

SUPPORTING MATERIAL 1-13

Barnaba: Software for Analysis of Nucleic Acids Structures and Trajectories

Sandro Bottaro, Giovanni Bussi, Giovanni Pinamonti,
Sabine Reißer, Wouter Boomsma, Kresten Lindorff-Larsen

October 1, 2018

Contents

1	Command-line options	2
2	RMSD/eRMSD calculation	4
3	Relative position and orientation between nucleobases	11
4	Annotate structures and simulations	18
4.1	Decypher the annotation	19
4.2	Dot-bracket annotation	19
5	Calculate torsion angles (backbone and sugar puckering)	20
6	Calculate 3J scalar couplings	25
7	Search for single-stranded RNA motifs	28
8	Search for double-stranded RNA motif in database	33
9	Snippet utility: extract fragments from structures with a given sequence	36
10	Clustering RNA structures	40
11	Dynamic secondary structure figures	44
12	Elastic Network Model	45
12.1	1: ENM construction	45
12.2	2: Eigenvalues and eigenvectors	45
12.3	3: Mean square fluctuations	47
12.4	4: C2-C2 fluctuations	48
12.5	5: Sparse diagonalization	49
12.5.1	5b: C2-C2 fluctuations with sparse matrices:	50
12.6	6: Visualize the eigenmodes	51
12.7	References:	51
12.8	Extra credits: Distance fluctuation matrix	52
13	eSCORE, a scoring function for RNA structure prediction	55

1 Command-line options

When properly installed, barnaba can be executed from the commandline. The files for executing the examples below can be found in the github repository.

1. minimal help:

```
barnaba --help
```

2. Calculate the ERMSD between structures

```
barnaba ERMSD --ref data/sample1.pdb
              --pdb data/sample2.pdb
```

trajectories can be provided as well, by specifying a topology file

```
barnaba ERMSD --ref data/sample1.pdb
              --top data/sample1.pdb
              --trj data/samples.xtc
```

other accepted options are shown in a function-specific help

```
barnaba ERMSD --help
```

3. Calculate the RMSD between structures. The dump option writes to disk the superimposed structure.

```
barnaba RMSD --ref data/sample1.pdb
             --pdb data/sample2.pdb --dump
```

4. Find structures similar to GNRA.pdb in 1S72.pdb. GNRA.pdb is a PDB file containing a single-stranded structure.

```
barnaba SS_MOTIF --query data/GNRA.pdb --pdb data/1S72.pdb
```

5. Find double stranded motif. l1 and l2 are the lengths of the two strands

```
barnaba DS_MOTIF --query data/SARCIN.pdb
                 --pdb data/1S72.pdb --l1 8 --l2 7
```

6. Annotate structures/trajectories according to the Leontis/Westhof classification.

```
barnaba ANNOTATE --pdb data/SARCIN.pdb
```

7. Calculate backbone, sugar and pseudorotation angles

```
barnaba TORSION --pdb data/GNRA.pdb
               --backbone --sugar --pucker
```

8. Calculate J-couplings

```
barnaba JCOUPLING --pdb data/sample1.pdb
```

9. Calculate elastic network models for RNA and predict SHAPE reactivity. NB: only works with PDB.

```
barnaba ENM --pdb data/GNRA.pdb --shape
```

10. Calculate relative positions between bases (R) and Gvectors for pairs within ellipsoidal cutoff

```
barnaba DUMP --pdb data/GNRA.pdb --dumpG --dumpR
```

11. Extract fragments from structures with a given sequence. This command only works for PDB files.

```
barnaba SNIPPET --pdb data/1S72.pdb --seq NNGNRANN
```

12. Calculate ESCORE

```
barnaba ESCORE --ff data/1S72.pdb --pdb data/sample1.pdb
```

2 RMSD/eRMSD calculation

We here show how to calculate distances between three-dimensional structures. eRMSD can be calculated using the function

```
ermsd = bb.ermsd(reference_file,target_file)
```

reference_file and target_file can be e.g. PDB files. eRMSD between reference and all frames in a simulation can be calculated by specifying the trajectory and topology files:

```
ermsd = bb.ermsd(reference_file,target_traj_file,topology=topology_file)
```

All trajectory formats accepted by MDTRAJ (e.g. pdb, xtc, trr, dcd, binpos, netcdf, mdcrd, prmtop) can be used. Let us see a practical example:

```
In [3]: # import barnaba
import barnaba as bb

# define trajectory and topology files
native="uucg2.pdb"
traj = "../test/data/UUCG.xtc"
top = "../test/data/UUCG.pdb"

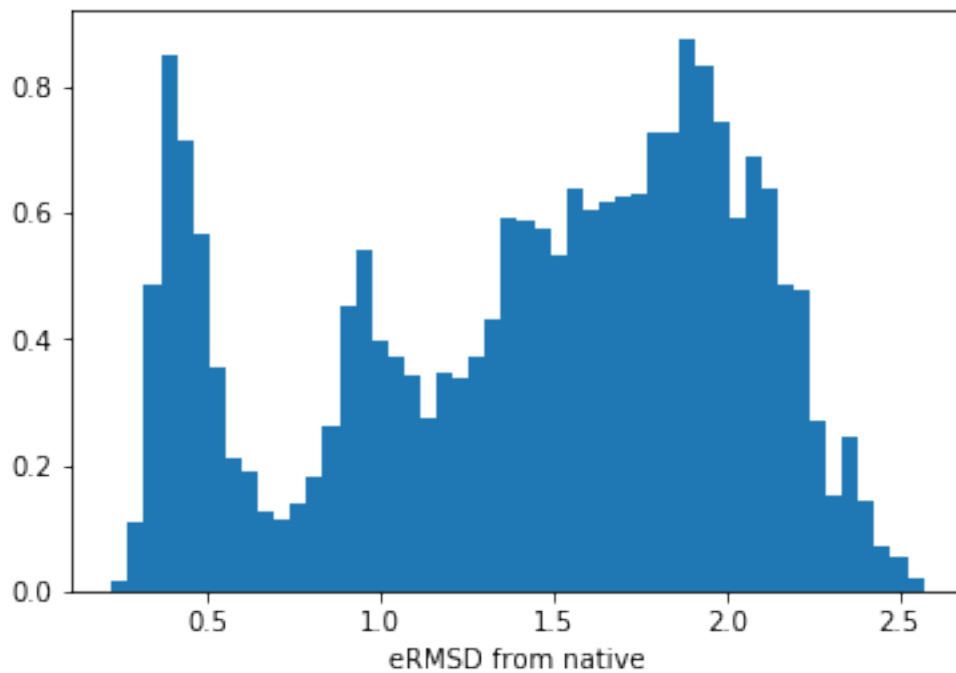
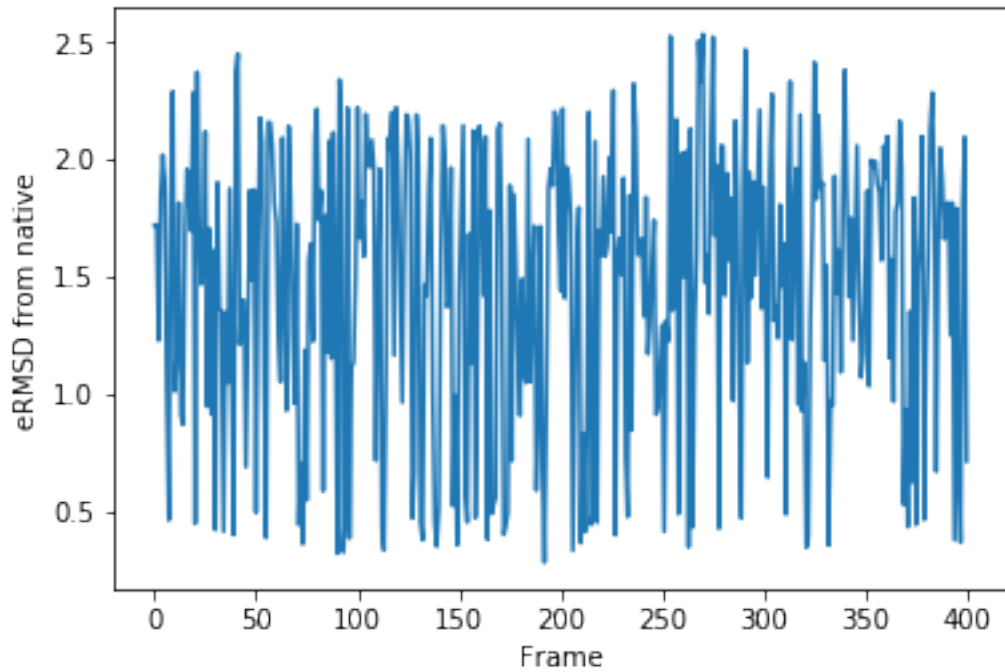
# calculate eRMSD between native and all frames in trajectory
ermsd = bb.ermsd(native,traj,topology=top)

# Loaded reference uucg2.pdb
# Loaded target ../test/data/UUCG.xtc
```

We plot the eRMSD over time (every 50 frames to make the plot nicer) and make an histogram

```
In [15]: import matplotlib.pyplot as plt
plt.xlabel("Frame")
plt.ylabel("eRMSD from native")
plt.plot(ermsd[::50])
plt.show()

plt.hist(ermsd,density=True,bins=50)
plt.xlabel("eRMSD from native")
plt.show()
```



As a rule of thumb, eRMSD below 0.7-0.8 can be considered *low*, as such the peak around 0.4 eRMSD corresponds to structures that are very similar to the native.

Nota Bene - eRMSD is a dimensionless number. - Remember to remove periodic boundary conditions before performing the analysis.

We can also calculate the root mean squared deviation (RMSD) after optimal superposition by using

```
rmsd = bb.rmsd(reference_file,target_file)
```

or

```
rmsd = bb.rmsd(reference_file,target_traj_file,topology=topology_file)
```

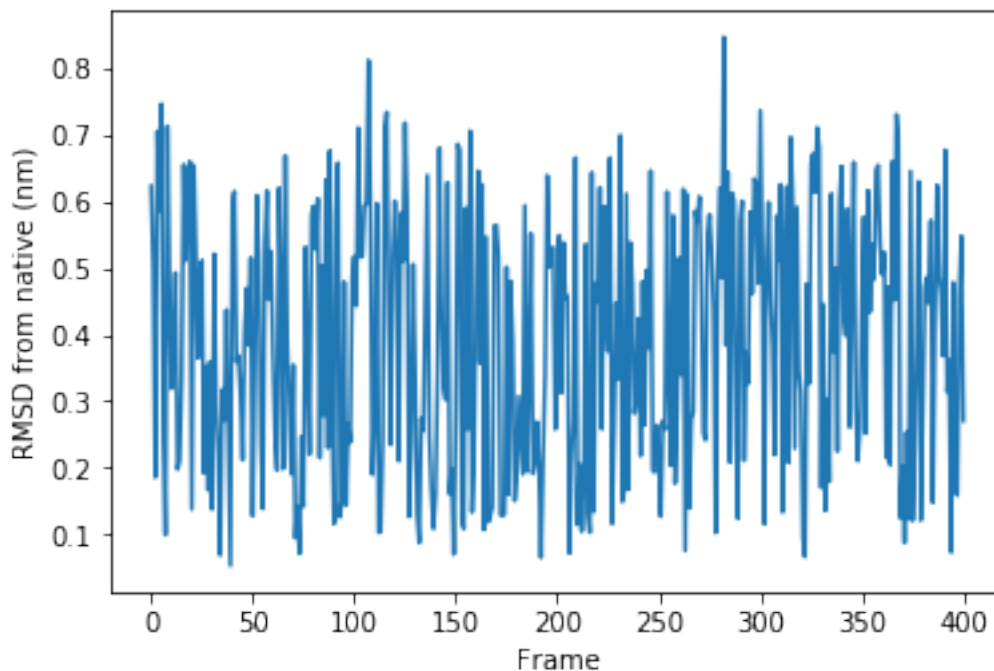
for trajectories. By default RMSD is calculated using backbone atoms only (`heavy_atom=False`): this makes it possible to calculate RMSD between structures with different sequences. If `heavy_atom=True`, RMSD is calculated using all heavy atoms. Values are expressed in nanometers.

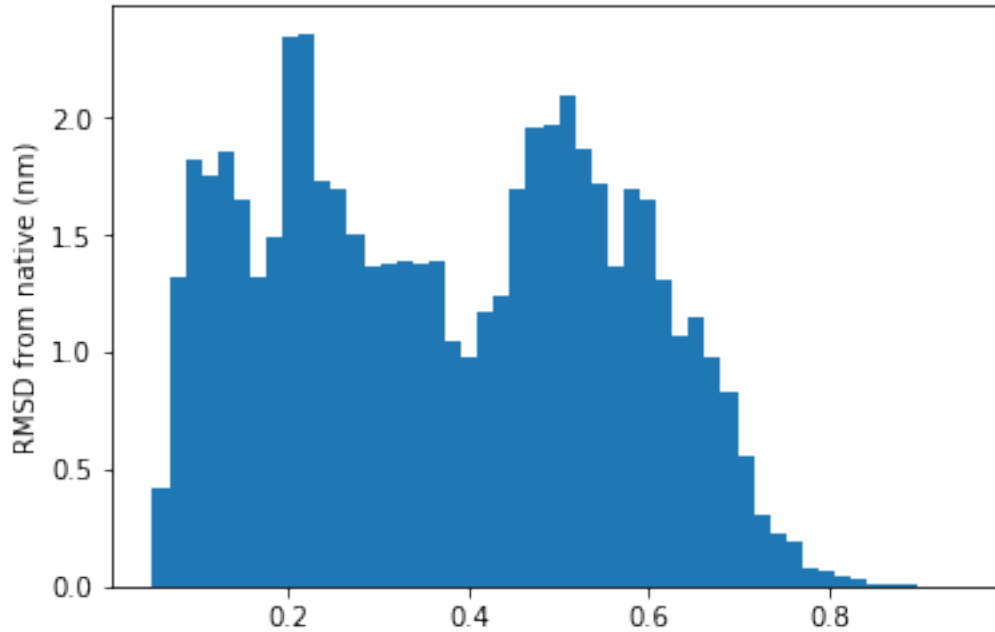
```
In [5]: # calculate RMSD
rmsd = bb.rmsd(native,traj,topology=top,topology_file=top,heavy_atom=False)

# plot time series
plt.xlabel("Frame")
plt.ylabel("RMSD from native (nm)")
plt.plot(rmsd[::50])
plt.show()

# make histogram
plt.hist(rmsd,density=True,bins=50)
plt.ylabel("RMSD from native (nm)")
plt.show()

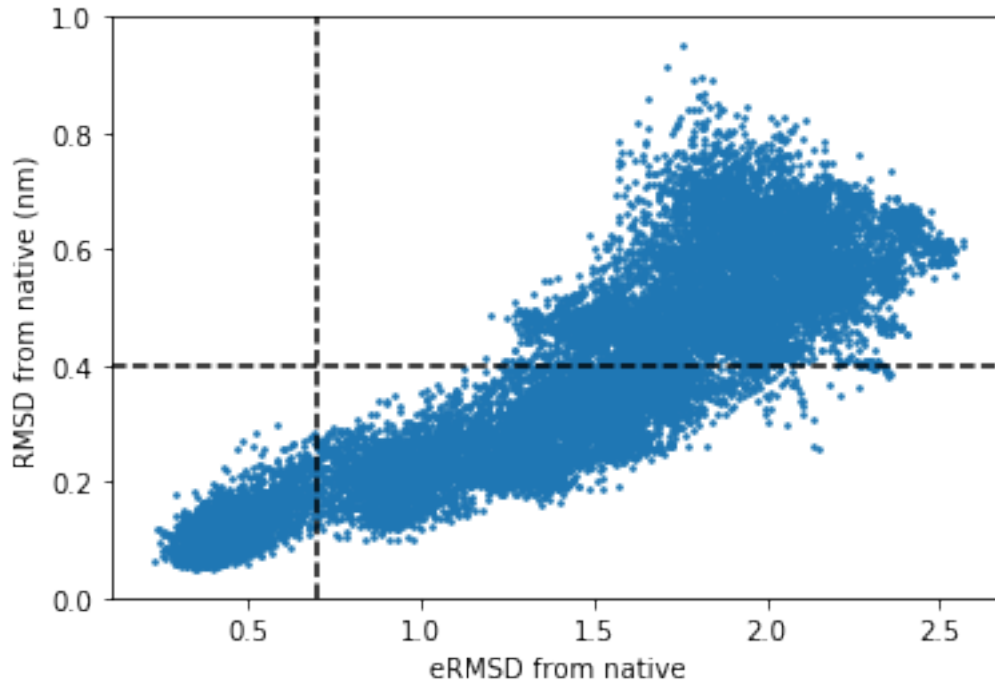
# found 93 atoms in common
```





