ProDy

Protein Dynamics & Sequence Analysis

Evol Tutorial

Release

Anindita Dutta, Ahmet Bakan, Cihan Kaya

December 04, 2024

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_** and **Matplotlib_** required.

1.2 Recommended Programs

IPython_ 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.

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. **:uniprot:'PIWI_ARCFU'**, or PDB identifier, e.g. **:pdb:'3luc'** 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.

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`[1] as follows:

```
In [4]: import pickle
```

Record is save using `dump()`[2] function into an open file:

Then, it can be loaded using `load()`[3] function:

```
In [5]: matches = pickle.load(open('pfam_search_PIWI_ARCFU.pkl'))

In [6]: matches
Out[6]:
{'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',
```

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

```
    'start': '110'}],
  'type': 'Pfam-A'}}
```

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.

2.2 Retrieve MSA files

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

Valid inputs are Pfam ID, e.g. **:pfam:‘Piwi‘**, or Pfam accession, e.g. **:pfam:‘PF02171‘** obtained from `searchPfam()`.

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

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.

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 the Pfam MSA file for the protein family that contains **:uniprot:'PIWI_ARCFU'**:

3.1 Parsing MSA files

This shows how to use `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. Like any file object, you can over iterate over a `MSAFile` object once.

```
In [4]: msafile = 'PF02171_seed.sth'

In [5]: msafobj = MSAFile(msafile)

In [6]: msafobj
Out[6]: <MSAFile: PF02171_seed (Stockholm; mode 'rt')>

In [7]: msa_seq_list = list(msafobj)

In [8]: msa_seq_list[0]
Out[8]: <Sequence: TAG76_CAEEL (length 395; 307 residues and 88 gaps)>
```

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

```
In [9]: fetchPfamMSA('PF02171', compressed=True)
Out[9]: 'PF02171_full.sth.gz'

In [10]: parseMSA('PF02171_full.sth.gz')
Out[10]: <MSA: PF02171_full (2067 sequences, 1392 residues)>

In [11]: fetchPfamMSA('PF02171', format='fasta')
Out[11]: 'PF02171_full.fasta.gz'

In [12]: parseMSA('PF02171_full.fasta.gz')
Out[12]: <MSA: PF02171_full (2067 sequences, 1392 residues)>
```

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

```
In [13]: msa = MSAFile('PF02171_seed.sth')

In [14]: seq_list = [seq for seq in msa]

In [15]: seq_list
Out[15]:
[<Sequence: TAG76_CAEEL (length 395; 307 residues and 88 gaps)>,
 <Sequence: O16720_CAEEL (length 395; 302 residues and 93 gaps)>,
 <Sequence: AGO10_ARATH (length 395; 322 residues and 73 gaps)>,
 <Sequence: AGO1_SCHPO (length 395; 300 residues and 95 gaps)>,
 <Sequence: AGO6_ARATH (length 395; 311 residues and 84 gaps)>,
 <Sequence: AGO4_ARATH (length 395; 309 residues and 86 gaps)>,
 <Sequence: YQ53_CAEEL (length 395; 328 residues and 67 gaps)>,
 <Sequence: Q21691_CAEEL (length 395; 329 residues and 66 gaps)>,
 <Sequence: O62275_CAEEL (length 395; 331 residues and 64 gaps)>,
 <Sequence: O16386_CAEEL (length 395; 300 residues and 95 gaps)>,
 <Sequence: Q21495_CAEEL (length 395; 285 residues and 110 gaps)>,
 <Sequence: O76922_DROME (length 395; 298 residues and 97 gaps)>,
 <Sequence: PIWI_DROME (length 395; 292 residues and 103 gaps)>,
 <Sequence: Q17567_CAEEL (length 395; 312 residues and 83 gaps)>,
 <Sequence: PIWL1_HUMAN (length 395; 293 residues and 102 gaps)>,
 <Sequence: PIWI_ARCFU (length 395; 297 residues and 98 gaps)>,
 <Sequence: Y1321_METJA (length 395; 274 residues and 121 gaps)>,
 <Sequence: O67434_AQUAE (length 395; 276 residues and 119 gaps)>]
```

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

In [17]: seq_list2 = [seq for seq in msafobj]

In [18]: seq_list2
Out[18]:
[<Sequence: AGO10_ARATH (length 395; 322 residues and 73 gaps)>,
 <Sequence: AGO6_ARATH (length 395; 311 residues and 84 gaps)>,
 <Sequence: AGO4_ARATH (length 395; 309 residues and 86 gaps)>]
```

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 label will give an object of `Sequence`. Here's an example using an index.

