ProDy

Protein Dynamics & Sequence Analysis

Ensemble Analysis

Release

Ahmet Bakan, Cihan Kaya

May 17, 2024

CHAPTER

ONE

INTRODUCTION

This tutorial shows how to analyze ensembles of experimental structures.

1.1 Required Programs

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

1.2 Recommended Programs

IPython_ and **Scipy_** 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.

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 **:wiki:‘ubiquitin‘**, and for illustration puposes, we will repeat the calculations for the ensemble of ubiquitin models that were analyzed in *[AB09]*.

A `PCA` object that stores the 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 only slightly different from that in the ProDy Tutorial[1]. Also, note that this example applies to any PDB file that contains multiple models.

2.2 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='ca')
```

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:

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

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

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

We can then make a new Atomgroup object where we replace the coordinates with the ones from the ensemble as follows:

```
In [10]: ubi_copy = ubi.copy()

In [11]: ubi_copy.delCoordset(range(ubi_copy.numCoordsets()))

In [12]: ubi_copy.addCoordset(ensemble.getCoordsets())
```

2.3 PCA calculations

Performing `PCA` is only three lines of code:

```
In [13]: pca = PCA('Ubiquitin')

In [14]: pca.buildCovariance(ensemble)

In [15]: pca.calcModes()

In [16]: repr(pca)
Out[16]: '<PCA: Ubiquitin (20 modes; 70 atoms)>'
```

Faster method

Principal modes can be calculated faster using singular value decomposition:

```
In [17]: svd = PCA('Ubiquitin')

In [18]: svd.performSVD(ensemble)
```

For heterogeneous NMR datasets, both methods yields identical results:

```
In [19]: abs(svd.getEigvals()[:20] - pca.getEigvals()).max()
Out[19]: 1.0658141036401503e-14

In [20]: abs(calcOverlap(pca, svd).diagonal()[:20]).min()
Out[20]: 0.9999999999999997
```

2.4 Write NMD file

Write principal modes into an NMD Format[2] file for NMWiz using `writeNMD()` function:

```
In [21]: writeNMD('ubi_pca.nmd', pca[:3], ubi)
Out[21]: 'ubi_pca.nmd'
```

2.5 Print data

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

```
In [22]: for mode in pca[:4]:
   ....:     print(calcFractVariance(mode).round(3))
   ....:
0.134
0.094
```

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

```
0.083
0.065
```

2.6 Compare with ANM results

We set the active coordinate set of ubi_copy to model 79, which is the one that is closest to the mean structure (note that indices start from 0 in Python so we give it 78). Then, we perform ANM calculations using `calcANM()` for the active coordset:

```
In [23]: ubi_copy.setACSIndex(78)

In [24]: anm, temp = calcANM(ubi_copy)

In [25]: 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 [26]: 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 the PDB. The protein of interest is **:wiki:‘cytochrome c‘** (cyt *c*). This dataset will contain structures sharing 44% or more sequence identity with human *cyt c*, i.e. its paralogs and/or orthologs.

A `PCA` instance that stores the 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]: ref_pdb = '1hrc'
```

In order to perform a BLAST search of the PDB, we will need the amino acid sequence of our reference protein. We could get the FASTA format from the PDB, or we could get the sequence from the PDB file itself. A more attractive method (to us) is to get the sequence using ProDy.

```
In [6]: ref_prot = parsePDB(ref_pdb)

In [7]: ref_hv = ref_prot.getHierView()['A']

In [8]: sequence = ref_hv.getSequence()
```

This is the same as simply using

```
In [9]: sequence = '''GDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKTGQAPGFTYTDANKNKGITWKEE
   ...: TLMEYLENPKKYIPGTKMIFAGIKKKTEREDLIAYLKKATNE'''
   ...:
```

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 [10]: exclude = ['3nbt', '3nbs']
```

3.2 Parameters

It is sometimes useful to set parameters in variables to use multiple times. In this case, we use `seqid` for the minimum sequence identity for including sequences at both selection of BLAST hits and ensemble building.

```
In [11]: seqid = 44
```

3.3 Blast and download

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

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

If this function times out, then you can ask the blast_record to try again using the `PDBBlastRecord.fetch()`. We can even do this in a loop to be sure:

```
In [13]: while not blast_record.isSuccess:
   ....:    blast_record.fetch()
   ....:
```

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 the Python standard library `pickle`[3] as follows:

```
In [14]: import pickle
```

The record is saved using the `dump()`[4] function:

```
In [15]: pickle.dump(blast_record, open('cytc_blast_record.pkl', 'wb'))
```

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

```
In [16]: blast_record = pickle.load(open('cytc_blast_record.pkl', 'rb'))
```

We then read information from the record to extract a list of PDB IDs and chain IDs.

```
In [17]: pdb_hits = []

In [18]: for key, item in blast_record.getHits(seqid).iteritems():
   ....:     pdb_hits.append((key, item['chain_id']))
   ....:
```

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

```
In [19]: pdbs = parsePDB([pdb for pdb, ch in pdb_hits], subset='ca', compressed=False)
```

```
In [20]: len(pdbs)
Out[20]: 120
```

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*

[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

```
In [21]: reference_structure = parsePDB(ref_pdb, subset='ca', chain='A')

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

In [23]: reference_chain = reference_hierview['A']
```

3.5 Prepare ensemble

X-ray structural ensembles are heterogenenous, 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*).

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 works[6] for more details.

