Structure Analysis

Release 1.5.1

Ahmet Bakan

December 24, 2013
## CONTENTS

1 Introduction 1  
   1.1 Required Programs 1  
   1.2 Recommended Programs 1  
   1.3 Getting Started 1  

2 PDB files 2  
   2.1 Fetch PDB files 2  
   2.2 Parse PDB files 3  
   2.3 Write PDB file 6  

3 Blast Search PDB 7  
   3.1 Blast search 7  
   3.2 Best match 8  
   3.3 PDB hits 8  
   3.4 Download hits 8  

4 Building Biomolecules 9  
   4.1 Build a Multimer 9  
   4.2 Build a Tetramer 11  

5 Alignments 13  
   5.1 Parse an NMR structure 13  
   5.2 Calculate RMSD 13  
   5.3 Align coordinate sets 14  
   5.4 Write aligned coordinates 15  

6 Structure Comparison 16  
   6.1 Match chains 16  
   6.2 Map onto a chain 19  

7 Intermolecular Contacts 21  
   7.1 Simple contact selections 21  
   7.2 Contacts between different atom groups 21  
   7.3 Composite contact selections 22  
   7.4 Spherical atom selections 22  
   7.5 Fast contact selections 22  

8 Ligand Extraction 24  
   8.1 Parse reference and blast search 24  
   8.2 Align structures and extract ligands 25
INTRODUCTION

This tutorial shows how to various ProDy features for managing, handling, and analyzing protein structures.

1.1 Required Programs

Latest version of ProDy\(^1\) and Matplotlib\(^2\) are required.

1.2 Recommended Programs

IPython\(^3\) is strongly recommended.

1.3 Getting Started

To follow this tutorial, you will need the following files:

There are no required files.

We recommend that you will follow this tutorial by typing commands in an IPython session, e.g.:

$ ipython

or with pylab environment:

$ ipython --pylab

First, we will make necessary imports from ProDy and Matplotlib packages.

In [1]: from prody import *

In [2]: from pylab import *

In [3]: ion()

We have included these imports in every part of the tutorial, so that code copied from the online pages is complete. You do not need to repeat imports in the same Python session.

\(^1\)http://prody.csb.pitt.edu

\(^2\)http://matplotlib.org

\(^3\)http://ipython.org
This example demonstrates how to use the flexible PDB fetcher, `fetchPDB()`. Valid inputs are PDB identifier, e.g. 2k39\(^1\), or a list of PDB identifiers, e.g. ['2k39', '1mkp', '1etc']. Compressed PDB files (pdb.gz) will be saved to the current working directory or a target folder.

### 2.1 Fetch PDB files

#### 2.1.1 Single file

We start by importing everything from the ProDy package:

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

The function will return a filename if the download is successful.

```python
In [2]: filename = fetchPDB('1p38')
In [3]: filename
Out[3]: '1p38.pdb'
```

#### 2.1.2 Multiple files

This function also accepts a list of PDB identifiers:

```python
In [4]: filenames = fetchPDB(['1p38', '1r39', '@!~#'])
In [5]: filenames
Out[5]: ['1p38.pdb', '1r39.pdb', None]
```

For failed downloads, None will be returned (or the list will contain None item).

Also note that in this case we passed a folder name. Files are saved in this folder, after it is created if it did not exist.

ProDy will give you a report of download results and return a list of filenames. The report will be printed on the screen, which in this case would be:

\(^1\)http://www.pdb.org/pdb/explore/explore.do?structureId=2k39
2.2 Parse PDB files

ProDy offers a fast and flexible PDB parser, parsePDB(). Parser can be used to read well defined subsets of atoms, specific chains or models (in NMR structures) to boost the performance. This example shows how to use the flexible parsing options.

Three types of input are accepted from user:

- PDB file path, e.g. "../1MKP.pdb"
- compressed (gzipped) PDB file path, e.g. "1p38.pdb.gz"
- PDB identifier, e.g. 2k39

Output is an AtomGroup instance that stores atomic data and can be used as input to functions and classes for dynamics analysis.

2.2.1 Parse a file

You can parse PDB files by passing a filename (gzipped files are handled). We do so after downloading a PDB file (see Fetch PDB files (page 2) for more information):

```
In [6]: fetchPDB('1p38')
Out[6]: '1p38.pdb'
```

```
In [7]: atoms = parsePDB('1p38')
```

```
In [8]: atoms
Out[8]: <AtomGroup: 1p38 (2962 atoms)>
```

Parser returns an AtomGroup instance.