```
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..SRIVADKITNKYLVDQP
```

Here we 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[['TAG76_CAEEL', 'O16720_CAEEL']]
Out[29]: <MSA: PF02171_seed (2 sequences, 395 residues)>
```

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 **:pdb:'3KG2'** containing pfam domains **:pfam:'PF01094'** and **:pfam:'PF00497'**.

```
In [33]: fetchPfamMSA('PF01094', alignment='seed')
Out[33]: 'PF01094_seed.sth'
```

```
In [34]: fetchPfamMSA('PF00497', alignment='seed')
Out[34]: 'PF00497_seed.sth'
```

Let's parse and merge the two files:

```
In [35]: msa1 = parseMSA('PF01094_seed.sth')

In [36]: msa1
Out[36]: <MSA: PF01094_seed (67 sequences, 686 residues)>

In [37]: msa2 = parseMSA('PF00497_seed.sth')

In [38]: msa2
Out[38]: <MSA: PF00497_seed (230 sequences, 503 residues)>

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

In [40]: merged
```

The merged MSA contains 4889 sequences with common labels.

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.

CHAPTER

FOUR

EVOLUTION ANALYSIS

This part follows from *MSA Files*. 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 **:pfam:'RnaseA'**. We can do this by specifying the PDB ID of a protein in this family.

```
In [4]: searchPfam('1K2A').keys()
Out[4]: ['PF00074']
```

```
In [5]: fetchPfamMSA('PF00074')
Out[5]: 'PF00074_full.sth'
```

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.

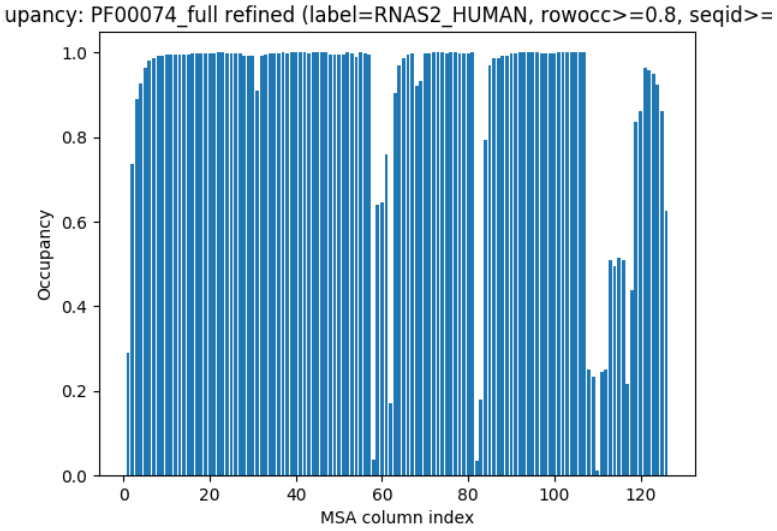
```
In [7]: msa_refine = refineMSA(msa, label='RNAS2_HUMAN', rowocc=0.8, seqid=0.98)

In [8]: msa_refine
Out[8]: <MSA: PF00074_full refined (label=RNAS2_HUMAN, rowocc>=0.8, seqid>=0.98) (698 sequences, 126
```

