ProDy

Protein Dynamics & Sequence Analysis

ProDy Documentation

Release 1.5.0

Ahmet Bakan

December 23, 2013

Contents

CHAPTER 1

Installation

1.1 Required Software

- Python[1] 2.6, 2.7, 3.2 or later

  Windows: You need to use **32-bit** Python on Windows to be able to install NumPy and ProDy.

- NumPy[2] 1.7 or later

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

1.2 Quick Install

If you have pip[4] installed, type the following:

```
pip install -U ProDy
```

If you don't have pip[5], please download an installation file and follow the instructions.

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 **??**).

Linux

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

[1] http://www.python.org
[2] http://www.numpy.org
[3] http://matplotlib.org
[4] http://www.pip-installer.org
[5] http://www.pip-installer.org

```
$ 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[6].

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-0.x.y.win32-py2.z.exe` and run to install ProDy.

To be able use *ProDy Applications* (page **??**) and *Evol Applications* (page **??**) in command prompt (**cmd.exe**), append Python and scripts folders (e.g. `C:\Python27` and `C:\Python27\Scripts`) to `PATH`[7] environment variable.

1.4 Recommended Software

- Scipy[8], 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] for reading molecular dynamics trajectories.

1.5 Included in ProDy

Following software is included in the ProDy installation packages:

- pyparsing[12] is used to define the atom selection grammar.
- Biopython[13] 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.

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

CHAPTER 2

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}
             ...

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}
    anm                 perform anisotropic network model 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

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 **??**):

```
$ 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[1].

[1] http://csb.pitt.edu/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[2] 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[3] environment variable yourself.

[2] http://matplotlib.sourceforge.net/faq/environment_variables_faq.html#envvar-PATH
[3] http://matplotlib.sourceforge.net/faq/environment_variables_faq.html#envvar-PATH

CHAPTER 3

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 **??**) 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 **??**) class, following classes that act as pointers provide convenient access subset of data:

- `Selection` (page **??**) - Points to an arbitrary subset of atoms. See *Atom Selections* (page **??**) and *Operations on Selections*[1] for usage examples.
- `Segment` (page **??**) - Points to atoms that have the same segment name.
- `Chain` (page **??**) - Points to atoms in a segment that have the same chain identifier.
- `Residue` (page **??**) - Points to atoms in a chain that have the same residue number and insertion code.
- `AtomMap` (page **??**) - 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 **??**) - Points to a single atom
- `Bond` (page **??**) - Points to two connected atoms

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

3.1.2 Atom data fields

Atom Data Fields (page **??**) 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:

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

3.1.3 Atom flags

Atom Flags (page **??**) 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 **??**) 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 **??**). 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

Following functions can be used for permanent data storage:

- `loadAtoms()` (page **??**)
- `saveAtoms()` (page **??**)

Following function can be used to identify fragments in a group (`AtomGroup` (page **??**)) or subset (`Selection` (page **??**)) of atoms:

- `findFragments()` (page **??**)
- `iterFragments()` (page **??**)

Following function can be used to get an `AtomMap` (page **??**) that sorts atoms based on a given property:

- `sortAtoms()` (page **??**)

Following function can be used check whether a word is reserved because it is used internally by `prody.atomic` (page **??**) classes:

- `isReserved()` (page **??**)
- `listReservedWords()` (page **??**)

3.1.6 Atom

This module defines classes to handle individual atoms.

class Atom(*ag*, *index*, *acsi*)

A class for handling individual atoms in an `AtomGroup` (page **??**).

copy()

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

getACSIndex()

Return index of the coordinate set.

getACSLabel()

Return 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()

Return anisotropic temperature factor of the atom.

getAnistd()

Return standard deviations for anisotropic temperature factor of the atom.

getAtomGroup()

Return 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'`.

getCSLabels()
Return 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 **??**) instance. Chain index can be used in atom selections, e.g. `'chindex 0'`.

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

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

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

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

getDataType(*label*)
Return 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*)
Return atom flag.

getFlagLabels(*which=None*)
Return 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 **??**) 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup` (page **??**) 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()
Return index of the atom.

getIndices()
Return index of the atom in an numpy.ndarray[2].

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

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 **??**) 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*.

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 **??**) 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()
Return selection string that will select this atom.

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'.

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

[2] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

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

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

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.

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

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

numCoordsets()
Return number of coordinate sets.

select(*selstr*, ***kwargs*)
Return atoms matching *selstr* criteria. See `select` (page **??**) 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(*data*)
Set anisotropic temperature factor of the atom.

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 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 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 _'`.

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*.

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'`.

3.1.7 Atom Group

This module defines `AtomGroup` (page **??**) class that stores atomic data and multiple coordinate sets in `numpy.ndarray`[3] instances.

[3] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

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 `numpy.ndarray`[4] instances.

Get and set methods

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

set methods, e.g. `setResnums()` (page **??**), accept data in `list()`[5] or `ndarray`[6] 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 **??**) method. `getCoords()` (page **??**) 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 **??**) instances may be used. A `Selection` (page **??**) has initially the same coordinate set as the *active coordinate set*, but it may be changed using `Selection.setACSIndex()` (page **??**) method.

Customizations

Following built-in functions are customized for this class:

- `len()`[7] returns the number of atoms, i.e. `numAtoms()` (page **??**)
- `iter()`[8] yields `Atom` (page **??**) instances

Indexing AtomGroup (page ??) instances by:

- *int* (`int()`[9]), e.g, `10`, returns an `Atom` (page **??**)
- *slice* (`slice()`[10]), e.g, `10:20:2`, returns a `Selection` (page **??**)
- *segment name* (`str()`[11]), e.g. `'PROT'`, returns a a `Segment` (page **??**)
- *chain identifier* (`str()`[12]), e.g. `'A'`, returns a a `Chain` (page **??**)
- *[segment name,] chain identifier, residue number[, insertion code]* (`tuple()`[13]), e.g. `'A', 10` or `'A', 10, 'B'` or `'PROT', 'A', 10, 'B'`, returns a `Residue` (page **??**)

Addition

Addition of two `AtomGroup` (page **??**) instances, let's say *A* and *B*, results in a new `AtomGroup` (page **??**) 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

[4] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[5] http://docs.python.org/library/functions.html#list
[6] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[7] http://docs.python.org/library/functions.html#len
[8] http://docs.python.org/library/functions.html#iter
[9] http://docs.python.org/library/functions.html#int
[10] http://docs.python.org/library/functions.html#slice
[11] http://docs.python.org/library/functions.html#str
[12] http://docs.python.org/library/functions.html#str
[13] http://docs.python.org/library/functions.html#tuple

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*)
Add a coordinate set. *coords* argument may be an object with `getCoordsets()` (page **??**) method.

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

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

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

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

getACSIndex()
Return index of the coordinate set.

getACSLabel()
Return 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()
Return a copy of anisotropic temperature factors.

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'`.

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`. *step* (default is 1) specifies increment. If atoms with matching serial numbers are not found, **None** will be returned.

getCSLabels()
Return 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 **??**) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

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

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

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

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

getDataType(*label*)
Return 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*)
Return 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 **??**) labels in addition to those present in the instance.

getFlags(*label*)
Return 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup` (page **??**) 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*)
Return 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 _'`.

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 **??**) 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*.

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 **??**) 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*.

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

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

iterAtoms()
Yield atom instances.

iterBonds()
Yield bonds. Use `setBonds()` (page **??**) for setting bonds.

iterChains()
Iterate over chains.

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

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

iterResidues()
Iterate over residues.

iterSegments()
Iterate over chains.

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

numBonds()
Return number of bonds. Use `setBonds()` (page **??**) for setting bonds.

numBytes(*all=False*)
Return 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()
Return number of chains.

numCoordsets()
Return number of coordinate sets.

numFragments()
Return number of connected atom subsets.

numResidues()
Return number of residues.

numSegments()
Return number of segments.

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

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

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

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

setAnisous(*data*)
Set anisotropic temperature factors.

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 **??**) 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.

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 **??**) 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 **??**). 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.

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 **??**))

data must be a `list()`[14] or a `ndarray`[15] 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.

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 _'`.

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'`.

[14] http://docs.python.org/library/functions.html#list
[15] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

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*.

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'`.

3.1.8 Atomic Base

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

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

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

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

3.1.9 Atom Map

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

How AtomMap's work

`AtomMap` (page **??**) class adds great flexibility to manipulating atomic data.

First let's see how an instance of `Selection` (page **??**) (`Chain` (page **??**), or `Residue` (page **??**)) works. Below table shows indices for a selection of atoms in an `AtomGroup` (page **??**) and values returned when `getNames()` (page **??**), `getResnames()` (page **??**) and `getResnums()` (page **??**) 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 **??**) instances keep indices ordered and do not allow duplicate values, hence their use is limited. In an `AtomMap` (page **??**), 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 **??**) 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 **??**) are used by `proteins` (page **??**) module functions that match or map protein chains. *Heterogeneous X-ray Structures*[16] and *Multimeric Structures*[17] examples that make use of these functions and `AtomMap` (page **??**) 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

[16] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray.html#pca-xray
[17] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/dimer.html#pca-dimer

- **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()`[18] returns the number of atoms in the instance.
- `iter()`[19] yields `Atom` (page **??**) instances.
- Indexing returns an `Atom` (page **??**) or an `AtomMap` (page **??**) instance depending on the type and value of the index.

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

getACSIndex()
Return index of the coordinate set.

getACSLabel()
Return active coordinate set label.

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

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

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

getAtomGroup()
Return associated atom group.

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

getCSLabels()
Return 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 **??**) instance. Entries for dummy atoms will be `0`.

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

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

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

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

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

getDataType(*label*)
Return 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*)
Return 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 **??**) labels in addition to those present in the instance.

getFlags(*label*)
Return 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup` (page **??**) instance. Entries for dummy atoms will be `0`.

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

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

getMapping()
Return 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 **??**) 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`.

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 **??**) instance. Entries for dummy atoms will be `0`.

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

getSelstr()
Return selection string that selects mapped atoms.

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

getTitle()
Return title of the instance.

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

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

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

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

iterCoordsets()
Yield copies of coordinate sets.

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

numCoordsets()
Return number of coordinate sets.

numDummies()
Return number of dummy atoms.

numMapped()
Return number of mapped atoms.

select(*selstr*, ***kwargs*)
Return atoms matching *selstr* criteria. See `select` (page **??**) 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.

3.1.10 Bond

This module defines `Bond` (page **??**) for dealing with bond information provided by using `AtomGroup.setBonds()` (page **??**) method.

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

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

- `len()`[20] returns bond length, i.e. `getLength()` (page **??**)
- `iter()`[21] yields `Atom` (page **??**) instances

getACSIndex()
Return index of the coordinate set.

getAtomGroup()
Return atom group.

getAtoms()
Return bonded atoms.

getIndices()
Return indices of bonded atoms.

getLength()
Return bond length.

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

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

3.1.11 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 **??**) class. Following built-in functions are customized for this class:

- `len()`[22] returns the number of residues in the chain
- `iter()`[23] yields `Residue` (page **??**) instances

Indexing `Chain` (page **??**) instances by:

- *residue number [, insertion code]* (`tuple()`[24]), e.g. `10` or `10, "B"`, returns a `Residue` (page **??**)
- *slice* (`slice()`[25]), e.g, `10:20`, returns a list of `Residue` (page **??**) instances

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

getACSIndex()
Return index of the coordinate set.

[20] http://docs.python.org/library/functions.html#len
[21] http://docs.python.org/library/functions.html#iter
[22] http://docs.python.org/library/functions.html#len
[23] http://docs.python.org/library/functions.html#iter
[24] http://docs.python.org/library/functions.html#tuple
[25] http://docs.python.org/library/functions.html#slice

getACSLabel()
Return 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()
Return a copy of anisotropic temperature factors.

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

getAtomGroup()
Return 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'`.

getCSLabels()
Return 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()
Return 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()
Return 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 **??**) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

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

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

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

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

getDataType(*label*)
Return 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*)
Return 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 **??**) labels in addition to those present in the instance.

getFlags(*label*)
Return 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup` (page **??**) 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()
Return 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*)
Return 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 **??**) 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*.

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 **??**) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegment()
Return segment of the chain.

getSegname()
Return 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()
Return selection string that selects atoms in this chain.

getSequence(***kwargs*)
Return 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'`.

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

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

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

iterAtoms()
Yield atoms.

iterCoordsets()
Yield copies of coordinate sets.

iterResidues()
Yield residues.

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

numCoordsets()
Return number of coordinate sets.

numResidues()
Return number of residues.

select(*selstr*, ***kwargs*)
Return atoms matching *selstr* criteria. See `select` (page **??**) 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(*data*)
Set anisotropic temperature factors.

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 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 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*.

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'.

3.1.12 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 **??**) instances. Available data fields are listed in the table below. Atomic classes, such as Selection (page **??**), offer get and set for handling parsed data:

Many of these data fields can be used to make *Atom Selections* (page **??**). 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 _'

anisou anisotropic temperature factor

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'

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'`

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*)
Return documentation string for the field.

`call`
list of `AtomGroup` (page **??**) 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 **??**) `_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 **??**) attributes to be set None, when `setMethod` is called

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

readonly
read-only attribute without a set method

selstr
list of selection string examples

synonym
synonym used in atom selections

3.1.13 Atom Flags

This module defines atom flags that are used in *Atom Selections* (page **??**). 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 **??**) 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[26], ARG[27], ASN[28], ASP[29], CYS[30], GLN[31], GLU[32], GLY[33], HIS[34], ILE[35], LEU[36], LYS[37], MET[38], PHE[39], PRO[40], SER[41], THR[42], TRP[43], TYR[44], and VAL[45]

nonstdaa indicates one of the following residues:

[26] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ALA
[27] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ARG
[28] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ASN
[29] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ASP
[30] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CYS
[31] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLN
[32] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLU
[33] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLY
[34] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HIS
[35] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ILE
[36] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=LEU
[37] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=LYS
[38] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MET
[39] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PHE
[40] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PRO
[41] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SER
[42] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=THR
[43] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TRP
[44] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TYR
[45] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=VAL

ASX[46] (B)	asparagine or aspartic acid
GLX[47] (Z)	glutamine or glutamic acid
CSO[48] (C)	S-hydroxycysteine
HIP[49] (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[50]	selenomethionine
SEC[51] (U)	selenocysteine
SEP[52] (S)	phosphoserine
TPO[53] (T)	phosphothreonine
PTR[54] (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 **??**), `delNonstdAminoacid()` (page **??**), and `listNonstdAAProps()` (page **??**).

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'`

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

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

acyclic residues ALA, ARG, ASN, ASP, ASX, CSO, CYS, GLN, GLU, GLX, GLY, ILE, EU, LYS, MET, MSE, SEC, SEP, SER, THR, TPO, VAL, XLE

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

aromatic residues HIS, PHE, PTR, TRP, TYR

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

buried residues ALA, CYS, ILE, LEU, MET, MSE, PHE, SEC, TRP, VAL, XLE

charged residues ARG, ASP, GLU, HIS, LYS

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

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

large residues ARG, GLN, GLU, GLX, HIP, HIS, HSD, HSE, HSP, ILE, LEU, LYS, ET, MSE, PHE, PTR, SEP, TPO, TRP, TYR, XLE

[46] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ASX
[47] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLX
[48] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CSO
[49] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HIP
[50] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MSE
[51] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SEC
[52] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SEP
[53] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TPO
[54] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PTR

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

neutral residues ALA, ASN, CSO, CYS, GLN, GLY, ILE, LEU, MET, MSE, PHE, PRO, EC, SER, THR, TRP, TYR, VAL

polar residues ARG, ASN, ASP, ASX, CSO, CYS, GLN, GLU, GLX, GLY, HIP, HIS, SD, HSE, HSP, LYS, PTR, SEC, SEP, SER, THR, TPO, TYR

small residues ALA, GLY, SER

surface residues ARG, ASN, ASP, ASX, CSO, GLN, GLU, GLX, GLY, HIP, HIS, HSD, SE, HSP, LYS, 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[55] (adenine), GUN[56] (guanine), CYT[57] (cytosine), THY[58] (thymine), and URA[59] (uracil).

nucleotide indicates residues with the following names:

DA[60]	2′-deoxyadenosine-5′-monophosphate
DC[61]	2′-deoxycytidine-5′-monophosphate
DG[62]	2′-deoxyguanosine-5′-monophosphate
DT[63]	2′-deoxythymidine-5′-monophosphate
DU[64]	2′-deoxyuridine-5′-monophosphate
A[65]	adenosine-5′-monophosphate
C[66]	cytidine-5′-monophosphate
G[67]	guanosine-5′-monophosphate
T[68]	2′-deoxythymidine-5′-monophosphate
U[69]	uridine-5′-monophosphate

nucleoside indicates following nucleoside derivatives that are recognized by *PDB*:

[55] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ADE
[56] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GUN
[57] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CYT
[58] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=THY
[59] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=URA
[60] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DA
[61] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DC
[62] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DG
[63] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DT
[64] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DU
[65] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=A
[66] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=C
[67] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=G
[68] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=T
[69] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=U

AMP[70]	adenosine monophosphate
ADP[71]	adenosine-5′-diphosphate
ATP[72]	adenosine-5′-triphosphate
CDP[73]	cytidine-5′-diphosphate
CTP[74]	cytidine-5′-triphosphate
GMP[75]	guanosine
GDP[76]	guanosine-5′-diphosphate
GTP[77]	guanosine-5′-triphosphate
TMP[78]	thymidine-5′-phosphate
TTP[79]	thymidine-5′-triphosphate
UMP[80]	2′-deoxyuridine 5′-monophosphate
UDP[81]	uridine 5′-diphosphate
UTP[82]	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'`

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[83] and DOD[84] 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[85]	aluminum	Yes		
BA[86]	barium	Yes		

Continued on next page

[70] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=AMP
[71] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ADP
[72] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ATP
[73] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CDP
[74] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CTP
[75] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GMP
[76] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GDP
[77] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GTP
[78] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TMP
[79] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TTP
[80] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=UMP
[81] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=UDP
[82] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=UTP
[83] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HOH
[84] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DOD
[85] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=AL
[86] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=BA

Table 3.3 – continued from previous page

CA[87]	calcium	Yes		
CD[88]	cadmium	Yes		
CL[89]	chloride	Yes		
CO[90]	cobalt (ii)	Yes		
CS[91]	cesium	Yes		
CU[92]	copper (ii)	Yes		
CU1[93]	copper (i)	Yes		
CUA[94]	dinuclear copper	Yes		
HG[95]	mercury (ii)	Yes		
IN[96]	indium (iii)	Yes		
IOD[97]	iodide	Yes		
K[98]	potassium	Yes		
MG[99]	magnesium	Yes		
MN3[100]	manganese (iii)	Yes		
NA[101]	sodium	Yes		
PB[102]	lead (ii)	Yes		
PT[103]	platinum (ii)	Yes		
RB[104]	rubidium	Yes		
TB[105]	terbium (iii)	Yes		
TL[106]	thallium (i)	Yes		
WO4[107]	thungstate (vi)	Yes		
YB[108]	ytterbium (iii)	Yes		
ZN[109]	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

Ion identifiers that are obsoleted by *PDB* (MO3, MO4, MO5, MO6, NAW, OC7, and ZN1) are removed from this definition.

[87] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CA
[88] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CD
[89] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CL
[90] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CO
[91] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CS
[92] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CU
[93] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CU1
[94] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=CUA
[95] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HG
[96] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=IN
[97] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=IOD
[98] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=K
[99] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MG
[100] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MN3
[101] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=NA
[102] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PB
[103] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=PT
[104] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=RB
[105] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TB
[106] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=TL
[107] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=WO4
[108] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=YB
[109] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=ZN

lipid indicates GPE[110], LPP[111], OLA[112], SDS[113], and STE[114] from *PDB*, and also POPC, LPPC, POPE, DLPE, PCGL, STEA, PALM, OLEO, DMPC from *CHARMM* force field.

sugar indicates BGC[115], GLC[116], and GLO[117] from *PDB*, and also AGLC from *CHARMM*.

heme indicates 1FH[118], 2FH[119], DDH[120], DHE[121], HAS[122], HDD[123], HDE[124], HDM[125], HEA[126], HEB[127], HEC[128], HEM[129], HEO[130], HES[131], HEV[132], NTE[133], SRM[134], and VER[135] from *PDB*, and also HEMO 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.

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'`

[110] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GPE
[111] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=LPP
[112] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=OLA
[113] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SDS
[114] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=STE
[115] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=BGC
[116] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLC
[117] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=GLO
[118] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=1FH
[119] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=2FH
[120] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DDH
[121] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=DHE
[122] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HAS
[123] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HDD
[124] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HDE
[125] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HDM
[126] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEA
[127] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEB
[128] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEC
[129] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEM
[130] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEO
[131] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HES
[132] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=HEV
[133] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=NTE
[134] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=SRM
[135] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=VER

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 **??**)

mapped indicates mapped atoms in an `AtomMap` (page **??**)

Functions

Following functions can be used to customize flag definitions:

- `flagDefinition()` (page **??**)
- `addNonstdAminoacid()` (page **??**)
- `delNonstdAminoacid()` (page **??**)
- `listNonstdAAProps()` (page **??**)

flagDefinition(**arg*, ***kwarg*)

Learn, change, or reset *Atom Flags* (page **??**) 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*)
Return 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 **??**) 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',
 'CSO',
 'GLX',
 'HIP',
 'HSD',
 'HSE',
 'HSP',
 'MSE',
 'SEC',
 'SEP',
 'TPO',
 'XAA',
 'XLE']
```

Default set of non-standard amino acids can be restored as follows:

```
In [3]: flagDefinition(reset='nonstdaa')
```

3.1.14 Supporting Functions

This module defines some functions for handling atomic classes and data.

iterFragments(*atoms*)
Yield fragments, connected subsets in *atoms*, as `Selection` (page **??**) instances.

findFragments(*atoms*)
Return list of fragments, connected subsets in *atoms*. See also `iterFragments()` (page **??**).

loadAtoms(*filename*)
Return `AtomGroup` (page **??**) instance loaded from *filename* using `numpy.load()`[136] function. See also `saveAtoms()` (page **??**).

saveAtoms(*atoms*, *filename=None*, ***kwargs*)
Save *atoms* in ProDy internal format. All `Atomic` (page **??**) classes are accepted as *atoms* argument. This function saves user set atomic data as well. Note that title of the `AtomGroup` (page **??**) instance is used as the filename when *atoms* is not an `AtomGroup` (page **??**). To avoid overwriting an existing file with the same name, specify a *filename*.

isReserved(*word*)
Return **True** if *word* is reserved for internal data labeling or atom selections. See `listReservedWords()` (page **??**) for a list of reserved words.

listReservedWords()
Return list of words that are reserved for atom selections and internal variables. These words are: *abs*, *acidic*, *acos*, *acyclic*, *aliphatic*, *all*, *altloc*, *aminoacid*, *and*, *anisou*, *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*, *secondary*, *secstr*,

[136] http://docs.scipy.org/doc/numpy/reference/generated/numpy.load.html#numpy.load

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*)
Return an `AtomMap` (page **??**) pointing to *atoms* sorted in ascending data *label* order, or optionally in *reverse* order.

3.1.15 Hierarchical Views

This module defines `HierView` (page **??**) class that builds a hierarchical views of atom groups.

class HierView (*atoms*, ***kwargs*)
Hierarchical views can be generated for `AtomGroup` (page **??**) and `Selection` (page **??**) instances. Indexing a `HierView` (page **??**) instance returns a `Chain` (page **??**) instance.

Some `object` methods are customized as follows:

- `len()`[137] returns the number of atoms, i.e. `numChains()` (page **??**)
- `iter()`[138] yields `Chain` (page **??**) instances
- **indexing by:**
  - *segment name* (`str()`[139]), e.g. `"PROT"`, returns a `Segment` (page **??**)
  - *chain identifier* (`str()`[140]), e.g. `"A"`, returns a `Chain` (page **??**)
  - *[segment name,] chain identifier, residue number[, insertion code]* (`tuple()`[141]), e.g. `"A", 10` or `"A", 10, "B"` or `"PROT", "A", 10, "B"`, returns a `Residue` (page **??**)

Note that when an `AtomGroup` (page **??**) instance have distinct segments, they will be considered when building the hierarchical view. A `Segment` (page **??**) 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 ()
Return atoms for which the hierarchical view was built.

getChain (*chid*, *segname=None*)
Return chain with identifier *chid*, if it is present.

getResidue (*chid*, *resnum*, *icode=None*, *segname=None*)
Return residue with number *resnum* and insertion code *icode* from the chain with identifier *chid* in segment with name *segname*.

getSegment (*segname*)
Return segment with name *segname*, if it is present.

iterChains ()
Yield chains.

iterResidues ()
Yield residues.

iterSegments ()
Yield segments.

[137] http://docs.python.org/library/functions.html#len
[138] http://docs.python.org/library/functions.html#iter
[139] http://docs.python.org/library/functions.html#str
[140] http://docs.python.org/library/functions.html#str
[141] http://docs.python.org/library/functions.html#tuple

numChains()
Return number of chains.

numResidues()
Return number of residues.

numSegments()
Return 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.

3.1.16 Atom Pointer

This module defines atom pointer base class.

class AtomPointer(*ag*, *acsi*)
A base for classes pointing to atoms in `AtomGroup` (page **??**) instances. Derived classes are:

- `Atom` (page **??**)
- `AtomSubset` (page **??**)
- `AtomMap` (page **??**)

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

getACSIndex()
Return index of the coordinate set.

getACSLabel()
Return active coordinate set label.

getAtomGroup()
Return associated atom group.

getCSLabels()
Return coordinate set labels.

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

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

getFlagLabels(*which=None*)
Return 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 **??**) labels in addition to those present in the instance.

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

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

numCoordsets()
Return number of coordinate sets.

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

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

3.1.17 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 **??**) class. Following built-in functions are customized for this class:

- `len()`[142] returns the number of atoms in the instance.
- `iter()`[143] yields `Atom` (page **??**) instances.

Indexing `Residue` (page **??**) instances by *atom name* (`str()`[144]), e.g. `"CA"` returns an `Atom` (page **??**) instance.

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

getACSIndex()
Return index of the coordinate set.

getACSLabel()
Return 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()
Return a copy of anisotropic temperature factors.

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

getAtom(*name*)
Return 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()
Return 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'`.

getCSLabels()
Return coordinate set labels.

getChain()
Return the chain that the residue belongs to.

[142] http://docs.python.org/library/functions.html#len
[143] http://docs.python.org/library/functions.html#iter
[144] http://docs.python.org/library/functions.html#str

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()
Return 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 **??**) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

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

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

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

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

getDataType(*label*)
Return 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*)
Return 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 **??**) labels in addition to those present in the instance.

getFlags(*label*)
Return 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup` (page **??**) instance. Fragment indices can be used in atom selections, e.g. `'fragindex 0'`, `'fragment 1'`. Note that *fragment* is a synonym for *fragindex*.

getIcode()
Return residue insertion code.

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

getIndices()
Return 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()
Return 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()
Return 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()
Return 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 **??**) instance. Residue indices can be used in atom selections, e.g. `'resindex 0'`.

getResname()
Return residue name.

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

getResnum()
Return 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*.

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 **??**) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegment()
Return segment of the residue.

getSegname()
Return 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()
Return selection string that will select this residue.

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'`.

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

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

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

iterAtoms()
Yield atoms.

iterCoordsets()
Yield copies of coordinate sets.

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

numCoordsets()
Return number of coordinate sets.

select(*selstr*, ***kwargs*)
Return atoms matching *selstr* criteria. See `select` (page **??**) 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(*data*)
Set anisotropic temperature factors.

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 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 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*.

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'`.

3.1.18 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 **??**) class. Following built-in functions are customized for this class:

- `len()`[145] returns the number of chains in the segment.
- `iter()`[146] yields `Chain` (page **??**) instances.

Indexing `Segment` (page **??**) instances by a *chain identifier* (`str()`[147]), e.g. `A`, returns a `Chain` (page **??**).

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

getACSIndex()
Return index of the coordinate set.

getACSLabel()
Return 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()
Return a copy of anisotropic temperature factors.

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

getAtomGroup()
Return 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'`.

getCSLabels()
Return coordinate set labels.

getChain(*chid*)
Return 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,

[145] http://docs.python.org/library/functions.html#len
[146] http://docs.python.org/library/functions.html#iter
[147] http://docs.python.org/library/functions.html#str

and are assigned in the order of appearance in `AtomGroup` (page **??**) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

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

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

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

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

getDataType(*label*)
Return 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*)
Return 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 **??**) labels in addition to those present in the instance.

getFlags(*label*)
Return 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup` (page **??**) 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()
Return 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 **??**) 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*.

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 **??**) instance. Segment indices can be used in atom selections, e.g. `'segindex 0'`.

getSegname()
Return 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()
Return selection string that selects atoms in this segment.

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'`.

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

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

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

iterAtoms()
Yield atoms.

iterChains()
Yield chains.

iterCoordsets()
Yield copies of coordinate sets.

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

numChains()
Return number of chains.

numCoordsets()
Return number of coordinate sets.

select(*selstr*, ***kwargs*)
Return atoms matching *selstr* criteria. See `select` (page **??**) 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(*data*)
Set anisotropic temperature factors.

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 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 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*.

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*.

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'`.

3.1.19 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 **??**). 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*[148] and *Operations on Selections*[149].

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 **??**) returns `AtomGroup` (page **??**) instances, `p` in this case, that stores all atomic data in the file. We can count different types of atoms using *Atom Flags* (page **??**) and `numAtoms()` (page **??**) method as follows:

```
In [3]: p.numAtoms('protein')
 Out[3]: 2606

In [4]: p.numAtoms('nucleic')
 Out[4]: 509

In [5]: p.numAtoms('hetero')
```

[148] http://prody.csb.pitt.edu/tutorials/structure_analysis/contacts.html#contacts
[149] http://prody.csb.pitt.edu/tutorials/prody_tutorial/selection.html#selection-operations

```
Out[5]: 96

In [6]: p.numAtoms('water')
Out[6]: 70
```

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 **??**) method. All *Atom Flags* (page **??**) 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 **??**) 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 **??**) 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 **??**), *Atom Data Fields* (page **??**) 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 **??**) 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 **??**).

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`[150].

Numerical comparisons

Atom Data Fields (page **??**) 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)>
```

[150] http://docs.python.org/library/re.html#re

```
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 Å 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 (Å) 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 Å 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 Å 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.f. `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 **??**) 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 and defaults to 1.

Selection macros

ProDy allows you to define a macro for any valid selection string. Below functions are for manipulating selection macros:

- `defSelectionMacro()` (page **??**)
- `delSelectionMacro()` (page **??**)
- `getSelectionMacro()` (page **??**)
- `isSelectionMacro()` (page **??**)

```
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 **??**) 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 **??**)) 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 **??**) 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 **??**) module documentation for selection grammar and examples. This class makes use of pyparsing[151] module.

