ProDy

Protein Dynamics & Sequence Analysis

Elastic Network Models

Release 1.5.1

Ahmet Bakan

December 24, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

This tutorial describes how to use elastic network models, in particular *Gaussian Network Model (GNM)* (page 3) and *Anisotropic Network Model (ANM)* (page 8), for studying protein dynamics.

1.1 Required Programs

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

1.2 Recommended Programs

IPython[3] is highly recommended for interactive usage.

1.3 Getting Started

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

```
 792  oanm_eigvals.txt
3.3M  oanm_hes.txt
215K  oanm_slwevs.txt
 94K  ognm_kirchhoff.txt
```

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 *
```

[1] http://prody.csb.pitt.edu
[2] http://matplotlib.org
[3] http://ipython.org

```
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

GAUSSIAN NETWORK MODEL (GNM)

This example shows how to perform GNM calculations using an X-ray structure of ubiquitin. A `GNM` instance that stores the Kirchhoff matrix and normal mode data describing the intrinsic dynamics of the protein structure will be obtained. `GNM` instances and individual normal modes (`Mode`) can be used as input to functions in `prody.dynamics`[1] module.

See [Bahar97] (page 31) and [Haliloglu97] (page 31) for more information on the theory of GNM.

2.1 Parse structure

We start by importing everything from the ProDy package:

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

In [2]: from matplotlib.pylab import *

In [3]: ion() # turn interactive mode on
```

First we parse a PDB file by passing its identifier to `parsePDB()` function. Note that if file is not found in the current working directory, it will be downloaded.

```
In [4]: ubi = parsePDB('1aar')

In [5]: ubi
Out[5]: <AtomGroup: 1aar (1218 atoms)>
```

This file contains 2 chains, and a flexible C-terminal (residues 71-76). We only want to use Cα atoms of first 70 residues from chain A, so we select them:

```
In [6]: calphas = ubi.select('calpha and chain A and resnum < 71')

In [7]: calphas
Out[7]: <Selection: 'calpha and chai...and resnum < 71' from 1aar (70 atoms)>
```

See definition of "calpha", "chain", and other selection keywords in *Atom Selections*[2].

Note that, flexible design of classes allows users to select atoms other than alpha carbons to be used in GNM calculations.

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

2.2 Build Kirchoff matrix

Instantiate a `GNM` instance:

```
In [8]: gnm = GNM('Ubiquitin')
```

We can build Kirchhoff matrix using selected atoms and `GNM.buildKirchhoff()` method:

```
In [9]: gnm.buildKirchhoff(calphas)
```

We can get a copy of the Kirchhoff matrix using `GNM.getKirchhoff()` method:

```
In [10]: gnm.getKirchhoff()
Out[10]:
array([[ 11.,  -1.,  -1., ...,   0.,   0.,   0.],
       [ -1.,  15.,  -1., ...,   0.,   0.,   0.],
       [ -1.,  -1.,  20., ...,   0.,   0.,   0.],
       ...,
       [  0.,   0.,   0., ...,  20.,  -1.,  -1.],
       [  0.,   0.,   0., ...,  -1.,  21.,  -1.],
       [  0.,   0.,   0., ...,  -1.,  -1.,  12.]])
```

2.3 Parameters

We didn't pass any parameters, but `GNM.buildKirchhoff()` method accepts two of them, which by default are `cutoff=10.0` and `gamma=1.0`, i.e. `buildKirchhoff(calphas, cutoff=10., gamma=1.)`

```
In [11]: gnm.getCutoff()
Out[11]: 10.0

In [12]: gnm.getGamma()
Out[12]: 1.0
```

Note that it is also possible to use an externally calculated Kirchhoff matrix. Just pass it to the GNM instance using `GNM.setKirchhoff()` method.

2.4 Calculate normal modes

We now calculate normal modes from the Kirchhoff matrix.

```
In [13]: gnm.calcModes()
```

Note that by default 20 non-zero (or non-trivial) modes and 1 trivial mode are calculated. Trivial modes are not retained. To calculate different numbers of non-zero modes or to keep zero modes, try `gnm.calcModes(50, zeros=True)`.

2.5 Normal mode data

Get eigenvalues and eigenvectors:

```
In [14]: gnm.getEigvals().round(3)
Out[14]:
array([  2.502,   2.812,   4.366,   5.05 ,   7.184,   7.65 ,   7.877,
         9.08 ,   9.713,  10.132,  10.502,  10.644,  10.888,  11.157,
        11.285,  11.632,  11.78 ,  11.936,  12.006,  12.218])

In [15]: gnm.getEigvecs().round(3)
Out[15]:
array([[-0.064, -0.131, -0.245, ..., -0.256,  0.538, -0.   ],
       [-0.073, -0.085, -0.19 , ...,  0.006, -0.069,  0.032],
       [-0.076, -0.043, -0.135, ...,  0.017, -0.047,  0.018],
       ...,
       [-0.092,  0.064,  0.105, ...,  0.032, -0.042,  0.006],
       [-0.07 ,  0.099,  0.054, ...,  0.031,  0.024, -0.014],
       [-0.081,  0.135,  0.124, ...,  0.013, -0.04 , -0.018]])
```

Get covariance matrix:

```
In [16]: gnm.getCovariance().round(2)
Out[16]:
array([[ 0.08,  0.02,  0.01, ..., -0.01, -0.01, -0.01],
       [ 0.02,  0.02,  0.01, ..., -0.  , -0.  , -0.01],
       [ 0.01,  0.01,  0.01, ...,  0.  , -0.  , -0.  ],
       ...,
       [-0.01, -0.  ,  0.  , ...,  0.01,  0.01,  0.01],
       [-0.01, -0.  , -0.  , ...,  0.01,  0.01,  0.02],
       [-0.01, -0.01, -0.  , ...,  0.01,  0.02,  0.05]])
```

Note that covariance matrices are calculated using the available modes in the model, which is the slowest 20 modes in this case. If the user calculates M modes, these M modes will be used in calculating the covariance matrix.

2.6 Individual modes

Normal mode indices start from 0, so slowest mode has index 0.