MSA is refined based on the sequence of **:uniprot:'RNAS2_HUMAN'**, corresponding to **:pdb:'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');
```

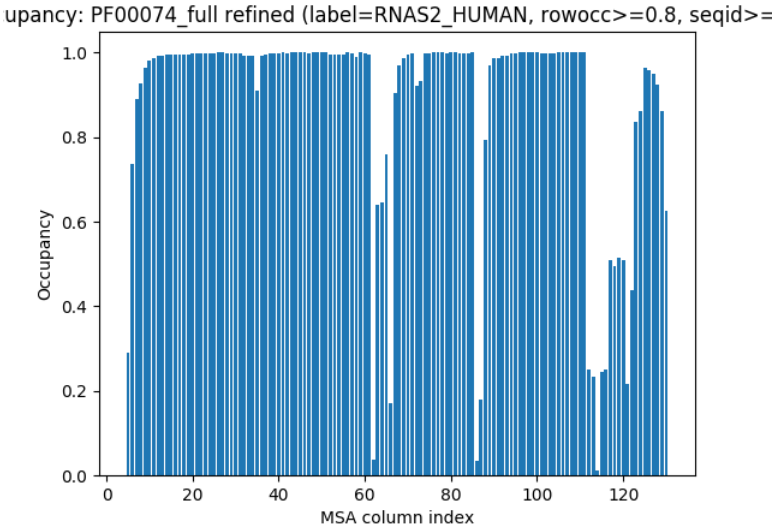


Let's find the minimum:

```
In [10]: calcMSAOccupancy(msa_refine, occ='res').min()
Out[10]: 0.011461318051575931
```

We can also specify indices based on the PDB.

```
In [11]: indices = list(range(5,131))

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.

