ProDy

Protein Dynamics & Sequence Analysis

Structure Analysis

Release

Ahmet Bakan, Cihan Kaya

December 04, 2024

CONTENTS

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

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

3 Blast Search PDB 9
- 3.1 Blast search 9
- 3.2 Best match 10
- 3.3 PDB hits 10
- 3.4 Download hits 10

4 Building Biomolecules 11
- 4.1 Build a Multimer 11
- 4.2 Build a Tetramer 13

5 Alignments 15
- 5.1 Parse an NMR structure 15
- 5.2 Calculate RMSD 15
- 5.3 Align coordinate sets 16
- 5.4 Write aligned coordinates 17

6 Structure Comparison 19
- 6.1 Match chains 19
- 6.2 Map onto a chain 22

7 Intermolecular Contacts 25
- 7.1 Simple contact selections 25
- 7.2 Contacts between different atom groups 25
- 7.3 Composite contact selections 26
- 7.4 Spherical atom selections 26
- 7.5 Fast contact selections 26

8 Ligand Extraction 29
- 8.1 Parse reference and blast search 29
- 8.2 Align structures and extract ligands 29

CHAPTER

ONE

INTRODUCTION

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

1.1 Required Programs

Latest version of **ProDy_** and **Matplotlib_** are required.

1.2 Recommended Programs

IPython_ 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.

CHAPTER TWO

PDB FILES

This examples demonstrates how to use the flexible PDB fetcher, `fetchPDB()`. Valid inputs are PDB identifier, e.g **:pdb:‘2k39‘**, 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:

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

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

```
In [2]: filename = fetchPDB('5uoj')

In [3]: filename
Out[3]: '5uoj.pdb.gz'
```

2.1.2 Multiple files

This function also accepts a list of PDB identifiers:

```
In [4]: filenames = fetchPDB(['5uoj', '1r39', '@!~#'])

In [5]: filenames
Out[5]: ['5uoj.pdb.gz', '1r39.pdb.gz', 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:

```
@> 5uoj (./5uoj.pdb.gz) is found in the target directory.
@> @!~# is not a valid identifier.
@> 1r39 downloaded (./1r39.pdb.gz)
@> PDB download completed (1 found, 1 downloaded, 1 failed).
```

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. `"5uoj.pdb.gz"`
- PDB identifier, e.g. **:pdb:'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* for more information):

```
In [6]: fetchPDB('5uoj')
Out[6]: '5uoj.pdb.gz'

In [7]: atoms = parsePDB('5uoj')

In [8]: atoms
Out[8]: <AtomGroup: 5uoj (3138 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)>
```

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: 3mkbA (1198 atoms)>

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

In [18]: chC
Out[18]: <AtomGroup: 3mkbC (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: 3mkbAC (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: 2k39A_ca (76 atoms)>
```

2.2.8 Header data

PDB parser can be used to extract header data in a `dict`[1] from PDB files as follows:

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

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

In [33]: header['experiment']
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.

---

[1] http://docs.python.org/library/stdtypes.html#dict

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'
```

CHAPTER THREE

BLAST SEARCH 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[2].

We will used amino acid sequence of a protein, e.g. `ASFPVEILPFLYLGCAKDSTNLDVLEEFGIKYILNVTPNLPNLF...YDIVKMKKSN`

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

Output will be `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
   ...: PNLFENAGEFKYKQIPISDHWSQNLSQFFPEAISFIDEAR
   ...: GKNCGVLVHSLAGISRSVTVTVAYLMQKLNLSMNDAYDIV
   ...: KMKKSNISPNFNFMGQLLDFERTL''')
   ...:
```

`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', 'wb'))
```

Then, it can be loaded using `load()`[5] function:

```
In [5]: blast_record = pickle.load(open('mkp3_blast_record.pkl', 'rb'))
```

---

[2] http://www.bahargroup.org/prody/tutorials/ensemble_analysis/index.html#pca

[3] http://docs.python.org/library/pickle.html#module-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:

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

In [7]: best['pdb_id']
Out[7]: '1mkp'

In [8]: best['percent_identity']
Out[8]: 100.0
```

3.3 PDB hits

```
In [9]: hits = blast_record.getHits(percent_identity=90, percent_overlap=70)

In [10]: list(hits)
Out[10]: ['1mkp']
```

