discard_empty_paths

If set to `True` empty paths are discarded.

3.5.2 Option sort_by_id

If set to `True` separate paths are sorted by ID.

3.5.3 Option apply_smoothing

If set to `True` smooth paths are precalculated according to **smooth** setting. This speed up access to smooth paths in later stages but makes dump data much bigger.

3.5.4 Option apply_soft_smoothing

If set to `True` raw paths are replaced by smooth paths calculated according to **smooth** section.

3.5.5 Option discard_short_paths

This option allows to discard paths that are shorter then the threshold.

3.6 Stage inlets_clusterization

3.6.1 Option recluster_outliers

If set to `True` reclusterization of outliers is executed according to the method defined in **reclusterization** section.

3.6.2 Option detect_outliers

If set detection of outliers is executed. See *Clusterization of inlets* for more details.

3.6.3 Option singletons_outliers

Maximal size of cluster to be considered as outliers. If set to number > 0 clusters of that size are removed and their objects are moved to outliers. See *Clusterization of inlets* for more details.

3.7 Stage analysis

3.7.1 Option dump_config

If set to `True` configuration options, as seen by Valve, are added to the head of results.

3.8 Stage visualize

3.8.1 Option simply_smooths

If set to float number simplification of smooth paths is applied. Simplification removes points which do not (or almost do not) change the shape of smooth path. For more details see *Recursive Vector Linearization*.

3.8.2 Option all_paths_raw

If True produces one object in PyMOL that holds all paths visualized by raw coordinates.

3.8.3 Option all_paths_smooth

If True produces one object in PyMOL that holds all paths visualized by smooth coordinates.

3.8.4 Option all_paths_split

If is set True objects produced by **all_paths_raw** and **all_paths_smooth** are split into Incoming, Object, and Outgoing parts and visualized as three different objects.

3.8.5 Options all_paths_raw_io and all_paths_smooth_io

If set True arrows pointing beginning and end of paths are displayed oriented accordingly to raw or smooth paths.

3.8.6 Option paths_raw

If set True raw paths are displayed as separate objects or as one object with states corresponding to number of path.

3.8.7 Option paths_raw

If set True smooth paths are displayed as separate objects or as one object with states corresponding to number of path.

3.8.8 Option paths_raw_io

If set True arrows indicating beginning and and of paths, oriented accordingly to raw paths, are displayed as separate objects or as one object with states corresponding to number of paths.

3.8.9 Option paths_smooth_io

If set True arrows indicating beginning and and of paths, oriented accordingly to smooth paths, are displayed as separate objects or as one object with states corresponding to number of paths.

3.8.10 Option paths_states

If True objects displayed by **paths_raw**, **paths_smooth**, **paths_raw_io**, and **paths_smooth_io** are displayed as one object with with states corresponding to number of paths. Otherwise they are displayed as separate objects.

3.8.11 Option ctypes_raw

Displays raw paths in a similar manner as non split **all_paths_raw** but each cluster type is displayed in separate object.

3.8.12 Option ctypes_smooth

Displays smooth paths in a similar manner as non split **all_paths_smooth** but each cluster type is displayed in separate object.

3.8.13 Option show_molecule

If is set to selection of some molecular object in the system, for example to `protein`, this object is displayed.

Note: Possibly due to limitations of `MDAnalysis` only whole molecules can be displayed. If **show_molecule** is set to `backbone` complete protein will be displayed any way. This may change in future version of `MDAnalysis` and or *`aqueduct`*.

3.8.14 Option show_molecule_frames

Allows to indicate which frames of object defined by **show_molecule** should be displayed. It is possible to set several frames. In that case frames would be displayed as states.

Note: If several frames are selected they are displayed as states which may interfere with other PyMOL objects displayed with several states.

Note: If several states are displayed protein tertiary structure data might be lost. This seems to be limitation of either `MDAnalysis` or PyMOL.

CHAPTER

FOUR

VALVE TUTORIAL

This is a tentative *Valve* manual. Created for the sake of Aqueduct training we have today. Eventually, it will be rewritten to the official version.

This tutorial assumes `aqueduct` and *Valve* is already installed - see *Aqueduct installation guide*. It is also assumed that user is acquainted with *Valve manual* and *Valve* *Configuration file options*.

4.1 *Valve* invocation

Usually *Valve* is run by:

```
valve.py
```

Due to specific setup in our laboratory *Valve* has to be run through simple wrapper script:

```
valve_run
```

Additionally, to speed up all calculations it is assumed that *Valve* is run with `--max-frame 1000` option:

```
valve_run --max-frame 1000
```

To check is *Valve* is installed and works properly try to issue following commands:

```
valve_run --help
valve_run --version
```

4.2 Test data

Mouse!

We will use 10ns Amber MD simulation data of sEH protein (PDBID **1cqz**). Necessary files can be downloaded here:

- Go to download server.
- Go to `1cqz` directory.
- Download all files and save them in sane location on your machine. Please note, that `.nc` file is ca. 3.5 GB so it may take a while to download it.

