Ensemble 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 NMR Models **2**
- 2.1 Notes . . . 2

3 Homologous Proteins **5**
- 3.1 Setup . . . 5
- 3.2 Parameters . . . 6
- 3.3 Blast and download . . . 6
- 3.4 Set reference . . . 6
- 3.5 Prepare ensemble . . . 7
- 3.6 Align PDB files . . . 8
- 3.7 Perform PCA . . . 8
- 3.8 Plot results . . . 9

4 Heterogeneous X-ray Structures **11**
- 4.1 Calculations . . . 11
- 4.2 Analysis . . . 15
- 4.3 Plotting . . . 17
- 4.4 Visualization . . . 25

5 Multimeric Structures **27**
- 5.1 Input and Parameters . . . 27
- 5.2 Prepare Ensemble . . . 29
- 5.3 Perform PCA . . . 31
- 5.4 Plot results . . . 31

Bibliography **33**

CHAPTER

ONE

INTRODUCTION

This tutorial shows how to analyze ensembles of experimental structures.

1.1 Required Programs

Latest version of ProDy[1] and Matplotlib[2] are required.

1.2 Recommended Programs

IPython[3] and Scipy[4] are recommended for this tutorial.

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
[4] http://www.scipy.org

CHAPTER

TWO

NMR MODELS

This example shows how to perform principal component analysis (PCA) of an ensemble of NMR models. The protein of interest is ubiquitin[1], and for illustration puposes, we will repeat the calculations for ubiquitin that was published in [AB09] (page 33).

A `PCA` object that stores covariance matrix and principal modes that describe the dominant changes in the dataset will be obtained. `PCA` and principal modes (`Mode`) can be used as input to functions in `dynamics` module for further analysis.

2.1 Notes

Note that this example is slightly different from that in the *ProDy Tutorial*[2]. This example uses `Ensemble` which has a method for performing iterative superposition.

Also, note that this example applies to any PDB file that contains multiple models.

2.1.1 Prepare ensemble

We start by importing everything from the ProDy package:

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

In [2]: from pylab import *

In [3]: ion()
```

We parse only Cα atoms using `parsePDB()` (note that it is possible to repeat this calculation for all atoms):

```
In [4]: ubi = parsePDB('2k39', subset='calpha')
```

We use residues 1 to 70, as residues 71 to 76 are very mobile and including them skews the results.

```
In [5]: ubi = ubi.select('resnum < 71').copy()

In [6]: ensemble = Ensemble('Ubiquitin NMR ensemble')

In [7]: ensemble.setCoords( ubi.getCoords() )
```

Then, we add all of the coordinate sets to the ensemble, and perform an iterative superposition:

[1] http://en.wikipedia.org/wiki/ubiquitin

[2] http://prody.csb.pitt.edu/tutorials/prody_tutorial/index.html#tutorial

```
In [8]: ensemble.addCoordset( ubi.getCoordsets() )

In [9]: ensemble.iterpose()
```

2.1.2 PCA calculations

Performing `PCA` is only three lines of code:

```
In [10]: pca = PCA('Ubiquitin')

In [11]: pca.buildCovariance(ensemble)

In [12]: pca.calcModes()

In [13]: repr(pca)
Out[13]: '<PCA: Ubiquitin (20 modes; 70 atoms)>'
```

Faster method

Principal modes can be calculated faster using singular value decomposition:

```
In [14]: svd = PCA('Ubiquitin')

In [15]: svd.performSVD(ensemble)
```

For heterogeneous NMR datasets, both methods yields identical results:

```
In [16]: abs(svd.getEigvals()[:20] - pca.getEigvals()).max()
Out[16]: 7.9936057773011271e-15

In [17]: abs(calcOverlap(pca, svd).diagonal()[:20]).min()
Out[17]: 0.99999999999999956
```

2.1.3 Write NMD file

Write principal modes into an *NMD Format*[3] file for NMWiz using `writeNMD()` function:

```
In [18]: writeNMD('ubi_pca.nmd', pca[:3], ubi)
Out[18]: 'ubi_pca.nmd'
```

2.1.4 Print data

Let's print fraction of variance for top ranking 4 PCs (listed in Table S3):

```
In [19]: for mode in pca[:4]:
   ....:     print calcFractVariance(mode).round(3)
   ....:
0.134
0.094
0.083
0.065
```

[3]http://prody.csb.pitt.edu/manual/reference/dynamics/nmdfile.html#nmd-format

2.1.5 Compare with ANM results

We set the active coordinate set to 79, which is the one that is closest to the mean structure (note that indices start from 0 in Python). Then, we perform ANM calculations using `calcANM()` for the active coordset:

```
In [20]: ubi.setACSIndex(78)

In [21]: anm, temp = calcANM(ubi)

In [22]: anm.setTitle('Ubiquitin')
```

We calculate overlaps between ANM and PCA modes (presented in Table 1). `printOverlapTable()` function is handy to print a formatted overlap table:

```
In [23]: printOverlapTable(pca[:4], anm[:4])
Overlap Table
                          ANM Ubiquitin
                     #1     #2     #3     #4
PCA Ubiquitin #1   -0.19  -0.30  +0.22  -0.62
PCA Ubiquitin #2   +0.09  -0.72  -0.16  +0.16
PCA Ubiquitin #3   +0.31  -0.06  -0.23   0.00
PCA Ubiquitin #4   +0.11  +0.02  +0.16  -0.31
```

CHAPTER

THREE

HOMOLOGOUS PROTEINS

This example shows how to perform PCA of a structural dataset obtained by blast searching PDB. The protein of interest is cytochrome c[1] (cyt *c*). Dataset will contain structures sharing 44% or more sequence identity with human *cyt c*, i.e. its homologs and/or orthologs.

A `PCA` instance that stores covariance matrix and principal modes that describe the dominant changes in the dataset will be obtained. `PCA` instance and principal modes (`Mode`) can be used as input to functions in `dynamics` module for further analysis.

Input is amino acid sequence of the protein, a reference PDB identifier, and some parameters.

3.1 Setup

Import ProDy and matplotlib into the current namespace.

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

In [2]: from pylab import *

In [3]: ion()
```

Name of the protein (a name without a white space is preferred)

```
In [4]: name = 'cyt_c'

In [5]: sequence = '''GDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKTGQAPGFTYTDANKNKGITWKEE
   ...: TLMEYLENPKKYIPGTKMIFAGIKKKTEREDLIAYLKKATNE'''
   ...:

In [6]: ref_pdb = '1hrc'
```

Optionally, a list of PDB files to be excluded from analysis can be provided. In this case dimeric Cyt c structures are excluded from the analysis. To use all PDB hits, provide an empty list.

```
In [7]: exclude = ['3nbt', '3nbs']
```

[1] http://en.wikipedia.org/wiki/cytochrome c

3.2 Parameters

```
# Minimum sequence identity of hits
In [8]: seqid = 44

In [9]: # Reference chain identifier

In [10]: ref_chid = 'A'

# Selection string ("all" can be used if all of the chain is to be analyzed)
In [11]: selstr = 'resnum 1 to 103'
```

3.3 Blast and download

List of PDB structures can be obtained using `blastPDB()` as follows:

```
In [12]: blast_record = blastPDB(sequence)
```

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`[2] as follows:

```
In [13]: import pickle
```

Record is save using `dump()`[3] function into an open file:

```
In [14]: pickle.dump(blast_record, open('cytc_blast_record.pkl', 'w'))
```

Then, it can be loaded using `load()`[4] function:

```
In [15]: blast_record = pickle.load(open('cytc_blast_record.pkl'))

In [16]: pdb_hits = []

In [17]: for key, item in blast_record.getHits(seqid).iteritems():
   ....:     pdb_hits.append((key, item['chain_id']))
   ....:
```

Let's fetch PDB files and see how many there are:

```
In [18]: pdb_files = fetchPDB(*[pdb for pdb, ch in pdb_hits], compressed=False)

In [19]: len(pdb_files)
Out[19]: 99
```

3.4 Set reference

We first parse the reference structure. Note that we parse only Cα atoms from chain A. The analysis will be performed for a single chain (monomeric) protein. For analysis of a dimeric protein see *Multimeric Structures* (page 27)

