ProDy

Protein Dynamics & Sequence Analysis

Evol Tutorial

Release 1.5.1

Anindita Dutta, Ahmet Bakan

December 24, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

This tutorial shows how to obtain multiple sequence alignments (MSA) from Pfam, how to manipulate MSA files, and obtain conservation and coevolution patterns.

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:

```
There are no required files.
```

We recommend that you will follow this tutorial by typing commands in an IPython session, e.g.:

```
$ ipython
```

or with pylab environment:

```
$ ipython --pylab
```

First, we will make necessary imports from ProDy and Matplotlib packages.

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

In [2]: from pylab import *

In [3]: ion()
```

We have included these imports in every part of the tutorial, so that code copied from the online pages is complete. You do not need to repeat imports in the same Python session.

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

CHAPTER

TWO

PFAM ACCESS

The part shows how to access Pfam database. You can search protein family accession numbers and information using a sequence or PDB/UniProt identifiers. MSA files for families of interest can be retrieved in a number of formats.

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

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

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

2.1 Search Pfam

We use `searchPfam()` for searching. Valid inputs are UniProt ID, e.g. PIWI_ARCFU[1], or PDB identifier, e.g. 3luc[2] or `"3lucA"` with chain identifier.

Matching Pfam accession (one or more) as keys will map to a dictionary that contains locations (alignment start, end, evalue etc), pfam family type, accession and id.

We query Pfam using the `searchPfam()`. with a UniProt ID.

```
In [4]: matches = searchPfam('PIWI_ARCFU')
```

It is a good practice to save this record on disk, as NCBI may not respond to repeated searches for the same sequence. We can do this using Python standard library `pickle`[3] as follows:

```
In [5]: import pickle
```

Record is save using `dump()`[4] function into an open file:

```
In [6]: pickle.dump(matches, open('pfam_search_PIWI_ARCFU.pkl', 'w'))
```

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

```
In [7]: matches = pickle.load(open('pfam_search_PIWI_ARCFU.pkl'))
```

```
In [8]: matches
Out[8]:
```

[1]http://www.uniprot.org/uniprot/PIWI_ARCFU
[2]http://www.pdb.org/pdb/explore/explore.do?structureId=3luc
[3]http://docs.python.org/library/pickle.html#pickle
[4]http://docs.python.org/library/pickle.html#pickle.dump
[5]http://docs.python.org/library/pickle.html#pickle.load

```
{'PF02171': {'accession': 'PF02171',
  'id': 'Piwi',
  'locations': [{'ali_end': '405',
    'ali_start': '111',
    'bitscore': '228.70',
    'end': '406',
    'evalue': '1.1e-64',
    'hmm_end': '301',
    'hmm_start': '2',
    'start': '110'}],
  'type': 'Pfam-A'}}
```

Input can also be a protein sequence or a file containing the sequence:

```
In [9]: sequence = ('PMFIVNTNVPRASVPDGFLSELTQQLAQATGKPPQYIAVHVVPDQLMAFGGSSE'
   ...:  'PCALCSLHSIGKIGGAQNRSYSKLLCGLLAERLRISPDRVYINYYDMNAANVGWNNSTFA')
   ...:

In [10]: matches = searchPfam(sequence)

In [11]: pickle.dump(matches, open('pfam_search_sequence.pkl', 'w'))

In [12]: matches = pickle.load(open('pfam_search_sequence.pkl'))

In [13]: matches
Out[13]:
{'PF01187': {'accession': 'PF01187.13',
  'class': 'Domain',
  'id': 'MIF',
  'locations': [{'ali_end': '114',
    'ali_start': '1',
    'bitscore': '174.2',
    'end': '114',
    'evalue': '7.7e-52',
    'evidence': 'hmmer v3.0',
    'hmm_end': '114',
    'hmm_start': '1',
    'significant': '1',
    'start': '1'}],
  'type': 'Pfam-A'}}
```

Input sequence cannot have gaps and should be at least 12 characters long.

For sequence searches, we can pass additional parameters to `searchPfam()` like *search_b* which will search pfam B and *skip_a* that will not search pfamA database. Additional parameters include *ga* that uses gathering threshold instead of e-value, *evalue* cutoff can also be specified and *timeout* that can be set higher especially when searching larger sequences, default is `timeout=60` seconds.