Also note that the time it took to parse the file is printed on the screen. This includes the time that it takes to evaluate coordinate lines and build an AtomGroup instance and excludes the time spent on reading the file from disk.

2.2.2 Use an identifier

PDB files can be parsed by passing simply an identifier. Parser will look for a PDB file that matches the given identifier in the current working directory. If a matching file is not found, ProDy will downloaded it from PDB FTP server automatically and saved it in the current working directory.

```
In [9]: atoms = parsePDB('1mkp')
```

```
In [10]: atoms
Out[10]: <AtomGroup: 1mkp (1183 atoms)>
```

__2http://www.pdb.org/pdb/explore/explore.do?structureId=2k39__
2.2.3 Subsets of atoms

Parser can be used to parse backbone or Cα atoms:

In [11]: backbone = parsePDB('1mkp', subset='bb')

In [12]: backbone
Out[12]: <AtomGroup: 1mkp_bb (576 atoms)>

In [13]: calpha = parsePDB('1mkp', subset='ca')

In [14]: calpha
Out[14]: <AtomGroup: 1mkp_ca (144 atoms)>

2.2.4 Specific chains

Parser can be used to parse a specific chain from a PDB file:

In [15]: chA = parsePDB('3mkb', chain='A')

In [16]: chA
Out[16]: <AtomGroup: 3mkb_A (1198 atoms)>

In [17]: chC = parsePDB('3mkb', chain='C')

In [18]: chC
Out[18]: <AtomGroup: 3mkb_C (1189 atoms)>

Multiple chains can also be parsed in the same way:

In [19]: chAC = parsePDB('3mkb', chain='AC')

In [20]: chAC
Out[20]: <AtomGroup: 3mkb_AC (2387 atoms)>

2.2.5 Specific models

Parser can be used to parse a specific model from a file:

In [21]: model1 = parsePDB('2k39', model=10)

In [22]: model1
Out[22]: <AtomGroup: 2k39 (1231 atoms)>

2.2.6 Alternate locations

When a PDB file contains alternate locations for some of the atoms, by default alternate locations with indicator A are parsed.

In [23]: altlocA = parsePDB('1ejg')

In [24]: altlocA
Out[24]: <AtomGroup: 1ejg (637 atoms)>

Specific alternate locations can be parsed as follows:
In [25]: altlocB = parsePDB('1ejg', altloc='B')

In [26]: altlocB
Out[26]: <AtomGroup: 1ejg (634 atoms)>

Note that in this case number of atoms are different between the two atom groups. This is because the residue types of atoms with alternate locations are different.

Also, all alternate locations can be parsed as follows:

In [27]: all_altlocs = parsePDB('1ejg', altloc=True)

In [28]: all_altlocs
Out[28]: <AtomGroup: 1ejg (637 atoms; active #0 of 3 coordsets)>

Note that this time parser returned three coordinate sets. One for each alternate location indicator found in this file (A, B, C). When parsing multiple alternate locations, parser will expect for the same residue type for each atom with an alternate location. If residue names differ, a warning message will be printed.

### 2.2.7 Composite arguments

Parser can be used to parse coordinates from a specific model for a subset of atoms of a specific chain:

In [29]: composite = parsePDB('2k39', model=10, chain='A', subset='ca')

In [30]: composite
Out[30]: <AtomGroup: 2k39_A_ca (76 atoms)>

### 2.2.8 Header data

PDB parser can be used to extract header data in a dict from PDB files as follows:

In [31]: atoms, header = parsePDB('1ubi', header=True)

In [32]: list(header)
Out[32]: ['A',
'sheet',
'classification',
'reference',
'title',
'polymers',
'resolution',
'space_group',
'chemicals',
'experiment',
'helix',
'version',
'authors',
'identifier',
'deposition_date',
'biomoltrans']

In [33]: header['experiment']

---

3http://docs.python.org/library/stdtypes.html#dict
Out[33]: 'X-RAY DIFFRACTION'

In [34]: header[‘resolution’]
Out[34]: 1.8

It is also possible to parse only header data by passing model=0 as an argument:

In [35]: header = parsePDB(‘1ubi’, header=True, model=0)

or using parsePDBHeader() function:

In [36]: header = parsePDBHeader(‘1ubi’)

2.3 Write PDB file

PDB files can be written using writePDB() function. This example shows how to write PDB files for AtomGroup instances and subsets of atoms.