[2] http://docs.python.org/library/pickle.html#pickle
[3] http://docs.python.org/library/pickle.html#pickle.dump
[4] http://docs.python.org/library/pickle.html#pickle.load

```
In [20]: reference_structure = parsePDB(ref_pdb, subset='ca', chain=ref_chid)

In [21]: # Get the reference chain from this structure

In [22]: reference_hierview = reference_structure.getHierView()

In [23]: reference_chain = reference_hierview[ref_chid]
```

3.5 Prepare ensemble

```
# Start a log file
In [24]: startLogfile('pca_blast')

In [25]: # Instantiate a PDB ensemble

In [26]: ensemble = PDBEnsemble(name)

In [27]: # Set ensemble atoms

In [28]: ensemble.setAtoms(reference_chain)

In [29]: # Set reference coordinates

In [30]: ensemble.setCoords(reference_chain.getCoords())

In [31]: for (pdb_id, chain_id), pdb_file in zip(pdb_hits, pdb_files):
   ....:     if pdb_id in exclude:
   ....:         continue
   ....:     structure = parsePDB(pdb_file, subset='calpha', chain=chain_id)
   ....:     if structure is None:
   ....:         plog('Failed to parse ' + pdb_file)
   ....:         continue
   ....:     mappings = mapOntoChain(structure, reference_chain, seqid=seqid)
   ....:     if len(mappings) == 0:
   ....:         plog('Failed to map', pdb_id)
   ....:         continue
   ....:     atommap = mappings[0][0]
   ....:     ensemble.addCoordset(atommap, weights=atommap.getFlags('mapped'))
   ....:

In [32]: ensemble.iterpose()

In [33]: saveEnsemble(ensemble)
Out[33]: 'cyt_c.ens.npz'
```

Let's check how many conformations are extracted from PDB files:

```
In [34]: len(ensemble)
Out[34]: 400
```

Note that number of conformations is larger than the number of PDB structures we retrieved. This is because some of the PDB files contained NMR structures with multiple models. Each model in NMR structures are added to the ensemble as individual conformations.

Write aligned conformations into a PDB file as follows:

```
In [35]: writePDB(name+'.pdb', ensemble)
Out[35]: 'cyt_c.pdb'
```

This file can be used to visualize the aligned conformations in a modeling software.

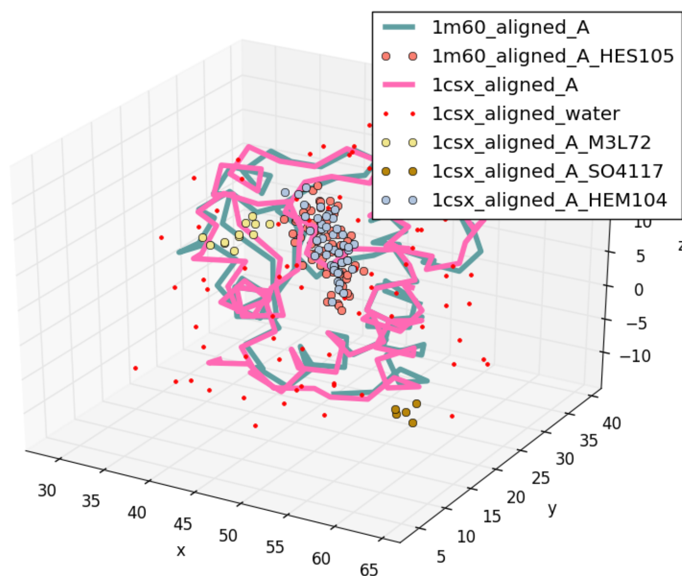
3.6 Align PDB files

`alignPDBEnsemble()` function can be used to align all PDB structures used in the analysis, e.g. `alignPDBEnsemble(ensemble)`. Outputted files will contain intact structures and can be used for visualization purposes in other software. In this case, we will align only select PDB files:

```
In [36]: conf1_alinged = alignPDBEnsemble(ensemble[0])

In [37]: conf2_alinged = alignPDBEnsemble(ensemble[1])
```

Let's take a quick look at the aligned structures:

```
In [38]: showProtein(parsePDB(conf1_alinged), parsePDB(conf2_alinged));

In [39]: legend();
```



3.7 Perform PCA

Once the ensemble is ready, performing PCA is 3 easy steps:

```
# Instantiate a PCA
In [40]: pca = PCA(name)

In [41]: # Build covariance matrix

In [42]: pca.buildCovariance(ensemble)

In [43]: # Calculate modes
```

```
In [44]: pca.calcModes()
```

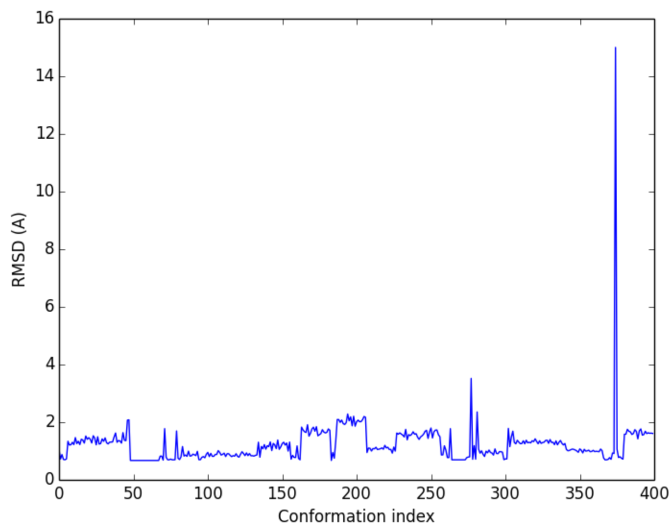
The calculated data can be saved as a compressed file using `saveModel()` function:

```
In [45]: saveModel(pca)
Out[45]: 'cyt_c.pca.npz'
```

3.8 Plot results

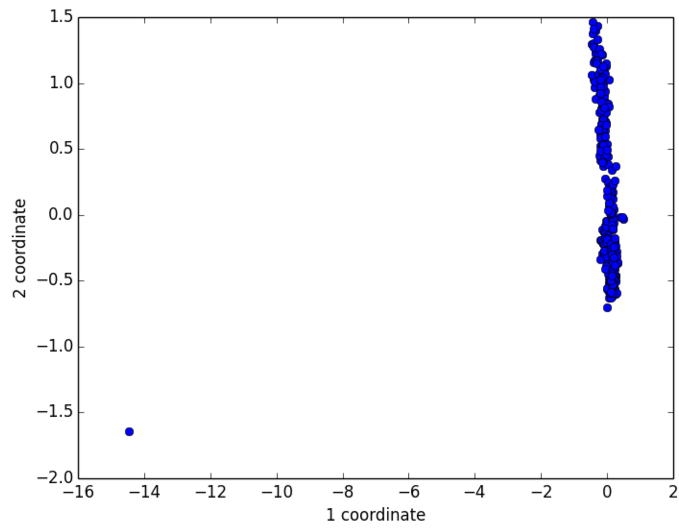
Let's plot RMSDs of all conformations from the average conformation:

```
In [46]: rmsd = calcRMSD(ensemble)

In [47]: plot(rmsd);

In [48]: xlabel('Conformation index');

In [49]: ylabel('RMSD (A)');
```



Let's show a projection of the ensemble onto PC1 and PC2:

```
In [50]: showProjection(ensemble, pca[:2]);
```


2 coordinate
1 coordinate
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−16
−14
−12
−10
−8
−6
−4
−2
0
2

CHAPTER

FOUR

HETEROGENEOUS X-RAY STRUCTURES

This example compares experimental structural data analyzed using Principal Component Analysis (PCA) with the theoretical data predicted by Anisotropic Network Model (ANM):

4.1 Calculations

This is the first part of a lengthy example. In this part, we perform the calculations using a p38 MAP kinase (MAPK) structural dataset. This will reproduce the calculations for p38 that were published in [AB09] (page 33).

We will obtain a `PCA` instance that stores the covariance matrix and principal modes describing the dominant changes in the dataset. The `PCA` instance and principal modes (`Mode`) can be used as input for the functions in `dynamics` module.

4.1.1 Retrieve dataset

We start by importing everything from the ProDy package:

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

In [2]: from pylab import *

In [3]: ion()
```

We use a list of PDB identifiers for the structures that we want to include in our analysis.