Structures with eRMSD lower than 0.7 are typically significantly similar to the reference. Note that structures with *low* RMSD (less than 0.4 nm) may be very different from native. We can check if this is true by comparing RMSD and eRMSD

```
In [6]: plt.xlabel("eRMSD from native")
plt.ylabel("RMSD from native (nm)")
plt.axhline(0.4,ls = "--", c= 'k')
plt.axvline(0.7,ls = "--", c= 'k')
plt.scatter(ermsd,rmsd,s=2.5)
plt.show()
```



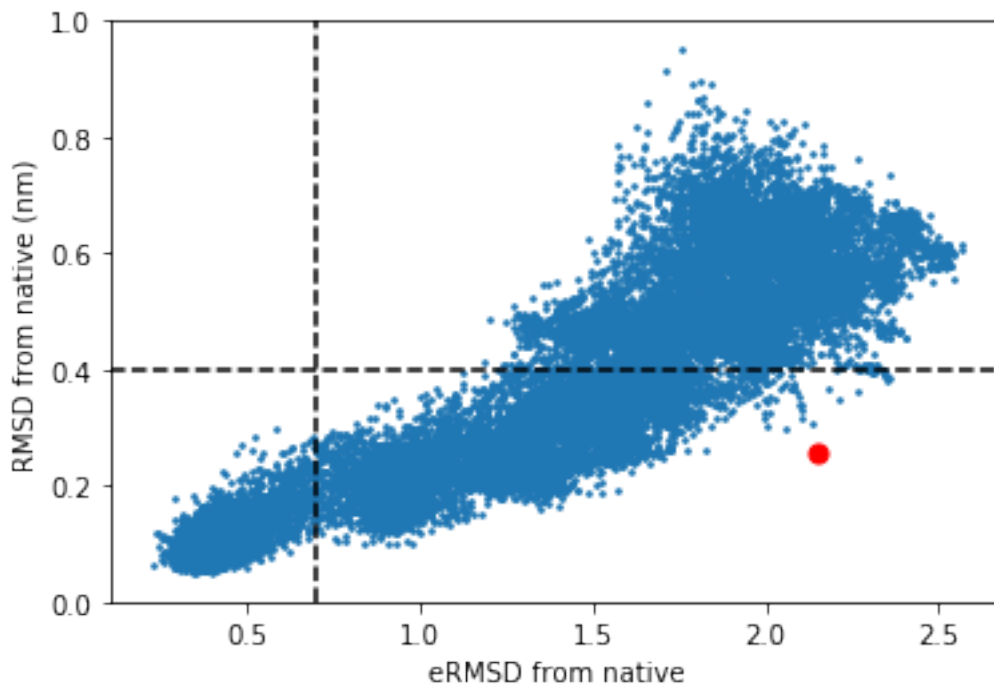
We can clearly see that the two measures are correlated, but several structures with low RMSD have very large eRMSD. We cherry-pick a structure with RMSD from native ≈ 0.3 nm, but high eRMSD.

```
In [7]: import numpy as np
low_rmsd = np.where(rmsd<0.3)
idx_a = np.argsort(ermsd[low_rmsd])[-1]
low_e = low_rmsd[0][idx_a]
print("Highest eRMSD for structures with RMSD ~ 0.3nm")
print("eRMSD:%5.3f; RMSD: %5.3f nm" % (ermsd[low_e],rmsd[low_e]))

plt.xlabel("eRMSD from native")
plt.ylabel("RMSD from native (nm)")
plt.axhline(0.4,ls = "--", c= 'k')
plt.axvline(0.7,ls = "--", c= 'k')

plt.scatter(ermsd,rmsd,s=2.5)
plt.scatter(ermsd[low_e],rmsd[low_e],s=50,c='r')
plt.show()
```

```
Highest eRMSD for structures with RMSD ~ 0.3nm
eRMSD:2.149; RMSD: 0.257 nm
```

We can extract a frame from the simulation using the save function from MDTraj. Aligned structures are written to disk by passing a string out to the rmsd function.

```
In [8]: import mdtraj as md

# load trajectory
tt = md.load(traj,top=top)

# save low ermsd
tt[low_e].save("low_rmsd.pdb")

# align to native and write aligned PDB to disk
rmsd1 = bb.rmsd(native,'low_rmsd.pdb',out='low_rmsd_align.pdb')

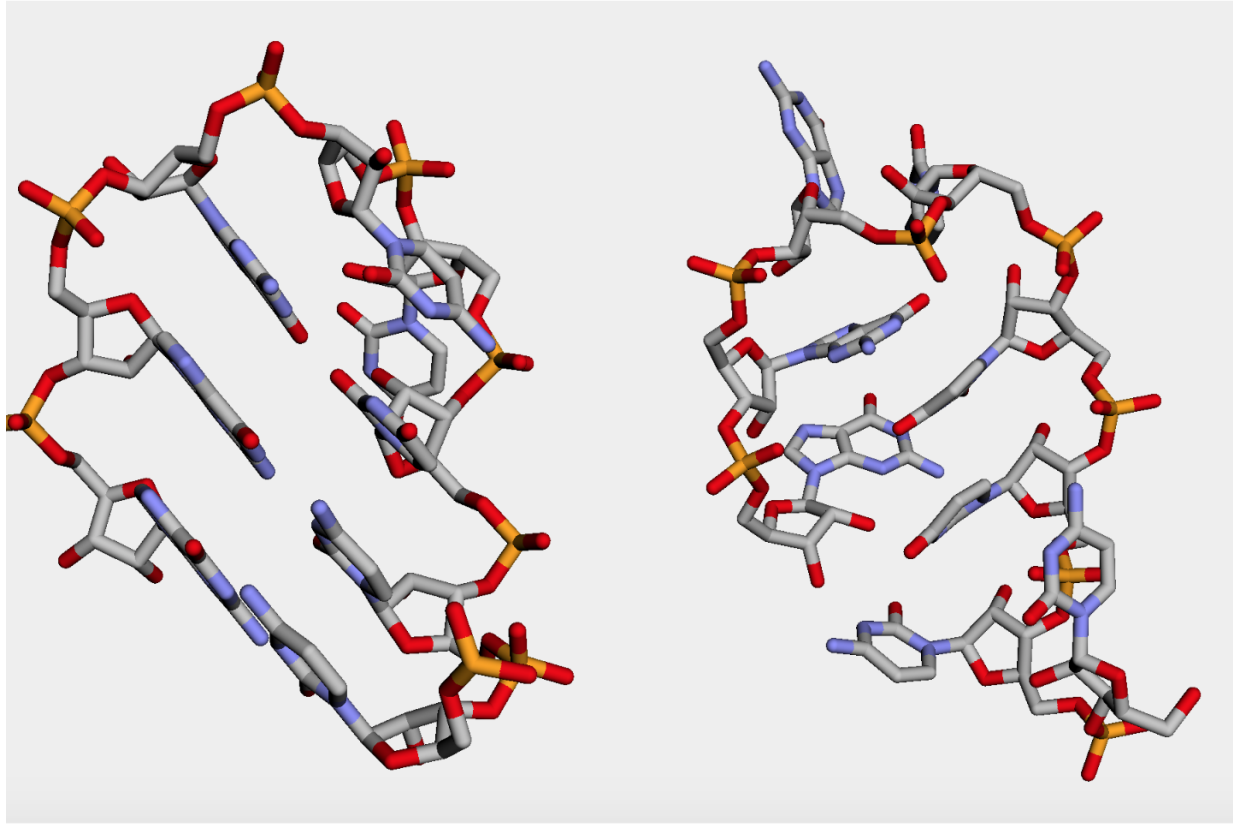
# found 93 atoms in common
```

Finally, we use py3Dmol module to visualize the native and the low-RMSD/high-eRMSD structure.

```
In [14]: import py3Dmol
pdb_e = open('low_rmsd_align.pdb','r').read()
pdb_n = open(native,'r').read()

p = py3Dmol.view(width=900,height=600,viewergrid=(1,2))
p.addModel(pdb_n,'pdb',viewer=(0,0))
p.addModel(pdb_e,'pdb',viewer=(0,1))

p.setStyle({'stick':{}})
p.setBackgroundColor('0xeeeeee')
p.zoomTo()
```



On the left the native UUCG and on the right the highest eRMSD among all structures with RMSD from native \approx 0.3nm.

3 Relative position and orientation between nucleobases

The relative position of a nucleobase i in the reference frame constructed on the base j carries interesting information, as described in Bottaro, Di Palma Bussi. Nucleic acids research (2014). It is possible to calculate the all the position vectors between all pairs in a molecule using the function

```
rvecs,res = bb.dump_rvec(pdb,cutoff=2.0)
```

`rvecs` is a matrix with dimensions $(nframes,n,n,3)$, where $nframes$ is the number of samples in the PDB/trajectory file, and n the sequence length. The position of base j in the reference frame constructed on base i in sample k is therefore stored in `rvecs[k,i,j]`. Note that $r_{i,j} \neq r_{j,i}$ and that $r_{j,j} = (0,0,0)$. Additionally, all pairs of bases with ellipsoidal distance larger than `cutoff` are set to zero. The meaning and rationale for this ellipsoidal distance will be clarified in the example below.

`res` contains the list of residues. The naming convention is `RESNAME_RESNUMBER_CHAININDEX`, where `RESNAME` and `RESNUMBER` are as in the PDB/topology file and `CHAININDEX` is the index of the chain starting from zero, in the same order as it appears in the PDB/topology file. It is not possible to get the chain name.

We here analyze the crystal structure of the large ribosomal subunit (PDB 1S72)

```
In [56]: # import barnaba
import barnaba as bb

pdb = "../test/data/1S72.pdb"
rvecs,res = bb.dump_rvec(pdb,cutoff=3.5)
```

```
# Loading ../test/data/1S72.pdb
# Treating nucleotide 1MA628 as A
# Treating nucleotide 0MU2587 as U
# Treating nucleotide 0MG2588 as G
# Treating nucleotide UR32619 as U
# Treating nucleotide PSU2621 as U
```

We remove all zero-vectors and scatter plot $\rho = \sqrt{x^2 + y^2}$ versus z

```
In [57]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.collections import PatchCollection
import seaborn as sns

# find all zero-elements
nonzero = np.where(np.sum(rvecs**2,axis=3)>0.01)
rr = rvecs[nonzero]
# calculate rho and zeta
z = rr[:,2]
rho = np.sqrt(rr[:,0]**2 + rr[:,1]**2)

# make a scatter plot
fig,ax = plt.subplots(figsize=(9,9))
ax.scatter(rho,z,s=0.15)
#ax.set_aspect(1)
patches = []
f1 = 0.5
f2 = 0.3
e1 = mpatches.Ellipse([0,0], 2*f1,2*f2, fc="none",ec='r',ls="--",lw=1.75)
ax.text(-0.08,f2*1.05,"cutoff=1")
```

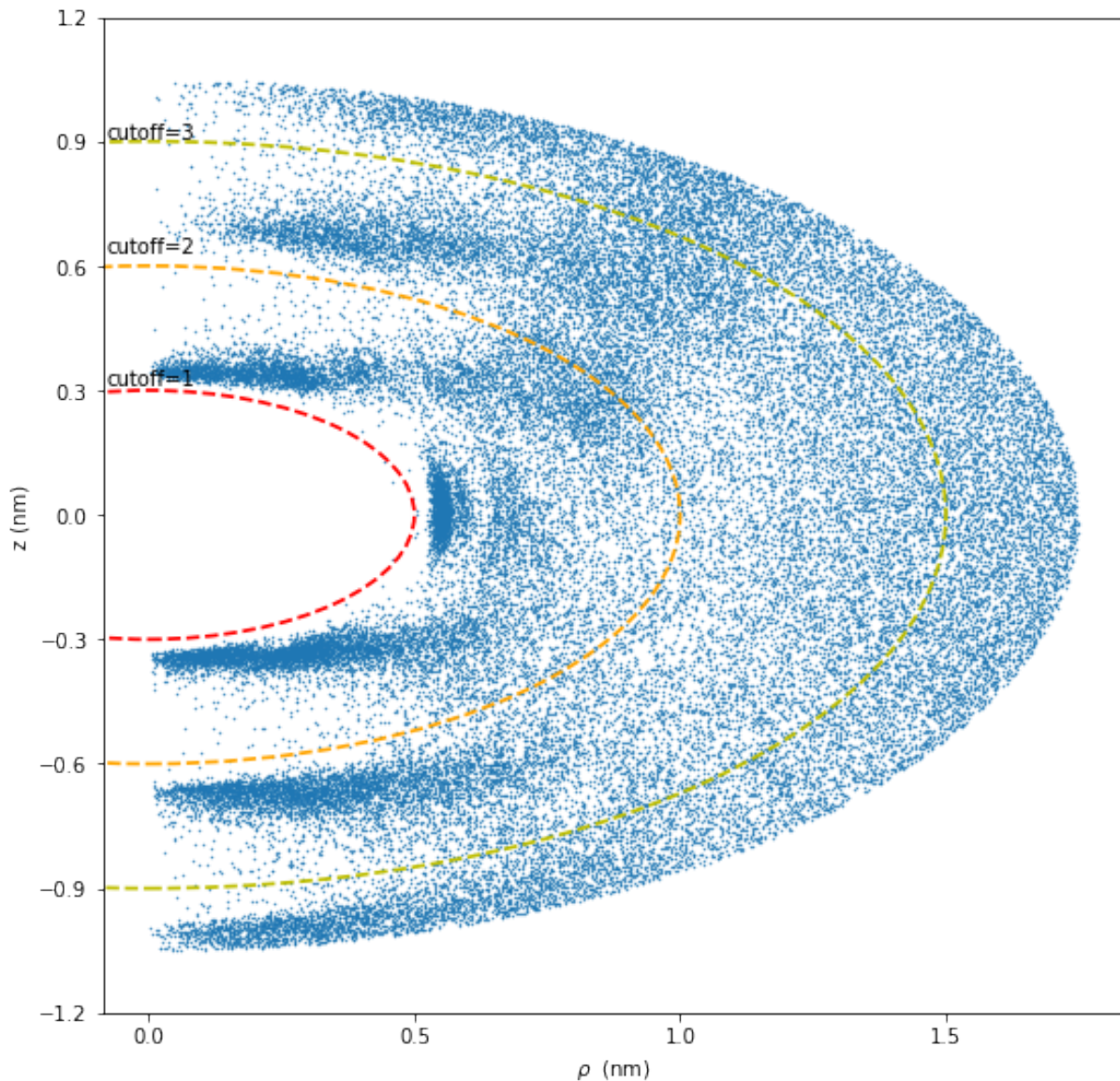
```

patches.append(e1)
e1 = mpatches.Ellipse([0,0], 4*f1,4*f2, fc="none",ec='orange',ls="--",lw=1.75)
ax.text(-0.08,2*f2*1.05,"cutoff=2")
patches.append(e1)
e1 = mpatches.Ellipse([0,0], 6*f1,6*f2, fc="none",ec='y',ls="--",lw=1.75)
ax.text(-0.08,3*f2*1.005,"cutoff=3")
patches.append(e1)
collection = PatchCollection(patches,match_original=True)
ax.add_collection(collection)

ax.set_xlabel(r'\rho$ (nm)')
ax.set_ylabel('z (nm)')
ax.set_yticks([-1.2,-0.9,-0.6,-0.3,0,0.3,0.6,0.9,1.2])
ax.set_xticks([0,0.5,1.0,1.5])

plt.show()

