Signature Dynamics

Release

She Zhang, James Krieger

June 07, 2024

CHAPTER ONE

INTRODUCTION

This tutorial describes how to calculate signature dynamics for a family of proteins with similar structures using Elastic Network Models. This method creates an ensemble of aligned structures and calculates statistics such as means and standard deviations on various dynamic properties including mode profiles, mean square fluctuations and cross-correlation matrices. It also includes tools for classifying family members based on their sequence, structure and dynamics.

The theory and usage of this toolkit will be described in *[SZ18]*.

1.1 Required Programs

The latest version of **ProDy_** is recommended along with **NumPy_** and **Matplotlib_**. **IPython_** is highly recommended for interactive usage.

1.2 Getting Started

This tutorial contains three parts. In the first part, we will have a quick walk-through on the SignDy[1] calculations and functions using the example of type-I periplasmic binding protein (PBP-I) domains, in which case the data is convienient collected from the Dali server[2] *[LH10]*. The second part will be a more detailed periplasmic binding protein (PBP-I) domains, in which case the data is convieniently collected from the Dali server[3] *[LH10]*. The second part will be review how to use the CATH database[4] *[IS21]* to build the ensemble. The third part will be a more detailed tutorial on building an ensemble 'manually' from scratch, and try to reproduce the figures presented in *[SZ18]*.

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 the **ProDy_**, **NumPy_** and **Matplotlib_** packages.

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

In [2]: from numpy import *

In [3]: from matplotlib.pyplot import *

In [4]: ion()
```

[1]http://prody.csb.pitt.edu/test_prody/tutorials/signdy_tutorial/
[2]http://ekhidna2.biocenter.helsinki.fi/dali/
[3]http://ekhidna2.biocenter.helsinki.fi/dali/
[4]https://www.cathdb.info/

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.

1.3 How to Cite

If you benefited from SignDy Calculations in your research, please cite the following paper:

CHAPTER

TWO

OVERVIEW

The first step in signature dynamics analysis is to collect a set of related protein structures and build a `PDBEnsemble`. This can be achieved by multiple routes: a query search of the PDB using `blastPDB()` or `searchDali()`, extraction of PDB IDs from the Pfam or CATH database, or input of a pre-defined list.

We demonstrate the Dali[5] method here in the first part of the tutorial. The usage of Pfam[6] and CATH[7] methods are described in the database tutorial (under construction) and the function `blastPDB()` is described in the Structure Analysis Tutorial[8].

We apply these methods to the PBP-I domains, a group of protein structures originally found in bacteria for transport of solutes across the periplasmic space and later seen in various eukaryotic receptors including ionotropic and metabotropic glutamate receptors. We use the N-terminal domain of AMPA receptor subunit GluA2 (gene name GRIA2[9]) as a query.

First, make necessary imports from ProDy and Matplotlib packages if you haven't already.

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

In [2]: from pylab import *

In [3]: ion()
```

2.1 Prepare ensemble (using Dali)

First we use the function `searchDali()` to search the PDB, which returns a `DaliRecord` object that contains a list of PDB IDs and their corresponding mappings to the reference structure.

```
In [4]: dali_rec = searchDali('3H5V','A')

In [5]: dali_rec
Out[5]: <prody.database.dali.DaliRecord at 0x7f1ddcfa7c50>
```

If DALI times out then, you can use the following code to fetch the data afterwards.

```
In [6]: while not dali_rec.isSuccess:
   ...:     dali_rec.fetch()
   ...:
```

Next, we get the lists of PDB IDs and mappings from **dali_rec**, parse the **pdb_ids** to get a list of `AtomGroup` instances:

[5] http://ekhidna2.biocenter.helsinki.fi/dali/
[6] https://pfam.xfam.org/
[7] http://www.cathdb.info/
[8] http://prody.csb.pitt.edu/tutorials/structure_analysis/blastpdb.html
[9] https://www.uniprot.org/uniprot/P42262

```
In [7]: pdb_ids = dali_rec.filter(cutoff_len=0.7, cutoff_rmsd=1.0, cutoff_Z=10)

In [8]: mappings = dali_rec.getMappings()
```

```
In [9]: ags = parsePDB(pdb_ids, subset='ca')

In [10]: len(ags)
Out[10]: 963
```

Then we provide **ags** together with **mappings** to `buildPDBEnsemble()`. We set the keyword argument `seqid=20` to account for the low sequence identity between some of the structures.

```
In [11]: dali_ens = buildPDBEnsemble(ags, mapping=mappings, seqid=20)

In [12]: dali_ens
Out[12]: <PDBEnsemble: Unknown (50 conformations; 773 atoms)>
```

Finally we save the ensemble for later processing:

```
In [13]: saveEnsemble(dali_ens, 'PBP-I')
Out[13]: 'PBP-I.ens.npz'
```

2.2 Mode ensemble

For this analysis we'll build a `ModeEnsemble` by calculating normal modes for each member of the `PDBEnsemble`. You can load a PDB ensemble at this stage if you already have one. We demonstrate this for the one we just saved.

```
In [14]: dali_ens = loadEnsemble('PBP-I.ens.npz')
```

Then we calculated `GNM` modes for each member of the ensemble. There are options to select the **model** (`GNM` by default) and the way of considering non-aligned residues by setting the **trim** option (default is `reduceModel()`, which treats them as environment).

```
In [15]: gnms = calcEnsembleENMs(dali_ens, model='GNM', trim='reduce')

In [16]: gnms
Out[16]: <ModeEnsemble: 50 modesets (20 modes, 773 atoms)>
```

We can also save the mode ensemble as follows:

```
In [17]: saveModeEnsemble(gnms, 'PBP-I')
Out[17]: 'PBP-I.modeens.npz'
```

We can load in a mode ensemble at this point as follows:

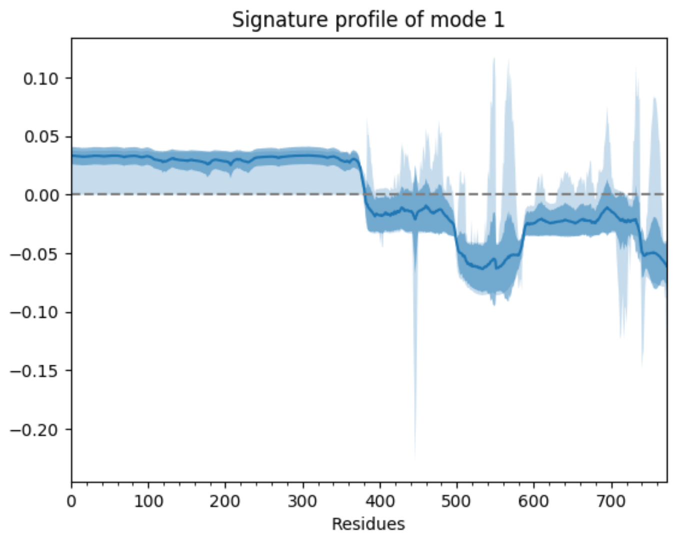
```
In [18]: gnms = loadModeEnsemble('PBP-I.modeens.npz')
```

2.3 Signature dynamics

Signatures are calculated as the mean and standard deviation of various properties such as mode shapes and mean square fluctations.