4.3 Inspect your system

Before we start any calculations lets have a look at the protein of interest. Start *PyMOL* and get `1cqz` PDB structure (for example by typing in *PyMOL* command prompt `fetch 1cqa`).

To setup *Valve* calculations we need to know active site of the protein. More precisely we need to know IDs or residues that are in the active site. This would allow us to create *Object definition*.

But wait. Is it really the correct structure? How many chains there are? What is the numeration of residues?

4.3.1 Create *Object definition*

Lets load another structure. Open file `first_frame_1cqz.pdb` downloaded from test data repository. It is a first frame of the MD simulation and it is en example of how the frame of MD looks like. In order to create *Object definition* you have to discover following things:

1. What is the name of water residue?
2. What are numbers of residues in the active site?
3. What size the active site is?

Note: It is also good idea to open `.pdb` file in your favorite text editor and look at residue numbers and names.

4.3.2 Create *Scope definition*

Scope definition is easy to create. We will use *Convex hull* version so the scope definition could be simply `backbone`.

4.4 Prepare config file

Valve performs calculations according to the configuration (aka *config*) file.

Lets start from dumping config file template to `config.txt` file. Open it in your favorite editor and fill all options. If you have troubles look at *Configuration file options* (and *Valve manual*).

Things to remember:

1. Provide correct paths to topology and trajectory data.
2. Enter correct *Object* and *Scope* definitions.
3. Provide file name of result in analysis section, for example `results.txt` (for future reference).
4. Make sure visualization is switched on and `save` option points to session file name (`.pse`)

4.5 Run *Valve*

Make sure all necessary data is in place. Open terminal, go to your working directory and type in:

```
valve_run --max-frame 1000 -c config.txt
```

Depending on your machine and current load it may take a while (matter of minutes) to complete all calculations.

4.5.1 Visual inspection

In the last stage *PyMOL* should pop up and *Valve* should start to feed it with visualization data. This would take a moment and if you set up `save` option a *PyMOL* session would be saved. Once it is done *Valve* quits and switches off *PyMOL*. Now, you can restart it and read saved session.

4.5.2 Analysis tables

Open `results.txt` file and look at summaries and tables. See also *Valve manual*.

4.6 Feedback

Give us your opinion. Send your questions, inquires, anything to developer(s): Tomasz Magdziarz. This are couple of questions that might be useful to form your opinion.

1. What do you like in *Valve* and *Aqueduct*?
2. What do you do not like in *Valve* or *Aqueduct*?
3. What is missing?
4. Do you find it useful?

CHAPTER

FIVE

AQUEDUCT

5.1 aqueduct package

5.1.1 Subpackages

aqueduct.geom package

Submodules

aqueduct.geom.cluster module This module provides functions for clusterization. Clusterization is done by `scikit-learn` module.

`MeanShiftBandwidth` (*X*, ***kwargs*)

class **`PerformClustering`** (*method*, ***kwargs*)

Bases: `object`

`__init__` (*method*, ***kwargs*)

`__call__` (*coords*)

`_get_noclusters` (*n*)

`fit` (*coords*)

`centers` ()

aqueduct.geom.convexhull module
`_vertices_ids` (*convexhull*)
`_vertices_points` (*convexhull*)

`_point_within_convexhull` (*convexhull*, *point*)

`_facets` (*convexhull*)

`is_point_within_convexhull` (*point_chull*)

aqueduct.geom.master module
`fit_trace_to_points` (*trace*, *points*)
`decide_on_type` (*cont*, *s2o_treshold=0.5*)

`simple_types_distribution` (*types*)

`get_weights_` (*spaths*, *smooth=None*)

`get_mean_coord_` (*coords*, *l*)

`concatenate` (**args*)

`create_master_spath` (*spaths*, *smooth=None*, *resid=0*, *ctype=None*, *bias_long=5*, *heartbeat=None*)

class **MasterTrace**
Bases: object

__init__()

aqueduct.geom.pca module

class **Center**(*X*)
Bases: object

__init__(*X*)

__call__(*X*)

undo(*X*)

class **Normalize**(*X*)
Bases: object

__init__(*X*)

__call__(*X*)

undo(*X*)

class **Standartize**(*X*)
Bases: *aqueduct.geom.pca.Center*, *aqueduct.geom.pca.Normalize*

__init__(*X*)

__call__(*X*)

undo(*X*)

class **PCA**(*X*, *prepro=None*)
Bases: object

__init__(*X*, *prepro=None*)

P

preprocess(*X*)

preprocess_undo(*X*)

__call__(*X*)

undo(*T*)

aqueduct.geom.smooth module Created on Dec 15, 2015

@author: tljm

class **Smooth**(*recursive=None*, ***kwargs*)
Bases: object

__init__(*recursive=None*, ***kwargs*)

smooth(*coords*)

__call__(*coords*)

class **GeneralWindow**

static **max_window_at_pos**(*pos*, *size*)

check_bounds_at_max_window_at_pos(*lb*, *ub*, *pos*, *size*)

class **WindowSmooth**(*window=5*, *function=<function mean>*, ***kwargs*)
Bases: *aqueduct.geom.smooth.Smooth*, *aqueduct.geom.smooth.GeneralWindow*