```
In [14]: matches = searchPfam(sequence, search_b=True, evalue=2.0)
```

2.2 Retrieve MSA files

Data from Pfam database can be fetched using `fetchPfamMSA()`.

Valid inputs are Pfam ID, e.g. Piwi[6], or Pfam accession, e.g. PF02171[7] obtained from `searchPfam()`.

[6]http://pfam.sanger.ac.uk/family/Piwi
[7]http://pfam.sanger.ac.uk/family/PF02171

Alignment type can be `"full'` (default), `"seed"`, `"ncbi"` or `"metagenomics"` or `"rp15"` or `"rp35"` or `"rp55"` or `"rp75"`.

```
In [15]: fetchPfamMSA('piwi', alignment='seed')
```

A compressed file can be downloaded by setting `compressed=True`. The `format` of the MSA can be of `"selex"` (default), `"stockholm"` or `"fasta"`. This will return the path of the downloaded MSA file. The `output` name can be specified, for by default it will have `"accession/ID_alignment.format"`.

Note that in this case we passed a folder name, the downloaded file is saved in this folder, after it is created if it did not exist. Also longer timeouts are necessary for larger families. Some other parameters like `gap`, `order` or `inserts` can be set, as shown in the following example.

```
In [16]: fetchPfamMSA('PF02171', compressed=True, gaps='mixed',
   ....:  inserts='lower', order='alphabetical', format='fasta')
   ....:
```

CHAPTER

THREE

MSA FILES

This part shows how to parse, refine, filter, slice, and write MSA files.

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

In [2]: from matplotlib.pylab import *

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

Let's get Pfam MSA file for protein family that contains PIWI_ARCFU[1]:

```
In [4]: searchPfam('PIWI_ARCFU').keys()

In [5]: fetchPfamMSA('PF02171', alignment='seed')
```

3.1 Parsing MSA files

This shows how to use the `MSAFile` or `parseMSA()` to read the MSA file.

Reading using `MSAFile` yields an MSAFile object. Iterating over the object will yield an object of `Sequence` from which labels, sequence can be obtained.

```
In [6]: msafile = 'PF02171_seed.sth'

In [7]: msafobj = MSAFile(msafile)

In [8]: msafobj
Out[8]: <MSAFile: PF02171_seed (Stockholm; mode 'rt')>

In [9]: msa_seq_list = list(msafobj)

In [10]: msa_seq_list[0]
Out[10]: <Sequence: TAG76_CAEEL (length 395; 307 residues and 88 gaps)>
```

`parseMSA()` returns an `MSA` object. We can parse compressed files, but reading uncompressed files are much faster.

```
In [11]: fetchPfamMSA('PF02171', compressed=True)

In [12]: parseMSA('PF02171_full.sth.gz')
```

[1]http://www.uniprot.org/uniprot/PIWI_ARCFU

```
In [13]: fetchPfamMSA('PF02171', format='fasta')

In [14]: parseMSA('PF02171_full.fasta.gz')
```

Iterating over a file will yield sequence id, sequence, residue start and end indices:

```
In [15]: msa = MSAFile('PF02171_seed.sth')

In [16]: for seq in msa:
   ....:     seq
   ....:
```

3.2 Filtering and Slicing

This shows how to use the `MSAFile` object or `MSA` object to refine MSA using filters and slices.

3.2.1 Filtering

Any function that takes label and sequence arguments and returns a boolean value can be used for filtering the sequences. A sequence will be yielded if the function returns **True**. In the following example, sequences from organism *ARATH* are filtered:

```
In [17]: msafobj = MSAFile(msafile, filter=lambda lbl, seq: 'ARATH' in lbl)

In [18]: for seq in msafobj:
   ....:     seq.getLabel()
   ....:
```

3.2.2 Slicing

A list of integers can be used to slice sequences as follows. This enables selective parsing of the MSA file.

```
In [19]: msafobj = MSAFile(msafile, slice=list(range(10)) + list(range(374,384)))

In [20]: list(msafobj)[0]
Out[20]: <Sequence: TAG76_CAEEL (length 20; 19 residues and 1 gaps)>
```

Slicing can also be done using `MSA`. The `MSA` object offers other functionalities like querying, indexing, slicing row and columns and refinement.

3.3 MSA objects

3.3.1 Indexing

Retrieving a sequence at a given index, or by id will give an object of `Sequence`:

```
In [21]: msa = parseMSA(msafile)

In [22]: seq = msa[0]

In [23]: seq
```

```
Out[23]: <Sequence: TAG76_CAEEL (PF02171_seed[0]; length 395; 307 residues and 88 gaps)>