`getBoolArray` (*atoms*, *selstr*, ***kwargs*)
Return a boolean array with **True** values for *atoms* matching *selstr*. The length of the boolean `numpy.ndarray`[152] will be equal to the length of *atoms* argument.

`getIndices` (*atoms*, *selstr*, ***kwargs*)
Return 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 **??**) instance.

`select` (*atoms*, *selstr*, ***kwargs*)
Return a `Selection` (page **??**) of atoms matching *selstr*, or **None**, if selection string does not match any atoms.

Parameters

- **atoms** (`Atomic` (page **??**)) – atoms to be evaluated
- **selstr** (*str*[153]) – selection string

Note that, if *atoms* is an `AtomMap` (page **??**) instance, an `AtomMap` (page **??**) is returned, instead of a a `Selection` (page **??**).

`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.

[151] http://pyparsing.wikispaces.com
[152] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[153] http://docs.python.org/library/functions.html#str

```
In [1]: defSelectionMacro('cbeta', 'name CB and protein')
```

delSelectionMacro(*name*)
Delete the macro *name*.

```
In [1]: delSelectionMacro('cbeta')
```

getSelectionMacro(*name=None*)
Return the definition of the macro *name*. If *name* is not given, returns a copy of the selection macros dictionary.

isSelectionMacro(*word*)
Return **True** if *word* is a user defined selection macro.

3.1.20 Selection

This module defines `Selection` (page **??**) 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 **??**) instance. Instances can be generated using `select()` (page **??**) method. Following built-in functions are customized for this class:

- `len()`[154] returns the number of selected atoms
- `iter()`[155] yields `Atom` (page **??**) instances

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

getACSIndex()
Return index of the coordinate set.

getACSLabel()
Return 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()
Return a copy of anisotropic temperature factors.

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

getAtomGroup()
Return 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'`.

getCSLabels()
Return coordinate set labels.

[154] http://docs.python.org/library/functions.html#len
[155] http://docs.python.org/library/functions.html#iter

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 **??**) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

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

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

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

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

getDataType(*label*)
Return 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*)
Return 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 **??**) labels in addition to those present in the instance.

getFlags(*label*)
Return 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup` (page **??**) 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*)
Return 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()
Return 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 **??**) 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*.

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 **??**) 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()
Return selection string that selects this atom subset.

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'`.

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

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

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

iterAtoms()
Yield atoms.

iterCoordsets()
Yield copies of coordinate sets.

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

numCoordsets()
Return number of coordinate sets.

select(*selstr*, ***kwargs*)
Return atoms matching *selstr* criteria. See `select` (page **??**) 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(*data*)
Set anisotropic temperature factors.

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 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 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*.

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'`.

update()
Update selection.

3.1.21 Atom Subsets

class AtomSubset(*ag*, *indices*, *acsi*, ***kwargs*)
A class for manipulating subset of atoms in an `AtomGroup` (page **??**). Derived classes are:

- `Selection` (page **??**)
- `Segment` (page **??**)
- `Chain` (page **??**)
- `Residue` (page **??**)

This class stores a reference to an `AtomGroup` (page **??**) instance, a set of atom indices, and active coordinate set index for the atom group.

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

getACSIndex()
Return index of the coordinate set.

getACSLabel()
Return 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()
Return a copy of anisotropic temperature factors.

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

getAtomGroup()
Return 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'`.

getCSLabels()
Return 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 **??**) instance. Chain indices can be used in atom selections, e.g. `'chindex 0'`.

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

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

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

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

getDataType(*label*)
Return 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*)
Return 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 **??**) labels in addition to those present in the instance.

getFlags(*label*)
Return 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 **??**) method. Fragment indices start from zero, are incremented by one, and are assigned in the order of appearance in `AtomGroup`

(page **??**) 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()
Return 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 **??**) 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*.

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 **??**) 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*.

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'`.

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

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

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

iterAtoms()
Yield atoms.

iterCoordsets()
Yield copies of coordinate sets.

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

numCoordsets()
Return number of coordinate sets.

select(*selstr*, ***kwargs*)
Return atoms matching *selstr* criteria. See `select` (page **??**) 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(*data*)
Set anisotropic temperature factors.

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 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 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*.

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'`.

3.2 Database Support

This module contains features for accessing databases containing protein related data.

3.2.1 Pfam

Following functions can be used to search and retrieve Pfam[156] data:

- `fetchPfamMSA()` (page **??**) - download MSA files
- `searchPfam()` (page **??**) - search families of a protein

[156] http://pfam.sanger.ac.uk/

3.2.2 Pfam Access Functions

This module defines functions for interfacing Pfam database.

searchPfam(*query*, *search_b=False*, *skip_a=False*, ***kwargs*)

Return Pfam search results in a dictionary. Matching Pfam accession as keys will map to evalue, alignment start and end residue positions.

Parameters

- **query** (*str*[157]) – UniProt ID, PDB identifier, protein sequence, or a sequence file, sequence queries must not contain without gaps and must be at least 16 characters long
- **search_b** (*bool*[158]) – search Pfam-B families when **True**
- **skip_a** (*bool*[159]) – do not search Pfam-A families when **True**
- **ga** (*bool*[160]) – use gathering threshold when **True**
- **evalue** (*float*[161]) – user specified e-value cutoff, must be smaller than 10.0
- **timeout** (*int*[162]) – timeout for blocking connection attempt in seconds, default is 60

query can also be a PDB identifier, e.g. `'1mkp'` or `'1mkpA'` with chain identifier. UniProt ID of the specified chain, or the first protein chain will be used for searching the Pfam database.

fetchPfamMSA(*acc*, *alignment='full'*, *compressed=False*, ***kwargs*)

Return a path to the downloaded Pfam MSA file.

Parameters

- **acc** (*str*[163]) – Pfam ID or Accession Code
- **alignment** – alignment type, one of `'full'` (default), `'seed'`, `'ncbi'`, `'metagenomics'`,`'rp15'`,`'rp35'`,`'rp55'`,or `'rp75'` where rp stands for representative proteomes
- **compressed** – gzip the downloaded MSA file, default is **False**

Alignment Options

Parameters

- **format** – a Pfam supported MSA file format, one of `'selex'`, (default), `'stockholm'` or `'fasta'`
- **order** – ordering of sequences, `'tree'` (default) or `'alphabetical'`
- **inserts** – letter case for inserts, `'upper'` (default) or `'lower'`
- **gaps** – gap character, one of `'dashes'` (default), `'dots'`,`'mixed'` or **None** for unaligned

Other Options

Parameters

- **timeout** – timeout for blocking connection attempt in seconds, default is 60

[157] http://docs.python.org/library/functions.html#str
[158] http://docs.python.org/library/functions.html#bool
[159] http://docs.python.org/library/functions.html#bool
[160] http://docs.python.org/library/functions.html#bool
[161] http://docs.python.org/library/functions.html#float
[162] http://docs.python.org/library/functions.html#int
[163] http://docs.python.org/library/functions.html#str

- **outname** – out filename, default is input `'acc_alignment.format'`
- **folder** – output folder, default is `'.'`

3.3 Dynamics Analysis

This module defines classes and functions for protein dynamics analysis.

3.3.1 Dynamics Models

Following classes are designed for modeling and analysis of protein dynamics:

- `ANM` (page **??**) - Anisotropic network model, for coarse-grained NMA
- `GNM` (page **??**) - Gaussian network model, for coarse-grained dynamics analysis
- `PCA` (page **??**) - Principal component analysis of conformation ensembles
- `EDA` (page **??**) - Essential dynamics analysis of dynamics trajectories
- `NMA` (page **??**) - Normal mode analysis, for analyzing data from external programs
- `RTB` (page **??**) - Rotations and Translation of Blocks method

Usage of these classes are shown in *Anisotropic Network Model (ANM)*[164], *Gaussian Network Model (GNM)*[165], *Ensemble Analysis*[166], and *Essential Dynamics Analysis*[167] examples.

Following classes are for analysis of individual modes or subsets of modes:

- `Mode` (page **??**) - analyze individual normal/principal/essential modes
- `ModeSet` (page **??**) - analyze subset of modes from a dynamics model
- `Vector` (page **??**) - analyze modified modes or deformation vectors

3.3.2 Customize ENMs

Following classes allow for using structure or distance based, or other custom force constants and cutoff distances in `ANM` (page **??**) and `GNM` (page **??**) calculations:

- `Gamma` (page **??**) - base class for developing property custom force constant calculation methods
- `GammaStructureBased` (page **??**) - secondary structure based force constants
- `GammaVariableCutoff` (page **??**) - atom type based variable cutoff function

3.3.3 Function library

Dynamics of the functions described below 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 **??**), `GNM` (page **??**), `NMA` (page **??**), `PCA` (page **??**), or `EDA` (page **??**)
- a `Mode` (page **??**) obtained by indexing an NMA model, e.g. `anm[0]`

[164] http://prody.csb.pitt.edu/tutorials/enm_analysis/anm.html#anm
[165] http://prody.csb.pitt.edu/tutorials/enm_analysis/gnm.html#gnm
[166] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#pca
[167] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda

- a `ModeSet` (page **??**) 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.

3.3.4 Analyze models

Following functions are for calculating atomic properties from normal modes:

- `calcCollectivity()` (page **??**) - degree of collectivity of a mode
- `calcCovariance()` (page **??**) - covariance matrix for given modes
- `calcCrossCorr()` (page **??**) - cross-correlations of fluctuations
- `calcFractVariance()` (page **??**) - fraction of variance explained by a mode
- `calcPerturbResponse()` (page **??**) - response to perturbations in positions
- `calcProjection()` (page **??**) - projection of conformations onto modes
- `calcSqFlucts()` (page **??**) - square-fluctuations
- `calcTempFactors()` (page **??**) - temperature factors fitted to exp. data

3.3.5 Compare models

Following functions are for comparing normal modes or dynamics models:

- `calcOverlap()` (page **??**) - overlap (correlation) between modes
- `calcCumulOverlap()` (page **??**) - cumulative overlap between modes
- `calcSubspaceOverlap()` (page **??**) - overlap between normal mode subspaces
- `calcCovOverlap()` (page **??**) - covariance overlap between models
- `printOverlapTable()` (page **??**) - formatted overlap table printed on screen

3.3.6 Generate conformers

Following functions can be used to generate conformers along normal modes:

- `deformAtoms()` (page **??**) - deform atoms along a mode
- `sampleModes()` (page **??**) - deform along random combination of a set of modes
- `traverseMode()` (page **??**) - traverse a mode along both directions

3.3.7 Editing models

Following functions can be used to reduce, slice, or extrapolate models:

- `sliceMode()` (page **??**) - take a slice of the normal mode
- `extendMode()` (page **??**) - extend a coarse-grained mode to all-atoms
- `sliceModel()` (page **??**) - take a slice of a model
- `extendModel()` (page **??**) - extend a coarse-grained model to all-atoms

- `reduceModel()` (page **??**) - reduce a model to a subset of atoms
- `sliceVector()` (page **??**) - take a slice of a vector
- `extendVector()` (page **??**) - extend a coarse-grained vector to all-atoms

3.3.8 Parse/write data

Following functions are parsing or writing normal mode data:

- `parseArray()` (page **??**) - numeric arrays, e.g. coordinates, eigenvectors
- `parseModes()` (page **??**) - normal modes
- `parseNMD()` (page **??**) - normal mode, coordinate, and atomic data for NMWiz
- `parseSparseMatrix()` (page **??**) - matrix data in sparse coordinate list format
- `writeArray()` (page **??**) - numeric arrays, e.g. coordinates, eigenvectors
- `writeModes()` (page **??**) - normal modes
- `writeNMD()` (page **??**) - normal mode, coordinate, and atomic data
- `writeOverlapTable()` (page **??**) - overlap between modes in a formatted table

3.3.9 Save/load models

Dynamics objects can be efficiently saved and loaded in later Python sessions using the following functions:

- `loadModel()` (page **??**), `saveModel()` (page **??**) - load/save dynamics models
- `loadVector()` (page **??**), `saveVector()` (page **??**) - load/save modes or vectors

3.3.10 Short-hand functions

Following allow for performing some dynamics calculations in one function call:

- `calcANM()` (page **??**) - perform ANM calculations
- `calcGNM()` (page **??**) - perform GNM calculations

3.3.11 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`[168] function utilized for actual plotting. Arguments and keyword arguments are passed to the Matplotlib functions.

- `showMode()` (page **??**) - mode shape
- `showOverlap()` (page **??**) - overlap between modes
- `showSqFlucts()` (page **??**) - square-fluctuations
- `showEllipsoid()` (page **??**) - depict projection of a normal mode space on another
- `showContactMap()` (page **??**) - contact map based on a Kirchhoff matrix
- `showProjection()` (page **??**) - projection of conformations onto normal modes

[168] http://matplotlib.sourceforge.net/api/pyplot_api.html#module-matplotlib.pyplot

- `showOverlapTable()` (page **??**) - overlaps between two models
- `showScaledSqFlucts()` (page **??**) - square-fluctuations fitted to experimental data
- `showNormedSqFlucts()` (page **??**) - normalized square-fluctuations
- `showCrossProjection()` (page **??**) - project conformations onto modes from different models
- `showCrossCorr()` (page **??**) - cross-correlations between fluctuations in atomic positions
- `showCumulOverlap()` (page **??**) - cumulative overlap of a mode with multiple modes from another model
- `showFractVars()` (page **??**) - fraction of variances
- `showCumulFractVars()` (page **??**) - cumulative fraction of variances
- `resetTicks()` (page **??**) - change ticks in a plot

3.3.12 Heat Mapper support

Following functions can be used to read, write, and plot VMD plugin Heat Mapper[169] files.

- `showHeatmap()` (page **??**)
- `parseHeatmap()` (page **??**)
- `writeHeatmap()` (page **??**)

3.3.13 Visualize modes

Finally, normal modes can be visualized and animated using VMD plugin *Normal Mode Wizard*[170]. Following functions allow for running NMWiz from within Python:

- `viewNMDinVMD()` (page **??**) - run VMD and load normal mode data
- `pathVMD()` (page **??**) - get/set path to VMD executable

3.3.14 Analysis Functions

This module defines functions for calculating atomic properties from normal modes.

calcCollectivity(*mode*, *masses=None*)
Return collectivity of the mode. This function implements collectivity as defined in equation 5 of [BR95] (page **??**). If *masses* are provided, they will be incorporated in the calculation. Otherwise, atoms are assumed to have uniform masses.

Parameters

- **mode** (`Mode` (page **??**) or `Vector` (page **??**)) – mode or vector
- **masses** (`numpy.ndarray`[171]) – atomic masses

calcCovariance(*modes*)
Return covariance matrix calculated for given *modes*.

[169] http://www.ks.uiuc.edu/Research/vmd/plugins/heatmapper/
[170] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz
[171] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

calcCrossCorr(*modes*, *n_cpu=1*)
Return 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. Covariance 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*)
Return 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*)
Return 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 **??**)) along the mode. For `PCA` (page **??**) and `EDA` (page **??**) models built using coordinate data in Å, unit of square-fluctuations is Å^2, for `ANM` (page **??**) and `GNM` (page **??**), on the other hand, it is arbitrary or relative units.

calcTempFactors(*modes*, *atoms*)
Return temperature (β) factors calculated using *modes* from a `ANM` (page **??**) or `GNM` (page **??**) instance scaled according to the experimental β-factors from *atoms*.

calcProjection(*ensemble*, *modes*, *rmsd=True*)
Return 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 **??**), `Conformation` (page **??**), `Vector` (page **??**), `Trajectory` (page **??**)) – an ensemble, trajectory or a conformation for which deviation(s) will be projected, or a deformation vector
- **modes** (`Mode` (page **??**), `ModeSet` (page **??**), `NMA` (page **??**)) – up to three normal modes

By default root-mean-square deviation (RMSD) along the normal mode is calculated. To calculate the projection pass `rmsd=True`. `Vector` (page **??**) instances are accepted as *ensemble* argument to allow for projecting a deformation vector onto normal modes.

calcCrossProjection(*ensemble*, *mode1*, *mode2*, *scale=None*, ***kwargs*)
Return projection of conformational deviations onto modes from different models.

Parameters

- **ensemble** (`Ensemble` (page **??**)) – ensemble for which deviations will be projected
- **mode1** (`Mode` (page **??**), `Vector` (page **??**)) – normal mode to project conformations onto
- **mode2** (`Mode` (page **??**), `Vector` (page **??**)) – normal mode to project conformations onto
- **scale** – 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

calcPerturbResponse(*model*, *atoms=None*, *repeats=100*)
Return a matrix of profiles from scanning of the response of the structure to random perturbations at specific atom (or node) positions. The function implements the perturbation response scanning (PRS)

method described in [CA09] (page **??**). 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_profile[i,j]` will give the response of residue/node *j* to perturbations in residue/node *i*. PRS is performed using the covariance matrix from *model*, e.t. `ANM` (page **??**) instance. Each residue/node is perturbed *repeats* times with a random unit force vector. When *atoms* instance is given, PRS profile for residues will be added as an attribute which then can be retrieved as `atoms.getData('prs_profile')`. *model* and *atoms* must have the same number of atoms. *atoms* must be an `AtomGroup` (page **??**) instance.

The RPS matrix can be save as follows:

```
prs_matrix = calcPerturbationResponse(p38_anm)
writeArray('prs_matrix.txt', prs_matrix, format='%8.6f', delimiter='      ')
```

3.3.15 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 **??**), [ARA01] (page **??**)).

See a usage example in *Anisotropic Network Model (ANM)*[172].

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`[173]) – a coordinate set or an object with `getCoords` method
- **cutoff** (*float*[174]) – 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*[175]) – elect to use sparse matrices, default is **False**. If Scipy is not found, `ImportError` is raised.
- **kdtree** (*bool*[176]) – 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.

calcModes(*n_modes=20*, *zeros=False*, *turbo=True*)

Calculate normal modes. This method uses `scipy.linalg.eigh()`[177] function to diagonalize the Hessian matrix. When Scipy is not found, `numpy.linalg.eigh()`[178] is used.

[172] http://prody.csb.pitt.edu/tutorials/enm_analysis/anm.html#anm
[173] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[174] http://docs.python.org/library/functions.html#float
[175] http://docs.python.org/library/functions.html#bool
[176] http://docs.python.org/library/functions.html#bool
[177] http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[178] http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eigh.html#numpy.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 `False`) – 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()
Return a copy of eigenvectors array.

getCovariance()
Return covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getCutoff()
Return cutoff distance.

getEigvals()
Return eigenvalues. For `PCA` (page **??**) and `EDA` (page **??**) models built using coordinate data in Å, unit of eigenvalues is Å^2. For `ANM` (page **??**), `GNM` (page **??**), and `RTB` (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Return a copy of eigenvectors array.

getGamma()
Return spring constant (or the gamma function or `Gamma` instance).

getHessian()
Return a copy of the Hessian matrix.

getKirchhoff()
Return a copy of the Kirchhoff matrix.

getModel()
Return self.

getTitle()
Return title of the model.

getVariances()
Return variances. For `PCA` (page **??**) and `EDA` (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For `ANM` (page **??**), `GNM` (page **??**), and `RTB` (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Return **True** if model is 3-dimensional.

numAtoms()
Return number of atoms.

numDOF()
Return number of degrees of freedom.

numModes()
Return 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.

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.

calcANM(*pdb*, *selstr='calpha'*, *cutoff=15.0*, *gamma=1.0*, *n_modes=20*, *zeros=False*)
Return an `ANM` (page **??**) 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 **??**) instance.

3.3.16 Comparison Functions

This module defines functions for comparing normal modes from different models.

calcOverlap(*rows*, *cols*)
Return 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.

calcCumulOverlap(*modes1*, *modes2*, *array=False*)
Return cumulative overlap of modes in *modes2* with those in *modes1*. Returns a number of *modes1* contains a single `Mode` (page **??**) or a `Vector` (page **??**) 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**, Return 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 upto that many number of modes from *modes2*.

calcSubspaceOverlap(*modes1*, *modes2*)
Return 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 **??**). This function returns a single number.

calcCovOverlap(*modelA*, *modelB*)
Return overlap between covariances of *modelA* and *modelB*. 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 **??**).

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 **??**).

3.3.17 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 **??**), `GNM` (page **??**), `PCA` (page **??**), or `NMA` (page **??**) 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*)
Return part of the *mode* for *atoms* matching *select*. This works slightly different from `sliceVector()` (page **??**). 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 **??**) instance is not normalized.

Parameters

- **mode** (`Mode` (page **??**)) – mode instance to be sliced
- **atoms** (`Atomic` (page **??**)) – atoms for which *mode* describes a deformation, motion, etc.
- **select** (`Selection` (page **??**), str) – an atom selection or a selection string

Returns (`Vector` (page **??**), `Selection` (page **??**))

sliceModel(*model*, *atoms*, *select*)
Return a part of the *model* for *atoms* matching *select*. Note that normal modes (eigenvectors) are not normalized.

Parameters

- **mode** (`NMA` (page **??**)) – NMA model instance to be sliced
- **atoms** (`Atomic` (page **??**)) – atoms for which the *model* was built
- **select** (`Selection` (page **??**), str) – an atom selection or a selection string

Returns (`NMA` (page **??**), `Selection` (page **??**))

sliceVector(*vector*, *atoms*, *select*)
Return part of the *vector* for *atoms* matching *select*. Note that returned `Vector` (page **??**) instance is not normalized.

Parameters

- **vector** (VectorBase) – vector instance to be sliced
- **atoms** (Atomic (page **??**)) – atoms for which *vector* describes a deformation, motion, etc.
- **select** (Selection (page **??**), str) – an atom selection or a selection string

Returns (Vector (page **??**), Selection (page **??**))

reduceModel(*model*, *atoms*, *select*)

Return reduced NMA model. Reduces a NMA (page **??**) model to a subset of *atoms* matching *select*. This function behaves differently depending on the type of the *model* argument. For ANM (page **??**) and GNM (page **??**) or other NMA (page **??**) 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 potential energy. This is based on the formulation in [KH00] (page **??**). For PCA (page **??**) models, this function simply takes the sub-covariance matrix for selection.

Parameters

- **model** (ANM (page **??**), GNM (page **??**), or PCA (page **??**)) – dynamics model
- **atoms** (Atomic (page **??**)) – atoms that were used to build the model
- **select** (Selection (page **??**), str) – an atom selection or a selection string

Returns (NMA (page **??**), Selection (page **??**))

3.3.18 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()[179] 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*[180]) – The string used to separate values. By default, this is any whitespace.
- **skiprows** (*int*[181]) – Skip the first *skiprows* lines, default is 0.
- **usecols** (*list*[182]) – 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[183].) – Data-type of the resulting array, default is float()[184].

[179] http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html#numpy.loadtxt
[180] http://docs.python.org/library/functions.html#str
[181] http://docs.python.org/library/functions.html#int
[182] http://docs.python.org/library/functions.html#list
[183] http://docs.scipy.org/doc/numpy/reference/generated/numpy.dtype.html#numpy.dtype
[184] http://docs.python.org/library/functions.html#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*)

Return `NMA` (page **??**) 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.
- **nm_delimiter** (*str*[185]) – 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*[186]) – 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*[187]) – 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*[188]) – 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*[189]) – 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 **??**) 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 **??**) 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.