2.3.1 Write all atoms

All atoms in an AtomGroup can be written in PDB format as follows:

In [37]: writePDB(‘MKP3.pdb’, atoms)
Out[37]: ‘MKP3.pdb’

Upon successful writing of PDB file, filename is returned.

2.3.2 Write a subset

It is also possible to write subsets of atoms in PDB format:

In [38]: alpha_carbons = atoms.select(‘calpha’)

In [39]: writePDB(‘1mkp_ca.pdb’, alpha_carbons)
Out[39]: ‘1mkp_ca.pdb’

In [40]: backbone = atoms.select(‘backbone’)

In [41]: writePDB(‘1mkp_bb.pdb’, backbone)
Out[41]: ‘1mkp_bb.pdb’
This example demonstrates how to use Protein Data Bank blast search function, `blastPDB()`.

`blastPDB()` is a utility function which can be used to check if structures matching a sequence exist in PDB or to identify a set of related structures for *Ensemble Analysis*\(^1\).

We will use amino acid sequence of a protein, e.g. `ASFPVEILPFLYLGCAKDSTNLDVLEEFGIKYILNVTPNL...YDIVKMKKSNISPNFNFMGQLLDFERTL`.

The `blastPDB()` function accepts sequence as a Python `str`\(^2\).

Output will be a `PDBBlastRecord` instance that stores PDB hits and returns to the user those sharing sequence identity above a user specified value.

### 3.1 Blast search

We start by importing everything from the ProDy package:

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

Let’s search for structures similar to that of MKP-3, using its sequence:

```
In [2]: blast_record = blastPDB('''ASFPVEILPFLYLGCAKDSTNLDVLEEFGIKYILNVTPNL
                     ...: PNLLENAGEFKIQIPISDHWSQNLQFFPEAISFIDEAR
                     ...: GKNCGVLYHSLAGRSVTVTAVYLMQKLNLSMNDAYDIV
                     ...: KMKKSNSPLNFNFMGQLLDFERTL''')
```

`blastPDB()` function returns a `PDBBlastRecord`. It is a good practice to save this record on disk, as NCBI may not respond to repeated searches for the same sequence. We can do this using Python standard library `pickle`\(^3\) as follows:

```
In [3]: import pickle
```

Record is save using `dump()`\(^4\) function into an open file:

```
In [4]: pickle.dump(blast_record, open('mkp3_blast_record.pkl', 'w'))
```

Then, it can be loaded using `load()`\(^5\) function:

---

\(^1\)[http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#pca]
\(^2\)[http://docs.python.org/library/functions.html#str]
\(^3\)[http://docs.python.org/library/pickle.html#pickle]
\(^4\)[http://docs.python.org/library/pickle.html#pickle.dump]
\(^5\)[http://docs.python.org/library/pickle.html#pickle.load]
3.2 Best match

To get the best match, PDBBlastRecord.getBest() method can be used:

```python
In [6]: best = blast_record.getBest()
```

```python
In [7]: best[‘pdb_id’]
Out[7]: ‘1mkp’
```

```python
In [8]: best[‘percent_identity’]
Out[8]: 100.0
```

3.3 PDB hits

```python
In [9]: hits = blast_record.getHits()
```

```python
In [10]: list(hits)
Out[10]: [‘1mkp’]
```

This results in only MKP-3 itself, since percent_identity argument was set to 90 by default:

```python
In [11]: hits = blast_record.getHits(percent_identity=50)
```

```python
In [12]: list(hits)
Out[12]: [‘1m3g’, ‘2hxp’, ‘3lj8’, ‘3ezz’, ‘1mkp’]
```

```python
In [13]: hits = blast_record.getHits(percent_identity=40)
```

```python
In [14]: list(hits)
Out[14]: [‘3lj8’, ‘1mkp’, ‘1zzw’, ‘2g6z’, ‘2hxp’, ‘3ezz’, ‘1m3g’, ‘2oud’]
```

This resulted in 7 hits, including structures of MKP-2, MKP-4, and MKP-5 More information on a hit can be obtained as follows:

```python
In [15]: hits[‘1zzw’][‘percent_identity’]
Out[15]: 49.27536231884058
```

```python
In [16]: hits[‘1zzw’][‘align-len’]
Out[16]: 138
```

```python
In [17]: hits[‘1zzw’][‘identity’]
Out[17]: 68
```

3.4 Download hits

PDB hits can be downloaded using fetchPDB() function:

```python
filenames = fetchPDB(hits.keys())
filenames
```
Some PDB files contain coordinates for a monomer of a functional/biological multimer (biomolecule). ProDy offers functions to build structures of biomolecules using the header data from the PDB file. We will use PDB file that contains the coordinates for a monomer of a biological multimeric protein and the transformations in the header section to generate the multimer coordinates. Output will be an AtomGroup instance that contains the multimer coordinates.