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

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

In [12]: list(hits)
Out[12]: ['1m3g', '2hxp', '3lj8', '3ezz', '1mkp']

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

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

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

```
In [15]: hits['1zzw']['percent_identity']
Out[15]: 49.27536231884058

In [16]: hits['1zzw']['align-len']
Out[16]: 138

In [17]: hits['1zzw']['identity']
Out[17]: 68
```

To obtain all hits, simply run the function without specifying parameters:

```
In [18]: all_hits = blast_record.getHits()
```

3.4 Download hits

PDB hits can be downloaded using `fetchPDB()` function:

```
filenames = fetchPDB(hits.keys())
filenames
```

CHAPTER

FOUR

BUILDING BIOMOLECULES

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:

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

In [2]: from pylab import *

In [3]: ion()
```

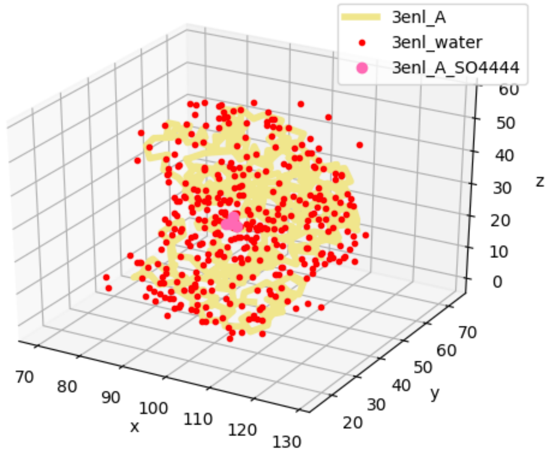
4.1 Build a Multimer

Let’s build the dimeric form of **:pdb:‘3enl‘** of **:wiki:‘enolase‘**:

```
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.

```
In [6]: showProtein(monomer);

In [7]: legend();
```

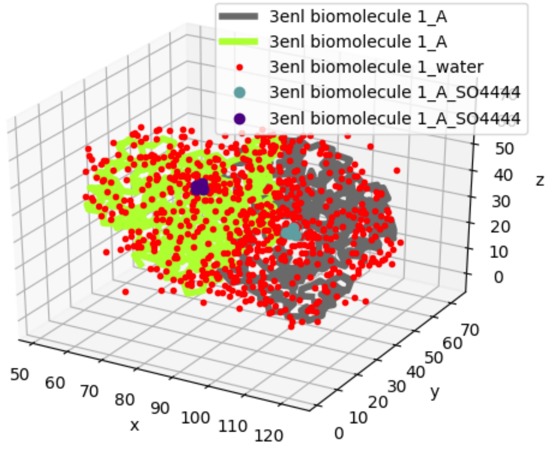


Let's get the dimer coordinates using `buildBiomolecules()` function:

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

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

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

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

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



The *dimer* object now has two chains:

```
In [12]: list(dimer.iterChains())
Out[12]:
```

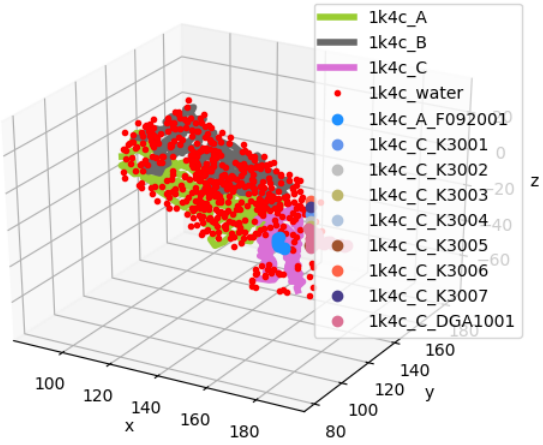
```
[<Chain: A from Segment 1 from 3enl biomolecule 1 (790 residues, 3647 atoms)>,
 <Chain: A from Segment 2 from 3enl biomolecule 1 (790 residues, 3647 atoms)>]
```

4.2 Build a Tetramer

Let’s build the tetrameric form of **:pdb:‘1k4c‘** of **:wiki:‘KcsA_potassium_channel‘**:

```
In [13]: monomer, header = parsePDB('1k4c', header=True)

In [14]: monomer
Out[14]: <AtomGroup: 1k4c (4534 atoms)>
```

```
In [15]: showProtein(monomer);

In [16]: legend();
```



