ProDy

Protein Dynamics & Sequence Analysis

ProDy Documentation

Release 2.4.1

Ahmet Bakan

June 07, 2024

CONTENTS

CHAPTER

ONE

INSTALLATION

1.1 Required Software

- Python[1] 2.7, 3.6 or later. We recommend using Anaconda[2], which provides the conda package and environment manager as well as many useful packages.
- NumPy[3] 1.10 or later
- *SciPy* - we recommend that you use the latest version, but all versions should be supported.
- *Biopython* - we recommend that you use the latest version, but all versions should be supported.

When compiling from source, on Linux for example, you will need a C compiler (e.g. **gcc**) and Python developer libraries (i.e. `python.h`). If you don't have Python developer libraries installed on your machine, use your package manager to install `python-dev` package.

In addition, matplotlib[4] is required for using plotting functions. ProDy, *ProDy Applications* (page 3), and *Evol Applications* (page 16) can be operated without this package.

1.2 Quick Install

If you have pip[5] installed, type the following:

```
pip install -U ProDy
```

If you don't have pip[6], please download an installation file and follow the instructions.

If you have conda installed, you can also type the following instead:

```
conda install ProDy
```

1.3 Download & Install

After installing the required packages, you will need to download a suitable ProDy source or installation file from http://python.org/pypi/ProDy. For changes and list of new features see *Release Notes* (page 345).

Linux

Download `ProDy-x.y.z.tar.gz`. Extract tarball contents and run `setup.py` as follows:

[1] http://www.python.org
[2] https://www.anaconda.com/products/individual
[3] http://www.numpy.org
[4] http://matplotlib.org
[5] https://pypi.python.org/pypi/pip
[6] https://pypi.python.org/pypi/pip

```
$ tar -xzf ProDy-x.y.z.tar.gz
$ cd ProDy-x.y.z
$ python setup.py build
$ python setup.py install
```

If you need root access for installation, try `sudo python setup.py install`. If you don't have root access, please consult alternate and custom installation schemes in Installing Python Modules[7].

Mac OS

For installing ProDy, please follow the Linux installation instructions.

Windows

Remove previously installed ProDy release from **`Uninstall a program`** in *Control Panel*.

Download `ProDy-1.x.y.win32-py2.z.exe` and run to install ProDy.

To be able use *ProDy Applications* (page 3) and *Evol Applications* (page 16) in command prompt (**`cmd.exe`**), append Python and scripts folders (e.g. `C:\Python27` and `C:\Python27\Scripts`) to `PATH`[8] environment variable.

1.4 Recommended Software

- **'Scipy'**_, when installed, replaces linear algebra module of Numpy. Scipy linear algebra module is more flexible and can be faster.
- IPython[9] is a must have for interactive ProDy sessions.
- PyReadline[10] for colorful IPython sessions on Windows.
- MDAnalysis[11] or MDTraj[12] for reading molecular dynamics trajectories.

1.5 Included in ProDy

Following software is included in the ProDy installation packages:

- pyparsing[13] is used to define the atom selection grammar.
- **'Biopython'**_ KDTree package and pairwise2 module are used for distance based atom selections and pairwise sequence alignment, respectively.
- argparse[14] is used to implement applications and provided for compatibility with Python 2.6.

1.6 Source Code

Source code is available at https://github.com/prody/ProDy.

[7] http://docs.python.org/install/index.html
[8] http://matplotlib.sourceforge.net/install/environment_variables_faq.html#envvar-PATH
[9] http://ipython.org
[10] http://ipython.org/pyreadline.html
[11] https://www.mdanalysis.org
[12] https://www.mdtraj.org
[13] http://pyparsing.wikispaces.com
[14] http://code.google.com/p/argparse/

CHAPTER

TWO

APPLICATIONS

ProDy comes with two sets of applications that automate structural dynamics and sequence coevolution analysis:

2.1 ProDy Applications

ProDy applications are command line programs that automates structure processing and structural dynamics analysis:

2.1.1 prody align

Usage

Running **`prody align -h`** displays:

```
usage: prody align [-h] [--quiet] [--examples] [-s SEL] [-m INT] [-i INT]
                   [-o INT] [-p STR] [-x STR]
                   pdb [pdb ...]

positional arguments:
  pdb                   PDB identifier(s) or filename(s)

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

atom/model selection:
  -s SEL, --select SEL  reference structure atom selection (default: calpha)
  -m INT, --model INT   for NMR files, reference model index (default: 1)

chain matching options:
  -i INT, --seqid INT   percent sequence identity (default: 90)
  -o INT, --overlap INT
                        percent sequence overlap (default: 90)

output options:
  -p STR, --prefix STR  output filename prefix (default: PDB filename)
  -x STR, --suffix STR  output filename suffix (default: _aligned)
```

Examples

Running **`prody align --examples`** displays:

```
Align models in a PDB structure or multiple PDB structures and save
aligned coordinate sets.  When multiple structures are aligned, ProDy
will match chains based on sequence alignment and use best match for
aligning the structures.

Fetch PDB structure 2k39 and align models (reference model is the
first model):

  $ prody align 2k39

Fetch PDB structure 2k39 and align models using backbone of residues
with number less than 71:

  $ prody align 2k39 --select "backbone and resnum < 71"

Align 1r39 and 1zz2 onto 1p38 using residues with number less than
300:

  $ prody align --select "resnum < 300" 1p38 1r39 1zz2

Align all models of 2k39 onto 1aar using residues 1 to 70 (inclusive):

  $ prody align --select "resnum 1 to 70" 1aar 2k39

Align 1fi7 onto 1hrc using heme atoms:

  $ prody align --select "noh heme and chain A" 1hrc 1fi7
```

2.1.2 prody anm

Usage

Running **`prody anm -h`** displays:

```
usage: prody anm [-h] [--quiet] [--examples] [-n INT] [-s SEL] [-c FLOAT]
                 [-g FLOAT] [-m INT] [-a] [-o PATH] [-e] [-r] [-u] [-q] [-v]
                 [-z] [-t STR] [-b] [-l] [-k] [-p STR] [-f STR] [-d STR]
                 [-x STR] [-A] [-R] [-Q] [-B] [-K] [-F STR] [-D INT]
                 [-W FLOAT] [-H FLOAT]
                 pdb

positional arguments:
  pdb                   PDB identifier or filename

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

parameters:
  -n INT, --number-of-modes INT
                        number of non-zero eigenvectors (modes) to calculate
                        (default: 10)
  -s SEL, --select SEL  atom selection (default: "protein and name CA or
                        nucleic and name P C4' C2")
  -c FLOAT, --cutoff FLOAT
                        cutoff distance (A) (default: 15.0)
  -g FLOAT, --gamma FLOAT
```

```
                        spring constant (default: 1.0)
  -m INT, --model INT   index of model that will be used in the calculations

output:
  -a, --all-output      write all outputs
  -o PATH, --output-dir PATH
                        output directory (default: .)
  -e, --eigenvs         write eigenvalues/vectors
  -r, --cross-correlations
                        write cross-correlations
  -u, --heatmap         write cross-correlations heatmap file
  -q, --square-fluctuations
                        write square-fluctuations
  -v, --covariance      write covariance matrix
  -z, --npz             write compressed ProDy data file
  -t STR, --extend STR  write NMD file for the model extended to "backbone"
                        ("bb") or "all" atoms of the residue, model must have
                        one node per residue
  -b, --beta-factors    write beta-factors calculated from GNM modes
  -l, --hessian         write Hessian matrix
  -k, --kirchhoff       write Kirchhoff matrix

output options:
  -p STR, --file-prefix STR
                        output file prefix (default: pdb_anm)
  -f STR, --number-format STR
                        number output format (default: %12g)
  -d STR, --delimiter STR
                        number delimiter (default: " ")
  -x STR, --extension STR
                        numeric file extension (default: .txt)

figures:
  -A, --all-figures     save all figures
  -R, --cross-correlations-figure
                        save cross-correlations figure
  -Q, --square-fluctuations-figure
                        save square-fluctuations figure
  -B, --beta-factors-figure
                        save beta-factors figure
  -K, --contact-map     save contact map (Kirchhoff matrix) figure

figure options:
  -F STR, --figure-format STR
                        pdf (default: pdf)
  -D INT, --dpi INT     figure resolution (dpi) (default: 300)
  -W FLOAT, --width FLOAT
                        figure width (inch) (default: 8.0)
  -H FLOAT, --height FLOAT
                        figure height (inch) (default: 6.0)
```

Examples

Running **`prody anm --examples`** displays:

```
Perform ANM calculations for given PDB structure and output results in
NMD format.  If an identifier is passed, structure file will be
downloaded from the PDB FTP server.
```

```
Fetch PDB 1p38, run ANM calculations using default parameters, and
write NMD file:

  $ prody anm 1p38

Fetch PDB 1aar, run ANM calculations using default parameters for
chain A carbon alpha atoms with residue numbers less than 70, and save
all of the graphical output files:

  $ prody anm 1aar -s "calpha and chain A and resnum < 70" -A
```

2.1.3 prody biomol

Usage

Running **`prody biomol -h`** displays:

```
usage: prody biomol [-h] [--quiet] [--examples] [-p STR] [-b INT] pdb

positional arguments:
  pdb                   PDB identifier or filename

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit
  -p STR, --prefix STR  prefix for output files (default: pdb_biomol_)
  -b INT, --biomol INT  index of the biomolecule, by default all are generated
```

Examples

Running **`prody biomol --examples`** displays:

```
Generate biomolecule coordinates:

  $ prody biomol 2bfu
```

2.1.4 prody blast

Usage

Running **`prody blast -h`** displays:

```
usage: prody blast [-h] [--quiet] [--examples] [-i FLOAT] [-o FLOAT] [-d PATH]
                   [-z] [-f STR] [-e FLOAT] [-l INT] [-s INT] [-t INT]
                   sequence

positional arguments:
  sequence              sequence or file in fasta format

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit
  -i FLOAT, --identity FLOAT
                        percent sequence identity (default: 90.0)
  -o FLOAT, --overlap FLOAT
```

```
                        percent sequence overlap (default: 90.0)
  -d PATH, --output-dir PATH
                        download uncompressed PDB files to given directory
  -z, --gzip            write compressed PDB file

Blast Parameters:
  -f STR, --filename STR
                        a filename to save the results in XML format
  -e FLOAT, --expect FLOAT
                        blast search parameter
  -l INT, --hit-list-size INT
                        blast search parameter
  -s INT, --sleep-time INT
                        how long to wait to reconnect for results (sleep time
                        is doubled when results are not ready)
  -t INT, --timeout INT
                        when to give up waiting for results
```

Examples

Running **`prody blast --examples`** displays:

```
Blast search PDB for the first sequence in a fasta file:

  $ prody blast seq.fasta -i 70

Blast search PDB for the sequence argument:

  $ prody blast MQIFVKTLTGKTITLEVEPSDTIENVKAKIQDKEGIPPDQQRLIFAGKQLEDGRTLSDYNIQKESTLHLVLRLRGG

Blast search PDB for avidin structures, download files, and align all
files onto the 2avi structure:

  $ prody blast -d . ARKCSLTGKWTNDLGSNMTIGAVNSRGEFTGTYITAVTATSNEIKESPLHGTQNTINKRTQPTFGFTVNWKFSESTTVF

  $ prody align 2avi.pdb *pdb
```

2.1.5 prody catdcd

Usage

Running **`prody catdcd -h`** displays:

```
usage: prody catdcd [-h] [--quiet] [--examples] [-s SEL] [-o FILE] [-n]
                    [--psf PSF] [--pdb PDB] [--first INT] [--last INT]
                    [--stride INT] [--align SEL]
                    dcd [dcd ...]

positional arguments:
  dcd                   DCD filename(s) (all must have same number of atoms)

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit
  -s SEL, --select SEL  atom selection (default: all)
  -o FILE, --output FILE
                        output filename (default: trajectory.dcd)
```

```
  -n, --num             print the number of frames in each file and exit
  --psf PSF             PSF filename (must have same number of atoms as DCDs)
  --pdb PDB             PDB filename (must have same number of atoms as DCDs)
  --first INT           index of the first output frame, default: 0
  --last INT            index of the last output frame, default: -1
  --stride INT          number of steps between output frames, default: 1
  --align SEL           atom selection for aligning frames, a PSF or PDB file
                        must be provided, if a PDB is provided frames will be
                        superposed onto PDB coordinates
```

Examples

Running **`prody catdcd --examples`** displays:

```
Concatenate two DCD files and output all atmos:

  $ prody catdcd mdm2.dcd mdm2sim2.dcd

Concatenate two DCD files and output backbone atoms:

  $ prody catdcd mdm2.dcd mdm2sim2.dcd --pdb mdm2.pdb -s bb
```

2.1.6 prody contacts

Usage

Running **`prody contacts -h`** displays:

```
usage: prody contacts [-h] [--quiet] [--examples] [-s SELSTR] [-r FLOAT]
                      [-t STR] [-p STR] [-x STR]
                      target ligand [ligand ...]

positional arguments:
  target                target PDB identifier or filename
  ligand                ligand PDB identifier(s) or filename(s)

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit
  -s SELSTR, --select SELSTR
                        selection string for target
  -r FLOAT, --radius FLOAT
                        contact radius (default: 4.0)
  -t STR, --extend STR  output same residue, chain, or segment as contacting
                        atoms
  -p STR, --prefix STR  output filename prefix (default: target filename)
  -x STR, --suffix STR  output filename suffix (default: _contacts)
```

Examples

Running **`prody contacts --examples`** displays:

```
Identify contacts of a target structure with one or more ligands.

Fetch PDB structure 1zz2, save PDB files for individual ligands, and
identify contacting residues of the target protein:
```

```
  $ prody select -o B11 "resname B11" 1zz2

  $ prody select -o BOG "resname BOG" 1zz2

  $ prody contacts -r 4.0 -t residue -s protein 1zz2 B11.pdb BOG.pdb
```

2.1.7 prody eda

Usage

Running **`prody eda -h`** displays:

```
usage: prody eda [-h] [--quiet] [--examples] [-n INT] [-s SEL] [-a] [-o PATH]
                 [-e] [-r] [-u] [-q] [-v] [-z] [-t STR] [-j] [-p STR] [-f STR]
                 [-d STR] [-x STR] [-A] [-R] [-Q] [-J STR] [-F STR] [-D INT]
                 [-W FLOAT] [-H FLOAT] [--psf PSF | --pdb PDB] [--aligned]
                 dcd

positional arguments:
  dcd                   file in DCD or PDB format

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit
  --psf PSF             PSF filename
  --pdb PDB             PDB filename
  --aligned             trajectory is already aligned

parameters:
  -n INT, --number-of-modes INT
                        number of non-zero eigenvectors (modes) to calculate
                        (default: 10)
  -s SEL, --select SEL  atom selection (default: "protein and name CA or
                        nucleic and name P C4' C2")

output:
  -a, --all-output      write all outputs
  -o PATH, --output-dir PATH
                        output directory (default: .)
  -e, --eigenvs         write eigenvalues/vectors
  -r, --cross-correlations
                        write cross-correlations
  -u, --heatmap         write cross-correlations heatmap file
  -q, --square-fluctuations
                        write square-fluctuations
  -v, --covariance      write covariance matrix
  -z, --npz             write compressed ProDy data file
  -t STR, --extend STR  write NMD file for the model extended to "backbone"
                        ("bb") or "all" atoms of the residue, model must have
                        one node per residue
  -j, --projection      write projections onto PCs

output options:
  -p STR, --file-prefix STR
                        output file prefix (default: pdb_pca)
  -f STR, --number-format STR
                        number output format (default: %12g)
```

```
  -d STR, --delimiter STR
                        number delimiter (default: " ")
  -x STR, --extension STR
                        numeric file extension (default: .txt)

figures:
  -A, --all-figures     save all figures
  -R, --cross-correlations-figure
                        save cross-correlations figure
  -Q, --square-fluctuations-figure
                        save square-fluctuations figure
  -J STR, --projection-figure STR
                        save projections onto specified subspaces, e.g. "1,2"
                        for projections onto PCs 1 and 2; "1,2 1,3" for
                        projections onto PCs 1,2 and 1, 3; "1 1,2,3" for
                        projections onto PCs 1 and 1, 2, 3

figure options:
  -F STR, --figure-format STR
                        pdf (default: pdf)
  -D INT, --dpi INT     figure resolution (dpi) (default: 300)
  -W FLOAT, --width FLOAT
                        figure width (inch) (default: 8.0)
  -H FLOAT, --height FLOAT
                        figure height (inch) (default: 6.0)
```

Examples

Running **`prody eda --examples`** displays:

```
This command performs PCA (or EDA) calculations for given multi-model
PDB structure or DCD format trajectory file and outputs results in NMD
format.  If a PDB identifier is given, structure file will be
downloaded from the PDB FTP server.  DCD files may be accompanied with
PDB or PSF files to enable atoms selections.

Fetch pdb 2k39, perform PCA calculations, and output NMD file:

  $ prody pca 2k39

Fetch pdb 2k39 and perform calculations for backbone of residues up to
71, and save all output and figure files:

  $ prody pca 2k39 --select "backbone and resnum < 71" -a -A

Perform EDA of MDM2 trajectory:

  $ prody eda mdm2.dcd

Perform EDA for backbone atoms:

  $ prody eda mdm2.dcd --pdb mdm2.pdb --select backbone
```

2.1.8 prody fetch

Usage

Running **`prody fetch -h`** displays:

```
usage: prody fetch [-h] [--quiet] [--examples] [-d PATH] [-z] pdb [pdb ...]

positional arguments:
  pdb                   PDB identifier(s) or a file that contains them

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit
  -d PATH, --dir PATH   target directory for saving PDB file(s)
  -z, --gzip            write compressed PDB file(S)
```

Examples

Running **`prody fetch --examples`** displays:

```
Download PDB file(s) by specifying identifiers:

  $ prody fetch 1mkp 1p38
```

2.1.9 prody gnm

Usage

Running **`prody gnm -h`** displays:

```
usage: prody gnm [-h] [--quiet] [--examples] [-n INT] [-s SEL] [-c FLOAT]
                 [-g FLOAT] [-m INT] [-a] [-o PATH] [-e] [-r] [-u] [-q] [-v]
                 [-z] [-t STR] [-b] [-k] [-p STR] [-f STR] [-d STR] [-x STR]
                 [-A] [-R] [-Q] [-B] [-K] [-M STR] [-F STR] [-D INT]
                 [-W FLOAT] [-H FLOAT]
                 pdb

positional arguments:
  pdb                   PDB identifier or filename

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

parameters:
  -n INT, --number-of-modes INT
                        number of non-zero eigenvectors (modes) to calculate
                        (default: 10)
  -s SEL, --select SEL  atom selection (default: "protein and name CA or
                        nucleic and name P C4' C2")
  -c FLOAT, --cutoff FLOAT
                        cutoff distance (A) (default: 10.0)
  -g FLOAT, --gamma FLOAT
                        spring constant (default: 1.0)
  -m INT, --model INT   index of model that will be used in the calculations

output:
  -a, --all-output      write all outputs
  -o PATH, --output-dir PATH
                        output directory (default: .)
  -e, --eigenvs         write eigenvalues/vectors
```

```
  -r, --cross-correlations
                        write cross-correlations
  -u, --heatmap         write cross-correlations heatmap file
  -q, --square-fluctuations
                        write square-fluctuations
  -v, --covariance      write covariance matrix
  -z, --npz             write compressed ProDy data file
  -t STR, --extend STR  write NMD file for the model extended to "backbone"
                        ("bb") or "all" atoms of the residue, model must have
                        one node per residue
  -b, --beta-factors    write beta-factors calculated from GNM modes
  -k, --kirchhoff       write Kirchhoff matrix

output options:
  -p STR, --file-prefix STR
                        output file prefix (default: pdb_gnm)
  -f STR, --number-format STR
                        number output format (default: %12g)
  -d STR, --delimiter STR
                        number delimiter (default: " ")
  -x STR, --extension STR
                        numeric file extension (default: .txt)

figures:
  -A, --all-figures     save all figures
  -R, --cross-correlations-figure
                        save cross-correlations figure
  -Q, --square-fluctuations-figure
                        save square-fluctuations figure
  -B, --beta-factors-figure
                        save beta-factors figure
  -K, --contact-map     save contact map (Kirchhoff matrix) figure
  -M STR, --mode-shape-figure STR
                        save mode shape figures for specified modes, e.g. "1-3
                        5" for modes 1, 2, 3 and 5

figure options:
  -F STR, --figure-format STR
                        pdf (default: pdf)
  -D INT, --dpi INT     figure resolution (dpi) (default: 300)
  -W FLOAT, --width FLOAT
                        figure width (inch) (default: 8.0)
  -H FLOAT, --height FLOAT
                        figure height (inch) (default: 6.0)
```

Examples

Running **`prody gnm --examples`** displays:

```
This command performs GNM calculations for given PDB structure and
outputs results in NMD format. If an identifier is passed, structure
file will be downloaded from the PDB FTP server.

Fetch PDB 1p38, run GNM calculations using default parameters, and
results:

  $ prody gnm 1p38
```

```
Fetch PDB 1aar, run GNM calculations with cutoff distance 7 angstrom
for chain A carbon alpha atoms with residue numbers less than 70, and
save all of the graphical output files:

  $ prody gnm 1aar -c 7 -s "calpha and chain A and resnum < 70" -A
```

2.1.10 prody pca

Usage

Running **`prody pca -h`** displays:

```
usage: prody pca [-h] [--quiet] [--examples] [-n INT] [-s SEL] [-a] [-o PATH]
                 [-e] [-r] [-u] [-q] [-v] [-z] [-t STR] [-j] [-p STR] [-f STR]
                 [-d STR] [-x STR] [-A] [-R] [-Q] [-J STR] [-F STR] [-D INT]
                 [-W FLOAT] [-H FLOAT] [--psf PSF | --pdb PDB] [--aligned]
                 dcd

positional arguments:
  dcd                   file in DCD or PDB format

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit
  --psf PSF             PSF filename
  --pdb PDB             PDB filename
  --aligned             trajectory is already aligned

parameters:
  -n INT, --number-of-modes INT
                        number of non-zero eigenvectors (modes) to calculate
                        (default: 10)
  -s SEL, --select SEL  atom selection (default: "protein and name CA or
                        nucleic and name P C4' C2")

output:
  -a, --all-output      write all outputs
  -o PATH, --output-dir PATH
                        output directory (default: .)
  -e, --eigenvs         write eigenvalues/vectors
  -r, --cross-correlations
                        write cross-correlations
  -u, --heatmap         write cross-correlations heatmap file
  -q, --square-fluctuations
                        write square-fluctuations
  -v, --covariance      write covariance matrix
  -z, --npz             write compressed ProDy data file
  -t STR, --extend STR  write NMD file for the model extended to "backbone"
                        ("bb") or "all" atoms of the residue, model must have
                        one node per residue
  -j, --projection      write projections onto PCs

output options:
  -p STR, --file-prefix STR
                        output file prefix (default: pdb_pca)
  -f STR, --number-format STR
                        number output format (default: %12g)
```

```
  -d STR, --delimiter STR
                        number delimiter (default: " ")
  -x STR, --extension STR
                        numeric file extension (default: .txt)

figures:
  -A, --all-figures     save all figures
  -R, --cross-correlations-figure
                        save cross-correlations figure
  -Q, --square-fluctuations-figure
                        save square-fluctuations figure
  -J STR, --projection-figure STR
                        save projections onto specified subspaces, e.g. "1,2"
                        for projections onto PCs 1 and 2; "1,2 1,3" for
                        projections onto PCs 1,2 and 1, 3; "1 1,2,3" for
                        projections onto PCs 1 and 1, 2, 3

figure options:
  -F STR, --figure-format STR
                        pdf (default: pdf)
  -D INT, --dpi INT     figure resolution (dpi) (default: 300)
  -W FLOAT, --width FLOAT
                        figure width (inch) (default: 8.0)
  -H FLOAT, --height FLOAT
                        figure height (inch) (default: 6.0)
```

Examples

Running **`prody pca --examples`** displays:

```
This command performs PCA (or EDA) calculations for given multi-model
PDB structure or DCD format trajectory file and outputs results in NMD
format.  If a PDB identifier is given, structure file will be
downloaded from the PDB FTP server.  DCD files may be accompanied with
PDB or PSF files to enable atoms selections.

Fetch pdb 2k39, perform PCA calculations, and output NMD file:

  $ prody pca 2k39

Fetch pdb 2k39 and perform calculations for backbone of residues up to
71, and save all output and figure files:

  $ prody pca 2k39 --select "backbone and resnum < 71" -a -A

Perform EDA of MDM2 trajectory:

  $ prody eda mdm2.dcd

Perform EDA for backbone atoms:

  $ prody eda mdm2.dcd --pdb mdm2.pdb --select backbone
```

2.1.11 prody select

Usage

Running **`prody select -h`** displays:

```
usage: prody select [-h] [--quiet] [--examples] [-o STR] [-p STR] [-x STR]
                    select pdb [pdb ...]

positional arguments:
  select                atom selection string
  pdb                   PDB identifier(s) or filename(s)

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

output options:
  -o STR, --output STR  output PDB filename (default: pdb_selected.pdb)
  -p STR, --prefix STR  output filename prefix (default: PDB filename)
  -x STR, --suffix STR  output filename suffix (default: _selected)
```

Examples

Running **`prody select --examples`** displays:

```
This command selects specified atoms and writes them in a PDB file.

Fetch PDB files 1p38 and 1r39 and write backbone atoms in a file:

  $ prody select backbone 1p38 1r39
```

Running **`prody`** command will provide a description of applications:

```
$ prody
```

```
usage: prody [-h] [-c] [-v] {anm,gnm,pca,eda,align,blast,biomol,catdcd,contacts,fetch,select,energy,c

ProDy: A Python Package for Protein Dynamics Analysis

optional arguments:
  -h, --help            show this help message and exit
  -c, --cite            print citation info and exit
  -v, --version         print ProDy version and exit

subcommands:
  {anm,gnm,pca,eda,align,blast,biomol,catdcd,contacts,fetch,select,energy,clustenm}
    anm                 perform anisotropic network model normal mode analysis calculations
    gnm                 perform Gaussian network model calculations
    pca                 perform principal component analysis calculations
    eda                 perform essential dynamics analysis calculations
    align               align models or structures
    blast               blast search Protein Data Bank
    biomol              build biomolecules
    catdcd              concatenate dcd files
    contacts            identify contacts between a target and ligand(s)
    fetch               fetch a PDB file
    select              select atoms and write a PDB file
    energy              fix missing atoms, solvate, minimise and calculate energy
    clustenm            run clustenm(d) simulations

See 'prody <command> -h' for more information on a specific command.
```

Detailed information on a specific application can be obtained by typing the command and application

names as **prody anm -h**.

Running **prody anm** application as follows will perform ANM calculations for the p38 MAP kinase structure, and will write eigenvalues/vectors in plain text and *NMD Format* (page 173):

```
$ prody anm 1p38
```

In the above example, the default parameters (`cutoff=15.` and `gamma=1.`) and all of the Cα atoms of the protein structure 1p38 are used.

In the example below, the *cutoff* distance is changed to 14 Å, and the Cα atoms of residues with numbers smaller than 340 are used, the output files are prefixed with `p38_anm`:

```
$ prody anm -c 14 -s "calpha resnum < 340" -p p38_anm 1p38
```

The output file `p38_anm.nmd` can be visualized using **'NMWiz'**_.

2.2 Evol Applications

Evol applications are command line programs that automate retrieval, refinement, and analysis of multiple sequence alignments:

2.2.1 evol coevol

Usage

Running **evol coevol -h** displays:

```
usage: evol coevol [-h] [--quiet] [--examples] [-n] [-c STR] [-m STR] [-t]
                   [-p STR] [-f STR] [-S] [-L FLOAT] [-U FLOAT] [-X STR]
                   [-T STR] [-D INT] [-H FLOAT] [-W FLOAT] [-F STR]
                   msa

positional arguments:
  msa                   refined MSA file

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

calculation options:
  -n, --no-ambiguity    treat amino acids characters B, Z, J, and X as non-
                        ambiguous
  -c STR, --correction STR
                        also save corrected mutual information matrix data and
                        plot, one of apc, asc
  -m STR, --normalization STR
                        also save normalized mutual information matrix data
                        and plot, one of sument, minent, maxent, mincon,
                        maxcon, joint

output options:
  -t, --heatmap         save heatmap files for all mutual information matrices
  -p STR, --prefix STR  output filename prefix, default is msa filename with
                        _coevol suffix
  -f STR, --number-format STR
                        number output format (default: %12g)
```

```
figure options:
  -S, --save-plot       save coevolution plot
  -L FLOAT, --cmin FLOAT
                        apply lower limits for figure plot
  -U FLOAT, --cmax FLOAT
                        apply upper limits for figure plot
  -X STR, --xlabel STR  specify xlabel, by default will be applied on ylabel
  -T STR, --title STR   figure title
  -D INT, --dpi INT     figure resolution (dpi) (default: 300)
  -H FLOAT, --height FLOAT
                        figure height (inch) (default: 6)
  -W FLOAT, --width FLOAT
                        figure width (inch) (default: 8)
  -F STR, --figure-format STR
                        figure file format, one of svgz, rgba, png, pdf, eps,
                        svg, ps, raw (default: pdf)
```

Examples

Running **`evol coevol --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.2 evol conserv

Usage

Running **`evol conserv -h`** displays:

```
usage: evol conserv [-h] [--quiet] [--examples] [-n] [-g] [-p STR] [-f STR]
                    [-S] [-H FLOAT] [-W FLOAT] [-F STR] [-D INT]
                    msa

positional arguments:
  msa                   refined MSA file

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

calculation options:
  -n, --no-ambiguity    treat amino acids characters B, Z, J, and X as non-
                        ambiguous
  -g, --gaps            do not omit gap characters

output options:
  -p STR, --prefix STR  output filename prefix, default is msa filename with
                        _conserv suffix
  -f STR, --number-format STR
                        number output format (default: %12g)

figure options:
  -S, --save-plot       save conservation plot
  -H FLOAT, --height FLOAT
                        figure height (inch) (default: 6)
  -W FLOAT, --width FLOAT
                        figure width (inch) (default: 8)
  -F STR, --figure-format STR
                        figure file format, one of raw, png, ps, svgz, eps,
                        pdf, rgba, svg (default: pdf)
  -D INT, --dpi INT     figure resolution (dpi) (default: 300)
```

Examples

Running **`evol conserv --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S
```

```
Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.3 evol fetch

Usage

Running **`evol fetch -h`** displays:

```
usage: evol fetch [-h] [--quiet] [--examples] [-a STR] [-f STR] [-o STR]
                  [-i STR] [-g STR] [-t INT] [-d PATH] [-p STR] [-z]
                  acc

positional arguments:
  acc                   Pfam accession or ID

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

download options:
  -a STR, --alignment STR
                        alignment type, one of full, seed, ncbi, metagenomics
                        (default: full)
  -f STR, --format STR  Pfam supported MSA format, one of selex, fasta,
                        stockholm (default: selex)
  -o STR, --order STR   ordering of sequences, one of tree, alphabetical
                        (default: tree)
  -i STR, --inserts STR
                        letter case for inserts, one of upper, lower (default:
                        upper)
  -g STR, --gaps STR    gap character, one of dashes, dots, mixed (default:
                        dashes)
  -t INT, --timeout INT
                        timeout for blocking connection attempts (default: 60)

output options:
  -d PATH, --outdir PATH
                        output directory (default: .)
  -p STR, --outname STR
                        output filename, default is accession and alignment
                        type
  -z, --compressed      gzip downloaded MSA file
```

Examples

Running **`evol fetch --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.4 evol filter

Usage

Running **`evol filter -h`** displays:

```
usage: evol filter [-h] [--quiet] [--examples] (-s | -e | -c) [-F] [-o STR]
                   [-f STR] [-z]
                   msa word [word ...]

positional arguments:
  msa                   MSA filename to be filtered
  word                  word to be compared to sequence label

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

filtering method (required):
  -s, --startswith      sequence label starts with given words
  -e, --endswith        sequence label ends with given words
  -c, --contains        sequence label contains with given words

filter option:
  -F, --full-label      compare full label with word(s)
```

```
output options:
  -o STR, --outname STR
                        output filename, default is msa filename with _refined
                        suffix
  -f STR, --format STR  output MSA file format, default is same as input
  -z, --compressed      gzip refined MSA output
```

Examples

Running **`evol filter --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.5 evol merge

Usage

Running **`evol merge -h`** displays:

```
usage: evol merge [-h] [--quiet] [--examples] [-o STR] [-f STR] [-z]
                  msa [msa ...]

positional arguments:
  msa                   MSA filenames to be merged

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
```

```
  --examples            show usage examples and exit

output options:
  -o STR, --outname STR
                        output filename, default is first input filename with
                        _merged suffix
  -f STR, --format STR  output MSA file format, default is same as first input
                        MSA
  -z, --compressed      gzip merged MSA output
```

Examples

Running **`evol merge --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.6 evol occupancy

Usage

Running **`evol occupancy -h`** displays:

```
usage: evol occupancy [-h] [--quiet] [--examples] [-o STR] [-p STR] [-l STR]
                      [-f STR] [-S] [-X STR] [-Y STR] [-T STR] [-D INT]
                      [-W FLOAT] [-F STR] [-H FLOAT]
                      msa

positional arguments:
```

```
  msa                   MSA file

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

calculation options:
  -o STR, --occ-axis STR
                        calculate row or column occupancy or both., one of
                        row, col, both (default: row)

output options:
  -p STR, --prefix STR  output filename prefix, default is msa filename with
                        _occupancy suffix
  -l STR, --label STR   index for column based on msa label
  -f STR, --number-format STR
                        number output format (default: %12g)

figure options:
  -S, --save-plot       save occupancy plot/s
  -X STR, --xlabel STR  specify xlabel
  -Y STR, --ylabel STR  specify ylabel
  -T STR, --title STR   figure title
  -D INT, --dpi INT     figure resolution (dpi) (default: 300)
  -W FLOAT, --width FLOAT
                        figure width (inch) (default: 8)
  -F STR, --figure-format STR
                        figure file format, one of png, pdf, raw, svg, eps,
                        ps, svgz, rgba (default: pdf)
  -H FLOAT, --height FLOAT
                        figure height (inch) (default: 6)
```

Examples

Running **`evol occupancy --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:
```

```
  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.7 evol rankorder

Usage

Running **`evol rankorder -h`** displays:

```
usage: evol rankorder [-h] [--quiet] [--examples] [-z] [-d STR] [-p STR]
                      [-m STR] [-l STR] [-n INT] [-q INT] [-t FLOAT] [-u]
                      [-o STR]
                      mutinfo

positional arguments:
  mutinfo               mutual information matrix

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

input options:
  -z, --zscore          apply zscore for identifying top ranked coevolving
                        pairs
  -d STR, --delimiter STR
                        delimiter used in mutual information matrix file
  -p STR, --pdb STR     PDB file that contains same number of residues as the
                        mutual information matrix, output residue numbers will
                        be based on PDB file
  -m STR, --msa STR     MSA file used for building the mutual info matrix,
                        output residue numbers will be based on the most
                        complete sequence in MSA if a PDB file or sequence
                        label is not specified
  -l STR, --label STR   label in MSA file for output residue numbers

output options:
  -n INT, --num-pairs INT
                        number of top ranking residue pairs to list (default:
                        100)
  -q INT, --seq-sep INT
                        report coevolution for residue pairs that are
                        sequentially separated by input value (default: 3)
  -t FLOAT, --min-dist FLOAT
                        report coevolution for residue pairs whose CA atoms
                        are spatially separated by at least the input value,
                        used when a PDB file is given and --use-dist is true
                        (default: 10.0)
  -u, --use-dist        use structural separation to report coevolving pairs
  -o STR, --outname STR
                        output filename, default is mutinfo_rankorder.txt
```

Examples

Running **`evol rankorder --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.8 evol refine

Usage

Running **`evol refine -h`** displays:

```
usage: evol refine [-h] [--quiet] [--examples] [-l STR] [-s FLOAT] [-c FLOAT]
                   [-r FLOAT] [-k] [-o STR] [-f STR] [-z]
                   msa

positional arguments:
  msa                   MSA filename to be refined

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

refinement options:
  -l STR, --label STR   sequence label, UniProt ID code, or PDB and chain
                        identifier
  -s FLOAT, --seqid FLOAT
                        identity threshold for selecting unique sequences
```

```
  -c FLOAT, --colocc FLOAT
                        column (residue position) occupancy
  -r FLOAT, --rowocc FLOAT
                        row (sequence) occupancy
  -k, --keep            keep columns corresponding to residues not resolved in
                        PDB structure, applies label argument is a PDB
                        identifier

output options:
  -o STR, --outname STR
                        output filename, default is msa filename with _refined
                        suffix
  -f STR, --format STR  output MSA file format, default is same as input
  -z, --compressed      gzip refined MSA output
```

Examples

Running **`evol refine --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx

Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

2.2.9 evol search

Usage

Running **`evol search -h`** displays:

```
usage: evol search [-h] [--quiet] [--examples] [-b] [-s] [-g] [-e FLOAT]
                   [-t INT] [-o STR] [-d STR]
                   query

positional arguments:
  query                 protein UniProt ID or sequence, a PDB identifier, or a
                        sequence file, where sequence have no gaps and 12 or
                        more characters

optional arguments:
  -h, --help            show this help message and exit
  --quiet               suppress info messages to stderr
  --examples            show usage examples and exit

sequence search options:
  -b, --searchBs        search Pfam-B families
  -s, --skipAs          do not search Pfam-A families
  -g, --ga              use gathering threshold
  -e FLOAT, --evalue FLOAT
                        e-value cutoff, must be less than 10.0
  -t INT, --timeout INT
                        timeout in seconds for blocking connection attempt
                        (default: 60)

output options:
  -o STR, --outname STR
                        name for output file, default is standard output
  -d STR, --delimiter STR
                        delimiter for output data columns (default: )
```

Examples

Running **`evol search --examples`** displays:

```
Sequence coevolution analysis involves several steps that including
retrieving data and refining it for calculations.  These steps are
illustrated below for RnaseA protein family.

Search Pfam database:

  $  evol search 2w5i

Download Pfam MSA file:

  $  evol fetch RnaseA

Refine MSA file:

  $ evol refine RnaseA_full.slx -l RNAS1_BOVIN --seqid 0.98 --rowocc 0.8

Checking occupancy:

  $ evol occupancy RnaseA_full.slx -l RNAS1_BOVIN -o col -S

Conservation analysis:

  $ evol conserv RnaseA_full_refined.slx
```

```
Coevolution analysis:

  $ evol coevol RnaseA_full_refined.slx -S -c apc

Rank order analysis:

  $ evol rankorder RnaseA_full_refined_mutinfo_corr_apc.txt -p 2w5i_1-121.pdb --seq-sep 3
```

Running **evol** command will provide a description of applications:

```
$ evol
```

```
usage: evol [-h] [-c] [-v] [-e]

            {search,fetch,filter,refine,merge,occupancy,conserv,coevol,rankorder}
            ...

Evol: Sequence Evolution and Dynamics Analysis

optional arguments:
  -h, --help            show this help message and exit
  -c, --cite            print citation info and exit
  -v, --version         print ProDy version and exit
  -e, --examples        show usage examples and exit

subcommands:
  {search,fetch,filter,refine,merge,occupancy,conserv,coevol,rankorder}
    search              search Pfam with given query
    fetch               fetch MSA files from Pfam
    filter              filter an MSA using sequence labels
    refine              refine an MSA by removing gapped rows/colums
    merge               merge multiple MSAs based on common labels
    occupancy           calculate occupancy of rows and columns in MSA
    conserv             analyze conservation using Shannon entropy
    coevol              analyze co-evolution using mutual information
    rankorder           identify highly coevolving pairs of residues

See 'evol <command> -h' for more information on a specific command.
```

Detailed information on a specific application can be obtained by typing the command and application names as **evol search -h**.

Running **prody search** application as follows will search Pfam database for protein families that match the proteins in PDB structure 2w5i:

```
$ evol search 2w5i
```

On Linux, when installing ProDy from source, application scripts are placed into a default folder that is included in PATH[15] environment variable, e.g. `/usr/local/bin/`.

On Windows, installer places the scripts into the `Scripts` folder under Python distribution folder, e.g. `C:\Python27\Scripts`. You may need to add this path to PATH[16] environment variable yourself.

[15] http://matplotlib.sourceforge.net/install/environment_variables_faq.html#envvar-PATH
[16] http://matplotlib.sourceforge.net/install/environment_variables_faq.html#envvar-PATH

CHAPTER

THREE

REFERENCE MANUAL

3.1 Atomic Data

This module defines classes for handling atomic data. Read this page using `help(atomic)`.

3.1.1 Atomic classes

ProDy stores atomic data in instances of *AtomGroup* (page 38) class, which supports multiple coordinate sets, e.g. models from an NMR structure or snapshots from a molecular dynamics trajectory.

Instances of the class can be obtained by parsing a PDB file as follows:

```
In [1]: from prody import *

In [2]: ag = parsePDB('1aar')

In [3]: ag
Out[3]: <AtomGroup: 1aar (1218 atoms)>
```

In addition to *AtomGroup* (page 38) class, following classes that act as pointers provide convenient access subset of data:

- *Selection* (page 100) - Points to an arbitrary subset of atoms. See *Atom Selections* (page 92) and Operations on Selections[17] for usage examples.
- *Segment* (page 86) - Points to atoms that have the same segment name.
- *Chain* (page 53) - Points to atoms in a segment that have the same chain identifier.
- *Residue* (page 80) - Points to atoms in a chain that have the same residue number and insertion code.
- *AtomMap* (page 49) - Points to arbitrary subsets of atoms while allowing for duplicates and missing atoms. Indices of atoms are stored in the order provided by the user.
- *Atom* (page 32) - Points to a single atom
- *Bond* (page 53) - Points to two connected atoms

3.1.2 Atom data fields

Atom Data Fields (page 61) defines an interface for handling data parsed from molecular data files, in particular PDB files. Aforementioned classes offer `get` and `set` functions for manipulating this data. For example, the following prints residue names:

[17] http://prody.csb.pitt.edu/tutorials/prody_tutorial/selection.html#selection-operations

```
In [4]: ag.getResnames()
Out[4]: array(['MET', 'MET', 'MET', ..., 'HOH', 'HOH', 'HOH'], dtype='|S6')
```

3.1.3 Atom flags

Atom Flags (page 64) module defines a way to mark atoms with certain properties, such as atoms that are part of a **protein**. Following example checks whether all atoms of *ag* are protein atoms:

```
In [5]: ag.isprotein
Out[5]: False
```

This indicates that there are some non-protein atoms, probably water atoms. We can easily make a count as follows:

```
In [6]: ag.numAtoms('protein')
Out[6]: 1203

In [7]: ag.numAtoms('hetero')
Out[7]: 15

In [8]: ag.numAtoms('water')
Out[8]: 15
```

3.1.4 Atom selections

Atom Selections (page 92) offer a flexible and powerful way to access subsets of selections and is one of the most important features of ProDy. The details of the selection grammar is described in *Atom Selections* (page 92). Following examples show how to make quick selections using the overloaded `.` operator:

```
In [9]: ag.chain_A  # selects chain A
Out[9]: <Selection: 'chain A' from 1aar (608 atoms)>

In [10]: ag.calpha  # selects alpha carbons
Out[10]: <Selection: 'calpha' from 1aar (152 atoms)>

In [11]: ag.resname_ALA  # selects alanine residues
Out[11]: <Selection: 'resname ALA' from 1aar (20 atoms)>
```

It is also possible to combine selections with `and` and `or` operators:

```
In [12]: ag.chain_A_and_backbone
Out[12]: <Selection: 'chain A and backbone' from 1aar (304 atoms)>

In [13]: ag.acidic_or_basic
Out[13]: <Selection: 'acidic or basic' from 1aar (422 atoms)>
```

Using dot operator will behave like the logical `and` operator:

```
In [14]: ag.chain_A.backbone
Out[14]: <Selection: '(backbone) and (chain A)' from 1aar (304 atoms)>
```

For this to work, the first word following the dot operator must be a flag label or a field name, e.g. `resname`, `name`, `apolar`, `protein`, etc. Underscores will be interpreted as white space, as obvious from the previous examples. The limitation of this is that parentheses, special characters cannot be used.

3.1.5 Functions

The following functions can be used for permanent data storage:

- *loadAtoms()* (page 75)
- *saveAtoms()* (page 75)

The following functions can be used to identify fragments in a group (*AtomGroup* (page 38)) or subset (*Selection* (page 100)) of atoms:

- *findFragments()* (page 74)
- *iterFragments()* (page 74)

The following function can be used to get an *AtomMap* (page 49) that sorts atoms based on a given property:

- *sortAtoms()* (page 75)

The following function can be used check whether a word is reserved because it is used internally by *prody.atomic* (page 29) classes:

- *isReserved()* (page 75)
- *listReservedWords()* (page 75)

3.1.6 Hydrogen Bond Acceptor

This module defines *Acceptor* (page 31) for dealing with acceptor information provided by using *AtomGroup.setAcceptors()* (page 44) method.

class Acceptor(*ag*, *indices*, *acsi=None*)

A pointer class for acceptored atoms. Following built-in functions are customized for this class:

- `len()`[18] returns acceptor length, i.e. *getLength()* (page 31)
- `iter()`[19] yields *Atom* (page 32) instances

getACSIndex()

Returns index of the coordinate set.

getAtomGroup()

Returns atom group.

getAtoms()

Returns acceptored atoms.

getIndices()

Returns indices of acceptored atoms.

getLength()

Returns acceptor length.

getVector()

Returns acceptor vector that originates from the first atom.

setACSIndex(*index*)

Set the coordinate set at *index* active.

3.1.7 Angle

This module defines *Angle* (page 31) for dealing with angle information provided by using *AtomGroup.setAngles()* (page 44) method.

class Angle(*ag*, *indices*, *acsi=None*)

A pointer class for angled atoms. Following built-in functions are customized for this class:

[18] http://docs.python.org/library/functions.html#len
[19] http://docs.python.org/library/functions.html#iter

- `len()`[20] returns angle length, i.e. *`getSize()`* (page 32)
- `iter()`[21] yields *`Atom`* (page 32) instances

getACSIndex()
Returns index of the coordinate set.

getAtomGroup()
Returns atom group.

getAtoms()
Returns angled atoms.

getIndices()
Returns indices of angled atoms.

getSize(*radian=False*)
Returns angle size.

getVectors()
Returns bond vectors that originate from the central atom.

setACSIndex(*index*)
Set the coordinate set at *index* active.

3.1.8 Atom

This module defines classes to handle individual atoms.

class Atom(*ag*, *index*, *acsi*)
A class for handling individual atoms in an *`AtomGroup`* (page 38).

copy()
Returns a copy of atoms (and atomic data) in an *`AtomGroup`* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAltloc()
Return alternate location indicator of the atom. Alternate location indicator can be used in atom selections, e.g. `'altloc A B','altloc _'`.

getAnisou()
Returns a copy of anisotropic temperature factors of the atom from the active coordinate set.

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAnistd()
Return standard deviations for anisotropic temperature factor of the atom.

getAtomGroup()
Returns associated atom group.

getBeta()
Return β-value (temperature factor) of the atom. β-value can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

[20] http://docs.python.org/library/functions.html#len
[21] http://docs.python.org/library/functions.html#iter

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

getCSLabels()
Returns coordinate set labels.

getCharge()
Return partial charge of the atom. Partial charge can be used in atom selections, e.g. `'charge 1','abs(charge) == 1','charge < 0'`.

getChid()
Return chain identifier of the atom. Chain identifier can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

getChindex()
Return chain index of the atom. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Chain index can be used in atom selections, e.g. `'chindex 0'`.

getCoords()
Returns a copy of coordinates of the atom from the active coordinate set.

getCoordsets(*indices=None*)
Returns a copy of coordinate set(s) at given *indices*.

getData(*label*)
Returns a copy of data associated with *label*, if it is present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getElement()
Return element symbol of the atom. Element symbol can be used in atom selections, e.g. `'element C O N'`.

getFlag(*label*)
Returns atom flag.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFragindex()
Return fragment index of the atom. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Fragment index can be used in atom selections, e.g. `'fragindex 0','fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getIcode()
Return insertion code of the atom. Insertion code can be used in atom selections, e.g. `'icode A','icode _'`.

getIndex()
Returns index of the atom.

getIndices()
Returns index of the atom in an `numpy.ndarray`.

getMass()
Return mass of the atom. Mass can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

getMasses()
get the mass atom.

getName()
Return name of the atom. Name can be used in atom selections, e.g. `'name CA CB'`.

getOccupancy()
Return occupancy value of the atom. Occupancy value can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

getRadius()
Return radius of the atom. Radius can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

getResindex()
Return residue index of the atom. Residue indices are assigned to subsets of atoms with distinct sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Residue index can be used in atom selections, e.g. `'resindex 0'`.

getResname()
Return residue name of the atom. Residue name can be used in atom selections, e.g. `'resname ALA GLY'`.

getResnum()
Return residue number of the atom. Residue number can be used in atom selections, e.g. `'resnum 1 2 3', 'resnum 120A 120B', 'resnum 10 to 20', 'resnum 10:20:2', 'resnum < 10'`. Note that *resid* is a synonym for *resnum*.

getSecclass()
Return secondary structure class of the atom. Secondary structure class can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

getSecid()
Return secondary structure identifier of the atom. Secondary structure identifier can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

getSecindex()
Return secondary structure index of the atom. Secondary structure index can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

getSecstr()
Return secondary structure assignment of the atom. Secondary structure assignment can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

getSegindex()
Return segment index of the atom. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Segment index can be used in atom selections, e.g. `'segindex 0'`.

getSegname()
Return segment name of the atom. Segment name can be used in atom selections, e.g. `'segment PROT','segname PROT'`. Note that *segname* is a synonym for *segment*.

getSelstr()
Returns selection string that will select this atom.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerial()
Return serial number (from file) of the atom. Serial number can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

getTitle()
Returns title of the instance.

getType()
Return type of the atom. Type can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterAtoms()
Yield atoms.

iterBonded()
Yield bonded atoms. Use `setBonds()` for setting bonds.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` for setting bonds.

iterCoordsets()
Yield copies of coordinate sets.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

numAtoms(*flag=None*)
Returns number of atoms, or number of atoms with given *flag*.

numBonds()
Returns number of bonds formed by this atom. Bonds must be set first using *AtomGroup.setBonds()* (page 44).

numCoordsets()
Returns number of coordinate sets.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

setAltloc(*data*)
Set alternate location indicator of the atom. Alternate location indicator can be used in atom selections, e.g. `'altloc A B','altloc _'`.

setAnisou(*anisou*)
Set anisotropic temperature factors of the atom in the active coordinate set.

setAnistd(*data*)
Set standard deviations for anisotropic temperature factor of the atom.

setBeta(*data*)
Set β-value (temperature factor) of the atom. β-value can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

setCharge(*data*)
Set partial charge of the atom. Partial charge can be used in atom selections, e.g. `'charge 1', 'abs(charge) == 1','charge < 0'`.

setChid(*data*)
Set chain identifier of the atom. Chain identifier can be used in atom selections, e.g. `'chain A', 'chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

setCoords(*coords*)
Set coordinates of the atom in the active coordinate set.

setData(*label*, *data*)
Update *data* associated with *label*.

Raises **AttributeError**[22] – when *label* is not in use or read-only

setElement(*data*)
Set element symbol of the atom. Element symbol can be used in atom selections, e.g. `'element C O N'`.

setFlag(*label*, *value*)
Update flag associated with *label*.

Raises **AttributeError**[23] – when *label* is not in use or read-only

setIcode(*data*)
Set insertion code of the atom. Insertion code can be used in atom selections, e.g. `'icode A', 'icode _'`.

[22] http://docs.python.org/library/exceptions.html#AttributeError
[23] http://docs.python.org/library/exceptions.html#AttributeError

setMass(*data*)
Set mass of the atom. Mass can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

setName(*data*)
Set name of the atom. Name can be used in atom selections, e.g. `'name CA CB'`.

setOccupancy(*data*)
Set occupancy value of the atom. Occupancy value can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

setRadius(*data*)
Set radius of the atom. Radius can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

setResname(*data*)
Set residue name of the atom. Residue name can be used in atom selections, e.g. `'resname ALA GLY'`.

setResnum(*data*)
Set residue number of the atom. Residue number can be used in atom selections, e.g. `'resnum 1 2 3','resnum 120A 120B','resnum 10 to 20','resnum 10:20:2','resnum < 10'`. Note that *resid* is a synonym for *resnum*.

setSecclass(*data*)
Set secondary structure class of the atom. Secondary structure class can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

setSecid(*data*)
Set secondary structure identifier of the atom. Secondary structure identifier can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

setSecindex(*data*)
Set secondary structure index of the atom. Secondary structure index can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

setSecstr(*data*)
Set secondary structure assignment of the atom. Secondary structure assignment can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

setSegname(*data*)
Set segment name of the atom. Segment name can be used in atom selections, e.g. `'segment PROT','segname PROT'`. Note that *segname* is a synonym for *segment*.

setSerial(*data*)
Set serial number (from file) of the atom. Serial number can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

setType(*data*)
Set type of the atom. Type can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtom()
Returns a TEMPy BioPyAtom or Structure object as appropriate

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

3.1.9 Atom Group

This module defines *AtomGroup* (page 38) class that stores atomic data and multiple coordinate sets in `ndarray` instances.

class AtomGroup(*title='Unnamed'*)
A class for storing and accessing atomic data. The number of atoms of the atom group is inferred at the first set method call from the size of the data array.

Atomic data

All atomic data is stored in `ndarray` instances.

Get and set methods

get methods, e.g. *getResnames()* (page 41), return copies of the data arrays.

set methods, e.g. *setResnums()* (page 46), accept data in `list()` or `ndarray` instances. The length of the list or array must match the number of atoms in the atom group. These methods set attributes of all atoms at once.

Coordinate sets

Atom groups with multiple coordinate sets may have one of these sets as the *active coordinate set*. The active coordinate set may be changed using *setACSIndex()* (page 44) method. *getCoords()* (page 40) returns coordinates from the *active set*.

Atom subsets

To access and modify data associated with a subset of atoms in an atom group, *Selection* (page 100) instances may be used. A *Selection* (page 100) has initially the same coordinate set as the *active coordinate set*, but it may be changed using *Selection.setACSIndex()* (page 103) method.

Customizations

Following built-in functions are customized for this class:

- `len()`[24] returns the number of atoms, i.e. *numAtoms()* (page 43)
- `iter()`[25] yields *Atom* (page 32) instances

Indexing *AtomGroup* (page 38) instances by:

- *int* (`int()`), e.g. `10`, returns an *Atom* (page 32)
- *slice* (`slice()`), e.g. `10:20:2`, returns a *Selection* (page 100)
- *segment name* (`str()`), e.g. `'PROT'`, returns a a *Segment* (page 86)
- *chain identifier* (`str()`), e.g. `'A'`, returns a a *Chain* (page 53)
- *[segment name,] chain identifier, residue number[, insertion code]* (`tuple()`), e.g. `'A', 10` or `'A', 10, 'B'` or `'PROT', 'A', 10, 'B'`, returns a *Residue* (page 80)

[24] http://docs.python.org/library/functions.html#len
[25] http://docs.python.org/library/functions.html#iter

Addition

Addition of two `AtomGroup` (page 38) instances, let's say *A* and *B*, results in a new `AtomGroup` (page 38) instance, say *C*. *C* stores an independent copy of the data of *A* and *B*. If *A* or *B* is missing a certain data type, zero values will be used for that part in *C*. If *A* and *B* has same number of coordinate sets, *C* will have a copy of all coordinate sets, otherwise *C* will have a single coordinate set, which is a copy of of active coordinate sets of *A* and *B*.

addCoordset(*coords*, *label=None*, *anisous=None*)
Add a coordinate set. *coords* argument may be an object with `getCoordsets()` (page 40) method.

copy()
Returns a copy of atoms (and atomic data) in an `AtomGroup` (page 38) instance.

delCoordset(*index*)
Delete a coordinate set from the atom group.

delData(*label*)
Returns data associated with *label* and remove from the instance. If data associated with *label* is not found, return **None**.

delFlags(*label*)
Returns flags associated with *label* and remove from the instance. If flags associated with *label* is not found, return **None**.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAcceptors()
Returns acceptors. Use `setAcceptors()` (page 44) for setting acceptors.

getAltlocs()
Return a copy of alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

getAngles()
Returns angles. Use `setAngles()` (page 44) for setting angles.

getAnisous()
Returns a copy of anisotropic temperature factors from active coordinate set.

getAnistds()
Return a copy of standard deviations for anisotropic temperature factors.

getBetas()
Return a copy of β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

getBonds()
Returns bonds. Use `setBonds()` (page 44) or `inferBonds()` (page 42) for setting bonds.

getBySerial(*serial*, *stop=None*, *step=None*)
Get an atom(s) by *serial* number (range). *serial* must be zero or a positive integer. *stop* may be **None**, or an integer greater than *serial*. `getBySerial(i, j)` will return atoms whose serial numbers are i+1, i+2, ..., j-1. Atom whose serial number is *stop* will be excluded as it would be in indexing a Python `list`[26]. *step* (default is 1) specifies increment. If atoms with matching serial

[26] http://docs.python.org/library/stdtypes.html#list

numbers are not found, **None** will be returned.

getCSLabels()
Returns coordinate set labels.

getCharges()
Return a copy of partial charges. Partial charges can be used in atom selections, e.g. `'charge 1','abs(charge) == 1','charge < 0'`.

getChids()
Return a copy of chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

getChindices()
Return a copy of chain indices. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

getCoords()
Returns a copy of coordinates from active coordinate set.

getCoordsets(*indices=None*)
Returns a copy of coordinate set(s) at given *indices*. *indices* may be an integer, a list of integers, or **None** meaning all coordinate sets.

getCrossterms()
Returns crossterms. Use *setCrossterms()* (page 45) for setting crossterms.

getData(*label*)
Returns a copy of the data array associated with *label*, or **None** if such data is not present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getDihedrals()
Returns dihedrals. Use *setDihedrals()* (page 45) for setting dihedrals.

getDonors()
Returns donors. Use *setDonors()* (page 45) for setting donors.

getElements()
Return a copy of element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFlags(*label*)
Returns a copy of atom flags for given *label*, or **None** when flags for *label* is not set.

getFragindices()
Return a copy of fragment indices. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by

one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Fragment indices can be used in atom selections, e.g. `'fragindex 0','fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getHierView(***kwargs*)
Returns a hierarchical view of the atom group.

getIcodes()
Return a copy of insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

getImpropers()
Returns impropers. Use *setImpropers()* (page 45) for setting impropers.

getMasses()
Return a copy of masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

getNBExclusions()
Returns nbexclusions. Use *setNBExclusions()* (page 46) for setting nbexclusions.

getNames()
Return a copy of names. Names can be used in atom selections, e.g. `'name CA CB'`.

getOccupancies()
Return a copy of occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

getRadii()
Return a copy of radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

getResindices()
Return a copy of residue indices. Residue indices are assigned to subsets of atoms with distinct sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Residue indices can be used in atom selections, e.g. `'resindex 0'`.

getResnames()
Return a copy of residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

getResnums()
Return a copy of residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3'`, `'resnum 120A 120B'`, `'resnum 10 to 20'`, `'resnum 10:20:2'`, `'resnum < 10'`. Note that *resid* is a synonym for *resnum*.

getSecclasses()
Return a copy of secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

getSecids()
Return a copy of secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

getSecindices()
Return a copy of secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

getSecstrs()
Return a copy of secondary structure assignments. Secondary structure assignments can be used

in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

getSegindices()
Return a copy of segment indices. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegnames()
Return a copy of segment names. Segment names can be used in atom selections, e.g. `'segment PROT','segname PROT'`. Note that *segname* is a synonym for *segment*.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerials()
Return a copy of serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

getTitle()
Returns title of the instance.

getTypes()
Return a copy of types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

inferBonds(*max_bond=1.6*, *min_bond=0*, *set_bonds=True*)
Returns bonds based on distances **max_bond** and **min_bond**.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors. Use *setAcceptors()* (page 44) for setting acceptors.

iterAngles()
Yield angles. Use *setAngles()* (page 44) for setting angles.

iterAtoms()
Yield atom instances.

iterBonds()
Yield bonds. Use *setBonds()* (page 44) or *inferBonds* for setting bonds.

iterChains()
Iterate over chains.

iterCoordsets()
Iterate over coordinate sets by returning a copy of each coordinate set.

iterCrossterms()
Yield crossterms. Use *setCrossterms()* (page 45) for setting crossterms.

iterDihedrals()
Yield dihedrals. Use *setDihedrals()* (page 45) for setting dihedrals.

iterDonors()
Yield donors. Use *setDonors()* (page 45) for setting donors.

iterFragments()
Yield connected atom subsets as *Selection* (page 100) instances.

iterImpropers()
Yield impropers. Use *setImpropers()* (page 45) for setting impropers.

iterNBExclusions()
Yield nbexclusions. Use *setNBExclusions()* (page 46) for setting nbexclusions.

iterResidues()
Iterate over residues.

iterSegments()
Iterate over segments.

numAcceptors()
Returns number of acceptors. Use *setAcceptors()* (page 44) for setting acceptors.

numAngles()
Returns number of angles. Use *setAngles()* (page 44) for setting angles.

numAtoms(*flag=None*)
Returns number of atoms, or number of atoms with given *flag*.

numBonds()
Returns number of bonds. Use *setBonds()* (page 44) or *inferBonds()* (page 42) for setting bonds.

numBytes(*all=False*)
Returns number of bytes used by atomic data arrays, such as coordinate, flag, and attribute arrays. If *all* is **True**, internal arrays for indexing hierarchical views, bonds, and fragments will also be included. Note that memory usage of Python objects is not taken into account and that this may change in the future.

numChains()
Returns number of chains.

numCoordsets()
Returns number of coordinate sets.

numCrossterms()
Returns number of crossterms. Use *setCrossterms()* (page 45) for setting crossterms.

numDihedrals()
Returns number of dihedrals. Use *setDihedrals()* (page 45) for setting dihedrals.

numDonors()
Returns number of donors. Use *setDonors()* (page 45) for setting donors.

numFragments()
Returns number of connected atom subsets.

numImpropers()
Returns number of impropers. Use *setImpropers()* (page 45) for setting impropers.

numNBExclusions()
Returns number of nbexclusions. Use *setNBExclusions()* (page 46) for setting nbexclusions.

numResidues()
Returns number of residues.

numSegments()
Returns number of segments.

select (*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex (*index*)
Set the coordinate set at *index* active.

setACSLabel (*label*)
Set active coordinate set label.

setAcceptors (*acceptors*)
Set covalent acceptors between atoms. *acceptors* must be a list or an array of pairs of indices. All acceptors must be set at once. Acceptoring information can be used to make atom selections, e.g. `"acceptored to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of acceptors will be generated and stored with label *numacceptors*. This can be used in atom selections, e.g. `'numacceptors 0'` can be used to select ions in a system. If *acceptors* is empty or **None**, then all acceptors will be removed for this *AtomGroup* (page 38).

setAltlocs (*data*)
Set alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

setAngles (*angles*)
Set covalent angles between atoms. *angles* must be a list or an array of triplets of indices. All angles must be set at once. Angle information can be used to make atom selections, e.g. `"angle to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of angles will be generated and stored with label *numangles*. This can be used in atom selections, e.g. `'numangles 0'` can be used to select ions in a system.

setAnisous (*anisous*, *label=''*)
Set anisotropic temperature factors of atoms. *anisous* may be any array like object or an object instance with *getAnisous()* (page 39) method. If the shape of anisou array is `(n_csets > 1, n_atoms, 3)`, it will replace all coordinate sets and the active coordinate set index will reset to zero. This situation can be avoided using *addCoordset()* (page 39). If shape of *coords* is `(n_atoms, 3)` or `(1, n_atoms, 3)`, it will replace the active coordinate set. *label* argument may be used to label coordinate set(s). *label* may be a string or a list of strings length equal to the number of coordinate sets.

setAnistds (*data*)
Set standard deviations for anisotropic temperature factors.

setBetas (*data*)
Set β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

setBonds (*bonds*)
Set covalent bonds between atoms. *bonds* must be a list or an array of pairs of indices. All bonds must be set at once. Bonding information can be used to make atom selections, e.g. `"bonded to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of bonds will be generated and stored with label *numbonds*. This can be used in atom selections, e.g. `'numbonds 0'` can be used to select ions in a system. If *bonds* is empty or **None**, then all bonds will be removed for this *AtomGroup* (page 38).

setCSLabels (*labels*)
Set coordinate set labels. *labels* must be a list of strings.

setCharges (*data*)
Set partial charges. Partial charges can be used in atom selections, e.g. `'charge 1'`, `'abs(charge) == 1','charge < 0'`.

setChids(*data*)
Set chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

setCoords(*coords*, *label=''*)
Set coordinates of atoms. *coords* may be any array like object or an object instance with *getCoords()* (page 40) method. If the shape of coordinate array is `(n_csets > 1, n_atoms, 3)`, it will replace all coordinate sets and the active coordinate set index will reset to zero. This situation can be avoided using *addCoordset()* (page 39). If shape of *coords* is `(n_atoms, 3)` or `(1, n_atoms, 3)`, it will replace the active coordinate set. *label* argument may be used to label coordinate set(s). *label* may be a string or a list of strings length equal to the number of coordinate sets.

setCrossterms(*crossterms*)
Set covalent crossterms between atoms. *crossterms* must be a list or an array of triplets of indices. All crossterms must be set at once. Crossterm information can be used to make atom selections, e.g. `"crossterm to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of crossterms will be generated and stored with label *numcrossterms*. This can be used in atom selections, e.g. `'numcrossterms 0'` can be used to select ions in a system.

setData(*label*, *data*)
Store atomic *data* under *label*, which must:

- start with a letter
- contain only alphanumeric characters and underscore
- not be a reserved word (see *listReservedWords()* (page 75))

data must be a `list()` or a `ndarray` and its length must be equal to the number of atoms. If the dimension of the *data* array is 1, i.e. `data.ndim==1`, *label* may be used to make atom selections, e.g. `"label 1 to 10"` or `"label C1 C2"`. Note that, if data with *label* is present, it will be overwritten.

setDihedrals(*dihedrals*)
Set covalent dihedrals between atoms. *dihedrals* must be a list or an array of triplets of indices. All dihedrals must be set at once. Dihedral information can be used to make atom selections, e.g. `"dihedral to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of dihedrals will be generated and stored with label *numdihedrals*. This can be used in atom selections, e.g. `'numdihedrals 0'` can be used to select ions in a system.

setDonors(*donors*)
Set covalent donors between atoms. *donors* must be a list or an array of pairs of indices. All donors must be set at once. Donoring information can be used to make atom selections, e.g. `"donored to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of donors will be generated and stored with label *numdonors*. This can be used in atom selections, e.g. `'numdonors 0'` can be used to select ions in a system. If *donors* is empty or **None**, then all donors will be removed for this *AtomGroup* (page 38).

setElements(*data*)
Set element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

setFlags(*label*, *flags*)
Set atom *flags* for *label*.

setIcodes(*data*)
Set insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

setImpropers(*impropers*)
Set covalent impropers between atoms. *impropers* must be a list or an array of triplets of indices. All impropers must be set at once. Improper information can be used to make atom selections, e.g. `"improper to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of impropers will be generated and stored with label *numimpropers*. This can be used in atom selections, e.g. `'numimpropers 0'` can be used to select ions in a system.

setMasses(*data*)
Set masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

setNBExclusions(*nbexclusions*)
Set nbexclusions between atoms. *nbexclusions* must be a list or an array of pairs of indices. All nbexclusions must be set at once. Acceptoring information can be used to make atom selections, e.g. `"nbexclusioned to index 1"`. See *select* (page 92) module documentation for details. Also, a data array with number of nbexclusions will be generated and stored with label *numnbexclusions*. This can be used in atom selections, e.g. `'numnbexclusions 0'` can be used to select ions in a system. If *nbexclusions* is empty or **None**, then all nbexclusions will be removed for this *AtomGroup* (page 38).

setNames(*data*)
Set names. Names can be used in atom selections, e.g. `'name CA CB'`.

setOccupancies(*data*)
Set occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1'`, `'occupancy > 0'`.

setRadii(*data*)
Set radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

setResnames(*data*)
Set residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

setResnums(*data*)
Set residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3','resnum 120A 120B','resnum 10 to 20','resnum 10:20:2','resnum < 10'`. Note that *resid* is a synonym for *resnum*.

setSecclasses(*data*)
Set secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

setSecids(*data*)
Set secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

setSecindices(*data*)
Set secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

setSecstrs(*data*)
Set secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

setSegnames(*data*)
Set segment names. Segment names can be used in atom selections, e.g. `'segment PROT'`, `'segname PROT'`. Note that *segname* is a synonym for *segment*.

setSerials(*data*)
Set serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1`

2 3','serial 1 to 10','serial 1:10:2','serial < 10'.

setTitle(*title*)
Set title of the instance.

setTypes(*data*)
Set types. Types can be used in atom selections, e.g. 'type CT1 CT2 CT3'.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a Bio.PDB.Structure object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a TEMPy.protein.prot_rep_biopy.Atom or list of them as appropriate

toTEMPyStructure()
Returns a protein.prot_rep_biopy.Structure object

3.1.10 Atomic Base

This module defines base class *Atomic* (page 47) that all other *atomic* (page 29) classes are derived from.

class Atomic
Base class for all atomic classes that can be used for type checking.

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getTitle()
Returns title of the instance.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a Bio.PDB.Structure object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a TEMPy.protein.prot_rep_biopy.Atom or list of them as appropriate

toTEMPyStructure()
Returns a protein.prot_rep_biopy.Structure object

3.1.11 Atom Map

This module defines *AtomMap* (page 49) class that allows for pointing atoms in arbitrary order.

How AtomMap works

AtomMap (page 49) class adds great flexibility to manipulating atomic data.

First let's see how an instance of *Selection* (page 100) (*Chain* (page 53), or *Residue* (page 80)) works. Below table shows indices for a selection of atoms in an *AtomGroup* (page 38) and values returned when *getNames()* (page 101), *getResnames()* (page 102) and *getResnums()* (page 102) methods are called.

Table 3.1: **Atom Subset**

Indices	Names	Resnames	Resnums
0	N	PHE	1
1	CA	PHE	1
2	C	PHE	1
3	O	PHE	1
4	CB	PHE	1
5	CG	PHE	1
6	CD1	PHE	1
7	CD2	PHE	1
8	CE1	PHE	1
9	CE2	PHE	1
10	CZ	PHE	1

Selection (page 100) instances keep indices ordered and do not allow duplicate values, hence their use is limited. In an *AtomMap* (page 49), indices do not need to be sorted, duplicate indices may exist, even "DUMMY" atoms are allowed.

Let's say we instantiate the following AtomMap:

```
amap = AtomMap(atomgroup, indices=[0, 1, 3, 8, 8, 9, 10],
               mapping=[5, 6, 7, 0, 1, 2, 3])
```

The size of the *AtomMap* (page 49) based on this mapping is 8, since the larger mapping is 7.

Calling the same functions for this AtomMap instance would result in the following:

Table 3.2: **Atom Map**

Mapping	Indices	Names	Resnames	Resnums	MappedFlags	DummyFlags
0	8	CE1	PHE	1	1	0
1	8	CE1	PHE	1	1	0
2	9	CE2	PHE	1	1	0
3	10	CZ	PHE	1	1	0
4				0	0	1
5	0	N	PHE	1	1	0
6	1	CA	PHE	1	1	0
7	3	O	PHE	1	1	0

For unmapped atoms, numeric attributes are set to 0, others to empty string, i.e. "".

See also:

AtomMap (page 49) are used by *proteins* (page 222) module functions that match or map protein chains. Heterogeneous X-ray Structures[27] and Multimeric Structures[28] examples that make use of these functions and *AtomMap* (page 49) class.

class AtomMap(*ag*, *indices*, *acsi=None*, ***kwargs*)

A class for mapping atomic data.

Instantiate an atom map.

Parameters

- **ag** – AtomGroup instance from which atoms are mapped
- **indices** – indices of mapped atoms
- **acsi** – active coordinate set index, defaults is that of *ag*
- **mapping** – mapping of atom *indices*
- **dummies** – dummy atom indices
- **title** – title of the instance, default is 'Unknown'

mapping and *dummies* arrays must be provided together. Length of *mapping* must be equal to length of *indices*. Elements of *mapping* must be an ordered in ascending order. When dummy atoms are present, number of atoms is the sum of lengths of *mapping* and *dummies*.

Following built-in functions are customized for this class:

- `len()`[29] returns the number of atoms in the instance.
- `iter()`[30] yields *Atom* (page 32) instances.
- Indexing returns an *Atom* (page 32) or an *AtomMap* (page 49) instance depending on the type and value of the index.

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAltlocs()
Return a copy of alternate location indicators. Entries for dummy atoms will be `''`.

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAnistds()
Return a copy of standard deviations for anisotropic temperature factors. Entries for dummy atoms will be `0.0`.

getAtomGroup()
Returns associated atom group.

getBetas()
Return a copy of β-values (or temperature factors). Entries for dummy atoms will be `0.0`.

[27] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray.html#pca-xray
[28] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/dimer.html#pca-dimer
[29] http://docs.python.org/library/functions.html#len
[30] http://docs.python.org/library/functions.html#iter

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

getCSLabels()
Returns coordinate set labels.

getCharges()
Return a copy of partial charges. Entries for dummy atoms will be `0.0`.

getChids()
Return a copy of chain identifiers. Entries for dummy atoms will be `''`.

getChindices()
Return a copy of chain indices. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Entries for dummy atoms will be `0`.

getCoords()
Returns a copy of coordinates from the active coordinate set.

getCoordsets(*indices=None*)
Returns coordinate set(s) at given *indices*, which may be an integer or a list/array of integers.

getData(*label*)
Returns a copy of data associated with *label*, if it is present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getElements()
Return a copy of element symbols. Entries for dummy atoms will be `''`.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFlags(*label*)
Returns a copy of atom flags for given *label*, or **None** when flags for *label* is not set.

getFragindices()
Return a copy of fragment indices. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Entries for dummy atoms will be `0`.

getHierView(***kwargs*)
Returns a hierarchical view of the this chain.

getIcodes()
Return a copy of insertion codes. Entries for dummy atoms will be `''`.

getIndices()
Returns a copy of indices of atoms, with maximum integer value dummies.

getMapping()
Returns a copy of mapping of indices.

getMasses()
Return a copy of masses. Entries for dummy atoms will be `0.0`.

getNames()
Return a copy of names. Entries for dummy atoms will be `''`.

getOccupancies()
Return a copy of occupancy values. Entries for dummy atoms will be `0.0`.

getRadii()
Return a copy of radii. Entries for dummy atoms will be `0.0`.

getResindices()
Return a copy of residue indices. Residue indices are assigned to subsets of atoms with distinct sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Entries for dummy atoms will be `0`.

getResnames()
Return a copy of residue names. Entries for dummy atoms will be `''`.

getResnums()
Return a copy of residue numbers. Entries for dummy atoms will be `0`.

getSecclasses()
Return a copy of secondary structure classs. Entries for dummy atoms will be `0`.

getSecids()
Return a copy of secondary structure identifiers. Entries for dummy atoms will be `''`.

getSecindices()
Return a copy of secondary structure indexs. Entries for dummy atoms will be `0`.

getSecstrs()
Return a copy of secondary structure assignments. Entries for dummy atoms will be `''`.

getSegindices()
Return a copy of segment indices. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Entries for dummy atoms will be `0`.

getSegnames()
Return a copy of segment names. Entries for dummy atoms will be `''`.

getSelstr()
Returns selection string that selects mapped atoms.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerials()
Return a copy of serial numbers (from file). Entries for dummy atoms will be `0`.

getTitle()
Returns title of the instance.

getTypes()
Return a copy of types. Entries for dummy atoms will be `''`.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterAtoms()
Yield atoms, and **None** for dummies.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` or `inferBonds()` for setting bonds.

iterCoordsets()
Yield copies of coordinate sets.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

numAtoms(*flag=None*)
Returns number of atoms.

numBonds()
Returns number of bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

numCoordsets()
Returns number of coordinate sets.

numDummies()
Returns number of dummy atoms.

numMapped()
Returns number of mapped atoms.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

setCoords(*coords*)
Set coordinates of atoms in the active coordinate set.

setTitle(*title*)
Set title of the instance.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

3.1.12 Bond

This module defines *Bond* (page 53) for dealing with bond information provided by using the *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42) method.

class Bond(*ag*, *indices*, *acsi=None*)
A pointer class for bonded atoms. Following built-in functions are customized for this class:

- `len()`[31] returns bond length, i.e. *getLength()* (page 53)
- `iter()`[32] yields *Atom* (page 32) instances

getACSIndex()
Returns index of the coordinate set.

getAtomGroup()
Returns atom group.

getAtoms()
Returns bonded atoms.

getIndices()
Returns indices of bonded atoms.

getLength()
Returns bond length.

getVector()
Returns bond vector that originates from the first atom.

setACSIndex(*index*)
Set the coordinate set at *index* active.

3.1.13 Chain

This module defines classes for handling polypeptide/nucleic acid chains.

class Chain(*ag*, *indices*, *hv*, *acsi=None*, ***kwargs*)
Instances of this class point to atoms with same chain identifiers and are generated by *HierView* (page 76) class. Following built-in functions are customized for this class:

- `len()`[33] returns the number of residues in the chain

[31] http://docs.python.org/library/functions.html#len
[32] http://docs.python.org/library/functions.html#iter
[33] http://docs.python.org/library/functions.html#len

- `iter()`[34] yields *Residue* (page 80) instances

Indexing *Chain* (page 53) instances by:

- *residue number [, insertion code]* (`tuple()`), e.g. `10` or `10, "B"`, returns a *Residue* (page 80)
- *slice* (`slice()`), e.g, `10:20`, returns a list of *Residue* (page 80) instances

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAltlocs()
Return a copy of alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAnistds()
Return a copy of standard deviations for anisotropic temperature factors.

getAtomGroup()
Returns associated atom group.

getBetas()
Return a copy of β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

getCSLabels()
Returns coordinate set labels.

getCharges()
Return a copy of partial charges. Partial charges can be used in atom selections, e.g. `'charge 1','abs(charge) == 1','charge < 0'`.

getChid()
Returns chain identifier.

getChids()
Return a copy of chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

getChindex()
Returns chain index.

getChindices()
Return a copy of chain indices. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

getCoords()
Returns a copy of coordinates from the active coordinate set.

[34] http://docs.python.org/library/functions.html#iter

getCoordsets(*indices=None*)
Returns coordinate set(s) at given *indices*, which may be an integer or a list/array of integers.

getData(*label*)
Returns a copy of data associated with *label*, if it is present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getElements()
Return a copy of element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFlags(*label*)
Returns a copy of atom flags for given *label*, or **None** when flags for *label* is not set.

getFragindices()
Return a copy of fragment indices. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Fragment indices can be used in atom selections, e.g. `'fragindex 0','fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getHierView(***kwargs*)
Returns a hierarchical view of the this chain.

getIcodes()
Return a copy of insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

getIndices()
Returns a copy of the indices of atoms.

getMasses()
Return a copy of masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

getNames()
Return a copy of names. Names can be used in atom selections, e.g. `'name CA CB'`.

getOccupancies()
Return a copy of occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

getRadii()
Return a copy of radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

getResidue(*resnum*, *icode=None*)
Returns residue with number *resnum* and insertion code *icode*.

getResindices()
Return a copy of residue indices. Residue indices are assigned to subsets of atoms with distinct

sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Residue indices can be used in atom selections, e.g. `'resindex 0'`.

getResnames()
Return a copy of residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

getResnums()
Return a copy of residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3'`, `'resnum 120A 120B'`, `'resnum 10 to 20'`, `'resnum 10:20:2'`, `'resnum < 10'`. Note that *resid* is a synonym for *resnum*.

getSecclasses()
Return a copy of secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

getSecids()
Return a copy of secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

getSecindices()
Return a copy of secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

getSecstrs()
Return a copy of secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

getSegindices()
Return a copy of segment indices. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegment()
Returns segment of the chain.

getSegname()
Returns segment name.

getSegnames()
Return a copy of segment names. Segment names can be used in atom selections, e.g. `'segment PROT','segname PROT'`. Note that *segname* is a synonym for *segment*.

getSelstr()
Returns selection string that selects atoms in this chain.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids in the chain. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerials()
Return a copy of serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

getTitle()
Returns title of the instance.

getTypes()
Return a copy of types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterAtoms()
Yield atoms.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` or `inferBonds()` for setting bonds.

iterCoordsets()
Yield copies of coordinate sets.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

iterResidues()
Yield residues.

numAtoms(*flag=None*)
Returns number of atoms, or number of atoms with given *flag*.

numBonds()
Returns number of bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

numCoordsets()
Returns number of coordinate sets.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

setAltlocs(*data*)
Set alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

setAnisous(*anisous*)
Set anisotropic temperature factors in the active coordinate set.

setAnistds(*data*)
Set standard deviations for anisotropic temperature factors.

setBetas(*data*)
Set β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

setCharges(*data*)
Set partial charges. Partial charges can be used in atom selections, e.g. `'charge 1', 'abs(charge) == 1','charge < 0'`.

setChid(*chid*)
Set chain identifier.

setChids(*data*)
Set chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

setCoords(*coords*)
Set coordinates in the active coordinate set.

setData(*label*, *data*)
Update *data* associated with *label*.

Raises **AttributeError**[35] – when *label* is not in use or read-only

setElements(*data*)
Set element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

setFlags(*label*, *value*)
Update flag associated with *label*.

Raises **AttributeError**[36] – when *label* is not in use or read-only

setIcodes(*data*)
Set insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

setMasses(*data*)
Set masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

setNames(*data*)
Set names. Names can be used in atom selections, e.g. `'name CA CB'`.

setOccupancies(*data*)
Set occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1', 'occupancy > 0'`.

setRadii(*data*)
Set radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

setResnames(*data*)
Set residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

[35] http://docs.python.org/library/exceptions.html#AttributeError
[36] http://docs.python.org/library/exceptions.html#AttributeError

setResnums(*data*)
Set residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3','resnum 120A 120B','resnum 10 to 20','resnum 10:20:2','resnum < 10'`. Note that *resid* is a synonym for *resnum*.

setSecclasses(*data*)
Set secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

setSecids(*data*)
Set secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

setSecindices(*data*)
Set secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

setSecstrs(*data*)
Set secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

setSegnames(*data*)
Set segment names. Segment names can be used in atom selections, e.g. `'segment PROT'`, `'segname PROT'`. Note that *segname* is a synonym for *segment*.

setSerials(*data*)
Set serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

setTypes(*data*)
Set types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

3.1.14 Cross-term

This module defines *Crossterm* (page 59) for dealing with crossterm information provided by using *AtomGroup.setCrossterms()* (page 45) method.

class Crossterm(*ag*, *indices*, *acsi=None*)
A pointer class for crosstermd atoms. Following built-in functions are customized for this class:

- `len()`[37] returns crossterm length, i.e. *getSize()* (page 60)

[37] http://docs.python.org/library/functions.html#len

- `iter()`[38] yields *Atom* (page 32) instances

getACSIndex()
Returns index of the coordinate set.

getAtomGroup()
Returns atom group.

getAtoms()
Returns crosstermd atoms.

getIndices()
Returns indices of crosstermd atoms.

getSize(*radian=False*)
Returns crossterm size.

getVectors()
Returns bond vectors that originate from the central atom.

setACSIndex(*index*)
Set the coordinate set at *index* active.

3.1.15 Dihedral

This module defines *Dihedral* (page 60) for dealing with dihedral information provided by using *AtomGroup.setDihedrals()* (page 45) method.

class Dihedral(*ag*, *indices*, *acsi=None*)
A pointer class for dihedrald atoms. Following built-in functions are customized for this class:

- `len()`[39] returns dihedral length, i.e. *getSize()* (page 60)
- `iter()`[40] yields *Atom* (page 32) instances

getACSIndex()
Returns index of the coordinate set.

getAtomGroup()
Returns atom group.

getAtoms()
Returns dihedrald atoms.

getIndices()
Returns indices of dihedrald atoms.

getSize(*radian=False*)
Returns dihedral size.

getVectors()
Returns bond vectors that originate from the central atom.

setACSIndex(*index*)
Set the coordinate set at *index* active.

[38] http://docs.python.org/library/functions.html#iter
[39] http://docs.python.org/library/functions.html#len
[40] http://docs.python.org/library/functions.html#iter

3.1.16 Hydrogen Bond Donor

This module defines *Donor* (page 61) for dealing with donor information provided by using *AtomGroup.setDonors()* (page 45) method.

class Donor(*ag*, *indices*, *acsi=None*)

A pointer class for donored atoms. Following built-in functions are customized for this class:

- `len()`[41] returns donor length, i.e. *getLength()* (page 61)
- `iter()`[42] yields *Atom* (page 32) instances

getACSIndex()

Returns index of the coordinate set.

getAtomGroup()

Returns atom group.

getAtoms()

Returns donored atoms.

getIndices()

Returns indices of donored atoms.

getLength()

Returns donor length.

getVector()

Returns donor vector that originates from the first atom.

setACSIndex(*index*)

Set the coordinate set at *index* active.

3.1.17 Atom Data Fields

This module defines atomic data fields. You can read this page in interactive sessions using `help(fields)`.

Data parsed from PDB and other supported files for these fields are stored in *AtomGroup* (page 38) instances. Available data fields are listed in the table below. `Atomic` classes, such as *Selection* (page 100), offer `get` and `set` for handling parsed data:

Many of these data fields can be used to make *Atom Selections* (page 92). Following table lists definitions of fields and selection examples. Note that fields noted as *read only* do not have a `set` method.

altloc alternate location indicator

E.g.: `'altloc A B','altloc _'`

beta β-value (temperature factor)

E.g.: `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`

chain, chid chain identifier

E.g.: `'chain A','chid A B C','chain _'`

charge partial charge

E.g.: `'charge 1','abs(charge) == 1','charge < 0'`

[41] http://docs.python.org/library/functions.html#len
[42] http://docs.python.org/library/functions.html#iter

chindex chain index *(read only)*

E.g.: `'chindex 0'`

element element symbol

E.g.: `'element C O N'`

fragindex, fragment fragment index *(read only)*

E.g.: `'fragindex 0','fragment 1'`

icode insertion code

E.g.: `'icode A','icode _'`

mass mass

E.g.: `'12 <= mass <= 13.5'`

name name

E.g.: `'name CA CB'`

numbonds number of bonds *(read only)*

E.g.: `'numbonds 0','numbonds 1'`

occupancy occupancy value

E.g.: `'occupancy 1','occupancy > 0'`

radius radius

E.g.: `'radii < 1.5','radii ** 2 < 2.3'`

resindex residue index *(read only)*

E.g.: `'resindex 0'`

resname residue name

E.g.: `'resname ALA GLY'`

resnum, resid residue number

E.g.: `'resnum 1 2 3', 'resnum 120A 120B', 'resnum 10 to 20', 'resnum 10:20:2', 'resnum < 10'`

secclass secondary structure class

E.g.: `'secclass 2','secclass -1'`

secid secondary structure identifier

E.g.: `'secid A B','secid 1 2'`

secindex secondary structure index

E.g.: `'s','e','c','i','n','d','e','x',' ','2'`

secondary, secstr secondary structure assignment

E.g.: `'secondary H E','secstr H E'`

segindex segment index *(read only)*

E.g.: `'segindex 0'`

segment, segname segment name

E.g.: `'segment PROT','segname PROT'`

serial serial number (from file)

E.g.: `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`

siguij standard deviations for anisotropic temperature factor

type type

E.g.: `'type CT1 CT2 CT3'`

class **Field**(*name*, *dtype*, ***kwargs*)
Atomic data field.

getDocstr(*meth*, *plural=True*, *selex=True*)
Returns documentation string for the field.

call
list of *AtomGroup* (page 38) methods to call when `getMethod` is called

depr
deprecated method name

depr_pl
deprecated method name in plural form

desc
description of data field, used in documentation

doc
internal variable name used as key for *AtomGroup* (page 38) `_data`

doc_pl
plural form for documentation

dtype
data type (primitive Python types)

flags
True when there are flags associated with the data field

meth
atomic get/set method name

meth_pl
get/set method name in plural form

name
data field name used in atom selections

ndim
expected dimension of the data array

none
AtomGroup (page 38) attributes to be set None, when `setMethod` is called

private
define only _getMethod for *AtomGroup* (page 38) to be used by *Select* (page 99) class

readonly
read-only attribute without a set method

selstr
list of selection string examples

synonym
synonym used in atom selections

3.1.18 Atom Flags

This module defines atom flags that are used in *Atom Selections* (page 92). You can read this page in interactive sessions using `help(flags)`.

Flag labels can be used in atom selections:

```
In [1]: from prody import *

In [2]: p = parsePDB('1ubi')

In [3]: p.select('protein')
Out[3]: <Selection: 'protein' from 1ubi (602 atoms)>
```

Flag labels can be combined with dot operator as follows to make selections:

```
In [4]: p.protein
Out[4]: <Selection: 'protein' from 1ubi (602 atoms)>

In [5]: p.protein.acidic  # selects acidic residues
Out[5]: <Selection: '(acidic) and (protein)' from 1ubi (94 atoms)>
```

Flag labels can be prefixed with `'is'` to check whether all atoms in an *Atomic* (page 47) instance are flagged the same way:

```
In [6]: p.protein.ishetero
Out[6]: False

In [7]: p.water.ishetero
Out[7]: True
```

Flag labels can also be used to make quick atom counts:

```
In [8]: p.numAtoms()
Out[8]: 683

In [9]: p.numAtoms('protein')
Out[9]: 602

In [10]: p.numAtoms('water')
Out[10]: 81
```

Protein

protein, aminoacid indicates the twenty standard amino acids (*stdaa*) and some non-standard amino acids (*nonstdaa*) described below. Residue must also have an atom named `'CA'` in addition to having a qualifying residue name.

stdaa indicates the standard amino acid residues: ALA[43], ARG[44], ASN[45], ASP[46], CYS[47], GLN[48], GLU[49], GLY[50], HIS[51], ILE[52], LEU[53], LYS[54], MET[55], PHE[56], PRO[57], SER[58], THR[59], TRP[60], TYR[61], and VAL[62]

nonstdaa indicates one of the following residues:

ASX[63] (B)	asparagine or aspartic acid
GLX[64] (Z)	glutamine or glutamic acid
CSO[65] (C)	S-hydroxycysteine
HIP[66] (H)	ND1-phosphohistidine
HSD (H)	prototropic tautomer of histidine, H on ND1 (CHARMM)
HSE (H)	prototropic tautomer of histidine, H on NE2 (CHARMM)
HSP (H)	protonated histidine
MSE[67]	selenomethionine
SEC[68] (U)	selenocysteine
SEP[69] (S)	phosphoserine
TPO[70] (T)	phosphothreonine
PTR[71] (Y)	O-phosphotyrosine
XLE (J)	leucine or isoleucine
XAA (X)	unspecified or unknown

You can modify the list of non-standard amino acids using `addNonstdAminoacid()` (page 73), `delNonstdAminoacid()` (page 74), and `listNonstdAAProps()` (page 73).

calpha, ca Cα atoms of *protein* residues, same as selection `'name CA and protein'`

backbone, bb non-hydrogen backbone atoms of *protein* residues, same as selection `'name CA C O N and protein'`

backbonefull, bbfull backbone atoms of *protein* residues, same as selection `'name CA C O N H H1 H2 H3 OXT and protein'`

[43] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ALA
[44] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ARG
[45] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ASN
[46] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ASP
[47] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CYS
[48] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLN
[49] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLU
[50] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLY
[51] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HIS
[52] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ILE
[53] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=LEU
[54] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=LYS
[55] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MET
[56] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PHE
[57] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PRO
[58] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SER
[59] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=THR
[60] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TRP
[61] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TYR
[62] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=VAL
[63] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ASX
[64] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLX
[65] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CSO
[66] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HIP
[67] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MSE
[68] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SEC
[69] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SEP
[70] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TPO
[71] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PTR

sidechain, sc side-chain atoms of *protein* residues, same as selection `'protein and not backbonefull'`

acidic residues ASP, GLU, HSP, PHD, PTR, SEP, TPO

acyclic residues ALA, ARG, ASN, ASP, ASX, CME, CSO, CYS, CYX, GLN, GLU, GLX, LY, ILE, LEU, LYS, MET, MSE, PHD, SEC, SEP, SER, THR, TPO, VAL, LE

aliphatic residues ALA, GLY, ILE, LEU, PRO, VAL, XLE

aromatic residues HIS, PHE, PTR, TRP, TYR

basic residues ARG, HID, HIE, HIP, HIS, HSD, HSE, LYS

buried residues ALA, CME, CYS, CYX, ILE, LEU, MET, MSE, PHE, SEC, TRP, VAL, LE

charged residues ARG, ASP, GLU, HIS, LYS

cyclic residues HID, HIE, HIP, HIS, HSD, HSE, HSP, PHE, PRO, PTR, TRP, TYR

hydrophobic residues ALA, ILE, LEU, MET, PHE, PRO, TRP, VAL, XLE

large residues ARG, CME, GLN, GLU, GLX, HID, HIE, HIP, HIS, HSD, HSE, HSP, LE, LEU, LYS, MET, MSE, PHD, PHE, PTR, SEP, TPO, TRP, TYR, XLE

medium residues ASN, ASP, ASX, CSO, CYS, CYX, PRO, SEC, THR, VAL

neutral residues ALA, ASN, CME, CSO, CYS, CYX, GLN, GLY, ILE, LEU, MET, MSE, HE, PRO, SEC, SER, THR, TRP, TYR, VAL

polar residues ARG, ASN, ASP, ASX, CSO, CYS, CYX, GLN, GLU, GLX, GLY, HID, IE, HIP, HIS, HSD, HSE, HSP, LYS, PHD, PTR, SEC, SEP, SER, THR, PO, TYR

small residues ALA, GLY, SER

surface residues ARG, ASN, ASP, ASX, CSO, GLN, GLU, GLX, GLY, HID, HIE, HIP, IS, HSD, HSE, HSP, LYS, PHD, PRO, PTR, SEP, SER, THR, TPO, TYR

Nucleic

nucleic indicates *nucleobase*, *nucleotide*, and some *nucleoside* derivatives that are described below, so it is same as `'nucleobase or nucleotide or nucleoside'`.

nucleobase indicates ADE[72] (adenine), GUN[73] (guanine), CYT[74] (cytosine), THY[75] (thymine), and URA[76] (uracil).

nucleotide indicates residues with the following names:

[72] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ADE
[73] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GUN
[74] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CYT
[75] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=THY
[76] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=URA

DA[77]	2′-deoxyadenosine-5′-monophosphate
DC[78]	2′-deoxycytidine-5′-monophosphate
DG[79]	2′-deoxyguanosine-5′-monophosphate
DT[80]	2′-deoxythymidine-5′-monophosphate
DU[81]	2′-deoxyuridine-5′-monophosphate
A[82]	adenosine-5′-monophosphate
C[83]	cytidine-5′-monophosphate
G[84]	guanosine-5′-monophosphate
T[85]	2′-deoxythymidine-5′-monophosphate
U[86]	uridine-5′-monophosphate

nucleoside indicates following nucleosides and their derivatives that are recognized by *PDB*:

[77] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DA
[78] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DC
[79] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DG
[80] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DT
[81] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DU
[82] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=A
[83] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=C
[84] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=G
[85] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=T
[86] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=U

ADN[87]	adenosine
AMP[88]	adenosine-5′-monophosphate
ADP[89]	adenosine-5′-diphosphate
ATP[90]	adenosine-5′-triphosphate
AGS[91]	adenosine-5′-triphosphate-gamma-S
CMP[92]	cyclic adenosine-3′,5′-monophosphate
A2P[93]	adenosine-2′,5′-diphosphate
A3P[94]	adenosine-3′,5′-diphosphate
CTN[95]	cytidine
C2P[96]	cytidine-2′-monophosphate
C3P[97]	cytidine-3′-monophosphate
C5P[98]	cytidine-5′-monophosphate
CDP[99]	cytidine-5′-diphosphate
CTP[100]	cytidine-5′-triphosphate
GMP[101]	guanosine
5GP[102]	guanosine-5′-monophosphate
GDP[103]	guanosine-5′-diphosphate
GTP[104]	guanosine-5′-triphosphate
THM[105]	thymidine
TMP[106]	thymidine-5′-monophosphate
TPP[107]	thymidine-5′-diphosphate
TTP[108]	thymidine-5′-triphosphate
URI[109]	uridine (uracil plus ribose)
UMP[110]	2′-deoxyuridine 5′-monophosphate
UDP[111]	uridine 5′-diphosphate
UTP[112]	uridine 5′-triphosphate

at same as selection `'resname ADE A THY T'`

cg same as selection `'resname CYT C GUN G'`

purine same as selection `'resname ADE A GUN G'`

[87] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ADN
[88] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=AMP
[89] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ADP
[90] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ATP
[91] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=AGS
[92] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CMP
[93] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=A2P
[94] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=A3P
[95] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CTN
[96] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=C2P
[97] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=C3P
[98] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=C5P
[99] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CDP
[100] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CTP
[101] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GMP
[102] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=5GP
[103] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GDP
[104] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GTP
[105] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=THM
[106] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TMP
[107] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TPP
[108] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TTP
[109] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=URI
[110] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=UMP
[111] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=UDP
[112] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=UTP

pyrimidine same as selection `'resname CYT C THY T URA U'`

Heteros

hetero indicates anything other than a *protein* or a *nucleic* residue, i.e. `'not (protein or nucleic)'`.

hetatm is available when atomic data is parsed from a PDB or similar format file and indicates atoms that are marked `'HETATM'` in the file.

water indices HOH[113] and DOD[114] recognized by *PDB* and also WAT, TIP3, H2O, OH2, TIP, TIP2, and TIP4 recognized by molecular dynamics (MD) force fields.