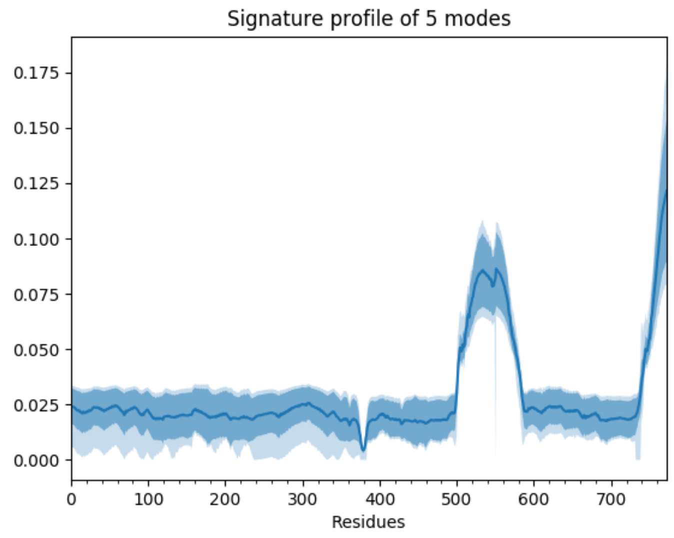
For example, we can show the average and standard deviation of the shape of the first mode (second index 0). The first index of the mode ensemble is over conformations.

```
In [19]: showSignatureMode(gnms[:, 0]);
```

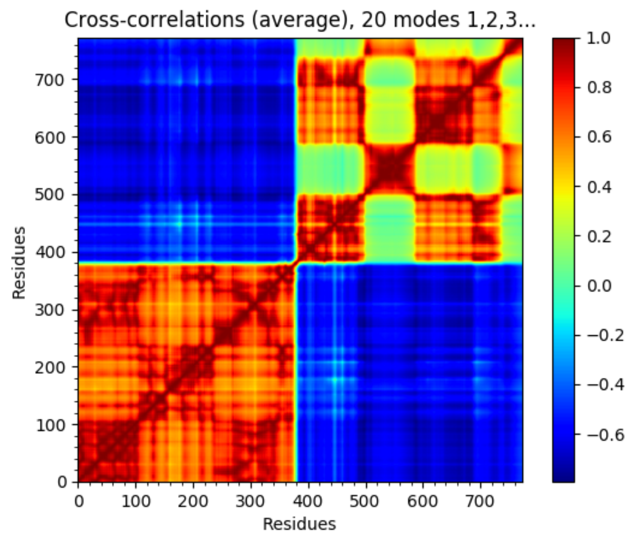


We can also show such results for properties involving multiple modes such as the mean square fluctuations from the first 5 modes or the cross-correlations from the first 20.

```
In [20]: showSignatureSqFlucts(gnms[:, :5]);
```



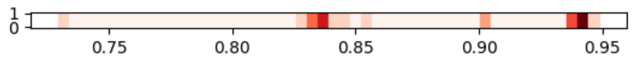
```
In [21]: showSignatureCrossCorr(gnms[:, :20]);
```

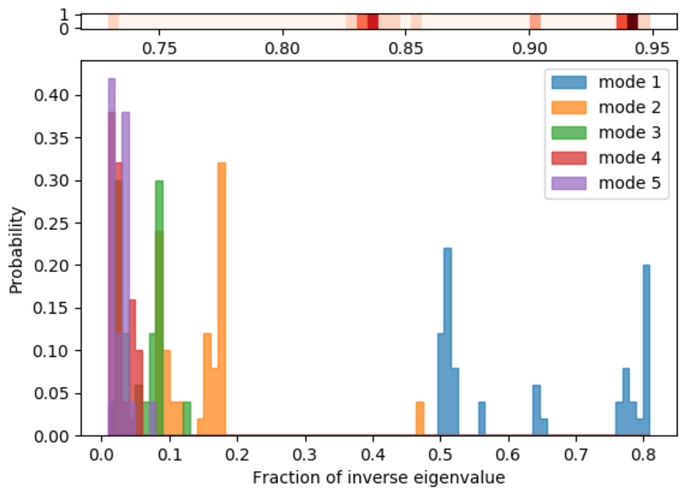


We can also look at distributions over values across different members of the ensemble such as inverse eigenvalue. We can show a bar above this with individual members labelled like *[JK15]*. We can select particular members to highlight with arrows by putting their names and labels in a dictionary.

We plot the variance bar for the first five modes (showing a function of the inverse eigenvalues related to the resultant relative size of motion) above the inverse eigenvalue distributions for each of those modes. To arrange the plots like this, we use the *:class:~matplotlib.gridspec.GridSpec* method of Matplotlib.

```
In [22]: highlights = {'3h5vA_ca': 'GluA2', '3o21C_ca': 'GluA3',
   ....:               '3h6gA_ca': 'GluK2'}
   ....:

In [23]: from matplotlib.gridspec import GridSpec

In [24]: gs = GridSpec(ncols=1, nrows=2, height_ratios=[1, 10], hspace=0.15)

In [25]: subplot(gs[0]);

In [26]: showVarianceBar(gnms[:, :5], fraction=True, highlights=highlights);

In [27]: xlabel('');

In [28]: subplot(gs[1]);

In [29]: showSignatureVariances(gnms[:, :5], fraction=True, bins=80, alpha=0.7);

In [30]: xlabel('Fraction of inverse eigenvalue');
```



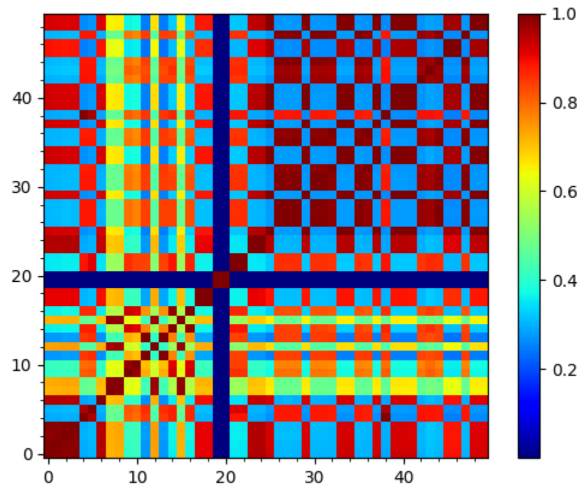


2.4 Spectral overlap and distance

Spectral overlap, also known as covariance overlap as defined in [BH02], measures the overlap between two covariance matrices, or the overlap of a subset of the modes (a mode spectrum).

```
In [31]: so_matrix = calcEnsembleSpectralOverlaps(gnms[:, :1])
```

```
In [32]: showMatrix(so_matrix);
```

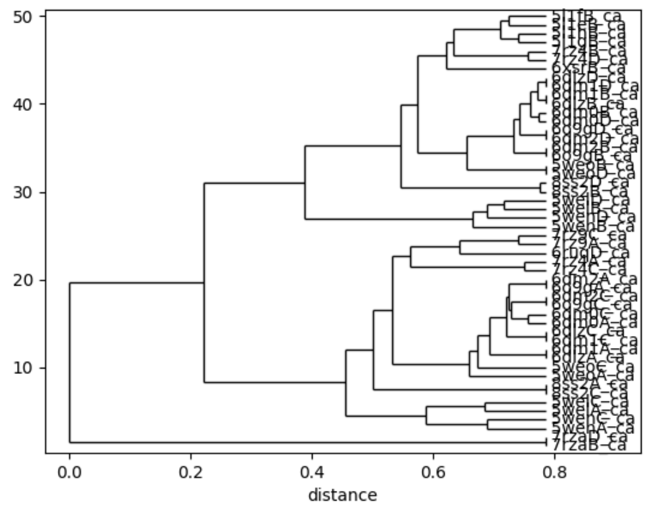


We can then calculate a tree from its arccosine, which converts the overlaps to distances:

```
In [33]: labels = gnms.getLabels()

In [34]: so_tree = calcTree(names=labels,
   ....:                    distance_matrix=arccos(so_matrix),
   ....:                    method='upgma')
   ....:
```

This tree can be displaced using the *:func:.showTree* function. The default format is ASCII text but we can change it to *plt* to get a figure:

```
In [35]: showTree(so_tree, 'plt');
```



We can reorder the spectral overlap matrix using the tree as follows:

```
In [36]: reordered_so, new_so_indices = reorderMatrix(names=labels,
   ....:                                              matrix=so_matrix,
   ....:                                               tree=so_tree)
   ....:
```

Both `PDBEnsemble` and `ModeEnsemble` objects can be reordered based on the new indices:

```
In [37]: reordered_ens = dali_ens[new_so_indices]

In [38]: reordered_gnms = gnms[new_so_indices, :]
```

2.5 Comparing with sequence and structural distances

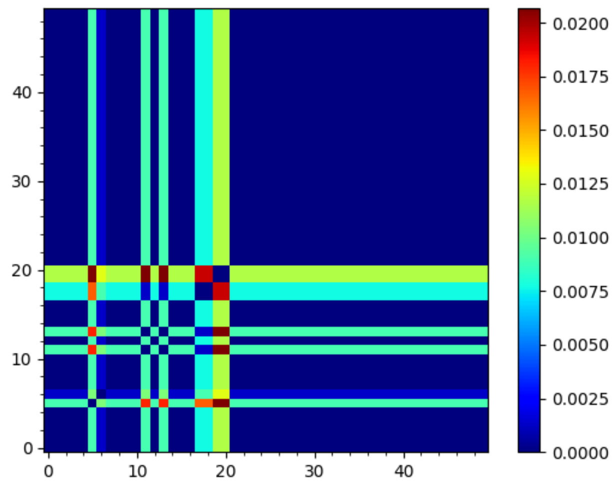
The sequence distance is given by the (normalized) Hamming distance, which is calculated by subtracting the percentage identity (fraction) from 1, and the structural distance is the RMSD. We can also calculate and show the matrices and trees for these from the PDB ensemble.