```



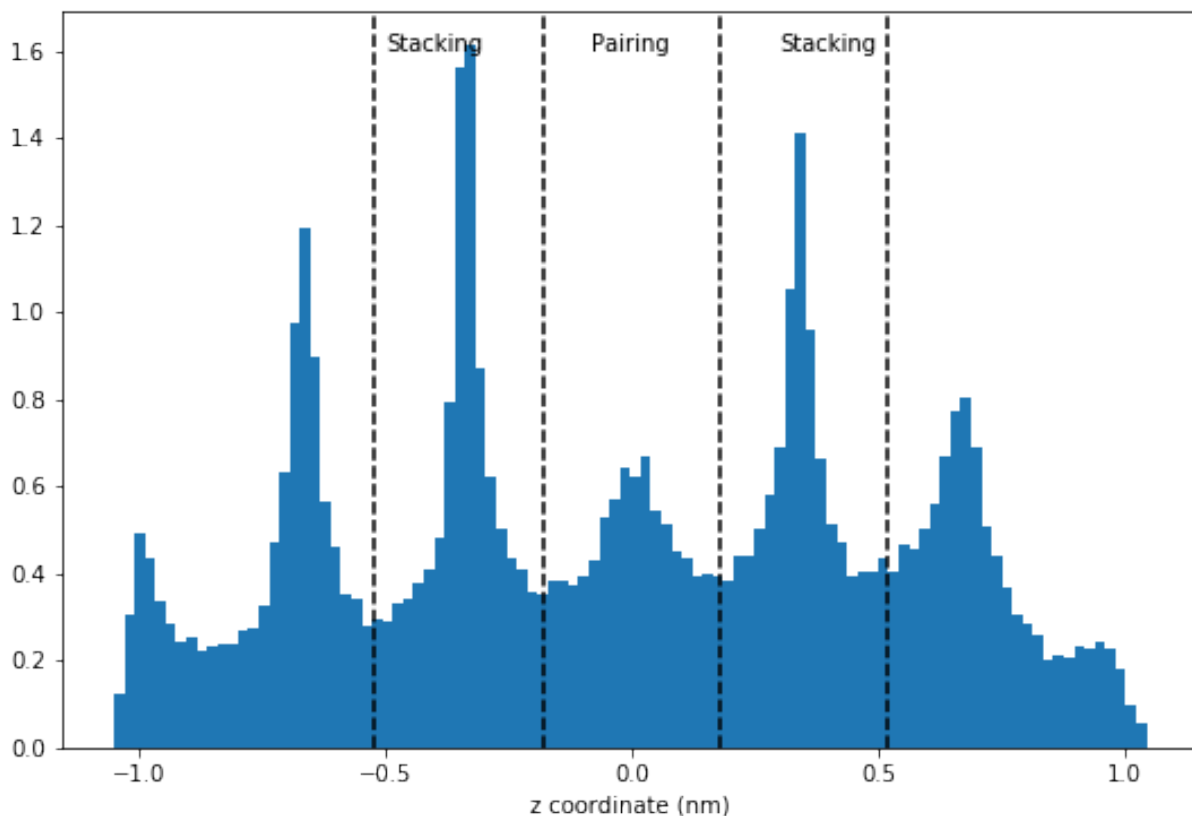
We can see that high-density points are observed around (0,0.6) (base-pairing), (0.3, ±0.33) (base stacking). Note also the ellipsoid with major axis $a = b = 0.5nm$ and minor axis $c = 0.3nm$ defines a natural metric. For values of the scaled distance $|\tilde{r}| = (a^2x^2 + b^2y^2 + c^2z^2)^{(1/2)}$ smaller than 1 (*cutoff=1*), no points are observed. Base-stacking and base-pairings are observed for cutoff distances smaller than 2.

This is also confirmed by looking at the histogram along the z coordinate:

```
In [58]: fig,ax = plt.subplots(figsize=(9,6))

plt.hist(z,bins=100,density=True)
plt.axvline(0.18,ls="--",c='k')
plt.axvline(-0.18,ls="--",c='k')
plt.axvline(0.52,ls="--",c='k')
plt.axvline(-0.52,ls="--",c='k')
plt.text(0,1.6,"Pairing",ha="center")
plt.text(-0.4,1.6,"Stacking",ha="center")
plt.text(0.4,1.6,"Stacking",ha="center")
plt.xlabel("z coordinate (nm)")

plt.show()
```



Another interesting exercise is to consider only the points in the pairing slice and project them on the (x,y) plane.

```

In [59]: # define an helper function to plot the nucleobase and some distances, as a reference.
def plot_grid():
    patches = []
    polygon = mpatches.RegularPolygon([0,0], 6, 0.28,fc='none',ec='k',lw=3,orientation=np.pi/2)
    patches.append(polygon)
    polygon = mpatches.RegularPolygon([-0.375,-0.225], 5, 0.24,fc='none',ec='k',lw=3,
    orientation=-0.42)
    patches.append(polygon)
    circle = mpatches.Circle([0,0], 0.5, fc="none",ec='k',ls="--",lw=0.75)
    plt.text(-0.53,0,"r=0.5 nm",rotation=90,ha="center",va='center',fontsize=13)
    patches.append(circle)
    circle = mpatches.Circle([0,0], 0.75, fc="none",ec='k',ls="--",lw=0.75)
    plt.text(-0.78,0,"r=0.75 nm ",rotation=90,ha="center",va='center',fontsize=13)
    patches.append(circle)
    circle = mpatches.Circle([0,0], 1.0, fc="none",ec='k',ls="--",lw=0.75)
    plt.text(-1.03,0,"r=1.0 nm ",rotation=90,ha="center",va='center',fontsize=13)
    patches.append(circle)
    collection = PatchCollection(patches,match_original=True)
    ax.add_collection(collection)

    plt.plot([0,1.], [0,0],c='gray',lw=1,ls="--")
    plt.text(1.1,0,r"$\theta=0^\circ$",ha="center",va='center',fontsize=13)
    plt.plot([0,-np.cos(np.pi/3)], [0,np.sin(np.pi/3)],c='gray',lw=1,ls="--")
    plt.text(-np.cos(np.pi/3)*1.1,np.sin(np.pi/3)*1.1,r"$\theta=120^\circ$",
    ha="center",va='center',fontsize=13)

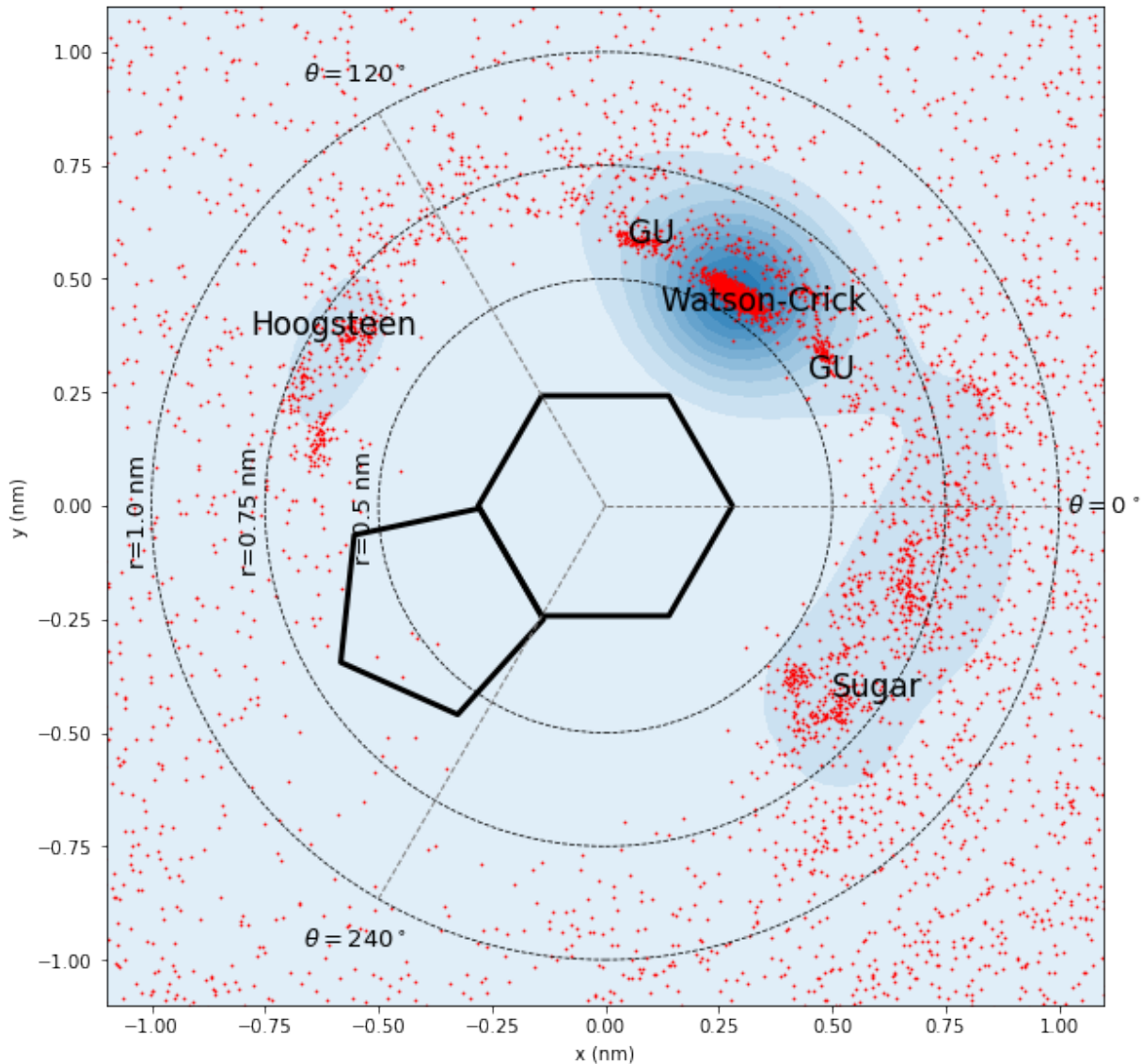
    plt.plot([0,-np.cos(np.pi/3)], [0,-np.sin(np.pi/3)],c='gray',lw=1,ls="--")
    plt.text(-np.cos(np.pi/3)*1.1,-np.sin(np.pi/3)*1.1,r"$\theta=240^\circ$",
    ha="center",va='center',fontsize=13)
    ax.set_aspect(1)
    ax.set_ylim(-1.1,1.1)
    ax.set_xlim(-1.1,1.1)
    ax.set_xlabel("x (nm)")
    ax.set_ylabel("y (nm)")

    # slice and take only where |z| is smaller than 0.18 nm
    pairs = rr[np.where(np.abs(rr[:,2])<0.18)]

    fig,ax = plt.subplots(figsize=(10,10))
    # do a KDE
    ax = sns.kdeplot(pairs[:,0],pairs[:,1], shade=True,bw=0.12)
    # scatter plot x and y
    ax.scatter(pairs[:,0],pairs[:,1],s=0.5,c='r')
    # make labels
    ax.text(0.35,0.45,"Watson-Crick",fontsize=17,ha='center',va='center',color='k')
    ax.text(0.1,0.6,"GU",fontsize=17,ha='center',va='center',color='k')
    ax.text(0.5,0.3,"GU",fontsize=17,ha='center',va='center',color='k')
    ax.text(-0.6,0.4,"Hoogsteen",fontsize=17,ha='center',va='center',color='k')
    ax.text(0.6,-0.4,"Sugar",fontsize=17,ha='center',va='center',color='k')

    plot_grid()
    plt.show()

```

The scatterplot above contains all contributions from all types of base-pairs. Still, we can clearly see many points around (0.4,0.5), corresponding to watson-crick base-pairs, wobble GU, hoogsteen and sugar interactions, as labeled. We can also scatterplot pairs at a fixed "sequence", for example A-U base pairing only:

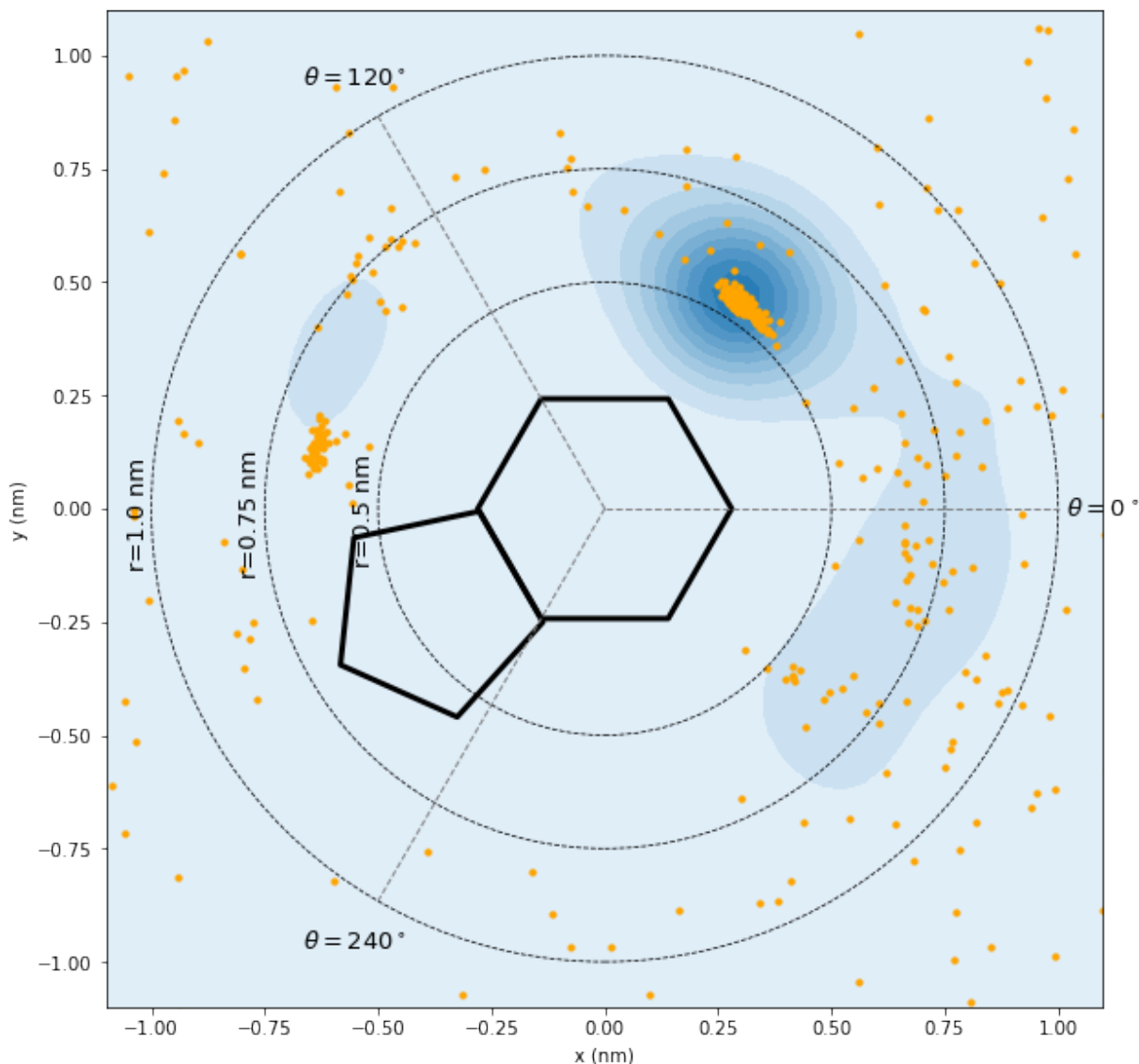
```
In [60]: # take only au-pairs. Need an explicit loop.
pp1 = []
for j in range(len(nonzero[0])):
    z = rvecs[0,nonzero[1][j],nonzero[2][j]][2]
    r1 = res[nonzero[1][j]][0]
    r2 = res[nonzero[2][j]][0]
    if(np.abs(z) < 0.18):
        if((r1=="A" and r2=="U")):
            pp1.append(rvecs[0,nonzero[1][j],nonzero[2][j]])

# plot KDE and scatter
```

```

fig,ax = plt.subplots(figsize=(10,10))
pp1 = np.array(pp1)
ax = sns.kdeplot(pairs[:,0],pairs[:,1], shade=True,bw=0.12)
ax.scatter(pp1[:,0],pp1[:,1],s=10,c='orange')
plot_grid()
plt.show()

```



Note that the distributions shown here are at the core of the eSCORE scoring function.