Previously used water types HH0, OHH, and SOL conflict with other compounds in the *PDB*, so are removed from the definition of this flag.

ion indicates the following ions most of which are recognized by the *PDB* and others by MD force fields.

		PDB	*Source*	*Conflict*
AL[115]	aluminum	Yes		
BA[116]	barium	Yes		
CA[117]	calcium	Yes		
CD[118]	cadmium	Yes		
CL[119]	chloride	Yes		
CO[120]	cobalt (ii)	Yes		
CS[121]	cesium	Yes		
CU[122]	copper (ii)	Yes		
CU1[123]	copper (i)	Yes		
CUA[124]	dinuclear copper	Yes		
HG[125]	mercury (ii)	Yes		
IN[126]	indium (iii)	Yes		
IOD[127]	iodide	Yes		
K[128]	potassium	Yes		
MG[129]	magnesium	Yes		
MN3[130]	manganese (iii)	Yes		
MN[131]	manganese (ii)	Yes		
NA[132]	sodium	Yes		
PB[133]	lead (ii)	Yes		
PT[134]	platinum (ii)	Yes		
RB[135]	rubidium	Yes		
TB[136]	terbium (iii)	Yes		
TL[137]	thallium (i)	Yes		
WO4[138]	thungstate (vi)	Yes		
YB[139]	ytterbium (iii)	Yes		
ZN[140]	zinc	Yes		
CAL	calcium	No	CHARMM	Yes
CES	cesium	No	CHARMM	Yes
CLA	chloride	No	CHARMM	Yes
POT	potassium	No	CHARMM	Yes
SOD	sodium	No	CHARMM	Yes
ZN2	zinc	No	CHARMM	No

[113] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HOH
[114] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DOD

Ion identifiers that are obsoleted by *PDB* (MO3, MO4, MO5, MO6, NAW, OC7, and ZN1) are removed from this definition.

lipid indicates GPE[141], LPP[142], OLA[143], SDS[144], and STE[145] from *PDB*, and also POPC, LPPC, POPE, DLPE, PCGL, STEA, PALM, OLEO, DMPC from *CHARMM* force field.

sugar indicates BGC[146], GLC[147], and GLO[148] from *PDB*, and also AGLC from *CHARMM*.

heme indicates 1FH[149], 2FH[150], DDH[151], DHE[152], HAS[153], HDD[154], HDE[155], HDM[156], HEA[157], HEB[158], HEC[159], HEM[160], HEO[161], HES[162], HEV[163], NTE[164], SRM[165], and VER[166] from *PDB*, and also HEMO

[115] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=AL
[116] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=BA
[117] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CA
[118] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CD
[119] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CL
[120] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CO
[121] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CS
[122] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CU
[123] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CU1
[124] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CUA
[125] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HG
[126] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=IN
[127] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=IOD
[128] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=K
[129] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MG
[130] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MN3
[131] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MN
[132] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=NA
[133] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PB
[134] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PT
[135] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=RB
[136] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TB
[137] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TL
[138] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=WO4
[139] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=YB
[140] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ZN
[141] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GPE
[142] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=LPP
[143] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=OLA
[144] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SDS
[145] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=STE
[146] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=BGC
[147] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLC
[148] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLO
[149] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=1FH
[150] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=2FH
[151] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DDH
[152] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DHE
[153] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HAS
[154] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HDD
[155] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HDE
[156] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HDM
[157] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEA
[158] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEB
[159] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEC
[160] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEM
[161] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEO
[162] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HES
[163] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEV
[164] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=NTE
[165] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SRM
[166] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=VER

and HEMR from CHARMM.

pdbter is available when atomic data is parsed from a PDB format file and indicates atoms that were followed by `'TER'` record.

selpdbter is available when atomic data is parsed from a PDB format file and then a selection is made and indicates selected atoms that should be followed by `'TER'` record.

Elements

Following elements found in proteins are recognized by applying regular expressions to atom names:

carbon carbon atoms, same as `'name "C.*" and not ion'`

nitrogen nitrogen atoms, same as `'name "N.*" and not ion'`

oxygen oxygen atoms, same as `'name "O.*" and not ion'`

sulfur sulfur atoms, same as `'name "S.*" and not ion'`

hydrogen hydrogen atoms, same as `'name "[1-9]?H.*" and not ion'`

noh, heavy non hydrogen atoms, same as `'not hydrogen`

`'not ion'` is appended to above definitions to avoid conflicts with *ion* atoms.

Structure

Following secondary structure flags are defined but before they can be used, secondary structure assignments must be made.

extended extended conformation, same as `'secondary E'`

helix α-helix conformation, same as `'secondary H'`

helix310 3_10-helix conformation, same as `'secondary G'`

helixpi π-helix conformation, same as `'secondary I'`

turn hydrogen bonded turn conformation, same as `'secondary T'`

bridge isolated beta-bridge conformation, same as `'secondary B'`

bend bend conformation, same as `'secondary S'`

coil not in one of above conformations, same as `'secondary C'`

Others

all indicates all atoms, returns a new view of the instance

none indicates no atoms, returns **None**

dummy indicates dummy atoms in an *AtomMap* (page 49)

mapped indicates mapped atoms in an *AtomMap* (page 49)

Functions

The following functions can be used to customize flag definitions:

- *flagDefinition()* (page 72)
- *addNonstdAminoacid()* (page 73)
- *delNonstdAminoacid()* (page 74)

- *listNonstdAAProps()* (page 73)

flagDefinition(**arg*, ***kwarg*)

Learn, change, or reset *Atom Flags* (page 64) definitions.

Learn a definition

Calling this function with no arguments will return list of flag names whose definitions you can learn:

```
In [1]: flagDefinition()
Out[1]:
['acidic',
 'acyclic',
 'aliphatic',
 'aminoacid',
 'aromatic',
 'at',
 'backbone',
 'backbonefull',
 'basic',
 'bb',
 'bbfull',
 'buried',
 'carbon',
 'cg',
 'charged',
 'cyclic',
 'heme',
 'hydrogen',
 'hydrophobic',
 'ion',
 'large',
 'lipid',
 'medium',
 'neutral',
 'nitrogen',
 'nonstdaa',
 'nucleic',
 'nucleobase',
 'nucleoside',
 'nucleotide',
 'oxygen',
 'polar',
 'protein',
 'purine',
 'pyrimidine',
 'small',
 'stdaa',
 'sugar',
 'sulfur',
 'surface',
 'water']
```

Passing a flag name will return its definition:

```
In [2]: flagDefinition('backbone')
Out[2]: ['C', 'CA', 'N', 'O']

In [3]: flagDefinition('hydrogen')
Out[3]: '[0-9]?H.*'
```

Change a definition

Calling the function with `editable=True` argument will return flag names those definitions that can be edited:

```
In [4]: flagDefinition(editable=True)
Out[4]:
['at',
 'backbone',
 'backbonefull',
 'bb',
 'bbfull',
 'carbon',
 'cg',
 'heme',
 'hydrogen',
 'ion',
 'lipid',
 'nitrogen',
 'nucleobase',
 'nucleoside',
 'nucleotide',
 'oxygen',
 'purine',
 'pyrimidine',
 'sugar',
 'sulfur',
 'water']
```

Pass an editable flag name with its new definition:

```
In [5]: flagDefinition(nitrogen='N.*')

In [6]: flagDefinition(backbone=['CA', 'C', 'O', 'N'])

In [7]: flagDefinition(nucleobase=['ADE', 'CYT', 'GUN', 'THY', 'URA'])
```

Note that the type of the new definition must be the same as the type of the old definition. Flags with editable definitions are: *at*, *backbone*, *backbonefull*, *bb*, *bbfull*, *carbon*, *cg*, *heme*, *hydrogen*, *ion*, *lipid*, *nitrogen*, *nucleobase*, *nucleoside*, *nucleotide*, *oxygen*, *purine*, *pyrimidine*, *sugar*, *sulfur*, and *water*

Reset definitions

Pass *reset* keyword as follows to restore all default definitions of editable flags and also non-standard amino acids.

```
In [8]: flagDefinition(reset='all')
```

Or, pass a specific editable flag label to restore its definition:

```
In [9]: flagDefinition(reset='nitrogen')
```

listNonstdAAProps (*resname*)
Returns properties of non-standard amino acid *resname*.

```
In [1]: listNonstdAAProps('PTR')
Out[1]: ['acidic', 'aromatic', 'cyclic', 'large', 'polar', 'surface']
```

getNonstdProperties (*resname*)
Deprecated for removal in v1.4, use *listNonstdAAProps()* (page 73) instead.

addNonstdAminoacid(*resname*, **properties*)

Add non-standard amino acid *resname* with *properties* selected from:

- *cyclic*: *acyclic*, or *cyclic*
- *charge*: *acidic*, *basic*, or *neutral*
- *depth*: *buried*, or *surface*
- *hydrophobicity*: *hydrophobic*, or *polar*
- *aromaticity*: *aliphatic*, or *aromatic*
- *size*: *large*, *medium*, or *small*

```
In [1]: addNonstdAminoacid('PTR', 'acidic', 'aromatic', 'cyclic', 'large',
   ...: 'polar', 'surface')
   ...:
```

Default set of non-standard amino acids can be restored as follows:

```
In [2]: flagDefinition(reset='nonstdaa')
```

delNonstdAminoacid(*resname*)

Delete non-standard amino acid *resname*.

```
In [1]: delNonstdAminoacid('PTR')

In [2]: flagDefinition('nonstdaa')
Out[2]:
['ASX',
 'CME',
 'CSB',
 'CSO',
 'CYX',
 'GLX',
 'HID',
 'HIE',
 'HIP',
 'HSD',
 'HSE',
 'HSP',
 'MEN',
 'MSE',
 'PHD',
 'SEC',
 'SEP',
 'TPO',
 'XAA',
 'XLE']
```

Default set of non-standard amino acids can be restored as follows:

```
In [3]: flagDefinition(reset='nonstdaa')
```

3.1.19 Supporting Functions

This module defines some functions for handling atomic classes and data.

iterFragments(*atoms*)

Yield fragments, connected subsets in *atoms*, as *Selection* (page 100) instances.

findFragments(*atoms*)
Returns list of fragments, connected subsets in *atoms*. See also *iterFragments()* (page 74).

loadAtoms(*filename*)
Returns *AtomGroup* (page 38) instance loaded from *filename* using `numpy.load()` function. See also *saveAtoms()* (page 75).

saveAtoms(*atoms*, *filename=None*, ***kwargs*)
Save *atoms* in ProDy internal format. All *Atomic* (page 47) classes are accepted as *atoms* argument. This function saves user set atomic data as well. Note that title of the *AtomGroup* (page 38) instance is used as the filename when *atoms* is not an *AtomGroup* (page 38). To avoid overwriting an existing file with the same name, specify a *filename*.

isReserved(*word*)
Returns **True** if *word* is reserved for internal data labeling or atom selections. See *listReservedWords()* (page 75) for a list of reserved words.

listReservedWords()
Returns list of words that are reserved for atom selections and internal variables. These words are: *abs*, *acidic*, *acos*, *acyclic*, *aliphatic*, *all*, *altloc*, *aminoacid*, *and*, *aromatic*, *as*, *asin*, *at*, *atan*, *backbone*, *backbonefull*, *basic*, *bb*, *bbfull*, *bend*, *beta*, *bmap*, *bonded*, *bonds*, *bridge*, *buried*, *ca*, *calpha*, *carbon*, *ceil*, *cg*, *chain*, *charge*, *charged*, *chid*, *chindex*, *coil*, *coordinates*, *cos*, *cosh*, *cslabels*, *cyclic*, *dummy*, *element*, *exbonded*, *exp*, *extended*, *exwithin*, *floor*, *fragindex*, *fragment*, *heavy*, *helix*, *helix310*, *helixpi*, *heme*, *hetero*, *hydrogen*, *hydrophobic*, *icode*, *index*, *ion*, *large*, *lipid*, *log*, *log10*, *mapped*, *mass*, *medium*, *n_atoms*, *n_csets*, *name*, *neutral*, *nitrogen*, *noh*, *none*, *nonstdaa*, *not*, *nucleic*, *nucleobase*, *nucleoside*, *nucleotide*, *numbonds*, *occupancy*, *of*, *or*, *oxygen*, *polar*, *protein*, *purine*, *pyrimidine*, *radius*, *resid*, *resindex*, *resname*, *resnum*, *same*, *sc*, *secclass*, *secid*, *secindex*, *secondary*, *secstr*, *segindex*, *segment*, *segname*, *sequence*, *serial*, *sidechain*, *siguij*, *sin*, *sinh*, *small*, *sq*, *sqrt*, *stdaa*, *sugar*, *sulfur*, *surface*, *tahn*, *tan*, *title*, *to*, *turn*, *type*, *water*, *within*, *x*, *y*, *z*.

sortAtoms(*atoms*, *label*, *reverse=False*)
Returns an *AtomMap* (page 49) pointing to *atoms* sorted in ascending data *label* order, or optionally in *reverse* order.

sliceAtoms(*atoms*, *select*)
Slice *atoms* using the selection defined by *select*.

Parameters

- **atoms** (*Atomic* (page 47)) – atoms to be selected from
- **select** (*Selection* (page 100), str) – a *Selection* (page 100) instance or selection string

extendAtoms(*nodes*, *atoms*, *is3d=False*)
Returns extended mapping indices and an *AtomMap* (page 49).

sliceAtomicData(*data*, *atoms*, *select*, *axis=None*)
Slice a matrix using indices extracted using *sliceAtoms()* (page 75).

Parameters

- **data** (`ndarray`) – any data array
- **atoms** (*Atomic* (page 47)) – atoms to be selected from
- **select** (*Selection* (page 100), str) – a *Selection* (page 100) instance or selection string
- **axis** (*int, list*) – the axis along which the data is sliced. See `numpy` for details of this parameter. Default is **None** (all axes)

extendAtomicData(*data*, *nodes*, *atoms*, *axis=None*)
Extend a coarse grained data obtained for *nodes* to *atoms*.

Parameters

- **data** (ndarray) – any data array
- **nodes** (*Atomic* (page 47)) – a set of atoms that has been used as nodes in data generation
- **atoms** (*Atomic* (page 47)) – atoms to be selected from
- **axis** (*int*[167]) – the axis/direction you want to use to slice data from the matrix. The options are **0** or **1** or **None** like in numpy. Default is **None** (all axes)

3.1.20 Hierarchical Views

This module defines *HierView* (page 76) class that builds a hierarchical views of atom groups.

class HierView(*atoms*, ***kwargs*)
Hierarchical views can be generated for *AtomGroup* (page 38), *Selection* (page 100), and *Chain* (page 53) instances. Indexing a *HierView* (page 76) instance returns a *Chain* (page 53) instance.

Some object[168] methods are customized as follows:

- len()[169] returns the number of chains, i.e. *numChains()* (page 77)
- iter()[170] yields *Chain* (page 53) instances
- **indexing by:**
  - *segment name* (str()), e.g. "PROT", returns a *Segment* (page 86)
  - *chain identifier* (str()), e.g. "A", returns a *Chain* (page 53)
  - *[segment name,] chain identifier, residue number[, insertion code]* (tuple()), e.g. "A", 10 or "A", 10, "B" or "PROT", "A", 10, "B", returns a *Residue* (page 80)

Note that when an *AtomGroup* (page 38) instance have distinct segments, they will be considered when building the hierarchical view. A *Segment* (page 86) instance will be generated for each distinct segment name. Then, for each segment chains and residues will be evaluated. Having segments in the structure will not change most behaviors of this class, except indexing. For example, when indexing a hierarchical view for chain P in segment PROT needs to be indexed as hv['PROT', 'P'].

getAtoms()
Returns atoms for which the hierarchical view was built.

getChain(*chid*, *segname=None*)
Returns chain with identifier *chid*, if it is present.

getResidue(*chid*, *resnum*, *icode=None*, *segname=None*)
Returns residue with number *resnum* and insertion code *icode* from the chain with identifier *chid* in segment with name *segname*.

getSegment(*segname*)
Returns segment with name *segname*, if it is present.

iterChains()
Yield chains.

[167] http://docs.python.org/library/functions.html#int
[168] http://docs.python.org/library/functions.html#object
[169] http://docs.python.org/library/functions.html#len
[170] http://docs.python.org/library/functions.html#iter

iterResidues()
Yield residues.

iterSegments()
Yield segments.

numChains()
Returns number of chains.

numResidues()
Returns number of residues.

numSegments()
Returns number of chains.

update(***kwargs*)
Update (or build) hierarchical view of atoms. This method is called at instantiation, but can be used to rebuild the hierarchical view when attributes of atoms change.

3.1.21 Improper

This module defines *Improper* (page 77) for dealing with improper information provided by using *AtomGroup.setImpropers()* (page 45) method.

class Improper(*ag*, *indices*, *acsi=None*)
A pointer class for improperd atoms. Following built-in functions are customized for this class:

- `len()`[171] returns improper length, i.e. *getSize()* (page 77)
- `iter()`[172] yields *Atom* (page 32) instances

getACSIndex()
Returns index of the coordinate set.

getAtomGroup()
Returns atom group.

getAtoms()
Returns improperd atoms.

getIndices()
Returns indices of improperd atoms.

getSize()
Returns improper size.

getVectors()
Returns bond vectors that originate from the central atom.

setACSIndex(*index*)
Set the coordinate set at *index* active.

3.1.22 Non-Bonded Exclusions

This module defines *NBExclusion* (page 77) for dealing with bond information provided by using *AtomGroup.setNBExclusions()* (page 46) method.

class NBExclusion(*ag*, *indices*, *acsi=None*)
A pointer class for nonnonbonded exclusion exclusion atoms. Following built-in functions are customized for this class:

[171] http://docs.python.org/library/functions.html#len
[172] http://docs.python.org/library/functions.html#iter

- `iter()`[173] yields *Atom* (page 32) instances

getACSIndex()
Returns index of the coordinate set.

getAtomGroup()
Returns atom group.

getAtoms()
Returns nonbonded exclusion atoms.

getIndices()
Returns indices of nonbonded exclusion atoms.

getVector()
Returns vector that originates from the first atom.

setACSIndex(*index*)
Set the coordinate set at *index* active.

3.1.23 Atom Pointer

This module defines atom pointer base class.

class AtomPointer(*ag*, *acsi*)
A base for classes pointing to atoms in *AtomGroup* (page 38) instances. Derived classes are:

- *Atom* (page 32)
- *AtomSubset* (page 105)
- *AtomMap* (page 49)

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAtomGroup()
Returns associated atom group.

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

getCSLabels()
Returns coordinate set labels.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided

[173] http://docs.python.org/library/functions.html#iter

flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getTitle()
Returns title of the instance.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` or `inferBonds()` for setting bonds.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

numBonds()
Returns number of bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

numCoordsets()
Returns number of coordinate sets.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

3.1.24 Residue

This module defines classes for handling residues.

class Residue(*ag*, *indices*, *hv*, *acsi=None*, ***kwargs*)
Instances of this class point to atoms with same residue numbers (and insertion codes) and are generated by *HierView* (page 76) class. Following built-in functions are customized for this class:

- `len()`[174] returns the number of atoms in the instance.
- `iter()`[175] yields *Atom* (page 32) instances.

Indexing *Residue* (page 80) instances by *atom name* (`str()`), e.g. `"CA"` returns an *Atom* (page 32) instance.

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAltlocs()
Return a copy of alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAnistds()
Return a copy of standard deviations for anisotropic temperature factors.

getAtom(*name*)
Returns atom with given *name*, **None** if not found. Assumes that atom names in the residue are unique. If more than one atoms with the given *name* exists, the one with the smaller index will be returned.

getAtomGroup()
Returns associated atom group.

getBetas()
Return a copy of β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

[174] http://docs.python.org/library/functions.html#len
[175] http://docs.python.org/library/functions.html#iter

getCSLabels()
Returns coordinate set labels.

getChain()
Returns the chain that the residue belongs to.

getCharges()
Return a copy of partial charges. Partial charges can be used in atom selections, e.g. `'charge 1','abs(charge) == 1','charge < 0'`.

getChid()
Returns chain identifier.

getChids()
Return a copy of chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

getChindices()
Return a copy of chain indices. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

getCoords()
Returns a copy of coordinates from the active coordinate set.

getCoordsets(*indices=None*)
Returns coordinate set(s) at given *indices*, which may be an integer or a list/array of integers.

getData(*label*)
Returns a copy of data associated with *label*, if it is present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getElements()
Return a copy of element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFlags(*label*)
Returns a copy of atom flags for given *label*, or **None** when flags for *label* is not set.

getFragindices()
Return a copy of fragment indices. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Fragment indices can be used in atom selections, e.g. `'fragindex 0','fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getIcode()
Returns residue insertion code.

getIcodes()
Return a copy of insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

getIndices()
Returns a copy of the indices of atoms.

getMasses()
Return a copy of masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

getNames()
Return a copy of names. Names can be used in atom selections, e.g. `'name CA CB'`.

getNext()
Returns following residue in the atom group.

getOccupancies()
Return a copy of occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

getPrev()
Returns preceding residue in the atom group.

getRadii()
Return a copy of radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

getResindex()
Returns residue index.

getResindices()
Return a copy of residue indices. Residue indices are assigned to subsets of atoms with distinct sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Residue indices can be used in atom selections, e.g. `'resindex 0'`.

getResname()
Returns residue name.

getResnames()
Return a copy of residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

getResnum()
Returns residue number.

getResnums()
Return a copy of residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3', 'resnum 120A 120B', 'resnum 10 to 20', 'resnum 10:20:2', 'resnum < 10'`. Note that *resid* is a synonym for *resnum*.

getSecclasses()
Return a copy of secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

getSecids()
Return a copy of secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

getSecindices()
Return a copy of secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

getSecstrs()
Return a copy of secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

getSegindices()
Return a copy of segment indices. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegment()
Returns segment of the residue.

getSegname()
Returns segment name.

getSegnames()
Return a copy of segment names. Segment names can be used in atom selections, e.g. `'segment PROT','segname PROT'`. Note that *segname* is a synonym for *segment*.

getSelstr()
Returns selection string that will select this residue.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerials()
Return a copy of serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

getTitle()
Returns title of the instance.

getTypes()
Return a copy of types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterAtoms()
Yield atoms.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` or `inferBonds()` for setting bonds.

iterCoordsets()
Yield copies of coordinate sets.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

numAtoms(*flag=None*)
Returns number of atoms, or number of atoms with given *flag*.

numBonds()
Returns number of bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

numCoordsets()
Returns number of coordinate sets.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

setAltlocs(*data*)
Set alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

setAnisous(*anisous*)
Set anisotropic temperature factors in the active coordinate set.

setAnistds(*data*)
Set standard deviations for anisotropic temperature factors.

setBetas(*data*)
Set β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

setCharges(*data*)
Set partial charges. Partial charges can be used in atom selections, e.g. `'charge 1', 'abs(charge) == 1','charge < 0'`.

setChids(*data*)
Set chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

setCoords(*coords*)
Set coordinates in the active coordinate set.

setData (*label*, *data*)
Update *data* associated with *label*.

Raises **AttributeError**[176] – when *label* is not in use or read-only

setElements (*data*)
Set element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

setFlags (*label*, *value*)
Update flag associated with *label*.

Raises **AttributeError**[177] – when *label* is not in use or read-only

setIcode (*icode*)
Set residue insertion code.

setIcodes (*data*)
Set insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

setMasses (*data*)
Set masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

setNames (*data*)
Set names. Names can be used in atom selections, e.g. `'name CA CB'`.

setOccupancies (*data*)
Set occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1', 'occupancy > 0'`.

setRadii (*data*)
Set radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

setResname (*name*)
Set residue name.

setResnames (*data*)
Set residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

setResnum (*number*)
Set residue number.

setResnums (*data*)
Set residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3','resnum 120A 120B','resnum 10 to 20','resnum 10:20:2','resnum < 10'`. Note that *resid* is a synonym for *resnum*.

setSecclasses (*data*)
Set secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

setSecids (*data*)
Set secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

setSecindices (*data*)
Set secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

[176] http://docs.python.org/library/exceptions.html#AttributeError
[177] http://docs.python.org/library/exceptions.html#AttributeError

setSecstrs(*data*)
Set secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

setSegnames(*data*)
Set segment names. Segment names can be used in atom selections, e.g. `'segment PROT'`, `'segname PROT'`. Note that *segname* is a synonym for *segment*.

setSerials(*data*)
Set serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

setTypes(*data*)
Set types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

3.1.25 Segment

This module defines a class to handle segments of atoms in an atom group.

class Segment(*ag*, *indices*, *hv*, *acsi=None*, ***kwargs*)
Instances of this class point to atoms with same segment names and are generated by *HierView* (page 76) class. Following built-in functions are customized for this class:

- `len()`[178] returns the number of chains in the segment.
- `iter()`[179] yields *Chain* (page 53) instances.

Indexing *Segment* (page 86) instances by a *chain identifier* (`str()`), e.g. `A`, returns a *Chain* (page 53).

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAltlocs()
Return a copy of alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

[178] http://docs.python.org/library/functions.html#len
[179] http://docs.python.org/library/functions.html#iter

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAnistds()
Return a copy of standard deviations for anisotropic temperature factors.

getAtomGroup()
Returns associated atom group.

getBetas()
Return a copy of β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

getCSLabels()
Returns coordinate set labels.

getChain(*chid*)
Returns chain with identifier *chid*.

getCharges()
Return a copy of partial charges. Partial charges can be used in atom selections, e.g. `'charge 1','abs(charge) == 1','charge < 0'`.

getChids()
Return a copy of chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

getChindices()
Return a copy of chain indices. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

getCoords()
Returns a copy of coordinates from the active coordinate set.

getCoordsets(*indices=None*)
Returns coordinate set(s) at given *indices*, which may be an integer or a list/array of integers.

getData(*label*)
Returns a copy of data associated with *label*, if it is present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getElements()
Return a copy of element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFlags(*label*)
Returns a copy of atom flags for given *label*, or **None** when flags for *label* is not set.

getFragindices()
Return a copy of fragment indices. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Fragment indices can be used in atom selections, e.g. `'fragindex 0','fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getHierView(***kwargs*)
Returns a hierarchical view of the this segment.

getIcodes()
Return a copy of insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

getIndices()
Returns a copy of the indices of atoms.

getMasses()
Return a copy of masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

getNames()
Return a copy of names. Names can be used in atom selections, e.g. `'name CA CB'`.

getOccupancies()
Return a copy of occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

getRadii()
Return a copy of radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

getResindices()
Return a copy of residue indices. Residue indices are assigned to subsets of atoms with distinct sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Residue indices can be used in atom selections, e.g. `'resindex 0'`.

getResnames()
Return a copy of residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

getResnums()
Return a copy of residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3', 'resnum 120A 120B', 'resnum 10 to 20', 'resnum 10:20:2', 'resnum < 10'`. Note that *resid* is a synonym for *resnum*.

getSecclasses()
Return a copy of secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

getSecids()
Return a copy of secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

getSecindices()
Return a copy of secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

getSecstrs()
Return a copy of secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

getSegindices()
Return a copy of segment indices. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegname()
Returns segment name.

getSegnames()
Return a copy of segment names. Segment names can be used in atom selections, e.g. `'segment PROT','segname PROT'`. Note that *segname* is a synonym for *segment*.

getSelstr()
Returns selection string that selects atoms in this segment.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerials()
Return a copy of serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

getTitle()
Returns title of the instance.

getTypes()
Return a copy of types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterAtoms()
Yield atoms.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` or `inferBonds()` for setting bonds.

iterChains()
Yield chains.

iterCoordsets()
Yield copies of coordinate sets.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

numAtoms(*flag=None*)
Returns number of atoms, or number of atoms with given *flag*.

numBonds()
Returns number of bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

numChains()
Returns number of chains.

numCoordsets()
Returns number of coordinate sets.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

setAltlocs(*data*)
Set alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

setAnisous(*anisous*)
Set anisotropic temperature factors in the active coordinate set.

setAnistds(*data*)
Set standard deviations for anisotropic temperature factors.

setBetas(*data*)
Set β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

setCharges(*data*)
Set partial charges. Partial charges can be used in atom selections, e.g. `'charge 1', 'abs(charge) == 1','charge < 0'`.

setChids(*data*)
Set chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

setCoords(*coords*)
Set coordinates in the active coordinate set.

setData(*label*, *data*)
Update *data* associated with *label*.

Raises **AttributeError**[180] – when *label* is not in use or read-only

setElements (*data*)
Set element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

setFlags (*label*, *value*)
Update flag associated with *label*.

Raises **AttributeError**[181] – when *label* is not in use or read-only

setIcodes (*data*)
Set insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

setMasses (*data*)
Set masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

setNames (*data*)
Set names. Names can be used in atom selections, e.g. `'name CA CB'`.

setOccupancies (*data*)
Set occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1'`, `'occupancy > 0'`.

setRadii (*data*)
Set radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

setResnames (*data*)
Set residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

setResnums (*data*)
Set residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3','resnum 120A 120B','resnum 10 to 20','resnum 10:20:2','resnum < 10'`. Note that *resid* is a synonym for *resnum*.

setSecclasses (*data*)
Set secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

setSecids (*data*)
Set secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

setSecindices (*data*)
Set secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

setSecstrs (*data*)
Set secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

setSegname (*segname*)
Set segment name.

setSegnames (*data*)
Set segment names. Segment names can be used in atom selections, e.g. `'segment PROT'`, `'segname PROT'`. Note that *segname* is a synonym for *segment*.

[180] http://docs.python.org/library/exceptions.html#AttributeError
[181] http://docs.python.org/library/exceptions.html#AttributeError

setSerials(*data*)
Set serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

setTypes(*data*)
Set types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

3.1.26 Atom Selections

This module defines a class for selecting subsets of atoms. You can read this page in interactive sessions using `help(select)`.

ProDy offers a fast and powerful atom selection class, *Select* (page 99). Selection features, grammar, and keywords are similar to those of VMD. Small differences, that is described below, should not affect most practical uses of atom selections. With added flexibility of Python, ProDy selection engine can also be used to identify intermolecular contacts. You may see this and other usage examples in Intermolecular Contacts[182] and Operations on Selections[183].

First, we import everything from ProDy and parse a protein-DNA-ligand complex structure:

```
In [1]: from prody import *

In [2]: p = parsePDB('3mht')
```

parsePDB() (page 268) returns *AtomGroup* (page 38) instances, `p` in this case, that stores all atomic data in the file. We can count different types of atoms using *Atom Flags* (page 64) and *numAtoms()* (page 43) method as follows:

```
In [3]: p.numAtoms('protein')
Out[3]: 2606

In [4]: p.numAtoms('nucleic')
Out[4]: 509

In [5]: p.numAtoms('hetero')
Out[5]: 96

In [6]: p.numAtoms('water')
Out[6]: 70
```

[182] http://prody.csb.pitt.edu/tutorials/structure_analysis/contacts.html#contacts
[183] http://prody.csb.pitt.edu/tutorials/prody_tutorial/selection.html#selection-operations

Last two counts suggest that ligand has 26 atoms, i.e. number of *hetero* atoms less the number of *water* atoms.

Atom flags

We select subset of atoms by using *AtomGroup.select()* (page 43) method. All *Atom Flags* (page 64) can be input arguments to this methods as follows:

```
In [7]: p.select('protein')
Out[7]: <Selection: 'protein' from 3mht (2606 atoms)>

In [8]: p.select('water')
Out[8]: <Selection: 'water' from 3mht (70 atoms)>
```

This operation returns *Selection* (page 100) instances, which can be an input to functions that accepts an *atoms* argument.

Logical operators

Flags can be combined using `'and'` and `'or'` operators:

```
In [9]: p.select('protein and water')
```

`'protein and water'` did not result in selection of *protein* and *water* atoms. This is because, no atom is flagged as a protein and a water atom at the same time.

Note: Interpreting selection strings

You may think as if a selection string, such as `'protein and water'`, is evaluated on a per atom basis and an atom is selected if it satisfies the given criterion. To select both water and protein atoms, `'or'` logical operator should be used instead. A protein or a water atom would satisfy `'protein or water'` criterion.

```
In [10]: p.select('protein or water')
Out[10]: <Selection: 'protein or water' from 3mht (2676 atoms)>
```

We can also use `'not'` operator to negate an atom flag. For example, the following selection will only select ligand atoms:

```
In [11]: p.select('not water and hetero')
Out[11]: <Selection: 'not water and hetero' from 3mht (26 atoms)>
```

If you omit the `'and'` operator, you will get the same result:

```
In [12]: p.select('not water hetero')
Out[12]: <Selection: 'not water hetero' from 3mht (26 atoms)>
```

Note: Default operator between two flags, or other selection tokens that will be discussed later, is `'and'`. For example, `'not water hetero'` is equivalent to `'not water and hetero'`.

We can select Cα atoms of acidic residues by omitting the default logical operator as follows:

```
In [13]: sel = p.select('acidic calpha')

In [14]: sel
Out[14]: <Selection: 'acidic calpha' from 3mht (39 atoms)>
```

```
In [15]: set(sel.getResnames())
Out[15]: {'ASP', 'GLU'}
```

Quick selections

For simple selections, such as shown above, following may be preferable over the *select()* (page 43) method:

```
In [16]: p.acidic_calpha
Out[16]: <Selection: 'acidic calpha' from 3mht (39 atoms)>
```

The result is the same as using `p.select('acidic calpha')`. Underscore, _, is considered as a whitespace. The limitation of this approach is that special characters cannot be used.

Atom data fields

In addition to *Atom Flags* (page 64), *Atom Data Fields* (page 61) can be used in atom selections when combined with some values. For example, we can select Cα and Cβ atoms of alanine residues as follows:

```
In [17]: p.select('resname ALA name CA CB')
Out[17]: <Selection: 'resname ALA name CA CB' from 3mht (32 atoms)>
```

Note that we omitted the default `'and'` operator.

Note: **Whitespace** or **empty string** can be specified using an `'_'`. Atoms with string data fields empty, such as those with no a chain identifiers or alternate location identifiers, can be selected using an underscore.

```
In [18]: p.select('chain _')  # chain identifiers of all atoms are specified in 3mht

In [19]: p.select('altloc _')  # altloc identifiers for all atoms are empty
Out[19]: <Selection: 'altloc _' from 3mht (3211 atoms)>
```

Numeric data fields can also be used to make selections:

```
In [20]: p.select('ca resnum 1 2 3 4')
Out[20]: <Selection: 'ca resnum 1 2 3 4' from 3mht (4 atoms)>
```

A special case for residues is having insertion codes. Residue numbers and insertion codes can be specified together as follows:

- `'resnum 5'` selects residue 5 (all insertion codes)
- `'resnum 5A'` selects residue 5 with insertion code A
- `'resnum 5_'` selects residue 5 with no insertion code

Number ranges

A range of numbers using `'to'` or Python style slicing with `':'`:

```
In [21]: p.select('ca resnum 1to4')
Out[21]: <Selection: 'ca resnum 1to4' from 3mht (4 atoms)>

In [22]: p.select('ca resnum 1:4')
Out[22]: <Selection: 'ca resnum 1:4' from 3mht (3 atoms)>
```

```
In [23]: p.select('ca resnum 1:4:2')
Out[23]: <Selection: 'ca resnum 1:4:2' from 3mht (2 atoms)>
```

Note: **Number ranges** specify continuous intervals:

- `'to'` is all inclusive, e.g. `'resnum 1 to 4'` means `'1 <= resnum <= 4'`
- `':'` is left inclusive, e.g. `'resnum 1:4'` means `'1 <= resnum < 4'`

Consecutive use of `':'`, however, specifies a discrete range of numbers, e.g. `'resnum 1:4:2'` means `'resnum 1 3'`

Special characters

Following characters can be specified when using *Atom Data Fields* (page 61) for atom selections:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
~@#$.:;_',
```

For example, `"name C' N` O~ C$ C#"` is a valid selection string.

Note: **Special characters** (`~!@#$%^&*()-_=+[{}]\|;:,<>./?()'"`) must be escaped using grave accent characters (`` ` ``).

Negative numbers

Negative numbers and number ranges must also be escaped using grave accent characters, since negative sign `'-'` is considered a special character unless it indicates subtraction operation (see below).

```
In [24]: p.select('x `-25 to 25`')
Out[24]: <Selection: 'x `-25 to 25`' from 3mht (1941 atoms)>

In [25]: p.select('x `-22.542`')
Out[25]: <Selection: 'x `-22.542`' from 3mht (1 atoms)>
```

Omitting the grave accent character will cause a *SelectionError* (page 99).

Regular expressions

Finally, you can specify regular expressions to select atoms based on data fields with type string. Following will select residues whose names start with capital letter A

```
In [26]: sel = p.select('resname "A.*"')

In [27]: set(sel.getResnames())
Out[27]: {'ALA', 'ARG', 'ASN', 'ASP'}
```

Note: **Regular expressions** can be specified using double quotes, `"..."`. For more information on regular expressions see `re`[184].

[184] http://docs.python.org/library/re.html#module-re

Numerical comparisons

Atom Data Fields (page 61) with numeric types can be used as operands in numerical comparisons:

```
In [28]: p.select('x < 0')
Out[28]: <Selection: 'x < 0' from 3mht (3095 atoms)>

In [29]: p.select('occupancy = 1')
Out[29]: <Selection: 'occupancy = 1' from 3mht (3211 atoms)>
```

Comparison	Description
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
=	equal
!=	not equal

It is also possible to chain comparison statements as follows:

```
In [30]: p.select('-10 <= x < 0')
Out[30]: <Selection: '-10 <= x < 0' from 3mht (557 atoms)>
```

This would be the same as the following selection:

```
In [31]: p.select('-10 <= x and x < 0') == p.select('-10 <= x < 0')
Out[31]: True
```

Furthermore, numerical comparisons may involve the following operations:

Operation	Description
x ** y	x to the power y
x ^ y	x to the power y
x * y	x times y
x / y	x divided by y
x // y	x divided by y (floor division)
x % y	x modulo y
x + y	x plus y
x - y	x minus y

These operations must be used with a numerical comparison, e.g.

```
In [32]: p.select('x ** 2 < 10')
Out[32]: <Selection: 'x ** 2 < 10' from 3mht (238 atoms)>

In [33]: p.select('x ** 2 ** 2 < 10')
Out[33]: <Selection: 'x ** 2 ** 2 < 10' from 3mht (134 atoms)>
```

Finally, following functions can be used in numerical comparisons:

Function	Description
abs(x)	absolute value of x
acos(x)	arccos of x
asin(x)	arcsin of x
atan(x)	arctan of x
ceil(x)	smallest integer not less than x
cos(x)	cosine of x
cosh(x)	hyperbolic cosine of x
floor(x)	largest integer not greater than x
exp(x)	e to the power x
log(x)	natural logarithm of x
log10(x)	base 10 logarithm of x
sin(x)	sine of x
sinh(x)	hyperbolic sine of x
sq(x)	square of x
sqrt(x)	square-root of x
tan(x)	tangent of x
tanh(x)	hyperbolic tangent of x

```
In [34]: p.select('sqrt(sq(x) + sq(y) + sq(z)) < 100')  # within 100 A of origin
Out[34]: <Selection: 'sqrt(sq(x) + sq(y) + sq(z)) < 100' from 3mht (1975 atoms)>
```

Distance based selections

Atoms within a user specified distance (A) from a set of user specified atoms can be selected using `'within . of .'` keyword, e.g. `'within 5 of water'` selects atoms that are within 5 A of water molecules. This setting will results selecting water atoms as well.

User can avoid selecting specified atoms using `exwithin . of ..` setting, e.g. `'exwithin 5 of water'` will not select water molecules and is equivalent to `'within 5 of water and not water'`

```
In [35]: p.select('exwithin 5 of water') == p.select('not water within 5 of water')
Out[35]: True
```

Sequence selections

One-letter amino acid sequences can be used to make atom selections. `'sequence SAR'` will select **SER-ALA-ARG** residues in a chain. Note that the selection does not consider connectivity within a chain. Regular expressions can also be used to make selections: `'sequence "MI.*KQ"'` will select **MET-ILE-(XXX)n-ASP-LYS-GLN** pattern, if present.

```
In [36]: sel = p.select('ca sequence "MI.*DKQ"')

In [37]: sel
Out[37]: <Selection: 'ca sequence "MI.*DKQ"' from 3mht (8 atoms)>

In [38]: sel.getResnames()
Out[38]:
array(['MET', 'ILE', 'GLU', 'ILE', 'LYS', 'ASP', 'LYS', 'GLN'],
      dtype='|S6')
```

Expanding selections

A selection can be expanded to include the atoms in the same *residue*, *chain*, or *segment* using `same .. as ..` setting, e.g. `'same residue as exwithin 4 of water'` will select residues that have at least an atom within 4 A of any water molecule.

```
In [39]: p.select('same residue as exwithin 4 of water')
Out[39]: <Selection: 'same residue as...thin 4 of water' from 3mht (1554 atoms)>
```

Additionally, a selection may be expanded to the immediately bonded atoms using `bonded [n] to ...` setting, e.g. `bonded 1 to calpha` will select atoms bonded to Cα atoms. For this setting to work, bonds must be set by the user using the *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42) method. It is also possible to select bonded atoms by excluding the originating atoms using `exbonded [n] to ...` setting. Number `'[n]'` indicates number of bonds to consider from the originating selection.

Selection macros

ProDy allows you to define a macro for any valid selection string. Below functions are for manipulating selection macros:

- *defSelectionMacro()* (page 99)
- *delSelectionMacro()* (page 99)
- *getSelectionMacro()* (page 99)
- *isSelectionMacro()* (page 99)

```
In [40]: defSelectionMacro('alanine', 'resname ALA')

In [41]: p.select('alanine') == p.select('resname ALA')
Out[41]: True
```

You can also use this macro as follows:

```
In [42]: p.alanine
Out[42]: <Selection: 'alanine' from 3mht (80 atoms)>
```

Macros are stored in ProDy configuration file permanently. You can delete them if you wish as follows:

```
In [43]: delSelectionMacro('alanine')
```

Keyword arguments

select() (page 99) method also accepts keyword arguments that can simplify some selections. Consider the following case where you want to select some protein atoms that are close to its center:

```
In [44]: protein = p.protein

In [45]: calcCenter(protein).round(2)
Out[45]: array([-21.17,  35.86,  79.97])

In [46]: sel1 = protein.select('sqrt(sq(x--21.17) + sq(y-35.86) + sq(z-79.97)) < 5')

In [47]: sel1
Out[47]: <Selection: '(sqrt(sq(x--21....) and (protein)' from 3mht (20 atoms)>
```

Instead, you could pass a keyword argument and use the keyword in the selection string:

```
In [48]: sel2 = protein.select('within 5 of center', center=calcCenter(protein))

In [49]: sel2
Out[49]: <Selection: 'index 1452 to 1...33 2935 to 2944' from 3mht (20 atoms)>
```

```
In [50]: sel1 == sel2
Out[50]: True
```

Note that selection string for *sel2* lists indices of atoms. This substitution is performed automatically to ensure reproducibility of the selection without the keyword *center*.

Keywords cannot be reserved words (see `listReservedWords()` (page 75)) and must be all alphanumeric characters.

exception SelectionError(*sel*, *loc=0*, *msg=''*, *tkns=None*)
Exception raised when there are errors in the selection string.

exception SelectionWarning(*sel=''*, *loc=0*, *msg=''*, *tkns=None*)
A class used for issuing warning messages when potential typos are detected in a selection string. Warnings are issued to `sys.stderr` via ProDy package logger. Use `confProDy()` (page 329) to selection warnings *on* or *off*, e.g. `confProDy(selection_warning=False)`.

class Select
Select subsets of atoms based on a selection string. See `select` (page 92) module documentation for selection grammar and examples. This class makes use of pyparsing[185] module.

getBoolArray(*atoms*, *selstr*, ***kwargs*)
Returns a boolean array with **True** values for *atoms* matching *selstr*. The length of the boolean `numpy.ndarray` will be equal to the length of *atoms* argument.

getIndices(*atoms*, *selstr*, ***kwargs*)
Returns indices of atoms matching *selstr*. Indices correspond to the order in *atoms* argument. If *atoms* is a subset of atoms, they should not be used for indexing the corresponding `AtomGroup` (page 38) instance.

select(*atoms*, *selstr*, ***kwargs*)
Returns a `Selection` (page 100) of atoms matching *selstr*, or **None**, if selection string does not match any atoms.

Parameters

- **atoms** (`Atomic` (page 47)) – atoms to be evaluated
- **selstr** (`str`[186]) – selection string

Note that, if *atoms* is an `AtomMap` (page 49) instance, an `AtomMap` (page 49) is returned, instead of a a `Selection` (page 100).

defSelectionMacro(*name*, *selstr*)
Define selection macro *selstr* with name *name*. Both *name* and *selstr* must be string. An existing keyword cannot be used as a macro name. If a macro with given *name* exists, it will be overwritten.

```
In [1]: defSelectionMacro('cbeta', 'name CB and protein')
```

delSelectionMacro(*name*)
Delete the macro *name*.

```
In [1]: delSelectionMacro('cbeta')
```

getSelectionMacro(*name=None*)
Returns the definition of the macro *name*. If *name* is not given, returns a copy of the selection macros dictionary.

[185] http://pyparsing.wikispaces.com
[186] http://docs.python.org/library/stdtypes.html#str

isSelectionMacro(*word*)
Returns **True** if *word* is a user defined selection macro.

3.1.27 Selection

This module defines *Selection* (page 100) class for handling arbitrary subsets of atom.

class Selection(*ag*, *indices*, *selstr*, *acsi=None*, ***kwargs*)
A class for accessing and manipulating attributes of selection of atoms in an *AtomGroup* (page 38) instance. Instances can be generated using *select()* (page 43) method. Following built-in functions are customized for this class:

- `len()`[187] returns the number of selected atoms
- `iter()`[188] yields *Atom* (page 32) instances

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAltlocs()
Return a copy of alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAnistds()
Return a copy of standard deviations for anisotropic temperature factors.

getAtomGroup()
Returns associated atom group.

getBetas()
Return a copy of β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

getCSLabels()
Returns coordinate set labels.

getCharges()
Return a copy of partial charges. Partial charges can be used in atom selections, e.g. `'charge 1','abs(charge) == 1','charge < 0'`.

getChids()
Return a copy of chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

getChindices()
Return a copy of chain indices. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one,

[187] http://docs.python.org/library/functions.html#len
[188] http://docs.python.org/library/functions.html#iter

and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

getCoords()
Returns a copy of coordinates from the active coordinate set.

getCoordsets(*indices=None*)
Returns coordinate set(s) at given *indices*, which may be an integer or a list/array of integers.

getData(*label*)
Returns a copy of data associated with *label*, if it is present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getElements()
Return a copy of element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFlags(*label*)
Returns a copy of atom flags for given *label*, or **None** when flags for *label* is not set.

getFragindices()
Return a copy of fragment indices. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Fragment indices can be used in atom selections, e.g. `'fragindex 0','fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getHierView(***kwargs*)
Returns a hierarchical view of the atom selection.

getIcodes()
Return a copy of insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

getIndices()
Returns a copy of the indices of atoms.

getMasses()
Return a copy of masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

getNames()
Return a copy of names. Names can be used in atom selections, e.g. `'name CA CB'`.

getOccupancies()
Return a copy of occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

getRadii()
Return a copy of radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

getResindices()
Return a copy of residue indices. Residue indices are assigned to subsets of atoms with distinct sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Residue indices can be used in atom selections, e.g. `'resindex 0'`.

getResnames()
Return a copy of residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

getResnums()
Return a copy of residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3'`, `'resnum 120A 120B'`, `'resnum 10 to 20'`, `'resnum 10:20:2'`, `'resnum < 10'`. Note that *resid* is a synonym for *resnum*.

getSecclasses()
Return a copy of secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

getSecids()
Return a copy of secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

getSecindices()
Return a copy of secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

getSecstrs()
Return a copy of secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

getSegindices()
Return a copy of segment indices. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegnames()
Return a copy of segment names. Segment names can be used in atom selections, e.g. `'segment PROT'`,`'segname PROT'`. Note that *segname* is a synonym for *segment*.

getSelstr()
Returns selection string that selects this atom subset.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerials()
Return a copy of serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

getTitle()
Returns title of the instance.

getTypes()
Return a copy of types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterAtoms()
Yield atoms.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` or `inferBonds()` for setting bonds.

iterCoordsets()
Yield copies of coordinate sets.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

numAtoms(*flag=None*)
Returns number of atoms, or number of atoms with given *flag*.

numBonds()
Returns number of bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

numCoordsets()
Returns number of coordinate sets.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

setAltlocs(*data*)
Set alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

setAnisous(*anisous*)
Set anisotropic temperature factors in the active coordinate set.

setAnistds (*data*)
Set standard deviations for anisotropic temperature factors.

setBetas (*data*)
Set β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

setCharges (*data*)
Set partial charges. Partial charges can be used in atom selections, e.g. `'charge 1', 'abs(charge) == 1','charge < 0'`.

setChids (*data*)
Set chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

setCoords (*coords*)
Set coordinates in the active coordinate set.

setData (*label*, *data*)
Update *data* associated with *label*.

Raises **AttributeError**[189] – when *label* is not in use or read-only

setElements (*data*)
Set element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

setFlags (*label*, *value*)
Update flag associated with *label*.

Raises **AttributeError**[190] – when *label* is not in use or read-only

setIcodes (*data*)
Set insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

setMasses (*data*)
Set masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

setNames (*data*)
Set names. Names can be used in atom selections, e.g. `'name CA CB'`.

setOccupancies (*data*)
Set occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1', 'occupancy > 0'`.

setRadii (*data*)
Set radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

setResnames (*data*)
Set residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

setResnums (*data*)
Set residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3','resnum 120A 120B','resnum 10 to 20','resnum 10:20:2','resnum < 10'`. Note that *resid* is a synonym for *resnum*.

setSecclasses (*data*)
Set secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

[189] http://docs.python.org/library/exceptions.html#AttributeError
[190] http://docs.python.org/library/exceptions.html#AttributeError

setSecids(*data*)
Set secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

setSecindices(*data*)
Set secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

setSecstrs(*data*)
Set secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

setSegnames(*data*)
Set segment names. Segment names can be used in atom selections, e.g. `'segment PROT'`, `'segname PROT'`. Note that *segname* is a synonym for *segment*.

setSerials(*data*)
Set serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

setTypes(*data*)
Set types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

toAtomGroup()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

update()
Update selection.

3.1.28 Atom Subsets

class AtomSubset(*ag*, *indices*, *acsi*, ***kwargs*)
A class for manipulating subset of atoms in an *AtomGroup* (page 38). Derived classes are:

- *Selection* (page 100)
- *Segment* (page 86)
- *Chain* (page 53)
- *Residue* (page 80)

This class stores a reference to an *AtomGroup* (page 38) instance, a set of atom indices, and active coordinate set index for the atom group.

copy()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

getACSIndex()
Returns index of the coordinate set.

getACSLabel()
Returns active coordinate set label.

getAltlocs()
Return a copy of alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

getAnisous()
Returns a copy of anisotropic temperature factors from the active coordinate set.

getAnistds()
Return a copy of standard deviations for anisotropic temperature factors.

getAtomGroup()
Returns associated atom group.

getBetas()
Return a copy of β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

getBonds()
Returns bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

getCSLabels()
Returns coordinate set labels.

getCharges()
Return a copy of partial charges. Partial charges can be used in atom selections, e.g. `'charge 1','abs(charge) == 1','charge < 0'`.

getChids()
Return a copy of chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

getChindices()
Return a copy of chain indices. Chain indices are assigned to subsets of atoms with distinct pairs of chain identifier and segment name. Chain indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

getCoords()
Returns a copy of coordinates from the active coordinate set.

getCoordsets(*indices=None*)
Returns coordinate set(s) at given *indices*, which may be an integer or a list/array of integers.

getData(*label*)
Returns a copy of data associated with *label*, if it is present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getElements()
Return a copy of element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

getFlagLabels(*which=None*)
Returns flag labels. For `which='user'`, return labels of user or parser (e.g. *hetatm*) provided flags, for `which='all'` return all possible *Atom Flags* (page 64) labels in addition to those present in the instance.

getFlags(*label*)
Returns a copy of atom flags for given *label*, or **None** when flags for *label* is not set.

getFragindices()
Return a copy of fragment indices. Fragment indices are assigned to connected subsets of atoms. Bonds needs to be set using *AtomGroup.setBonds()* (page 44) or *AtomGroup.inferBonds()* (page 42). Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Fragment indices can be used in atom selections, e.g. `'fragindex 0','fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getIcodes()
Return a copy of insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

getIndices()
Returns a copy of the indices of atoms.

getMasses()
Return a copy of masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

getNames()
Return a copy of names. Names can be used in atom selections, e.g. `'name CA CB'`.

getOccupancies()
Return a copy of occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1','occupancy > 0'`.

getRadii()
Return a copy of radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

getResindices()
Return a copy of residue indices. Residue indices are assigned to subsets of atoms with distinct sequences of residue number, insertion code, chain identifier, and segment name. Residue indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Residue indices can be used in atom selections, e.g. `'resindex 0'`.

getResnames()
Return a copy of residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

getResnums()
Return a copy of residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3', 'resnum 120A 120B', 'resnum 10 to 20', 'resnum 10:20:2', 'resnum < 10'`. Note that *resid* is a synonym for *resnum*.

getSecclasses()
Return a copy of secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

getSecids()
Return a copy of secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

getSecindices()
Return a copy of secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

getSecstrs()
Return a copy of secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

getSegindices()
Return a copy of segment indices. Segment indices are assigned to subsets of atoms with distinct segment names. Segment indices start from zero, are incremented by one, and are assigned in the order of appearance in *AtomGroup* (page 38) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegnames()
Return a copy of segment names. Segment names can be used in atom selections, e.g. `'segment PROT','segname PROT'`. Note that *segname* is a synonym for *segment*.

getSequence(***kwargs*)
Returns one-letter sequence string for amino acids. When *allres* keyword argument is **True**, sequence will include all residues (e.g. water molecules) in the chain and **X** will be used for non-standard residue names.

getSerials()
Return a copy of serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

getTitle()
Returns title of the instance.

getTypes()
Return a copy of types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isFlagLabel(*label*)
Returns **True** if flags associated with *label* are present.

iterAcceptors()
Yield acceptors formed by the atom. Use `setAcceptors()` for setting acceptors.

iterAngles()
Yield angles formed by the atom. Use `setAngles()` for setting angles.

iterAtoms()
Yield atoms.

iterBonds()
Yield bonds formed by the atom. Use `setBonds()` or `inferBonds()` for setting bonds.

iterCoordsets()
Yield copies of coordinate sets.

iterCrossterms()
Yield crossterms formed by the atom. Use `setCrossterms()` for setting crossterms.

iterDihedrals()
Yield dihedrals formed by the atom. Use `setDihedrals()` for setting dihedrals.

iterDonors()
Yield donors formed by the atom. Use `setDonors()` for setting donors.

iterImpropers()
Yield impropers formed by the atom. Use `setImpropers()` for setting impropers.

iterNBExclusions()
Yield nbexclusions formed by the atom. Use `setNBExclusions()` for setting nbexclusions.

numAtoms(*flag=None*)
Returns number of atoms, or number of atoms with given *flag*.

numBonds()
Returns number of bonds. Use `setBonds()` or `inferBonds()` from parent AtomGroup for setting bonds.

numCoordsets()
Returns number of coordinate sets.

numResidues()
Returns number of residues.

select(*selstr*, ***kwargs*)
Returns atoms matching *selstr* criteria. See *select* (page 92) module documentation for details and usage examples.

setACSIndex(*index*)
Set coordinates at *index* active.

setAltlocs(*data*)
Set alternate location indicators. Alternate location indicators can be used in atom selections, e.g. `'altloc A B','altloc _'`.

setAnisous(*anisous*)
Set anisotropic temperature factors in the active coordinate set.

setAnistds(*data*)
Set standard deviations for anisotropic temperature factors.

setBetas(*data*)
Set β-values (or temperature factors). β-values can be used in atom selections, e.g. `'beta 555.55','beta 0 to 500','beta 0:500','beta < 500'`.

setCharges(*data*)
Set partial charges. Partial charges can be used in atom selections, e.g. `'charge 1', 'abs(charge) == 1','charge < 0'`.

setChids(*data*)
Set chain identifiers. Chain identifiers can be used in atom selections, e.g. `'chain A','chid A B C','chain _'`. Note that *chid* is a synonym for *chain*.

setCoords(*coords*)
Set coordinates in the active coordinate set.

setData(*label*, *data*)
Update *data* associated with *label*.

Raises **AttributeError**[191] – when *label* is not in use or read-only

setElements(*data*)
Set element symbols. Element symbols can be used in atom selections, e.g. `'element C O N'`.

[191] http://docs.python.org/library/exceptions.html#AttributeError

setFlags (*label*, *value*)
Update flag associated with *label*.

Raises **AttributeError**[192] – when *label* is not in use or read-only

setIcodes (*data*)
Set insertion codes. Insertion codes can be used in atom selections, e.g. `'icode A','icode _'`.

setMasses (*data*)
Set masses. Masses can be used in atom selections, e.g. `'12 <= mass <= 13.5'`.

setNames (*data*)
Set names. Names can be used in atom selections, e.g. `'name CA CB'`.

setOccupancies (*data*)
Set occupancy values. Occupancy values can be used in atom selections, e.g. `'occupancy 1', 'occupancy > 0'`.

setRadii (*data*)
Set radii. Radii can be used in atom selections, e.g. `'radii < 1.5','radii ** 2 < 2.3'`.

setResnames (*data*)
Set residue names. Residue names can be used in atom selections, e.g. `'resname ALA GLY'`.

setResnums (*data*)
Set residue numbers. Residue numbers can be used in atom selections, e.g. `'resnum 1 2 3','resnum 120A 120B','resnum 10 to 20','resnum 10:20:2','resnum < 10'`. Note that *resid* is a synonym for *resnum*.

setSecclasses (*data*)
Set secondary structure classs. Secondary structure classs can be used in atom selections, e.g. `'secclass 2','secclass -1'`.

setSecids (*data*)
Set secondary structure identifiers. Secondary structure identifiers can be used in atom selections, e.g. `'secid A B','secid 1 2'`.

setSecindices (*data*)
Set secondary structure indexs. Secondary structure indexs can be used in atom selections, e.g. `'s','e','c','i','n','d','e','x',' ','2'`.

setSecstrs (*data*)
Set secondary structure assignments. Secondary structure assignments can be used in atom selections, e.g. `'secondary H E','secstr H E'`. Note that *secstr* is a synonym for *secondary*.

setSegnames (*data*)
Set segment names. Segment names can be used in atom selections, e.g. `'segment PROT', 'segname PROT'`. Note that *segname* is a synonym for *segment*.

setSerials (*data*)
Set serial numbers (from file). Serial numbers can be used in atom selections, e.g. `'serial 1 2 3','serial 1 to 10','serial 1:10:2','serial < 10'`.

setTypes (*data*)
Set types. Types can be used in atom selections, e.g. `'type CT1 CT2 CT3'`.

toAtomGroup ()
Returns a copy of atoms (and atomic data) in an *AtomGroup* (page 38) instance.

[192] http://docs.python.org/library/exceptions.html#AttributeError

toBioPythonStructure(*header=None*, ***kwargs*)
Returns a `Bio.PDB.Structure` object

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets

toTEMPyAtoms()
Returns a `TEMPy.protein.prot_rep_biopy.Atom` or list of them as appropriate

toTEMPyStructure()
Returns a `protein.prot_rep_biopy.Structure` object

3.2 Chromatin Dynamics Analysis

This module defines classes and functions to parse and write Hi-C data files, visualize, and perform elastic network analysis on Hi-C data.

3.2.1 Parse/write Hi-C files

Following ProDy functions are for parsing and writing Hi-C files:

- *parseHiC()* (page 114) - parse Hi-C data file
- *parseHiCStream()* (page 114) - parse Hi-C data stream
- *writeMap()* (page 114) - write Hi-C data to text file

3.2.2 Visualize Hi-C data

Following ProDy functions are for visualizing Hi-C data:

- `showMap()` - show Hi-C contact map
- *showDomains()* (page 113) - show Hi-C structural domains

Save/load HiC class

- *saveHiC()* (page 114)
- *loadHiC()* (page 114)

3.2.3 Chromatin clustering

calcGNMDomains(*modes*, *method=<function Discretize>*, ***kwargs*)
Uses spectral clustering to separate structural domains in chromosomes and proteins.

Parameters

- **modes** (`ModeSet`) – GNM modes used for segmentation
- **method** (*func*) – Label assignment algorithm used after Laplacian embedding of loci.

KMeans(*V*, ***kwargs*)
Performs k-means clustering on *V*. The function uses `sklearn.cluster.KMeans()`. See sklearn documents for details.

Parameters

- **V** (ndarray) – row-normalized eigenvectors for the purpose of clustering.
- **n_clusters** (*int*[193]) – specifies the number of clusters.

Hierarchy(*V*, ***kwargs*)

Performs hierarchical clustering on *V*. The function essentially uses two scipy functions: `linkage` and `fcluster`. See `scipy.cluster.hierarchy.linkage()`[194] and `scipy.cluster.hierarchy.fcluster()`[195] for the explaination of the arguments. Here lists arguments that are different from those of scipy.

Parameters

- **V** (ndarray) – row-normalized eigenvectors for the purpose of clustering.
- **inconsistent_percentile** – if the clustering *criterion* for `scipy.cluster.hierarchy.fcluster()`[196]

is `inconsistent` and threshold *t* is not given (default), then the function will use the percentile specified by this argument as the threshold. :type inconsistent_percentile: double

Parameters **n_clusters** – specifies the maximal number of clusters. If this argument is given, then the function will

automatically set *criterion* to `maxclust` and *t* equal to *n_clusters*. :type n_clusters: int

Discretize(*V*, ***kwargs*)

Adapted from `discretize()`. Copyright please see LICENSE.rst.

showLinkage(*V*, ***kwargs*)

Shows the dendrogram of hierarchical clustering on *V*. See `scipy.cluster.hierarchy.dendrogram()`[197] for details.

Parameters **V** (ndarray) – row-normalized eigenvectors for the purpose of clustering.

GaussianMixture(*V*, ***kwargs*)

Performs clustering on *V* by using Gaussian mixture models. The function uses `sklearn.micture.GaussianMixture()`. See sklearn documents for details.

Parameters

- **V** (ndarray) – row-normalized eigenvectors for the purpose of clustering.
- **n_clusters** (*int*[198]) – specifies the number of clusters.

BayesianGaussianMixture(*V*, ***kwargs*)

Performs clustering on *V* by using Gaussian mixture models with variational inference. The function uses `sklearn.micture.GaussianMixture()`. See sklearn documents for details.

Parameters

- **V** (ndarray) – row-normalized eigenvectors for the purpose of clustering.
- **n_clusters** (*int*[199]) – specifies the number of clusters.

[193] http://docs.python.org/library/functions.html#int
[194] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.cluster.hierarchy.linkage.html#scipy.cluster.hierarchy.linkage
[195] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.cluster.hierarchy.fcluster.html#scipy.cluster.hierarchy.fcluster
[196] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.cluster.hierarchy.fcluster.html#scipy.cluster.hierarchy.fcluster
[197] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.cluster.hierarchy.dendrogram.html#scipy.cluster.hierarchy.dendrogram
[198] http://docs.python.org/library/functions.html#int
[199] http://docs.python.org/library/functions.html#int

3.2.4 Chromatin functions

showDomains(*domains*, *linespec='-'*, ***kwargs*)
A convenient function that can be used to visualize Hi-C structural domains. *kwargs* will be passed to `matplotlib.pyplot.plot()`[200].

Parameters **domains** (`numpy.ndarray`) – a 2D array of Hi-C domains, such as [[start1, end1], [start2, end2], ...].

showEmbedding(*modes*, *labels=None*, *trace=True*, *headtail=True*, *cmap='prism'*)
Visualizes Laplacian embedding of Hi-C data.

Parameters **modes** – modes in which loci are embedded. It can only have 2 or 3 modes for the purpose

of visualization. :type modes: *ModeSet* (page 171)

Parameters

- **labels** (*list*[201]) – a list of integers indicating the segmentation of the sequence.
- **trace** (*bool*[202]) – if **True** then the trace of the sequence will be indicated by a grey dashed line.
- **headtail** – if **True** then a star and a closed circle will indicate the head and the tail

of the sequence respectively. :type headtail: bool

Parameters **cmap** (*str*[203]) – the color map used to render the *labels*.

getDomainList(*labels*)
Returns a list of domain separations. The list has two columns: the first is for the domain starts and the second is for the domain ends.

3.2.5 HiC

class HiC(*title='Unknown'*, *map=None*, *bin=None*)
This class is used to store and preprocess Hi-C contact map. A *GNM* (page 164) instance for analyzing the contact map can be also created by using this class.

calcGNM(*n_modes=None*, ***kwargs*)
Calculates GNM on the current Hi-C map. By default, `n_modes` is set to **None** and `zeros` to **True**.

getCompleteMap()
Obtains the complete contact map with unmapped regions.

getDomainList()
Returns a list of domain separations. The list has two columns: the first is for the domain starts and the second is for the domain ends.

getDomains()
Returns an 1D `numpy.ndarray` whose length is the number of loci. Each element is an index denotes to which domain the locus belongs.

getKirchhoff()
Builds a Kirchhoff matrix based on the contact map.

[200] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
[201] http://docs.python.org/library/stdtypes.html#list
[202] http://docs.python.org/library/functions.html#bool
[203] http://docs.python.org/library/stdtypes.html#str

getTitle()
Returns title of the instance.

getTrimedMap()
Obtains the contact map without unmapped regions.

normalize(*method=<function VCnorm>*, ***kwargs*)
Applies chosen normalization on the current Hi-C map.

setDomains(*labels*, ***kwargs*)
Uses spectral clustering to identify structural domains on the chromosome.

Parameters

- **labels** (ndarray, list) – domain labels
- **method** (*func*) – Label assignment algorithm used after Laplacian embedding.

setTitle(*title*)
Sets title of the instance.

view(*spec='p'*, ***kwargs*)
Visualization of the Hi-C map and domains (if present). The function makes use of *showMatrix()* (page 304).

Parameters

- **spec** (*str*[204]) – a string specifies how to preprocess the matrix. Blank for no preprocessing, 'p' for showing only data from *p*-th to *100-p*-th percentile. '_' is to suppress creating a new figure and paint to the current one instead. The letter specifications can be applied sequentially, e.g. 'p_'.
- **p** (*double*) – specifies the percentile threshold.

parseHiC(*filename*, ***kwargs*)
Returns an *HiC* (page 113) from a Hi-C data file.

This function extends *parseHiCStream()* (page 114).

Parameters **filename** (*str*[205]) – the filename to the Hi-C data file.

parseHiCStream(*stream*, ***kwargs*)
Returns an *HiC* (page 113) from a stream of Hi-C data lines.

Parameters **stream** – Anything that implements the method read, seek (e.g. file, buffer, stdin)

saveHiC(*hic*, *filename=None*, *map=True*, ***kwargs*)
Saves *HiC* model data as filename.hic.npz. If *map* is **True**, Hi-C contact map will not be saved and it can be loaded from raw data file later. If *filename* is **None**, name of the Hi-C instance will be used as the filename, after " " (white spaces) in the name are replaced with "_" (underscores). Upon successful completion of saving, filename is returned. This function makes use of numpy.savez() function.

loadHiC(*filename*)
Returns HiC instance after loading it from file (*filename*). This function makes use of numpy.load() function. See also *saveHiC()* (page 114).

writeMap(*filename*, *map*, *bin=None*, *format='%f'*)
Writes *map* to the file designated by *filename*.

Parameters

[204] http://docs.python.org/library/stdtypes.html#str
[205] http://docs.python.org/library/stdtypes.html#str

- **filename** (*str*[206]) – the file to be written.
- **map** (numpy.ndarray) – a Hi-C contact map.
- **bin** (*int*[207]) – bin size of the *map*. If bin is *None*, *map* will be written in full matrix format.
- **format** (*str*[208]) – output format for map elements.

3.2.6 Chromatin normalization

VCnorm(*M*, ***kwargs*)
Performs vanilla coverage normalization on matrix *M*.

SQRTVCnorm(*M*, ***kwargs*)
Performs square-root vanilla coverage normalization on matrix *M*.

Filenorm(*M*, ***kwargs*)
Performs normalization on matrix *M* given a file. *filename* specifies the path to the file. The file should be one-column, and ideally has the same number of entries with the size of *M* (extra entries will be ignored). Say *F* is vector of the normalization factors, *N* is the normalized matrix, if *expected* is **True**, `N[i,j] = M[i,j]/F[i]/F[j]`. If *expected* is **True**, `N[i,j] = M[i,j]/F[|i-j|]`.

SCN(*M*, ***kwargs*)
Performs Sequential Component Normalization on matrix *M*.

3.2.7 Straw HiC API

Straw module

Straw enables programmatic access to .hic files. .hic files store the contact matrices from Hi-C experiments and the normalization and expected vectors, along with meta-data in the header.

The main function, straw, takes in the normalization, the filename or URL, chromosome1 (and optional range), chromosome2 (and optional range), whether the bins desired are fragment or base pair delimited, and bin size.

It then reads the header, follows the various pointers to the desired matrix and normalization vector, and stores as [x, y, count]

Usage: straw <NONE/VC/VC_SQRT/KR> <hicFile(s)> <chr1>[:x1:x2] <chr2>[:y1:y2] <BP/FRAG> <binsize>

Example: >>>import straw >>>result = straw.straw('NONE', 'HIC001.hic', 'X', 'X', 'BP', 1000000) >>>for i in range(len(result[0])): ... print("{0} {1} {2}".format(result[0][i], result[1][i], result[2][i]))

See https://github.com/theaidenlab/straw/wiki/Python for more documentation

getBlockNumbersForRegionFromBinPosition(*regionIndices*, *blockBinCount*, *blockColumnCount*, *intra*)
Gets the block numbers we will need for a specific region; used when the range to extract is sent in as a parameter

Args: regionIndices (array): Array of ints giving range blockBinCount (int): The block bin count of the matrix blockColumnCount (int): The block column count of the matrix intra: Flag indicating if this is an intrachromosomal matrix

Returns: blockSet (set): A set of blocks to print

[206] http://docs.python.org/library/stdtypes.html#str
[207] http://docs.python.org/library/functions.html#int
[208] http://docs.python.org/library/stdtypes.html#str

printme(*norm*, *infile*, *chr1loc*, *chr2loc*, *unit*, *binsize*, *outfile*)
Reads a .hic file and extracts and prints the given contact matrix to a text file

Args: norm(str): Normalization type, one of VC, KR, VC_SQRT, or NONE infile(str): File name or URL of .hic file chr1loc(str): Chromosome name and (optionally) range, i.e. "1" or "1:10000:25000" chr2loc(str): Chromosome name and (optionally) range, i.e. "1" or "1:10000:25000" unit(str): One of BP or FRAG binsize(int): Resolution, i.e. 25000 for 25K outfile(str): Name or stream of text file to write to

readBlock(*req*, *size*)
Reads the block - reads the compressed bytes, decompresses, and stores results in array. Presumes file pointer is in correct position.

Args: req (file): File to read from. Presumes file pointer is in correct position size (int): How many bytes to read

Returns: array containing row, column, count data for this block

readFooter(*req*, *c1*, *c2*, *norm*, *unit*, *resolution*)
Reads the footer, which contains all the expected and normalization vectors. Presumes file pointer is in correct position Args:

req (file): File to read from; presumes file pointer is in correct position chr1 (str): Chromosome 1 chr2 (str): Chromosome 2 norm (str): Normalization type, one of NONE, VC, KR, VC_SQRT unit (str): One of BP or FRAG resolution (int): Bin size

Returns:

list: File position of matrix, position+size chr1 normalization vector, position+size chr2 normalization vector

readHeader(*req*, *chr1*, *chr2*, *posilist*)
Reads the header

Args: req (file): File to read from chr1 (str): Chromosome 1 chr2 (str): Chromosome 2 c1pos1 (int, optional): Starting range of chromosome1 output c1pos2 (int, optional): Stopping range of chromosome1 output c2pos1 (int, optional): Starting range of chromosome2 output c2pos2 (int, optional): Stopping range of chromosome2 output

Returns: list: master index, chromosome1 index, chromosome2 index

readMatrix(*req*, *unit*, *binsize*)
Reads the matrix - that is, finds the appropriate pointers to block data and stores them. Needs to read through headers of zoom data to find appropriate matrix. Presumes file pointer is in correct position.

Args: req (file): File to read from; presumes file pointer is in correct position unit (str): Unit to search for (BP or FRAG) binsize (int): Resolution to search for

Returns: list containing block bin count and block column count of matrix

readMatrixZoomData(*req*, *myunit*, *mybinsize*)
Reads the Matrix Zoom Data, which gives pointer list for blocks for the data. Presumes file pointer is in correct position

Args: req (file): File to read from; presumes file pointer is in correct position myunit (str): Unit (BP or FRAG) we're searching for mybinsize (int): Resolution we're searching for

Returns: list containing boolean indicating if we found appropriate matrix, and if so, the counts for the bins and columns

readNormalizationVector(*req*)
Reads the normalization vector from the file; presumes file pointer is in correct position

Args: req (file): File to read from; presumes file pointer is in correct position

Returns: Array of normalization values

straw(*norm*, *infile*, *chr1loc*, *chr2loc*, *unit*, *binsize*)

This is the main workhorse method of the module. Reads a .hic file and extracts the given contact matrix. Stores in an array in sparse upper triangular format: row, column, (normalized) count

Args: norm(str): Normalization type, one of VC, KR, VC_SQRT, or NONE infile(str): File name or URL of .hic file chr1loc(str): Chromosome name and (optionally) range, i.e. "1" or "1:10000:25000" chr2loc(str): Chromosome name and (optionally) range, i.e. "1" or "1:10000:25000" unit(str): One of BP or FRAG binsize(int): Resolution, i.e. 25000 for 25K

3.3 Compounds Analysis

This module defines classes and functions to fetch, parse, and write structural data files associated with compounds and to access and search structural databases, e.g. ProteinDataBank[209].

3.3.1 Ligand data

The following function can be used to fetch metadata on PDB ligands:

- *fetchPDBLigand()* (page 119) - retrieve ligand from Ligand-Expo

3.3.2 BIRD data

The following functions can be used to fetch metadata from the Biologically Interesting Reference Dictionary (BIRD) for peptide-like ligands:

- *fetchBIRDviaFTP()* (page 117) - retrieve BIRD data from PDB
- *parseBIRD()* (page 118) - parse BIRD data for a particular compound or family

3.3.3 CCD data

The following function can be used to fetch metadata from the Chemical Component Dictionary (CCD) for residues within ligands:

- *parseCCD()* (page 118) - parse CCD data for a particular component

3.3.4 Functions

The following functions can be used to calculate 2D similarity using Morgan Fingerprints:

- *calc2DSimilarity()* (page 118) - calculate 2D similarity for a pair of SMILES
- `calc2DSimilarityMatrix()` - calculate 2D similarity matrix for a collection of SMILES

3.3.5 BIRD Classes and Functions

This module defines functions for fetching and parsing files in the PDB for the Biologically Interesting Molecule Reference Dictionary (BIRD[210]).

The chemical information is stored in Peptide Reference Dictionary (PRD) files, whereas the biological function is documented in a separate family file.

[209] http://wwpdb.org
[210] https://www.wwpdb.org/data/bird

fetchBIRDviaFTP(***kwargs*)
Retrieve the whole Biologically Interesting Molecule Reference Dictionary (BIRD) resource, which is updated every week. This includes 2 kinds of keys, which can be selected with the **keys** keyword argument.

The chemical information is found in a zipped (tar.gz) directory at https://files.rcsb.org/pub/pdb/data/bird/prd/prd-all.cif.gz, which contains individual CIF files within it. This data will be downloaded and extracted to `.prody/bird-prd`.

Biological function information is also found in a zipped (tar.gz) directory at https://files.rcsb.org/pub/pdb/data/bird/family/family-all.cif.gz, which contains individual CIF files within it. This data will be downloaded and extracted to `.prody/bird-family`.

Parameters **keys** (str, tuple, list, `ndarray`) – keys specifying which data to fetch out of `'prd'`,`'family'` or `'both'` default is `'both'`

The underlying data can be accessed using *parseBIRD()* (page 118).

parseBIRD(**ids*, ***kwargs*)
Parse data from the Biologically Interesting Molecule Reference Dictionary (BIRD) resource, which is updated every week. This includes 2 kinds of keys, which can be selected with the **keys** keyword argument.

The chemical information is found in a single CIF file at https://files.rcsb.org/pub/pdb/data/bird/prd/prd-all.cif.gz. This data will be downloaded and extracted to `.prody/bird-prd`.

Biological function information is also found in a single CIF file at https://files.rcsb.org/pub/pdb/data/bird/family/family-all.cif.gz. This data will be downloaded and extracted to `.prody/bird-family`.

Individual compounds can be selected using **ids**. If needed, BIRD files are downloaded using *fetchBIRDviaFTP()* (page 117) function.

You can also provide arguments that you would like passed on to fetchBIRDviaFTP.

Parameters

- **ids** (str, tuple, list, `ndarray`, **None**) – one BIRD identifier (starting with PRD or FAM) or a list of them. If **None** is provided then all of them are returned.
- **key** (*str*[211]) – key specifying which data to fetch out of `'prd'` or `'family'` default is `'prd'`

Returns *StarDataBlock* (page 271) object or list of them.

3.3.6 CCD Classes and Functions

This module defines functions for fetching and parsing files in the PDB for the Chemical Compound Dictionary (CCD[212]).

parseCCD(*ids*)
Retrieve the whole Chemical Component Dictionary (CCD) resource.

3.3.7 Supporting Functions

This module defines functions for using compounds from the PDB and elsewhere.

calc2DSimilarity(*smiles1*, *smiles2*)
Calculate 2D similarity using Morgan Fingerprints

[211] http://docs.python.org/library/stdtypes.html#str
[212] https://www.wwpdb.org/data/ccd

Parameters

- **smiles1** (str, *PDBLigandRecord* (page 119)) – first SMILES string or PDBLigandRecord containing one
- **smiles2** (str, *PDBLigandRecord* (page 119)) – second SMILES string or PDBLigandRecord containing one

3.3.8 PDB Ligands

This module defines functions for fetching PDB ligand data.

class PDBLigandRecord(*data*)

Class for handling the output of fetchPDBLigand

getCanonicalSMILES()

fetchPDBLigand(*cci*, *filename=None*)

Fetch PDB ligand data from PDB[213] for chemical component *cci*. *cci* may be 3-letter chemical component identifier or a valid XML filename. If *filename* is given, XML file will be saved with that name.

If you query ligand data frequently, you may configure ProDy to save XML files in your computer. Set `ligand_xml_save` option **True**, i.e. `confProDy(ligand_xml_save=True)`. Compressed XML files will be save to ProDy package folder, e.g. `/home/user/.prody/pdbligands`. Each file is around 5Kb when compressed.

This function is compatible with PDBx/PDBML v 4.0.

Ligand data is returned in a dictionary. Ligand coordinate atom data with *model* and *ideal* coordinate sets are also stored in this dictionary. Note that this dictionary will contain data that is present in the XML file and all Ligand Expo XML files do not contain every possible data field. So, it may be better if you use `dict.get()`[214] instead of indexing the dictionary, e.g. to retrieve formula weight (or relative molar mass) of the chemical component use `data.get('formula_weight')` instead of `data['formula_weight']` to avoid exceptions when this data field is not found in the XML file. URL and/or path of the XML file are returned in the dictionary with keys `url` and `path`, respectively.

Following example downloads data for ligand STI (a.k.a. Gleevec and Imatinib) and calculates RMSD between model (X-ray structure 1IEP) and ideal (energy minimized) coordinate sets:

```
In [1]: from prody import *

In [2]: ligand_data = fetchPDBLigand('STI')
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-2-65da2e637be2> in <module>()
----> 1 ligand_data = fetchPDBLigand('STI')

/home/exx/ProDy-website/ProDy/prody/compounds/pdbligands.pyc in fetchPDBLigand(cci, filename)
     95             except IOError:
     96                 raise IOError('XML file for ligand {0} is not found online'
---> 97                               .format(cci))
     98             else:
     99                 xml = inp.read()

IOError: XML file for ligand STI is not found online

In [3]: ligand_data['model_coordinates_db_code']
---------------------------------------------------------------------------
```

[213] http://www.pdb.org
[214] http://docs.python.org/library/stdtypes.html#dict.get

```
NameError                                 Traceback (most recent call last)
<ipython-input-3-87a0a2c276f6> in <module>()
----> 1 ligand_data['model_coordinates_db_code']

NameError: name 'ligand_data' is not defined

In [4]: ligand_model = ligand_data['model']
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-c3b5fe43bd1d> in <module>()
----> 1 ligand_model = ligand_data['model']

NameError: name 'ligand_data' is not defined

In [5]: ligand_ideal = ligand_data['ideal']
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-3a4a6d842406> in <module>()
----> 1 ligand_ideal = ligand_data['ideal']

NameError: name 'ligand_data' is not defined

In [6]: transformation = superpose(ligand_ideal.noh, ligand_model.noh)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-47d2aaf39930> in <module>()
----> 1 transformation = superpose(ligand_ideal.noh, ligand_model.noh)

NameError: name 'ligand_ideal' is not defined

In [7]: calcRMSD(ligand_ideal.noh, ligand_model.noh)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-3a9f73b6f35d> in <module>()
----> 1 calcRMSD(ligand_ideal.noh, ligand_model.noh)