```
In [17]: slowest_mode = gnm[0]

In [18]: slowest_mode.getEigval().round(3)
Out[18]: 2.5019999999999998

In [19]: slowest_mode.getEigvec().round(3)
Out[19]:
array([-0.064, -0.073, -0.076, -0.112, -0.092, -0.143, -0.164, -0.205,
       -0.24 , -0.313, -0.192, -0.152, -0.066, -0.07 , -0.025, -0.031,
        0.001, -0.006, -0.015,  0.027,  0.042,  0.055,  0.063,  0.09 ,
        0.09 ,  0.069,  0.132,  0.175,  0.145,  0.121,  0.195,  0.218,
        0.158,  0.217,  0.245,  0.214,  0.225,  0.171,  0.2  ,  0.151,
        0.102,  0.043, -0.029, -0.064, -0.072, -0.086, -0.09 , -0.078,
       -0.057, -0.011,  0.016,  0.061,  0.058,  0.043,  0.029,  0.013,
        0.004,  0.011, -0.013, -0.037, -0.05 , -0.059, -0.07 , -0.094,
       -0.094, -0.099, -0.097, -0.092, -0.07 , -0.081])
```

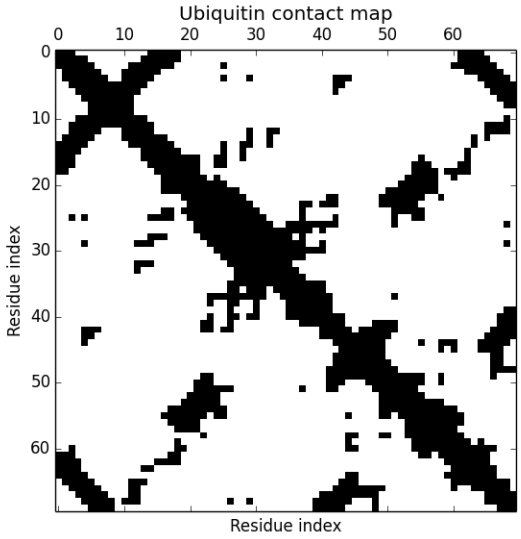
By default, modes with 0 eigenvalue are excluded. If they were retained, slowest non-trivial mode would have index 6.

2.7 Plot results

ProDy plotting functions are prefixed with `show`. Let's use some of them to plot data:

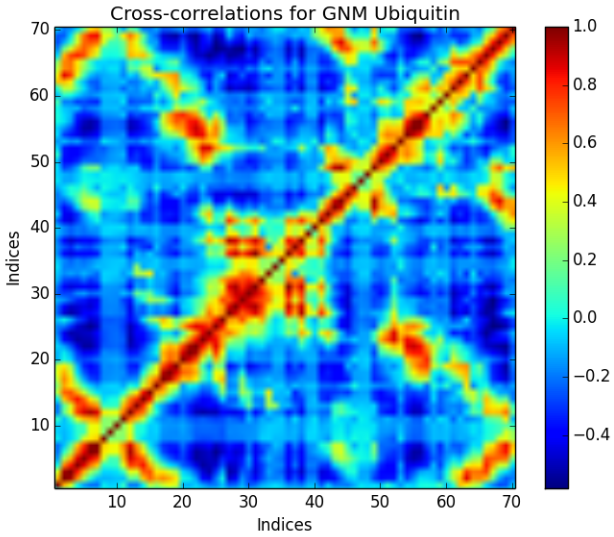
2.7.1 Contact Map

```
In [20]: showContactMap(gnm);
```



2.7.2 Cross-correlations

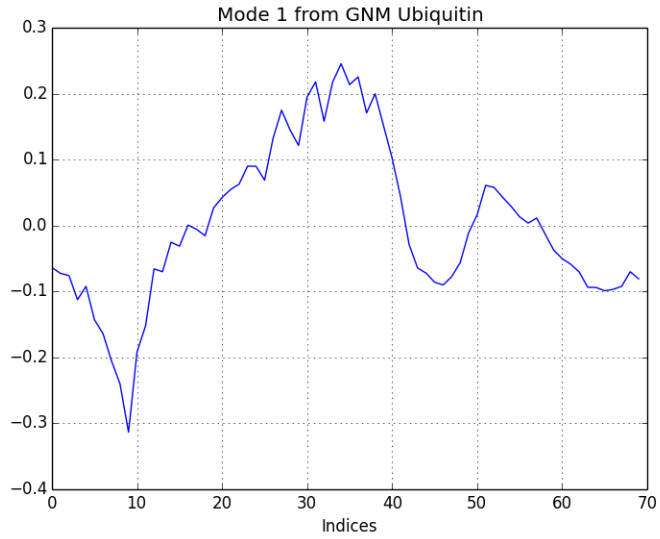
```
In [21]: showCrossCorr(gnm);
```



2.7.3 Slow mode shape

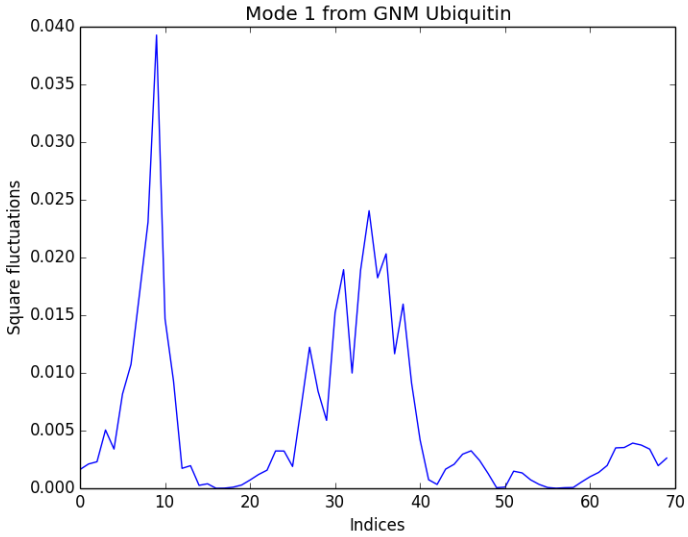
```
In [22]: showMode(gnm[0]);
```

```
In [23]: plt.grid();
```



2.7.4 Square fluctuations

```
In [24]: showSqFlucts(gnm[0]);
```



CHAPTER

THREE

ANISOTROPIC NETWORK MODEL (ANM)

This example shows how to perform ANM calculations, and retrieve normal mode data. An `ANM` instance that stores Hessian matrix (and also Kirchhoff matrix) and normal mode data describing the intrinsic dynamics of the protein structure will be obtained. `ANM` instances and individual normal modes (`Mode`) can be used as input to functions in `dynamics` module.

See [Doruker00] (page 31) and [Atilgan01] (page 31) for more information on the theory of ANM.

3.1 Parse structure

We start by importing everything from the ProDy package:

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

In [2]: from pylab import *

In [3]: ion()
```

We start with parsing a PDB file by passing an identifier. Note that if a file is not found in the current working directory, it will be downloaded.

```
In [4]: p38 = parsePDB('1p38')

In [5]: p38
Out[5]: <AtomGroup: 1p38 (2962 atoms)>
```

We want to use only Cα atoms, so we select them:

```
In [6]: calphas = p38.select('protein and name CA')

In [7]: calphas
Out[7]: <Selection: 'protein and name CA' from 1p38 (351 atoms)>
```

We can also make the same selection like this:

```
In [8]: calphas2 = p38.select('calpha')

In [9]: calphas2
Out[9]: <Selection: 'calpha' from 1p38 (351 atoms)>
```

To check whether the selections are the same, we can try:

```
In [10]: calphas == calphas2
Out[10]: True
```

Note that, ProDy atom selector gives the flexibility to select any set of atoms to be used in ANM calculations.