__init__(*window=5*, *function=<function mean>*, ***kwargs*)

smooth (**args, **kwargs*)

class **DistanceWindowSmooth** (*window=5, function=<function mean>, **kwargs*)
Bases: `aqueduct.geom.smooth.Smooth`, `aqueduct.geom.smooth.GeneralWindow`

__init__ (*window=5, function=<function mean>, **kwargs*)

smooth (**args, **kwargs*)

class **ActiveWindowSmooth** (*window=5, function=<function mean>, **kwargs*)
Bases: `aqueduct.geom.smooth.Smooth`, `aqueduct.geom.smooth.GeneralWindow`

__init__ (*window=5, function=<function mean>, **kwargs*)

smooth (**args, **kwargs*)

class **MaxStepSmooth** (*step=1.0, **kwargs*)
Bases: `aqueduct.geom.smooth.Smooth`

__init__ (*step=1.0, **kwargs*)

smooth (**args, **kwargs*)

class **WindowOverMaxStepSmooth** (***kwargs*)
Bases: `aqueduct.geom.smooth.Smooth`

__init__ (***kwargs*)

smooth (*coords*)

class **ActiveWindowOverMaxStepSmooth** (***kwargs*)
Bases: `aqueduct.geom.smooth.Smooth`

__init__ (***kwargs*)

smooth (*coords*)

class **DistanceWindowOverMaxStepSmooth** (***kwargs*)
Bases: `aqueduct.geom.smooth.Smooth`

__init__ (***kwargs*)

smooth (*coords*)

aqueduct.geom.test_triangle_height module

class **TestTriangle_height** (*methodName='runTest'*)
Bases: `unittest.case.TestCase`

test_OutType ()

test_2dim ()

test_3dim ()

aqueduct.geom.test_vectors_angle_anorm module

class **TestVectors_angle_anorm** (*methodName='runTest'*)
Bases: `unittest.case.TestCase`

test_output_value ()

test_vectors_angle_anorm ()

test_list_input ()

test_zero ()

test_zero2 ()

test_huge_values ()

test_mixed_values ()

test_negative_vector()

test_one_negative_value()

aqueduct.geom.traces module

vector_norm(*V*)

triangle_angles(*A, B, C*)

triangle_angles_last(*A, B, C*)

triangle_height(*A, B, C*)

vectors_angle(*A, B*)

vectors_angle_alt(*A, B*)

vectors_angle_alt_anorm(*A, B, A_norm*)

vectors_angle_anorm(*A, B, A_norm*)

class **LinearizeOneWay**

Bases: `object`

here(*coords*)

class **LinearizeHobbit**

Bases: *aqueduct.geom.traces.LinearizeOneWay*

and_back_again(*coords*)

__call__(*coords*)

class **LinearizeRecursive**

Bases: `object`

Base class for linearization methods classes.

It implements recursive algorithm.

here(*coords, depth=0*)

Core of recursive linearization argorithm.

It checks if the first, the last and the middle point are linear acoording to the criterion. The middle point is selected a point that is in the middle of lenght of the paths made by input coordinates.

If these points are linear their indices are returned. Otherwise, coordinates are split into two parts. First part spans points from the first point to the middle point (inclusive) and the second parth spans points from the middle (inclusive) to the last point. Next, these two parts are submitted recursively to *here()*.

Results of these recursive calls are joined, redundant indecies are removed and sorted reslult is returned.

Parameters

- **coords** (*numpy.ndarray*) – Input coordinates.
- **depth** (*int*) – Depth of recurence.

Returns Indices of **:arg:‘coords‘** points that can be used instead of all points in visulatization.

Return type list of int

__call__(*coords*)

class **TrianlgeLinearize**(*treshold*)

Bases: `object`

__init__(*treshold*)

is_linear (*coords*, ***kwargs*)

class **VectorLinearize** (*treshold*)

Bases: `object`

Base class for linearization methods classes.

It implements vectro linearization criterion.

__init__ (*treshold*)

is_linear_core (*coords*, *depth=None*)

Method checks if input coordinates are linear acoording to the threshold and depth.

It begins with calculation of the threshold. If *depth* is None it is set to 1. Current treshold is calculated with following simple equation:

$$treshold_{current} = treshold_{initial} * (2 - 0.9^{depth})$$

Next, in a loop over all points but the first and the last the angle is calculated between two vectors. The first one made by the point and the firs point, and the second vector made by the last and the first point. If any of the calculated angles is bigger the the treshold methods returns False; otherwise method returns True.

Parameters

- **coords** (*numpy.ndarray*) – Coodrdinates for which linearizetion criterion is checked.
- **depth** (*int*) – Depth of recurence.

Returns True if input coordinates are linear and False otherwise.

Return type bool

is_linear (*coords*, *depth=None*, ***kwargs*)

For more detail see `is_linear_core()` which is used as the criterion of linearity in this method.