[185] http://docs.python.org/library/functions.html#str
[186] http://docs.python.org/library/functions.html#list
[187] http://docs.python.org/library/functions.html#str
[188] http://docs.python.org/library/functions.html#list
[189] http://docs.python.org/library/functions.html#list

- **symmetric** (*bool*[190]) – Set `True` if the file contains triangular part of a symmetric matrix, default is `False`.
- **delimiter** (*str*[191]) – The string used to separate values. By default, this is any whitespace.
- **skiprows** (*int*[192]) – Skip the first *skiprows* lines, default is `0`.
- **irow** (*int*[193]) – Index of the column in data file corresponding to row indices, default is `0`.
- **icol** (*int*[194]) – Index of the column in data file corresponding to row indices, default is `0`.
- **first** (*int*[195]) – First index in the data file (0 or 1), default is `1`.

Data-type of the resulting array, default is `float()`[196].

writeArray(*filename*, *array*, *format='%d'*, *delimiter=' '*)
Write 1-d or 2-d array data into a delimited text file.

This function is using `numpy.savetxt()`[197] 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 **??**).

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 `numpy.savez()`[198] function.

loadModel(*filename*)
Return NMA instance after loading it from file (*filename*). This function makes use of `numpy.load()`[199] function. See also `saveModel()` (page **??**).

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()`[200] function.

loadVector(*filename*)
Return `Vector` (page **??**) instance after loading it from *filename* using `numpy.load()`[201]. See also `saveVector()` (page **??**).

[190] http://docs.python.org/library/functions.html#bool
[191] http://docs.python.org/library/functions.html#str
[192] http://docs.python.org/library/functions.html#int
[193] http://docs.python.org/library/functions.html#int
[194] http://docs.python.org/library/functions.html#int
[195] http://docs.python.org/library/functions.html#int
[196] http://docs.python.org/library/functions.html#float
[197] http://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html#numpy.savetxt
[198] http://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html#numpy.savez
[199] http://docs.scipy.org/doc/numpy/reference/generated/numpy.load.html#numpy.load
[200] http://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html#numpy.savez
[201] http://docs.scipy.org/doc/numpy/reference/generated/numpy.load.html#numpy.load

3.3.19 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 **??**)
- `GammaVariableCutoff` (page **??**)

gamma (*dist2*, *i*, *j*)

Return 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 recent systematic study [LT10] (page **??**) of a large set of NMR-structures analyzed using a method based on entropy maximization showed that taking into consideration properties such as sequential separation 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: 1aar_A_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 **??**)) – A set of atoms with chain identifiers, residue numbers, and secondary structure assignments are set.
- **gamma** (*float*[202]) – Force constant in arbitrary units. Default is 1.0.
- **helix** (*float*[203]) – 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*[204]) – 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*[205]) – Force constant factor for residue pairs that are connected. Default is 10.0, i.e. `10.0`*gamma.

gamma(*dist2*, *i*, *j*)
Return force constant.

getChids()
Return a copy of chain identifiers.

getResnums()
Return a copy of residue numbers.

getSecstrs()
Return 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:

[202] http://docs.python.org/library/functions.html#float
[203] http://docs.python.org/library/functions.html#float
[204] http://docs.python.org/library/functions.html#float
[205] http://docs.python.org/library/functions.html#float

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', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA',
       'CA', 'CA', '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,    7.5,    7.5,    7.5,    7.5,    7.5,    7.5,    7.5,    7.5,
        7.5,    7.5,    7.5,    7.5,    7.5,    7.5,    7.5,    7.5,    7.5,
        7.5,    7.5,    7.5,    7.5,    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 **??**) 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 **??**) 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 **??**) method returns `0`. This annuls the interaction between those atom pairs.

Set the radii of atoms.

Parameters

- **identifiers** (list or `numpy.ndarray`[206]) – List of atom names or types, or residue names.

[206] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

- **gamma** (*float*[207]) – Uniform force constant value. Default is 1.0.
- **default_radius** (*float*[208]) – 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`.

gamma(*dist2*, *i*, *j*)
Return force constant.

getGamma()
Return the uniform force constant value.

getRadii()
Return a copy of radii array.

3.3.20 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 **??**), [TH97] (page **??**)).

See example *Gaussian Network Model (GNM)*[209].

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`[210] or `Atomic` (page **??**)) – a coordinate set or an object with `getCoords` method
- **cutoff** (*float*[211]) – cutoff distance (Å) for pairwise interactions default is 10.0 Å, , minimum is 4.0 Å
- **gamma** (*float*[212]) – spring constant, default is 1.0
- **sparse** (*bool*[213]) – elect to use sparse matrices, default is **False**. If Scipy is not found, `ImportError` is raised.
- **kdtree** (*bool*[214]) – 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.

[207] http://docs.python.org/library/functions.html#float
[208] http://docs.python.org/library/functions.html#float
[209] http://prody.csb.pitt.edu/tutorials/enm_analysis/gnm.html#gnm
[210] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[211] http://docs.python.org/library/functions.html#float
[212] http://docs.python.org/library/functions.html#float
[213] http://docs.python.org/library/functions.html#bool
[214] http://docs.python.org/library/functions.html#bool

calcModes(*n_modes=20*, *zeros=False*, *turbo=True*)
Calculate normal modes. This method uses `scipy.linalg.eigh()`[215] function to diagonalize the Kirchhoff matrix. When Scipy is not found, `numpy.linalg.eigh()`[216] 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 `False`) – 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()
Return a copy of eigenvectors array.

getCovariance()
Return covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getCutoff()
Return cutoff distance.

getEigvals()
Return eigenvalues. For `PCA` (page **??**) and `EDA` (page **??**) models built using coordinate data in Å, unit of eigenvalues is Å^2. For `ANM` (page **??**), `GNM` (page **??**), and `RTB` (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Return a copy of eigenvectors array.

getGamma()
Return spring constant (or the gamma function or `Gamma` instance).

getKirchhoff()
Return a copy of the Kirchhoff matrix.

getModel()
Return self.

getTitle()
Return title of the model.

getVariances()
Return variances. For `PCA` (page **??**) and `EDA` (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For `ANM` (page **??**), `GNM` (page **??**), and `RTB` (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Return **True** if model is 3-dimensional.

numAtoms()
Return number of atoms.

numDOF()
Return number of degrees of freedom.

numModes()
Return number of modes in the instance (not necessarily maximum number of possible modes).

[215] http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[216] http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eigh.html#numpy.linalg.eigh

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.

setKirchhoff(*kirchhoff*)
Set Kirchhoff matrix.

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

calcGNM(*pdb*, *selstr='calpha'*, *cutoff=15.0*, *gamma=1.0*, *n_modes=20*, *zeros=False*)
Return a `GNM` (page **??**) 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 **??**) instance.

3.3.21 Heatmapper Functions

This module defines functions for supporting VMD plugin Heat Mapper[217] format files.

parseHeatmap(*heatmap*, ***kwargs*)
Return 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*)
Return *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*[218]) – title of the heatmap
- **xlabel** (*str*[219]) – x-axis lab, default is `'unknown'`
- **ylabel** (*str*[220]) – y-axis lab, default is `'unknown'`
- **xorigin** (*float*[221]) – x-axis origin, default is 0
- **xstep** (*float*[222]) – x-axis step, default is 1
- **min** (*float*[223]) – minimum value, default is minimum in *heatmap*
- **max** (*float*[224]) – maximum value, default is maximum in *heatmap*
- **format** (*str*[225]) – 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()`[226] function. Default values passed to this function are `interpolation='nearest'`, `aspect='auto'`, and `origin='lower'`.

[217] http://www.ks.uiuc.edu/Research/vmd/plugins/heatmapper/
[218] http://docs.python.org/library/functions.html#str
[219] http://docs.python.org/library/functions.html#str
[220] http://docs.python.org/library/functions.html#str
[221] http://docs.python.org/library/functions.html#float
[222] http://docs.python.org/library/functions.html#float
[223] http://docs.python.org/library/functions.html#float
[224] http://docs.python.org/library/functions.html#float
[225] http://docs.python.org/library/functions.html#str
[226] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.imshow

3.3.22 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 **??**), GNM (page **??**), or PCA (page **??**)) – a normal mode analysis instance
- **index** (*int*[227]) – index of the mode

getArray()

Return a copy of the normal mode array (eigenvector).

getArrayNx3()

Return a copy of array with shape (N, 3).

getEigval()

Return normal mode eigenvalue. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of eigenvalues is Å^2. For ANM (page **??**) and GNM (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvec()

Return a copy of the normal mode array (eigenvector).

getIndex()

Return the index of the mode. Note that mode indices are zero-based.

getModel()

Return the model that the mode instance belongs to.

getTitle()

A descriptive title for the mode instance.

getVariance()

Return variance along the mode. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For ANM (page **??**) and GNM (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()

Return **True** if mode instance is from a 3-dimensional model.

numAtoms()

Return number of atoms.

numDOF()

Return number of degrees of freedom (three times the number of atoms).

numModes()

Return 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 **??**) instance or addition of two Mode (page **??**) instances results in a Vector (page **??**) instance.

[227] http://docs.python.org/library/functions.html#int

Instantiate with a name, an array, and a 3d flag.

getArray()
Return a copy of array.

getArrayNx3()
Return a copy of array with shape (N, 3).

getNormed()
Return mode after normalizing it.

getTitle()
Get the descriptive title for the vector instance.

is3d()
Return **True** if vector instance describes a 3-dimensional property, such as a deformation for a set of atoms.

numAtoms()
Return number of atoms. For a 3-dimensional vector, returns length of the vector divided by 3.

numDOF()
Return number of degrees of freedom.

numModes()
Return 1.

setTitle(*title*)
Set the descriptive title for the vector instance.

3.3.23 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 **??**), GNM (page **??**), or PCA (page **??**)). 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()
Return a copy of eigenvectors array.

getEigvals()
Return eigenvalues. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of eigenvalues is $Å^2$. For ANM (page **??**) and GNM (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Return a copy of eigenvectors array.

getIndices()
Return indices of modes in the mode set.

getModel()
Return the model that the modes belongs to.

getTitle()
Return title of the mode set.

getVariances()
Return variances. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For ANM (page **??**) and GNM (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Return **True** is model is 3-dimensional.

numAtoms()
Return number of atoms.

numDOF()
Return number of degrees of freedom.

numModes()
Return number of modes in the instance (not necessarily maximum number of possible modes).

3.3.24 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()
Return a copy of eigenvectors array.

getCovariance()
Return covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getEigvals()
Return eigenvalues. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of eigenvalues is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Return a copy of eigenvectors array.

getModel()
Return self.

getTitle()
Return title of the model.

getVariances()
Return variances. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Return **True** if model is 3-dimensional.

numAtoms()
Return number of atoms.

numDOF()
Return number of degrees of freedom.

numModes()
Return 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.

3.3.25 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*[228]. ProDy functions `writeNMD()` (page **??**) and `parseNMD()` (page **??**) 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.

[228] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

```
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  ...
```

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=None*)
Return `NMA` (page **??**) and `AtomGroup` (page **??**) instances storing data parsed from *filename* in `.nmd` format. Type of `NMA` (page **??**) instance, e.g. `PCA` (page **??**), `ANM` (page **??**), or `GNM` (page **??**) will be determined based on mode data.

writeNMD(*filename*, *modes*, *atoms*)
Return *filename* that contains *modes* and *atoms* data in NMD format described in *NMD Format* (page **??**). `.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 **??**) 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*)
Return VMD path, or set it to be a user specified *path*.

getVMDpath()
Deprecated for removal in v1.5, use `pathVMD()` (page **??**) instead.

setVMDpath(*path*)
Deprecated for removal in v1.5, use `pathVMD()` (page **??**) instead.

viewNMDinVMD(*filename*)
Start VMD in the current Python session and load NMD data.

3.3.26 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*[229].

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 **??**)
- `Ensemble` (page **??**)
- `TrajBase` (page **??**)
- `numpy.ndarray`[230] 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 **??**), covariance will be built by superposing frames onto the reference coordinate set (see `Frame.superpose()` (page **??**)). If frames are already aligned, use `aligned=True` argument to skip this step.

Note: If *coordsets* is a `PDBEnsemble` (page **??**) 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*)
Calculate principal (or essential) modes. This method uses `scipy.linalg.eigh()`[231], or `numpy.linalg.eigh()`[232], function to diagonalize the covariance matrix.

Parameters

- **n_modes** (*int*[233]) – number of non-zero eigenvalues/vectors to calculate, default is 20, for **None** all modes will be calculated

[229] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#pca
[230] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[231] http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[232] http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eigh.html#numpy.linalg.eigh
[233] http://docs.python.org/library/functions.html#int

- **turbo** (*bool*[234]) – when available, use a memory intensive but faster way to calculate modes, default is **True**

getArray()
Return a copy of eigenvectors array.

getCovariance()
Return covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getEigvals()
Return eigenvalues. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of eigenvalues is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Return a copy of eigenvectors array.

getModel()
Return self.

getTitle()
Return title of the model.

getVariances()
Return variances. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Return **True** if model is 3-dimensional.

numAtoms()
Return number of atoms.

numDOF()
Return number of degrees of freedom.

numModes()
Return 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 **??**), Ensemble (page **??**), or numpy.ndarray[235] 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*[236] example for comparison of results from SVD and covariance methods.

setCovariance(*covariance*)
Set covariance matrix.

[234] http://docs.python.org/library/functions.html#bool
[235] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[236] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_calculations.html#pca-xray-calculations

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'*)
A class for Essential Dynamics Analysis (EDA) [AA93] (page **??**). See examples in *Essential Dynamics Analysis*[237].

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 **??**)
- `Ensemble` (page **??**)
- `TrajBase` (page **??**)
- `numpy.ndarray`[238] 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 **??**), covariance will be built by superposing frames onto the reference coordinate set (see `Frame.superpose()` (page **??**)). If frames are already aligned, use `aligned=True` argument to skip this step.

Note: If *coordsets* is a `PDBEnsemble` (page **??**) 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*)
Calculate principal (or essential) modes. This method uses `scipy.linalg.eigh()`[239], or `numpy.linalg.eigh()`[240], function to diagonalize the covariance matrix.

Parameters

- **n_modes** (*int*[241]) – number of non-zero eigenvalues/vectors to calculate, default is 20, for **None** all modes will be calculated
- **turbo** (*bool*[242]) – when available, use a memory intensive but faster way to calculate modes, default is **True**

getArray()
Return a copy of eigenvectors array.

[237] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda
[238] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[239] http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[240] http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eigh.html#numpy.linalg.eigh
[241] http://docs.python.org/library/functions.html#int
[242] http://docs.python.org/library/functions.html#bool

getCovariance()
Return covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

getEigvals()
Return eigenvalues. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of eigenvalues is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Return a copy of eigenvectors array.

getModel()
Return self.

getTitle()
Return title of the model.

getVariances()
Return variances. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Return **True** if model is 3-dimensional.

numAtoms()
Return number of atoms.

numDOF()
Return number of degrees of freedom.

numModes()
Return 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 **??**), Ensemble (page **??**), or numpy.ndarray[243] 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*[244] example for comparison of results from SVD and covariance methods.

setCovariance(*covariance*)
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.

[243] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[244] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_calculations.html#pca-xray-calculations

3.3.27 Plotting Functions

This module defines plotting functions for protein dynamics analysis.

Plotting functions are called by the name of the plotted data/property and are prefixed with `show`. Function documentations refers to the `matplotlib.pyplot`[245] function utilized for actual plotting. Arguments and keyword arguments are passed to the Matplotlib functions.

showContactMap(*enm*, **args*, ***kwargs*)
Show Kirchhoff matrix using `spy()`[246].

showCrossCorr(*modes*, **args*, ***kwargs*)
Show cross-correlations using `imshow()`[247]. By default, *origin=lower* and *interpolation=bilinear* keyword arguments are passed to this function, but user can overwrite these parameters. See also `calcCrossCorr()` (page **??**).

showCumulOverlap(*mode*, *modes*, **args*, ***kwargs*)
Show cumulative overlap using `plot()`[248].

Parameters **modes** (`ModeSet` (page **??**), `ANM` (page **??**), `GNM` (page **??**), `PCA` (page **??**)) – multiple modes

showFractVars(*modes*, **args*, ***kwargs*)
Show fraction of variances using `bar()`[249]. 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 **??**) function.

showMode(*mode*, **args*, ***kwargs*)
Show mode array using `plot()`[250].

showOverlap(*mode*, *modes*, **args*, ***kwargs*)
Show overlap `bar()`[251].

Parameters

- **mode** (`Mode` (page **??**), `Vector` (page **??**)) – a single mode/vector
- **modes** (`ModeSet` (page **??**), `ANM` (page **??**), `GNM` (page **??**), `PCA` (page **??**)) – multiple modes

showOverlapTable(*modes_x*, *modes_y*, ***kwargs*)
Show overlap table using `pcolor()`[252]. *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()`[253]:

- `cmap=plt.cm.jet`
- `norm=plt.normalize(0, 1)`

[245] http://matplotlib.sourceforge.net/api/pyplot_api.html#module-matplotlib.pyplot
[246] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.spy
[247] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.imshow
[248] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
[249] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.bar
[250] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
[251] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.bar
[252] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.pcolor
[253] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.pcolor

showProjection(*ensemble*, *modes*, **args*, ***kwargs*)

Show a projection of conformational deviations onto up to three normal modes from the same model.

Parameters

- **ensemble** (`Ensemble` (page **??**), `Conformation` (page **??**), `Vector` (page **??**), `Trajectory` (page **??**)) – an ensemble, trajectory or a conformation for which deviation(s) will be projected, or a deformation vector
- **modes** (`Mode` (page **??**), `ModeSet` (page **??**), `NMA` (page **??**)) – up to three normal modes
- **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*[254]) – line style, default is `'None'`
- **text** (*list*[255]) – list of text labels, one for each conformation
- **fontsize** (*int*[256]) – 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:

- 1 mode: `hist()`[257]
- 2 modes: `plot()`[258]
- 3 modes: `plot()`[259]

showCrossProjection(*ensemble*, *mode_x*, *mode_y*, *scale=None*, **args*, ***kwargs*)

Show a projection of conformational deviations onto modes from different models using `plot()`[260]. This function differs from `showProjection()` (page **??**) by accepting modes from two different models.

Parameters

- **ensemble** (`Ensemble` (page **??**), `Conformation` (page **??**), `Vector` (page **??**), `Trajectory` (page **??**)) – an ensemble or a conformation for which deviation(s) will be projected, or a deformation vector
- **mode_x** (`Mode` (page **??**), `Vector` (page **??**)) – projection onto this mode will be shown along x-axis
- **mode_y** (`Mode` (page **??**), `Vector` (page **??**)) – projection onto this mode will be shown along y-axis
- **scale** (*str*[261]) – 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

[254] http://docs.python.org/library/functions.html#str
[255] http://docs.python.org/library/functions.html#list
[256] http://docs.python.org/library/functions.html#int
[257] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.hist
[258] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
[259] http://matplotlib.sourceforge.net/mpl_toolkits/mplot3d/tutorial.html#mpl_toolkits.mplot3d.Axes3D.plot
[260] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
[261] http://docs.python.org/library/functions.html#str

- **scalar** (*float*[262]) – 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*[263]) – line style, default is `'None'`
- **text** (*list*[264]) – list of text labels, one for each conformation
- **fontsize** (*int*[265]) – font size for text labels

The projected values are by default converted to RMSD. Pass `rmsd=False` to calculate raw projection values. See *Plotting*[266] for a more elaborate example.

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

- **modes** (`ModeSet` (page **??**), `PCA` (page **??**), `ANM` (page **??**), `NMA` (page **??**)) – 3 modes for which ellipsoid will be drawn.
- **onto** – 3 modes onto which ellipsoid will be projected.
- **n_std** (*float*[267]) – Number of standard deviations to scale the ellipsoid.
- **scale** (*float*[268]) – Used for scaling the volume of ellipsoid. This can be obtained from `sampleModes()` (page **??**).

showSqFlucts(*modes*, **args*, ***kwargs*)
Show square fluctuations using `plot()`[269]. See also `calcSqFlucts()` (page **??**).

showScaledSqFlucts(*modes*, **args*, ***kwargs*)
Show scaled square fluctuations using `plot()`[270]. 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()`[271].

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 `imshow()`[272]. When `NMA` (page **??**) models are passed instead of matrices, the functions could call `calcCrossCorr()` (page **??**) function to calculate the matrices for given modes.

[262] http://docs.python.org/library/functions.html#float
[263] http://docs.python.org/library/functions.html#str
[264] http://docs.python.org/library/functions.html#list
[265] http://docs.python.org/library/functions.html#int
[266] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_plotting.html#pca-xray-plotting
[267] http://docs.python.org/library/functions.html#float
[268] http://docs.python.org/library/functions.html#float
[269] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
[270] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
[271] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
[272] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.imshow

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.

3.3.28 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 **??**)). Optional arguments permit imposing constrains along Z-direction as in *imANM* method described in [TL12] (page **??**).

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`[273]) – a coordinate set or an object with `getCoords` method
- **blocks** (list, `numpy.ndarray`[274]) – a list or array of block identifiers
- **cutoff** (*float*[275]) – cutoff distance (Å) for pairwise interactions, default is 15.0 Å
- **gamma** (*float*[276]) – spring constant, default is 1.0
- **scale** (*float*[277]) – scaling factor for force constant along Z-direction, default is 1.0

calcModes(*n_modes=20*, *zeros=False*, *turbo=True*)

Calculate normal modes. This method uses `scipy.linalg.eigh()`[278] function to diagonalize the Hessian matrix. When Scipy is not found, `numpy.linalg.eigh()`[279] 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 `False`) – 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()

Return a copy of eigenvectors array.

getCovariance()

Return covariance matrix. If covariance matrix is not set or yet calculated, it will be calculated using available modes.

[273] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[274] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[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.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh
[279] http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eigh.html#numpy.linalg.eigh

getEigvals()
Return eigenvalues. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of eigenvalues is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, eigenvalues are in arbitrary or relative units but they correlate with stiffness of the motion along associated eigenvector.

getEigvecs()
Return a copy of eigenvectors array.

getHessian()
Return a copy of the Hessian matrix.

getModel()
Return self.

getProjection()
Return a copy of the projection matrix.

getTitle()
Return title of the model.

getVariances()
Return variances. For PCA (page **??**) and EDA (page **??**) models built using coordinate data in Å, unit of variance is Å^2. For ANM (page **??**), GNM (page **??**), and RTB (page **??**), on the other hand, variance is the inverse of the eigenvalue, so it has arbitrary or relative units.

is3d()
Return **True** if model is 3-dimensional.

numAtoms()
Return number of atoms.

numDOF()
Return number of degrees of freedom.

numModes()
Return 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.

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.

3.3.29 Sampling Functions

This module defines functions for generating alternate conformations along normal modes.

deformAtoms(*atoms*, *mode*, *rmsd=None*)
Generate a new coordinate set for *atoms* along the *mode*. *atoms* must be a AtomGroup (page **??**) 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*)
Return 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 **??**), `ModeSet` (page **??**), `PCA` (page **??**), `ANM` (page **??**) or `NMA` (page **??**)) – modes along which sampling will be performed
- **atoms** (`Atomic` (page **??**)) – atoms whose active coordinate set will be used as the initial conformation
- **n_confs** – number of conformations to generate, default is 1000
- **rmsd** (*float*[280]) – average RMSD that the conformations will have with respect to the initial conformation, default is 1.0 Å

Returns `Ensemble` (page **??**)

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()`[281].

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 **??**), variances are used instead of inverse eigenvalues, i.e. $\sigma_i \sim \lambda_i^{-1}$.

See also `showEllipsoid()` (page **??**).

traverseMode(*mode*, *atoms*, *n_steps=10*, *rmsd=1.5*)

Generates a trajectory along a given *mode*, which can be used to animate fluctuations in an external program.

Parameters

- **mode** (`Mode` (page **??**)) – mode along which a trajectory will be generated
- **atoms** (`Atomic` (page **??**)) – atoms whose active coordinate set will be used as the initial conformation

[280] http://docs.python.org/library/functions.html#float
[281] http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html#numpy.random.randn

- **n_steps** (*int*[282]) – number of steps to take along each direction, for example, for `n_steps=10`, 20 conformations will be generated along the first mode, default is 10.
- **rmsd** (*float*[283]) – maximum RMSD that the conformations will have with respect to the initial conformation, default is 1.5 Å

Returns `Ensemble` (page **??**)

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.

3.4 Ensemble Analysis

This module defines classes for handling conformational ensembles.

3.4.1 Conformational ensembles

The following two classes are implemented for handling arbitrary but uniform conformational ensembles, e.g. NMR models, MD snapshots:

- `Ensemble` (page **??**)
- `Conformation` (page **??**)

See usage examples in *NMR Models*[284] and *Essential Dynamics Analysis*[285].

3.4.2 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 **??**)
- `PDBConformation` (page **??**)

Following functions are for editing PDB ensembles, e.g. finding and removing residues that are missing in too many structures:

- `alignPDBEnsemble()` (page **??**)
- `calcOccupancies()` (page **??**)
- `showOccupancies()` (page **??**)
- `trimPDBEnsemble()` (page **??**)

See usage examples in *Heterogeneous X-ray Structures*[286], *Multimeric Structures*[287], *Homologous Proteins*[288].

[282] http://docs.python.org/library/functions.html#int
[283] http://docs.python.org/library/functions.html#float
[284] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/nmr.html#pca-nmr
[285] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda
[286] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray.html#pca-xray
[287] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/dimer.html#pca-dimer
[288] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/blast.html#pca-blast

3.4.3 Save/load ensembles

- `saveEnsemble()` (page **??**)
- `loadEnsemble()` (page **??**)

3.4.4 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()

Return associated atom group.

getCoords()

Return 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.

getDeviations()

Return deviations from the ensemble reference coordinates. Deviations are calculated for (selected) atoms.

getEnsemble()

Return the ensemble that this conformation belongs to.

getIndex()

Return conformation index.

getRMSD()

Return RMSD from the ensemble reference coordinates. RMSD is calculated for (selected) atoms.

getWeights()

Return coordinate weights for (selected) atoms.

numAtoms()

Return number of atoms.

numSelected()

Return number of selected atoms.

class **PDBConformation**(*ensemble*, *index*)

This class is the same as `Conformation` (page **??**), except that the conformation has a name (or identifier), e.g. PDB identifier.

getAtoms()

Return associated atom group.

getCoords()

Return 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()

Return deviations from the ensemble reference coordinates. Deviations are calculated for (selected) atoms.

getEnsemble()
Return the ensemble that this conformation belongs to.

getIndex()
Return conformation index.

getLabel()
Return the label of the conformation.

getRMSD()
Return RMSD from the ensemble reference coordinates. RMSD is calculated for (selected) atoms.

getTransformation()
Return the `Transformation` (page **??**) used to superpose this conformation onto reference coordinates. The transformation can be used to superpose original PDB file onto the reference PDB file.

getWeights()
Return coordinate weights for (selected) atoms.

numAtoms()
Return number of atoms.

numSelected()
Return number of selected atoms.

setLabel(*label*)
Set the label of the conformation.

3.4.5 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 **??**) instance that points to a coordinate set in the ensemble. Slicing (e.g. `ens[0:10]`) returns an `Ensemble` (page **??**) instance that contains a copy of the subset of conformations (coordinate sets).