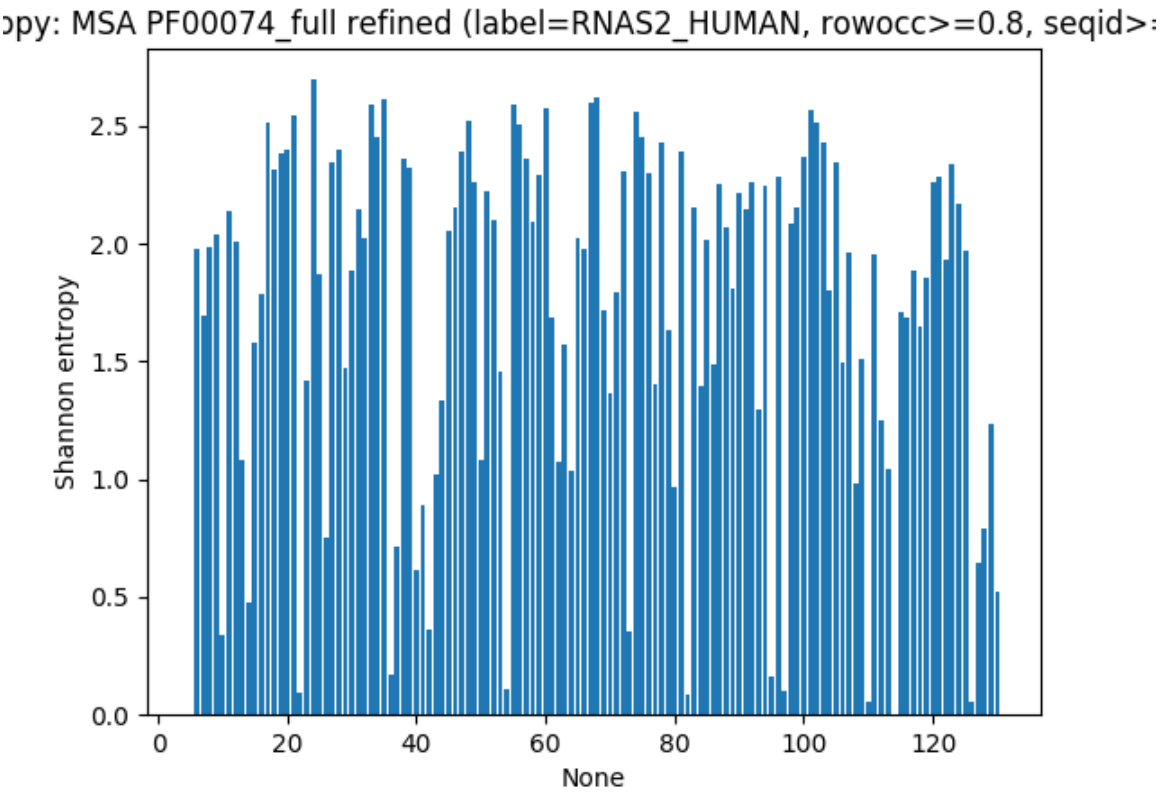
4.4 Entropy Calculation

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([-0.        ,  1.97536341,  1.6976014 ,  1.98686002,  2.04097927,
        0.33881322,  2.13977922,  2.00661895,  1.08055377,  0.47729866,
        1.58125405,  1.78890686,  2.51747686,  2.31397501,  2.38803313,
        2.40126663,  2.54295735,  0.09338611,  1.41579988,  2.69688922,
        1.8686074 ,  0.75606438,  2.34383631,  2.39720547,  1.47431149,
        1.88334663,  2.14950549,  2.02528004,  2.58853012,  2.45108208,
        2.61723915,  0.16813412,  0.71292546,  2.36093797,  2.32632049,
        0.61799546,  0.89021762,  0.36043225,  1.01850997,  1.33382837,
        2.05607945,  2.15459689,  2.3895889 ,  2.52272713,  2.26277362,
        1.07883485,  2.22209291,  2.10332436,  1.46075956,  0.10580037,
        2.5880002 ,  2.50810725,  2.35814643,  2.09300504,  2.29386782,
        2.5724491 ,  1.68877804,  1.07047455,  1.57433048,  1.03752672,
        2.0206458 ,  1.97820357,  2.60225   ,  2.62010838,  1.71632538,
        1.36403271,  1.79131851,  2.30781167,  0.35730727,  2.55844119,
        2.45589515,  2.30325997,  1.40702068,  2.42986786,  1.63284318,
        0.96868799,  2.39301813,  0.08317819,  2.15806993,  1.39390954,
        2.01888973,  1.48477641,  2.25039015,  2.071024  ,  1.80876533,
        2.21861345,  2.1495278 ,  2.26531537,  1.29842729,  2.2432391 ,
        0.16590041,  2.28183392,  0.10492724,  2.08455959,  2.15120138,
        2.36710596,  2.56442544,  2.51541107,  2.42719479,  1.80265513,
        2.34569871,  1.4964265 ,  1.96057778,  0.98329793,  1.51199785,
        0.05404462,  1.95389007,  1.24837514,  1.04713865, -0.        ,
        1.71085363,  1.68585224,  1.88812929,  1.64865814,  1.8563798 ,
        2.26222015,  2.28490399,  1.93257978,  2.34153018,  2.17263465,
        1.97322381,  0.05898735,  0.64841109,  0.79058895,  1.23695064,
        0.52622679])
```

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