The resulting `AtomMap` instances are used to prepare a `PDBEnsemble` by mapping each structure against the reference chain and adding a coordinates set corresponding to the mapped atoms. The overall procedure is shown in detail below so you can understand the process and think about case specific changes such as those in the Multimeric Structures tutorial[7]. This process can also be automated using `buildPDBEnsemble()` as shown in the Heterogeneous X-ray Structures tutorial[8].

```
In [24]: startLogfile('pca_blast')

In [25]: ensemble = PDBEnsemble(name)

In [26]: ensemble.setAtoms(reference_chain)

In [27]: ensemble.setCoords(reference_chain.getCoords())
```

```
In [28]: for structure in pdbs:
   ....:     if structure.getTitle()[:4] in exclude:
   ....:         continue
   ....:     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', structure.getTitle()[:4])
   ....:         continue
   ....:     atommap = mappings[0][0]
   ....:     ensemble.addCoordset(atommap, weights=atommap.getFlags('mapped'))
   ....:

In [29]: ensemble.iterpose()

In [30]: saveEnsemble(ensemble)
Out[30]: 'cyt_c.ens.npz'
```

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

```
In [31]: len(ensemble)
Out[31]: 407
```

Note that the 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

[6] http://prody.csb.pitt.edu/manual/reference/atomic/atommap.html#atommaps
[7] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/dimer.html
[8] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_calculations.html

ensemble as individual conformations.

Write aligned conformations into a PDB file as follows:

```
In [32]: writePDB(name+'.pdb', ensemble)
Out[32]: 'cyt_c.pdb'
```

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

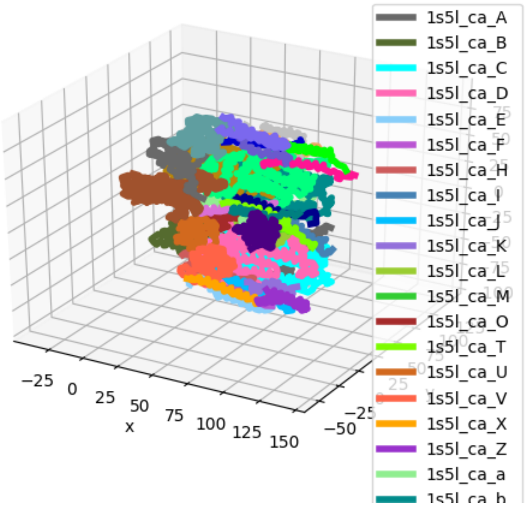
3.6 Align PDB files

`alignByEnsemble()` function can be used to align PDB structures used in the analysis from which you can write new PDB files. The resulting files will contain intact structures and can be used for visualization purposes. In this case, we will align only select PDB files:

```
In [33]: conf1_aligned, conf2_aligned = alignByEnsemble(pdbs[:2], ensemble[:2])
```

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

```
In [34]: showProtein(conf1_aligned, conf2_aligned);

In [35]: legend();
```



3.7 Perform PCA

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

```
In [36]: pca = PCA(name)

In [37]: pca.buildCovariance(ensemble)

In [38]: pca.calcModes()
```

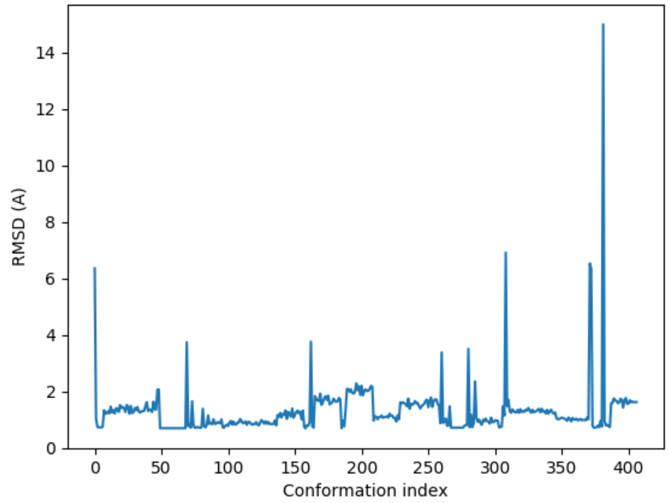
The calculated data can be saved as a compressed file using `saveModel()` function:

```
In [39]: saveModel(pca)
Out[39]: 'cyt_c.pca.npz'
```

3.8 Plot results

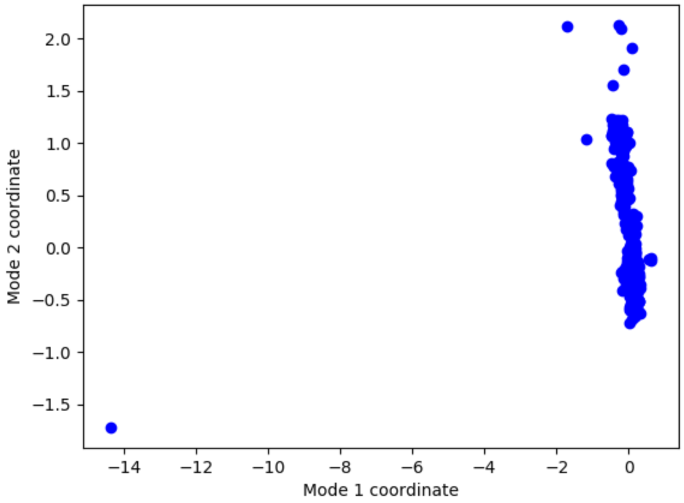
Let's plot RMSDs of all conformations from the average conformation:

```
In [40]: rmsd = calcRMSD(ensemble)

In [41]: plot(rmsd);

In [42]: xlabel('Conformation index');

In [43]: ylabel('RMSD (A)');
```



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

```
In [44]: showProjection(ensemble, pca[: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]*.

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 = ['5UOJ', '1A9U', '1BL6', '1BL7', '1BMK', '1DI9', '1IAN', '1KV1',
   ...:           '1KV2', '1LEW', '1LEZ', '1M7Q', '1OUK', '1OUY', '1OVE', '1OZ1',
   ...:           '5UOJ', '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]*. Some structures have been refined and their identifiers 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(90, 70)
```

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

```
In [8]: structures = parsePDB(pdbids, subset='ca', compressed=False)
```

The `structures` variable contains a list of `AtomGroup` instances.