```
In [4]: pdbids = ['1A9U', '1BL6', '1BL7', '1BMK', '1DI9', '1IAN', '1KV1', '1KV2',
   ...:           '1LEW', '1LEZ', '1M7Q', '1OUK', '1OUY', '1OVE', '1OZ1', '1P38',
   ...:           '1R39', '1R3C', '1W7H', '1W82', '1W83', '1W84', '1WBN', '1WBO',
   ...:           '1WBS', '1WBT', '1WBV', '1WBW', '1WFC', '1YQJ', '1YW2', '1YWR',
   ...:           '1ZYJ', '1ZZ2', '1ZZL', '2BAJ', '2BAK', '2BAL', '2BAQ', '2EWA',
   ...:           '2FSL', '2FSM', '2FSO', '2FST', '2GFS', '2GHL', '2GHM', '2GTM',
   ...:           '2GTN', '2I0H', '2NPQ', '2OKR', '2OZA', '3HVC', '3MH0', '3MH3',
   ...:           '3MH2', '2PUU', '3MGY', '3MH1', '2QD9', '2RG5', '2RG6', '2ZAZ',
   ...:           '2ZB0', '2ZB1', '3BV2', '3BV3', '3BX5', '3C5U', '3L8X', '3CTQ',
   ...:           '3D7Z', '3D83', '2ONL']
   ...:
```

Note that we used a list of identifiers that are different from what was listed in the supporting material of [AB09] (page 33). Some structure has been refined and their identifier have been changed by the Protein Data Bank. These changes are reflected in the above list.

Also note that it is possible to update this list to include all of the p38 structures currently available in the PDB using the `blastPDB()` function as follows:

```
In [5]: p38_sequence = '''GLVPRGSHMSQERPTFYRQELNKTIWEVPERYQNLSPVGSGAYGSVCAAFDTKTGHRV
   ...: AVKKLSRPFQSIIHAKRTYRELRLLKHMKHENVIGLLDVFTPARSLEEFNDVYLVTHLMGADLNNIVKCQKLTDDH
   ...: VQFLIYQILRGLKYIHSADIIHRDLKPSNLAVNEDCELKILDFGLARHTDDEMTGYVATRWYRAPEIMLNWMHYNQ
   ...: TVDIWSVGCIMAELLTGRTLFPGTDHIDQLKLILRLVGTPGAELLKKISSESARNYIQSLAQMPKMNFANVFIGAN
   ...: PLAVDLLEKMLVLDSDKRITAAQALAHAYFAQYHDPDDEPVADPYDQSFESRDLLIDEWKSLTYDEVISFVPPPLD
   ...: QEEMES'''
   ...:
```

To update list of p38 MAPK PDB files, you can make a blast search as follows:

```
In [6]: blast_record = blastPDB(p38_sequence)

In [7]: pdbids = blast_record.getHits()
```

We use the same set of structures to reproduce the results. After we listed the PDB identifiers, we obtain them using `fetchPDB()` function as follows:

```
In [8]: pdbfiles = fetchPDB(*pdbids, compressed=False)
```

`pdbfiles` variable contains a list of PDB filenames.

4.1.2 Set reference chain

The next step is setting one of the p38 structures as the reference structure. We use 1p38 chain A. Note that we won't use all of the resolved residues in this structure. We select only those residues which are resolved in at least 90% of the dataset.

```
In [9]: ref_structure = parsePDB('1p38')

In [10]: ref_selection = ref_structure.select('resnum 5 to 31 36 to 114 122 to '
   ....:                                      '169 185 to 351 and calpha')
   ....:
```

Retrieve protein chain A from the reference selection:

```
In [11]: ref_chain = ref_selection.getHierView().getChain('A')

In [12]: repr(ref_chain)
Out[12]: '<Chain: A from 1p38 (321 residues, 321 atoms)>'
```

We use the `parsePDB()` function to parse a PDB file. This returns a `AtomGroup` instance. We make a copy of α-carbon atoms of select residues for analysis.

See *Atom Selections*[1] for making selections.

4.1.3 Prepare ensemble

X-ray structural ensembles are heterogenous, i.e. different structures have different sets of unresolved residues. Hence, it is not straightforward to analyzed them as it would be for NMR models (see *NMR Models* (page 2)).

[1] http://prody.csb.pitt.edu/manual/reference/atomic/select.html#selections

ProDy has special functions and classes for facilitating efficient analysis of the PDB X-ray data. In this example we use `mapOntoChain()` function which returns an `AtomMap` instance.

See *How AtomMap's work*[2] for more details.

Start a logfile to save screen output:

```
In [13]: startLogfile('p38_pca')
```

Instantiate an `PDBEnsemble` object:

```
In [14]: ensemble = PDBEnsemble('p38 X-ray')
```

Set atoms and reference coordinate set of the ensemble:

```
In [15]: ensemble.setAtoms(ref_chain)

In [16]: ensemble.setCoords(ref_chain)
```

For each PDB file, we find the matching chain and add it to the ensemble:

```
In [17]: for pdbfile in pdbfiles:
   ....:     # Parse next PDB file. (only alpha carbons, since it's faster)
   ....:     structure = parsePDB(pdbfile, subset='calpha')
   ....:     # Get mapping to the reference chain
   ....:     mappings = mapOntoChain(structure, ref_chain)
   ....:     atommap = mappings[0][0]
   ....:     # Add the atommap (mapped coordinates) to the ensemble
   ....:     # Note that some structures do not completely map (missing residues)
   ....:     # so we pass weights (1 for mapped atoms, 0 for unmapped atoms)
   ....:     ensemble.addCoordset(atommap, weights=atommap.getFlags('mapped'))
   ....:

In [18]: repr(ensemble)
Out[18]: '<PDBEnsemble: p38 X-ray (75 conformations; 321 atoms)>'

In [19]: len(ensemble) == len(pdbfiles)
Out[19]: True
```

Perform an iterative superimposition:

```
In [20]: ensemble.iterpose()
```

Close the logfile (file content shows how chains were paired/mapped):

```
In [21]: closeLogfile('p38_pca')
```

4.1.4 Save coordinates

We use `PDBEnsemble` to store coordinates of the X-ray structures. The `PDBEnsemble` instances do not store any other atomic data. If we want to write aligned coordinates into a file, we need to pass the coordinates to an `AtomGroup` instance. Then we use `writePDB()` function to save coordinates:

```
In [22]: writePDB('p38_xray_ensemble.pdb', ensemble)
Out[22]: 'p38_xray_ensemble.pdb'
```

[2] http://prody.csb.pitt.edu/manual/reference/atomic/atommap.html#atommaps

4.1.5 PCA calculations

Once the coordinate data are prepared, it is straightforward to perform the PCA calculations:

```
In [23]: pca = PCA('p38 xray')            # Instantiate a PCA instance

In [24]: pca.buildCovariance(ensemble)    # Build covariance for the ensemble

In [25]: pca.calcModes()                  # Calculate modes (20 of the by default)
```

Approximate method

In the following we are using singular value decomposition for faster and more memory efficient calculation of principal modes:

```
In [26]: pca_svd = PCA('p38 svd')

In [27]: pca_svd.performSVD(ensemble)
```

The resulting eigenvalues and eigenvectors may show small differences due to missing atoms in the datasets:

```
In [28]: abs(pca_svd.getEigvals()[:20] - pca.getEigvals()).max()
Out[28]: 0.40185220465125226

In [29]: abs(calcOverlap(pca, pca_svd).diagonal()[:20]).min()
Out[29]: 0.99801084515679994
```

Note that building and diagonalizing the covariance matrix is the preferred method for heterogeneous ensembles. For NMR models or MD trajectories SVD method may be preferred over covariance method.

4.1.6 ANM calculations

To perform ANM calculations:

```
In [30]: anm = ANM('1p38')              # Instantiate a ANM instance

In [31]: anm.buildHessian(ref_chain)    # Build Hessian for the reference chain

In [32]: anm.calcModes()                # Calculate slowest non-trivial 20 modes
```

4.1.7 Save your work

Calculated data can be saved in a ProDy internal format to use in a later session or to share it with others.

If you are in an interactive Python session, and wish to continue without leaving your session, you do not need to save the data. Saving data is useful if you want to use it in another session or at a later time, or if you want to share it with others.