First we calculate the sequence distance matrix:

```
In [39]: seqid_matrix = buildSeqidMatrix(dali_ens.getMSA())

In [40]: seqdist_matrix = 1. - seqid_matrix
```

We can visualize the matrix using `showMatrix()`:

```
In [41]: showMatrix(seqdist_matrix);
```

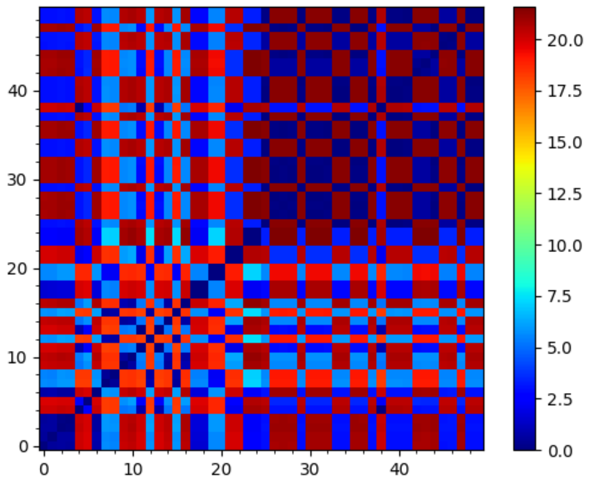


We can also construct a tree based on this distance matrix:

```
In [42]: seqdist_tree = calcTree(names=labels,
   ....:                         distance_matrix=seqdist_matrix,
   ....:                         method='upgma')
   ....:
```

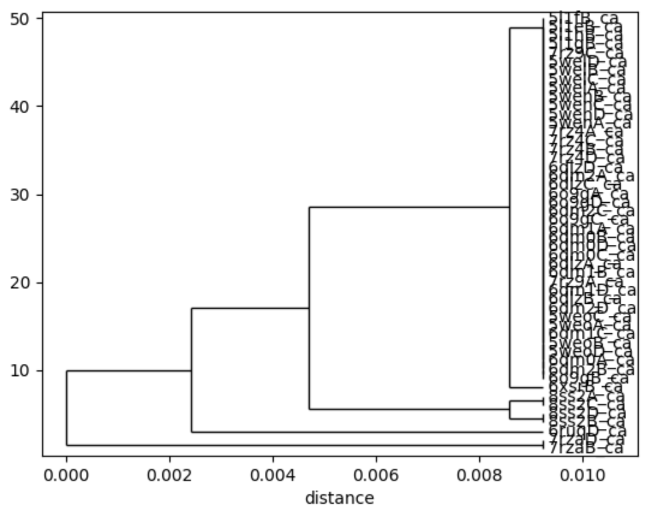
Similarily, once we obtain the RMSD matrix using `PDBEnsemble.getRMSDs()`, we can calculate the structure-based tree:

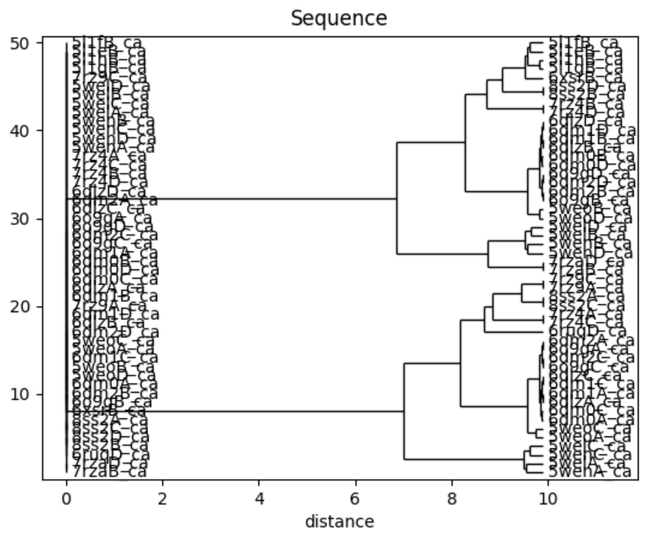
```
In [43]: rmsd_matrix = dali_ens.getRMSDs(pairwise=True)

In [44]: showMatrix(rmsd_matrix);
In [45]: rmsd_tree = calcTree(names=labels,
   ....:                      distance_matrix=rmsd_matrix,
   ....:                      method='upgma')
   ....:
```

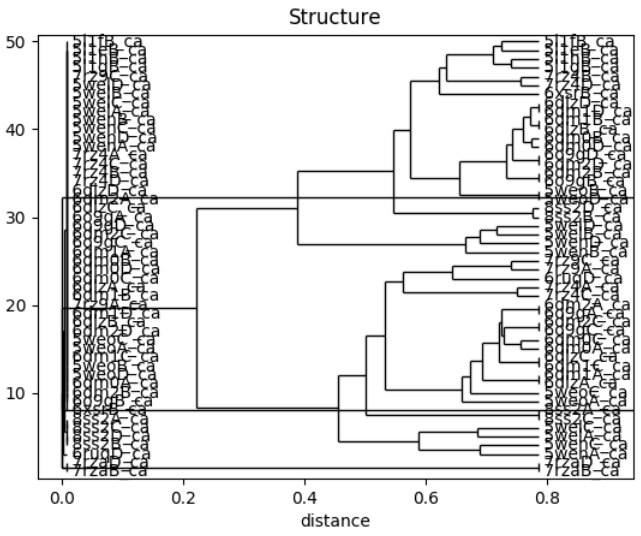


We could plot the three trees one by one. Or, it could be of interest to put all three trees constructed based on different distance metrics side by side and compare them:

```
In [46]: showTree(seqdist_tree, format='plt')

In [47]: title('Sequence')
Out[47]: Text(0.5,1,'Sequence')

In [48]: showTree(rmsd_tree, format='plt')

In [49]: title('Structure')
Out[49]: Text(0.5,1,'Structure')

In [50]: showTree(so_tree, format='plt')

In [51]: title('Dynamics')
Out[51]: Text(0.5,1,'Dynamics')
```


distance
0.000
0.002
0.004
0.006
0.008
0.010
10
20
30
40
50


Sequence
distance
0
2
4
6
8
10
20
30
40
50



This analysis is quite sensitive to how many modes are used. As the number of modes approaches the full number, the dynamic distance order approaches the RMSD order. With smaller numbers, we see finer distinctions. This is particularly clear in the current case where we used just one mode.

CHAPTER

THREE

DATA COLLECTION WITH CATH

3.1 Navigating the CATH tree

The first step in signature dynamics analysis is to collect a set of related protein structures and build a `PDBEnsemble`. This can be achieved by multiple routes: a query search of the PDB using `blastPDB()` or `searchDali()`, extraction of PDB IDs from the Pfam or CATH database, or input of a pre-defined list.

Here, we demonstrate the usage of CATH for ensemble building.

First, make necessary imports from **ProDy_**, **NumPy_** and **Matplotlib_** packages if you haven't already.

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

In [2]: from pylab import *

In [3]: ion()
```

First, we initialise a `CATHDB` object. By default, this is done by downloading data from the CATH website.