4.1.2 Prepare ensemble

X-ray structural ensembles are heterogenenous, 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*). However, ProDy has a function `buildPDBEnsemble()` that makes this process a lot easier. It depends on mapping each structure against the reference structure using a function such as `mapOntoChain()` demonstrated in the BLAST example. The reference structure is automatically the first member of list provided, which in this case is 5uoj.

```
In [9]: ensemble = buildPDBEnsemble(structures, title='p38 X-ray')

In [10]: ensemble
Out[10]: <PDBEnsemble: p38 X-ray (78 conformations; 343 atoms)>
```

Perform an iterative superimposition:

```
In [11]: ensemble.iterpose()
```

4.1.3 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 [12]: writePDB('p38_xray_ensemble.pdb', ensemble)
Out[12]: 'p38_xray_ensemble.pdb'
```

4.1.4 PCA calculations

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

```
In [13]: pca = PCA('p38 xray')           # Instantiate a PCA instance

In [14]: pca.buildCovariance(ensemble)   # Build covariance for the ensemble

In [15]: pca.calcModes()                 # Calculate modes (20 of the by default)
```

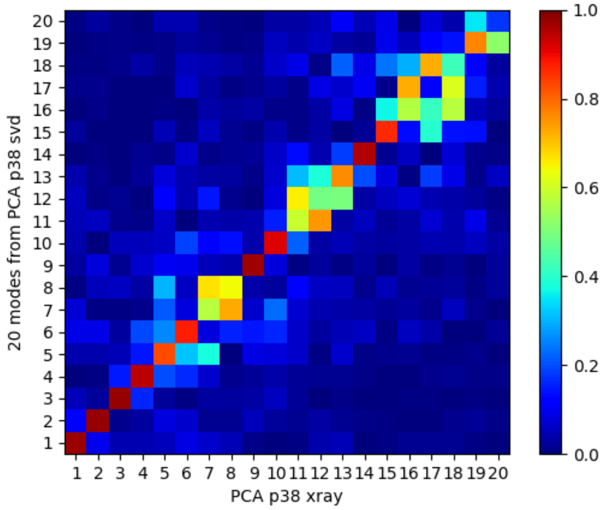
Approximate method

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

```
In [16]: pca_svd = PCA('p38 svd')

In [17]: pca_svd.performSVD(ensemble)
```

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

```
In [18]: abs(pca_svd.getEigvals()[:20] - pca.getEigvals()).max()
Out[18]: 28.681810668526865

In [19]: abs(calcOverlap(pca, pca_svd).diagonal()[:20]).min()
Out[19]: 0.12871820775089923

In [20]: showOverlapTable(pca, pca_svd[:20])
Out[20]:
(<matplotlib.image.AxesImage at 0x7ff482506d10>,
 [],
 <matplotlib.colorbar.Colorbar at 0x7ff483f3aa10>)
```



If we remove the most variable loop from the analysis, then these results become much more similar:

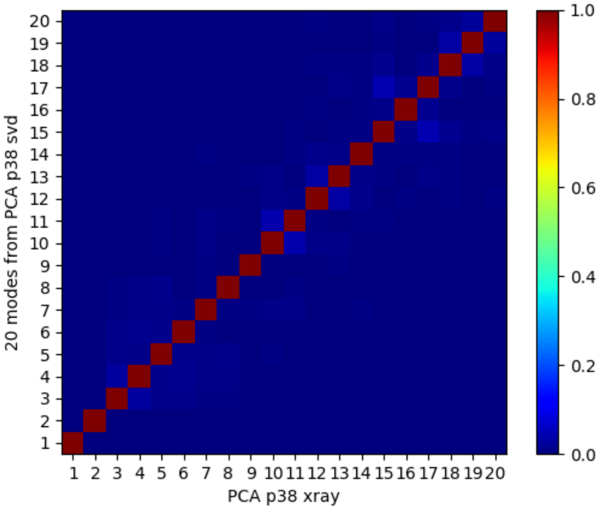
```
In [21]: ref_selection = structures[0].select('resnum 5 to 31 36 to 114 122 to 169 185 to 351')

In [22]: ensemble.setAtoms(ref_selection)
```

```
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)
```

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

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

```
In [28]: abs(pca_svd.getEigvals()[:20] - pca.getEigvals()).max()
Out[28]: 0.4086249606714958
```

```
In [29]: abs(calcOverlap(pca, pca_svd).diagonal()[:20]).min()
Out[29]: 0.9984941988041348

In [30]: showOverlapTable(pca, pca_svd[:20]);
```



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

4.1.5 ANM calculations

We also calculate ANM modes for p38 in order to make comparisons later:

```
In [31]: anm = ANM('5uoj')                    # Instantiate a ANM instance

In [32]: anm.buildHessian(ref_selection)      # Build Hessian for the reference chain

In [33]: anm.calcModes()                      # Calculate slowest non-trivial 20 modes
```

4.1.6 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 [34]: saveModel(pca)
Out[34]: 'p38_xray.pca.npz'

In [35]: saveModel(anm)
Out[35]: '5uoj.anm.npz'

In [36]: saveEnsemble(ensemble)
Out[36]: 'p38_X-ray.ens.npz'

In [37]: writePDB('p38_ref_selection.pdb', ref_selection)
Out[37]: 'p38_ref_selection.pdb'
```

We use the `saveModel()` and `saveEnsemble()` functions to save calculated data. In *Analysis*, 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*. 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]*.