3.2 Build Hessian

We instantiate an `ANM` instance:

```
In [11]: anm = ANM('p38 ANM analysis')
```

Then, build the Hessian matrix by passing selected atoms (351 Cα's) to `ANM.buildHessian()` method:

```
In [12]: anm.buildHessian(calphas)
```

We can get a copy of the Hessian matrix using `ANM.getHessian()` method:

```
In [13]: anm.getHessian().round(3)
Out[13]:
array([[ 9.959, -3.788,  0.624, ...,  0.   ,  0.   ,  0.   ],
       [-3.788,  7.581,  1.051, ...,  0.   ,  0.   ,  0.   ],
       [ 0.624,  1.051,  5.46 , ...,  0.   ,  0.   ,  0.   ],
       ...,
       [ 0.   ,  0.   ,  0.   , ...,  1.002, -0.282,  0.607],
       [ 0.   ,  0.   ,  0.   , ..., -0.282,  3.785, -2.504],
       [ 0.   ,  0.   ,  0.   , ...,  0.607, -2.504,  4.214]])
```

3.3 Parameters

We didn't pass any parameters to `ANM.buildHessian()` method, but it accepts *cutoff* and *gamma* parameters, for which default values are `cutoff=15.0` and `gamma=1.0`.

```
In [14]: anm.getCutoff()
Out[14]: 15.0

In [15]: anm.getGamma()
Out[15]: 1.0
```

Note that it is also possible to use an externally calculated Hessian matrix. Just pass it to the ANM instance using `ANM.setHessian()` method.

3.4 Calculate normal modes

Calculate modes using `ANM.calcModes()` method:

```
In [16]: anm.calcModes()
```

Note that by default 20 non-zero (or non-trivial) and 6 trivial modes are calculated. Trivial modes are not retained. To calculate a different number of non-zero modes or to keep zero modes, try `anm.calcModes(50, zeros=True)`.

3.5 Normal modes data

```
In [17]: anm.getEigvals().round(3)
Out[17]:
array([ 0.179,  0.334,  0.346,  0.791,  0.942,  1.012,  1.188,  1.304,
        1.469,  1.546,  1.608,  1.811,  1.925,  1.983,  2.14 ,  2.298,
        2.33 ,  2.364,  2.69 ,  2.794])

In [18]: anm.getEigvecs().round(3)
Out[18]:
array([[ 0.039, -0.045,  0.007, ...,  0.105,  0.032, -0.038],
       [ 0.009, -0.096, -0.044, ...,  0.091,  0.036, -0.037],
       [ 0.058, -0.009,  0.08 , ..., -0.188, -0.08 , -0.063],
       ...,
       [ 0.046, -0.093, -0.131, ...,  0.018, -0.008,  0.006],
       [ 0.042, -0.018, -0.023, ...,  0.014, -0.043,  0.037],
       [ 0.08 , -0.002, -0.023, ...,  0.024, -0.023, -0.009]])
```

You can get the covariance matrix as follows:

```
In [19]: anm.getCovariance().round(2)
Out[19]:
array([[ 0.03,  0.03, -0.  , ...,  0.  ,  0.  ,  0.01],
       [ 0.03,  0.06, -0.03, ...,  0.01, -0.  ,  0.01],
       [-0.  , -0.03,  0.09, ..., -0.01, -0.  ,  0.01],
       ...,
       [ 0.  ,  0.01, -0.01, ...,  1.21,  0.  , -0.17],
       [ 0.  , -0.  , -0.  , ...,  0.  ,  0.41,  0.38],
       [ 0.01,  0.01,  0.01, ..., -0.17,  0.38,  0.4 ]])
```

Covariance matrices are calculated using the available modes (slowest 20 modes in this case). If the user calculates M slowest modes, only they will be used in the calculation of covariances.

3.6 Individual modes

Normal mode indices in Python start from 0, so the slowest mode has index 0. By default, modes with zero eigenvalues are excluded. If they were retained, the slowest non-trivial mode would have index 6.

Get the slowest mode by indexing `ANM` instance as follows:

```
In [20]: slowest_mode = anm[0]

In [21]: slowest_mode.getEigval().round(3)
Out[21]: 0.17899999999999999

In [22]: slowest_mode.getEigvec().round(3)
Out[22]: array([ 0.039,  0.009,  0.058, ...,  0.046,  0.042,  0.08 ])
```

3.7 Write NMD file

ANM results in NMD format can be visualized using *Normal Mode Wizard*[1] VMD[2] plugin. The following statement writes the slowest 3 ANM modes into an NMD file:

[1] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz
[2] http://www.ks.uiuc.edu/Research/vmd

```
In [23]: writeNMD('p38_anm_modes.nmd', anm[:3], calphas)
Out[23]: 'p38_anm_modes.nmd'
```

Note that slicing an ANM objects returns a list of modes. In this case, slowest 3 ANM modes were written into NMD file.

3.8 View modes in VMD

First make sure that the VMD path is correct

```
In [24]: pathVMD()
Out[24]: '/usr/local/bin/vmd'

# if this is incorrect use setVMDpath to correct it
In [25]: viewNMDinVMD('p38_anm_modes.nmd')
```

This will show the slowest 3 modes in VMD using NMWiz. This concludes the ANM example. Many of the methods demonstrated here apply to other NMA models, such as GNM and EDA.

CHAPTER

FOUR

USING AN EXTERNAL MATRIX

This example shows how to use matrices from external software in ANM or GNM analysis of protein dynamics.