```
In [4]: cath = CATHDB()

In [5]: cath
Out[5]: <CATHDB: 5 members>
```

We can also use this object to save this data to an `.xml` file and load it later:

```
In [6]: cath.save('cath.xml')
```

```
In [7]: cath = CATHDB('cath.xml')
```

One way of using the `CATHDB` class is to navigate the CATH tree, using modified versions of methods and properties inherited from base classes in **:module:'~xml.etree.ElementTree'**.

The root of the tree and all other elements in it are instances of the `CATHElement` class, which is based on `Element`[10], allowing us to easily navigate the CATH tree structure using parent/child relationships as follows:

```
In [8]: root = cath.root

In [9]: root
Out[9]: <CATHElement: root (5 members)>
```

```
In [10]: node = root.getchildren()

In [11]: node
Out[11]:
```

[10]http://docs.python.org/library/xml.etree.elementtree.html#xml.etree.ElementTree.Element

```
<CATHCollection:
[<CATHElement: 1 (5 members)>
<CATHElement: 2 (21 members)>
<CATHElement: 3 (14 members)>
<CATHElement: 4 (1 members)>
<CATHElement: 6 (2 members)>]>
```

Any branching point node containing a collection of children is an instance of the `CATHCollection` class, which is based on the `CATHElement` class but has additional and modified properties and methods.

For example, collections return a list of values for the properties *cath* (CATH ID) and *name*, while elements return single values:

```
In [12]: node.name
Out[12]:
['Mainly Alpha',
 'Mainly Beta',
 'Alpha Beta',
 'Few Secondary Structures',
 'Special']
```

```
In [13]: node.cath
Out[13]: ['1', '2', '3', '4', '6']
```

```
In [14]: element = node[0]

In [15]: element.name
Out[15]: 'Mainly Alpha'
```

```
In [16]: element.cath
Out[16]: '1'
```

3.2 Searching CATH

We can also use the `CATHDB` class to find a particular part of the CATH hierarchy by CATH ID:

```
In [17]: node = cath.find('1.10.8')

In [18]: node.name
Out[18]: 'Helicase, Ruva Protein; domain 3'
```

We can also then examine its children:

```
In [19]: node.getchildren().name
Out[19]:
['DNA helicase RuvA subunit, C-terminal domain',
 'N-terminal domain of phosphatidylinositol transfer protein sec14p',
 'Albumin-binding domain',
 'Superfamily 1.10.8.50',
 'Superfamily 1.10.8.60',
 'Glutamate-tRNA synthetase, class I, anticodon-binding domain 1',
 'Magnesium chelatase subunit I, C-Terminal domain',
 'Ribosomal RNA adenine dimethylase-like, domain 2',
 'Photosystem I PsaF, reaction centre subunit III',
 'Superfamily 1.10.8.130',
 'PDCD5-like',
 'DNA primase S; domain 2',
 'Carbon monoxide dehydrogenase alpha subunit. Chain M, domain 1',
 'Replisome organizer (g39p helicase loader/inhibitor protein)',
 'Sirohaem synthase, dimerisation domain',
 'CofD-like domain',
 'DNA polymerase III clamp loader domain like',
 'HI0933 insert domain-like',
 'putative rabgap domain of human tbc1 domain family member 14 like domains',
 'ABC transporter ATPase domain-like',
 'uncharacterized protein sp1917 domain',
 'PG0816-like',
 'PG0816-like',
 'Bacterial muramidase',
 '3,6-anhydro-alpha-l-galactosidase',
 'nsp7 replicase',
 'Uncharacterised protein PF01937, DUF89, domain 1',
 'Internalin N-terminal Cap domain-like',
 'Enoyl acyl carrier protein reductase',
 'RecR Domain 1',
 'Helical domain of apoptotic protease-activating factors',
 'Vesicular stomatitis virus phosphoprotein C-terminal domain',
 'ppGaNTase-T1 linker domain-like',
 'Superfamily 1.10.8.470',
 'Superfamily 1.10.8.480',
 'Ced-4 linker helical domain-like',
 'HAMP domain in histidine kinase',
 'ExsD N-terminal domain-like',
 'DNA polymerase alpha-primase, subunit B, N-terminal domain',
 'FHIPEP family, domain 3',
 'Proto-chlorophyllide reductase 57 kD subunit B',
 'Antirestriction protein ArdA, domain 2',
 'Hypothetical protein YfmB',
 'Superfamily 1.10.8.590',
 'Phage phi29 replication organiser protein p16.7-like',
 'SirC, precorrin-2 dehydrogenase, C-terminal helical domain-like',
 'ORF12 helical bundle domain-like',
 'Cytochrome C biogenesis protein',
 'Uncharacterised protein PF13642 yp_926445, C-terminal domain',
 'Superfamily 1.10.8.660',
 'Ypt/Rab-GAP domain of gyp1p, domain 2',
 'Bacteriophage clamp loader A subunit, A domain',
 'Superfamily 1.10.8.710',
 'Region D6 of dynein motor',
 'Superfamily 1.10.8.730',
 'Superfamily 1.10.8.740',
 'Phosphoribosylformylglycinamidine synthase, linker domain',
 'Haem-binding uptake, Tiki superfamily, ChaN, domain 2',
 'Superfamily 1.10.8.770',
 'RNA-dependent RNA polymerase, slab domain, helical subdomain-like',
 'D-family DNA polymerase, DP1 subunit N-terminal domain',
 'Daxx helical bundle domain',
 'Ribosome-associated complex head domain',
 'Histone-lysine N methyltransferase , C-terminal domain-like',
 'Enterocin 7a-like',
 'Alpha-glycerophosphate oxidase, cap domain',
 'Birnavirus VP3 protein, domain 2',
 'Superfamily 1.10.8.890',
 'Superfamily 1.10.8.900',
 'Protein of unknown function DUF1465',
 'Uncharacterised protein, phage p2 ORF12',
 'Filoviridae VP35, C-terminal inhibitory domain, helical subdomain',
 'Superfamily 1.10.8.960',
 'Flavivirus envelope glycoprotein M-like',
 'Superfamily 1.10.8.990',
 'Ornithine 4,5 aminomutase S component, alpha subunit-like',
 'Superfamily 1.10.8.1010',
 'RecQ-mediated genome instability protein 1, N-terminal domain',
 'Superfamily 1.10.8.1040',
 'Antitoxin VbhA-like',
 'Corynebacterium glutamicum thioredoxin-dependent arsenate reductase, N-terminal domain',
 'Superfamily 1.10.8.1070',
 'Superfamily 1.10.8.1080',
 'Histone RNA hairpin-binding protein RNA-binding domain',
 'Bacterial toxin RNase RnlA/LsoA, C-terminal Dmd-binding domain',
 'Superfamily 1.10.8.1160',
 'Superfamily 1.10.8.1170',
 'Superfamily 1.10.8.1180',
 'Superfamily 1.10.8.1190',
 'Superfamily 1.10.8.1210',
 'Superfamily 1.10.8.1220',
 'Superfamily 1.10.8.1240',
 'Glutaminyl-tRNA synthetase, non-specific RNA binding region part 1, domain 1',
 'Superfamily 1.10.8.1310',
 'Superfamily 1.10.8.1320',
 'Intein homing endonuclease, domain III']
```

Lastly, the `CATHDB` object can be used to find different CATH domains within a particular PDB structure:

```
In [20]: result = cath.search('3kg2A')

In [21]: result.name
Out[21]:
['Superfamily 1.10.287.70',
 'Superfamily 3.40.50.2300',
 'Superfamily 3.40.50.2300',
 'Periplasmic binding protein-like II',
 'Periplasmic binding protein-like II']
```

```
In [22]: result.getSelstrs()
Out[22]:
['resindex 500 to 621 or resindex 777 to 808',
 'resindex 104 to 239 or resindex 346 to 376',
 'resindex 1 to 103 or resindex 240 to 345',
 'resindex 384 to 489 or resindex 724 to 773',
 'resindex 490 to 499 or resindex 622 to 723']
```

This iGluR example also illustrates that CATH domains may also not correspond to biological domains identified by other methods.

The N-terminal domain (NTD; residues 1 to 376), a type-I PBP domain, is split into CATH domains corresponding to the two lobes, which each belong to ‘Superfamily 3.40.50.2300’.

Likewise, the two lobes of the ligand-binding domain (LBD) are assigned as separate domains that both belong to ‘Periplasmic binding protein-like II’, which is usually the whole bi-lobed clamshell structure.

3.3 Getting atomic structures from CATH and building ensembles

We can also get PDB IDs associated with particular levels:

```
In [23]: node = cath.find('1.10.8.40')

In [24]: node.getPDBs()
Out[24]:
['2j5yA',
 '2j5yB',
 '2vdbB',
 '1tf0B',
 '1gabA',
 '1prbA',
 '2n35A',
 '2jwsA',
 '2kdlA',
 '2lhcA',
 '2lhgA',
 '2mh8A',
 '2fs1A',
 '1gjsA',
 '1gjtA']
```

Two other useful methods retrieve the associated CATH domains and selection strings.