In [24]: str(seq)
Out[24]: 'CIIVVLQS.KNSDI.YMTVKEQSDIVHGIMSQCVLMKNVSRP........TPATCANIVLKLNMKMGGIN..SRIVADKITNKYLVDQPT
```

Retrieve a sequence by UniProt ID:

```
In [25]: msa['YQ53_CAEEL']
Out[25]: <Sequence: YQ53_CAEEL (PF02171_seed[6]; length 395; 328 residues and 67 gaps)>
```

3.3.2 Querying

You can query whether a sequence in contained in the instance using the UniProt identifier of the sequence as follows:

```
In [26]: 'YQ53_CAEEL' in msa
Out[26]: True
```

3.3.3 Slicing

Slice an MSA instance to give a new `MSA`. object :

```
In [27]: new_msa = msa[:2]

In [28]: new_msa
Out[28]: <MSA: PF02171_seed' (2 sequences, 395 residues)>
```

Slice using a list of UniProt IDs:

```
In [29]: msa[:2] == msa[['TAG76_CAEEL', 'O16720_CAEEL']]
Out[29]: True
```

Retrieve a character or a slice of a sequence:

```
In [30]: msa[0,0]
Out[30]: <Sequence: TAG76_CAEEL (length 1; 1 residues and 0 gaps)>

In [31]: msa[0,0:10]
Out[31]: <Sequence: TAG76_CAEEL (length 10; 9 residues and 1 gaps)>
```

Slice MSA rows and columns:

```
In [32]: msa[:10,20:40]
Out[32]: <MSA: PF02171_seed' (10 sequences, 20 residues)>
```

3.4 Merging MSAs

`mergeMSA()` can be used to merge two or more MSAs. Based on their labels only those sequences that appear in both MSAs are retained, and concatenated horizontally to give a joint or merged MSA. This can be useful while evaluating covariance patterns for proteins with multiple domains or protein-protein interactions. The example shows merging for the multi-domain receptor 3KG2[2] containing pfam domains

[2]http://www.pdb.org/pdb/explore/explore.do?structureId=3KG2

PF01094[3] and PF00497[4].

```
In [33]: fetchPfamMSA('PF00017', alignment='seed')

In [34]: fetchPfamMSA('PF07714', alignment='seed')
```

Let's parse and merge the two files:

```
In [35]: msa1 = parseMSA('PF00017_seed.sth')

In [36]: msa1
Out[36]: <MSA: PF00017_seed (58 sequences, 109 residues)>

In [37]: msa2 = parseMSA('PF07714_seed.sth')

In [38]: msa2
Out[38]: <MSA: PF07714_seed (145 sequences, 484 residues)>

In [39]: merged = mergeMSA(msa1, msa2)

In [40]: merged
Out[40]: <MSA: PF00017_seed + PF07714_seed (14 sequences, 593 residues)>
```

Merged MSA contains 14 sequences.

3.5 Writing MSAs

`writeMSA()` can be used to write MSA. It takes filename as input which should contain appropriate extension that can be `".slx"` or `".sth"` or `".fasta"` or format should be specified as `"SELEX"`, `"Stockholm"` or `"FASTA"`. Input MSA should be `MSAFile` or `MSA` object. Filename can contain `".gz"` extension, in which case a compressed file will be written.

```
In [41]: writeMSA('sliced_MSA.gz', msa, format='SELEX')
Out[41]: 'sliced_MSA.gz'

In [42]: writeMSA('sliced_MSA.fasta', msafobj)
Out[42]: 'sliced_MSA.fasta'
```

`writeMSA()` returns the name of the MSA file that is written.

[3] http://pfam.sanger.ac.uk/family/PF01094
[4] http://pfam.sanger.ac.uk/family/PF00497

CHAPTER

FOUR

EVOLUTION ANALYSIS

This part follows from *MSA Files* (page 5). The aim of this part is to show how to calculate residue conservation and coevolution properties based on multiple sequence alignments (MSAs). MSA

First, we import everything from the ProDy package.

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

In [2]: from pylab import *

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

4.1 Get MSA data

Let's download full MSA file for protein family RnaseA[1]. We can do this by specifying the PDB ID of a protein in this family.

```
In [4]: searchPfam('1K2A').keys()

In [5]: fetchPfamMSA('PF00074')
```

Let's parse the downloaded file:

```
In [6]: msa = parseMSA('PF00074_full.sth')
```

4.2 Refine MSA

Here, we refine the MSA to decrease the number of gaps. We will remove any columns in the alignment for which there is a gap in the specified PDB file, and then remove any rows that have more than 20% gaps. `refineMSA()` does all of this and returns an `MSA` object.