```
In [15]: showShannonEntropy(msa_refine, 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.        , 0.20631824, 0.16401097, ..., 0.06095202, 0.14566864,
        0.05380968],
       [0.20631824, 0.        , 0.60253758, ..., 0.22563382, 0.22440521,
        0.18667669],
       [0.16401097, 0.60253758, 0.        , ..., 0.35108633, 0.37773013,
        0.24761374],
       ...,
       [0.06095202, 0.22563382, 0.35108633, ..., 0.        , 0.4087536 ,
        0.22060682],
       [0.14566864, 0.22440521, 0.37773013, ..., 0.4087536 , 0.        ,
        0.27725374],
       [0.05380968, 0.18667669, 0.24761374, ..., 0.22060682, 0.27725374,
        0.        ]])
```

Result is a 2D NumPy array.

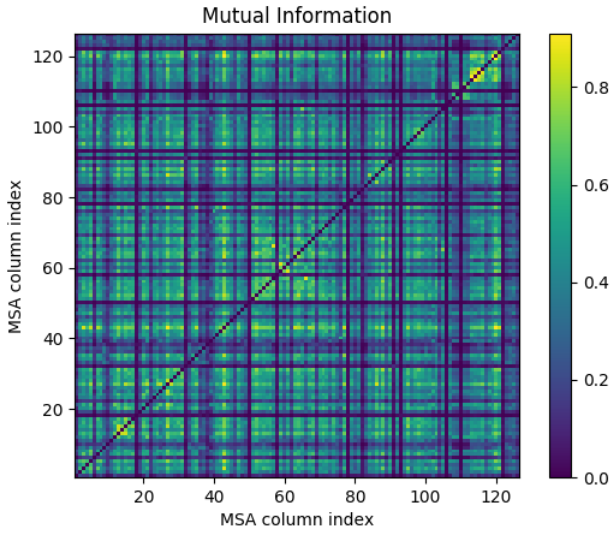
We can also apply normalization using `applyMutinfoNorm()` and correction using `applyMutinfoCorr()` to the mutual information matrix based on references *[Martin05]* and *[Dunn08]*, respectively.

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

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

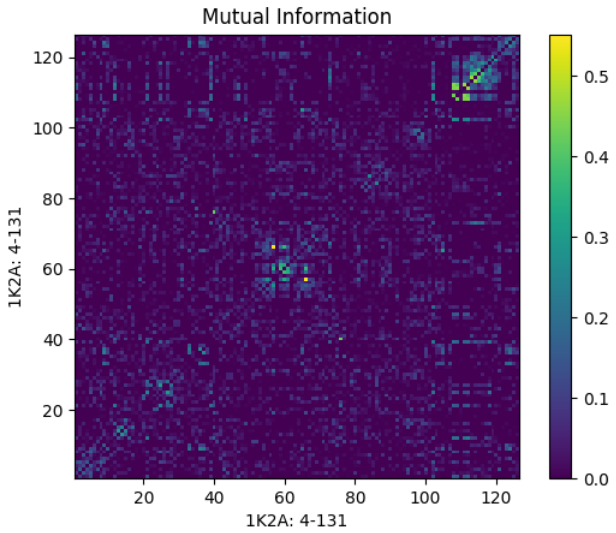
Note that by default `norm="sument"` normalization is applied in `applyMutinfoNorm` and `corr="prod"` is applied in `applyMutinfoCorr`.

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([127,  31,  31, 128, 115])

In [25]: asarray(indices)[rank_col[:5]]
Out[25]: array([126,  26,  29, 126, 113])

In [26]: zscore_sort[:5]
Out[26]: array([4.50792926, 3.55425262, 3.42345689, 3.27298657, 3.21999586])
```

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 **:pfam:'PF00074'**:

```
In [4]: fetchPfamMSA('PF00074')
Out[4]: 'PF00074_full.sth'
```

We parse the MSA file:

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

Then, we refine it using `refineMSA()` based on the sequence of **:uniprot:'RNAS1_BOVIN'**:

```
In [6]: msa_refine = refineMSA(msa, label='RNAS1_BOVIN', 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 **:pdb:'2W5I'**.

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

We can use the function `alignSequenceToMSA()` to identify the part of the PDB file that matches with the MSA. In cases where this fails, a label keyword argument can be provided to the function as well.

```
In [9]: aln, idx_1, idx_2 = alignSequenceToMSA(pdb, msa_refine, label='RNAS1_BOVIN')

In [10]: showAlignment(aln, indices=[idx_1, idx_2])

                                        19        29        39        49        59
```

```
2W5IB                   KETAAAKFERQHMDSSTSAASSSNYCNQMMKSRNLTKDRCKPVNTFVHESLADVQAVCSQ

                                        18        28        38        48        58
RNAS1_BOVIN             --TAAAKFERQHMDSSTSAASSSNYCNQMMKSRNLTKDRCKPVNTFVHESLADVQAVCSQ


                                69        79        89        99       109       119
2W5IB                   KNVACKNGQTNCYQSYSTMSITDCRETGSSKYPNCAYKTTQANKHIIVACEGNPYVPVHF

                                68        78        88        98       108       118
RNAS1_BOVIN             KNVACKNGQTNCYQSYSTMSITDCRETGSSKYPNCAYKTTQANKHIIVACEGNPYVPVHF



2W5IB                   DASV


RNAS1_BOVIN             D---
```

This tells us that the first two residues are missing as are the last three, ending the sequence at residue 121. Hence, we make a selection accordingly:

```
In [11]: chB = pdb.select('resnum 3 to 121')
```

We can see from the sequence that this gives us the right portion:

```
In [12]: chB.ca.getSequence()
Out[12]: 'TAAAKFERQHMDSSTSAASSSNYCNQMMKSRNLTKDRCKPVNTFVHESLADVQAVCSQKNVACKNGQTNCYQSYSTMSITDCRETGSSKYP
```

We write this selection to a PDB file for use later, e.g. with evol apps.