We start by importing everything from the ProDy package:

```python
In [1]: from prody import *
In [2]: from pylab import *
In [3]: ion()
```

### 4.1 Build a Multimer

Let’s build the dimeric form of 3enl¹ of enolase²:

```python
In [4]: monomer, header = parsePDB('3enl', header=True)
In [5]: monomer
Out[5]: <AtomGroup: 3enl (3647 atoms)>
```

Note that we passed header=True argument to parse header data in addition to coordinates.

```python
In [6]: showProtein(monomer);
In [7]: legend();
```

¹[http://www.pdb.org/pdb/explore/explore.do?structureId=3enl](http://www.pdb.org/pdb/explore/explore.do?structureId=3enl)

Let's get the dimer coordinates using `buildBiomolecules()` function:

```python
In [8]: dimer = buildBiomolecules(header, monomer)
```

```python
In [9]: dimer
Out[9]: <AtomGroup: 3enl biomolecule 1 (7294 atoms)>
```

This function takes biomolecular transformations from the `header` dictionary (item with key `'biomoltrans'`) and applies them to the `monomer`.

```python
In [10]: showProtein(dimer);
```

```python
In [11]: legend();
```

The `dimer` object now has two chains:

```python
In [12]: list(dimer.iterChains())
Out[12]:
```
4.2 Build a Tetramer

Let’s build the tetrameric form of 1k4c\(^3\) of \texttt{KcsA\_potassium\_channel}\(^4\):

\begin{verbatim}
In [13]: monomer, header = parsePDB('1k4c', header=True)
In [14]: monomer
Out[14]: <AtomGroup: 1k4c (4534 atoms)>
In [15]: showProtein(monomer);
In [16]: legend();
\end{verbatim}

![Image](image.png)

Note that we do not want to replicate potassium ions, so we will exclude them:

\begin{verbatim}
In [17]: potassium = monomer.name_K
In [18]: potassium
Out[18]: <Selection: 'name K' from 1k4c (7 atoms)>
In [19]: without_K = ~ potassium
In [20]: without_K
Out[20]: <Selection: 'not (name K)' from 1k4c (4527 atoms)>
In [21]: tetramer = buildBiomolecules(header, without_K)
In [22]: tetramer
Out[22]: <AtomGroup: 1k4c Selection 'not (name K)' biomolecule 1 (18108 atoms)>
\end{verbatim}

\(^3\)http://www.pdb.org/pdb/explore/explore.do?structureId=1k4c
\(^4\)http://en.wikipedia.org/wiki/KcsA\_potassium\_channel
Now, let’s append potassium ions to the tetramer:

```python
In [23]: potassium.setChids('K')
```

```python
In [24]: kcsa = tetramer + potassium.copy()
```

```python
In [25]: kcsa.setTitle('KcsA')
```

Here is a view of the tetramer:

```python
In [26]: showProtein(kcsa);
```

```python
In [27]: legend();
```

Let’s get a list of all the chains:

```python
In [28]: list(kcsa.iterChains())
```

```json
Out[28]:
[<Chain: A from Segment A from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment A from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment A from KcsA (162 residues, 854 atoms)>,
 <Chain: A from Segment B from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment B from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment B from KcsA (162 residues, 854 atoms)>,
 <Chain: A from Segment C from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment C from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment C from KcsA (162 residues, 854 atoms)>,
 <Chain: A from Segment D from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment D from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment D from KcsA (162 residues, 854 atoms)>,
 <Chain: K from KcsA (7 residues, 7 atoms)>]
```

You see that chain identifiers are preserved within monomers, and monomers have different segment names. To get chain B from first monomer with segment name A, we would do the following:

```python
In [29]: kcsa['A', 'B']
```

```json
Out[29]: <Chain: B from Segment A from KcsA (417 residues, 1851 atoms)>
```

---

### 4.2. Build a Tetramer

12
AtomGroup instances can store multiple coordinate sets, i.e. multiple models from an NMR structure. This example shows how to align such coordinate sets using alignCoordsets() function. Resulting AtomGroup will have its coordinate sets superposed onto the active coordinate set selected by the user.