```
In [33]: saveModel(pca)
Out[33]: 'p38_xray.pca.npz'

In [34]: saveModel(anm)
Out[34]: '1p38.anm.npz'

In [35]: saveEnsemble(ensemble)
Out[35]: 'p38_X-ray.ens.npz'
```

```
In [36]: writePDB('p38_ref_chain.pdb', ref_chain)
Out[36]: 'p38_ref_chain.pdb'
```

We use the `saveModel()` and `saveEnsemble()` functions to save calculated data. In *Analysis* (page 15), we will use the `loadModel()` and `loadEnsemble()` functions to load the data.

4.2 Analysis

4.2.1 Synopsis

This example is continued from *Calculations* (page 11). The aim of this part is to perform a quantitative comparison of experimental and theoretical data and to print/save the numerical data that were presented in [AB09] (page 33).

We start by importing everything from the ProDy package:

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

In [2]: from pylab import *

In [3]: ion()
```

Then we load data saved in the previous part:

```
In [4]: pca = loadModel('p38_xray.pca.npz')

In [5]: anm = loadModel('1p38.anm.npz')
```

4.2.2 Variance along PCs

Of interest is the fraction of variance that is explained by principal components, which are the dominant modes of variability in the dataset. We can print this information to screen for top 6 PCs as follows:

```
In [6]: for mode in pca[:3]:    # Print % variance explained by top PCs
   ...:     var = calcFractVariance(mode)*100
   ...:     print('{0:s}  % variance = {1:.2f}'.format(mode, var))
   ...:
Mode 1 from PCA p38 xray  % variance = 29.19
Mode 2 from PCA p38 xray  % variance = 16.51
Mode 3 from PCA p38 xray  % variance = 10.63
```

These data were included in Table 1 in [AB09] (page 33).

4.2.3 Collectivity of modes

Collectivity of a normal mode ([BR95]) can be obtained using `calcCollectivity()`:

```
In [7]: for mode in pca[:3]:    # Print PCA mode collectivity
   ...:     coll = calcCollectivity(mode)
   ...:     print('{0:s}  collectivity = {1:.2f}'.format(mode, coll))
   ...:
Mode 1 from PCA p38 xray  collectivity = 0.50
```

```
Mode 2 from PCA p38 xray  collectivity = 0.50
Mode 3 from PCA p38 xray  collectivity = 0.32
```

Similarly, we can get collectivity of ANM modes:

```
In [8]: for mode in anm[:3]:    # Print ANM mode collectivity
   ...:     coll = calcCollectivity(mode)
   ...:     print('{0:s}  collectivity = {1:.2f}'.format(mode, coll))
   ...:
Mode 1 from ANM 1p38  collectivity = 0.65
Mode 2 from ANM 1p38  collectivity = 0.55
Mode 3 from ANM 1p38  collectivity = 0.68
```

This shows that top PCA modes and slow ANM modes are highly collective.

4.2.4 PCA - ANM overlap

We calculate the overlap between PCA and ANM modes in order to see whether structural changes observed upon inhibitor binding correlate with the intrinsic fluctuations of the p38 MAP kinase (Table 1 in [AB09] (page 33)).

There are a number of functions to calculate or show overlaps between modes (see list of them in *Dynamics Analysis*[3]). In an interactive session, most useful is `printOverlapTable()`:

```
In [9]: printOverlapTable(pca[:3], anm[:3]) # Top 3 PCs vs slowest 3 ANM modes
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
```

This formatted table can also be written into a file using `writeOverlapTable()` function.

4.2.5 Save numeric data

`ANM` and `PCA` instances store calculated numeric data. Their class documentation lists methods that return eigenvalue, eigenvector, covariance matrix etc. data to the user. Such data can easily be written into text files for analysis using external software. The function is to use is `writeArray()`:

```
In [10]: writeArray('p38_PCA_eigvecs.txt', pca.getEigvecs() ) # PCA eigenvectors
Out[10]: 'p38_PCA_eigvecs.txt'

In [11]: writeModes('p38_ANM_modes.txt', anm) # ANM modes, same as using above func
Out[11]: 'p38_ANM_modes.txt'
```

It is also possible to write arbitrary arrays:

```
In [12]: overlap = calcOverlap(pca[:3], anm[:3])

In [13]: writeArray('p38_PCA_ANM_overlap.txt', abs(overlap), format='%.2f')
Out[13]: 'p38_PCA_ANM_overlap.txt'
```

[3]http://prody.csb.pitt.edu/manual/reference/dynamics/index.html#dynamics

4.3 Plotting

4.3.1 Synopsis

This example is continued from *Analysis* (page 15). The aim of this part is to produce graphical comparison of experimental and theoretical data. We will reproduce the plots that was presented in our paper [AB09] (page 33).

4.3.2 Load data

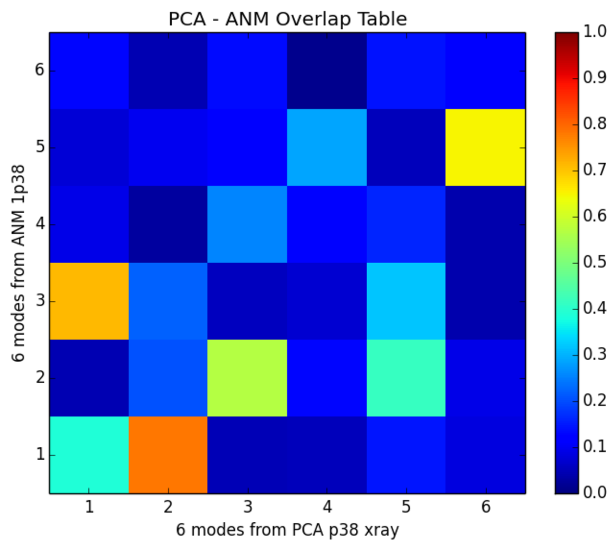
First, we load data saved in *Calculations* (page 11):

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

In [2]: from pylab import *

In [3]: ion()

In [4]: pca = loadModel('p38_xray.pca.npz')

In [5]: anm = loadModel('1p38.anm.npz')

In [6]: ensemble = loadEnsemble('p38_X-ray.ens.npz')

In [7]: ref_chain = parsePDB('p38_ref_chain.pdb')
```

4.3.3 PCA - ANM overlap

In previous page, we compared PCA and ANM modes to get some numbers. In this case, we will use plotting functions to make similar comparisons:

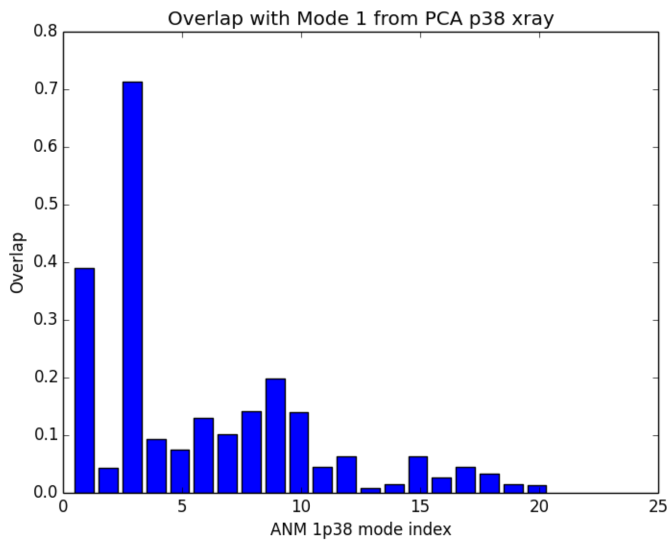
```
In [8]: showOverlapTable(pca[:6], anm[:6]);

In [9]: # Let's change the title of the figure

In [10]: title('PCA - ANM Overlap Table');
```

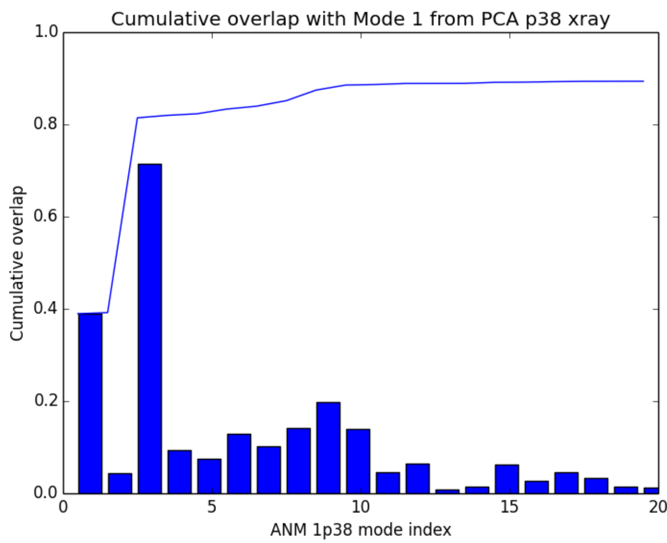


It is also possible to plot overlap of a single mode from one model with multiple modes from another:

```
In [11]: showOverlap(pca[0], anm);
```

```
In [12]: showCumulOverlap(pca[0], anm);
```



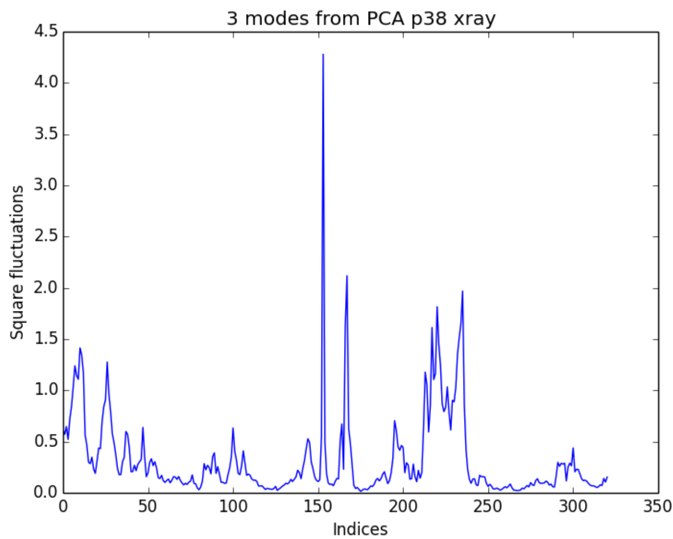


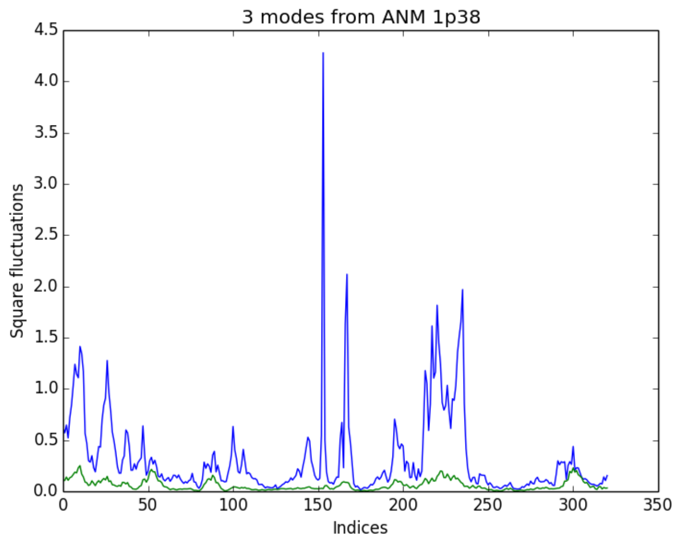
Let's also plot the cumulative overlap in the same figure:

4.3.4 Square fluctuations

```
In [13]: showSqFlucts(pca[:3]);
```

```
In [14]: showSqFlucts(anm[:3]);
```

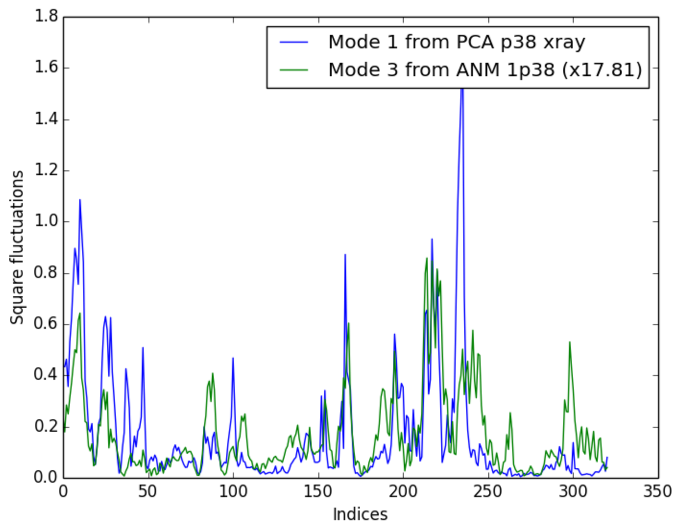




Now let's plot square fluctuations along PCA and ANM modes in the same plot:

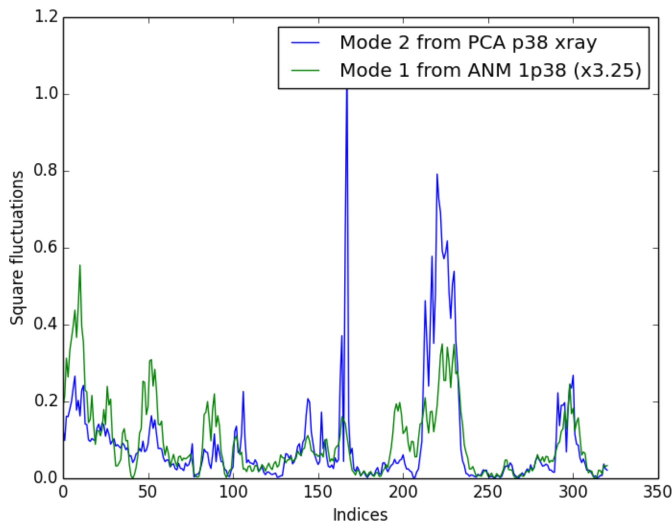
```
In [15]: showScaledSqFlucts(pca[0], anm[2]);
```

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



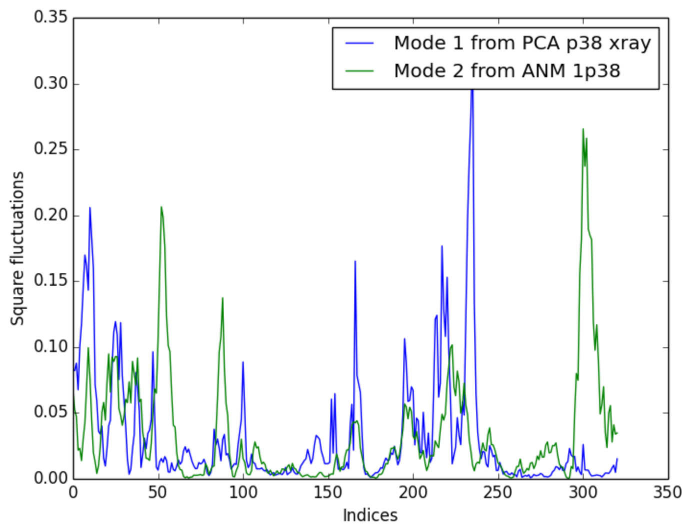

```
In [17]: showScaledSqFlucts(pca[1], anm[0]);
```

```
In [18]: legend();
```



In above example, ANM modes are scaled to have the same mean as PCA modes. Alternatively, we could plot normalized square fluctuations:

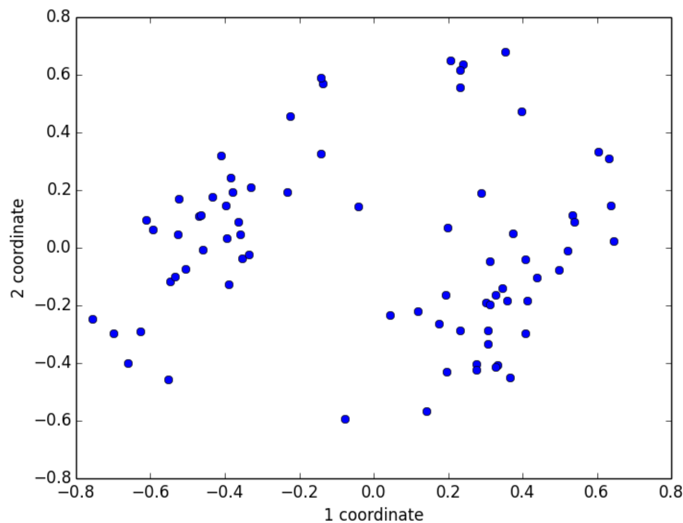
```
In [19]: showNormedSqFlucts(pca[0], anm[1]);