NameError: name 'ligand_ideal' is not defined
```

parsePDBLigand(*cci*, *filename=None*)
See *fetchPDBLigand()* (page 119)

3.4 Database Support

This module contains features for accessing databases containing protein related data.

3.4.1 Pfam

The following functions can be used to search and retrieve Pfam[215] data:

- *fetchPfamMSA()* (page 126) - download MSA files
- *searchPfam()* (page 126) - search for domain families of a protein

[215] https://www.ebi.ac.uk/interpro/entry/pfam/

3.4.2 UniProt

The following functions and class can be used to search and retrieve UniProt[216] data:

- *queryUniprot()* (page 131) - query UniProt and parse the results as a dictionary
- *UniprotRecord* (page 131) - a wrapper from UniProt data with functions including parsing PDBs
- *searchUniprot()* (page 131) - search UniProt and return a UniprotRecord

3.4.3 CATH

The following class and its functions can be used to search and retrieve CATH[217] data:

- *CATHDB* (page 122) - parse, handle and navigate the tree-like structure of the CATH database

3.4.4 DALI

The following class and functions can be used to search and retrieve data using the DALI[218] structure alignment server:

- *searchDali()* (page 124) - search for similar structures using DALI
- *DaliRecord* (page 122) - fetch and handle outputs from DALI searches
- *daliFilterMultimers()* (page 124) - filter DALI results to obtain multimers of a particular size

3.4.5 QuartataWeb

The following classes and functions can be used to search and retrieve data using the QuartataWeb[219] structure alignment server:

- *QuartataWebBrowser* (page 127) - class based on the Splinter web browser package to search QuartataWeb
- *QuartataChemicalRecord* (page 130) - class to handle the outputs of QuartataWeb searches
- *searchQuartataWeb()* (page 130) - perform QuartataWeb searches and return the output in a QuartataChemicalRecord

3.4.6 Gene Ontology Annotation (GOA)

The following classes and functions can be used to search and retrieve data from the EBI GOA[220] database:

- *queryGOA()* (page 125) - query GOA using a PDB ID
- *GOADictList* (page 124) - class to handle data from GOA queries
- *parseOBO()* (page 124) - parse an OBO file containing the Gene Ontology.
- *parseGAF()* (page 124) - parse a Gene Association File (GAF)
- *showGoLineage()* (page 125) - visualize GO tree
- *calcGoOverlap()* (page 125) - Calculate overlap between GO terms from their distance in the graph

[216] https://www.uniprot.org/
[217] http://download.cathdb.info
[218] http://ekhidna2.biocenter.helsinki.fi/dali/
[219] http://quartata.csb.pitt.edu
[220] https://www.ebi.ac.uk/GOA/

3.4.7 Interpro

The following functions can be used to search and retrieve Interpro[221] data:

- `searchInterpro()` - search for domain families of a protein

3.4.8 BioExcel-CV19

The following functions can be used to retrieve BioExcel-CV19[222] data:

- `fetchBioexcelPDB()` - fetch PDB files for starting structures for trajectories

3.4.9 CATH Access Functions

This module defines functions for query CATH database.

class **CATHDB**(*source='http://download.cathdb.info'*)

This class is for handing the data in the CATH database. It facilitates tree navigation and ID-based searching.

The class is initialised using data from *source*, which is downloaded from the CATH website by default, but can also be parsed from an XML file.

find(*identifier='root'*)

Find an `xml.etree.Element` object given *identifier*.

save(*filename='cath.xml'*)

Write local CATH database to an XML file. *filename* can either be a file name or a handle.

search(*identifier*)

Search the *identifier* against the local CATH database and return a list of `ElementTree` instances. *identifier* can be a 4-digit PDB ID plus an 1-digit chain ID (optional), 7-digit domain ID, or CATH ID.

update(*source=None*)

Update data and files from CATH.

class **CATHElement**(*tag*, *attrib={}*, *parent=None*, ***extra*)

This class handles individual elements from the cath tree including tree navigation and the getting of associated CATH domains, PDB IDs and selection strings.

parsePDBs(***kwargs*)

Load PDB into memory as *AtomGroup* (page 38) instances using *parsePDB()* (page 268) and perform selection based on residue ranges given by CATH.

3.4.10 Dali Server Functions

This module defines functions for Dali searching Protein Data Bank.

class **DaliRecord**(*url*, *pdbId*, *chain*, *subset='fullPDB'*, *localFile=False*, ***kwargs*)

A class to store results from Dali PDB search.

Instantiate a DaliRecord object instance.

Parameters

- **url** – url of Dali results page or local dali results file
- **pdbId** – PDB code for searched protein

[221] https://www.ebi.ac.uk/interpro/
[222] https://bioexcel-cv19-dev.bsc.es/

- **chain** – chain identifier (only one chain can be assigned for PDB)
- **subset** – fullPDB, PDB25, PDB50, PDB90. Ignored if localFile=True (url is a local file)
- **localFile** – whether provided url is a path for a local dali results file

fetch(*url=None*, *localFile=False*, ***kwargs*)
Get Dali record from url or file.

Parameters

- **url** (*str*[223]) – url of Dali results page or local dali results file If None then the url already associated with the DaliRecord object is used.
- **localFile** (*bool*[224]) – whether provided url is a path for a local dali results file
- **timeout** (*int*[225]) – amount of time until the query times out in seconds default value is 120
- **localfolder** (*str*[226]) – folder in which to find the local file default is the current folder

filter(*cutoff_len=None*, *cutoff_rmsd=None*, *cutoff_Z=None*, *cutoff_identity=None*, *stringency=False*)
Filters a PDBList by a given set of cutoffs and returns a list of PDB IDs where PDBs that matched have been removed. PDBs that satisfy _any_ of the following criteria will be filtered out. (1) Length of aligned residues < cutoff_len (must be an integer or a float between 0 and 1); (2) RMSD < cutoff_rmsd (must be a positive number); (3) Z score < cutoff_Z (must be a positive number); (4) Identity > cutoff_identity (must be an integer or a float between 0 and 1).

Setting the stringency flag to True changes the behavior so that all filtering is by stringency. In this case, PDBs that satisfy _any_ of the following criteria will be filtered out:

1. Length of aligned residues < cutoff_len (must be an integer or a float between 0 and 1); (unchanged)
2. RMSD > cutoff_rmsd (must be a positive number); (inverted)
3. Z score < cutoff_Z (must be a positive number); (unchanged)
4. Identity < cutoff_identity (must be an integer or a float between 0 and 1). (inverted)

Leaving stringency False maintains old behavior.

getFilterList()
Returns a list of PDB IDs and chains for the entries that were filtered out

getHits()
Returns the dictionary associated with the DaliRecord

getMapping(*key*)
Get mapping for a particular entry in the DaliRecord

getMappings()
Get all mappings in the DaliRecord

getPDBs(*filtered=True*)
Returns PDB list (filters may be applied)

[223] http://docs.python.org/library/stdtypes.html#str
[224] http://docs.python.org/library/functions.html#bool
[225] http://docs.python.org/library/functions.html#int
[226] http://docs.python.org/library/stdtypes.html#str

getTitle()
Return the title of the record

mappings
Get all mappings in the DaliRecord

searchDali(*pdb*, *chain=None*, *subset='fullPDB'*, *daliURL=None*, ***kwargs*)
Search Dali server with input of PDB ID (or local PDB file) and chain ID. Dali server: http://ekhidna2.biocenter.helsinki.fi/dali/

Parameters

- **pdb** – PDB code or local PDB file for the protein to be searched
- **chain** (*str*[227]) – chain identifier (only one chain can be assigned for PDB)
- **subset** (*str*[228]) – fullPDB, PDB25, PDB50, PDB90

daliFilterMultimer(*atoms*, *dali_rec*, *n_chains=None*)
Filters multimers to only include chains with Dali mappings.

Parameters

- **atoms** (*Atomic* (page 47)) – the multimer to be filtered
- **dali_rec** (*DaliRecord* (page 122)) – the DaliRecord object with which to filter chains

daliFilterMultimers(*structures*, *dali_rec*, *n_chains=None*)
A wrapper for daliFilterMultimer to apply to multiple structures.

3.4.11 Gene Ontology Annotation (GOA) Server Functions

This module defines functions for interfacing with the EBI's Gene Ontology Annotation (GOA) database for analysing gene/protein functions through the Gene Ontology (GO).

This module is based on the tutorial notebook at https://nbviewer.jupyter.org/urls/dessimozlab.github.io/go-handbook/GO%20Tutorial%20in%20Python%20-%20Solutions.ipynb

class GOADictList(*parsingList*, *title='unnamed'*, ***kwargs*)
A class for handling the list of GOA Dictionaries returned by queryGOA

pop(*index*)
Pop dataBlock with the given index from the list of dataBlocks in GOADictList

parseOBO(***kwargs*)
Parse a GO OBO file containing the GO itself. See OBO[229] for more information on the file format.

parseGAF(*database='PDB'*, ***kwargs*)
Parse a GO Association File (GAF) corresponding to a particular database collection into a dictionary for ease of querying.

See GAF[230] for more information on the file format

Parameters

- **database** (*str*[231]) – name of the database of interest default is PDB. Others include UNIPROT and common names of many organisms.

[227] http://docs.python.org/library/stdtypes.html#str
[228] http://docs.python.org/library/stdtypes.html#str
[229] http://owlcollab.github.io/oboformat/doc/obo-syntax.html
[230] http://geneontology.org/docs/go-annotation-file-gaf-format-2.1/
[231] http://docs.python.org/library/stdtypes.html#str

- **filename** (*str*[232]) – filename for the gaf of interest default is goa_ and the database name in lower case and .gaf.gz

queryGOA(**ids*, ***kwargs*)

Query a GOA database by identifier.

Parameters

- **ids** (str, tuple, list, ndarray) – an identifier or a list-like of identifiers
- **database** (*str*[233]) – name of the database of interest default is PDB. Others include UNIPROT and common names of many organisms.

showGoLineage(*go_term*, ***kwargs*)

Use pygraphviz and IPython notebook to show the lineage of a GO term

Parameters **go** (*~goatools.obo_parser.GODag*) – object containing a gene ontology (GO) directed acyclic graph (DAG) default is to parse with *parseOBO()* (page 124)

arg out_format: format for output. Currently only output to file. This file will be displayed in Jupyter Notebook.

type out_format: str

arg filename: filename for output default behaviour is to use the GO term ID and append '_lineage.png'

type filename: str

calcGoOverlap(**go_terms*, ***kwargs*)

Calculate overlap between GO terms based on their distance in the graph. GO terms in different namespaces (molecular function, cellular component, and biological process) have undefined distances.

Parameters

- **go_terms** (list, tuple, *~numpy.ndarray*) – a list of GO terms or GO IDs
- **pairwise** (*bool*[234]) – whether to calculate to a matrix of pairwise overlaps default is False
- **distance** (*bool*[235]) – whether to return distances rather than calculating overlaps default is False
- **go** (*~goatools.obo_parser.GODag*) – GO graph. Default behaviour is to parse it with *parseOBO()* (page 124).

calcDeepFunctionOverlaps(**goa_data*, ***kwargs*)

Calculate function overlaps between the deep (most detailed) molecular functions in particular from two sets of GO terms.

Parameters

- **goa1** (tuple, list, ndarray) – the first set of GO terms
- **goa2** (tuple, list, ndarray) – the second set of GO terms

[232] http://docs.python.org/library/stdtypes.html#str
[233] http://docs.python.org/library/stdtypes.html#str
[234] http://docs.python.org/library/functions.html#bool
[235] http://docs.python.org/library/functions.html#bool

calcEnsembleFunctionOverlaps(*ens*, ***kwargs*)
Calculate function overlaps for an ensemble as the mean of the value from *calcDeepFunctionOverlaps()* (page 125).

Parameters **ens** (Ensemble) – an ensemble with labels

findDeepestFunctions(*go_terms*, ***kwargs*)
Find the deepest (most detailed) molecular functions in a list of GO terms.

Parameters **go_terms** (*GOADictList* (page 124)) – a list of GO terms

findDeepestCommonAncestor(*terms*, *go*)
Find the nearest common ancestor. Only returns single most specific - assumes unique exists.

Parameters

- **terms** (tuple, list, ndarray) – a list of GO terms
- **go** (*~goatools.obo_parser.GODag*) – object containing a gene ontology (GO) directed acyclic graph (DAG)

calcMinBranchLength(*go_id1*, *go_id2*, *go*)
Find the minimum branch length between two terms in the GO DAG.

Parameters

- **go_id1** (*str*[236]) – the first GO ID
- **go_id2** – the second GO ID

:type go_id2:str

Parameters **go** (*~goatools.obo_parser.GODag*) – object containing a gene ontology (GO) directed acyclic graph (DAG)

findCommonParentGoIds(*terms*, *go*)
This function finds the common ancestors in the GO tree of the list of terms in the input.

Parameters

- **terms** (tuple, list, ndarray) – a list of GO terms
- **go** (*~goatools.obo_parser.GODag*) – object containing a gene ontology (GO) directed acyclic graph (DAG)

3.4.12 Pfam Access Functions

This module defines functions for interfacing Pfam database.

searchPfam(*query*, ***kwargs*)
Returns Pfam search results in a dictionary. Matching Pfam accession as keys will map to evalue, alignment start and end residue positions.

Parameters

- **query** (*str*[237]) – UniProt ID or PDB identifier with or without a chain identifier, e.g. '1mkp' or '1mkpA'. UniProt ID of the specified chain, or the first protein chain will be used for searching the Pfam database
- **timeout** (*int*[238]) – timeout for blocking connection attempt in seconds, default is 60

[236] http://docs.python.org/library/stdtypes.html#str
[237] http://docs.python.org/library/stdtypes.html#str
[238] http://docs.python.org/library/functions.html#int

fetchPfamMSA(*acc*, *alignment='seed'*, *compressed=False*, ***kwargs*)
Returns a path to the downloaded Pfam MSA file.

Parameters

- **acc** (*str*[239]) – Pfam ID or Accession Code
- **alignment** – alignment type, one of `'full'`, `'seed'` (default), `'ncbi'`, `'metagenomics'`, `'rp15'`, `'rp35'`, `'rp55'`, `'rp75'` or `'uniprot'` where rp stands for representative proteomes. InterPro Pfam seems to only have seed alignments easily accessible in most cases
- **compressed** – gzip the downloaded MSA file, default is **False**
- **timeout** – timeout for blocking connection attempt in seconds, default is 60
- **outname** (*str*[240]) – out filename, default is input `'acc_alignment.format'`
- **folder** (*str*[241]) – output folder, default is `'.'`

parsePfamPDBs(*query*, *data=[]*, ***kwargs*)
Returns a list of *AtomGroup* (page 38) objects containing sections of chains that correspond to a particular PFAM domain family. These are defined by alignment start and end residue numbers.

Parameters

- **query** (*str*[242]) – Pfam ID, UniProt ID or PDB ID If a PDB ID is provided the corresponding UniProt ID is used. If this returns multiple matches then start or end must also be provided. This query is also used for label refinement of the Pfam domain MSA.
- **data** (*list*[243]) – If given the data list from the Pfam mapping table will be output through this argument.
- **start** (*int*[244]) – Residue number for defining the start of the domain. The PFAM domain that starts closest to this will be selected. Default is **1**
- **end** (*int*[245]) – Residue number for defining the end of the domain. The PFAM domain that ends closest to this will be selected.

3.4.13 QuartataWeb Server Functions

This module defines classes and functions for browsing QuartataWeb.

Based on code written by the CHARMM-GUI team (http://charmm-gui.org) and modified by James Krieger

This suite uses the following softwares: - python Splinter package (https://splinter.readthedocs.org/en/latest/) - a web browser, such as Google Chrome or Mozilla Firefox - the corresponding driver such as chromedriver (https://sites.google.com/a/chromium.org/chromedriver/downloads)

for Chrome or geckodriver (https://github.com/mozilla/geckodriver/releases) for Firefox

[239] http://docs.python.org/library/stdtypes.html#str
[240] http://docs.python.org/library/stdtypes.html#str
[241] http://docs.python.org/library/stdtypes.html#str
[242] http://docs.python.org/library/stdtypes.html#str
[243] http://docs.python.org/library/stdtypes.html#list
[244] http://docs.python.org/library/functions.html#int
[245] http://docs.python.org/library/functions.html#int

class **QuartataWebBrowser**(*data_source=None*, *drug_group=None*, *input_type=None*, *query_type=None*, *data=None*, *num_predictions=None*, *browser_type=None*, *job_id=None*, *tsv=None*, *chem_type='known'*)

Class to browse the QuartataWeb website.

Parameters

- **data_source** (*str*[246]) – source database for QuartataWeb analysis options are `"DrugBank"` or `"STITCH"`. Default is `"DrugBank"`
- **drug_group** (*str*[247]) – group of drugs if using DrugBank options are `"Approved"` or `"All"`. Default is `"All"`
- **input_type** (*int*[248]) – number corresponding to the input type, options are `1` (Chemical and/or target) or `2` (A list of chemicals, targets or chemical combinations). Default is `1`
- **query_type** (*int*[249]) – number corresponding to the query type. Options are dependent on input_type.

  With input_type 1, they are: * `1` (chemical-target interaction) * `2` (chemical-chemical similarity) * `3` (target-target similarity)

  With input_type 2, they are: * `1` (chemicals) * `2` (targets) * `3` (chemical combinations)

  Default is `1`
- **data** (*list*[250]) – data to enter into the box or boxes. This varies depending on input type and query type, but will always be a list of strings.

  For input_type 1, a list with two items is expected. These will be one of the following depending on query_type: * With query_type 1, the first would be a chemical and the second a target.

  One of these can also be left blank.

  – With query_type 2, the first would be a chemical and the second a chemical.

  – With query_type 3, the first would be a target and the second a target.

  For input_type 2, a list with any length is expected. These will be one of the following depending on query_type: * With query_type 1, these would be chemicals. * With query_type 2, these would be targets. * With query_type 3, these would be pairs of chemicals, separated by semicolons.
- **num_predictions** (*int, list*) – number of predictions to show or consider in addition to known interactions. Default is `0`. With DrugBank and input_type 1, a second number can be provided in a list for secondary interactions.
- **browser_type** (*str*[251]) – browser type for navigation Default is `"Chrome"`
- **job_id** (*int*[252]) – job ID for accessing previous jobs Default is **None**
- **tsv** (*str*[253]) – a filename for a file that contains the results or a file to save the results in tsv format

[246] http://docs.python.org/library/stdtypes.html#str
[247] http://docs.python.org/library/stdtypes.html#str
[248] http://docs.python.org/library/functions.html#int
[249] http://docs.python.org/library/functions.html#int
[250] http://docs.python.org/library/stdtypes.html#list
[251] http://docs.python.org/library/stdtypes.html#str
[252] http://docs.python.org/library/functions.html#int
[253] http://docs.python.org/library/stdtypes.html#str

goToDownloads()
Go to downloads page

goToWorkDir()
Go to working directory

parseChemicals(*filename=None*, *chem_type='known'*)
Go to working directory and parse chemicals for query protein. Updates self.chemical_data

setBrowserType(*browser_type*)
Set browser_type and update home page

Parameters **browser_type** (*str*[254]) – browser type for navigation Default is `"Chrome"`

setData(*data*)
Set data and update home page

Parameters **data** (*list*[255]) – data to enter into the box or boxes. This varies depending on input type and query type, but will always be a list of strings.

For input_type 1, a list with two items is expected. These will be one of the following depending on query_type: * With query_type 1, the first would be a chemical and the second a target.

One of these can also be left blank.

- With query_type 2, the first would be a chemical and the second a chemical.
- With query_type 3, the first would be a target and the second a target.

For input_type 2, a list with any length is expected. These will be one of the following depending on query_type: * With query_type 1, these would be chemicals. * With query_type 2, these would be targets. * With query_type 3, these would be pairs of chemicals, separated by semicolons.

setDataSource(*data_source*)
Set data_source and update home page

Parameters **data_source** (*str*[256]) – source database for QuartataWeb analysis options are `"DrugBank"` or `"STITCH"`. Default is `"DrugBank"`

setDrugGroup(*group*)
Set drug_group and update home page

Parameters **group** (*str*[257]) – group of drugs if using DrugBank options are `"Approved"` or `"All"`. Default is `"All"`

setInputType(*input_type*)
Set input_type and update home page

Parameters **input_type** (*int*[258]) – number corresponding to the input type, options are `1` (Chemical and/or target) or `2` (A list of chemicals, targets or chemical combinations). Default is `1`

setJObID(*job_id*)
Set job_id and view results

[254] http://docs.python.org/library/stdtypes.html#str
[255] http://docs.python.org/library/stdtypes.html#list
[256] http://docs.python.org/library/stdtypes.html#str
[257] http://docs.python.org/library/stdtypes.html#str
[258] http://docs.python.org/library/functions.html#int

Parameters **job_id** (*int*[259]) – job ID for accessing previous jobs Default is **None**

setNumPredictions(*num_predictions*)
Set num_predictions and update home page

Parameters **num_predictions** (*int, list*) – number of predictions to show or consider in addition to known interactions. Default is 0. With DrugBank and input_type 1, a second number can be provided in a list for secondary interactions.

setQueryType(*query_type*)
Set query_type and update home page

Parameters **query_type** (*int*[260]) – number corresponding to the query type. Options are dependent on input_type.

With input_type 1, they are: * 1 (chemical-target interaction) * 2 (chemical-chemical similarity) * 3 (target-target similarity)

With input_type 2, they are: * 1 (chemicals) * 2 (targets) * 3 (chemical combinations)

Default is 1

updateHomePage()
Update the home page with data from setting variables

viewResults()
View results by clicking submit or using a job_id

class **QuartataChemicalRecord**(*data_source=None, drug_group=None, input_type=None, query_type=None, data=None, num_predictions=None, browser_type=None, job_id=None, filename=None*)
Class for handling chemical data from QuartataWebBrowser

Instantiate a QuartataChemicalRecord object instance. Inputs are the same as QuartataWebBrowser.

fetch(*data_source=None, drug_group=None, input_type=None, query_type=None, data=None, num_predictions=None, browser_type=None, job_id=None, filename=None*)
Fetch data

filter(*lower_weight=None, upper_weight=None, cutoff_score=None*)
Filters out chemicals from the list and returns the updated list. Chemicals that satisfy any of the following criterion will be filtered out. (1) Molecular weight < lower_weight (must be a positive number); (2) Molecular weight > upper_weight (must be a positive number); (3) Confidence score < cutoff_score (must be a positive number);

Please note that every time this function is run, this overrides any previous runs. Therefore, please provide all filters at once.

getChemicalList(*filtered=True*)
Returns chemical list (filters may be applied)

getFilterList()
Returns a list of chemicals for the entries that were filtered out

getParticularSMILES(*key*)
Returns SMILES for a particular chemical

getSMILESList(*filtered=True*)
Returns SMILES list (filters may be applied)

[259] http://docs.python.org/library/functions.html#int
[260] http://docs.python.org/library/functions.html#int

searchQuartataWeb(*data_source=None*, *drug_group=None*, *input_type=None*, *query_type=None*, *data=None*, *num_predictions=None*, *browser_type=None*, *job_id=None*, *filename=None*, *result_type='Chemical'*)

Wrapper function for searching QuartataWeb.

Parameters **result_type** (*str*[261]) – type of results to get from QuartataWeb. So far only `'Chemical'` is supported.

All other arguments are the same as *QuartataWebBrowser* (page 127).

3.4.14 UniProt Access Functions

class UniprotRecord(*data*)

This class provides a wrapper for UniProt data including functions for accessing particular fields and parsing associated PDB entries.

parsePDBs(***kwargs*)

Load PDB into memory as *AtomGroup* (page 38) instances using *parsePDB()* (page 268) and perform selection based on residue ranges given by CATH.

searchUniprot(*id*)

Search Uniprot with *id* and return a *UniprotRecord* (page 131) containing the results.

queryUniprot(*id*, *expand=[]*, *regex=True*)

Query Uniprot with *id* and return a *dict* containing the raw results. Regular users should use *searchUniprot()* (page 131) instead.

Parameters **expand** (*list*[262]) – entries through which you want to loop dictElements until there aren't any elements left

3.5 Dynamical Domain Decomposition

This module defines functions for computing structural/dynamical domain decompositions, and related properties, from either ANM modes or analysis of structural ensembles.

3.5.1 Spectrus

3.6 Dynamics Analysis

This module defines classes and functions for protein dynamics analysis.

3.6.1 Dynamics Models

The following classes are designed for modeling and analysis of protein dynamics:

- *ANM* (page 140) - Anisotropic network model, for coarse-grained NMA
- *GNM* (page 164) - Gaussian network model, for coarse-grained dynamics analysis
- *PCA* (page 175) - Principal component analysis of conformation ensembles
- *EDA* (page 176) - Essential dynamics analysis of dynamics trajectories
- *NMA* (page 172) - Normal mode analysis, for analyzing data from external programs
- *RTB* (page 187) - Rotations and Translation of Blocks method

[261] http://docs.python.org/library/stdtypes.html#str
[262] http://docs.python.org/library/stdtypes.html#list

Usage of these classes are shown in Anisotropic Network Model (ANM)[263], Gaussian Network Model (GNM)[264], Ensemble Analysis[265], and Essential Dynamics Analysis[266] examples.

The following classes are for analysis of individual modes or subsets of modes:

- *Mode* (page 169) - analyze individual normal/principal/essential modes
- *ModeSet* (page 171) - analyze subset of modes from a dynamics model
- *Vector* (page 170) - analyze modified modes or deformation vectors

3.6.2 Customize ENMs

The following classes allow for using structure or distance based, or other custom force constants and cutoff distances in *ANM* (page 140) and *GNM* (page 164) calculations:

- *Gamma* (page 159) - base class for developing property custom force constant calculation methods
- *GammaStructureBased* (page 159) - secondary structure based force constants
- *GammaVariableCutoff* (page 161) - atom type based variable cutoff function

3.6.3 Membrane Models

The following classes are designed for modeling and analysis of protein dynamics in membranes:

- *exANM* (page 150) - perform explicit membrane ANM, creating an explicit membrane lattice
- *exGNM* (page 152) - perform explicit membrane GNM, creating an explicit membrane lattice
- `imANM()` - perform implicit membrane GNM, creating anisotropic constraints in the membrane domain using RTB

Usage of these classes are shown in exanm and imanm examples.

3.6.4 Signature Dynamics (SignDy)

The following classes are designed for signature dynamics analysis of protein/domain families, together with those in the database module:

- *calcEnsembleENMs()* (page 193) - perform NMA on a protein family ensemble using ENMs
- *ModeEnsemble* (page 191) - handle outputs of ensemble NMA, an ensemble of normal modes
- *sdarray* (page 192) - handle signature dynamics data in an array based on a numpy array

There are many other functions starting *showSignature* or *calcSignature* for plotting and analysis. There are also load and save functions for mode ensembles and signature arrays.

Usage of these classes are shown in Overview[267], Core Calculations[268], and Classification using sequence, structure and dynamics distances[269] examples.

[263] http://prody.csb.pitt.edu/tutorials/enm_analysis/anm.html#anm
[264] http://prody.csb.pitt.edu/tutorials/enm_analysis/gnm.html#gnm
[265] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#pca
[266] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda
[267] http://prody.csb.pitt.edu/tutorials/signdy_tutorial/overview.html#signdy-overview
[268] http://prody.csb.pitt.edu/tutorials/signdy_tutorial/core.html#signdy-core
[269] http://prody.csb.pitt.edu/tutorials/signdy_tutorial/classifications.html#signdy-class

3.6.5 Function library

Dynamics of the functions in this library accept a *modes* argument (may also appear in different names), which may refer to one or more of the following:

- a dynamics model, *ANM* (page 140), SM, *GNM* (page 164), *NMA* (page 172), *PCA* (page 175), or *EDA* (page 176)
- a *Mode* (page 169) obtained by indexing an NMA model, e.g. `anm[0]`
- a *ModeSet* (page 171) obtained by slicing an NMA model, e.g. `anm[0:10]`

Some of these functions may also accept `Vector` instances as *mode* argument. These are noted in function documentations.

3.6.6 Analyze models

The following functions are for calculating atomic properties from normal modes:

- *calcCollectivity()* (page 137) - degree of collectivity of a mode
- *calcCovariance()* (page 137) - covariance matrix for given modes
- *calcCrossCorr()* (page 137) - cross-correlations of fluctuations
- *calcFractVariance()* (page 137) - fraction of variance explained by a mode
- *calcPerturbResponse()* (page 178) - response to perturbations in positions
- *calcProjection()* (page 138) - projection of conformations onto modes
- *calcSqFlucts()* (page 137) - square-fluctuations
- *calcTempFactors()* (page 137) - temperature factors fitted to exp. data

3.6.7 Compare models

The following functions are for comparing normal modes or dynamics models:

- *calcOverlap()* (page 142) - overlap (correlation) between modes
- *calcCumulOverlap()* (page 142) - cumulative overlap between modes
- *calcSubspaceOverlap()* (page 143) - overlap between normal mode subspaces
- *calcCovOverlap()* (page 143) - covariance overlap between models
- *printOverlapTable()* (page 143) - formatted overlap table printed on screen

3.6.8 Generate conformers

The following functions can be used to generate conformers along normal modes:

- *deformAtoms()* (page 189) - deform atoms along a mode
- *sampleModes()* (page 189) - deform along random combination of a set of modes
- *traverseMode()* (page 190) - traverse a mode along both directions

3.6.9 Adaptive ANM

The following class and its functions can be used to generate conformers using adaptive ANM:

- `AdaptiveANM` - generate transitions between two conformers using best overlapping modes

3.6.10 Essential Site Scanning Analysis (ESSA)

The following class and its functions can be used to perform Essential Site Scanning Analysis:

- *ESSA* (page 147)

3.6.11 Editing models

The following functions can be used to reduce, slice, or extrapolate models:

- *sliceMode()* (page 144) - take a slice of the normal mode
- *extendMode()* (page 144) - extend a coarse-grained mode to all-atoms
- *sliceModel()* (page 145) - take a slice of a model
- *extendModel()* (page 144) - extend a coarse-grained model to all-atoms
- *reduceModel()* (page 145) - reduce a model to a subset of atoms
- *sliceVector()* (page 145) - take a slice of a vector
- *extendVector()* (page 144) - extend a coarse-grained vector to all-atoms

3.6.12 Parse/write data

The following functions are parsing or writing normal mode data:

- *parseArray()* (page 155) - numeric arrays, e.g. coordinates, eigenvectors
- *parseModes()* (page 155) - normal modes
- *parseNMD()* (page 174) - normal mode, coordinate, and atomic data for NMWiz
- *parseSparseMatrix()* (page 156) - matrix data in sparse coordinate list format
- *writeArray()* (page 157) - numeric arrays, e.g. coordinates, eigenvectors
- *writeModes()* (page 158) - normal modes
- *writeNMD()* (page 174) - normal mode, coordinate, and atomic data
- *writeOverlapTable()* (page 143) - overlap between modes in a formatted table
- `writeBILD()` - normal mode and coordinate data for ChimeraX

3.6.13 Save/load models

Dynamics objects can be efficiently saved and loaded in later Python sessions using the following functions:

- *loadModel()* (page 158), *saveModel()* (page 158) - load/save dynamics models
- *loadVector()* (page 158), *saveVector()* (page 158) - load/save modes or vectors

3.6.14 Short-hand functions

Following allow for performing some dynamics calculations in one function call:

- *calcANM()* (page 142) - perform ANM calculations
- *calcGNM()* (page 166) - perform GNM calculations
- `calcSM()` - perform GNM calculations

3.6.15 Plotting functions

Plotting functions are called by the name of the plotted data/property and are prefixed with "show". Function documentations refers to the `matplotlib.pyplot`[270] function utilized for actual plotting. Arguments and keyword arguments are passed to the Matplotlib functions.

- *showMode()* (page 180) - mode shape
- *showOverlap()* (page 180) - overlap between modes
- *showSqFlucts()* (page 183) - square-fluctuations
- *showEllipsoid()* (page 182) - depict projection of a normal mode space on another
- *showContactMap()* (page 180) - contact map based on a Kirchhoff matrix
- *showProjection()* (page 181) - projection of conformations onto normal modes
- *showOverlapTable()* (page 181) - overlaps between two models
- *showScaledSqFlucts()* (page 183) - square-fluctuations fitted to experimental data
- *showNormedSqFlucts()* (page 183) - normalized square-fluctuations
- *showCrossProjection()* (page 182) - project conformations onto modes from different models
- *showCrossCorr()* (page 180) - cross-correlations between fluctuations in atomic positions
- *showCumulOverlap()* (page 180) - cumulative overlap of a mode with multiple modes from another model
- *showFractVars()* (page 180) - fraction of variances
- *showCumulFractVars()* (page 180) - cumulative fraction of variances
- *resetTicks()* (page 183) - change ticks in a plot

3.6.16 Heat Mapper support

The following functions can be used to read, write, and plot VMD plugin Heat Mapper[271] files.

- *showHeatmap()* (page 167)
- *parseHeatmap()* (page 166)
- *writeHeatmap()* (page 166)

3.6.17 Visualize modes

Finally, normal modes can be visualized and animated using VMD plugin Normal Mode Wizard[272]. The following functions allow for running NMWiz from within Python:

- *viewNMDinVMD()* (page 174) - run VMD and load normal mode data
- *pathVMD()* (page 174) - get/set path to VMD executable

3.6.18 Adaptive ANM

This module defines functions for performing adaptive ANM.

[270] http://matplotlib.sourceforge.net/api/pyplot_summary.html#module-matplotlib.pyplot
[271] http://www.ks.uiuc.edu/Research/vmd/plugins/heatmapper/
[272] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

calcAdaptiveANM(*a*, *b*, *n_steps*, *mode=0*, ***kwargs*)

Runs adaptive ANM analysis of proteins ([ZY09] (page 395)) that creates a path that connects two conformations using normal modes.

This function can be run in three modes:

1. *AANM_ONEWAY*: all steps are run in one direction: from *a* to *b*.
2. ***AANM_ALTERNATING*: steps are run in alternating directions: from *a* to *b*,** then *b* to *a*, then back again, and so on.
3. ***AANM_BOTHWAYS*: steps are run in one direction (from *a* to *b*)** until convergence is reached and then the other way.

This also implementation differs from the original one in that it sorts the modes by overlap prior to cumulative overlap calculations for efficiency.

Parameters

- **a** (*Atomic* (page 47), ndarray) – structure A for the transition
- **b** (*Atomic* (page 47), ndarray) – structure B for the transition
- **n_steps** (*int*[273]) – the maximum number of steps to be calculated. For *AANM_BOTHWAYS*, this means the maximum number of steps from each direction
- **mode** (*int*[274]) – the way of the calculation to be performed, which can be either *AANM_ONEWAY*, *AANM_ALTERNATING*, or *AANM_BOTHWAYS*. Default is *AANM_ALTERNATING*
- **f** (*float*[275]) – step size. Default is 0.2
- **Fmin** (*float*[276]) – cutoff for selecting modes based on square cumulative overlaps Default is **None**, which automatically determines and adapts *Fmin* on the fly.
- **Fmin_max** (*float*[277]) – maximum value for *Fmin* when it is automatically determined Default is 0.6
- **min_rmsd_diff** (*float*[278]) – cutoff for rmsds converging. Default is 0.05
- **target_rmsd** (*float*[279]) – target rmsd for stopping. Default is 1.0
- **n_modes** (*int*[280]) – the number of modes to be calculated for the first run. *n_modes* will be dynamically adjusted later as the calculation progresses. Default is 20
- **n_max_modes** (*int*[281]) – the maximum number of modes to be calculated in each run. Default is **None**, which allows as many as degree of freedom
- **callback_func** (*func*) – a callback function that can be used to collect quantities from each iteration. The function must accept ***kwargs* as its only input. Keywords in *kwargs* are: 'init': the initial coordinate; 'tar': the target coordinate; 'modes': a *ModeSet* (page 171) of selected modes; 'defvec': the deformation vector; 'c_sq': the critical square cumulative overlap; 'rmsd': the RMSD between the two structures after the deformation.

[273] http://docs.python.org/library/functions.html#int
[274] http://docs.python.org/library/functions.html#int
[275] http://docs.python.org/library/functions.html#float
[276] http://docs.python.org/library/functions.html#float
[277] http://docs.python.org/library/functions.html#float
[278] http://docs.python.org/library/functions.html#float
[279] http://docs.python.org/library/functions.html#float
[280] http://docs.python.org/library/functions.html#int
[281] http://docs.python.org/library/functions.html#int

Please see keyword arguments for calculating the modes in *calcENM()* (page 158).

3.6.19 Analysis Functions

This module defines functions for calculating physical properties from normal modes.

calcCollectivity(*mode*, *masses=None*, *is3d=None*)
Returns collectivity of the mode. This function implements collectivity as defined in equation 5 of *[BR95]* (page 395). If *masses* are provided, they will be incorporated in the calculation. Otherwise, atoms are assumed to have uniform masses.

Parameters

- **mode** (*Mode* (page 169), *Vector* (page 170), *ModeSet* (page 171), *NMA* (page 172), ndarray) – mode(s) or vector(s)
- **masses** (numpy.ndarray) – atomic masses
- **is3d** (*bool*[282]) – whether mode is 3d. Default is **None** which means determine the value based on mode.is3d().

calcCovariance(*modes*)
Returns covariance matrix calculated for given *modes*. This is 3Nx3N for 3-d models and NxN (equivalent to cross-correlations) for 1-d models such as GNM.

calcCrossCorr(*modes*, *n_cpu=1*, *norm=True*)
Returns cross-correlations matrix. For a 3-d model, cross-correlations matrix is an NxN matrix, where N is the number of atoms. Each element of this matrix is the trace of the submatrix corresponding to a pair of atoms. Cross-correlations matrix may be calculated using all modes or a subset of modes of an NMA instance. For large systems, calculation of cross-correlations matrix may be time consuming. Optionally, multiple processors may be employed to perform calculations by passing n_cpu=2 or more.

calcFractVariance(*mode*)
Returns fraction of variance explained by the *mode*. Fraction of variance is the ratio of the variance along a mode to the trace of the covariance matrix of the model.

calcSqFlucts(*modes*)
Returns sum of square-fluctuations for given set of normal *modes*. Square fluctuations for a single mode is obtained by multiplying the square of the mode array with the variance (*Mode.getVariance()* (page 170)) along the mode. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of square-fluctuations is Å^2, for *ANM* (page 140) and *GNM* (page 164), on the other hand, it is arbitrary or relative units.

calcRMSFlucts(*modes*)
Returns root mean square fluctuation(s) (RMSF) for given set of normal *modes*. This is calculated just by doing the square root of the square fluctuations

calcMostMobileNodes(*modes*, ***kwargs*)
Returns indices for nodes with highest root mean square fluctuations (RMSFs) for given set of normal *modes* above a particular *percentile* and/or *cutoff*.

Parameters

- **percentile** (*int*[283]) – percentile for internal cutoff (between 0 and 100). Default 0 takes all values
- **cutoff** (*float*[284]) – user-defined cutoff, default is to take all values

[282] http://docs.python.org/library/functions.html#bool
[283] http://docs.python.org/library/functions.html#int
[284] http://docs.python.org/library/functions.html#float

calcTempFactors(*modes*, *atoms*)
Returns temperature (β) factors calculated using *modes* from a *ANM* (page 140) or *GNM* (page 164) instance scaled according to the experimental B-factors from *atoms*.

calcProjection(*ensemble*, *modes*, *rmsd=True*, *norm=False*)
Returns projection of conformational deviations onto given modes. *ensemble* coordinates are used to calculate the deviations that are projected onto *modes*. For K conformations and M modes, a (K,M) matrix is returned.

Parameters

- **ensemble** (*Ensemble* (page 202), *Conformation* (page 201), *Vector* (page 170), *Trajectory* (page 296)) – an ensemble, trajectory or a conformation for which deviation(s) will be projected, or a deformation vector
- **modes** (*Mode* (page 169), *ModeSet* (page 171), *NMA* (page 172)) – up to three normal modes

By default, root-mean-square deviation (RMSD) along the normal mode is calculated. To calculate the raw projection pass `rmsd=False`.

By default, the projection is not normalized. If you would like it to be, pass `norm=True`.

Vector (page 170) instances are accepted as *ensemble* argument to allow for projecting a deformation vector onto normal modes.

calcCrossProjection(*ensemble*, *mode1*, *mode2*, *scale=None*, ***kwargs*)
Returns projection of conformational deviations onto modes from different models.

Parameters

- **ensemble** (*Ensemble* (page 202)) – ensemble for which deviations will be projected
- **mode1** (*Mode* (page 169), *Vector* (page 170)) – normal mode to project conformations onto
- **mode2** (*Mode* (page 169), *Vector* (page 170)) – normal mode to project conformations onto
- **scale** – scale width of the projection onto mode1 (`x`) or mode2(`y`), an optimized scaling factor (scalar) will be calculated by default or a value of scalar can be passed.

This function uses calcProjection and its arguments can be passed to it as keyword arguments. By default, this function applies RMSD scaling and normalisation. These can be turned off with `rmsd=False` and `norm=False`.

calcSpecDimension(*mode*)

Parameters **mode** (*Mode* (page 169) or *Vector* (page 170)) – mode or vector

calcPairDeformationDist(*model*, *coords*, *ind1*, *ind2*, *kbt=1.0*)
Returns distribution of the deformations in the distance contributed by each mode for selected pair of residues *ind1 ind2* using *model* from a *ANM* (page 140). Method described in *[EB08]* (page 395) equation (10) and figure (2).

Parameters

- **model** (*ANM* (page 140)) – this is an 3-dimensional `NMA` instance from a *ANM* (page 140) calculations.
- **coords** (`ndarray`.) – a coordinate set or an object with `getCoords()` method. Recommended: `coords = parsePDB('pdbfile').select('protein and name CA')`.

- **ind1** (*int*[285]) – first residue number.
- **ind2** (*int*[286]) – second residue number.

calcDistFlucts(*modes*, *n_cpu=1*, *norm=True*)
Returns the matrix of distance fluctuations (i.e. an NxN matrix where N is the number of residues, of MSFs in the inter-residue distances) computed from the cross-correlation matrix (see Eq. 12.E.1 in *[IB18]* (page 395)). The arguments are the same as in `calcCrossCorr()`.

calcHinges(*modes*, *atoms=None*, *flag=False*)
Returns the hinge sites identified using normal modes.

Parameters

- **modes** (*GNM* (page 164)) – normal modes of which will be used to identify hinge sites
- **atoms** (*Atomic* (page 47)) – an Atomic object on which to map hinges. The output will then be a selection.
- **flag** (*bool*[287]) – whether return flag or index array. Default is **False**

calcHitTime(*model*, *method='standard'*)
Returns the hit and commute times between pairs of nodes calculated based on a *NMA* (page 172) object.

to Equilibrium Fluctuations. *PLoS Comput Biol* **2007** 3(9).

Parameters

- **model** (*NMA* (page 172)) – model to be used to calculate hit times
- **method** (*str*[288]) – method to be used to calculate hit times. Available options are `"standard"` or `"kirchhoff"`. Default is `"standard"`

Returns (ndarray, ndarray)

calcAnisousFromModel(*model*)
Returns a Nx6 matrix containing anisotropic B factors (ANISOU lines) from a covariance matrix calculated from **model**.

Parameters **model** (*ANM* (page 140), *PCA* (page 175)) – 3D model from which to calculate covariance matrix

```
In [1]: from prody import *

In [2]: protein = parsePDB('1ejg')

In [3]: anm, calphas = calcANM(protein)

In [4]: adp_matrix = calcAnisousFromModel(anm)
```

calcScipionScore(*modes*)
Calculate the score from hybrid electron microscopy normal mode analysis (HEMNMA) *[CS14]* (page 395) as implemented in the Scipion continuousflex plugin *[MH20]* (page 395). This score prioritises modes as a function of mode number and collectivity order.

Parameters **modes** (*Mode* (page 169), *Vector* (page 170), *ModeSet* (page 171), *NMA* (page 172)) – mode(s) or vector(s)

[285] http://docs.python.org/library/functions.html#int
[286] http://docs.python.org/library/functions.html#int
[287] http://docs.python.org/library/functions.html#bool
[288] http://docs.python.org/library/stdtypes.html#str

calcHemnmaScore(*modes*)
Calculate the score from hybrid electron microscopy normal mode analysis (HEMNMA) *[CS14]* (page 395) as implemented in the Scipion continuousflex plugin *[MH20]* (page 395). This score prioritises modes as a function of mode number and collectivity order.

Parameters **modes** (*Mode* (page 169), *Vector* (page 170), *ModeSet* (page 171), *NMA* (page 172)) – mode(s) or vector(s)

3.6.20 Anisotropic Network Model

This module defines a class and a function for anisotropic network model (ANM) calculations.

class ANM(*name='Unknown'*)
Class for Anisotropic Network Model (ANM) analysis of proteins (*[PD00]* (page 395), *[ARA01]* (page 395)).

See a usage example in Anisotropic Network Model (ANM)[289].

addEigenpair(*vector*, *value=None*)
Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Inverse eigenvalues are set as variances.

buildHessian(*coords*, *cutoff=15.0*, *gamma=1.0*, ***kwargs*)
Build Hessian matrix for given coordinate set.

Parameters

- **coords** (`numpy.ndarray`, `Atomic`, `Ensemble`, `Trajectory`) – a coordinate set or an object with `getCoords` method
- **cutoff** (*float*[290]) – cutoff distance (Å) for pairwise interactions, default is 15.0 Å, minimum is 4.0 Å
- **gamma** (float, `Gamma`) – spring constant, default is 1.0
- **sparse** (*bool*[291]) – elect to use sparse matrices, default is **False**. If Scipy is not found, `ImportError`[292] is raised.
- **kdtree** (*bool*[293]) – elect to use KDTree for building Hessian matrix, default is **False** since KDTree method is slower

Instances of `Gamma` classes and custom functions are accepted as *gamma* argument.

When Scipy is available, user can select to use sparse matrices for efficient usage of memory at the cost of computation speed.

Any atoms or points can be used for building a Hessian matrix, including calphas, phosphorus and carbon atoms from nucleic acids, all atoms, or pseudoatoms fitted to density maps with algorithms such as TRN.

The cutoff distance may need to be adjusted depending on the coarse graining level of the atoms or points used.

buildKirchhoff(*coords*, *cutoff=10.0*, *gamma=1.0*, ***kwargs*)
Build Kirchhoff matrix for given coordinate set.

Parameters

[289] http://prody.csb.pitt.edu/tutorials/enm_analysis/anm.html#anm
[290] http://docs.python.org/library/functions.html#float
[291] http://docs.python.org/library/functions.html#bool
[292] http://docs.python.org/library/exceptions.html#ImportError
[293] http://docs.python.org/library/functions.html#bool

- **coords** (numpy.ndarray or *Atomic* (page 47)) – a coordinate set or an object with getCoords method
- **cutoff** (*float*[294]) – cutoff distance (Å) for pairwise interactions default is 10.0 Å, , minimum is 4.0 Å
- **gamma** (*float*[295]) – spring constant, default is 1.0
- **sparse** (*bool*[296]) – elect to use sparse matrices, default is **False**. If Scipy is not found, ImportError[297] is raised.
- **kdtree** (*bool*[298]) – elect to use KDTree for building Kirchhoff matrix faster, default is **True**

Instances of Gamma classes and custom functions are accepted as *gamma* argument.

When Scipy is available, user can select to use sparse matrices for efficient usage of memory at the cost of computation speed.

calcModes(*n_modes=20*, *zeros=False*, *turbo=True*, ***kwargs*)

Calculate normal modes. This method uses scipy.linalg.eigh()[299] function to diagonalize the Hessian matrix. When Scipy is not found, numpy.linalg.eigh() is used.

Parameters

- **n_modes** (*int or None, default is 20*) – number of non-zero eigenvalues/vectors to calculate. If **None** or 'all' is given, all modes will be calculated.
- **zeros** (bool, default is **True**) – If **True**, modes with zero eigenvalues will be kept.
- **turbo** (bool, default is **True**) – Use a memory intensive, but faster way to calculate modes.
- **nproc** (*int*[300]) – number of processors for thread pool limit, default is **0**, meaning don't impose limit

getArray()

Returns a copy of eigenvectors array.

getCovariance()

Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getCutoff()

Returns cutoff distance.

getEigvals()

Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is $Å^2$. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()

Returns a copy of eigenvectors array.

[294] http://docs.python.org/library/functions.html#float
[295] http://docs.python.org/library/functions.html#float
[296] http://docs.python.org/library/functions.html#bool
[297] http://docs.python.org/library/exceptions.html#ImportError
[298] http://docs.python.org/library/functions.html#bool
[299] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[300] http://docs.python.org/library/functions.html#int

getGamma()
Returns spring constant (or the gamma function or `Gamma` instance).

getHessian()
Returns a copy of the Hessian matrix.

getKirchhoff()
Returns a copy of the Kirchhoff matrix.

getModel()
Returns self.

getNormDistFluct(*coords*)
Normalized distance fluctuation

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is $Å^2$. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

setHessian(*hessian*)
Set Hessian matrix. A symmetric matrix is expected, i.e. not a lower- or upper-triangular matrix.

setKirchhoff(*kirchhoff*)
Set Kirchhoff matrix.

setTitle(*title*)
Set title of the model.

calcANM(*pdb*, *selstr='calpha'*, *cutoff=15.0*, *gamma=1.0*, *n_modes=20*, *zeros=False*, *title=None*)
Returns an *ANM* (page 140) instance and atoms used for the calculations. By default only alpha carbons are considered, but selection string helps selecting a subset of it. *pdb* can be a PDB code, *Atomic* (page 47) instance, or a Hessian matrix (`ndarray`).

3.6.21 Comparison Functions

This module defines functions for comparing normal modes from different models.

calcOverlap(*rows*, *cols*, *diag=False*)
Returns overlap (or correlation) between two sets of modes (*rows* and *cols*). Returns a matrix whose rows correspond to modes passed as *rows* argument, and columns correspond to those passed as *cols* argument. Both rows and columns are normalized prior to calculating overlap.

This function can now return the diagonal of the overlap matrix if *diag* is set to **True**.

calcCumulOverlap(*modes1*, *modes2*, *array=False*)
Returns cumulative overlap of modes in *modes2* with those in *modes1*. Returns a number if *modes1* contains a single `Mode` (page 169) or a `Vector` (page 170) instance. If *modes1* contains multiple modes, returns an array. Elements of the array correspond to cumulative overlaps for modes in *modes1* with those in *modes2*. If *array* is **True**, returns an array of cumulative overlaps. Returned array has the shape `(len(modes1), len(modes2))`. Each row corresponds to cumulative overlaps calculated for modes in *modes1* with those in *modes2*. Each value in a row corresponds to cumulative overlap calculated using up to that many number of modes from *modes2*.

calcSubspaceOverlap(*modes1*, *modes2*)
Returns subspace overlap between two sets of modes (*modes1* and *modes2*). Also known as the root mean square inner product (RMSIP) of essential subspaces *[AA99]* (page 395). This function returns a single number.

calcSpectralOverlap(*modes1*, *modes2*, *weighted=False*, *turbo=False*)
Returns overlap between covariances of *modes1* and *modes2*. Overlap between covariances are calculated using normal modes (eigenvectors), hence modes in both models must have been calculated. This function implements equation 11 in *[BH02]* (page 395).

Parameters **weighted** (*bool*[301]) – if **True** then covariances are weighted by the trace.

calcCovOverlap(*modes1*, *modes2*, *turbo=False*)
Returns overlap between covariances of *modes1* and *modes2*. Overlap between covariances are calculated using normal modes (eigenvectors), hence modes in both models must have been calculated. This function implements equation 11 in *[BH02]* (page 395).

printOverlapTable(*rows*, *cols*)
Print table of overlaps (correlations) between two sets of modes. *rows* and *cols* are sets of normal modes, and correspond to rows and columns of the printed table. This function may be used to take a quick look into mode correspondences between two models.

```
>>> # Compare top 3 PCs and slowest 3 ANM modes
>>> printOverlapTable(p38_pca[:3], p38_anm[:3])
Overlap Table
                         ANM 1p38
                    #1     #2     #3
PCA p38 xray #1   -0.39  +0.04  -0.71
PCA p38 xray #2   -0.78  -0.20  +0.22
PCA p38 xray #3   +0.05  -0.57  +0.06
```

writeOverlapTable(*filename*, *rows*, *cols*)
Write table of overlaps (correlations) between two sets of modes to a file. *rows* and *cols* are sets of normal modes, and correspond to rows and columns of the overlap table. See also `printOverlapTable()` (page 143).

calcSquareInnerProduct(*modes1*, *modes2*)
Returns the square inner product (SIP) of fluctuations *[SK02]* (page 396). This function returns a single number.

pairModes(*modes1*, *modes2*, ***kwargs*)
Returns the optimal matches between *modes1* and *modes2*. *modes1* and *modes2* should have equal number of modes, and the function will return a nested list where each item is a list containing a pair of modes.

Parameters **index** (*bool*[302]) – if **True** then indices of modes will be returned instead of `Mode` instances.

[301] http://docs.python.org/library/functions.html#bool
[302] http://docs.python.org/library/functions.html#bool

matchModes (**modesets*, ***kwargs*)
Returns the matches of modes among *modesets*. Note that the first modeset will be treated as the reference so that only the matching of each modeset to the first modeset is guaranteed to be optimal.

Parameters

- **index** (*bool*[303]) – if **True** then indices of modes will be returned instead of `Mode` instances
- **turbo** (*bool, int*) – if **True** then the computation will be performed in parallel. The number of threads is set to be the same as the number of CPUs. Assigning a number will specify the number of threads to be used. Note that if writing a script, `if __name__ == '__main__'` is necessary to protect your code when multitasking. See https://docs.python.org/2/library/multiprocessing.html for details. Default is **False**

calcRMSIP (*modes1*, *modes2*)
Returns subspace overlap between two sets of modes (*modes1* and *modes2*). Also known as the root mean square inner product (RMSIP) of essential subspaces *[AA99]* (page 395). This function returns a single number.

calcSIP (*modes1*, *modes2*)
Returns the square inner product (SIP) of fluctuations *[SK02]* (page 396). This function returns a single number.

calcRWSIP (*modes1*, *modes2*)
Returns root weighted square inner product (RWSIP) of essential subspaces *[VC07]* (page 396). This function returns a single number.

3.6.22 NMA Model Editing

This module defines functions for editing normal mode data.

extendModel (*model*, *nodes*, *atoms*, *norm=False*)
Extend a coarse grained *model* built for *nodes* to *atoms*. *model* may be *ANM* (page 140), *GNM* (page 164), *PCA* (page 175), or *NMA* (page 172) instance. This function will take part of the normal modes for each node (i.e. Cα atoms) and extend it to all other atoms in the same residue. For each atom in *nodes* argument *atoms* argument must contain a corresponding residue. If *norm* is **True**, extended modes are normalized.

extendMode (*mode*, *nodes*, *atoms*, *norm=False*)
Extend a coarse grained normal *mode* built for *nodes* to *atoms*. This function will take part of the normal modes for each node (i.e. Cα atoms) and extend it to all other atoms in the same residue. For each atom in *nodes* argument *atoms* argument must contain a corresponding residue. Extended mode is multiplied by the square root of variance of the mode. If *norm* is **True**, extended mode is normalized.

extendVector (*vector*, *nodes*, *atoms*)
Extend a coarse grained *vector* for *nodes* to *atoms*. This function will take part of the normal modes for each node (i.e. Cα atoms) and extend it to all other atoms in the same residue. For each atom in *nodes*, *atoms* argument must contain a corresponding residue.

sliceMode (*mode*, *atoms*, *select*)
Returns part of the *mode* for *atoms* matching *select*. This works slightly different from *sliceVector()* (page 145). Mode array (eigenvector) is multiplied by square-root of the variance along the mode. If mode is from an elastic network model, variance is defined as the inverse of the eigenvalue. Note that returned *Vector* (page 170) instance is not normalized.

Parameters

[303] http://docs.python.org/library/functions.html#bool

- **mode** (*Mode* (page 169)) – mode instance to be sliced
- **atoms** (*Atomic* (page 47)) – atoms for which *mode* describes a deformation, motion, etc.
- **select** (*Selection* (page 100), str) – an atom selection or a selection string

Returns (*Vector* (page 170), *Selection* (page 100))

sliceModel(*model*, *atoms*, *select*, *norm=False*)

Returns a part of the *model* (modes calculated) for *atoms* matching *select*. Note that normal modes are sliced instead of the connectivity matrix. Sliced normal modes (eigenvectors) are not normalized unless *norm* is **True**.

Parameters

- **mode** (*NMA* (page 172)) – NMA model instance to be sliced
- **atoms** (*Atomic* (page 47)) – atoms for which the *model* was built
- **select** (*Selection* (page 100), str) – an atom selection or a selection string
- **norm** (*bool*[304]) – whether to normalize eigenvectors, default **False**

Returns (*NMA* (page 172), *Selection* (page 100))

sliceModelByMask(*model*, *mask*, *norm=False*)

Returns a part of the *model* indicated by *mask*. Note that normal modes (eigenvectors) are not normalized unless *norm* is **True**.

Parameters

- **mode** (*NMA* (page 172)) – NMA model instance to be sliced
- **mask** (list, `ndarray`) – an Integer array or a Boolean array where `"True"` indicates the parts being selected
- **norm** (*bool*[305]) – whether to normalize eigenvectors, default **False**

Returns *NMA* (page 172)

sliceVector(*vector*, *atoms*, *select*)

Returns part of the *vector* for *atoms* matching *select*. Note that returned *Vector* (page 170) instance is not normalized.

Parameters

- **vector** (`VectorBase`) – vector instance to be sliced
- **atoms** (*Atomic* (page 47)) – atoms for which *vector* describes a deformation, motion, etc.
- **select** (*Selection* (page 100), str) – an atom selection or a selection string

Returns (*Vector* (page 170), *Selection* (page 100))

reduceModel(*model*, *atoms*, *select*)

Returns reduced NMA model. Reduces a *NMA* (page 172) model to a subset of *atoms* matching *select*. This function behaves differently depending on the type of the *model* argument. For *ANM* (page 140) and *GNM* (page 164) or other *NMA* (page 172) models, force constant matrix for system of interest (specified by the *select*) is derived from the force constant matrix for the *model* by assuming that for any given displacement of the system of interest, other atoms move along in such a way as to minimize the

[304] http://docs.python.org/library/functions.html#bool
[305] http://docs.python.org/library/functions.html#bool

potential energy. This is based on the formulation in *[KH00]* (page 396). For *PCA* (page 175) models, this function simply takes the sub-covariance matrix for selection.

Parameters

- **model** (*ANM* (page 140), *GNM* (page 164), or *PCA* (page 175)) – dynamics model
- **atoms** (*Atomic* (page 47)) – atoms that were used to build the model
- **select** (*Selection* (page 100), str) – an atom selection or a selection string

Returns (*NMA* (page 172), *Selection* (page 100))

reduceModelByMask(*model*, *mask*)

Returns NMA model reduced based on *mask*.

Parameters

- **model** (*ANM* (page 140), *GNM* (page 164), or *PCA* (page 175)) – dynamics model
- **mask** (list, ndarray) – an Integer array or a Boolean array where "True" indicates the parts being selected

Returns *NMA* (page 172)

trimModel(*model*, *atoms*, *select*)

Returns a part of the *model* for *atoms* matching *select*. This method removes columns and rows in the connectivity matrix and fix the diagonal sums. Normal modes need to be calculated again after the trim.

Parameters

- **mode** (*NMA* (page 172)) – NMA model instance to be sliced
- **atoms** (*Atomic* (page 47)) – atoms for which the *model* was built
- **select** (*Selection* (page 100), str) – an atom selection or a selection string

Returns (*NMA* (page 172), *Selection* (page 100))

trimModelByMask(*model*, *mask*)

Returns a part of the *model* indicated by *mask*. This method removes columns and rows in the connectivity matrix indicated by *mask* and fix the diagonal sums. Normal modes need to be calculated again after the trim.

Parameters

- **mode** (*NMA* (page 172)) – NMA model instance to be sliced
- **mask** (list, ndarray) – an Integer array or a Boolean array where "True" indicates the parts being selected

Returns *NMA* (page 172)

interpolateModel(*model*, *nodes*, *coords*, *norm=False*, ***kwargs*)

Interpolate a coarse grained *model* built for *nodes* to *coords*.

model may be *ANM* (page 140), *PCA* (page 175), or *NMA* (page 172) instance

Parameters

- **nodes** (*Atomic* (page 47), ndarray) – the coordinate set or object with getCoords() method that corresponds to the model
- **coords** (*Atomic* (page 47), ndarray) – a coordinate set or an object with getCoords() method onto which the model should be interpolated

This function will take the part of the normal modes for each node (i.e. Cα atoms) and extend it to nearby atoms.

If *norm* is **True**, extended modes are normalized.

Adapted from ModeHunter as described in *[JS09]* (page 396).

elastic network simulation of biomolecular motion. *J Chem Phys* **2009** 131:074112.

Legal notice: # # This software is copyrighted, (c) 2009-10, by Joseph N. Stember and Willy Wriggers # under the following terms: # # The authors hereby grant permission to use, copy, modify, and re-distribute this # software and its documentation for any purpose, provided that existing copyright # notices are retained in all copies and that this notice is included verbatim in # any distributions. No written agreement, license, or royalty fee is required for # any of the authorized uses. # # IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, # INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE # OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE # AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. # # THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, # BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A # PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS # IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE # MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS. # # ++

3.6.23 Essential Site Scanning Analysis

Copyright (c) 2020 Burak Kaynak, Pemra Doruker.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

class ESSA

ESSA determines the essentiality score of each residue based on the extent to which it can alter the global dynamics (*[KB20]* (page 396)). It can also rank potentially allosteric pockets by calculating their ESSA and local hydrophibic density z-scores using Fpocket algorithm (*[LGV09]* (page 396)).

Instantiate an ESSA object.

getESSAEnsemble()

Returns ESSA mode ensemble, comprised of ENMS calculated for each scanned/perturbed residue.

getESSAZscores()

Returns ESSA z-scores.

getEigvals()
Returns eigenvalues of the matched modes.

getEigvecs()
Returns eigenvectors of the matched modes.

getLigandResidueCodes()
Returns chain ids and residue numbers of the residues interacting with ligands.

getLigandResidueESSAZscores()
Returns ESSA z-scores of the residues interacting with ligands as a dictionary. The keys of which are the corresponding chain ids and residue numbers of the ligands. Each value comprises the indices of the residue ESSA z-scores in the profile and the corresponding scores as separate arrays.

getLigandResidueIndices()
Returns residue indices of the residues interacting with ligands.

getPocketFeatures()
Returns pocket features as a Pandas dataframe.

getPocketRanks()
Returns pocket ranks (allosteric potential).

getPocketZscores()
Returns ESSA and local hydrophobic density (LHD) z-scores for pockets as a Pandas dataframe.

rankPockets()
Ranks pockets in terms of their allosteric potential, based on their ESSA z-scores (max/median) with local hydrophobic density (LHD) screening.

saveESSAEnsemble()
Saves ESSA mode ensemble, comprised of ENMS calculated for each scanned/perturbed residue.

saveESSAZscores()
Saves ESSA z-scores to a binary file in Numpy *.npy* format.

saveEigvals()
Saves eigenvalues of the matched modes in Numpy *.npy* format.

saveEigvecs()
Saves eigenvectors of the matched modes in Numpy *.npy* format.

saveLigandResidueCodes()
Saves chain ids and residue numbers of the residues interacting with ligands.

saveLigandResidueESSAZscores()
Saves the dictionary of ESSA z-scores of the residues interacting with ligands to a pickle *.pkl* file. The keys of the dictionary are the corresponding chain ids and residue numbers of the ligands. Each value comprises the indices of the residue ESSA z-scores in the profile and the corresponding scores as separate arrays.

savePocketFeatures()
Saves pocket features to a pickle *.pkl* file.

savePocketRanks()
Saves pocket ranks to a binary file in Numpy *.npy* format.

savePocketZscores()
Saves ESSA and local hydrophobic density (LHD) z-scores of pockets to a pickle *.pkl* file.

scanPockets()
Generates ESSA z-scores for pockets and parses pocket features. It requires both Fpocket 3.0 and Pandas being installed in your system.

scanResidues(*n_modes=10*, *enm='gnm'*, *cutoff=None*)
Scans residues to generate ESSA z-scores.

Parameters

- **n_modes** (*int*[306]) – Number of global modes.
- **enm** (*str*[307]) – Type of elastic network model, 'gnm' or 'anm', default is 'gnm'.
- **cutoff** (*float*[308]) – Cutoff distance (A) for pairwise interactions, default is 10 A for GNM and 15 A for ANM.

setSystem(*atoms*, ***kwargs*)
Sets atoms, ligands and a cutoff distance for protein-ligand interactions.

Parameters

- **atoms** – *atoms* parsed by parsePDB
- **lig** (*str*[309]) – String of ligands' chainIDs and resSeqs (resnum) separated by a whitespace, e.g., 'A 300 B 301'. Default is None.
- **dist** (*float*[310]) – Atom-atom distance (A) to select the protein residues that are in contact with a ligand, default is 4.5 A.
- **lowmem** (*bool*[311]) – If True, a ModeEnsemble is not generated due to the lack of memory resources, and eigenvalue/eigenvectors are only stored, default it False.

showESSAProfile(*q=0.75*, *rescode=False*, *sel=None*)
Shows ESSA profile.

Parameters

- **q** (*float*[312]) – Quantile value to plot a baseline for z-scores, default is 0.75. If it is set to 0.0, then the baseline is not drawn.
- **rescode** (*bool*[313]) – If it is True, the ligand interacting residues with ESSA scores larger than the baseline are denoted by their single letter codes and chain ids on the profile. If quantile vaue is 0.0, then all ligand-interacting residues denoted by their codes.
- **sel** (*str*[314]) – It is a selection string, default is None. If it is provided, then selected residues are shown with their single letter codes and residue numbers on the profile. For example, 'chain A and resnum 33 47'.

showPocketZscores()
Plots maximum/median ESSA and local hydrophobic density (LHD) z-scores for pockets.

writeESSAZscoresToPDB()
Writes a pdb file with ESSA z-scores placed in the B-factor column.

[306] http://docs.python.org/library/functions.html#int
[307] http://docs.python.org/library/stdtypes.html#str
[308] http://docs.python.org/library/functions.html#float
[309] http://docs.python.org/library/stdtypes.html#str
[310] http://docs.python.org/library/functions.html#float
[311] http://docs.python.org/library/functions.html#bool
[312] http://docs.python.org/library/functions.html#float
[313] http://docs.python.org/library/functions.html#bool
[314] http://docs.python.org/library/stdtypes.html#str

writePocketRanksToCSV()
Writes pocket ranks to a *.csv* file.

3.6.24 Explicit Membrane Anisotropic Network Model

This module defines a class and a function for explicit membrane ANM calculations.

class exANM(*name='Unknown'*)
Class for explicit ANM (exANM) method ([FT00] (page 397)). Optional arguments build a membrane lattice permit analysis of membrane

effect on elastic network models in *exANM* method described in [TL12] (page 396).

addEigenpair(*vector*, *value=None*)
Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Inverse eigenvalues are set as variances.

buildHessian(*coords*, *cutoff=15.0*, *gamma=1.0*, ***kwargs*)
Build Hessian matrix for given coordinate set. **kwargs** are passed to *buildMembrane()* (page 150).

Parameters

- **coords** (numpy.ndarray) – a coordinate set or an object with getCoords method
- **cutoff** (*float*[315]) – cutoff distance (Å) for pairwise interactions, default is 15.0 Å
- **gamma** (*float*[316]) – spring constant, default is 1.0

buildKirchhoff(*coords*, *cutoff=10.0*, *gamma=1.0*, ***kwargs*)
Build Kirchhoff matrix for given coordinate set.

Parameters

- **coords** (numpy.ndarray or *Atomic* (page 47)) – a coordinate set or an object with getCoords method
- **cutoff** (*float*[317]) – cutoff distance (Å) for pairwise interactions default is 10.0 Å, , minimum is 4.0 Å
- **gamma** (*float*[318]) – spring constant, default is 1.0
- **sparse** (*bool*[319]) – elect to use sparse matrices, default is **False**. If Scipy is not found, ImportError[320] is raised.
- **kdtree** (*bool*[321]) – elect to use KDTree for building Kirchhoff matrix faster, default is **True**

Instances of Gamma classes and custom functions are accepted as *gamma* argument.

When Scipy is available, user can select to use sparse matrices for efficient usage of memory at the cost of computation speed.

buildMembrane(*coords*, ***kwargs*)
Build membrane lattice around **coords**.

[315] http://docs.python.org/library/functions.html#float
[316] http://docs.python.org/library/functions.html#float
[317] http://docs.python.org/library/functions.html#float
[318] http://docs.python.org/library/functions.html#float
[319] http://docs.python.org/library/functions.html#bool
[320] http://docs.python.org/library/exceptions.html#ImportError
[321] http://docs.python.org/library/functions.html#bool

Parameters

- **coords** (numpy.ndarray) – a coordinate set or an object with getCoords method
- **membrane_high** (*float*[322]) – the maximum z coordinate of the membrane. Default is **13.0**
- **membrane_low** (*float*[323]) – the minimum z coordinate of the membrane. Default is **-13.0**
- **R** (*float*[324]) – radius of all membrane in x-y direction. Default is **80**
- **Ri** (*float*[325]) – inner radius of the membrane in x-y direction if it needs to be hollow. Default is **0**, which is not hollow
- **r** (*float*[326]) – radius of each membrane node. Default is **3.1**
- **lat** (*str*[327]) – lattice type which could be **FCC** (face-centered-cubic, default), **SC** (simple cubic), **SH** (simple hexagonal)
- **exr** (*float*[328]) – exclusive radius of each protein node. Default is **5.0**
- **hull** (*bool*[329]) – whether use convex hull to determine the protein's interior. Turn it off if protein is multimer. Default is **True**
- **center** (*bool*[330]) – whether transform the structure to the origin (only x- and y-axis). Default is **True**

calcModes(*n_modes=20*, ***kwargs*)

Calculate normal modes. This method uses scipy.linalg.eigh()[331] function to diagonalize the Hessian matrix. When Scipy is not found, numpy.linalg.eigh() is used.

Parameters

- **n_modes** (*int or None, default is 20*) – number of non-zero eigenvalues/vectors to calculate. If **None** is given, all modes will be calculated.
- **zeros** (bool, default is **True**) – If **True**, modes with zero eigenvalues will be kept.
- **turbo** (bool, default is **True**) – Use a memory intensive, but faster way to calculate modes.

getArray()

Returns a copy of eigenvectors array.

getCombined()

Returns a copy of the combined atoms or coordinates.

getCovariance()

Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getCutoff()

Returns cutoff distance.

[322] http://docs.python.org/library/functions.html#float
[323] http://docs.python.org/library/functions.html#float
[324] http://docs.python.org/library/functions.html#float
[325] http://docs.python.org/library/functions.html#float
[326] http://docs.python.org/library/functions.html#float
[327] http://docs.python.org/library/stdtypes.html#str
[328] http://docs.python.org/library/functions.html#float
[329] http://docs.python.org/library/functions.html#bool
[330] http://docs.python.org/library/functions.html#bool
[331] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Returns a copy of eigenvectors array.

getGamma()
Returns spring constant (or the gamma function or `Gamma` instance).

getHessian()
Returns a copy of the Hessian matrix.

getKirchhoff()
Returns a copy of the Kirchhoff matrix.

getMembrane()
Returns a copy of the membrane coordinates.

getModel()
Returns self.

getNormDistFluct(*coords*)
Normalized distance fluctuation

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

setHessian(*hessian*)
Set Hessian matrix. A symmetric matrix is expected, i.e. not a lower- or upper-triangular matrix.

setKirchhoff(*kirchhoff*)
Set Kirchhoff matrix.

setTitle(*title*)
Set title of the model.

3.6.25 Explicit Membrane Gaussian Network Model

This module defines a class and a function for explicit membrane GNM calculations.

class exGNM(*name='Unknown'*)

Class for explicit GNM (exGNM) method ([FT00] (page 397)). Optional arguments build a membrane lattice permit analysis of membrane

effect on elastic network models in *exGNM* method described in [TL12] (page 396).

addEigenpair(*vector*, *value=None*)

Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Inverse eigenvalues are set as variances.

buildKirchhoff(*coords*, *cutoff=15.0*, *gamma=1.0*, ***kwargs*)

Build Kirchhoff matrix for given coordinate set. **kwargs** are passed to `buildMembrane()` (page 153).

Parameters

- **coords** (`numpy.ndarray`) – a coordinate set or an object with `getCoords` method
- **cutoff** (*float*[332]) – cutoff distance (Å) for pairwise interactions, default is 15.0 Å
- **gamma** (*float*[333]) – spring constant, default is 1.0

buildMembrane(*coords*, ***kwargs*)

Build Kirchhoff matrix for given coordinate set.

Parameters

- **coords** (`numpy.ndarray`) – a coordinate set or an object with `getCoords` method
- **membrane_high** (*float*[334]) – the maximum z coordinate of the membrane. Default is **13.0**
- **membrane_low** (*float*[335]) – the minimum z coordinate of the membrane. Default is **-13.0**
- **R** (*float*[336]) – radius of all membrane in x-y direction. Default is **80**
- **Ri** (*float*[337]) – inner radius of the membrane in x-y direction if it needs to be hollow. Default is **0**, which is not hollow
- **r** (*float*[338]) – radius of each membrane node. Default is *.1**
- **lat** (*str*[339]) – lattice type which could be **FCC** (face-centered-cubic, default), **SC** (simple cubic), **SH** (simple hexagonal)
- **exr** (*float*[340]) – exclusive radius of each protein node. Default is **5.0**
- **hull** (*bool*[341]) – whether use convex hull to determine the protein's interior. Turn it off if protein is multimer. Default is **True**

[332] http://docs.python.org/library/functions.html#float
[333] http://docs.python.org/library/functions.html#float
[334] http://docs.python.org/library/functions.html#float
[335] http://docs.python.org/library/functions.html#float
[336] http://docs.python.org/library/functions.html#float
[337] http://docs.python.org/library/functions.html#float
[338] http://docs.python.org/library/functions.html#float
[339] http://docs.python.org/library/stdtypes.html#str
[340] http://docs.python.org/library/functions.html#float
[341] http://docs.python.org/library/functions.html#bool

- **center** (*bool*[342]) – whether transform the structure to the origin (only x- and y-axis). Default is **True**

calcModes(*n_modes=20*, ***kwargs*)
Calculate normal modes. This method uses `scipy.linalg.eigh()`[343] function to diagonalize the Kirchhoff matrix. When Scipy is not found, `numpy.linalg.eigh()` is used.

Parameters

- **n_modes** (*int or None, default is 20*) – number of non-zero eigenvalues/vectors to calculate. If **None** is given, all modes will be calculated.
- **zeros** (bool, default is **True**) – If **True**, modes with zero eigenvalues will be kept.
- **turbo** (bool, default is **True**) – Use a memory intensive, but faster way to calculate modes.

getArray()
Returns a copy of eigenvectors array.

getCombined()
Returns a copy of the combined atoms or coordinates.

getCovariance()
Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getCutoff()
Returns cutoff distance.

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Returns a copy of eigenvectors array.

getGamma()
Returns spring constant (or the gamma function or `Gamma` instance).

getKirchhoff()
Returns a copy of the Kirchhoff matrix.

getMembrane()
Returns a copy of the membrane coordinates.

getModel()
Returns self.

getNormDistFluct(*coords*)
Normalized distance fluctuation

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

[342] http://docs.python.org/library/functions.html#bool
[343] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

setKirchhoff(*kirchhoff*)
Set Kirchhoff matrix.

setTitle(*title*)
Set title of the model.

3.6.26 Supporting Functions

This module defines input and output functions.

parseArray(*filename*, *delimiter=None*, *skiprows=0*, *usecols=None*, *dtype=<type 'float'>*)
Parse array data from a file.

This function is using `numpy.loadtxt()` to parse the file. Each row in the text file must have the same number of values.

Parameters

- **filename** (*str or file*) – File or filename to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed.
- **delimiter** (*str*[344]) – The string used to separate values. By default, this is any whitespace.
- **skiprows** (*int*[345]) – Skip the first *skiprows* lines, default is `0`.
- **usecols** (*list*[346]) – Which columns to read, with 0 being the first. For example, `usecols = (1,4,5)` will extract the 2nd, 5th and 6th columns. The default, **None**, results in all columns being read.
- **dtype** (`numpy.dtype`.) – Data-type of the resulting array, default is `float()`.

parseModes(*normalmodes*, *eigenvalues=None*, *nm_delimiter=None*, *nm_skiprows=0*, *nm_usecols=None*, *ev_delimiter=None*, *ev_skiprows=0*, *ev_usecols=None*, *ev_usevalues=None*)
Returns *NMA* (page 172) instance with normal modes parsed from *normalmodes*.

In normal mode file *normalmodes*, columns must correspond to modes (eigenvectors). Optionally, *eigenvalues* can be parsed from a separate file. If eigenvalues are not provided, they will all be set to 1.

Parameters

- **normalmodes** (*str or file*) – File or filename that contains normal modes. If the filename extension is `.gz` or `.bz2`, the file is first decompressed.
- **eigenvalues** (*str or file*) – Optional, file or filename that contains eigenvalues. If the filename extension is `.gz` or `.bz2`, the file is first decompressed.

[344] http://docs.python.org/library/stdtypes.html#str
[345] http://docs.python.org/library/functions.html#int
[346] http://docs.python.org/library/stdtypes.html#list

- **nm_delimiter** (*str*[347]) – The string used to separate values in *normalmodes*. By default, this is any whitespace.
- **nm_skiprows** (*0*) – Skip the first *skiprows* lines in *normalmodes*. Default is `0`.
- **nm_usecols** (*list*[348]) – Which columns to read from *normalmodes*, with 0 being the first. For example, `usecols = (1,4,5)` will extract the 2nd, 5th and 6th columns. The default, **None**, results in all columns being read.
- **ev_delimiter** (*str*[349]) – The string used to separate values in *eigenvalues*. By default, this is any whitespace.
- **ev_skiprows** (*0*) – Skip the first *skiprows* lines in *eigenvalues*. Default is `0`.
- **ev_usecols** (*list*[350]) – Which columns to read from *eigenvalues*, with 0 being the first. For example, `usecols = (1,4,5)` will extract the 2nd, 5th and 6th columns. The default, **None**, results in all columns being read.
- **ev_usevalues** (*list*[351]) – Which columns to use after the eigenvalue column is parsed from *eigenvalues*, with 0 being the first. This can be used if *eigenvalues* contains more values than the number of modes in *normalmodes*.

See *parseArray()* (page 155) for details of parsing arrays from files.

parseSparseMatrix(*filename*, *symmetric=False*, *delimiter=None*, *skiprows=0*, *irow=0*, *icol=1*, *first=1*)

Parse sparse matrix data from a file.

This function is using *parseArray()* (page 155) to parse the file. Input must have the following format:

```
1       1    9.958948135375977e+00
1       2   -3.788214445114136e+00
1       3    6.236155629158020e-01
1       4   -7.820609807968140e-01
```

Each row in the text file must have the same number of values.

Parameters

- **filename** (*str or file*) – File or filename to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed.
- **symmetric** (*bool*[352]) – Set **True** if the file contains triangular part of a symmetric matrix, default is **True**.
- **delimiter** (*str*[353]) – The string used to separate values. By default, this is any whitespace.
- **skiprows** (*int*[354]) – Skip the first *skiprows* lines, default is `0`.
- **irow** (*int*[355]) – Index of the column in data file corresponding to row indices, default is `0`.

[347] http://docs.python.org/library/stdtypes.html#str
[348] http://docs.python.org/library/stdtypes.html#list
[349] http://docs.python.org/library/stdtypes.html#str
[350] http://docs.python.org/library/stdtypes.html#list
[351] http://docs.python.org/library/stdtypes.html#list
[352] http://docs.python.org/library/functions.html#bool
[353] http://docs.python.org/library/stdtypes.html#str
[354] http://docs.python.org/library/functions.html#int
[355] http://docs.python.org/library/functions.html#int

- **icol** (*int*[356]) – Index of the column in data file corresponding to column indices, default is `1`.
- **first** (*int*[357]) – First index in the data file (0 or 1), default is `1`.

Data-type of the resulting array, default is `float()`.

parseGromacsModes(*run_path*, *title=''*, *model='nma'*, ***kwargs*)

Returns *NMA* (page 172) containing eigenvectors and eigenvalues parsed from a run directory containing results from gmx covar or gmx nmeig followed by gmx anaeig including eigenvalues in an xvg file and eigenvectors in pdb files (see http://www.strodel.info/index_files/lecture/html/analysis-9.html).

Parameters

- **run_path** (*str*[358]) – path to the run directory
- **title** (*str*[359]) – title for resulting object Default is `""`
- **model** (*str*[360]) – type of calculated that was performed. It can be either `"nma"` or `"pca"`. If it is not changed to `"pca"` then `"nma"` will be assumed.
- **eigval_fname** (*str*[361]) – filename or path for xvg file containing eigenvalues Default is `"eigenval.xvg"` as this is the default from Gromacs
- **eigvec_fname** (*str*[362]) – filename or path for trr file containing eigenvectors Default is `"eigenvec.trr"` as this is the default from Gromacs
- **pdb_fname** (*str*[363]) – filename or path for pdb file containing the reference structure Default is `"average.pdb"` although this is probably suboptimal

parseScipionModes(*metadata_file*, *title=None*, *pdb=None*, *parseIndices=False*)

Returns *NMA* (page 172) or *GNM* (page 164) containing eigenvectors and eigenvalues parsed from an NMA, GNM or PCA protocol path from ContinuousFlex or Scipion-EM-ProDy.

Parameters

- **metadata_file** (*str*[364]) – metadata sqlite file in Scipion protocol path The location of this file is currently limited to the top level of the project path and not to deep directories like the extra path.
- **title** (*str*[365]) – title for *NMA* (page 172) object
- **pdb** (*str*[366]) – pdb file to help define dof
- **parseIndices** (*bool*[367]) – whether to parse indices and output a ModeSet default *False*

writeArray(*filename*, *array*, *format='%3.2f'*, *delimiter=' '*)

Write 1-d or 2-d array data into a delimited text file.

[356] http://docs.python.org/library/functions.html#int
[357] http://docs.python.org/library/functions.html#int
[358] http://docs.python.org/library/stdtypes.html#str
[359] http://docs.python.org/library/stdtypes.html#str
[360] http://docs.python.org/library/stdtypes.html#str
[361] http://docs.python.org/library/stdtypes.html#str
[362] http://docs.python.org/library/stdtypes.html#str
[363] http://docs.python.org/library/stdtypes.html#str
[364] http://docs.python.org/library/stdtypes.html#str
[365] http://docs.python.org/library/stdtypes.html#str
[366] http://docs.python.org/library/stdtypes.html#str
[367] http://docs.python.org/library/functions.html#bool

This function is using `numpy.savetxt()` to write the file, after making some type and value checks. Default *format* argument is `"%d"`. Default *delimiter* argument is white space, `" "`.

filename will be returned upon successful writing.

writeModes(*filename*, *modes*, *format='%.18e'*, *delimiter=' '*)
Write *modes* (eigenvectors) into a plain text file with name *filename*. See also *writeArray()* (page 157).

writeScipionModes(*output_path*, *modes*, *write_star=False*, *scores=None*, *only_sqlite=False*, *collectivityThreshold=0.0*)
Writes *modes* to a set of files that can be recognised by Scipion. A directory called **"modes"** will be created if it doesn't already exist. Filenames inside will start with **"vec"** and have the mode number as the extension.

Parameters

- **output_path** (*str*[368]) – path to the directory where the modes directory will be
- **modes** (*Mode* (page 169), *ModeSet* (page 171), *NMA* (page 172)) – modes to be written to files
- **write_star** (*bool*[369]) – whether to write modes.xmd STAR file. Default is **False** as qualifyModesStep writes it with scores.
- **scores** (*list*[370]) – scores from qualifyModesStep for re-writing sqlite Default is **None** and then it uses *calcScipionScore()* (page 139)
- **only_sqlite** (*bool*[371]) – whether to write only the sqlite file instead of everything. Default is **False** but it can be useful to set it to **True** for updating the sqlite file.
- **collectivityThreshold** (*float*[372]) – collectivity threshold below which modes are not enabled Default is 0.

saveModel(*nma*, *filename=None*, *matrices=False*, ***kwargs*)
Save *nma* model data as `filename.nma.npz`. By default, eigenvalues, eigenvectors, variances, trace of covariance matrix, and name of the model will be saved. If *matrices* is **True**, covariance, Hessian or Kirchhoff matrices are saved too, whichever are available. If *filename* is **None**, name of the NMA instance will be used as the filename, after `" "` (white spaces) in the name are replaced with `"_"` (underscores). Extension may differ based on the type of the NMA model. For ANM models, it is `.anm.npz`. Upon successful completion of saving, filename is returned. This function makes use of `savez()` function.

loadModel(*filename*, ***kwargs*)
Returns NMA instance after loading it from file (*filename*). This function makes use of `load()` function. See also *saveModel()* (page 158).

saveVector(*vector*, *filename*, ***kwargs*)
Save *vector* data as `filename.vec.npz`. Upon successful completion of saving, filename is returned. This function makes use of `numpy.savez()` function.

loadVector(*filename*)
Returns *Vector* (page 170) instance after loading it from *filename* using `numpy.load()`. See also *saveVector()* (page 158).

[368] http://docs.python.org/library/stdtypes.html#str
[369] http://docs.python.org/library/functions.html#bool
[370] http://docs.python.org/library/stdtypes.html#list
[371] http://docs.python.org/library/functions.html#bool
[372] http://docs.python.org/library/functions.html#float

calcENM(*atoms*, *select=None*, *model='anm'*, *trim='trim'*, *gamma=1.0*, *title=None*, *n_modes=None*, ***kwargs*)

Returns an *ANM* (page 140) or *GNM* (page 164) instance and *atoms* used for the calculations. The model can be trimmed, sliced, or reduced based on the selection.

Parameters

- **atoms** (*Atomic* (page 47), *AtomGroup* (page 38), *Selection* (page 100), ndarray) – atoms on which the ENM is performed. It can be any Atomic class that supports selection or a ndarray.
- **select** (str, *Selection* (page 100), ndarray) – part of the atoms that is considered as the system. If set to **None**, then all atoms will be considered as the system
- **model** (*str*[373]) – type of ENM that will be performed. It can be either "anm" or "gnm" or "exanm"
- **trim** (*str*[374]) – type of method that will be used to trim the model. It can be either "trim" , "slice", or "reduce". If set to "trim", the parts that is not in the selection will simply be removed

realignModes(*modes*, *atoms*, *ref*)

Align *modes* in the original frame based on *atoms* onto another frame based on *ref* using the transformation from alignment of *atoms* to *ref*

Parameters

- **modes** (*ModeSet* (page 171), *ANM* (page 140), *PCA* (page 175)) – multiple 3D modes
- **atoms** (*Atomic* (page 47)) – central structure related to *modes* to map onto *ref* Inserting *atoms* into an ensemble and projecting onto *modes* should give all zeros
- **ref** (*Atomic* (page 47)) – reference structure for mapping

3.6.27 Custom Gamma Functions

This module defines customized gamma functions for elastic network model analysis.

class Gamma

Base class for facilitating use of atom type, residue type, or residue property dependent force constants (γ).

Derived classes:

- *GammaStructureBased* (page 159)
- *GammaVariableCutoff* (page 161)

gamma(*dist2*, *i*, *j*)

Returns force constant.

For efficiency purposes square of the distance between interacting atom/residue (node) pairs is passed to this function. In addition, node indices are passed.

class GammaStructureBased(*atoms*, *gamma=1.0*, *helix=6.0*, *sheet=6.0*, *connected=10.0*)

Facilitate setting the spring constant based on the secondary structure and connectivity of the residues.

A systematic study *[LT10]* (page 396) of a large set of NMR-structures analyzed using a method based on entropy maximization showed that taking into consideration properties such as sequential sep-

[373] http://docs.python.org/library/stdtypes.html#str
[374] http://docs.python.org/library/stdtypes.html#str

aration between contacting residues and the secondary structure types of the interacting residues provides refinement in the ENM description of proteins.

This class determines pairs of connected residues or pairs of proximal residues in a helix or a sheet, and assigns them a larger user defined spring constant value.

DSSP single letter abbreviations are recognized:

- **H**: α-helix
- **G**: 3-10-helix
- **I**: π-helix
- **E**: extended part of a sheet

helix: Applies to residue (or Cα atom) pairs that are in the same helical segment, at most 7 Å apart, and separated by at most 3 (3-10-helix), 4 (α-helix), or 5 (π-helix) residues.

sheet: Applies to Cα atom pairs that are in different β-strands and at most 6 Å apart.

connected: Applies to Cα atoms that are at most 4 Å apart.

Note that this class does not take into account insertion codes.

Example:

Let's parse coordinates and header data from a PDB file, and then assign secondary structure to the atoms.

```
In [1]: from prody import *

In [2]: ubi, header = parsePDB('1aar', chain='A', subset='calpha', header=True)

In [3]: assignSecstr(header, ubi)
Out[3]: <AtomGroup: 1aarA_ca (76 atoms)>
```

In the above we parsed only the atoms needed for this calculation, i.e. Cα atoms from chain A.

We build the Hessian matrix using structure based force constants as follows;

```
In [4]: gamma = GammaStructureBased(ubi)

In [5]: anm = ANM('')

In [6]: anm.buildHessian(ubi, gamma=gamma)
```

We can obtain the force constants assigned to residue pairs from the Kirchhoff matrix as follows:

```
In [7]: k = anm.getKirchhoff()

In [8]: k[0,1] # a pair of connected residues
Out[8]: -10.0

In [9]: k[0,16] # a pair of residues from a sheet
Out[9]: -6.0
```

Setup the parameters.

Parameters

- **atoms** (*Atomic* (page 47)) – A set of atoms with chain identifiers, residue numbers, and secondary structure assignments are set.

- **gamma** (*float*[375]) – Force constant in arbitrary units. Default is 1.0.
- **helix** (*float*[376]) – Force constant factor for residues hydrogen bonded in α-helices, 3,10-helices, and π-helices. Default is 6.0, i.e. `6.0`*gamma`.
- **sheet** (*float*[377]) – Force constant factor for residue pairs forming a hydrogen bond in a β-sheet. Default is 6.0, i.e. `6.0`*gamma`.
- **connected** (*float*[378]) – Force constant factor for residue pairs that are connected. Default is 10.0, i.e. `10.0`*gamma`.

gamma(*dist2, i, j*)
Returns force constant.

getChids()
Returns a copy of chain identifiers.

getResnums()
Returns a copy of residue numbers.

getSecstrs()
Returns a copy of secondary structure assignments.

class **GammaVariableCutoff**(*identifiers, gamma=1.0, default_radius=7.5, **kwargs*)
Facilitate setting the cutoff distance based on user defined atom/residue (node) radii.

Half of the cutoff distance can be thought of as the radius of a node. This class enables setting different radii for different node types.

Example:

Let's think of a protein-DNA complex for which we want to use different radius for different residue types. Let's say, for protein Cα atoms we want to set the radius to 7.5 Å, and for nucleic acid phosphate atoms to 10 Å. We use the HhaI-DNA complex structure `1mht`.

```
In [1]: hhai = parsePDB('1mht')

In [2]: ca_p = hhai.select('(protein and name CA) or (nucleic and name P)')

In [3]: ca_p.getNames()
Out[3]:
array(['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P',
       'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
```

[375] http://docs.python.org/library/functions.html#float
[376] http://docs.python.org/library/functions.html#float
[377] http://docs.python.org/library/functions.html#float
[378] http://docs.python.org/library/functions.html#float

```
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA'], dtype='|S6')
```

We set the radii of atoms:

```
In [4]: varcutoff = GammaVariableCutoff(ca_p.getNames(), gamma=1,
   ...:     default_radius=7.5, debug=False, P=10)
   ...:

In [5]: varcutoff.getRadii()
Out[5]:
array([10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. ,
       10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. , 10. ,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,
        7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5,  7.5])
```

The above shows that for phosphate atoms radii is set to 10 Å, because we passed the `P=10` argument.

As for Cα atoms, the default 7.5 Å is set as the radius (`default_radius=7.5`). You can also try this with `debug=True` argument to print debugging information on the screen.

We build *ANM* (page 140) Hessian matrix as follows:

```
In [6]: anm = ANM('HhaI-DNA')

In [7]: anm.buildHessian(ca_p, gamma=varcutoff, cutoff=20)
```

Note that we passed `cutoff=20.0` to the *ANM.buildHessian()* (page 140) method. This is equal to the largest possible cutoff distance (between two phosphate atoms) for this system, and ensures that all of the potential interactions are evaluated.

For pairs of atoms for which the actual distance is larger than the effective cutoff, the *GammaVariableCutoff.gamma()* (page 163) method returns `0`. This annuls the interaction between those atom pairs.

Set the radii of atoms.

Parameters

- **identifiers** (list or `numpy.ndarray`) – List of atom names or types, or residue names.
- **gamma** (*float*[379]) – Uniform force constant value. Default is 1.0.
- **default_radius** (*float*[380]) – Default radius for atoms whose radii is not set as a keyword argument. Default is 7.5

Keywords in keyword arguments must match those in *atom_identifiers*. Values of keyword arguments must be `float`[381].

gamma(*dist2*, *i*, *j*)
Returns force constant.

getGamma()
Returns the uniform force constant value.

getRadii()
Returns a copy of radii array.

class GammaED(*atoms*, *Ccart=6.0*, *Ex=6*, *Cseq=60.0*, *Slim=3*)
Facilitate setting the spring constant based on sequence distance and spatial distance as in ed-ENM *[OL10]* (page 396). This ENM is refined based on comparison to essential dynamics from MD simulations and can reproduce flexibility in NMR ensembles.

It has also been implemented in FlexServ *[CJ09]* (page 396) and used in MDdMD *[SP12]* (page 396) and GOdMD *[SP13]* (page 396).

The sequence distance-dependent term is Cseq/(S**2) for S=abs(i,j) <= Slim

The structure distance-dependent term is (Ccart/dist)**Ex

Example:

Let's parse coordinates from a PDB file.

```
In [1]: from prody import *

In [2]: ubi = parsePDB('1aar', chain='A', subset='calpha')
```

[379] http://docs.python.org/library/functions.html#float
[380] http://docs.python.org/library/functions.html#float
[381] http://docs.python.org/library/functions.html#float

In the above we parsed only the atoms needed for this calculation, i.e. Cα atoms from chain A.

We build the Hessian matrix using GOdMD spring constants as follows;

```
In [3]: gamma = GammaGOdMD(ubi)

In [4]: anm = ANM('')

In [5]: anm.buildHessian(ubi, gamma=gamma)
```

Setup the parameters.

Parameters

- **atoms** (*Atomic* (page 47)) – A set of atoms
- **Ccart** (*float*[382]) – Multiplication constant inside the exponential in the spatial distance-dependent term Default is 6.
- **Ex** – Exponent in spatial distance-dependent term Default is 6
- **Cseq** (*float*[383]) – Multiplication constant in sequence distance-dependent term Default is 60.
- **Slim** (*float*[384]) – Sequence distance limit for sequence distance-dependence This limit is used with a less-or-equal operator Default is 3

gamma (*dist2*, *i*, *j*)
Returns force constant.

GammaGOdMD
alias of *GammaED* (page 163)

3.6.28 Gaussian Network Model

This module defines a class and a function for Gaussian network model (GNM) calculations.

class GNM (*name='Unknown'*)
A class for Gaussian Network Model (GNM) analysis of proteins ([*IB97*] (page 396), [*TH97*] (page 396)).

See example Gaussian Network Model (GNM)[385].

addEigenpair (*vector*, *value=None*)
Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Inverse eigenvalues are set as variances.

buildKirchhoff (*coords*, *cutoff=10.0*, *gamma=1.0*, ***kwargs*)
Build Kirchhoff matrix for given coordinate set.

Parameters

- **coords** (`numpy.ndarray` or *Atomic* (page 47)) – a coordinate set or an object with `getCoords` method
- **cutoff** (*float*[386]) – cutoff distance (Å) for pairwise interactions default is 10.0 Å, , minimum is 4.0 Å

[382] http://docs.python.org/library/functions.html#float
[383] http://docs.python.org/library/functions.html#float
[384] http://docs.python.org/library/functions.html#float
[385] http://prody.csb.pitt.edu/tutorials/enm_analysis/gnm.html#gnm
[386] http://docs.python.org/library/functions.html#float

- **gamma** (*float*[387]) – spring constant, default is 1.0
- **sparse** (*bool*[388]) – elect to use sparse matrices, default is **False**. If Scipy is not found, `ImportError`[389] is raised.
- **kdtree** (*bool*[390]) – elect to use KDTree for building Kirchhoff matrix faster, default is **True**

Instances of `Gamma` classes and custom functions are accepted as *gamma* argument.

When Scipy is available, user can select to use sparse matrices for efficient usage of memory at the cost of computation speed.

calcModes(*n_modes=20*, *zeros=False*, *turbo=True*, ***kwargs*)

Calculate normal modes. This method uses `scipy.linalg.eigh()`[391] function to diagonalize the Kirchhoff matrix. When Scipy is not found, `numpy.linalg.eigh()` is used.

Parameters

- **n_modes** (*int or None, default is 20*) – number of non-zero eigenvalues/vectors to calculate. If **None** or `'all'` is given, all modes will be calculated.
- **zeros** (bool, default is **True**) – If **True**, modes with zero eigenvalues will be kept.
- **turbo** (bool, default is **True**) – Use a memory intensive, but faster way to calculate modes.

getArray()

Returns a copy of eigenvectors array.

getCovariance()

Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getCutoff()

Returns cutoff distance.

getEigvals()

Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is $Å^2$. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()

Returns a copy of eigenvectors array.

getGamma()

Returns spring constant (or the gamma function or `Gamma` instance).

getKirchhoff()

Returns a copy of the Kirchhoff matrix.

getModel()

Returns self.

getNormDistFluct(*coords*)

Normalized distance fluctuation

[387] http://docs.python.org/library/functions.html#float
[388] http://docs.python.org/library/functions.html#bool
[389] http://docs.python.org/library/exceptions.html#ImportError
[390] http://docs.python.org/library/functions.html#bool
[391] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

setKirchhoff(*kirchhoff*)
Set Kirchhoff matrix.

setTitle(*title*)
Set title of the model.

calcGNM(*pdb*, *selstr='calpha'*, *cutoff=10*, *gamma=1.0*, *n_modes=20*, *zeros=False*)
Returns a *GNM* (page 164) instance and atoms used for the calculations. By default only alpha carbons are considered, but selection string helps selecting a subset of it. *pdb* can be *Atomic* (page 47) instance.

3.6.29 Heatmapper Functions

This module defines functions for supporting VMD plugin Heatmapper[392] format files.

parseHeatmap(*heatmap*, ***kwargs*)
Returns a two dimensional array and a dictionary with information parsed from *heatmap*, which may be an input stream or an `.hm` file in VMD plugin Heat Mapper format.

writeHeatmap(*filename*, *heatmap*, ***kwargs*)
Returns *filename* that contains *heatmap* in Heat Mapper `.hm` file (extension is automatically added when not found). *filename* may also be an output stream.

Parameters

- **title** (*str*[393]) – title of the heatmap
- **xlabel** (*str*[394]) – x-axis lab, default is `'unknown'`
- **ylabel** (*str*[395]) – y-axis lab, default is `'unknown'`
- **xorigin** (*float*[396]) – x-axis origin, default is 0
- **xstep** (*float*[397]) – x-axis step, default is 1

[392] http://www.ks.uiuc.edu/Research/vmd/plugins/heatmapper/
[393] http://docs.python.org/library/stdtypes.html#str
[394] http://docs.python.org/library/stdtypes.html#str
[395] http://docs.python.org/library/stdtypes.html#str
[396] http://docs.python.org/library/functions.html#float
[397] http://docs.python.org/library/functions.html#float

- **min** (*float*[398]) – minimum value, default is minimum in *heatmap*
- **max** (*float*[399]) – maximum value, default is maximum in *heatmap*
- **format** (*str*[400]) – number format, default is `'%f'`

Other keyword arguments that are arrays with length equal to the y-axis (second dimension of heatmap) will be considered as *numbering*.

showHeatmap(*heatmap*, **args*, ***kwargs*)
Show *heatmap*, which can be an two dimensional array or a Heat Mapper `.hm` file.

Heatmap is plotted using `imshow()`[401] function. Default values passed to this function are `interpolation='nearest'`, `aspect='auto'`, and `origin='lower'`.

3.6.30 Implicit Membrane Anisotropic Network Model

This module defines a class and a function for explicit membrane ANM calculations.

class imANM(*name='Unknown'*)
Class for implicit ANM (imANM) method (*[FT00]* (page 397)).

addEigenpair(*vector*, *value=None*)
Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Inverse eigenvalues are set as variances.

buildHessian(*coords*, *blocks*, *cutoff=15.0*, *gamma=1.0*, ***kwargs*)
Build Hessian matrix for given coordinate set.

Parameters

- **coords** (`numpy.ndarray`) – a coordinate set or an object with `getCoords` method
- **blocks** (list, `numpy.ndarray`) – a list or array of block identifiers
- **cutoff** (*float*[402]) – cutoff distance (Å) for pairwise interactions, default is 15.0 Å
- **gamma** (*float*[403]) – spring constant, default is 1.0
- **scale** (*float*[404]) – scaling factor for force constant along Z-direction, default is 16.0
- **membrane_low** (*float*[405]) – minimum z-coordinate at which membrane scaling is applied default is 1.0
- **membrane_high** (*float*[406]) – maximum z-coordinate at which membrane scaling is applied. If membrane_high < membrane_low, scaling will be applied to the entire structure default is -1.0

calcModes(*n_modes=20*, *zeros=False*, *turbo=True*)
Calculate normal modes. This method uses `scipy.linalg.eigh()`[407] function to diagonalize the Hessian matrix. When Scipy is not found, `numpy.linalg.eigh()` is used.

[398] http://docs.python.org/library/functions.html#float
[399] http://docs.python.org/library/functions.html#float
[400] http://docs.python.org/library/stdtypes.html#str
[401] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow
[402] http://docs.python.org/library/functions.html#float
[403] http://docs.python.org/library/functions.html#float
[404] http://docs.python.org/library/functions.html#float
[405] http://docs.python.org/library/functions.html#float
[406] http://docs.python.org/library/functions.html#float
[407] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh

Parameters

- **n_modes** (*int or None, default is 20*) – number of non-zero eigenvalues/vectors to calculate. If **None** is given, all modes will be calculated.
- **zeros** (bool, default is **True**) – If **True**, modes with zero eigenvalues will be kept.
- **turbo** (bool, default is **True**) – Use a memory intensive, but faster way to calculate modes.

getArray()
Returns a copy of eigenvectors array.

getCovariance()
Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Returns a copy of eigenvectors array.

getHessian()
Returns a copy of the Hessian matrix.

getModel()
Returns self.

getProjection()
Returns a copy of the projection matrix.

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

setHessian(*hessian*)
Set Hessian matrix. A symmetric matrix is expected, i.e. not a lower- or upper-triangular matrix.

setTitle(*title*)
Set title of the model.

3.6.31 Mechanical Stiffness Calculations

This module defines functions for Mechanical Stiffness calculations.

calcMechStiff(*modes*, *coords*, *kbt=1.0*)
Calculate stiffness matrix calculated using *ANM* (page 140) instance. Method described in *[EB08]* (page 395) and [KMR17].

Parameters

- **coords** (numpy.ndarray.) – a coordinate set or an object with getCoords method
- **n_modes** (int or **None**, default is 20.) – number of non-zero eigenvalues/vectors to calculate. If **None** is given, all modes will be calculated (3x number of atoms).

Authors: Mustafa Tekpinar & Karolina Mikulska-Ruminska & Cihan Kaya

calcStiffnessRange(*stiffness*)
Return the range of effective spring constant.

calcMechStiffStatistic(*stiffness*, *rangeK*, *minAA=0*, *AA='all'*)
Returns number of effective spring constant with set range of amino acids of protein structure. AA can be a list with a range of analysed amino acids as: [first_aa, last_aa, first_aa2, last_aa2], minAA - eliminate amino acids that are within 20aa and rangeK is a list [minK, maxK]

calcStiffnessRangeSel(*stiffness*, *value*, *minAA=20*, *AA='all'*)
Returns minimum or maximum value of sping constant from mechanical stiffness calculations for residues that are within more than min_aa from each other. Value should be 'minK' or 'maxK'. It alow to avoid residues near each other. AA is a number of residues from both terminus (N and C) of protein strcuture, it can be all or int value (than first and last AA residues will be analysed. With minAA=0 it can be used to search the highest/lowest values of interactions between N-C terminus if protein structure has a shear, zipper or SD1-disconnected mechanical clamp -it is common in FnIII/Ig like domains and determines the maximum unfolding force in AFM or SMD method.

3.6.32 Normal Mode

This module defines classes for handling mode data.

class Mode(*model*, *index*)
A class to provide access to and operations on mode data.

Initialize mode object as part of an NMA model.

Parameters

- **model** (*NMA* (page 172), *GNM* (page 164), or *PCA* (page 175)) – a normal mode analysis instance
- **index** (*int*[408]) – index of the mode

getArray()
Returns a copy of the normal mode array (eigenvector).

getArrayNx3()
Returns a copy of array with shape (N, 3).

getEigval()
Returns normal mode eigenvalue. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140) and *GNM* (page 164), on the

[408] http://docs.python.org/library/functions.html#int

other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvals()
Returns normal mode eigenvalue. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140) and *GNM* (page 164), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvec()
Returns a copy of the normal mode array (eigenvector).

getEigvecs()
Returns a copy of the normal mode array (eigenvector).

getIndex()
Returns the index of the mode. Note that mode indices are zero-based.

getModel()
Returns the model that the mode instance belongs to.

getTitle()
A descriptive title for the mode instance.

getVariance()
Returns variance along the mode. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140) and *GNM* (page 164), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

getVariances()
Returns variance along the mode. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140) and *GNM* (page 164), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if mode instance is from a 3-dimensional model.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in the eigenvector.

numModes()
Returns 1.

class Vector(*array*, *title='Unknown'*, *is3d=True*)
A class to provide operations on a modified mode array. This class holds only mode array (i.e. eigenvector) data, and has no associations with an NMA instance. Scalar multiplication of *Mode* (page 169) instance or addition of two *Mode* (page 169) instances results in a *Vector* (page 170) instance.

Instantiate with a name, an array, and a 3d flag.

getArray()
Returns a copy of array.

getArrayNx3()
Returns a copy of array with shape (N, 3).

getNormed()
Returns mode after normalizing it.

getTitle()
Get the descriptive title for the vector instance.

is3d()
Returns **True** if vector instance describes a 3-dimensional property, such as a deformation for a set of atoms.

numAtoms()
Returns number of atoms. For a 3-dimensional vector, returns length of the vector divided by 3.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in the vector.

numModes()
Returns 1.

setTitle(*title*)
Set the descriptive title for the vector instance.

3.6.33 Mode Set

This module defines a pointer class for handling subsets of normal modes.

class ModeSet(*model*, *indices*)
A class for providing access to subset of mode data. Instances are obtained by slicing an NMA model (*ANM* (page 140), *GNM* (page 164), or *PCA* (page 175)). ModeSet's contain a reference to the model and a list of mode indices. Methods common to NMA models are also defined for mode sets.

getArray()
Returns a copy of eigenvectors array.

getCovariance()
Returns covariance matrix. It will be calculated using available modes.

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is $Å^2$. For *ANM* (page 140) and *GNM* (page 164), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Returns a copy of eigenvectors array.

getIndices()
Returns indices of modes in the mode set.

getModel()
Returns the model that the modes belongs to.

getTitle()
Returns title of the mode set.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is $Å^2$. For *ANM* (page 140) and *GNM* (page 164), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** is model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

3.6.34 Normal Mode Analysis

This module defines a class handling normal mode analysis data.

class NMA(*title='Unknown'*)
A class for handling Normal Mode Analysis (NMA) data.

addEigenpair(*vector*, *value=None*)
Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Inverse eigenvalues are set as variances.

getArray()
Returns a copy of eigenvectors array.

getCovariance()
Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is $Å^2$. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Returns a copy of eigenvectors array.

getModel()
Returns self.

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is $Å^2$. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

setEigens(*vectors*, *values=None*)
Set eigen *vectors* and eigen *values*. If eigen *values* are omitted, they will be set to 1. Inverse eigenvalues are set as variances.

setTitle(*title*)
Set title of the model.

3.6.35 NMD File

This module defines input and output functions for NMD format.

NMD Format

Description

NMD files (extension `.nmd`) are plain text files that contain at least normal mode and system coordinate data.

NMD files can be visualized using Normal Mode Wizard[409]. ProDy functions *writeNMD()* (page 174) and *parseNMD()* (page 174) can be used to read and write NMD files.

Data fields

Data fields in bold face are required. All data arrays and lists must be in a single line and items must be separated by one or more space characters.

coordinates: system coordinates as a list of decimal numbers Coordinate array is the most important line in an NMD file. All mode array lengths must match the length of the coordinate array. Also, number of atoms in the system is deduced from the length of the coordinate array.