```
In [25]: node.getDomains()
Out[25]:
[<CATHElement: 2j5yA00>,
 <CATHElement: 2j5yB00>,
 <CATHElement: 2vdbB00>,
 <CATHElement: 1tf0B00>,
 <CATHElement: 1gabA00>,
 <CATHElement: 1prbA00>,
 <CATHElement: 2n35A00>,
 <CATHElement: 2jwsA00>,
 <CATHElement: 2kdlA00>,
 <CATHElement: 2lhcA00>,
 <CATHElement: 2lhgA00>,
 <CATHElement: 2mh8A00>,
 <CATHElement: 2fs1A00>,
 <CATHElement: 1gjsA00>,
 <CATHElement: 1gjtA00>]
```

```
In [26]: node.getSelstrs()
Out[26]:
['resindex 1 to 61',
 'resindex 1 to 61',
 'resindex 1 to 55',
 'resindex 1 to 53',
 'resindex 1 to 53',
 'resindex 1 to 53',
 'resindex 1 to 52',
 'resindex 1 to 56',
 'resindex 1 to 56',
 'resindex 1 to 56',
 'resindex 1 to 56',
 'resindex 1 to 56',
 'resindex 1 to 56',
 'resindex 1 to 65',
 'resindex 1 to 65']
```

We can combine all of these together to fetch and parse structures from the PDB and make the appropriate selections at the same time:

```
In [27]: proteins = node.parsePDBs(subset='ca')

In [28]: proteins
Out[28]:
[<Selection: 'resindex 1 to 61' from 2j5yA_ca (60 atoms)>,
 <Selection: 'resindex 1 to 61' from 2j5yB_ca (60 atoms)>,
 <Selection: 'resindex 1 to 55' from 2vdbB_ca (54 atoms)>,
 <Selection: 'resindex 1 to 53' from 1tf0B_ca (52 atoms)>,
 <Selection: 'resindex 1 to 53' from 1gabA_ca (52 atoms; active #0 of 20 coordsets)>,
 <Selection: 'resindex 1 to 53' from 1prbA_ca (52 atoms)>,
 <Selection: 'resindex 1 to 52' from 2n35A_ca (51 atoms; active #0 of 20 coordsets)>,
 <Selection: 'resindex 1 to 56' from 2jwsA_ca (55 atoms; active #0 of 20 coordsets)>,
 <Selection: 'resindex 1 to 56' from 2kdlA_ca (55 atoms; active #0 of 20 coordsets)>,
 <Selection: 'resindex 1 to 56' from 2lhcA_ca (55 atoms; active #0 of 20 coordsets)>,
 <Selection: 'resindex 1 to 56' from 2lhgA_ca (55 atoms; active #0 of 10 coordsets)>,
 <Selection: 'resindex 1 to 56' from 2mh8A_ca (55 atoms; active #0 of 10 coordsets)>,
 <Selection: 'resindex 1 to 56' from 2fs1A_ca (55 atoms; active #0 of 20 coordsets)>,
 <Selection: 'resindex 1 to 65' from 1gjsA_ca (64 atoms; active #0 of 30 coordsets)>,
 <Selection: 'resindex 1 to 65' from 1gjtA_ca (64 atoms)>]
```

This then allows us to build a `PDBEnsemble` from them:

```
In [29]: ens = buildPDBEnsemble(proteins, mapping='CE')

In [30]: ens
Out[30]: <PDBEnsemble: Unknown (15 conformations; 60 atoms)>
```

CHAPTER

FOUR

DATA COLLECTION WITH PDB IDS

The first step in signature dynamics analysis is to collect a set of related protein structures and build a `PDBEnsemble`. This can be achieved by multiple routes: a query search of the PDB using `blastPDB()` or `searchDali()`, extraction of PDB IDs from the Pfam or CATH database, or input of a pre-defined list.

We demonstrate the usage of SignDy with a pre-defined list of transporter proteins sharing the common LeuT fold *[YS13]*. These proteins cycle through four typical states to transport a substrate molecule: outward-facing open (OFo), outward-facing closed (OFc), inward-facing open (IFo), inward-facing closed (IFc), but only the first three states have PDB structures available. If you know how to prepare an ensemble of structural homologs and wish to skip this part, you can download the ensemble file used in *[SZ18]* from here and proceed to the next tutorial.

First, make necessary imports from **ProDy_** and **Matplotlib_** packages if you haven't already.

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

In [2]: from pylab import *

In [3]: ion()
```

4.1 Prepare ensemble

For convinience and clarity, we define LeuT folds in separate lists taxonomically. For example, the PDB identifiers for bacterial Leucine transporters are defined as follows:

```
In [4]: LeuTs = ['2A65', '2Q6H', '2Q72', '2QB4', '2QEI', '2QJU', '3F3A', '3F3C',
   ...:          '3F3D', '3F3E', '3F48', '3F4I', '3F4J', '3GJC', '3GJD', '3GWU',
   ...:          '3GWV', '3GWW', '3MPN', '3MPQ', '3QS4', '3QS5', '3QS6', '3TT1',
   ...:          '3TU0', '3USG', '3USI', '3USJ', '3USK', '3USL', '3USM', '3USO',
   ...:          '3USP', '4FXZ', '4FY0', '4HMK', '4HOD', '4MM4', '4MM5', '4MM6',
   ...:          '4MM7', '4MM8', '4MM9', '4MMA', '4MMB', '4MMC', '4MMD', '4MME',
   ...:          '4MMF', '3TT3']
   ...:
```

Despite the fact that bacterial Leucine transporters can form dimers, we will only take the chain A in each structure:

```
In [5]: LeuTs = [protID + 'A' for protID in LeuTs]
```

Note that in the above line, we use list comprehension[11] to add a letter 'A' to each PDB identifier in the list to select chain A. We define other LeuT folds similarly:

```
In [6]: DATs = ['4M48', '4XNU', '4XNX', '4XP1', '4XP4', '4XP5', '4XP6',
   ...:         '4XP9C', '4XPA', '4XPB', '4XPF', '4XPG', '4XPH', '4XPT']
   ...:
```

[11] https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions

```
In [7]: DATs = [protID + 'A' for protID in DATs if protID is not '4XP9C']

In [8]: MhsTs = ['4US4A', '4US3A']

In [9]: vSGLTs = ['2XQ2A']

In [10]: Mhp1s = ['2JLN', '2X79', '4D1A', '4D1B', '4D1C', '4D1D']

In [11]: Mhp1s = [protID + 'A' for protID in Mhp1s]

In [12]: BetPs = ['2WITA', '2WITB', '2WITC', '3P03A', '3P03B', '3P03C',
   ....:          '4AINA', '4AINB', '4AINC', '4C7RA', '4C7RB', '4C7RC',
   ....:          '4DOJA', '4DOJB', '4DOJC', '4LLHA', '4LLHB', '4LLHC']
   ....:

In [13]: AdiCs = ['3L1L', '3LRB', '3LRC', '3NCY', '3OB6', '5J4I', '5J4N']

In [14]: AdiCs = [protID + 'A' for protID in AdiCs]

In [15]: CaiTs = ['4M8JA', '2WSXA', '2WSXB', '2WSXC', '2WSWA', '3HFXA']
```

`parsePDB()` allows us to parse multiple structures all at once, and we can use it to load all the PDB structures into **ProDy_** in one line. We only need the alpha carbon for our purpose, so we set `subset='ca'`:

```
In [16]: pdb_ids = LeuTs + DATs + MhsTs + vSGLTs + Mhp1s + BetPs + AdiCs + CaiTs

In [17]: ags = parsePDB(pdb_ids)

In [18]: len(ags)
Out[18]: 103
```