Another possible application of the `dump_rvec` function is to analyze trajectories. For example, we can monitor the distance between the center of two six-membered rings during a simulation. To do so, we use a very large cutoff, so that the only zero vectors are on the diagonal.

```

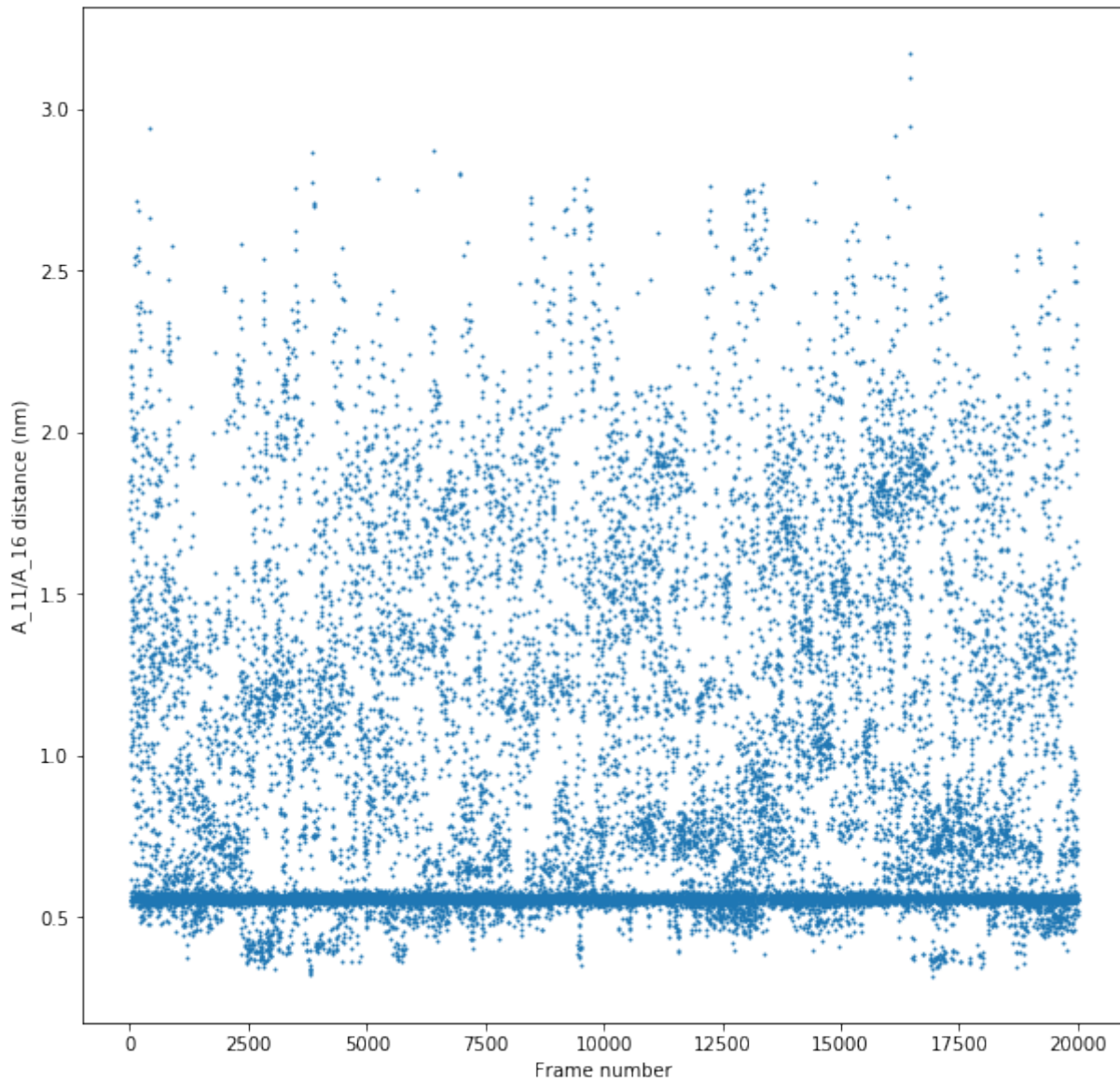
In [61]: traj = "../test/data/UUCG.xtc"
         top = "../test/data/UUCG.pdb"
         rvecs_traj,res_traj = bb.dump_rvec(traj,topology=top,cutoff=100.0)

```



```
# Loading ../test/data/UUCG.xtc
```

```
In [62]: fig,ax = plt.subplots(figsize=(10,10))
dist = np.sqrt(np.sum(rvecs_traj[:,1,6]**2,axis=1))
ax.scatter(np.arange(len(dist)),dist,s=1)
ax.set_xlabel("Frame number")
ax.set_ylabel("%s/%s distance (nm)" % (res[1][:-2],res[6][:-2]))
plt.show()
```



Note that the base-pair C2-G7 is often formed (distance $\approx 0.57nm$).

4 Annotate structures and simulations

We here show how to find base-pairs and stacking interactions in structures and simulations. The function

```
stackings, pairings, res = bb.annotate(pdb)
```

returns three lists: - a list of stacking interactions - a list of pairing interactions - the list of residue names following the usual convention RESNAME_RESNUMBER_CHAININDEX

stackings and pairings contain the list of interactions for the N frames in the PDB/trajectory file and it is organized in the following way: for a given frame $i = 1..N$ there are $k = 1..Q$ interactions between residues with index pairings[i][0][k][0] and pairings[i][0][k][1]. The type of interaction is specified at the element pairings[i][1][k].

But let's make an example by annotating a PDB file containing a sarcin-ricin motif:

```
In [2]: import barnaba as bb
```

```
# annotate
pdb = "../test/data/SARCIN.pdb"
stackings, pairings, res = bb.annotate(pdb)

# list base pairings
print("BASE-PAIRS")
for p in range(len(pairings[0][0])):
    res1 = res[pairings[0][0][p][0]]
    res2 = res[pairings[0][0][p][1]]
    interaction = pairings[0][1][p]
    print("%10s %10s %4s" % (res1,res2,interaction))

# list base-stackings
print()
print("STACKING")
for p in range(len(stackings[0][0])):
    res1 = res[stackings[0][0][p][0]]
    res2 = res[stackings[0][0][p][1]]
    interaction = stackings[0][1][p]
    print("%10s %10s %4s" % (res1,res2,interaction))
```

BASE-PAIRS

C_6_0	G_24_0	WCc
U_7_0	C_23_0	WHc
C_8_0	C_22_0	SHt
A_9_0	A_21_0	HHT
G_10_0	U_11_0	SHc
U_11_0	A_20_0	WHt
A_12_0	G_19_0	HSc
U_13_0	A_18_0	WCc

STACKING

C_6_0	U_7_0	>>
U_7_0	C_8_0	>>
C_8_0	A_9_0	<<
A_12_0	U_13_0	>>
A_12_0	A_20_0	<>
A_18_0	G_19_0	>>
A_20_0	A_21_0	>>

```
A_21_0    C_22_0    >>
C_22_0    C_23_0    >>
```

```
# Loading ../test/data/SARCIN.pdb
```

4.1 Decypher the annotation

Base-pairing are classified according to the Leontis-Westhof classification, where - W = Watson-Crick edge - H = Hoogsteen edge - S= Sugar edge - c/t = cis/trans - XXx = when two bases are close in space, but they do not fall in any of the categories. This happens frequently for low-resolution structures or from molecular simulations.

WWc pairs between complementary bases are called WCc or GUc.

Stacking are classified according to the MCanotate classification: - ">>" Upward - "<<" Downward - "<>" Outward - "><" Inward

4.2 Dot-bracket annotation

From the list of base-pairing, we can obtain the dot-bracket annotation using the function

```
dotbracket = bb.dot_bracket(pairings,res)
```

this function returns a string for each frame in the PDB/simulation. Let's see this in action:

```
In [3]: dotbr, seq = bb.dot_bracket(pairings,res)
        print(">",seq)
        for j in range(len(dotbr)):
            print(dotbr[j])
```

```
> CUCAGUAUAGAACCG
```

```
(...())...
```

5 Calculate torsion angles (backbone and sugar puckering)

Here we calculate torsion angles in a PDB or trajectory.
Backbone torsion angles are calculated using the function

```
angles,res = bb.backbone_angles(pdb_file)
```

for PDB files. For trajectories, it is necessary to specify a topology file

```
angles,res = bb.backbone_angles(trajectory_file,topology=topology_file)
```

All trajectory formats accepted by MDTRAJ (e.g. pdb, xtc, trr, dcd, binpos, netcdf, mdcrd, prmtop) can be used. Two objects are returned: *angles* and *res*. *angles* is an array with shape $n \times m \times 7$, where n = # number of frames ($n=1$ in the example below) m = # number of residues 7 = number of torsion angles ($\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \chi$). Angles are expressed in radians and in the $(-\pi, \pi)$ range.

res is the name of the residues. The naming convention is RESNAME_RESNUMBER_CHAININDEX, where RESNAME and RESNUMBER are as in the PDB/topology file and CHAININDEX is the index of the chain starting from zero, in the same order as it appears in the PDB/topology file. It is not possible to get the chain name. Note that non-nucleic acids (proteins, water, ions) are ignored. Modified nucleotides are mapped to standard nucleotides if present in the *modified_dict* in the file *definitions.py*.

We start by calculating all backbone angle in the structure of the UUCG tetraloop.

```
In [12]: # import barnaba
import barnaba as bb
from barnaba import definitions

# calculate backbone angles
pdb_file="uucg2.pdb"
angles,res = bb.backbone_angles(pdb_file)

# print angles
header = "# Residue " + "".join(["%10s " % aa for aa in definitions.bb_angles])
print(header)
for j in range(angles.shape[1]):
    stri = "%10s" % res[j]
    for k in range(angles.shape[2]):
        stri += "%10.3f " % angles[0,j,k]
    print(stri)
```

# Residue	alpha	beta	gamma	delta	eps	zeta	chi
C_1448_0	nan	-2.912	0.746	1.502	-2.421	-1.251	-3.006
C_1449_0	-1.126	2.961	0.970	1.397	-2.559	-1.016	-2.945
U_1450_0	-1.304	3.092	1.087	1.368	-2.704	-1.706	-2.823
U_1451_0	-2.766	2.523	0.898	2.472	-1.755	-1.290	-2.889
C_1452_0	-2.632	-2.658	2.179	2.443	-1.498	2.513	-2.407
G_1453_0	-0.759	-1.666	-2.560	1.493	-2.551	-1.055	0.711
G_1454_0	2.459	-2.238	-3.131	1.469	-2.495	-1.258	-3.103
G_1455_0	-1.086	3.136	0.894	1.361	nan	nan	-2.904

```
# Loading uucg2.pdb
```

It is also possible to calculate single angles and residues by specifying a list of *angles* and *residues*. In the following example, we calculate χ angles in residues U_4_0 and G_6_0.

```
In [3]: traj = "../test/data/UUCG.xtc"
top = "../test/data/UUCG.pdb"
angles_s,res_s = bb.backbone_angles(traj,topology=top,\
                                   residues=["U_4_0","G_6_0"],angles=["chi"])
print(angles_s.shape)
```

(20001, 2, 1)

```
# Loading ../test/data/UUCG.xtc
```

We now plot the angle distributions, setting the domain to (0,360) to make the plot a bit nicer.

```
In [14]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

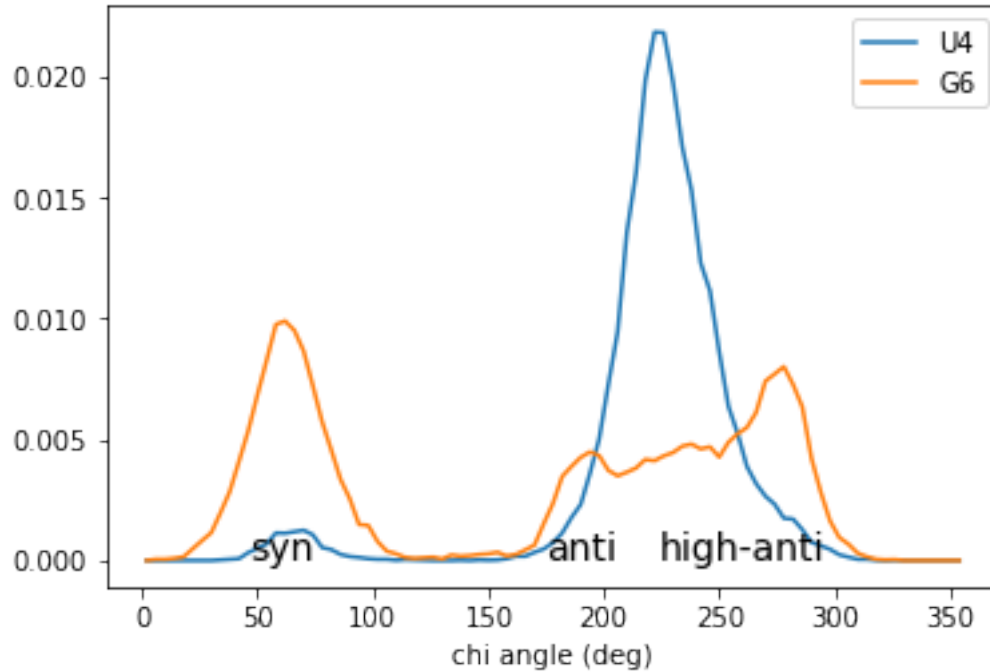
# move from -pi,pi to 0-2pi range
aa = np.copy(angles_s)
aa[np.where(aa<0.0)] += 2.*np.pi

# from radians to deg
aa *= 180.0/np.pi

# create histogram
bins = np.arange(0,360,4)
hh1,ee1 = np.histogram(aa[:,0,0],density=True,bins=bins)
hh2,ee2 = np.histogram(aa[:,1,0],density=True,bins=bins)

# make plot
plt.plot(0.5*(ee1[1:]+ee1[:-1]),hh1,label="U4")
plt.plot(0.5*(ee2[1:]+ee2[:-1]),hh2,label="G6")
plt.legend()
plt.xlabel("chi angle (deg)")
plt.text(60,0,"syn",fontsize=14,ha='center')
plt.text(190,0,"anti",fontsize=14,ha='center')
plt.text(260,0,"high-anti",fontsize=14,ha='center')
```

Out[14]: Text(260,0,'high-anti')



We can also calculate sugar torsion angles v_0 , v_1 , v_2 , v_3 and v_4 by calling the function

```
angles, residues = bb.sugar_angles(traj,topology=top)
```

```
In [15]: # calculate sugar angles for C_5. If angles is not specified, all
# torsion angles in the sugar v0, v1, v2, v3, v4 are calculated
```

```
angles_s,rr = bb.sugar_angles(traj,topology=top, residues=["C_5_0"])
print(angles_s.shape)
```

```
(20001, 1, 5)
```

```
# Loading ../test/data/UUCG.xtc
```