5.1 Parse an NMR structure

We start by importing everything from the ProDy package:

```
In [1]: from prody import *
In [2]: from pylab import *
In [3]: ion()
```

We use 1joy\(^1\) that contains 21 models homodimeric domain of EnvZ protein from E. coli.

```
In [4]: pdb = parsePDB('1joy')
In [5]: pdb.numCoordsets()
Out[5]: 21
```

5.2 Calculate RMSD

```
In [6]: rmsds = calcRMSD(pdb)
In [7]: rmsds.mean()
Out[7]: 37.506911678400989
```

This function calculates RMSDs with respect to the active coordinate set, which is the first model in this case.

```
In [8]: showProtein(pdb);
In [9]: pdb.setACSIndex(1) # model 2 in PDB is now the active coordinate set
In [10]: showProtein(pdb);
```

\(^1\)http://www.pdb.org/pdb/explore/explore.do?structureId=1joy
5.3 Align coordinate sets

We will superpose all models onto the first model in the file using based on Cα atom positions:

In [11]: legend();

To use all backbone atoms, `pdb.backbone` can be passed as argument. See *Atom Selections*\(^2\) for more information on making selections.

Coordinate sets are superposed onto the first model (the active coordinate set).

In [12]: pdb.setACSIndex(0)

In [13]: alignCoordsets(pdb.calpha);

In [14]: rmsds = calcRMSD(pdb)

In [15]: rmsds.mean()
Out[15]: 3.2768912151768554

In [16]: showProtein(pdb);

In [17]: pdb.setACSIndex(1) # model 2 in PDB is now the active coordinate set

In [18]: showProtein(pdb);

In [19]: legend();

\(^2\)http://prody.csb.pitt.edu/manual/reference/atomic/select.html#selections
5.4 Write aligned coordinates

Using `writePDB()` function, we can write the aligned coordinate sets in PDB format:

```python
In [20]: writePDB('1joy_aligned.pdb', pdb)
Out[20]: '1joy_aligned.pdb'
```
CHAPTER SIX

STRUCTURE COMPARISON

This section shows how to find identical or similar protein chains in two structures files and align them.

proteins module contains functions for matching and mapping chains. Results can be used for RMSD fitting and PCA analysis.

Output will be AtomMap instances that can be used as input to ProDy classes and functions.

6.1 Match chains

We start by importing everything from the ProDy package:

```
In [1]: from prody import *
In [2]: from pylab import *
In [3]: ion()
```

Matching chains is useful when comparing two structures. We will find matching chains in two different HIV Reverse Transcriptase\(^1\) structures.

First we define a function that prints information on paired (matched) chains:

```
In [4]: def printMatch(match):
    ...:     print('Chain 1 : {}'.format(match[0]))
    ...:     print('Chain 2 : {}'.format(match[1]))
    ...:     print('Length : {}'.format(len(match[0])))
    ...:     print('Seq identity: {}'.format(match[2]))
    ...:     print('Seq overlap : {}'.format(match[3]))
    ...:     print('RMSD : {}
    ...:
```

Now let’s parse bound RT structure 1vrt\(^2\) and unbound structure 1dlo\(^3\):

```
In [5]: bound = parsePDB('1vrt')
In [6]: unbound = parsePDB('1dlo')
```

Let’s verify that these structures are not aligned:

---

\(^1\)http://en.wikipedia.org/wiki/Reverse_Transcriptase
\(^2\)http://www.pdb.org/pdb/explore/explore.do?structureId=1vrt
\(^3\)http://www.pdb.org/pdb/explore/explore.do?structureId=1dlo
We find matching chains as follows:

In [9]: matches = matchChains(bound, unbound)

In [10]: for match in matches:
   ....:     printMatch(match)
   ....:
Chain 1 : AtomMap Chain B from 1vrt -> Chain B from 1dlo
Chain 2 : AtomMap Chain B from 1dlo -> Chain B from 1vrt
Length  : 400
Seq identity: 99.2518703242
Seq overlap : 96
RMSD      : 110.45149192