In [20]: legend();
```



4.3.5 Projections

Now we will project the ensemble onto PC 1 and 2 using `showProjection()`:

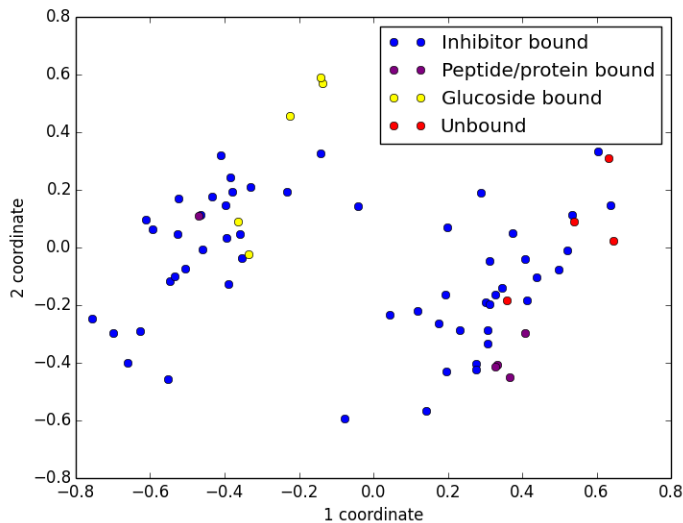
```
In [21]: showProjection(ensemble, pca[:2]);

In [22]: axis([-0.8, 0.8, -0.8, 0.8]);
```



Now we will do a little more work, and get a colorful picture:

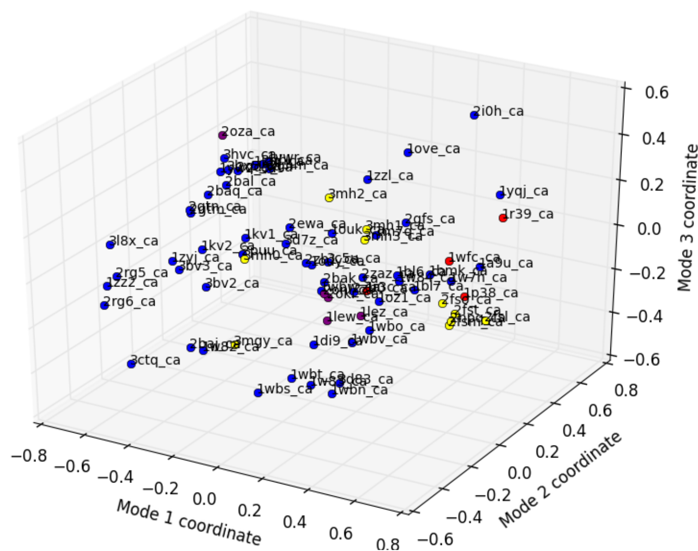
red	unbound
blue	inhibitor bound
yellow	glucoside bound
purple	peptide/protein bound

```
In [23]: color_list = ['blue', 'blue', 'blue', 'blue', 'blue', 'blue', 'blue',
   ....:                'blue', 'purple', 'purple', 'blue', 'blue', 'blue',
   ....:                'blue', 'blue', 'red', 'red', 'red', 'blue', 'blue',
   ....:                'blue', 'blue', 'blue','blue', 'blue', 'blue', 'blue',
   ....:                'blue', 'red', 'blue', 'blue','blue', 'blue', 'blue',
   ....:                'blue', 'blue', 'blue', 'blue', 'blue', 'blue', 'yellow',
   ....:                'yellow', 'yellow', 'yellow', 'blue', 'blue','blue',
   ....:                'blue', 'blue', 'blue', 'yellow', 'purple', 'purple',
   ....:                'blue', 'yellow', 'yellow', 'yellow', 'blue', 'yellow',
   ....:                'yellow', 'blue', 'blue', 'blue', 'blue', 'blue', 'blue',
   ....:                'blue', 'blue', 'blue', 'blue', 'blue', 'blue', 'blue',
   ....:                'blue', 'purple']
   ....:

In [24]: color2label = {'red': 'Unbound', 'blue': 'Inhibitor bound',
   ....:                 'yellow': 'Glucoside bound',
   ....:                 'purple': 'Peptide/protein bound'}
   ....:

In [25]: label_list = [color2label[color] for color in color_list]

In [26]: showProjection(ensemble, pca[:2], color=color_list,
   ....:                 label=label_list);
   ....:

In [27]: axis([-0.8, 0.8, -0.8, 0.8]);

In [28]: legend();
```



Now let's project conformations onto 3d principal space and label conformations using `text` keyword argument and `PDBEnsemble.getLabels()` method:

```
In [29]: showProjection(ensemble, pca[:3], color=color_list, label=label_list,
   ....:                text=ensemble.getLabels(), fontsize=10);
   ....:
```



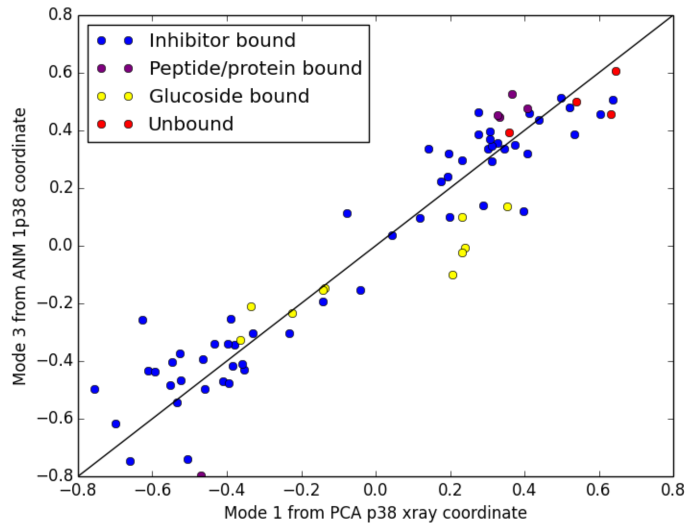
The figure with all conformation labels is crowded, but in an interactive session you can zoom in and out to make text readable.

4.3.6 Cross-projections

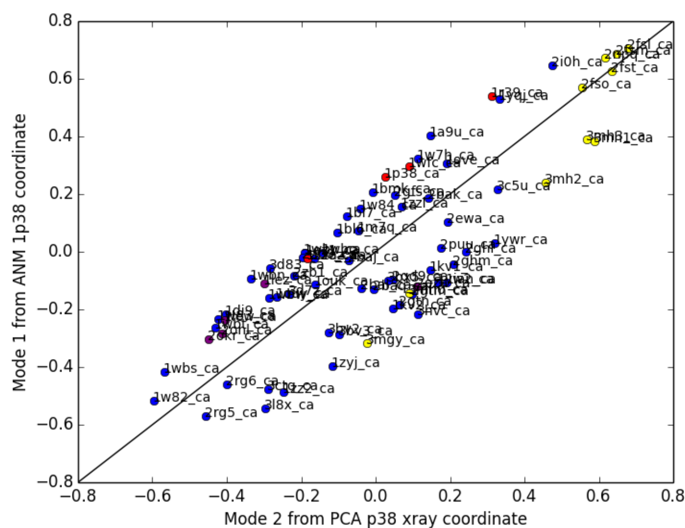
Finally, we will make a cross-projection plot using `showCrossProjection()`. We will pass `scale='y'` argument, which will scale the width of the projection along ANM mode:

```
In [30]: showCrossProjection(ensemble, pca[0], anm[2], scale="y",
   ....:                     color=color_list, label=label_list);
```

```
   ....:

In [31]: plot([-0.8, 0.8], [-0.8, 0.8], 'k');

In [32]: axis([-0.8, 0.8, -0.8, 0.8]);

In [33]: legend(loc='upper left');
```




```
In [34]: showCrossProjection(ensemble, pca[1], anm[0], scale="y",
   ....:                      color=color_list, label=label_list);
   ....:

In [35]: plot([-0.8, 0.8], [-0.8, 0.8], 'k');