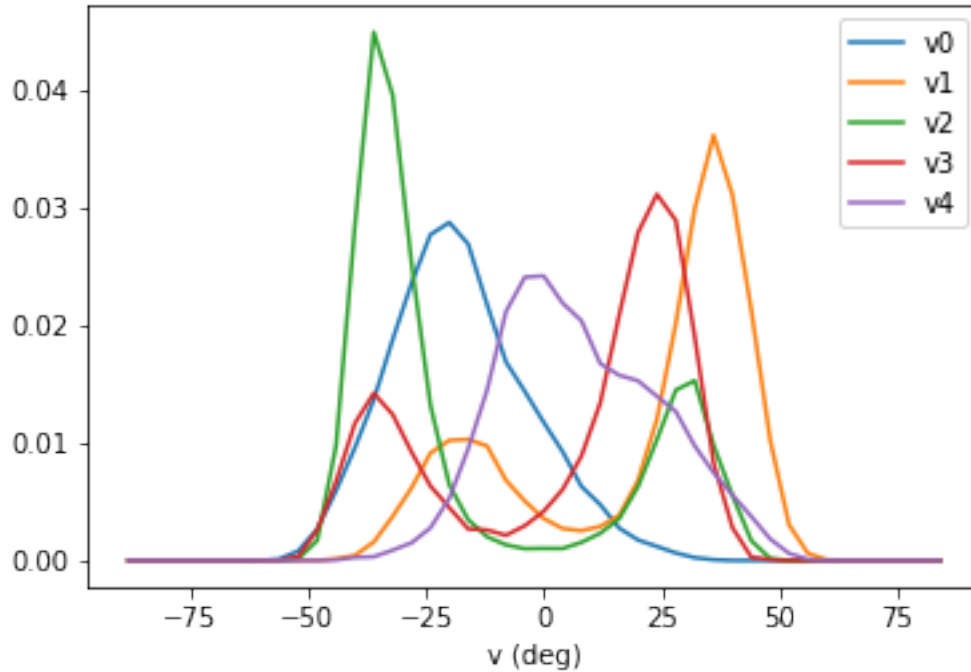
```
In [16]: aa1 = np.copy(angles_s)
```

```
# from radians to deg
aa1 *= 180.0/np.pi

bins = np.arange(-90,90,4)
for j in range(5):
    hh,ee = np.histogram(aa1[:,0,j],density=True,bins=bins)
    # make plot
    plt.plot(0.5*(ee[1:]+ee[:-1]),hh,label="v%d"%j)

plt.legend()
plt.xlabel("v (deg)")
```

```
Out[16]: Text(0.5,0,'v (deg)')
```



Amplitude and phase of the pucker is calculated by calling the function

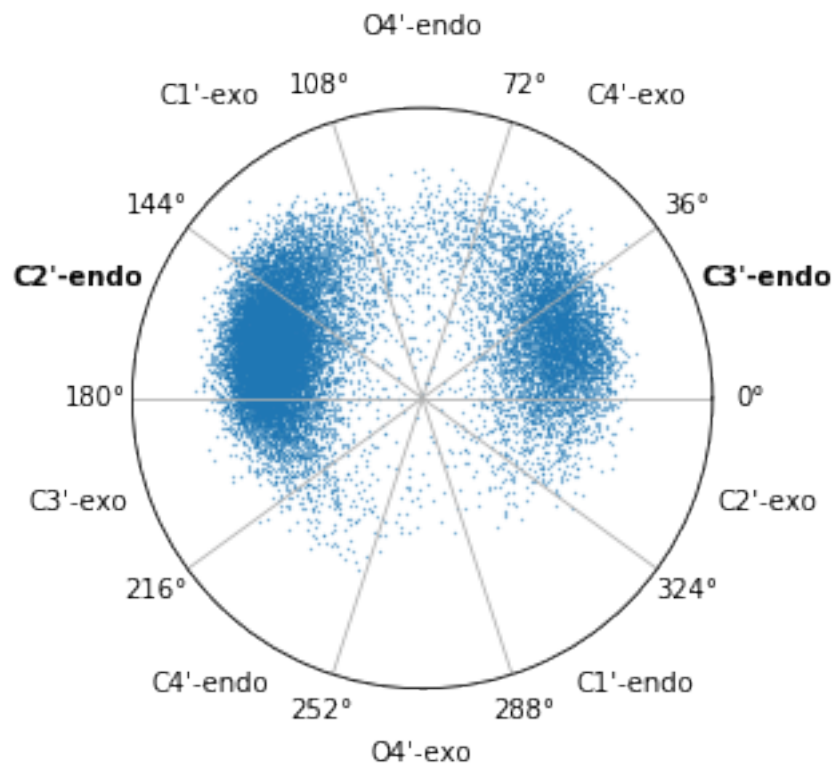
```
angles, residues = bb.pucker_angles(traj, topology=top)
```

again, a list of residues can be specified.

```
In [17]: angles_p, rr = bb.pucker_angles(traj, topology=top, residues=["C_5_0"])
```

```
ax = plt.subplot(111, polar=True)
c = plt.scatter(angles_p[:,0,0], angles_p[:,0,1], s=0.05)
p3 = np.pi/5
plt.ylim(0, 1.2)
xt = np.arange(0, 2*np.pi, p3)
plt.text(0.5*p3, 1.5, "C3'-endo", ha='center', fontweight='bold')
plt.text(1.5*p3, 1.5, "C4'-exo", ha='center')
plt.text(2.5*p3, 1.5, "O4'-endo", ha='center')
plt.text(3.5*p3, 1.5, "C1'-exo", ha='center')
plt.text(4.5*p3, 1.5, "C2'-endo", ha='center', fontweight='bold')
plt.text(5.5*p3, 1.5, "C3'-exo", ha='center')
plt.text(6.5*p3, 1.5, "C4'-endo", ha='center')
plt.text(7.5*p3, 1.5, "O4'-exo", ha='center')
plt.text(8.5*p3, 1.5, "C1'-endo", ha='center')
plt.text(9.5*p3, 1.5, "C2'-exo", ha='center')
plt.xticks(xt)
plt.yticks([])
```

```
# Loading ../test/data/UUCG.xtc
```



6 Calculate 3J scalar couplings

We here calculate 3J scalar couplings for an entire trajectory. The following couplings can be calculated:

In the sugar:

- H1H2. function of torsion angle H1'-C1'-C2'-H2' - H2H3. function of torsion angle H2'-C2'-C3'-H3' - H3H4. function of torsion angle H3'-C3'-C4'-H4'

In the backbone:

- 1H5P, 2H5P, C4Pb. Function of beta torsion angle - 1H5H4, 2H5H4. Function of gamma torsion angle - H3P, C4Pe. Function of epsilon torsion angle

In the nucleobase:

- H1C2/4, H1C6/8. Function of chi torsion angle

by default, all scalar couplings for all residues are calculated.

This means that the output of the `jcouplings` function is a `n x m x 12` array, where

`n` = # of frames

`m` = # of nucleobases

12 = total number of couplings

`rr` is the list of residue names

ATT! it is important that the atom names follow the amber naming conventions. This is tricky for hydrogens. Other names may not be recognized!

```
In [8]: # import barnaba
import barnaba as bb

# define trajectory and topology files
traj = "../test/data/samples.xtc"
top = "../test/data/sample1.pdb"

couplings,rr = bb.jcouplings(traj,topology=top)
print(couplings.shape)
```

```
(101, 71, 12)
```

```
# Loading ../test/data/samples.xtc
```

We now print only the couplings relative to the first residue of the first frame

```
In [9]: from barnaba import definitions
for e in range(1):
    stri = ""
    for k in range(1):
        for l in range(couplings.shape[2]):
            stri += "%10s " % list(definitions.couplings_idx.keys())[l]
            stri += " %10.4f Hz\n " % couplings[e,k,l]
        stri += "\n"
    stri += "\n"
print(stri)
```

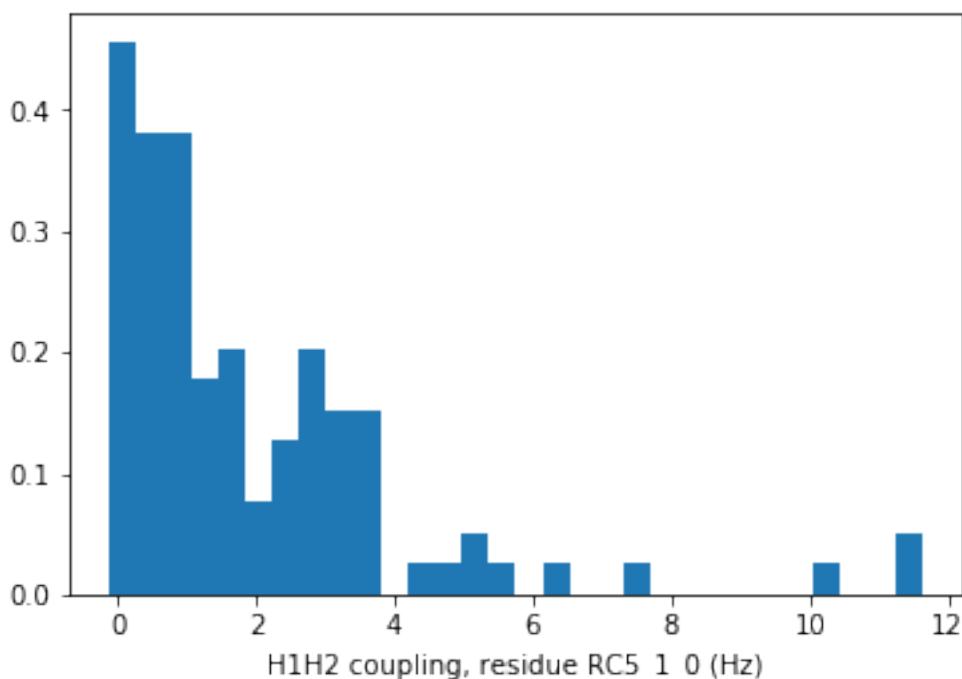
```
H1H2      0.4028 Hz
H2H3      4.7901 Hz
H3H4     10.1837 Hz
1H5P           nan Hz
2H5P           nan Hz
C4Pb           nan Hz
1H5H4      8.7789 Hz
```

2H5H4	0.0109 Hz
H3P	4.1050 Hz
C4Pe	10.7100 Hz
H1C2/4	0.4799 Hz
H1C6/8	2.2005 Hz

But we can also plot the histogram of the H1H2 couplings for all frames

```
In [10]: %matplotlib inline
import matplotlib.pyplot as plt
plt.hist(couplings[:,0,0],bins=30,density=True)
plt.xlabel("H1H2 coupling, residue %s (Hz)" % rr[0])
```

```
Out[10]: Text(0.5,0,'H1H2 coupling, residue RC5_1_0 (Hz)')
```



If the keyword raw=True, the output is the value of the torsion angle, and not the coupling. For example

```
angles,rr = bb.jcouplings(traj,topology=top,raw=True,residues=["RC5_1_0"])
```

returns the angles in radians for all frames of the residue RC3_1_0. As usual, the naming convention is RESNAME_RESNUMBER_CHAININDEX. Note that the shape of the array angles is n x m x 6, where

n = # number of frames

m = # number of residues in the list (in this specific example m=1)

6 = number of torsion angles that are needed for the couplings calculations (H1'-H2',H2'-H3',H3'-H4',beta,gamma,epsilon,chi)

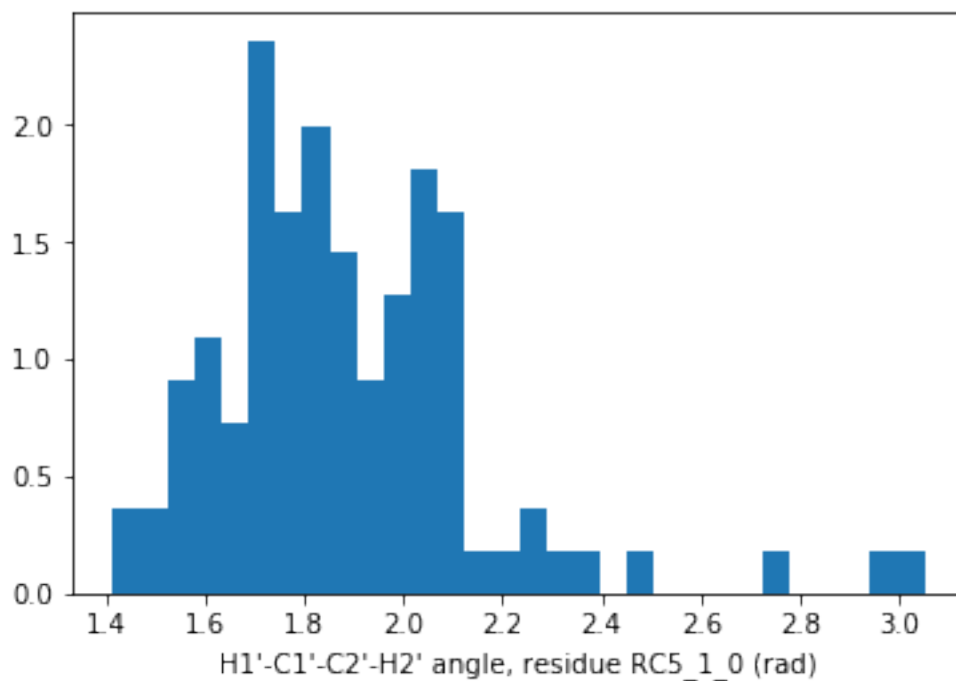
```
In [11]: angles,rr = bb.jcouplings(traj,topology=top,residues=["RC5_1_0"],raw=True)
         print(angles.shape)
```

```
plt.hist(angles[:,0,0],bins=30,density=True)
plt.xlabel("H1'-C1'-C2'-H2' angle, residue %s (rad)" % rr[0])
```

```
(101, 1, 7)
```

```
# Loading ../test/data/samples.xtc
```

```
Out[11]: Text(0.5,0,"H1'-C1'-C2'-H2' angle, residue RC5_1_0 (rad)")
```



7 Search for single-stranded RNA motifs

We will now search for single-stranded motifs within a structure/trajectory. This is performed by using the `ss_motif` function.

```
results = bb.ss_motif(query,target,threshold=0.6,out=None,bulges=0)
```

- *query* is a PDB file with the structure you want to search for within the file *target*. If the keyword *topology* is specified, the query structure is searched in the target trajectory file.
- *threshold* is the eRMSD threshold to consider a substructure in *target* to be significantly similar to *query*. Typical relevant hits have eRMSD in the 0.6-0.9 range.
- If you specify the optional string keyword *out*, PDB structures below the threshold are written with the specified prefix.
- It is possible to specify the maximum number of allowed inserted or bulged bases with the option *bulges*.
- The search is performed not considering the sequence. It is possible to specify a sequence with the *sequence* option. Abbreviations (i.e N/R/Y) are accepted.

The function returns a list of hits. Each element in this list is in turn a list containing the following information: - element 0 is the frame index. This is relevant if the search is performed over a trajectory/multi model PDB. - element 1 is the eRMSD distance from the query - element 2 is the list of residues.