```
coordinates 27.552 4.354 23.629 24.179 4.807 21.907 ...
```

mode: normal mode array as a list of decimal numbers Optionally, mode index and a scaling factor may be provided in the same line as a mode array. Both of these must precede the mode array. Providing a scaling factor enables relative scaling of the mode arrows and the amplitude of the fluctuations in animations. For NMA, scaling factors may be chosen to be the square-root of the inverse-eigenvalue associated with the mode. Analogously, for PCA data, scaling factor would be the square-root of the eigenvalue.

If a mode line contains numbers preceding the mode array, they are evaluated based on their type. If an integer is encountered, it is considered the mode index. If a decimal number is encountered, it is considered the scaling factor. Scaling factor may be the square-root of the inverse eigenvalue if data is from an elastic network model, or the square-root of the eigenvalue if data is from an essential dynamics (or principal component) analysis.

For example, all of the following lines are valid. The first line contains mode index and scaling factor. Second and third lines contain mode index or scaling factor. Last line contains only the mode array.

```
mode 1 2.37    0.039 0.009 0.058 0.038 -0.011 0.052  ...
mode 1    0.039 0.009 0.058 0.038 -0.011 0.052  ...
mode 2.37    0.039 0.009 0.058 0.038 -0.011 0.052  ...
mode 0.039 0.009 0.058 0.038 -0.011 0.052 0.043  ...
```

[409] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

name: name of the model

The length of all following data fields must be equal to the number of atoms in the system. NMWiz uses such data when writing a temporary PDB files for loading coordinate data into VMD.

***atomnames*: list of atom names** If not provided, all atom names are set to "CA".

***resnames*: list of residue names** If not provided, all residue names are set to "GLY".

***chainids*: list of chain identifiers** If not provided, all chain identifiers are set to "A".

***resids*: list of residue numbers** If not provided, residue numbers are started from 1 and incremented by one for each atom.

***bfactors*: list of experimental beta-factors** If not provided, all beta-factors are set to zero. Beta-factors can be used to color the protein representation.

NMD files may contain additional lines. Only lines that start with one of the above field names are evaluated by NMWiz.

Autoload Trick

By adding a special line in an NMD file, file content can be automatically loaded into VMD at startup. The first line calls a NMWiz function to load the file itself (`xyzeros.nmd`).

```
nmwiz_load xyzeros.nmd
coordinates 0 0 0 0 0 0  ...
mode 0.039 0.009 0.058 0.038 -0.011 0.052 ...
mode -0.045 -0.096 -0.009 -0.040 -0.076 -0.010 ...
mode 0.007 -0.044 0.080 0.015 -0.037 0.062 ...
```

In this case, VMD must be started from the command line by typing **`vmd -e xyzeros.nmd`**.

`parseNMD`(*filename*, *type=<class 'prody.dynamics.nma.NMA'>*)
Returns *NMA* (page 172) and *AtomGroup* (page 38) instances storing data parsed from *filename* in `.nmd` format. Type should be *NMA* (page 172) or a subclass such as *PCA* (page 175), *ANM* (page 140), or *GNM* (page 164).

`writeNMD`(*filename*, *modes*, *atoms*)
Returns *filename* that contains *modes* and *atoms* data in NMD format described in *NMD Format* (page 173). `.nmd` extension is appended to filename, if it does not have an extension.

Note:

1. This function skips modes with zero eigenvalues.
2. If a *Vector* (page 170) instance is given, it will be normalized before it is written. It's length before normalization will be written as the scaling factor of the vector.

`pathVMD`(**path*)
Returns VMD path, or set it to be a user specified *path*.

`getVMDpath`()
Deprecated for removal in v1.5, use *pathVMD()* (page 174) instead.

`setVMDpath`(*path*)
Deprecated for removal in v1.5, use *pathVMD()* (page 174) instead.

`viewNMDinVMD`(*filename*)
Start VMD in the current Python session and load NMD data.

3.6.36 Principal Component Analysis

This module defines classes for principal component analysis (PCA) and essential dynamics analysis (EDA) calculations.

class PCA(*name='Unknown'*)

A class for Principal Component Analysis (PCA) of conformational ensembles. See examples in Ensemble Analysis[410].

addEigenpair(*eigenvector*, *eigenvalue=None*)

Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Eigenvalues are set as variances.

buildCovariance(*coordsets*, ***kwargs*)

Build a covariance matrix for *coordsets* using mean coordinates as the reference. *coordsets* argument may be one of the following:

- *Atomic* (page 47)
- *Ensemble* (page 202)
- *TrajBase* (page 294)
- `numpy.ndarray` with shape `(n_csets, n_atoms, 3)`

For ensemble and trajectory objects, `update_coords=True` argument can be used to set the mean coordinates as the coordinates of the object.

When *coordsets* is a trajectory object, such as *DCDFile* (page 290), covariance will be built by superposing frames onto the reference coordinate set (see *Frame.superpose()* (page 293)). If frames are already aligned, use `aligned=True` argument to skip this step.

Note: If *coordsets* is a *PDBEnsemble* (page 207) instance, coordinates are treated specially. Let's say **C**_ij is the element of the covariance matrix that corresponds to atoms *i* and *j*. This super element is divided by number of coordinate sets (PDB models or structures) in which both of these atoms are observed together.

---

calcModes(*n_modes=20*, *turbo=True*, ***kwargs*)

Calculate principal (or essential) modes. This method uses `scipy.linalg.eigh()`[411], or `numpy.linalg.eigh()`, function to diagonalize the covariance matrix.

Parameters

- **n_modes** (*int*[412]) – number of non-zero eigenvalues/vectors to calculate, default is 20, if **None** or `'all'` is given, all modes will be calculated
- **turbo** (*bool*[413]) – when available, use a memory intensive but faster way to calculate modes, default is **True**

getArray()

Returns a copy of eigenvectors array.

getCovariance()

Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

[410] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#pca
[411] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[412] http://docs.python.org/library/functions.html#int
[413] http://docs.python.org/library/functions.html#bool

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Returns a copy of eigenvectors array.

getModel()
Returns self.

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

performSVD(*coordsets*)
Calculate principal modes using singular value decomposition (SVD). *coordsets* argument may be a *Atomic* (page 47), *Ensemble* (page 202), or `numpy.ndarray` instance. If *coordsets* is a numpy array, its shape must be `(n_csets, n_atoms, 3)`. Note that coordinate sets must be aligned prior to SVD calculations.

This is a considerably faster way of performing PCA calculations compared to eigenvalue decomposition of covariance matrix, but is an approximate method when heterogeneous datasets are analyzed. Covariance method should be preferred over this one for analysis of ensembles with missing atomic data. See Calculations[414] example for comparison of results from SVD and covariance methods.

setCovariance(*covariance*, *is3d=True*)
Set covariance matrix.

setEigens(*vectors*, *values=None*)
Set eigen *vectors* and eigen *values*. If eigen *values* are omitted, they will be set to 1. Eigenvalues are set as variances.

setTitle(*title*)
Set title of the model.

class EDA(*name='Unknown'*)

[414] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_calculations.html#pca-xray-calculations

A class for Essential Dynamics Analysis (EDA) *[AA93]* (page 396). See examples in Essential Dynamics Analysis[415].

addEigenpair(*eigenvector*, *eigenvalue=None*)
Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Eigenvalues are set as variances.

buildCovariance(*coordsets*, ***kwargs*)
Build a covariance matrix for *coordsets* using mean coordinates as the reference. *coordsets* argument may be one of the following:

- *Atomic* (page 47)
- *Ensemble* (page 202)
- *TrajBase* (page 294)
- `numpy.ndarray` with shape `(n_csets, n_atoms, 3)`

For ensemble and trajectory objects, `update_coords=True` argument can be used to set the mean coordinates as the coordinates of the object.

When *coordsets* is a trajectory object, such as *DCDFile* (page 290), covariance will be built by superposing frames onto the reference coordinate set (see *Frame.superpose()* (page 293)). If frames are already aligned, use `aligned=True` argument to skip this step.

Note: If *coordsets* is a *PDBEnsemble* (page 207) instance, coordinates are treated specially. Let's say **C**_ij is the element of the covariance matrix that corresponds to atoms *i* and *j*. This super element is divided by number of coordinate sets (PDB models or structures) in which both of these atoms are observed together.

---

calcModes(*n_modes=20*, *turbo=True*, ***kwargs*)
Calculate principal (or essential) modes. This method uses `scipy.linalg.eigh()`[416], or `numpy.linalg.eigh()`, function to diagonalize the covariance matrix.

Parameters

- **n_modes** (*int*[417]) – number of non-zero eigenvalues/vectors to calculate, default is 20, if **None** or `'all'` is given, all modes will be calculated
- **turbo** (*bool*[418]) – when available, use a memory intensive but faster way to calculate modes, default is **True**

getArray()
Returns a copy of eigenvectors array.

getCovariance()
Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

[415] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda
[416] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[417] http://docs.python.org/library/functions.html#int
[418] http://docs.python.org/library/functions.html#bool

getEigvecs()
Returns a copy of eigenvectors array.

getModel()
Returns self.

getTitle()
Returns title of the model.

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

performSVD(*coordsets*)
Calculate principal modes using singular value decomposition (SVD). *coordsets* argument may be a *Atomic* (page 47), *Ensemble* (page 202), or `numpy.ndarray` instance. If *coordsets* is a numpy array, its shape must be `(n_csets, n_atoms, 3)`. Note that coordinate sets must be aligned prior to SVD calculations.

This is a considerably faster way of performing PCA calculations compared to eigenvalue decomposition of covariance matrix, but is an approximate method when heterogeneous datasets are analyzed. Covariance method should be preferred over this one for analysis of ensembles with missing atomic data. See Calculations[419] example for comparison of results from SVD and covariance methods.

setCovariance(*covariance*, *is3d=True*)
Set covariance matrix.

setEigens(*vectors*, *values=None*)
Set eigen *vectors* and eigen *values*. If eigen *values* are omitted, they will be set to 1. Eigenvalues are set as variances.

setTitle(*title*)
Set title of the model.

3.6.37 Perturbation Response Scanning

This module defines functions for performing perturbation response scanning from PCA and normal modes.

calcPerturbResponse(*model*, ***kwargs*)
This function implements the perturbation response scanning (PRS) method described in *[CA09]* (page 397) and *[IG14]* (page 396). It returns a PRS matrix, and effectiveness and sensitivity profiles.

[419] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_calculations.html#pca-xray-calculations

Rows of the matrix are the average magnitude of the responses obtained by perturbing the atom/node position at that row index, i.e. `prs_matrix[i,j]` will give the response of residue/node *j* to perturbations in residue/node *i*.

PRS is performed using the covariance matrix from a *model*, e.g. a *ANM* (page 140) instance. To use an external matrix, please provide it to a *PCA* (page 175) instance using the *PCA.setCovariance()* (page 176).

When an *atoms* instance is given, the PRS matrix will be added as data, which can be retrieved with `atoms.getData('prs_matrix')`.

model and *atoms* must have the same number of atoms. *atoms* must be an *AtomGroup* (page 38) instance.

If *turbo* is **True** (default), then PRS is approximated by the limit of large numbers of forces and no perturbation forces are explicitly applied. If set to **False**, then each residue/node is perturbed *repeats* times (default 100) with a random unit force vector as in ProDy v1.8 and earlier.

calcDynamicFlexibilityIndex(*matrix*, *atoms*, *select*, ***kwargs*)

Calculate the dynamic flexibility index for the selected residue(s). This function implements the dynamic flexibility index (Dfi) method described in *[ZNG13]* (page 396).

Parameters

- **matrix** (ndarray, *ANM* (page 140), *PCA* (page 175), *GNM* (page 164)) – PRS or covariance matrix, or a model from which to calculate them
- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **select** (str, *Selection* (page 100)) – a selection string or selection for residues of interest
- **norm** (*bool*[420]) – whether to normalise the covariance to a PRS matrix, default False This option is only valid when providing a model.

calcDynamicCouplingIndex(*matrix*, *atoms*, *select*, *func_sel*, ***kwargs*)

Calculate the dynamic coupling index for the selected residue(s). This function implements the dynamic coupling index (DCI) or functional DFI method described in *[AK15]* (page 397).

Parameters

- **matrix** (ndarray, *ANM* (page 140), *PCA* (page 175), *GNM* (page 164)) – PRS or covariance matrix, or a model from which to calculate them
- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **select** (str, *Selection* (page 100)) – a selection string or selection for residues of interest
- **func_sel** (str, *Selection* (page 100)) – a selection string or selection for functional residues
- **norm** (*bool*[421]) – whether to normalise the covariance to a PRS matrix, default False This option is only valid when providing a model.

3.6.38 Plotting Functions

This module defines plotting functions for protein dynamics analysis.

[420] http://docs.python.org/library/functions.html#bool
[421] http://docs.python.org/library/functions.html#bool

Plotting functions are called by the name of the plotted data/property and are prefixed with `show`. Function documentations refers to the `matplotlib.pyplot`[422] function utilized for actual plotting. Arguments and keyword arguments are passed to the Matplotlib functions.

showContactMap(*enm*, ***kwargs*)
Show contact map using *showAtomicMatrix()* (page 184). *enm* can be either a *GNM* (page 164) or *Atomic* (page 47) object.

showCrossCorr(*modes*, **args*, ***kwargs*)
Show cross-correlations using *showAtomicMatrix()* (page 184). By default, *origin=lower* and *interpolation=bilinear* keyword arguments are passed to this function, but user can overwrite these parameters. See also *calcCrossCorr()* (page 137).

showCovarianceMatrix(*modes*, **args*, ***kwargs*)
Show 3Nx3N covariance matrix (or NxN matrix for GNM) using *showAtomicMatrix()* (page 184). By default, *origin=lower* and *interpolation=bilinear* keyword arguments are passed to this function, but user can overwrite these parameters. See also *calcCovariance()* (page 137).

showCumulOverlap(*mode*, *modes*, **args*, ***kwargs*)
Show cumulative overlap using `plot()`[423].

Parameters **modes** (*ModeSet* (page 171), *ANM* (page 140), *GNM* (page 164), *PCA* (page 175)) – multiple modes

showFractVars(*modes*, **args*, ***kwargs*)
Show fraction of variances using `bar()`[424]. Note that mode indices are incremented by 1.

showCumulFractVars(*modes*, **args*, ***kwargs*)
Show fraction of variances of *modes* using `plot()`. Note that mode indices are incremented by 1. See also *showFractVars()* (page 180) function.

showMode(*mode*, **args*, ***kwargs*)
Show mode array using `plot()`[425].

showOverlap(*mode*, *modes*, **args*, ***kwargs*)
Show overlap `bar()`[426].

Parameters

- **mode** (*Mode* (page 169), *Vector* (page 170), *ModeSet* (page 171), *ANM* (page 140), *GNM* (page 164), *PCA* (page 175)) – a single mode/vector or multiple modes. If multiple modes are provided, then the overlaps are calculated by going through them one by one, i.e. mode i from this set is compared with mode i from the other set.
- **modes** (*ModeSet* (page 171), *ANM* (page 140), *GNM* (page 164), *PCA* (page 175)) – multiple modes

showOverlaps(*mode*, *modes*, **args*, ***kwargs*)
Show overlap `bar()`[427].

Parameters

- **mode** (*Mode* (page 169), *Vector* (page 170), *ModeSet* (page 171), *ANM* (page 140), *GNM* (page 164), *PCA* (page 175)) – a single mode/vector or multiple modes. If multiple modes are provided, then the overlaps are calculated by going through

[422] http://matplotlib.sourceforge.net/api/pyplot_summary.html#module-matplotlib.pyplot
[423] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
[424] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar
[425] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
[426] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar
[427] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar

them one by one, i.e. mode i from this set is compared with mode i from the other set.

- **modes** (*ModeSet* (page 171), *ANM* (page 140), *GNM* (page 164), *PCA* (page 175)) – multiple modes

showOverlapTable(*modes_x*, *modes_y*, ***kwargs*)

Show overlap table using `pcolor()`[428]. *modes_x* and *modes_y* are sets of normal modes, and correspond to x and y axes of the plot. Note that mode indices are incremented by **1**. List of modes is assumed to contain a set of contiguous modes from the same model.

Default arguments for `pcolor()`[429]:

- `cmap='jet'`
- `norm=matplotlib.colors.Normalize(0, 1)`

showProjection(*ensemble=None*, *modes=None*, *projection=None*, **args*, ***kwargs*)

Show a projection of conformational deviations onto up to three normal modes from the same model.

Parameters

- **ensemble** (*Ensemble* (page 202), *Conformation* (page 201), *Vector* (page 170), *Trajectory* (page 296)) – an ensemble, trajectory or a conformation for which deviation(s) will be projected, or a deformation vector
- **modes** (*Mode* (page 169), *ModeSet* (page 171), *NMA* (page 172)) – up to three normal modes
- **show_density** (*bool*[430]) – whether to show a density histogram or kernel density estimate rather than a 2D scatter of points or a 1D projection by time (number of steps) on the x-axis. This option is not valid for 3D projections. Default is **True** for 1D and **False** for 2D to maintain old behaviour.
- **use_weights** (*bool*[431]) – whether to use weights in a density histogram or kernel density estimate or for the size of points in 2D scatter of points or a 1D projection by time (number of steps) on the x-axis. This option is not valid for 3D projections. Default is **False** to maintain old behaviour.
- **weights** (int, list, `ndarray`) – weights for histograms or point sizes Default is to use ensemble.getData('size')
- **color** (*str, list*) – a color name or a list of color names or values, default is `'blue'`
- **label** (*str, list*) – label or a list of labels
- **marker** (*str, list*) – a marker or a list of markers, default is `'o'`
- **linestyle** (*str*[432]) – line style, default is `'None'`
- **text** (*list*[433]) – list of text labels, one for each conformation
- **fontsize** (*int*[434]) – font size for text labels

The projected values are by default converted to RMSD. Pass `rmsd=False` to use projection itself.

Matplotlib function used for plotting depends on the number of modes:

[428] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.pcolor.html#matplotlib.pyplot.pcolor
[429] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.pcolor.html#matplotlib.pyplot.pcolor
[430] http://docs.python.org/library/functions.html#bool
[431] http://docs.python.org/library/functions.html#bool
[432] http://docs.python.org/library/stdtypes.html#str
[433] http://docs.python.org/library/stdtypes.html#list
[434] http://docs.python.org/library/functions.html#int

- 1 mode: `hist()`[435]
- 2 modes: `scatter()`[436]
- 3 modes: `scatter()`

showCrossProjection(*ensemble*, *mode_x*, *mode_y*, *scale=None*, **args*, ***kwargs*)

Show a projection of conformational deviations onto modes from different models using `plot()`[437]. This function differs from *showProjection()* (page 181) by accepting modes from two different models.

Parameters

- **ensemble** (*Ensemble* (page 202), *Conformation* (page 201), *Vector* (page 170), *Trajectory* (page 296)) – an ensemble or a conformation for which deviation(s) will be projected, or a deformation vector
- **mode_x** (*Mode* (page 169), *Vector* (page 170)) – projection onto this mode will be shown along x-axis
- **mode_y** (*Mode* (page 169), *Vector* (page 170)) – projection onto this mode will be shown along y-axis
- **scale** (*str*[438]) – scale width of the projection onto mode `x` or `y`, best scaling factor will be calculated and printed on the console, absolute value of scalar makes the with of two projection same, sign of scalar makes the projections yield a positive correlation
- **scalar** (*float*[439]) – scalar factor for projection onto selected mode
- **color** (*str, list*) – a color name or a list of color name, default is `'blue'`
- **label** (*str, list*) – label or a list of labels
- **marker** (*str, list*) – a marker or a list of markers, default is `'o'`
- **linestyle** (*str*[440]) – line style, default is `'None'`
- **text** (*list*[441]) – list of text labels, one for each conformation
- **fontsize** (*int*[442]) – font size for text labels

This function uses calcProjection and its arguments can be passed to it as keyword arguments.

The projected values are by default converted to RMSD. Pass `rmsd=False` to calculate raw projection values. See Plotting[443] for a more elaborate example.

Likewise, normalisation is applied by default and can be turned off with `norm=False`.

showEllipsoid(*modes*, *onto=None*, *n_std=2*, *scale=1.0*, **args*, ***kwargs*)

Show an ellipsoid using `plot_wireframe()`.

Ellipsoid volume gives an analytical view of the conformational space that given modes describe.

Parameters

[435] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.hist.html#matplotlib.pyplot.hist
[436] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter
[437] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
[438] http://docs.python.org/library/stdtypes.html#str
[439] http://docs.python.org/library/functions.html#float
[440] http://docs.python.org/library/stdtypes.html#str
[441] http://docs.python.org/library/stdtypes.html#list
[442] http://docs.python.org/library/functions.html#int
[443] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_plotting.html#pca-xray-plotting

- **modes** (*ModeSet* (page 171), *PCA* (page 175), *ANM* (page 140), *NMA* (page 172)) – 3 modes for which ellipsoid will be drawn.
- **onto** – 3 modes onto which ellipsoid will be projected.
- **n_std** (*float*[444]) – Number of standard deviations to scale the ellipsoid.
- **scale** (*float*[445]) – Used for scaling the volume of ellipsoid. This can be obtained from *sampleModes()* (page 189).

showSqFlucts(*modes*, **args*, ***kwargs*)
Show square fluctuations using *showAtomicLines()* (page 186). See also *calcSqFlucts()* (page 137).

showRMSFlucts(*modes*, **args*, ***kwargs*)
Show square fluctuations using *showAtomicLines()* (page 186). See also *calcRMSFlucts()* (page 137).

showScaledSqFlucts(*modes*, **args*, ***kwargs*)
Show scaled square fluctuations using `plot()`[446]. Modes or mode sets given as additional arguments will be scaled to have the same mean squared fluctuations as *modes*.

showNormedSqFlucts(*modes*, **args*, ***kwargs*)
Show normalized square fluctuations via `plot()`[447].

resetTicks(*x*, *y=None*)
Reset X (and Y) axis ticks using values in given *array*. Ticks in the current figure should not be fractional values for this function to work as expected.

showDiffMatrix(*matrix1*, *matrix2*, **args*, ***kwargs*)
Show the difference between two cross-correlation matrices from different models. For given *matrix1* and *matrix2* show the difference between them in the form of (matrix2 - matrix1) and plot the difference matrix using *showAtomicMatrix()* (page 184). When *NMA* (page 172) models are passed instead of matrices, the functions could call *calcCrossCorr()* (page 137) function to calculate the matrices for given modes.

To display the absolute values in the difference matrix, user could set *abs* keyword argument **True**.

By default, `origin="lower"` and `interpolation="bilinear"` keyword arguments are passed to this function, but user can overwrite these parameters.

showMechStiff(*stiffness*, *atoms*, ***kwargs*)
Show mechanical stiffness matrix using `imshow()`[448]. By default, `origin="lower"` keyword arguments are passed to this function, but user can overwrite these parameters.

showNormDistFunct(*model*, *coords*, ***kwargs*)
Show normalized distance fluctuation matrix using `imshow()`[449]. By default, `origin="lower"` keyword arguments are passed to this function, but user can overwrite these parameters.

showPairDeformationDist(*model*, *coords*, *ind1*, *ind2*, **args*, ***kwargs*)
Show distribution of deformations in distance contributed by each mode for selected pair of residues *ind1 ind2* using `plot()`[450].

showMeanMechStiff(*stiffness*, *atoms*, *header*, *chain='A'*, **args*, ***kwargs*)
Show mean value of effective spring constant with secondary structure taken from MechStiff. Header

[444] http://docs.python.org/library/functions.html#float
[445] http://docs.python.org/library/functions.html#float
[446] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
[447] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
[448] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow
[449] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow
[450] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

is needed to obatin secondary structure range. Using `"jet_r"` as argument color map will be reverse (similar to VMD program coding).

showPerturbResponse(*model*, *atoms=None*, *show_matrix=True*, *select=None*, ***kwargs*)
Plot the PRS matrix with the profiles along the right and bottom.

If atoms are provided then residue numbers can be used from there. *model* and *atoms* must have the same number of atoms. *atoms* must be an *Atomic* (page 47) instance.

Parameters

- **model** (*NMA* (page 172), ndarray) – any object with a `calcCovariance()` method from which to calculate a PRS matrix (e.g. *ANM* (page 140) instance) or a PRS matrix itself
- **atoms** (*AtomGroup* (page 38)) – a :class: *AtomGroup* instance for matching residue numbers and chain identifiers
- **select** – a `Selection` instance or selection string for showing residue-specific profiles. This can only be used with `show_matrix=False`.
- **show_matrix** (*bool*[451]) – whether to show the matrix, default is **True**
- **suppress_diag** (*bool*[452]) – whether to suppress the diagonal default is **True**

Tye select `Selection`, str

showTree(*tree*, *format='matplotlib'*, ***kwargs*)
Given a tree, creates visualization in different formats.

arg tree: Tree needs to be unrooted and should be generated by tree generator from Phylo in biopython, which is used by `calcTree()`

type tree: `Tree`

arg format: depending on the format, you will see different forms of trees. Acceptable formats are `"plt"` (or `"mpl"` or `"matplotlib"`), `"ascii"` and `"networkx"`. Default is `"matplotlib"`.

type format: str

keyword font_size: font size for branch labels type font_size: float

keyword line_width: the line width for each branch type line_width: float

showTree_networkx(*tree*, *node_size=20*, *node_color='red'*, *node_shape='o'*, *withlabels=True*, *scale=1.0*, *iterations=500*, *k=None*, ***kwargs*)
Given a tree, creates visualization using `networkx`. See `spring_layout()` and `draw_networkx_nodes()` for more details.

arg tree: Tree needs to be unrooted and should be generated by tree generator from Phylo in biopython, which is used by `calcTree()`

type tree: `Tree`

showAtomicMatrix(*matrix*, *x_array=None*, *y_array=None*, *atoms=None*, ***kwargs*)
Show a matrix using `imshow()`[453]. Curves on x- and y-axis can be added. The first return value is the `Axes`[454] object for the upper plot, and the second return value is equivalent object for the left plot. The

[451] http://docs.python.org/library/functions.html#bool
[452] http://docs.python.org/library/functions.html#bool
[453] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.imshow.html#matplotlib.axes.Axes.imshow
[454] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes

third return value is the AxesImage[455] object for the matrix plot. The last return value is the Axes[456] object for the color bar.

Parameters

- **matrix** (ndarray) – matrix to be displayed
- **x_array** (ndarray) – data to be plotted above the matrix
- **y_array** (ndarray) – data to be plotted on the left side of the matrix
- **percentile** (*float*[457]) – A percentile threshold to remove outliers, i.e. only showing data within *p*-th to *100-p*-th percentile.
- **atoms** – a :class: *AtomGroup* instance for matching residue numbers and chain identifiers
- **chain** (*bool*[458]) – display a bar at the bottom to show chain separations. If set to **None**, it will be decided depends on whether *atoms* is provided. Default is **None**.
- **domain** (*bool*[459]) – the same with *chains* but show domain separations instead. *atoms* needs to have *domain* data associated to it. Default is **None**.
- **figure** (Figure[460], int, str) – if set to **None**, then a new figure will be created if *auto_show* is *True*, otherwise it will be plotted on the current figure. If set to a figure number or a Figure[461] instance, no matter what 'auto_show' value is, plots will be drawn on the *figure*. Default is **None**.
- **interactive** (*bool*[462]) – turn on or off the interactive options

pimshow(*matrix*, *x_array=None*, *y_array=None*, *atoms=None*, ***kwargs*)

Show a matrix using imshow()[463]. Curves on x- and y-axis can be added. The first return value is the Axes[464] object for the upper plot, and the second return value is equivalent object for the left plot. The third return value is the AxesImage[465] object for the matrix plot. The last return value is the Axes[466] object for the color bar.

Parameters

- **matrix** (ndarray) – matrix to be displayed
- **x_array** (ndarray) – data to be plotted above the matrix
- **y_array** (ndarray) – data to be plotted on the left side of the matrix
- **percentile** (*float*[467]) – A percentile threshold to remove outliers, i.e. only showing data within *p*-th to *100-p*-th percentile.
- **atoms** – a :class: *AtomGroup* instance for matching residue numbers and chain identifiers

[455] http://matplotlib.sourceforge.net/api/image_api.html#matplotlib.image.AxesImage
[456] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[457] http://docs.python.org/library/functions.html#float
[458] http://docs.python.org/library/functions.html#bool
[459] http://docs.python.org/library/functions.html#bool
[460] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[461] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[462] http://docs.python.org/library/functions.html#bool
[463] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.imshow.html#matplotlib.axes.Axes.imshow
[464] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[465] http://matplotlib.sourceforge.net/api/image_api.html#matplotlib.image.AxesImage
[466] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes
[467] http://docs.python.org/library/functions.html#float

- **chain** (*bool*[468]) – display a bar at the bottom to show chain separations. If set to **None**, it will be decided depends on whether *atoms* is provided. Default is **None**.
- **domain** (*bool*[469]) – the same with *chains* but show domain separations instead. *atoms* needs to have *domain* data associated to it. Default is **None**.
- **figure** (Figure[470], int, str) – if set to **None**, then a new figure will be created if *auto_show* is *True*, otherwise it will be plotted on the current figure. If set to a figure number or a Figure[471] instance, no matter what 'auto_show' value is, plots will be drawn on the *figure*. Default is **None**.
- **interactive** (*bool*[472]) – turn on or off the interactive options

showAtomicLines(**args*, ***kwargs*)

Show a plot with the option to use residue numbers and include chain/domain color bars using provided atoms.

Parameters

- **atoms** – a :class: *AtomGroup* instance for matching residue numbers and chain identifiers.
- **chain** (*bool*[473]) – display a bar at the bottom to show chain separations. If set to **None**, it will be decided depends on whether *atoms* is provided. Default is **None**.
- **domain** (*bool*[474]) – the same as *chain* but show domain separations instead. *atoms* needs to have *domain* data associated to it. Default is **None**.
- **gap** (*bool*[475]) – whether to show the gaps in the *atoms* or not. Default is **False**.
- **overlay** (*bool*[476]) – whether to overlay the curves based on the chain separations in *atoms* or not. Default is **False**.
- **figure** (Figure[477], int, str) – if set to **None**, then a new figure will be created if *auto_show* is **True**, otherwise it will be plotted on the current figure. If set to a figure number or string or a Figure[478] instance, no matter what 'auto_show' value is, plots will be drawn on the *figure*. Default is **None**.
- **final** (*bool*[479]) – if set to **False**, *chain* and *domain* will be set to **False** no matter what their values are. This is used to stack plots onto one another, and show only one domain/chain bar.

pplot(**args*, ***kwargs*)

Show a plot with the option to use residue numbers and include chain/domain color bars using provided atoms.

Parameters

- **atoms** – a :class: *AtomGroup* instance for matching residue numbers and chain identifiers.

[468] http://docs.python.org/library/functions.html#bool
[469] http://docs.python.org/library/functions.html#bool
[470] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[471] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[472] http://docs.python.org/library/functions.html#bool
[473] http://docs.python.org/library/functions.html#bool
[474] http://docs.python.org/library/functions.html#bool
[475] http://docs.python.org/library/functions.html#bool
[476] http://docs.python.org/library/functions.html#bool
[477] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[478] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[479] http://docs.python.org/library/functions.html#bool

- **chain** (*bool*[480]) – display a bar at the bottom to show chain separations. If set to **None**, it will be decided depends on whether *atoms* is provided. Default is **None**.
- **domain** (*bool*[481]) – the same as *chain* but show domain separations instead. *atoms* needs to have *domain* data associated to it. Default is **None**.
- **gap** (*bool*[482]) – whether to show the gaps in the *atoms* or not. Default is **False**.
- **overlay** (*bool*[483]) – whether to overlay the curves based on the chain separations in *atoms* or not. Default is **False**.
- **figure** (Figure[484], int, str) – if set to **None**, then a new figure will be created if *auto_show* is **True**, otherwise it will be plotted on the current figure. If set to a figure number or string or a Figure[485] instance, no matter what 'auto_show' value is, plots will be drawn on the *figure*. Default is **None**.
- **final** (*bool*[486]) – if set to **False**, *chain* and *domain* will be set to **False** no matter what their values are. This is used to stack plots onto one another, and show only one domain/chain bar.

showDomainBar(*domains*, *x=None*, *loc=0.0*, *axis='x'*, ***kwargs*)

Plot a bar on top of the current axis which is colored based on domain separations.

Parameters

- **domains** (list, tuple, ndarray) – a list of domain labels
- **loc** (*float*[487]) – relative position of the domain bar. **0** means at bottom/left and **1** means at top/right
- **axis** (*str*[488]) – on which axis the bar will be plotted. It can be either **x** or **y**
- **text** (*bool*[489]) – whether to show the text or not. Default is **True**
- **text_loc** (*str*[490]) – location of text labels. It can be either **above** or **below**
- **text_color** (*str, tuple, list*) – color of the text labels
- **color** (*dict*[491]) – a dictionary of colors where keys are the domain names
- **relim** (*bool*[492]) – whether to rescale the axes' limits after adding the bar. Default is **True**

3.6.39 Rotation Translation Blocks

This module defines a class and a function for rotating translating blocks (RTB) calculations.

class RTB(*name='Unknown'*)

Class for Rotations and Translations of Blocks (RTB) method ([*FT00*] (page 397)).

[480] http://docs.python.org/library/functions.html#bool
[481] http://docs.python.org/library/functions.html#bool
[482] http://docs.python.org/library/functions.html#bool
[483] http://docs.python.org/library/functions.html#bool
[484] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[485] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure
[486] http://docs.python.org/library/functions.html#bool
[487] http://docs.python.org/library/functions.html#float
[488] http://docs.python.org/library/stdtypes.html#str
[489] http://docs.python.org/library/functions.html#bool
[490] http://docs.python.org/library/stdtypes.html#str
[491] http://docs.python.org/library/stdtypes.html#dict
[492] http://docs.python.org/library/functions.html#bool

addEigenpair(*vector*, *value=None*)
Add eigen *vector* and eigen *value* pair(s) to the instance. If eigen *value* is omitted, it will be set to 1. Inverse eigenvalues are set as variances.

buildHessian(*coords*, *blocks*, *cutoff=15.0*, *gamma=1.0*, ***kwargs*)
Build Hessian matrix for given coordinate set.

Parameters

- **coords** (`numpy.ndarray`) – a coordinate set or an object with `getCoords` method
- **blocks** (list, `numpy.ndarray`) – a list or array of block identifiers
- **cutoff** (*float*[493]) – cutoff distance (Å) for pairwise interactions, default is 15.0 Å
- **gamma** (*float*[494]) – spring constant, default is 1.0

calcModes(*n_modes=20*, *zeros=False*, *turbo=True*)
Calculate normal modes. This method uses `scipy.linalg.eigh()`[495] function to diagonalize the Hessian matrix. When Scipy is not found, `numpy.linalg.eigh()` is used.

Parameters

- **n_modes** (*int or None, default is 20*) – number of non-zero eigenvalues/vectors to calculate. If **None** is given, all modes will be calculated.
- **zeros** (bool, default is **True**) – If **True**, modes with zero eigenvalues will be kept.
- **turbo** (bool, default is **True**) – Use a memory intensive, but faster way to calculate modes.

getArray()
Returns a copy of eigenvectors array.

getCovariance()
Returns covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getEigvals()
Returns eigenvalues. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of eigenvalues is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Returns a copy of eigenvectors array.

getHessian()
Returns a copy of the Hessian matrix.

getModel()
Returns self.

getProjection()
Returns a copy of the projection matrix.

getTitle()
Returns title of the model.

[493] http://docs.python.org/library/functions.html#float
[494] http://docs.python.org/library/functions.html#float
[495] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh

getVariances()
Returns variances. For *PCA* (page 175) and *EDA* (page 176) models built using coordinate data in Å, unit of variance is Å^2. For *ANM* (page 140), *GNM* (page 164), and *RTB* (page 187), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Returns **True** if model is 3-dimensional.

numAtoms()
Returns number of atoms.

numDOF()
Returns number of degrees of freedom.

numEntries()
Returns number of entries in one eigenvector.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

setHessian(*hessian*)
Set Hessian matrix. A symmetric matrix is expected, i.e. not a lower- or upper-triangular matrix.

setTitle(*title*)
Set title of the model.

3.6.40 Sampling Functions

This module defines functions for generating alternate conformations along normal modes.

deformAtoms(*atoms*, *mode*, *rmsd=None*, *replace=False*, *scale=None*)
Generate a new coordinate set for *atoms* along the *mode*. *atoms* must be a *AtomGroup* (page 38) instance. New coordinate set will be appended to *atoms*. If *rmsd* is provided, *mode* will be scaled to generate a coordinate set with given RMSD distance to the active coordinate set.

sampleModes(*modes*, *atoms=None*, *n_confs=1000*, *rmsd=1.0*)
Returns an ensemble of randomly sampled conformations along given *modes*. If *atoms* are provided, sampling will be around its active coordinate set. Otherwise, sampling is around the 0 coordinate set.

Parameters

- **modes** (*Mode* (page 169), *ModeSet* (page 171), *PCA* (page 175), *ANM* (page 140) or *NMA* (page 172)) – modes along which sampling will be performed
- **atoms** (*Atomic* (page 47)) – atoms whose active coordinate set will be used as the initial conformation
- **n_confs** – number of conformations to generate, default is 1000
- **rmsd** (*float*[496]) – average RMSD that the conformations will have with respect to the initial conformation, default is 1.0 Å

Returns *Ensemble* (page 202)

For given normal modes $[u_1 u_2 ... u_m]$ and their eigenvalues $[\lambda_1 \lambda_2 ... \lambda_m]$, a new conformation is sampled using the relation:

$$R_k = R_0 + s \sum_{i=1}^{m} r_i^k \lambda_i^{-0.5} u_i \tag{3.1}$$

R_0 is the active coordinate set of *atoms*. $[r_1^k r_2^k ... r_m^k]$ are normally distributed random numbers generated for conformation k using `numpy.random.randn()`.

[496] http://docs.python.org/library/functions.html#float

RMSD of the new conformation from R_0 can be calculated as

$$RMSD^k = \sqrt{\left[s \sum_{i=1}^{m} r_i^k \lambda_i^{-0.5} u_i \right]^2 / N} = \frac{s}{\sqrt{N}} \sqrt{\sum_{i=1}^{m} (r_i^k)^2 \lambda_i^{-1}} \tag{3.2}$$

Average $RMSD$ of the generated conformations from the initial conformation is:

$$\left\langle RMSD^k \right\rangle = \frac{s}{\sqrt{N}} \left\langle \sqrt{\sum_{i=1}^{m} (r_i^k)^2 \lambda_i^{-1}} \right\rangle \tag{3.3}$$

From this relation s scaling factor obtained using the relation

$$s = \left\langle RMSD^k \right\rangle \sqrt{N} \left\langle \sqrt{\sum_{i=1}^{m} (r_i)^2 \lambda_i^{-1}} \right\rangle^{-1} \tag{3.4}$$

Note that random numbers are generated before conformations are sampled, hence exact value of s is known from this relation to ensure that the generated ensemble will have user given average *rmsd* value.

Note that if modes are from a `PCA` (page 175), variances are used instead of inverse eigenvalues, i.e. $\sigma_i \sim \lambda_i^{-1}$.

See also `showEllipsoid()` (page 182).

traverseMode(*mode*, *atoms*, *n_steps=10*, *rmsd=1.5*, ***kwargs*)

Generates a trajectory along a given *mode*, which can be used to animate fluctuations in an external program.

Parameters

- **mode** (`Mode` (page 169)) – mode along which a trajectory will be generated
- **atoms** (`Atomic` (page 47)) – atoms whose active coordinate set will be used as the initial conformation
- **n_steps** (`int`[497]) – number of steps to take along each direction, for example, for `n_steps=10`, 20 conformations will be generated along *mode* with structure *atoms* in between, default is 10.
- **rmsd** (`float`[498]) – maximum RMSD that the conformations will have with respect to the initial conformation, default is 1.5 Å
- **pos** (`bool`[499]) – whether to include steps in the positive mode direction, default is **True**
- **neg** (`bool`[500]) – whether to include steps in the negative mode direction, default is **True**
- **reverse** (`bool`[501]) – whether to reverse the direction default is **False**

Returns `Ensemble` (page 202)

[497] http://docs.python.org/library/functions.html#int
[498] http://docs.python.org/library/functions.html#float
[499] http://docs.python.org/library/functions.html#bool
[500] http://docs.python.org/library/functions.html#bool
[501] http://docs.python.org/library/functions.html#bool

For given normal mode u_i, its eigenvalue λ_i, number of steps n, and maximum $RMSD$ conformations $[R_{-n}R_{-n+1}...R_{-1}R_0R_1...R_n]$ are generated.

R_0 is the active coordinate set of *atoms*. $R_k = R_0 + sk\lambda_i u_i$, where s is found using $s = ((N(\frac{RMSD}{n})^2)/\lambda_i^{-1})^{0.5}$, where N is the number of atoms.

3.6.41 Signature Dynamics of Protein Families (SignDy)

This module defines functions for signature dynamics (SignDy), analyzing normal modes obtained for conformations in an ensemble.

class ModeEnsemble(*title=None*)

A collection of ENMs calculated for conformations in an `Ensemble`. or `PDBEnsemble`.

addModeSet(*modeset*, *weights=None*, *label=None*, *matched=False*, *reweighted=False*)
Adds a modeset or modesets to the mode ensemble.

delModeSet(*index*)
Removes a modeset or modesets from the mode ensemble.

getArray(*mode_index=0*)
Returns a sdarray of row arrays.

getAtoms()
Returns associated atoms of the mode ensemble.

getEigval(*mode_index=0*)
Returns eigenvalue of a given mode index with respect to the reference.

getEigvals(*mode_indices=None*)
Returns a sdarray of eigenvalues across modesets.

getEigvec(*mode_index=0*, *sign_correction=True*)
Returns a sdarray of eigenvector across modesets.

getEigvecs(*mode_indices=None*, *sign_correction=True*)
Returns a sdarray of eigenvectors across modesets.

getIndex(*mode_index=0*)
Returns indices of modes matched to the reference modeset.

getIndices(*mode_indices=None*)
Returns indices of modes in the mode ensemble.

getLabels()
Returns the labels of the mode ensemble.

getMatchingStatus()
Returns the matching status of each mode ensemble.

getModeSets(*index=None*)
Returns the modeset of the given index. If index is **None** then all modesets are returned.

getReweightingStatus()
Returns the reweighting status of each mode ensemble.

getTitle()
Returns title of the signature.

getVariance(*mode_index=0*)
Returns variances of a given mode index with respect to the reference.

getVariances(*mode_indices=None*)
Returns a sdarray of variances across modesets.

getWeights()
Returns a copy of weights.

is3d()
Returns **True** is model is 3-dimensional.

isMatched()
Returns whether the modes are matched across ALL modesets in the mode ensemble

isReweighted()
Returns whether the modes are matched across ALL modesets in the mode ensemble

match(*turbo=False*, *method=None*)
Matches the modes across mode sets according the mode overlaps.

Parameters **turbo** (*bool, int*) – if **True** then the computation will be performed in parallel. The number of threads is set to be the same as the number of CPUs. Assigning a number will specify the number of threads to be used. Default is **False**

numAtoms()
Returns number of atoms.

numModeSets()
Returns number of modesets in the instance.

numModes()
Returns number of modes in the instance (not necessarily maximum number of possible modes).

reorder()
Reorders the modes across mode sets according to their collectivity

reweight()
Reweight the modes based on matched orders

setAtoms(*atoms*)
Sets the atoms of the mode ensemble.

setLabels(*labels*)
Returns the labels of the mode ensemble.

setMatchingStatus(*status*)
Returns the matching status of each mode ensemble.

setReweightingStatus(*status*)
Returns the reweighting status of each mode ensemble.

setWeights(*weights*)
Set atomic weights.

undoMatching()
Restores the original orders of modes

undoReweighting()
Restores the original weighting of modes

class sdarray
A class for representing a collection of arrays. It is derived from `ndarray`, and the first axis is reserved for indexing the collection.

sdarray (page 192) functions exactly the same as `ndarray`, except that *sdarray.mean()* (page 193), *sdarray.std()* (page 193), *sdarray.max()* (page 193), *sdarray.min()* (page 193) are overriden. Average, standard deviation, minimum, maximum, etc. are weighted and calculated over the first axis by default. "sdarray" stands for "signature dynamics array".

Note for developers: please read the following article about subclassing `ndarray` before modifying this class:

https://docs.scipy.org/doc/numpy-1.14.0/user/basics.subclassing.html

getArray()
Returns the signature as an numpy array.

getLabels()
Returns the labels of the signature.

getTitle()
Returns the title of the signature.

getWeights()
Returns the weights of the signature.

is3d()
Returns **True** is model is 3-dimensional.

max(*axis=0*, ***kwargs*)
Calculates the maximum values of the sdarray over modesets (*axis=0*).

mean(*axis=0*, ***kwargs*)
Calculates the weighted average of the sdarray over modesets (*axis=0*).

min(*axis=0*, ***kwargs*)
Calculates the minimum values of the sdarray over modesets (*axis=0*).

numAtoms()
Returns the number of atoms assuming it is represented by the second axis.

numModeSets()
Returns the number of modesets in the instance

setWeights(*weights*)
Sets the weights of the signature.

std(*axis=0*, ***kwargs*)
Calculates the weighted standard deviations of the sdarray over modesets (*axis=0*).

weights
Returns the weights of the signature.

calcEnsembleENMs(*ensemble*, *model='gnm'*, *trim='reduce'*, *n_modes=20*, ***kwargs*)
Calculates normal modes for each member of *ensemble*.

Parameters

- **ensemble** (*PDBEnsemble* (page 207)) – normal modes of whose members to be computed
- **model** (*str*[502]) – type of ENM that will be performed. It can be either 'anm' or 'gnm'
- **trim** (*int*[503]) – type of method that will be used to trim the model. It can be either 'trim' , 'slice', or 'reduce'. If set to 'trim', the parts that is not in the selection will simply be removed
- **n_modes** – number of modes to be computed

[502] http://docs.python.org/library/stdtypes.html#str
[503] http://docs.python.org/library/functions.html#int

- **turbo** (*bool*[504]) – if **True** then the computation will be performed in parallel. The number of threads is set to be the same as the number of CPUs. Assigning a number to specify the number of threads to be used. Default is **False**
- **match** (*bool*[505]) – whether the modes should be matched using *matchModes()* (page 143). Default is **True**
- **method** (*function*) – the alternative function that is used to match the modes. Default is **None**
- **turbo** – whether use `Pool` to accelerate the computation. Note that if writing a script, `if __name__ == '__main__'` is necessary to protect your code when multi-tasking. See https://docs.python.org/2/library/multiprocessing.html for details. Default is **False**

Returns *ModeEnsemble* (page 191)

showSignature1D(*signature*, *linespec='-'*, ***kwargs*)
Show *signature* using `showAtomicLines()`.

Parameters

- **signature** (*sdarray* (page 192)) – the signature dynamics to be plotted
- **linespec** (*str*[506]) – line specifications that will be passed to `showAtomicLines()`
- **atoms** (*Atomic* (page 47)) – an object with method `getResnums()` for use on the x-axis.
- **alpha** (*float*[507]) – the transparency of the band(s).
- **range** (*bool*[508]) – whether shows the minimum and maximum values. Default is **True**

psplot(*signature*, *linespec='-'*, ***kwargs*)
Show *signature* using `showAtomicLines()`.

Parameters

- **signature** (*sdarray* (page 192)) – the signature dynamics to be plotted
- **linespec** (*str*[509]) – line specifications that will be passed to `showAtomicLines()`
- **atoms** (*Atomic* (page 47)) – an object with method `getResnums()` for use on the x-axis.
- **alpha** (*float*[510]) – the transparency of the band(s).
- **range** (*bool*[511]) – whether shows the minimum and maximum values. Default is **True**

showSignatureAtomicLines(*y*, *std=None*, *min=None*, *max=None*, *atoms=None*, ***kwargs*)
Show the signature dynamics data using `showAtomicLines()`.

Parameters

[504] http://docs.python.org/library/functions.html#bool
[505] http://docs.python.org/library/functions.html#bool
[506] http://docs.python.org/library/stdtypes.html#str
[507] http://docs.python.org/library/functions.html#float
[508] http://docs.python.org/library/functions.html#bool
[509] http://docs.python.org/library/stdtypes.html#str
[510] http://docs.python.org/library/functions.html#float
[511] http://docs.python.org/library/functions.html#bool

- **y** (ndarray) – the mean values of signature dynamics to be plotted
- **std** (ndarray) – the standard deviations of signature dynamics to be plotted
- **min** (ndarray) – the minimum values of signature dynamics to be plotted
- **max** (ndarray) – the maximum values of signature dynamics to be plotted
- **linespec** (*str*[512]) – line specifications that will be passed to showAtomicLines()
- **atoms** (*Atomic* (page 47)) – an object with method getResnums() for use on the x-axis.

showSignatureMode(*mode_ensemble*, ***kwargs*)
Show signature mode profile.

Parameters

- **mode_ensemble** (*ModeEnsemble* (page 191)) – mode ensemble from which to extract an eigenvector If this is not indexed already then index 0 is used by default
- **atoms** (*Atomic* (page 47)) – atoms for showing residues along the x-axis Default option is to use mode_ensemble.getAtoms()
- **scale** (*float*[513]) – scaling factor. Default is 1.0

showSignatureDistribution(*signature*, ***kwargs*)
Show the distribution of signature values using hist()[514].

showSignatureCollectivity(*mode_ensemble*, ***kwargs*)
Show the distribution of signature variances using *showSignatureDistribution()* (page 195).

showSignatureSqFlucts(*mode_ensemble*, ***kwargs*)
Show signature profile of square fluctations.

Parameters

- **mode_ensemble** (*ModeEnsemble* (page 191)) – mode ensemble from which to calculate square fluctutations
- **atoms** (*Atomic* (page 47)) – atoms for showing residues along the x-axis Default option is to use mode_ensemble.getAtoms()
- **scale** (*float*[515]) – scaling factor. Default is 1.0
- **show_zero** (*bool*[516]) – where to show a grey line at y=0 Default is False

calcEnsembleSpectralOverlaps(*ensemble*, *distance=False*, *turbo=False*, ***kwargs*)
Calculate the spectral overlaps between each pair of conformations in the *ensemble*.

Parameters

- **ensemble** – an ensemble of structures or ENMs
- **distance** (*bool*[517]) – if set to **True**, spectral overlap will be converted to spectral distance via arccos.

[512] http://docs.python.org/library/stdtypes.html#str
[513] http://docs.python.org/library/functions.html#float
[514] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.hist.html#matplotlib.pyplot.hist
[515] http://docs.python.org/library/functions.html#float
[516] http://docs.python.org/library/functions.html#bool
[517] http://docs.python.org/library/functions.html#bool

- **turbo** (*bool*[518]) – if **True**, extra memory will be used to remember previous calculation results to accelerate the next calculation, so this option is particularly useful if spectral overlaps of the same ensemble are calculated repeatedly, e.g. using different number of modes. Note that for single calculation, *turbo* will compromise the speed. Default is **False**

calcSignatureSqFlucts(*mode_ensemble*, ***kwargs*)
Get the signature square fluctuations of *mode_ensemble*.

Parameters

- **mode_ensemble** – an ensemble of ENMs
- **norm** (*bool*[519]) – whether to normalize the square fluctuations. Default is **True**
- **scale** (*bool*[520]) – whether to rescale the square fluctuations based on the reference. Default is **False**

calcSignatureCollectivity(*mode_ensemble*, *masses=None*)
Calculate average collectivities for a ModeEnsemble.

calcSignatureFractVariance(*mode_ensemble*)
Calculate signature fractional variance for a ModeEnsemble.

calcSignatureModes(*mode_ensemble*)
Calculate mean eigenvalues and eigenvectors and return a new GNM or ANM object containing them.

calcSignatureCrossCorr(*mode_ensemble*, *norm=True*)
Calculate the signature cross-correlations based on a *ModeEnsemble* (page 191) instance.

Parameters

- **mode_ensemble** – an ensemble of ENMs
- **norm** (*bool*[521]) – whether to normalize the cross-correlations. Default is **True**

showSignatureCrossCorr(*mode_ensemble*, *std=False*, ***kwargs*)
Show average cross-correlations using `showAtomicMatrix()`. By default, *origin=lower* and *interpolation=bilinear* keyword arguments are passed to this function, but user can overwrite these parameters. See also *calcSignatureCrossCorr()* (page 196).

Parameters

- **ensemble** – an ensemble of structures or ENMs, or a signature profile
- **atoms** (*Atomic* (page 47)) – an object with method `getResnums()` for use on the x-axis.

showVarianceBar(*mode_ensemble*, *highlights=None*, ***kwargs*)
Show the distribution of variances (cumulative if multiple modes) using `histogram()`.

Parameters

- **mode_ensemble** (*ModeEnsemble* (page 191)) – an ensemble of modes whose variances are displayed
- **highlights** (*list*[522]) – labels of conformations whose locations on the bar will be highlighted by arrows and texts

[518] http://docs.python.org/library/functions.html#bool
[519] http://docs.python.org/library/functions.html#bool
[520] http://docs.python.org/library/functions.html#bool
[521] http://docs.python.org/library/functions.html#bool
[522] http://docs.python.org/library/stdtypes.html#list

- **fraction** (*bool*[523]) – whether the variances should be weighted or not. Default is **True**

showSignatureVariances(*mode_ensemble*, ***kwargs*)
Show the distribution of signature variances using *showSignatureDistribution()* (page 195).

calcSignatureOverlaps(*mode_ensemble*, *diag=True*, *collapse=False*)
Calculate average mode-mode overlaps for a ModeEnsemble.

If *diag* is **True** (default) then only diagonal values will be calculated. Otherwise, the whole overlap matrices will be calculated.

By default (*collapse* is **False**), the whole overlap matrices are returned as a 4-dimensional sdarray that is a matrix of overlap matrices.

If *collapse* is **True** then these will be collapsed together, giving a 2-dimensional array for full matrices. This operation is not defined for diagonal values.

showSignatureOverlaps(*mode_ensemble*, ***kwargs*)
Show a curve of mode-mode overlaps against mode number with shades for standard deviation and range

Parameters

- **diag** (*bool*[524]) – Whether to calculate the diagonal values only. Default is **False** and `showMatrix()` is used. If set to **True**, *showSignatureAtomicLines()* (page 194) is used.
- **std** – Whether to show the standard deviation matrix when **diag** is **False** (and whole matrix is shown). Default is **False**, meaning the mean matrix is shown.

type: std: bool

saveModeEnsemble(*mode_ensemble*, *filename=None*, *atoms=False*, ***kwargs*)
Save *mode_ensemble* as `filename.modeens.npz`. If *filename* is **None**, title of the *mode_ensemble* will be used as the filename, after `" "` (white spaces) in the title are replaced with `"_"` (underscores). Upon successful completion of saving, filename is returned. This function makes use of `savez_compressed()` function.

loadModeEnsemble(*filename*, ***kwargs*)
Returns ModeEnsemble instance after loading it from file (*filename*). This function makes use of `numpy.load()` function. See also *saveModeEnsemble()* (page 197).

saveSignature(*signature*, *filename=None*, ***kwargs*)
Save *signature* as `filename.sdarray.npz`. If *filename* is **None**, title of the *signature* will be used as the filename, after `" "` (white spaces) in the title are replaced with `"_"` (underscores). Upon successful completion of saving, filename is returned. This function makes use of `savez_compressed()` function.

loadSignature(*filename*, ***kwargs*)
Returns *sdarray* (page 192) instance after loading it from file (*filename*). This function makes use of `numpy.load()` function. See also *saveSignature()* (page 197).

calcSubfamilySpectralOverlaps(*mode_ens*, *subfamily_dict*, ***kwargs*)
Calculate average spectral overlaps (or distances) within and between subfamilies in a mode ensemble defined using a dictionary where each key is an ensemble member and the associate value is a subfamily name.

[523] http://docs.python.org/library/functions.html#bool
[524] http://docs.python.org/library/functions.html#bool

To use a range of modes, please index the mode ensemble e.g. mode_ens=mode_ensemble[:,3:20] to use modes 4 to 20 inclusive. Alternatively, there is the option to provide first and last keyword arguments, which would be used as the 3 and 20 above.

Parameters

- **mode_ensemble** (*ModeEnsemble* (page 191)) – an ensemble of modes corresponding to a set of modes for each family member
- **subfamily_dict** (*dict*[525]) – a dictionary providing a subfamily label for each family member
- **first** (*int*[526]) – the first index for a range of modes
- **last** (*int*[527]) – the last index for a range of modes
- **remove_small** (*bool*[528]) – whether to remove small subfamilies with fewer than 4 members. Default is True
- **return_reordered_subfamilies** – whether to return the reordered subfamilies in addition to the matrix. Default is False

type return_reordered_subfamilies: bool

showSubfamilySpectralOverlaps(*mode_ens*, *subfamily_dict*, ***kwargs*)

Calculate and show the matrix of spectral overlaps or distances averaged over subfamilies. Inputs are the same as calcSubfamilySpectralOverlaps plus the following and those of showDomainBar if you wish.

Parameters **show_subfamily_bar** (*bool*[529]) – whether to show the subfamilies as colored bars using showDomainBar. Default is False

calcSignaturePerturbResponse(*mode_ensemble*, ***kwargs*)

Calculate the signature perturbation response scanning based on a *ModeEnsemble* (page 191) instance.

Parameters **mode_ensemble** – an ensemble of ENMs

3.6.42 Writing TCL scripts for VMD

This module defines TCL file for VMD program. Questions/Problems: karolami@pitt.edu

writeVMDstiffness(*stiffness*, *pdb*, *indices*, *k_range*, *filename='vmd_out'*, *select='protein and name CA'*, *loadToVMD=False*)

Returns three files starting with the provided filename and having their own extensions:

1.A PDB file that can be used in the TCL script.

(2) TCL file containing vmd commands for loading PDB file with accurate vmd representation. Pair of residues with selected *k_range* of effective spring constant are shown in VMD respresentation with solid line between them. If more than one residue is selected in *indices*, different pair for each residue will be colored in the different colors.

(3) TXT file containing pair of residues with effective spring constant in selected range *k_range*.

Note:

[525] http://docs.python.org/library/stdtypes.html#dict
[526] http://docs.python.org/library/functions.html#int
[527] http://docs.python.org/library/functions.html#int
[528] http://docs.python.org/library/functions.html#bool
[529] http://docs.python.org/library/functions.html#bool

1.This function skips modes with zero eigenvalues.

2.If a *Vector* (page 170) instance is given, it will be normalized before it is written. It's length before normalization will be written as the scaling factor of the vector.

Parameters

- **stiffness** (ndarray) – mechanical stiffness profile calculated with *calcMechStiff()* (page 169)
- **pdb** (ndarray, *Atomic* (page 47)) – a coordinate set or an object with getCoords method
- **indices** (*list*[530]) – an amino acid number or a pair of amino acid numbers
- **k_range** (*int, float, list*) – effective force constant value or range of values
- **select** (*Select* (page 99), str) – a selection or selection string default is 'protein and name CA'
- **loadToVMD** (*bool*[531]) – whether to load VMD and run the tcl file default is False

writeDeformProfile(*stiffness, pdb, filename='dp_out', select='protein and name CA', pdb_selstr='protein', loadToVMD=False*)

Calculate deformability (plasticity) profile of molecule based on mechanical stiffness matrix (see *[EB08]* (page 395)).

Parameters

- **stiffness** (:class:'~numpy.ndarray) – mechanical stiffness matrix
- **pdb** (ndarray) – a coordinate set or an object with getCoords method

Note: selection can be done using select and pdb_selstr. select defines model selection (used for building *ANM* (page 140) model) and pdb_selstr will be used in VMD program for visualization.

By default files are saved as *filename* and loaded to VMD program. To change it use loadToVMD=False.

Mean value of mechanical stiffness for molecule can be found in occupancy column in PDB file.

calcChainsNormDistFluct(*coords, ch1, ch2, cutoff=10.0, percent=5, rangeAng=5, filename='ch_ndf_out', loadToVMD=False*)

Calculate protein-protein interaction using getNormDistFluct() from *GNM* (page 164) model. It is assigned to protein complex.

Parameters

- **coords** (ndarray.) – a coordinate set or an object with getCoords method.
- **ch1** (*'A' or other letter as a string*) – first chain name
- **ch2** (*string*[532]) – second chain name
- **cutoff** (*float*[533]) – cutoff distance (Å) for pairwise interactions in Kirchhoff matrix, default is 10.0 Å
- **percent** (*int or float*) – percent of the highest and lowest results displayed in _VMD program, default is 5%

[530]http://docs.python.org/library/stdtypes.html#list
[531]http://docs.python.org/library/functions.html#bool
[532]http://docs.python.org/library/string.html#module-string
[533]http://docs.python.org/library/functions.html#float

- **rangeAng** (*int or float*) – cutoff range of protein-protein interactions, default is 5 Å
- **filename** (*str*[534]) – name of tcl file from _VMD program

By default files are saved as *filename* and loaded to VMD program. To change it use `loadToVMD=False`.

UNDER PREPARATION.. problems with not complete structures

3.7 Ensemble Analysis

This module defines classes for handling conformational ensembles.

3.7.1 Conformational ensembles

The following two classes are implemented for handling arbitrary but uniform conformational ensembles, e.g. NMR models, MD snapshots:

- *Ensemble* (page 202)
- *Conformation* (page 201)

See usage examples in NMR Models[535] and Essential Dynamics Analysis[536].

3.7.2 PDB ensembles

PDB ensembles, such as multiple structures of the same protein, are in general heterogeneous. This just means that different residues in different structures are missing. The following classes extend above to support this heterogeneity:

- *PDBEnsemble* (page 207)
- *PDBConformation* (page 201)

The following functions are for creating or editing PDB ensembles, e.g. finding and removing residues that are missing in too many structures:

- *buildPDBEnsemble()* (page 206)
- *alignByEnsemble()* (page 207)
- *calcOccupancies()* (page 205)
- *showOccupancies()* (page 206)
- *trimPDBEnsemble()* (page 205)

See usage examples in Heterogeneous X-ray Structures[537], Multimeric Structures[538], Homologous Proteins[539].

3.7.3 Save/load ensembles

- *saveEnsemble()* (page 205)
- *loadEnsemble()* (page 205)

[534] http://docs.python.org/library/stdtypes.html#str
[535] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/nmr.html#pca-nmr
[536] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda
[537] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray.html#pca-xray
[538] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/dimer.html#pca-dimer
[539] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/blast.html#pca-blast

3.7.4 Conformation

This module defines classes handling individual conformations.

class **Conformation**(*ensemble*, *index*)

A class to provide methods on a conformation in an ensemble. Instances of this class do not keep coordinate and weights data.

getAtoms()

Returns associated atom group.

getCoords()

Returns a copy of the coordinates of the conformation. If a subset of atoms are selected in the ensemble, coordinates for selected atoms will be returned.

getData(*label*)

Returns a copy of the data array associated with *label*, or **None** if such data is not present.

getDeviations()

Returns deviations from the ensemble reference coordinates. Deviations are calculated for (selected) atoms.

getEnsemble()

Returns the ensemble that this conformation belongs to.

getIndex()

Returns conformation index.

getRMSD()

Returns RMSD from the ensemble reference coordinates. RMSD is calculated for (selected) atoms.

getWeights()

Returns coordinate weights for (selected) atoms.

numAtoms()

Returns number of atoms.

numSelected()

Returns number of selected atoms.

class **PDBConformation**(*ensemble*, *index*)

This class is the same as *Conformation* (page 201), except that the conformation has a name (or identifier), e.g. PDB identifier.

getAtoms()

Returns associated atom group.

getCoords()

Returns a copy of the coordinates of the conformation. If a subset of atoms are selected in the ensemble, coordinates for selected atoms will be returned.

> **Warning:** When there are atoms with weights equal to zero (0), their coordinates will be replaced with the coordinates of the ensemble reference coordinate set.

getDeviations()

Returns deviations from the ensemble reference coordinates. Deviations are calculated for (selected) atoms.

getEnsemble()

Returns the ensemble that this conformation belongs to.

getIndex()
Returns conformation index.

getLabel()
Returns the label of the conformation.

getRMSD()
Returns RMSD from the ensemble reference coordinates. RMSD is calculated for (selected) atoms.

getSequence()
Returns the sequence of the conformation.

getTransformation()
Returns the *Transformation* (page 218) used to superpose this conformation onto reference coordinates. The transformation can be used to superpose original PDB file onto the reference PDB file.

getWeights()
Returns coordinate weights for (selected) atoms.

numAtoms()
Returns number of atoms.

numSelected()
Returns number of selected atoms.

setLabel(*label*)
Set the label of the conformation.

3.7.5 Conformational Ensemble

This module defines a class for handling ensembles of conformations.

class Ensemble(*title='Unknown'*)
A class for analysis of arbitrary conformational ensembles.

Indexing (e.g. `ens[0]`) returns a *Conformation* (page 201) instance that points to a coordinate set in the ensemble. Slicing (e.g. `ens[0:10]`) returns an *Ensemble* (page 202) instance that contains a copy of the subset of conformations (coordinate sets).

Instantiate with a *title* or a *Atomic* (page 47) instance. All coordinate sets from atomic instances will be added to the ensemble.

addCoordset(*coords*, ***kwargs*)
Add coordinate set(s) to the ensemble.

Parameters

- **coords** – must be a `ndarray` with suitable data type, shape and dimensionality, or an object with *getCoordsets()* (page 203) method.
- **data** (*dict*[540]) – associated data to be added along with *coords*.

delCoordset(*index*)
Delete a coordinate set from the ensemble.

delData(*label*)
Return data associated with *label* and remove from the instance. If data associated with *label* is not found, return **None**.

[540] http://docs.python.org/library/stdtypes.html#dict

deselect()
Undoes the selection.

getAtoms(*selected=True*)
Returns associated/selected atoms.

getConformation(*index*)
Returns conformation at given index.

getCoords(*selected=True*)
Returns a copy of reference coordinates for selected atoms.

getCoordsets(*indices=None*, *selected=True*)
Returns a copy of coordinate set(s) at given *indices*, which may be an integer, a list of integers or **None**. **None** returns all coordinate sets. For reference coordinates, use *getCoords()* (page 203) method.

getData(*label*)
Returns a copy of the data array associated with *label*, or **None** if such data is not present.

getDataLabels(*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType(*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getDefvecs(*pairwise=False*)
Calculate and return deformation vectors (defvecs). Note that you might need to align the conformations using *superpose()* (page 205) or *iterpose()* (page 204) before calculating defvecs.

Parameters **pairwise** (*bool*[541]) – if **True** then it will return pairwise defvecs as an n-by-n matrix. n is the number of conformations.

getDeviations()
Returns deviations from reference coordinates for selected atoms. Conformations can be aligned using one of *superpose()* (page 205) or *iterpose()* (page 204) methods prior to calculating deviations.

getIndices()
Returns a copy of indices of selected columns

getMSFs()
Returns mean square fluctuations (MSFs) for selected atoms. Conformations can be aligned using one of *superpose()* (page 205) or *iterpose()* (page 204) methods prior to MSF calculation.

getRMSDs(*pairwise=False*)
Returns root mean square deviations (RMSDs) for selected atoms. Conformations can be aligned using one of *superpose()* (page 205) or *iterpose()* (page 204) methods prior to RMSD calculation.

Parameters **pairwise** (*bool*[542]) – if **True** then it will return pairwise RMSDs as an n-by-n matrix. n is the number of conformations.

getRMSFs()
Returns root mean square fluctuations (RMSFs) for selected atoms. Conformations can be aligned using one of *superpose()* (page 205) or *iterpose()* (page 204) methods prior to RMSF calculation.

[541] http://docs.python.org/library/functions.html#bool
[542] http://docs.python.org/library/functions.html#bool

getTitle()
Returns title of the ensemble.

getWeights(*selected=True*)
Returns a copy of weights of selected atoms.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isSelected()
Returns if a subset of atoms are selected.

iterCoordsets()
Iterate over coordinate sets. A copy of each coordinate set for selected atoms is returned. Reference coordinates are not included.

iterpose(*rmsd=0.0001*, *quiet=False*)
Iteratively superpose the ensemble until convergence. Initially, all conformations are aligned with the reference coordinates. Then mean coordinates are calculated, and are set as the new reference coordinates. This is repeated until reference coordinates do not change. This is determined by the value of RMSD between the new and old reference coordinates. Note that at the end of the iterative procedure the reference coordinate set will be average of conformations in the ensemble.

Parameters **rmsd** (*float*[543]) – change in reference coordinates to determine convergence, default is 0.0001 Å RMSD

numAtoms(*selected=True*)
Returns number of atoms.

numConfs()
Returns number of conformations.

numCoordsets()
Returns number of conformations.

numSelected()
Returns number of selected atoms. Number of all atoms will be returned if a selection is not made. A subset of atoms can be selected by passing a selection to *setAtoms()* (page 204).

select(*selection*)
Selects columns corresponding to a part of the atoms.

setAtoms(*atoms*)
Set *atoms* or specify a selection of atoms to be considered in calculations and coordinate requests. When a selection is set, corresponding subset of coordinates will be considered in, for example, alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, *Ensemble* (page 202) and *Conformation* (page 201) instances become suitable arguments for *writePDB()* (page 270). Passing **None** as *atoms* argument will deselect atoms.

setCoords(*coords*)
Set *coords* as the ensemble reference coordinate set. *coords* may be an array with suitable data type, shape, and dimensionality, or an object with *getCoords()* (page 203) method.

setData(*label*, *data*)
Store atomic *data* under *label*, which must:

- start with a letter

[543] http://docs.python.org/library/functions.html#float

- contain only alphanumeric characters and underscore
- not be a reserved word (see *listReservedWords()* (page 75))

data must be a `list()` or a `ndarray` and its length must be equal to the number of atoms. If the dimension of the *data* array is 1, i.e. `data.ndim==1`, *label* may be used to make atom selections, e.g. `"label 1 to 10"` or `"label C1 C2"`. Note that, if data with *label* is present, it will be overwritten.

setTitle(*title*)
Set title of the ensemble.

setWeights(*weights*)
Set atomic weights.

superpose(***kwargs*)
Superpose the ensemble onto the reference coordinates.

Parameters **ref** (*int*[544]) – index of the reference coordinate. If **None**, the average coordinate will be assumed as the reference. Default is **None**

3.7.6 Supporting Functions

This module defines a functions for handling conformational ensembles.

saveEnsemble(*ensemble*, *filename=None*, ***kwargs*)
Save *ensemble* model data as `filename.ens.npz`. If *filename* is **None**, title of the *ensemble* will be used as the filename, after white spaces in the title are replaced with underscores. Extension is `.ens.npz`. Upon successful completion of saving, filename is returned. This function makes use of `savez()` function.

loadEnsemble(*filename*, ***kwargs*)
Returns ensemble instance loaded from *filename*. This function makes use of `load()` function. See also *saveEnsemble()* (page 205)

trimPDBEnsemble(*pdb_ensemble*, *occupancy=None*, ***kwargs*)
Returns a new PDB ensemble obtained by trimming given *pdb_ensemble*. This function helps selecting atoms in a pdb ensemble based on one of the following criteria, and returns them in a new *PDBEnsemble* (page 207) instance.

Resulting PDB ensemble will contain atoms whose occupancies are greater or equal to *occupancy* keyword argument. Occupancies for atoms will be calculated using `calcOccupancies(pdb_ensemble, normed=True)`.

Parameters

- **occupancy** (*float*[545]) – occupancy for selecting atoms, must satisfy `0 < occupancy <= 1`. If set to *None* then *hard* trimming will be performed.
- **hard** (*bool*[546]) – Whether to perform hard trimming. Default is **False** If set to **True**, atoms will be completely removed from *pdb_ensemble*. If set to **False**, a soft trimming of *pdb_ensemble* will be done where atoms will be removed from the active selection. This is useful, for example, when one uses `calcEnsembleENMs()` together with `sliceModel()` or `reduceModel()` to calculate the modes from the remaining part while still taking the removed part into consideration (e.g. as the environment).

[544] http://docs.python.org/library/functions.html#int
[545] http://docs.python.org/library/functions.html#float
[546] http://docs.python.org/library/functions.html#bool

calcOccupancies(*pdb_ensemble*, *normed=False*)

Returns occupancy calculated from weights of a *PDBEnsemble* (page 207). Any non-zero weight will be considered equal to one. Occupancies are calculated by binary weights for each atom over the conformations in the ensemble. When *normed* is **True**, total weights will be divided by the number of atoms. This function can be used to see how many times a residue is resolved when analyzing an ensemble of X-ray structures.

showOccupancies(*pdbensemble*, **args*, ***kwargs*)

Show occupancies for the PDB ensemble using `plot()`. Occupancies are calculated using *calcOccupancies()* (page 205).

buildPDBEnsemble(*atomics*, *ref=None*, *title='Unknown'*, *labels=None*, *atommaps=None*, *unmapped=None*, ***kwargs*)

Builds a *PDBEnsemble* (page 207) from a given reference structure and a list of structures (*Atomic* (page 47) instances). Note that the reference should be included in the list as well.

Parameters

- **atomics** (*list*[547]) – a list of *Atomic* (page 47) instances
- **ref** (int, *Chain* (page 53), *Selection* (page 100), or *AtomGroup* (page 38)) – reference structure or the index to the reference in *atomics*. If **None**, then the first item in *atomics* will be considered as the reference. If it is a *PDBEnsemble* (page 207) instance, then *atomics* will be appended to the existing ensemble. Default is **None**
- **title** (*str*[548]) – the title of the ensemble
- **labels** (*list*[549]) – labels of the conformations
- **degeneracy** (*bool*[550]) – whether only the active coordinate set (**True**) or all the coordinate sets (**False**) of each structure should be added to the ensemble. Default is **True**
- **occupancy** (*float*[551]) – minimal occupancy of columns (range from 0 to 1). Columns whose occupancy is below this value will be trimmed
- **atommaps** (*list*[552]) – atom maps for *atomics* that were mapped and added into the ensemble. This is an output argument
- **unmapped** (*list*[553]) – labels of *atomics* that cannot be included in the ensemble. This is an output argument
- **subset** (*str*[554]) – a subset for selecting particular atoms from the input structures. Default is `"all"`
- **superpose** (*str, bool*) – if set to `'iter'`,`PDBEnsemble.iterpose()` will be used to superpose the structures, otherwise conformations will be superposed with respect to the reference specified by *ref* unless set to `False`. Default is `'iter'`

refineEnsemble(*ensemble*, *lower=0.5*, *upper=10.0*, ***kwargs*)

Refine a *PDBEnsemble* (page 207) based on RMSD criterions.

Parameters

[547] http://docs.python.org/library/stdtypes.html#list
[548] http://docs.python.org/library/stdtypes.html#str
[549] http://docs.python.org/library/stdtypes.html#list
[550] http://docs.python.org/library/functions.html#bool
[551] http://docs.python.org/library/functions.html#float
[552] http://docs.python.org/library/stdtypes.html#list
[553] http://docs.python.org/library/stdtypes.html#list
[554] http://docs.python.org/library/stdtypes.html#str

- **ensemble** (*Ensemble* (page 202), *PDBEnsemble* (page 207)) – the ensemble to be refined
- **lower** (*float*[555]) – the smallest allowed RMSD between two conformations with the exception of **protected**
- **upper** (*float*[556]) – the highest allowed RMSD between two conformations with the exception of **protected**
- **protected** (*list*[557]) – a list of either the indices or labels of the conformations needed to be kept in the refined ensemble
- **ref** (*int or str*) – the index or label of the reference conformation which will also be kept. Default is 0
- **data_type** (*str*[558]) – type of data to use for refinement. This can be either "rmsd" or "seqid" Default is "rmsd"

combineEnsembles(*target*, *mobile*, ***kwargs*)
Combines two ensembles by mapping the **atoms** of **mobile** to that of **target**.

alignByEnsemble(*atomics*, *ensemble*)
Align a set of *Atomic* (page 47) objects using transformations from *ensemble*, which may be a *PDBEnsemble* (page 207) or a *PDBConformation* (page 201) instance.

Transformations will be applied based on indices so *atomics* and *ensemble* must have the same number of members.

Parameters

- **atomics** (tuple, list, ndarray) – a set of *Atomic* (page 47) objects to be aligned
- **ensemble** (*PDBEnsemble* (page 207), *PDBConformation* (page 201)) – a *PDBEnsemble* (page 207) or a *PDBConformation* (page 201) from which transformations can be extracted

3.7.7 PDB Structure Ensemble

This module defines a class for handling ensembles of PDB conformations.

class PDBEnsemble(*title='Unknown'*)
This class enables handling coordinates for heterogeneous structural datasets and stores identifiers for individual conformations.

See usage usage in Heterogeneous X-ray Structures[559], Multimeric Structures[560], and Homologous Proteins[561].

Note: This class is designed to handle conformations with missing coordinates, e.g. atoms that are note resolved in an X-ray structure. For unresolved atoms, the coordinates of the reference structure is assumed in RMSD calculations and superpositions.

---

addCoordset(*coords*, *weights=None*, *label=None*, ***kwargs*)
Add coordinate set(s) to the ensemble. *coords* must be a Numpy array with suitable shape and

[555] http://docs.python.org/library/functions.html#float
[556] http://docs.python.org/library/functions.html#float
[557] http://docs.python.org/library/stdtypes.html#list
[558] http://docs.python.org/library/stdtypes.html#str
[559] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray.html#pca-xray
[560] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/dimer.html#pca-dimer
[561] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/blast.html#pca-blast

dimensionality, or an object with *getCoordsets()* (page 208). *weights* is an optional argument. If provided, its length must match number of atoms. Weights of missing (not resolved) atoms must be `0` and weights of those that are resolved can be anything greater than `0`. If not provided, weights of all atoms for this coordinate set will be set equal to `1`. *label*, which may be a PDB identifier or a list of identifiers, is used to label conformations.

delCoordset (*index*)
Delete a coordinate set from the ensemble.

delData (*label*)
Return data associated with *label* and remove from the instance. If data associated with *label* is not found, return **None**.

deselect ()
Undoes the selection.

getAtoms (*selected=True*)
Returns associated/selected atoms.

getConformation (*index*)
Returns conformation at given index.

getCoords (*selected=True*)
Returns a copy of reference coordinates for selected atoms.

getCoordsets (*indices=None*, *selected=True*)
Returns a copy of coordinate set(s) at given *indices* for selected atoms. *indices* may be an integer, a list of integers or **None**. **None** returns all coordinate sets.

> **Warning:** When there are atoms with weights equal to zero (0), their coordinates will be replaced with the coordinates of the ensemble reference coordinate set.

getData (*label*)
Returns a copy of the data array associated with *label*, or **None** if such data is not present.

getDataLabels (*which=None*)
Returns data labels. For `which='user'`, return only labels of user provided data.

getDataType (*label*)
Returns type of the data (i.e. `data.dtype`) associated with *label*, or **None** label is not used.

getDefvecs (*pairwise=False*)
Calculate and return deformation vectors (defvecs). Note that you might need to align the conformations using *superpose()* (page 210) or *iterpose()* (page 209) before calculating defvecs.

Parameters **pairwise** (*bool*[562]) – if **True** then it will return pairwise defvecs as an n-by-n matrix. n is the number of conformations.

getDeviations ()
Returns deviations from reference coordinates for selected atoms. Conformations can be aligned using one of *superpose()* (page 210) or *iterpose()* (page 209) methods prior to calculating deviations.

getIndices ()
Returns a copy of indices of selected columns

getLabels ()
Returns identifiers of the conformations in the ensemble.

[562] http://docs.python.org/library/functions.html#bool

getMSA(*indices=None*, *selected=True*)
Returns an MSA of selected atoms.

getMSFs()
Calculate and return mean square fluctuations (MSFs). Note that you might need to align the conformations using *superpose()* (page 210) or *iterpose()* (page 209) before calculating MSFs.

getRMSDs(*pairwise=False*)
Calculate and return root mean square deviations (RMSDs). Note that you might need to align the conformations using *superpose()* (page 210) or *iterpose()* (page 209) before calculating RMSDs.

Parameters **pairwise** (*bool*[563]) – if **True** then it will return pairwise RMSDs as an n-by-n matrix. n is the number of conformations.

getRMSFs()
Returns root mean square fluctuations (RMSFs) for selected atoms. Conformations can be aligned using one of *superpose()* (page 210) or *iterpose()* (page 209) methods prior to RMSF calculation.

getTitle()
Returns title of the ensemble.

getTransformations()
Returns the *Transformation* (page 218) used to superpose this conformation onto reference coordinates. The transformation can be used to superpose original PDB file onto the reference PDB file.

getWeights(*selected=True*)
Returns a copy of weights of selected atoms.

isDataLabel(*label*)
Returns **True** if data associated with *label* is present.

isSelected()
Returns if a subset of atoms are selected.

iterCoordsets()
Iterate over coordinate sets. A copy of each coordinate set for selected atoms is returned. Reference coordinates are not included.

iterpose(*rmsd=0.0001*)
Iteratively superpose the ensemble until convergence. Initially, all conformations are aligned with the reference coordinates. Then mean coordinates are calculated, and are set as the new reference coordinates. This is repeated until reference coordinates do not change. This is determined by the value of RMSD between the new and old reference coordinates. Note that at the end of the iterative procedure the reference coordinate set will be average of conformations in the ensemble.

Parameters **rmsd** (*float*[564]) – change in reference coordinates to determine convergence, default is 0.0001 Å RMSD

numAtoms(*selected=True*)
Returns number of atoms.

numConfs()
Returns number of conformations.

[563] http://docs.python.org/library/functions.html#bool
[564] http://docs.python.org/library/functions.html#float

numCoordsets()
Returns number of conformations.

numSelected()
Returns number of selected atoms. Number of all atoms will be returned if a selection is not made. A subset of atoms can be selected by passing a selection to *setAtoms()* (page 210).

select(*selection*)
Selects columns corresponding to a part of the atoms.

setAtoms(*atoms*)
Set *atoms* or specify a selection of atoms to be considered in calculations and coordinate requests. When a selection is set, corresponding subset of coordinates will be considered in, for example, alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, *Ensemble* (page 202) and *Conformation* (page 201) instances become suitable arguments for *writePDB()* (page 270). Passing **None** as *atoms* argument will deselect atoms.

setCoords(*coords*)
Set *coords* as the ensemble reference coordinate set. *coords* may be an array with suitable data type, shape, and dimensionality, or an object with *getCoords()* (page 208) method.

setData(*label*, *data*)
Store atomic *data* under *label*, which must:

- start with a letter
- contain only alphanumeric characters and underscore
- not be a reserved word (see *listReservedWords()* (page 75))

data must be a `list()` or a `ndarray` and its length must be equal to the number of atoms. If the dimension of the *data* array is 1, i.e. `data.ndim==1`, *label* may be used to make atom selections, e.g. `"label 1 to 10"` or `"label C1 C2"`. Note that, if data with *label* is present, it will be overwritten.

setTitle(*title*)
Set title of the ensemble.

setWeights(*weights*)
Set atomic weights.

superpose(***kwargs*)
Superpose the ensemble onto the reference coordinates obtained by *getCoords()* (page 208).

3.8 KDTree

This module provides *KDTree* (page 210) class as an interface to Thomas Hamelryck's KDTree C module distributed with Biopython.

3.8.1 KD Tree

This module defines *KDTree* (page 210) class for dealing with atomic coordinate sets and handling periodic boundary conditions.

class KDTree(*coords*, ***kwargs*)
An interface to Thomas Hamelryck's C KDTree module that can handle periodic boundary conditions. Both point and pair search are performed using the single *search()* (page 212) method and results are retrieved using *getIndices()* (page 212) and *getDistances()* (page 211).

Periodic Boundary Conditions

Point search

A point search around a *center*, indicated with a question mark (`?`) below, involves making images of the point in cells sharing a wall or an edge with the unitcell that contains the system. The search is performed for all images of the *center* (27 in 3-dimensional space) and unique indices with the minimum distance from them to the *center* are returned.

```
 ___________________________________
|         1|         2|         3|
|        ? |        ? |       ?  |
|__________|__________|__________|
|         4|o  h h  5|         6| ? and H interact in periodic image 4
|        ?H| h  o  ? |       ?  | but not in the original unitcell (5)
|__________|__________|__________|
|         7|         8|         9|
|        ? |        ? |       ?  |
|__________|__________|__________|
```

There are two requirements for this approach to work: (i) the *center* must be in the original unitcell, and (ii) the system must be in the original unitcell with parts in its immediate periodic images.

Pair search

A pair search involves making 26 (or 8 in 2-d) replicas of the system coordinates. A KDTree is built for the system (`O` and `H`) and all its replicas (`o` and `h`). After pair search is performed, unique pairs of indices and minimum distance between them are returned.