Instantiate with a *title* or a `Atomic` (page **??**) instance. All coordinate sets from atomic instances will be added to the ensemble.

addCoordset(*coords*)
Add coordinate set(s) to the ensemble. *coords* must be a Numpy array with suitable data type, shape and dimensionality, or an object with `getCoordsets()` (page **??**) method.

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

getAtoms()
Return associated/selected atoms.

getConformation(*index*)
Return conformation at given index.

getCoords()
Return a copy of reference coordinates for selected atoms.

getCoordsets(*indices=None*)
Return 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 `getCoordinates()` method.

getDeviations()
Return deviations from reference coordinates for selected atoms. Conformations can be aligned using one of `superpose()` (page **??**) or `iterpose()` (page **??**) methods prior to calculating deviations.

getMSFs()
Return mean square fluctuations (MSFs) for selected atoms. Conformations can be aligned using one of `superpose()` (page **??**) or `iterpose()` (page **??**) methods prior to MSF calculation.

getRMSDs()
Return root mean square deviations (RMSDs) for selected atoms. Conformations can be aligned using one of `superpose()` (page **??**) or `iterpose()` (page **??**) methods prior to RMSD calculation.

getRMSFs()
Return root mean square fluctuations (RMSFs) for selected atoms. Conformations can be aligned using one of `superpose()` (page **??**) or `iterpose()` (page **??**) methods prior to RMSF calculation.

getTitle()
Return title of the ensemble.

getWeights()
Return a copy of weights of selected atoms.

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*[289]) – change in reference coordinates to determine convergence, default is 0.0001 Å RMSD

numAtoms()
Return number of atoms.

numConfs()
Return number of conformations.

numCoordsets()
Return number of conformations.

numSelected()
Return 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 **??**).

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,

[289] http://docs.python.org/library/functions.html#float

alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, `Ensemble` (page **??**) and `Conformation` (page **??**) instances become suitable arguments for `writePDB()` (page **??**). 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 **??**) method.

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

setWeights(*weights*)
Set atomic weights.

superpose()
Superpose the ensemble onto the reference coordinates.

3.4.6 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 `numpy.savez()`[290] function.

loadEnsemble(*filename*)
Return ensemble instance loaded from *filename*. This function makes use of `numpy.load()`[291] function. See also `saveEnsemble()` (page **??**)

trimPDBEnsemble(*pdb_ensemble*, ***kwargs*)
Return 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 **??**) instance.

Occupancy

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*[292]) – occupancy for selecting atoms, must satisfy `0 < occupancy <= 1`

calcOccupancies(*pdb_ensemble*, *normed=False*)
Return occupancy calculated from weights of a `PDBEnsemble` (page **??**). 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 **??**).

[290] http://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html#numpy.savez
[291] http://docs.scipy.org/doc/numpy/reference/generated/numpy.load.html#numpy.load
[292] http://docs.python.org/library/functions.html#float

alignPDBEnsemble(*ensemble*, *suffix='_aligned'*, *outdir='.'*, *gzip=False*)

Align PDB files using transformations from *ensemble*, which may be a `PDBEnsemble` (page **??**) or a `PDBConformation` (page **??**) instance. Label of the conformation (see `getLabel()` (page **??**)) will be used to determine the PDB structure and model number. First four characters of the label is expected to be the PDB identifier and ending numbers to be the model number. For example, the `Transformation` (page **??**) from conformation with label *2k39_ca_selection_'resnum_<_71'_m116* will be applied to 116th model of structure **2k39**. After applicable transformations are made, structure will be written into *outputdir* as `2k39_aligned.pdb`. If *gzip* is **True**, output files will be compressed. Return value is the output filename or list of filenames, in the order files are processed. Note that if multiple models from a file are aligned, that filename will appear in the list multiple times.

3.4.7 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*[293], *Multimeric Structures*[294], and *Homologous Proteins*[295].

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*)

Add coordinate set(s) to the ensemble. *coords* must be a Numpy array with suitable shape and dimensionality, or an object with `getCoordsets()` (page **??**) method. *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.

getAtoms()

Return associated/selected atoms.

getConformation(*index*)

Return conformation at given index.

getCoords()

Return a copy of reference coordinates for selected atoms.

getCoordsets(*indices=None*)

Return 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.

[293] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray.html#pca-xray
[294] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/dimer.html#pca-dimer
[295] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/blast.html#pca-blast

getDeviations()
Return deviations from reference coordinates for selected atoms. Conformations can be aligned using one of `superpose()` (page **??**) or `iterpose()` (page **??**) methods prior to calculating deviations.

getLabels()
Return identifiers of the conformations in the ensemble.

getMSFs()
Calculate and return mean square fluctuations (MSFs). Note that you might need to align the conformations using `superpose()` (page **??**) or `iterpose()` (page **??**) before calculating MSFs.

getRMSDs()
Calculate and return root mean square deviations (RMSDs). Note that you might need to align the conformations using `superpose()` (page **??**) or `iterpose()` (page **??**) before calculating RMSDs.

getRMSFs()
Return root mean square fluctuations (RMSFs) for selected atoms. Conformations can be aligned using one of `superpose()` (page **??**) or `iterpose()` (page **??**) methods prior to RMSF calculation.

getTitle()
Return title of the ensemble.

getWeights()
Return a copy of weights of selected atoms.

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*[296]) – change in reference coordinates to determine convergence, default is 0.0001 Å RMSD

numAtoms()
Return number of atoms.

numConfs()
Return number of conformations.

numCoordsets()
Return number of conformations.

numSelected()
Return 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 **??**).

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,

[296] http://docs.python.org/library/functions.html#float

alignments and RMSD calculations. Setting atoms also allows some functions to access atomic data when needed. For example, `Ensemble` (page **??**) and `Conformation` (page **??**) instances become suitable arguments for `writePDB()` (page **??**). 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 **??**) method.

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

setWeights(*weights*)
Set atomic weights.

superpose()
Superpose the ensemble onto the reference coordinates.

3.5 KDTree

This module provides `KDTree` (page **??**) class as an interface to Thomas Hamelryck's KDTree C module distributed with Biopython.

3.5.1 KD Tree

This module defines `KDTree` (page **??**) 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 **??**) method and results are retrieved using `getIndices()` (page **??**) and `getDistances()` (page **??**).

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 **??**) can be used for wrapping atoms into the single periodic image of the system.

Parameters

- **coords** (`numpy.ndarray`[297], `Atomic` (page **??**), `Frame` (page **??**)) – coordinate array with shape `(N, 3)`, where N is number of atoms
- **unitcell** (`numpy.ndarray`[298]) – orthorhombic unitcell dimension array with shape `(3,)`
- **bucketsize** (*int*[299]) – number of points per tree node, default is 10

getCount()
Return number of points or pairs.

getDistances()
Return array of distances.

getIndices()
Return 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*[300]) – distance (Å)
- **center** (`numpy.ndarray`[301]) – a point in Cartesian coordinate system

3.6 Measurement Tools

This module defines classes measuring quantities, transforming coordinates, and identifying contacts.

[297] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[298] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[299] http://docs.python.org/library/functions.html#int
[300] http://docs.python.org/library/functions.html#float
[301] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

3.6.1 Identify contacts

Following class and functions are for contact identifications:

- `Contacts` (page **??**) - identify intermolecular contacts
- `findNeighbors()` (page **??**) - identify interacting atom pairs
- `iterNeighbors()` (page **??**) - identify interacting atom pairs

3.6.2 Measure quantities

Following functions are for measuring simple quantities:

- `calcDistance()` (page **??**) - calculate distance(s)
- `calcAngle()` (page **??**) - calculate bond angle
- `calcDihedral()` (page **??**) - calculate dihedral angle
- `calcOmega()` (page **??**) - calculate omega (ω) angle
- `calcPhi()` (page **??**) - calculate phi (ϕ) angle
- `calcPsi()` (page **??**) - calculate psi (ψ) angle
- `calcGyradius()` (page **??**) - calculate radius of gyration
- `calcCenter()` (page **??**) - calculate geometric (or mass) center
- `calcDeformVector()` (page **??**) - calculate deformation vector

3.6.3 Anisotropic factors

Following functions handle anisotropic displacement parameter (ADP) present in some X-ray structures.

- `buildADPMatrix()` (page **??**) - build ADP matrix
- `calcADPAxes()` (page **??**) - calculate ADP axes
- `calcADPs()` (page **??**) - calculate ADPs

3.6.4 Transformations

Following class and functions are for handling coordinate transformations:

- `Transformation` (page **??**) - store transformation matrix
- `alignCoordsets()` (page **??**) - align multiple coordinate sets
- `applyTransformation()` (page **??**) - apply a transformation
- `calcTransformation()` (page **??**) - calculate a transformation
- `calcRMSD()` (page **??**) - calculate root-mean-square distance
- `superpose()` (page **??**) - superpose atoms or coordinate sets
- `moveAtoms()` (page **??**) - move atoms by given offset

3.6.5 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 **??**) instance. When an orthorhombic *unitcell* array is given

getAtoms()

Return atoms, or coordinate array, provided at instantiation..

getUnitcell()

Return 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`[302] with shape `(n_atoms, 3)`) or a set of atoms, e.g. `Selection` (page **??**).

iterNeighbors(*atoms*, *radius*, *atoms2=None*, *unitcell=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 **??**) and also `wrapAtoms()` for details). If *atoms* is a `Frame` (page **??**) instance and *unitcell* is not provided, unitcell information from frame will be if available.

findNeighbors(*atoms*, *radius*, *atoms2=None*, *unitcell=None*)

Return list of neighbors that are within *radius* of each other and the distance between them. See `iterNeighbors()` (page **??**) for more details.

3.6.6 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'*)

Return 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 **??**), `numpy.ndarray`[303]) – atom or coordinate data
- **atoms2** (`Atomic` (page **??**), `numpy.ndarray`[304]) – atom or coordinate data
- **unitcell** (`numpy.ndarray`[305]) – orthorhombic unitcell dimension array with shape `(3,)`
- **format** (*bool*[306]) – 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)

[302] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[303] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[304] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[305] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[306] http://docs.python.org/library/functions.html#bool

calcDistance(*atoms1*, *atoms2*, *unitcell=None*)
Return the Euclidean distance between *atoms1* and *atoms2*. Arguments may be `Atomic` (page **??**) 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 **??**), `numpy.ndarray`[307]) – atom or coordinate data
- **atoms2** (`Atomic` (page **??**), `numpy.ndarray`[308]) – atom or coordinate data
- **unitcell** (`numpy.ndarray`[309]) – orthorhombic unitcell dimension array with shape `(3,)`

calcCenter(*atoms*, *weights=None*)
Return 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*)
Return the angle between atoms in degrees.

calcDihedral(*atoms1*, *atoms2*, *atoms3*, *atoms4*, *radian=False*)
Return the dihedral angle between atoms in degrees.

calcOmega(*residue*, *radian=False*, *dist=4.1*)
Return 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*)
Return 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.

calcPsi(*residue*, *radian=False*, *dist=4.1*)
Return 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 **??**), `TrajBase` (page **??**), or `Atomic` (page **??**). For trajectory objects, e.g. `DCDFile` (page **??**), 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 **??**) 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*)
Return root mean square fluctuation(s) (RMSF). *coordsets* may be an instance of `Ensemble` (page **??**), `TrajBase` (page **??**), or `Atomic` (page **??**). For trajectory objects, e.g. `DCDFile` (page **??**), frames will be considered after they are superposed. For other ProDy objects, coordinate sets should be aligned prior to MSF calculation.

[307] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[308] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[309] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

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 **??**) 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*)
Return deformation from *from_atoms* to *atoms_to* as a `Vector` (page **??**) instance.

buildADPMatrix (*atoms*)
Return 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)
```

calcADPAxes (*atoms*, ***kwargs*)
Return a 3Nx3 array containing principal axes defining anisotropic displacement parameter (ADP, or anisotropic temperature factor) ellipsoids.

Parameters

- **atoms** (`Atomic` (page **??**)) – a ProDy object for handling atomic data
- **fract** (*float*[310]) – 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*[311]) – 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*[312]) – 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*[313]) – Same as *ratio3*.

Keyword arguments *fract*, *ratio3*, 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')
```

[310] http://docs.python.org/library/functions.html#float
[311] http://docs.python.org/library/functions.html#float
[312] http://docs.python.org/library/functions.html#float
[313] http://docs.python.org/library/functions.html#float

```
In [3]: calphas = protein.select('calpha')

In [4]: adp_axes = calcADPAxes( calphas )

In [5]: adp_axes.shape
 Out[5]: (138, 3)
```

These can be written in NMD format as follows:

```
In [6]: nma = NMA('ADPs')

In [7]: nma.setEigens(adp_axes)

In [8]: nma
 Out[8]: <NMA: ADPs (3 modes; 46 atoms)>

In [9]: writeNMD('adp_axes.nmd', nma, calphas)
 Out[9]: 'adp_axes.nmd'
```

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*)
Return `Atom` (page **??**) or `Conformation` (page **??**) that is closest to the center of *obj*, which may be an `Atomic` (page **??**) or `Ensemble` (page **??**) instance. See also `pickCentralAtom()` (page **??**), and `pickCentralConf()` (page **??**) functions.

pickCentralAtom(*atoms*, *weights=None*)
Return `Atom` (page **??**) that is closest to the center, which is calculated using `calcCenter()` (page **??**).

pickCentralConf(*ens*, *weights=None*)
Return `Conformation` (page **??**) that is closest to the center of *ens*. In addition to `Ensemble` (page **??**) instances, `Atomic` (page **??**) instances are accepted as *ens* argument. In this case a `Selection` (page **??**) with central coordinate set as active will be returned.

3.6.7 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 **??**) for details.

getMatrix()
Returns a copy of the 4x4 transformation matrix whose top left is rotation matrix and last column is translation vector.

getRotation()
Return rotation matrix.

getTranslation()
Return translation vector.

setRotation(*rotation*)
Set rotation matrix.

setTranslation(*translation*)
Set translation vector.

applyTransformation(*transformation*, *atoms*)
Return *atoms* after applying *transformation*. If *atoms* is a `Atomic` (page **??**) instance, it will be returned after *transformation* is applied to its active coordinate set. If *atoms* is an `AtomPointer` (page **??**) instance, *transformation* will be applied to the corresponding coordinate set in the associated `AtomGroup` (page **??**).

alignCoordsets(*atoms*, *weights=None*)
Return *atoms* after superposing coordinate sets onto its active coordinate set. Transformations will be calculated for *atoms* and applied to its `AtomGroup` (page **??**), when applicable. Optionally, atomic *weights* can be passed for weighted superposition.

calcRMSD(*reference*, *target=None*, *weights=None*)
Return root-mean-square deviation(s) (RMSD) between reference and target coordinates.

calcTransformation(*mobile*, *target*, *weights=None*)
Returns a `Transformation` (page **??**) 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 **??**) instances, e.g. `AtomGroup` (page **??**), `Chain` (page **??**), or `Selection` (page **??**).

superpose(*mobile*, *target*, *weights=None*)
Return *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[314]) – an offset array with shape `([1,] 3)` or `(n_atoms, 3)` or a transformation matrix with shape `(4, 4)`
- **to** (numpy.ndarray[315]) – a point array with shape `([1,] 3)`
- **ag** (*bool*[316]) – when *atoms* is a `AtomSubset` (page **??**), apply translation vector (*to*) or transformation matrix to the `AtomGroup` (page **??**), default is **False**
- **weights** (numpy.ndarray[317]) – array of atomic weights with shape `(n_atoms[, 1])`

When *to* argument is passed, `calcCenter()` (page **??**) 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 **??**), 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 **??**), `AtomGroup` (page **??**), numpy.ndarray[318]) – a frame instance or a coordinate set

[314] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[315] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[316] http://docs.python.org/library/functions.html#bool
[317] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[318] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

- **unitcell** (numpy.ndarray[319]) – orthorhombic unitcell array with shape (3,)
- **center** (numpy.ndarray[320]) – 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 **??**) function.

3.7 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[321].

3.7.1 PDB resources

- fetchPDB() (page **??**) - retrieve PDB files
- fetchPDBviaFTP() (page **??**) - download PDB/PDBML/mmCIF files
- fetchPDBviaHTTP() (page **??**) - download PDB files

You can use following functions to manage PDB file resources:

- pathPDBFolder() (page **??**) - local folder for storing PDB files
- pathPDBMirror() (page **??**) - local PDB mirror path
- wwPDBServer() (page **??**) - set wwPDB FTP/HTTP server for downloads

Following functions can be used to handle local PDB files:

- findPDBFiles() (page **??**) - return a dictionary containing files in a path
- iterPDBFilenames() (page **??**) - yield file names in a path or local PDB mirror

3.7.2 Blast search PDB

The following are for blast searching PDB content.

- blastPDB() (page **??**) - blast search NCBI PDB database
- PDBBlastRecord (page **??**) - 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 **??**) - get list of identifiers in a PDB sequence cluster
- loadPDBClusters() (page **??**) - load PDB clusters into memory
- fetchPDBClusters() (page **??**) - retrieve PDB sequence cluster data from wwPDB

[319] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[320] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[321] http://wwpdb.org

3.7.3 Parse/write PDB files

Following ProDy functions are for parsing and writing `.pdb` files:

- `parsePDB()` (page **??**) - parse `.pdb` formated file
- `parsePDBStream()` (page **??**) - parse `.pdb` formated stream
- `writePDB()` (page **??**) - write `.pdb` formatted file
- `writePDBStream()` (page **??**) write `.pdb` formated stream

Since `.pqr` format is similar to `.pdb` format, following functions come as bonus features:

- `writePQR()` (page **??**) - write atomic data to a file in `.pqr` format
- `parsePQR()` (page **??**) - parse atomic data from files in `.pqr` format

See Also:

Atom data (coordinates, atom names, residue names, etc.) parsed from PDB/PSF/PQR files are stored in `AtomGroup` (page **??**) instances. See `atomic` (page **??**) module documentation for more details.

3.7.4 Quick visualization

`showProtein()` (page **??**) function can be used to take a quick look at protein structures.

3.7.5 Edit structures

Following functions allow editing structures using structural data from PDB header records:

- `assignSecstr()` (page **??**) - add secondary structure data from header to atoms
- `buildBiomolecules()` (page **??**) - build biomolecule from header records

3.7.6 PDB header data

Use the following to parse and access header data in PDB files:

- `parsePDBHeader()` (page **??**) - parse header data from `.pdb` files
- `Chemical` (page **??**) - store PDB chemical (heterogen) component data
- `Polymer` (page **??**) - store PDB polymer (macromolecule) component data
- `DBRef` (page **??**) - store polymer sequence database reference records

3.7.7 Ligand data

Following function can be used to fetch meta data on PDB ligands:

- `fetchPDBLigand()` (page **??**) - retrieve ligand from Ligand-Expo

3.7.8 Compare/align chains

Following functions can be used to match, align, and map polypeptide chains:

- `matchChains()` (page **??**) - finds matching chains in two protein structures
- `matchAlign()` (page **??**) - finds best matching chains and aligns structures
- `mapOntoChain()` (page **??**) - maps chains in a structure onto a reference chain

Following functions can be used to adjust alignment parameters:

- `getAlignmentMethod()` (page **??**), `setAlignmentMethod()` (page **??**)
- `getMatchScore()` (page **??**), `setMatchScore()` (page **??**)
- `getMismatchScore()` (page **??**), `setMismatchScore()` (page **??**)
- `getGapPenalty()` (page **??**), `setGapPenalty()` (page **??**)
- `getGapExtPenalty()` (page **??**), `setGapExtPenalty()` (page **??**)

3.7.9 Execute DSSP

Following functions can be used to execute DSSP structural analysis program and/or parse results:

- `execDSSP()` (page **??**) - execute **dssp**
- `performDSSP()` (page **??**) - execute **dssp** and parse results
- `parseDSSP()` (page **??**) - parse structural data from **dssp** output

3.7.10 Execute STRIDE

Following functions can be used to execute STRIDE structural analysis program and/or parse results:

- `execSTRIDE()` (page **??**) - execute **stride**
- `performSTRIDE()` (page **??**) - execute **stride** and parse results
- `parseSTRIDE()` (page **??**) - parse structural data from **stride** output

3.7.11 PDB Blast Search

This module defines functions for blast searching Protein Data Bank.

class **`PDBBlastRecord`**(*xml*, *sequence=None*)

A class to store results from ProteinDataBank blast search.

Instantiate a PDBlast object instance.

Parameters

- **xml** (*str*[322]) – blast search results in XML format or an XML file that contains the results
- **sequence** (*str*[323]) – query sequence

[322] http://docs.python.org/library/functions.html#str
[323] http://docs.python.org/library/functions.html#str

getBest()
Return a dictionary containing structure and alignment information for the hit with highest sequence identity.

getHits(*percent_identity=90.0*, *percent_overlap=70.0*, *chain=False*)
Return a dictionary in which PDB identifiers are mapped to structure and alignment information.

Parameters

- **percent_identity** (*float*[324]) – PDB hits with percent sequence identity equal to or higher than this value will be returned, default is `90.0`
- **percent_overlap** (*float*[325]) – PDB hits with percent coverage of the query sequence equivalent or better will be returned, default is `70.0`
- **chain** (*bool*[326]) – if chain is **True**, individual chains in a PDB file will be considered as separate hits , default is **False**

getParameters()
Return parameters used in blast search.

getSequence()
Return the query sequence that was used in the search.

blastPDB(*sequence*, *filename=None*, ***kwargs*)
Return a `PDBBlastRecord` (page **??**) instance that contains results from blast searching of ProteinDataBank database *sequence* using NCBI blastp.

Parameters

- **sequence** (*str*[327]) – single-letter code amino acid sequence of the protein without any gap characters, all white spaces will be removed
- **filename** (*str*[328]) – 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 doubled when results are not ready. *timeout* (default is 120s) determines when to give up waiting for the results.

3.7.12 Structure Comparison

This module defines functions for comparing and mapping polypeptide chains.

matchChains(*atoms1*, *atoms2*, ***kwargs*)
Return 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 **??**)) of atom groups. This function returns a list of matching chains in a tuples that contain 4 items:

- matching chain from *atoms1* as a `AtomMap` (page **??**) instance,
- matching chain from *atoms2* as a `AtomMap` (page **??**) instance,
- percent sequence identity of the match,
- percent sequence overlap of the match.

[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#bool
[327] http://docs.python.org/library/functions.html#str
[328] http://docs.python.org/library/functions.html#str

List of matches are sorted in decreasing percent sequence identity order. `AtomMap` (page **??**) instances can be used to calculate RMSD values and superpose atom groups.

Parameters

- **atoms1** (`Chain` (page **??**), `AtomGroup` (page **??**), `Selection` (page **??**)) – atoms that contain a chain
- **atoms2** (`Chain` (page **??**), `AtomGroup` (page **??**), `Selection` (page **??**)) – atoms that contain a chain
- **subset** (*string*[329]) – 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*[330]) – percent sequence identity, default is 90
- **overlap** (*float*[331]) – percent overlap, default is 90
- **pwalign** (*bool*[332]) – 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, `Bio.pairwise2` is used for pairwise sequence alignment, and matching is performed based on the 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 **??**) 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 **??**) instance,
- matching chain from *target* as a `AtomMap` (page **??**) instance,
- percent sequence identity of the match,
- percent sequence overlap of the match.

Parameters

- **mobile** (`Chain` (page **??**), `AtomGroup` (page **??**), `Selection` (page **??**)) – atoms that contain a protein chain
- **target** (`Chain` (page **??**), `AtomGroup` (page **??**), `Selection` (page **??**)) – atoms that contain a protein chain
- **tarsel** (*str*[333]) – *target* atoms that will be used for alignment, default is `'calpha'`
- **allcsets** (*bool*[334]) – align all coordinate sets of *mobile*, default is **True**
- **seqid** (*float*[335]) – percent sequence identity, default is 90

[329] http://docs.python.org/library/string.html#string
[330] http://docs.python.org/library/functions.html#float
[331] http://docs.python.org/library/functions.html#float
[332] http://docs.python.org/library/functions.html#bool
[333] http://docs.python.org/library/functions.html#str
[334] http://docs.python.org/library/functions.html#bool
[335] http://docs.python.org/library/functions.html#float

- **overlap** (*float*[336]) – percent overlap, default is 90
- **pwalign** (*bool*[337]) – perform pairwise sequence alignment

mapOntoChain(*atoms*, *chain*, ***kwargs*)

Map *atoms* onto *chain*. This function returns a list of mappings. Each mapping is a tuple that contains 4 items:

- Mapped chain as an `AtomMap` (page **??**) instance,
- *chain* as an `AtomMap` (page **??**) instance,
- Percent sequence identitity,
- Percent sequence overlap

Mappings are returned in decreasing percent sequence identity order. `AtomMap` (page **??**) that keeps mapped atom indices contains dummy atoms in place of unmapped atoms.

Parameters

- **atoms** (`Chain` (page **??**), `AtomGroup` (page **??**), `Selection` (page **??**)) – atoms that will be mapped to the target *chain*
- **chain** (`Chain` (page **??**)) – chain to which atoms will be mapped
- **subset** (*string*[338]) – 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*[339]) – percent sequence identity, default is 90
- **overlap** (*float*[340]) – percent overlap, default is 90
- **pwalign** (*bool*[341]) – perform pairwise sequence alignment

This function tries to map *atoms* to *chain* based on residue numbers and types. Each individual chain in *atoms* is compared to target *chain*. This works well for different structures of the same protein. When it fails, `Bio.pairwise2` is used for sequence alignment, and mapping is performed based on the sequence alignment. User can control, whether sequence alignment is performed or not with *pwalign* keyword. If `pwalign=True` is passed, pairwise alignment is enforced.

getMatchScore()

Return match score used to align sequences.

setMatchScore(*match_score*)

Set match score used to align sequences.

getMismatchScore()

Return mismatch score used to align sequences.

setMismatchScore(*mismatch_score*)

Set mismatch score used to align sequences.

getGapPenalty()

Return gap opening penalty used for pairwise alignment.

[336] http://docs.python.org/library/functions.html#float
[337] http://docs.python.org/library/functions.html#bool
[338] http://docs.python.org/library/string.html#string
[339] http://docs.python.org/library/functions.html#float
[340] http://docs.python.org/library/functions.html#float
[341] http://docs.python.org/library/functions.html#bool

setGapPenalty(*gap_penalty*)
Set gap opening penalty used for pairwise alignment.

getGapExtPenalty()
Return gap extension penalty used for pairwise alignment.

setGapExtPenalty(*gap_ext_penalty*)
Set gap extension penalty used for pairwise alignment.

getAlignmentMethod()
Return pairwise alignment method.

setAlignmentMethod(*method*)
Set pairwise alignment method (global or local).

3.7.13 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`[342] 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 **??**).

parseDSSP(*dssp*, *ag*, *parseall=False*)
Parse DSSP data from file *dssp* into `AtomGroup` (page **??**) 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.
- *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.

[342] http://docs.python.org/library/gzip.html#gzip

- *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 **??**) instance. See also `execDSSP()` (page **??**) and `parseDSSP()` (page **??**).

3.7.14 Miscellaneous Tools

This module defines miscellaneous functions dealing with protein data.

showProtein (**atoms*, ***kwargs*)

Show protein representation using `Axes3D()`. This function is designed for generating a quick view of the contents of a `AtomGroup` (page **??**) or `Selection` (page **??**).