4.1 Parse Hessian

We start by importing everything from the ProDy package:

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

In [2]: from matplotlib.pylab import *

In [3]: ion()  # turn interactive mode on
```

The input file that contains the Hessian matrix has the following format (`oanm_hes.txt`):

```
1       1    9.958948135375977e+00
1       2   -3.788214445114136e+00
1       3    6.236155629158020e-01
1       4   -7.820609807968140e-01
1       5    1.050322428345680e-01
1       6   -3.992616236209869e-01
1       7   -7.818332314491272e-01
1       8   -1.989762037992477e-01
1       9   -3.619094789028168e-01
1      10   -5.224789977073669e-01
...
```

`parseSparseMatrix()` can be used for parsing the above file:

```
In [4]: hessian = parseSparseMatrix('oanm_hes.txt', symmetric=True)

In [5]: hessian.shape
Out[5]: (1053, 1053)
```

4.2 ANM calculations

Rest of the calculations can be performed as follows:

```
In [6]: anm = ANM('Using external Hessian')

In [7]: anm.setHessian(hessian)
```

```
In [8]: anm.calcModes()

In [9]: anm
Out[9]: <ANM: Using external Hessian (20 modes; 351 nodes)>
```

For more information, see *Anisotropic Network Model (ANM)* (page 8).

4.3 Parse Kirchhoff

The input file that contains the Kirchhoff matrix has the following format (`ognm_kirchhoff.txt`):

```
 3316
1       1       5.00
1       2      -1.00
1       3      -1.00
1       4      -1.00
1      91      -1.00
1     343      -1.00
2       2      10.00
2       3      -1.00
2       4      -1.00
...
```

```
In [10]: kirchhoff = parseSparseMatrix('ognm_kirchhoff.txt',
   ....:                               symmetric=True, skiprows=1)
   ....:

In [11]: kirchhoff.shape
Out[11]: (351, 351)
```

4.4 GNM calculations

Rest of the GNM calculations can be performed as follows:

```
In [12]: gnm = GNM('Using external Kirchhoff')

In [13]: gnm.setKirchhoff(kirchhoff)

In [14]: gnm.calcModes()

In [15]: gnm
Out[15]: <GNM: Using external Kirchhoff (20 modes; 351 nodes)>
```

For more information, see *Gaussian Network Model (GNM)* (page 3).

CHAPTER

FIVE

CUSTOM GAMMA FUNCTIONS

This example shows how to develop custom force constant functions for `ANM` (or `GNM`) calculations.

We will use the relation shown in the figure below. For Cα atoms that are 10 to 15 Å apart from each other, we use a unit force constant. For those that are 4 to 10 Å apart, we use a 2 times stronger force constant. For those that are within 4 Å of each other (i.e. those from connected residue pairs), we use a 10 times stronger force constant.

We will obtain an `ANM` instance that stores Hessian and Kirchhoff matrices and normal mode data describing the intrinsic dynamics of the protein structure. `ANM` instances and individual normal modes (`Mode`) can be used as input to functions in `prody.dynamics`[1] module.

5.1 Parse structure

We start by importing everything from ProDy, Numpy, and Matplotlib packages:

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

In [2]: from matplotlib.pylab import *

In [3]: ion() # turn interactive mode on
```

We start with parsing a PDB file by passing an identifier.

```
In [4]: p38 = parsePDB('1p38')

In [5]: p38
Out[5]: <AtomGroup: 1p38 (2962 atoms)>
```

We want to use only Cα atoms, so we select them:

```
In [6]: calphas = p38.select('protein and name CA')

In [7]: calphas
Out[7]: <Selection: 'protein and name CA' from 1p38 (351 atoms)>
```

5.2 Force Constant Function

We define the aformentioned function as follows:

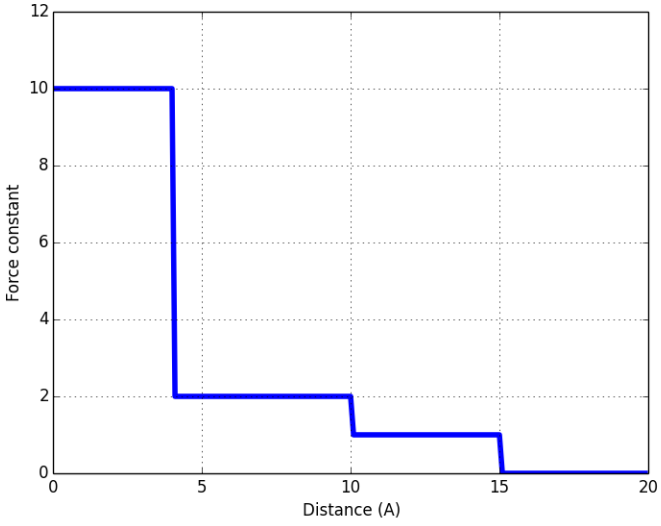
[1] http://prody.csb.pitt.edu/manual/reference/dynamics/index.html#prody.dynamics

```
In [8]: def gammaDistanceDependent(dist2, *args):
   ...:     """Return a force constant based on the given square distance."""
   ...:     if dist2 <= 16:
   ...:         return 10
   ...:     elif dist2 <= 100:
   ...:         return 2
   ...:     elif dist2 <= 225:
   ...:         return 1
   ...:     else:
   ...:         return 0
   ...:
```

Note that the input to this function from `ANM` or `GNM` is the square of the distance. In addition, node (atom or residue) indices are passed to this function, that's why we used `*args` in the function definition.

Let's test how it works:

```
In [9]: dist = arange(0, 20, 0.1)

In [10]: gamma = map(gammaDistanceDependent, dist ** 2)

In [11]: plot(dist, gamma, lw=4);

In [12]: axis([0, 20, 0, 12]);

In [13]: xlabel('Distance (A)');

In [14]: ylabel('Force constant');

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



5.3 ANM calculations

We use selected atoms (351 Cα's) and `gammaDistanceDependent` function for ANM calculations as follows:

```
In [16]: anm = ANM('1p38')

In [17]: anm.buildHessian(calphas, cutoff=15, gamma=gammaDistanceDependent)

In [18]: anm.calcModes()
```

For more detailed examples see *Anisotropic Network Model (ANM)* (page 8) or *Gaussian Network Model (GNM)* (page 3).

CHAPTER

SIX

EDITING A MODEL

This example shows how to analyze the normal modes corresponding to a system of interest. In this example, ANM calculations will be performed for HIV-1 reverse transcriptase (RT) subunits p66 and p51. Analysis will be made for subunit p66. Output is a reduced/sliced model that can be used as input to analysis and plotting functions.

6.1 ANM calculations

We start by importing everything from the ProDy package:

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

In [2]: from matplotlib.pylab import *

In [3]: ion()
```

We start with parsing the Cα atoms of the RT structure 1DLO and performing ANM calculations for them:

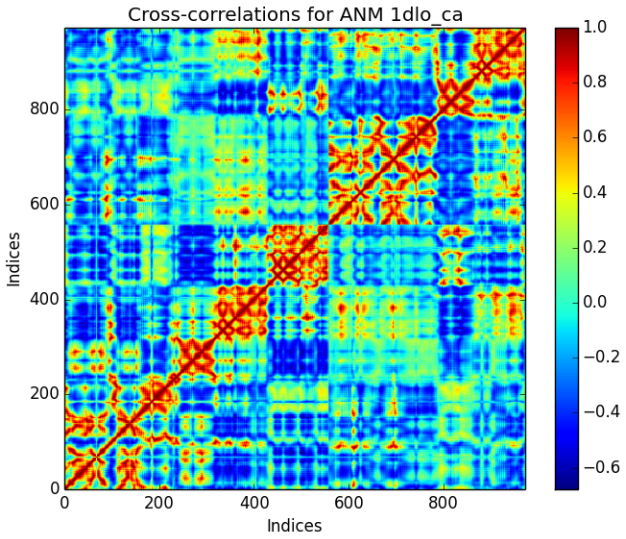
```
In [4]: rt = parsePDB('1dlo', subset="ca")

In [5]: anm, sel = calcANM(rt)

In [6]: anm
Out[6]: <ANM: 1dlo_ca (20 modes; 971 nodes)>

In [7]: saveModel(anm, 'rt_anm')
Out[7]: 'rt_anm.anm.npz'

In [8]: anm[:5].getEigvals().round(3)
Out[8]: array([ 0.039,  0.063,  0.126,  0.181,  0.221])

In [9]: (anm[0].getArray() ** 2).sum() ** 0.5
Out[9]: 1.0000000000000002
```

6.1.1 Analysis

We can plot the cross-correlations and square fluctuations for the full model as follows:

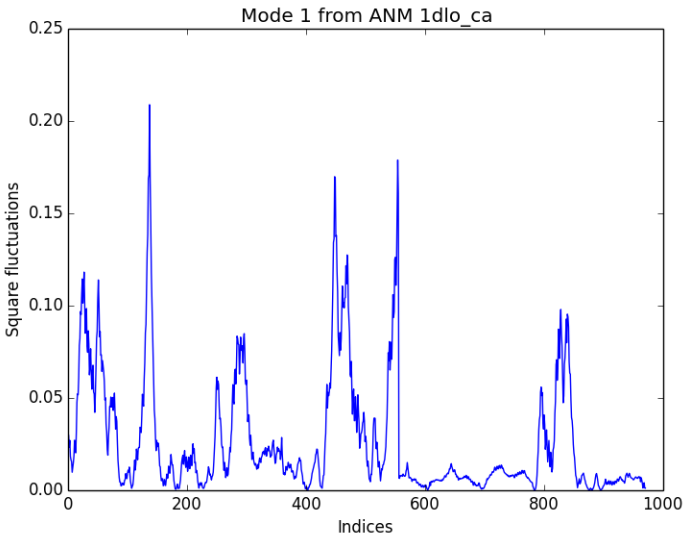
Cross-correlations

```
In [10]: showCrossCorr(anm);
```



Square fluctuations

```
In [11]: showSqFlucts(anm[0]);
```



6.2 Slicing a model

Slicing a model is analogous to slicing a list, i.e.:

```
In [12]: numbers = list(range(10))
```

```
In [13]: numbers
Out[13]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [14]: slice_first_half = numbers[:10]

In [15]: slice_first_half
Out[15]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In this case, we want to slice normal modes, so that we will handle mode data corresponding to subunit p66, which is chain A in the structure. We use `sliceModel()` function:

```
In [16]: anm_slc_p66, sel_p66 = sliceModel(anm, rt, 'chain A')

In [17]: anm_slc_p66
Out[17]: <ANM: 1dlo_ca slice chain A (20 modes; 556 nodes)>
```

You see that now the sliced model contains 556 nodes out of the 971 nodes in the original model.

```
In [18]: saveModel(anm_slc_p66, 'rt_anm_sliced')
Out[18]: 'rt_anm_sliced.anm.npz'

In [19]: anm_slc_p66[:5].getEigvals().round(3)
Out[19]: array([ 0.039,  0.063,  0.126,  0.181,  0.221])

In [20]: '%.3f' % (anm_slc_p66[0].getArray() ** 2).sum() ** 0.5
Out[20]: '0.895'
```

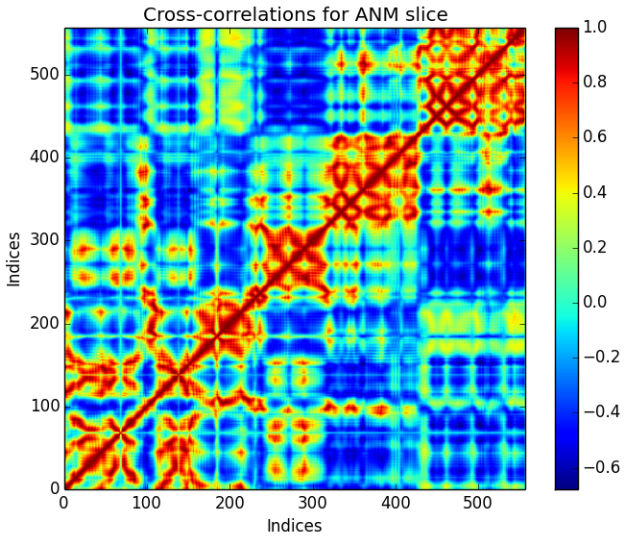
Note that slicing does not change anything in the model apart from taking parts of the modes matching the selection. The sliced model contains fewer nodes, has the same eigenvalues, and modes in the model are not normalized.

6.2.1 Analysis

We plot the cross-correlations and square fluctuations for the sliced model in the same way. Note that the plots contain the selected part of the model without any change:

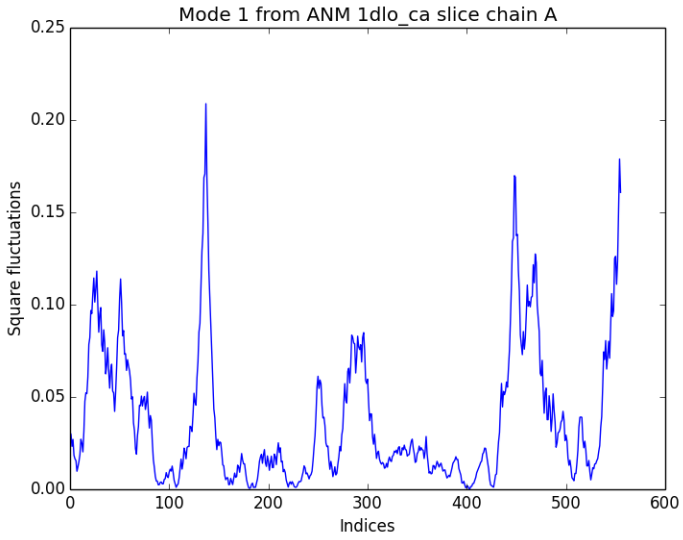
Cross-correlations