We start by importing everything from the ProDy package and loading data from the previous part. These steps can be skipped if you are continuing in the same iPython session.

```
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('5uoj.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 3 PCs as follows:

```
In [6]: for mode in pca[:3]:
   ...:     var = calcFractVariance(mode)*100
   ...:     print('{0:s}  % variance = {1:.2f}'.format(mode, var))
   ...:
Mode 1 from PCA p38 xray  % variance = 28.65
Mode 2 from PCA p38 xray  % variance = 16.81
Mode 3 from PCA p38 xray  % variance = 10.32
```

These data were included in Table 1 in *[AB09]*.

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 = 0.52
Mode 3 from PCA p38 xray  collectivity = 0.31
```

We can also calculate the collectivity of ANM modes:

```
In [8]: 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 = 0.52
Mode 3 from PCA p38 xray  collectivity = 0.31
```

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

4.2.4 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 [9]: writeArray('p38_PCA_eigvecs.txt', pca.getEigvecs() ) # PCA eigenvectors
Out[9]: 'p38_PCA_eigvecs.txt'

In [10]: writeModes('p38_ANM_modes.txt', anm) # This function is based on writeArray
Out[10]: 'p38_ANM_modes.txt'
```

4.3 Plotting

4.3.1 Synopsis

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

4.3.2 Load data

First, we load data saved in *Calculations*:

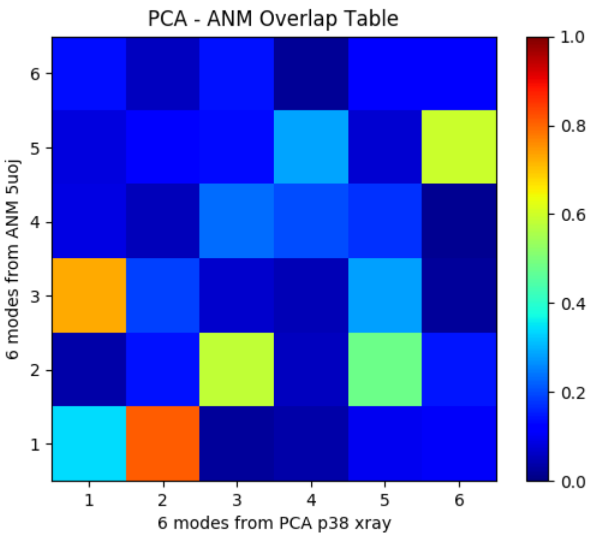
```
In [1]: from prody import *

In [2]: from pylab import *

In [3]: ion()
```

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

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

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

In [7]: ref_chain = parsePDB('p38_ref_selection.pdb')
```

4.3.3 PCA - ANM overlap

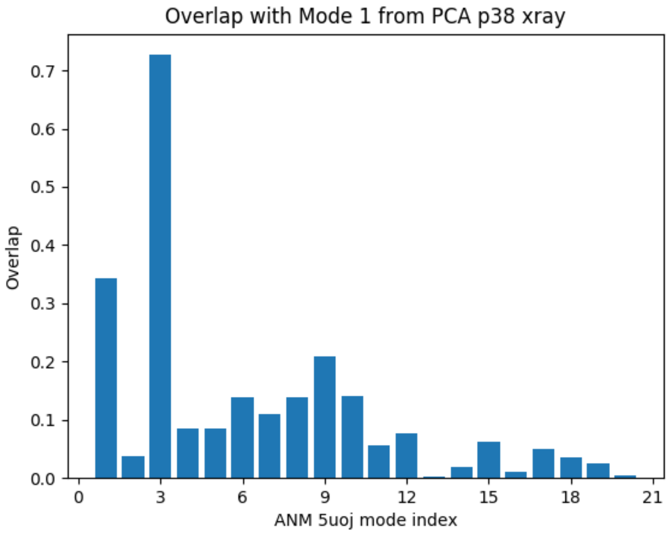
In previous page, we compared PCA and ANM modes to get some numbers. In this case, we will use plotting functions to make similar comparisons:

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

# Let's change the title of the figure
In [9]: title('PCA - ANM Overlap Table');
```



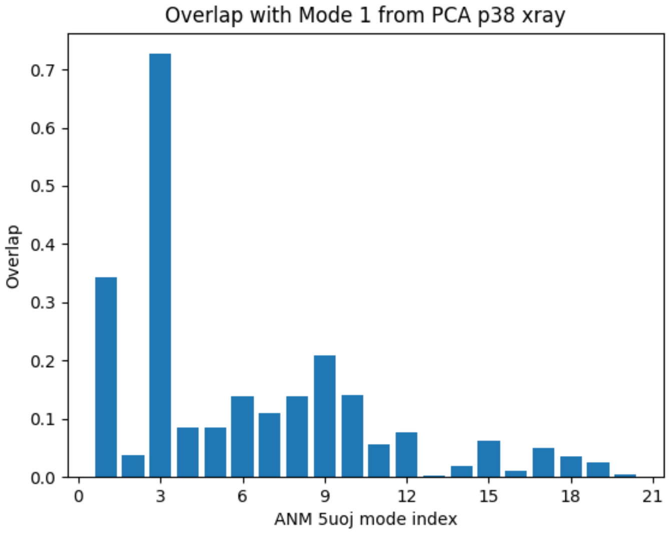
It is also possible to plot overlap of a single mode from one model with multiple modes from another:

```
In [10]: showOverlap(pca[0], anm);
```



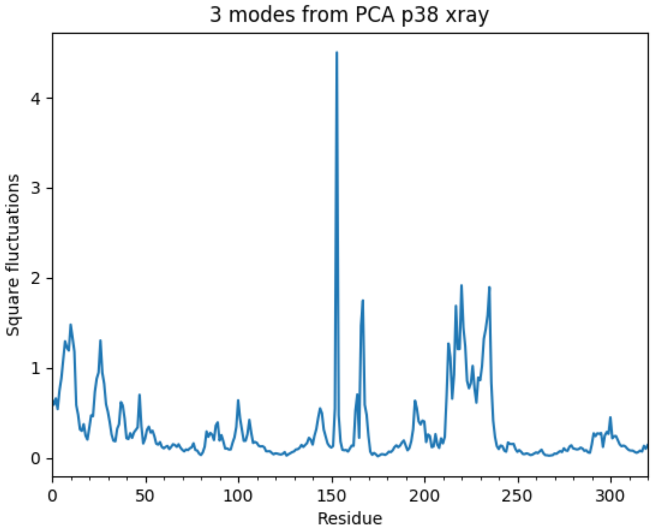
Let’s also plot the cumulative overlap in the same figure:

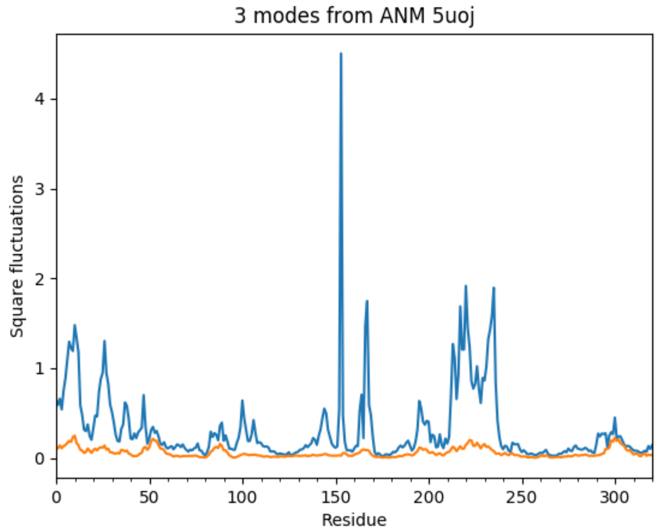
```
In [11]: showOverlap(pca[0], anm);

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



4.3.4 Square fluctuations

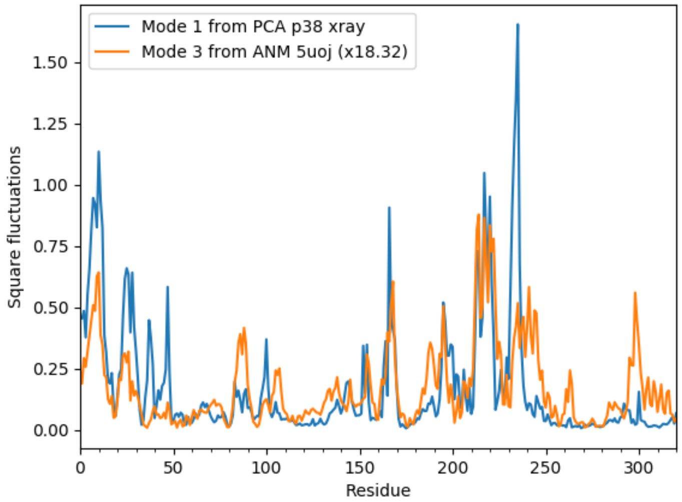
```
In [13]: showSqFlucts(pca[:3]);

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



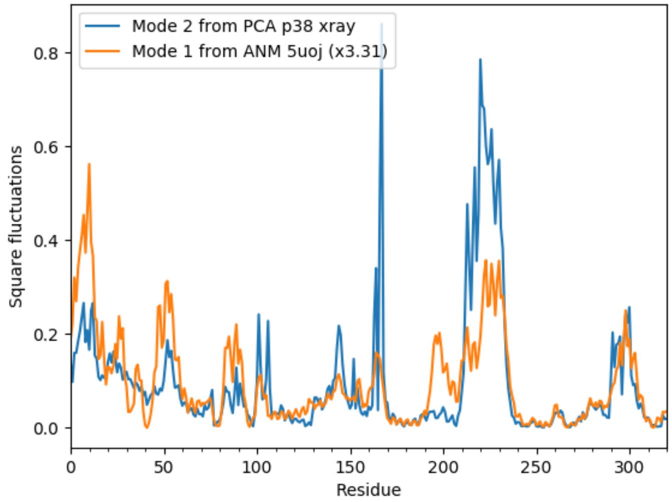


Now let's plot square fluctuations along PCA and ANM modes in the same plot:

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

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

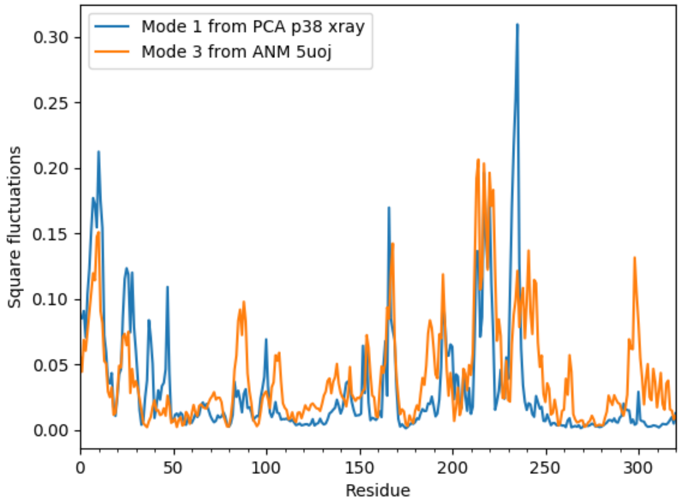