Parameters

- **coords** (*numpy.ndarray*) – Coodrdinates for which linearizetion criterion is checked.
- **depth** (*int*) – Depth of recurence.

Returns True if input coordinates are linear and False otherwise. Criterion is checked for coordinates in normal and reverse order.

Return type bool

class **LinearizeRecursiveVector** (*treshold*)

Bases: `aqueduct.geom.traces.LinearizeRecursive`, `aqueduct.geom.traces.VectorLinearize`
Class provides recursive linearization of coordinates with `LinearizeRecursive` algorithm and the criterion of linearity implemented by `VectorLinearize`.

diff (*trace*)

tracepoints (*start*, *stop*, *nr*)

midpoints (*paths*)

length_step_std (*trace*)

derrivative (*values*)

Module contents

aqueduct.traj package

Submodules

aqueduct.traj.dumps module

class TmpDumpWriterOfMDA

Bases: `object`

`__init__`()

`dump_frames`(*reader*, *frames*, *selection='protein'*)

`close`()

`__del__`()

aqueduct.traj.inlets module

class ProtoInletTypeCodes

`surface` = 'surface'

`internal` = 'internal'

`incoming` = 'inin'

`outgoing` = 'inout'

class InletTypeCodes

Bases: *`aqueduct.traj.inlets.ProtoInletTypeCodes`*

`all_surface` = [('surface', 'inin'), ('surface', 'inout')]

`all_internal` = [('internal', 'inin'), ('internal', 'inout')]

`all_incoming` = [('surface', 'inin'), ('internal', 'inin')]

`all_outgoing` = [('surface', 'inout'), ('internal', 'inout')]

`surface_incoming` = ('surface', 'inin')

`internal_incoming` = ('internal', 'inin')

`internal_outgoing` = ('internal', 'inout')

`surface_outgoing` = ('surface', 'inout')

`itype` = 'internal'

class InletClusterGenericType(*inp*, *out*)

Bases: `object`

`__init__`(*inp*, *out*)

`input`

`output`

`cluster2str`(*cl*)

`__getitem__`(*item*)

`__len__`()

`__str__`()

`__repr__`()

`make_val`(*base*)

`__cmp__`(*other*)

`__hash__`()

class `InletClusterExtendedType`(*surfin*, *interin*, *interout*, *surfout*)
Bases: `aqueduct.traj.inlets.InletClusterGenericType`

`__init__`(*surfin*, *interin*, *interout*, *surfout*)

`generic`

class `Inlet`(*coords*, *type*, *reference*)
Bases: `tuple`

`__getnewargs__`()
Return self as a plain tuple. Used by copy and pickle.

`__getstate__`()
Exclude the OrderedDict from pickling

static `__new__`(*_cls*, *coords*, *type*, *reference*)
Create new instance of Inlet(coords, type, reference)

`__repr__`()
Return a nicely formatted representation string

`__slots__` = ()

`_asdict`()
Return a new OrderedDict which maps field names to their values

`_fields` = ('coords', 'type', 'reference')

classmethod `_make`(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new Inlet object from a sequence or iterable

`_replace`(*_self*, ***kwds*)
Return a new Inlet object replacing specified fields with new values

`coords`
Alias for field number 0

`reference`
Alias for field number 2

`type`
Alias for field number 1

class `Inlets`(*spaths, onlytype=[('surface', 'inin'), ('surface', 'inout')]*)
Bases: `object`

`__init__`(*spaths, onlytype=[('surface', 'inin'), ('surface', 'inout')]*)

`extend_inlets`(*spath*, *onlytype=None*)

`add_cluster_annotations`(*clusters*)

`size`

`coords`

`types`

`refs`

`perform_clustering`(*method*)

`recluster_outliers`(*method*)

`small_clusters_to_outliers`(*maxsize*)

`renumber_clusters`()

sort_clusters ()

clusters_list

clusters_centers

clusters_size

clusters_std

spaths2ctypes (**args*, ***kwargs*)

lim_to (*what*, *towhat*)

lim2spaths (*spaths*)

lim2types (*types*)

lim2clusters (*clusters*)

limspaths2 (**args*, ***kwargs*)

aqueduct.traj.paths module Created on Dec 10, 2015

@author: tljm

union (*a*, *b*)

glue (*a*, *b*)

xor (**args*, ***kwargs*)

left (*a*, *b*)

right (*a*, *b*)

class **PathTypesCodes**

path_in_code = **'i'**

path_object_code = **'c'**

path_out_code = **'o'**

class **GenericPathTypeCodes**

object_name = **'c'**

scope_name = **'s'**

out_name = **'n'**

class **GenericPaths** (*id*, *min_pf=None*, *max_pf=None*)