```
In [7]: msa_refine = refineMSA(msa, label='RNAS2_HUMAN', rowocc=0.8, seqid=0.98)

In [8]: msa_refine
Out[8]: <MSA: PF00074_full refined (label=RNAS2_HUMAN, rowocc>=0.8, seqid>=0.98) (533 sequences, 128
```

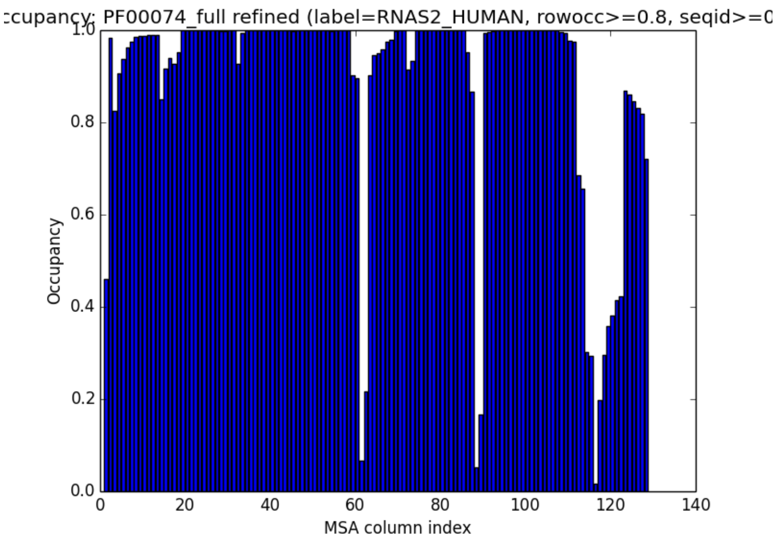
MSA is refined based on the sequence of RNAS2_HUMAN[2], corresponding to 1K2A[3].

[1]http://pfam.sanger.ac.uk/family/RnaseA
[2]http://www.uniprot.org/uniprot/RNAS2_HUMAN
[3]http://www.pdb.org/pdb/explore/explore.do?structureId=1K2A

4.3 Occupancy calculation

Evol plotting functions are prefixed with `show`. We can plot the occupancy for each column to see if there are any positions in the MSA that have a lot of gaps. We use the function `showMSAOccupancy()` that uses `calcMSAOccupancy()` to calculate occupancy for MSA.

```
In [9]: showMSAOccupancy(msa_refine, occ='res');
```



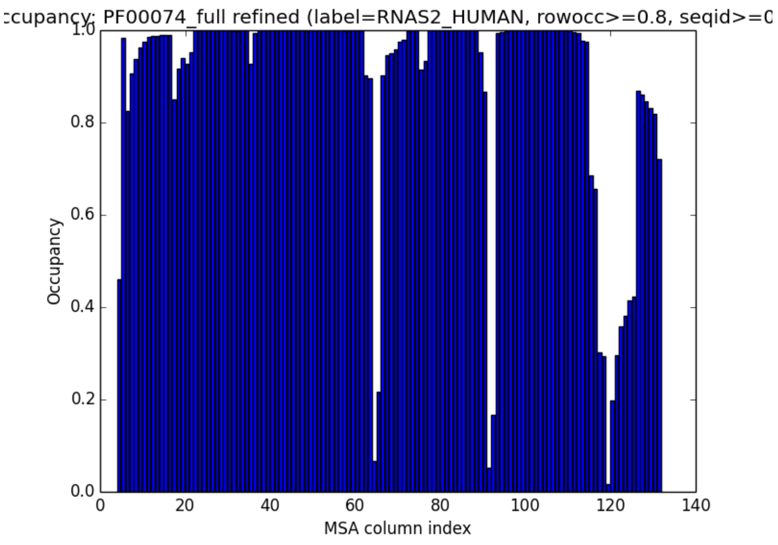
Let's find the minimum:

```
In [10]: calcMSAOccupancy(msa_refine, occ='res').min();
```

We can also specify indices based on the PDB.