```
  ___________________________________
 |o  h h  1|o  h h  2|o  h h  3|
h| h  o   h| h  o   h| h  o    |
 |__________|__________|__________|
 |o  h h  4|O  H H  5|o  h h  6|
h| h  o   H| H  O   h| h  o    |
 |__________|__________|__________|
 |o  h h  7|o  h h  8|o  h h  9|
h| h  o   h| h  o   h| h  o    |
 |__________|__________|__________|
```

Only requirement for this approach to work is that the system must be in the original unitcell with parts in its immediate periodic images.

See also:

wrapAtoms() (page 221) can be used for wrapping atoms into the single periodic image of the system.

Parameters

- **coords** (numpy.ndarray, *Atomic* (page 47), *Frame* (page 293)) – coordinate array with shape `(N, 3)`, where N is number of atoms
- **unitcell** (numpy.ndarray) – orthorhombic unitcell dimension array with shape `(3,)`
- **bucketsize** (*int*[565]) – number of points per tree node, default is 10

getCount()

Returns number of points or pairs.

[565] http://docs.python.org/library/functions.html#int

getDistances()
Returns array of distances.

getIndices()
Returns array of indices for points or pairs, depending on the type of the most recent search.

search(*radius*, *center=None*)
Search pairs within *radius* of each other or points within *radius* of *center*.

Parameters

- **radius** (*float*[566]) – distance (Å)
- **center** (numpy.ndarray) – a point in Cartesian coordinate system

3.9 Measurement Tools

This module defines classes measuring quantities, transforming coordinates, and identifying contacts.

3.9.1 Identify contacts

The following class and functions are for contact identifications:

- *Contacts* (page 213) - identify intermolecular contacts
- *findNeighbors()* (page 213) - identify interacting atom pairs
- *iterNeighbors()* (page 213) - identify interacting atom pairs

3.9.2 Measure quantities

The following functions are for measuring simple quantities:

- *calcDistance()* (page 214) - calculate distance(s)
- *calcAngle()* (page 214) - calculate bond angle
- *calcDihedral()* (page 214) - calculate dihedral angle
- *calcOmega()* (page 214) - calculate omega (ω) angle
- *calcPhi()* (page 214) - calculate phi (ϕ) angle
- *calcPsi()* (page 214) - calculate psi (ψ) angle
- *calcGyradius()* (page 214) - calculate radius of gyration
- *calcCenter()* (page 214) - calculate geometric (or mass) center
- *calcDeformVector()* (page 215) - calculate deformation vector

3.9.3 Anisotropic factors

The following functions handle anisotropic displacement parameter (ADP) present in some X-ray structures.

- *buildADPMatrix()* (page 215) - build ADP matrix
- *calcADPAxes()* (page 215) - calculate ADP axes
- *calcADPs()* (page 217) - calculate ADPs

[566] http://docs.python.org/library/functions.html#float

3.9.4 Transformations

The following class and functions are for handling coordinate transformations:

- *Transformation* (page 218) - store transformation matrix
- *alignCoordsets()* (page 219) - align multiple coordinate sets
- *applyTransformation()* (page 219) - apply a transformation
- *calcTransformation()* (page 220) - calculate a transformation
- *calcRMSD()* (page 219) - calculate root-mean-square distance
- *superpose()* (page 220) - superpose atoms or coordinate sets
- *moveAtoms()* (page 220) - move atoms by given offset

3.9.5 Contact Identification

This module defines a class and function for identifying contacts.

class Contacts(*atoms*, *unitcell=None*)

A class for contact identification. Contacts are identified using the coordinates of atoms at the time of instantiation.

atoms must be an *Atomic* (page 47) instance.

Parameters **unitcell** (ndarray) – orthorhombic unitcell dimension array with shape (3,) for KDTree. Default is **None**.

getAtoms()

Returns atoms, or coordinate array, provided at instantiation..

getUnitcell()

Returns unitcell array, or **None** if one was not provided.

select(*radius*, *center*)

Select atoms radius *radius* (Å) of *center*, which can be point(s) in 3-d space (numpy.ndarray with shape (n_atoms, 3)) or a set of atoms, e.g. *Selection* (page 100).

iterNeighbors(*atoms*, *radius*, *atoms2=None*, *unitcell=None*, *seqsep=None*)

Yield pairs of *atoms* that are within *radius* of each other and the distance between them.

If *atoms2* is also provided, one atom from *atoms* and another from *atoms2* will be yielded. If one of *atoms* or *atoms2* is a coordinate array, pairs of indices and distances will be yielded.

When orthorhombic *unitcell* dimensions are provided, periodic boundary conditions will be taken into account (see *KDTree* (page 210) and also wrapAtoms() for details). If *atoms* is a *Frame* (page 293) instance and *unitcell* is not provided, unitcell information from frame will be used if available.

If *seqsep* is provided, neighbors will be kept if the sequence separation >= *seqsep*. Note that *seqsep* will be ignored if atoms are not provided.

findNeighbors(*atoms*, *radius*, *atoms2=None*, *unitcell=None*, *seqsep=None*)

Returns list of neighbors that are within *radius* of each other and the distance between them. See *iterNeighbors()* (page 213) for more details.

3.9.6 Measurement Tools

This module defines a class and methods and for comparing coordinate data and measuring quantities.

buildDistMatrix(*atoms1*, *atoms2=None*, *unitcell=None*, *format='mat'*, *seqsep=None*)
Returns distance matrix. When *atoms2* is given, a distance matrix with shape `(len(atoms1), len(atoms2))` is built. When *atoms2* is **None**, a symmetric matrix with shape `(len(atoms1), len(atoms1))` is built. If *unitcell* array is provided, periodic boundary conditions will be taken into account.

Parameters

- **atoms1** (*Atomic* (page 47), `numpy.ndarray`) – atom or coordinate data
- **atoms2** (*Atomic* (page 47), `numpy.ndarray`) – atom or coordinate data
- **unitcell** (`numpy.ndarray`) – orthorhombic unitcell dimension array with shape `(3,)`
- **format** (*bool*[567]) – format of the resulting array, one of `'mat'` (matrix, default), `'rcd'` (arrays of row indices, column indices, and distances), or `'arr'` (only array of distances)
- **seqsep** (*int*[568]) – if provided, distances will only be measured between atoms with resnum differences that are greater than or equal to seqsep.

calcDistance(*atoms1*, *atoms2*, *unitcell=None*)
Returns the Euclidean distance between *atoms1* and *atoms2*. Arguments may be *Atomic* (page 47) instances or NumPy arrays. Shape of numpy arrays must be `([M,]N,3)`, where *M* is number of coordinate sets and *N* is the number of atoms. If *unitcell* array is provided, periodic boundary conditions will be taken into account.

Parameters

- **atoms1** (*Atomic* (page 47), `numpy.ndarray`) – atom or coordinate data
- **atoms2** (*Atomic* (page 47), `numpy.ndarray`) – atom or coordinate data
- **unitcell** (`numpy.ndarray`) – orthorhombic unitcell dimension array with shape `(3,)`

calcCenter(*atoms*, *weights=None*)
Returns geometric center of *atoms*. If *weights* is given it must be a flat array with length equal to number of atoms. Mass center of atoms can be calculated by setting weights equal to atom masses, i.e. `weights=atoms.getMasses()`.

calcGyradius(*atoms*, *weights=None*)
Calculate radius of gyration of *atoms*.

calcAngle(*atoms1*, *atoms2*, *atoms3*, *radian=False*)
Returns the angle between atoms in degrees.

calcDihedral(*atoms1*, *atoms2*, *atoms3*, *atoms4*, *radian=False*)
Returns the dihedral angle between atoms in degrees.

calcOmega(*residue*, *radian=False*, *dist=4.1*)
Returns ω (omega) angle of *residue* in degrees. This function checks the distance between Cα atoms of two residues and raises an exception if the residues are disconnected. Set *dist* to **None**, to avoid this.

calcPhi(*residue*, *radian=False*, *dist=4.1*)
Returns ϕ (phi) angle of *residue* in degrees. This function checks the distance between Cα atoms of two residues and raises an exception if the residues are disconnected. Set *dist* to **None**, to avoid this.

[567] http://docs.python.org/library/functions.html#bool
[568] http://docs.python.org/library/functions.html#int

calcPsi (*residue*, *radian=False*, *dist=4.1*)
Returns ψ (psi) angle of *residue* in degrees. This function checks the distance between Cα atoms of two residues and raises an exception if the residues are disconnected. Set *dist* to **None**, to avoid this.

calcMSF (*coordsets*)
Calculate mean square fluctuation(s) (MSF). *coordsets* may be an instance of *Ensemble* (page 202), *TrajBase* (page 294), or *Atomic* (page 47). For trajectory objects, e.g. *DCDFile* (page 290), frames will be considered after they are superposed. For other ProDy objects, coordinate sets should be aligned prior to MSF calculation.

Note that using trajectory files that store 32-bit coordinate will result in lower precision in calculations. Over 10,000 frames this may result in up to 5% difference from the values calculated using 64-bit arrays. To ensure higher-precision calculations for *DCDFile* (page 290) instances, you may use *astype* argument, i.e. `astype=float`, to auto recast coordinate data to double-precision (64-bit) floating-point format.

calcRMSF (*coordsets*)
Returns root mean square fluctuation(s) (RMSF). *coordsets* may be an instance of *Ensemble* (page 202), *TrajBase* (page 294), or *Atomic* (page 47). For trajectory objects, e.g. *DCDFile* (page 290), frames will be considered after they are superposed. For other ProDy objects, coordinate sets should be aligned prior to MSF calculation.

Note that using trajectory files that store 32-bit coordinate will result in lower precision in calculations. Over 10,000 frames this may result in up to 5% difference from the values calculated using 64-bit arrays. To ensure higher-precision calculations for *DCDFile* (page 290) instances, you may use *astype* argument, i.e. `astype=float`, to auto recast coordinate data to double-precision (64-bit) floating-point format.

calcDeformVector (*from_atoms*, *to_atoms*, *weights=None*)
Returns deformation from *from_atoms* to *atoms_to* as a *Vector* (page 170) instance.

buildADPMatrix (*atoms*)
Returns a 3Nx3N symmetric matrix containing anisotropic displacement parameters (ADPs) along the diagonal as 3x3 super elements.

```
In [1]: from prody import *

In [2]: protein = parsePDB('1ejg')

In [3]: calphas = protein.select('calpha')

In [4]: adp_matrix = buildADPMatrix(calphas)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-d4af2f2cc2ed> in <module>()
----> 1 adp_matrix = buildADPMatrix(calphas)

/home/exx/ProDy-website/ProDy/prody/measure/measure.pyc in buildADPMatrix(atoms)
    812         element[0, 2] = element[2, 0] = anisou[4]
    813         element[1, 2] = element[2, 1] = anisou[5]
--> 814         adp[i*3:i*3, i*3:i*3] = element
    815     return adp
    816

ValueError: could not broadcast input array from shape (3,3) into shape (0,0)
```

calcADPAxes (*atoms*, ***kwargs*)
Returns a 3Nx3 array containing principal axes defining anisotropic displacement parameter (ADP, or anisotropic temperature factor) ellipsoids.

Parameters

- **atoms** (*Atomic* (page 47)) – a ProDy object for handling atomic data
- **fract** (*float*[569]) – For an atom, if the fraction of anisotropic displacement explained by its largest axis/eigenvector is less than given value, all axes for that atom will be set to zero. Values larger than 0.33 and smaller than 1.0 are accepted.
- **ratio2** (*float*[570]) – For an atom, if the ratio of the second-largest eigenvalue to the largest eigenvalue axis less than or equal to the given value, all principal axes for that atom will be returned. Values less than 1 and greater than 0 are accepted.
- **ratio3** (*float*[571]) – For an atom, if the ratio of the smallest eigenvalue to the largest eigenvalue is less than or equal to the given value, all principal axes for that atom will be returned. Values less than 1 and greater than 0 are accepted.
- **ratio** (*float*[572]) – Same as *ratio3*.

Keyword arguments *fract*, *ratio2*, or *ratio3* can be used to set principal axes to 0 for atoms showing relatively lower degree of anisotropy.

3Nx3 axis contains N times 3x3 matrices, one for each given atom. Columns of these 3x3 matrices are the principal axes which are weighted by square root of their eigenvalues. The first columns correspond to largest principal axes.

The direction of the principal axes for an atom is determined based on the correlation of the axes vector with the principal axes vector of the previous atom.

```
In [1]: from prody import *

In [2]: protein = parsePDB('1ejg')

In [3]: calphas = protein.select('calpha')

In [4]: adp_axes = calcADPAxes( calphas )
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-5a7bd3de55e3> in <module>()
----> 1 adp_axes = calcADPAxes( calphas )

/home/exx/ProDy-website/ProDy/prody/measure/measure.pyc in calcADPAxes(atoms, **kwargs)
    730         # is selected
    731         vals = vals * sign((vecs * axes[(i-1)*3:(i)*3, :]).sum(0))
--> 732         axes[i*3:i*3, :] = vals * vecs
    733     # Resort the columns before returning array
    734     axes = axes[:, [2, 1, 0]]

ValueError: could not broadcast input array from shape (3,3) into shape (0,3)

In [5]: adp_axes.shape
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-71e545866239> in <module>()
----> 1 adp_axes.shape

NameError: name 'adp_axes' is not defined
```

[569] http://docs.python.org/library/functions.html#float
[570] http://docs.python.org/library/functions.html#float
[571] http://docs.python.org/library/functions.html#float
[572] http://docs.python.org/library/functions.html#float

These can be written in NMD format as follows:

```
In [6]: nma = NMA('ADPs')

In [7]: nma.setEigens(adp_axes)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-e3908103788d> in <module>()
----> 1 nma.setEigens(adp_axes)

NameError: name 'adp_axes' is not defined

In [8]: nma
Out[8]: <NMA: ADPs (0 modes; 0 atoms)>

In [9]: writeNMD('adp_axes.nmd', nma, calphas)
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-9-2a12f0fa27fa> in <module>()
----> 1 writeNMD('adp_axes.nmd', nma, calphas)

/home/exx/ProDy-website/ProDy/prody/dynamics/nmdfile.pyc in writeNMD(filename, modes, atoms)
    388                          'not {0}'.format(type(modes)))
    389     if modes.numAtoms() != atoms.numAtoms():
--> 390         raise Exception('number of atoms do not match')
    391     out = openFile(filename, 'w')
    392

Exception: number of atoms do not match
```

calcADPs(*atom*)
Calculate anisotropic displacement parameters (ADPs) from anisotropic temperature factors (ATFs).

atom must have ATF values set for ADP calculation. ADPs are returned as a tuple, i.e. (eigenvalues, eigenvectors).

pickCentral(*obj*, *weights=None*)
Returns *Atom* (page 32) or *Conformation* (page 201) that is closest to the center of *obj*, which may be an *Atomic* (page 47) or *Ensemble* (page 202) instance. See also *pickCentralAtom()* (page 217), and *pickCentralConf()* (page 217) functions.

pickCentralAtom(*atoms*, *weights=None*)
Returns *Atom* (page 32) that is closest to the center, which is calculated using *calcCenter()* (page 214).

pickCentralConf(*ens*, *weights=None*)
Returns *Conformation* (page 201) that is closest to the center of *ens*. In addition to *Ensemble* (page 202) instances, *Atomic* (page 47) instances are accepted as *ens* argument. In this case a *Selection* (page 100) with central coordinate set as active will be returned.

calcInertiaTensor(*coords*)
"Calculate inertia tensor from coords

calcPrincAxes(*coords*, *turbo=True*)
Calculate principal axes from coords

calcDistanceMatrix(*coords*, *cutoff=None*)
Calculate matrix of distances between coordinates within *cutoff*. Other matrix entries are set to maximum of calculated distances.

Parameters

- **coords** (ndarray, *Atomic* (page 47)) – a coordinate set or an object with `getCoords()` method.
- **cutoff** (*None, float*) – cutoff distance for searching the KDTree. Default (**None**) is to use the length of the longest coordinate axis.

assignBlocks(*atoms*, *res_per_block=None*, *secstr=False*, ***kwargs*)

Assigns blocks to protein from *atoms* using a block size of *res_per_block* or secondary structure information if *secstr* is **True**.

Returns an array of block IDs and an AtomMap corresponding to protein atoms.

Parameters

- **atoms** (Atomic) – atoms to be assigned blocks
- **res_per_block** (*int*[573]) – number of residues per block The last block may be smaller or larger than this. Default is **None**, allowing *secstr* to be used easily instead.
- **secstr** (*bool*[574]) – use secondary structure information to assign blocks. Default is **False**, allowing *res_per_block* to be used easily instead. Any set of strings that can be retrieved by *getSecstr()* (page 34) is acceptable including from PDB header, DSSP or STRIDE.
- **shortest_block** (*int*[575]) – smallest number of residues to be included in a block before merging with the previous block Default is **4** as smaller numbers can cause problems for distance matrices.
- **longest_block** (*int*[576]) – largest number of residues to be included in a block before splitting it in half. Default is the length of the protein so it isn't triggered.
- **min_dist_cutoff** (*Number*) – minimum distance of a residue from others beyond which it is not included in the same block as them using `findSubgroups()`. Default is 20 A, which was found to work well with *res_per_block*=10.

3.9.7 Transformations

This module defines a class for identifying contacts.

class Transformation(**args*)

A class for storing a transformation matrix.

Either 4x4 transformation *matrix*, or *rotation* matrix and *translation* vector must be provided at instantiation.

apply(*atoms*)

Apply transformation to *atoms*, see *applyTransformation()* (page 219) for details.

getMatrix()

Returns a copy of the 4x4 transformation matrix whose top left is rotation matrix and last column is translation vector.

getRotation()

Returns rotation matrix.

getTranslation()

Returns translation vector.

[573] http://docs.python.org/library/functions.html#int
[574] http://docs.python.org/library/functions.html#bool
[575] http://docs.python.org/library/functions.html#int
[576] http://docs.python.org/library/functions.html#int

setRotation (*rotation*)
Set rotation matrix.

setTranslation (*translation*)
Set translation vector.

applyTransformation (*transformation*, *atoms*)
Returns *atoms* after applying *transformation*. If *atoms* is a *Atomic* (page 47) instance, it will be returned after *transformation* is applied to its active coordinate set. If *atoms* is an *AtomPointer* (page 78) instance, *transformation* will be applied to the corresponding coordinate set in the associated *AtomGroup* (page 38).

alignCoordsets (*atoms*, *weights=None*)
Returns *atoms* after superposing coordinate sets onto its active coordinate set. Transformations will be calculated for *atoms* and applied to its *AtomGroup* (page 38), when applicable. Optionally, atomic *weights* can be passed for weighted superposition.

calcRMSD (*reference*, *target=None*, *weights=None*)
Returns root-mean-square deviation (RMSD) between reference and target coordinates.

```
In [1]: ens = loadEnsemble('p38_X-ray.ens.npz')
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-1-dd128c0f7ede> in <module>()
----> 1 ens = loadEnsemble('p38_X-ray.ens.npz')

/home/exx/ProDy-website/ProDy/prody/ensemble/functions.pyc in loadEnsemble(filename, **kwargs)
     95         kwargs['allow_pickle'] = True
     96
---> 97     attr_dict = np.load(filename, **kwargs)
     98     if '_weights' in attr_dict:
     99         weights = attr_dict['_weights']

/home/exx/miniconda3/envs/py27/lib/python2.7/site-packages/numpy/lib/npyio.pyc in load(file, mma
    420         own_fid = False
    421     else:
--> 422         fid = open(os_fspath(file), "rb")
    423         own_fid = True
    424

IOError: [Errno 2] No such file or directory: 'p38_X-ray.ens.npz'

In [2]: ens.getRMSDs()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-9e688260fdc1> in <module>()
----> 1 ens.getRMSDs()

NameError: name 'ens' is not defined

In [3]: calcRMSD(ens)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-f69c1b48e438> in <module>()
----> 1 calcRMSD(ens)

NameError: name 'ens' is not defined

In [4]: calcRMSD(ens.getCoords(), ens.getCoordsets(), ens.getWeights())
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-1a40d17246f8> in <module>()
----> 1 calcRMSD(ens.getCoords(), ens.getCoordsets(), ens.getWeights())

NameError: name 'ens' is not defined
```

calcTransformation(*mobile*, *target*, *weights=None*)
Returns a *Transformation* (page 218) instance which, when applied to the atoms in *mobile*, minimizes the weighted RMSD between *mobile* and *target*. *mobile* and *target* may be NumPy coordinate arrays, or *Atomic* (page 47) instances, e.g. *AtomGroup* (page 38), *Chain* (page 53), or *Selection* (page 100).

superpose(*mobile*, *target*, *weights=None*)
Returns *mobile*, after its RMSD minimizing superposition onto *target*, and the transformation that minimizes the RMSD.

moveAtoms(*atoms*, ***kwargs*)
Move *atoms* *to* a new location or *by* an offset. This method will change the active coordinate set of the *atoms*. Note that only one of *to* or *by* keyword arguments is expected.

Move protein so that its centroid is at the origin, `[0., 0., 0.]`:

```
In [1]: from prody import *

In [2]: from numpy import ones, zeros

In [3]: protein = parsePDB('1ubi')

In [4]: calcCenter(protein).round(3)
Out[4]: array([30.173, 28.658, 15.262])

In [5]: moveAtoms(protein, to=zeros(3))

In [6]: calcCenter(protein).round(3)
Out[6]: array([ 0.,  0., -0.])
```

Move protein so that its mass center is at the origin:

```
In [7]: protein.setMasses(ones(len(protein)))

In [8]: protein.carbon.setMasses(12)

In [9]: protein.nitrogen.setMasses(14)

In [10]: protein.oxygen.setMasses(16)

In [11]: moveAtoms(protein, to=zeros(3), weights=protein.getMasses())

In [12]: calcCenter(protein, weights=protein.getMasses()).round(3)
Out[12]: array([-0., -0.,  0.])
```

Move protein so that centroid of Cα atoms is at the origin:

```
In [13]: moveAtoms(protein.ca, to=zeros(3), ag=True)

In [14]: calcCenter(protein).round(3)
Out[14]: array([-0.268, -0.343, -0.259])
```

```
In [15]: calcCenter(protein.ca).round(3)
Out[15]: array([ 0., -0., -0.])
```

Move protein by 10 A along each direction:

```
In [16]: moveAtoms(protein, by=ones(3) * 10)

In [17]: calcCenter(protein).round(3)
Out[17]: array([9.732, 9.657, 9.741])

In [18]: calcCenter(protein.ca).round(3)
Out[18]: array([10., 10., 10.])
```

Parameters

- **by** (numpy.ndarray) – an offset array with shape ([1,] 3) or (n_atoms, 3) or a transformation matrix with shape (4, 4)
- **to** (numpy.ndarray) – a point array with shape ([1,] 3)
- **ag** (*bool*[577]) – when *atoms* is a *AtomSubset* (page 105), apply translation vector (*to*) or transformation matrix to the *AtomGroup* (page 38), default is **False**
- **weights** (numpy.ndarray) – array of atomic weights with shape (n_atoms[, 1])

When *to* argument is passed, *calcCenter()* (page 214) function is used to calculate centroid or mass center.

wrapAtoms(*frame*, *unitcell=None*, *center=array([0., 0., 0.])*)

Wrap atoms into an image of the system simulated under periodic boundary conditions. When *frame* is a *Frame* (page 293), unitcell information will be retrieved automatically.

Note: This function will wrap all atoms into the specified periodic image, so covalent bonds will be broken.

Parameters

- **frame** (*Frame* (page 293), *AtomGroup* (page 38), numpy.ndarray) – a frame instance or a coordinate set
- **unitcell** (numpy.ndarray) – orthorhombic unitcell array with shape (3,)
- **center** (numpy.ndarray) – coordinates of the center of the wrapping cell, default is the origin of the Cartesian coordinate system

printRMSD(*reference*, *target=None*, *weights=None*, *log=True*, *msg=None*)

Print RMSD to the screen. If *target* has multiple coordinate sets, minimum, maximum and mean RMSD values are printed. If *log* is **True** (default), RMSD is written to the standard error using package logger, otherwise standard output is used. When *msg* string is given, it is printed before the RMSD value. See also *calcRMSD()* (page 219) function.

[577] http://docs.python.org/library/functions.html#bool

3.10 Protein Structure

This module defines classes and functions to fetch, parse, and write structural data files, execute structural analysis programs, and to access and search structural databases, e.g. ProteinDataBank[578].

3.10.1 PDB resources

- *fetchPDB()* (page 266) - retrieve PDB files
- *fetchPDBviaFTP()* (page 277) - download PDB/PDBML/mmCIF files
- *fetchPDBviaHTTP()* (page 278) - download PDB files

You can use following functions to manage PDB file resources:

- *pathPDBFolder()* (page 266) - local folder for storing PDB files
- *pathPDBMirror()* (page 266) - local PDB mirror path
- *wwPDBServer()* (page 277) - set wwPDB FTP/HTTP server for downloads

The following functions can be used to handle local PDB files:

- *findPDBFiles()* (page 266) - return a dictionary containing files in a path
- *iterPDBFilenames()* (page 266) - yield file names in a path or local PDB mirror

3.10.2 Blast search PDB

The following are for blast searching PDB content.

- *blastPDB()* (page 226) - blast search NCBI PDB database
- *PDBBlastRecord* (page 225) - store/evaluate NCBI PDB blast search results

PDB clusters biopolymer chains using blast weekly. These clusters can be retrieved using the following functions. Using cluster data is as good as blast searching PDB most of the time and incredibly faster always.

- *listPDBCluster()* (page 267) - get list of identifiers in a PDB sequence cluster
- *loadPDBClusters()* (page 267) - load PDB clusters into memory
- *fetchPDBClusters()* (page 267) - retrieve PDB sequence cluster data from wwPDB

3.10.3 Parse/write PDB files

Following ProDy functions are for parsing and writing `.pdb` files:

- *parsePDB()* (page 268) - parse `.pdb` formated file
- *parsePDBStream()* (page 267) - parse `.pdb` formated stream
- *writePDB()* (page 270) - write `.pdb` formatted file
- *writePDBStream()* (page 270) write `.pdb` formated stream

Since `.pqr` format is similar to `.pdb` format, following functions come as bonus features:

- *writePQR()* (page 271) - write atomic data to a file in `.pqr` format
- *parsePQR()* (page 269) - parse atomic data from files in `.pqr` format

[578] http://wwpdb.org

See also:

Atom data (coordinates, atom names, residue names, etc.) parsed from PDB/PSF/PQR/mmCIF files are stored in *AtomGroup* (page 38) instances. See *atomic* (page 29) module documentation for more details.

3.10.4 Parse mmCIF files

Following ProDy functions are for parsing `.cif` files:

- *parseMMCIF()* (page 227) - parse `.cif` formated file
- *parseMMCIFStream()* (page 226) - parse `.cif` formated stream

See also:

Atom data (coordinates, atom names, residue names, etc.) parsed from PDB/PSF/PQR/mmCIF files are stored in *AtomGroup* (page 38) instances. See *atomic* (page 29) module documentation for more details.

3.10.5 Quick visualization

showProtein() (page 236) function can be used to take a quick look at protein structures.

3.10.6 Edit structures

The following functions allow editing structures using structural data from PDB header records:

- *assignSecstr()* (page 242) - add secondary structure data from header to atoms
- *buildBiomolecules()* (page 242) - build biomolecule from header records

3.10.7 PDB header data

Use the following to parse and access header data in PDB files:

- *parsePDBHeader()* (page 240) - parse header data from `.pdb` files
- *Chemical* (page 236) - store PDB chemical (heterogen) component data
- *Polymer* (page 238) - store PDB polymer (macromolecule) component data
- *DBRef* (page 239) - store polymer sequence database reference records

3.10.8 Analyze interactions and stability with InSty and find water bridges with WatFinder

Use the following to analyze interactions within protein structure or between protein and ligand structure in single PDB file or in trajectory:

- `addHydrogens()` - add missing hydrogens to `.pdb` files
- *calcHydrogenBonds()* (page 252) - compute hydrogen bonds in proteins
- *calcSaltBridges()* (page 253) - compute salt bridges in proteins
- *calcRepulsiveIonicBonding()* (page 254) - compute repulsive ionic bonding in proteins
- *calcPiStacking()* (page 254) - compute Pi-stacking interactions in proteins
- *calcPiCation()* (page 255) - compute Pi-cation interactions in proteins
- *calcHydrophobic()* (page 255) - compute hydrophobic interactions in proteins
- *calcProteinInteractions()* (page 260) - compute all above interaction types at once

- *showProteinInteractions_VMD()* (page 263) - return TCL file for visualization in VMD
- calcHydrogenBondsDCD() - compute hydrogen bonds in a trajectory for proteins
- calcSaltBridgesDCD() - ompute salt bridges in a trajectory for proteins
- calcRepulsiveIonicBondingDCD() - compute repulsive ionic bonding in a trajectory for proteins
- calcPiStackingDCD() - compute Pi-stacking interactions in a trajectory for proteins
- calcPiCationDCD() - compute Pi-cation interactions in a trajectory for proteins
- calcHydrophobicDCD() - compute hydrophobic interactions in a trajectory for proteins
- *calcStatisticsInteractions()* (page 260) - return statistical information for each interaction type
- *calcLigandInteractions()* (page 262) - compute all type of interactions between protein and ligand
- *listLigandInteractions()* (page 263) - return list of interactions between protein and ligand
- *showLigandInteraction_VMD()* (page 263) - return TCL file for visualization of interactions for VMD
- *Interactions* (page 243) - store inteactions for a single PDB structure
- InteractionsDCD - store interactions for a trajectory

3.10.9 Compare/align chains

The following functions can be used to match, align, and map polypeptide chains:

- *matchChains()* (page 228) - finds matching chains in two protein structures
- *matchAlign()* (page 229) - finds best matching chains and aligns structures
- *mapOntoChain()* (page 230) - maps chains in a structure onto a reference chain

The following functions can be used to adjust alignment parameters:

- *getAlignmentMethod()* (page 232), *setAlignmentMethod()* (page 232)
- *getMatchScore()* (page 231), *setMatchScore()* (page 231)
- *getMismatchScore()* (page 231), *setMismatchScore()* (page 231)
- *getGapPenalty()* (page 231), *setGapPenalty()* (page 231)
- *getGapExtPenalty()* (page 231), *setGapExtPenalty()* (page 231)

3.10.10 Execute DSSP

The following functions can be used to execute DSSP structural analysis program and/or parse results:

- *execDSSP()* (page 232) - execute **dssp**
- *performDSSP()* (page 233) - execute **dssp** and parse results
- *parseDSSP()* (page 232) - parse structural data from **dssp** output

3.10.11 Execute STRIDE

The following functions can be used to execute STRIDE structural analysis program and/or parse results:

- *execSTRIDE()* (page 273) - execute **stride**
- *parseSTRIDE()* (page 273) - parse structural data from **stride** output
- *performSTRIDE()* (page 273) - execute **stride** and parse results

3.10.12 Handle EMD Map Files and Build Pseudoatoms into them

Use the following to parse and access header data in EMD files:

- *parseEMD()* (page 235) - parse structural data from `.emd` files
- *EMDMAP* (page 234) - access structural data from `.emd` files
- *TRNET* (page 233) - fit pseudoatoms to EM density maps using the TRN algorithm

3.10.13 Add missing atoms including hydrogens

Use the following to add missing atoms

- `addMissingAtoms()` - add missing atoms with separately installed OpenBabel or PDBFixer

Missing residues can also be added if a PDB or mmCIF file with SEQRES entries is provided.

3.10.14 PDB Blast Search

This module defines functions for blast searching the Protein Data Bank.

class **PDBBlastRecord**(*xml=None*, *sequence=None*, ***kwargs*)

A class to store results from blast searches.

Instantiate a PDBBlastRecord object instance.

Parameters

- **xml** (*str*[579]) – blast search results in XML format or an XML file that contains the results
- **sequence** (*str*[580]) – query sequence

fetch(*xml=None*, *sequence=None*, ***kwargs*)

Get Blast record from url or file.

Parameters

- **sequence** (*Atomic* (page 47), *Sequence* (page 289), str) – an object with an associated sequence string or a sequence string itself
- **xml** (*str*[581]) – blast search results in XML format or an XML file that contains the results or a filename for saving the results or None
- **timeout** (*int*[582]) – amount of time until the query times out in seconds default value is 120

[579] http://docs.python.org/library/stdtypes.html#str
[580] http://docs.python.org/library/stdtypes.html#str
[581] http://docs.python.org/library/stdtypes.html#str
[582] http://docs.python.org/library/functions.html#int

getBest()
Returns a dictionary containing structure and alignment information for the hit with highest sequence identity.

getHits(*percent_identity=0.0*, *percent_overlap=0.0*, *chain=False*)
Returns a dictionary in which PDB identifiers are mapped to structure and alignment information.

Parameters

- **percent_identity** (*float*[583]) – PDB hits with percent sequence identity equal to or higher than this value will be returned, default is `0`.
- **percent_overlap** (*float*[584]) – PDB hits with percent coverage of the query sequence equivalent or better will be returned, default is `0`.
- **chain** (*bool*[585]) – if chain is **True**, individual chains in a PDB file will be considered as separate hits, default is **False**

getParameters()
Returns parameters used in blast search.

getSequence()
Returns the query sequence that was used in the search.

writeSequences(*filename*, ***kwargs*)
Returns a plot that contains a dendrogram of the sequence similarities among the sequences in given hit list.

Parameters **hits** (*dict*[586]) – A dictionary that contains hits that are obtained from a blast record object.

Arguments of getHits can be parsed as kwargs.

blastPDB(*sequence*, *filename=None*, ***kwargs*)
Returns a *PDBBlastRecord* (page 225) instance that contains results from blast searching *sequence* against the PDB using NCBI blastp.

Parameters

- **sequence** (*Atomic* (page 47), *Sequence* (page 289), str) – an object with an associated sequence string or a sequence string itself
- **filename** (*str*[587]) – a *filename* to save the results in XML format

hitlist_size (default is `250`) and *expect* (default is `1e-10`) search parameters can be adjusted by the user. *sleep* keyword argument (default is `2` seconds) determines how long to wait to reconnect for results. Sleep time is multiplied by 1.5 when results are not ready. *timeout* (default is 120 s) determines when to give up waiting for the results.

3.10.15 mmCIF File

This module defines functions for parsing mmCIF files[588].

[583] http://docs.python.org/library/functions.html#float
[584] http://docs.python.org/library/functions.html#float
[585] http://docs.python.org/library/functions.html#bool
[586] http://docs.python.org/library/stdtypes.html#dict
[587] http://docs.python.org/library/stdtypes.html#str
[588] http://mmcif.wwpdb.org/docs/tutorials/mechanics/pdbx-mmcif-syntax.html

parseMMCIFStream(*stream*, ***kwargs*)

Returns an *AtomGroup* (page 38) and/or a class:.*StarDict* containing header data parsed from a stream of CIF lines.

Parameters

- **stream** – Anything that implements the method `readlines` (e.g. `file`, buffer, stdin)
- **title** (*str*[589]) – title of the *AtomGroup* (page 38) instance, default is the PDB filename or PDB identifier
- **chain** (*str*[590]) – chain identifiers for parsing specific chains, e.g. `chain='A'`, `chain='B'`, `chain='DE'`, by default all chains are parsed
- **segment** (*str*[591]) – segment identifiers for parsing specific chains, e.g. `segment='A'`, `segment='B'`, `segment='DE'`, by default all segment are parsed
- **subset** (*str*[592]) – a predefined keyword to parse subset of atoms, valid keywords are `'calpha'` (`'ca'`), `'backbone'` (`'bb'`), or **None** (read all atoms), e.g. `subset='bb'`
- **model** (*int, list*) – model index or None (read all models), e.g. `model=10`
- **altloc** (*str*[593]) – if a location indicator is passed, such as `'A'` or `'B'`, only indicated alternate locations will be parsed as the single coordinate set of the AtomGroup, if *altloc* is set `'all'` then all alternate locations will be parsed and each will be appended as a distinct coordinate set, default is `"A"`
- **unite_chains** (*bool*[594]) – unite chains with the same segment name (auth_asym_id), making chain ids be auth_asym_id instead of label_asym_id. This can be helpful in some cases e.g. alignments, but can cause some problems too. For example, using `buildBiomolecules()` afterwards requires original chain id (label_asym_id). Using biomol=True, inside parseMMCIF is fine. Default is *False*

parseMMCIF(*pdb*, ***kwargs*)

Returns an *AtomGroup* (page 38) and/or a *StarDict* (page 271) containing header data parsed from an mmCIF file. If not found, the mmCIF file will be downloaded from the PDB. It will be downloaded in uncompressed format regardless of the compressed keyword.

This function extends *parseMMCIFStream()* (page 226).

Parameters **pdb** (*str*[595]) – a PDB identifier or a filename If needed, mmCIF files are downloaded using *fetchPDB()* (page 266) function.

parseCIF(*pdb*, ***kwargs*)

Returns an *AtomGroup* (page 38) and/or a *StarDict* (page 271) containing header data parsed from an mmCIF file. If not found, the mmCIF file will be downloaded from the PDB. It will be downloaded in uncompressed format regardless of the compressed keyword.

This function extends *parseMMCIFStream()* (page 226).

Parameters **pdb** (*str*[596]) – a PDB identifier or a filename If needed, mmCIF files are downloaded using *fetchPDB()* (page 266) function.

[589] http://docs.python.org/library/stdtypes.html#str
[590] http://docs.python.org/library/stdtypes.html#str
[591] http://docs.python.org/library/stdtypes.html#str
[592] http://docs.python.org/library/stdtypes.html#str
[593] http://docs.python.org/library/stdtypes.html#str
[594] http://docs.python.org/library/functions.html#bool
[595] http://docs.python.org/library/stdtypes.html#str
[596] http://docs.python.org/library/stdtypes.html#str

writeMMCIF(*filename*, *atoms*, *csets=None*, *autoext=True*, ***kwargs*)

Write *atoms* in MMTF format to a file with name *filename* and return *filename*. If *filename* ends with `.gz`, a compressed file will be written.

Parameters

- **atoms** (*Atomic* (page 47)) – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets
- **autoext** – when not present, append extension `.cif` to *filename*
- **header** (*dict*[597]) – header to write too

3.10.16 Structure Comparison

This module defines functions for comparing and mapping polypeptide chains.

matchChains(*atoms1*, *atoms2*, ***kwargs*)

Returns pairs of chains matched based on sequence similarity. Makes an all-to-all comparison of chains in *atoms1* and *atoms2*. Chains are obtained from hierarchical views (*HierView* (page 76)) of atom groups. This function returns a list of matching chains in a tuple that contain 4 items:

- matching chain from *atoms1* as a *AtomMap* (page 49) instance,
- matching chain from *atoms2* as a *AtomMap* (page 49) instance,
- percent sequence identity of the match,
- percent sequence overlap of the match.

List of matches are sorted in decreasing percent sequence identity order. *AtomMap* (page 49) instances can be used to calculate RMSD values and superpose atom groups.

Parameters

- **atoms1** (*Chain* (page 53), *AtomGroup* (page 38), *Selection* (page 100)) – atoms that contain a chain
- **atoms2** (*Chain* (page 53), *AtomGroup* (page 38), *Selection* (page 100)) – atoms that contain a chain
- **subset** (*str*[598]) – one of the following well-defined subsets of atoms: `"calpha"` (or `"ca"`), `"backbone"` (or `"bb"`), `"heavy"` (or `"noh"`), or `"all"`, default is `"calpha"`
- **seqid** (*float*[599]) – percent sequence identity, default is 90
- **overlap** (*float*[600]) – percent overlap, default is 90
- **pwalign** (*bool*[601]) – perform pairwise sequence alignment

If *subset* is set to *calpha* or *backbone*, only alpha carbon atoms or backbone atoms will be paired. If set to *all*, all atoms common to matched residues will be returned.

This function tries to match chains based on residue numbers and names. All chains in *atoms1* is compared to all chains in *atoms2*. This works well for different structures of the same protein. When it fails, Biopython is used for pairwise sequence alignment, and matching is performed based on the

[597] http://docs.python.org/library/stdtypes.html#dict
[598] http://docs.python.org/library/stdtypes.html#str
[599] http://docs.python.org/library/functions.html#float
[600] http://docs.python.org/library/functions.html#float
[601] http://docs.python.org/library/functions.html#bool

sequence alignment. User can control, whether sequence alignment is performed or not with *pwalign* keyword. If `pwalign=True` is passed, pairwise alignment is enforced.

matchAlign(*mobile*, *target*, ***kwargs*)

Superpose *mobile* onto *target* based on best matching pair of chains. This function uses *matchChains()* (page 228) for matching chains and returns a tuple that contains the following items:

- *mobile* after it is superposed,
- matching chain from *mobile* as a *AtomMap* (page 49) instance,
- matching chain from *target* as a *AtomMap* (page 49) instance,
- percent sequence identity of the match,
- percent sequence overlap of the match.

Parameters

- **mobile** (*Chain* (page 53), *AtomGroup* (page 38), *Selection* (page 100)) – atoms that contain a protein chain
- **target** (*Chain* (page 53), *AtomGroup* (page 38), *Selection* (page 100)) – atoms that contain a protein chain
- **tarsel** (*str*[602]) – *target* atoms that will be used for alignment, default is `'calpha'`
- **allcsets** (*bool*[603]) – align all coordinate sets of *mobile*, default is **True**
- **seqid** (*float*[604]) – percent sequence identity, default is 90
- **overlap** (*float*[605]) – percent overlap, default is 90
- **pwalign** (*bool*[606]) – perform pairwise sequence alignment

mapChainOntoChain(*mobile*, *target*, ***kwargs*)

Map *mobile* chain onto *target* chain. This function returns a mapping that contains 4 items:

- Mapped chain as an *AtomMap* (page 49) instance,
- *chain* as an *AtomMap* (page 49) instance,
- Percent sequence identitity,
- Percent sequence overlap

Mappings are returned in decreasing percent sequence identity order. *AtomMap* (page 49) that keeps mapped atom indices contains dummy atoms in place of unmapped atoms.

Parameters

- **mobile** (*Chain* (page 53)) – mobile that will be mapped to the *target* chain
- **target** (*Chain* (page 53)) – chain to which atoms will be mapped
- **seqid** (*float*[607]) – percent sequence identity, default is **90**. Note that this parameter is only effective for sequence alignment
- **overlap** (*float*[608]) – percent overlap with *target*, default is **70**

[602] http://docs.python.org/library/stdtypes.html#str
[603] http://docs.python.org/library/functions.html#bool
[604] http://docs.python.org/library/functions.html#float
[605] http://docs.python.org/library/functions.html#float
[606] http://docs.python.org/library/functions.html#bool
[607] http://docs.python.org/library/functions.html#float
[608] http://docs.python.org/library/functions.html#float

- **mapping** (*list, str, bool*) – what method will be used if the trivial mapping based on residue numbers fails. If `"ce"` or `"cealign"`, then the CE structural alignment *[IS98]* (page 397) will be performed. It can also be a list of prealigned sequences, a *MSA* (page 285) instance, or a dict of indices such as that derived from a *DaliRecord* (page 122). If set to **True** then the sequence alignment from Biopython will be used. If set to **False**, only the trivial mapping will be performed. Default is **"auto"**, which means try sequence alignment then CE.
- **pwalign** (*bool*[609]) – if **True**, then pairwise sequence alignment will be performed. If **False** then a simple mapping will be performed based on residue numbers (as well as insertion codes). This will be overridden by the *mapping* keyword's value.

mapOntoChain (*atoms*, *chain*, ***kwargs*)

Map *atoms* onto *chain*. This function is a wrapper of *mapChainOntoChain()* (page 229) that manages to map chains onto target *chain*. The function returns a list of mappings. Each mapping is a tuple that contains 4 items:

- Mapped chain as an *AtomMap* (page 49) instance,
- *chain* as an *AtomMap* (page 49) instance,
- Percent sequence identitity,
- Percent sequence overlap

Mappings are returned in decreasing percent sequence identity order. *AtomMap* (page 49) that keeps mapped atom indices contains dummy atoms in place of unmapped atoms.

Parameters

- **atoms** (*Chain* (page 53), *AtomGroup* (page 38), *Selection* (page 100)) – atoms that will be mapped to the target *chain*
- **chain** (*Chain* (page 53)) – chain to which atoms will be mapped
- **subset** (*str*[610]) – one of the following well-defined subsets of atoms: `"calpha"` (or `"ca"`), `"backbone"` (or `"bb"`), `"heavy"` (or `"noh"`), or `"all"`, default is `"calpha"`

See *mapChainOntoChain()* (page 229) for other keyword arguments. This function tries to map *atoms* to *chain* based on residue numbers and types. Each individual chain in *atoms* is compared to target *chain*.

alignChains (*atoms*, *target*, *match_func=<function bestMatch>*, ***kwargs*)

Aligns chains of *atoms* to those of *target* using *mapOntoChains()* (page 230) and *combineAtomMaps()* (page 231). Please check out those two functions for details about the parameters.

mapOntoChains (*atoms*, *ref*, *match_func=<function bestMatch>*, ***kwargs*)

This function is a generalization and wrapper of *mapOntoChain()* (page 230) that manages to map chains onto chains (instead of a single chain).

Parameters

- **atoms** (*Atomic* (page 47)) – atoms to map onto the reference
- **ref** (*Atomic* (page 47)) – reference structure for mapping
- **match_func** (*func*) – function determines which chains from `ref` and `atoms` are matched. Default is to use the best match.

[609] http://docs.python.org/library/functions.html#bool
[610] http://docs.python.org/library/stdtypes.html#str

bestMatch(*chain1*, *chain2*)

sameChid(*chain1*, *chain2*)

userDefined(*chain1*, *chain2*, *correspondence*)

sameChainPos(*chain1*, *chain2*)

mapOntoChainByAlignment(*atoms*, *chain*, ***kwargs*)
This function is similar to *mapOntoChain()* (page 230) but correspondence of chains is found by alignment provided.

Parameters **alignments** (list, dict, MSA) – A list of predefined alignments. It can be also a dictionary or MSA instance where the keys or labels are the title of *atoms* or *chains*.

getMatchScore()
Returns match score used to align sequences.

setMatchScore(*match_score*)
Set match score used to align sequences.

getMismatchScore()
Returns mismatch score used to align sequences.

setMismatchScore(*mismatch_score*)
Set mismatch score used to align sequences.

getGapPenalty()
Returns gap opening penalty used for pairwise alignment.

setGapPenalty(*gap_penalty*)
Set gap opening penalty used for pairwise alignment.

getGapExtPenalty()
Returns gap extension penalty used for pairwise alignment.

setGapExtPenalty(*gap_ext_penalty*)
Set gap extension penalty used for pairwise alignment.

getGoodSeqId()
Returns good sequence identity.

setGoodSeqId(*seqid*)
Set good sequence identity.

getGoodCoverage()
Returns good sequence coverage.

combineAtomMaps(*mappings*, *target=None*, ***kwargs*)
Builds a grand *AtomMap* (page 49) instance based on *mappings* obtained from *mapOntoChains()* (page 230). The function also accepts the output *mapOntoChain()* (page 230) but will trivially return all the *AtomMap* (page 49) in *mappings*. *mappings* should be a list or an array of matching chains in a tuple that contain 4 items:

- matching chain from *atoms1* as a *AtomMap* (page 49) instance,
- matching chain from *atoms2* as a *AtomMap* (page 49) instance,
- percent sequence identity of the match,
- percent sequence overlap of the match.

Parameters

- **mappings** (tuple, list, ndarray) – a list or an array of matching chains in a tuple, or just the tuple
- **target** (*Atomic* (page 47)) – reference structure for superposition and checking RMSD
- **drmsd** (*float*[611]) – amount deviation of the RMSD with respect to the top ranking atommap. This is to allow multiple matches when *mobile* has more chains than *target*. Default is 3.0
- **rmsd_reject** (*float*[612]) – upper RMSD cutoff that rejects an atommap. Default is 15.0
- **least** (*int*[613]) – the least number of atommaps requested. If **None**, it will be automatically determined by the number of chains present in *target* and *mobile*. Default is **None**
- **debug** (*dict*[614]) – a container (dict) that saves the following information for debugging purposes: * coverage: original coverage matrix, rows and columns correspond to the reference and the mobile, respectively, * solutions: matched index groups that obtained by modeling the coverage matrix as a linear assignment problem, * rmsd: a list of ranked RMSDs of identified atommaps.

setGoodCoverage(*coverage*)
Set good sequence coverage.

getAlignmentMethod()
Returns pairwise alignment method.

setAlignmentMethod(*method*)
Set pairwise alignment method (global or local).

3.10.17 DSSP Tools

This module defines functions for executing DSSP program and parsing its output.

execDSSP(*pdb*, *outputname=None*, *outputdir=None*, *stderr=True*)
Execute DSSP for given *pdb*. *pdb* can be a PDB identifier or a PDB file path. If *pdb* is a compressed file, it will be decompressed using Python gzip[615] library. When no *outputname* is given, output name will be `pdb.dssp`. `.dssp` extension will be appended automatically to *outputname*. If `outputdir` is given, DSSP output and uncompressed PDB file will be written into this folder. Upon successful execution of **dssp pdb > out** command, output filename is returned. On Linux platforms, when *stderr* is false, standard error messages are suppressed, i.e. `dssp pdb > outputname 2> /dev/null`.

For more information on DSSP see http://swift.cmbi.ru.nl/gv/dssp/. If you benefited from DSSP, please consider citing *[WK83]* (page 397).

parseDSSP(*dssp*, *ag*, *parseall=False*)
Parse DSSP data from file *dssp* into *AtomGroup* (page 38) instance *ag*. DSSP output file must be in the new format used from July 1995 and onwards. When *dssp* file is parsed, following attributes are added to *ag*:

- *dssp_resnum*: DSSP's sequential residue number, starting at the first residue actually in the data set and including chain breaks; this number is used to refer to residues throughout.

[611] http://docs.python.org/library/functions.html#float
[612] http://docs.python.org/library/functions.html#float
[613] http://docs.python.org/library/functions.html#int
[614] http://docs.python.org/library/stdtypes.html#dict
[615] http://docs.python.org/library/gzip.html#module-gzip

- *dssp_acc*: number of water molecules in contact with this residue *10. or residue water exposed surface in Angstrom^2.
- *dssp_kappa*: virtual bond angle (bend angle) defined by the three Cα atoms of residues I-2,I,I+2. Used to define bend (structure code 'S').
- *dssp_alpha*: virtual torsion angle (dihedral angle) defined by the four Cα atoms of residues I-1,I,I+1,I+2.Used to define chirality (structure code '+' or '-').
- *dssp_phi* and *dssp_psi*: IUPAC peptide backbone torsion angles

The following attributes are parsed when `parseall=True` is passed:

- *dssp_bp1*, *dssp_bp2*, and *dssp_sheet_label*: residue number of first and second bridge partner followed by one letter sheet label
- *dssp_tco*: cosine of angle between C=O of residue I and C=O of residue I-1. For α-helices, TCO is near +1, for β-sheets TCO is near -1. Not used for structure definition.
- *dssp_NH_O_1_index*, *dssp_NH_O_1_energy*, etc.: hydrogen bonds; e.g. -3,-1.4 means: if this residue is residue i then N-H of I is h-bonded to C=O of I-3 with an electrostatic H-bond energy of -1.4 kcal/mol. There are two columns for each type of H-bond, to allow for bifurcated H-bonds.

See http://swift.cmbi.ru.nl/gv/dssp/DSSP_3.html for details.

performDSSP(*pdb*, *parseall=False*, *stderr=True*)
Perform DSSP calculations and parse results. DSSP data is returned in an *AtomGroup* (page 38) instance. See also *execDSSP()* (page 232) and *parseDSSP()* (page 232).

3.10.18 EMD File

This module defines functions for parsing and writing EMD map files[616].

class TRNET(*n_nodes*)
Class for building topology representing networks using EM density maps. It uses the algorithm described in *[TM94]* (page 397).

inputMap(*emdmap*, *sample='density'*)

outputEdges()

run(***kwargs*)

Parameters

- **tmax** (*int*[617]) – multiplicative factor such that the maximum total number of iterations is tmax times the number of beads default 200
- **li** (*float*[618]) – initial Gaussian bandwidth for determining how much each node is moved As the iterations progress, the bandwidth increases from li to lf. default 0.2
- **lf** (*float*[619]) – final Gaussian bandwidth for determining how much each node is moved As the iterations progress, the bandwidth increases from li to lf. default 0.01

[616] http://emdatabank.org/mapformat.html
[617] http://docs.python.org/library/functions.html#int
[618] http://docs.python.org/library/functions.html#float
[619] http://docs.python.org/library/functions.html#float

- **ei** (*float*[620]) – initial value of the adaptive step size As the iterations progress, the step size increases from ei to ef. default 0.3
- **ef** (*float*[621]) – final value of the adaptive step size As the iterations progress, the step size increases from ei to ef. default 0.05
- **c** (*float*[622]) – cutoff for moving the nodes. When c=0, all nodes are moved in each iteration. When c>0, only the nearest c/#nodes nodes are moved. This parameter is used for optimization. default 0
- **calcC** (*bool*[623]) – whether to calculate the connectivity matrix from TRN. This is **False** by default because the connectivity is usually built by ANM or GNM. default **False**
- **Ti** (*float*[624]) – initial value of the adaptive threshold for building the connectivity. Not used if calcC is False. default 0.1
- **Tf** (*float*[625]) – final value of the adaptive threshold for building the connectivity. Not used if calcC is False. default 2

runOnce(*t*, *l*, *ep*, *T*, *c=0*)

run_n_pause(*k0*, *k*, *tmax=200*, *li=0.2*, *lf=0.01*, *ei=0.3*, *ef=0.05*, *Ti=0.1*, *Tf=2*)

class EMDMAP(*stream*, *min_cutoff*, *max_cutoff*)

Class for handling EM density maps in EMD/MRC2014 format.

Parameters

- **stream** – a file stream containing data from an EMD/MRC file.
- **min_cutoff** (*None, float*) – minimum cutoff for thresholding
- **max_cutoff** (*None, float*) – maximum cutoff for thresholding

center()

coordinate(*sec*, *row*, *col*)

Given a position as *sec*, *row* and *col*, it will return its coordinate in Angstroms.

copyMap()

Copy to a new object.

drawsample()

drawsample_uniform()

getApix()

getOrigin()

getTitle()

numidx2matidx(*numidx*)

Given index of the position, it will return the numbers of section, row and column.

setApix(*apix*)

setOrigin(*x0*, *y0*, *z0*)

[620] http://docs.python.org/library/functions.html#float
[621] http://docs.python.org/library/functions.html#float
[622] http://docs.python.org/library/functions.html#float
[623] http://docs.python.org/library/functions.html#bool
[624] http://docs.python.org/library/functions.html#float
[625] http://docs.python.org/library/functions.html#float

setTitle(*title*)

thresholdMap(*min_cutoff=None*, *max_cutoff=None*)
Thresholds a map and returns a new map like the equivalent function in TEMPy

toTEMPyMap()
Convert to a TEMPy Map.

apix

filename

origin

parseEMDStream(*stream*, ***kwargs*)
Parse lines of data stream from an EMD/MRC2014 file and optionally return an *AtomGroup* (page 38) containing TRN nodes based on it.

Parameters **stream** – Any object with the method `readlines` (e.g. `file`, buffer, stdin)

parseEMD(*emd*, ***kwargs*)
Parses an EM density map in EMD/MRC2014 format and optionally returns an *AtomGroup* (page 38) containing beads built in the density using the TRN algorithm [_TM94].

This function extends *parseEMDStream()* (page 235).

See Cryo-EM Dynamics (CryoDy)[626] for a usage example.

Parameters

- **emd** (*str*[627]) – an EMD identifier or a file name. A 4-digit EMDataBank identifier can be provided to download it via FTP.
- **min_cutoff** (*float*[628]) – minimum density cutoff to read EMD map. The regions with lower density than this cutoff are discarded. This corresponds to the previous cutoff and take values from it.
- **max_cutoff** (*float*[629]) – maximum density cutoff to read EMD map. The regions with higher density than this cutoff are discarded.
- **n_nodes** (*int*[630]) – A bead based network will be constructed into the provided density map. This parameter will set the number of beads to fit to density map. Default is 0. Please change it to some number to run the TRN algorithm. Other parameters are passed through as kwargs to *TRNET.run()* (page 233) as described in its docs.
- **map** (*bool*[631]) – Return the density map itself. Default is **False** in line with previous behaviour. This value is reset to **True** if n_nodes is 0 or less.

writeEMD(*filename*, *emd*)
Writes a map file in MRC2014 format (counting words 25 to 49 as 'extra').

Parameters

- **filename** (*str*[632]) – name for output file
- **emd** (`EMD`) – an EMD object containing data to be written to file

[626] http://prody.csb.pitt.edu/tutorials/cryoem_tutorial/index.html#cryoem-analysis
[627] http://docs.python.org/library/stdtypes.html#str
[628] http://docs.python.org/library/functions.html#float
[629] http://docs.python.org/library/functions.html#float
[630] http://docs.python.org/library/functions.html#int
[631] http://docs.python.org/library/functions.html#bool
[632] http://docs.python.org/library/stdtypes.html#str

3.10.19 Miscellaneous Tools

This module defines miscellaneous functions dealing with protein data.

view3D(**alist*, ***kwargs*)

Return a py3Dmol view instance for interactive visualization in Jupyter notebooks. Available arguments are: width, height (of the viewer), backgroundColor, zoomTo (a py3Dmol selection to center around), and style, which is a py3Dmol style object that will be applied to all atoms in the scene. More complex styling can be achieved by manipulating the view object directly.

The default style is to show the protein in a rainbow cartoon and hetero atoms in sticks/spheres.

GNM/ANM Coloring

An array of fluctuation values can be provided with the *data* kwarg for visualization of GNM/ANM calculations. The array is assumed to correpond to a calpha selection of the provided protein. The default color will be set to a RWB color scheme on a per-residue basis. If the fluctuation vector contains negative values, the midpoint (white) will be at zero. Otherwise the midpoint is the mean.

An array of displacement vectors can be provided with the *mode* kwarg. The animation of these motions can be controlled with frames (number of frames to animate over), amplitude (scaling factor), and animate (3Dmol.js animate options). If animation isn't enabled, by default arrows are drawn. Drawing of arrows is controlled by the boolean arrows option and the arrowcolor option.

If multiple structures are provided with the data or mode arguments, these arguments must be provided as lists of arrays of the appropriate dimension.

If a 3Dmol.js viewer as specified as the view argument, that viewer will be modified and returned. After modification, update instead of show should be called on the viewer object if it is desired to update in-place instead of instantiating a new viewer.

showProtein(**atoms*, ***kwargs*)

Show protein representation using `Axes3D()`. This function is designed for generating a quick view of the contents of a *AtomGroup* (page 38) or *Selection* (page 100).

Protein atoms matching `"calpha"` selection are displayed using solid lines by picking a random and unique color per chain. Line with can be adjusted using *lw* argument, e.g. `lw=12`. Default width is 4. Chain colors can be overwritten using chain identifier as in `A='green'`.

Water molecule oxygen atoms are represented by red colored circles. Color can be changed using *water* keyword argument, e.g. `water='aqua'`. Water marker and size can be changed using *wmarker* and *wsize* keywords, defaults values are `wmarker='.', wsize=6`.

Hetero atoms matching `"hetero and noh"` selection are represented by circles and unique colors are picked at random on a per residue basis. Colors can be customized using residue name as in `NAH='purple'`. Note that this will color all distinct residues with the same name in the same color. Hetero atom marker and size can be changed using *hmarker* and *hsize* keywords, default values are `hmarker='o', hsize=6`.

ProDy will set the size of axis so the representation is not distorted when the shape of figure window is close to a square. Colors are picked at random, except for water oxygens which will always be colored red.

*** Interactive 3D Rendering in Jupyter Notebook ***

If py3Dmol has been imported then it will be used instead to display an interactive viewer. See *view3D()* (page 236)

3.10.20 PDB File Header

This module defines functions for parsing header data from PDB files.

class Chemical (*resname*)

A data structure for storing information on chemical components (or heterogens) in PDB structures.

A *Chemical* (page 236) instance has the following attributes:

Attribute	Type	Description (RECORD TYPE)
resname	str	residue name (or chemical component identifier) (HET)
name	str	chemical name (HETNAM)
chain	str	chain identifier (HET)
resnum	int	residue (or sequence) number (HET)
icode	str	insertion code (HET)
natoms	int	number of atoms present in the structure (HET)
description	str	description of the chemical component (HET)
synonyms	list	synonyms (HETSYN)
formula	str	chemical formula (FORMUL)
pdbentry	str	PDB entry that chemical data is extracted from

Chemical class instances can be obtained as follows:

```
In [1]: from prody import *

In [2]: chemical = parsePDBHeader('1zz2', 'chemicals')[0]

In [3]: chemical
Out[3]: <Chemical: B11 (1ZZ2_A_362)>

In [4]: chemical.name
Out[4]: 'N-[3-(4-FLUOROPHENOXY)PHENYL]-4-[(2-HYDROXYBENZYL)AMINO]PIPERIDINE-1-SULFONAMIDE'

In [5]: chemical.natoms
Out[5]: 33

In [6]: len(chemical)
Out[6]: 33
```

chain
chain identifier

description
description of the chemical component

formula
chemical formula

icode
insertion code

name
chemical name

natoms
number of atoms present in the structure

pdbentry
PDB entry that chemical data is extracted from

resname
residue name (or chemical component identifier)

resnum
residue (or sequence) number

synonyms
list of synonyms

class Polymer(*chid*)
A data structure for storing information on polymer components (protein or nucleic) of PDB structures.

A *Polymer* (page 238) instance has the following attributes:

Attribute	Type	Description (RECORD TYPE)
chid	str	chain identifier
name	str	name of the polymer (macromolecule) (COMPND)
fragment	str	specifies a domain or region of the molecule (COMPND)
synonyms	list	synonyms for the polymer (COMPND)
ec	list	associated Enzyme Commission numbers (COMPND)
engineered	bool	indicates that the polymer was produced using recombinant technology or by purely chemical synthesis (COMPND)
mutation	bool	indicates presence of a mutation (COMPND)
comments	str	additional comments
sequence	str	polymer chain sequence (SEQRES)
dbrefs	list	sequence database records (DBREF[1|2] and SEQADV), see *DBRef* (page 239)
modified	list	modified residues (MODRES) when modified residues are present, each will be represented as: `(resname, chid, resnum, icode, stdname, comment)`
pdbentry	str	PDB entry that polymer data is extracted from

Polymer class instances can be obtained as follows:

```
In [1]: polymer = parsePDBHeader('2k39', 'polymers')[0]

In [2]: polymer
Out[2]: <Polymer: UBIQUITIN (2K39_A)>

In [3]: polymer.pdbentry
Out[3]: '2K39'

In [4]: polymer.chid
Out[4]: 'A'

In [5]: polymer.name
```

```
Out[5]: 'UBIQUITIN'

In [6]: polymer.sequence
Out[6]: 'MQIFVKTLTGKTITLEVEPSDTIENVKAKIQDKEGIPPDQQRLIFAGKQLEDGRTLSDYNIQKESTLHLVLRLRGG'

In [7]: len(polymer.sequence)
Out[7]: 76

In [8]: len(polymer)
Out[8]: 76

In [9]: dbref = polymer.dbrefs[0]

In [10]: dbref.database
Out[10]: 'UniProt'

In [11]: dbref.accession
Out[11]: 'P62972'

In [12]: dbref.idcode
Out[12]: 'UBIQ_XENLA'
```

chid
chain identifier

comments
additional comments

dbrefs
sequence database reference records

ec
list of associated Enzyme Commission numbers

engineered
indicates that the molecule was produced using recombinant technology or by purely chemical synthesis

fragment
specifies a domain or region of the molecule

modified
modified residues

mutation
indicates presence of a mutation

name
name of the polymer (macromolecule)

pdbentry
PDB entry that polymer data is extracted from

sequence
polymer chain sequence

synonyms
list of synonyms for the molecule

class **DBRef**

A data structure for storing reference to sequence databases for polymer components in PDB structures. Information if parsed from **DBREF[1|2]** and **SEQADV** records in PDB header.

accession
database accession code

database
sequence database, one of UniProt, GenBank, Norine, UNIMES, or PDB

dbabbr
database abbreviation, one of UNP, GB, NORINE, UNIMES, or PDB

diff
list of differences between PDB and database sequences, `(resname, resnum, icode, dbResname, dbResnum, comment)`

first
initial residue numbers, `(resnum, icode, dbnum)`

idcode
database identification code, i.e. entry name in UniProt

last
ending residue numbers, `(resnum, icode, dbnum)`

parsePDBHeader(*pdb*, **keys*)
Returns header data dictionary for *pdb*. This function is equivalent to `parsePDB(pdb, header=True, model=0, meta=False)`, likewise *pdb* may be an identifier or a filename.

List of header records that are parsed.

Record type	Dictionary key(s)	Description
HEADER	classification deposition_date identifier	molecule classification deposition date PDB identifier
TITLE	title	title for the experiment or analysis
SPLIT	split	list of PDB entries that make up the whole structure when combined with this one
COMPND	polymers	see *Polymer* (page 238)
EXPDTA	experiment	information about the experiment
NUMMDL	n_models	number of models
MDLTYP	model_type	additional structural annotation
AUTHOR	authors	list of contributors
JRNL	reference	**reference information dictionary:** • *authors*: list of authors • *title*: title of the article • *editors*: list of editors • *issn*: • *reference*: journal, vol, issue, etc. • *publisher*: publisher information • *pmid*: pubmed identifier • *doi*: digital object identifier
DBREF[1|2]	polymers	see *Polymer* (page 238) and *DBRef* (page 239)
SEQADV	polymers	see *Polymer* (page 238)
SEQRES	polymers	see *Polymer* (page 238)
MODRES	polymers	see *Polymer* (page 238)
HELIX	polymers	see *Polymer* (page 238)
SHEET	polymers	see *Polymer* (page 238)
HET	chemicals	see *Chemical* (page 236)
HETNAM	chemicals	see *Chemical* (page 236)
HETSYN	chemicals	see *Chemical* (page 236)
FORMUL	chemicals	see *Chemical* (page 236)
REMARK 2	resolution	resolution of structures, when applicable
REMARK 4	version	PDB file version
REMARK 350	biomoltrans	biomolecular transformation lines (unprocessed)
REMARK 900	related_entries	related entries in the PDB or EMDB

Header records that are not parsed are: OBSLTE, CAVEAT, SOURCE, KEYWDS, REVDAT, SPRSDE,

SSBOND, LINK, CISPEP, CRYST1, ORIGX1, ORIGX2, ORIGX3, MTRIX1, MTRIX2, MTRIX3, and REMARK X not mentioned above.

assignSecstr(*header*, *atoms*, *coil=True*)

Assign secondary structure from *header* dictionary to *atoms*. *header* must be a dictionary parsed using the *parsePDB()* (page 268). *atoms* may be an instance of *AtomGroup* (page 38), *Selection* (page 100), *Chain* (page 53) or *Residue* (page 80). ProDy can be configured to automatically parse and assign secondary structure information using `confProDy(auto_secondary=True)` command. See also *confProDy()* (page 329) function.

The Dictionary of Protein Secondary Structure, in short DSSP, type single letter code assignments are used:

- **G** = 3-turn helix (310 helix). Min length 3 residues.
- **H** = 4-turn helix (alpha helix). Min length 4 residues.
- **I** = 5-turn helix (pi helix). Min length 5 residues.
- **T** = hydrogen bonded turn (3, 4 or 5 turn)
- **E** = extended strand in parallel and/or anti-parallel beta-sheet conformation. Min length 2 residues.
- **B** = residue in isolated beta-bridge (single pair beta-sheet hydrogen bond formation)
- **S** = bend (the only non-hydrogen-bond based assignment).
- **C** = residues not in one of above conformations.

See http://en.wikipedia.org/wiki/Protein_secondary_structure#The_DSSP_code for more details.

Following PDB helix classes are omitted:

- Right-handed omega (2, class number)
- Right-handed gamma (4)
- Left-handed alpha (6)
- Left-handed omega (7)
- Left-handed gamma (8)
- 2 - 7 ribbon/helix (9)
- Polyproline (10)

Secondary structures are assigned to all atoms in a residue. Amino acid residues without any secondary structure assignments in the header section will be assigned coil (C) conformation. This can be prevented by passing `coil=False` argument.

buildBiomolecules(*header*, *atoms*, *biomol=None*)

Returns *atoms* after applying biomolecular transformations from *header* dictionary. Biomolecular transformations are applied to all coordinate sets in the molecule.

Some PDB files contain transformations for more than 1 biomolecules. A specific set of transformations can be choosen using *biomol* argument. Transformation sets are identified by numbers, e.g. `"1"`, `"2"`, ...

If multiple biomolecular transformations are provided in the *header* dictionary, biomolecules will be returned as *AtomGroup* (page 38) instances in a `list()`.

If the resulting biomolecule has more than 26 chains, the molecular assembly will be split into multiple *AtomGroup* (page 38) instances each containing at most 26 chains. These *AtomGroup* (page 38) instances will be returned in a tuple.

Note that atoms in biomolecules are ordered according to chain identifiers. When multiple chains in a biomolecule have the same chain identifier, they are given different segment names to distinguish them.

3.10.21 Interactions and Stability (InSty)

This module called InSty defines functions for calculating different types of interactions in protein structure, between proteins or between protein and ligand. The following interactions are available for protein interactions:

1. Hydrogen bonds
2. Salt Bridges
3. Repulsive Ionic Bonding
4. Pi stacking interactions
5. Pi-cation interactions
6. Hydrophobic interactions
7. Disulfide Bonds

class **Interactions**(*title='Unknown'*)

Class for Interaction analysis of proteins.

buildInteractionMatrix(***kwargs*)

Build matrix with protein interactions. Interactions are counted as follows:

1. Hydrogen bonds (HBs) +1
2. Salt Bridges (SBs) +1 (Salt bridges might be included in hydrogen bonds)
3. Repulsive Ionic Bonding (RIB) +1
4. Pi stacking interactions (PiStack) +1
5. Pi-cation interactions (PiCat) +1
6. Hydrophobic interactions (HPh) +1
7. Disulfide bonds (DiBs) +1

Some type of interactions could be removed from the analysis by replacing value 1 by 0. Example: >>> interactions = Interactions() >>> atoms = parsePDB(PDBfile).select('protein') >>> interactions.calcProteinInteractions(atoms) >>> interactions.buildInteractionMatrix(RIB=0, HBs=0, HPh=0)

Parameters

- **HBs** (*int, float*) – score per single hydrogen bond
- **SBs** (*int, float*) – score per single salt bridge
- **RIB** (*int, float*) – score per single repulsive ionic bonding
- **PiStack** (*int, float*) – score per pi-stacking interaction
- **PiCat** (*int, float*) – score per pi-cation interaction
- **HPh** (*int, float*) – score per hydrophobic interaction
- **DiBs** (*int, float*) – score per disulfide bond

calcProteinInteractions(*atoms*, ***kwargs*)

Compute all protein interactions (shown below) using default parameters.

1. Hydrogen bonds
2. Salt Bridges
3. RepulsiveIonicBonding
4. Pi stacking interactions
5. Pi-cation interactions
6. Hydrophobic interactions
7. Disulfide Bonds

Parameters **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected

getAtoms()
Returns associated atoms.

getDisulfideBonds(***kwargs*)
Returns the list of disulfide bonds.

Parameters

- **selection** (*str*[633]) – selection string
- **selection2** (*str*[634]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getFrequentInteractors(*contacts_min=3*)
Provide a list of residues with the most frequent interactions based on the following interactions:

1.Hydrogen bonds (hb)

2.Salt Bridges (sb)

3.Repulsive Ionic Bonding (rb)

4.Pi stacking interactions (ps)

5.Pi-cation interactions (pc)

6.Hydrophobic interactions (hp)

7.Disulfide bonds (disb)

Parameters **contacts_min** (*int*[635]) – Minimal number of contacts which residue may form with other residues, by default 3.

getHydrogenBonds(***kwargs*)
Returns the list of hydrogen bonds.

Parameters

[633] http://docs.python.org/library/stdtypes.html#str
[634] http://docs.python.org/library/stdtypes.html#str
[635] http://docs.python.org/library/functions.html#int

- **selection** (*str*[636]) – selection string
- **selection2** (*str*[637]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getHydrophobic(***kwargs*)

Returns the list of hydrophobic interactions.

Parameters

- **selection** (*str*[638]) – selection string
- **selection2** (*str*[639]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getInteractions(***kwargs*)

Returns the list of all interactions.

Parameters

- **selection** (*str*[640]) – selection string
- **selection2** (*str*[641]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getPiCation(***kwargs*)

Returns the list of Pi-cation interactions.

Parameters

- **selection** (*str*[642]) – selection string
- **selection2** (*str*[643]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

[636] http://docs.python.org/library/stdtypes.html#str
[637] http://docs.python.org/library/stdtypes.html#str
[638] http://docs.python.org/library/stdtypes.html#str
[639] http://docs.python.org/library/stdtypes.html#str
[640] http://docs.python.org/library/stdtypes.html#str
[641] http://docs.python.org/library/stdtypes.html#str
[642] http://docs.python.org/library/stdtypes.html#str
[643] http://docs.python.org/library/stdtypes.html#str

getPiStacking(***kwargs*)
Returns the list of Pi-stacking interactions.

Parameters

- **selection** (*str*[644]) – selection string
- **selection2** (*str*[645]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getRepulsiveIonicBonding(***kwargs*)
Returns the list of repulsive ionic bonding.

Parameters

- **selection** (*str*[646]) – selection string
- **selection2** (*str*[647]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getSaltBridges(***kwargs*)
Returns the list of salt bridges.

Parameters

- **selection** (*str*[648]) – selection string
- **selection2** (*str*[649]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

saveInteractionsPDB(***kwargs*)
Save the number of potential interactions to PDB file in occupancy column.

Parameters **filename** (*str*[650]) – name of the PDB file which will be saved for visualization, it will contain the results in occupancy column.

setNewDisulfideBonds(*interaction*)
Replace default calculation of hydrophobic interactions by the one provided by user.

setNewHydrogenBonds(*interaction*)
Replace default calculation of hydrogen bonds by the one provided by user.

[644] http://docs.python.org/library/stdtypes.html#str
[645] http://docs.python.org/library/stdtypes.html#str
[646] http://docs.python.org/library/stdtypes.html#str
[647] http://docs.python.org/library/stdtypes.html#str
[648] http://docs.python.org/library/stdtypes.html#str
[649] http://docs.python.org/library/stdtypes.html#str
[650] http://docs.python.org/library/stdtypes.html#str

setNewHydrophobic(*interaction*)
Replace default calculation of hydrophobic interactions by the one provided by user.

setNewPiCation(*interaction*)
Replace default calculation of pi-cation interactions by the one provided by user.

setNewPiStacking(*interaction*)
Replace default calculation of pi-stacking interactions by the one provided by user.

setNewRepulsiveIonicBonding(*interaction*)
Replace default calculation of repulsive ionic bonding by the one provided by user.

setNewSaltBridges(*interaction*)
Replace default calculation of salt bridges by the one provided by user.

setTitle(*title*)
Set title of the model.

showCumulativeInteractionTypes(***kwargs*)
Bar plot with the number of potential inetractions per residue. Particular type of interactions can be excluded by using keywords HBs=0, RIB=0, etc.

Parameters

- **HBs** (*int, float*) – score per single hydrogen bond
- **SBs** (*int, float*) – score per single salt bridge
- **RIB** (*int, float*) – score per single repulsive ionic bonding
- **PiStack** (*int, float*) – score per pi-stacking interaction
- **PiCat** (*int, float*) – score per pi-cation interaction
- **HPh** (*int, float*) – score per hydrophobic interaction
- **DiBs** (*int, float*) – score per disulfide bond

showFrequentInteractors(*cutoff=5*, ***kwargs*)
Plots regions with the most frequent interactions.

Parameters **cutoff** (*int, float*) – minimal score per residue which will be displayed. If cutoff value is to big, top 30% with the higest values will be returned. Default is 5.

Nonstandard resiudes can be updated in a following way: d = {'CYX': 'X', 'CEA': 'Z'} >>> name.showFrequentInteractors(d)

showInteractors(***kwargs*)
Display protein residues and their number of potential interactions with other residues from protein structure.

class InteractionsTrajectory(*name='Unknown'*)
Class for Interaction analysis of DCD trajectory or multi-model PDB (Ensemble PDB).

calcProteinInteractionsTrajectory(*atoms*, *trajectory=None*, *filename=None*, ***kwargs*)

Compute all protein interactions (shown below) for DCD trajectory or multi-model PDB
using default parameters. (1) Hydrogen bonds (2) Salt Bridges (3) RepulsiveIonicBonding (4) Pi stacking interactions (5) Pi-cation interactions (6) Hydrophobic interactions (7) Disulfide Bonds

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected

- **trajectory** (class:.*Trajectory*) – trajectory file
- **filename** (*pkl*) – Name of pkl filename in which interactions will be storage
- **selection** (*str*[651]) – selection string
- **selection2** (*str*[652]) – selection string
- **start_frame** (*int*[653]) – index of first frame to read
- **stop_frame** (*int*[654]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getAtoms()
Returns associated atoms.

getDisulfideBonds(**kwargs*)
Return the list of disulfide bonds computed from DCD trajectory.

Parameters

- **selection** (*str*[655]) – selection string
- **selection2** (*str*[656]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getHydrogenBonds(**kwargs*)
Return the list of hydrogen bonds computed from DCD trajectory.

Parameters

- **selection** (*str*[657]) – selection string
- **selection2** (*str*[658]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getHydrophobic(**kwargs*)
Return the list of hydrophobic interactions computed from DCD trajectory.

Parameters

[651] http://docs.python.org/library/stdtypes.html#str
[652] http://docs.python.org/library/stdtypes.html#str
[653] http://docs.python.org/library/functions.html#int
[654] http://docs.python.org/library/functions.html#int
[655] http://docs.python.org/library/stdtypes.html#str
[656] http://docs.python.org/library/stdtypes.html#str
[657] http://docs.python.org/library/stdtypes.html#str
[658] http://docs.python.org/library/stdtypes.html#str

- **selection** (*str*[659]) – selection string
- **selection2** (*str*[660]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getInteractions(***kwargs*)
Return the list of all interactions.

Parameters

- **selection** (*str*[661]) – selection string
- **selection2** (*str*[662]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getInteractionsNumber()
Return the number of interactions in each frame.

getPiCation(***kwargs*)
Return the list of Pi-cation interactions computed from DCD trajectory.

Parameters

- **selection** (*str*[663]) – selection string
- **selection2** (*str*[664]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getPiStacking(***kwargs*)
Return the list of Pi-stacking interactions computed from DCD trajectory.

Parameters

- **selection** (*str*[665]) – selection string
- **selection2** (*str*[666]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

[659] http://docs.python.org/library/stdtypes.html#str
[660] http://docs.python.org/library/stdtypes.html#str
[661] http://docs.python.org/library/stdtypes.html#str
[662] http://docs.python.org/library/stdtypes.html#str
[663] http://docs.python.org/library/stdtypes.html#str
[664] http://docs.python.org/library/stdtypes.html#str
[665] http://docs.python.org/library/stdtypes.html#str
[666] http://docs.python.org/library/stdtypes.html#str

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getRepulsiveIonicBonding(***kwargs*)
Return the list of repulsive ionic bonding computed from DCD trajectory.

Parameters

- **selection** (*str*[667]) – selection string
- **selection2** (*str*[668]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getSaltBridges(***kwargs*)
Return the list of salt bridges computed from DCD trajectory.

Parameters

- **selection** (*str*[669]) – selection string
- **selection2** (*str*[670]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

getTimeInteractions(*filename=None*, ***kwargs*)
Return a bar plots with the number of interactions per each frame.

Parameters **filename** (*str*[671]) – PNG file name

parseInteractions(*filename*)
Import interactions from analysis of trajectory which was saved via calcProteinInteractionsTrajectory().

Parameters **filename** (*pkl*) – Name of pkl file in which interactions will be storage

setNewDisulfideBondsTrajectory(*interaction*)
Replace default calculation of disulfide bonds by the one provided by user.

setNewHydrogenBondsTrajectory(*interaction*)
Replace default calculation of hydrogen bonds by the one provided by user.

setNewHydrophobicTrajectory(*interaction*)
Replace default calculation of hydrophobic interactions by the one provided by user.

setNewPiCationTrajectory(*interaction*)
Replace default calculation of pi-cation interactions by the one provided by user.

setNewPiStackingTrajectory(*interaction*)
Replace default calculation of pi-stacking interactions by the one provided by user.

[667] http://docs.python.org/library/stdtypes.html#str
[668] http://docs.python.org/library/stdtypes.html#str
[669] http://docs.python.org/library/stdtypes.html#str
[670] http://docs.python.org/library/stdtypes.html#str
[671] http://docs.python.org/library/stdtypes.html#str

setNewRepulsiveIonicBondingTrajectory(*interaction*)
Replace default calculation of repulsive ionic bonding by the one provided by user.

setNewSaltBridgesTrajectory(*interaction*)
Replace default calculation of salt bridges by the one provided by user.

class **LigandInteractionsTrajectory**(*name='Unknown'*)
Class for protein-ligand interaction analysis of DCD trajectory or multi-model PDB (Ensemble PDB). This class is using PLIP to provide the interactions. Install PLIP before using it.

Instalation of PLIP using conda: >>> conda install -c conda-forge plip ## https://anaconda.org/conda-forge/plip # https://github.com/pharmai/plip/blob/master/DOCUMENTATION.md

Instalation using PIP: >>> pip install plip

PLIP: fully automated protein–ligand interaction profiler *Nucl. Acids Res.* **2015** 43:W443-W447.

calcFrequentInteractors(***kwargs*)
Returns a dictonary with residues involved in the interaction with ligand and their number of counts.

Parameters **selection** (*str*[672]) – selection string of ligand with chain ID e.g. "MESA" where MES is ligand resname and A is chain ID. Selection pointed as None will return all interactions together without ligands separation.

calcLigandInteractionsTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute protein-ligand interactions for DCD trajectory or multi-model PDB using PLIP library.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **filename** (*pkl*) – Name of pkl filename in which interactions will be storage
- **start_frame** (*int*[673]) – index of first frame to read
- **stop_frame** (*int*[674]) – index of last frame to read
- **output** (*bool*[675]) – parameter to print the interactions on the screen while analyzing the structure, use 'info'.

getAtoms()
Returns associated atoms.

getInteractionTypes()
Show which interaction types were detected for ligands.

getLigandInteractions(***kwargs*)
Return the list of protein-ligand interactions.

Parameters

- **filters** (*str*[676]) – selection string of ligand with chain ID or interaction type e.g. 'SBs' (HBs, SBs, HPh, PiStack, PiCat, HPh, watBridge)

[672] http://docs.python.org/library/stdtypes.html#str
[673] http://docs.python.org/library/functions.html#int
[674] http://docs.python.org/library/functions.html#int
[675] http://docs.python.org/library/functions.html#bool
[676] http://docs.python.org/library/stdtypes.html#str

- **include_frames** (*bool*[677]) – used with filters, it will leave selected keyword in orginal lists, if False it will collect selected interactions in one list, Use True to assign new selection using setLigandInteractions. by default True

getLigandInteractionsNumber(***kwargs*)
Return the number of interactions per each frame. Number of interactions can be a total number of interactions or it can be divided into interaction types.

Parameters **types** (*bool*[678]) – Interaction types can be included (True) or not (False). by default is True.

getLigandsNames()
Show which ligands are in a system.

parseLigandInteractions(*filename*)
Import interactions from analysis of trajectory which was saved via calcLigandInteractionsTrajectory().

Parameters **filename** (*pkl*) – Name of pkl file in which interactions will be storage

saveInteractionsPDB(***kwargs*)
Save the number of interactions with ligand to PDB file in occupancy column It will recognize the chains. If the system will contain one chain and many segments the PDB file will not be created in a correct way.

Parameters

- **filename** (*str*[679]) – name of the PDB file which will be saved for visualization, it will contain the results in occupancy column.
- **ligand_sele** (*str*[680]) – ligand selection, by default is 'all not (protein or water or ion)'.

setLigandInteractions(*atoms*, *interaction*)
Replace protein-ligand interactions for example byb using getLigandInteractions() with filters to select particular ligand.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **interactions** (*list*[681]) – list of interactions

calcHydrogenBonds(*atoms*, ***kwargs*)
Compute hydrogen bonds for proteins and other molecules.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distDA** (*int, float*) – non-zero value, maximal distance between donor and acceptor. default is 3.5 distA also works
- **angleDHA** (*int, float*) – non-zero value, maximal (180 - D-H-A angle) (donor, hydrogen, acceptor). default is 40. angle also works

[677] http://docs.python.org/library/functions.html#bool
[678] http://docs.python.org/library/functions.html#bool
[679] http://docs.python.org/library/stdtypes.html#str
[680] http://docs.python.org/library/stdtypes.html#str
[681] http://docs.python.org/library/stdtypes.html#list

- **seq_cutoff_HB** (*int*[682]) – non-zero value, interactions will be found between atoms with index differences that are higher than seq_cutoff_HB. default is 25 atoms. seq_cutoff also works
- **donors** (*list*[683]) – which atoms to count as donors default is ['N', 'O', 'S', 'F']
- **acceptors** (*list*[684]) – which atoms to count as acceptors default is ['N', 'O', 'S', 'F']
- **selection** (*str*[685]) – selection string
- **selection2** (*str*[686]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

Structure should contain hydrogens. If not they can be added using addMissingAtoms(pdb_name) function available in ProDy after Openbabel installation. *conda install -c conda-forge openbabel*

Note that the angle which it is considering is 180-defined angle D-H-A (in a good agreement with VMD) Results can be displayed in VMD by using showVMDinteraction()

calcChHydrogenBonds(*atoms*, ***kwargs*)

Finds hydrogen bonds between different chains. See more details in calcHydrogenBonds().

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distA** (*int, float*) – non-zero value, maximal distance between donor and acceptor. default is 3.5
- **angle** (*int, float*) – non-zero value, D-H-A angle (donor, hydrogen, acceptor). default is 40.
- **seq_cutoff** (*int*[687]) – non-zero value, interactions will be found between atoms with index differences that are higher than seq_cutoff. default is 25 atoms.

Structure should contain hydrogens. If not they can be added using addMissingAtoms(pdb_name) function available in ProDy after Openbabel installation. *conda install -c conda-forge openbabel*

Note that the angle which it is considering is 180-defined angle D-H-A (in a good agreement with VMD) Results can be displayed in VMD.

calcSaltBridges(*atoms*, ***kwargs*)

Finds salt bridges in protein structure. Histidines are considered as charge residues.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distSB** (*int, float*) – non-zero value, maximal distance between center of masses of N and O atoms of negatively and positevely charged residues. default is 5. distA also works

[682] http://docs.python.org/library/functions.html#int
[683] http://docs.python.org/library/stdtypes.html#list
[684] http://docs.python.org/library/stdtypes.html#list
[685] http://docs.python.org/library/stdtypes.html#str
[686] http://docs.python.org/library/stdtypes.html#str
[687] http://docs.python.org/library/functions.html#int

- **selection** (*str*[688]) – selection string
- **selection2** (*str*[689]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

Results can be displayed in VMD.

calcRepulsiveIonicBonding(*atoms*, ***kwargs*)

Finds repulsive ionic bonding in protein structure i.e. between positive-positive or negative-negative residues. Histidine is not considered as a charged residue.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distRB** (*int, float*) – non-zero value, maximal distance between center of masses between N-N or O-O atoms of residues. default is 4.5. distA works too
- **selection** (*str*[690]) – selection string
- **selection2** (*str*[691]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

Results can be displayed in VMD.

calcPiStacking(*atoms*, ***kwargs*)

Finds π–π stacking interactions (between aromatic rings).

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distPS** (*int, float*) – non-zero value, maximal distance between center of masses of residues aromatic rings. default is 5. distA works too
- **angle_min_PS** (*int, float*) – minimal angle between aromatic rings. default is 0. angle_min works too
- **angle_max_PS** (*int, float*) – maximal angle between rings. default is 360. angle_max works too
- **selection** (*str*[692]) – selection string
- **selection2** (*str*[693]) – selection string
- **non_standard_PS** (*dict*[694]) – dictionary of non-standard residue in the protein structure that need to be included in calculations non_standard works too

[688] http://docs.python.org/library/stdtypes.html#str
[689] http://docs.python.org/library/stdtypes.html#str
[690] http://docs.python.org/library/stdtypes.html#str
[691] http://docs.python.org/library/stdtypes.html#str
[692] http://docs.python.org/library/stdtypes.html#str
[693] http://docs.python.org/library/stdtypes.html#str
[694] http://docs.python.org/library/stdtypes.html#dict

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

Results can be displayed in VMD. By default three residues are included TRP, PHE, TYR and HIS. Additional selection can be added:

```
>>> non_standard = {"HSE": "noh and not backbone and not name CB",
            "HSD": "noh and not backbone and not name CB"}
>>> calcPiStacking(atoms, non_standard)
```

Predictions for proteins only. To compute protein-ligand interactions use calcLigandInteractions() or define **kwargs.

calcPiCation(*atoms*, ***kwargs*)

Finds cation-Pi interaction i.e. between aromatic ring and positively charged residue (ARG and LYS).

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distPC** (*int, float*) – non-zero value, maximal distance between center of masses of aromatic ring and positively charge group. default is 5. distA works too
- **selection** (*str*[695]) – selection string
- **selection2** (*str*[696]) – selection string
- **non_standard_PC** (*dict*[697]) – dictionary of non-standard residue in the protein structure that need to be included in calculations non_standard also works

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

By default three residues are included TRP, PHE, TYR and HIS. Additional selection can be added:

```
>>> calcPiCation(atoms, 'HSE'='noh and not backbone and not name CB')
or
>>> non_standard = {"HSE": "noh and not backbone and not name CB",
        "HSD": "noh and not backbone and not name CB"}
>>> calcPiCation(atoms, non_standard)
```

Results can be displayed in VMD. Predictions for proteins only. To compute protein-ligand interactions use calcLigandInteractions() or define **kwargs

calcHydrophobic(*atoms*, ***kwargs*)

Prediction of hydrophobic interactions between hydrophobic residues (ALA, ILE, LEU, MET, PHE, TRP, VAL, CG of ARG, and CG and CD of LYS).

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected

[695] http://docs.python.org/library/stdtypes.html#str
[696] http://docs.python.org/library/stdtypes.html#str
[697] http://docs.python.org/library/stdtypes.html#dict

- **distHPh** (*int, float*) – non-zero value, maximal distance between atoms of hydrophobic residues. default is 4.5. distA works too
- **non_standard_Hph** (*dict*[698]) – dictionary of non-standard residue in the protein structure that need to be included in calculations non_standard works too
- **zerosHPh** (*bool*[699]) – zero values of hydrophobic overlaping areas included default is False

Last value in the output corresponds to the total hydrophobic overlaping area for two residues not only for the atoms that are included in the list. Atoms that which are listed are the closest between two residues and they will be inluded to draw the line in VMD[700].

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

Additional selection can be added as shown below (with selection that includes only hydrophobic part):

```
>>> calcHydrophobic(atoms, non_standard_Hph={'XLE'='noh and not backbone',
                                             'XLI'='noh and not backbone'})
```

Predictions for proteins only. To compute protein-ligand interactions use calcLigandInteractions(). Results can be displayed in VMD by using showVMDinteraction()

Note that interactions between aromatic residues are omitted becasue they are provided by calcPiStacking().

calcDisulfideBonds(*atoms*, ***kwargs*)
Prediction of disulfide bonds.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distDB** (*int, float*) – non-zero value, maximal distance between atoms of hydrophobic residues. default is 3. distA works too

calcMetalInteractions(*atoms, distA=3.0, extraIons=['FE'], excluded_ions=['SOD', 'CLA']*)
Interactions with metal ions (includes water, ligands and other ions).

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **distA** (*int, float*) – non-zero value, maximal distance between ion and residue. default is 3.0
- **extraIons** (*list*[701]) – ions to be included in the analysis.
- **excluded_ions** (*list*[702]) – ions which should be excluded from the analysis.

calcHydrogenBondsTrajectory(*atoms*, *trajectory=None*, ***kwargs*)
Compute hydrogen bonds for DCD trajectory or multi-model PDB using default parameters.

Parameters

[698] http://docs.python.org/library/stdtypes.html#dict
[699] http://docs.python.org/library/functions.html#bool
[700] http://www.ks.uiuc.edu/Research/vmd
[701] http://docs.python.org/library/stdtypes.html#list
[702] http://docs.python.org/library/stdtypes.html#list

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between donor and acceptor. default is 3.5
- **angle** (*int, float*) – non-zero value, maximal (180 - D-H-A angle) (donor, hydrogen, acceptor). default is 40.
- **seq_cutoff** (*int*[703]) – non-zero value, interactions will be found between atoms with index differences that are higher than seq_cutoff. default is 20 atoms.
- **selection** (*str*[704]) – selection string
- **selection2** (*str*[705]) – selection string
- **start_frame** (*int*[706]) – index of first frame to read
- **stop_frame** (*int*[707]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcSaltBridgesTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute salt bridges for DCD trajectory or multi-model PDB using default parameters.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between center of masses of N and O atoms of negatively and positevely charged residues. default is 5.
- **selection** (*str*[708]) – selection string
- **selection2** (*str*[709]) – selection string
- **start_frame** (*int*[710]) – index of first frame to read
- **stop_frame** (*int*[711]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcRepulsiveIonicBondingTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute repulsive ionic bonding for DCD trajectory or multi-model PDB using default parameters.

[703] http://docs.python.org/library/functions.html#int
[704] http://docs.python.org/library/stdtypes.html#str
[705] http://docs.python.org/library/stdtypes.html#str
[706] http://docs.python.org/library/functions.html#int
[707] http://docs.python.org/library/functions.html#int
[708] http://docs.python.org/library/stdtypes.html#str
[709] http://docs.python.org/library/stdtypes.html#str
[710] http://docs.python.org/library/functions.html#int
[711] http://docs.python.org/library/functions.html#int

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between center of masses between N-N or O-O atoms of residues. default is 4.5.
- **selection** (*str*[712]) – selection string
- **selection2** (*str*[713]) – selection string
- **start_frame** (*int*[714]) – index of first frame to read
- **stop_frame** (*int*[715]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcPiStackingTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute Pi-stacking interactions for DCD trajectory or multi-model PDB using default parameters.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between center of masses of residues aromatic rings. default is 5.
- **angle_min** (*int*[716]) – minimal angle between aromatic rings. default is 0.
- **angle_max** (*int, float*) – maximal angle between rings. default is 360.
- **selection** (*str*[717]) – selection string
- **selection2** (*str*[718]) – selection string
- **start_frame** (*int*[719]) – index of first frame to read
- **stop_frame** (*int*[720]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcPiCationTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute Pi-cation interactions for DCD trajectory or multi-model PDB using default parameters.

Parameters

[712] http://docs.python.org/library/stdtypes.html#str
[713] http://docs.python.org/library/stdtypes.html#str
[714] http://docs.python.org/library/functions.html#int
[715] http://docs.python.org/library/functions.html#int
[716] http://docs.python.org/library/functions.html#int
[717] http://docs.python.org/library/stdtypes.html#str
[718] http://docs.python.org/library/stdtypes.html#str
[719] http://docs.python.org/library/functions.html#int
[720] http://docs.python.org/library/functions.html#int

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between center of masses of aromatic ring and positively charge group. default is 5.
- **selection** (*str*[721]) – selection string
- **selection2** (*str*[722]) – selection string
- **start_frame** (*int*[723]) – index of first frame to read
- **stop_frame** (*int*[724]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcHydrophobicTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute hydrophobic interactions for DCD trajectory or multi-model PDB using default parameters.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between atoms of hydrophobic residues. default is 4.5.
- **selection** (*str*[725]) – selection string
- **selection2** (*str*[726]) – selection string
- **start_frame** (*int*[727]) – index of first frame to read
- **stop_frame** (*int*[728]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcDisulfideBondsTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute disulfide bonds for DCD trajectory or multi-model PDB using default parameters.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between atoms of hydrophobic residues. default is 2.5.

[721] http://docs.python.org/library/stdtypes.html#str
[722] http://docs.python.org/library/stdtypes.html#str
[723] http://docs.python.org/library/functions.html#int
[724] http://docs.python.org/library/functions.html#int
[725] http://docs.python.org/library/stdtypes.html#str
[726] http://docs.python.org/library/stdtypes.html#str
[727] http://docs.python.org/library/functions.html#int
[728] http://docs.python.org/library/functions.html#int

- **start_frame** (*int*[729]) – index of first frame to read
- **stop_frame** (*int*[730]) – index of last frame to read

calcProteinInteractions(*atoms*, ***kwargs*)

Compute all protein interactions (shown below) using default parameters.

1. Hydrogen bonds
2. Salt Bridges
3. RepulsiveIonicBonding
4. Pi stacking interactions
5. Pi-cation interactions
6. Hydrophobic interactions
7. Disulfide Bonds

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **selection** (*str*[731]) – selection string
- **selection2** (*str*[732]) – selection string

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcStatisticsInteractions(*data*, ***kwargs*)

Return the statistics of interactions from PDB Ensemble or trajectory including: (1) the weight for each residue pair: corresponds to the number of counts divided by the number of frames (values >1 are obtained when residue pair creates multiple contacts); (2) average distance of interactions for each pair [in Ang] and (3) standard deviation [Ang.].

Parameters

- **data** (*list*[733]) – list with interactions from calcHydrogenBondsTrajectory() or other types
- **weight_cutoff** (*int, float*) – value above which results will be displayed 1 or more means that residue contact is present in all conformations/frames default value is 0.2 (in 20% of conformations contact appeared)

Example of usage: >>> atoms = parsePDB('PDBfile.pdb') >>> dcd = Trajectory('DCDfile.dcd') >>> dcd.link(atoms) >>> dcd.setCoords(atoms)