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.

writePQR (*filename*, *atoms*)

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.

3.7.15 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 **??**) 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 **??**) 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 **??**)
modified	list	modified residues (SEQMOD) when modified residues are present, each will be represented as: `(resname, resnum, icode, stdname, comment)`
pdbentry	str	PDB entry that polymer data is extracted from

Polymer class instances can be obtained as follows:

```
In [7]: polymer = parsePDBHeader('2k39', 'polymers')[0]

In [8]: polymer
 Out[8]: <Polymer: UBIQUITIN (2K39_A)>

In [9]: polymer.pdbentry
 Out[9]: '2K39'

In [10]: polymer.chid
Out[10]: 'A'

In [11]: polymer.name
Out[11]: 'UBIQUITIN'

In [12]: polymer.sequence
```

```
Out[12]: 'MQIFVKTLTGKTITLEVEPSDTIENVKAKIQDKEGIPPDQQRLIFAGKQLEDGRTLSDYNIQKESTLHLVLRLRGG'

In [13]: len(polymer.sequence)
Out[13]: 76

In [14]: len(polymer)
Out[14]: 76

In [15]: dbref = polymer.dbrefs[0]

In [16]: dbref.database
Out[16]: 'UniProt'

In [17]: dbref.accession
Out[17]: 'P62972'

In [18]: dbref.idcode
Out[18]: '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*)
Return 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 **??**)
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 **??**) and `DBRef` (page **??**)
SEQADV	polymers	see `Polymer` (page **??**)
SEQRES	polymers	see `Polymer` (page **??**)
MODRES	polymers	see `Polymer` (page **??**)
HELIX	polymers	see `Polymer` (page **??**)
SHEET	polymers	see `Polymer` (page **??**)
HET	chemicals	see `Chemical` (page **??**)
HETNAM	chemicals	see `Chemical` (page **??**)
HETSYN	chemicals	see `Chemical` (page **??**)
FORMUL	chemicals	see `Chemical` (page **??**)
REMARK 2	resolution	resolution of structures, when applicable
REMARK 4	version	PDB file version
REMARK 350	biomoltrans	biomolecular transformation lines (unprocessed)

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=False*)

Assign secondary structure from *header* dictionary to *atoms*. *header* must be a dictionary parsed using the `parsePDB()` (page **??**). *atoms* may be an instance of `AtomGroup` (page **??**), `Selection` (page **??**), `Chain` (page **??**) or `Residue` (page **??**). ProDy can be configured to automatically parse and assign secondary structure information using `confProDy(auto_secondary=True)` command. See also `confProDy()` (page **??**) 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*)

Return *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 **??**) instances in a `list()`[343].

If the resulting biomolecule has more than 26 chains, the molecular assembly will be split into multiple `AtomGroup` (page **??**) instances each containing at most 26 chains. These `AtomGroup` (page **??**) instances will be returned in a tuple.

[343] http://docs.python.org/library/functions.html#list

Note that atoms in biomolecules are ordered according to chain identifiers.

3.7.16 Local PDB Handlers

This module defines functions for handling local PDB folders.

pathPDBFolder (*folder=None*, *divided=False*)

Return or specify local PDB folder for storing PDB files downloaded from wwPDB[344] servers. Files stored in this folder can be accessed via `fetchPDB()` (page **??**) 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*)

Return or specify PDB mirror path to be used by `fetchPDB()` (page **??**). 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*)

Return path(s) to PDB file(s) for specified *pdb* identifier(s). Files will be sought in user specified *folder* or current working director, 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. See `pathPDBFolder()` (page **??**) and `pathPDBMirror()` (page **??**) for managing local resources, `fetchPDBviaFTP()` (page **??**) and `fetchPDBviaFTP()` (page **??**) for downloading files from PDB servers.

fetchPDBfromMirror (**pdb*, ***kwargs*)

Return 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[345] and mmCIF[346] 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 **??**)). 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*)

Return 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 **??**).

[344] http://www.wwpdb.org/
[345] http://pdbml.pdb.org/
[346] http://mmcif.pdb.org/

3.7.17 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: ftp://resources.rcsb.org/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 **??**) function and be accessed using `listPDBCluster()` (page **??**).

loadPDBClusters(*sqid=None*)
Load previously fetched PDB sequence clusters from disk to memory.

listPDBCluster(*pdb*, *ch*, *sqid=95*)
Return 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 **??**) needs to be called. This function will load the PDB sequence clusters for *sqid* automatically using `loadPDBClusters()` (page **??**).

3.7.18 PDB File

This module defines functions for parsing and writing PDB files[347].

parsePDBStream(*stream*, ***kwargs*)
Return an `AtomGroup` (page **??**) 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*[348]) – title of the `AtomGroup` (page **??**) instance, default is the PDB filename or PDB identifier
- **ag** (`AtomGroup` (page **??**)) – `AtomGroup` (page **??**) 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*[349]) – chain identifiers for parsing specific chains, e.g. `chain='A'`, `chain='B'`, `chain='DE'`, by default all chains are parsed
- **subset** (*str*[350]) – a predefined keyword to parse subset of atoms, valid keywords are `'calpha'` (`'ca'`), `'backbone'` (`'bb'`), or **None** (read all atoms), e.g. `subset='bb'`

[347] http://www.wwpdb.org/documentation/format32/v3.2.html
[348] http://docs.python.org/library/functions.html#str
[349] http://docs.python.org/library/functions.html#str
[350] http://docs.python.org/library/functions.html#str

- **model** (*int, list*) – model index or None (read all models), e.g. `model=10`
- **header** (*bool*[351]) – if **True** PDB header content will be parsed and returned
- **altloc** (*str*[352]) – 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 **True** all alternate locations will be parsed and each will be appended as a distinct coordinate set, default is `"A"`
- **biomol** (*False*[353]) – if **True**, biomolecule obtained by transforming the coordinates using information from header section will be returned
- **secondary** (*False*[354]) – if **True**, secondary structure information from header section will be assigned atoms

If `model=0` and `header=True`, return header dictionary only.

Note that this function does not evaluate `CONECT` records.

parsePDB(*pdb*, ***kwargs*)

Return an `AtomGroup` (page **??**) and/or dictionary containing header data parsed from a PDB file.

This function extends `parsePDBStream()` (page **??**).

See *Parse PDB files*[355] for a detailed usage example.

Parameters

- **pdb** – a PDB identifier or a filename If needed, PDB files are downloaded using `fetchPDB()` (page **??**) function.
- **title** (*str*[356]) – title of the `AtomGroup` (page **??**) instance, default is the PDB filename or PDB identifier
- **ag** (`AtomGroup` (page **??**)) – `AtomGroup` (page **??**) 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*[357]) – chain identifiers for parsing specific chains, e.g. `chain='A'`, `chain='B'`, `chain='DE'`, by default all chains are parsed
- **subset** (*str*[358]) – 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*[359]) – if **True** PDB header content will be parsed and returned
- **altloc** (*str*[360]) – 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

[351] http://docs.python.org/library/functions.html#bool
[352] http://docs.python.org/library/functions.html#str
[353] http://docs.python.org/library/constants.html#False
[354] http://docs.python.org/library/constants.html#False
[355] http://prody.csb.pitt.edu/tutorials/structure_analysis/pdbfiles.html#parsepdb
[356] http://docs.python.org/library/functions.html#str
[357] http://docs.python.org/library/functions.html#str
[358] http://docs.python.org/library/functions.html#str
[359] http://docs.python.org/library/functions.html#bool
[360] http://docs.python.org/library/functions.html#str

altloc is set **True** all alternate locations will be parsed and each will be appended as a distinct coordinate set, default is `"A"`

- **biomol** (*False*[361]) – if **True**, biomolecule obtained by transforming the coordinates using information from header section will be returned
- **secondary** (*False*[362]) – if **True**, secondary structure information from header section will be assigned atoms

If `model=0` and `header=True`, return header dictionary only.

Note that this function does not evaluate `CONECT` records.

parsePQR(*filename*, ***kwargs*)

Return an `AtomGroup` (page **??**) containing data parsed from PDB lines.

Parameters

- **filename** (*str*[363]) – a PQR filename
- **title** (*str*[364]) – title of the `AtomGroup` (page **??**) instance, default is the PDB filename or PDB identifier
- **ag** (`AtomGroup` (page **??**)) – `AtomGroup` (page **??**) 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*[365]) – chain identifiers for parsing specific chains, e.g. `chain='A'`, `chain='B'`, `chain='DE'`, by default all chains are parsed
- **subset** (*str*[366]) – a predefined keyword to parse subset of atoms, valid keywords are `'calpha'` (`'ca'`), `'backbone'` (`'bb'`), or **None** (read all atoms), e.g. `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)
- **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

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

- **atoms** – an object with atom and coordinate data

[361] http://docs.python.org/library/constants.html#False
[362] http://docs.python.org/library/constants.html#False
[363] http://docs.python.org/library/functions.html#str
[364] http://docs.python.org/library/functions.html#str
[365] http://docs.python.org/library/functions.html#str
[366] http://docs.python.org/library/functions.html#str

- **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
- **autoext** – when not present, append extension `.pdb` to *filename*

3.7.19 PDB Ligands

This module defines functions for fetching PDB ligand data.

fetchPDBLigand(*cci*, *filename=None*)

Fetch PDB ligand data from PDB[367] 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()`[368] 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')

In [3]: ligand_data['model_coordinates_db_code']
 Out[3]: '1IEP'

In [4]: ligand_model = ligand_data['model']

In [5]: ligand_ideal = ligand_data['ideal']

In [6]: transformation = superpose(ligand_ideal.noh, ligand_model.noh)

In [7]: calcRMSD(ligand_ideal.noh, ligand_model.noh)
 Out[7]: 2.2678638214526532
```

3.7.20 Stride Tools

This module defines functions for executing STRIDE program and parsing its output.

[367] http://www.pdb.org
[368] http://docs.python.org/library/stdtypes.html#dict.get

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`[369] 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 **??**).

parseSTRIDE(*stride*, *ag*)
Parse STRIDE output from file *stride* into `AtomGroup` (page **??**) 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 **??**) instance. See also `execSTRIDE()` (page **??**) and `parseSTRIDE()` (page **??**).

3.7.21 wwPDB Tools

This module defines functions for accessing wwPDB servers.

wwPDBServer(**key*)
Set/get wwPDB[370] 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, or mmCIF 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*. *format* keyword argument can be used to retrieve PDBML[371] and mmCIF[372] 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.

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*.

[369] http://docs.python.org/library/gzip.html#gzip
[370] http://www.wwpdb.org/
[371] http://pdbml.pdb.org/
[372] http://mmcif.pdb.org/

3.8 Sequence Analysis

This module contains features for analyzing protein sequences.

3.8.1 Classes

- `MSA` (page **??**) - store MSA data indexed by label
- `Sequence` (page **??**) - store sequence data

3.8.2 MSA IO

- `MSAFile` (page **??**) - read/write MSA files in FASTA/SELEX/Stockholm formats
- `parseMSA()` (page **??**) - parse MSA files
- `writeMSA()` (page **??**) - parse MSA files

3.8.3 Editing

- `mergeMSA()` (page **??**) - merge MSA data for multi-domain proteins
- `refineMSA()` (page **??**) - refine MSA by removing gapped columns and/or sequences

3.8.4 Analysis

- `calcMSAOccupancy()` (page **??**) - calculate row (sequence) or column occupancy
- `calcShannonEntropy()` (page **??**) - calculate Shannon entropy
- `buildMutinfoMatrix()` (page **??**) - build mutual information matrix
- `buildOMESMatrix()` (page **??**) - build mutual observed minus expected squared covariance matrix
- `buildSCAMatrix()` (page **??**)- build statistical coupling analysis matrix
- `buildSeqidMatrix()` (page **??**)- build sequence identity matrix
- `buildDirectInfoMatrix()` (page **??**) - build direct information matrix
- `uniqueSequences()` (page **??**) - select unique sequences
- `applyMutinfoCorr()` (page **??**) - apply correction to mutual information matrix
- `applyMutinfoNorm()` (page **??**) - apply normalization to mutual information matrix
- `calcMeff()` (page **??**) - calculate sequence weights
- `calcRankorder()` (page **??**) - rank order scores

3.8.5 Plotting

- `showShannonEntropy()` (page **??**) - plot Shannon entropy
- `showMSAOccupancy()` (page **??**) - plot row (sequence) or column occupancy
- `showMutinfoMatrix()` (page **??**) - show mutual information matrix

3.8.6 Analysis Functions

This module defines MSA analysis functions.

calcShannonEntropy(*msa*, *ambiguity=True*, *omitgaps=True*, ***kwargs*)
Return Shannon entropy array calculated for *msa*, which may be an `MSA` (page **??**) 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*)
Return mutual information matrix calculated for *msa*, which may be an `MSA` (page **??**) 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*)
Return 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 **??**) 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'*)
Return a copy of *mutinfo* array after average product correction (default) or average sum correction is applied. See [DSD08] (page **??**) for details.

applyMutinfoNorm(*mutinfo*, *entropy*, *norm='sument'*)

Apply one of the normalizations discussed in [MLC05] (page **??**) 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 **??**) *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 smymetric only lower triangle indices will be returned, with diagonal elements if *diag* is **True** (default).

buildSeqidMatrix(*msa*, *turbo=True*)

Return sequence identity matrix for *msa*.

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*)

Return a boolean array marking unique sequences in *msa*. A sequence sharing sequence identity of *sqid* or more with another sequence coming before itself in *msa* will have a **False** 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*)

Return OMES (Observed Minus Expected Squared) covariance matrix calculated for *msa*, which may be an `MSA` (page **??**) instance or a 2D NumPy character array. OMES is defined as:

```
                    (N_OBS - N_EX)^2                (f_i,j - f_i * f_j)^2
OMES_(i,j) = sum(------------------) = N * sum(-----------------------)
                         N_EX                           f_i * f_j
```

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*)

Return SCA matrix calculated for *msa*, which may be an `MSA` (page **??**) 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*)

Return direct information matrix calculated for *msa*, which may be an `MSA` (page **??**) 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 **??**).

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*)

Return the Meff for *msa*, which may be an `MSA` (page **??**) 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**.

3.8.7 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*)
Return the number of sequences that *label* maps onto.

getArray()
Return a copy of the MSA character array.

getIndex(*label*)
Return 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*)
Return label of the sequence at given *index*. Residue numbers will be removed from the sequence label, unless *full* is **True**.

getResnums(*index*)
Return starting and ending residue numbers (*resnum*) for the sequence at given *index*.

getTitle()
Return title of the instance.

isAligned()
Return **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()
Return 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()
Return number of residues (or columns in the MSA), if MSA is aligned.

numSequences()
Return number of sequences.

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

split
Return split label when iterating or indexing.

refineMSA(*msa*, *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 **??**)) – multiple sequence alignment
- **label** (*str*[373]) – remove columns that are gaps in the sequence matching label, `msa.getIndex(label)` must return a sequence index, a PDB identifier is also acceptable

[373] http://docs.python.org/library/functions.html#str

- **rowocc** (*float*[374]) – row occupancy, sequences with less occupancy will be removed after *label* refinement is applied
- **seqid** (*float*[375]) – keep unique sequences at specified sequence identity level, unique sequences are identified using `uniqueSequences()` (page **??**)
- **colocc** (*float*[376]) – 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 **??**) function (see also `Polymer` (page **??**) and `DBRef` (page **??**)).

The order of refinements are applied in the order of arguments. If *label* and *unique* is specified is specified, sequence matching *label* will be kept in the refined `MSA` (page **??**) although it may be similar to some other sequence.

mergeMSA(**msa*, ***kwargs*)

Return an `MSA` (page **??**) 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.

3.8.8 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 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`), or SELEX (`.slx`) *format*. For spesified extensions, *format* argument is not needed. If *aligned* is **True**, unaligned sequences in the file or stream will cause an `IOError` exception. *filter*, a function that returns a boolean, can be used for filtering sequences, see `setFilter()` (page **??**) for details. *slice* can be used to slice sequences, and is applied after filtering, see `setSlice()` (page **??**) for details.

close()

Close the file. This method will not affect a stream.

getFilename()

Return filename, or **None** if instance is handling a stream.

getFilter()

Return function used for filtering sequences.

getFormat()

Return file format.

getSlice()

Return object used to slice sequences.

[374] http://docs.python.org/library/functions.html#float
[375] http://docs.python.org/library/functions.html#float
[376] http://docs.python.org/library/functions.html#float

getTitle()
Return title of the instance.

isAligned()
Return **True** if MSA is aligned.

reset()
Return 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()`[377] or a `list()`[378] of numbers.

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

write(*seq*)
Write *seq*, an `Sequence` (page **??**) instance, into the MSA file.

closed
True for closed file.

format
Format of the MSA file.

splitSeqLabel(*label*)
Return label, starting residue number, and ending residue number parsed from sequence label.

parseMSA(*filename*, ***kwargs*)
Return an `MSA` (page **??**) instance that stores multiple sequence alignment and sequence labels parsed from Stockholm, SELEX, 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*)
Return *filename* containing *msa*, a `MSA` (page **??**) or `MSAFile` (page **??**) 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 **??**) instances will be written using C function into uncompressed files.

3.8.9 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 **??**) instance *msa* using `calcMSAOccupancy()` (page **??**). *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()`[379] function.

showShannonEntropy(*entropy*, *indices=None*, ***kwargs*)
Show a bar plot of Shannon *entropy* array. `MSA` (page **??**) instances or Numpy character arrays storing sequence alignments are also accepted as *entropy* argument, in which case `calcShannonEntropy()`

[377] http://docs.python.org/library/functions.html#slice
[378] http://docs.python.org/library/functions.html#list
[379] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.bar

(page **??**) 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()`[380] function.

showMutinfoMatrix(*mutinfo*, **args*, ***kwargs*)
Show a heatmap of mutual information array. `MSA` (page **??**) instances or Numpy character arrays storing sequence alignment are also accepted as *mutinfo* argument, in which case `buildMutinfoMatrix()` (page **??**) will be used for calculations. Note that x, y axes contain indices of the matrix starting from 1.

Mutual information is plotted using `imshow()`[381] 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 **??**) instances or Numpy character arrays storing sequence alignment are also accepted as *dirinfo* argument, in which case `buildDirectInfoMatrix()` (page **??**) will be used for calculations. Note that x, y axes contain indices of the matrix starting from 1.

Direct information is plotted using `imshow()`[382] 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 **??**) instances. blah

or Numpy character arrays storing sequence alignment are also accepted as *scainfo* argument, in which case `buildSCAMatrix()` (page **??**) will be used for calculations. Note that x, y axes contain indices of the matrix starting from 1.

SCA information is plotted using `imshow()`[383] function. vmin and vmax values can be set by user to achieve better signals using this function.

3.8.10 Sequence

This module handles individual sequences.

class Sequence(**args*)
Handle individual sequences of an `MSA` (page **??**) object

Depending on input arguments, instances may point to an `MSA` (page **??**) object or store its own data:

MSA Pointer

An `MSA` (page **??**) instance and an index:

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

In [2]: msa = parseMSA('piwi_seed.sth')
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-2-fce130cfb7ce> in <module>()
----> 1 msa = parseMSA('piwi_seed.sth')

/home/docs/checkouts/readthedocs.org/user_builds/prody/envs/v1.5/local/lib/python2.7/site-packag
    483         if not fileok:
```

[380] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.bar
[381] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.imshow
[382] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.imshow
[383] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.imshow

```
    484             raise IOError('[Errno 2] No such file or directory: ' +
--> 485                           repr(filename))
    486
    487     # if MSA is a compressed file or filter/slice is passed, use

IOError: [Errno 2] No such file or directory: 'piwi_seed.sth'

In [3]: Sequence(msa, 0)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-8c5a985dfe08> in <module>()
----> 1 Sequence(msa, 0)

NameError: name 'msa' is not defined

In [4]: msa[0]
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-ac8524cb1326> in <module>()
----> 1 msa[0]

NameError: name 'msa' is not defined
```

Independent

Instantiation with sequence and label (optional) string:

```
In [5]: Sequence('SOME-SEQUENCE-STRING', 'MySeq/1-18')
 Out[5]: <Sequence: MySeq (length 20; 18 residues and 2 gaps)>
```

copy()
Return a copy of the instance that owns its sequence data.

getIndex()
Return sequence index or **None**.

getLabel(*full=False*)
Return label of the sequence.

getMSA()
Return MSA (page **??**) instance or **None**.

getResnums(*gaps=False*)
Return 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. When label does not contain residue number information, indices a range of numbers starting from 1 is returned.

numGaps()
Return number of gap characters.

numResidues()
Return the number of alphabet characters.

3.9 Trajectory I/O

This module defines classes for handling trajectory files in DCD format.

3.9.1 Parse/write DCD files

- `DCDFile` (page **??**)
- `parseDCD()` (page **??**)
- `writeDCD()` (page **??**)

3.9.2 Parse structure files

- `parsePSF()` (page **??**)

3.9.3 Handle multiple files

- `Trajectory` (page **??**)

3.9.4 Handle frame data

- `Frame` (page **??**)

3.9.5 Examples

Following examples show how to use trajectory classes and functions:

- *Trajectory Analysis*[384]
- *Trajectory Analysis II*[385]
- *Essential Dynamics Analysis*[386]

3.9.6 DCD File

This module defines classes for handling trajectory files in DCD format[387].

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.

[384] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory.html#trajectory
[385] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory2.html#trajectory2
[386] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda
[387] http://www.ks.uiuc.edu/Research/namd/2.6/ug/node13.html

getAtoms()
Return associated/selected atoms.

getCoords()
Return 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*)
Return relative path to the current file. For absolute path, pass `absolute=True` argument.

getFirstTimestep()
Return first timestep value.

getFrame(*index*)
Return frame at given *index*.

getFrameFreq()
Return timesteps between frames.

getLinked()
Return linked `AtomGroup` (page **??**) instance, or **None** if a link is not established.

getRemarks()
Return remarks parsed from DCD file.

getTimestep()
Return timestep size.

getTitle()
Return title of the ensemble.

getWeights()
Return 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 **??**) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) methods is called.

hasUnitcell()
Return `True` if trajectory has unitcell data.

isLinked()
Return **True** if trajectory is linked to an `AtomGroup` (page **??**) 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 **??**) 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 **??**) will be overwritten.

next()
Return next coordinate set in a `Frame` (page **??**) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Return next coordinate set.

nextIndex()
Return the index of the next frame.

numAtoms()
Return number of atoms.

numCoordsets()
Return number of frames.

numFixed()
Return number of fixed atoms.

numFrames()
Return number of frames.

numSelected()
Return number of selected atoms. A subset of atoms can be selected by passing a selection to `setAtoms()` (page **??**).

reset()
Go to first frame at index 0. First frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) 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 **??**) and `Frame` (page **??**) instances become suitable arguments for `writePDB()` (page **??**). 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 **??**) method.

setCoords(*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with `getCoords()` (page **??**) 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 **??**)) Next frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) 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 **??**) or `Ensemble` (page **??**) 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 **??**) class for parsing coordinates of a subset of atoms.

Parameters

- **filename** (*str*[388]) – DCD filename
- **start** (*int*[389]) – index of first frame to read
- **stop** (*int*[390]) – index of the frame that stops reading
- **step** (*int*[391]) – steps between reading frames, default is 1 meaning every frame
- **astype** (*type*[392]) – 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 :class:'Trajectory', :class:'DCDFile', or :class:'Ensemble' instance. *filename* is returned upon successful output of file.

3.9.7 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()

Return associated/selected atoms.

getCoords()

Return a copy of coordinates of (selected) atoms.

getDeviations()

Return deviations from the trajectory reference coordinates.

getIndex()

Return index.

[388] http://docs.python.org/library/functions.html#str
[389] http://docs.python.org/library/functions.html#int
[390] http://docs.python.org/library/functions.html#int
[391] http://docs.python.org/library/functions.html#int
[392] http://docs.python.org/library/functions.html#type

getRMSD()
Return RMSD from the trajectory reference coordinates. If weights for the trajectory are set, weighted RMSD will be returned.

getTrajectory()
Return the trajectory.

getUnitcell()
Return a copy of unitcell array.

getVelocities()
Return a copy of velocities of (selected) atoms.

getWeights()
Return coordinate weights for selected atoms.

numAtoms()
Return number of atoms.

numSelected()
Return number of selected atoms.

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.

3.9.8 PSF File

This module defines a function for parsing protein structure files in PSF format[393].

parsePSF(*filename*, *title=None*, *ag=None*)
Return an `AtomGroup` (page **??**) 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 **??**) instance. An `AtomGroup` (page **??**) 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. Note that this function does not evaluate angles, dihedrals, and impropers sections.

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.

3.9.9 Trajectory Base

This module defines base class for trajectory handling.

class TrajBase(*title='Unknown'*)
Base class for `Trajectory` (page **??**) and `TrajFile` (page **??**). Derived classes must implement functions described in this class.

close()
Close trajectory file.

getAtoms()
Return associated/selected atoms.

[393] http://www.ks.uiuc.edu/Training/Tutorials/namd/namd-tutorial-unix-html/node21.html

getCoords()
Return 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*)
Return frame at given *index*.

getLinked()
Return linked `AtomGroup` (page **??**) instance, or **None** if a link is not established.

getTitle()
Return title of the ensemble.

getWeights()
Return 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 **??**) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) methods is called.

hasUnitcell()
Return `True` if trajectory has unitcell data.

isLinked()
Return **True** if trajectory is linked to an `AtomGroup` (page **??**) 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 **??**) 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 **??**) will be overwritten.

next()
Return next coordinate set in a `Frame` (page **??**) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Return next coordinate set.

nextIndex()
Return the index of the next frame.

numAtoms()
Return number of atoms.

numCoordsets()
Return number of frames.

numFrames()
Return number of frames.

numSelected()
Return number of selected atoms. A subset of atoms can be selected by passing a selection to `setAtoms()` (page **??**).

reset()
Go to first frame at index 0. First frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) 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 **??**) and `Frame` (page **??**) instances become suitable arguments for `writePDB()` (page **??**). 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 **??**) method.

setCoords(*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with `getCoords()` (page **??**) 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 **??**)) Next frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) methods is called.

3.9.10 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()
Return associated/selected atoms.

getCoords()
Return 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*)
Return list of filenames opened for reading.

getFirstTimestep()
Return list of first timestep values, one number from each file.

getFrameFreq()
Return list of timesteps between frames, one number from each file.

getLinked()
Return linked `AtomGroup` (page **??**) instance, or **None** if a link is not established.

getTimestep()
Return list of timestep sizes, one number from each file.

getTitle()
Return title of the ensemble.

getWeights()
Return 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 **??**) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) methods is called.

hasUnitcell()
Return `True` if trajectory has unitcell data.

isLinked()
Return **True** if trajectory is linked to an `AtomGroup` (page **??**) 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 **??**) 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 **??**) will be overwritten.

next()
Return next coordinate set in a `Frame` (page **??**) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Return next coordinate set.

nextIndex()
Return the index of the next frame.

numAtoms()
Return number of atoms.

numCoordsets()
Return number of frames.

numFiles()
Return number of open trajectory files.

numFixed()
Return a list of fixed atom numbers, one from each file.

numFrames()
Return number of frames.

numSelected()
Return number of selected atoms. A subset of atoms can be selected by passing a selection to `setAtoms()` (page **??**).

reset()
Go to first frame at index 0. First frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) 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 **??**) and `Frame` (page **??**) instances become suitable arguments for `writePDB()` (page **??**). 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 **??**) method.

setCoords(*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with `getCoords()` (page **??**) 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 **??**)) Next frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) methods is called.

3.9.11 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 **??**)

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()
Return associated/selected atoms.

getCoords()
Return 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*)
Return relative path to the current file. For absolute path, pass `absolute=True` argument.

getFirstTimestep()
Return first timestep value.

getFrame(*index*)
Return frame at given *index*.

getFrameFreq()
Return timesteps between frames.

getLinked()
Return linked `AtomGroup` (page **??**) instance, or **None** if a link is not established.

getTimestep()
Return timestep size.

getTitle()
Return title of the ensemble.

getWeights()
Return 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 **??**) method. `n=-1` will go to the last frame. Frame *n* will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) methods is called.

hasUnitcell()
Return `True` if trajectory has unitcell data.

isLinked()
Return **True** if trajectory is linked to an `AtomGroup` (page **??**) 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 **??**) 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 **??**) will be overwritten.

next()
Return next coordinate set in a `Frame` (page **??**) instance. Note that when atoms are set for the trajectory, this method will return the same frame instance after updating its coordinates.

nextCoordset()
Return next coordinate set.

nextIndex()
Return the index of the next frame.

numAtoms()
Return number of atoms.

numCoordsets()
Return number of frames.

numFixed()
Return number of fixed atoms.

numFrames()
Return number of frames.

numSelected()
Return number of selected atoms. A subset of atoms can be selected by passing a selection to `setAtoms()` (page **??**).

reset()
Go to first frame at index 0. First frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) 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 **??**) and `Frame` (page **??**) instances become suitable arguments for `writePDB()` (page **??**). 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 **??**) method.

setCoords(*coords*)
Set *coords* as the trajectory reference coordinate set. *coords* must be an object with `getCoords()` (page **??**) 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 **??**)) Next frame will not be parsed until one of `next()` (page **??**) or `nextCoordset()` (page **??**) methods is called.

3.10 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`.

3.10.1 Package utilities

- `PackageLogger` (page **??**)
- `PackageSettings` (page **??**)
- `getPackagePath()` (page **??**)
- `setPackagePath()` (page **??**)

3.10.2 Type/Value checkers

- `checkCoords()` (page **??**)
- `checkWeights()` (page **??**)
- `checkTypes()` (page **??**)

3.10.3 Path/file handling

- `gunzip()` (page **??**)
- `openFile()` (page **??**)
- `openDB()` (page **??**)
- `openSQLite()` (page **??**)
- `openURL()` (page **??**)
- `copyFile()` (page **??**)
- `isExecutable()` (page **??**)
- `isReadable()` (page **??**)
- `isWritable()` (page **??**)
- `makePath()` (page **??**)
- `relpath()` (page **??**)
- `which()` (page **??**)
- `pickle()` (page **??**)
- `unpickle()` (page **??**)
- `glob()` (page **??**)

3.10.4 Documentation tools

- `joinRepr()` (page **??**)
- `joinRepr()` (page **??**)
- `joinTerms()` (page **??**)
- `tabulate()` (page **??**)
- `wrapText()` (page **??**)

3.10.5 Miscellaneous tools

- `rangeString()` (page **??**)
- `alnum()` (page **??**)
- `importLA()` (page **??**)
- `dictElement()` (page **??**)

3.10.6 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'*)

Return **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

Raises `TypeError` when *coords* is not an instance of `numpy.ndarray`[394]

Raises `ValueError` when wrong shape, dimensionality, or data type is encountered

checkWeights(*weights*, *natoms*, *ncsets=None*, *dtype=<type 'float'>*)

Return *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*)

Return **True** if types of all *args* match those given in *types*.

Raises `TypeError` when type of an argument is not one of allowed types

[394] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