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

```
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)>
```

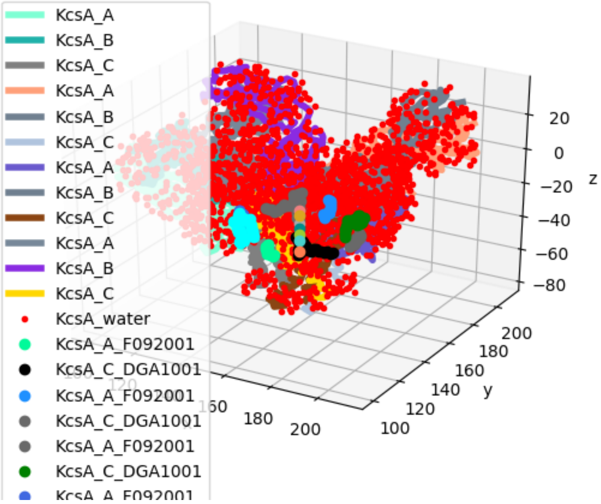
```
In [21]: tetramer = buildBiomolecules(header, without_K)

In [22]: tetramer
Out[22]: <AtomGroup: 1k4c Selection 'not (name K)' biomolecule 1 (18108 atoms)>
```

Now, let’s append potassium ions to the tetramer:

```
In [23]: potassium.setChids('K')
In [24]: kcsa = tetramer + potassium.copy()

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

Here is a view of the tetramer:

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

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



Let's get a list of all the chains:

```
In [28]: list(kcsa.iterChains())
Out[28]:
[<Chain: A from Segment 1 from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment 1 from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment 1 from KcsA (162 residues, 854 atoms)>,
 <Chain: A from Segment 2 from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment 2 from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment 2 from KcsA (162 residues, 854 atoms)>,
 <Chain: A from Segment 3 from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment 3 from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment 3 from KcsA (162 residues, 854 atoms)>,
 <Chain: A from Segment 4 from KcsA (426 residues, 1822 atoms)>,
 <Chain: B from Segment 4 from KcsA (417 residues, 1851 atoms)>,
 <Chain: C from Segment 4 from KcsA (162 residues, 854 atoms)>,
 <Chain: K from Segment  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:

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

CHAPTER

FIVE

ALIGNMENTS

`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 **:pdb:‘1joy‘** that contains 21 models homodimeric domain of EnvZ protein from E. coli.

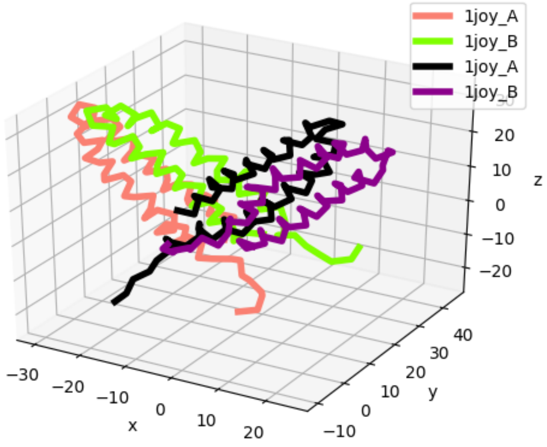
```
In [4]: pdb = parsePDB('1joy')

In [5]: pdb.numCoordsets()
Out[5]: 21
```

5.2 Calculate RMSD

```
In [6]: rmsds = calcRMSD(pdb)

In [7]: rmsds.mean()
Out[7]: 37.506911678400954
```

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);

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



5.3 Align coordinate sets

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

```
In [12]: pdb.setACSIndex(0)

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

To use all backbone atoms, `pdb.backbone` can be passed as argument. See Atom Selections[6] for more information on making selections.

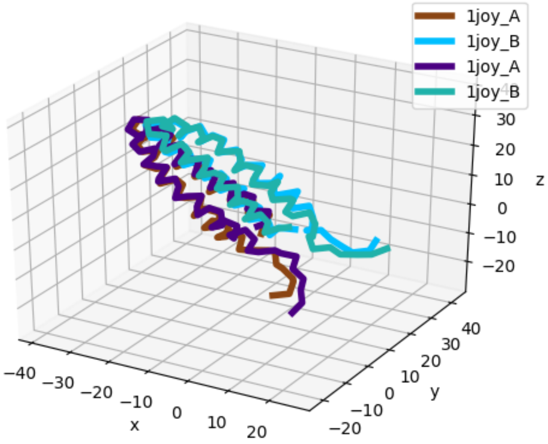
Coordinate sets are superposed onto the first model (the active coordinate set).