```
>>> data = calcPiCationTrajectory(atoms, dcd, distA=5)
or
>>> interactionsTrajectory = InteractionsTrajectory()
>>> data = interactionsTrajectory.getPiCation()
```

[729] http://docs.python.org/library/functions.html#int
[730] http://docs.python.org/library/functions.html#int
[731] http://docs.python.org/library/stdtypes.html#str
[732] http://docs.python.org/library/stdtypes.html#str
[733] http://docs.python.org/library/stdtypes.html#list

calcDistribution(*interactions*, *residue1*, *residue2=None*, ***kwargs*)
Distributions/histograms of pairs of amino acids. Histograms are normalized.

Parameters

- **interactions** (*list*[734]) – list of interactions
- **residue1** (*str*[735]) – residue name in 3-letter code and residue number
- **residue2** (*str*[736]) – residue name in 3-letter code and residue number
- **metrics** (*str*[737]) – name of the data type 'distance' or 'angle' depends on the type of interaction

calcSASA(*atoms*, ***kwargs*)
Provide information about solvent accessible surface area (SASA) based on the sasa program. To use this function compiled hpb.so is needed.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **selection** (*str*[738]) – selection string by default all the protein structure is used
- **sasa_cutoff** (*float, int*) – cutoff for SASA values default is 0.0
- **resnames** (*bool*[739]) – residues name included default is False

calcVolume(*atoms*, ***kwargs*)
Provide information about volume for each residue/molecule/chain or other selection". To use this function compiled hpb.so is needed.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **selection** (*str*[740]) – selection string by default all the protein structure is used
- **volume_cutoff** – cutoff for volume default is 0.0 to include all the results
- **split_residues** (*bool*[741]) – it will provide values for each residue default is False
- **resnames** (*bool*[742]) – residues name included default is False True - will give residue names and values for each residue False - will give only the values for each residues

compareInteractions(*data1*, *data2*, ***kwargs*)
Comparison of two outputs from interactions. It will provide information about the disappearance and appearance of some interactions as well as the similarities in the interactions for the same system. Two conformations can be compared.

Parameters

- **data1** (*list*[743]) – list with interactions from calcHydrogenBonds() or other types

[734] http://docs.python.org/library/stdtypes.html#list
[735] http://docs.python.org/library/stdtypes.html#str
[736] http://docs.python.org/library/stdtypes.html#str
[737] http://docs.python.org/library/stdtypes.html#str
[738] http://docs.python.org/library/stdtypes.html#str
[739] http://docs.python.org/library/functions.html#bool
[740] http://docs.python.org/library/stdtypes.html#str
[741] http://docs.python.org/library/functions.html#bool
[742] http://docs.python.org/library/functions.html#bool
[743] http://docs.python.org/library/stdtypes.html#list

- **data2** (*list*[744]) – list with interactions from calcHydrogenBonds() or other types
- **filename** – name of text file in which the comparison between two sets of interactions will be saved

type filename: str

Example of usage: >>> atoms1 = parsePDB('PDBfile1.pdb').select('protein') >>> atoms2 = parsePDB('PDBfile2.pdb').select('protein') >>> compareInteractions(calcHydrogenBonds(atoms1), calcHydrogenBonds(atoms2))

showInteractionsGraph(*statistics*, ***kwargs*)

Return residue-residue interactions as graph/network.

Parameters

- **statistics** (*list*[745]) – Results obtained from calcStatisticsInteractions analysis
- **cutoff** (*int, float*) – Minimal number of counts per residue in the trajectory by default 0.1.
- **code** (*str*[746]) – representation of the residues, 3-letter or 1-letter by default 3-letter
- **multiple_chains** (*bool*[747]) – display chain name for structure with many chains by default False
- **edge_cmap** (*str*[748]) – color of the residue connection by default plt.cm.Blues (blue color)
- **node_size** (*int*[749]) – size of the nodes which describes residues by default 300
- **node_distance** (*int*[750]) – value which will scale residue-residue interactions by default 5
- **font_size** (*int*[751]) – size of the font by default 14
- **seed** (*int*[752]) – random number which affect the distribution of residues by default 42

calcLigandInteractions(*atoms*, ***kwargs*)

Provide ligand interactions with other elements of the system including protein, water and ions. Results are computed by PLIP *[SS15]* (page 397) which should be installed. Note that PLIP will not recognize ligand unless it will be HETATM in the PDB file.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **sele** (*str*[753]) – a selection string for residues of interest default is 'all not (water or protein or ion)'
- **ignore_ligs** (*list*[754]) – List of ligands which will be excluded from the analysis.

[744] http://docs.python.org/library/stdtypes.html#list
[745] http://docs.python.org/library/stdtypes.html#list
[746] http://docs.python.org/library/stdtypes.html#str
[747] http://docs.python.org/library/functions.html#bool
[748] http://docs.python.org/library/stdtypes.html#str
[749] http://docs.python.org/library/functions.html#int
[750] http://docs.python.org/library/functions.html#int
[751] http://docs.python.org/library/functions.html#int
[752] http://docs.python.org/library/functions.html#int
[753] http://docs.python.org/library/stdtypes.html#str
[754] http://docs.python.org/library/stdtypes.html#list

To display results as a list of interactions use listLigandInteractions() and for visualization in VMD please use showLigandInteraction_VMD()

Requirements of usage: ## Instalation of Openbabel: >>> conda install -c conda-forge openbabel ## https://anaconda.org/conda-forge/openbabel

Instalation of PLIP >>> conda install -c conda-forge plip ## https://anaconda.org/conda-forge/plip # https://github.com/pharmai/plip/blob/master/DOCUMENTATION.md

PLIP: fully automated protein–ligand interaction profiler *Nucl. Acids Res.* **2015** 43:W443-W447.

listLigandInteractions (*PLIP_output*, ***kwargs*)

Create a list of interactions from PLIP output created using calcLigandInteractions(). Results can be displayed in VMD.

Parameters

- **PLIP_output** (*PLIP object obtained from calcLigandInteractions()*) – Results from PLIP for protein-ligand interactions.
- **output** (*bool*[755]) – parameter to print the interactions on the screen while analyzing the structure (True | False) by default is False

Note that five types of interactions are considered: hydrogen bonds, salt bridges, pi-stacking, cation-pi, hydrophobic and water bridges.

showProteinInteractions_VMD (*atoms*, *interactions*, *color='red'*, ***kwargs*)

Save information about protein interactions to a TCL file (filename) which can be further use in VMD to display all intercations in a graphical interface (in TKConsole: play script_name.tcl). Different types of interactions can be saved separately (color can be selected) or all at once for all types of interactions (hydrogen bonds - blue, salt bridges - yellow, pi stacking - green, cation-pi - orangem, hydrophobic - silver, and disulfide bonds - black).

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **interactions** (*List of lists*) – List of interactions for protein interactions.
- **color** (*str*[756]) – color to draw interactions in VMD, not used only for single interaction type. default **"red"**
- **filename** (*str*[757]) – name of TCL file where interactions will be saved.

Example (hydrogen bonds for protein only): >>> interactions = calcHydrogenBonds(atoms.protein, distA=3.2, angle=30) or all interactions at once: >>> interactions = calcProteinInteractions(atoms.protein)

showLigandInteraction_VMD (*atoms*, *interactions*, ***kwargs*)

Save information from PLIP for ligand-protein interactions in a TCL file which can be further used in VMD to display all intercations in a graphical interface (hydrogen bonds - *blue*, salt bridges - *yellow*, pi stacking - *green*, cation-pi - *orange*, hydrophobic - *silver* and water bridges - *cyan*).

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **interactions** (*list*[758]) – List of interactions lists for protein-ligand interactions.

[755] http://docs.python.org/library/functions.html#bool
[756] http://docs.python.org/library/stdtypes.html#str
[757] http://docs.python.org/library/stdtypes.html#str
[758] http://docs.python.org/library/stdtypes.html#list

- **filename** (*str*[759]) – name of TCL file where interactions will be saved.

To obtain protein-ligand interactions: >>> calculations = calcLigandInteractions(atoms) >>> interactions = listLigandInteractions(calculations)

calcHydrogenBondsTrajectory(*atoms*, *trajectory=None*, ***kwargs*)

Compute hydrogen bonds for DCD trajectory or multi-model PDB using default parameters.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **trajectory** (class:.*Trajectory*) – trajectory file
- **distA** (*int, float*) – non-zero value, maximal distance between donor and acceptor. default is 3.5
- **angle** (*int, float*) – non-zero value, maximal (180 - D-H-A angle) (donor, hydrogen, acceptor). default is 40.
- **seq_cutoff** (*int*[760]) – non-zero value, interactions will be found between atoms with index differences that are higher than seq_cutoff. default is 20 atoms.
- **selection** (*str*[761]) – selection string
- **selection2** (*str*[762]) – selection string
- **start_frame** (*int*[763]) – index of first frame to read
- **stop_frame** (*int*[764]) – index of last frame to read

Selection: If we want to select interactions for the particular residue or group of residues:

selection='chain A and resid 1 to 50'

If we want to study chain-chain interactions: selection='chain A', selection2='chain B'

calcHydrophobicOverlapingAreas(*atoms*, ***kwargs*)

Provide information about hydrophobic contacts between pairs of residues based on the regsurf program. To use this function compiled hpb.so is needed.

Parameters

- **atoms** (*Atomic* (page 47)) – an Atomic object from which residues are selected
- **selection** (*str*[765]) – selection string of hydrophobic residues
- **hpb_cutoff** (*float, int*) – cutoff for hydrophobic overlaping area values default is 0.0
- **cumulative_values** (*'pairs' or 'single_residues'*) – sum of results for pairs of residues or for single residues without distinguishing pairs, default is None

calcSminaBindingAffinity(*atoms*, *trajectory=None*, ***kwargs*)

Computing binding affinity of ligand toward protein structure using SMINA package *[DRK13]* (page 397).

Parameters

[759] http://docs.python.org/library/stdtypes.html#str
[760] http://docs.python.org/library/functions.html#int
[761] http://docs.python.org/library/stdtypes.html#str
[762] http://docs.python.org/library/stdtypes.html#str
[763] http://docs.python.org/library/functions.html#int
[764] http://docs.python.org/library/functions.html#int
[765] http://docs.python.org/library/stdtypes.html#str

- **atoms** (*Atomic* (page 47), *LigandInteractionsTrajectory* (page 251)) – an Atomic object from which residues are selected
- **protein_selection** (*str*[766]) – selection string for the protein and other component of the system that should be included, e.g. "protein and chain A", by default "protein"
- **ligand_selection** (*str*[767]) – selection string for ligand, e.g. "resname ADP", by default "all not protein"
- **ligand_selection** – scoring function (vina or vinardo) by default is "vina"
- **atom_terms** (*bool*[768]) – write per-atom interaction term values. It will provide more information as dictionary for each frame/model, and details for atom terms will be saved in terms_*frame_number*.txt, by default is False

SMINA installation is required to compute ligand binding affinity: >> conda install -c conda-forge smina (for Anaconda)

For more information on SMINA see https://sourceforge.net/projects/smina/. If you benefited from SMINA, please consider citing *[DRK13]* (page 397).

Empirical Scoring with smina from the CSAR 2011 Benchmarking Exercise, *J. Chem. Inf. Model.* **2013** 53: 1893–1904.

calcSminaPerAtomInteractions(*atoms*, *list_terms*)

Computing the summary of per-atom interaction term values using SMINA package *[DRK13]* (page 397). It will return dictionary with per-atom interaction term values.

Parameters

- **atoms** (*Atomic* (page 47), *LigandInteractionsTrajectory* (page 251)) – an Atomic object from which residues are selected
- **list_terms** (*list*[769]) – List of *terms.txt* files obtained from meth:.*calcSminaBindingAffinity* using *atom_terms = True*

Important: First text file in the list should be reference structure which correspond to the provided coordinates as atoms.

calcSminaTermValues(*data*)

Computing weights multiplied by term values, before weighting for each Term. As a results will are obtaining a dictionary.

Parameters **data** (*list*[770]) – List of results provided by Smina using meth:.*calcSminaBindingAffinity* with *atom_terms = True*

showSminaTermValues(*data*)

Display a histogram of weights multiplied by term values, before weighting for each Term. As a results will are obtaining a dictionary.

Parameters **data** (*list*[771]) – List of results provided by Smina using meth:.*calcSminaBindingAffinity* with *atom_terms = True*

[766] http://docs.python.org/library/stdtypes.html#str
[767] http://docs.python.org/library/stdtypes.html#str
[768] http://docs.python.org/library/functions.html#bool
[769] http://docs.python.org/library/stdtypes.html#list
[770] http://docs.python.org/library/stdtypes.html#list
[771] http://docs.python.org/library/stdtypes.html#list

3.10.22 Local PDB Handlers

This module defines functions for handling local PDB folders.

pathPDBFolder (*folder=None*, *divided=False*)

Returns or specify local PDB folder for storing PDB files downloaded from wwPDB[772] servers. Files stored in this folder can be accessed via *fetchPDB()* (page 266) from any working directory. To release the current folder, pass an invalid path, e.g. `folder=''`.

If *divided* is **True**, the divided folder structure of wwPDB servers will be assumed when reading from and writing to the local folder. For example, a structure with identifier **1XYZ** will be present as `pdblocalfolder/yz/pdb1xyz.pdb.gz`.

If *divided* is **False**, a plain folder structure will be expected and adopted when saving files. For example, the same structure will be present as `pdblocalfolder/1xyz.pdb.gz`.

Finally, in either case, lower case letters will be used and compressed files will be stored.

pathPDBMirror (*path=None*, *format=None*)

Returns or specify PDB mirror path to be used by *fetchPDB()* (page 266). To release the current mirror, pass an invalid path, e.g. `path=''`. If you are keeping a partial mirror, such as PDB files in `/data/structures/divided/pdb/` folder, specify *format*, which is `'pdb'` in this case.

fetchPDB (**pdb*, ***kwargs*)

Returns path(s) to PDB file(s) for specified *pdb* identifier(s). Files will be sought in user specified *folder* or current working directory, and then in local PDB folder and mirror, if they are available. If *copy* is set **True**, files will be copied into *folder*. If *compressed* is **False**, all files will be decompressed into *folder*. See *pathPDBFolder()* (page 266) and *pathPDBMirror()* (page 266) for managing local resources, *fetchPDBviaFTP()* (page 277) and *fetchPDBviaHTTP()* (page 278) for downloading files from PDB servers.

fetchPDBfromMirror (**pdb*, ***kwargs*)

Returns path(s) to PDB (default), PDBML, or mmCIF file(s) for specified *pdb* identifier(s). If a *folder* is specified, files will be copied into this folder. If *compressed* is **False**, files will decompressed. *format* argument can be used to get PDBML[773] and mmCIF[774] files: `format='cif'` will fetch an mmCIF file, and `format='xml'` will fetch a PDBML file. If PDBML header file is desired, `noatom=True` argument will do the job.

iterPDBFilenames (*path=None*, *sort=False*, *unique=True*, ***kwargs*)

Yield PDB filenames in *path* specified by the user or in local PDB mirror (see *pathPDBMirror()* (page 266)). When *unique* is **True**, files one of potentially identical files will be yielded (e.g. `1mkp.pdb` and `pdb1mkp.ent.gz1`). `.pdb` and `.ent` extensions, and compressed files are considered.

findPDBFiles (*path*, *case=None*, ***kwargs*)

Returns a dictionary that maps PDB filenames to file paths. If *case* is specified (`'u[pper]'` or `'l[ower]'`), dictionary keys (filenames) will be modified accordingly. If a PDB filename has `pdb` prefix, it will be trimmed, for example `'1mkp'` will be mapped to file path `./pdb1mkp.pdb.gz`). If a file is present with multiple extensions, only one of them will be returned. See also *iterPDBFilenames()* (page 266).

fetchPDBs (**pdb*, ***kwargs*)

"Wrapper function to fetch multiple files from the PDB. If no format is given, it tries PDB then mmCIF then EMD.

Parameters **pdb** – one PDB identifier or filename, or a list of them. If needed, PDB files are downloaded using *fetchPDB()* (page 266) function.

[772] http://www.wwpdb.org/
[773] http://pdbml.pdb.org/
[774] http://mmcif.pdb.org/

3.10.23 PDB Sequence Clusters

This module defines functions for handling PDB sequence clusters.

fetchPDBClusters(*sqid=None*)

Retrieve PDB sequence clusters. PDB sequence clusters are results of the weekly clustering of protein chains in the PDB generated by blastclust. They are available at FTP site: https://cdn.rcsb.org/resources/sequence/clusters/

This function will download about 10 Mb of data and save it after compressing in your home directory in `.prody/pdbclusters`. Compressed files will be less than 4 Mb in size. Cluster data can be loaded using *loadPDBClusters()* (page 267) function and be accessed using *listPDBCluster()* (page 267).

loadPDBClusters(*sqid=None*)

Load previously fetched PDB sequence clusters from disk to memory.

listPDBCluster(*pdb*, *ch*, *sqid=95*)

Returns the PDB sequence cluster that contains chain *ch* in structure *pdb* for sequence identity level *sqid*. PDB sequence cluster will be returned in as a list of tuples, e.g. `[('1XXX', 'A'), ]`. Note that PDB clusters individual chains, so the same PDB identifier may appear twice in the same cluster if the corresponding chain is present in the structure twice.

Before this function is used, *fetchPDBClusters()* (page 267) needs to be called. This function will load the PDB sequence clusters for *sqid* automatically using *loadPDBClusters()* (page 267).

3.10.24 PDB File

This module defines functions for parsing and writing PDB files[775].

parsePDBStream(*stream*, ***kwargs*)

Returns an *AtomGroup* (page 38) and/or dictionary containing header data parsed from a stream of PDB lines.

Parameters

- **stream** – Anything that implements the method `readlines` (e.g. `file`, buffer, stdin)
- **title** (*str*[776]) – title of the *AtomGroup* (page 38) instance, default is the PDB filename or PDB identifier
- **ag** (*AtomGroup* (page 38)) – *AtomGroup* (page 38) instance for storing data parsed from PDB file, number of atoms in *ag* and number of atoms parsed from the PDB file must be the same and atoms in *ag* and those in PDB file must be in the same order. Non-coordinate data stored in *ag* will be overwritten with those parsed from the file.
- **chain** (*str*[777]) – chain identifiers for parsing specific chains, e.g. `chain='A'`, `chain='B'`, `chain='DE'`, by default all chains are parsed
- **subset** (*str*[778]) – a predefined keyword to parse subset of atoms, valid keywords are `'calpha'` (`'ca'`), `'backbone'` (`'bb'`), or **None** (read all atoms), e.g. `subset='bb'`
- **model** (*int, list*) – model index or None (read all models), e.g. `model=10`

[775] http://www.wwpdb.org/documentation/format32/v3.2.html
[776] http://docs.python.org/library/stdtypes.html#str
[777] http://docs.python.org/library/stdtypes.html#str
[778] http://docs.python.org/library/stdtypes.html#str

- **header** (*bool*[779]) – if **True** PDB header content will be parsed and returned
- **altloc** (*str*[780]) – if a location indicator is passed, such as `'A'` or `'B'`, only indicated alternate locations will be parsed as the single coordinate set of the AtomGroup, if *altloc* is set `'all'` then all alternate locations will be parsed and each will be appended as a distinct coordinate set, default is `"A"`
- **biomol** (*bool*[781]) – if **True**, biomolecules are obtained by transforming the coordinates using information from header section will be returned. This option uses *buildBiomolecules()* (page 242) and as noted there, atoms in biomolecules are ordered according to the original chain IDs. Chains may have the same chain ID, in which case they are given different segment names. Default is **False**
- **secondary** (*bool*[782]) – if **True**, secondary structure information from header section will be assigned to atoms. Default is **False**

If `model=0` and `header=True`, return header dictionary only.

parsePDB(**pdb*, ***kwargs*)

Returns an *AtomGroup* (page 38) and/or dictionary containing header data parsed from a PDB file.

This function extends *parsePDBStream()* (page 267).

See Parse PDB files[783] for a detailed usage example.

Parameters **pdb** – one PDB identifier or filename, or a list of them. If needed, PDB files are downloaded using *fetchPDB()* (page 266) function.

You can also provide arguments that you would like passed on to fetchPDB().

Parameters **extend_biomol** (*bool*[784]) – whether to extend the list of results with a list rather than appending, which can create a mixed list, especially when biomol=True. Default value is False to reproduce previous behaviour. This value is ignored when result is not a list (header=True or model=0).

Please note that resnames are only taken as 3 characters and chids can be 2. Hence, TIP3S is split into resname TIP and chid 3S.

Parameters

- **title** (*str*[785]) – title of the *AtomGroup* (page 38) instance, default is the PDB filename or PDB identifier
- **ag** (*AtomGroup* (page 38)) – *AtomGroup* (page 38) instance for storing data parsed from PDB file, number of atoms in *ag* and number of atoms parsed from the PDB file must be the same and atoms in *ag* and those in PDB file must be in the same order. Non-coordinate data stored in *ag* will be overwritten with those parsed from the file.
- **chain** (*str*[786]) – chain identifiers for parsing specific chains, e.g. `chain='A'`, `chain='B'`, `chain='DE'`, by default all chains are parsed

[779] http://docs.python.org/library/functions.html#bool
[780] http://docs.python.org/library/stdtypes.html#str
[781] http://docs.python.org/library/functions.html#bool
[782] http://docs.python.org/library/functions.html#bool
[783] http://prody.csb.pitt.edu/tutorials/structure_analysis/pdbfiles.html#parsepdb
[784] http://docs.python.org/library/functions.html#bool
[785] http://docs.python.org/library/stdtypes.html#str
[786] http://docs.python.org/library/stdtypes.html#str

- **subset** (*str*[787]) – a predefined keyword to parse subset of atoms, valid keywords are `'calpha'` (`'ca'`), `'backbone'` (`'bb'`), or **None** (read all atoms), e.g. `subset='bb'`
- **model** (*int, list*) – model index or None (read all models), e.g. `model=10`
- **header** (*bool*[788]) – if **True** PDB header content will be parsed and returned
- **altloc** (*str*[789]) – if a location indicator is passed, such as `'A'` or `'B'`, only indicated alternate locations will be parsed as the single coordinate set of the AtomGroup, if *altloc* is set `'all'` then all alternate locations will be parsed and each will be appended as a distinct coordinate set, default is `"A"`
- **biomol** (*bool*[790]) – if **True**, biomolecules are obtained by transforming the coordinates using information from header section will be returned. This option uses *buildBiomolecules()* (page 242) and as noted there, atoms in biomolecules are ordered according to the original chain IDs. Chains may have the same chain ID, in which case they are given different segment names. Default is **False**
- **secondary** (*bool*[791]) – if **True**, secondary structure information from header section will be assigned to atoms. Default is **False**

If `model=0` and `header=True`, return header dictionary only.

parseChainsList(*filename*)

Parse a set of PDBs and extract chains based on a list in a text file.

Parameters **filename** (*str*[792]) – the name of the file to be read

Returns: lists containing an :class:'.AtomGroup' for each PDB, the headers for those PDBs, and the requested *Chain* (page 53) objects

parsePQR(*filename*, ***kwargs*)

Returns an *AtomGroup* (page 38) containing data parsed from PDB lines.

Parameters

- **filename** (*str*[793]) – a PQR filename
- **title** (*str*[794]) – title of the *AtomGroup* (page 38) instance, default is the PDB filename or PDB identifier
- **ag** (*AtomGroup* (page 38)) – *AtomGroup* (page 38) instance for storing data parsed from PDB file, number of atoms in *ag* and number of atoms parsed from the PDB file must be the same and atoms in *ag* and those in PDB file must be in the same order. Non-coordinate data stored in *ag* will be overwritten with those parsed from the file.
- **chain** (*str*[795]) – chain identifiers for parsing specific chains, e.g. `chain='A'`, `chain='B'`, `chain='DE'`, by default all chains are parsed
- **subset** (*str*[796]) – a predefined keyword to parse subset of atoms, valid keywords are `'calpha'` (`'ca'`), `'backbone'` (`'bb'`), or **None** (read all atoms), e.g.

[787] http://docs.python.org/library/stdtypes.html#str
[788] http://docs.python.org/library/functions.html#bool
[789] http://docs.python.org/library/stdtypes.html#str
[790] http://docs.python.org/library/functions.html#bool
[791] http://docs.python.org/library/functions.html#bool
[792] http://docs.python.org/library/stdtypes.html#str
[793] http://docs.python.org/library/stdtypes.html#str
[794] http://docs.python.org/library/stdtypes.html#str
[795] http://docs.python.org/library/stdtypes.html#str
[796] http://docs.python.org/library/stdtypes.html#str

subset='bb'

writePDBStream(*stream*, *atoms*, *csets=None*, ***kwargs*)

Write *atoms* in PDB format to a *stream*.

Parameters

- **stream** – anything that implements a `write()` method (e.g. file, buffer, stdout)
- **renumber** (*bool*[797]) – whether to renumber atoms with serial indices Default is **True**
- **hybrid36** (*bool*[798]) – whether to use hybrid36 format for atom residue numbers Default is **False**, which means using hexadecimal instead. NB: ChimeraX seems to prefer hybrid36 and may have problems with hexadecimal.
- **full_ter** (*bool*[799]) – whether to write full TER lines with atoms info Default is **True**
- **write_remarks** (*bool*[800]) – whether to write REMARK lines Default is **True**
- **atoms** – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets
- **beta** – a list or array of number to be outputted in beta column
- **occupancy** – a list or array of number to be outputted in occupancy column
- **hybrid36** – whether to use hybrid36 format for atoms with serial greater than 99999. Hexadecimal is used otherwise. Default is False

writePDB(*filename*, *atoms*, *csets=None*, *autoext=True*, ***kwargs*)

Write *atoms* in PDB format to a file with name *filename* and return *filename*. If *filename* ends with `.gz`, a compressed file will be written.

Parameters

- **renumber** (*bool*[801]) – whether to renumber atoms with serial indices Default is **True**
- **hybrid36** (*bool*[802]) – whether to use hybrid36 format for atom residue numbers Default is **False**, which means using hexadecimal instead. NB: ChimeraX seems to prefer hybrid36 and may have problems with hexadecimal.

Please note that resnames longer than 3 characters will be trimmed.

Parameters

- **atoms** – an object with atom and coordinate data
- **csets** – coordinate set indices, default is all coordinate sets
- **beta** – a list or array of number to be outputted in beta column
- **occupancy** – a list or array of number to be outputted in occupancy column
- **hybrid36** (*bool*[803]) – whether to use hybrid36 format for atoms with serial greater than 99999. Hexadecimal is used otherwise. Default is False

[797] http://docs.python.org/library/functions.html#bool
[798] http://docs.python.org/library/functions.html#bool
[799] http://docs.python.org/library/functions.html#bool
[800] http://docs.python.org/library/functions.html#bool
[801] http://docs.python.org/library/functions.html#bool
[802] http://docs.python.org/library/functions.html#bool
[803] http://docs.python.org/library/functions.html#bool

- **autoext** – when not present, append extension `.pdb` to *filename*

writeChainsList(*chains*, *filename*)
Write a text file containing a list of chains that can be parsed.

Parameters

- **chains** (*list*[804]) – a list of *Chain* (page 53) objects
- **filename** (*str*[805]) – the name of the file to be written

writePQR(*filename*, *atoms*, ***kwargs*)
Write *atoms* in PQR format to a file with name *filename*. Only current coordinate set is written. Returns *filename* upon success. If *filename* ends with `.gz`, a compressed file will be written.

writePQRStream(*stream*, *atoms*, ***kwargs*)

3.10.25 STAR File

This module defines functions for parsing STAR[806] (Self-defining Text Archiving and Retrieval) files. This includes metadata files from cryo-EM image analysis programs including RELION and XMIPP as well as the Crystallographic Information File (CIF) format used for much of the PDB.

class StarDict(*parsingDict*, *prog*, *title='unnamed'*, *indices=None*)

getDict()

getTitle()

numDataBlocks()

pop(*index*)
Pop dataBlock with the given index from the list of dataBlocks in StarDict

printData()

search(*substr*, *return_indices=False*)

setTitle(*value*)

class StarDataBlock(*starDict*, *key*, *indices=None*)

getDict()

getLoop(*index*)

getTitle()

numEntries()

numLoops()

pop(*index*)
Pop loop with the given index from the list of loops in dataBlock

printData()

search(*substr*, *return_indices=False*)

setTitle(*title*)

[804] http://docs.python.org/library/stdtypes.html#list
[805] http://docs.python.org/library/stdtypes.html#str
[806] https://www2.mrc-lmb.cam.ac.uk/relion/index.php/Conventions_%26_File_formats#The_STAR_format

class StarLoop(*dataBlock*, *key*, *indices=None*)

getData(*key*)

getDict()

getTitle()

numFields()

numRows()

printData()

search(*substr*, *return_indices=False*)

setTitle(*title*)

parseSTAR(*filename*, ***kwargs*)
Returns a dictionary containing data parsed from a STAR file.

Parameters

- **filename** (*str*[807]) – a filename The .star extension can be omitted.
- **start** (*int, None*) – line number for starting Default is **None**, meaning start at the beginning
- **stop** (*int, None*) – line number for stopping Default is **None**, meaning don't stop.
- **shlex** (*bool*[808]) – whether to use shlex for splitting lines so as to preserve quoted substrings Default is **False**

writeSTAR(*filename*, *starDict*, ***kwargs*)
Writes a STAR file from a dictionary containing data such as that parsed from a Relion STAR file.

Parameters

- **filename** (*str*[809]) – a filename The .star extension can be omitted.
- **starDict** (*dict*[810]) – a dictionary in STAR format This should have nested entries starting with data blocks then loops/tables then field names and finally data.

kwargs can be given including the program style to follow (*prog*)

parseImagesFromSTAR(*particlesSTAR*, ***kwargs*)
Parses particle images using data from a STAR file containing information about them.

Parameters

- **particlesSTAR** (*str*[811]) – a filename for a STAR file.
- **block_indices** (list, ndarray) – indices for data blocks containing rows corresponding to images of interest The indexing scheme is similar to that for numpy arrays. Default behavior is use all data blocks about images
- **row_indices** (list, ndarray) – indices for rows corresponding to images of interest The indexing scheme is similar to that for numpy arrays. row_indices should be a 1D or 2D array-like. 2D row_indices should contain an entry for each relevant

[807] http://docs.python.org/library/stdtypes.html#str
[808] http://docs.python.org/library/functions.html#bool
[809] http://docs.python.org/library/stdtypes.html#str
[810] http://docs.python.org/library/stdtypes.html#dict
[811] http://docs.python.org/library/stdtypes.html#str

loop. If a 1D array-like is given the same row indices will be applied to all loops. Default behavior is to use all rows about images

- **particle_indices** (list, ndarray) – indices for particles regardless of STAR structure default is take all particles Please note: this acts after block_indices and row_indices
- **saveImageArrays** (*bool*[812]) – whether to save the numpy array for each image to file default is False
- **saveDirectory** (*str*[813]) – directory where numpy image arrays are saved default is **None**, which means save to the current working directory
- **rotateImages** (*bool*[814]) – whether to apply in plane translations and rotations using provided psi and origin data, default is True

parseSTARSection(*lines*, *key*, *report=True*)

Parse a section of data from *lines* from a STAR file corresponding to a *key* (part before the dot). This can be a loop or data block.

Returns data encapulated in a list and the associated fields.

Parameters **report** (*bool*[815]) – whether to report warnings about not finding data default True

3.10.26 Stride Tools

This module defines functions for executing STRIDE program and parsing its output.

execSTRIDE(*pdb*, *outputname=None*, *outputdir=None*)

Execute STRIDE program for given *pdb*. *pdb* can be an identifier or a PDB file path. If *pdb* is a compressed file, it will be decompressed using Python gzip[816] library. When no *outputname* is given, output name will be pdb.stride. .stride extension will be appended automatically to *outputname*. If outputdir is given, STRIDE output and uncompressed PDB file will be written into this folder. Upon successful execution of **stride pdb > out** command, output filename is returned.

For more information on STRIDE see http://webclu.bio.wzw.tum.de/stride/. If you benefited from STRIDE, please consider citing *[DF95]* (page 397).

parseSTRIDE(*stride*, *ag*)

Parse STRIDE output from file *stride* into *AtomGroup* (page 38) instance *ag*. STRIDE output file must be in the new format used from July 1995 and onwards. When *stride* file is parsed, following attributes are added to *ag*:

- *stride_resnum*: STRIDE's sequential residue number, starting at the first residue actually in the data set.
- *stride_phi*, *stride_psi*: peptide backbone torsion angles phi and psi
- *stride_area*: residue solvent accessible area

performSTRIDE(*pdb*)

Perform STRIDE calculations and parse results. STRIDE data is returned in an *AtomGroup* (page 38) instance. See also *execSTRIDE()* (page 273) and *parseSTRIDE()* (page 273).

[812] http://docs.python.org/library/functions.html#bool
[813] http://docs.python.org/library/stdtypes.html#str
[814] http://docs.python.org/library/functions.html#bool
[815] http://docs.python.org/library/functions.html#bool
[816] http://docs.python.org/library/gzip.html#module-gzip

3.10.27 Water bridge finder (WatFinder)

This module provides the the WatFinder toolkit that detects, predicts and analyzes water bridges.

calcWaterBridges(*atoms*, ***kwargs*)

Compute water bridges for a protein that has water molecules.

Parameters

- **atoms** (*Atomic* (page 47)) – Atomic object from which atoms are considered
- **method** (*string 'cluster' | 'chain'*) – cluster or chain, where chain find shortest water bridging path between two protein atoms default is 'chain'
- **distDA** (*int, float*) – maximal distance between water/protein donor and acceptor default is 3.5
- **distWR** (*int, float*) – maximal distance between considered water and any residue default is 4
- **anglePDWA** (*(int, int)*) – angle range where protein is donor and water is acceptor default is (100, 200)
- **anglePAWD** – angle range where protein is acceptor and water is donor default is (100, 140)
- **angleWW** (*(int, int)*) – angle between water donor/acceptor default is (140, 180)
- **maxDepth** (*int, None*) – maximum number of waters in chain/depth of residues in cluster default is 2
- **maxNumRes** (*int, None*) – maximum number of water+protein residues in cluster default is None
- **donors** (*list*[817]) – which atoms to count as donors default is ['N', 'O', 'S', 'F']
- **acceptors** (*list*[818]) – which atoms to count as acceptors default is ['N', 'O', 'S', 'F']
- **output** (*bool*[819]) – return information arrays, (protein atoms, water atoms), or just atom indices per bridge default is 'atomic'
- **isInfoLog** – should log information default is True

calcWaterBridgesTrajectory(*atoms*, *trajectory*, ***kwargs*)

Computes water bridges for a given trajectory. Kwargs for options are the same as in calcWaterBridges.

Parameters

- **atoms** (*Atomic* (page 47)) – Atomic object from which atoms are considered
- **trajectory** (Trajectory', :class:.Ensemble', *Atomic* (page 47)) – Trajectory data coming from a DCD or multi-model PDB file.
- **start_frame** (*int*[820]) – frame to start from
- **stop_frame** (*int*[821]) – frame to stop

[817] http://docs.python.org/library/stdtypes.html#list
[818] http://docs.python.org/library/stdtypes.html#list
[819] http://docs.python.org/library/functions.html#bool
[820] http://docs.python.org/library/functions.html#int
[821] http://docs.python.org/library/functions.html#int

getWaterBridgesInfoOutput(*waterBridgesAtomic*)
Converts single frame/trajectory atomic output from calcWaterBridges/Trajectory to info output.

Parameters **waterBridgesAtomic** (*list*[822]) – water bridges from calcWaterBridges/Trajectory

calcWaterBridgesStatistics(*frames*, *trajectory*, ***kwargs*)
Returns statistics. Value is percentage of bridge appearance of frames for each residue.

Parameters

- **frames** (*list*[823]) – list of water bridges from calcWaterBridgesTrajectory(), output='atomic'
- **output** (*'info' | 'indices'*) – return dictorinary whose keys are tuples of resnames or resids default is 'indices'
- **filename** (*string*[824]) – name of file to save statistic information if wanted default is None

getWaterBridgeStatInfo(*stats*, *atoms*)
Converts calcWaterBridgesStatistic indices output to info output from stat.

Parameters

- **stats** (*dictionary*) – statistics returned by calcWaterBridgesStatistics, output='indices'
- **atoms** (*Atomic* (page 47)) – Atomic object from which atoms are considered

calcWaterBridgeMatrix(*data*, *metric*)
Returns matrix which has metric as value and residue ids as ax indices.

Parameters

- **data** (*dict*[825]) – dictionary returned by calcWaterBridgesStatistics, output='indices'
- **metric** (*'percentage' | 'distAvg' | 'distStd'*) – dict key from data

showWaterBridgeMatrix(*data*, *metric*)
Shows matrix which has percentage/avg distance as value and residue ids as ax indices.

Parameters

- **data** (*dict*[826]) – dictionary returned by calcWaterBridgesStatistics, output='indices'
- **metric** (*'percentage' | 'distAvg' | 'distStd'*) – dict key from data

calcBridgingResiduesHistogram(*frames*, ***kwargs*)
Calculates, plots and returns number of frames that each residue is involved in making water bridges, sorted by value.

Parameters

- **frames** (*list*[827]) – list of water bridges from calcWaterBridgesTrajectory(), output='atomic'

[822] http://docs.python.org/library/stdtypes.html#list
[823] http://docs.python.org/library/stdtypes.html#list
[824] http://docs.python.org/library/string.html#module-string
[825] http://docs.python.org/library/stdtypes.html#dict
[826] http://docs.python.org/library/stdtypes.html#dict
[827] http://docs.python.org/library/stdtypes.html#list

- **clip** (*int*[828]) – maximal number of residues on graph; to represent all set None default is 20

calcWaterBridgesDistribution(*frames*, *res_a*, *res_b=None*, ***kwargs*)
Returns distribution for certain metric and plots if possible.

Parameters

- **res_a** – name of first residue
- **res_b** – name of second residue default is None
- **metric** (*'residues' | 'waters' | 'distance' | 'location'*) – 'residues' returns names and frame count of residues interacting with res_a, 'waters' returns water count for each bridge between res_a and res_b 'distance' returns distance between each pair of protein atoms involved in bridge between res_a and res_b 'location' returns dictionary with backbone/sidechain count information
- **output** (*'dict' | 'indices'*) – return 2D matrices or dictionary where key is residue info default is 'dict'

Trajectory DCD file - necessary for distance distribution

savePDBWaterBridges(*bridges*, *atoms*, *filename*)
Saves single PDB with occupancy on protein atoms and waters involved bridges.

Parameters

- **bridges** (*list*[829]) – atomic output from calcWaterBridges
- **atoms** (*Atomic* (page 47)) – Atomic object from which atoms are considered
- **filename** (*string*[830]) – name of file to be saved

savePDBWaterBridgesTrajectory(*bridgeFrames*, *atoms*, *filename*, *trajectory=None*)
Saves one PDB per frame with occupancy and beta on protein atoms and waters forming bridges in frame.

Parameters

- **bridgeFrames** (*list*[831]) – atomic output from calcWaterBridgesTrajectory
- **atoms** (*Atomic* (page 47)) – Atomic object from which atoms are considered
- **filename** (*string*[832]) – name of file to be saved; must end in .pdb
- **trajectory** – DCD trajectory (not needed for multimodal PDB)

saveWaterBridges(*atomicBridges*, *filename*)
Save water bridges as information (.txt) or WaterBridges (.wb) parsable file.

Parameters

- **atomicBridges** (*list*[833]) – atomic output from calcWaterBridges/Trajectory
- **filename** (*string*[834]) – path where file should be saved

parseWaterBridges(*filename*, *atoms*)
Parse water bridges from .wb file saved by saveWaterBridges, returns atomic type.

[828] http://docs.python.org/library/functions.html#int
[829] http://docs.python.org/library/stdtypes.html#list
[830] http://docs.python.org/library/string.html#module-string
[831] http://docs.python.org/library/stdtypes.html#list
[832] http://docs.python.org/library/string.html#module-string
[833] http://docs.python.org/library/stdtypes.html#list
[834] http://docs.python.org/library/string.html#module-string

Parameters

- **filename** (*string*[835]) – path of file where bridges are stored
- **atoms** (*Atomic* (page 47)) – Atomic object on which calcWaterBridges was performed

findClusterCenters(*file_pattern*, ***kwargs*)
Find molecules that are forming cluster in 3D space.

Parameters

- **file_pattern** (*str*[836]) – file pattern for anlaysis it can include '*' example:'file_*.pdb' will analyze file_1.pdb, file_2.pdb, etc.
- **selection** (*str*[837]) – selection string by default water and name OH2 is used
- **distC** (*int, float default is 0.3*) – distance to other molecules
- **numC** (*int*[838]) – min number of molecules in a cluster default is 3

filterStructuresWithoutWater(*structures*, *min_water=0*, *filenames=None*)
This function will filter out structures from *structures* that have no water or fewer water molecules than *min_water*.

Parameters

- **structures** (*list*[839]) – list of *Atomic* (page 47) structures to be filtered
- **min_water** (*int*[840]) – minimum number of water O atoms, default is 0
- **filenames** (*list*[841]) – an optional list of filenames to filter too This is an output argument

3.10.28 wwPDB Tools

This module defines functions for accessing wwPDB servers.

wwPDBServer(**key*)
Set/get wwPDB[842] FTP/HTTP server location used for downloading PDB structures. Use one of the following keywords for setting a server:

wwPDB FTP server	*Key* (case insensitive)
RCSB PDB (USA) (default)	RCSB, USA, US
PDBe (Europe)	PDBe, Europe, Euro, EU
PDBj (Japan)	PDBj, Japan, Jp

fetchPDBviaFTP(**pdb*, ***kwargs*)
Retrieve PDB (default), PDBML, mmCIF, or EMD file(s) for specified *pdb* identifier(s) and return path(s). Downloaded files will be stored in local PDB folder, if one is set using `pathPDBFolder()`, and copied into *folder*, if specified by the user. If no destination folder is specified, files will be saved in the current working directory. If *compressed* is **False**, decompressed files will be copied

[835] http://docs.python.org/library/string.html#module-string
[836] http://docs.python.org/library/stdtypes.html#str
[837] http://docs.python.org/library/stdtypes.html#str
[838] http://docs.python.org/library/functions.html#int
[839] http://docs.python.org/library/stdtypes.html#list
[840] http://docs.python.org/library/functions.html#int
[841] http://docs.python.org/library/stdtypes.html#list
[842] http://www.wwpdb.org/

into *folder*. *format* keyword argument can be used to retrieve PDBML[843], mmCIF[844] and PDBML[845] files: `format='cif'` will fetch an mmCIF file, `format='emd'` will fetch an EMD file, and `format='xml'` will fetch a PDBML file. If PDBML header file is desired, `noatom=True` argument will do the job.

fetchPDBviaHTTP(**pdb*, ***kwargs*)

Retrieve PDB file(s) for specified *pdb* identifier(s) and return path(s). Downloaded files will be stored in local PDB folder, if one is set using `pathPDBFolder()`, and copied into *folder*, if specified by the user. If no destination folder is specified, files will be saved in the current working directory. If *compressed* is **False**, decompressed files will be copied into *folder*.

3.11 Sequence Analysis

This module contains features for analyzing protein sequences.

3.11.1 Classes

- *MSA* (page 285) - store MSA data indexed by label
- *Sequence* (page 289) - store sequence data

3.11.2 MSA IO

- *MSAFile* (page 287) - read/write MSA files in FASTA/SELEX/Stockholm formats
- *parseMSA()* (page 287) - parse MSA files
- *writeMSA()* (page 288) - parse MSA files

3.11.3 Editing

- *mergeMSA()* (page 286) - merge MSA data for multi-domain proteins
- *refineMSA()* (page 286) - refine MSA by removing gapped columns and/or sequences

3.11.4 Analysis

- *calcMSAOccupancy()* (page 279) - calculate row (sequence) or column occupancy
- *calcShannonEntropy()* (page 279) - calculate Shannon entropy
- *buildMutinfoMatrix()* (page 279) - build mutual information matrix
- *buildOMESMatrix()* (page 281) - build mutual observed minus expected squared covariance matrix
- *buildSCAMatrix()* (page 281)- build statistical coupling analysis matrix
- *buildSeqidMatrix()* (page 280)- build sequence identity matrix
- *buildDirectInfoMatrix()* (page 282) - build direct information matrix
- *uniqueSequences()* (page 281) - select unique sequences
- *applyMutinfoCorr()* (page 280) - apply correction to mutual information matrix
- *applyMutinfoNorm()* (page 280) - apply normalization to mutual information matrix
- *calcMeff()* (page 282) - calculate sequence weights

[843] http://pdbml.pdb.org/
[844] http://mmcif.pdb.org/
[845] ftp://ftp.wwpdb.org/pub/emdb/doc/Map-format/current/EMDB_map_format.pdf

- *calcRankorder()* (page 280) - rank order scores

3.11.5 Plotting

- *showShannonEntropy()* (page 288) - plot Shannon entropy
- *showMSAOccupancy()* (page 288) - plot row (sequence) or column occupancy
- *showMutinfoMatrix()* (page 288) - show mutual information matrix

3.11.6 Analysis Functions

This module defines MSA analysis functions.

calcShannonEntropy(*msa*, *ambiguity=True*, *omitgaps=True*, ***kwargs*)

Returns Shannon entropy array calculated for *msa*, which may be an *MSA* (page 285) instance or a 2D Numpy character array. Implementation is case insensitive and handles ambiguous amino acids as follows:

- **B** (Asx) count is allocated to *D* (Asp) and *N* (Asn)
- **Z** (Glx) count is allocated to *E* (Glu) and *Q* (Gln)
- **J** (Xle) count is allocated to *I* (Ile) and *L* (Leu)
- **X** (Xaa) count is allocated to the twenty standard amino acids

Selenocysteine (**U**, Sec) and pyrrolysine (**O**, Pyl) are considered as distinct amino acids. When *ambiguity* is set **False**, all alphabet characters as considered as distinct types.

All non-alphabet characters are considered as gaps, and they are handled in two ways:

- non-existent, the probability of observing amino acids in a given column is adjusted, by default
- as a distinct character with its own probability, when *omitgaps* is **False**

buildMutinfoMatrix(*msa*, *ambiguity=True*, *turbo=True*, ***kwargs*)

Returns mutual information matrix calculated for *msa*, which may be an *MSA* (page 285) instance or a 2D Numpy character array. Implementation is case insensitive and handles ambiguous amino acids as follows:

- **B** (Asx) count is allocated to *D* (Asp) and *N* (Asn)
- **Z** (Glx) count is allocated to *E* (Glu) and *Q* (Gln)
- **J** (Xle) count is allocated to *I* (Ile) and *L* (Leu)
- **X** (Xaa) count is allocated to the twenty standard amino acids
- Joint probability of observing a pair of ambiguous amino acids is allocated to all potential combinations, e.g. probability of **XX** is allocated to 400 combinations of standard amino acids, similarly probability of **XB** is allocated to 40 combinations of *D* and *N* with the standard amino acids.

Selenocysteine (**U**, Sec) and pyrrolysine (**O**, Pyl) are considered as distinct amino acids. When *ambiguity* is set **False**, all alphabet characters as considered as distinct types. All non-alphabet characters are considered as gaps.

Mutual information matrix can be normalized or corrected using `applyMINormalization()` and `applyMICorrection()` methods, respectively. Normalization by joint entropy can performed using this function with *norm* option set **True**.

By default, *turbo* mode, which uses memory as large as the MSA array itself but runs four to five times faster, will be used. If memory allocation fails, the implementation will fall back to slower and memory efficient mode.

calcMSAOccupancy(*msa*, *occ='res'*, *count=False*)
Returns occupancy array calculated for residue positions (default, `'res'` or `'col'` for *occ*) or sequences (`'seq'` or `'row'` for *occ*) of *msa*, which may be an *MSA* (page 285) instance or a 2D NumPy character array. By default, occupancy [0-1] will be calculated. If *count* is **True**, count of non-gap characters will be returned. Implementation is case insensitive.

applyMutinfoCorr(*mutinfo*, *corr='prod'*)
Returns a copy of *mutinfo* array after average product correction (default) or average sum correction is applied. See *[DSD08]* (page 397) for details.

applyMutinfoNorm(*mutinfo*, *entropy*, *norm='sument'*)
Apply one of the normalizations discussed in *[MLC05]* (page 397) to *mutinfo* matrix. *norm* can be one of the following:

- `'sument'`: $H(X) + H(Y)$, sum of entropy of columns
- `'minent'`: $min\{H(X), H(Y)\}$, minimum entropy
- `'maxent'`: $max\{H(X), H(Y)\}$, maximum entropy
- **`'mincon'`: $min\{H(X|Y), H(Y|X)\}$, minimum conditional** entropy
- **`'maxcon'`: $max\{H(X|Y), H(Y|X)\}$, maximum conditional** entropy

where $H(X)$ is the entropy of a column, and $H(X|Y) = H(X) - MI(X, Y)$. Normalization with joint entropy, i.e. $H(X, Y)$, can be done using *buildMutinfoMatrix()* (page 279) *norm* argument.

calcRankorder(*matrix*, *zscore=False*, ***kwargs*)
Returns indices of elements and corresponding values sorted in descending order, if *descend* is **True** (default). Can apply a zscore normalization; by default along *axis* - 0 such that each column has `mean=0` and `std=1`. If *zcore* analysis is used, return value contains the zscores. If matrix is symmetric only lower triangle indices will be returned, with diagonal elements if *diag* is **True** (default).

filterRankedPairs(*pdb*, *indices*, *msa_indices*, *rank_row*, *rank_col*, *zscore_sort*, *num_of_pairs=20*, *seqDistance=5*, *resi_range=None*, *pdbDistance=8*, *chain1='A'*, *chain2='A'*)
indices and msa_indices are lists output from alignSequenceToMSA

rank_row, rank_col and zscore_sort are the outputs from calcRankorder

Parameters

- **num_of_pairs** (*int*[846]) – The number of pairs to be output, if no value is given then all pairs are output. Default is 20
- **seqDistance** (*int*[847]) – Remove pairs that are closer than this in the reference sequence Default is 5
- **pdbDistance** (*int*[848]) – Remove pairs with Calpha atoms further apart than this in the PDB Default is 8
- **chain1** (*str*[849]) – The chain used for the residue specified by rank_row when measuring distances
- **chain2** (*str*[850]) – The chain used for the residue specified by rank_col when measuring distances

buildSeqidMatrix(*msa*, *turbo=True*)
Returns sequence identity matrix for *msa*.

[846] http://docs.python.org/library/functions.html#int
[847] http://docs.python.org/library/functions.html#int
[848] http://docs.python.org/library/functions.html#int
[849] http://docs.python.org/library/stdtypes.html#str
[850] http://docs.python.org/library/stdtypes.html#str

By default, *turbo* mode, which uses memory as large as the MSA array itself but runs four to five times faster, will be used. If memory allocation fails, the implementation will fall back to slower and memory efficient mode.

uniqueSequences(*msa*, *seqid=0.98*, *turbo=True*)

Returns a boolean array marking unique sequences in *msa*. A sequence sharing sequence identity of *seqid* or more with another sequence coming before itself in *msa* will have a **True** value in the array.

By default, *turbo* mode, which uses memory as large as the MSA array itself but runs four to five times faster, will be used. If memory allocation fails, the implementation will fall back to slower and memory efficient mode.

buildOMESMatrix(*msa*, *ambiguity=True*, *turbo=True*, ***kwargs*)

Returns OMES (Observed Minus Expected Squared) covariance matrix calculated for *msa*, which may be an *MSA* (page 285) instance or a 2D NumPy character array. OMES is defined as:

$$OMES\_(i,j) = \sum\left(\frac{(N\_OBS - N\_EX)^2}{N\_EX}\right) = N * \sum\left(\frac{(f\_i,j - f\_i * f\_j)^2}{f\_i * f\_j}\right)$$

Implementation is case insensitive and handles ambiguous amino acids as follows:

- **B** (Asx) count is allocated to *D* (Asp) and *N* (Asn)
- **Z** (Glx) count is allocated to *E* (Glu) and *Q* (Gln)
- **J** (Xle) count is allocated to *I* (Ile) and *L* (Leu)
- **X** (Xaa) count is allocated to the twenty standard amino acids
- Joint probability of observing a pair of ambiguous amino acids is allocated to all potential combinations, e.g. probability of **XX** is allocated to 400 combinations of standard amino acids, similarly probability of **XB** is allocated to 40 combinations of *D* and *N* with the standard amino acids.

Selenocysteine (**U**, Sec) and pyrrolysine (**O**, Pyl) are considered as distinct amino acids. When *ambiguity* is set **False**, all alphabet characters as considered as distinct types. All non-alphabet characters are considered as gaps.

By default, *turbo* mode, which uses memory as large as the MSA array itself but runs four to five times faster, will be used. If memory allocation fails, the implementation will fall back to slower and memory efficient mode.

buildSCAMatrix(*msa*, *turbo=True*, ***kwargs*)

Returns SCA matrix calculated for *msa*, which may be an *MSA* (page 285) instance or a 2D Numpy character array.

Implementation is case insensitive and handles ambiguous amino acids as follows:

- **B** (Asx) count is allocated to *D* (Asp) and *N* (Asn)
- **Z** (Glx) count is allocated to *E* (Glu) and *Q* (Gln)
- **J** (Xle) count is allocated to *I* (Ile) and *L* (Leu)
- **X** (Xaa) count is allocated to the twenty standard amino acids
- Joint probability of observing a pair of ambiguous amino acids is allocated to all potential combinations, e.g. probability of **XX** is allocated to 400 combinations of standard amino acids, similarly probability of **XB** is allocated to 40 combinations of *D* and *N* with the standard amino acids.

Selenocysteine (**U**, Sec) and pyrrolysine (**O**, Pyl) are considered as distinct amino acids. When *ambiguity* is set **False**, all alphabet characters as considered as distinct types. All non-alphabet characters are considered as gaps.

By default, *turbo* mode, which uses memory as large as the MSA array itself but runs four to five times faster, will be used. If memory allocation fails, the implementation will fall back to slower and memory efficient mode.

buildDirectInfoMatrix(*msa*, *seqid=0.8*, *pseudo_weight=0.5*, *refine=False*, ***kwargs*)
Returns direct information matrix calculated for *msa*, which may be an `MSA` (page 285) instance or a 2D Numpy character array.

Sequences sharing sequence identity of *seqid* or more with another sequence are regarded as similar sequences for calculating their weights using `calcMeff()` (page 282).

pseudo_weight are the weight for pseudo count probability.

Sequences are not refined by default. When *refine* is set **True**, the MSA will be refined by the first sequence and the shape of direct information matrix will be smaller.

calcMeff(*msa*, *seqid=0.8*, *refine=False*, *weight=False*, ***kwargs*)
Returns the Meff for *msa*, which may be an `MSA` (page 285) instance or a 2D Numpy character array.

Since similar sequences in an *msa* decreases the diversity of *msa*, *Meff* gives a weight for sequences in the *msa*.

For example: One sequence in MSA has 5 other similar sequences in this MSA(itself included). The weight of this sequence is defined as 1/5=0.2. Meff is the sum of all sequence weights. In another word, Meff can be understood as the effective number of independent sequences.

Sequences sharing sequence identity of *seqid* or more with another sequence are regarded as similar sequences to calculate Meff.

Sequences are not refined by default. When *refine* is set **True**, the MSA will be refined by the first sequence.

The weight for each sequence are returned when *weight* is **True**.

buildPCMatrix(*msa*, *turbo=False*, ***kwargs*)
Returns PC matrix calculated for *msa*, which may be an `MSA` (page 285) instance or a 2D Numpy character array.

Implementation is case insensitive and handles ambiguous amino acids as follows:

- **B** (Asx) count is allocated to *D* (Asp) and *N* (Asn)
- **Z** (Glx) count is allocated to *E* (Glu) and *Q* (Gln)
- **J** (Xle) count is allocated to *I* (Ile) and *L* (Leu)
- **X** (Xaa) count is allocated to the twenty standard amino acids
- Joint probability of observing a pair of ambiguous amino acids is allocated to all potential combinations, e.g. probability of **XX** is allocated to 400 combinations of standard amino acids, similarly probability of **XB** is allocated to 40 combinations of *D* and *N* with the standard amino acids.

Selenocysteine (**U**, Sec) and pyrrolysine (**O**, Pyl) are considered as distinct amino acids. When *ambiguity* is set **False**, all alphabet characters as considered as distinct types. All non-alphabet characters are considered as gaps.

buildMSA(*sequences*, *title='Unknown'*, *labels=None*, ***kwargs*)
Aligns sequences with clustalw or clustalw2 or Biopython and returns the resulting MSA.

Parameters

- **sequences** (`Atomic` (page 47), `MSA` (page 285), `ndarray`, str) – a file, MSA object or a list or array containing sequences as Atomic objects with `getSequence()` or

Sequence objects or strings. If strings are used then labels must be provided using `labels`

- **title** (*str*[851]) – the title for the MSA and it will be used as the prefix for output files.
- **labels** (*list*[852]) – a list of labels to go with the sequences
- **align** (*str*[853]) – whether to align the sequences default True
- **method** – alignment method, one of either Biopython 'global', Biopython 'local', clustalw(2), or another software in your path. Default is 'local'

showAlignment(*alignment*, ***kwargs*)

Prints out an alignment as sets of short rows with labels.

Parameters

- **alignment** – any object with aligned sequences
- **row_size** (*int*[854]) – the size of each row default 60
- **indices** (ndarray, list) – a set of indices for some or all sequences that will be shown above the relevant sequences
- **index_start** (*int*[855]) – how far along the alignment to start putting indices default 0
- **index_stop** (*int*[856]) – how far along the alignment to stop putting indices default the point when the shortest sequence stops
- **labels** (*list*[857]) – a list of labels

alignTwoSequencesWithBiopython(*seq1*, *seq2*, ***kwargs*)

Easily align two sequences with Biopython's globalms or localms. Returns an MSA and indices for use with *showAlignment()* (page 283).

Alignment parameters can be provided as keyword arguments. Default values are as originally set in the proteins.compare module, but now found in utilities.seqtools.

Parameters

- **match** (*int*[858]) – a positive integer, used to reward finding a match
- **mismatch** (*int*[859]) – a negative integer, used to penalise finding a mismatch
- **gap_opening** (*int*[860]) – a negative integer, used to penalise opening a gap
- **gap_extension** (*int*[861]) – a negative integer, used to penalise extending a gap
- **method** (*str*[862]) – method for pairwise2 alignment. Possible values are 'local' and 'global'

[851] http://docs.python.org/library/stdtypes.html#str
[852] http://docs.python.org/library/stdtypes.html#list
[853] http://docs.python.org/library/stdtypes.html#str
[854] http://docs.python.org/library/functions.html#int
[855] http://docs.python.org/library/functions.html#int
[856] http://docs.python.org/library/functions.html#int
[857] http://docs.python.org/library/stdtypes.html#list
[858] http://docs.python.org/library/functions.html#int
[859] http://docs.python.org/library/functions.html#int
[860] http://docs.python.org/library/functions.html#int
[861] http://docs.python.org/library/functions.html#int
[862] http://docs.python.org/library/stdtypes.html#str

alignSequenceToMSA(*seq*, *msa*, ***kwargs*)

Align a sequence from a PDB or Sequence to a sequence from an MSA and create two sets of indices. The resulting MSA and the two sets of indices are returned.

The first set (*seq_indices*) maps the residue indices in the PDB to the reference sequence. The second set (*msa_indices*) indexes the reference sequence in the msa and is used for retrieving values from the first indices.

Parameters

- **seq** (*Atomic* (page 47), *Sequence* (page 289), str) – an object with an associated sequence string or a sequence string itself
- **msa** (*MSA* (page 285)) – a multiple sequence alignment
- **label** (*str*[863]) – a label for a sequence in msa or a PDB ID `msa.getIndex(label)` must return a sequence index
- **chain** (*str*[864]) – which chain from pdb to use for alignment, default is **None**, which does no selection on *seq*. This value will be ignored if seq is not an *Atomic* (page 47) object.

Parameters for Biopython `pairwise2` alignments can be provided as keyword arguments. Default values are originally from `proteins.compare` module, but now found in `utilities.seqtools`.

Parameters

- **match** (*int*[865]) – a positive integer, used to reward finding a match
- **mismatch** (*int*[866]) – a negative integer, used to penalise finding a mismatch
- **gap_opening** (*int*[867]) – a negative integer, used to penalise opening a gap
- **gap_extension** (*int*[868]) – a negative integer, used to penalise extending a gap
- **method** (*str*[869]) – method for pairwise2 alignment. Possible values are `"local"` and `"global"`

calcPercentIdentities(*msa*)

alignSequencesByChain(*PDBs*, ***kwargs*)

Runs *buildMSA()* (page 282) for each chain and optionally joins the results. Returns either a single `MSA` or a dictionary containing an `MSA` for each chain.

Parameters

- **PDBs** (*list*[870]) – a list of `AtomGroup` objects
- **join_chains** (*bool*[871]) – whether to join chain alignments default is True
- **join_char** (*str*[872]) – a character for joining chain alignments default is '/' as used by PIR format alignments

trimAtomsUsingMSA(*atoms*, *msa*, ***kwargs*)

This function uses *alignSequenceToMSA()* (page 283) and has the same kwargs.

[863] http://docs.python.org/library/stdtypes.html#str
[864] http://docs.python.org/library/stdtypes.html#str
[865] http://docs.python.org/library/functions.html#int
[866] http://docs.python.org/library/functions.html#int
[867] http://docs.python.org/library/functions.html#int
[868] http://docs.python.org/library/functions.html#int
[869] http://docs.python.org/library/stdtypes.html#str
[870] http://docs.python.org/library/stdtypes.html#list
[871] http://docs.python.org/library/functions.html#bool
[872] http://docs.python.org/library/stdtypes.html#str

Parameters

- **atoms** (*Atomic* (page 47)) – an atomic structure for trimming
- **msa** (*MSA* (page 285)) – a multiple sequence alignment

3.11.7 Multiple Sequence Alignment

This module defines MSA analysis functions.

class MSA(*msa*, *title='Unknown'*, *labels=None*, ***kwargs*)

Store and manipulate multiple sequence alignments.

msa must be a 2D Numpy character array. *labels* is a list of sequence labels (or titles). *mapping* should map label or part of label to sequence index in *msa* array. If *mapping* is not given, one will be build from *labels*.

countLabel(*label*)

Returns the number of sequences that *label* maps onto.

extend(*other*)

Adds *other* to this MSA.

getArray()

Returns a copy of the MSA character array.

getIndex(*label*)

Returns index of the sequence that *label* maps onto. If *label* maps onto multiple sequences or *label* is a list of labels, a list of indices is returned. If an index for a label is not found, return **None**.

getLabel(*index*, *full=False*)

Returns label of the sequence at given *index*. Residue numbers will be removed from the sequence label, unless *full* is **True**.

getLabels(*full=False*)

Returns all labels

getResnums(*index*)

Returns starting and ending residue numbers (*resnum*) for the sequence at given *index*.

getTitle()

Returns title of the instance.

isAligned()

Returns **True** if MSA is aligned.

iterLabels(*full=False*)

Yield sequence labels. By default the part of the label used for indexing sequences is yielded.

numIndexed()

Returns number of sequences that are indexed using the identifier part or all of their labels. The return value should be equal to number of sequences.

numResidues()

Returns number of residues (or columns in the MSA), if MSA is aligned.

numSequences()

Returns number of sequences.

setTitle(*title*)

Set title of the instance.

split

Return split label when iterating or indexing.

refineMSA(*msa*, *index=None*, *label=None*, *rowocc=None*, *seqid=None*, *colocc=None*, ***kwargs*)

Refine *msa* by removing sequences (rows) and residues (columns) that contain gaps.

Parameters

- **msa** (*MSA* (page 285)) – multiple sequence alignment
- **index** (*int*[873]) – remove columns that are gaps in the sequence with that index
- **label** (*str*[874]) – remove columns that are gaps in the sequence matching label, `msa.getIndex(label)` must return a sequence index, a PDB identifier is also acceptable
- **rowocc** (*float*[875]) – row occupancy, sequences with less occupancy will be removed after *label* refinement is applied
- **seqid** (*float*[876]) – keep unique sequences at specified sequence identity level, unique sequences are identified using *uniqueSequences()* (page 281)
- **colocc** (*float*[877]) – column occupancy, residue positions with less occupancy will be removed after other refinements are applied
- **keep** – keep columns corresponding to residues not resolved in the PDB structure, default is **False**, applies when *label* is a PDB identifier
- **type** – bool

For Pfam MSA data, *label* is UniProt entry name for the protein. You may also use PDB structure and chain identifiers, e.g. `'1p38'` or `'1p38A'`, for *label* argument and UniProt entry names will be parsed using *parsePDBHeader()* (page 240) function (see also *Polymer* (page 238) and *DBRef* (page 239)).

The order of refinements are applied in the order of arguments. If *label* and *unique* is specified, sequence matching *label* will be kept in the refined *MSA* (page 285) although it may be similar to some other sequence.

mergeMSA(**msa*, ***kwargs*)

Returns an *MSA* (page 285) obtained from merging parts of the sequences of proteins present in multiple *msa* instances. Sequences are matched based on protein identifiers found in the sequence labels. Order of sequences in the merged MSA will follow the order of sequences in the first *msa* instance. Note that protein identifiers that map to multiple sequences will be excluded.

MSAs with different identifiers can be merged with the *ignore_ids* kwarg. This only works when all MSAs have the same number of sequences.

Parameters

- **msa** (list, tuple, `ndarray`) – a set of *MSA* (page 285) objects to be analysed
- **ignore_ids** (*bool*[878]) – where to ignore identifiers instead of matching them Default is False

specMergeMSA(**msa*, ***kwargs*)

Returns an *MSA* (page 285) obtained from merging parts of the sequences of proteins present in multiple *msa* instances. Sequences are matched based on species section of protein identifiers found in the sequence labels. Order of sequences in the merged MSA will follow the order of sequences in the first *msa* instance. Note that protein identifiers that map to multiple sequences will be excluded.

[873] http://docs.python.org/library/functions.html#int
[874] http://docs.python.org/library/stdtypes.html#str
[875] http://docs.python.org/library/functions.html#float
[876] http://docs.python.org/library/functions.html#float
[877] http://docs.python.org/library/functions.html#float
[878] http://docs.python.org/library/functions.html#bool

3.11.8 MSA File

This module defines functions and classes for parsing, manipulating, and analyzing multiple sequence alignments.

class **MSAFile**(*msa*, *mode='r'*, *format=None*, *aligned=True*, ***kwargs*)
Handle MSA files in FASTA, SELEX, CLUSTAL and Stockholm formats.

msa may be a filename or a stream. Multiple sequence alignments can be read from or written in FASTA (`.fasta`), Stockholm (`.sth`), CLUSTAL (`.aln`), or SELEX (`.slx`) *format*. For specified extensions, *format* argument is not needed. If *aligned* is **True**, unaligned sequences in the file or stream will cause an `IOError`[879] exception. *filter*, a function that returns a boolean, can be used for filtering sequences, see *`MSAFile.setFilter()`* (page 287) for details. *slice* can be used to slice sequences, and is applied after filtering, see *`MSAFile.setSlice()`* (page 287) for details.

close()
Close the file. This method will not affect a stream.

getFilename()
Returns filename, or **None** if instance is handling a stream.

getFilter()
Returns function used for filtering sequences.

getFormat()
Returns file format.

getSlice()
Returns object used to slice sequences.

getTitle()
Returns title of the instance.

isAligned()
Returns **True** if MSA is aligned.

reset()
Returns to the beginning of the file.

setFilter(*filter*, *filter_full=False*)
Set function used for filtering sequences. *filter* will be applied to split sequence label, by default. If *filter_full* is **True**, filter will be applied to the full label.

setSlice(*slice*)
Set object used to *slice* sequences, which may be a `slice()` or a `list()` of numbers.

setTitle(*title*)
Set title of the instance.

write(*seq*)
Write *seq*, an *`Sequence`* (page 289) instance, into the MSA file.

closed
True for closed file.

format
Format of the MSA file.

splitSeqLabel(*label*)
Returns label, starting residue number, and ending residue number parsed from sequence label.

[879] http://docs.python.org/library/exceptions.html#IOError

parseMSA(*filename*, ***kwargs*)
Returns an *MSA* (page 285) instance that stores multiple sequence alignment and sequence labels parsed from Stockholm, SELEX, CLUSTAL, PIR, or FASTA format *filename* file, which may be a compressed file. Uncompressed MSA files are parsed using C code at a fraction of the time it would take to parse compressed files in Python.

writeMSA(*filename*, *msa*, ***kwargs*)
Returns *filename* containing *msa*, a *MSA* (page 285) or *MSAFile* (page 287) instance, in the specified *format*, which can be *SELEX*, *Stockholm*, or *FASTA*. If *compressed* is **True** or *filename* ends with `.gz`, a compressed file will be written. *MSA* (page 285) instances will be written using C function into uncompressed files.

Can also write *CLUSTAL* or *PIR* format files using Python functions.

3.11.9 Plotting Functions

This module defines MSA analysis functions.

showMSAOccupancy(*msa*, *occ='res'*, *indices=None*, *count=False*, ***kwargs*)
Show a bar plot of occupancy calculated for *MSA* (page 285) instance *msa* using *calcMSAOccupancy()* (page 279). *occ* may be `'res'` or `'col'`, or a a pre-calculated occupancy array. If x-axis *indices* are not specified, they will be inferred from *msa* or given *label* that may correspond to a sequence in the msa.

Occupancy is plotted using `bar()`[880] function.

showShannonEntropy(*entropy*, *indices=None*, ***kwargs*)
Show a bar plot of Shannon *entropy* array. *MSA* (page 285) instances or Numpy character arrays storing sequence alignments are also accepted as *entropy* argument, in which case *calcShannonEntropy()* (page 279) will be used for calculations. *indices* may be residue numbers, when not specified they will be inferred from *msa* or indices starting from 1 will be used.

Entropy is plotted using `bar()`[881] function.

showMutinfoMatrix(*mutinfo*, **args*, ***kwargs*)
Show a heatmap of mutual information array. *MSA* (page 285) instances or Numpy character arrays storing sequence alignment are also accepted as *mutinfo* argument, in which case *buildMutinfoMatrix()* (page 279) will be used for calculations. Note that x, y axes contain indices of the matrix starting from 1.

Mutual information is plotted using `imshow()`[882] function. vmin and vmax values can be set by user to achieve better signals using this function.

showDirectInfoMatrix(*dirinfo*, **args*, ***kwargs*)
Show a heatmap of direct information array. *MSA* (page 285) instances or Numpy character arrays storing sequence alignment are also accepted as *dirinfo* argument, in which case *buildDirectInfoMatrix()* (page 282) will be used for calculations. Note that x, y axes contain indices of the matrix starting from 1.

Direct information is plotted using `imshow()`[883] function. vmin and vmax values can be set by user to achieve better signals using this function.

showSCAMatrix(*scainfo*, **args*, ***kwargs*)
Show a heatmap of SCA (statistical coupling analysis) array. *MSA* (page 285) instances. blah

[880] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar
[881] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar
[882] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow
[883] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow

or Numpy character arrays storing sequence alignment are also accepted as *scainfo* argument, in which case *buildSCAMatrix()* (page 281) will be used for calculations. Note that x, y axes contain indices of the matrix starting from 1.

SCA information is plotted using imshow()[884] function. vmin and vmax values can be set by user to achieve better signals using this function.

3.11.10 Sequence

This module handles individual sequences.

class Sequence (**args*)

Handle individual sequences of an *MSA* (page 285) object

Depending on input arguments, instances may point to an *MSA* (page 285) object or store its own data.

copy()

Returns a copy of the instance that owns its sequence data.

getArray()

getIndex()

Returns sequence index or **None**.

getLabel (*full=False*)

Returns label of the sequence.

getMSA()

Returns *MSA* (page 285) instance or **None**.

getResnums (*gaps=False*, *report_match=False*)

Returns list of residue numbers associated with non-gapped *seq*. When *gaps* is **True**, return a list containing the residue numbers with gaps appearing as **None**.

Residue numbers are inferred from the full label if possible. When the label does not contain residue number information, a range of numbers starting from 1 is returned.

numGaps()

Returns number of gap characters.

numResidues()

Returns the number of alphabet characters.

3.12 Trajectory I/O

This module defines classes for handling trajectory files in DCD format.

3.12.1 Parse/write DCD files

- *DCDFile* (page 290)
- *parseDCD()* (page 292)
- *writeDCD()* (page 293)

3.12.2 Parse structure files

- *parsePSF()* (page 294)

[884] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow

3.12.3 Handle multiple files

- *Trajectory* (page 296)

3.12.4 Handle frame data

- *Frame* (page 293)

3.12.5 Examples

Following examples show how to use trajectory classes and functions:

- Trajectory Analysis[885]
- Trajectory Analysis II[886]
- Essential Dynamics Analysis[887]

3.12.6 DCD File

This module defines classes for handling trajectory files in DCD format[888].

class DCDFile (*filename*, *mode='rb'*, ***kwargs*)

A class for reading and writing DCD files. DCD header and first frame is parsed at instantiation. Coordinates from the first frame is set as the reference coordinate set. This class has been tested for 32-bit DCD files. 32-bit floating-point coordinate array can be casted automatically to a specified type, such as 64-bit float, using *astype* keyword argument, i.e. `astype=float`, using `ndarray.astype()` method.

Open *filename* for reading (default, `mode="r"`), writing (`mode="w"`), or appending (`mode="r+"` or `mode="a"`). Binary mode option will be appended automatically.

close()

Close trajectory file.

flush()

Flush the internal output buffer.

getAtoms()

Returns associated/selected atoms.

getCoords()

Returns a copy of reference coordinates for (selected) atoms.

getCoordsets(*indices=None*)

Returns coordinate sets at given *indices*. *indices* may be an integer, a list of ordered integers or **None**. **None** returns all coordinate sets. If a list of indices is given, unique numbers will be selected and sorted. That is, this method will always return unique coordinate sets in the order they appear in the trajectory file. Shape of the coordinate set array is (n_sets, n_atoms, 3).

getFilename(*absolute=False*)

Returns relative path to the current file. For absolute path, pass `absolute=True` argument.

getFirstTimestep()

Returns first timestep value.

[885] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory.html#trajectory
[886] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory2.html#trajectory2
[887] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda
[888] http://www.ks.uiuc.edu/Research/namd/2.6/ug/node13.html

getFrame(*index*)
Returns frame at given *index*.

getFrameFreq()
Returns timesteps between frames.

getLinked()
Returns linked *AtomGroup* (page 38) instance, or **None** if a link is not established.

getRemarks()
Returns remarks parsed from DCD file.

getTimestep()
Returns timestep size.

getTitle()
Returns title of the ensemble.

getWeights()
Returns a copy of weights of (selected) atoms.

goto(*n*)
Go to the frame at index *n*. `n=0` will rewind the trajectory to the beginning, same as calling *reset()* (page 292) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of *next()* (page 291) or *nextCoordset()* (page 291) methods is called.

hasUnitcell()
Returns **True** if trajectory has unitcell data.

isLinked()
Returns **True** if trajectory is linked to an *AtomGroup* (page 38) instance.

iterCoordsets()
Yield coordinate sets for (selected) atoms. Reference coordinates are not included. Iteration starts from the next frame in line.

link(**ag*)
Link, return, or unlink an *AtomGroup* (page 38) instance. When a link to *ag* is established, coordinates of new frames parsed from the trajectory file will be set as the coordinates of *ag* and this will update coordinates of all selections and atom subsets pointing to it. At link time, if *ag* does not have any coordinate sets and reference coordinates of the trajectory is set, reference coordinates of the trajectory will be passed to *ag*. To break an established link, pass **None** argument, or to return the linked atom group instance, call with no arguments.

> **Warning:** Every time a frame is parsed from the trajectory, all coordinate sets present in the linked *AtomGroup* (page 38) will be overwritten.

next()
Returns next coordinate set in a *Frame* (page 293) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Returns next coordinate set.

nextIndex()
Returns the index of the next frame.

numAtoms()
Returns number of atoms.

numCoordsets()
Returns number of frames.

numFixed()
Returns number of fixed atoms.

numFrames()
Returns number of frames.

numSelected()
Returns number of selected atoms. A subset of atoms can be selected by passing a selection to *setAtoms()* (page 292).

reset()
Go to first frame at index 0. First frame will not be parsed until one of *next()* (page 291) or *nextCoordset()* (page 291) methods is called.

setAtoms(*atoms*)
Set *atoms* or specify a selection of atoms to be considered in calculations and coordinate requests. When a selection is set, corresponding subset of coordinates will be considered in, for example, alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, *Trajectory* (page 296) and *Frame* (page 293) instances become suitable arguments for *writePDB()* (page 270). Passing **None** as *atoms* argument will deselect atoms. Note that setting atoms does not change the reference coordinates of the trajectory. To change the reference, use *setCoords()* (page 292) method.

setCoords(*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with *getCoords()* (page 290) method, or a Numpy array with suitable data type, shape, and dimensionality.

setTitle(*title*)
Set title of the ensemble.

setWeights(*weights*)
Set atomic weights.

skip(*n*)
Skip *n* frames. *n* must be a positive integer. Skipping some frames will only change the next frame index (*nextIndex()* (page 291)) Next frame will not be parsed until one of *next()* (page 291) or *nextCoordset()* (page 291) methods is called.

write(*coords*, *unitcell=None*, ***kwargs*)
Write *coords* to a file open in 'a' or 'w' mode. *coords* may be a NUmpy array or a ProDy object that stores or points to coordinate data. Note that all coordinate sets of ProDy object will be written. Number of atoms will be determined from the file or based on the size of the first coordinate set written. If *unitcell* is provided for the first coordinate set, it will be expected for the following coordinate sets as well. If *coords* is an *Atomic* (page 47) or *Ensemble* (page 202) all coordinate sets will be written.

Following keywords are used when writing the first coordinate set:

Parameters

- **timestep** – timestep used for integration, default is 1
- **firsttimestep** – number of the first timestep, default is 0
- **framefreq** – number of timesteps between frames, default is 1

parseDCD(*filename*, *start=None*, *stop=None*, *step=None*, *astype=None*)
Parse CHARMM format DCD files (also NAMD 2.1 and later). Returns an `Ensemble` instance.

Conformations in the ensemble will be ordered as they appear in the trajectory file. Use *DCDFile* (page 290) class for parsing coordinates of a subset of atoms.

Parameters

- **filename** (*str*[889]) – DCD filename
- **start** (*int*[890]) – index of first frame to read
- **stop** (*int*[891]) – index of the frame that stops reading
- **step** (*int*[892]) – steps between reading frames, default is 1 meaning every frame
- **astype** (*type*[893]) – cast coordinate array to specified type

writeDCD(*filename*, *trajectory*, *start=None*, *stop=None*, *step=None*, *align=False*)
Write 32-bit CHARMM format DCD file (also NAMD 2.1 and later). *trajectory* can be an `Trajectory`, *DCDFile* (page 290), or `Ensemble` instance. *filename* is returned upon successful output of file.

3.12.7 Frame

This module defines a class for handling trajectory frames.

class Frame(*traj*, *index*, *coords*, *unitcell=None*, *velocs=None*)
A class for storing trajectory frame coordinates and provide methods acting on them.

getAtoms()
Returns associated/selected atoms.

getCoords()
Returns a copy of coordinates of (selected) atoms.

getDeviations()
Returns deviations from the trajectory reference coordinates.

getIndex()
Returns index.

getRMSD()
Returns RMSD from the trajectory reference coordinates. If weights for the trajectory are set, weighted RMSD will be returned.

getTrajectory()
Returns the trajectory.

getUnitcell()
Returns a copy of unitcell array.

getVelocities()
Returns a copy of velocities of (selected) atoms.

getWeights()
Returns coordinate weights for selected atoms.

numAtoms()
Returns number of atoms.

numSelected()
Returns number of selected atoms.

[889] http://docs.python.org/library/stdtypes.html#str
[890] http://docs.python.org/library/functions.html#int
[891] http://docs.python.org/library/functions.html#int
[892] http://docs.python.org/library/functions.html#int
[893] http://docs.python.org/library/functions.html#type

superpose()
Superpose frame onto the trajectory reference coordinates. Note that transformation matrix is calculated based on selected atoms and applied to all atoms. If atom weights for the trajectory are set, they will be used to calculate the transformation.

3.12.8 PSF File

This module defines a function for parsing protein structure files in PSF format[894].

parsePSF(*filename*, *title=None*, *ag=None*)
Returns an `AtomGroup` (page 38) instance storing data parsed from X-PLOR format PSF file *filename*. Atom and bond information is parsed from the file. If *title* is not given, *filename* will be set as the title of the `AtomGroup` (page 38) instance. An `AtomGroup` (page 38) instance may be provided as *ag* argument. When provided, *ag* must have the same number of atoms in the same order as the file. Data from PSF file will be added to the *ag*. This may overwrite present data if it overlaps with PSF file content.

This function now includes the angles, dihedrals, and impropers sections as well as donors, acceptors and crossterms!

writePSF(*filename*, *atoms*)
Write atoms in X-PLOR format PSF file with name *filename* and return *filename*. This function will write available atom and bond information only.

3.12.9 Trajectory Base

This module defines base class for trajectory handling.

class TrajBase(*title='Unknown'*)
Base class for `Trajectory` (page 296) and `TrajFile` (page 298). Derived classes must implement functions described in this class.

close()
Close trajectory file.

getAtoms()
Returns associated/selected atoms.

getCoords()
Returns a copy of reference coordinates for (selected) atoms.

getCoordsets(*indices=None*)
Returns coordinate sets at given *indices*. *indices* may be an integer, a list of ordered integers or **None**. **None** returns all coordinate sets. If a list of indices is given, unique numbers will be selected and sorted. That is, this method will always return unique coordinate sets in the order they appear in the trajectory file. Shape of the coordinate set array is (n_sets, n_atoms, 3).

getFrame(*index*)
Returns frame at given *index*.

getLinked()
Returns linked `AtomGroup` (page 38) instance, or **None** if a link is not established.

getTitle()
Returns title of the ensemble.

getWeights()
Returns a copy of weights of (selected) atoms.

[894] http://www.ks.uiuc.edu/Training/Tutorials/namd/namd-tutorial-unix-html/node23.html

goto(*n*)
Go to the frame at index *n*. `n=0` will rewind the trajectory to the beginning, same as calling *reset()* (page 295) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of *next()* (page 295) or *nextCoordset()* (page 295) methods is called.

hasUnitcell()
Returns **True** if trajectory has unitcell data.

isLinked()
Returns **True** if trajectory is linked to an *AtomGroup* (page 38) instance.

iterCoordsets()
Yield coordinate sets for (selected) atoms. Reference coordinates are not included. Iteration starts from the next frame in line.

link(**ag*)
Link, return, or unlink an *AtomGroup* (page 38) instance. When a link to *ag* is established, coordinates of new frames parsed from the trajectory file will be set as the coordinates of *ag* and this will update coordinates of all selections and atom subsets pointing to it. At link time, if *ag* does not have any coordinate sets and reference coordinates of the trajectory is set, reference coordinates of the trajectory will be passed to *ag*. To break an established link, pass **None** argument, or to return the linked atom group instance, call with no arguments.

> **Warning:** Every time a frame is parsed from the trajectory, all coordinate sets present in the linked *AtomGroup* (page 38) will be overwritten.

next()
Returns next coordinate set in a *Frame* (page 293) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Returns next coordinate set.

nextIndex()
Returns the index of the next frame.

numAtoms()
Returns number of atoms.

numCoordsets()
Returns number of frames.

numFrames()
Returns number of frames.

numSelected()
Returns number of selected atoms. A subset of atoms can be selected by passing a selection to *setAtoms()* (page 295).

reset()
Go to first frame at index 0. First frame will not be parsed until one of *next()* (page 295) or *nextCoordset()* (page 295) methods is called.

setAtoms(*atoms*)
Set *atoms* or specify a selection of atoms to be considered in calculations and coordinate requests. When a selection is set, corresponding subset of coordinates will be considered in, for example, alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, *Trajectory* (page 296) and *Frame* (page 293) instances become suitable arguments for *writePDB()* (page 270). Passing **None** as *atoms* argument will

deselect atoms. Note that setting atoms does not change the reference coordinates of the trajectory. To change the reference, use *setCoords()* (page 296) method.

setCoords(*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with *getCoords()* (page 294) method, or a Numpy array with suitable data type, shape, and dimensionality.

setTitle(*title*)
Set title of the ensemble.

setWeights(*weights*)
Set atomic weights.

skip(*n*)
Skip *n* frames. *n* must be a positive integer. Skipping some frames will only change the next frame index (*nextIndex()* (page 295)) Next frame will not be parsed until one of *next()* (page 295) or *nextCoordset()* (page 295) methods is called.

3.12.10 Trajectory

This module defines a class for handling multiple trajectories.

class Trajectory(*name*, ***kwargs*)
A class for handling trajectories in multiple files.

Trajectory can be instantiated with a *name* or a filename. When name is a valid path to a trajectory file it will be opened for reading.

addFile(*filename*, ***kwargs*)
Add a file to the trajectory instance. Currently only DCD files are supported.

close()
Close trajectory file.

getAtoms()
Returns associated/selected atoms.

getCoords()
Returns a copy of reference coordinates for (selected) atoms.

getCoordsets(*indices=None*)
Returns coordinate sets at given *indices*. *indices* may be an integer, a list of ordered integers or **None**. **None** returns all coordinate sets. If a list of indices is given, unique numbers will be selected and sorted. That is, this method will always return unique coordinate sets in the order they appear in the trajectory file. Shape of the coordinate set array is (n_sets, n_atoms, 3).

getFilenames(*absolute=False*)
Returns list of filenames opened for reading.

getFirstTimestep()
Returns list of first timestep values, one number from each file.

getFrameFreq()
Returns list of timesteps between frames, one number from each file.

getLinked()
Returns linked *AtomGroup* (page 38) instance, or **None** if a link is not established.

getTimestep()
Returns list of timestep sizes, one number from each file.

getTitle()
Returns title of the ensemble.

getWeights()
Returns a copy of weights of (selected) atoms.

goto(*n*)
Go to the frame at index *n*. `n=0` will rewind the trajectory to the beginning, same as calling *reset()* (page 297) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of *next()* (page 297) or *nextCoordset()* (page 297) methods is called.

hasUnitcell()
Returns **True** if trajectory has unitcell data.

isLinked()
Returns **True** if trajectory is linked to an *AtomGroup* (page 38) instance.

iterCoordsets()
Yield coordinate sets for (selected) atoms. Reference coordinates are not included. Iteration starts from the next frame in line.

link(**ag*)
Link, return, or unlink an *AtomGroup* (page 38) instance. When a link to *ag* is established, coordinates of new frames parsed from the trajectory file will be set as the coordinates of *ag* and this will update coordinates of all selections and atom subsets pointing to it. At link time, if *ag* does not have any coordinate sets and reference coordinates of the trajectory is set, reference coordinates of the trajectory will be passed to *ag*. To break an established link, pass **None** argument, or to return the linked atom group instance, call with no arguments.

> **Warning:** Every time a frame is parsed from the trajectory, all coordinate sets present in the linked *AtomGroup* (page 38) will be overwritten.

next()
Returns next coordinate set in a *Frame* (page 293) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Returns next coordinate set.

nextIndex()
Returns the index of the next frame.

numAtoms()
Returns number of atoms.

numCoordsets()
Returns number of frames.

numFiles()
Returns number of open trajectory files.

numFixed()
Returns a list of fixed atom numbers, one from each file.

numFrames()
Returns number of frames.

numSelected()
Returns number of selected atoms. A subset of atoms can be selected by passing a selection to *setAtoms()* (page 298).

reset()
Go to first frame at index 0. First frame will not be parsed until one of *next()* (page 297) or

nextCoordset() (page 297) methods is called.

setAtoms (*atoms*)
Set *atoms* or specify a selection of atoms to be considered in calculations and coordinate requests. When a selection is set, corresponding subset of coordinates will be considered in, for example, alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, *Trajectory* (page 296) and *Frame* (page 293) instances become suitable arguments for *writePDB()* (page 270). Passing **None** as *atoms* argument will deselect atoms. Note that setting atoms does not change the reference coordinates of the trajectory. To change the reference, use *setCoords()* (page 298) method.

setCoords (*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with *getCoords()* (page 296) method, or a Numpy array with suitable data type, shape, and dimensionality.

setTitle (*title*)
Set title of the ensemble.

setWeights (*weights*)
Set atomic weights.

skip (*n*)
Skip *n* frames. *n* must be a positive integer. Skipping some frames will only change the next frame index (*nextIndex()* (page 297)) Next frame will not be parsed until one of *next()* (page 297) or *nextCoordset()* (page 297) methods is called.

3.12.11 Trajectory File

This module defines a base class for format specific trajectory classes.

class TrajFile (*filename*, *mode='r'*)
A base class for trajectory file classes:

- *DCDFile* (page 290)

Open *filename* for reading (default, `mode="r"`), writing (`mode="w"`), or appending (`mode="r+"` or `mode="a"`). Binary mode option will be appended automatically.

close ()
Close trajectory file.

getAtoms ()
Returns associated/selected atoms.

getCoords ()
Returns a copy of reference coordinates for (selected) atoms.

getCoordsets (*indices=None*)
Returns coordinate sets at given *indices*. *indices* may be an integer, a list of ordered integers or **None**. **None** returns all coordinate sets. If a list of indices is given, unique numbers will be selected and sorted. That is, this method will always return unique coordinate sets in the order they appear in the trajectory file. Shape of the coordinate set array is (n_sets, n_atoms, 3).

getFilename (*absolute=False*)
Returns relative path to the current file. For absolute path, pass `absolute=True` argument.

getFirstTimestep ()
Returns first timestep value.

getFrame (*index*)
Returns frame at given *index*.

getFrameFreq()
Returns timesteps between frames.

getLinked()
Returns linked *AtomGroup* (page 38) instance, or **None** if a link is not established.

getTimestep()
Returns timestep size.

getTitle()
Returns title of the ensemble.

getWeights()
Returns a copy of weights of (selected) atoms.

goto(*n*)
Go to the frame at index *n*. `n=0` will rewind the trajectory to the beginning, same as calling *reset()* (page 300) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of *next()* (page 299) or *nextCoordset()* (page 299) methods is called.

hasUnitcell()
Returns **True** if trajectory has unitcell data.

isLinked()
Returns **True** if trajectory is linked to an *AtomGroup* (page 38) instance.

iterCoordsets()
Yield coordinate sets for (selected) atoms. Reference coordinates are not included. Iteration starts from the next frame in line.

link(**ag*)
Link, return, or unlink an *AtomGroup* (page 38) instance. When a link to *ag* is established, coordinates of new frames parsed from the trajectory file will be set as the coordinates of *ag* and this will update coordinates of all selections and atom subsets pointing to it. At link time, if *ag* does not have any coordinate sets and reference coordinates of the trajectory is set, reference coordinates of the trajectory will be passed to *ag*. To break an established link, pass **None** argument, or to return the linked atom group instance, call with no arguments.

> **Warning:** Every time a frame is parsed from the trajectory, all coordinate sets present in the linked *AtomGroup* (page 38) will be overwritten.

next()
Returns next coordinate set in a *Frame* (page 293) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Returns next coordinate set.

nextIndex()
Returns the index of the next frame.

numAtoms()
Returns number of atoms.

numCoordsets()
Returns number of frames.

numFixed()
Returns number of fixed atoms.

numFrames()
Returns number of frames.

numSelected()
Returns number of selected atoms. A subset of atoms can be selected by passing a selection to *setAtoms()* (page 300).

reset()
Go to first frame at index 0. First frame will not be parsed until one of *next()* (page 299) or *nextCoordset()* (page 299) methods is called.

setAtoms(*atoms*)
Set *atoms* or specify a selection of atoms to be considered in calculations and coordinate requests. When a selection is set, corresponding subset of coordinates will be considered in, for example, alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, *Trajectory* (page 296) and *Frame* (page 293) instances become suitable arguments for *writePDB()* (page 270). Passing **None** as *atoms* argument will deselect atoms. Note that setting atoms does not change the reference coordinates of the trajectory. To change the reference, use *setCoords()* (page 300) method.

setCoords(*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with *getCoords()* (page 298) method, or a Numpy array with suitable data type, shape, and dimensionality.

setTitle(*title*)
Set title of the ensemble.

setWeights(*weights*)
Set atomic weights.

skip(*n*)
Skip *n* frames. *n* must be a positive integer. Skipping some frames will only change the next frame index (*nextIndex()* (page 299)) Next frame will not be parsed until one of *next()* (page 299) or *nextCoordset()* (page 299) methods is called.

3.13 ProDy Utilities

This module provides utility functions and classes for handling files, logging, type checking, etc. Contents of this module are not included in ProDy namespace, as it is not safe to import them all due to name conflicts. Required or classes should be imported explicitly, e.g. `from prody.utilities import PackageLogger, openFile`.

3.13.1 Package utilities

- *PackageLogger* (page 308)
- *PackageSettings* (page 314)
- *getPackagePath()* (page 314)
- *setPackagePath()* (page 314)

3.13.2 Type/Value checkers

- *checkCoords()* (page 306)
- *checkWeights()* (page 307)
- *checkTypes()* (page 307)

3.13.3 Path/file handling

- *gunzip()* (page 312)
- *openFile()* (page 312)
- *openDB()* (page 312)
- *openSQLite()* (page 312)
- *openURL()* (page 312)
- *copyFile()* (page 312)
- *isExecutable()* (page 312)
- *isReadable()* (page 312)
- *isWritable()* (page 312)
- *makePath()* (page 312)
- *relpath()* (page 313)
- *which()* (page 313)
- *pickle()* (page 313)
- *unpickle()* (page 313)
- *glob()* (page 313)

3.13.4 Documentation tools

- *joinRepr()* (page 307)
- *joinRepr()* (page 307)
- *joinTerms()* (page 307)
- *tabulate()* (page 307)
- *wrapText()* (page 308)

3.13.5 Miscellaneous tools

- *rangeString()* (page 310)
- *alnum()* (page 310)
- *importLA()* (page 310)
- *dictElement()* (page 310)

3.13.6 Tree Construction Tools

Classes and methods for tree construction.

class **DistanceMatrix**(*names*, *matrix=None*)

Distance matrix class that can be used for distance based tree algorithms.

All diagonal elements will be zero no matter what the users provide.

Initialize the class.

format_phylip(*handle*)
Write data in Phylip format to a given file-like object or handle.

The output stream is the input distance matrix format used with Phylip programs (e.g. 'neighbor'). See: http://evolution.genetics.washington.edu/phylip/doc/neighbor.html

Parameters

handle [file or file-like object] A writeable file handle or other object supporting the 'write' method, such as StringIO or sys.stdout. On Python 3, should be open in text mode.

class TreeConstructor
Base class for all tree constructor.

build_tree(*msa*)
Caller to built the tree from a MultipleSeqAlignment object.

This should be implemented in subclass.

class DistanceTreeConstructor(*method='nj'*)
Distance based tree constructor.

Parameters

method [str] Distance tree construction method, 'nj'(default) or 'upgma'.

Loading a small PHYLIP alignment from which to compute distances, and then build a upgma Tree:

```
from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
from Bio.Phylo.TreeConstruction import DistanceCalculator
from Bio import AlignIO
aln = AlignIO.read(open('TreeConstruction/msa.phy'), 'phylip')
constructor = DistanceTreeConstructor()
calculator = DistanceCalculator('identity')
dm = calculator.get_distance(aln)
upgmatree = constructor.upgma(dm)
print(upgmatree)
```

Output:

```
Tree(rooted=True)
    Clade(branch_length=0, name='Inner4')
        Clade(branch_length=0.18749999999999994, name='Inner1')
            Clade(branch_length=0.07692307692307693, name='Epsilon')
            Clade(branch_length=0.07692307692307693, name='Delta')
        Clade(branch_length=0.11057692307692304, name='Inner3')
            Clade(branch_length=0.038461538461538464, name='Inner2')
                Clade(branch_length=0.11538461538461536, name='Gamma')
                Clade(branch_length=0.11538461538461536, name='Beta')
            Clade(branch_length=0.15384615384615383, name='Alpha')
```

Build a NJ Tree:

```
njtree = constructor.nj(dm)
print(njtree)
```

Output:

```
Tree(rooted=False)
    Clade(branch_length=0, name='Inner3')
        Clade(branch_length=0.18269230769230765, name='Alpha')
        Clade(branch_length=0.04807692307692307, name='Beta')
```

```
            Clade(branch_length=0.04807692307692307, name='Inner2')
                Clade(branch_length=0.27884615384615385, name='Inner1')
                    Clade(branch_length=0.051282051282051266, name='Epsilon')
                    Clade(branch_length=0.10256410256410259, name='Delta')
                Clade(branch_length=0.14423076923076922, name='Gamma')
```

Initialize the class.

nj(*distance_matrix*)
Construct and return a Neighbor Joining tree.

Parameters

distance_matrix [DistanceMatrix] The distance matrix for tree construction.

upgma(*distance_matrix*)
Construct and return an UPGMA tree.

Constructs and returns an Unweighted Pair Group Method with Arithmetic mean (UPGMA) tree.

Parameters

distance_matrix [DistanceMatrix] The distance matrix for tree construction.

3.13.7 Additional utilities

This module defines miscellaneous utility functions that is public to users.

calcTree(*names*, *distance_matrix*, *method='upgma'*, *linkage=False*)
Given a distance matrix, it creates an returns a tree structure.

Parameters

- **names** (list, `ndarray`) – a list of names
- **distance_matrix** (`ndarray`) – a square matrix with length of ensemble. If numbers does not match *names* it will raise an error
- **method** (*str*[895]) – method used for constructing the tree. Acceptable options are `"upgma"`, `"nj"`, or methods supported by `linkage()`[896] such as `"single"`, `"average"`, `"ward"`, etc. Default is `"upgma"`
- **linkage** (*bool*[897]) – whether the linkage matrix is returned. Note that NJ trees do not support linkage

clusterMatrix(*distance_matrix=None*, *similarity_matrix=None*, *labels=None*, *return_linkage=None*, ***kwargs*)
Cluster a distance matrix using scipy.cluster.hierarchy and return the sorted matrix, indices used for sorting, sorted labels (if **labels** are passed), and linkage matrix (if **return_linkage** is **True**).

Parameters

- **distance_matrix** (`ndarray`) – an N-by-N matrix containing some measure of distance such as 1. - seqid_matrix (Hamming distance), rmsds, or distances in PCA space
- **similarity_matrix** (`ndarray`) – an N-by-N matrix containing some measure of similarity such as sequence identity, mode-mode overlap, or spectral overlap. Each element will be subtracted from 1. to get distance, so make sure this is reasonable.

[895] http://docs.python.org/library/stdtypes.html#str
[896] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.cluster.hierarchy.linkage.html#scipy.cluster.hierarchy.linkage
[897] http://docs.python.org/library/functions.html#bool

- **labels** (*list*[898]) – labels for each matrix row that can be returned sorted
- **no_plot** (*bool*[899]) – if **True**, don't plot the dendrogram. default is **True**
- **reversed** (*bool*[900]) – if set to **True**, then the sorting indices will be reversed.

Other arguments for `linkage()` and `dendrogram()` can also be provided and will be taken as **kwargs**.

showLines(**args*, ***kwargs*)

Show 1-D data using `plot()`.

Parameters

- **x** (`ndarray`) – (optional) x coordinates. *x* can be an 1-D array or a 2-D matrix of column vectors.
- **y** (`ndarray`) – data array. *y* can be an 1-D array or a 2-D matrix of column vectors.
- **dy** (`ndarray`) – an array of variances of *y* which will be plotted as a band along *y*. It should have the same shape with *y*.
- **lower** (`ndarray`) – an array of lower bounds which will be plotted as a band along *y*. It should have the same shape with *y* and should be paired with *upper*.
- **upper** (`ndarray`) – an array of upper bounds which will be plotted as a band along *y*. It should have the same shape with *y* and should be paired with *lower*.
- **alpha** (*float*[901]) – the transparency of the band(s) for plotting *dy*.
- **beta** (*float*[902]) – the transparency of the band(s) for plotting *miny* and *maxy*.
- **ticklabels** (*list*[903]) – user-defined tick labels for x-axis.

showMatrix(*matrix*, *x_array=None*, *y_array=None*, ***kwargs*)

Show a matrix using `imshow()`[904] or `scatter()`[905] if *markersize* is provided.

Curves on x- and y-axis can be added.

Parameters

- **matrix** (`ndarray`) – matrix to be displayed
- **x_array** (`ndarray`) – data to be plotted above the matrix
- **y_array** (`ndarray`) – data to be plotted on the left side of the matrix
- **percentile** (*float*[906]) – a percentile threshold to remove outliers, i.e. only showing data within *p*-th to *100-p*-th percentile
- **interactive** (*bool*[907]) – turn on or off the interactive options
- **xtickrotation** (*float*[908]) – how much to rotate the xticklabels in degrees default is 0

[898] http://docs.python.org/library/stdtypes.html#list
[899] http://docs.python.org/library/functions.html#bool
[900] http://docs.python.org/library/functions.html#bool
[901] http://docs.python.org/library/functions.html#float
[902] http://docs.python.org/library/functions.html#float
[903] http://docs.python.org/library/stdtypes.html#list
[904] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.imshow.html#matplotlib.axes.Axes.imshow
[905] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.scatter.html#matplotlib.axes.Axes.scatter
[906] http://docs.python.org/library/functions.html#float
[907] http://docs.python.org/library/functions.html#bool
[908] http://docs.python.org/library/functions.html#float

- **markersize** (*float*[909]) – size of square markers for using `scatter()`[910] to help show matrices with small data regions compared to zeros. Note only non-zeros are plotted so the colorbar range may change if not using norm Default is None, which results in using `imshow()`[911]

reorderMatrix(*names*, *matrix*, *tree*, *axis=None*)

Reorder a matrix based on a tree and return the reordered matrix and indices for reordering other things.

Parameters

- **names** (*list*[912]) – a list of names associated with the rows of the matrix These names must match the ones used to generate the tree
- **matrix** (ndarray) – any square matrix
- **tree** (Tree) – any tree from *calcTree()* (page 303)
- **axis** (*int*[913]) – along which axis the matrix should be reordered. Default is **None** which reorder along all the axes

findSubgroups(*tree*, *c*, *method='naive'*, ***kwargs*)

Divide **tree** into subgroups using a criterion **method** and a cutoff **c**. Returns a list of lists with labels divided into subgroups.

getCoords(*data*)

Get coordinates from *data* if possible and handle errors well.

Parameters **data** (numpy.ndarray, Atomic, Ensemble, Trajectory) – a coordinate set or an object with `getCoords` method

getLinkage(*names*, *tree*)

Obtain the `linkage()`[914] matrix encoding `tree`.

Parameters

- **names** (list, ndarray) – a list of names, the order determines the values in the linkage matrix
- **tree** (Tree) – tree to be converted

getTreeFromLinkage(*names*, *linkage*)

Obtain the tree encoded by `linkage`.

Parameters

- **names** (list, ndarray) – a list of names, the order should correspond to the values in linkage
- **linkage** (ndarray) – linkage matrix

clusterSubfamilies(*similarities*, *n_clusters=0*, *linkage='all'*, *method='tsne'*, *cutoff=0.0*, ***kwargs*)

Perform clustering based on members of the *ensemble* projected into lower a reduced dimension.

Parameters

[909] http://docs.python.org/library/functions.html#float
[910] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.scatter.html#matplotlib.axes.Axes.scatter
[911] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.axes.Axes.imshow.html#matplotlib.axes.Axes.imshow
[912] http://docs.python.org/library/stdtypes.html#list
[913] http://docs.python.org/library/functions.html#int
[914] http://docs.scipy.org/doc/scipy/reference/reference/generated/scipy.cluster.hierarchy.linkage.html#scipy.cluster.hierarchy.linkage

- **similarities** (ndarray) – a matrix of similarities for each structure in the ensemble, such as RMSD-matrix, dynamics-based spectral overlap, sequence similarity
- **n_clusters** (*int*[915]) – the number of clusters to generate. If **0**, will scan a range of number of clusters and return the best one based on highest silhouette score. Default is **0**.
- **linkage** (str, list, tuple, ndarray) – if **all**, will test all linkage types (ward, average, complete, single). Otherwise will use only the one(s) given as input. Default is **all**.
- **method** (*str*[916]) – if set to **spectral**, will generate a Kirchoff matrix based on the cutoff value given and use that as input as clustering instead of the values themselves. Default is **tsne**.
- **cutoff** (*float*[917]) – only used if *method* is set to **spectral**. This value is used for generating the Kirchoff matrix to use for generating clusters when doing spectral clustering. Default is **0.0**.

calcRMSDclusters(*rmsd_matrix*, *c*, *labels=None*)

Divide **rmsd_matrix** into clusters using the gromos method with a cutoff **c** as implemented in gromacs (see https://manual.gromacs.org/documentation/current/onlinehelp/gmx-cluster.html)

Returns a list of lists with labels divided into clusters.

calcGromosClusters(*rmsd_matrix*, *c*, *labels=None*)

Divide **rmsd_matrix** into clusters using the gromos method with a cutoff **c** as implemented in gromacs (see https://manual.gromacs.org/documentation/current/onlinehelp/gmx-cluster.html)

Returns a list of lists with labels divided into clusters.

calcGromacsClusters(*rmsd_matrix*, *c*, *labels=None*)

Divide **rmsd_matrix** into clusters using the gromos method with a cutoff **c** as implemented in gromacs (see https://manual.gromacs.org/documentation/current/onlinehelp/gmx-cluster.html)

Returns a list of lists with labels divided into clusters.

3.13.8 Type Checkers

This module defines functions for type, value, and/or attribute checking.

checkCoords(*coords*, *csets=False*, *natoms=None*, *dtype=(<type 'float'>, <type 'numpy.float32'>)*, *name='coords'*)

Returns **True** if shape, dimensionality, and data type of *coords* array are as expected.

Parameters

- **coords** – coordinate array
- **csets** – whether multiple coordinate sets (i.e. `.ndim in (2, 3)`) are allowed, default is **False**
- **natoms** – number of atoms, if **None** number of atoms is not checked
- **dtype** – allowed data type(s), default is `(float, numpy.float32)`, if **None** data type is not checked
- **name** – name of the coordinate argument

[915] http://docs.python.org/library/functions.html#int
[916] http://docs.python.org/library/stdtypes.html#str
[917] http://docs.python.org/library/functions.html#float

Raises `TypeError`[918] when *coords* is not an instance of `numpy.ndarray`

Raises `ValueError`[919] when wrong shape, dimensionality, or data type is encountered

checkWeights(*weights*, *natoms*, *ncsets=None*, *dtype=<type 'float'>*)
Returns *weights* if it has correct shape ([ncsets,]natoms, 1). after its shape and data type is corrected. otherwise raise an exception. All items of *weights* must be greater than zero.

checkTypes(*args*, ***types*)
Returns **True** if types of all *args* match those given in *types*.

Raises `TypeError`[920] when type of an argument is not one of allowed types

```
    def incr(n, i):
        '''Return sum of *n* and *i*.'''

        checkTypes(locals(), n=(float, int), i=(float, int))
        return n + i
```

checkAnisous(*anisous*, *csets=False*, *natoms=None*, *dtype=(<type 'float'>, <type 'numpy.float32'>)*, *name='anisous'*)
Returns **True** if shape, dimensionality, and data type of *anisous* array are as expected.

Parameters

- **anisous** – anisous array
- **csets** – whether multiple coordinate sets (i.e. `.ndim in (2, 3)`) are allowed, default is **False**
- **natoms** – number of atoms, if **None** number of atoms is not checked
- **dtype** – allowed data type(s), default is `(float, numpy.float32)`, if **None** data type is not checked
- **name** – name of the coordinate argument

Raises `TypeError`[921] when *anisous* is not an instance of `numpy.ndarray`

Raises `ValueError`[922] when wrong shape, dimensionality, or data type is encountered

3.13.9 Documentation Tools

This module defines miscellaneous utility functions.

joinLinks(*links*, *sep=', '*, *last=None*, *sort=False*)
Returns a string joining *links* as reStructuredText.

joinRepr(*items*, *sep=', '*, *last=None*, *sort=False*)
Returns a string joining representations of *items*.

joinTerms(*terms*, *sep=', '*, *last=None*, *sort=False*)
Returns a string joining *terms* as reStructuredText.

tabulate(**cols*, ***kwargs*)
Returns a table for columns of data.

Parameters

[918] http://docs.python.org/library/exceptions.html#TypeError
[919] http://docs.python.org/library/exceptions.html#ValueError
[920] http://docs.python.org/library/exceptions.html#TypeError
[921] http://docs.python.org/library/exceptions.html#TypeError
[922] http://docs.python.org/library/exceptions.html#ValueError

- **header** (*bool*[923]) – make first row a header, default is **True**
- **width** (*int*[924]) – 79

Kwargs space number of white space characters between columns, default is 2

wrapText(*text*, *width=70*, *join='\n'*, ***kwargs*)
Returns wrapped lines from `textwrap.wrap()`[925] after *join*ing them.

3.13.10 Drawing/Plotting Tools

This module defines utility functions for drawing.

class Arrow3D(*xs*, *ys*, *zs*, **args*, ***kwargs*)
This function is implemented by tacaswell on stackoverflow: https://stackoverflow.com/a/29188796.

class IndexFormatter(*labels*)
Function taken from Matplotlib version 3.3.1 as soon to be deprecated.

Format the position x to the nearest i-th label where `i = int(x + 0.5)`. Positions where `i < 0` or `i > len(list)` have no tick labels.

Parameters **labels** – List of labels.

type labels: list

drawTree(*tree*, *label_func=<type 'str'>*, *show_confidence=False*, *orientation='horizontal'*, *inverted=False*, *branch_labels=None*, *label_colors=None*, **args*, ***kwargs*)
Plot the given tree using matplotlib. This function is adapted from `draw()` for more versatile usages.

3.13.11 Eigen Decomposition Tools

This module defines utility functions for solving eigenvalues.

3.13.12 Linear Assignment Problems Tools

This module defines functions for solving linear assignment problems.

multilap(*cost_matrix*, *nodes=[]*, *BIG_NUMBER=1000000.0*)
Finds the (next) optimal solution to the linear assignment problem. The function can handle the cases where each row can be assigned to multiple columns.

3.13.13 Package Logger

This module defines class that can be used a package wide logger.

class PackageLogger(*name*, ***kwargs*)
A class for package wide logging functionality.

Start logger for the package. Returns a logger instance.

Parameters

- **prefix** – prefix to console log messages, default is `'@> '`
- **console** – log level for console (`sys.stderr`) messages, default is `'debug'`
- **info** – prefix to log messages at *info* level
- **warning** – prefix to log messages at *warning* level, default is `'WARNING '`

[923] http://docs.python.org/library/functions.html#bool
[924] http://docs.python.org/library/functions.html#int
[925] http://docs.python.org/library/textwrap.html#textwrap.wrap

- **error** – prefix to log messages at *error* level, default is `'ERROR '`

addHandler(*hdlr*)
Add the specified handler to this logger.

clear()
Clear current line in `sys.stderr`.

close(*filename*)
Close logfile *filename*.

critical(*msg*)
Log *msg* with severity 'CRITICAL'.

debug(*msg*)
Log *msg* with severity 'DEBUG'.

delHandler(*index*)
Remove handler at given *index* from the logger instance.

error(*msg*)
Log *msg* with severity 'ERROR' and terminate with status 2.

exit(*status=0*)
Exit the interpreter.

getHandlers()
Returns handlers.

info(*msg*)
Log *msg* with severity 'INFO'.

progress(*msg*, *steps*, *label=None*, ***kwargs*)
Instantiate a labeled process with message and number of steps.

report(*msg='Completed in %.2fs.'*, *label=None*)
Write *msg* with timing information for a labeled or default process at *debug* logging level.

sleep(*seconds*, *msg=''*)
Sleep for seconds while updating screen message every second. Message will start with `'Waiting for Xs '` followed by *msg*.

start(*filename*, ***kwargs*)
Start a logfile. If *filename* does not have an extension. `.log` will be appended to it.

Parameters

- **filename** – name of the logfile
- **mode** – mode in which logfile will be opened, default is "w"
- **backupcount** – number of existing *filename.log* files to backup, default is 1

timeit(*label=None*)
Start timing a process. Use *timing()* (page 309) and *report()* (page 309) to learn and report timing, respectively.

timing(*label=None*)
Returns timing for a labeled or default (**None**) process.

update(*step*, *msg=None*, *label=None*)
Update progress status to current line in the console.

warn(*msg*)
Log *msg* with severity 'WARNING'.

warning (*msg*)
Log *msg* with severity 'WARNING'.

write (*line*)
Write *line* into `sys.stderr`.

prefix
String prepended to console log messages.

verbosity
Verbosity *level* of the logger, default level is **debug**. Log messages are written to `sys.stderr`. Following logging levers are recognized:

Level	Description
debug	Everything will be printed to the sys.stderr.
info	Only brief information will be printed.
warning	Only warning messages will be printed.
none	Nothing will be printed.

3.13.14 Miscellaneous Tools

This module defines miscellaneous utility functions.

class **Everything**
A place for everything.

rangeString (*lint*, *sep=' '*, *rng=' to '*, *exc=False*, *pos=True*)
Returns a structured string for a given list of integers.

Parameters

- **lint** – integer list or array
- **sep** – range or number separator
- **rng** – range symbol
- **exc** – set **True** if range symbol is exclusive
- **pos** – only consider zero and positive integers

```
In [1]: from prody.utilities import rangeString

In [2]: lint = [1, 2, 3, 4, 10, 15, 16, 17]

In [3]: rangeString(lint)
Out[3]: '1 to 4 10 15 to 17'

In [4]: rangeString(lint, sep=',', rng='-')
Out[4]: '1-4,10,15-17'

In [5]: rangeString(lint, ',', ':', exc=True)
Out[5]: '1:5,10,15:18'
```

alnum (*string*, *alt='_'*, *trim=False*, *single=False*)
Replace non alpha numeric characters with *alt*. If *trim* is **True** remove preceding and trailing *arg* characters. If *single* is **True**, contain only a single joining *alt* character.

importLA ()
Returns one of `scipy.linalg`[926] or `numpy.linalg`.

[926] http://docs.scipy.org/doc/scipy/reference/reference/linalg.html#module-scipy.linalg

dictElement (*element*, *prefix=None*, *number_multiples=False*)
Returns a dictionary built from the children of *element*, which must be a `xml.etree.ElementTree.Element`[927] instance. Keys of the dictionary are *tag* of children without the *prefix*, or namespace. Values depend on the content of the child. If a child does not have any children, its text attribute is the value. If a child has children, then the child is the value.

intorfloat (*x*)
Returns `int(x)`, or `float(x)` upon `ValueError`[928].

startswith (*this*, *that*)
Returns **True** if *this* or *that* starts with the other.

showFigure ()
Call `show()`[929] function with `block=False` argument to avoid blocking behavior in non-interactive sessions. If *block* keyword argument is not recognized, try again without it.

countBytes (*arrays*, *base=False*)
Returns total number of bytes consumed by elements of arrays. If *base* is **True**, use number of bytes from the base array.

sqrtm (*matrix*)
Returns the square root of a matrix.

addEnds (*x*, *y*, *axis=0*)
Finds breaks in *x*, extends them by one position and adds **nan** at the corresponding position in *y*. *x* needs to be an 1-D array, *y* can be a matrix of column (or row) vectors

isPDB ()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

isURL ()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

isSymmetric (*M*, *rtol=1e-05*, *atol=1e-08*)
Checks if the matrix is symmetric.

makeSymmetric (*M*)
Makes sure the matrix is symmetric.

div0 (*a*, *b*, *defval=0.0*)
Performs `true_divide` but ignores the error when division by zero (result is set to zero instead).

wmean (*array*, *weights*, *axis=None*)
Calculates the weighted average of *array* given *axis*.

bin2dec (*x*)
Converts the binary array to decimal.

fixArraySize (*arr*, *sizes*, *value=0*)
Makes sure that **arr** is of **sizes**. If not, pad with **value**.

decToHybrid36 (*x*, *resnum=False*)
Convert a regular decimal number to a string in hybrid36 format

hybrid36ToDec (*x*, *resnum=False*)
Convert string in hybrid36 format to a regular decimal number

[927] http://docs.python.org/library/xml.etree.elementtree.html#xml.etree.ElementTree.Element
[928] http://docs.python.org/library/exceptions.html#ValueError
[929] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.show.html#matplotlib.pyplot.show

checkIdentifiers(**pdb*, ***kwargs*)
Check whether *pdb* identifiers are valid, and replace invalid ones with **None** in place.

importDec()
Returns one of `scipy.linalg`[930] or `numpy.linalg`.

3.13.15 Path Tools

This module defines functions for handling files and paths.

gunzip(*filename*, *outname=None*)
Returns output name that contains decompressed contents of *filename*. When no *outname* is given, *filename* is used as the output name as it is or after `.gz` extension is removed. *filename* may also be a string buffer, in which case decompressed string buffer or *outname* that contains buffer will be returned.

backupFile(*filename*, *backup=None*, *backup_ext='.BAK'*, ***kwargs*)
Rename *filename* with *backup_ext* appended to its name for backup purposes, if *backup* is **True** or if automatic backups is turned on using *confProDy()* (page 329). Default extension `.BAK` is used when one is not set using *confProDy()* (page 329). If *filename* does not exist, no action will be taken and *filename* will be returned. If file is successfully renamed, new filename will be returned.

openFile(*filename*, **args*, ***kwargs*)
Open *filename* for reading, writing, or appending. First argument in *args* is treated as the mode. Opening `.gz` and `.zip` files for reading and writing is handled automatically.

Parameters

- **backup** (*bool*[931]) – backup existing file using *backupFile()* (page 312) when opening in append or write modes, default is obtained from package settings
- **backup_ext** (*str*[932]) – extension for backup file, default is `.BAK`

openDB(*filename*, **args*)
Open a database with given *filename*.

openSQLite(*filename*, **args*)
Returns a connection to SQLite database *filename*. If `'n'` argument is passed, remove any existing databases with the same name and return connection to a new empty database.

openURL(*url*, *timeout=5*, ***kwargs*)
Open *url* for reading. Raise an `IOError`[933] if *url* cannot be reached. Small *timeout* values are suitable if *url* is an ip address. *kwargs* will be used to make `urllib.request.Request`[934] instance for opening the *url*.

copyFile(*src*, *dst*)
Returns *dst*, a copy of *src*.

isExecutable(*path*)
Returns true if *path* is an executable.

isReadable(*path*)
Returns true if *path* is readable by the user.

isWritable(*path*)
Returns true if *path* is writable by the user.

[930] http://docs.scipy.org/doc/scipy/reference/reference/linalg.html#module-scipy.linalg
[931] http://docs.python.org/library/functions.html#bool
[932] http://docs.python.org/library/stdtypes.html#str
[933] http://docs.python.org/library/exceptions.html#IOError
[934] http://docs.python.org/library/urllib.request.html#urllib.request.Request

makePath (*path*)
Make all directories that does not exist in a given *path*.

relpath (*path*)
Returns *path* on Windows, and relative path elsewhere.

sympath (*path*, *beg=2*, *end=1*, *ellipsis='...'*)
Returns a symbolic path for a long *path*, by replacing folder names in the middle with *ellipsis*. *beg* and *end* specified how many folder (or file) names to include from the beginning and end of the path.

which (*program*)
This function is based on the example in: http://stackoverflow.com/questions/377017/

pickle (*obj*, *filename*, *protocol=2*, ***kwargs*)
Pickle *obj* using `pickle.dump()`[935] in *filename*. *protocol* is set to 2 for compatibility between Python 2 and 3.

unpickle (*filename*, ***kwargs*)
Unpickle object in *filename* using `pickle.load()`[936].

glob (**pathnames*)
Returns concatenation of ordered lists of paths matching patterns in *pathnames*.

addext (*filename*, *extension*)
Returns *filename*, with *extension* if it does not have one.

3.13.16 Sequence Tools

This module defines functions and constants related to sequences.

splitSeqLabel (*label*)
Returns label, starting residue number, and ending residue number parsed from sequence label.

alignBioPairwise (*a_sequence*, *b_sequence*, *ALIGNMENT_METHOD='local'*, *MATCH_SCORE=1.0*, *MISMATCH_SCORE=0.0*, *GAP_PENALTY=-1.0*, *GAP_EXT_PENALTY=-0.1*, *max_alignments=1*)
Wrapper function to align two sequences using Biopython to support deprecation and associated warnings.

It first attempts to use `Bio.Align.PairwiseAligner` and formats the output to match `Bio.pairwise2` methods and if it doesn't find it then it defaults back to `Bio.pairwise2` methods.

Parameters

- **a_sequence** (*str*[937]) – first sequence to align
- **b_sequence** (*str*[938]) – second sequence to align
- **ALIGNMENT_METHOD** (*str*[939]) – method for pairwise2 alignment Possible values are `"local"` and `"global"`
- **MATCH_SCORE** (*int*[940]) – a positive integer, used to reward finding a match
- **MISMATCH_SCORE** (*int*[941]) – a negative integer, used to penalise finding a mismatch

[935] http://docs.python.org/library/pickle.html#pickle.dump
[936] http://docs.python.org/library/pickle.html#pickle.load
[937] http://docs.python.org/library/stdtypes.html#str
[938] http://docs.python.org/library/stdtypes.html#str
[939] http://docs.python.org/library/stdtypes.html#str
[940] http://docs.python.org/library/functions.html#int
[941] http://docs.python.org/library/functions.html#int

- **GAP_PENALTY** (*int*[942]) – a negative integer, used to penalise opening a gap
- **GAP_EXT_PENALTY** (*int*[943]) – a negative integer, used to penalise extending a gap
- **max_alignments** (*int*[944]) – maximum number of alignments to extract

3.13.17 Package Settings

This module defines class for handling and storing package settings.

class PackageSettings(*pkg*, *rcfile=None*, *logger=None*)
A class for managing package settings. Settings are saved in user's home director. When settings are changed by the users, the changes are automatically saved. Settings are stored in a *dict*[945] instance. The dictionary is pickled in user's home directory for permanent storage.

rcfile is the filename for pickled settings dictionary, and by default is set to `.pkgrc`.

get(*key*, *default=None*)
Returns value corresponding to specified *key*, or *default* if *key* is not found.

load()
Load settings by unpickling the settings dictionary.

pop(*key*, *default=None*)
Remove specified *key* and return corresponding value. If *key* is not found, *default* is returned.

save(*backup=False*)
Save settings by pickling the settings dictionary.

update(**args*, ***kwargs*)
Update settings dictionary.

getPackagePath()
Returns package path.

setPackagePath(*path*)
Set package path.

3.14 Applications API

This module contains ProDy applications.

3.14.1 Dynamics analysis

- *prody_anm()* (page 321)
- *prody_gnm()* (page 326)
- *prody_pca()* (page 327)

3.14.2 Structure analysis

- *prody_align()* (page 320)
- *prody_biomol()* (page 322)
- *prody_blast()* (page 322)

[942] http://docs.python.org/library/functions.html#int
[943] http://docs.python.org/library/functions.html#int
[944] http://docs.python.org/library/functions.html#int
[945] http://docs.python.org/library/stdtypes.html#dict

- *prody_catdcd()* (page 323)
- *prody_contacts()* (page 326)
- *prody_fetch()* (page 326)
- *prody_select()* (page 328)

3.14.3 Sequence analysis

- *evol_search()* (page 320)
- *evol_fetch()* (page 316)
- *evol_filter()* (page 317)
- *evol_refine()* (page 319)
- *evol_merge()* (page 318)
- *evol_conserv()* (page 316)
- *evol_coevol()* (page 315)
- *evol_occupancy()* (page 318)
- *evol_rankorder()* (page 318)

3.14.4 Coevolution Application

MSA residue coevolution calculation application.

evol_coevol (*msa*, ***kwargs*)

Analyze co-evolution using mutual information.

Parameters **msa** – refined MSA file

Calculation Options

Parameters

- **ambiguity** (*bool*[946]) – treat amino acids characters B, Z, J, and X as non- ambiguous, default is `True`
- **correction** (*str*[947]) – also save corrected mutual information matrix data and plot, one of `'apc'`,`'asc'`
- **normalization** (*str*[948]) – also save normalized mutual information matrix data and plot, one of `'sument'`,`'minent'`,`'maxent'`,`'mincon'`,`'maxcon'`,`'joint'`

Output Options

Parameters

- **heatmap** (*bool*[949]) – save heatmap files for all mutual information matrices
- **prefix** (*str*[950]) – output filename prefix, default is msa filename with _coevol suffix

[946] http://docs.python.org/library/functions.html#bool
[947] http://docs.python.org/library/stdtypes.html#str
[948] http://docs.python.org/library/stdtypes.html#str
[949] http://docs.python.org/library/functions.html#bool
[950] http://docs.python.org/library/stdtypes.html#str

- **numformat** (*str*[951]) – number output format, default is '%12g'

3.14.5 Conservation Application

Calculate conservation in an MSA using Shannon entropy.

evol_conserv(*msa*, ***kwargs*)

Analyze conservation using Shannon entropy.

Parameters **msa** – refined MSA file

Calculation Options

Parameters

- **ambiguity** (*bool*[952]) – treat amino acids characters B, Z, J, and X as non- ambiguous, default is True
- **omitgaps** (*bool*[953]) – do not omit gap characters, default is True

Output Options

Parameters

- **prefix** (*str*[954]) – output filename prefix, default is msa filename with _conserv suffix
- **numformat** (*str*[955]) – number output format, default is '%12g'

3.14.6 Pfam MSA Fetcher

Pfam MSA download application.

evol_fetch(*acc*, ***kwargs*)

Fetch MSA files from Pfam.

Parameters **acc** (*str*[956]) – Pfam accession or ID

Download Options

Parameters

- **alignment** (*str*[957]) – alignment type, one of 'full', 'seed', 'ncbi', 'metagenomics', default is 'full'
- **format** (*str*[958]) – Pfam supported MSA format, one of 'selex', 'fasta', 'stockholm', default is 'stockholm'
- **order** (*str*[959]) – ordering of sequences, one of 'tree', 'alphabetical', default is 'tree'
- **inserts** (*str*[960]) – letter case for inserts, one of 'upper', 'lower', default is 'upper'

[951] http://docs.python.org/library/stdtypes.html#str
[952] http://docs.python.org/library/functions.html#bool
[953] http://docs.python.org/library/functions.html#bool
[954] http://docs.python.org/library/stdtypes.html#str
[955] http://docs.python.org/library/stdtypes.html#str
[956] http://docs.python.org/library/stdtypes.html#str
[957] http://docs.python.org/library/stdtypes.html#str
[958] http://docs.python.org/library/stdtypes.html#str
[959] http://docs.python.org/library/stdtypes.html#str
[960] http://docs.python.org/library/stdtypes.html#str

- **gaps** (*str*[961]) – gap character, one of `'dashes'`, `'dots'`, `'mixed'`, default is `'dashes'`
- **timeout** (*int*[962]) – timeout for blocking connection attempts, default is `60`

Output Options

Parameters

- **folder** (*str*[963]) – output directory, default is `'.'`
- **outname** (*str*[964]) – output filename, default is accession and alignment type
- **compressed** (*bool*[965]) – gzip downloaded MSA file

3.14.7 MSA File Filter

Refine MSA application.

evol_filter(*msa*, **word*, ***kwargs*)

Filter an MSA using sequence labels.

Parameters

- **msa** – MSA filename to be filtered
- **word** – word to be compared to sequence label

Filtering Method (Required)

Parameters

- **startswith** (*bool*[966]) – sequence label starts with given words
- **endswith** (*bool*[967]) – sequence label ends with given words
- **contains** (*bool*[968]) – sequence label contains with given words

Filter Option

Parameters **filter_full** (*bool*[969]) – compare full label with word(s)

Output Options

Parameters

- **outname** (*str*[970]) – output filename, default is msa filename with _refined suffix
- **format** (*str*[971]) – output MSA file format, default is same as input
- **compressed** (*bool*[972]) – gzip refined MSA output

[961] http://docs.python.org/library/stdtypes.html#str
[962] http://docs.python.org/library/functions.html#int
[963] http://docs.python.org/library/stdtypes.html#str
[964] http://docs.python.org/library/stdtypes.html#str
[965] http://docs.python.org/library/functions.html#bool
[966] http://docs.python.org/library/functions.html#bool
[967] http://docs.python.org/library/functions.html#bool
[968] http://docs.python.org/library/functions.html#bool
[969] http://docs.python.org/library/functions.html#bool
[970] http://docs.python.org/library/stdtypes.html#str
[971] http://docs.python.org/library/stdtypes.html#str
[972] http://docs.python.org/library/functions.html#bool

3.14.8 MSA File Merger

Merge multiple MSAs based on common labels.

evol_merge (**msa*, ***kwargs*)

Merge multiple MSAs based on common labels.

Parameters **msa** – MSA filenames to be merged

Output Options

Parameters

- **outname** (*str*[973]) – output filename, default is first input filename with _merged suffix
- **format** (*str*[974]) – output MSA file format, default is same as first input MSA
- **compressed** (*bool*[975]) – gzip merged MSA output

3.14.9 MSA Occupancy Calculation

MSA residue coevolution calculation application.

evol_occupancy (*msa*, ***kwargs*)

Calculate occupancy of rows and columns in MSA.

Parameters **msa** – MSA file

Calculation Options

Parameters **occaxis** (*str*[976]) – calculate row or column occupancy or both., one of `'row'`, `'col'`,`'both'`, default is `'row'`

Output Options

Parameters

- **prefix** (*str*[977]) – output filename prefix, default is msa filename with _occupancy suffix
- **label** (*str*[978]) – index for column based on msa label
- **numformat** (*str*[979]) – number output format, default is `'%12g'`

3.14.10 Identify Coevolving Pairs

Refine MSA application.

evol_rankorder (*mutinfo*, ***kwargs*)

Identify highly coevolving pairs of residues.

Parameters **mutinfo** – mutual information matrix

Input Options

Parameters

[973] http://docs.python.org/library/stdtypes.html#str
[974] http://docs.python.org/library/stdtypes.html#str
[975] http://docs.python.org/library/functions.html#bool
[976] http://docs.python.org/library/stdtypes.html#str
[977] http://docs.python.org/library/stdtypes.html#str
[978] http://docs.python.org/library/stdtypes.html#str
[979] http://docs.python.org/library/stdtypes.html#str

- **zscore** (*bool*[980]) – apply zscore for identifying top ranked coevolving pairs
- **delimiter** (*str*[981]) – delimiter used in mutual information matrix file
- **pdb** (*str*[982]) – PDB file that contains same number of residues as the mutual information matrix, output residue numbers will be based on PDB file
- **msa** (*str*[983]) – MSA file used for building the mutual info matrix, output residue numbers will be based on the most complete sequence in MSA if a PDB file or sequence label is not specified
- **label** (*str*[984]) – label in MSA file for output residue numbers

Output Options

Parameters

- **numpairs** (*int*[985]) – number of top ranking residue pairs to list, default is `100`
- **seqsep** (*int*[986]) – report coevolution for residue pairs that are sequentially separated by input value, default is `3`
- **dist** (*float*[987]) – report coevolution for residue pairs whose CA atoms are spatially separated by at least the input value, used when a PDB file is given and –usedist is true, default is `10.0`
- **usedist** (*bool*[988]) – use structural separation to report coevolving pairs
- **outname** (*str*[989]) – output filename, default is mutinfo_rankorder.txt

3.14.11 MSA Refinement

Refine MSA application.

evol_refine(*msa*, ***kwargs*)

Refine an MSA by removing gapped rows/colums.

Parameters **msa** – MSA filename to be refined

Refinement Options

Parameters

- **label** (*str*[990]) – sequence label, UniProt ID code, or PDB and chain identifier
- **seqid** (*float*[991]) – identity threshold for selecting unique sequences
- **colocc** (*float*[992]) – column (residue position) occupancy
- **rowocc** (*float*[993]) – row (sequence) occupancy

[980] http://docs.python.org/library/functions.html#bool
[981] http://docs.python.org/library/stdtypes.html#str
[982] http://docs.python.org/library/stdtypes.html#str
[983] http://docs.python.org/library/stdtypes.html#str
[984] http://docs.python.org/library/stdtypes.html#str
[985] http://docs.python.org/library/functions.html#int
[986] http://docs.python.org/library/functions.html#int
[987] http://docs.python.org/library/functions.html#float
[988] http://docs.python.org/library/functions.html#bool
[989] http://docs.python.org/library/stdtypes.html#str
[990] http://docs.python.org/library/stdtypes.html#str
[991] http://docs.python.org/library/functions.html#float
[992] http://docs.python.org/library/functions.html#float
[993] http://docs.python.org/library/functions.html#float

- **pdbres** (*bool*[994]) – keep columns corresponding to residues not resolved in PDB structure, applies label argument is a PDB identifier

Output Options

Parameters

- **outname** (*str*[995]) – output filename, default is msa filename with _refined suffix
- **format** (*str*[996]) – output MSA file format, default is same as input
- **compressed** (*bool*[997]) – gzip refined MSA output

3.14.12 Pfam Search

Pfam search application.

evol_search(*query*, ***kwargs*)

Search Pfam with given *query*.

Parameters **query** – protein UniProt ID or sequence, a PDB identifier, or a sequence file, where sequence have no gaps and 12 or more characters

Sequence Search Options

Parameters

- **search_b** (*bool*[998]) – search Pfam-B families
- **skip_a** (*bool*[999]) – do not search Pfam-A families
- **ga** (*bool*[1000]) – use gathering threshold
- **evalue** (*float*[1001]) – e-value cutoff, must be less than 10.0
- **timeout** (*int*[1002]) – timeout in seconds for blocking connection attempt, default is `60`

Output Options

Parameters

- **outname** (*str*[1003]) – name for output file, default is standard output
- **delimiter** (*str*[1004]) – delimiter for output data columns, default is `'\t'`

3.14.13 PDB Model/Structure Alignment

Align models in a PDB file or multiple structures in separate PDB files.

prody_align(**pdbs*, ***kwargs*)

Align models in a PDB file or multiple structures in separate PDB files. By default, protein chains will be matched based on selected atoms and alignment will be performed based on matching residues. If

[994] http://docs.python.org/library/functions.html#bool
[995] http://docs.python.org/library/stdtypes.html#str
[996] http://docs.python.org/library/stdtypes.html#str
[997] http://docs.python.org/library/functions.html#bool
[998] http://docs.python.org/library/functions.html#bool
[999] http://docs.python.org/library/functions.html#bool
[1000] http://docs.python.org/library/functions.html#bool
[1001] http://docs.python.org/library/functions.html#float
[1002] http://docs.python.org/library/functions.html#int
[1003] http://docs.python.org/library/stdtypes.html#str
[1004] http://docs.python.org/library/stdtypes.html#str

non-protein atoms are selected and selected atoms match in multiple structures, they will be used for alignment.

Parameters

- **pdbs** – PDB identifier(s) or filename(s)
- **select** – atom selection string, default is *calpha*, see *Atom Selections* (page 92)
- **model** – for NMR files, reference model index, default is `1`
- **seqid** – percent sequence identity, default is `90`
- **overlap** – percent sequence overlap, default is `90`
- **prefix** – prefix for output file, default is PDB filename
- **suffix** – output filename suffix, default is `_aligned`

3.14.14 ANM Application

Perform ANM calculations and output the results in plain text, NMD, and graphical formats.

prody_anm(*pdb*, ***kwargs*)

Perform ANM calculations for *pdb*.

Parameters

- **altloc** – alternative location identifiers for residues used in the calculations, default is `'A'`
- **cutoff** – cutoff distance (A), default is `'15.'`
- **extend** – write NMD file for the model extended to "backbone" ("bb") or "all" atoms of the residue, model must have one node per residue, default is `''`
- **figall** – save all figures, default is `False`
- **figbeta** – save beta-factors figure, default is `False`
- **figcc** – save cross-correlations figure, default is `False`
- **figcmap** – save contact map (Kirchhoff matrix) figure, default is `False`
- **figdpi** – figure resolution (dpi), default is `300`
- **figformat** – figure file format, default is `'pdf'`
- **figheight** – figure height (inch), default is `6.0`
- **figmode** – save mode shape figures for specified modes, e.g. "1-3 5" for modes 1, 2, 3 and 5, default is `''`
- **figsf** – save square-fluctuations figure, default is `False`
- **figwidth** – figure width (inch), default is `8.0`
- **gamma** – spring constant, default is `'1.'`
- **hessian** – write Hessian matrix, default is `False`
- **kdtree** – use kdtree for Hessian, default is `False`
- **kirchhoff** – write Kirchhoff matrix, default is `False`
- **membrane** – whether to include the explicit membrane model, default is `False`
- **model** – index of model that will be used in the calculations, default is `1`

- **nmodes** – number of non-zero eigenvectors (modes) to calculate, default is `10`
- **nproc** – number of processors, default is `0`
- **npzmatrices** – write matrix to compressed ProDy data file, default is `False`
- **numdelim** – number delimiter, default is `' '`
- **numext** – numeric file extension, default is `'.txt'`
- **numformat** – number output format, default is `'%12g'`
- **outall** – write all outputs, default is `False`
- **outbeta** – write beta-factors calculated from ANM modes, default is `False`
- **outcc** – write cross-correlations, default is `False`
- **outcov** – write covariance matrix, default is `False`
- **outdir** – output directory, default is `'.'`
- **outeig** – write eigenvalues/vectors, default is `False`
- **outhm** – write cross-correlations heatmap file, default is `False`
- **outnpz** – write compressed ProDy data file, default is `False`
- **outscipion** – write continuousflex modes directory and sqlite, default is `False`
- **outsf** – write square-fluctuations, default is `False`
- **prefix** – output file prefix, default is `'_anm'`
- **select** – atom selection, default is `"protein and name CA or nucleic and name P C4' C2"`
- **sparse** – use sparse matrices, default is `False`
- **turbo** – use memory-intensive turbo option for modes, default is `False`
- **zeros** – calculate zero modes, default is `False`

3.14.15 Biomolecule Builder

Generate biomolecule structure using the transformation from the header section of the PDB file.

prody_biomol (*pdbname*, ***kwargs*)
Generate biomolecule coordinates.

Parameters

- **pdb** – PDB identifier or filename
- **prefix** – prefix for output files, default is `_biomol`
- **biomol** – index of the biomolecule, by default all are generated

3.14.16 Blast Search PDB

Blast Protein Data Bank for structures matching a user given sequence.

prody_blast (*sequence*, ***kwargs*)
Blast search PDB and download hits.

Parameters

- **sequence** – sequence or file in fasta format

- **identity** (*float*[1005]) – percent sequence identity for blast search, default is 90.0
- **overlap** (*float*[1006]) – percent sequence overlap between sequences, default is 90.0
- **outdir** (*str*[1007]) – download uncompressed PDB files to given directory
- **gzip** – write compressed PDB file

Blast Parameters

Parameters

- **filename** (*str*[1008]) – a *filename* to save the results in XML format
- **hitlist_size** (*int*[1009]) – search parameters, default is 250
- **expect** (*float*[1010]) – search parameters, default is 1e-10
- **sleep** (*int*[1011]) – how long to wait to reconnect for results, default is 2 sleep time is doubled when results are not ready.
- **timeout** (*int*[1012]) – when to give up waiting for results. default is 30

3.14.17 DCD Files Concatenation

Concatenate, slice, and/or select DCD files.

prody_catdcd(**dcd*, ***kwargs*)
Concatenate *dcd* files.

Parameters

- **select** – atom selection
- **align** – atom selection for aligning frames
- **pdb** – PDB file used in atom selections and as reference for alignment
- **psf** – PSF file used in atom selections
- **output** – output filename
- **first** – index of the first output frame
- **last** – index of the last output frame
- **stride** – number of steps between output frames

3.14.18 ANM Application

Run the ClustENM(D) hybrid simulation method, combining clustering, ENM NMA and MD.

prody_clustenm(*pdb*, ***kwargs*)
Run ClustENM(D) for *pdb*.

Parameters

[1005] http://docs.python.org/library/functions.html#float
[1006] http://docs.python.org/library/functions.html#float
[1007] http://docs.python.org/library/stdtypes.html#str
[1008] http://docs.python.org/library/stdtypes.html#str
[1009] http://docs.python.org/library/functions.html#int
[1010] http://docs.python.org/library/functions.html#float
[1011] http://docs.python.org/library/functions.html#int
[1012] http://docs.python.org/library/functions.html#int

- **altloc** – alternative location identifiers for residues used in the simulations, default is 'A'
- **cutoff** – cutoff distance (A), default is '15.'
- **extend** – write NMD file for the model extended to "backbone" ("bb") or "all" atoms of the residue, model must have one node per residue, default is ''
- **figall** – save all figures, default is False
- **figcc** – save cross-correlations figure, default is False
- **figdpi** – figure resolution (dpi), default is 300
- **figformat** – figure file format, default is 'pdf'
- **figheight** – figure height (inch), default is 6.0
- **figsf** – save square-fluctuations figure, default is False
- **figwidth** – figure width (inch), default is 8.0
- **fit_resolution** – resolution for blurring structures for fitting cc, default is 5
- **fitmap** – map to fit by filtering conformations like MDeNMD-EMFit, default is None
- **forcefield** – Alternative force field pair tuple for protein then solvent from openmm, default is 'None'
- **gamma** – spring constant, default is '1.'
- **ionicStrength** – total concentration (M) of ions (positive and negative), excluding those to neutralize, default is 0.0
- **kdtree** – use kdtree for Hessian, default is False
- **map_cutoff** – min_cutoff for passing map for fitting, default is 0
- **maxIterations** – maximum number of iterations of energy minimization, 0 means until convergence, default is 0
- **maxclust** – maximum number of clusters for each generation, can be tuple of floats, default is 'None'
- **membrane** – whether to include the explicit membrane model, default is False
- **model** – index of model that will be used in the simulations, default is 1
- **multiple** – whether each conformer will be saved as a separate PDB file, default is False
- **mzscore** – modified z-score threshold to label conformers as outliers, default is 3.5
- **nconfs** – number of new conformers from each one from previous generation, default is 50
- **ngens** – number of generations, default is 5
- **nmodes** – number of non-zero eigenvectors (modes) to calculate, default is 10
- **no_outlier** – whether to not exclude outliers in each generation when using implicit solvent (always False for explicit), default is False
- **no_sim** – whether a short MD simulation is not performed after energy minimization (otherwise it is), default is False

- **nproc** – number of processors, default is `0`
- **npzmatrices** – write matrix to compressed ProDy data file, default is `False`
- **numdelim** – number delimiter, default is `' '`
- **numext** – numeric file extension, default is `'.txt'`
- **numformat** – number output format, default is `'%12g'`
- **outall** – write all outputs, default is `False`
- **outcc** – write cross-correlations, default is `False`
- **outcov** – write covariance matrix, default is `False`
- **outdir** – output directory, default is `'.'`
- **outeig** – write eigenvalues/vectors, default is `False`
- **outhm** – write cross-correlations heatmap file, default is `False`
- **outnpz** – write compressed ProDy data file, default is `False`
- **outscipion** – write continuousflex modes directory and sqlite, default is `False`
- **outsf** – write square-fluctuations, default is `False`
- **padding** – padding distance to use for solvation (nm), default is `1.0`
- **parallel** – whether conformer generation will be parallelized, default is `False`
- **platform** – openmm platform (OpenCL, CUDA, CPU or None), default is `None`
- **prefix** – output file prefix, default is `'_clustenm'`
- **replace_filtered** – whether to keep sampling again to replace filtered conformers, default is `False`
- **rmsd** – average RMSD of new conformers from previous ones, can be tuple of floats, default is `'1.'`
- **select** – atom selection, default is `"protein and name CA or nucleic and name P C4' C2"`
- **solvent** – solvent model to be used, either imp for implicit or exp for explicit, default is `'imp'`
- **sparse** – use sparse matrices, default is `False`
- **t_steps_g** – number of 2.0 fs MD time steps in each generation, can be tuple of floats, default is `'7500'`
- **t_steps_i** – number of 2.0 fs MD time steps for initial structure, default is `1000`
- **temp** – temperature at which simulations are conducted, default is `303.15`
- **threshold** – RMSD threshold to apply when forming clusters, can be tuple of floats, default is `'None'`
- **tolerance** – energy tolerance to which the system should be minimized in kJ/mole, default is `10.0`
- **turbo** – use memory-intensive turbo option for modes, default is `False`
- **v1** – whether to use original sampling method with complete enumeration of ANM modes, default is `False`
- **write_params** – whether to write parameters, default is `False`

3.14.19 Contact Identification

This module defines a routine for contact identification.

prody_contacts(***kwargs*)

Identify contacts of a target structure with one or more ligands. Contacting atoms (or extended subset of atoms, such as residues) are outputted in PDB file format.

Parameters

- **target** – target PDB identifier or filename
- **ligand** – ligand PDB identifier(s) or filename(s)
- **select** – atom selection string for target structure
- **radius** – contact radius (Å), default is `4.0`
- **extend** – output same `'residue'`, `'chain'`, or `'segment'` along with contacting atoms
- **prefix** – prefix for output file, default is *target* filename
- **suffix** – output filename suffix, default is *ligand* filename

3.14.20 PDB File Fetcher

Download PDB files for given identifiers.

prody_fetch(**pdb*, ***kwargs*)

Fetch PDB files from PDB FTP server.

Parameters

- **pdbs** – PDB identifier(s) or filename(s)
- **dir** – target directory for saving PDB file(s), default is `'.'`
- **gzip** – gzip fetched files or not, default is **True**

3.14.21 GNM Application

Perform GNM calculations and output the results in plain text NMD, and graphical formats.

prody_gnm(*pdb*, ***kwargs*)

Perform GNM calculations for *pdb*.

Parameters

- **altloc** – alternative location identifiers for residues used in the calculations, default is `'A'`
- **cutoff** – cutoff distance (A), default is `10.0`
- **extend** – write NMD file for the model extended to “backbone” (“bb”) or “all” atoms of the residue, model must have one node per residue, default is `''`
- **figall** – save all figures, default is `False`
- **figbeta** – save beta-factors figure, default is `False`
- **figcc** – save cross-correlations figure, default is `False`
- **figcmap** – save contact map (Kirchhoff matrix) figure, default is `False`
- **figdpi** – figure resolution (dpi), default is `300`

- **figformat** – figure file format, default is 'pdf'
- **figheight** – figure height (inch), default is 6.0
- **figmode** – save mode shape figures for specified modes, e.g. "1-3 5" for modes 1, 2, 3 and 5, default is ''
- **figsf** – save square-fluctuations figure, default is False
- **figwidth** – figure width (inch), default is 8.0
- **gamma** – spring constant, default is '1.'
- **kirchhoff** – write Kirchhoff matrix, default is False
- **membrane** – whether to include the explicit membrane model, default is False
- **model** – index of model that will be used in the calculations, default is 1
- **nmodes** – number of non-zero eigenvectors (modes) to calculate, default is 10
- **nproc** – number of processors, default is 0
- **npzmatrices** – write matrix to compressed ProDy data file, default is False
- **numdelim** – number delimiter, default is ' '
- **numext** – numeric file extension, default is '.txt'
- **numformat** – number output format, default is '%12g'
- **outall** – write all outputs, default is False
- **outbeta** – write beta-factors calculated from GNM modes, default is False
- **outcc** – write cross-correlations, default is False
- **outcov** – write covariance matrix, default is False
- **outdir** – output directory, default is '.'
- **outeig** – write eigenvalues/vectors, default is False
- **outhm** – write cross-correlations heatmap file, default is False
- **outnpz** – write compressed ProDy data file, default is False
- **outscipion** – write continuousflex modes directory and sqlite, default is False
- **outsf** – write square-fluctuations, default is False
- **prefix** – output file prefix, default is '_gnm'
- **select** – atom selection, default is "protein and name CA or nucleic and name P C4' C2"
- **zeros** – calculate zero modes, default is False

3.14.22 PCA Application

Perform PCA/EDA calculations and output the results in plain text, NMD, and graphical formats.

prody_pca(*coords*, ***kwargs*)

Perform PCA calculations for PDB or DCD format *coords* file.

Parameters

- **aligned** – trajectory is already aligned, default is False

- **altloc** – alternative location identifiers for residues used in the calculations, default is `'A'`
- **extend** – write NMD file for the model extended to "backbone" ("bb") or "all" atoms of the residue, model must have one node per residue, default is `''`
- **figall** – save all figures, default is `False`
- **figcc** – save cross-correlations figure, default is `False`
- **figdpi** – figure resolution (dpi), default is `300`
- **figformat** – figure file format, default is `'pdf'`
- **figheight** – figure height (inch), default is `6.0`
- **figproj** – save projections onto specified subspaces, e.g. "1,2" for projections onto PCs 1 and 2; "1,2 1,3" for projections onto PCs 1,2 and 1, 3; "1 1,2,3" for projections onto PCs 1 and 1, 2, 3, default is `''`
- **figsf** – save square-fluctuations figure, default is `False`
- **figwidth** – figure width (inch), default is `8.0`
- **membrane** – whether to include the explicit membrane model, default is `False`
- **nmodes** – number of non-zero eigenvectors (modes) to calculate, default is `10`
- **nproc** – number of processors, default is `0`
- **npzmatrices** – write matrix to compressed ProDy data file, default is `False`
- **numdelim** – number delimiter, default is `' '`
- **numext** – numeric file extension, default is `'.txt'`
- **numformat** – number output format, default is `'%12g'`
- **outall** – write all outputs, default is `False`
- **outcc** – write cross-correlations, default is `False`
- **outcov** – write covariance matrix, default is `False`
- **outdir** – output directory, default is `'.'`
- **outeig** – write eigenvalues/vectors, default is `False`
- **outhm** – write cross-correlations heatmap file, default is `False`
- **outnpz** – write compressed ProDy data file, default is `False`
- **outproj** – write projections onto PCs, default is `False`
- **outscipion** – write continuousflex modes directory and sqlite, default is `False`
- **outsf** – write square-fluctuations, default is `False`
- **prefix** – output file prefix, default is `'_pca'`
- **select** – atom selection, default is `"protein and name CA or nucleic and name P C4' C2"`

3.14.23 Atom Selection

Extract a selection of atoms from a PDB file.

prody_select(*selstr*, **pdbs*, ***kwargs*)

Write selected atoms from a PDB file in PDB format.

Parameters

- **selstr** – atom selection string, see *Atom Selections* (page 92)
- **pdbs** – PDB identifier(s) or filename(s)
- **output** – output filename, default is `pdb_selected.pdb`
- **prefix** – prefix for output file, default is PDB filename
- **suffix** – output filename suffix, default is `_selected`

3.15 Configuration & Logging

This module defines functions for logging in files, configuring ProDy, and running tests.

- *confProDy()* (page 329)
- *checkUpdates()* (page 329)
- *startLogfile()* (page 329)
- *closeLogfile()* (page 330)
- *plog()* (page 330)

confProDy(**args*, ***kwargs*)

Configure ProDy.

Option	Default (acceptable values)
auto_bonds	False
auto_secondary	False
auto_show	False
backup	False
backup_ext	'.BAK'
check_updates	0
ligand_xml_save	False
local_pdb_folder	'' See also *pathPDBFolder()* (page 266).
pdb_mirror_path	'' See also *pathPDBMirror()* (page 266).
selection_warning	True
typo_warnings	True
verbosity	'debug' ('critical', 'debug', 'error', 'info', 'none', 'progress', or 'warning')

Usage example:

```
confProDy('backup')
confProDy('backup', 'backup_ext')
confProDy(backup=True, backup_ext='.bak')
confProDy(backup_ext='.BAK')
```

checkUpdates()

Check PyPI to see if there is a newer ProDy version available. Setting ProDy configuration parameter *check_updates* to a positive integer will make ProDy automatically check updates, e.g.:

```
confProDy(check_updates=7) # check at most once a week
confProDy(check_updates=0) # do not auto check updates
confProDy(check_updates=-1) # check at the start of every session
```

startLogfile(*filename*, ***kwargs*)

Start a logfile. If *filename* does not have an extension, `.log` will be appended to it.

Parameters

- **filename** – name of the logfile
- **mode** – mode in which logfile will be opened, default is "w"
- **backupcount** – number of existing *filename.log* files to backup, default is 1

closeLogfile(*filename*)
Close logfile with *filename*.

plog(**text*)
Log *text* using ProDy logger at log level info. Multiple arguments are accepted. Each argument will be converted to string and joined using a white space as delimiter.

CHAPTER

FOUR

DEVELOPER'S GUIDE

4.1 Contributing to ProDy

4.1.1 Install Git and a GUI

ProDy source code is managed using Git[1013] distributed revision controlling system. You need to install **`git`**, and if you prefer a GUI for it, on your computer to be able to contribute to development of ProDy.

On Debian/Ubuntu Linux, for example, you can run the following to install **`git`** and **`gitk`**:

```
$ sudo apt-get install git gitk
```

For other operating systems, you can obtain installation instructions and files from Git[1014].

You will only need to use a few basic **`git`** commands. These commands are provided below, but usually without an adequate description. Please refer to Git book[1015] and Git docs[1016] for usage details and examples.

4.1.2 Fork and Clone ProDy

ProDy source code an issue tracker are hosted on Github[1017]. You need to create an account on this service, if you do not have one already.

If you work on Mac OS or Windows, you may consider getting GitHub Mac[1018] or GitHub Windows[1019] to help you manage a copy of the repository.

Once you have an account, you need to make a fork of ProDy, which is creating a copy of the repository in your account. You will see a link for this on ProDy[1020] source code page. You will have write access to this

[1013] http://git-scm.com/downloads
[1014] http://git-scm.com/downloads
[1015] http://git-scm.com/book
[1016] http://git-scm.com/docs
[1017] http://github.com/prody/ProDy
[1018] http://mac.github.com
[1019] http://windows.github.com
[1020] http://prody.csb.pitt.edu

fork and later will use it share your changes with others.

The next step is cloning the fork from your online account (e.g. jamesmkrieger) to your local system. If you are not using the GitHub software, you can do it as follows:

```
$ git clone https://github.com/jamesmkrieger/ProDy.git
```

This will create `ProDy` folder with a copy of the project files in it:

```
$ cd ProDy
$ ls
bdist_wininst.bat  docs    INSTALL.rst  LICENSE.rst  Makefile
MANIFEST.in        prody   README.rst   scripts      setup.py
```

4.1.3 Setup Working Environment

You can use ProDy directly from this clone by adding `ProDy` folder to your `PYTHONPATH`[1021] environment variable, e.g.:

```
export PYTHONPATH=$PYTHONPATH:$/home/USERNAME/path/to/ProDy
```

This will not be enough though, since you also need to compile C extensions. You can run the following series of commands to build and copy C modules to where they need to be:

```
$ cd ProDy
$ python setup.py build_ext --inplace --force
```

or, on Linux you can:

```
$ make build
```

You may also want to make sure that you can run *ProDy Applications* (page 3) from anywhere on your system. One way to do this by adding `ProDy/scripts` folder to your `PATH`[1022] environment variable, e.g.:

```
export PATH=$PATH:$/home/USERNAME/path/to/ProDy/scripts
```

4.1.4 Modify, Test, and Commit

When modifying ProDy files you may want to follow the *Style Guide for ProDy* (page 336). Closely following the guidelines therein will allow for incorporation of your changes to ProDy quickly.

If you changed `.py` files, you should ensure to check the integrity of the package. To do this, you should at least run fast ProDy tests as follows:

```
$ cd ProDy
$ nosetests
```

See *Testing ProDy* (page 338) for alternate and more comprehensive ways of testing. ProDy unittest suit may not include a test for the function or the class that you just changed, but running the tests will ensure that the ProDy package can be imported and run without problems.

After ensuring that the package runs, you can commit your changes as follows:

```
$ git commit modified_file_1.py modified_file_2.py
```

or:

[1021] http://docs.python.org/using/cmdline.html#envvar-PYTHONPATH
[1022] http://matplotlib.sourceforge.net/install/environment_variables_faq.html#envvar-PATH

```
$ git commit -a
```

This command will open a text editor for you to describe the changes that you just committed.

4.1.5 Push and Pull Request

After you have committed your changes, you will need to push them to your GitHub account:

```
git push origin master
```

This step will ask for your account user name. If you are going to push to your GitHub account frequently, you may add an SSH key for automatic authentication. To add an SSH key for your system, go to *Edit Your Profile → SSH keys* page on GitHub.

After pushing your changes, you will need to make a pull request from your to notify ProDy developers of the changes you made and facilitate their incorporation to ProDy.

4.1.6 Update Local Copy

You can also keep an up-to-date copy of ProDy by pulling changes from the master ProDy[1023] repository on a regular basis. You need add to the master repository as a remote to your local copy. You can do this running the following command from the ProDy project folder:

```
$ cd prody
$ git remote add prodymaster git@github.com:prody/ProDy.git
```

You may use any name other than *prodymaster*, except for *origin*, which points to the ProDy fork in your account.

After setting up this remote, calling **`git pull`** command will fetch latest changes from ProDy[1024] master repository and merge them to your local copy:

```
$ git pull prodymaster master
```

Note that when there are changes in C modules, you need to run the following commands again to update the binary module files:

```
$ python setup.py build_ext --inplace --force
```

4.2 Documenting ProDy

ProDy documentation is written using reStructuredText[1025] markup and prepared using Sphinx[1026]. You may install Sphinx using **`easy_install`**, i.e. `easy_install -U Sphinx`, or using package manager on your Linux machine.

[1023] http://prody.csb.pitt.edu
[1024] http://prody.csb.pitt.edu
[1025] http://docutils.sf.net/rst.html
[1026] http://sphinx.pocoo.org/

4.2.1 Building Manual

ProDy Manual in HTML and PDF formats can be build as follows:

```
$ cd docs
$ make html
$ make pdf
```

If all documentation strings and pages are properly formatted according to reStructuredText[1027] markup, documentation pages should compile without any warnings. Note that to build PDF files, you need to install **`latex`** and **`pdflatex`** programs.

Read the Docs

A copy of ProDy manual is hosted on Read the Docs[1028] and can be viewed at http://prody.readthedocs.org/. Read the Docs is configured to build manual pages for the `devel` branch (latest) and the recent stable versions. The user name for Read the Docs is `prody`.

4.2.2 Building Website

ProDy-website source is hosted at https://github.com/prody/ProDy-website This project contains tutorial files and the home pages for ProDy and other related software.

Latest version

To build website on ProDy server, start with pulling changes:

```
$ cd ProDy-website
$ git pull
```

Running the following command will build HTML pages for the latest stable release of ProDy:

```
$ make html
```

HTML pages for manual and all tutorials are build as a single project, which allows for referencing from manual to tutorials.

PDF files for the manual and tutorials, and also download files are build as follows:

```
$ make pdf
```

PDF and TGZ/ZIP files are copied to appropriate places after they are built.

4.3 How to Make a Release

1. Make sure ProDy imports and passes all unit tests both Python 2 and Python 3, and using nose **`nosetests`** command:

   ```
   $ cd ProDy
   $ nosetests
   $ nosetests3
   ```

   See *Testing ProDy* (page 338) for more on testing.

2. Update the version number in:

   - `prody/__init__.py`
   - `./PKG-INFO`

[1027] http://docutils.sf.net/rst.html
[1028] https://readthedocs.org/

Also, comment `+ '-dev'` out, so that documentation will build for a stable release.

3. Update the most recent changes and the latest release date in:

   - `docs/release/vX.Y_series.rst`.

   If there is a new incremental release, start a new file.

4. Make sure the following files are up-to-date.

   - `README.txt`
   - `MANIFEST.in`
   - `setup.py`

   If there is a new file format, that is a new extensions not captured in `MANIFEST.in`, it should be included.

   If there is a new C extension, it should be listed in `setup.py`.

   After checking these files, commit change and push them to GitHub[1029].

5. Generate the source distributions:

```
$ cd ..
$ python setup.py sdist --formats=gztar,zip
```

6. Prepare and test Python Wheels[1030] on Windows (see *Making Windows Installers* (page 343)).

   Wheels should be prepared for the following versions of Python:

```
$ C:\Python27\python setup.py bdist_wheel
$ C:\Python35\python setup.py bdist_wheel
$ C:\Python36\python setup.py bdist_wheel
```

   Alternatively, use **`bdist_wheel.bat`** to run these commands. When there is a newer Python major release, it should be added to this list. Don't forget to pull most recent changes to your Windows machine.

   A good practice is installing ProDy using all newly created installers and checking that it works. ProDy script can be used to check that, e.g.:

```
$ C:\Python33\Scripts\prody.bat anm 1ubi
```

   If this command runs for all supported Python versions, release is good to go.

7. Put all installation source and executable in dist directory.

8. Upload the new release files to the PyPI[1031] using twine (NOTE: this step is **irreversible**! If there were to be a change to ProDy after this step, then it needs to be prepared as a whole new release):

```
$ twine upload dist/*
```

   This will offer a number of options. ProDy on PyPI is owned by user `prody.devel`.

9. Commit final changes, if there are any:

```
$ cd ..
$ git commit -a
```

10. Tag the repository with the current version number and push new tag:

[1029] http://github.com/prody/ProDy
[1030] https://pythonwheels.com/
[1031] http://pypi.python.org/pypi/ProDy

```
$ git tag vX.Y
$ git push --tags
```

11. Rebase `devel` branch to `master`:

```
$ git checkout master
$ git rebase devel
$ git push
```

12. Update the documentation on ProDy[1032] website. See *Documenting ProDy* (page 333).
13. Now that you made a release, you can go back to development. You may start with appending `'-dev'` to `__release__` in `prody/__init__.py`.

4.4 Style Guide for ProDy

4.4.1 Introduction

PEP 8[1033], the *Style Guide for Python Code*, is adopted in the development of ProDy package. Contributions to ProDy shall follow **PEP 8**[1034] and the specifications and additions provided in this addendum.

4.4.2 Code Layout

Indentation

Use 4 spaces per indentation level in source code (`.py`) and never use tabs as a substitute.

In documentation files (`.rst`), use 2 spaces per indentation level.

Maximum line length

Limit all lines to a maximum of 79 characters in both source code and documentation files. Exceptions may be made when tabulating data in documentation files and strings. The length of lines in a paragraph may be much less than 79 characters if the line ends align better with the first line, as in this paragraph.

Encodings

In cases where an encoding for a `.py` file needs to be specified, such as when characters like α, β, or Å are used in docstrings, use UTF-8 encoding, i.e. start the file with the following line:

```
# -*- coding: utf-8 -*-
```

Imports

In addition to **PEP 8#imports**[1035] recommendations regarding imports, the following should be applied:

[1032] http://prody.csb.pitt.edu
[1033] https://www.python.org/dev/peps/pep-0008
[1034] https://www.python.org/dev/peps/pep-0008
[1035] https://www.python.org/dev/peps/pep-0008#imports

- relative intra-ProDy imports are discouraged, use `from prody.atomic import AtomGroup` not `from atomic import AtomGroup`
- always import from second top level module, use `from prody.atomic import AtomGroup` and not `from prody.atomic.atomgroup import AtomGroup`, because file names may change or files that grow too big may be split into smaller modules, etc.

Here is a series of properly formatted imports following a module documentation string:

```
"""This module defines a function to calculate something interesting."""

import os.path
from collections import defaultdict
from time import time

import numpy as np

from prody.atomic import AtomGroup
from prody.measure import calcRMSD
from prody.tools import openFile
from prody import LOGGER, SETTINGS

__all__ = ['calcSomething']
```

4.4.3 Whitespaces

In addition to recommendations regarding whitespace use in Python code (**PEP 8#whitespace-in-expressions-and-statements**[1036]), two whitespace characters should follow a period in documentation files and strings to help reading documentation in terminal windows and text editors.

4.4.4 Naming Conventions

ProDy naming conventions aim at making the library suitable for interactive sessions, i.e. easy to remember and type.

Class names

Naming style for classes is `CapitalizedWords` (or `CapWords`, or `CamelCase`). Abbreviations and/or truncated names should be used to keep class names short. Some class name examples are:

- *ANM* (page 140) for Anisotropic Network Model
- *HierView* (page 76) for Hierarchical View

Exception names

Prefer using a suitable standard-library exception over defining a new one. If you absolutely need to define one, use the class naming convention. Use the suffix "Error" for exception names, when exception is an error:

- *SelectionError* (page 99), the only exception defined in ProDy package

Method and function names

Naming style for methods and functions is `mixedCase`, that differs from `CapWords` by initial lowercase character. Starting with a lowercase (no shift key) and using no underscore characters decreases the number of key strokes by half in many cases in interactive sessions.

Method and function names should start with a verb, suggestive on the action, and followed by one or two names, where the second name may start with a lower case letter. Some examples are *moveAtoms()*

[1036] https://www.python.org/dev/peps/pep-0008#whitespace-in-expressions-and-statements

(page 220), *wrapAtoms()* (page 221), *assignSecstr()* (page 242), and *calcSubspaceOverlap()* (page 143).

Abbreviations and/or truncated names should be used and obvious words should be omitted to limit number of names to 20 characters. For example, *buildHessian()* (page 140) is preferred over `buildHessianMatrix()`. Another example is the change from using `getResidueNames()` to using *AtomGroup.getResnames()* (page 41). In fact, this was part of a series of major *Release Notes* (page 345) aimed at refining the library for interactive usage.

In addition, the following should be applied to enable grouping of methods and functions based on their action and/or return value:

- `buildSomething()`: methods and functions that calculate a matrix should start with `build`, e.g. *GNM.buildKirchhoff()* (page 164) and *buildDistMatrix()* (page 213)
- `calcSomething()`: methods that calculate new data but does not necessarily return anything and especially those that take timely actions, should start with `calc`, e.g. *PCA.calcModes()* (page 175)
- `getSomething()`: methods, and sometimes functions, that return a copy of data should start with `get`, such as *listReservedWords()* (page 75)
- `setSomething()`: methods, and sometimes functions, that alter internal data should start with `set`

4.4.5 Variable Names

Variable names in functions and methods should contain only lower case letters, and may contain underscore characters to increase readability.

4.5 Testing ProDy

4.5.1 Running Unittests

The easiest way to run ProDy unit tests is using nose[1037]. The following will run all tests:

```
$ nosetests prody
```

To skip tests that are slow, use the following:

```
$ nosetests prody -a '!slow'
```

To run tests for a specific module do as follows:

```
$ nosetests prody.tests.atomic prody.tests.sequence
```

4.5.2 Unittest Development

Unit test development should follow these guidelines:

1. For comparing Python numerical types and objects, e.g. int, list, tuple, use methods of `unittest.TestCase`[1038].

[1037] http://nose.readthedocs.org
[1038] http://docs.python.org/library/unittest.html#unittest.TestCase

2. For comparing Numpy arrays, use assertions available in `numpy.testing` module.
3. All test files should be stored in `tests` folder in the ProDy package directory, i.e. `prody/tests/`
4. All tests for functions and classes in a ProDy module should be in a single test file named after the module, e.g. `test_atomic/test_select.py`.
5. Data files for testing should be located in `tests/test_datafiles`.

4.6 Writing Tutorials

This is a short guide for writing ProDy tutorials that are published as part of online documentation pages, and also as individual downloadable PDF files.

4.6.1 Tutorial Setup

First go to `doc` folder in ProDy package and generate necessary files for your tutorial using **`start-tutorial.sh`** script:

```
$ cd doc
$ ./start-tutorial.sh
Enter tutorial title: ENM Analysis using ProDy
Enter a short title: ENM Analysis
Enter author name: First Last

Tutorial folders and files are prepared, see tutorials/enm_analysis
```

This will generate following folder and files:

```
$ cd tutorials/enm_analysis/
$ ls -lgo
-rw-r--r-- 1  328 Apr 30 16:48 conf.py
-rw-r--r-- 1  395 Apr 30 16:48 index.rst
-rw-r--r-- 1  882 Apr 30 16:48 intro.rst
-rw-r--r-- 1 1466 Apr 30 16:48 Makefile
lrwxrwxrwx 1   13 Apr 30 16:48 _static -> ../../_static
```

Note that short title will be used as filename and part of the URL of the online documentation pages.

If tutorial logo/image that you want to use is different from ProDy logo, update the following line in `conf.py`:

```
tutorial_logo = u'enm.png'      # default is ProDy logo
tutorial_prody_version = u''        # default is latest ProDy version
```

Also, note ProDy version if the tutorial is developed for a specific release.

4.6.2 Style and Organization

ProDy documentation and tutorials are written using reStructuredText[1039], an easy-to-read/write file format. See reStructuredText Primer[1040] for a quick introduction.

reStructuredText is stored in plain-text files with `.rst` extension, and converted to HTML and PDF pages using Sphinx[1041].

`index.rst` and `intro.rst` files are automatically generated. `index.rst` file should include title and table of contents of the tutorial. Table of contents is just a list of `.rst` files that are part of the tutorial. They be listed in the order that they should appear in the final PDF file:

```
.. _enm-analysis:

.. use "enm-analysis" to refer to this file, i.e. :ref:`enm-analysis`

*****************************************************************************
ENM Analysis using ProDy
*****************************************************************************


.. add .rst files to `toctree` in the order that you want them

.. toctree::
   :glob:
   :maxdepth: 2

   intro
```

Add more `.rst` files as needed. See other tutorials in `doc/tutorials` folder as examples.

4.6.3 Input/Output Files

All files needed to follow the tutorial should be stored in `tutorial_name_files` folder. There is usually no need to provide PDB files, as ProDy automatically downloads them when needed. Optionally, output files can also be provided.

Note: Small input and output files that contain textual information may be included in the **`git`** repository, but please avoid including large files in particular those that contain binary data.

4.6.4 Including Code

Python code in tutorials should be included using IPython Sphinx directive[1042]. In the beginning of each `.rst` file, you should make necessary imports as follows:

```
.. ipython:: python

   from prody import *
   from matplotlib.pylab import *
   ion()
```

This will convert to the following:

[1039] http://docutils.sourceforge.net/rst.html
[1040] http://sphinx-doc.org/rest.html
[1041] http://sphinx-doc.org/
[1042] http://ipython.org/ipython-doc/dev/development/ipython_directive.html

```
In [1]: from prody import *

In [2]: from matplotlib.pylab import *

In [3]: ion()
```

Then you can add the code for the tutorial:

```
.. ipython:: python

   pdb = parsePDB('1p38')
```

```
In [4]: pdb = parsePDB('1p38')
```

4.6.5 Including Figures

IPython directive should also be used for including figures:

```
.. ipython:: python

   @savefig tutorial_name_figure_name.png width=4in
   plot(range(10))

   @savefig tutorial_name_figure_two.png width=4in
   plot(range(100)); # used ; to suppress output
```

`@savefig` decorator was used to save the figure.

Note: Figure names needs to be unique within the tutorial and should be prefixed with the tutorial name.

Note that in the second `plot()` [1043] call, we used a semicolon to suppress the output of the function.

If you want to make modifications to the figure, save it after the last modification:

```
.. ipython:: python

   plot(range(10));
   grid();
   xlabel('X-axis')
   @savefig tutorial_name_figure_three.png width=4in
   ylabel('Y-axis')
```

4.6.6 Testing Code

If there is any particular code output that you want to test, you can use `@doctest` decorator as follows:

```
.. ipython::

   @doctest
   In [1]: 2 + 2
   Out[1]: 4
```

```
In [5]: 2 + 2
Out[5]: 4
```

Failing to produce the correct output will prevent building the documentation.

[1043] http://matplotlib.sourceforge.net/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

4.6.7 Publishing Tutorial

To see how your `.rst` files convert to HTML format, use the following command:

```
$ make html
```

You will find HTML files in `_build/html` folder.

Once your tutorial is complete and looks good in HTML (no code execution problems), following commands can be used to generate a PDF file and tutorial file achieves:

```
$ make pdf
$ make files
```

ProDy online documentation will contain these files as well as tutorial pages in HTML format.

4.7 Building the Website

This is a short guide for building the ProDy website.

4.7.1 Environment Setup

First log in to the ProDy webserver (prody.csb.pitt.edu) then run the following:

```
$ conda deactivate
```

This will then bring you into an environment with sphinx-build 1.3.5 and python 2.7, which is necessary for building the website.

Next change directory to the website root dir:

```
$ cd /var/www/html/prody
```

In this directory, you will find a number of directories, one of which will be attached to a symbolic link to ProDy-website. That directory will contain the current website and should not be changed!

Instead navigate to one of the others and build the website in there, such as ProDy-website-workdir. You can then copy files back over afterwards.

It's recommended to have the symbolic link called test_prody pointing to your build directory instead and then you can monitor changes by going to http://prody.csb.pitt.edu/test_prody/_build/html/ in your web browser.

4.7.2 Updating from GitHub

Changes to the website code are made using reStructuredText, which is stored in plain-text files with `.rst` extension, and converted to HTML and PDF pages using Sphinx[1044].

More information about the style etc. can be found in the instructions on making tutorials.

Any changes should be made on your local computer and added to the ProDy-Website GitHub repository via pull requests. These can then be pulled onto the ProDy webserver in your working directory:

[1044] http://sphinx-doc.org/

```
$ make pull
```

You may also need to install the ProDy in that directory again to make it get used during the building of the website. This can be done as follows:

```
$ cd ProDy
$ pip install -U . --user
$ cd ..
```

DruGUI and its tutorial are handled by their own GitHub repo. The following command can be used to update them:

```
$ make drugui
```

4.7.3 Publishing Changes

For generally making `.rst` files convert to HTML format, use the following command:

```
$ make html
```

You will find HTML files in `_build/html` folder.

It may also help to run the following first:

```
$ make clean
```

For tutorials, once a tutorial is complete and looks good in HTML (no code execution problems), the following commands can be used to generate a PDF file and tutorial file achieves:

```
$ make pdf
$ make files
```

ProDy online documentation will contain these files as well as tutorial pages in HTML format.

You can then copy files over to the main ProDy-Website directory to have them incorporated into the main prody website.

4.8 Making Windows Installers

MinGW[1045] (for 32-bit system) or MinGW-w64[1046] (for 64-bit system) can be used for compiling C modules when making Windows installers. Please follow the instructions[1047] to install and configure them.

libpython[1048] is also required to be installed. If the compiler complains, such as, "'::hypot' has not been declared", please refer to this link[1049].

4.9 Cross-platform Issues

This section describes cross-platform issues that may emerge and provides possible solutions for them.

[1045] http://www.mingw.org/
[1046] http://mingw-w64.org/
[1047] https://wiki.python.org/moin/WindowsCompilers
[1048] https://anaconda.org/anaconda/libpython
[1049] https://stackoverflow.com/questions/10660524/error-building-boost-1-49-0-with-gcc-4-7-0

4.9.1 Numpy integer type

Issues may arise when comparing Numpy integer types with Python `int()`. Python `int()` equivalent Numpy integer type on Windows (Win7 64bit, Python 32bit) is `int32`, while on Linux (Ubuntu 64bit) it is `int64`. For example, the statement `isinstance(np.array([1], np.int64), int)` may return **True** resulting in unexpected behavior in ProDy functions or methods. If Numpy integer type needs to be specified, using `int` seems a safe option.

4.9.2 Relative paths

`os.path.relpath()`[1050] function raises exceptions when the working directory and the path of interest are on separate drives, e.g. trying to write a `C:\temp` while running tests on `D:\ProDy`. Instead of this `os.path.relpath()`[1051], ProDy function *relpath()* (page 313) should be used to avoid problems.

[1050] http://docs.python.org/library/os.path.html#os.path.relpath
[1051] http://docs.python.org/library/os.path.html#os.path.relpath

CHAPTER

FIVE

RELEASE NOTES

5.1 ProDy 2.0 Series

5.1.1 2.0.2 (February 17, 2022)

New Features:

- Added **:function:'.realignModes'** for comparing sets of modes calculated on unaligned

structures

- New option *by_time* in **:function:'.showProjection'**, allowing projections onto 1 PC

and the frames sampled over the time course of a simulation or the conformers in an ensemble.

- Updates to ClustENM(D) to use latest and more efficient OpenMM version, 7.6
- New **:function:'.parseGromacsModes'** for parsing NMA and PCA modes calculated in Gromacs,

allowing the use of all-atom force fields and more efficient handling of large trajectories

Bug Fixes and Improvements:

- Bug fixes to searchPfam and fetchPfamMSA, which no longer worked due to some security change.
- Bug fix to Ensembles and Conformations for weights being indexed twice
- Bug fix to –quiet option of apps that fixes the VMD ProDy interface
- General documentation and error improvements

Full Changelog: https://github.com/prody/ProDy/compare/v2.0.1...v2.0.2

5.1.2 2.0.1 (Dec 20, 2021)

New Features:

- Added **:function:'.calcRWSIP'** for comparing sets of modes
- New methods to convert `Atom` (page 32), `Atomic` (page 47) and `EMDMAP` (page 234)

to TEMPy objects, allowing calculations such as cross-correlation coefficient (CCC) to EM maps.

- Added *calcDynamicFlexibilityIndex()* (page 179) and *calcDynamicCouplingIndex()* (page 179)

for further interpretation of PRS results.

- Reinstated the option of using forces in PRS as in ProDy v1.8 (*turbo=**False***)
- Added GitHub Actions Continuous Integration in place of Travis.

Bug Fixes and Improvements:

- Compatibility and bug fixes for various functions, including pyparsing for selections.
- Extended `AdaptiveANM` to work with other models including explicit membrane

ANM, *exANM* (page 150).

- Improved capabilities for fetching and parsing mmCIF and EMD files.
- Improved handling of residue and serial numbers including hexadecimal and hybrid36 formats.
- Consistency fixes.
- More non-standard amino acids (MEN, CSB, CME).

Full Changelog: https://github.com/prody/ProDy/compare/v2.0...v2.0.1

5.1.3 2.0 (Dec 30, 2020)

New Features:

ESSA

- New classes and functions for Essential Site Scanning Analysis (ESSA)

Updates for CryoDy

- Finalised the `AdaptiveANM` (initially added in v1.10.11) for exploring transitions between conformations.
- Improved domain decomposition

Expanded database module

- New interfaces including for QuartataWeb

New compounds module

- New modules for fetching and parsing compound data from the PDB including Biologically Interesting Reference Dictionary (BIRD) and Chemical Component Dictionary (CCD) CIF files
- New functions module including 2D chemical similarity calculations using Morgan Fingerprint Similarity.

Improved membrane ENMs

- New implementation of exANM based on iterative Schur complements and block-wise inversion
- New exGNM based on improved exANM

Bug Fixes and Improvements:

- New function `inferBonds()` in *AtomGroup* (page 38) for inferring bonds based on distances without information from `PSF` files.

5.2 ProDy 1.11 Series

5.2.1 1.11 (Oct 1, 2020)

New Features:

Rework of the Structure Mapping Functions

- Implemented `mapOntoChains()` that calculates mappings of chains between two structures in a pairwise fashion.
- Now newly added `mapChainOntoChain()` functions as the elementary operation of *mapOntoChains()* (page 230) use *mapChainOntoChain()* (page 229) that maps a *Chain* (page 53) instance onto another.
- Implemented *combineAtomMaps()* (page 231) for optimally combining atommaps obtained from `mapOntoChains` or `mapOntoChain`, based on mapping coverages.
- Implemented *alignChains()* (page 230) for aligning two structures using *mapOntoChains()* (page 230) and *combineAtomMaps()* (page 231).
- Now *buildPDBEnsemble()* (page 206) uses *alignChains()* (page 230) to perform the alignment. Therefore, instead of *mapping_func*, a *match_func* argument should be passed to the function for controlling how chains are (pre)matched. The *match_func* takes **exactly two** parameters, *chain1* and *chain2*, and determines if these two chains should be tried to be mapped.

  Built-in options for *match_func* includes:

  - *bestMatch()* (page 230) which allows all pairwise mappings between chains, and it is called *bestMatch()* (page 230) because the optimal mapping will be chosen later automatically.
  - *sameChid()* (page 231) which only maps the ones with the same chain ID, and the later optimization of mappings would not matter since the same chain IDs usually indicate a unique mapping.
  - *userDefined()* (page 231). It takes three parameters, which are two chains and a `dict`[1052] instance indicating correspondence Since it has three parameters, a wrapper is needed. Its usage is demonstrated by the following example:

```
>>> ref = parsePDB('3qel', subset='ca').select('chain A or chain B')
... mob = parsePDB('4pe5', subset='ca')
...
... chmap = {'3qel_ca': 'AB', '4pe5_ca': 'CD'}
... GluRChains = lambda chain1, chain2: userDefined(chain1, chain2, chmap)
... atommaps = alignChains(mob, ref, mapping='ce', match_func=GluRChains)
```

Bug Fixes and Improvements:

- Changed *showProtein()* (page 236) so that it does not display dummy atoms.
- Added warnings to `calcTransform()` when either *mobile* or *target* is an *AtomMap* (page 49) instance.
- Fixed a bug related to how the ticks are located in *showMatrix()* (page 304).
- Deprecated *mapChainByChain* and *addPDBEnsemble* whose rules are filled respectively by *mapOntoChains()* (page 230) and *buildPDBEnsemble()* (page 206) when *ref* is an *PDBEnsemble* (page 207) instance.

[1052] http://docs.python.org/library/stdtypes.html#dict

5.3 ProDy 1.10 Series

5.3.1 1.10.11 (Oct 25, 2019)

New Features:

- Added `calcDynamicFlexiblityIndex()`.
- Added "related_entries" to PDB header.
- Enabled *RTB* (page 187) to take a customized Hessian as input.
- Added numEntries to *NMA* (page 172), *Mode* (page 169), and *ModeSet* (page 171) objects.

Bug Fixes and Improvements:

- Added a keyword argument to `mapChainByChain()` to allow user-defined

correspondence of chain IDs between the target and the mobile.

- Compatibility and bug fixes for various functions.
- Fixed compilation error on MacOS if a user-installed gcc is present.
- Consistency fixes.
- Added new modules and functions for handling angles, dihedrals, impropers,

cross-terms, non-bonded exclusions, hydrogen bond donor and acceptor information from `PSF` files and including them in *AtomGroup* (page 38).

5.3.2 1.10.10 (May 9, 2019)

Bug Fixes and Improvements:

- Compatibility fix for the RTB module on Windows.
- Fixed compilation error on MacOS.
- Consistency fixes.

5.3.3 1.10.9 (May 2, 2019)

New Features:

- Added a *method* option to *pairModes()* (page 143) to use other solvers for the

linear assignment problem.

- Added functions for handling user-defined data for `Ensemble` and `PDBEnsemble`
- Added the *reweight* option to *ModeEnsemble* (page 191) to reweight the modes based on matched eigenvalues.
- *showSqFlucts()* (page 183) now plot color-coded square fluctuations using domain information.
- Added *calcSquareInnerProduct()* (page 143) for calculating the square inner product between two square fluctuations.
- Allowed *parseHiC()* (page 114) to automatically identify and read bindary hic files.

Bug Fixes and Improvements:

- Fixes to *parsePDB()* (page 268) for compatibility with large PDB files.
- Fixes and improvements to membrane ANM functions for faster computation.
- PY3K compatibility fixes.

5.3.4 1.10.8 (Sept 18, 2018)

New Features:

- Added *turbo* option to *calcEnsembleSpectralOverlaps()* (page 195) and *matchModes()* (page 143). Both options allow speed-up about 40%.

Bug Fixes and Improvements:

- *writePDB()* (page 270) now can accept *Ensemble* (page 202) as input.
- PY3K compatibility fixes.

5.3.5 1.10.7 (May 21, 2018)

New Features:

- Added *overlay* and *gap* option to *showAtomicLines()* (page 186).

Bug Fixes and Improvements:

- Reorganized MechStiff functions. Now MechStiff functions are moved out from the *ANM* (page 140) class, and *get...* were renamed to *calc...*
- Fixed a problem that an error will be raised when searching a sequence against *Pfam*.
- PY3K compatibility fixes.

5.3.6 1.10.4 (May 15, 2018)

- Minor fixes.

5.3.7 1.10.3 (May 14, 2018)

New Features:

- Added *CATHDB* (page 122) for querying information from CATH database.
- Added *sliceAtomicData()* (page 75) and *extendAtomicData()* (page 75) to slice more extend data based on *Selection* (page 100) or other types of *Atomic* (page 47) objects.

Bug Fixes and Improvements:

- Switched to Python Wheel for distributing on Windows. This will allow Windows users to *pip install prody*.
- Fixed a error when compiling C++ extensions on MacOS.
- Various bug fixes according to issues proposed on the GitHub.

5.3.8 1.10.2 (May 2, 2018)

- Minor fixes.

5.3.9 1.10.1 (May 1, 2018)

- Added the function sliceAtomicData for slicing data based on slicing atoms.
- Updated the documentation for making a release.
- Other documentation and minor fixes.

5.3.10 1.10 (Apr 30, 2018)

Signature Dynamics

- Added *calcEnsembleENMs()* (page 193) to compute ENMs on each conformation of a given ensemble to obtain an ensemble of modes.
- Added *ModeEnsemble* (page 191) and *sdarray* (page 192) classes as the basic data types for signature dynamics.
- Added functions such as *calcSignatureSqFlucts()* (page 196), *calcSignatureCrossCorr()* (page 196), *calcSignatureFractVariance()* (page 196) to extract signature dynamics.
- Added *calcEnsembleSpectralOverlaps()* (page 195) to obtain dynamical overlaps/distances among the conformations in a given ensemble.

New Features:

Visualization

- Added *showAtomicLines()* (page 186) and *showAtomicMatrix()* (page 184) functions to improve visualization.
- Added an *networkx* option to *showTree()* (page 184) so that the user can choose to use `networkx` to visualize a given tree.

Ensemble and PDBEnsemble

- Associated an *MSA* (page 285) object to the *PDBEnsemble* (page 207) class.
- Added an *pairwise* option to *Ensemble.getRMSDs()* (page 203) to obtain an RMSD table of every pair of conformations in the ensemble.
- Improved *Ensemble.setAtoms()* (page 204) for selecting a subset of residues/atoms of the ensemble.

Databases and Web Services

- Added methods and classes for obtaining data from *CATH* and *Dali*.
- Added additional functions for *Uniprot* and *Pfam* such as `queryUniprot()` (page 131) and `parsePfamPDBs()` (page 127).

Bug Fixes and Improvements:

- Fixed compatibility problems for Python 2 and 3.
- Improved the `saveModel()` (page 158) function to include class-specific features.
- Fixed a bug related to the `Atomgroup` addition method.
- Bug fixes to `NMA` (page 172) classes.
- Fixed a problem with `MSA` (page 285) indexing.
- Reorganized file structures and functions for consistency.
- Other bug fixes.

5.4 ProDy 1.9 Series

5.4.1 1.9.4 (Feb 02, 2018)

- Undocumented release and fixes.

5.4.2 1.9.3 (Oct 09, 2017)

Bugfixes

- Bug fix about http and ftp based pdb downloads.
- Bug fixes in PRS calculations.

5.4.3 1.9.2 (Aug 29, 2017)

** New Features**:

Migration to pypi.org

- All repositories are moved to pypi.org

5.4.4 1.9.1 (Aug 18, 2017)

** New Features**:

PDB Secondary Structures

- It is possible to write secondary structure infrmation to PDBs.

Bugfixes

- Fixed the problem about clang compiler for saxs tools.
- If FTP client is not working, HTTP client will be used when downloading PDBs.

5.4.5 1.9 (May 23, 2017)

New Features:

Perturbation Response Scanning

- Perturbation Response Scanning method is fully implemented with new plotting tools.
- Effectors and sensors are calculated from PRS tool.

Visualization with py3Dmol

- In jupyter notebook, if you have installed py3Dmol you can use py3Dmol visualization directly instead of simple matplotlib visualization.

mmcif parser

- Another structural format cif is also a part of ProDy parser now.

Bugfixes

- Various indexing issues are fixed.
- Some of the obsolete pdbs will not be downloaded anymore, instead replaced pdbs will be downloaded. This will change the priority between ftp and http servers.

5.5 ProDy 1.8 Series

5.5.1 1.8.2 (Jun 5, 2016)

- `addCoordset()` in *`PDBEnsemble`* (page 207) class, has an additional argument for NMR models.

5.5.2 1.8.1 (May 28, 2016)

Bugfixes

- `getHits()` in *`PDBBlastRecord`* (page 225) class, default overlap threshold changed to 0.7 to match with *`mapOntoChain()`* (page 230).
- `calcModes()` in *`RTB`* (page 187) have a bug on number of modes and fixed.
- Tab and indentation errors with Python 3.4 are fixed.

5.5.3 1.8 (May 13, 2016)

MechStiff

- Identification of the weakest/strongest elements of the structure architecture provided together with 3D vizualization and statistics analysis.

- Determination of the effective spring constant for selected pair of residues - useful for Single Molecule Force Spectroscopy (SMFS, AFM) and Steered Molecular Dynamics simulations.
- Evaluating the contributions of each mode to selected deformations

New Features:

Python 2 and 3 Support

- ProDy has been refactored to support Python 2.7 and 3.4. Windows installers for Python 2.7 and 3.4 are available in *Installation* (page 1).
- Unit tests are compatible with Python 2.7 and 3.4, and running them with other versions gives errors due to unavailability of some `unittest`[1053] features.

Bugfixes

- Various indexing issues are fixed.
- Compatibility issue of `searchPfam()` (page 126) with Python 2.7.11 is fixed.

5.6 ProDy 1.7 Series

5.6.1 1.7.1 (May 31, 2015)

Changes:

- `searchPfam()` uses hmmer for given sequence inputs instead of pfam search.

5.6.2 1.7 (Dec 23, 2013)

New Features:

- `buildPCMatrix()` (page 282) is implemented for calculation of coevolution with PSICOV method from multiple sequence alignments.
- `specMergeMSA()` is implemented for merging multiple sequence alignment files based on the species identifiers of sequences.
- `exANM` (page 150) is implemented for explicit membrane ANM calculations.
- `writeMembranePDB()` is implemented for writing PDB structures of created membranes for exANM class.

5.7 ProDy 1.6 Series

[1053] http://docs.python.org/library/unittest.html#module-unittest

5.7.1 1.6.1 (May 31, 2015)

Changes:

- `searchPfam()` uses hmmer for given sequence inputs instead of pfam search.

5.7.2 1.5 (Dec 23, 2013)

New Features:

- *buildPCMatrix()* (page 282) is implemented for calculation of coevolution with PSICOV method from multiple sequence alignments.
- `specMergeMSA()` is implemented for merging multiple sequence alignment files based on the species identifiers of sequences.
- *exANM* (page 150) is implemented for explicit membrane ANM calculations.
- `writeMembranePDB()` is implemented for writing PDB structures of created membranes for exANM class.

5.8 ProDy 1.5 Series

5.8.1 1.5.1 (Dec 24, 2013)

Changes:

- `PDBBlastRecord` become picklable.

5.8.2 1.5 (Dec 23, 2013)

New Features:

- *buildDirectInfoMatrix()* (page 282) and *calcMeff()* (page 282) are implemented for calculation of direct information from multiple sequence alignments.
- *showDirectInfoMatrix()* (page 288) and *showSCAMatrix()* (page 288) functions are implemented for displaying coevolutionary data.
- *RTB* (page 187) is implemented for Rotations-Translations of Blocks calculations. Optional arguments also permit *imANM* calculations.

Availability:

- Source is moved from `lib/prody` to `prody`.
- Source code will be hosted only at GitHub[1054].

Improvements:

- *DCDFile* (page 290) and *parseDCD()* (page 292) support DCD files written by **`cpptraj`**.

Testing:

- ProDy test command (**`prody test`**) and function `prody.test()` has been removed for easier maintenance of testing functions. See *Testing ProDy* (page 338) for more information on how to test ProDy.

[1054] http://github.com/prody/ProDy

5.9 ProDy 1.4 Series

5.9.1 1.4.9 (Nov 14, 2013)

Upcoming changes:

- Support for Python 3.1 and NumPy 1.5 will be dropped, meaning no Windows installers will be built for these versions of them.

Improvements:

- *HierView* (page 76) can handle *Residue* (page 80) instances that have same *segment* name, *chain* identifier, and *resnum*, if PDB file contains TER lines to terminate these residues. If these three identifiers are shared by multiple residues, indexing *AtomGroup* (page 38) instances will return a list of residues. This behavior can be used as follows. Note that in v1.5, this will be the default behavior.