```
In [17]: showScaledSqFlucts(pca[1], anm[0]);

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



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

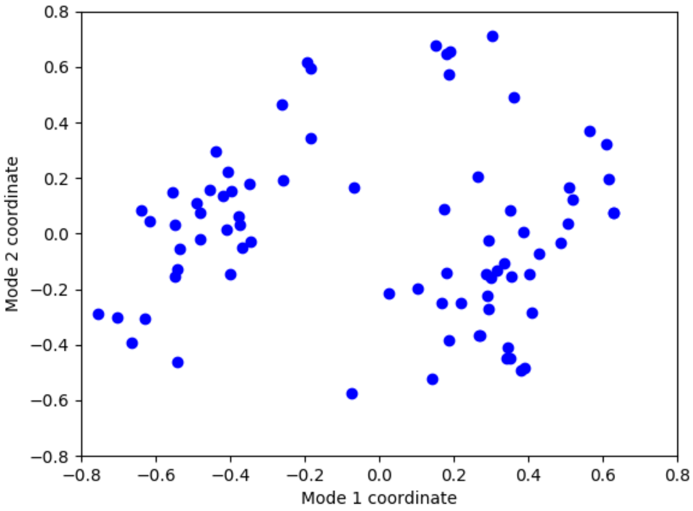
```
In [19]: showNormedSqFlucts(pca[0], anm[2]);

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



4.3.5 Projections

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

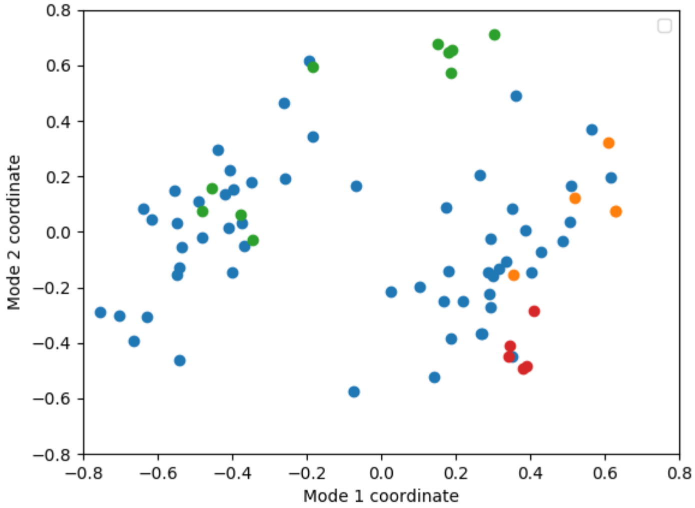
```
In [21]: showProjection(ensemble, pca[:2]);

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



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

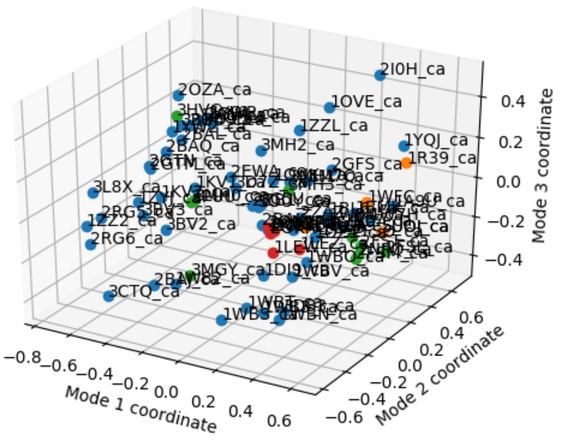
red	unbound
blue	inhibitor bound
yellow	glucoside bound
purple	peptide/protein bound

```
In [23]: color_list = ['red', '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', '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);
   ....:
```

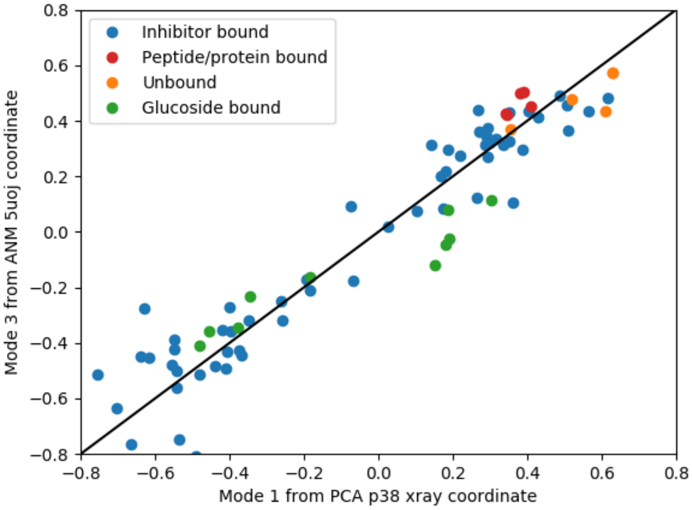


The figure with all conformation labels is crowded, but in an interactive session you can zoom in and out to make text readable.