```
In [14]: rmsds = calcRMSD(pdb)

In [15]: rmsds.mean()
Out[15]: 3.276891215176855
```

```
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();
```

---

[6] http://www.bahargroup.org/prody/manual/reference/atomic/select.html#selections



5.4 Write aligned coordinates

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

```
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 PDB 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 **:wiki:‘Reverse Transcriptase‘** structures.

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

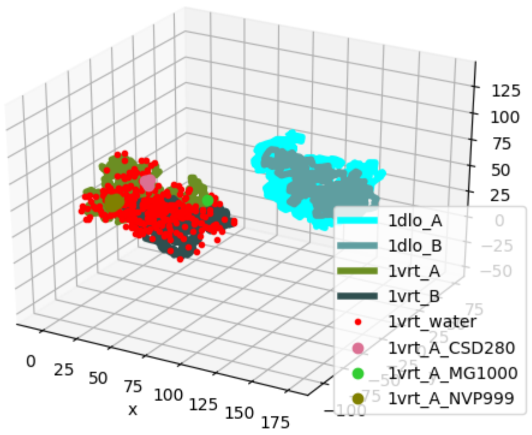
```
In [4]: def printMatch(match):
   ...:     print('Chain 1     : {}'.format(match[0]))
   ...:     print('Chain 2     : {}'.format(match[1]))
   ...:     print('Length      : {}'.format(len(match[0])))
   ...:     print('Seq identity: {}'.format(match[2]))
   ...:     print('Seq overlap : {}'.format(match[3]))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

Now let’s parse bound RT structure **:pdb:‘1vrt‘** and unbound structure **:pdb:‘1dlo‘**:

```
In [5]: bound_all = parsePDB('1vrt')

In [6]: unbound_all = parsePDB('1dlo')
```

Let’s verify that these structures are not aligned:

```
In [7]: showProtein(unbound_all, bound_all);

In [8]: legend();
```



We find matching chains as follows:

We first select just the protein for matching:

```
In [9]: bound = bound_all.protein

In [10]: unbound = unbound_all.protein
```

```
In [11]: matches = matchChains(bound, unbound)

In [12]: 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. The chains in the matches contain only Cα atoms:

```
In [13]: match[0][0].iscalpha
Out[13]: True

In [14]: match[0][1].iscalpha
Out[14]: True
```

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

```
In [15]: bound_ca = matches[0][0] + matches[1][0]

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

In [17]: unbound_ca = matches[0][1] + matches[1][1]

In [18]: unbound_ca
Out[18]: <AtomMap: (AtomMap Chain B from 1dlo -> Chain B from 1vrt) + (AtomMap Chain A from 1dlo -> (
```

Let’s calculate RMSD:

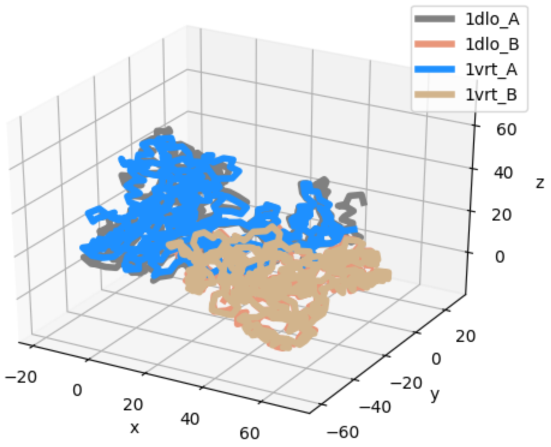
```
In [19]: calcRMSD(bound_ca, unbound_ca)
Out[19]: 129.34348658001392
```

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

```
In [20]: calcTransformation(unbound_ca, bound_ca).apply(unbound);

In [21]: calcRMSD(bound_ca, unbound_ca)
Out[21]: 6.0020747465625375
```