```
In [21]: showCrossCorr(anm_slc_p66);
   ....:

In [22]: title('Cross-correlations for ANM slice');
```



Square fluctuations

```
In [23]: showSqFlucts(anm_slc_p66[0]);
```



6.3 Reducing a model

We reduce the ANM model to subunit p66 using `reduceModel()` function. This function implements the method described in 2000 paper of Hinsen et al. [KH00]

```
In [24]: anm_red_p66, sel_p66 = reduceModel(anm, rt, 'chain A')

In [25]: anm_red_p66.calcModes()

In [26]: anm_red_p66
```

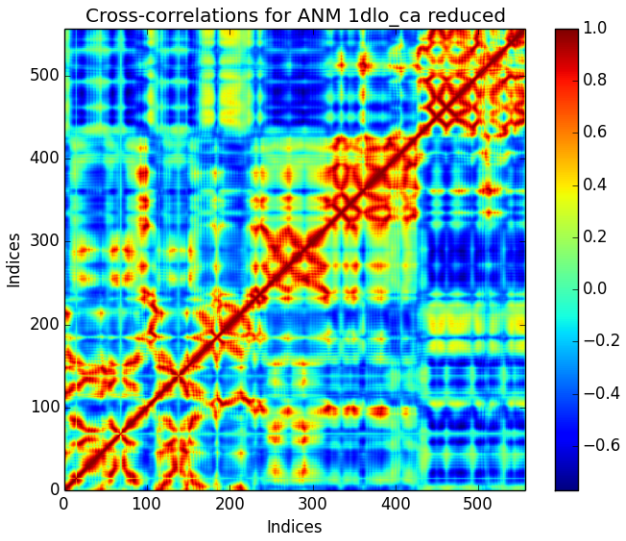
```
Out[26]: <ANM: 1dlo_ca reduced (20 modes; 556 nodes)>

In [27]: saveModel(anm_red_p66, 'rt_anm_reduced')
Out[27]: 'rt_anm_reduced.anm.npz'

In [28]: anm_red_p66[:5].getEigvals().round(3)
Out[28]: array([ 0.05 ,  0.098,  0.214,  0.289,  0.423])

In [29]: '%.3f' % (anm_red_p66[0].getArray() ** 2).sum() ** 0.5
Out[29]: '1.000'
```

6.3.1 Analysis

We plot the cross-correlations and square fluctuations for the reduced model in the same way. Note that in this case the plots are not identical to the full model:

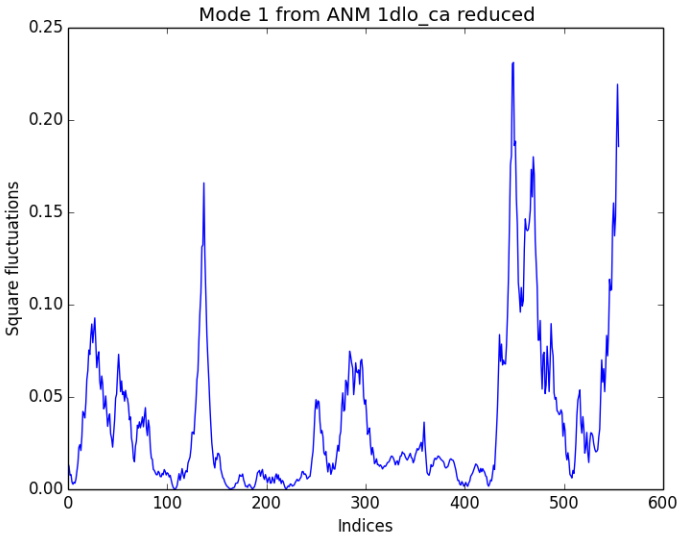
Cross-correlations

```
In [30]: showCrossCorr(anm_red_p66);
```



Square fluctuations

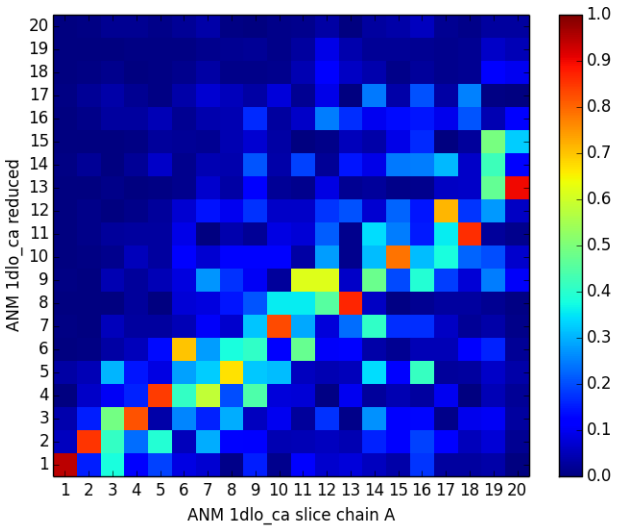
```
In [31]: showSqFlucts(anm_red_p66[0]);
```



6.4 Compare reduced and sliced models

We can compare the sliced and reduced models by plotting the overlap table between modes:

```
In [32]: showOverlapTable(anm_slc_p66, anm_red_p66);
```



The sliced and reduced models are not the same. While the purpose of slicing is simply enabling easy plotting/analysis of properties of a part of the system, reducing has other uses as in [WZ05] (page 31).

CHAPTER

SEVEN

EXTEND A COARSE-GRAINED MODEL

This example shows how to extend normal modes calculated for a coarse-grained model to a larger set of atoms. Extended model can be used to generate alternate conformers that can be saves in PDB format.

We start by importing everything from the ProDy package:

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

In [2]: from matplotlib.pylab import *

In [3]: ion()
```

Conformers can be generated along any set of normal modes. In this example, we will calculate normal modes for the unbound structure of p38 MAP kinase and generate backbone trace conformations.

```
In [4]: p38 = parsePDB('1p38')

In [5]: p38_ca = p38.select('calpha')

In [6]: anm = ANM('1p38')

In [7]: anm.buildHessian(p38_ca)

In [8]: anm.calcModes()
```

7.1 Extrapolation

ANM modes are extended using the `extendModel()` function:

```
In [9]: bb_anm, bb_atoms = extendModel(anm, p38_ca, p38.select('backbone'))

In [10]: bb_anm
Out[10]: <NMA: Extended ANM 1p38 (20 modes; 1404 atoms)>

In [11]: bb_atoms
Out[11]: <AtomMap: Selection 'backbone' from 1p38 (1404 atoms)>
```

Note that `GNM`, `PCA`, and `NMA` instances can also be used as input to this function.