Any element in the list *ags* should be an `AtomGroup` instance. We can conveniently feed this list to `buildPDBEnsemble()` and let it build an `PDBEnsemble` for downstream analyses. We set `mapping=ce` to tell the function to use a structure alignment algorithm, CEalign [IS98], for building the ensemble. We also set `seqid=0` and `overlap=0` to make sure we apply no threshold of sequence identity or coverage/overlap to the building process.

```
In [19]: ens = buildPDBEnsemble(ags, mapping='ce', seqid=0, overlap=0, title='LeuT', subset='ca')

In [20]: ens
Out[20]: <PDBEnsemble: LeuT (103 conformations; 510 atoms)>
```

Finally we save the ensemble for later processing:

```
In [21]: saveEnsemble(ens, 'LeuT')
Out[21]: 'LeuT.ens.npz'
```

A more refined alignment procedure was adopted in the *[SZ18]* paper. A representative structure is chosen from each subtype of the proteins, e.g. LeuT, DAT, etc., and they are aligned to the LeuT representative using CEalign [IS98]. Then the rest are aligned to the representative structure of their own kind using the pairwise alignment algorithm because they are sequentially the same despite small differences.

CHAPTER

FIVE

CORE CALCULATIONS

In order to infer signature dynamics features we create mode ensembles from the PDB ensembles by calculating normal modes for each member of the `PDBEnsemble` as in the previous page or *[SZ18]*.

First, make necessary imports from ProDy and Matplotlib packages if you haven't already.

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

In [2]: from pylab import *

In [3]: ion()
```

5.1 Mode Ensemble

For this analysis we'll use the `PDBEnsemble` object, which we just created, to build a `ModeEnsemble`. There are options to select the model (`GNM` by default) and the way of considering non-aligned residues (default is `reduceModel()`, which treats them as environment).

If necessary, we first load the ensemble:

```
In [4]: ens = loadEnsemble('LeuT.ens.npz')
```

Then we calculated first 20 GNM modes for each member of the ensemble:

```
In [5]: gnms = calcEnsembleENMs(ens, model=GNM, trim='reduce', n_modes=20, match=True)

In [6]: gnms
Out[6]: <ModeEnsemble: 103 modesets (20 modes, 510 atoms)>
```

In this way, we will obtain one `ModeSet` for each member and 85 in total. Finding a consistent order of modes across different mode sets is critical to the accuracy of SignDy[12] calculations. Therefore, in the above code, *match* is set to *True* so that all other `ModeSet`'s are sorted to match the order of the reference` `:class:`.ModeSet`, which is the first `ModeSet` in the `ModeEnsemble` by default.

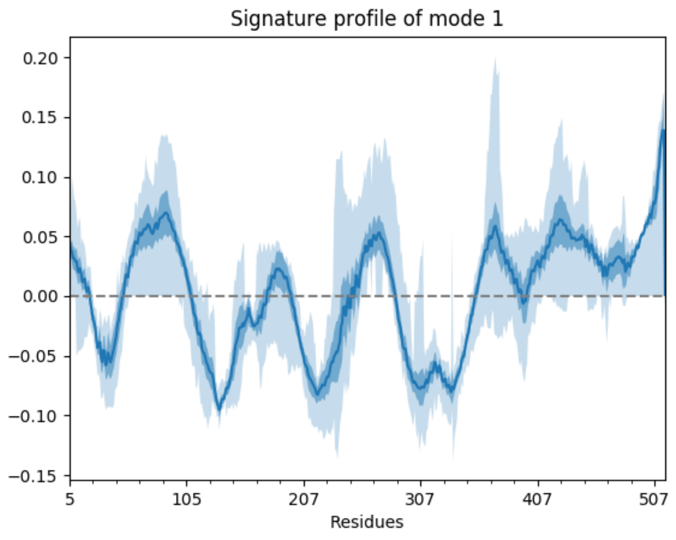
5.2 Signature Dynamics

Signature dynamics are calculated as the mean and standard deviation of various properties such as mode shapes and mean square fluctations.

For example, we can show the average and standard deviation of the shape of the first mode (second index 0). The first index of the mode ensemble is over conformations.

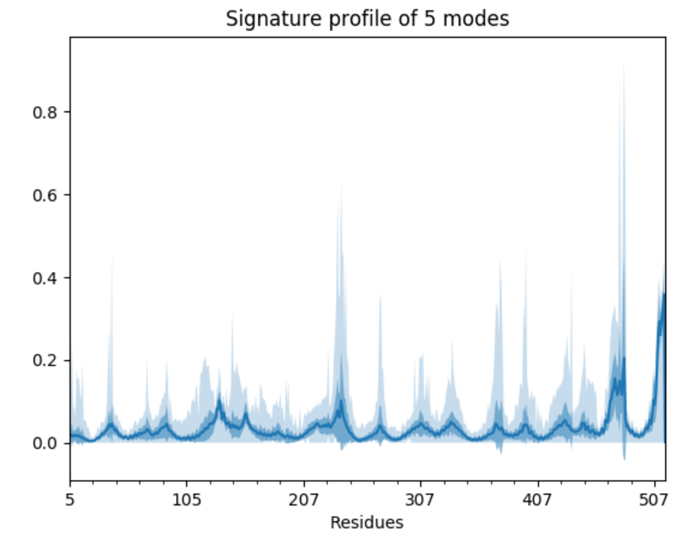
[12]http://prody.csb.pitt.edu/test_prody/tutorials/signdy_tutorial/

```
In [7]: showSignatureMode(gnms[:, 0]);
```



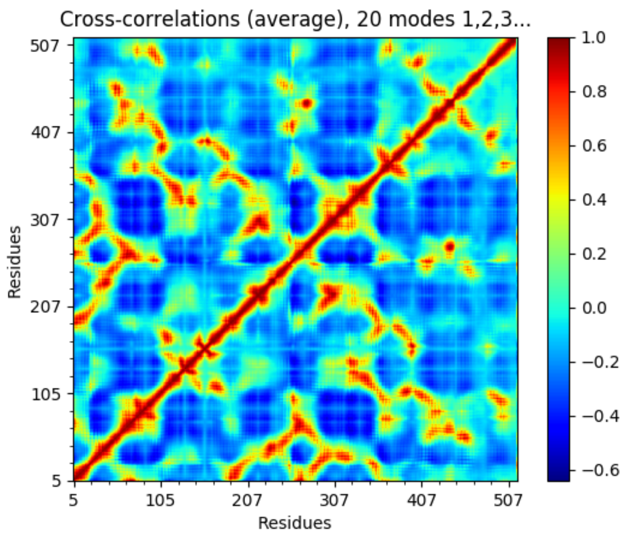
In the plot, the curve shows the mean values, the darker shade shows the standard deviations, and the lighter shade shows the range (minimum and maximum values). We can also show such things for properties involving multiple modes such as the mean square fluctuations from the first 5 modes,

```
In [8]: showSignatureSqFlucts(gnms[:, :5]);
```



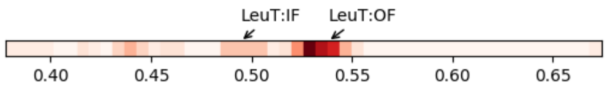
or the cross-correlations from the first 20.

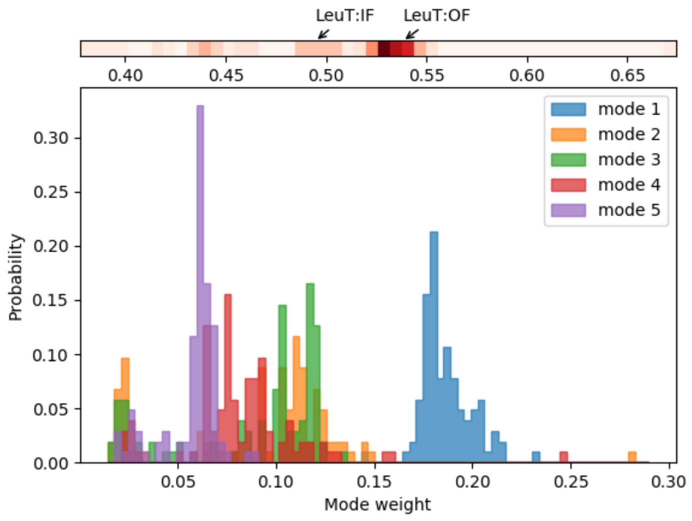
```
In [9]: showSignatureCrossCorr(gnms[:, :20]);
```



We can also look at distributions over values across different members of the ensemble such as inverse eigenvalue. We can show a bar above this with individual members labelled like *[JK15]*.