Let’s see the aligned structures now:

```
In [22]: showProtein(unbound, bound);

In [23]: 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 [24]: matches = matchChains(bound, unbound, subset='bb')

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

In [27]: 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 for the analysis of heterogeneous datasets (see Ensemble Analysis[7]).

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

```
In [28]: 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 : {}\n'.format(mapping[3]))
   ....:
```

```
In [29]: unbound_hv = unbound.getHierView()

In [30]: unbound_A = unbound_hv['A']

In [31]: mappings = mapOntoChain(bound, unbound_A)
```

---

[7] http://www.bahargroup.org/prody/tutorials/ensemble_analysis/index.html#pca

```
In [32]: for mapping in mappings:
   ....:     printMapping(mapping)
   ....:
Mapped chain     : AtomMap Chain A from 1vrt -> Chain A from 1dlo
Target chain     : AtomMap Chain A from 1dlo -> Chain A from 1vrt
Mapping length   : 4370
# of mapped atoms: 4159
# of dummy atoms : 211
Sequence identity: 99
Sequence overlap : 94

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: 3209
# of dummy atoms : 1161
Sequence identity: 99
Sequence overlap : 72
```

`mapOntoChain()` mapped all atoms. *subset* argument allows for matching other sets of atoms. We can map backbone atoms as follows:

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

In [34]: for mapping in mappings:
   ....:     printMapping(mapping)
   ....:
Mapped chain     : AtomMap Chain A from 1vrt -> Chain A from 1dlo
Target chain     : AtomMap Chain A from 1dlo -> Chain A from 1vrt
Mapping length   : 2224
# of mapped atoms: 2096
# of dummy atoms : 128
Sequence identity: 99
Sequence overlap : 94

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: 1604
# of dummy atoms : 620
Sequence identity: 99
Sequence overlap : 72
```

Or, we can map all atoms as follows:

```
In [35]: mappings = mapOntoChain(bound, unbound_A, subset='all')

In [36]: for mapping in mappings:
   ....:     printMapping(mapping)
   ....:
Mapped chain     : AtomMap Chain A from 1vrt -> Chain A from 1dlo
Target chain     : AtomMap Chain A from 1dlo -> Chain A from 1vrt
Mapping length   : 4370
# of mapped atoms: 4159
# of dummy atoms : 211
Sequence identity: 99
Sequence overlap : 94
```

```
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: 3209
# of dummy atoms : 1161
Sequence identity: 99
Sequence overlap : 72
```

CHAPTER

SEVEN

INTERMOLECULAR CONTACTS

This examples 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 points 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')
```

:pdb:‘1zz2‘ contains an inhibitor bound p38 MAP kinase structure. Residue name of inhibitor is **:pdbhet:‘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 wit... of resname B11' from 1zz2 (50 atoms)>"
```

`’protein and within 4 of resname B11’` is interpreted as select protein atoms that are within 4 A of residue whose name is B11. This selects protein atoms that within 4 A 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.

```
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 A 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.

CHAPTER

EIGHT

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:

```
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:

```
In [4]: p38 = parsePDB('5uoj')

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`[8] as follows:

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

Record is save using `dump()`[9] function into an open file:

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

Then, it can be loaded using `load()`[10] function:

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

8.2 Align structures and extract ligands

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

---

[8] http://docs.python.org/library/pickle.html#module-pickle

[9] http://docs.python.org/library/pickle.html#pickle.dump

[10] http://docs.python.org/library/pickle.html#pickle.load

```
In [10]: for pdb_id in blast_record.getHits(90, 70):
   ....:     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
1w83_ligand.pdb  3d7z_ligand.pdb  3fmk_ligand.pdb  3oef_ligand.pdb
1wbo_ligand.pdb  3d83_ligand.pdb  3fmn_ligand.pdb  3zsg_ligand.pdb
1wbs_ligand.pdb  3e92_ligand.pdb  3iph_ligand.pdb  3zya_ligand.pdb
2fst_ligand.pdb  3e93_ligand.pdb  3k3i_ligand.pdb
2npq_ligand.pdb  3f14_ligand.pdb  3kq7_ligand.pdb
2zaz_ligand.pdb  3fln_ligand.pdb  3mpt_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* and associated programs is partially supported by the NIH[11]-funded R01 GM139297 entitled *"Toward a deeper understanding of allostery and allotargeting by computational approaches"*.

---

[11] http://www.nih.gov/