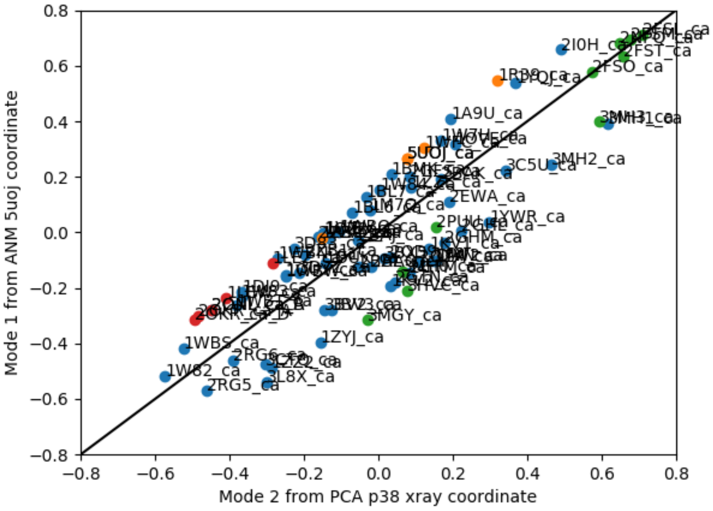
4.3.6 Cross-projections

Finally, we will make a cross-projection plot comparing PCA modes and ANM modes using `showCrossProjection()`. We will pass the `scale='y'` argument, which will scale the width of the projection along the 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]);
```



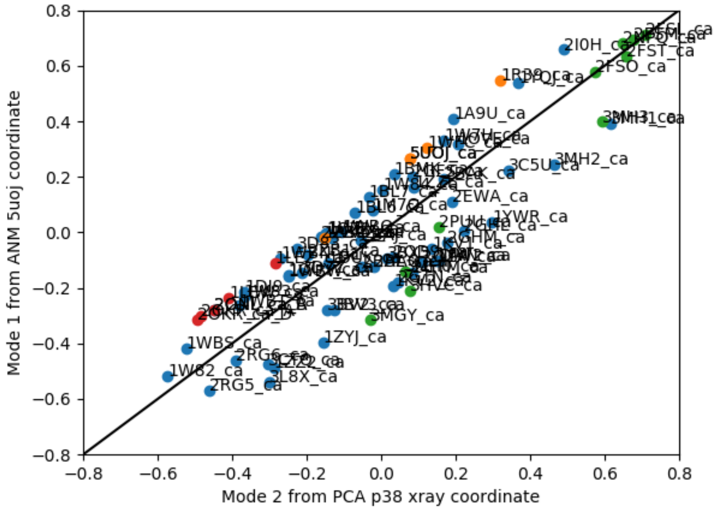
It is also possible to find the correlation between these projections:

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

In [38]: print(np.corrcoef(pca_coords, anm_coords))
[[ 1.          -0.95605067]
 [-0.95605067  1.        ]]
```

This shows 0.95 for the 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*. 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]*.

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_** plugin designed to complement ProDy.

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*:

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

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

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

In [7]: ref_selection = parsePDB('p38_ref_selection.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_selection)
Out[8]: 'p38_pca.nmd'

In [9]: writeNMD('p38_anm.nmd', anm[:3], ref_selection)
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]: '/home/exx/miniconda3/envs/py27/bin/vmd'
```

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

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

CHAPTER

FIVE

MULTIMERIC STRUCTURES

In this part, first we will build an ensemble of HIV **:wiki:'Reverse Transcriptase'** (RT), which is a heterodimer. The ensemble can be conveniently built by `buildPDBEnsemble()` automatically, but we will build it step by step first for a better understanding of the process. Then we will show how to use `buildPDBEnsemble()` to build ensemble in one line. Then we will perform PCA on the ensemble.

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

The following parameters are for comparing two structures to determine matching chains.

```
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.

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* and Blast Search PDB[9] examples.

5.1.4 Set reference

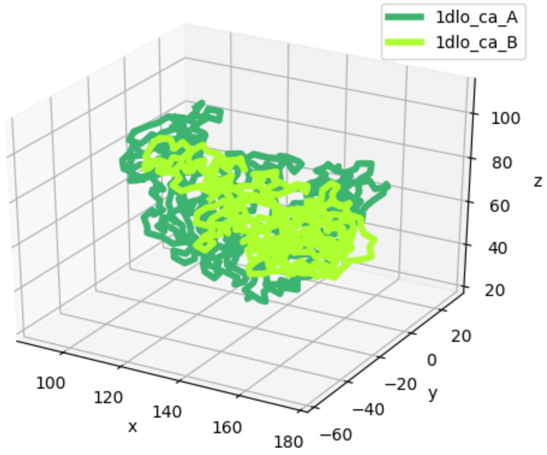
Now we set the reference chains that will be used for compared to the structures in the ensemble and will form the basis of the structural alignment.

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

# Get the reference chain from this structure
In [11]: reference_hierview = reference_structure.getHierView()

In [12]: reference_chains = [reference_hierview[chid] for chid in ref_chids]

In [13]: reference_chains
Out[13]:
[<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 [14]: showProtein(reference_structure);

In [15]: legend();
```

[9] 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 [16]: ensemble = PDBEnsemble(name)
```

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

```
In [17]: reference_chain = reference_chains[0] + reference_chains[1]

In [18]: ensemble.setAtoms(reference_chain)

In [19]: 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 saved in this file, and can be used to check if structures are added to the ensemble as expected.