Bases: `object`, `aqueduct.traj.paths.GenericPathTypeCodes`

__init__ (*id*, *min_pf=None*, *max_pf=None*)

add_coord (*coord*)

add_object (*frame*)

add_scope (*frame*)

add_type (*frame*, *ftype*)

max_frame

min_frame

get_paths_in ()

get_paths_out ()

get_paths_for_frames_range (**args*, ***kwargs*)

find_paths (*fullonly=False*)

find_paths_coords (*fullonly=False*)

find_paths_types (*fullonly=False*)

find_paths_coords_types (*fullonly=False*)

get_single_path_coords (*spath*)

get_single_path_types (*spath*)

class SinglePathID (*id=None*, *nr=None*)

Bases: `object`

__init__ (*id=None*, *nr=None*)

__str__ ()

yield_single_paths (*gps*, *fullonly=False*, *progress=False*)

class SinglePath (*id*, *paths*, *coords*, *types*)

Bases: `object`, `aqueduct.traj.paths.PathTypesCodes`, `aqueduct.traj.inlets.InletTypeCodes`

empty_coords = array([], shape=(0, 3), dtype=float64)

__init__ (*id*, *paths*, *coords*, *types*)

coords_first_in

coords_last_out

coords_filo

get_inlets ()

coords

coords_cont

paths

paths_cont

types

types_cont

gtypes

gtypes_cont

etypes

etypes_cont

size

begins

ends

has_in

has_object

has_out

get_coords (**args*, ***kwargs*)

get_coords_cont (*smooth=None*)

_make_smooth_coords (**args*, ***kwargs*)

apply_smoothing (*smooth*)

get_distance_cont (*smooth=None*, *normalize=False*)

get_distance_rev_cont (**args*, ***kwargs*)

get_distance_both_cont (**args*, ***kwargs*)

get_velocity_cont (**args*, ***kwargs*)

get_acceleration_cont (**args*, ***kwargs*)

class **MasterPath** (*sp*)
Bases: `aqueduct.traj.paths.SinglePath`

__init__ (*sp*)

add_width (*width*)

aqueduct.traj.reader module Created on Nov 19, 2015

@author: tljm

class **Reader** (*topology*, *trajectory*)
Bases: `object`

__init__ (*topology*, *trajectory*)

open_trajectory ()

number_of_frames

set_current_frame (*frame*)

next_frame ()

iterate_over_frames ()

parse_selection (*selection*)

select_resnum (*resnum*)

select_multiple_resnum (*resnum_list*)

class **ReadViaMDA** (*topology*, *trajectory*)
Bases: `aqueduct.traj.reader.Reader`

number_of_frames

set_current_frame (*frame*)

next_frame ()

parse_selection (*selection*)

select_resnum (*resnum*)

select_multiple_resnum (*resnum_list*)

__enter__ ()

__exit__ (*typ*, *value*, *traceback*)

open_trajectory ()

class **ReadAmberNetCDFviaMDA** (*topology*, *trajectory*)
Bases: `aqueduct.traj.reader.ReadViaMDA`

open_trajectory ()

class **ReadDCDviaMDA** (*topology*, *trajectory*)
Bases: `aqueduct.traj.reader.ReadViaMDA`

open_trajectory ()

aqueduct.traj.selections module

class `Selection`

Bases: `object`

def __init__(self,selection,selection_string=None):

self.selection_object = selection self.selection_string = selection_string

`center_of_mass``()`

`iterate_over_residues``()`

`unique_resids``()`

`unique_resids_number``()`

`atom_positions``()`

`center_of_mass_of_residues``()`

`get_convexhull_of_atom_positions``()`

`uniquify``()`

`__add__``(`*other*`)`

`first_resid``()`

class `SelectionMDA``(`*atoms*`)`

Bases: `MDAnalysis.core.AtomGroup.AtomGroup`, *`aqueduct.traj.selections.Selection`*

`iterate_over_residues``()`

`unique_resids``(`*ikwid=False*`)`

`atom_positions``()`

`__add__``(`*other*`)`

`uniquify``()`

class `CompactSelectionMDA``(`*sMDA*`)`

Bases: `object`

`__init__``(`*sMDA*`)`

`toSelectionMDA``(`*reader*`)`

Module contents

aqueduct.utils package

Submodules

aqueduct.utils.helpers module Collection of helpers - functions and decorators.

`combine``(`*seqin*`)`

This is an alien function. It is not extensively used.

Directly taken form http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/302478/index_txt

Returns a list of all combinations of argument sequences. For example, following call:

```
combine(((1,2),(3,4)))
```

gives following list of combinations:

```
[[1, 3], [1, 4], [2, 3], [2, 4]]
```

Parameters **sequin** (*tuple*) – Tuple of sequences to combine.

Returns All possible combinations of all input sequences.

Return type list of lists

is_number(*s*)

lind(*l*, *ind*)

Indexes lists using lists of integers as identificators. For example:

```
lind(['a','b','c','d','e'],[1,4,2])
```

returns:

```
['b', 'e', 'c']
```

Parameters

- **l** (*list*) – List to be indexed.
- **ind** (*list*) – Integer indexes.

Returns Reindexed list.

Return type list

class **Auto**

Auto type definition. The class is used as an alternative value for options (if particular option supports it). If options (or variables/parameters etc.) have value of *Auto* it means that an automatic process for parametrization should be performed.