```
In [11]: indices = list(range(4,132))
```

```
In [12]: showMSAOccupancy(msa_refine, occ='res', indices=indices);
```



Further refining the MSA to remove positions that have low occupancy will change the start and end positions of the labels in the MSA. This is not corrected automatically on refinement. We can also plot occupancy based on rows for the sequences in the MSA.

4.4 Entropy Calculation

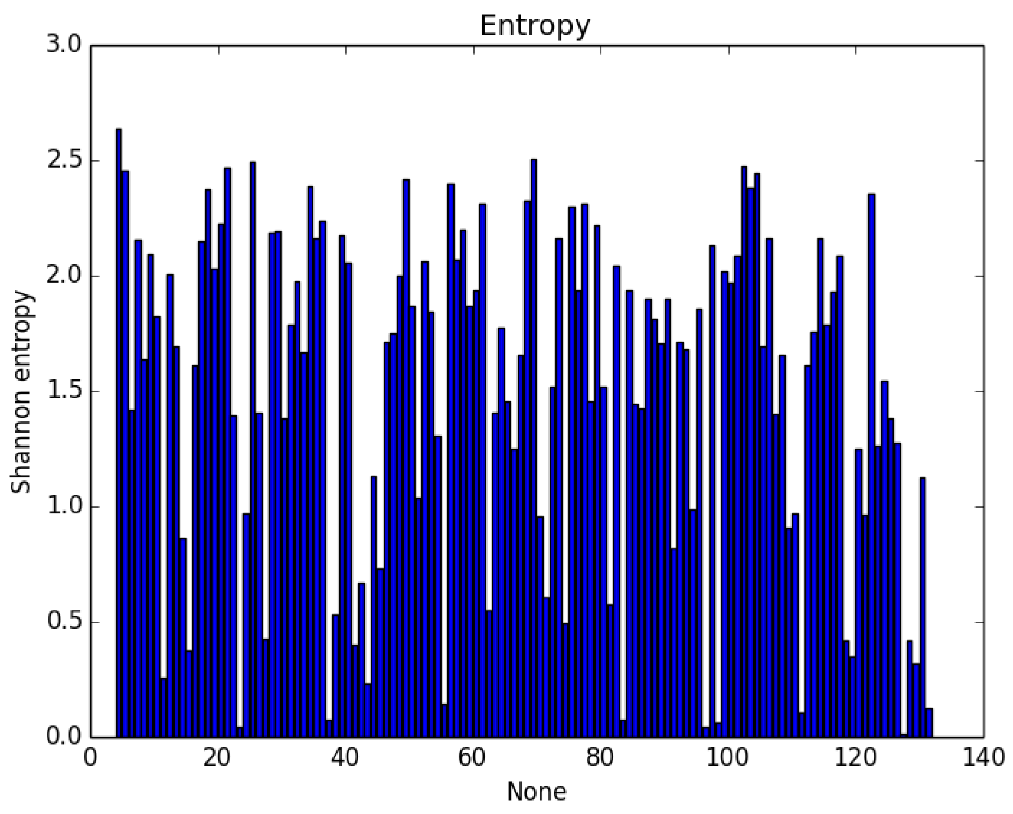
Here, we show how to calculate and plot Shannon Entropy. Entropy for each position in the MSA is calculated using `calcShannonEntropy()`. It takes `MSA` object or a numpy 2D array containg MSA as input and returns a 1D numpy array.

```
In [13]: entropy = calcShannonEntropy(msa_refine)

In [14]: entropy
Out[14]:
array([ 2.63800919,  2.45450938,  1.42041785,  2.15639571,  1.63907879,
        2.09393186,  1.82330678,  0.25908259,  2.00605129,  1.69565943,
        0.86225618,  0.37467677,  1.61399529,  2.152871  ,  2.37773811,
        2.03304806,  2.22500579,  2.47038886,  1.39355386,  0.04095142,
        0.96975454,  2.49565623,  1.40664469,  0.42308606,  2.1870084 ,
        2.1912902 ,  1.37821019,  1.78935302,  1.9773083 ,  1.66724354,
        2.38991496,  2.16435735,  2.23861691,  0.07666912,  0.53090604,
        2.17685992,  2.05601916,  0.39718599,  0.66921337,  0.22814177,
        1.13067592,  0.73408808,  1.7134237 ,  1.75080943,  1.99947939,
        2.42045936,  1.86751707,  1.03884166,  2.06000272,  1.84548923,
        1.30756389,  0.14081204,  2.40116039,  2.06824591,  2.20118327,
        1.86602952,  1.93722842,  2.31294099,  0.55243632,  1.40868441,
        1.77233537,  1.45889218,  1.25211336,  1.65401333,  2.32637287,
        2.50318713,  0.95322287,  0.6088434 ,  1.516984  ,  2.16086557,
        0.49502443,  2.2973572 ,  1.93483556,  2.31261882,  1.4581374 ,
        2.2167319 ,  1.51758558,  0.5770289 ,  2.04402151,  0.07524241,
        1.93959113,  1.44071683,  1.42274434,  1.9023148 ,  1.81370905,
        1.70753063,  1.89902445,  0.81650334,  1.71242241,  1.68083773,
        0.98479639,  1.8581422 ,  0.04095142,  2.13080516,  0.06205112,
        2.01730025,  1.96933716,  2.09058798,  2.47214553,  2.38254327,
        2.44521881,  1.69081665,  2.15967129,  1.40280104,  1.65347276,
        0.90701065,  0.96741081,  0.1092195 ,  1.61548777,  1.75714973,
        2.15960565,  1.78756768,  1.92852948,  2.08465504,  0.42115912,
        0.3488321 ,  1.25056073,  0.9596198 ,  2.35731046,  1.26015433,
        1.54531754,  1.3825499 ,  1.27762709,  0.01552932,  0.41832099,
        0.32073906,  1.12347468,  0.12739974])
```

entropy is a 1D Numpy array. Plotting is done using `showShannonEntropy()`.