In the following example we search for structures similar to GNRA.pdb in a crystal structure of the H.Marismortui large ribosomal subunit (PDB 1S72).

```
In [3]: import barnaba as bb

# find all GNRA tetraloops in H.Marismortui large ribosomal subunit (PDB 1S72)
query = "../test/data/GNRA.pdb"
target = "../test/data/1S72.pdb"

# call function.
results = bb.ss_motif(query,target,threshold=0.6,out='gnra_loops',bulges=1)

# Loaded query ../test/data/GNRA.pdb
# Loaded target ../test/data/1S72.pdb
# Treating nucleotide 1MA628 as A
# Treating nucleotide 0MU2587 as U
# Treating nucleotide 0MG2588 as G
# Treating nucleotide UR32619 as U
# Treating nucleotide PSU2621 as U
```

Now we print the fragment residues and their eRMSD distance from the query structure.

```
In [4]: for j in range(len(results)):
#seq = "".join([r.split("_")[0] for r in results[j][2]])
print("%2d eRMSD:%5.3f " % (j,results[j][1]))
print("    Sequence: %s" % ",".join(results[j][2]))
print()

0 eRMSD:0.436
  Sequence: C_252_0,U_253_0,C_254_0,A_255_0,C_256_0,G_257_0

1 eRMSD:0.448
  Sequence: U_468_0,G_469_0,U_470_0,G_471_0,A_472_0,A_473_0

2 eRMSD:0.441
```

Sequence: C_576_0,G_577_0,C_578_0,G_579_0,A_580_0,G_581_0

3 eRMSD:0.461
Sequence: G_690_0,G_691_0,A_692_0,A_693_0,A_694_0,C_695_0

4 eRMSD:0.352
Sequence: C_804_0,G_805_0,A_806_0,A_807_0,A_808_0,G_809_0

5 eRMSD:0.555
Sequence: U_1326_0,G_1327_0,A_1328_0,A_1329_0,A_1330_0,A_1331_0

6 eRMSD:0.429
Sequence: G_1628_0,G_1629_0,A_1630_0,A_1631_0,A_1632_0,C_1633_0

7 eRMSD:0.276
Sequence: C_1862_0,G_1863_0,C_1864_0,A_1865_0,A_1866_0,G_1867_0

8 eRMSD:0.210
Sequence: C_2248_0,G_2249_0,G_2250_0,G_2251_0,A_2252_0,G_2253_0

9 eRMSD:0.314
Sequence: C_2411_0,G_2412_0,A_2413_0,A_2414_0,A_2415_0,G_2416_0

10 eRMSD:0.426
Sequence: C_2629_0,G_2630_0,U_2631_0,G_2632_0,A_2633_0,G_2634_0

11 eRMSD:0.350
Sequence: C_2695_0,G_2696_0,A_2697_0,G_2698_0,A_2699_0,G_2700_0

12 eRMSD:0.475
Sequence: G_2876_0,G_2877_0,U_2878_0,A_2879_0,A_2880_0,C_2881_0

13 eRMSD:0.239
Sequence: C_89_1,G_90_1,C_91_1,G_92_1,A_93_1,G_94_1

14 eRMSD:0.488
Sequence: C_1594_0,G_1595_0,A_1597_0,A_1598_0,U_1599_0,G_1600_0

15 eRMSD:0.508
Sequence: U_493_0,C_494_0,A_495_0,G_496_0,A_498_0,G_499_0

16 eRMSD:0.445
Sequence: G_1054_0,G_1055_0,U_1056_0,A_1057_0,A_1058_0,C_1060_0

17 eRMSD:0.302
Sequence: C_1793_0,G_1794_0,G_1795_0,A_1796_0,A_1797_0,G_1799_0

We can also calculate RMSD distances for the different hits

```
In [5]: import glob

        pdbs = glob.glob("gnra_loops*.pdb")
        dists = [bb.rmsd(query,f)[0] for f in pdbs]
```

```

for j in range(len(results)):
    seq = "".join([r.split("_")[0] for r in results[j][2]])
    print("%2d eRMSD:%5.3f RMSD: %6.4f" % (j,results[j][1],10.*dists[j]), end="")
    print("    Sequence: %s" % seq)

    #print "%50s %6.4f AA" % (f,10.*dist[0])

```

```

0 eRMSD:0.436 RMSD: 0.7048    Sequence: CUCACG
1 eRMSD:0.448 RMSD: 0.6999    Sequence: UGUGAA
2 eRMSD:0.441 RMSD: 0.7403    Sequence: CGCGAG
3 eRMSD:0.461 RMSD: 0.6419    Sequence: GGAAAC
4 eRMSD:0.352 RMSD: 0.8012    Sequence: CGAAAG
5 eRMSD:0.555 RMSD: 0.8040    Sequence: UGAAAA
6 eRMSD:0.429 RMSD: 0.8208    Sequence: GGAAAC
7 eRMSD:0.276 RMSD: 0.7992    Sequence: CGCAAG
8 eRMSD:0.210 RMSD: 0.6920    Sequence: CGGGAG
9 eRMSD:0.314 RMSD: 0.7170    Sequence: CGAAAG
10 eRMSD:0.426 RMSD: 0.6651    Sequence: CGUGAG
11 eRMSD:0.350 RMSD: 0.6953    Sequence: CGAGAG
12 eRMSD:0.475 RMSD: 0.9468    Sequence: GGUAAC
13 eRMSD:0.239 RMSD: 0.6417    Sequence: CGCGAG
14 eRMSD:0.488 RMSD: 2.2815    Sequence: CGAAUG
15 eRMSD:0.508 RMSD: 1.4962    Sequence: UCAGAG
16 eRMSD:0.445 RMSD: 1.0643    Sequence: GGUAAC
17 eRMSD:0.302 RMSD: 1.2767    Sequence: CGGAAG

```

Note that the first hit has a low eRMSD, but no GNRA sequence. Let's have a look at this structure:

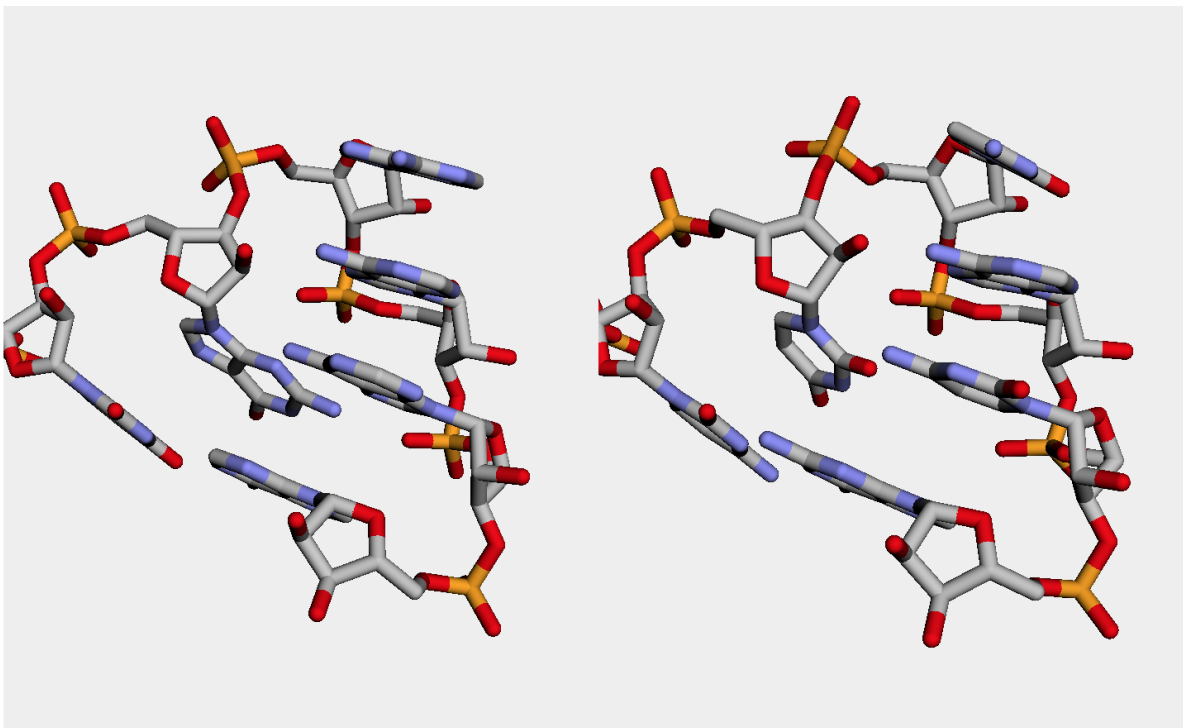
```
In [6]: import py3Dmol
```

```

query_s = open(query, 'r').read()
hit_0 = open(pdb[0], 'r').read()

p = py3Dmol.view(width=900,height=600,viewergrid=(1,2))
p.addModel(query_s, 'pdb', viewer=(0,0))
p.addModel(hit_0, 'pdb', viewer=(0,1))
p.setStyle({'stick':{}})
p.setBackgroundColor('Oxeeeeee')
p.zoomTo()
p.show()

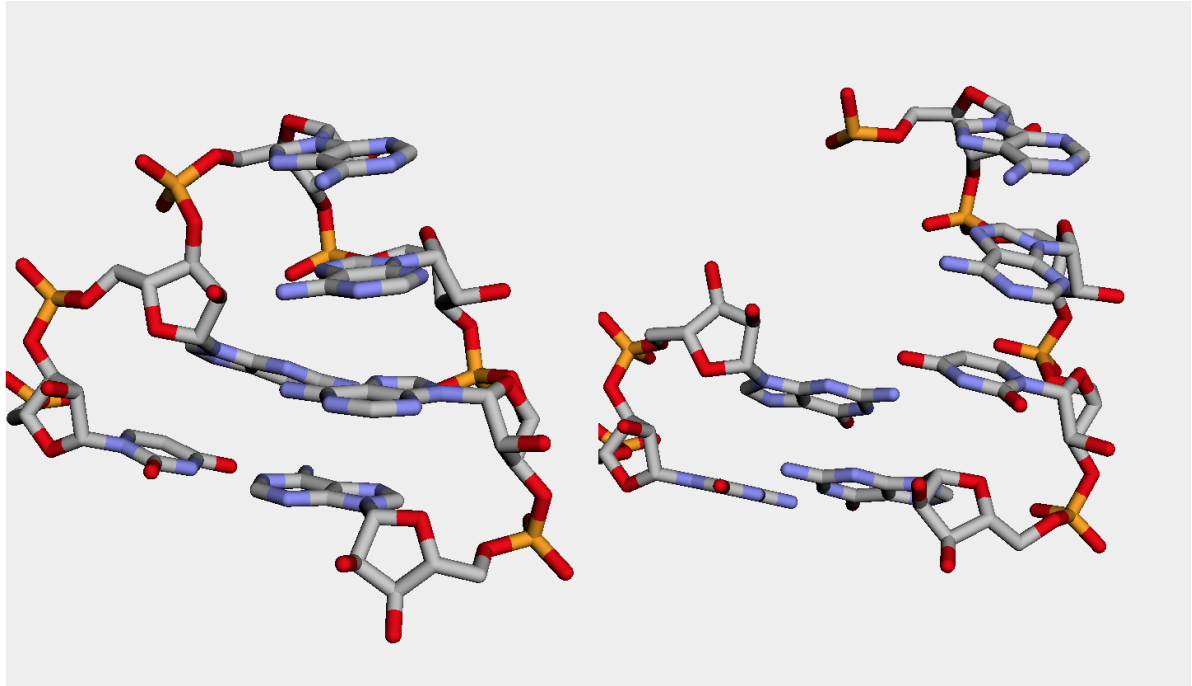
```



We can also check hit 14, that has low eRMSD but (relatively) high RMSD

```
In [7]: hit_14 = open(pdb[14], 'r').read()
```

```
p = py3Dmol.view(width=900,height=600,viewergrid=(1,2))
p.addModel(query_s, 'pdb', viewer=(0,0))
p.addModel(hit_14, 'pdb', viewer=(0,1))
p.setStyle({'stick':{}})
p.setBackgroundColor('0xeeeeee')
p.zoomTo()
p.show()
```



8 Search for double-stranded RNA motif in database

We will now search for a double-stranded motif within the crystal structure of the large ribosomal subunit.

This can be performed using the `ds_motif` function, that is very similar to the function `ss_motif`. It is necessary to specify the number of nucleotides in the first (l1) and second (l2) strand.

It is possible to specify the maximum number of allowed inserted or bulged residues with the option *bulges*.

In the example we use a threshold in ERMSD of 0.6 - relevant hits have a distance between 0.6-0.9 eRMSD. If you specify the optional keyword *out*, PDB structures are written to disk.

By default the search is performed not considering the sequence. It is possible to specify a sequence with the sequence option. abbreviations (i.e N/R/Y) are accepted.

```
In [2]: import barnaba as bb
```

```
# find all SARCIN motifs in H.Marismortui large ribosomal subunit (PDB 1S72)

query = "../test/data/SARCIN.pdb"
pdb = "../test/data/1S72.pdb"

# call function.
results = bb.ds_motif(query,pdb,l1=8,l2=7,bulges=0,threshold=0.7,out='sarcin_motif')
```

```
# Loaded query ../test/data/SARCIN.pdb
# Loaded target ../test/data/1S72.pdb
# Treating nucleotide 1MA628 as A
# Treating nucleotide 0MU2587 as U
# Treating nucleotide 0MG2588 as G
# Treating nucleotide UR32619 as U
# Treating nucleotide PSU2621 as U
```

Now we print distances and sequences

```
In [3]: import glob
```

```
pdb = glob.glob("sarcin*.pdb")

for j in range(len(results)):
    seq = ",".join([r for r in results[j][2]])
    print("%2d eRMSD:%5.3f" % (j,results[j][1]))
    print("    Sequence: %s" % seq)
```

```
0 eRMSD:0.394
Sequence: C_171_0,U_172_0,C_173_0,A_174_0,G_175_0,U_176_0,A_177_0,U_178_0,
A_158_0,G_159_0,A_160_0,A_161_0,C_162_0,U_163_0,G_164_0
1 eRMSD:0.511
Sequence: G_209_0,U_210_0,U_211_0,A_212_0,G_213_0,U_214_0,A_215_0,A_216_0,
U_224_0,G_225_0,A_226_0,A_227_0,C_228_0,G_229_0,C_230_0
2 eRMSD:0.602
Sequence: A_354_0,C_355_0,C_356_0,A_357_0,G_358_0,U_359_0,A_360_0,C_361_0,
C_291_0,G_292_0,A_293_0,C_294_0,C_295_0,G_296_0,U_297_0
3 eRMSD:0.599
Sequence: U_584_0,C_585_0,C_586_0,A_587_0,G_588_0,U_589_0,A_590_0,A_591_0,
U_567_0,G_568_0,A_569_0,C_570_0,C_571_0,G_572_0,A_573_0
4 eRMSD:0.572
Sequence: C_1366_0,A_1367_0,U_1368_0,A_1369_0,G_1370_0,U_1371_0,A_1372_0,G_1373_0,
```

```

U_2052_0,G_2053_0,A_2054_0,A_2055_0,C_2056_0,U_2057_0,G_2058_0
5 eRMSD:0.481
Sequence: U_2688_0,A_2689_0,U_2690_0,A_2691_0,G_2692_0,U_2693_0,A_2694_0,C_2695_0,
G_2700_0,G_2701_0,A_2702_0,A_2703_0,C_2704_0,U_2705_0,A_2706_0
6 eRMSD:0.536
Sequence: G_74_1,G_75_1,G_76_1,A_77_1,G_78_1,U_79_1,A_80_1,C_81_1,
G_101_1,G_102_1,A_103_1,A_104_1,A_105_1,C_106_1,C_107_1