```
In [10]: highlights = {'2A65A': 'LeuT:OF', '3TT3A': 'LeuT:IF'}

In [11]: from matplotlib.gridspec import GridSpec

In [12]: gs = GridSpec(ncols=1, nrows=2, height_ratios=[1, 10], hspace=0.15)

In [13]: subplot(gs[0]);

In [14]: showVarianceBar(gnms[:, :5], fraction=True, highlights=highlights);

In [15]: xlabel('');

In [16]: subplot(gs[1]);

In [17]: showSignatureVariances(gnms[:, :5], fraction=True, bins=80, alpha=0.7);

In [18]: xlabel('Mode weight');
```





5.3 Saving the ModeEnsemble

Finally we save the mode ensemble for later processing:

```
In [19]: saveModeEnsemble(gnms, 'LeuT')
Out[19]: 'LeuT.modeens.npz'
```

CHAPTER

SIX

CLASSIFICATION USING SEQUENCE, STRUCTURE AND DYNAMICS DISTANCES

We can compare the dynamics of individual proteins using the spectral overlap, also known as covariance overlap. The arccosine of this value provides a distance metric. Calculating this for all pairs in a mode ensemble gives us the spectral distance matrix, which can be used to calculate a dynamics-based "phylogenetic" tree. This can be compared against matrices and trees calculated using sequence and structure distances.

Again here are the imports if you need them.

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

In [2]: from pylab import *

In [3]: ion()
```

6.1 Load PDBEnsemble and ModeEnsemble

We first load the `PDBEnsemble`:

```
In [4]: ens = loadEnsemble('LeuT.ens.npz')
```

Then we load the `ModeEnsemble`:

```
In [5]: gnms = loadModeEnsemble('LeuT.modeens.npz')
```

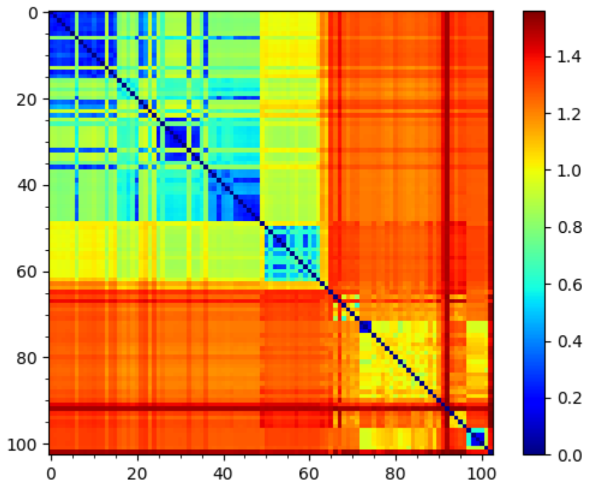
6.2 Spectral overlap and distance

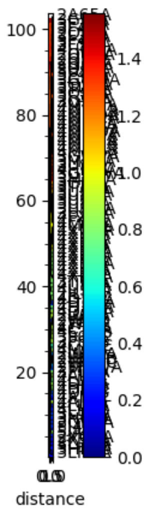
We calculate a distance matrix based on spectral overlaps (calculated as their arccosines), calculate a tree from it and reorder the matrix using the tree as follows:

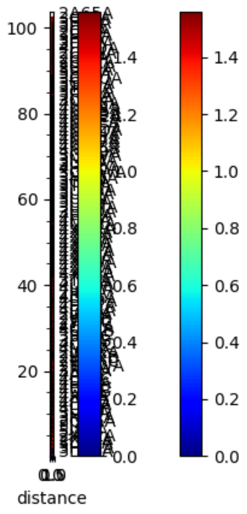
```
In [6]: sd_matrix = calcEnsembleSpectralOverlaps(gnms[:,:1], distance=True)

In [7]: labels = gnms.getLabels()

In [8]: so_tree = calcTree(names=labels,
   ...:                    distance_matrix=sd_matrix,
   ...:                    method='upgma')
   ...:

In [9]: reordered_sd, new_sd_indices = reorderMatrix(names=labels,
   ...:                                              matrix=sd_matrix,
   ...:                                              tree=so_tree)
   ...:
```

We can show the original and reordered spectral distance matrices and the tree as follows. `showTree()` has multiple *format* options. Here we show the output of using *plt*. This layout allows us to directly compare against the output from `showMatrix()` using the option *origin='upper'*.

```
In [10]: showMatrix(sd_matrix, origin='upper');

In [11]: showTree(so_tree, format='plt');

In [12]: showMatrix(reordered_sd, origin='upper');
```



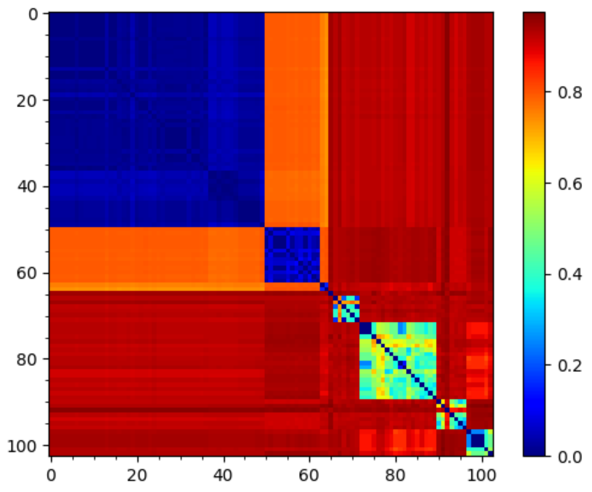


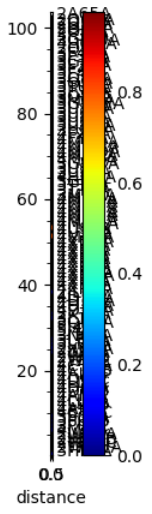


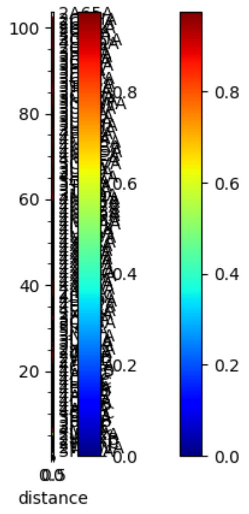
6.3 Sequence and structural distances

The sequence distance is given by the Hamming distance, which is calculated by subtracting the percentage identity (fraction) from 1, and the structural distance is the RMSD. We can also calculate and show the matrices and trees for these from the PDB ensemble.