```
>>> pdb_lines = """
... ATOM      1  O   WAT A   1       4.694  -3.891  -0.592  1.00  1.00
... ATOM      2  H1  WAT A   1       5.096  -3.068  -0.190  1.00  1.00
... ATOM      3  H2  WAT A   1       5.420  -4.544  -0.808  1.00  1.00
... TER
... ATOM      4  O   WAT A   1     -30.035  19.116  -2.193  1.00  1.00
... ATOM      5  H1  WAT A   1     -30.959  18.736  -2.244  1.00  1.00
... ATOM      6  H2  WAT A   1     -29.993  19.960  -2.728  1.00  1.00
... TER
... ATOM      7  O   WAT A   1     -77.584 -21.524 -37.894  1.00  1.00
... ATOM      8  H1  WAT A   1     -77.226 -21.966 -38.717  1.00  1.00
... ATOM      9  H2  WAT A   1     -77.023 -20.726 -37.674  1.00  1.00
... TER"""
>>> from StringIO import StringIO
>>> atoms = parsePDBStream(StringIO(pdb_lines))
```

Current behavior:

```
>>> print(atoms.numResidues())
1
>>> atoms['A', 1]
<Residue: WAT 1 from Chain A from Unknown (9 atoms)>
```

To activate the new behavior (which will be the default behavior in v1.5):

```
>>> hv = atoms.getHierView(ter=True)
>>> print(hv.numResidues())
>>> hv['A', 1]
```

- *parsePDB()* (page 268) reads TER records in PDB files. Atoms and hetero atoms (*hetatm*) that are followed by a TER record are now flagged as *pdbter*.

Bugfixes:

- Fixed memory leaks in *uniqueSequences()* (page 281) and *buildSeqidMatrix()* (page 280).

5.9.2 1.4.8 (Nov 4, 2013)

New Features:

- New analysis functions *buildOMESMatrix()* (page 281) and *buildSCAMatrix()* (page 281) are implemented.
- New *AtomGroup.numBytes()* (page 43) method returns an estimate of memory usage.
- New *countBytes()* (page 311) utility function is added for counting bytes used by NumPy arrays.

Improvements:

- *parsePDB()* (page 268) resizes data arrays to decrease memory usage.

Bugfixes:

- Fixed memory leaks in MSA *analysis* (page 279) functions.
- Fixed potential problems with importing contributed libraries.

5.9.3 1.4.7 (Oct 29, 2013)

Improvements:

- *AtomGroup* (page 38), *Selection* (page 100), and other *Atomic* (page 47) classes are picklable.
- Improved equality tests for *AtomGroup* (page 38). Two different instances are considered equal if they contain identical data and coordinate sets.

5.9.4 1.4.6 (Oct 16, 2013)

Bugfixes:

- Selection problem with using *resid* is fixed (issue 160[1055])
- Fixed a memory leak in MSA parsers written in C. When dealing with large files, leak would cause a segmentation fault.
- Fixed a memory leak in MSA parsers written in C. When dealing with large files, leak would cause a segmentation fault.
- Fixed a reference counting problem in MSA parsers in C that would cause segmentation fault when reading files that uses the same label for multiple sequences.
- Updated *fetchPDBLigand()* (page 119) to use PDB for fetching XML files.
- Revised handling of MSA file formats to avoid exceptions for unknown extensions.

[1055] https://github.com/prody/ProDy/issues/160

5.9.5 1.4.5 (Sep 6, 2013)

New Features:

- *parsePDBHeader()* (page 240) function can parse space group information from header section specified as `REMARK 290`, e.g. `parsePDBHeader('1mkp', 'space_group')` or `parsePDBHeader('1mkp')['space_group']`
- *heavy* selection flag is defined as an alias for *noh*.
- *matchChains()* (page 228) function can match non-hydrogen atoms using `subset='heavy'` keyword argument.
- Added `update_coords` keyword argument to `PCA.builCovariance()`, so that average coordinates calculated internally can be stored in ensemble or trajectory objects used as input.

Improvements:

- Unit tests can be run with Python 2.6 when *unittest2* module is installed.

Bugfixes:

- Fixed problems with reading compressed PDB files using Python 3.3.
- Fixed a bug in *parseSTRIDE()* (page 273) function that prevented reading files.
- Improved parsing of biomolecular transformations.
- Fixed memory allocation in C code used by *parseMSA()* (page 287) (Python 2.6).
- Fixed a potential name error in trajectory classes.
- Fixed problems in handling compressed files when using Python 2.6 and 3.3.
- Fixed a problem with indexing *NMA* (page 172) instances in Python 3 series.

5.9.6 1.4.4 (July 22, 2013)

Improvements:

- *writeNMD()* (page 174) and *parseNMD()* (page 174) write and read segment names. NMWiz is also improved to handle segment names. Improvements will be available in VMD v1.9.2.

Bugfixes:

- A bug in *saveAtoms()* (page 75) that would cause `KeyError`[1056] when bonds are set but fragments are not determined is fixed.
- Import ProDy would fail when `HOME`[1057] is not set. Changed *PackageSettings* (page 314) to handle this case graciously.

5.9.7 1.4.3 (June 14, 2013)

Changes:

- *getVMDpath()* (page 174) and *setVMDpath()* (page 174) functions are deprecated for removal, use *pathVMD()* (page 174) instead.
- Increased *blastPDB()* (page 226) *timeout* to 60 seconds.
- *extendModel()* (page 144) and *extendMode()* (page 144) functions have a new option for normalizing extended mode(s).

[1056] http://docs.python.org/library/exceptions.html#KeyError
[1057] http://matplotlib.sourceforge.net/install/environment_variables_faq.html#envvar-HOME

- *sampleModes()* (page 189) and *traverseMode()* (page 190) automatically normalizes input modes.

Bugfixes:

- A bug in *applyTransformation()* (page 219) is fixed. The function would interpret some external transformation matrices incorrectly.
- A bug in *fetchPDBLigand()* (page 119) function is fixed.

5.9.8 1.4.2 (April 19, 2013)

Improvements:

- *fetchPDB()* (page 266) and *fetchPDBfromMirror()* (page 266) functions can handle partial PDB mirrors. See *pathPDBMirror()* (page 266) for setting a mirror path.

Changes:

- MSE[1058] is included in the definition of non-standard amino acids, i.e. *nonstdaa*.

Bugfixes:

- Atom selection problems related to using *all* and *none* in composite selections, e.g. `'calpha and all'`, is fixed by defining these keywords as *Atom Flags* (page 64).
- Fasta files with sequence labels using multiple pipe characters would cause C parser (and so *parseMSA()* (page 287)) to fail. This issue is fixed by completely disregarding pipe characters.
- Empty chain identifiers for PDB hits would cause a problem in parsing XML results file and *blastPDB()* (page 226) would throw an exception. This case is handled by slicing the chain identifier string.
- A problem in *viewNMDinVMD()* (page 174) related to module imports is fixed.
- A problem with handling weights in *loadEnsemble()* (page 205) is fixed.

5.9.9 1.4.1 (Dec 16, 2012)

New Features:

- *buildSeqidMatrix()* (page 280) and *uniqueSequences()* (page 281) functions are implemented for comparing sequences in an *MSA* (page 285) object.
- *showHeatmap()* (page 167), *parseHeatmap()* (page 166), and *writeHeatmap()* (page 166) functions are implemented to support VMD plugin Heat Mapper[1059] file format.
- *Sequence* (page 289) is implemented to handle individual sequence records and point to sequences in *MSA* (page 285) instances.
- *evol occupancy* (page 22) application is implemented for refined MSA quality checking purposes.
- *mergeMSA()* (page 286) function and *evol merge* (page 21) application are implemented for merging Pfam MSA to study multi-domain proteins.

Improvements:

- *refineMSA()* (page 286) function and *evol refine* (page 25) application can perform MSA refinements by removing similar sequences.
- *writePDB()* (page 270) function takes *beta* and *occupancy* arguments to be outputted in corresponding columns.

[1058] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MSE
[1059] http://www.ks.uiuc.edu/Research/vmd/plugins/heatmapper/

- *MSA* (page 285) indexing and slicing are revised and improved.
- *parseMSA()* (page 287) is improved to handle indexing of sequences that have the same label in an MSA file, e.g. domains repeated in a protein.
- *prody anm* (page 4), *prody gnm* (page 11), and *prody pca* (page 13) applications can write heatmap files for visualization using NMWiz and Heatmapper plugins.
- Several improvements made to handling sequence labels in Pfam MSA files. Files that contain sequence parts with same protein UniProt ID are handled delicately.

Changes:

- ProDy will not emit a warning message when a wwPDB server is not set using *wwPDBServer()* (page 277), and use the default US server.
- Indexing *MSA* (page 285) returns *Sequence* (page 289) instances.
- Iterating over *MSA* (page 285) and *MSAFile* (page 287) yields *Sequence* (page 289) instances.

Bugfixes:

- Fixed a syntax problem that prevented running ProDy using Python 2.6.
- Fixed *NMA* (page 172) indexing problem that was introduced in v1.4.

Normal Mode Wizard

- NMWiz can visualize heatmaps linked to structural view via Heatmapper. Clicking on the heatmap will highlight atom or residue pairs.
- ProDy interface has the option to write and load cross-correlations.
- NMWiz can determined whether a model is an extended model. For extended models plotting mobility has been improved. Only a single value per residue will be plotted, and clicking on the plot will highlight all of the residue atoms.

5.9.10 1.4 (Dec 2, 2012)

New Features:

Python 3 Support

- ProDy has been refactored to support Python 3. Windows installers for Python 2.6, 2.7, 3.1, and 3.2 are available in *Installation* (page 1).
- Unit tests are compatible with Python 2.7 and 3.2, and running them with other versions gives errors due to unavailability of some `unittest`[1060] features.

Sequence Analysis

- New applications *Evol Applications* (page 16) are available.
- *searchPfam()* (page 126) and *fetchPfamMSA()* (page 126) functions are implemented for searching and retrieving Pfam data. See MSA Files[1061] for usage examples.
- *MSAFile* (page 287) class, *parseMSA()* (page 287) and *writeMSA()* (page 288) functions are implemented for reading and writing multiple sequence alignments. See MSA Files[1062] for usage examples.
- *MSA* (page 285) class has been implemented for storing and manipulating MSAs in memory.

[1060] http://docs.python.org/library/unittest.html#module-unittest
[1061] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msafiles.html#msafiles
[1062] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msafiles.html#msafiles

- *calcShannonEntropy()* (page 279), *buildMutinfoMatrix()* (page 279), and *calcMSAOccupancy()* (page 279) functions are implemented implemented for MSA analysis. See Evolution Analysis[1063] for usage examples.
- *showShannonEntropy()* (page 288), *showMutinfoMatrix()* (page 288), and *showMSAOccupancy()* (page 288) functions are implemented implemented for MSA analysis. See Evolution Analysis[1064] for usage examples.
- *applyMutinfoCorr()* (page 280) and *applyMutinfoNorm()* (page 280) functions are implemented for applying normalization and corrections to mutual information matrices.
- *calcRankorder()* (page 280) function is implemented for identifying highly correlated/co-evolving pairs of residues.

Bugfix:

- Fixed selection issues involving use of `x` or negative numbers.

5.10 ProDy 1.3 Series

5.10.1 1.3.1 (Nov 6, 2012)

New Features:

- Added *fetchPDBviaHTTP()* (page 278) and *fetchPDBviaFTP()* (page 277) functions.
- Added *copyFile()* (page 312) function to *utilities* (page 300).
- Added `prody test` command for convenient testing of ProDy package.

Improvements:

- Improved *gunzip()* (page 312) function to handle `.gz` extensions and string buffers.

Changes:

- `getWWPDBFTPServer()` and `setWWPDBFTPServer()` are deprecated for removal in v1.4, use *wwPDBServer()* (page 277) instead.
- `getPDBLocalFolder()` and `setPDBLocalFolder()` are deprecated for removal in v1.4, use *pathPDBFolder()* (page 266) instead.
- `getPDBMirrorPath()` and `setPDBMirrorPath()` are deprecated for removal in v1.4, use *pathPDBMirror()* (page 266) instead.
- `getPDBCluster()` is deprecated for removal in v1.4, use *listPDBCluster()* (page 267) instead.
- `getReservedWords()` is deprecated for removal in v1.4, use *listReservedWords()* (page 75) instead.
- *getNonstdProperties()* (page 73) is deprecated for removal in v1.4, use *listNonstdAAProps()* (page 73) instead.

Bugfix:

[1063] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msaanalysis.html#msa-analysis
[1064] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msaanalysis.html#msa-analysis

- Fixed a bug in *HierView* (page 76) that would cause wrong assignment of residue/chain indices to atoms when residue or chain atoms are separated by atoms of other entities. This would also caused problems when making keyword selections, such as *protein*.
- Added dummy atom check in *Ensemble.setAtoms()* (page 204) and *Trajectory.setAtoms()* (page 298) methods to avoid indexing problems.

5.10.2 1.3 (Sep 30, 2012)

Improvements:

- *select* (page 92) module and its documentation are completely rewritten. *Select* (page 99) class uses simplest possible parser to evaluate selection strings and achieves more than 25% speed-up on average.
- *Atom Selections* (page 92) become more forgiving of small typos, but will issue warning messages when they are detected via *SelectionWarning* (page 99). These messages can be turned of using *confProDy()* (page 329)
- Functions used in *ProDy Applications* (page 3) have been refactored to allow for using them directly. See *apps* (page 314) for their documentation.

Bugfix:

- A problem in *prody catdcd* (page 7) command that was introduced when refactoring *trajectory* (page 289) classes is fixed.

5.11 ProDy 1.2 Series

5.11.1 1.2.1 (Sep 6, 2012)

If you are upgrading from ProDy v1.1, see also the below changes introduced in v1.2.

Bugfix:

- A problem in `select`[1065] module regarding Numpy numeric types is fixed. Problem would emerge on platforms which do not offer some numeric types, e.f. `np.float16`.
- Fixed problems in *prody anm* (page 4), *prody gnm* (page 11), and *prody fetch* (page 10) related to writing output files.

Changes:

- The way that *prody fetch* (page 10) command handles files containing PDB identifiers has changed.

5.11.2 1.2 (Aug 30, 2012)

Important Changes:

Package folder `prody` is moved into `lib` folder to prevent exceptions related to importing compiled packages from the installation folder.

[1065] http://docs.python.org/library/select.html#module-select

Some changes in *Trajectory* (page 296) and *Ensemble* (page 202) methods related to linking, setting, and selecting atoms were made to make the interface more intuitive. These changes, which may break your code, are as follows:

- *AtomGroup* (page 38) instances can be linked to a *Trajectory* (page 296) using *Trajectory.link()* (page 297) method and linking status of an instance can be checked using *Trajectory.isLinked()* (page 297) medhod.
- *Trajectory.setAtoms()* (page 298) method accepts *AtomGroup* (page 38) and *Selection* (page 100) instances and should be used to select a subset of atoms. This method will not link *AtomGroup* (page 38) instance to the trajectory and also will not update the reference coordinates of the instance.
- `Trajectory.select()` and *Ensemble.select()* (page 204) methods are removed and their functions are overloaded to *Trajectory.setAtoms()* (page 298) and *Ensemble.setAtoms()* (page 204) methods, respectively.
- `Trajectory.getSelection()` and `Ensemble.getSelection()` methods are removed, use *Trajectory.getAtoms()* (page 296) and *Ensemble.getAtoms()* (page 203) instead.
- *Trajectory* (page 296) reference coordinates must be changed using *Trajectory.setCoords()* (page 298) method.

For usage examples see Trajectory Analysis[1066], Trajectory Analysis II[1067], Frames and Atom Groups[1068], and Trajectory Output[1069].

New Features:

- *Atom Flags* (page 64), that are used in *Atom Selections* (page 92), is implemented. See its documentation for handy usage examples.
- *sortAtoms()* (page 75) function is implemented.
- *pickCentralConf()* (page 217) function is implemented to pick the conformation or the active coordinate set that is closest to the average of coordinate sets.
- *writePSF()* (page 294), a simple PSF file writer, is implemented.
- *glob()* (page 313) utility function is implemented.
- *iterPDBFilenames()* (page 266) function is implemented, which can be used to iterate over all PDB files stored in a local mirror of Protein Data Bank.
- *findPDBFiles()* (page 266) function is implemented, which can be used to access PDB files in a path.

Improvements:

- *HierView* (page 76) instances are built more efficiently. Two times speed-up is achieved by delaying instantiation of *Chain* (page 53) and *Residue* (page 80) instances until they are needed.
- Multiple *Atom Flags* (page 64) can be used in *Atom Selections* (page 92) without using `'and'` operator, e.g. `'sidechain carbon'` is the same as `'sidechain and carbon'`.
- *writePDB()* (page 270) accepts *Ensemble* (page 202), *Conformation* (page 201), and *Frame* (page 293) instances as atoms argument.
- *writePDB()* (page 270) function is around 25% faster.

[1066] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory.html#trajectory
[1067] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory2.html#trajectory2
[1068] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/frame.html#frame
[1069] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/outputtraj.html#outputtraj

- *pickCentral()* (page 217) is extended to accept *Atomic* (page 47) and *Ensemble* (page 202) instances. Old function is now *pickCentralAtom()* (page 217).
- *prody align* (page 3) command and *prody_align()* (page 320) function can handle non-protein atom selections (see examples for *prody align* (page 3)).
- *parsePDB()* (page 268) and *writePDB()* (page 270) supports 100K and more atoms.

Changes:

- *showOverlapTable()* (page 181) displays first set of modes along x axis of the plot.
- *AtomGroup.setData()* (page 45) does not accept arrays with boolean data type, use *AtomGroup.setFlags()* (page 45) instead.
- *writePDB()* (page 270) function argument *model* is changed to *csets* that indicates the coordinate set index of *atoms* argument.
- *PackageLogger.timing()* (page 309) does not return elapsed time, only logs this information.
- `PackageLogger.startLogfile()` is deprecated for removal in v1.3, use *PackageLogger.start()* (page 309) instead.
- `PackageLogger.closeLogfile()` is deprecated for removal in v1.3, use *PackageLogger.close()* (page 309) instead.
- `from prody.utilities import *` will not work anymore due to potential name conflicts with Python standard library functions. Import required functions explicitly.
- *writePDB()* (page 270) appends `.pdb` extension to filename when it is not present
- *prody select* (page 14) command positional argument order is changed to allow for handling multiple PDBs at a time. Old older will be supported until v1.4, but a warning message will be issued.
- *select* argument in *alignCoordsets()* (page 219) is removed, make selection outside of the function instead.

Deprecations:

- `AtomGroup.getHeteros()` method has been deprecated for removal in v1.3, use `getFlags('hetatm')` instead.
- `AtomMap.getMappedFlags()` and `AtomMap.getDummyFlags()` methods have been deprecated for removal in v1.3, use `getFlags('mapped')` and `getFlags('dummy')` instead.
- `getVerbosity()` and `setVerbosity()` are deprecated for removal in v1.3, use *confProDy()* (page 329) instead which save changes permanently.
- `NMA.getModes()` and `ModeSet.getModes()` methods are deprecated for removal in v1.3, use `list()`, e.g. `list(model)`, instead.

Bugfixes:

- Fixed a bug in *prody contacts* (page 8) command that arose problems when when selecting a subset of the target atoms.

Normal Mode Wizard

Improvements:

- *ProDy Interface* shows the size of the trajectory output file for PCA calculations.
- *Mode Graphics Options* allows for copying arrows settings from one mode to another.
- Color scale method and midpoint for protein coloring based on mobility and bfactors can be adjusted from *Protein Graphics Options* panel.

5.12 ProDy 1.1 Series

5.12.1 1.1 (June 1, 2012)

New Features:

- `iterFragments()` (page 74) function is added.
- `findNeighbors()` (page 213) function is added.
- `calcMSF()` (page 215) and `calcRMSF()` (page 215) functions are added.
- `wrapAtoms()` (page 221) functions is added.
- `extendMode()` (page 144) and `extendVector()` (page 144) functions are added.
- *prody contacts* (page 8) command is added.

Improvements:

- `moveAtoms()` (page 220) function is improved to move atoms to a specified location.
- `DCDFile` (page 290) and `parseDCD()` (page 292) take *astype* keyword argument for automatic type recasting for coordinate arrays. This option can be used to convert 32-bit coordinate arrays to 64-bit automatically for higher precision calculations.
- Commands *prody anm* (page 4), *prody gnm* (page 11), and *prody pca* (page 13) can extend a coarse grained model to backbone or all atoms of the residues. See their documentation pages.

Changes:

- Color scale used by `showOverlapTable()` (page 181) is normalized by default.
- `tools` module is deprecated for removal, use `utilities` (page 300) instead.
- *array* argument in `moveAtoms()` (page 220) is replaced with *by* keyword argument.
- *which* argument in `AtomGroup.copy()` (page 39) method is deprecated for removal in version 1.2.
- `DCDFile` (page 290) does not log information for most common type of DCD file, i.e. 32-bit CHARMM format.
- `Trajectory.getNextIndex()` method is deprecated for removal in v1.2, use `nextIndex()` (page 297) instead.

Bugfixes:

- Fixed several problems in `iterNeighbors()` (page 213) function and `Contacts` (page 213) class that were introduced after transition to new `KDTree` (page 210) interface.
- Fixed a problem in setting selection strings of fragments identified using `findFragments()` (page 74).
- Fixed a problem in `calcCenter()` (page 214) related to weighted center calculation.
- Fixed a problem of in copying `AtomMap` (page 49) instances, which would emerge when bond information was present in unusual mappings, such as when atom orders are changed or an atom is present multiple times in the mapping.

Normal Mode Wizard

Improvements:

- Mode scaling options are improved.
- Options added for extending coarse grained NMA models to residue backbone or all atoms.

5.13 ProDy 1.0 Series

5.13.1 1.0.4 (May 2, 2012)

Bugfixes:

- Fixed a problem in *calcPhi()* (page 214) function that raised a name error.
- Fixed a problem in *KDTree.getDistances()* (page 211) method that raised a name error when unitcell is provided.
- Fixed a problem in *buildDistMatrix()* (page 213) and *calcDistance()* (page 214) functions causing miscalculations when unitcell is given.
- Revised *KDTree* (page 210) methods dealing with to handle special cases where unitcell might have some dimensions zero.

Changes:

- `buildKDTree()` method is removed, earlier than planned due to unexpected bugfix releases.

5.13.2 1.0.3 (May 1, 2012)

Bugfixes:

- Fixed *kdtree* (page 210) import problem.

New Features:

- *buildDistMatrix()* (page 213) function that can take periodic boundary conditions is implemented.

Improvements:

- *calcDistance()* (page 214) function is improved to take periodic boundary conditions into account when provided by the users.

5.13.3 1.0.2 (May 1, 2012)

New Features:

- Methods to deal with connected subsets of atoms are implemented, see *AtomGroup.iterFragments()* (page 42) and *AtomGroup.numFragments()* (page 43).
- *pickCentral()* (page 217) method is implemented for picking the atom that is closest to the centroid of a group or subset of atoms.

- ProDy configuration option *auto_secondary* is implemented to allow for parsing and assigning secondary structure information from PDB file header data automatically. See *assignSecstr()* (page 242) and *confProDy()* (page 329) for usage details.
- **prody align** makes use of *select* when aligning multiple structures. See usage examples: *prody align* (page 3)
- *printRMSD()* (page 221) function that prints minimum, maximum, and mean RMSD values when comparing multiple coordinate sets is implemented.
- *findFragments()* (page 74) function that identifies fragments in atom subsets, e.g. *Selection* (page 100), is implemented.
- A new *KDTree* (page 210) interface with coherent method names and capability to handle periodic boundary conditions is implemented.

Improvements:

- Performance improvements made in *saveAtoms()* (page 75) and *loadAtoms()* (page 75).
- *sliceMode()* (page 144), *sliceModel()* (page 145), *sliceVector()* (page 145), and *reduceModel()* (page 145) functions accept *Selection* (page 100) instances as well as selection strings. In repeated use of this function, if selections are already made out of the function, considerable speed-ups are achieved when selection is passed instead of selection string.
- Fragment iteration (*AtomGroup.iterFragments()* (page 42)) is improved to yield items faster.

Changes:

- There is a change in the behavior of addition operation on instances of *AtomGroup* (page 38). When operands do not have same number of coordinate sets, the result will have one coordinate set that is concatenation of the *active coordinate sets* of operands.
- `buildKDTree()` function is deprecated for removal, use the new *KDTree* (page 210) class instead.

Bugfixes:

- A problem in building hierarchical views when making selections using *resindex*, *chindex*, and *segindex* keywords is fixed.
- A problem in *Chain* (page 53) and *Residue* (page 80) selection strings that would emerge when a *HierView* (page 76) is build using a selection is fixed.
- A problem with copying *AtomGroup* (page 38) instances whose coordinates are not set is fixed.
- *AtomGroup* (page 38) fragment detection algorithm is rewritten to avoid the problem of reaching maximum recursion depth for large molecules with the old recursive algorithm.
- A problem with picking central atom of *AtomGroup* (page 38) instances in *pickCentral()* (page 217) function is fixed.
- A problem in *Select* (page 99) class that caused exceptions when evaluating complex macro definitions is fixed.
- Fixed a problem in handling multiple trajectory files. The problem would emerge when a file was added (*addFile()* (page 296)) to a *Trajectory* (page 296) after atoms were set (*setAtoms()* (page 298)). Newly added file would not be associated with the atoms and coordinates parsed from this file would not be set for the *AtomGroup* (page 38) instance.

5.13.4 1.0.1 (Apr 6, 2012)

New Features:

- ProDy can be configured to automatically check for updates on a regular basis, see *checkUpdates()* (page 329) and *confProDy()* (page 329) functions for details.
- `alignPDBEnsemble()` function is implemented to align PDB files using transformations calculated in ensemble analysis. See usage example in Homologous Proteins[1070] example.
- *PDBConformation.getTransformation()* (page 202) is implemented to return the transformation that was used to superpose conformation onto reference coordinates. This transformation can be used to superpose the original PDB file onto the reference PDB file.
- Amino acid sequences with regular expressions can be used to make atom selections, e.g. `'sequence "C..C"'`. See *Atom Selections* (page 92) for usage details.
- *calcCrossProjection()* (page 138) function is implemented.

Improvements:

- *Select* (page 99) class raises a *SelectionError* (page 99) when potential typos are detected in a selection string, e.g. `'chain AB'` is a grammatically correct selection string that will return **None** since no atoms have chain identifier `'AB'`. In such cases, an exception noting that values exceed maximum number of characters is raised.
- **`prody align`** command accepts percent sequence identity and overlap parameters used when matching chains from given multiple structures.
- When using **`prody align`** command to align multiple structure, all models in NMR structures are aligned onto the reference structure.
- **`prody catdcd`** command accepts `--align SELSTR` argument that can be used to align frames when concatenating files.
- *showProjection()* (page 181) and *showCrossProjection()* (page 182) functions are improved to evaluate list of markers, color, labels, and texts. See usage example in Plotting[1071].
- *Trajectory* (page 296) instances can be used for calculating and plotting projections using *calcProjection()* (page 138), *showProjection()* (page 181), *calcCrossProjection()* (page 138), and *showCrossProjection()* (page 182) functions.

Changes:

- Phosphorylated amino acids, phosphothreonine (*TPO*), O-phosphotyrosine (*PTR*), and phosphoserine (*SEP*), are recognized as acidic protein residues. This prevents having breaks in protein chains which contains phosphorylated residues. See *Atom Selections* (page 92) for definitions of *protein* and *acidic* keywords.
- Hit dictionaries from *PDBBlastRecord* (page 225) will use *percent_overlap* instead of *percent_coverage*. Older key will be removed in v1.1.
- `Transformation.get4x4Matrix()` method is deprecated for removal in v1.1, use *Transformation.getMatrix()* (page 218) method instead.

Bugfixes:

- A bug in some *ProDy Applications* (page 3) is fixed. The bug would emerge when invalid arguments were passed to effected commands and throw an unrelated exception hiding the error message related to the arguments.
- A bug in `'bonded to ...'` is fixed that emerged when `'...'` selected nothing.
- A bug in `'not'` selections using `.` operator is fixed.

[1070] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/blast.html#pca-blast
[1071] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_plotting.html#pca-xray-plotting

5.13.5 1.0 (Mar 7, 2012)

Improvements:

- *ANM.buildHessian()* (page 140) method is not using a KDTree by default, since with some code optimization the version not using KDTree is running faster. Same optimization has gone into *GNM.buildKirchhoff()* (page 164) too, but for Kirchoff matrix, version using KDTree is faster and is the default. Both methods have *kdtree* argument to choose whether to use it or not.
- **prody** script is updated. Importing Prody and Numpy libraries are avoided. Script responses to help queries faster. See *ProDy Applications* (page 3) for script usage details.
- Added `bonded to ...` selection method that expands a selection to immediately bound atoms. See *Atom Selections* (page 92) for its description.
- *fetchPDBLigand()* (page 119) parses bond data from the XML file.
- *fetchPDBLigand()* (page 119) can optionally save compressed XML files into ProDy package folder so that frequent access to same files will be more rapid. See *confProDy()* (page 329) function for setting this option.
- *Select* (page 99) class is revised. All exceptions are handled delicately to increase the stability of the class.
- Distance based atom selection is 10 to 15% faster for atom groups with more than 5K atoms.
- Added uncompressed file saving option to *prody blast* (page 6) command.

Changes:

- All deprecated method and functions scheduled for removal are removed.
- `getEigenvector()` and `getEigenvalue()` methods are deprecated for removal in v1.1, use *Mode.getEigvec()* (page 170) and *Mode.getEigval()* (page 169) instead.
- `getEigenvectors()` and `getEigenvalues()` methods are deprecated for removal in v1.1, use *NMA.getEigvecs()* (page 172) and *NMA.getEigvals()* (page 172) instead.
- `Mode.getCovariance()` and *ModeSet.getCovariance()* (page 171) methods are deprecated for removal in v1.1, use *calcCovariance()* (page 137) method instead.
- `Mode.getCollectivity()` method is removed, use *calcCollectivity()* (page 137) function instead.
- `Mode.getFractOfVariance()` method is removed, use the new *calcFractVariance()* (page 137) function instead.
- `Mode.getSqFlucts()` method is removed, use *calcSqFlucts()* (page 137) function instead.
- Renamed `showFractOfVar()` function as *showFractVars()* (page 180) function instead.
- Removed `calcCumOverlapArray()`, use *calcCumulOverlap()* (page 142) with `array=True` argument instead.
- Renamed `extrapolateModel()` as *extendModel()* (page 144).
- The relation between *AtomGroup* (page 38), *Trajectory* (page 296), and *Frame* (page 293) instances have changed. See Trajectory Analysis II[1072] and Trajectory Output[1073], and Frames and Atom Groups[1074] usage examples.
- *AtomGroup* (page 38) cannot be deformed by direct addition with a vector instance.

[1072] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory2.html#trajectory2
[1073] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/outputtraj.html#outputtraj
[1074] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/frame.html#frame

- Unmapped atoms in *AtomMap* (page 49) instances are called dummies. `AtomMap.numUnmapped()` method, for example, is renamed as *AtomMap.numDummies()* (page 52).
- *fetchPDBLigand()* (page 119) accepts only *filename* (instead of *save* and *folder*) argument to save an XML file.

Bugfixes:

- A problem in distance based atom selection which would could cause problems when a distance based selection is made from a selection is fixed.
- Changed *prody blast* (page 6) so that when a path for downloading files are given files are not save to local PDB folder.

5.14 ProDy 0.9 Series

5.14.1 0.9.4 (Feb 4, 2012)

Changes:

- `setAtomGroup()` and `getAtomGroup()` methods are renamed as *Ensemble.setAtoms()* (page 204) and *Ensemble.getAtoms()* (page 203).
- *AtomGroup* (page 38) class trajectory methods, i.e. `AtomGroup.setTrajectory()`, `AtomGroup.getTrajectory()`, `AtomGroup.nextFrame()`, `AtomGroup.nextFrame()`, and `AtomGroup.gotoFrame()` methods are deprecated. Version 1.0 will feature a better integration of *AtomGroup* (page 38) and *Trajectory* (page 296) classes.

Bugfixes:

- Bugfixes in *Bond.setACSIndex()* (page 53), *saveAtoms()* (page 75), and *HierView.getSegment()* (page 76).
- Bugfixes in *GammaVariableCutoff* (page 161) and *GammaStructureBased* (page 159) classes.
- Bugfix in *calcCrossCorr()* (page 137) function.
- Bugfixes in *Ensemble.getWeights()* (page 204), *showOccupancies()* (page 206), *DCDFile.flush()* (page 290).
- Bugfixes in ProDy commands *prody blast* (page 6), *prody fetch* (page 10), and *prody pca* (page 13).
- Bugfix in *calcCenter()* (page 214) function.

5.14.2 0.9.3 (Feb 1, 2012)

New Features:

- *DBRef* (page 239) class is implemented for storing references to sequence databases parsed from PDB header records.

- Methods for storing coordinate set labels in *AtomGroup* (page 38) instances are implemented: *getACSLabel()* (page 39), and *getACSLabel()* (page 39).
- *calcCenter()* (page 214) and *moveAtoms()* (page 220) functions are implemented for dealing with coordinate translation.
- Hierarchical view, *HierView* (page 76), is completely redesigned. PDB files that contain non-empty segment name column (or when such information is parsed from a PSF file), new design delicately handles this information to identify distinct chains and residues. This prevents merging distinct chains in different segments but with same identifiers and residues in those with same numbers. New design is also using ordered dictionaries `collections.OrderedDict`[1075] and lists so that chain and residue iterations yield them in the order they are parsed from file. These improvements also bring modest improvements in speed.
- *Segment* (page 86) class is implemented for handling segments of atoms defined in molecular dynamics simulations setup, using **psfgen** for example.
- Context manager methods are added to trajectory classes. A trajectory file can be opened as follows:

```
with Trajectory('mdm2.dcd') as traj:
    for frame in traj:
        calcGyradius(frame)
```

- *Chain* (page 53) slicing is implemented:

```
p38 = parsePDB('1p38')
chA = p38['A']
res_4to10 = chA[4:11]
res_100toLAST = chA[100:]
```

- Some support for bonds is implemented to *AtomGroup* (page 38) class. Bonds can be set using *setBonds()* (page 44) method. All bonds must be set at once. *iterBonds()* (page 42) or *iterBonds()* (page 35) methods can be used to iterate over bonds in an AtomGroup or an Atom.
- *parsePSF()* (page 294) parses bond information and sets to the atom group.
- *Selection.update()* (page 105) method is implemented, which may be useful to update a distance based selection after coordinate changes.
- `buildKDTree()` and *iterNeighbors()* (page 213) methods are implemented for facilitating identification of pairs of atoms that are proximal.
- *iterAtoms()* (page 42) method is implemented to all *atomic* (page 29) classes to provide uniformity for atom iterations.
- *calcAngle()* (page 214), *calcDihedral()* (page 214), *calcPhi()* (page 214), *calcPsi()* (page 214), and *calcOmega()* (page 214) methods are implemented.

Improvements:

- *Chain.getSelstr()* (page 56) and *Residue.getSelstr()* (page 83) methods are improved to include the selection string of a *Selection* (page 100) when they are built using one.

Changes:

- *Residue* (page 80) methods `getNumber()`, `setNumber()`, `getName()`, `setName()` methods are deprecated and will be removed in v1.0.
- *Chain* (page 53) methods `getIdentifier()` and `setIdentifier()` methods are deprecated and will be removed in v1.0.
- *Polymer* (page 238) attribute `identifier` is renamed as *chid* (page 239).

[1075] http://docs.python.org/library/collections.html#collections.OrderedDict

- *Chemical* (page 236) attribute `identifier` is renamed as *resname* (page 237).
- `getACSI()` and `setACSI()` are renamed as *getACSIndex()* (page 39) and *setACSIndex()* (page 44), respectively.
- `calcRadiusOfGyration()` is deprecated and will be removed in v1.0. Use *calcGyradius()* (page 214) instead.

Bugfixes:

- Fixed a problem in *parsePDB()* (page 268) that caused loosing existing coordinate sets in an *AtomGroup* (page 38) when passed as *ag* argument.
- Fixed a problem with `"same ... as ..."` argument of *Select* (page 99) that selected atoms when followed by an incorrect atom selection.
- Fixed another problem with `"same ... as ..."` which result in selecting multiple chains when same chain identifier is found in multiple segments or multiple residues when same residue number is found in multiple segments.
- Improved handling of negative integers in indexing *AtomGroup* (page 38) instances.

5.14.3 0.9.2 (Jan 11, 2012)

New Features:

- **`prody catdcd`** command is implemented for concatenating and/or slicing `.dcd` files. See *prody catdcd* (page 7) for usage examples.
- *DCDFile* (page 290) can be opened in write or append mode, and coordinate sets can be added using *write()* (page 292) method.
- `getReservedWords()` can be used to get a list of words that cannot be used to label user data.
- *confProDy()* (page 329) function is added for configuring ProDy.
- ProDy can optionally backup existing files with `.BAK` (or another) extension instead of overwriting them. This behavior can be activated using *confProDy()* (page 329) function.

Improvements:

- *writeDCD()* (page 293) file accepts *AtomGroup* (page 38) or other *Atomic* (page 47) instances as *trajectory* argument.
- **`prody align`** command can be used to align multiple PDB structures.
- **`prody pca`** command allows atom selections for DCD files that are accompanied with a PDB or PSF file.

Changes:

- *DCDFile* (page 290) instances, when closed, raise exception, similar to behavior of `file` objects in Python.
- Title of *AtomGroup* (page 38) instances resulting from copying an *Atomic* (page 47) instances does not start with 'Copy of'.
- `changeVerbosity()` and `getVerbosityLevel()` are renamed as `setVerbosity()` and `getVerbosity()`, respectively. Old names will be removed in v1.0.
- ProDy applications (commands) module is rewritten to use new `argparse`[1076] module. See *ProDy Applications* (page 3) for details of changes.

[1076] http://docs.python.org/library/argparse.html#module-argparse

- argparse[1077] module is added to the package for Python versions 2.6 and older.

Bugfixes:

- Fixed problems in *loadAtoms()* (page 75) and *saveAtoms()* (page 75) functions.
- Bugfixes in *parseDCD()* (page 292) and *writeDCD()* (page 293) functions for Windows compatability.

5.14.4 0.9.1 (Nov 9, 2011)

Bug Fixes:

- Fixed problems with reading and writing configuration files.
- Fixed problem with importing nose for testing.

5.14.5 0.9 (Nov 8, 2011)

New Features:

- PDBML[1078] and mmCIF[1079] files can be retrieved using *fetchPDB()* (page 266) function.
- `getPDBLocalFolder()` and `setPDBLocalFolder()` functions are implemented for local PDB folder management.
- *parsePDBHeader()* (page 240) is implemented for convenient parsing of header data from `.pdb` files.
- *showProtein()* (page 236) is implemented to allow taking a quick look at protein structure.
- *Chemical* (page 236) and *Polymer* (page 238) classes are implemented for storing chemical and polymer component data parsed from PDB header records.

Changes:

> **Warning:** This release introduces numerous changes in method and function names all aiming to improve the interactive usage experience. All changes are listed below. Currently these functions and methods are present in both old and new names, so code using ProDy must not be affected. Old function names will be removed from version 1.0, which is expected to happen late in the first quarter of 2012.
> Old function names are marked as deprecated, but ProDy will not issue any warnings until the end of 2011. In 2012, ProDy will automatically start issuing `DeprecationWarning` upon calls using old names to remind the user of the name change.
> For deprecated methods that are present in multiple classes, only the affected modules are listed for brevity.
>
> http://docs.python.org/library/exceptions.html#DeprecationWarning

> **Note:** When modifying code using ProDy to adjust the name changes, turning on deprecation warnings may help locating all use cases of the deprecated names. See `turnonDeprecationWarnings()` for this purpose.

Functions:

[1077] http://docs.python.org/library/argparse.html#module-argparse
[1078] http://pdbml.pdb.org/
[1079] http://mmcif.pdb.org/

The following function name changes are mainly to reduce the length of the name in order to make them more suitable for interactive sessions:

Old name	New name
applyBiomolecularTransformations()	*buildBiomolecules()* (page 242)
assignSecondaryStructure()	*assignSecstr()* (page 242)
scanPerturbationResponse()	*calcPerturbResponse()* (page 178)
calcCrossCorrelations()	*calcCrossCorr()* (page 137)
calcCumulativeOverlap()	*calcCumulOverlap()* (page 142)
calcCovarianceOverlap()	*calcCovOverlap()* (page 143)
showFractOfVariances()	*showFractVars()* (page 180)
showCumFractOfVariances()	*showCumulFractVars()* (page 180)
showCrossCorrelations()	*showCrossCorr()* (page 180)
showCumulativeOverlap()	*showCumulOverlap()* (page 180)
deform()	*deformAtoms()* (page 189)
calcSumOfWeights()	*calcOccupancies()* (page 205)
showSumOfWeights()	*showOccupancies()* (page 206)
trimEnsemble()	*trimPDBEnsemble()* (page 205)
getKeywordResidueNames()	getKeywordResnames()
setKeywordResidueNames()	setKeywordResnames()
getPairwiseAlignmentMethod()	*getAlignmentMethod()* (page 232)
setPairwiseAlignmentMethod()	*setAlignmentMethod()* (page 232)
getPairwiseMatchScore()	*getMatchScore()* (page 231)
setPairwiseMatchScore()	*setMatchScore()* (page 231)
getPairwiseMismatchScore()	*getMismatchScore()* (page 231)
setPairwiseMismatchScore()	*setMismatchScore()* (page 231)
getPairwiseGapOpeningPenalty()	*getGapPenalty()* (page 231)
setPairwiseGapOpeningPenalty()	*setGapPenalty()* (page 231)
getPairwiseGapExtensionPenalty()	*getGapExtPenalty()* (page 231)
setPairwiseGapExtensionPenalty()	*setGapExtPenalty()* (page 231)

Coordinate methods:

All `getCoordinates()` and `setCoordinates()` methods in *atomic* (page 29) and *ensemble* (page 200) classes are renamed as `getCoords()` and `setCoords()`, respectively.

`getNumOf` **methods**:

All method names starting with `getNumOf` now start with `num`. This change brings two advantages: method names (i) are considerably shorter, and (ii) do not suggest that there might also be corresponding `set` methods.

Old name	New name	Affected modules
getNumOfAtoms()	numAtoms()	*atomic* (page 29), *ensemble* (page 200), *dynamics* (page 131)
getNumOfChains()	numChains()	*atomic* (page 29)
getNumOfConfs()	numConfs()	*ensemble* (page 200)
getNumOfCoordsets()	numCoordsets()	*atomic* (page 29), *ensemble* (page 200)
getNumOfDegOfFreedom()	numDOF()	*dynamics* (page 131)
getNumOfFixed()	numFixed()	*ensemble* (page 200)
getNumOfFrames()	numFrames()	*ensemble* (page 200)
getNumOfResidues()	numResidues()	*atomic* (page 29)
getNumOfMapped()	numMapped()	*atomic* (page 29)
getNumOfModes()	numModes()	*dynamics* (page 131)
getNumOfSelected()	numSelected()	*ensemble* (page 200)
getNumOfUnmapped()	numUnmapped()	*atomic* (page 29)

`getName` **method**:

`getName()` methods are renamed as `getTitle()` to avoid confusions that might arise from changes in *atomic* (page 29) method names listed below. All classes in *atomic* (page 29), *ensemble* (page 200), and *dynamics* (page 131) are affected from this change.

In line with this change, *parsePDB()* (page 268) and *parsePQR()* (page 269) *name* arguments are changed to *title*, but *name* argument will also work until release 1.0.

This name change conflicted with *DCDFile.getTitle()* (page 291) method. The conflict is resolved in favor of the general `getTitle()` method. An alternative method will be implemented to handle title strings in `DCD` files.

`get/set` **methods of atomic classes**:

Names of `get` and `set` methods allowing access to atomic data are all shortened as follows:

Old name	New name
`getAtomNames()`	`getNames()`
`getAtomTypes()`	`getTypes()`
`getAltLocIndicators()`	`getAltlocs()`
`getAnisoTempFactors()`	`getAnisos()`
`getAnisoStdDevs()`	`getAnistds()`
`getChainIdentifiers()`	`getChains()`
`getElementSymbols()`	`getElements()`
`getHeteroFlags()`	`getHeteros()`
`getInsertionCodes()`	`getIcodes()`
`getResidueNames()`	`getResnames()`
`getResidueNumbers()`	`getResnums()`
`getSecondaryStrs()`	`getSecstrs()`
`getSegmentNames()`	`getSegnames()`
`getSerialNumbers()`	`getSerials()`
`getTempFactors()`	`getBetas()`

This change affects all *atomic* (page 29) classes, *AtomGroup* (page 38), *Atom* (page 32), *Chain* (page 53), *Residue* (page 80), *Selection* (page 100) and *AtomMap* (page 49).

Other changes in atomic methods:

- `getSelectionString()` renamed as `getSelstr()`

Methods handling user data (which was previously called attribute) are renamed as follows:

Old name	New name
`getAttribute()`	`getData()`
`getAttrNames()`	`getDataLabels()`
`getAttrType()`	`getDataType()`
`delAttribute()`	`delData()`
`isAttribute()`	`isData()`
`setAttribute()`	`setData()`

To be removed:

Finally, the following methods will be removed, but other suitable methods are overloaded to perform their action:

- removed `AtomGroup.getBySerialRange()`, overloaded *AtomGroup.getBySerial()* (page 39)
- removed `getProteinResidueNames()`, overloaded `getKeywordResnames()`

- removed `setProteinResidueNames()`, overloaded `setKeywordResnames()`

Scripts:

The way ProDy scripts work has changed. See *ProDy Applications* (page 3) for details. Using older scripts will start issuing deprecation warnings in 2012.

Bug Fixes:

- Bugs in *`execDSSP()`* (page 232) and *`execSTRIDE()`* (page 273) functions that caused exceptions when compressed files were passed is fixed.
- A problem in scripts for PCA of DCD files is fixed.

Normal Mode Wizard

Development of NMWiz is finalized and it will not be distributed in the ProDy installation package anymore. See Normal Mode Wizard[1081] pages for instructions on installing it.

5.15 ProDy 0.8 Series

[1081] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

5.15.1 0.8.3 (Oct 16, 2011)

New Features:

- Functions to read and write PQR files: *`parsePQR()`* (page 269) and *`writePQR()`* (page 271).
- Added `PDBEnsemble.getIdentifiers()` method that returns identifiers of all conformations in the ensemble.
- ProDy tests are incorporated to the package installer. If you are using Python version 2.7, you can run the tests by calling `prody.test()`.

Improvements:

- *`blastPDB()`* (page 226) function and *`PDBBlastRecord`* (page 225) class are rewritten to use faster and more compact code.
- New *`PackageLogger`* (page 308) function is implemented to unify logging and reporting task progression.
- Improvements in PDB ensemble support functions, e.g. *`trimPDBEnsemble()`* (page 205), are made.
- Improvements in ensemble concatenations are made.

Bug Fixes:

- Bugfixes in `PDBEnsemble()` slicing operation. This may have affected users when slicing a PDB ensemble for plotting projections in color for different forms of the protein.

[1081] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

5.15.2 0.8.2 (Oct 14, 2011)

New Features:

- *fetchPDBClusters()* (page 267), *loadPDBClusters()* (page 267), and `getPDBCluster()` functions are implemented for handling PDB sequence cluster data. These functions can be used instead of *blastPDB()* (page 226) function for fast access to structures of the same protein (at 95% sequence identity level) or similar proteins.
- Perturbation response scanning method described in *[CA09]* (page 397) is implemented as `scanPerturbationResponse()` based on the code provided by Ying Liu.

Changes:

- *fetchPDBLigand()* (page 119) returns the URL of the XML file in the ligand data dictionary.
- Name of the ProDy configuration file in user `home` directory is renamed as `.prodyrc` (used to be `.prody`).
- `applyBiomolecularTransformations()` and `assignSecondaryStructure()` functions raise `ValueError`[1083] when the function fails to perform its action due to missing data in header dictionary.
- *fetchPDB()* (page 266) decompresses PDB files found in the working directory when user asks for decompressed files.
- *parsePDB()* (page 268) appends *chain* and *subset* arguments to `AtomGroup()` name.
- *chain* argument is added to *PDBBlastRecord.getHits()* (page 226).

Improvements:

- Atom selection class *Select* (page 99) is completely redesigned to prevent breaking of the parser when evaluating invalid selection strings.
- Improved type checking in *parsePDB()* (page 268) function.

Bug Fixes:

- Bugfixes in *parseDSSP()* (page 232): one emerged problems in lines indicating chain breaks, another did not parse bridge-partners correctly. Both fixes are contributed by Kian Ho.
- Bugfix in *parsePDB()* (page 268) function. When only header is desired (`header=True`, `model=0`), would return a tuple containing an empty atom group and the header.

Developmental:

- Unit tests for *proteins* (page 222) and `select` modules are developed.

5.15.3 0.8.1 (Sep 16, 2011)

New Features:

- `fetchLigandData()` is implemented for fetching ligand data from Ligand Expo.
- *parsePSF()* (page 294) function is implemented for parsing X-PLOR format PSF files.

Changes:

- __slots__ is used in *AtomGroup* (page 38) and *Atomic* (page 47) classes. This change prevents user from assigning new variables to instances of all classes derived from the base *Atomic* (page 47).
- `pyparsing` is updated to version 1.5.6.

[1083] http://docs.python.org/library/exceptions.html#ValueError

Bug Fixes:

- A bug in *AtomGroup.copy()* (page 39) method is fixed. When AtomGroup instance itself is copied, deep copies of data arrays were not made.
- A bug in *Select* (page 99) class raising exceptions when negative residue number values are present is fixed.
- Another bug in *Select* (page 99) class misinterpreting `same residue as ...` statement when specific chains are involved is fixed.
- A bug in *AtomGroup.addCoordset()* (page 39) method duplicating coordinates when no coordinate sets are present in the instance is fixed.

Normal Mode Wizard

Changes:

- Version number in main window is iterated.
- Mode graphics material is stored for individual modes.
- Mode scaling factor is printed when active mode or RMSD is changed.
- All selections are deleted to avoid memory leaks.

5.15.4 0.8 (Aug 24, 2011)

Note: After installing v0.8, you may need to make a small change in your existing scripts. If you are using *Ensemble* (page 202) class for analyzing PDB structures, rename it as *PDBEnsemble* (page 207). See the other changes that may affect your work below and the class documentation for more information.

New Features:

- *DCDFile* (page 290) is implemented for handling DCD files.
- *Trajectory* (page 296) is implemented for handling multiple trajectory files.
- *writeDCD()* (page 293) is implemented for writing DCD files.
- Trajectory Analysis[1084] example to illustrate usage of new classes for handling DCD files. Essential Dynamics Analysis[1085] example is updated to use new ProDy classes.
- *PCA* (page 175) supports *Trajectory* (page 296) and *DCDFile* (page 290) instances.
- *Ensemble* (page 202) and *PDBEnsemble* (page 207) classes can be associated with *AtomGroup* (page 38) instances. This allows selecting and evaluating coordinates of subset of atoms. See `setAtomGroup()`, `select()`, `getAtomGroup()`, and `getSelection()` methods.
- *execDSSP()* (page 232), *parseDSSP()* (page 232), and *performDSSP()* (page 233) functions are implemented for executing and parsing DSSP calculations.
- *execSTRIDE()* (page 273), *parseSTRIDE()* (page 273), and *performSTRIDE()* (page 273) functions are implemented for executing and parsing DSSP calculations.
- *parsePDB()* (page 268) function parses atom serial numbers. Atoms can be retrieved from an *AtomGroup* (page 38) instance by their serial numbers using *getBySerial()* (page 39) and `getBySerialRange()` methods.

[1084] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory.html#trajectory
[1085] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda

- *calcADPs()* (page 217) function can be used to calculate anisotropic displacement parameters for atoms with anisotropic temperature factor data.
- *getRMSFs()* (page 203) is implemented for calculating root mean square fluctuations.
- *AtomGroup* (page 38) and *Mode* (page 169) or *Vector* (page 170) additions are supported. This adds a new coordinate set to the *AtomGroup* (page 38) instance.
- `getAttrNames()` is implemented for listing user set attribute names.

Improvements:

- *calcProjection()* (page 138), *showProjection()* (page 181), and *showCrossProjection()* (page 182) functions can optionally calculate/display RMSD along the normal mode.
- ANM, GNM, and PCA applications can optionally write compressed ProDy data files.
- *fetchPDB()* (page 266) function can optionally write decompressed files and force copying a file from local mirror to target folder.
- *PCA.buildCovariance()* (page 175) and *PCA.performSVD()* (page 176) methods accept Numpy arrays as coordinate sets.
- Performance of *PCA.buildCovariance()* (page 175) method is optimized for evaluation of PDB ensembles.
- *calcRMSD()* (page 219) and *superpose()* (page 220) functions are optimized for speed and memory usage.
- *Ensemble.getMSFs()* (page 203) is optimized for speed and memory usage.
- Improvements in memory operations in *atomic* (page 29), *ensemble* (page 200), and *dynamics* (page 131) modules for faster data (PDB/NMD) output.
- Optimizations in *Select* (page 99) and *Contacts* (page 213) classes.

Changes:

- *Ensemble* (page 202) does not store conformation names. Instead, newly implemented *PDBEnsemble* (page 207) class stores identifiers for individual conformations (PDB IDs). This class should be used in cases where source of individual conformations is important.
- *calcProjection()* (page 138), *showProjection()* (page 181), and *showCrossProjection()* (page 182) function calculate/display root mean square deviations, by default.
- Oxidized cysteine residue abbreviation `CSO` is added to the definition of `protein` keyword.
- `getMSF()` method is renamed as *getMSFs()* (page 203).
- *parseDCD()* (page 292) function returns *Ensemble* (page 202) instances.

Bug Fixes:

- A bug in `select` module causing exceptions when regular expressions are used is fixed.
- Another bug in `select` module raising exception when "(not ..," is passed is fixed.
- Various bugfixes in *ensemble* (page 200) module.
- Problem in **`prody fetch`** that occurred when a file is found in a local mirror is fixed.
- Bugfix in *AtomPointer.copy()* (page 78) method.

Normal Mode Wizard[1086]

New Features:

- NMWiz can be used to compare two structures by calculating and depicting structural changes.
- Arrow graphics is scaled based on a user specified RMSD value.

Improvements:

- NMWiz writes DCD format trajectories for PCA using ProDy. This provides significant speed up in cases where IO rate is the bottleneck.

Changes:

- Help is provided in a text window to provide a cleaner GUI.

5.16 ProDy 0.7 Series

5.16.1 0.7.2 (Jun 21, 2011)

New Features:

- `parseDCD()` (page 292) is implemented for parsing coordinate sets from DCD files.

Improvements:

- `parsePDB()` (page 268) parses `SEQRES` records in header sections.

Changes:

- Major classes can be instantiated without passing a name argument.
- Default selection in NMWiz ProDy interface is changed to ensure selection only protein Cα atoms.

Bug Fixes:

- A bug in `writeNMD()` (page 174) function causing problems when writing a single mode is fixeed.
- Other bugfixes in `dynamics` (page 131) module functions.

5.16.2 0.7.1 (Apr 28, 2011)

Highlights:

- `Atomic` (page 47) `__getattribute__()` is overloaded to interpret atomic selections following the dot operator. For example, `atoms.calpha` is interpreted as `atoms.select('calpha')`. See :ref:'' for more details.
- `AtomGroup` (page 38) class is integrated with `HierView` (page 76) class. Atom group instances now can be indexed to get chains or residues and number of chains/residues can be retrieved. A hierarchical view is generated and updated when needed. See :ref:'' for more details.

New Features:

[1086] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

- *matchAlign()* (page 229) is implemented for quick alignment of protein structures. See Ligand Extraction[1087] usage example.
- setAttribute(), getAttribute(), delAttribute(), and isAttribute() functions are implemented for *AtomGroup* (page 38) class to facilitate storing user provided atomic data. See Storing data in AtomGroup[1088] example.
- *saveAtoms()* (page 75) and *loadAtoms()* (page 75) functions are implemented to allow for saving atomic data and loading it This saves custom atomic attributes and much faster than parsing data from PDB files.
- *calcCollectivity()* (page 137) function is implemented to allow for calculating collectivity of deformation vectors.

Improvements:

- *parsePDB()* (page 268) can optionally return biomolecule when biomol=True keyword argument is passed.
- *parsePDB()* (page 268) can optionally make secondary structure assignments when secondary=True keyword argument is passed.
- *calcSqFlucts()* (page 137) function is changed to accept *Vector* (page 170) instances, e.g. deformation vectors.

Changes:

- Changes were made in *calcADPAxes()* (page 215) function to follow the conventions in analysis ADPs. See its documentation.

Bug Fixes:

- A in *Ensemble* (page 202) slicing operations is fixed. Weights are now copied to the new instances obtained by slicing.
- Bug fixes in *dynamics* (page 131) plotting functions *showScaledSqFlucts()* (page 183), *showNormedSqFlucts()* (page 183),

5.16.3 0.7 (Apr 4, 2011)

New Features:

- Regular expressions can be used in atom selections. See select module for details.
- User can define selection macros using defSelectionMacro() function. Macros are saved in ProDy configuration and loaded in later sessions. See select module for other related functions.
- *parseSparseMatrix()* (page 156) function is implemented for parsing matrices in sparse format. See the usage example in Using an External Matrix[1089].
- deform() function is implemented for deforming coordinate sets along a normal mode or linear combination of multiple modes.
- *sliceModel()* (page 145) function is implemented for slicing normal mode data to be used with functions calculating atomic properties using normal modes.

Improvements:

- Atom selections using bare keyword arguments is optimized. New keyword definitions are added. See select module for the complete list.

[1087] http://prody.csb.pitt.edu/tutorials/structure_analysis/ligands.html#extract-ligands
[1088] http://prody.csb.pitt.edu/tutorials/prody_tutorial/atomgroup.html#id1
[1089] http://prody.csb.pitt.edu/tutorials/enm_analysis/external.html#external-matrix

- A new keyword argument for *calcADPAxes()* (page 215) allows for comparing largest axis to the second largest one.

Changes:

- There are changes in function used to alter definitions of selection keywords. See `select` for details.
- `assignSecondaryStructure()` function assigns SS identifiers to all atoms in a residue. Residues with no SS information specified is assigned coil conformation.
- When *Ensemble* (page 202) and *NMA* (page 172) classes are instantiated with an empty string, instances are called "Unnamed".
- *sliceMode()* (page 144), *sliceVector()* (page 145) and *reduceModel()* (page 145) functions return the atom selection in addition to the sliced vector/mode/model instance.

Bug Fixes:

- Default selection for *calcGNM()* (page 166) function is set to "calpha".

Normal Mode Wizard

New Features:

- NMWiz supports GNM data and can use ProDy for GNM calculations.
- NMWiz can gather normal mode data from molecules loaded into VMD. This allows NMWiz to support all formats supported by VMD.
- User can write data loaded into NMWiz in NMD format.
- An Arrow Graphics option allows the user to draw arrows in both directions.
- User can select Licorice representation for the protein if model is an all atom mode.
- User can select Custom as the representation of the protein to prevent NMWiz from chancing a user set representation.
- Trace is added as a protein backbone representation option.

Improvements:

- NMWiz remembers all adjustments on arrow graphics for all modes.
- Plotting *Clear* button clears only atom labels that are associated with the dataset.
- Removing a dataset removes all associated molecule objects.
- Selected atom representations are turned on based on atom index.
- Padding around interface button has been standardized to provide a uniform experience between different platforms.

5.17 ProDy 0.6 Series

5.17.1 0.6.2 (Mar 16, 2011)

New Features:

- *performSVD()* (page 176) function is implemented for faster and more memory efficient principal compoment analysis.
- `extrapolateModel()` function is implemented for extrapolating a coarse-grained model to an all atom model. See the usage example Extend a coarse-grained model[1090].
- `plog()` is implemented for enabling users to make log entries.

Improvements:

- `compare` functions are improved to handle insertion codes.
- *HierView* (page 76) allows for indexing using chain identifier and residue numbers. See usage example Hierarchical Views[1091].
- *Chain* (page 53) allows for indexing using residue number and insertion code. See usage example Hierarchical Views[1092].
- *addCoordset()* (page 39) function accepts *Atomic* (page 47) and *Ensemble* (page 202) instances as *coords* argument.
- New method *HierView.getAtoms()* (page 76) is implemented.
- *AtomGroup* (page 38) set functions check the correctness of dimension of data arrays to prevent runtime problems.
- **prody pca** script is updated to use the faster PCA method that uses SVD.

Changes:

- "backbone" definition now includes the backbone hydrogen atom (Thanks to Nahren Mascarenhas for pointing to this discrepancy in the keyword definition).

Bug Fixes:

- A bug in *PCA* (page 175) allowed calculating covariance matrix for less than 3 coordinate sets is fixed.
- A bug in *mapOntoChain()* (page 230) function that caused problems when mapping all atoms is fixed.

5.17.2 0.6.1 (Mar 2, 2011)

New Features:

- `setWWPDBFTPServer()` and `getWWPDBFTPServer()` functions allow user to change or learn the WWPDB FTP server that ProDy uses to download PDB files. Default server is RCSB PDB in USA. User can change the default server to one in Europe or Japan.
- `setPDBMirrorPath()` and `getPDBMirrorPath()` functions allow user to specify or learn the path to a local PDB mirror. When specified, a local PDB mirror is preferred for accessing PDB files, over downloading them from FTP servers.
- *mapOntoChain()* (page 230) function is improved to map backbone or all atoms.

Improvements:

- `WWPDB_PDBFetcher` can download PDB files from different WWPDB FTP servers.

[1090] http://prody.csb.pitt.edu/tutorials/enm_analysis/extend.html#extendmodel
[1091] http://prody.csb.pitt.edu/tutorials/prody_tutorial/hierview.html#hierview
[1092] http://prody.csb.pitt.edu/tutorials/prody_tutorial/hierview.html#hierview

- `WWPDB_PDBFetcher` can also use local PDB mirrors for accessing PDB files.

Changes:

- `RCSB_PDBFetcher` is renamed as `WWPDB_PDBFetcher`.
- *mapOntoChain()* (page 230) and *matchChains()* (page 228) functions accept `"ca"` and `"bb"` as *subset* arguments.
- Definition of selection keyword "protein" is updated to include some non-standard amino acid abbreviations.

Bug Fixes:

- A bug in `WWPDB_PDBFetcher` causing exceptions when non-string items passed in a list is fixed.
- An important bug in *parsePDB()* (page 268) is fixed. When parsing backbone or Cα atoms, residue names were not checked and this caused parsing water atoms with name `"O"` or calcium ions with name `"CA"`.

5.17.3 0.6 (Feb 22, 2011)

New Features:

- Biopython module pairwise2 and packages KDTree and Blast are incorporated in ProDy package to make installation easier. Only NumPy needs to be installed before ProDy can be used. For plotting, Matplotlib is still required.
- Normal Mode Wizard[1093] is distributed with ProDy source. On Linux, if VMD is installed, ProDy installer locates VMD plugins folder and installs NMWiz. On Windows, user needs to follow a separate set of instructions (see Normal Mode Wizard[1094]).
- *Gamma* (page 159) class is implemented for facilitating use of force constants based on atom type, residue type, or property. An example derived classes are *GammaStructureBased* (page 159) and *GammaVariableCutoff* (page 161).
- *calcTempFactors()* (page 137) function is implemented to calculate theoretical temperature factors.
- 5 new *ProDy Applications* (page 3) are implemented, and existing scripts are improved to output figures.
- *getModel()* (page 172) method is implemented to make function development easier.
- *resetTicks()* (page 183) function is implemented to change X and/or Y axis ticks in plots when there are discontinuities in the plotted data.

Improvements:

- *ANM.buildHessian()* (page 140) and *GNM.buildKirchhoff()* (page 164) classes are improved to accept *Gamma* (page 159) instances or other custom function as *gamma* argument. See also Custom Gamma Functions[1095].
- *Select* (page 99) class is changed to treat single word keywords differently, e.g. "backbone" or "protein". They are interpreted 10 times faster and in use achieve much higher speed-ups when compared to composite selections. For example, using the keyword "calpha" instead of the `name CA and protein`, which returns the same selection, works >20 times faster.
- Optimizations in `Select` class to increase performance (Thanks to Paul McGuire for providing several Pythonic tips and Pyparsing specific advice).

[1093] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz
[1094] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz
[1095] http://prody.csb.pitt.edu/tutorials/enm_analysis/gamma.html#gamma

- `applyBiomolecularTransformations()` function is improved to handle large biomolecular assemblies.
- Performance optimizations in *parsePDB()* (page 268) and other functions.
- *Ensemble* (page 202) class accepts *Atomic* (page 47) instances and automatically adds coordinate sets to the ensemble.

Changes:

- `PDBlastRecord` is renamed as *PDBBlastRecord* (page 225).
- *NMA* (page 172) instances can be index using a list or tuple of integers, e.g. `anm[1,3,5]`.
- "ca", "bb", and "sc" keywords are defined as short-hands for "calpha", "backbone", and "sidechain", respectively.
- Behavior of *calcANM()* (page 142) and *calcGNM()* (page 166) functions have changed. They return the atoms used for calculation as well.

Bug Fixes:

- A bug in `assignSecondaryStructure()` function is fixed.
- Bug fixes in *prody anm* (page 4) and *prody gnm* (page 11).
- Bug fixes in *showSqFlucts()* (page 183) and *showProjection()* (page 181) functions.

Normal Mode Wizard

- NMWiz can be used as a graphical interface to ProDy. ANM or PCA calculations can be performed for molecules that are loaded in VMD.
- User can set default color for arrow graphics and paths to ANM and PCA scripts.
- Optionally, NMWiz can preserve the current view in VMD display window when loading a new dataset. Check the box in the NMWiz GUI main window.
- A bug that prevented selecting residues from plot window is fixed.

5.18 ProDy 0.5 Series

5.18.1 0.5.3 (Feb 11, 2011)

New Features:

- Membership, equality, and non-equality test operation are defined for all *atomic* (page 29) classes. See Operations on Selections[1096].
- Two functions are implemented for dealing with anisotropic temperature factors: *calcADPAxes()* (page 215) and *buildADPMatrix()* (page 215).
- *NMA.setEigens()* (page 173) and *NMA.addEigenpair()* (page 172) methods are implemented to assist analysis of normal modes calculated using external software.

[1096] http://prody.csb.pitt.edu/tutorials/prody_tutorial/selection.html#selection-operations

- *parseNMD()* (page 174) is implemented for parsing NMD files.
- *parseModes()* (page 155) is implemented for parsing normal mode data.
- *parseArray()* (page 155) is implementing for reading numeric data, particularly normal mode data calculated using other software for analysis using ProDy.
- The method in *[BH02]* (page 395) to calculate overlap between covariance matrices is implemented as *calcCovOverlap()* (page 143) function.
- `trimEnsemble()` to trim *Ensemble* (page 202) instances is implemented.
- *checkUpdates()* (page 329) to check for ProDy updates is implemented.

Changes:

- Change in default behavior of *parsePDB()* (page 268) function. When alternate locations exist, those indicated by A are parsed. For parsing all alternate locations user needs to pass `altloc=True` argument.
- `getSumOfWeights()` is renamed as `calcSumOfWeights()`.
- `mapAtomsToChain()` is renamed as *mapOntoChain()* (page 230).
- `ProDyStartLogFile()` is renamed as *startLogfile()* (page 329).
- `ProDyCloseLogFile()` is renamed as *closeLogfile()* (page 330).
- `ProDySetVerbosity()` is renamed as `changeVerbosity()`.

Improvements:

- A few bugs in ensemble and dynamics classes are fixed.
- Improvements in `RCSB_PDBFetcher` allow it not to miss a PDB file if it exists in the target folder.
- *writeNMD()* (page 174) is fixed to output B-factors (Thanks to Dan Holloway for pointing it out).

5.18.2 0.5.2 (Jan 12, 2011)

Bug Fixes:

- An important fix in *sampleModes()* (page 189) function was made (Thanks to Alberto Perez for finding the bug and suggesting a solution).

Improvements:

- Improvements in *ANM.calcModes()* (page 141), *GNM.calcModes()* (page 165), and *PCA.calcModes()* (page 175) methods prevent Numpy/Scipy throwing an exception when more than available modes are requested by the user.
- Improvements in *blastPDB()* (page 226) enable ProDy throw an exception when no internet connection is found, and warn user when downloads fail due to restriction in network regulations (Thanks to Serkan Apaydin for helping identify these improvements).
- New example Write PDB file[1097].

5.18.3 0.5.1 (Dec 31, 2010)

Changes in dependencies:

- Scipy (linear algebra module) is not required package anymore. When available it replaces Numpy (linear algebra module) for greater flexibility and efficiency. A warning message is printed when Scipy is not found.

[1097] http://prody.csb.pitt.edu/tutorials/structure_analysis/pdbfiles.html#writepdb

- Biopython KDTree module is not required for ENM calculations (specifically for building Hessian (ANM) or Kirchoff (GNM) matrices). When available it is used to increase the performance. A warning message is printed when KDTree is not found.

5.18.4 0.5 (Dec 21, 2010)

New Features:

- *AtomPointer* (page 78) base class for classes pointing to atoms in an *AtomGroup* (page 38).
- *AtomPointer* (page 78) instances (Selection, Residue, etc.) can be added. See Operations on Selections[1098] for examples.
- *Select.getIndices()* (page 99) and *Select.getBoolArray()* (page 99) methods to expand the usage of *Select* (page 99).
- *sliceVector()* (page 145) and *sliceMode()* (page 144) functions.
- *saveModel()* (page 158) and *loadModel()* (page 158) functions for saving and loading NMA data.
- *parsePDBStream()* (page 267) can now parse specific chains or alternate locations from a PDB file.
- *alignCoordsets()* (page 219) is implemented to superimpose coordinate sets of an *AtomGroup* (page 38) instance.

Bug Fixes:

- A bug in *parsePDBStream()* (page 267) that caused unidentified errors when a model in a multiple model file did not have the same number of atoms is fixed.

Changes:

- Iterating over a *Chain* (page 53) instance yields *Residue* (page 80) instances.
- *Vector* (page 170) instantiation requires an *array* only. *name* is an optional argument.
- Functions starting with `get` and performing a calculations are renamed to start with `calc`, e.g. `getRMSD()` is now *calcRMSD()* (page 219).

5.19 ProDy 0.2 Series

5.19.1 0.2 (Nov 16, 2010)

Important Changes:

- Single word keywords *not* followed by "and" logical operator are not accepted, e.g. "protein within 5 of water" will raise a *SelectionError* (page 99), use "protein and within 5 of water" instead.
- `findMatchingChains()` is renamed to *matchChains()* (page 228).
- `showOverlapMatrix()` is renamed to *showOverlapTable()* (page 181).
- Modules are reorganized.

New Features:

[1098] http://prody.csb.pitt.edu/tutorials/prody_tutorial/selection.html#selection-operations

- *Atomic* (page 47) for easy type checking.
- *Contacts* (page 213) for faster intermolecular contact identification.
- *Select* (page 99) can identify intermolecular contacts. See Intermolecular Contacts[1099] for an examples and details.
- *sampleModes()* (page 189) implemented for sampling conformations along normal modes.

Improvements:

- *proteins.compare* (page 228) functions are improved. Now they perform sequence alignment if simple residue number/identity based matchin does not work, or if user passes `pwalign=True` argument. This impacts the speed of X-ray ensemble analysis.
- *Select* (page 99) can cache data optionally. This results in speeds up from 2 to 50 folds depending on number of atoms and selection operations.
- Implementation of *showProjection()* (page 181) is completed.

Normal Mode Wizard

Release 0.2.3

- For each mode a molecule for drawing arrows and a molecule for showing animation is formed in VMD on demand. NMWiz remembers a color associated with a mode.
- Deselecting a residue by clicking on a plot is possible.
- A bug causing incorrect parsing of NMD files from ANM server is fixed.

Release 0.2.2

- Selection string option allows user to show a subset of arrows matching a VMD selection string. Optionally, this selection string may affect protein and animation representations.
- A bug that caused problems when over plotting modes is removed.
- A bug affecting line width changes in plots is removed.
- Selected residue representations are colored according to the color of the plot.

Release 0.2.1

- Usability improvements.
- Loading the same data file more than once is prevented.
- If a GUI window for a dataset is closed, it can be reloaded from the main window.
- A dataset and GUI can be deleted from the VMD session via the main window.

Release 0.2

- Instant documentation is improved.
- Problem with clearing selections is fixed.
- Plotting options frame is populated.
- Multiple modes can be plotted on the same canvas.

[1099] http://prody.csb.pitt.edu/tutorials/structure_analysis/contacts.html#contacts

5.20 ProDy 0.1 Series

5.20.1 0.1.2 (Nov 9, 2010)

- Important bug fixes and improvements in NMA helper and plotting functions.
- Documentation updates and improvements.

5.20.2 0.1.1 (Nov 8, 2010)

- Important bug fixes and improvements in chain comparison functions.
- Bug fixes.
- Source clean up.
- Documentation improvements.

5.20.3 0.1 (Nov 7, 2010)

- First release.

CHAPTER

SIX

ABOUT PRODY

ProDy is a free and open-source Python package for protein structural dynamics and sequence evolution analysis. It is designed as a flexible and responsive API suitable for interactive usage and application development.

6.1 People

ProDy is being developed in the Bahar Lab[1100] at the University of Pittsburgh[1101] with support from NIH R01 GM099738 award.

6.1.1 Development Team

Ahmet Bakan[1102] initiated the *ProDy* project, designed and developed *ProDy*, *NMWiz*, *Evol*, and *DruGUI*.

Cihan Kaya[1103] was overseeing the overall development of *ProDy* from 2015 to 2018.

She (John) Zhang[1104] was helping with maintaining and developing *ProDy* from 2016, and took on the overseeing role from 2018 to mid 2020.

James Krieger[1105] was helping develop *ProDy* from 2017 and became the main overseer and developer in mid 2020.

Hongchun Li[1106] is currently maintaining and developing ANM and GNM servers, and made significant contributions to `database` (page 120) and `dynamics` (page 131) including *SignDy* and Adaptive ANM.

Yan Zhang[1107] contributed significantly to the development of the *cryo-EM* module, `protein.emdmap`.

Burak Kaynak[1108] contributed significantly to the development of `domain_decomposition` (page 131), `dynamics.essa` (page 147), and `dynamics.clustenm`.

Karolina Mikulska-Ruminska[1109] contributed significantly to the development of `protein.interactions` (*InSty*), `protein.waterbridges` (*WatFinder*), and `dynamics.mechstiff` (page 169) (*MechStiff*).

Anthony Bogetti[1110] is overseeing the overall development of *ProDy* since 2024.

[1100] http://www.bahargroup.org/Faculty/bahar/
[1101] http://www.pitt.edu/
[1102] https://scholar.google.com/citations?user=-QAYVgMAAAAJ&hl=en
[1103] https://www.linkedin.com/in/cihan-kaya/
[1104] https://www.linkedin.com/in/she-zhang-49164399/
[1105] https://scholar.google.pl/citations?user=DoiCjkUAAAAJ&hl=pl
[1106] http://www.pitt.edu/~hongchun/
[1107] https://scholar.google.pl/citations?user=VxwU0pgAAAAJ&hl=pl&oi=sra
[1108] https://scholar.google.pl/citations?user=gP8RokwAAAAJ&hl=pl&oi=ao
[1109] https://scholar.google.pl/citations?user=IpyPHRwAAAAJ&hl=pl
[1110] https://scholar.google.pl/citations?hl=pl&user=9qQClIcAAAAJ

Anindita Dutta[1111] contributed to the development of *Evol*, `database` (page 120) and `sequence` (page 278) modules.

Tim Lezon[1112] contributed to development of Rotations and Translation of Blocks and Membrane ENM.

Wenzhi Mao[1113] contributed to development of MSA analysis functions.

Lidio Meireles[1114] provided insightful comments on the design of *ProDy*, and contributed to the development of *ProDy Applications* (page 3).

6.1.2 Contributors

In addition to the development team members, we acknowledge contributions and feedback from the following individuals:

Ying Liu[1115] provided the code for Perturbation Response Scanning method.

Frane Doljanin[1116] provided the code for the water bridge detection.

Kian Ho[1117] contributed with bug fixes and unit tests for DSSP functions.

Gökçen Eraslan[1118] contributed with bug fixes and development and maintenance insights.

6.2 Citing

When using *ProDy* or *NMWiz* in published work, please cite:

> Bakan A, Meireles LM, Bahar I.
> *ProDy*: Protein Dynamics Inferred from Theory and Experiments.
> *Bioinformatics* **2011** 27(11):1575-1577.

> Bakan A, Dutta A, Mao W, Liu Y, Chennubhotla C, Lezon TR, Bahar I.
> *Evol* and *ProDy* for Bridging Protein Sequence Evolution and Structural Dynamics.
> *Bioinformatics* **2014** 30(18):2681-2683.

> Zhang S, Krieger JM, Zhang Y, Kaya C, Kaynak B, Mikulska-Ruminska K, Doruker P, Li H, Bahar I.
> *ProDy* 2.0: Increased Scale and Scope after 10 Years of Protein Dynamics Modelling with Python.
> *Bioinformatics* **2021** 37(20):3657-3659.

When using *pairwise2* or *KDTree* modules in published work, please cite:

> Cock PJ, Antao T, Chang JT, Chapman BA, Cox CJ, Dalke A, Friedberg I, Hamelryck T, Kauff F, Wilczynski B, de Hoon MJ.
> Biopython: freely available Python tools for computational molecular biology and bioinformatics.

[1111] http://www.linkedin.com/pub/anindita-dutta/5a/568/a90
[1112] https://scholar.google.pl/citations?user=1MwNI3EAAAAJ&hl=pl&oi=ao
[1113] http://www.linkedin.com/pub/wenzhi-mao/2a/29a/29
[1114] http://www.linkedin.com/in/lidio
[1115] http://www.linkedin.com/pub/ying-liu/15/48b/5a9
[1116] https://github.com/fdoljanin
[1117] https://github.com/kianho
[1118] http://blog.yeredusuncedernegi.com/

Bioinformatics **2009** 25(11):1422-3.

6.3 Credits

ProDy makes use of the following great software:

pyparsing[1119] is used to define the sophisticated atom selection grammar. This makes every user a power user by enabling fast access to and easy handling of atomic data via simple selection statements.

Biopython[1120] KDTree package and TreeConstruction module, which are distributed ProDy, significantly enrich and improve the ProDy user experience. KDtree package allows for fast distance based selections making atom selections suitable for contact identification. TreeConstruction module enables the calculation of phylogenetic trees.

ProDy requires NumPy[1121] for almost all major functionality including, but not limited to, storing atomic data and performing normal mode calculations. The power and speed of NumPy makes ProDy suitable for interactive and high-throughput structural analysis.

CEalign is distributed with ProDy. The original CE method was developed by Ilya Shindyalov and Philip Bourne. The Python version which is used by ProDy is developed by Jason Vertrees and available under the New BSD license.

Finally, ProDy can benefit from SciPy[1122] and Matplotlib[1123] packages. SciPy makes ProDy normal calculations more flexible and on low memory machines possible. Matplotlib allows greatly enriches user experience by allowing plotting protein dynamics data calculated using ProDy.

6.4 Funding

Continued development of protein dynamics software *ProDy* is supported by NIH through R01 GM099738 award.

6.5 License

6.5.1 ProDy

ProDy is available under the MIT License[1124]:

```
ProDy: A Python Package for Protein Dynamics Analysis

Copyright (C) 2010-2014 University of Pittsburgh

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
```

[1119] http://pyparsing.wikispaces.com
[1120] http://biopython.org
[1121] http://www.numpy.org
[1122] http://www.scipy.org
[1123] http://matplotlib.org
[1124] http://opensource.org/licenses/MIT

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

6.5.2 Biopython

'Biopython<https://biopython.org/>'_ KDTree package and TreeConstruction module are distributed with the ProDy package. Biopython is developed by The Biopython Consortium and is available under the Biopython license[1125]:

```
                  Biopython License Agreement

Permission to use, copy, modify, and distribute this software and its
documentation with or without modifications and for any purpose and
without fee is hereby granted, provided that any copyright notices
appear in all copies and that both those copyright notices and this
permission notice appear in supporting documentation, and that the
names of the contributors or copyright holders not be used in
advertising or publicity pertaining to distribution of the software
without specific prior permission.

THE CONTRIBUTORS AND COPYRIGHT HOLDERS OF THIS SOFTWARE DISCLAIM ALL
WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL THE
CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
OR PERFORMANCE OF THIS SOFTWARE.
```

6.5.3 Pyparsing

The pyparsing[1126] module is distributed with the ProDy package. Pyparsing is developed by Paul T. McGuire and is available under the MIT License[1127]:

```
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
```

[1125] http://www.biopython.org/DIST/LICENSE
[1126] http://pyparsing.wikispaces.com
[1127] http://opensource.org/licenses/MIT

```
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

6.5.4 Argparse

The argparse module[1128] is distributed with the ProDy package. Argparse is developed by Steven J. Bethard and is available under the Python Software Foundation License[1129].

6.5.5 CEalign

CEalign module is distributed with ProDy. The original CE method was developed by Ilya Shindyalov and Philip Bourne. The Python version which is used by ProDy is developed by Jason Vertrees and available under the New BSD license:

```
Copyright (c) 2007, Jason Vertrees.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in
    the documentation and/or other materials provided with the
    distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

6.5.6 scikit-learn

The discretization method for spectral clustering is redistributed and adapted with ProDy. scikit-learn is a Python module for machine learning built on top of SciPy and is distributed under the 3-Clause BSD license. The project was started in 2007 by David Cournapeau as a Google Summer of Code project, and since then many volunteers have contributed.

> New BSD License
>
> Copyright (c) 2007–2019 The scikit-learn developers. All rights reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

[1128] http://code.google.com/p/argparse/
[1129] http://docs.python.org/license.html

1. Redistributions of source code must retain the above copyright notice,

   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright

   notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the Scikit-learn Developers nor the names of

   its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

[AC12] Cournac A, Marie-Nelly H, Marbouty M, Koszul R, Mozziconacci J. Normalization of a chromosomal contact map. *BMC Genomics* **2012**.

[ZY09] Zheng Yang, Peter Májek, Ivet Bahar. Allosteric Transitions of Supramolecular Systems Explored by Network Models: Application to Chaperonin GroEL. *PLOS Comp Biol* **2009** 40:512-524.

[BR95] Bruschweiler R. Collective protein dynamics and nuclear spin relaxation. *J Chem Phys* **1995** 102:3396-3403.

[EB08] Eyal E., Bahar I. Toward a Molecular Understanding of the Anisotropic Response of Proteins to External Forces: Insights from Elastic Network Models. *Biophys J* **2008** 94:3424-34355.

[IB18] Dill K, Jernigan RL, Bahar I. Protein Actions: Principles and Modeling. *Garland Science* **2017**.

[CB95] Chennubhotla C., Bahar I. Signal Propagation in Proteins and Relation

[CS14] Sorzano COS, de la Rosa-Trevín JM, Tama F, Jonić S. Hybrid Electron Microscopy Normal Mode Analysis graphical interface and protocol. *J Struct Biol* **2014** 188:134-41.

[MH20] Harastani M, Sorzano COS, Jonić S. Hybrid Electron Microscopy Normal Mode Analysis with Scipion. *Protein Sci* **2020** 29:223-236.

[CS14] Sorzano COS, de la Rosa-Trevín JM, Tama F, Jonić S. Hybrid Electron Microscopy Normal Mode Analysis graphical interface and protocol. *J Struct Biol* **2014** 188:134-41.

[MH20] Harastani M, Sorzano COS, Jonić S. Hybrid Electron Microscopy Normal Mode Analysis with Scipion. *Protein Sci* **2020** 29:223-236.

[PD00] Doruker P, Atilgan AR, Bahar I. Dynamics of proteins predicted by molecular dynamics simulations and analytical approaches: Application to a-amylase inhibitor. *Proteins* **2000** 40:512-524.

[ARA01] Atilgan AR, Durrell SR, Jernigan RL, Demirel MC, Keskin O, Bahar I. Anisotropy of fluctuation dynamics of proteins with an elastic network model. *Biophys. J.* **2001** 80:505-515.

[AA99] Amadei A, Ceruso MA, Di Nola A. On the convergence of the conformational coordinates basis set obtained by the essential dynamics analysis of proteins' molecular dynamics simulations. *Proteins* **1999** 36(4):419-424.

[BH02] Hess B. Convergence of sampling in protein simulations. *Phys Rev E* **2002** 65(3):031910.

[SK02] Kundu S, Melton JS, Sorensen DC, Phillips GN: Dynamics of proteins in crystals: comparison of experiment with simple models. *Biophys J.* **2002**, 83:723-732.

[AA99] Amadei A, Ceruso MA, Di Nola A. On the convergence of the conformational coordinates basis set obtained by the essential dynamics analysis of proteins' molecular dynamics simulations. *Proteins* **1999** 36(4):419-424.

[SK02] Kundu S, Melton JS, Sorensen DC, Phillips GN: Dynamics of proteins in crystals: comparison of experiment with simple models. *Biophys J.* **2002**, 83:723-732.

[VC07] Carnevale V, Pontiggia F, Micheletti C. Structural and dynamical alignment of enzymes with partial structural similarity. *J Phys Condens Matter.* **2007** 19:285206.

[KH00] Hinsen K, Petrescu A-J, Dellerue S, Bellissent-Funel M-C, Kneller GR. Harmonicity in slow protein dynamics. *Chem Phys* **2000** 261:25-37.

[JS09] Stember JN, Wriggers W. Bend-twist-stretch model for coarse

[KB20] Kaynak B.T., Bahar I., Doruker P., Essential site scanning analysis: A new approach for detecting sites that modulate the dispersion of protein global motions, *Comput. Struct. Biotechnol. J.* **2020** 18:1577-1586.

[LGV09] Le Guilloux, V., Schmidtke P., Tuffery P., Fpocket: An open source platform for ligand pocket detection, *BMC Bioinformatics* **2009** 10:168.

[TL12] Lezon TR, Bahar I, Constraints Imposed by the Membrane Selectively Guide the Alternating Access Dynamics of the Glutamate Transporter GltPh

[TL12] Lezon TR, Bahar I, Constraints Imposed by the Membrane Selectively Guide the Alternating Access Dynamics of the Glutamate Transporter GltPh

[LT10] Lezon TR, Bahar I. Using entropy maximization to understand the determinants of structural dynamics beyond native contact topology. *PLoS Comput Biol* **2010** 6(6):e1000816.

[OL10] Orellana L, Rueda M, Ferrer-Costa C, Lopez-Blanco JR, Chacón P, Orozco M. Approaching Elastic Network Models to Molecular Dynamics Flexibility. *J Chem Theory Comput* **2010** 6(9):2910-23.

[CJ09] Camps J, Carrillo O, Emperador A, Orellana L, Hospital A, Rueda M, Cicin-Sain D, D'Abramo M, Gelpí JL, Orozco M. FlexServ: an integrated tool for the analysis of protein flexibility. *Bioinformatics* **2009** 25(13):1709-10.

[SP12] Sfriso P, Emperador A, Orellana L, Hospital A, Gelpí JL, Orozco M. Finding Conformational Transition Pathways from Discrete Molecular Dynamics Simulations. *J Chem Theory Comput* **2012** 8(11):4707-18.

[SP13] Sfriso P, Hospital A, Emperador A, Orozco M. Exploration of conformational transition pathways from coarse-grained simulations. *Bioinformatics* **2013** 29(16):1980-6.

[IB97] Bahar I, Atilgan AR, Erman B. Direct evaluation of thermal fluctuations in protein using a single parameter harmonic potential. *Folding & Design* **1997** 2:173-181.

[TH97] Haliloglu T, Bahar I, Erman B. Gaussian dynamics of folded proteins. *Phys. Rev. Lett.* **1997** 79:3090-3093.

[TL12] Lezon TR, Bahar I, Constraints Imposed by the Membrane Selectively Guide the Alternating Access Dynamics of the Glutamate Transporter GltPh

[KMR] Mikulska-Ruminska K., Kulik A.J., Benadiba C., Bahar I., Dietler G., Nowak W. Nanomechanics of multidomain neuronal cell adhesion protein contactin revealed by single molecule AFM and SMD. *Sci Rep* **2017** 7:8852.

[AA93] Amadei A, Linssen AB, Berendsen HJ. Essential dynamics of proteins. *Proteins* **1993** 17(4):412-25.

[CA09] Atilgan C, Atilgan AR, Perturbation-Response Scanning Reveals Ligand Entry-Exit Mechanisms of Ferric Binding Protein. *PLoS Comput Biol* **2009** 5(10):e1000544.

[IG14] General IJ, Liu Y, Blackburn ME, Mao W, Gierasch LM, Bahar I. ATPase subdomain IA is a mediator of interdomain allostery in Hsp70 molecular chaperones. *PLoS Comput. Biol.* **2014** 10:e1003624.

[ZNG13] Gerek ZN, Kumar S, Ozkan SB, Structural dynamics flexibility informs function and evolution at a proteome scale. *Evol Appl.* **2013** 6(3):423-33.

[AK15] **Kumar A, Glembo TJ, Ozkan SB. The Role of Conformational Dynamics and Allostery** in the Disease Development of Human Ferritin.

Biophys J. **2015** 109(6):1273-81.

[FT00] Tama F, Gadea FJ, Marques O, Sanejouand YH. Building-block approach for determining low-frequency normal modes of macromolecules. *Proteins* **2000** 41:1-7.

[IS98] Shindyalov IN, Bourne PE. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein engineering* **1998** 11(9):739-47.

[IS98] Shindyalov IN, Bourne PE. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein engineering* **1998** 11(9):739-47.

[WK83] Kabsch W, Sander C. Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers* **1983** 22:2577-2637.

[TM94] Martinetz T, Schulten K, Topology Representing Networks. *Neural Networks* **1994** 7(3):507-552.

[SS15] Salentin S., Schreiber S., Haupt V. J., Adasme M. F., Schroeder M.

[SS15] Salentin S., Schreiber S., Haupt V. J., Adasme M. F., Schroeder M.

[DRK13] Koes D. R., Baumgartner M. P., Camacho C. J., Lessons Learned in

[DF95] Frishman D, Argos P. Knowledge-Based Protein Secondary Structure Assignment. *Proteins* **1995** 23:566-579.

[DSD08] Dunn SD, Wahl LM, Gloor GB. Mutual information without the influence of phylogeny or entropy dramatically improves residue contact prediction. *Bioinformatics* **2008** 24(3):333-340.

[MLC05] Martin LC, Gloor GB, Dunn SD, Wahl LM. Using information theory to search for co-evolving residues in proteins. *Bioinformatics* **2005** 21(22):4116-4124.

[KM68] Murty KG. Letter to the editor-An algorithm for ranking all the assignments in order of increasing cost. *Operations research* **1968** 16(3):682-687.

[CA09] Atilgan C, Atilgan AR. Perturbation-response scanning reveals ligand entry-exit mechanisms of ferric binding protein. *PLoS Comput. Biol.* **2009** 5:e1000544.

a

c

d

e

k

m

p

s

t

u

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W