```
    def incr(n, i):
        '''Return sum of *n* and *i*.'''

        checkTypes(locals(), n=(float, int), i=(float, int))
        return n + i
```

3.10.7 Documentation Tools

This module defines miscellaneous utility functions.

joinLinks(*links*, *sep=', '*, *last=None*, *sort=False*)
Return a string joining *links* as reStructuredText.

joinRepr(*items*, *sep=', '*, *last=None*, *sort=False*)
Return a string joining representations of *items*.

joinTerms(*terms*, *sep=', '*, *last=None*, *sort=False*)
Return a string joining *terms* as reStructuredText.

tabulate(**cols*, ***kwargs*)
Return a table for columns of data.

Parameters

- **header** (*bool*[395]) – make first row a header, default is **True**
- **width** (*int*[396]) – 79

Kwargs space number of white space characters between columns, default is 2

wrapText(*text*, *width=70*, *join='\n'*, ***kwargs*)
Return wrapped lines from `textwrap.wrap()`[397] after *join*ing them.

3.10.8 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 '`
- **error** – prefix to log messages at *error* level, default is `'ERROR '`

addHandler(*hdlr*)
Add the specified handler to this logger.

[395] http://docs.python.org/library/functions.html#bool
[396] http://docs.python.org/library/functions.html#int
[397] http://docs.python.org/library/textwrap.html#textwrap.wrap

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()
Return 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 **??**) and `report()` (page **??**) to learn and report timing, respectively.

timing(*label=None*)
Return timing for a labeled or default (**None**) process.

update(*step*, *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.

3.10.9 Miscellaneous Tools

This module defines miscellaneous utility functions.

class Everything
A place for everything.

rangeString (*lint*, *sep=' '*, *rng=' to '*, *exc=False*, *pos=True*)
Return 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 ()
Return one of `scipy.linalg`[398] or `numpy.linalg`.

[398] http://docs.scipy.org/doc/scipy/reference/linalg.html#scipy.linalg

dictElement (*element*, *prefix=None*)
Returns a dictionary built from the children of *element*, which must be a `xml.etree.ElementTree.Element`[399] 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*)
Return `int(x)`, or `float(x)` upon `ValueError`.

startswith (*this*, *that*)
Return **True** if *this* or *that* starts with the other.

showFigure ()
Call `show()`[400] 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*)
Return total number of bytes consumed by elements of arrays. If *base* is **True**, use number of bytes from the base array.

3.10.10 Path Tools

This module defines functions for handling files and paths.

gunzip (*filename*, *outname=None*)
Return 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 **??**). Default extension `.BAK` is used when one is not set using `confProDy()` (page **??**). 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*[401]) – backup existing file using `backupFile()` (page **??**) when opening in append or write modes, default is obtained from package settings
- **backup_ext** (*str*[402]) – extension for backup file, default is `.BAK`

openDB (*filename*, **args*)
Open a database with given *filename*.

openSQLite (*filename*, **args*)
Return 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.

[399] http://docs.python.org/library/xml.etree.elementtree.html#xml.etree.ElementTree.Element
[400] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.show
[401] http://docs.python.org/library/functions.html#bool
[402] http://docs.python.org/library/functions.html#str

openURL(*url*, *timeout=5*, ***kwargs*)
Open *url* for reading. Raise an `IOError` 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` instance for opening the *url*.

copyFile(*src*, *dst*)
Return *dst*, a copy of *src*.

isExecutable(*path*)
Return true if *path* is an executable.

isReadable(*path*)
Return true if *path* is readable by the user.

isWritable(*path*)
Return true if *path* is writable by the user.

makePath(*path*)
Make all directories that does not exist in a given *path*.

relpath(*path*)
Return *path* on Windows, and relative path elsewhere.

sympath(*path*, *beg=2*, *end=1*, *ellipsis='...'*)
Return 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()`[403] in *filename*. *protocol* is set to 2 for compatibility between Python 2 and 3.

unpickle(*filename*, ***kwargs*)
Unpickle object in *filename* using `pickle.load()`[404].

glob(**pathnames*)
Return concatenation of ordered lists of paths matching patterns in *pathnames*.

addext(*filename*, *extension*)
Return *filename*, with *extension* if it does not have one.

3.10.11 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`[405] 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*)
Return value corresponding to specified *key*, or *default* if *key* is not found.

[403] http://docs.python.org/library/pickle.html#pickle.dump
[404] http://docs.python.org/library/pickle.html#pickle.load
[405] http://docs.python.org/library/stdtypes.html#dict

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()
Return package path.

setPackagePath(*path*)
Set package path.

3.11 Applications API

This module contains ProDy applications.

3.11.1 Dynamics analysis

- `prody_anm()` (page **??**)
- `prody_gnm()` (page **??**)
- `prody_pca()` (page **??**)

3.11.2 Structure analysis

- `prody_align()` (page **??**)
- `prody_biomol()` (page **??**)
- `prody_blast()` (page **??**)
- `prody_catdcd()` (page **??**)
- `prody_contacts()` (page **??**)
- `prody_fetch()` (page **??**)
- `prody_select()` (page **??**)

3.11.3 Sequence analysis

- `evol_search()` (page **??**)
- `evol_fetch()` (page **??**)
- `evol_filter()` (page **??**)
- `evol_refine()` (page **??**)
- `evol_merge()` (page **??**)
- `evol_conserv()` (page **??**)

- `evol_coevol()` (page **??**)
- `evol_occupancy()` (page **??**)
- `evol_rankorder()` (page **??**)

3.11.4 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*[406]) – treat amino acids characters B, Z, J, and X as non- ambiguous, default is `True`
- **correction** (*str*[407]) – also save corrected mutual information matrix data and plot, one of `'apc','asc'`
- **normalization** (*str*[408]) – also save normalized mutual information matrix data and plot, one of `'sument','minent','maxent','mincon','maxcon','joint'`

Output Options

Parameters

- **heatmap** (*bool*[409]) – save heatmap files for all mutual information matrices
- **prefix** (*str*[410]) – output filename prefix, default is msa filename with _coevol suffix
- **numformat** (*str*[411]) – number output format, default is `'%12g'`

3.11.5 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*[412]) – treat amino acids characters B, Z, J, and X as non- ambiguous, default is `True`
- **omitgaps** (*bool*[413]) – do not omit gap characters, default is `True`

[406] http://docs.python.org/library/functions.html#bool
[407] http://docs.python.org/library/functions.html#str
[408] http://docs.python.org/library/functions.html#str
[409] http://docs.python.org/library/functions.html#bool
[410] http://docs.python.org/library/functions.html#str
[411] http://docs.python.org/library/functions.html#str
[412] http://docs.python.org/library/functions.html#bool
[413] http://docs.python.org/library/functions.html#bool

Output Options

Parameters

- **prefix** (*str*[414]) – output filename prefix, default is msa filename with _conserv suffix
- **numformat** (*str*[415]) – number output format, default is `'%12g'`

3.11.6 Pfam MSA Fetcher

Pfam MSA download application.

evol_fetch (*acc*, ***kwargs*)

Fetch MSA files from Pfam.

Parameters **acc** (*str*[416]) – Pfam accession or ID

Download Options

Parameters

- **alignment** (*str*[417]) – alignment type, one of `'full'`, `'seed'`, `'ncbi'`, `'metagenomics'`, default is `'full'`
- **format** (*str*[418]) – Pfam supported MSA format, one of `'selex'`, `'fasta'`, `'stockholm'`, default is `'selex'`
- **order** (*str*[419]) – ordering of sequences, one of `'tree'`, `'alphabetical'`, default is `'tree'`
- **inserts** (*str*[420]) – letter case for inserts, one of `'upper'`, `'lower'`, default is `'upper'`
- **gaps** (*str*[421]) – gap character, one of `'dashes'`, `'dots'`, `'mixed'`, default is `'dashes'`
- **timeout** (*int*[422]) – timeout for blocking connection attempts, default is `60`

Output Options

Parameters

- **folder** (*str*[423]) – output directory, default is `'.'`
- **outname** (*str*[424]) – output filename, default is accession and alignment type
- **compressed** (*bool*[425]) – gzip downloaded MSA file

[414] http://docs.python.org/library/functions.html#str
[415] http://docs.python.org/library/functions.html#str
[416] http://docs.python.org/library/functions.html#str
[417] http://docs.python.org/library/functions.html#str
[418] http://docs.python.org/library/functions.html#str
[419] http://docs.python.org/library/functions.html#str
[420] http://docs.python.org/library/functions.html#str
[421] http://docs.python.org/library/functions.html#str
[422] http://docs.python.org/library/functions.html#int
[423] http://docs.python.org/library/functions.html#str
[424] http://docs.python.org/library/functions.html#str
[425] http://docs.python.org/library/functions.html#bool

3.11.7 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*[426]) – sequence label starts with given words
- **endswith** (*bool*[427]) – sequence label ends with given words
- **contains** (*bool*[428]) – sequence label contains with given words

Filter Option

Parameters **filter_full** (*bool*[429]) – compare full label with word(s)

Output Options

Parameters

- **outname** (*str*[430]) – output filename, default is msa filename with _refined suffix
- **format** (*str*[431]) – output MSA file format, default is same as input
- **compressed** (*bool*[432]) – gzip refined MSA output

3.11.8 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*[433]) – output filename, default is first input filename with _merged suffix
- **format** (*str*[434]) – output MSA file format, default is same as first input MSA
- **compressed** (*bool*[435]) – gzip merged MSA output

[426] http://docs.python.org/library/functions.html#bool
[427] http://docs.python.org/library/functions.html#bool
[428] http://docs.python.org/library/functions.html#bool
[429] http://docs.python.org/library/functions.html#bool
[430] http://docs.python.org/library/functions.html#str
[431] http://docs.python.org/library/functions.html#str
[432] http://docs.python.org/library/functions.html#bool
[433] http://docs.python.org/library/functions.html#str
[434] http://docs.python.org/library/functions.html#str
[435] http://docs.python.org/library/functions.html#bool

3.11.9 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*[436]) – calculate row or column occupancy or both., one of `'row'`, `'col'`, `'both'`, default is `'row'`

Output Options

Parameters

- **prefix** (*str*[437]) – output filename prefix, default is msa filename with _occupancy suffix
- **label** (*str*[438]) – index for column based on msa label
- **numformat** (*str*[439]) – number output format, default is `'%12g'`

3.11.10 Identify Coevolving Pairs

Refine MSA application.

evol_rankorder (*mutinfo*, ***kwargs*)
Identify highly coevolving pairs of residues.

Parameters **mutinfo** – mutual information matrix

Input Options

Parameters

- **zscore** (*bool*[440]) – apply zscore for identifying top ranked coevolving pairs
- **delimiter** (*str*[441]) – delimiter used in mutual information matrix file
- **pdb** (*str*[442]) – PDB file that contains same number of residues as the mutual information matrix, output residue numbers will be based on PDB file
- **msa** (*str*[443]) – 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*[444]) – label in MSA file for output residue numbers

Output Options

Parameters

[436] http://docs.python.org/library/functions.html#str
[437] http://docs.python.org/library/functions.html#str
[438] http://docs.python.org/library/functions.html#str
[439] http://docs.python.org/library/functions.html#str
[440] http://docs.python.org/library/functions.html#bool
[441] http://docs.python.org/library/functions.html#str
[442] http://docs.python.org/library/functions.html#str
[443] http://docs.python.org/library/functions.html#str
[444] http://docs.python.org/library/functions.html#str

- **numpairs** (*int*[445]) – number of top ranking residue pairs to list, default is `100`
- **seqsep** (*int*[446]) – report coevolution for residue pairs that are sequentially separated by input value, default is `3`
- **dist** (*float*[447]) – 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 is `10.0`
- **usedist** (*bool*[448]) – use structural separation to report coevolving pairs
- **outname** (*str*[449]) – output filename, default is mutinfo_rankorder.txt

3.11.11 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*[450]) – sequence label, UniProt ID code, or PDB and chain identifier
- **seqid** (*float*[451]) – identity threshold for selecting unique sequences
- **colocc** (*float*[452]) – column (residue position) occupancy
- **rowocc** (*float*[453]) – row (sequence) occupancy
- **pdbres** (*bool*[454]) – keep columns corresponding to residues not resolved in PDB structure, applies label argument is a PDB identifier

Output Options

Parameters

- **outname** (*str*[455]) – output filename, default is msa filename with _refined suffix
- **format** (*str*[456]) – output MSA file format, default is same as input
- **compressed** (*bool*[457]) – gzip refined MSA output

[445] http://docs.python.org/library/functions.html#int
[446] http://docs.python.org/library/functions.html#int
[447] http://docs.python.org/library/functions.html#float
[448] http://docs.python.org/library/functions.html#bool
[449] http://docs.python.org/library/functions.html#str
[450] http://docs.python.org/library/functions.html#str
[451] http://docs.python.org/library/functions.html#float
[452] http://docs.python.org/library/functions.html#float
[453] http://docs.python.org/library/functions.html#float
[454] http://docs.python.org/library/functions.html#bool
[455] http://docs.python.org/library/functions.html#str
[456] http://docs.python.org/library/functions.html#str
[457] http://docs.python.org/library/functions.html#bool

3.11.12 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*[458]) – search Pfam-B families
- **skip_a** (*bool*[459]) – do not search Pfam-A families
- **ga** (*bool*[460]) – use gathering threshold
- **evalue** (*float*[461]) – e-value cutoff, must be less than 10.0
- **timeout** (*int*[462]) – timeout in seconds for blocking connection attempt, default is `60`

Output Options

Parameters

- **outname** (*str*[463]) – name for output file, default is standard output
- **delimiter** (*str*[464]) – delimiter for output data columns, default is `'\t'`

3.11.13 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 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 **??**)
- **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`

[458] http://docs.python.org/library/functions.html#bool
[459] http://docs.python.org/library/functions.html#bool
[460] http://docs.python.org/library/functions.html#bool
[461] http://docs.python.org/library/functions.html#float
[462] http://docs.python.org/library/functions.html#int
[463] http://docs.python.org/library/functions.html#str
[464] http://docs.python.org/library/functions.html#str

3.11.14 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

- **cutoff** – cutoff distance (A), default is `15.0`
- **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 `1.0`
- **hessian** – write Hessian matrix, default is `False`
- **kirchhoff** – write Kirchhoff matrix, 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`
- **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`
- **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"`

3.11.15 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

3.11.16 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*[465]) – percent sequence identity for blast search, default is 90.0
- **overlap** (*float*[466]) – percent sequence overlap between sequences, default is 90.0
- **outdir** (*str*[467]) – download uncompressed PDB files to given directory
- **gzip** – write compressed PDB file

Blast Parameters

Parameters

- **filename** (*str*[468]) – a *filename* to save the results in XML format
- **hitlist_size** (*int*[469]) – search parameters, default is 250
- **expect** (*float*[470]) – search parameters, default is 1e-10
- **sleep** (*int*[471]) – how long to wait to reconnect for results, default is 2 sleep time is doubled when results are not ready.
- **timeout** (*int*[472]) – when to give up waiting for results. default is 30

[465] http://docs.python.org/library/functions.html#float
[466] http://docs.python.org/library/functions.html#float
[467] http://docs.python.org/library/functions.html#str
[468] http://docs.python.org/library/functions.html#str
[469] http://docs.python.org/library/functions.html#int
[470] http://docs.python.org/library/functions.html#float
[471] http://docs.python.org/library/functions.html#int
[472] http://docs.python.org/library/functions.html#int

3.11.17 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

3.11.18 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

3.11.19 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 `False`

3.11.20 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

- **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 `1.0`
- **kirchhoff** – write Kirchhoff matrix, 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`
- **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`
- **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"

3.11.21 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
- **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
- **nmodes** – number of non-zero eigenvectors (modes) to calculate, default is 10
- **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
- **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"

3.11.22 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 **??**)
- **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`

3.12 Configuration & Logging

This module defines functions for logging in files, configuring ProDy, and running tests.

- `confProDy()` (page **??**)
- `checkUpdates()` (page **??**)
- `startLogfile()` (page **??**)
- `closeLogfile()` (page **??**)
- `plog()` (page **??**)

confProDy(**args*, ***kwargs*)
Configure ProDy.

Option	Default (acceptable values)
auto_secondary	False
auto_show	True
backup	False
backup_ext	'.BAK'
check_updates	0
ligand_xml_save	False
local_pdb_folder	'' See also `pathPDBFolder()` (page **??**).
pdb_mirror_path	'' See also `pathPDBMirror()` (page **??**).
selection_warning	True
typo_warnings	True
verbosity	'debug' ('critical', 'debug', 'error', 'info', 'none', 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 4

Developer's Guide

4.1 Contributing to ProDy

4.1.1 Install Git and a GUI

ProDy source code is managed using Git[1] 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[2].

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[3] and Git docs[4] for usage details and examples.

4.1.2 Fork and Clone ProDy

ProDy source code an issue tracker are hosted on Github[5]. You need to create an account on this service, if you do not have one already.

[1] http://git-scm.com/downloads
[2] http://git-scm.com/downloads
[3] http://git-scm.com/book
[4] http://git-scm.com/docs
[5] http://github.com/prody/ProDy

If you work on Mac OS or Windows, you may consider getting GitHub Mac[6] or GitHub Windows[7] 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[8] source code page. You will have write access to this fork and later will use it share your changes with others.

The next step is cloning the fork from your online account to your local system. If you are not using the GitHub software, you can do it as follows:

```
$ git clone https://github.com/prody/ProDy.git
```

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`[9] 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 **??**) from anywhere on your system. One way to do this by adding `ProDy/scripts` folder to your `PATH`[10] 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 **??**). 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:

[6] http://mac.github.com
[7] http://windows.github.com
[8] http://prody.csb.pitt.edu
[9] http://matplotlib.sourceforge.net/faq/environment_variables_faq.html#envvar-PYTHONPATH
[10] http://matplotlib.sourceforge.net/faq/environment_variables_faq.html#envvar-PATH

```
$ cd ProDy
$ nosetests
```

See *Testing ProDy* (page **??**) 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:

```
$ 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 Bitbucket account:

```
git push origin master
```

This step will ask for your account user name. If you are going to push to your GitHub/Bitbucket 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 or *Manage Account → SSH keys* page on Bitbucket.

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[11] 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:abakan/ProDy.git
```

or:

```
$ cd prody
$ git remote add prodymaster git@bitbucket.org:abakan/prody.git
```

You may use any name other than *prodymaster*, but *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[12] 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:

[11] http://prody.csb.pitt.edu
[12] http://prody.csb.pitt.edu

```
$ python setup.py build_ext --inplace --force
```

4.2 Documenting ProDy

ProDy documentation is written using reStructuredText[13] markup and prepared using Sphinx[14]. You may install Sphinx using **easy_install**, i.e. `easy_install -U Sphinx`, or using package manager on your Linux machine.

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[15] 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[16] 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.

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 other software in the

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 **??**) for more on testing.

2. Update the version number in:

[13] http://docutils.sf.net/rst.html
[14] http://sphinx.pocoo.org/
[15] http://docutils.sf.net/rst.html
[16] https://readthedocs.org/

- `prody/__init__.py`

Also, commend `+ '-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[17].

5. Generate the source distributions:

```
$ cd ..
$ python setup.py sdist --formats=gztar,zip
```

6. Prepare and test Windows installers (see *Making Windows Installers* (page **??**)):

```
$ C:\Python26\python setup.py bdist_wininst
$ C:\Python27\python setup.py bdist_wininst
$ C:\Python32\python setup.py bdist_wininst
$ C:\Python33\python setup.py bdist_wininst
```

   Alternatively, use **bdist_wininst.bat** to run these commands. When there is a newer Python major release, it should be added to this list.

7. Register new release to PyPI:

```
$ python setup.py register
```

8. Upload the new release files to the PyPI[18].
9. Commit final changes, if there are any:

```
$ cd ..
$ git commit -a
```

10. Tag the repository with the current version number:

```
$ git tag vX.Y
```

11. Rebase `devel` branch to `master`:

```
$ git checkout master
$ git rebase devel
```

12. Push the changes with the new tag:

[17] http://github.com/prody/ProDy
[18] http://pypi.python.org/pypi/ProDy

```
$ git checkout master
$ git push --tags
$ git checkout devel
$ git push --tags
```

13. Finally, update the documentation on ProDy[19] website. See *Documenting ProDy* (page **??**).
14. Now that you made a release, you can go back to development. You may stat with append `'-dev'` to `__release__` in `prody/__init__.py`.

4.4 Style Guide for ProDy

4.4.1 Introduction

PEP 8[20], the *Style Guide for Python Code*, is adopted in the development of ProDy package. Contributions to ProDy shall follow **PEP 8**[21] 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**[22] recommendations regarding imports, the following should be applied:

- relative intra-ProDy imports are discouraged, use `from prody.atomic import AtomGroup` not `from atomic import AtomGroup`

[19] http://prody.csb.pitt.edu
[20] http://www.python.org/dev/peps/pep-0008
[21] http://www.python.org/dev/peps/pep-0008
[22] http://www.python.org/dev/peps/pep-0008#imports

- 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**[23]), 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 **??**) for Anisotropic Network Model
- `HierView` (page **??**) 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 **??**), 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()`

[23] http://www.python.org/dev/peps/pep-0008#whitespace-in-expressions-and-statements

(page **??**), `wrapAtoms()` (page **??**), `assignSecstr()` (page **??**), and `calcSubspaceOverlap()` (page **??**).

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 **??**) is preferred over `buildHessianMatrix()`. Another example is the change from using `getResidueNames()` to using `AtomGroup.getResnames()` (page **??**). In fact, this was part of a series of major *Release Notes* (page **??**) 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 **??**) and `buildDistMatrix()` (page **??**)
- `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 **??**)
- `getSomething()`: methods, and sometimes functions, that return a copy of data should start with `get`, such as `listReservedWords()` (page **??**)
- `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[24]. 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
```

[24] http://nose.readthedocs.org

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`[25].
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
```

[25] http://docs.python.org/library/unittest.html#unittest.TestCase

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[26], an easy-to-read/write file format. See reStructuredText Primer[27] for a quick introduction.

reStructuredText is stored in plain-text files with `.rst` extension, and converted to HTML and PDF pages using Sphinx[28].

`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.

---

[26] http://docutils.sourceforge.net/rst.html
[27] http://sphinx-doc.org/rest.html
[28] http://sphinx-doc.org/

4.6.4 Including Code

Python code in tutorials should be included using IPython Sphinx directive[29]. 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:

```
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()`[30] 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')
```

[29] http://ipython.org/ipython-doc/dev/development/ipython_directive.html
[30] http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot

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.

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 Making Windows Installers

MinGW[31] can be used for compiling C modules when making Windows installers. Install MinGW and make `distutils.cfg` file in `PythonXY\Lib\distutils` folder that contains:

```
[build]
compiler = mingw32
```

4.8 Cross-platform Issues

This section describes cross-platform issues that may emerge and provides possible solutions for them.

[31] http://www.mingw.org/

4.8.1 Numpy integer type

Issues may arise when comparing Numpy integer types with Python `int()`[32]. Python `int()`[33] 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 `False` resulting in unexpected behavior in ProDy functions or methods. If Numpy integer type needs to be specified, using `int` seems a safe option.

4.8.2 Relative paths

`os.path.relpath()`[34] 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()`[35], ProDy function `relpath()` (page **??**) should be used to avoid problems.

[32] http://docs.python.org/library/functions.html#int
[33] http://docs.python.org/library/functions.html#int
[34] http://docs.python.org/library/os.path.html#os.path.relpath
[35] http://docs.python.org/library/os.path.html#os.path.relpath

CHAPTER 5

Release Notes

5.1 ProDy 1.5 Series

5.1.1 1.5 (Dec 23, 2013)

New Features:

- `buildDirectInfoMatrix()` (page **??**) and `calcMeff()` (page **??**) are implemented for calculation of direct information from multiple sequence alignments.
- `showDirectInfoMatrix()` (page **??**) and `showSCAMatrix()` (page **??**) functions are implemented for displaying coevolutionary data.
- `RTB` (page **??**) 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[1].

Improvements:

- `DCDFile` (page **??**) and `parseDCD()` (page **??**) support DCD files written by **cpptraj**.

Testing:

- ProDy test command (**prody test**) and function `prody.test()`[2] has been removed for easier maintenance of testing functions. See *Testing ProDy* (page **??**) for more information on how to test ProDy.

[1] http://github.com/prody/ProDy
[2] http://prody.csb.pitt.edu/reference/prody.html#prody.test

5.2 ProDy 1.4 Series

5.2.1 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 **??**) can handle `Residue` (page **??**) 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 **??**) 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 **??**) 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 **??**) and `buildSeqidMatrix()` (page **??**).

5.2.2 1.4.8 (Nov 4, 2013)

New Features:

- New analysis functions `buildOMESMatrix()` (page **??**) and `buildSCAMatrix()` (page **??**) are implemented.
- New `AtomGroup.numBytes()` (page **??**) method returns an estimate of memory usage.
- New `countBytes()` (page **??**) utility function is added for counting bytes used by NumPy arrays.

Improvements:

- `parsePDB()` (page **??**) resizes data arrays to decrease memory usage.

Bugfixes:

- Fixed memory leaks in MSA `analysis` (page **??**) functions.
- Fixed potential problems with importing contributed libraries.

5.2.3 1.4.7 (Oct 29, 2013)

Improvements:

- `AtomGroup` (page **??**), `Selection` (page **??**), and other `Atomic` (page **??**) classes are picklable.
- Improved equality tests for `AtomGroup` (page **??**). Two different instances are considered equal if they contain identical data and coordinate sets.

5.2.4 1.4.6 (Oct 16, 2013)

Bugfixes:

- Selection problem with using *resid* is fixed (issue 160[3])
- 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 **??**) to use PDB for fetching XML files.
- Revised handling of MSA file formats to avoid exceptions for unknown extensions.

[3] https://github.com/prody/ProDy/issues/160

5.2.5 1.4.5 (Sep 6, 2013)

New Features:

- `parsePDBHeader()` (page **??**) 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 **??**) 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 **??**) function that prevented reading files.
- Improved parsing of biomolecular transformations.
- Fixed memory allocation in C code used by `parseMSA()` (page **??**) (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 **??**) instances in Python 3 series.

5.2.6 1.4.4 (July 22, 2013)

Improvements:

- `writeNMD()` (page **??**) and `parseNMD()` (page **??**) 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 **??**) that would cause `KeyError` when bonds are set but fragments are not determined is fixed.
- Import ProDy would fail when `HOME`[4] is not set. Changed `PackageSettings` (page **??**) to handle this case graciously.

5.2.7 1.4.3 (June 14, 2013)

Changes:

- `getVMDpath()` (page **??**) and `setVMDpath()` (page **??**) functions are deprecated for removal, use `pathVMD()` (page **??**) instead.
- Increased `blastPDB()` (page **??**) *timeout* to 60 seconds.

[4] http://matplotlib.sourceforge.net/faq/environment_variables_faq.html#envvar-HOME

- `extendModel()` (page **??**) and `extendMode()` (page **??**) functions have a new option for normalizing extended mode(s).
- `sampleModes()` (page **??**) and `traverseMode()` (page **??**) automatically normalizes input modes.

Bugfixes:

- A bug in `applyTransformation()` (page **??**) is fixed. The function would interpret some external transformation matrices incorrectly.
- A bug in `fetchPDBLigand()` (page **??**) function is fixed.

5.2.8 1.4.2 (April 19, 2013)

Improvements:

- `fetchPDB()` (page **??**) and `fetchPDBfromMirror()` (page **??**) functions can handle partial PDB mirrors. See `pathPDBMirror()` (page **??**) for setting a mirror path.

Changes:

- MSE[5] 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 **??**).
- Fasta files with sequence labels using multiple pipe characters would cause C parser (and so `parseMSA()` (page **??**)) 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 **??**) would throw an exception. This case is handled by slicing the chain identifier string.
- A problem in `viewNMDinVMD()` (page **??**) related to module imports is fixed.
- A problem with handling weights in `loadEnsemble()` (page **??**) is fixed.

5.2.9 1.4.1 (Dec 16, 2012)

New Features:

- `buildSeqidMatrix()` (page **??**) and `uniqueSequences()` (page **??**) functions are implemented for comparing sequences in an `MSA` (page **??**) object.
- `showHeatmap()` (page **??**), `parseHeatmap()` (page **??**), and `writeHeatmap()` (page **??**) functions are implemented to support VMD plugin Heat Mapper[6] file format.
- `Sequence` (page **??**) is implemented to handle individual sequence records and point to sequences in `MSA` (page **??**) instances.
- *evol occupancy* (page **??**) application is implemented for refined MSA quality checking purposes.
- `mergeMSA()` (page **??**) function and *evol merge* (page **??**) application are implemented for merging Pfam MSA to study multi-domain proteins.

Improvements:

[5] http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=MSE
[6] http://www.ks.uiuc.edu/Research/vmd/plugins/heatmapper/

- `refineMSA()` (page **??**) function and *evol refine* (page **??**) application can perform MSA refinements by removing similar sequences.
- `writePDB()` (page **??**) function takes *beta* and *occupancy* arguments to be outputted in corresponding columns.
- `MSA` (page **??**) indexing and slicing are revised and improved.
- `parseMSA()` (page **??**) 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 **??**), *prody gnm* (page **??**), and *prody pca* (page **??**) 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 **??**), and use the default US server.
- Indexing `MSA` (page **??**) returns `Sequence` (page **??**) instances.
- Iterating over `MSA` (page **??**) and `MSAFile` (page **??**) yields `Sequence` (page **??**) instances.

Bugfixes:

- Fixed a syntax problem that prevented running ProDy using Python 2.6.
- Fixed `NMA` (page **??**) 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.

5.2.10 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 **??**).
- 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`[7] features.

Sequence Analysis

- New applications *Evol Applications* (page **??**) are available.

[7] http://docs.python.org/library/unittest.html#unittest

- `searchPfam()` (page **??**) and `fetchPfamMSA()` (page **??**) functions are implemented for searching and retrieving Pfam data. See *MSA Files*[8] for usage examples.
- `MSAFile` (page **??**) class, `parseMSA()` (page **??**) and `writeMSA()` (page **??**) functions are implemented for reading and writing multiple sequence alignments. See *MSA Files*[9] for usage examples.
- `MSA` (page **??**) class has been implemented for storing and manipulating MSAs in memory.
- `calcShannonEntropy()` (page **??**), `buildMutinfoMatrix()` (page **??**), and `calcMSAOccupancy()` (page **??**) functions are implemented implemented for MSA analysis. See *Evolution Analysis*[10] for usage examples.
- `showShannonEntropy()` (page **??**), `showMutinfoMatrix()` (page **??**), and `showMSAOccupancy()` (page **??**) functions are implemented implemented for MSA analysis. See *Evolution Analysis*[11] for usage examples.
- `applyMutinfoCorr()` (page **??**) and `applyMutinfoNorm()` (page **??**) functions are implemented for applying normalization and corrections to mutual information matrices.
- `calcRankorder()` (page **??**) function is implemented for identifying highly correlated/co-evolving pairs of residues.

Bugfix:

- Fixed selection issues involving use of `x` or negative numbers.

5.3 ProDy 1.3 Series

5.3.1 1.3.1 (Nov 6, 2012)

New Features:

- Added `fetchPDBviaHTTP()` (page **??**) and `fetchPDBviaFTP()` (page **??**) functions.
- Added `copyFile()` (page **??**) function to `utilities` (page **??**).
- Added `prody test` command for convenient testing of ProDy package.

Improvements:

- Improved `gunzip()` (page **??**) function to handle `.gz` extensions and string buffers.

Changes:

- `getWWPDBFTPServer()` and `setWWPDBFTPServer()` are deprecated for removal in v1.4, use `wwPDBServer()` (page **??**) instead.
- `getPDBLocalFolder()` and `setPDBLocalFolder()` are deprecated for removal in v1.4, use `pathPDBFolder()` (page **??**) instead.

[8] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msafiles.html#msafiles
[9] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msafiles.html#msafiles
[10] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msaanalysis.html#msa-analysis
[11] http://prody.csb.pitt.edu/tutorials/evol_tutorial/msaanalysis.html#msa-analysis

- `getPDBMirrorPath()` and `setPDBMirrorPath()` are deprecated for removal in v1.4, use `pathPDBMirror()` (page **??**) instead.
- `getPDBCluster()` is deprecated for removal in v1.4, use `listPDBCluster()` (page **??**) instead.
- `getReservedWords()` is deprecated for removal in v1.4, use `listReservedWords()` (page **??**) instead.
- `getNonstdProperties()` (page **??**) is deprecated for removal in v1.4, use `listNonstdAAProps()` (page **??**) instead.

Bugfix:

- Fixed a bug in `HierView` (page **??**) 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 **??**) and `Trajectory.setAtoms()` (page **??**) methods to avoid indexing problems.

5.3.2 1.3 (Sep 30, 2012)

Improvements:

- `select` (page **??**) module and its documentation are completely rewritten. `Select` (page **??**) class uses simplest possible parser to evaluate selection strings and achieves more than 25% speed-up on average.
- *Atom Selections* (page **??**) become more forgiving of small typos, but will issue warning messages when they are detected via `SelectionWarning` (page **??**). These messages can be turned of using `confProDy()` (page **??**)
- Functions used in *ProDy Applications* (page **??**) have been refactored to allow for using them directly. See `apps` (page **??**) for their documentation.

Bugfix:

- A problem in *prody catdcd* (page **??**) command that was introduced when refactoring `trajectory` (page **??**) classes is fixed.

5.4 ProDy 1.2 Series

5.4.1 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`[12] module regarding Numpy numeric types is fixed. Problem would emerge on platforms which do not offer some numeric types, e.f. `np.float16`.

[12] http://docs.python.org/library/select.html#select

- Fixed problems in *prody anm* (page **??**), *prody gnm* (page **??**), and *prody fetch* (page **??**) related to writing output files.

Changes:

- The way that *prody fetch* (page **??**) command handles files containing PDB identifiers has changed.

5.4.2 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.

Some changes in `Trajectory` (page **??**) and `Ensemble` (page **??**) 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 **??**) instances can be linked to a `Trajectory` (page **??**) using `Trajectory.link()` (page **??**) method and linking status of an instance can be checked using `Trajectory.isLinked()` (page **??**) medhod.
- `Trajectory.setAtoms()` (page **??**) method accepts `AtomGroup` (page **??**) and `Selection` (page **??**) instances and should be used to select a subset of atoms. This method will not link `AtomGroup` (page **??**) instance to the trajectory and also will not update the reference coordinates of the instance.
- `Trajectory.select()` and `Ensemble.select()` methods are removed and their functions are overloaded to `Trajectory.setAtoms()` (page **??**) and `Ensemble.setAtoms()` (page **??**) methods, respectively.
- `Trajectory.getSelection()` and `Ensemble.getSelection()` methods are removed, use `Trajectory.getAtoms()` (page **??**) and `Ensemble.getAtoms()` (page **??**) instead.
- `Trajectory` (page **??**) reference coordinates must be changed using `Trajectory.setCoords()` (page **??**) method.

For usage examples see *Trajectory Analysis*[13], *Trajectory Analysis II*[14], *Frames and Atom Groups*[15], and *Trajectory Output*[16].

New Features:

- *Atom Flags* (page **??**), that are used in *Atom Selections* (page **??**), is implemented. See its documentation for handy usage examples.
- `sortAtoms()` (page **??**) function is implemented.
- `pickCentralConf()` (page **??**) function is implemented to pick the conformation or the active coordinate set that is closest to the average of coordinate sets.
- `writePSF()` (page **??**), a simple PSF file writer, is implemented.
- `glob()` (page **??**) utility function is implemented.
- `iterPDBFilenames()` (page **??**) function is implemented, which can be used to iterate over all PDB files stored in a local mirror of Protein Data Bank.
- `findPDBFiles()` (page **??**) function is implemented, which can be used to access PDB files in a path.

[13] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory.html#trajectory
[14] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory2.html#trajectory2
[15] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/frame.html#frame
[16] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/outputtraj.html#outputtraj

Improvements:

- `HierView` (page **??**) instances are built more efficiently. Two times speed-up is achieved by delaying instantiation of `Chain` (page **??**) and `Residue` (page **??**) instances until they are needed.
- Multiple *Atom Flags* (page **??**) can be used in *Atom Selections* (page **??**) without using `'and'` operator, e.g. `'sidechain carbon'` is the same as `'sidechain and carbon'`.
- `writePDB()` (page **??**) accepts `Ensemble` (page **??**), `Conformation` (page **??**), and `Frame` (page **??**) instances as atoms argument.
- `writePDB()` (page **??**) function is around 25% faster.
- `pickCentral()` (page **??**) is extended to accept `Atomic` (page **??**) and `Ensemble` (page **??**) instances. Old function is now `pickCentralAtom()` (page **??**).
- *prody align* (page **??**) command and `prody_align()` (page **??**) function can handle non-protein atom selections (see examples for *prody align* (page **??**)).
- `parsePDB()` (page **??**) and `writePDB()` (page **??**) supports 100K and more atoms.

Changes:

- `showOverlapTable()` (page **??**) displays first set of modes along x axis of the plot.
- `AtomGroup.setData()` (page **??**) does not accept arrays with boolean data type, use `AtomGroup.setFlags()` (page **??**) instead.
- `writePDB()` (page **??**) function argument *model* is changed to *csets* that indicates the coordinate set index of *atoms* argument.
- `PackageLogger.timing()` (page **??**) does not return elapsed time, only logs this information.
- `PackageLogger.startLogfile()` is deprecated for removal in v1.3, use `PackageLogger.start()` (page **??**) instead.
- `PackageLogger.closeLogfile()` is deprecated for removal in v1.3, use `PackageLogger.close()` (page **??**) 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 **??**) appends `.pdb` extension to filename when it is not present
- *prody select* (page **??**) 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 **??**) 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 **??**) instead which save changes permanently.
- `NMA.getModes()` and `ModeSet.getModes()` methods are deprecated for removal in v1.3, use `list()`[17], e.g. `list(model)`, instead.

[17] http://docs.python.org/library/functions.html#list

Bugfixes:

- Fixed a bug in *prody contacts* (page **??**) 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.

5.5 ProDy 1.1 Series

5.5.1 1.1 (June 1, 2012)

New Features:

- `iterFragments()` (page **??**) function is added.
- `findNeighbors()` (page **??**) function is added.
- `calcMSF()` (page **??**) and `calcRMSF()` (page **??**) functions are added.
- `wrapAtoms()` (page **??**) functions is added.
- `extendMode()` (page **??**) and `extendVector()` (page **??**) functions are added.
- *prody contacts* (page **??**) command is added.

Improvements:

- `moveAtoms()` (page **??**) function is improved to move atoms to a specified location.
- `DCDFile` (page **??**) and `parseDCD()` (page **??**) 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 **??**), *prody gnm* (page **??**), and *prody pca* (page **??**) 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 **??**) is normalized by default.
- `tools` module is depracated for removal, use `utilities` (page **??**) instead.
- *array* argument in `moveAtoms()` (page **??**) is replaced with *by* keyword argument.
- *which* argument in `AtomGroup.copy()` (page **??**) method is deprecated for removal in version 1.2.

- `DCDFile` (page **??**) 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 **??**) instead.

Bugfixes:

- Fixed several problems in `iterNeighbors()` (page **??**) function and `Contacts` (page **??**) class that were introduced after transition to new `KDTree` (page **??**) interface.
- Fixed a problem in setting selection strings of fragments identified using `findFragments()` (page **??**).
- Fixed a problem in `calcCenter()` (page **??**) related to weighted center calculation.
- Fixed a problem of in copying `AtomMap` (page **??**) 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.

5.6 ProDy 1.0 Series

5.6.1 1.0.4 (May 2, 2012)

Bugfixes:

- Fixed a problem in `calcPhi()` (page **??**) function that raised a name error.
- Fixed a problem in `KDTree.getDistances()` (page **??**) method that raised a name error when unitcell is provided.
- Fixed a problem in `buildDistMatrix()` (page **??**) and `calcDistance()` (page **??**) functions causing miscalculations when unitcell is given.
- Revised `KDTree` (page **??**) 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.

5.6.2 1.0.3 (May 1, 2012)

Bugfixes:

- Fixed `kdtree` (page **??**) import problem.

New Features:

- `buildDistMatrix()` (page **??**) function that can take periodic boundary conditions is implemented.

Improvements:

- `calcDistance()` (page **??**) function is improved to take periodic boundary conditions into account when provided by the users.

5.6.3 1.0.2 (May 1, 2012)

New Features:

- Methods to deal with connected subsets of atoms are implemented, see `AtomGroup.iterFragments()` (page **??**) and `AtomGroup.numFragments()` (page **??**).
- `pickCentral()` (page **??**) 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 **??**) and `confProDy()` (page **??**) for usage details.
- **prody align** makes use of `--select` when aligning multiple structures. See usage examples: *prody align* (page **??**)
- `printRMSD()` (page **??**) function that prints minimum, maximum, and mean RMSD values when comparing multiple coordinate sets is implemented.
- `findFragments()` (page **??**) function that identifies fragments in atom subsets, e.g. `Selection` (page **??**), is implemented.
- A new `KDTree` (page **??**) interface with coherent method names and capability to handle periodic boundary conditions is implemented.

Improvements:

- Performance improvements made in `saveAtoms()` (page **??**) and `loadAtoms()` (page **??**).
- `sliceMode()` (page **??**), `sliceModel()` (page **??**), `sliceVector()` (page **??**), and `reduceModel()` (page **??**) functions accept `Selection` (page **??**) 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 **??**)) is improved to yield items faster.

Changes:

- There is a change in the behavior of addition operation on instances of `AtomGroup` (page **??**). 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 **??**) 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 **??**) and `Residue` (page **??**) selection strings that would emerge when a `HierView` (page **??**) is build using a selection is fixed.
- A problem with copying `AtomGroup` (page **??**) instances whose coordinates are not set is fixed.
- `AtomGroup` (page **??**) 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 **??**) instances in `pickCentral()` (page **??**) function is fixed.
- A problem in `Select` (page **??**) 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 **??**)) to a `Trajectory` (page **??**) after atoms were set (`setAtoms()` (page **??**)). Newly added file would not be associated with the atoms and coordinates parsed from this file would not be set for the `AtomGroup` (page **??**) instance.

5.6.4 1.0.1 (Apr 6, 2012)

New Features:

- ProDy can be configured to automatically check for updates on a regular basis, see `checkUpdates()` (page **??**) and `confProDy()` (page **??**) functions for details.
- `alignPDBEnsemble()` (page **??**) function is implemented to align PDB files using transformations calculated in ensemble analysis. See usage example in *Homologous Proteins*[18] example.
- `PDBConformation.getTransformation()` (page **??**) 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 **??**) for usage details.
- `calcCrossProjection()` (page **??**) function is implemented.

Improvements:

- `Select` (page **??**) class raises a `SelectionError` 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 **??**) and `showCrossProjection()` (page **??**) functions are improved to evaluate list of markers, color, labels, and texts. See usage example in *Plotting*[19].

[18] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/blast.html#pca-blast
[19] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_plotting.html#pca-xray-plotting

- `Trajectory` (page **??**) instances can be used for calculating and plotting projections using `calcProjection()` (page **??**), `showProjection()` (page **??**), `calcCrossProjection()` (page **??**), and `showCrossProjection()` (page **??**) 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 **??**) for definitions of *protein* and *acidic* keywords.
- Hit dictionaries from `PDBBlastRecord` (page **??**) 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 **??**) method instead.

Bugfixes:

- A bug in some *ProDy Applications* (page **??**) 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.

5.6.5 1.0 (Mar 7, 2012)

Improvements:

- `ANM.buildHessian()` (page **??**) 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 **??**) 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 **??**) for script usage details.
- Added `bonded to ...` selection method that expands a selection to immediately bound atoms. See *Atom Selections* (page **??**) for its description.
- `fetchPDBLigand()` (page **??**) parses bond data from the XML file.
- `fetchPDBLigand()` (page **??**) can optionally save compressed XML files into ProDy package folder so that frequent access to same files will be more rapid. See `confProDy()` (page **??**) function for setting this option.
- `Select` (page **??**) 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 **??**) 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 **??**) and `Mode.getEigval()` (page **??**) instead.

- `getEigenvectors()` and `getEigenvalues()` methods are deprecated for removal in v1.1, use `NMA.getEigvecs()` (page **??**) and `NMA.getEigvals()` (page **??**) instead.
- `Mode.getCovariance()` and `ModeSet.getCovariance()` methods are deprecated for removal in v1.1, use `calcCovariance()` (page **??**) method instead.
- `Mode.getCollectivity()` method is removed, use `calcCollectivity()` (page **??**) function instead.
- `Mode.getFractOfVariance()` method is removed, use the new `calcFractVariance()` (page **??**) function instead.
- `Mode.getSqFlucts()` method is removed, use `calcSqFlucts()` (page **??**) function instead.
- Renamed `showFractOfVar()` function as `showFractVars()` (page **??**) function instead.
- Removed `calcCumOverlapArray()`, use `calcCumulOverlap()` (page **??**) with `array=True` argument instead.
- Renamed `extrapolateModel()` as `extendModel()` (page **??**).
- The relation between `AtomGroup` (page **??**), `Trajectory` (page **??**), and `Frame` (page **??**) instances have changed. See *Trajectory Analysis II*[20] and *Trajectory Output*[21], and *Frames and Atom Groups*[22] usage examples.
- `AtomGroup` (page **??**) cannot be deformed by direct addition with a vector instance.
- Unmapped atoms in `AtomMap` (page **??**) instances are called dummies. `AtomMap.numUnmapped()` method, for example, is renamed as `AtomMap.numDummies()` (page **??**).
- `fetchPDBLigand()` (page **??**) 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 **??**) so that when a path for downloading files are given files are not save to local PDB folder.

5.7 ProDy 0.9 Series

5.7.1 0.9.4 (Feb 4, 2012)

Changes:

[20] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory2.html#trajectory2
[21] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/outputtraj.html#outputtraj
[22] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/frame.html#frame

- `setAtomGroup()` and `getAtomGroup()` methods are renamed as `Ensemble.setAtoms()` (page **??**) and `Ensemble.getAtoms()` (page **??**).
- `AtomGroup` (page **??**) 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 **??**) and `Trajectory` (page **??**) classes.

Bugfixes:

- Bugfixes in `Bond.setACSIndex()` (page **??**), `saveAtoms()` (page **??**), and `HierView.getSegment()` (page **??**).
- Bugfixes in `GammaVariableCutoff` (page **??**) and `GammaStructureBased` (page **??**) classes.
- Bugfix in `calcCrossCorr()` (page **??**) function.
- Bugfixes in `Ensemble.getWeights()` (page **??**), `showOccupancies()` (page **??**), `DCDFile.flush()` (page **??**).
- Bugfixes in ProDy commands *prody blast* (page **??**), *prody fetch* (page **??**), and *prody pca* (page **??**).
- Bugfix in `calcCenter()` (page **??**) function.

5.7.2 0.9.3 (Feb 1, 2012)

New Features:

- `DBRef` (page **??**) class is implemented for storing references to sequence databases parsed from PDB header records.
- Methods for storing coordinate set labels in `AtomGroup` (page **??**) instances are implemented: `getACSLabel()` (page **??**), and `getACSLabel()` (page **??**).
- `calcCenter()` (page **??**) and `moveAtoms()` (page **??**) functions are implemented for dealing with coordinate translation.
- Hierarchical view, `HierView` (page **??**), 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`[23] 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 **??**) 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 **??**) slicing is implemented:

```
p38 = parsePDB('1p38')
chA = p38['A']
res_4to10 = chA[4:11]
res_100toLAST = chA[100:]
```

[23] http://docs.python.org/library/collections.html#collections.OrderedDict

- Some support for bonds is implemented to `AtomGroup` (page **??**) class. Bonds can be set using `setBonds()` (page **??**) method. All bonds must be set at once. `iterBonds()` (page **??**) or `iterBonds()` (page **??**) methods can be used to iterate over bonds in an AtomGroup or an Atom.
- `parsePSF()` (page **??**) parses bond information and sets to the atom group.
- `Selection.update()` (page **??**) method is implemented, which may be useful to update a distance based selection after coordinate changes.
- `buildKDTree()` and `iterNeighbors()` (page **??**) methods are implemented for facilitating identification of pairs of atoms that are proximal.
- `iterAtoms()` (page **??**) method is implemented to all `atomic` (page **??**) classes to provide uniformity for atom iterations.
- `calcAngle()` (page **??**), `calcDihedral()` (page **??**), `calcPhi()` (page **??**), `calcPsi()` (page **??**), and `calcOmega()` (page **??**) methods are implemented.

Improvements:

- `Chain.getSelstr()` (page **??**) and `Residue.getSelstr()` (page **??**) methods are improved to include the selection string of a `Selection` (page **??**) when they are built using one.

Changes:

- `Residue` (page **??**) methods `getNumber()`, `setNumber()`, `getName()`, `setName()` methods are deprecated and will be removed in v1.0.
- `Chain` (page **??**) methods `getIdentifier()` and `setIdentifier()` methods are deprecated and will be removed in v1.0.
- `Polymer` (page **??**) attribute `identifier` is renamed as `chid` (page **??**).
- `Chemical` (page **??**) attribute `identifier` is renamed as `resname` (page **??**).
- `getACSI()` and `setACSI()` are renamed as `getACSIndex()` (page **??**) and `setACSIndex()` (page **??**), respectively.
- `calcRadiusOfGyration()` is deprecated and will be removed in v1.0. Use `calcGyradius()` (page **??**) instead.

Bugfixes:

- Fixed a problem in `parsePDB()` (page **??**) that caused loosing existing coordinate sets in an `AtomGroup` (page **??**) when passed as *ag* argument.
- Fixed a problem with `"same ... as ..."` argument of `Select` (page **??**) 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 **??**) instances.

5.7.3 0.9.2 (Jan 11, 2012)

New Features:

- **prody catdcd** command is implemented for concatenating and/or slicing `.dcd` files. See *prody catdcd* (page **??**) for usage examples.
- `DCDFile` (page **??**) can be opened in write or append mode, and coordinate sets can be added using `write()` (page **??**) method.

- `getReservedWords()` can be used to get a list of words that cannot be used to label user data.
- `confProDy()` (page **??**) 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 **??**) function.

Improvements:

- `writeDCD()` (page **??**) file accepts `AtomGroup` (page **??**) or other `Atomic` (page **??**) 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 **??**) instances, when closed, raise exception, similar to behavior of `file` objects in Python.
- Title of `AtomGroup` (page **??**) instances resulting from copying an `Atomic` (page **??**) 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`[24] module. See *ProDy Applications* (page **??**) for details of changes.
- `argparse`[25] module is added to the package for Python versions 2.6 and older.

Bugfixes:

- Fixed problems in `loadAtoms()` (page **??**) and `saveAtoms()` (page **??**) functions.
- Bugfixes in `parseDCD()` (page **??**) and `writeDCD()` (page **??**) functions for Windows compatability.

5.7.4 0.9.1 (Nov 9, 2011)

Bug Fixes:

- Fixed problems with reading and writing configuration files.
- Fixed problem with importing nose for testing.

5.7.5 0.9 (Nov 8, 2011)

New Features:

- PDBML[26] and mmCIF[27] files can be retrieved using `fetchPDB()` (page **??**) function.
- `getPDBLocalFolder()` and `setPDBLocalFolder()` functions are implemented for local PDB folder management.
- `parsePDBHeader()` (page **??**) is implemented for convenient parsing of header data from `.pdb` files.
- `showProtein()` (page **??**) is implemented to allow taking a quick look at protein structure.

[24] http://docs.python.org/library/argparse.html#argparse
[25] http://docs.python.org/library/argparse.html#argparse
[26] http://pdbml.pdb.org/
[27] http://mmcif.pdb.org/

- `Chemical` (page **??**) and `Polymer` (page **??**) 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.

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 `turnonDepracationWarnings()` for this purpose.

Functions:

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 **??**)
`assignSecondaryStructure()`	`assignSecstr()` (page **??**)
`scanPerturbationResponse()`	`calcPerturbResponse()` (page **??**)
`calcCrossCorrelations()`	`calcCrossCorr()` (page **??**)
`calcCumulativeOverlap()`	`calcCumulOverlap()` (page **??**)
`calcCovarianceOverlap()`	`calcCovOverlap()` (page **??**)
`showFractOfVariances()`	`showFractVars()` (page **??**)
`showCumFractOfVariances()`	`showCumulFractVars()` (page **??**)
`showCrossCorrelations()`	`showCrossCorr()` (page **??**)
`showCumulativeOverlap()`	`showCumulOverlap()` (page **??**)
`deform()`	`deformAtoms()` (page **??**)
`calcSumOfWeights()`	`calcOccupancies()` (page **??**)
`showSumOfWeights()`	`showOccupancies()` (page **??**)
`trimEnsemble()`	`trimPDBEnsemble()` (page **??**)
`getKeywordResidueNames()`	`getKeywordResnames()`
`setKeywordResidueNames()`	`setKeywordResnames()`
`getPairwiseAlignmentMethod()`	`getAlignmentMethod()` (page **??**)
`setPairwiseAlignmentMethod()`	`setAlignmentMethod()` (page **??**)
`getPairwiseMatchScore()`	`getMatchScore()` (page **??**)
`setPairwiseMatchScore()`	`setMatchScore()` (page **??**)
`getPairwiseMismatchScore()`	`getMismatchScore()` (page **??**)
`setPairwiseMismatchScore()`	`setMismatchScore()` (page **??**)
`getPairwiseGapOpeningPenalty()`	`getGapPenalty()` (page **??**)
`setPairwiseGapOpeningPenalty()`	`setGapPenalty()` (page **??**)
`getPairwiseGapExtensionPenalty()`	`getGapExtPenalty()` (page **??**)
`setPairwiseGapExtensionPenalty()`	`setGapExtPenalty()` (page **??**)

Coordinate methods:

All `getCoordinates()` and `setCoordinates()` methods in `atomic` (page **??**) and `ensemble` (page **??**) 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 **??**), `ensemble` (page **??**), `dynamics` (page **??**)
`getNumOfChains()`	`numChains()`	`atomic` (page **??**)
`getNumOfConfs()`	`numConfs()`	`ensemble` (page **??**)
`getNumOfCoordsets()`	`numCoordsets()`	`atomic` (page **??**), `ensemble` (page **??**)
`getNumOfDegOfFreedom()`	`numDOF()`	`dynamics` (page **??**)
`getNumOfFixed()`	`numFixed()`	`ensemble` (page **??**)
`getNumOfFrames()`	`numFrames()`	`ensemble` (page **??**)
`getNumOfResidues()`	`numResidues()`	`atomic` (page **??**)
`getNumOfMapped()`	`numMapped()`	`atomic` (page **??**)
`getNumOfModes()`	`numModes()`	`dynamics` (page **??**)
`getNumOfSelected()`	`numSelected()`	`ensemble` (page **??**)
`getNumOfUnmapped()`	`numUnmapped()`	`atomic` (page **??**)

`getName` **method**:

`getName()` methods are renamed as `getTitle()` to avoid confusions that might arise from changes in `atomic` (page **??**) method names listed below. All classes in `atomic` (page **??**), `ensemble` (page **??**), and `dynamics` (page **??**) are affected from this change.

In line with this change, `parsePDB()` (page **??**) and `parsePQR()` (page **??**) *name* arguments are changed to *title*, but *name* argument will also work until release 1.0.

This name change conflicted with `DCDFile.getTitle()` (page **??**) 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 **??**) classes, `AtomGroup` (page **??**), `Atom` (page **??**), `Chain` (page **??**), `Residue` (page **??**), `Selection` (page **??**) and `AtomMap` (page **??**).

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 **??**)
- removed `getProteinResidueNames()`, overloaded `getKeywordResnames()`
- removed `setProteinResidueNames()`, overloaded `setKeywordResnames()`

Scripts:

The way ProDy scripts work has changed. See *ProDy Applications* (page **??**) for details. Using older scripts will start issuing deprecation warnings in 2012.

Bug Fixes:

- Bugs in `execDSSP()` (page **??**) and `execSTRIDE()` (page **??**) 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*[28] pages for instructions on installing it.

5.8 ProDy 0.8 Series

[28] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

- 0.8.3 (Oct 16, 2011) (page **??**)
- 0.8.2 (Oct 14, 2011) (page **??**)
- 0.8.1 (Sep 16, 2011) (page **??**)
  - Normal Mode Wizard (page **??**)
- 0.8 (Aug 24, 2011) (page **??**)
  - *Normal Mode Wizard*[a] (page **??**)

[a] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

5.8.1 0.8.3 (Oct 16, 2011)

New Features:

- Functions to read and write PQR files: `parsePQR()` (page **??**) and `writePQR()` (page **??**).
- 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 **??**) function and `PDBBlastRecord` (page **??**) class are rewritten to use faster and more compact code.
- New `PackageLogger` (page **??**) function is implemented to unify logging and reporting task progression.
- Improvements in PDB ensemble support functions, e.g. `trimPDBEnsemble()` (page **??**), 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.

5.8.2 0.8.2 (Oct 14, 2011)

New Features:

- `fetchPDBClusters()` (page **??**), `loadPDBClusters()` (page **??**), and `getPDBCluster()` functions are implemented for handling PDB sequence cluster data. These functions can be used instead of `blastPDB()` (page **??**) 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 **??**) is implemented as `scanPerturbationResponse()` based on the code provided by Ying Liu.

Changes:

- `fetchPDBLigand()` (page **??**) 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` when the function fails to perform its action due to missing data in header dictionary.
- `fetchPDB()` (page **??**) decompresses PDB files found in the working directory when user asks for decompressed files.
- `parsePDB()` (page **??**) appends *chain* and *subset* arguments to `AtomGroup()` name.
- *chain* argument is added to `PDBBlastRecord.getHits()` (page **??**).

Improvements:

- Atom selection class `Select` (page **??**) is completely redesigned to prevent breaking of the parser when evaluating invalid selection strings.
- Improved type checking in `parsePDB()` (page **??**) function.

Bug Fixes:

- Bugfixes in `parseDSSP()` (page **??**): 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 **??**) 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 **??**) and `select` modules are developed.

5.8.3 0.8.1 (Sep 16, 2011)

New Features:

- `fetchLigandData()` is implemented for fetching ligand data from Ligand Expo.
- `parsePSF()` (page **??**) function is implemented for parsing X-PLOR format PSF files.

Changes:

- __slots__ is used in `AtomGroup` (page **??**) and `Atomic` (page **??**) classes. This change prevents user from assigning new variables to instances of all classes derived from the base `Atomic` (page **??**).
- `pyparsing` is updated to version 1.5.6.

Bug Fixes:

- A bug in `AtomGroup.copy()` (page **??**) method is fixed. When AtomGroup instance itself is copied, deep copies of data arrays were not made.
- A bug in `Select` (page **??**) class raising exceptions when negative residue number values are present is fixed.
- Another bug in `Select` (page **??**) class misinterpreting `same residue as ...` statement when specific chains are involved is fixed.
- A bug in `AtomGroup.addCoordset()` (page **??**) 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.

5.8.4 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 **??**) class for analyzing PDB structures, rename it as `PDBEnsemble` (page **??**). See the other changes that may affect your work below and the class documentation for more information.

New Features:

- `DCDFile` (page **??**) is implemented for handling DCD files.
- `Trajectory` (page **??**) is implemented for handling multiple trajectory files.
- `writeDCD()` (page **??**) is implemented for writing DCD files.
- *Trajectory Analysis*[29] example to illustrate usage of new classes for handling DCD files. *Essential Dynamics Analysis*[30] example is updated to use new ProDy classes.
- `PCA` (page **??**) supports `Trajectory` (page **??**) and `DCDFile` (page **??**) instances.
- `Ensemble` (page **??**) and `PDBEnsemble` (page **??**) classes can be associated with `AtomGroup` (page **??**) instances. This allows selecting and evaluating coordinates of subset of atoms. See `setAtomGroup()`, `select()`, `getAtomGroup()`, and `getSelection()` methods.
- `execDSSP()` (page **??**), `parseDSSP()` (page **??**), and `performDSSP()` (page **??**) functions are implemented for executing and parsing DSSP calculations.
- `execSTRIDE()` (page **??**), `parseSTRIDE()` (page **??**), and `performSTRIDE()` (page **??**) functions are implemented for executing and parsing DSSP calculations.
- `parsePDB()` (page **??**) function parses atom serial numbers. Atoms can be retrieved from an `AtomGroup` (page **??**) instance by their serial numbers using `getBySerial()` (page **??**) and `getBySerialRange()` methods.
- `calcADPs()` (page **??**) function can be used to calculate anisotropic displacement parameters for atoms with anisotropic temperature factor data.
- `getRMSFs()` (page **??**) is implemented for calculating root mean square fluctuations.
- `AtomGroup` (page **??**) and `Mode` (page **??**) or `Vector` (page **??**) additions are supported. This adds a new coordinate set to the `AtomGroup` (page **??**) instance.
- `getAttrNames()` is implemented for listing user set attribute names.

Improvements:

- `calcProjection()` (page **??**), `showProjection()` (page **??**), and `showCrossProjection()` (page **??**) functions can optionally calculate/display RMSD along the normal mode.

[29] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/trajectory.html#trajectory
[30] http://prody.csb.pitt.edu/tutorials/trajectory_analysis/eda.html#eda

- ANM, GNM, and PCA applications can optionally write compressed ProDy data files.
- `fetchPDB()` (page **??**) function can optionally write decompressed files and force copying a file from local mirror to target folder.
- `PCA.buildCovariance()` (page **??**) and `PCA.performSVD()` (page **??**) methods accept Numpy arrays as coordinate sets.
- Performance of `PCA.buildCovariance()` (page **??**) method is optimized for evaluation of PDB ensembles.
- `calcRMSD()` (page **??**) and `superpose()` (page **??**) functions are optimized for speed and memory usage.
- `Ensemble.getMSFs()` (page **??**) is optimized for speed and memory usage.
- Improvements in memory operations in `atomic` (page **??**), `ensemble` (page **??**), and `dynamics` (page **??**) modules for faster data (PDB/NMD) output.
- Optimizations in `Select` (page **??**) and `Contacts` (page **??**) classes.

Changes:

- `Ensemble` (page **??**) does not store conformation names. Instead, newly implemented `PDBEnsemble` (page **??**) class stores identifiers for individual conformations (PDB IDs). This class should be used in cases where source of individual conformations is important.
- `calcProjection()` (page **??**), `showProjection()` (page **??**), and `showCrossProjection()` (page **??**) 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 **??**).
- `parseDCD()` (page **??**) function returns `Ensemble` (page **??**) 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 **??**) module.
- Problem in **prody fetch** that occurred when a file is found in a local mirror is fixed.
- Bugfix in `AtomPointer.copy()` (page **??**) method.

Normal Mode Wizard

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.

5.9 ProDy 0.7 Series

5.9.1 0.7.2 (Jun 21, 2011)

New Features:

- `parseDCD()` (page **??**) is implemented for parsing coordinate sets from DCD files.

Improvements:

- `parsePDB()` (page **??**) 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 **??**) function causing problems when writing a single mode is fixeed.
- Other bugfixes in `dynamics` (page **??**) module functions.

5.9.2 0.7.1 (Apr 28, 2011)

Highlights:

- `Atomic` (page **??**) `__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 **??**) class is integrated with `HierView` (page **??**) 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:

- `matchAlign()` (page **??**) is implemented for quick alignment of protein structures. See *Ligand Extraction*[31] usage example.
- `setAttribute()`, `getAttribute()`, `delAttribute()`, and `isAttribute()` functions are implemented for `AtomGroup` (page **??**) class to facilitate storing user provided atomic data. See *Storing data in AtomGroup*[32] example.
- `saveAtoms()` (page **??**) and `loadAtoms()` (page **??**) 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.

[31] http://prody.csb.pitt.edu/tutorials/structure_analysis/ligands.html#extract-ligands
[32] http://prody.csb.pitt.edu/tutorials/prody_tutorial/atomgroup.html#id1

- `calcCollectivity()` (page **??**) function is implemented to allow for calculating collectivity of deformation vectors.

Improvements:

- `parsePDB()` (page **??**) can optionally return biomolecule when `biomol=True` keyword argument is passed.
- `parsePDB()` (page **??**) can optionally make secondary structure assignments when `secondary=True` keyword argument is passed.
- `calcSqFlucts()` (page **??**) function is changed to accept `Vector` (page **??**) instances, e.g. deformation vectors.

Changes:

- Changes were made in `calcADPAxes()` (page **??**) function to follow the conventions in analysis ADPs. See its documentation.

Bug Fixes:

- A in `Ensemble` (page **??**) slicing operations is fixed. Weights are now copied to the new instances obtained by slicing.
- Bug fixes in `dynamics` (page **??**) plotting functions `showScaledSqFlucts()` (page **??**), `showNormedSqFlucts()` (page **??**),

5.9.3 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 **??**) function is implemented for parsing matrices in sparse format. See the usage example in *Using an External Matrix*[33].
- `deform()` function is implemented for deforming coordinate sets along a normal mode or linear combination of multiple modes.
- `sliceModel()` (page **??**) 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.
- A new keyword argument for `calcADPAxes()` (page **??**) 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 **??**) and `NMA` (page **??**) classes are instantiated with an empty string, instances are called "Unnamed".

[33] http://prody.csb.pitt.edu/tutorials/enm_analysis/external.html#external-matrix

- `sliceMode()` (page **??**), `sliceVector()` (page **??**) and `reduceModel()` (page **??**) functions return the atom selection in addition to the sliced vector/mode/model instance.

Bug Fixes:

- Default selection for `calcGNM()` (page **??**) 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.

5.10 ProDy 0.6 Series

5.10.1 0.6.2 (Mar 16, 2011)

New Features:

- `performSVD()` (page **??**) 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*[34].

[34] http://prody.csb.pitt.edu/tutorials/enm_analysis/extend.html#extendmodel

- `plog()` is implemented for enabling users to make log entries.

Improvements:

- `compare` functions are improved to handle insertion codes.
- `HierView` (page **??**) allows for indexing using chain identifier and residue numbers. See usage example *Hierarchical Views*[35].
- `Chain` (page **??**) allows for indexing using residue number and insertion code. See usage example *Hierarchical Views*[36].
- `addCoordset()` (page **??**) function accepts `Atomic` (page **??**) and `Ensemble` (page **??**) instances as *coords* argument.
- New method `HierView.getAtoms()` (page **??**) is implemented.
- `AtomGroup` (page **??**) 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 **??**) allowed calculating covariance matrix for less than 3 coordinate sets is fixed.
- A bug in `mapOntoChain()` (page **??**) function that caused problems when mapping all atoms is fixed.

5.10.2 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 **??**) function is improved to map backbone or all atoms.

Improvements:

- `WWPDB_PDBFetcher` can download PDB files from different WWPDB FTP servers.
- `WWPDB_PDBFetcher` can also use local PDB mirrors for accessing PDB files.

Changes:

- `RCSB_PDBFetcher` is renamed as `WWPDB_PDBFetcher`.
- `mapOntoChain()` (page **??**) and `matchChains()` (page **??**) functions accept `"ca"` and `"bb"` as *subset* arguments.
- Definition of selection keyword "protein" is updated to include some non-standard amino acid abbreviations.

[35] http://prody.csb.pitt.edu/tutorials/prody_tutorial/hierview.html#hierview
[36] http://prody.csb.pitt.edu/tutorials/prody_tutorial/hierview.html#hierview

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 **??**) 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"`.

5.10.3 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*[37] 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*[38]).
- `Gamma` (page **??**) class is implemented for facilitating use of force constants based on atom type, residue type, or property. An example derived classes are `GammaStructureBased` (page **??**) and `GammaVariableCutoff` (page **??**).
- `calcTempFactors()` (page **??**) function is implemented to calculate theoretical temperature factors.
- 5 new *ProDy Applications* (page **??**) are implemented, and existing scripts are improved to output figures.
- `getModel()` (page **??**) method is implemented to make function development easier.
- `resetTicks()` (page **??**) 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 **??**) and `GNM.buildKirchhoff()` (page **??**) classes are improved to accept `Gamma` (page **??**) instances or other custom function as *gamma* argument. See also *Custom Gamma Functions*[39].
- `Select` (page **??**) 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).
- `applyBiomolecularTransformations()` function is improved to handle large biomolecular assemblies.
- Performance optimizations in `parsePDB()` (page **??**) and other functions.
- `Ensemble` (page **??**) class accepts `Atomic` (page **??**) instances and automatically adds coordinate sets to the ensemble.

Changes:

- `PDBlastRecord` is renamed as `PDBBlastRecord` (page **??**).

[37] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz
[38] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz
[39] http://prody.csb.pitt.edu/tutorials/enm_analysis/gamma.html#gamma

- `NMA` (page **??**) 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 **??**) and `calcGNM()` (page **??**) 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 **??**) and *prody gnm* (page **??**).
- Bug fixes in `showSqFlucts()` (page **??**) and `showProjection()` (page **??**) 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.

5.11 ProDy 0.5 Series

5.11.1 0.5.3 (Feb 11, 2011)

New Features:

- Membership, equality, and non-equality test operation are defined for all `atomic` (page **??**) classes. See *Operations on Selections*[40].
- Two functions are implemented for dealing with anisotropic temperature factors: `calcADPAxes()` (page **??**) and `buildADPMatrix()` (page **??**).
- `NMA.setEigens()` (page **??**) and `NMA.addEigenpair()` (page **??**) methods are implemented to assist analysis of normal modes calculated using external software.
- `parseNMD()` (page **??**) is implemented for parsing NMD files.
- `parseModes()` (page **??**) is implemented for parsing normal mode data.
- `parseArray()` (page **??**) is implementing for reading numeric data, particularly normal mode data calculated using other software for analysis using ProDy.

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

- The method in [BH02] (page **??**) to calculate overlap between covariance matrices is implemented as `calcCovOverlap()` (page **??**) function.
- `trimEnsemble()` to trim `Ensemble` (page **??**) instances is implemented.
- `checkUpdates()` (page **??**) to check for ProDy updates is implemented.

Changes:

- Change in default behavior of `parsePDB()` (page **??**) 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 **??**).
- `ProDyStartLogFile()` is renamed as `startLogfile()` (page **??**).
- `ProDyCloseLogFile()` is renamed as `closeLogfile()` (page **??**).
- `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 **??**) is fixed to output B-factors (Thanks to Dan Holloway for pointing it out).

5.11.2 0.5.2 (Jan 12, 2011)

Bug Fixes:

- An important fix in `sampleModes()` (page **??**) function was made (Thanks to Alberto Perez for finding the bug and suggesting a solution).

Improvements:

- Improvements in `ANM.calcModes()` (page **??**), `GNM.calcModes()` (page **??**), and `PCA.calcModes()` (page **??**) methods prevent Numpy/Scipy throwing an exception when more than available modes are requested by the user.
- Improvements in `blastPDB()` (page **??**) 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*[41].

5.11.3 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.
- 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.

[41] http://prody.csb.pitt.edu/tutorials/structure_analysis/pdbfiles.html#writepdb

5.11.4 0.5 (Dec 21, 2010)

New Features:

- `AtomPointer` (page **??**) base class for classes pointing to atoms in an `AtomGroup` (page **??**).
- `AtomPointer` (page **??**) instances (Selection, Residue, etc.) can be added. See *Operations on Selections*[42] for examples.
- `Select.getIndices()` (page **??**) and `Select.getBoolArray()` (page **??**) methods to expand the usage of `Select` (page **??**).
- `sliceVector()` (page **??**) and `sliceMode()` (page **??**) functions.
- `saveModel()` (page **??**) and `loadModel()` (page **??**) functions for saving and loading NMA data.
- `parsePDBStream()` (page **??**) can now parse specific chains or alternate locations from a PDB file.
- `alignCoordsets()` (page **??**) is implemented to superimpose coordinate sets of an `AtomGroup` (page **??**) instance.

Bug Fixes:

- A bug in `parsePDBStream()` (page **??**) 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 **??**) instance yields `Residue` (page **??**) instances.
- `Vector` (page **??**) 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 **??**).

5.12 ProDy 0.2 Series

5.12.1 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`, use "protein and within 5 of water" instead.
- `findMatchingChains()` is renamed to `matchChains()` (page **??**).
- `showOverlapMatrix()` is renamed to `showOverlapTable()` (page **??**).
- Modules are reorganized.

New Features:

- `Atomic` (page **??**) for easy type checking.
- `Contacts` (page **??**) for faster intermolecular contact identification.

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

- `Select` (page **??**) can identify intermolecular contacts. See *Intermolecular Contacts*[43] for an examples and details.
- `sampleModes()` (page **??**) implemented for sampling conformations along normal modes.

Improvements:

- `proteins.compare` (page **??**) 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 **??**) 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 **??**) 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.

[43] http://prody.csb.pitt.edu/tutorials/structure_analysis/contacts.html#contacts

5.13 ProDy 0.1 Series

5.13.1 0.1.2 (Nov 9, 2010)

- Important bug fixes and improvements in NMA helper and plotting functions.
- Documentation updates and improvements.

5.13.2 0.1.1 (Nov 8, 2010)

- Important bug fixes and improvements in chain comparison functions.
- Bug fixes.
- Source clean up.
- Documentation improvements.

5.13.3 0.1 (Nov 7, 2010)

- First release.

CHAPTER 6

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 Bahar Lab[1] at the University of Pittsburgh[2] with support from NIH R01 GM099738 award.

6.1.1 Development Team

Ahmet Bakan[3] initiated the *ProDy* project, designed and developed *ProDy*, *NMWiz*, *Evol*, and *DruGUI*.

Chakra Chennubhotla[4] is currently overseeing the overall development of *ProDy*.

Anindita Dutta[5] contributed to the development of *Evol*, `database` (page **??**) and `sequence` (page **??**) modules.

Tim Lezon contributed to development of Rotations and Translation of Blocks and Membrane ENM models.

Wenzhi Mao[6] contributed to development of MSA analysis functions.

Lidio Meireles[7] provided insightful comments on the design of *ProDy*, and contributed to the development of *ProDy Applications* (page **??**).

6.1.2 Contributors

In addition to the development team members, we acknowledge contributions and feedback from the following individuals:

[1] http://www.ccbb.pitt.edu/faculty/bahar/
[2] http://www.pitt.edu/
[3] http://ahmetbakan.com
[4] http://www.csb.pitt.edu/Faculty/Chakra/
[5] http://www.linkedin.com/pub/anindita-dutta/5a/568/a90
[6] http://www.linkedin.com/pub/wenzhi-mao/2a/29a/29
[7] http://www.linkedin.com/in/lidio

Ying Liu[8] provided the code for Perturbation Response Scanning method.

Kian Ho[9] contributed with bug fixes and unit tests for DSSP functions.

Gökçen Eraslan[10] 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.

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.
> *Bioinformatics* **2009** 25(11):1422-3.

6.3 Credits

ProDy makes use of the following great software:

pyparsing[11] 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[12] KDTree package and pairwise2 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. pairwise2 module enables performing sequence alignment for protein structure comparison and ensemble analysis.

ProDy requires NumPy[13] 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.

Finally, ProDy can benefit from SciPy[14] and Matplotlib[15] 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.

[8] http://www.linkedin.com/pub/ying-liu/15/48b/5a9
[9] https://github.com/kianho
[10] http://blog.yeredusuncedernegi.com/
[11] http://pyparsing.wikispaces.com
[12] http://biopython.org
[13] http://www.numpy.org
[14] http://www.scipy.org
[15] http://matplotlib.org

6.5 License

6.5.1 ProDy

ProDy is available under the MIT License[16]:

```
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.

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[17] KDTree package and pairwise2 module are distributed with the ProDy package. Biopython is developed by The Biopython Consortium and is available under the Biopython license[18]:

```
                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.
```

[16] http://opensource.org/licenses/MIT
[17] http://biopython.org
[18] http://www.biopython.org/DIST/LICENSE

6.5.3 Pyparsing

The pyparsing[19] module is distributed with the ProDy package. Pyparsing is developed by Paul T. McGuire and is available under the MIT License[20]:

```
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.
```

6.5.4 Argparse

The argparse module[21] is distributed with the ProDy package. Argparse is developed by Steven J. Bethard and is available under the Python Software Foundation License[22].

[19] http://pyparsing.wikispaces.com
[20] http://opensource.org/licenses/MIT
[21] http://code.google.com/p/argparse/
[22] http://docs.python.org/license.html

Bibliography

[BR95] Bruschweiler R. Collective protein dynamics and nuclear spin relaxation. *J Chem Phys* **1995** 102:3396-3403.

[CA09] Atilgan C, Atilgan AR, Perturbation-Response Scanning Reveals Ligand Entry-Exit Mechanisms of Ferric Binding Protein. *PLoS Comput Biol* **2009** 5(10):e1000544.

[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.

[KH00] Konrad H, Andrei-Jose P, Serge D, Marie-Claire BF, Gerald RK. Harmonicity in slow protein dynamics. *Chem Phys* **2000** 261:25-37.

[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.

[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.

[AA93] Amadei A, Linssen AB, Berendsen HJ. Essential dynamics of proteins. *Proteins* **1993** 17(4):412-25.

[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.

[TL12] Lezon TR, Bahar I, Constraints Imposed by the Membrane Selectively Guide the Alternating Access Dynamics of the Glutamate Transporter GltPh

[WK83] Kabsch W, Sander C. Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers* **1983** 22:2577-2637.

[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.

Python Module Index

p

s

t

u