```
In [13]: writePDB('2W5IB_3-121.pdb', chB)
Out[13]: '2W5IB_3-121.pdb'
```

We perform GNM as follows:

```
In [14]: gnm = GNM('2W5I')

In [15]: gnm.buildKirchhoff(chB.ca)

In [16]: gnm.calcModes(n_modes=None)   # calculate all modes
```

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

```
In [17]: mobility_1 = calcSqFlucts(gnm[0])

In [18]: mobility_1to8 = calcSqFlucts(gnm[:8])

In [19]: 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:

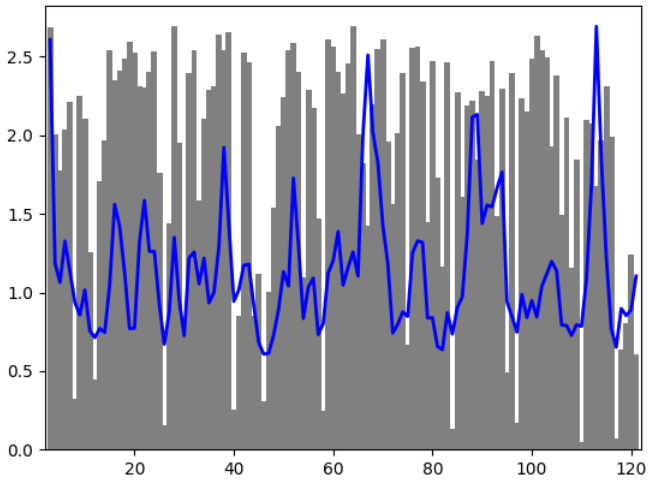
[4] http://www.bahargroup.org/prody/tutorials/enm_analysis/gnm.html#gnm

```
In [20]: result = corrcoef(mobility_all, entropy)

In [21]: result.round(3)[0,1]
Out[21]: 0.396
```

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 [22]: indices = chB.ca.getResnums()

In [23]: bar(indices, entropy, width=1.2, color='grey');

In [24]: xlim(min(indices)-1, max(indices)+1);

In [25]: plot(indices, mobility_all*(max(entropy)/max(mobility_all)), color='b',
   ....: linewidth=2);
   ....:
```



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 [26]: selprot = chB.copy()

In [27]: resindex = selprot.getResindices()

In [28]: entropy_prot = [entropy[ind] for ind in resindex]

In [29]: mobility_prot = [mobility_all[ind]*10 for ind in resindex]

In [30]: selprot.setBetas(entropy_prot)

In [31]: writePDB('2W5I_entropy.pdb', selprot)
```

```
Out[31]: '2W5I_entropy.pdb'

In [32]: selprot.setBetas(mobility_prot)

In [33]: writePDB('2W5I_mobility.pdb', selprot)
Out[33]: '2W5I_mobility.pdb'
```

We can see on the structure just as we could in the bar graph that there is some correlation with highly conserved (low entropy) regions having low mobility and high entropy regions have higher mobility.

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:

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

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

```
evol refine -h
evol refine PF00074_full.sth -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.:

```
evol conserv PF00074_full_refined.sth -S
evol coevol PF00074_full_refined.sth -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 a PDB file, such as the one we made earlier:

```
evol rankorder -h
evol rankorder PF00074_full_refined_mutinfo_corr_apc.txt -q 5 -p 2W5IB_3-121.pdb
evol rankorder PF00074_full_refined_mutinfo_corr_apc.txt -u -t 15 -p 2W5IB_3-121.pdb
```

We can also provide a PDB ID and chain if we provide an MSA to match it against:

```
evol rankorder PF00074_full_refined_mutinfo_corr_apc.txt -q 5 -p 2W5IB -m PF00074_full_refined.sth
```

Or even use the MSA directly if it has start and end in the labels:

```
evol rankorder PF00074_full_refined_mutinfo_corr_apc.txt -q 5 -m PF00074_full_refined.sth -l RNAS1_BO
```

Acknowledgments

Continued development of Protein Dynamics Software *ProDy* and associated programs is partially supported by the NIH[5]-funded R01 GM139297 entitled *"Toward a deeper understanding of allostery and allotargeting by computational approaches"*.

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

[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.