```

Finally, we visualize the query and the first hit

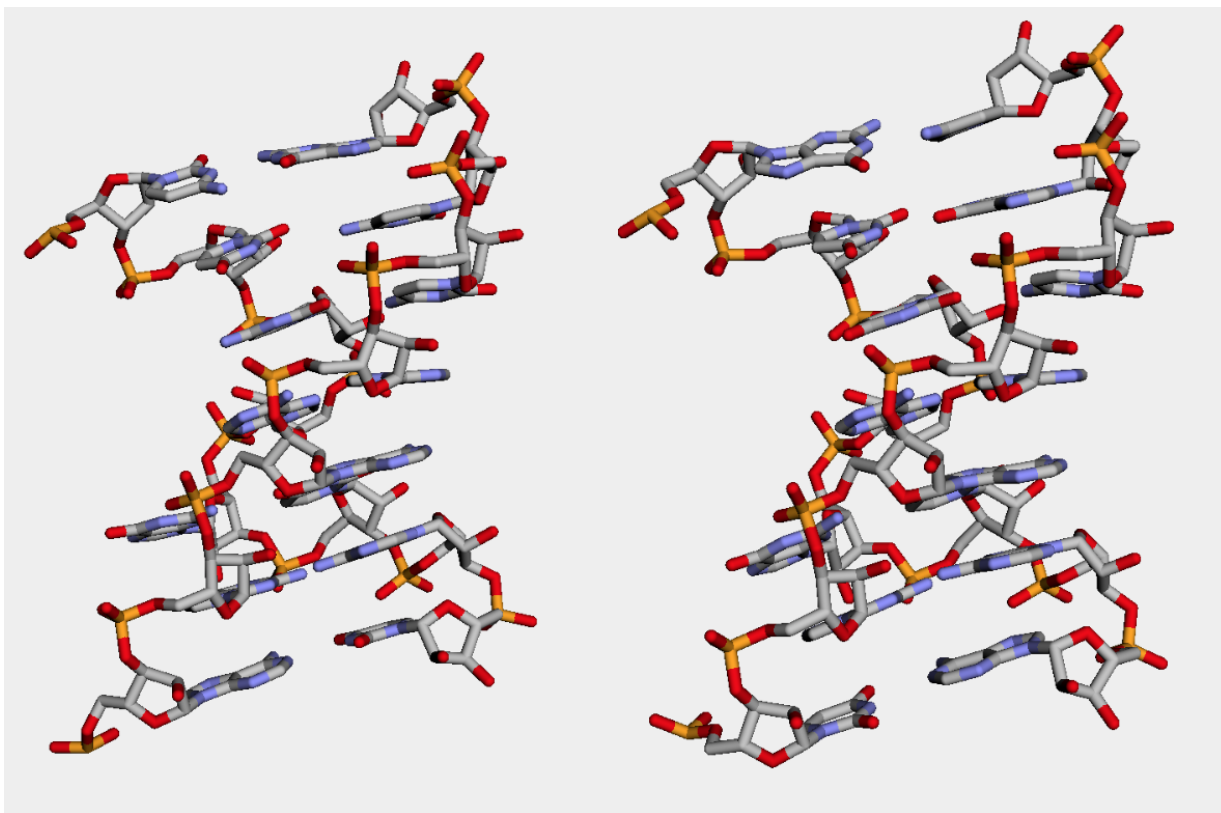
```
In [4]: import py3Dmol
```

```

query_s = open(query, 'r').read()
hit_1 = open(pdb[1], 'r').read()

p = py3Dmol.view(width=900,height=600,viewergrid=(1,2))
#p = py3Dmol.view(width=900,height=600)
#p.addModel(query_s, 'pdb')
p.addModel(query_s, 'pdb', viewer=(0,0))
p.addModel(hit_1, 'pdb', viewer=(0,1))
p.setStyle({'stick':{}})
p.setBackgroundColor('0xeeeeee')
p.zoomTo()
p.show()

```



```
In [5]: # annotate native
stackings_query, pairings_query, res_query = bb.annotate(query)
print("Query BASE-PAIR")
```

```

for p in range(len(pairings_query[0][0])):
    print(res_query[pairings_query[0][0][p][0]], end=" ")
    print(res_query[pairings_query[0][0][p][1]], end=" ")
    print(pairings_query[0][1][p])

print()
stackings_hit, pairings_hit, res_hit = bb.annotate(pdb[1])

print("Hit 2 base-pairs")
for p in range(len(pairings_hit[0][0])):
    print(res_hit[pairings_hit[0][0][p][0]], end=" ")
    print(res_hit[pairings_hit[0][0][p][1]], end=" ")
    print(pairings_hit[0][1][p])

print()

```

```

Query BASE-PAIR
C_6_0 G_24_0 WCc
U_7_0 C_23_0 WHc
C_8_0 C_22_0 SHt
A_9_0 A_21_0 HHt
G_10_0 U_11_0 SHc
U_11_0 A_20_0 WHt
A_12_0 G_19_0 HSc
U_13_0 A_18_0 WCc

```

```

Hit 2 base-pairs
G_209_0 C_230_0 WCc
U_210_0 G_229_0 GUc
U_211_0 C_228_0 SHt
A_212_0 A_227_0 HHt
G_213_0 U_214_0 SHc
U_214_0 A_226_0 WHt
A_215_0 G_225_0 HSc
A_216_0 U_224_0 WCc

```

```

# Loading ../test/data/SARCIN.pdb
# Loading sarcin_motif_00002_U_0.pdb

```

9 Snippet utility: extract fragments from structures with a given sequence

We here show how to extract fragments from PDB files with a given sequence. Given a PDB file (trajectories are NOT supported), it is possible to extract fragments with a given sequence using the function

```
bb.snippet(fname, sequence, outdir=outdir)
```

PDB fragments are written in the directory outdir. See below an example where we extract all fragments with sequence GNRA from the structure of the large ribosomal subunit

```
In [2]: import barnaba as bb
import os

fname = "../test/data/1S72.pdb"
sequence = "GNRA"
os.system("mkdir -p snippet")
bb.snippet(fname, sequence, outdir="snippet")
```

```
# Initializing file ../test/data/1S72.pdb
```

We now list the files and print the sequence:

```
In [3]: import glob

flist = glob.glob("snippet/*.pdb")
for f in flist:
    seq = [line[17:20].strip() for line in open(f) if line[12:16].strip()=="C2"]
    print("%-40s %4s" % (f, "".join(seq)))

print("Total number of fragments with sequence GNRA:", len(flist))
```

```
snippet/1S72_100_G_9_0_00146.pdb      GGGA
snippet/1S72_101_G_9_0_00147.pdb      GGAA
snippet/1S72_102_G_9_0_00148.pdb      GAAA
snippet/1S72_1037_G_0_0_00055.pdb     GGGA
snippet/1S72_1055_G_0_0_00056.pdb     GUAA
snippet/1S72_1075_G_0_0_00057.pdb     GGGA
snippet/1S72_1076_G_0_0_00058.pdb     GGAA
snippet/1S72_1087_G_0_0_00059.pdb     GAGA
snippet/1S72_1121_G_0_0_00060.pdb     GUAA
snippet/1S72_1158_G_0_0_00061.pdb     GGGA
snippet/1S72_1163_G_0_0_00062.pdb     GUGA
snippet/1S72_116_G_0_0_00003.pdb      GAGA
snippet/1S72_1190_G_0_0_00063.pdb     GAAA
snippet/1S72_1197_G_0_0_00064.pdb     GUAA
snippet/1S72_1239_G_0_0_00065.pdb     GGGA
snippet/1S72_1258_G_0_0_00066.pdb     GAGA
snippet/1S72_1284_G_0_0_00067.pdb     GUAA
snippet/1S72_1315_G_0_0_00068.pdb     GGAA
snippet/1S72_1325_G_0_0_00069.pdb     GUGA
snippet/1S72_1327_G_0_0_00070.pdb     GAAA
snippet/1S72_1349_G_0_0_00071.pdb     GUGA
snippet/1S72_1354_G_0_0_00072.pdb     GAAA
snippet/1S72_1376_G_0_0_00073.pdb     GCGA
```

snippet/1S72_1387_G_0_0_00074.pdb	GUGA
snippet/1S72_1389_G_0_0_00075.pdb	GAGA
snippet/1S72_142_G_0_0_00004.pdb	GCAA
snippet/1S72_1468_G_0_0_00076.pdb	GCAA
snippet/1S72_1484_G_0_0_00077.pdb	GAAA
snippet/1S72_1489_G_0_0_00078.pdb	GGGA
snippet/1S72_1490_G_0_0_00079.pdb	GGAA
snippet/1S72_1491_G_0_0_00080.pdb	GAAA
snippet/1S72_149_G_0_0_00005.pdb	GGAA
snippet/1S72_1523_G_0_0_00081.pdb	GUGA
snippet/1S72_1525_G_0_0_00082.pdb	GAAA
snippet/1S72_157_G_0_0_00006.pdb	GAGA
snippet/1S72_1588_G_0_0_00083.pdb	GGAA
snippet/1S72_1595_G_0_0_00084.pdb	GUAA
snippet/1S72_1604_G_0_0_00085.pdb	GGAA
snippet/1S72_1627_G_0_0_00086.pdb	GGGA
snippet/1S72_1628_G_0_0_00087.pdb	GGAA
snippet/1S72_1629_G_0_0_00088.pdb	GAAA
snippet/1S72_1634_G_0_0_00089.pdb	GUGA
snippet/1S72_164_G_0_0_00007.pdb	GAAA
snippet/1S72_1655_G_0_0_00090.pdb	GAAA
snippet/1S72_1681_G_0_0_00091.pdb	GAGA
snippet/1S72_1707_G_0_0_00092.pdb	GCGA
snippet/1S72_1709_G_0_0_00093.pdb	GAAA
snippet/1S72_1726_G_0_0_00094.pdb	GGGA
snippet/1S72_1730_G_0_0_00095.pdb	GCAA
snippet/1S72_1743_G_0_0_00096.pdb	GGGA
snippet/1S72_1744_G_0_0_00097.pdb	GGAA
snippet/1S72_1752_G_0_0_00098.pdb	GCAA
snippet/1S72_1773_G_0_0_00099.pdb	GGAA
snippet/1S72_1780_G_0_0_00100.pdb	GGGA
snippet/1S72_1794_G_0_0_00101.pdb	GGAA
snippet/1S72_180_G_0_0_00008.pdb	GGGA
snippet/1S72_1812_G_0_0_00102.pdb	GUGA
snippet/1S72_1819_G_0_0_00103.pdb	GGAA
snippet/1S72_1837_G_0_0_00104.pdb	GUAA
snippet/1S72_1849_G_0_0_00105.pdb	GUGA
snippet/1S72_184_G_0_0_00009.pdb	GGAA
snippet/1S72_1855_G_0_0_00106.pdb	GCAA
snippet/1S72_1863_G_0_0_00107.pdb	GCAA
snippet/1S72_190_G_0_0_00010.pdb	GAAA
snippet/1S72_196_G_0_0_00011.pdb	GCAA
snippet/1S72_201_G_0_0_00012.pdb	GUGA
snippet/1S72_2051_G_0_0_00108.pdb	GUGA
snippet/1S72_2080_G_0_0_00109.pdb	GAGA
snippet/1S72_2092_G_0_0_00110.pdb	GGGA
snippet/1S72_2093_G_0_0_00111.pdb	GGAA
snippet/1S72_2097_G_0_0_00112.pdb	GCGA
snippet/1S72_213_G_0_0_00013.pdb	GUAA
snippet/1S72_219_G_0_0_00014.pdb	GCGA
snippet/1S72_223_G_0_0_00015.pdb	GUGA
snippet/1S72_2249_G_0_0_00113.pdb	GGGA
snippet/1S72_2299_G_0_0_00114.pdb	GAAA
snippet/1S72_229_G_0_0_00016.pdb	GCGA

snippet/1S72_2337_G_0_0_00115.pdb	GGGA
snippet/1S72_2350_G_0_0_00116.pdb	GCGA
snippet/1S72_2359_G_0_0_00117.pdb	GCAA
snippet/1S72_2365_G_0_0_00118.pdb	GCAA
snippet/1S72_2399_G_0_0_00119.pdb	GGAA
snippet/1S72_2410_G_0_0_00120.pdb	GCGA
snippet/1S72_2412_G_0_0_00121.pdb	GAAA
snippet/1S72_2426_G_0_0_00122.pdb	GCGA
snippet/1S72_2453_G_0_0_00123.pdb	GCAA
snippet/1S72_2466_G_0_0_00124.pdb	GAAA
snippet/1S72_2480_G_0_0_00125.pdb	GGGA
snippet/1S72_2501_G_0_0_00126.pdb	GCAA
snippet/1S72_2574_G_0_0_00127.pdb	GCAA
snippet/1S72_2580_G_0_0_00128.pdb	GUGA
snippet/1S72_259_G_0_0_00017.pdb	GCAA
snippet/1S72_2609_G_0_0_00129.pdb	GUGA
snippet/1S72_2630_G_0_0_00130.pdb	GUGA
snippet/1S72_2632_G_0_0_00131.pdb	GAGA
snippet/1S72_2696_G_0_0_00132.pdb	GAGA
snippet/1S72_2700_G_0_0_00133.pdb	GGAA
snippet/1S72_2738_G_0_0_00134.pdb	GAGA
snippet/1S72_2740_G_0_0_00135.pdb	GAGA
snippet/1S72_2773_G_0_0_00136.pdb	GUAA
snippet/1S72_2798_G_0_0_00137.pdb	GAAA
snippet/1S72_2809_G_0_0_00138.pdb	GGAA
snippet/1S72_2810_G_0_0_00139.pdb	GAAA
snippet/1S72_2815_G_0_0_00140.pdb	GAGA
snippet/1S72_2877_G_0_0_00141.pdb	GUAA
snippet/1S72_2882_G_0_0_00142.pdb	GAGA
snippet/1S72_314_G_0_0_00018.pdb	GGAA
snippet/1S72_324_G_0_0_00019.pdb	GUGA
snippet/1S72_334_G_0_0_00020.pdb	GUGA
snippet/1S72_351_G_0_0_00021.pdb	GAGA
snippet/1S72_404_G_0_0_00022.pdb	GCGA
snippet/1S72_426_G_0_0_00023.pdb	GCGA
snippet/1S72_446_G_0_0_00024.pdb	GAGA
snippet/1S72_456_G_0_0_00025.pdb	GUGA
snippet/1S72_469_G_0_0_00026.pdb	GUGA
snippet/1S72_482_G_0_0_00027.pdb	GCAA
snippet/1S72_499_G_0_0_00028.pdb	GGGA
snippet/1S72_49_G_9_0_00143.pdb	GGAA
snippet/1S72_504_G_0_0_00029.pdb	GCGA
snippet/1S72_506_G_0_0_00030.pdb	GAAA
snippet/1S72_518_G_0_0_00031.pdb	GAAA
snippet/1S72_529_G_0_0_00032.pdb	GCGA
snippet/1S72_537_G_0_0_00033.pdb	GCGA
snippet/1S72_577_G_0_0_00034.pdb	GCGA
snippet/1S72_588_G_0_0_00035.pdb	GUAA
snippet/1S72_599_G_0_0_00036.pdb	GGGA
snippet/1S72_600_G_0_0_00037.pdb	GGAA
snippet/1S72_627_G_0_0_00038.pdb	GAAA
snippet/1S72_640_G_0_0_00039.pdb	GGGA
snippet/1S72_64_G_0_0_00000.pdb	GCGA
snippet/1S72_657_G_0_0_00040.pdb	GCAA

snippet/1S72_679_G_0_0_00041.pdb	GGGA
snippet/1S72_689_G_0_0_00042.pdb	GGGA
snippet/1S72_690_G_0_0_00043.pdb	GGAA
snippet/1S72_691_G_0_0_00044.pdb	GAAA
snippet/1S72_743_G_0_0_00045.pdb	GGGA
snippet/1S72_74_G_9_0_00144.pdb	GGGA
snippet/1S72_77_G_0_0_00001.pdb	GGGA
snippet/1S72_805_G_0_0_00046.pdb	GAAA
snippet/1S72_816_G_0_0_00047.pdb	GGAA
snippet/1S72_854_G_0_0_00048.pdb	GUGA
snippet/1S72_871_G_0_0_00049.pdb	GUGA
snippet/1S72_873_G_0_0_00050.pdb	GAAA
snippet/1S72_892_G_0_0_00051.pdb	GCAA
snippet/1S72_90_G_9_0_00145.pdb	GCGA
snippet/1S72_911_G_0_0_00052.pdb	GAAA
snippet/1S72_92_G_0_0_00002.pdb	GCGA
snippet/1S72_948_G_0_0_00053.pdb	GUGA
snippet/1S72_958_G_0_0_00054.pdb	GCGA

Total number of fragments with sequence GNRA: 149

We show in the example_cluster how to visualise and analyse these data

10 Clustering RNA structures

We start by clustering the structures obtained from the previous example “example_08_snippet.ipynb”, where we extracted all fragments with sequence GNRA from the PDB of the large ribosomal subunit.