```
In [15]: showShannonEntropy(entropy, indices);
```



4.5 Mutual Information

We can calculate mutual information between the positions of the MSA using `buildMutinfoMatrix()` which also takes an `MSA` object or a numpy 2D array containing MSA as input.

```
In [16]: mutinfo = buildMutinfoMatrix(msa_refine)

In [17]: mutinfo
Out[17]:
array([[ 0.        ,  0.80235226,  0.55114673, ...,  0.16578316,
         0.31217035,  0.10893036],
       [ 0.80235226,  0.        ,  0.59997967, ...,  0.13780672,
         0.40552489,  0.10436392],
       [ 0.55114673,  0.59997967,  0.        , ...,  0.16524   ,
         0.40110279,  0.13758264],
       ...,
       [ 0.16578316,  0.13780672,  0.16524   , ...,  0.        ,
         0.57198283,  0.35523316],
       [ 0.31217035,  0.40552489,  0.40110279, ...,  0.57198283,
         0.        ,  0.39735979],
       [ 0.10893036,  0.10436392,  0.13758264, ...,  0.35523316,
         0.39735979,  0.        ]])
```

Result is a 2D Numpy array.

We can also apply normalization using `applyMutinfoNorm()` and correction using applyMutinfoCorr() to the mutual information matrix based on references [Martin05] (page 19) and [Dunn08] (page 19), respectively.

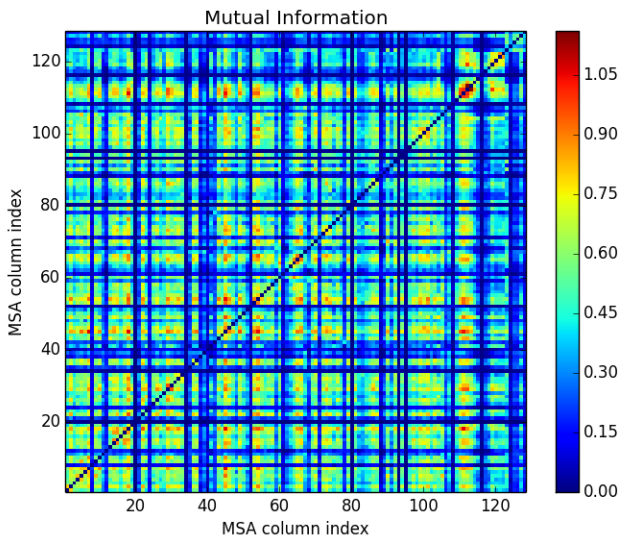
```
In [18]: mutinfo_norm = applyMutinfoNorm(mutinfo, entropy, norm='minent')
```

```
In [19]: mutinfo_corr = applyMutinfoCorr(mutinfo, corr='apc')
```

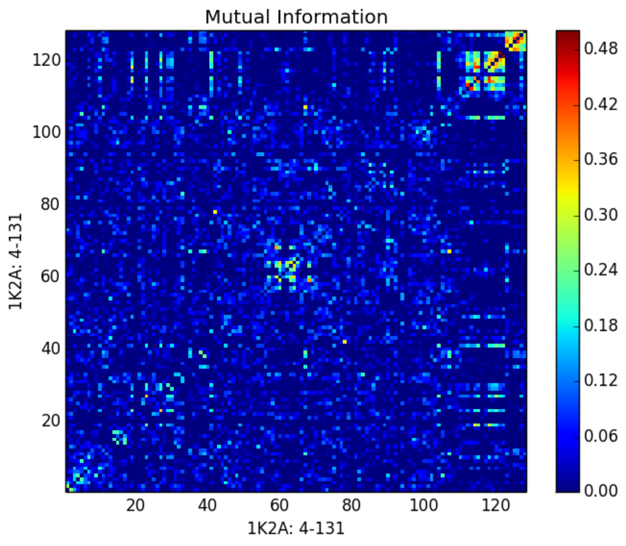
Note that by default norm="sument" normalization is applied in applyMutinfoNorm and corr="prod" is applied in applyMutinfoCorr.

Now we plot the mutual information matrices that we obtained above and see the effects of different corrections and normalizations.