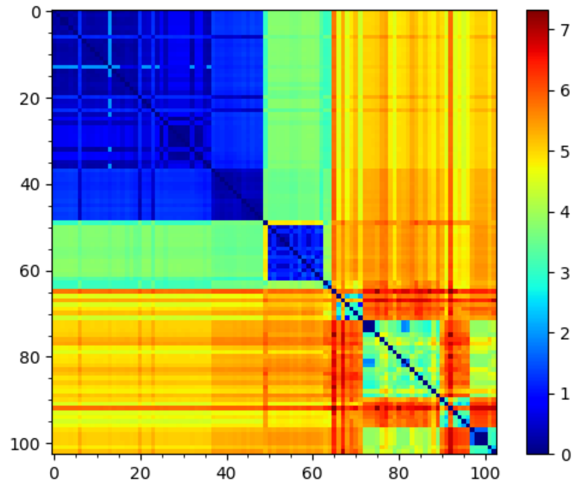
```
In [13]: seqid_matrix = buildSeqidMatrix(ens.getMSA())

In [14]: seqd_matrix = 1. - seqid_matrix

In [15]: showMatrix(seqd_matrix, origin='upper');

In [16]: seqd_tree = calcTree(names=labels,
   ....:                      distance_matrix=seqd_matrix,
   ....:                      method='upgma')
   ....:

In [17]: showTree(seqd_tree, format='plt');

In [18]: reordered_seqd, indices = reorderMatrix(labels, seqd_matrix, seqd_tree)

In [19]: showMatrix(reordered_seqd, origin='upper');
```

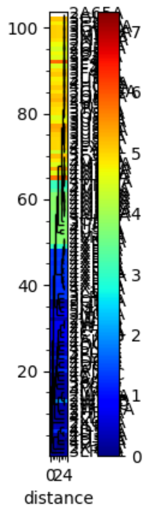

0
20
40
60
80
100
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8

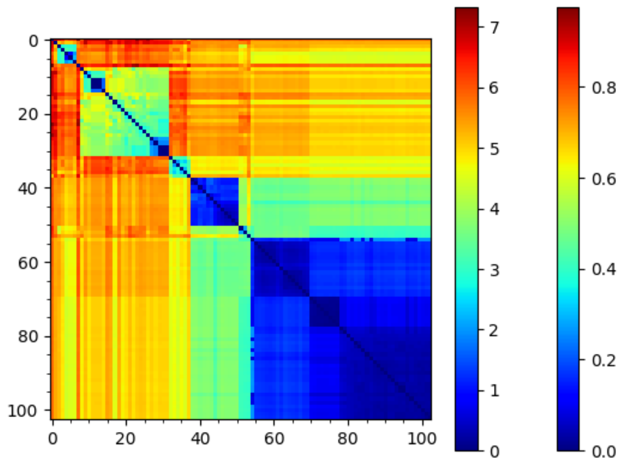

100
80
60
40
20
0.6
distance
0.0
0.2
0.4
0.6
0.8



```
In [20]: rmsd_matrix = ens.getRMSDs(pairwise=True)

In [21]: showMatrix(rmsd_matrix, origin='upper');

In [22]: rmsd_tree = calcTree(names=labels,
   ....:                      distance_matrix=rmsd_matrix,
   ....:                      method='upgma')
   ....:

In [23]: showTree(rmsd_tree, format='plt');

In [24]: reordered_rmsd, indices = reorderMatrix(labels, rmsd_matrix, rmsd_tree)

In [25]: showMatrix(reordered_rmsd, origin='upper');
```





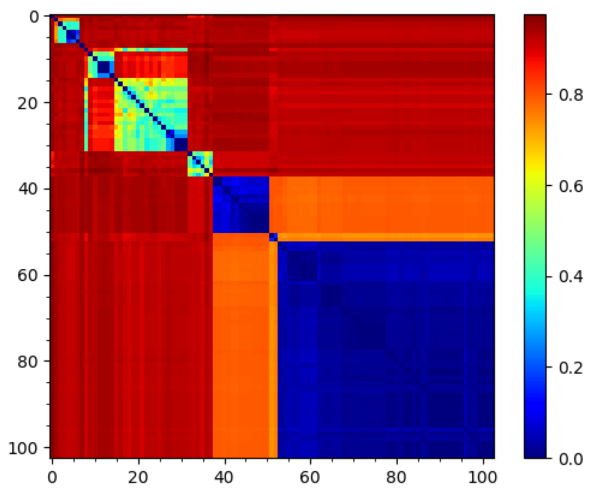


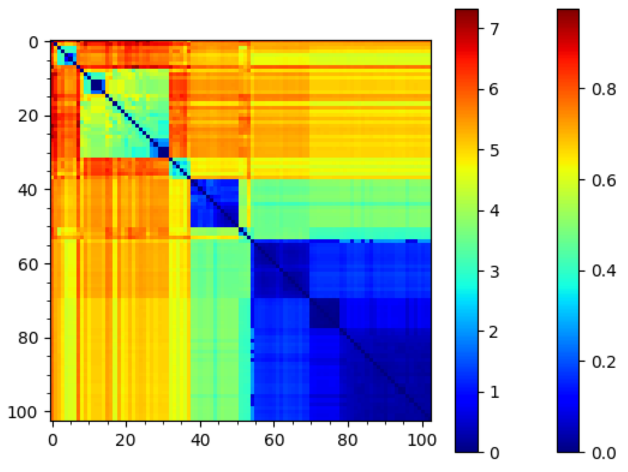
6.4 Comparing sequence, structural and dynamic classifications

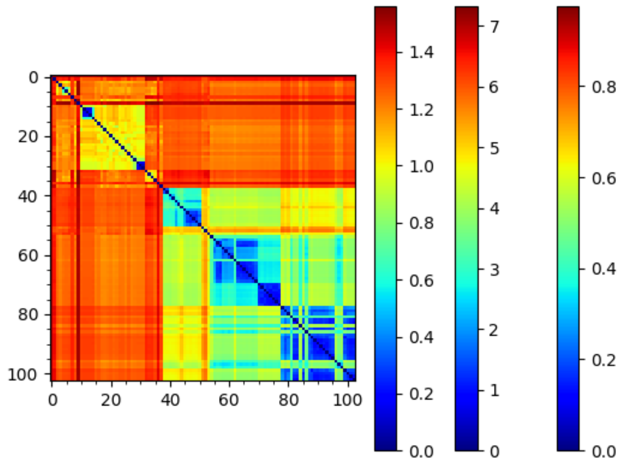
We can reorder the seqd and sod matrices by the RMSD tree too to compare them:

```
In [26]: reordered_seqd, indices = reorderMatrix(names=labels, matrix=seqd_matrix, tree=rmsd_tree)

In [27]: reordered_sd, indices = reorderMatrix(names=labels, matrix=sd_matrix, tree=rmsd_tree)
```

```
In [28]: showMatrix(reordered_seqd, origin='upper')
Out[28]:
(<matplotlib.image.AxesImage at 0x7f1ddf5a7e90>,
 [],
 <matplotlib.colorbar.Colorbar at 0x7f1ddf48d410>)

In [29]: showMatrix(reordered_rmsd, origin='upper')
Out[29]:
(<matplotlib.image.AxesImage at 0x7f1dd84b3950>,
 [],
 <matplotlib.colorbar.Colorbar at 0x7f1dd7e85810>)

In [30]: showMatrix(reordered_sd, origin='upper')
Out[30]:
(<matplotlib.image.AxesImage at 0x7f1dd8493250>,
 [],
 <matplotlib.colorbar.Colorbar at 0x7f1dd7e159d0>)
```







This analysis is quite sensitive to how many modes are used. As the number of modes approaches the full number, the dynamic distance order approaches the RMSD order. With smaller numbers, we see finer distinctions. This is particularly clear in the current case where we used just one mode.

Acknowledgments

Continued development of Protein Dynamics Software *ProDy* and associated programs is partially supported by the NIH[13]-funded Biomedical Technology and Research Center (BTRC) on *High Performance Computing for Multiscale Modeling of Biological Systems* (MMBios[14]) (P41 GM103712).

[13] http://www.nih.gov/
[14] http://mmbios.org/

[SZ18] Zhang S, Li H, Krieger J, Bahar I. Shared signature dynamics tempered by local fluctuations enables fold adaptability and specificity. *Mol. Biol. Evol.* **2019** 36(9):2053–2068

[LH10] Holm L, Rosenström P. Dali server: conservation mapping in 3D. *Nucleic Acids Res.* **2010** 10(38):W545-9

[IS21] Sillitoe I, Bordin N, Dawson N, Waman VP, Ashford P, Scholes HM, Pang CSM, Woodridge L, Rauer C, Sen N, Abbasian M, Le Cornu S, Lam SD, Berka K, Varekova IH, Svobodova R, Lees J, Orengo CA. CATH: increased structural coverage of functional space. *Nucleic Acids Res.* **2021** 49(D1):D266-D273

[JK15] Krieger J, Bahar I, Greger IH. Structure, Dynamics, and Allosteric Potential of Ionotropic Glutamate Receptor N-Terminal Domains. *Biophys. J.* **2015** 109(6):1136-48

[YS13] Shi Y. Common folds and transport mechanisms of secondary active transporters. *Annu. Rev. Biophys.* **2013** 42:51-72