For example, if the input parameter is set to *Auto* it is supposed that its value is calculated on the basis of input data or other parameters.

__repr__()

Returns String `Auto`.

Return type str

__str__()

Calls *__repr__()*.

create_tmpfile(*ext=None*)

Creates temporary file. File is created, closed and its file name is returned.

Note: It is responsibility of the caller to delete the file.

Parameters **ext** (*str*) – Optional extension of the file.

Returns File name of created temporary file.

Return type str

range2int(*r*, *uniq=True*)

Transforms a string range in to a list of integers (with added missing elements from given ranges).

For example, a following string:

```
'0:2 4:5 7 9'
```

is transformed into:

```
[0,1,2,4,5,7,9]
```

Parameters

- **r** (*str*) – String of input range.
- **uniq** (*bool*) – Optional parameter, if set to *True* only unique and sorted integers are returned.

Returns List of integers.

Return type list of int

int2range(*l*)

Transforms a list of integers in to a string of ranges.

For example, a following list:

```
[0,1,2,4,5,7,9]
```

is transformed into:

```
0:2 4:5 7 9
```

Parameters **l** (*list*) – input list of int

Returns String of ranges.

Return type str

is_iterable(*l*)

Checks if provided object is iterable. Returns True is it is iterable, otherwise returns False.

Parameters **l** (*list*) – input object

Returns True if submitted object is iterable otherwise returns False.

Return type bool

Warning: Current implementation cannot be used with generators!

Todo

Current implementation is primitive and HAVE TO be replaced.

sortify(*gen*)

Decorator to convert functions' outputs into a sorted list. If the output is iterable it is converted in to a list of appropriate length. If the output is not iterable it is converted in to a list of length 1.

Written on the basis of `listify()`.

Returns Output of decorated function converted to a sorted list.

Return type list

listify(*gen*)

Decorator to convert functions' outputs into a list. If the output is iterable it is converted in to a list of appropriate length. If the output is not iterable it is converted in to a list of length 1.

This function was copied from:

http://argandgahandapandpa.wordpress.com/2009/03/29/python-generator-to-list-decorator/

and further improved by tljm@wp.pl.

Returns Output of decorated function converted to a list.

Return type list

tupleify(*gen*)
Decorator to convert functions' outputs into a tuple. If the output is iterable it is converted in to a tuple of apropriate length. If the output is not iterable it is converted in to a tuple of length 1.

Written on the basis of `listify()`.

Returns Output of decorated function converted to a tuple.

Return type tuple

arrayify(*gen*)
Decorator to convert functions' outputs into a 2D numpy array. If the output is iterable it is converted in to a 2D numpy array of appropriate shape. If the output is not iterable it is converted in to a 2D numpy array of shape 1x1.

Written on the basis of `listify()`.

Returns Output of decorated function converted to a 2D numpy array.

Return type numpy.ndarray

arrayify1(*gen*)
Decorator to convert functions' outputs into a 1D numpy array. If the output is iterable it is converted in to a 2D numpy array of appropriate shape. If the output is not iterable it is converted in to a 2D numpy array of shape 1x1.

Written on the basis of `listify()`.

Returns Output of decorated function converted to a 1D numpy array.

Return type numpy.ndarray

list_blocks_to_slices(*l*)
Slices list in to block according to its elements identity. Resulting slices correspond to blocks of identical elements.

Parameters **l** (`list`) – List of any objects.

Returns Generator of slices.

Return type generator

what2what(*what*, *towhat*)
This function search if elements of the one list (:attr: 'what') are present in the other list (:attr: 'towhat') and returns indices of elements form :attr:'what' list as a tuple. If elements from the first list are not present in the second list the tuple is empty. :param list what: Input list for which indices of elements present in `towhat` are returned. :param list towhat: List of elements which input list is indexed to. :return: Indices of `what` list that are present in `towhat` list. :rtype: tuple

make_iterable(*something*)
If input object is not iterable returns it as one element list. Otherwise returns the object.

Parameters **something** (`object`) – Input object.

Returns Iterable object.

Return type iterable or list

strech_zip(**args*)

compress_zip(**args*)

zip_zip(**args*, ***kwargs*)

aqueduct.utils.log module Module comprises convieniences functions and definitios for logging purposes including progress bar helpers.

gregorian_year_in_days = 365.2425
Length of Gregorian year in days. Average value. Source: https://en.wikipedia.org/wiki/Year

smart_time_string (*s*, *rl=0*, *t=1.1*, *maximal_length=None*, *maximal_units=5*)

Function transforms time in seconds to nicely formatted string of length defined by `maximal_length`. Depending on number of seconds time is represented with one or more of the following units:

Unit name	Unit abbreviation
seconds	s
minutes	m
hours	h
days	d
years	y

Maximal number of units used in time string can be set with `maximal_units`.

Parameters

- **s** (*int*) – Input time in seconds.
- **rl** (*int*) – Number of units already used for representing time.
- **t** (*float*) – Exces above standard number of current time units.
- **maximal_length** (*int*) – Maximal length of the output string. Must be greater then 0.
- **maximal_units** (*int*) – Maximal number of units used in the output string. Must be greater then 0 and lower then 6.