```
In [20]: showMutinfoMatrix(mutinfo);
```



```
In [21]: showMutinfoMatrix(mutinfo_corr, clim=[0, mutinfo_corr.max()],
   ....:     xlabel='1K2A: 4-131');
   ....:
```



4.6 Output Results

Here we show how to write the mutual information and entropy arrays to file. We use the `writeArray()` to write Numpy array data.

```
In [22]: writeArray('1K2A_MI.txt', mutinfo)
Out[22]: '1K2A_MI.txt'
```

This can be later loaded using `parseArray()`.

4.7 Rank-ordering

Further analysis can also be done by rank ordering the matrix and analyzing the pairs with highest mutual information or the most co-evolving residues. This is done using `calcRankorder()`. A z-score normalization can also be applied to select coevolving pairs based on a z score cutoff.

```
In [23]: rank_row, rank_col, zscore_sort = calcRankorder(mutinfo, zscore=True)

In [24]: asarray(indices)[rank_row[:5]]
Out[24]: array([128, 129, 130, 130, 129])

In [25]: asarray(indices)[rank_col[:5]]
Out[25]: array([127, 127, 129, 127, 128])

In [26]: zscore_sort[:5]
Out[26]: array([ 4.86370837,  4.35821895,  4.04974775,  4.03299096,  3.42713102])
```

CHAPTER

FIVE

SEQUENCE-STRUCTURE COMPARISON

The part shows how to compare sequence conservation properties with structural mobility obtained from Gaussian network model (GNM) calculations.

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

In [2]: from matplotlib.pylab import *

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

5.1 Entropy Calculation

First, we retrieve MSA for protein for protein family PF00074[1]:

```
In [4]: fetchPfamMSA('PF00074')
```

We parse the MSA file:

```
In [5]: msa = parseMSA('PF00074_full.sth')
```

Then, we refine it using `refineMSA()` based on the sequence of RNAS1_BOVIN[2]:

```
In [6]: msa_refine = refineMSA(msa, label='RNAS1_BOVIN', rowocc=0.8, seqid=0.98)
```

We calculate the entropy for the refined MSA:

```
In [7]: entropy = calcShannonEntropy(msa_refine)
```

5.2 Mobility Calculation

Next, we obtain residue fluctuations or mobility for protein member of the above family. We will use chain B of 2W5I[3].

[1]http://pfam.sanger.ac.uk/family/PF00074
[2]http://www.uniprot.org/uniprot/RNAS1_BOVIN
[3]http://www.pdb.org/pdb/explore/explore.do?structureId=2W5I

```
In [8]: pdb = parsePDB('2W5I', chain='B')

In [9]: chB_ca = pdb.select('protein and name CA and resid 1 to 121')
```

We perform GNM as follows:

```
In [10]: gnm = GNM('2W5I')

In [11]: gnm.buildKirchhoff(chB_ca)

In [12]: gnm.calcModes(n_modes=None)  # calculate all modes
```

Now, let's obtain residue mobility using slowest mode, slowest 8 modes, and all modes:

```
In [13]: mobility_1 = calcSqFlucts(gnm[0])

In [14]: mobility_1to8 = calcSqFlucts(gnm[:8])

In [15]: mobility_all = calcSqFlucts(gnm[:])
```

See *Gaussian Network Model (GNM)*[4] for details.

5.3 Comparison of mobility and conservation

We use the above data to compare structural mobility and degree of conservation. We can calculate a correlation coefficient between the two quantities:

```
In [16]: result = corrcoef(mobility_all, entropy)