7.2 Write NMD file

Extended modes can be visualized in VMD using *Normal Mode Wizard*[1] using an NMD file:

```
In [12]: writeNMD('p38_anm_backbone.nmd', bb_anm, bb_atoms)
Out[12]: 'p38_anm_backbone.nmd'
```

7.3 Sample conformers

We can use the extended model to sample backbone conformers:

```
In [13]: ensemble = sampleModes(bb_anm[:3], bb_atoms, n_confs=40, rmsd=0.8)

In [14]: ensemble
Out[14]: <Ensemble: Conformations along 3 modes from NMA Extended ANM 1p38 (40 conformations; 1404 at
```

Note that we made use of ANM modes to generate full atomic conformers. These conformers would need geometry optimization before they can be used for modeling.

7.4 Write PDB file

Generated conformers can be written in PDB format as follows:

```
In [15]: backbone = bb_atoms.copy()

In [16]: backbone.addCoordset(ensemble)

In [17]: writePDB('p38_backbone_ensemble.pdb', backbone)
Out[17]: 'p38_backbone_ensemble.pdb'
```

[1] http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

CHAPTER

EIGHT

NORMAL MODE ALGEBRA

This part shows how to use some handy features of `Mode` objects.

8.1 ANM Calculations

We will compare modes from two ANMs for the same protein, but everything applies to comparison of ANMs and PCAs (as long as they contain same number of atoms).

Let's get started by getting ANM models for two related protein structures:

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

In [2]: str1 = parsePDB('1p38')

In [3]: str2 = parsePDB('1r39')
```

Find and align matching chains

```
In [4]: matches = matchChains(str1, str2)

In [5]: match = matches[0]

In [6]: ch1 = match[0]

In [7]: ch2 = match[1]
```

Minimize RMSD by superposing `ch2` onto `ch1`:

```
In [8]: ch2, t = superpose(ch2, ch1)  # t is transformation, already applied to ch2

In [9]: calcRMSD(ch1, ch2)
Out[9]: 0.89840163398680684
```

Get ANM models for each chain

```
In [10]: anm1, ch1 = calcANM(ch1)

In [11]: anm2, ch2 = calcANM(ch2)

In [12]: anm1[0]
Out[12]: <Mode: 1 from ANM 1p38>
```

Let's rename these `ANM` instances, so that they print short:

```
In [13]: anm1.setTitle('1p38_anm')

In [14]: anm2.setTitle('1r39_anm')
```

This is how they print now:

```
In [15]: anm1[0]
Out[15]: <Mode: 1 from ANM 1p38_anm>

In [16]: anm2[0]
Out[16]: <Mode: 1 from ANM 1r39_anm>
```

8.2 Calculate overlap

We need Numpy in this part:

```
In [17]: from numpy import *
```

Multiplication of two `Mode` instances returns dot product of their eigenvectors. This dot product is the overlap or cosine correlation between modes.

Let's calculate overlap for slowest modes:

```
In [18]: overlap = anm1[0] * anm2[0]

In [19]: overlap
Out[19]: -0.98402119545045141
```

This shows that the overlap between these two modes is 0.98, which is not surprising since ANM modes come from structures of the *same* protein.

To compare multiple modes, convert a list of modes to a `numpy.array()`[1]:

```
In [20]: array(list(anm1[:3])) * array(list(anm2[:3]))
Out[20]: array([-0.98402119545045141, -0.98158348544972518, -0.99135781183188298], dtype=object)
```

This shows that slowest three modes are almost identical.

We could also generate a matrix of overlaps using `numpy.outer()`[2]:

```
In [21]: outer_product = outer(array(list(anm1[:3])), array(list(anm2[:3])))

In [22]: outer_product
Out[22]:
array([[-0.98402119545045141, -0.14494461667586134, -0.0021711558324487876],
       [0.14836678827912717, -0.98158348544972518, 0.080773610952805816],
       [0.010432872162820008, -0.08407811447295388, -0.99135781183188298]], dtype=object)
```

This could also be printed in a pretty table format using `printOverlapTable()`:

```
In [23]: printOverlapTable(anm1[:3], anm2[:3])
Overlap Table
                     ANM 1r39_anm
                    #1     #2     #3
ANM 1p38_anm #1   -0.98  -0.14   0.00
```

[1] http://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html#numpy.array
[2] http://docs.scipy.org/doc/numpy/reference/generated/numpy.outer.html#numpy.outer

```
ANM 1p38_anm #2    +0.15  -0.98  +0.08
ANM 1p38_anm #3    +0.01  -0.08  -0.99
```

Scaling

`Mode` instances can be scaled, but after this operation they will become `Vector` instances:

```
In [24]: anm1[0] * 10
Out[24]: <Vector: (Mode 1 from ANM 1p38_anm)*10>
```

8.3 Linear combination

It is also possible to linearly combine normal modes:

```
In [25]: anm1[0] * 3 + anm1[1] + anm1[2] * 2
Out[25]: <Vector: (((Mode 1 from ANM 1p38_anm)*3) + (Mode 2 from ANM 1p38_anm)) + ((Mode 3 from ANM 1
```

Or, we could use eigenvalues for linear combination:

```
In [26]: lincomb = anm1[0] * anm1[0].getEigval() + anm1[1] * anm1[1].getEigval()
```

It is the name of the `Vector` instance that keeps track of operations.