Chain 1 : AtomMap Chain A from 1vrt -> Chain A from 1dlo
Chain 2 : AtomMap Chain A from 1dlo -> Chain A from 1vrt
Length  : 524
Seq identity: 99.0458015267
Seq overlap : 94
RMSD      : 142.084163869

This resulted in two matches. Chains A and B of two structures are paired. These chains contain only Cα atoms:

In [11]: match[0][0].iscalpha
Out[11]: True

In [12]: match[0][1].iscalpha
Out[12]: True

For a structural alignment based on both chains, we merge these matches as follows:

In [13]: bound_ca = matches[0][0] + matches[1][0]

In [14]: bound_ca
Out[14]: <AtomMap: (AtomMap Chain B from 1vrt -> Chain B from 1dlo) + (AtomMap Chain A from 1vrt -> Chain A from 1dlo) from 1vrt (924 atoms)>

In [15]: unbound_ca = matches[0][1] + matches[1][1]

In [16]: unbound_ca
Out[16]: <AtomMap: (AtomMap Chain B from 1dlo -> Chain B from 1vrt) + (AtomMap Chain A from 1dlo -> Chain A from 1vrt) from 1dlo (924 atoms)>

Let’s calculate RMSD:

In [17]: calcRMSD(bound_ca, unbound_ca)
Out[17]: 129.34348658001386

We find the transformation that minimizes RMSD between these two selections and apply it to unbound structure:

In [18]: calcTransformation(unbound_ca, bound_ca).apply(unbound);

In [19]: calcRMSD(bound_ca, unbound_ca)
Out[19]: 6.0020747465625393

Let’s see the aligned structures now:

In [20]: showProtein(unbound, bound);

In [21]: legend();

By default, matchChains() function matches Cα atoms. subset argument allows for matching larger numbers of atoms. We can match backbone atoms as follows:

In [22]: matches = matchChains(bound, unbound, subset='bb')

In [23]: for match in matches:
   ..:    printMatch(match)
   ..:    Chain 1   : AtomMap Chain B from 1vrt -> Chain B from 1dlo
   ..:    Chain 2   : AtomMap Chain B from 1dlo -> Chain B from 1vrt
   ..:    Length    : 1600
   ..:    Seq identity: 99.2518703242
   ..:    Seq overlap  : 96
RMSD : 1.71102621571
Chain 1 : AtomMap Chain A from 1vrt -> Chain A from 1dlo
Chain 2 : AtomMap Chain A from 1dlo -> Chain A from 1vrt
Length : 2096
Seq identity: 99.0458015267
Seq overlap : 94
RMSD : 7.78386812028

Or, we can match all atoms as follows:

In [24]: matches = matchChains(bound, unbound, subset='all')

In [25]: for match in matches:
   ..: printMatch(match)
   ....: Chain 1 : AtomMap Chain B from 1vrt -> Chain B from 1dlo
   ....: Chain 2 : AtomMap Chain B from 1dlo -> Chain B from 1vrt
   Length : 3225
   Seq identity: 99.2518703242
   Seq overlap : 96
   RMSD : 2.20947196284

Chain 1 : AtomMap Chain A from 1vrt -> Chain A from 1dlo
Chain 2 : AtomMap Chain A from 1dlo -> Chain A from 1vrt
Length : 4159
Seq identity: 99.0458015267
Seq overlap : 94
RMSD : 7.83814068858

6.2 Map onto a chain

Mapping is different from matching. When chains are matched, all matching atoms are returned as AtomMap instances. When atoms are mapped onto a chain, missing atoms are replaced by dummy atoms. The length of the mapping is equal to the length of chain. Mapping is used particularly useful in assembling coordinate data in analysis of heterogeneous datasets (see Ensemble Analysis).

Let’s map bound structure onto unbound chain A (subunit p66):

In [26]: def printMapping(mapping):
   ..: print('Mapped chain : {}'.format(mapping[0]))
   ..: print('Target chain : {}'.format(mapping[1]))
   ..: print('Mapping length : {}'.format(len(mapping[0])))
   ..: print('# of mapped atoms: {}'.format(mapping[0].numMapped()))
   ..: print('# of dummy atoms : {}'.format(mapping[0].numDummies()))
   ..: print('Sequence identity: {}'.format(mapping[2]))
   ..: print('Sequence overlap : {}'.format(mapping[3]))
   ..:

In [27]: unbound_hv = unbound.getHierView()

In [28]: unbound_A = unbound_hv['A']