In [36]: axis([-0.8, 0.8, -0.8, 0.8]);
```



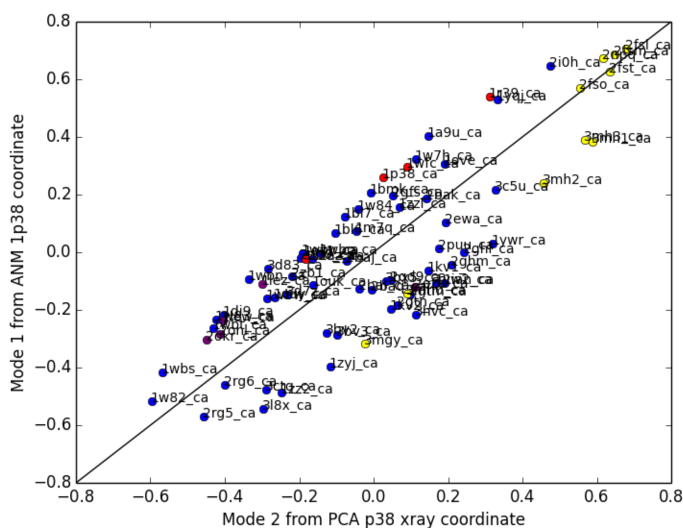
It is also possible to find the correlation between these projections:

```
In [37]: pca_coords, anm_coords = calcCrossProjection(ensemble, pca[0], anm[2])

In [38]: print(np.corrcoef(pca_coords, anm_coords))
[[ 1.         -0.94621454]
 [-0.94621454  1.        ]]
```

This is going to print 0.95 for PC 1 and ANM mode 2 pair.

Finally, it is also possible to label conformations in cross projection plots too:

```
In [39]: showCrossProjection(ensemble, pca[1], anm[0], scale="y",
   ....:     color=color_list, label=label_list, text=ensemble.getLabels(),
   ....:     fontsize=10);
   ....:

In [40]: plot([-0.8, 0.8], [-0.8, 0.8], 'k');

In [41]: axis([-0.8, 0.8, -0.8, 0.8]);
```



4.4 Visualization

4.4.1 Synopsis

This example is continued from *Plotting* (page 17). The aim of this part is visual comparison of experimental and theoretical modes. We will generate molecular graphics that was presented in our paper [AB09] (page 33).

Notes

To make a comparative visual analysis of PCA and ANM modes that were calculated in the previous parts, NMWiz needs to be installed. NMWiz is a VMD[4] plugin designed to complement ProDy.

[4] http://www.ks.uiuc.edu/Research/vmd

4.4.2 Load data

We start by importing everything from the ProDy package:

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

In [2]: from pylab import *

In [3]: ion()
```

Then we load data saved in *Calculations* (page 11):

```
In [4]: pca = loadModel('p38_xray.pca.npz')

In [5]: anm = loadModel('1p38.anm.npz')

In [6]: ensemble = loadEnsemble('p38_X-ray.ens.npz')

In [7]: ref_chain = parsePDB('p38_ref_chain.pdb')
```

4.4.3 Write NMD files

We will save PCA and ANM data in NMD format. NMWiz can read and visualize multiple NMD files at once. Interested user is referred to NMWiz documentation for more information. NMD files are saved as follows using `writeNMD()` functions:

```
In [8]: writeNMD('p38_pca.nmd', pca[:3], ref_chain)
Out[8]: 'p38_pca.nmd'

In [9]: writeNMD('p38_anm.nmd', anm[:3], ref_chain)
Out[9]: 'p38_anm.nmd'
```

It is also possible to load VMD to visualize normal mode data from within an interactive Python session. For this to work, you need VMD and NMWiz plugin installed. Check if VMD path is correct using `pathVMD()`:

```
In [10]: pathVMD()
Out[10]: '/usr/local/bin/vmd'
```

If this is not the correct path to your VMD executable you can change it using the same function.

```
In [11]: viewNMDinVMD('1p38_pca.nmd')
```

CHAPTER

FIVE

MULTIMERIC STRUCTURES

In this part, we perform PCA of HIV Reverse Transcriptase[1] (RT), which is a heterodimer.

5.1 Input and Parameters

First, we make necessary imports:

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

```
In [2]: from pylab import *
```

```
In [3]: ion()
```

5.1.1 Reference structure

We set the name of the protein/dataset (a name without a white space is preferred) and also reference structure id and chain identifiers:

```
In [4]: name = 'HIV-RT'   # dataset name
```

```
In [5]: ref_pdb = '1dlo'   # reference PDB file
```

```
In [6]: ref_chids = 'AB'   # reference chain identifiers
```

5.1.2 Parameters

Following parameters are for comparing two structures to determine matching chain.

```
In [7]: sequence_identity = 94
```

```
In [8]: sequence_coverage = 85
```

Chains from two different structures will be paired if they share 94% sequence identity and the aligned part of the sequences cover 85% of the longer sequence.

[1]http://en.wikipedia.org/wiki/Reverse Transcriptase

5.1.3 Structures

We are going to use the following list of structures:

```
In [9]: pdb_ids = ['3kk3', '3kk2', '3kk1', '1suq', '1dtt', '3i0r', '3i0s', '3m8p',
   ...:            '3m8q', '1jlq', '3nbp', '1klm', '2ops', '2opr', '1s9g', '2jle',
   ...:            '1s9e', '1jla', '1jlc', '1jlb', '1jle', '1jlg', '1jlf', '3drs',
   ...:            '3e01', '3drp', '1hpz', '3ith', '1s1v', '1s1u', '1s1t', '1ep4',
   ...:            '3klf', '2wom', '2won', '1s1x', '2zd1', '3kle', '1hqe', '1n5y',
   ...:            '1fko', '1hnv', '1hni', '1hqu', '1iky', '1ikx', '1t03', '1ikw',
   ...:            '1ikv', '1t05', '3qip', '3jsm', '1c0t', '1c0u', '2ze2', '1hys',
   ...:            '1rev', '3dle', '1uwb', '3dlg', '3qo9', '1tv6', '2i5j', '3meg',
   ...:            '3mee', '3med', '3mec', '3dya', '2be2', '2opp', '3di6', '1tl3',
   ...:            '1jkh', '1sv5', '1tl1', '1n6q', '2rki', '1tvr', '3klh', '3kli',
   ...:            '1dtq', '1bqn', '3klg', '1bqm', '3ig1', '2b5j', '1r0a', '3dol',
   ...:            '1fk9', '2ykm', '1rtd', '1hmv', '3dok', '1rti', '1rth', '1rtj',
   ...:            '1dlo', '1fkp', '3bgr', '1c1c', '1c1b', '3lan', '3lal', '3lam',
   ...:            '3lak', '3drr', '2rf2', '1rt1', '1j5o', '1rt3', '1rt2', '1rt5',
   ...:            '1rt4', '1rt7', '1rt6', '3lp1', '3lp0', '2iaj', '3lp2', '1qe1',
   ...:            '3dlk', '1s1w', '3isn', '3kjv', '3jyt', '2ban', '3dmj', '2vg5',
   ...:            '1vru', '1vrt', '1lw2', '1lw0', '2ic3', '3c6t', '3c6u', '3is9',
   ...:            '2ykn', '1hvu', '3irx', '2b6a', '3hvt', '1tkz', '1eet', '1tkx',
   ...:            '2vg7', '2hmi', '1lwf', '1tkt', '2vg6', '1s6p', '1s6q', '3dm2',
   ...:            '1lwc', '3ffi', '1lwe']
   ...:
```

A predefined set of structures will be used, but an up-to-date list can be obtained by blast searching PDB. See *Homologous Proteins* (page 5) and *Blast Search PDB*[2] examples.