In [17]: result.round(3)[0,1]
Out[17]: 0.39800000000000002
```

We can plot the two curves simultaneously to visualize the correlation. We have to scale the values of mobility to display them in the same plot.

5.3.1 Plotting

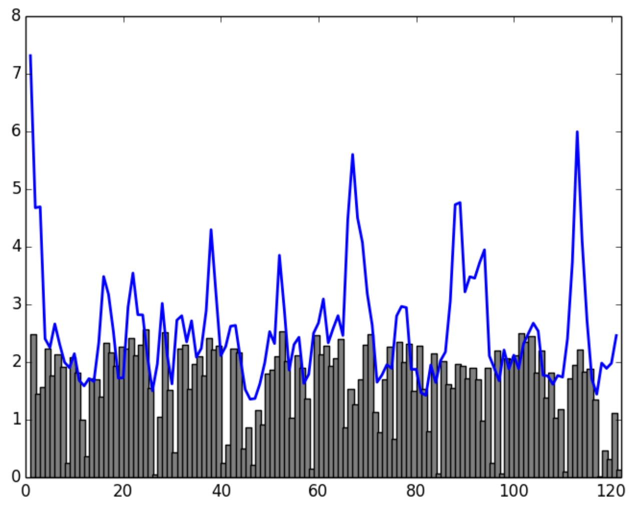
```
In [18]: indices = range(1,122)

In [19]: bar(indices, entropy, width=1.2, color='grey', hold='True');

In [20]: xlim(min(indices)-1, max(indices)+1);

In [21]: plot(indices, mobility_all*(max(entropy)/mean(mobility_all)), color='b',
   ....: linewidth=2);
   ....:
```

[4]http://prody.csb.pitt.edu/tutorials/enm_analysis/gnm.html#gnm



5.4 Writing PDB files

We can also write PDB with b-factor column replaced by entropy and mobility values respectively. We can then load the PDB structure in VMD or PyMol to see the distribution of entropy and mobility on the structure.

```
In [22]: selprot = pdb.select('protein and resid 1 to 121')

In [23]: resindex = selprot.getResindices()

In [24]: index = unique(resindex)

In [25]: count = 0

In [26]: entropy_prot = []

In [27]: mobility_prot = []

In [28]: for ind in index:
   ....:     while(count < len(resindex)):
   ....:         if(ind == resindex[count]):
   ....:             entropy_prot.append(entropy[ind])
   ....:             mobility_prot.append(mobility_all[ind]*100)
   ....:         count = count + 1
   ....:

In [29]: selprot.setBetas(entropy_prot)

In [30]: writePDB('2W5I_entropy.pdb', selprot)
Out[30]: '2W5I_entropy.pdb'

In [31]: selprot.setBetas(mobility_prot)

In [32]: writePDB('2W5I_mobility.pdb', selprot)
Out[32]: '2W5I_mobility.pdb'
```

CHAPTER

SIX

EVOL APPLICATION

Evol Applications have similar fuctionality as the python API. We can `search` Pfam, `fetch` from Pfam and also `refine MSA`, `merge` two or more MSA and calculate `conservation` and `coevolution` properties and also `rankorder` results from mutual information to get top-ranking pairs.

All `evol` functions and their options can be obtained using the -h option. We should be in /prody/scripts directory to run the following commands:

```
python evol -h
python evol search -h
python evol search 2W5IB
python evol fetch PF00074
```

Using the above we can search and fetch MSA. Next we can refine the MSA:

```
python evol refine -h
python evol refine PF00074_full.slx -l RNAS1_BOVIN -s 0.98 -r 0.8
```

Next we can calculate conservation using shannon entropy and coevolution using mutual information with correction and also save the plots.:

```
python evol conserv PF00074_full_refined.slx -S
python evol coevol PF00074_full_refined.slx -S -F png -c apc -cmin 0.0
```

We can rank order the residues with highest covariance and apply filters like reporting only those pairs that are at a separation of at least 5 residues sequentially or are 15 Ang apart in structure. The residues may be numbered based on PDB:

```
python evol rankorder -h
python evol rankorder PF00074_full_refined_mutinfo_corr_apc.txt -q 5 -p
2W5IBI_1-121.pdb
python evol rankorder PF00074_full_refined_mutinfo_corr_apc.txt -u -t 15 -p 2W5IB_1-121.pdb
```

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

[Martin05] Martin LC, Gloor GB, Dunn SD, Wahl LM. Using information theory to search for co-evolving residues in proteins. *Bioinformatics* **2005** 21(22):4116-4124.

[Dunn08] Dunn SD, Wahl LM, Gloor GB. Mutual information without the influence of phylogeny or entropy dramatically improves residue contact prediction. *Bioinformatics* **2008** 24(3):333-340.