Returns string of nicely formated time

Return type str

gsep (*sep='-'*, *times=72*, *length=None*)

Generic separator.

Parameters

- **sep** (*str*) – Element(s) of separator.
- **times** (*int*) – Number of times `sep` is printed.
- **length** (*int*) – Optional maximal length of output.

Returns String separator.

Return type str

tsep (*line*)

Parameters **line** (*str*) – Input line.

Returns Returns default `gsep()` of length of `line`.

underline (*line*)

Parameters **line** (*str*) – Input line.

Returns String made by concatenation of `line`, `os.linesep`, and output of `tsep()` called with `line`.

Return type str

thead (*line*)

Parameters **line** (*str*) – Input line.

Returns String made by concatenation of output of `tsep()` called with `line`, `line`, `os.linesep`, and again output of `tsep()` called with `line`.

Return type str

class SimpleProgressBar(*mess=None*, *maxval=None*)

Bases: object

Simple progress bar displaying progress with percent indicator, progress bar and ETA. Progress is measured by iterations.

Variables

- **rotate** (*str*) – String comprising characters with frames of a rotating toy.
- **barlenght** (*int*) – Length of progress bar.
- **maxval** (*int*) – maximal number of iterations
- **current** (*int*) – current number of iterations
- **overrun_notice** (*bool*) – if True, overrun above **:ivar:'maxval'** iterations causes insert of newline
- **overrun** (*bool*) – flag of overrun
- **begin** (*int*) – time in seconds at the initialization of the *SimpleProgressBar* class.
- **tcurrent** (*int*) – time in seconds of current iteration

rotate = '\\|/-'

barlenght = 24

__init__(*mess=None*, *maxval=None*)

Parameters

- **maxval** (*int*) – Maximal number of iterations stored to `maxval`.
- **mess** (*str*) – Optional message displayed at progress bar initialization.

bar()

ETA()

Returns ETA calculated on the basis of current number of iterations `current` and current time `tcurrent`. If number of iterations is 0 returns `?`. Time is formated wiht *smart_time_string()*.

Returns ETA as string.

Return type str

percent()

Returns float number of precent progress calculated in the basis of current number of iterations `current`. Should return number between 0 and 100.

Returns percent progress number

Return type float

show()

Shows current progress.

If value returned by *percent()* is =< 100 then progres is printed as percent indicator leaded by ETA calculated by *ETA()*.

If value returned by *percent()* is > 100 then progress is printed as number of iterations and total time.

Progress bar is writen to standard error.

heartbeat()

update(*step*)

Updates number of current iterations `current` by one if `step` is > 0. Otherwise number of current iterations is not updated. In boths cases time of current iteration `tcurrent` is updated and *show()* is called.

Parameters **step** (*int*) – update step

ttime()

Calculates and returns total time string formated with *smart_time_string()*.

Returns string of total time

Return type str

finish()

Finishes progress bar. First, *update()* is called with `step` = 0. Next message of total time is writen to standard error.

class pbar(*maxval=100*, *kind='simple'*)

Bases: `object`

Progress bar wrapper class. It can use several types of progress bars, including *SimpleProgressBar*. Additionaly, it can handle progress bars with following packages (must be installed separately):

- progressbar
- tqdm
- pyprind

Variables

- **__maxval** (*int*) – maximal number of iterations
- **__kind** (*str*) – type of progress bar
- **__curval** (*int*) – current number of iterations
- **__pbar** – progress bar child object

Warning: *SimpleProgressBar* is the only one kind of progress bar recommended.

__init__(*maxval=100*, *kind='simple'*)

Parameters

- **maxval** (*int*) – maximal number of iterations stored to **:ivar:'__maxval'** and passed child progress bar object
- **kind** (*str*) – type of progress bar, available types: simple, progressbar, tqdm, pyprind

update(*val*)

Updates progress bar with value of `val` parameter. Exact behavior depends on the type of progress bar.

Parameters **val** (*int*) – value used to update progress bar

heartbeat()

finish()

Finishes progress bar. It cals appropriate, if exist, method of child progress bar object. is writen to standard error.

get_str_timestamp()

class fbm(*info*)

Bases: `object`

__init__(*info*)

__enter__()

__exit__(*typ*, *value*, *traceback*)

`__call__`(*info*)

`message`(*mess*, *cont=False*)
Prints message to standard error.