5.1.4 Set reference

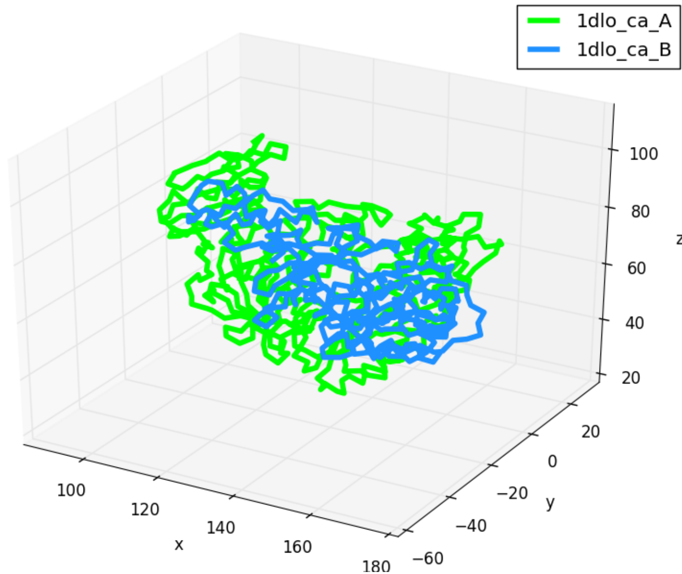
Now we set the reference chains that will be used for compared to the structures in the ensemble and will form the basis of the structural alignment.

```
# Parse reference structure
In [10]: reference_structure = parsePDB(ref_pdb, subset='calpha')

In [11]: # Get the reference chain from this structure

In [12]: reference_hierview = reference_structure.getHierView()

In [13]: reference_chains = [reference_hierview[chid] for chid in ref_chids]

In [14]: reference_chains
Out[14]:
[<Chain: A from 1dlo_ca (556 residues, 556 atoms)>,
 <Chain: B from 1dlo_ca (415 residues, 415 atoms)>]
```

Chain A is the p66 subunit, and chain B is the p51 subunit of HIV-RT. Let's take a quick look at that:

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

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

[2] http://prody.csb.pitt.edu/tutorials/structure_analysis/blastpdb.html#blastpdb



5.2 Prepare Ensemble

We handle an ensemble of heterogeneous conformations using `PDBEnsemble` objects, so let's instantiate one:

```
In [17]: ensemble = PDBEnsemble(name)
```

We now combine the reference chains and set the reference coordinates of the ensemble.

```
In [18]: reference_chain = reference_chains[0] + reference_chains[1]

In [19]: ensemble.setAtoms(reference_chain)

In [20]: ensemble.setCoords(reference_chain.getCoords())
```

Coordinates of the reference structure are set as the coordinates of the ensemble onto which other conformations will be superposed.

We can also start a log file using `startLogfile()`. Screen output will be save in this file, and can be used to check if structures are added to the ensemble as expected.

```
In [21]: startLogfile(name)
```

Let's also start a list to keep track of PDB files that are not added to the ensemble:

```
In [22]: unmapped = []
```

Now, we parse the PDB files one by one and add them to the ensemble:

```
In [23]: for pdb in pdb_ids:
   ....:     # Parse the PDB file
   ....:     structure = parsePDB(pdb, subset='calpha', model=1)
   ....:     atommaps = []
   ....:     for reference_chain in reference_chains:
   ....:         # Map current PDB file to the reference chain
   ....:         mappings = mapOntoChain(structure, reference_chain,
   ....:                                 seqid=sequence_identity,
```

```
   ....:                                  coverage=sequence_coverage)
   ....:         if len(mappings) == 0:
   ....:             print 'Failed to map', pdb
   ....:             break
   ....:         atommaps.append(mappings[0][0])
   ....:         # Make sure all chains are mapped
   ....:     if len(atommaps) != len(reference_chains):
   ....:         unmapped.append(pdb)
   ....:         continue
   ....:     atommap = atommaps[0] + atommaps[1]
   ....:     ensemble.addCoordset(atommap, weights=atommap.getFlags('mapped'))
   ....:

In [24]: ensemble
Out[24]: <PDBEnsemble: HIV-RT (155 conformations; 971 atoms)>

In [25]: ensemble.iterpose()

In [26]: saveEnsemble(ensemble)
Out[26]: 'HIV-RT.ens.npz'
```

We can now close the logfile using `closeLogfile()`:

```
In [27]: closeLogfile(name)
```

Let's check which structures, if any, are not mapped (added to the ensemble):

```
In [28]: unmapped
Out[28]: []
```

We can write the aligned conformations into a PDB file as follows:

```
In [29]: writePDB(name + '.pdb', ensemble)
Out[29]: 'HIV-RT.pdb'
```

This file can be used to visualize the aligned conformations in modeling software.

This is a heterogeneous dataset, i.e. many structures had missing residues. We want to make sure that we include residues in PCA analysis if they are resolved in more than 94% of the time.

We can find out this using `calcOccupancies()` function:

```
In [30]: calcOccupancies(ensemble, normed=True).min()
Out[30]: 0.25161290322580643
```

This shows that some residues were resolved in only 24% of the dataset. We trim the ensemble to contain residues resolved in more than 94% of the ensemble:

```
In [31]: ensemble = trimPDBEnsemble(ensemble, occupancy=0.94)
```

After trimming, another round of iterative superposition may be useful:

```
In [32]: ensemble.iterpose()

In [33]: saveEnsemble(ensemble)
Out[33]: 'HIV-RT.ens.npz'
```

5.3 Perform PCA

Once the ensemble is ready, performing `PCA` is 3 easy steps:

```
In [34]: pca = PCA(name)

In [35]: pca.buildCovariance(ensemble)

In [36]: pca.calcModes()
```

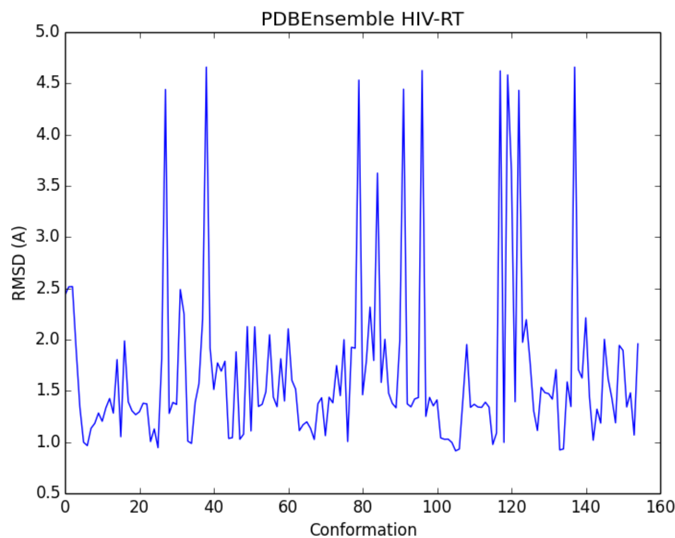
The calculated data can be saved as a compressed file using `saveModel()`

```
In [37]: saveModel(pca)
Out[37]: 'HIV-RT.pca.npz'
```

5.4 Plot results

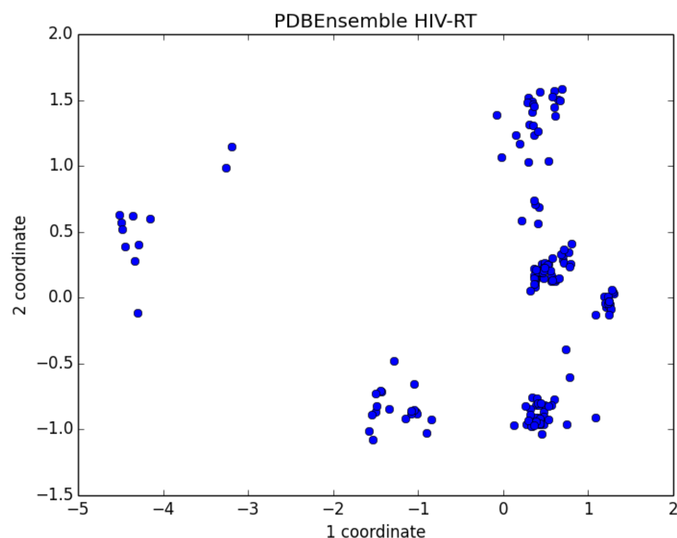
Let's plot RMSD from the average structure:

```
In [38]: plot(calcRMSD(ensemble));

In [39]: xlabel('Conformation');

In [40]: ylabel('RMSD (A)');

In [41]: title(ensemble);
```



Let's show a projection of the ensemble onto PC1 and PC2:

```
In [42]: showProjection(ensemble, pca[:2]);

In [43]: title(ensemble);
```



Only some of the ProDy plotting functions are shown here. A complete list can be found in *Dynamics Analysis*[3] module.

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* (MMBios[4]) (P41 GM103712).

[3] http://prody.csb.pitt.edu/manual/reference/dynamics/index.html#dynamics
[4] http://mmbios.org/

BIBLIOGRAPHY

[AB09] Bakan A, Bahar I. The intrinsic dynamics of enzymes plays a dominant role in determining the structural changes induced upon inhibitor binding. *Proc Natl Acad Sci U S A.* **2009** 106(34):14349-54.