In [29]: mappings = mapOntoChain(bound, unbound_A)

4http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#pca

6.2. Map onto a chain
In [30]: for mapping in mappings:
    ....:     printMapping(mapping)
    ....:
Mapped chain  : AtomMap Chain B from 1vrt -> Chain A from 1dlo
Target chain  : AtomMap Chain A from 1dlo -> Chain B from 1vrt
Mapping length : 556
# of mapped atoms: 524
# of dummy atoms : 32
Sequence identity: 99
Sequence overlap : 94

mapOntoChain() mapped only Cα atoms. `subset` argument allows for matching larger numbers of atoms. We can map backbone atoms as follows:

In [31]: mappings = mapOntoChain(bound, unbound_A, subset='bb')

In [32]: for mapping in mappings:
    ....:     printMapping(mapping)
    ....:
Mapped chain  : AtomMap Chain B from 1vrt -> Chain A from 1dlo
Target chain  : AtomMap Chain A from 1dlo -> Chain B from 1vrt
Mapping length : 2224
# of mapped atoms: 2096
# of dummy atoms : 128
Sequence identity: 99
Sequence overlap : 94

Or, we can map all atoms as follows:

In [33]: mappings = mapOntoChain(bound, unbound_A, subset='all')

In [34]: for mapping in mappings:
    ....:     printMapping(mapping)
    ....:
Mapped chain  : AtomMap Chain B from 1vrt -> Chain A from 1dlo
Target chain  : AtomMap Chain A from 1dlo -> Chain B from 1vrt
Mapping length : 4370
# of mapped atoms: 4159
# of dummy atoms : 211
Sequence identity: 99
Sequence overlap : 94

6.2. Map onto a chain
Chapter Seven

Intermolecular Contacts

This example shows how to identify intermolecular contacts, e.g., protein atoms interacting with a bound inhibitor. A structure of a protein-ligand complex in PDB format will be used. Output will be Selection instances that point to atoms matching the contact criteria given by the user. Selection instances can be used as input to other functions for further analysis.

7.1 Simple contact selections

We start by importing everything from the ProDy package:

In [1]: from prody import *

In [2]: from pylab import *

In [3]: ion()

ProDy selection engine has a powerful feature that enables identifying intermolecular contacts very easily. We will see this by identifying protein atoms interacting with an inhibitor.

We start with parsing a PDB file that contains a protein and a bound ligand.

In [4]: pdb = parsePDB('1zz2')

1zz2 contains an inhibitor bound p38 MAP kinase structure. Residue name of inhibitor is B11. Protein atoms interacting with the inhibitor can simply be identified as follows:

In [5]: contacts = pdb.select('protein and within 4 of resname B11')

In [6]: repr(contacts)
Out[6]: "<Selection: 'protein and within 4 of resname B11' from 1zz2 (50 atoms)>

'protein and within 4 of resname B11' is interpreted as select protein atoms that are within 4 Å of residue whose name is B11. This selects protein atoms that within 4 Å of the inhibitor.

7.2 Contacts between different atom groups

In some cases, the protein and the ligand may be in separate files. We will imitate this case by making copies of protein and ligand.

http://www.pdb.org/pdb/explore/explore.do?structureId=1zz2

http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=B11

---

1 http://www.pdb.org/pdb/explore/explore.do?structureId=1zz2

2 http://www.pdb.org/pdb/ligand/ligandsummary.do?hetId=B11
In [7]: inhibitor = pdb.select('resname B11').copy()

In [8]: repr(inhibitor)
Out[8]: "<AtomGroup: 1zz2 Selection ‘resname B11’ (33 atoms)>"

In [9]: protein = pdb.select('protein').copy()

In [10]: repr(protein)
Out[10]: "<AtomGroup: 1zz2 Selection ‘protein’ (2716 atoms)>"

We see that inhibitor molecule contains 33 atoms.

Now we have two different atom groups, and we want protein atoms that are within 4 Å of the inhibitor.

In [11]: contacts = protein.select('within 4 of inhibitor', inhibitor=inhibitor)

In [12]: repr(contacts)
Out[12]: "<Selection: ‘index 227 230 2... 1354 1356 1358’ from 1zz2 Selection ‘protein’ (50 atoms)>"

We found that 50 protein atoms are contacting with the inhibitor. In this case, we passed the atom group inhibitor as a keyword argument to the selection function. Note that the keyword must match that is used in the selection string.