Parameters

- **`mess`** (`str`) – message to print
- **`cont`** (`bool`) – if set True no new line is printed

aqueduct.utils.multip module Created on Feb 3, 2016

@author: tljm

Module contents

aqueduct.visual package

Submodules

aqueduct.visual.cmaps module

aqueduct.visual.helpers module

`cc_safe`(*c*)

`cc`(*c*)

`color_codes`(*code*, *custom_codes=None*)

`get_cmap`(*size*)

class **`ColorMapDistMap`**(*name='hsv'*, *size=None*)
Bases: `object`

`default_cm_size` = 256

`grey` = (0.5, 0.5, 0.5, 1)

`__init__`(*name='hsv'*, *size=None*)

`__call__`(*node*)

`f_like`(*n*)

aqueduct.visual.pymol_cgo module

aqueduct.visual.pymol_connector module

class **`BasicPymolCGO`**
Bases: `object`

`cgo_entity_begin` = []

`cgo_entity_end` = []

`__init__`()

`clean`()

`new`()

`get`()

class **`BasicPymolCGOLines`**
Bases: `aqueduct.visual.pymol_connector.BasicPymolCGO`

cgo_entity_begin = [2.0, 1.0]

cgo_entity_end = [3.0]

add (*coords=None*, *color=None*)

class **BasicPymolCGOSpheres**
Bases: aqueduct.visual.pymol_connector.BasicPymolCGO

cgo_entity_begin = []

cgo_entity_end = []

add (*coords=None*, *radius=None*, *color=None*)

class **BasicPymolCGOPointers**
Bases: aqueduct.visual.pymol_connector.BasicPymolCGO

cgo_entity_begin = []

cgo_entity_end = []

add_cone (*coords1=None*, *coords2=None*, *radius1=None*, *radius2=None*, *color1=None*, *color2=None*)

add_pointer (*point=None*, *direction=None*, *length=None*, *color=None*, *reverse=False*)

class **SimpleTarWriteHelper**
Bases: object

__init__ ()

open (*filename*)

save_object2tar (*obj*, *name*)

save_file2tar (*filename*, *name*)

__del__ ()

class **ConnectToPymol**
Bases: object

cgo_line_width = 2.0

ct_pymol = 'pymol'

ct_file = 'file'

__init__ ()

init_pymol ()

init_script (*filename*)

add_cgo_object (*name*, *cgo_object*, *state=None*)

del_cgo_object (*name*, *state=None*)

load_pdb (*name*, *filename*, *state=None*)

orient_on (*name*)

__del__ ()

class **SinglePathPlotter** (*pymol_connector*, *linearize=None*)
Bases: object

__init__ (*pymol_connector*, *linearize=None*)

add_single_path_continous_trace (*spath*, *smooth=None*, *plot_in=True*, *plot_object=True*, *plot_out=True*, ***kwargs*)

paths_trace (*spaths*, *smooth=None*, *name='paths'*, *state=None*, ***kwargs*)

paths_inlets (*spaths, smooth=None, color=None, plot_in=True, plot_out=True, name='in-out-let', state=None, **kwargs*)

scatter (*coords, radius=0.4, color='r', name='scatter', state=None*)

convexhull (*chull, color='m', name='convexhull', state=None*)

aqueduct.visual.quickplot module

yield_spath_len_and_smooth_diff_in_types_slices (*sp, smooth=None, smooth_len=None, smooth_diff=None, types='etypes'*)

plot_colorful_lines (*x, y, c, **kwargs*)

spaths_spectra (*spaths, **kwargs*)

plot_spath_spectrum (*sp, **kwargs*)

spath_spectrum (*sp, **kwargs*)

showit (*gen*)

get_ax3d (*fig, sub=111*)

class **SimpleTracePlotter**

Bases: object

plot_line (*coords, color, **kwargs*)

single_trace (*coords, color='r', **kwargs*)

path_trace (*path, color=('r', 'g', 'b'), plot_in=True, plot_object=True, plot_out=True, **kwargs*)

class **SimpleProteinPlotter**

Bases: *aqueduct.visual.quickplot.SimpleTracePlotter*

protein_trace (*protein, smooth=None, color=('c', 'm', 'y'), **kwargs*)

class **SimplePathPlotter**

Bases: *aqueduct.visual.quickplot.SimpleTracePlotter*

single_path_traces (*spaths, smooth=None, color=('r', 'g', 'b'), **kwargs*)

class **MPLTracePlotter**

Bases: *aqueduct.visual.quickplot.SimplePathPlotter*, *aqueduct.visual.quickplot.SimpleProteinPlotter*

init_ax (**args, **kwargs*)

plot_line (**args, **kwargs*)

scatter (**args, **kwargs*)

Module contents

5.1.2 Module contents

Aqueduct - a collection of tools to trace residues in MD simulation.

version ()

Returns *aqueduct* version number.

Returns 3 element tuple of int numbers

Return type tuple

version_nice()

Returns *aqueduct* version number as nicely formated string.

Returns string composed on the basis of the number returned by *version()*.

Return type str

greetings()

Returns fancy greetings of *aqueduct*. It has a form of ASCII-like graphic. Currently it returns following string:

```
------------------------------------------------
           ~ ~ ~ A Q U E D U C T ~ ~ ~
################################################
####        ########        ########        ####
##            ####            ####            ##
#              ##              ##              #
#              ##              ##              #
#              ##              ##              #
#              ##              ##              #
------------------------------------------------
```

Returns *aqueduct* fancy greetings.

Return type str

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

Symbols