```
In [20]: startLogfile(name)
```

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

```
In [21]: unmapped = []
```

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

```
In [22]: for pdb in pdb_ids:
   ....:     structure = parsePDB(pdb, subset='calpha', model=1)
   ....:     atommaps = []
   ....:     for reference_chain in reference_chains:
   ....:         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])
   ....:     if len(atommaps) != len(reference_chains):
   ....:         unmapped.append(pdb)
   ....:         continue
   ....:     atommap = atommaps[0] + atommaps[1]
   ....:     ensemble.addCoordset(atommap, weights=atommap.getFlags('mapped'))
   ....:

In [23]: ensemble
Out[23]: <PDBEnsemble: HIV-RT (155 conformations; 971 atoms)>

In [24]: ensemble.iterpose()

In [25]: saveEnsemble(ensemble)
Out[25]: 'HIV-RT.ens.npz'
```

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

```
In [26]: closeLogfile(name)
```

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

```
In [27]: unmapped
Out[27]: []
```

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

```
In [28]: writePDB(name + '.pdb', ensemble)
Out[28]: 'HIV-RT.pdb'
```

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

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

We can check this using the `calcOccupancies()` function:

```
In [29]: calcOccupancies(ensemble, normed=True).min()
Out[29]: 0.3096774193548387
```

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

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

```
In [31]: ensemble.iterpose()

In [32]: ensemble
Out[32]: <PDBEnsemble: HIV-RT (155 conformations; selected 908 of 971 atoms)>

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

5.3 Use buildPDBEnsemble Function

As mentioned at the beginning, the ensemble can be also built by `buildPDBEnsemble()` in several lines of code:

```
In [34]: unmapped = []

In [35]: prot = parsePDB('1dlo', subset='ca', model=1)

In [36]: pdbs = parsePDB(pdb_ids, subset='ca', model=1)

In [37]: ensemble = buildPDBEnsemble(pdbs, ref=prot, title='HIV-RT', labels=pdb_ids,
   ....:                              seqid=94, coverage=85, occupancy=0.94, unmapped=unmapped)
   ....:

In [38]: ensemble
Out[38]: <PDBEnsemble: HIV-RT (162 conformations; selected 893 of 971 atoms)>
```

5.4 Perform PCA

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

```
In [39]: pca = PCA(name)

In [40]: pca.buildCovariance(ensemble)

In [41]: pca.calcModes()
```

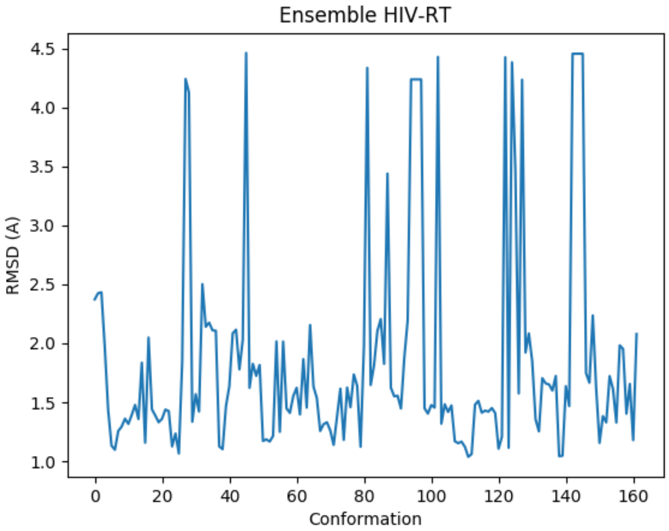
The calculated data can be saved as a compressed file using `saveModel()`

```
In [42]: saveModel(pca)
Out[42]: 'HIV-RT.pca.npz'
```

5.5 Plot results

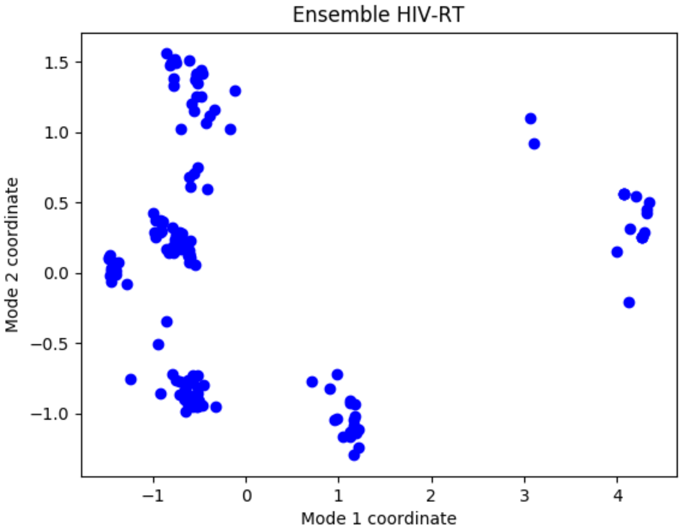
Let's plot RMSD from the average structure:

```
In [43]: plot(calcRMSD(ensemble));

In [44]: xlabel('Conformation');

In [45]: ylabel('RMSD (A)');

In [46]: title(ensemble);
```



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

```
In [47]: showProjection(ensemble, pca[:2]);

In [48]: title(ensemble);
```



Only some of the ProDy plotting functions are shown here. A complete list can be found in Dynamics Analysis[10] module.

Acknowledgments

Continued development of Protein Dynamics Software *ProDy* and associated programs is partially supported by the NIH[11]-funded Biomedical Technology and Research Center (BTRC) on *High Performance Computing for Multiscale Modeling of Biological Systems* (MMBios[12]) (P41 GM103712).

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

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

[12] http://mmbios.org/

[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.