### 7.3 Composite contact selections

Now, let’s try something more sophisticated. We select Cα atoms of residues that have at least one atom interacting with the inhibitor:

In [13]: contacts_ca = protein.select(
   ....:   'calpha and (same residue as within 4 of inhibitor)',
   ....:   inhibitor=inhibitor)

In [14]: repr(contacts_ca)
Out[14]: "<Selection: ‘index 225 232 2... 1328 1351 1359’ from 1zz2 Selection ‘protein’ (20 atoms)>"

In this case, ‘calpha and (same residue as within 4 of inhibitor)’ is interpreted as select Cα atoms of residues that have at least one atom within 4 Å of any inhibitor atom.

This shows that, 20 residues have atoms interacting with the inhibitor.

### 7.4 Spherical atom selections

Similarly, one can give arbitrary coordinate arrays as keyword arguments to identify atoms in a spherical region. Let’s find backbone atoms within 5 Å of point (25, 73, 13):

In [15]: sel = protein.select('backbone and within 5 of somepoint',
   ....:   somepoint=np.array((25, 73, 13)))

### 7.5 Fast contact selections

For repeated and faster contact identification Contacts class is recommended.
We pass the protein as argument:

```
In [16]: protein_contacts = Contacts(protein)
```

The following corresponds to "within 4 of inhibitor":

```
In [17]: contants = protein_contacts.select(4, inhibitor)
```

```
In [18]: repr(contacts)
Out[18]: "<Selection: 'index 227 230 2... 1354 1356 1358' from 1zz2 Selection 'protein' (50 atoms)>"
```

This method is 20 times faster than the one in the previous part, but it is limited to selecting only contacting atoms (other selection arguments cannot be passed). Again, it should be noted that `Contacts` does not update the KDTree that it uses, so it should be used if protein coordinates does not change between selections.
LIGAND EXTRACTION

This example shows how to align structures of the same protein and extract bound ligands from these structures.

matchAlign() function can be used for aligning protein structures. This example shows how to use it to extract ligands from multiple PDB structures after superposing the structures onto a reference. Output will be PDB files that contain ligands superposed onto the reference structure.

8.1 Parse reference and blast search

We start by importing everything from the ProDy package:

```python
In [1]: from prody import *
In [2]: from pylab import *
In [3]: ion()
```

First, we parse the reference structure and blast search PDB for similar structure:

```python
In [4]: p38 = parsePDB('1p38')
In [5]: seq = p38['A'].getSequence()
In [6]: blast_record = blastPDB(seq)
```

It is a good practice to save this record on disk, as NCBI may not respond to repeated searches for the same sequence. We can do this using Python standard library pickle\(^1\) as follows:

```python
In [7]: import pickle
```

Record is save using `dump()`\(^2\) function into an open file:

```python
In [8]: pickle.dump(blast_record, open('p38_blast_record.pkl', 'w'))
```

Then, it can be loaded using `load()`\(^3\) function:

```python
In [9]: blast_record = pickle.load(open('p38_blast_record.pkl'))
```

---

\(^1\)http://docs.python.org/library/pickle.html#pickle
\(^2\)http://docs.python.org/library/pickle.html#pickle.dump
\(^3\)http://docs.python.org/library/pickle.html#pickle.load
8.2 Align structures and extract ligands

Then, we parse the hits one-by-one, superpose them onto the reference structure, and extract ligands:

```python
In [10]: for pdb_id in blast_record.getHits():
....:     # blast search may return PDB identifiers of deprecated structures,
....:     # so we parse structures within a try statement
....:     try:
....:         pdb = parsePDB(pdb_id)
....:         pdb = matchAlign(pdb, p38)[0]
....:     except:
....:         continue
....:     else:
....:         ligand = pdb.select('not protein and not water')
....:         repr(ligand)
....:         if ligand:
....:             writePDB(pdb_id + '_ligand.pdb', ligand)

In [11]: !ls *_ligand.pdb
```

Ligands bound to p38 are outputted. Note that output PDB files may contain multiple ligands.

The output can be loaded into a molecular visualization tool for analysis.

Acknowledgments

Continued development of Protein Dynamics Software ProDy is supported by NIH through R01 GM099738
award. Development of this tutorial is supported by NIH funded Biomedical Technology and Research Center (BTRC) on *High Performance Computing for Multiscale Modeling of Biological Systems* (MMBios⁴) (P41 GM103712).

⁴http://mmbios.org/