```
In [27]: lincomb.getTitle()
Out[27]: '((Mode 1 from ANM 1p38_anm)*0.148971269751) + ((Mode 2 from ANM 1p38_anm)*0.24904210757)'
```

8.4 Approximate a deformation vector

Let's get the deformation vector between *ch1* and *ch2*:

```
In [28]: defvec = calcDeformVector(ch1, ch2)

In [29]: abs(defvec)
Out[29]: 16.687069727870341
```

Let's see how deformation projects onto ANM modes:

```
In [30]: array(list(anm1[:3])) * defvec
Out[30]: array([-5.6086059478387353, 2.1539336595931999, -3.1370160919866956], dtype=object)
```

We can use these numbers to combine ANM modes:

```
In [31]: approximate_defvec = sum((array(list(anm1[:3])) * defvec) *
   ....:                          array(list(anm1[:3]))))
   ....:

In [32]: approximate_defvec
Out[32]: <Vector: ((-5.60860594784*(Mode 1 from ANM 1p38_anm)) + (2.15393365959*(Mode 2 from ANM 1p38
```

Let's deform 1r39 chain along this approximate deformation vector and see how RMSD changes:

```
In [33]: ch2.setCoords(ch2.getCoords() - approximate_defvec.getArrayNx3())

In [34]: calcRMSD(ch1, ch2)
Out[34]: 0.82096008703377332
```

RMSD decreases from 0.89 A to 0.82 A.

CHAPTER

NINE

DEFORMATION ANALYSIS

This example shows how to calculate the deformation vector describing the change between two structures of a protein. Two structures of the same protein in PDB format will be used. A `Vector` instance that contains the deformation vector describing the change in protein structure will be calculated. This object will be compared to `ANM` modes.

9.1 Parse structures

We start by importing everything from the ProDy package:

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

In [2]: from matplotlib.pylab import *

In [3]: ion()
```

Let's parse two p38 MAP Kinase structures: 1p38 and 1zz2

```
In [4]: reference = parsePDB('1p38')

In [5]: mobile = parsePDB('1zz2')    # this is the one we want to superimpose
```

9.2 Match chains

ProDy offers the function `matchChains()` to find matching chains in two structures easily. We use it to find the chains for which we will calculate the deformation vector:

```
In [6]: matches = matchChains(reference, mobile)
```

`matchChains()` function returns a list. If there are no matching chains, list is empty, else the list contains a tuple for each pair of matching chains.

```
In [7]: len(matches)
Out[7]: 1

In [8]: match = matches[0]
```

There is only one match in this case. First item is a subset of atoms from the first structure (*reference*). Second item is a subset of atoms from the second structure (*mobile*).

```
In [9]: ref_chain = match[0]

In [10]: mob_chain = match[1]
```

Matched atoms are returned in `AtomMap` instances. We can get information on matched subset of atoms by entering the variable name:

```
In [11]: ref_chain
Out[11]: <AtomMap: Chain A from 1p38 -> Chain A from 1zz2 from 1p38 (337 atoms)>

In [12]: mob_chain
Out[12]: <AtomMap: Chain A from 1zz2 -> Chain A from 1p38 from 1zz2 (337 atoms)>
```

Both `AtomMap` instances refer to same number of atoms, and their name suggests how they were retrieved.

In addition, we can find out the sequence identity that the matched atoms (residues) share (third item in the tuple):

```
In [13]: match[2]
Out[13]: 99.40652818991099
```

The fourth item in the tuple shows the coverage of the matching:

```
In [14]: match[3]
Out[14]: 96
```

This is the percentage of matched residues with respect to the longer chain. 1p38 chain A contains 351 resiudes, 96% of it is 337 residues, which is the number of atoms in the returned atom maps.

9.3 RMSD and superpose

We calculate the RMSD using `calcRMSD()` function:

```
In [15]: calcRMSD(ref_chain, mob_chain).round(2)
Out[15]: 72.930000000000007
```

Let's find the transformation that minimizes RMSD between these chains using `calcTransformation()` function:

```
In [16]: t = calcTransformation(mob_chain, ref_chain)
```

We apply this transformation to *mobile* structure (not to *mob_chain*, to preserve structures integrity).

```
In [17]: t.apply(mobile)
Out[17]: <AtomGroup: 1zz2 (2872 atoms)>

In [18]: calcRMSD(ref_chain, mob_chain).round(2)
Out[18]: 1.8600000000000001
```

9.4 Deformation vector

Once matching chains are identified it is straightforward to calculate the deformation vector using `calcDeformVector()`

```
In [19]: defvec = calcDeformVector(ref_chain, mob_chain)

In [20]: abs(defvec).round(3)
Out[20]: 34.195999999999998
```

To show how RMSD and deformation vector are related, we can calculate RMSD from the magnitude of the deformation vector:

```
In [21]: (abs(defvec)**2 / len(ref_chain)) ** 0.5
Out[21]: 1.86280149086955
```

Array of numbers for this deformation can be obtained as follows

```
In [22]: arr = defvec.getArray() # arr is a NumPy array

In [23]: arr.round(2)
Out[23]: array([-1.11, -0.52, -1.89, ...,  0.85, -0.18,  0.54])
```

Following yields the normalized deformation vector

```
In [24]: defvecnormed = defvec.getNormed()

In [25]: abs(defvecnormed)
Out[25]: 1.0000000000000004
```

9.5 Compare with ANM modes

Let's get ANM model for the reference chain using `calcANM()` (a shorthand function for ANM calculations):

```
In [26]: anm = calcANM(ref_chain)[0]
```

Calculate overlap between slowest ANM mode and the deformation vector

```
In [27]: (anm[0] * defvecnormed).round(2) # used normalized deformation vector
Out[27]: -0.41999999999999998
```

We can do this for a set of ANM modes (slowest 6) as follows

```
In [28]: (array(list(anm[:6])) * defvecnormed).astype(float64).round(2)
Out[28]: array([-0.42, -0.14,  0.49,  0.03, -0.17, -0.1 ])
```

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[1]) (P41 GM103712).

[1] http://mmbios.org/

BIBLIOGRAPHY

[Bahar97] Bahar I, Atilgan AR, Erman B. Direct evaluation of thermal fluctuations in protein using a single parameter harmonic potential. *Folding & Design* **1997** 2:173-181.

[Haliloglu97] Haliloglu T, Bahar I, Erman B. Gaussian dynamics of folded proteins. *Phys. Rev. Lett.* **1997** 79:3090-3093.

[Doruker00] Doruker P, Atilgan AR, Bahar I. Dynamics of proteins predicted by molecular dynamics simulations and analytical approaches: Application to a-amylase inhibitor. *Proteins* **2000** 40:512-524.

[Atilgan01] Atilgan AR, Durrell SR, Jernigan RL, Demirel MC, Keskin O, Bahar I. Anisotropy of fluctuation dynamics of proteins with an elastic network model. *Biophys. J.* **2001** 80:505-515.

[WZ05] Zheng W, Brooks BR. Probing the Local Dynamics of Nucleotide-Binding Pocket Coupled to the Global Dynamics: Myosin versus Kinesin. *Biophysical Journal* **2005** 89:167–178.