First, we calculate the g-vectors for all PDB files

```
In [11]: import glob
import barnaba as bb
import numpy as np

flist = glob.glob("snippet/*.pdb")
if(len(flist)==0):
    print("# You need to run the example example8_snippet.ipynb")
    exit()

# calculate G-VECTORS for all files
gvecs = []
for f in flist:
    gvec,seq = bb.dump_gvec(f)
    assert len(seq)==4
    gvecs.extend(gvec)
```

Then, we reshape the array so that has the dimension $(N, n * 4 * 4)$, where N is the number of frames and n is the number of nucleotides

```
In [12]: gvecs = np.array(gvecs)
gvecs = gvecs.reshape(149,-1)
print(gvecs.shape)
```

(149, 64)

C. We project the data using a simple principal component analysis on the g-vectors

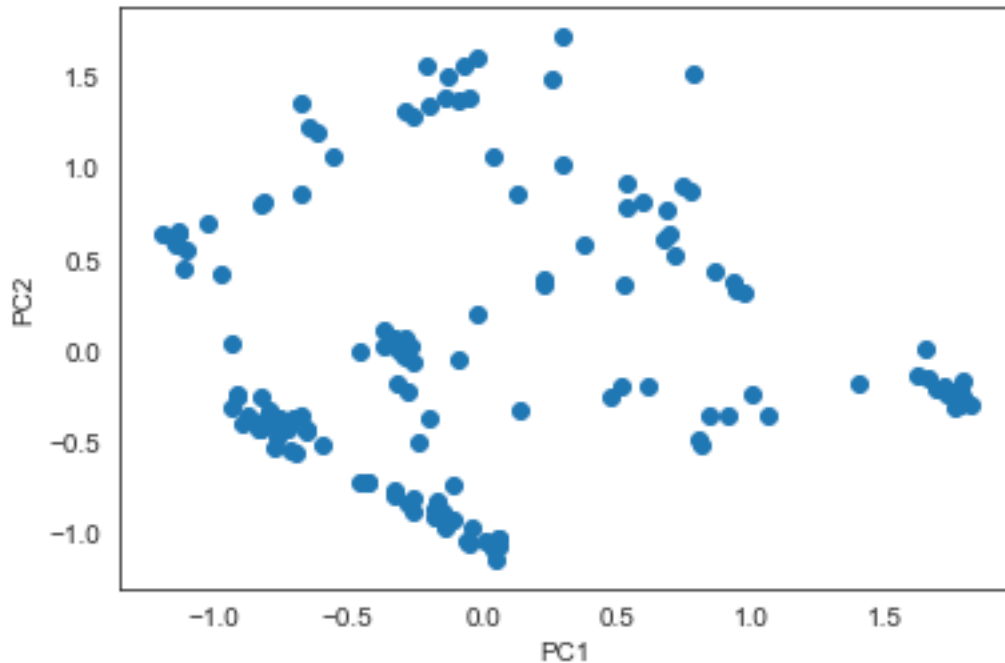
```
In [13]: import barnaba.cluster as cc
# calculate PCA
v,w = cc.pca(gvecs,nevecs=3)
print("# Explained variance 1=%5.1f 2:=%5.1f 3=%5.1f" % (v[0]*100,v[1]*100,v[2]*100))
```

Cumulative explained variance of component: 1= 26.6 2:= 48.7 3= 64.9

```
In [14]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")

plt.scatter(w[:,0],w[:,1])
plt.xlabel("PC1")
plt.ylabel("PC2")
```

Out[14]: Text(0,0.5, 'PC2')



D. We make use of DBSCAN in sklearn to perform clustering. The function `cc.dbSCAN` takes four arguments:

- i. the list of G-vectors `gvec`
- ii. the list of labels for each point
- iii. the `eps` value
- iv. `min_samples`
- v. (optional) the weight of the samples for non-uniform clustering

The function outputs some information on the clustering: the number of clusters, the number of samples assigned to clusters (non noise), and silhouette.

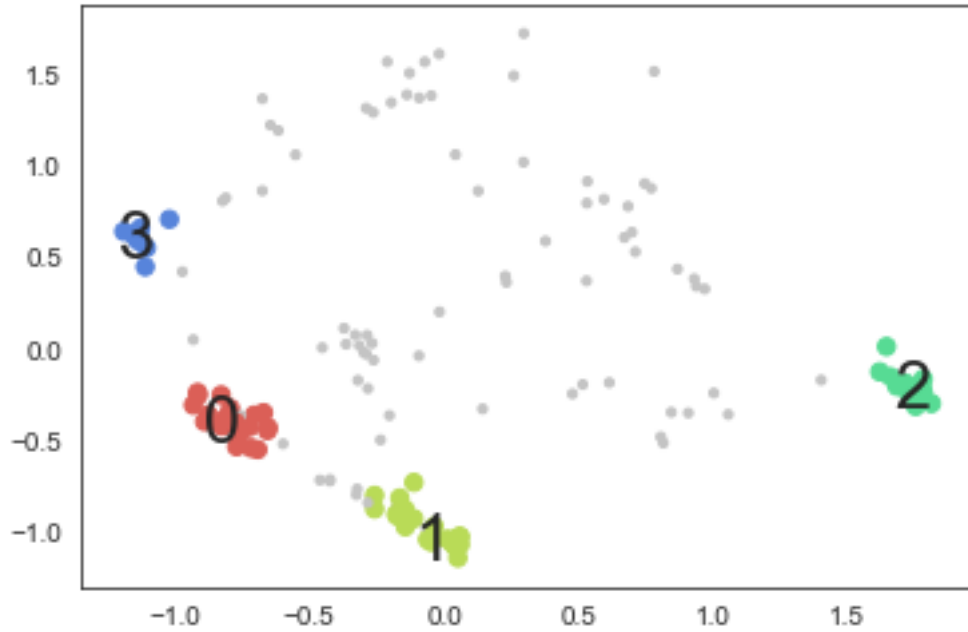
For each cluster it reports the size, the maximum eRMSD distance between samples in a cluster (IC=intra-cluster), the median intra-cluster eRMSD, the maximum and median distance from the centroid.

```
In [15]: new_labels, center_idx = cc.dbSCAN(gvecs, range(gvecs.shape[0]), eps=0.35, min_samples=8)

# eps:0.700 min_samples:8 nclusters: 4
# silhouette score: 0.1728
# Avg silhouette: 0.6111
# assigned samples :71 total samples:149
# N size      max eRMSD (IC)      med eRMSD (IC) max eRMSD (centroid) med eRMSD (centroid) center
# 00 0026      0.613      0.309      0.440      0.222 00 118
# 01 0022      0.553      0.337      0.372      0.280 01 11
# 02 0015      0.543      0.286      0.370      0.229 02 4
# 03 0008      0.453      0.228      0.311      0.158 03 8
```

We can now color the PCA according to the different cluster and display the centroid as a label:

```
In [16]: cp = sns.color_palette("hls", len(center_idx)+1)
colors = [cp[j-1] if(j!=0) else (0.77,0.77,0.77) for j in new_labels]
size = [40 if(j!=0) else 10 for j in new_labels]
#do scatterplot
plt.scatter(w[:,0], w[:,1], s=size, c=colors)
for i, k in enumerate(center_idx):
    plt.text(w[k,0], w[k,1], str(i), ha='center', va='center', fontsize=25)
```



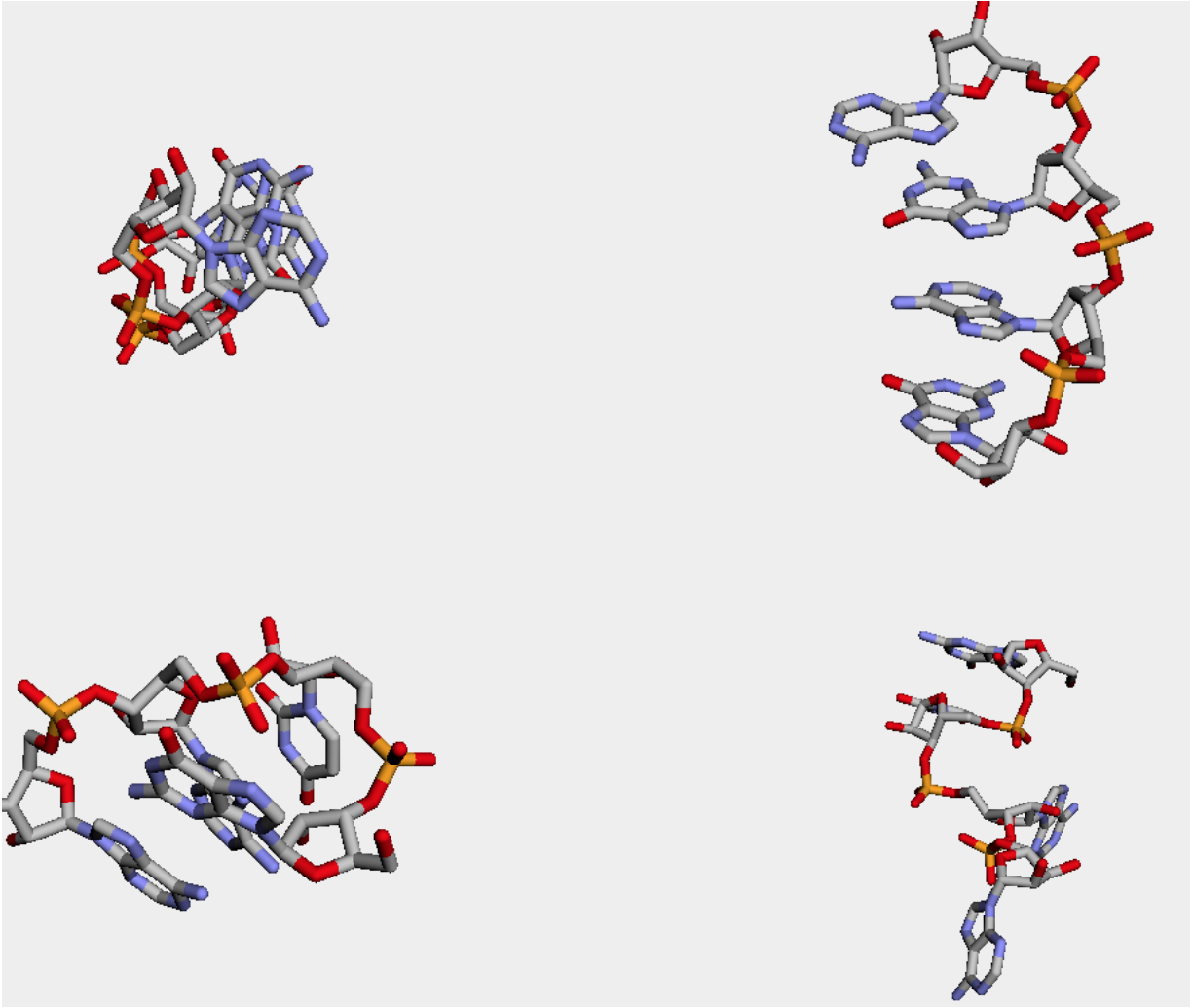
E. We finally visualise the 4 centroids:

In [17]: `import py3Dmol`

```
cluster_0 = open(flist[center_idx[0]], 'r').read()
cluster_1 = open(flist[center_idx[1]], 'r').read()
cluster_2 = open(flist[center_idx[2]], 'r').read()
cluster_3 = open(flist[center_idx[3]], 'r').read()

p = py3Dmol.view(width=900,height=600,viewergrid=(2,2))
#p = py3Dmol.view(width=900,height=600)
#p.addModel(query_s, 'pdb')
p.addModel(cluster_0, 'pdb', viewer=(0,0))
p.addModel(cluster_1, 'pdb', viewer=(0,1))
p.addModel(cluster_2, 'pdb', viewer=(1,0))
p.addModel(cluster_3, 'pdb', viewer=(1,1))

#p.addModel(hit_0, 'pdb', viewer=(0,1))
p.setStyle({'stick':{}})
p.setBackgroundColor('0xeeeeee')
p.zoomTo()
p.show()
```



It is interesting to observe that cluster 0 and 1 (in close proximity in the PCA projection), correspond to A-form-like structures. Cluster 2 corresponds to the classic GNRA fold.

12 Elastic Network Model

Here we show how to use BaRNABA to construct an elastic network model (ENM) of an RNA molecule.

```
In [2]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# import barnaba
import barnaba.enm as enm

# define the input file
fname = "../test/data/sample1.pdb"
```

12.1 1: ENM construction

```
In [3]: %time enm_obj=enm.Enm(fname,sparse=False)

# Read (212, 3) coordinates
CPU times: user 345 ms, sys: 27.7 ms, total: 373 ms
Wall time: 291 ms
```

The input parameter `>sele_atoms`

enables the user to choose which atoms she/he wants to use as beads when constructing the ENM. Standard options are: - S-ENM (C1'): best 1-bead choice [2,3] - SBP-ENM (C1', C2, P): optimal compromise between accuracy and computational burden [3] - AA-ENM (all heavy (non-hydrogen) atoms): best level of accuracy [3,4]

For each of these beads choices there is an optimal cutoff radius (see the analysis in [2] for more details).

The default model implemented by BaRNABA is SBP-ENM, with a cutoff of 0.9 nm.

Let's try to build an AA-ENM

```
In [4]: %time enm_AA=enm.Enm(fname,sele_atoms="AA",cutoff=0.7)

# Read (1499, 3) coordinates
CPU times: user 35.4 s, sys: 588 ms, total: 36 s
Wall time: 22.3 s
```

We can see that this takes considerably more time compared to the 3-beads choice.

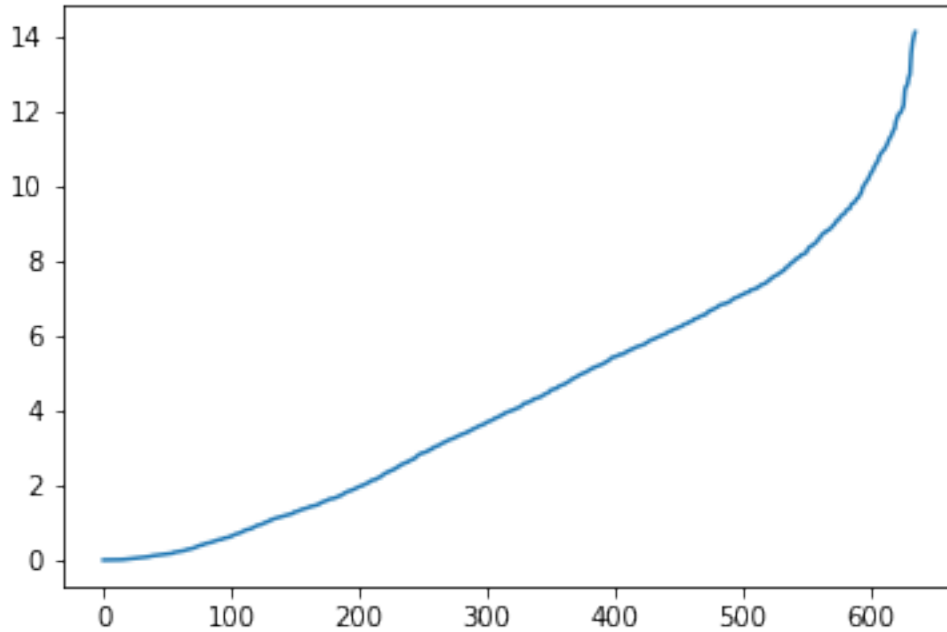
12.2 2: Eigenvalues and eigenvectors

Let's take a look of the eigenvalues:

```
In [5]: e_val=enm_obj.get_eval()

In [6]: plt.plot(e_val)

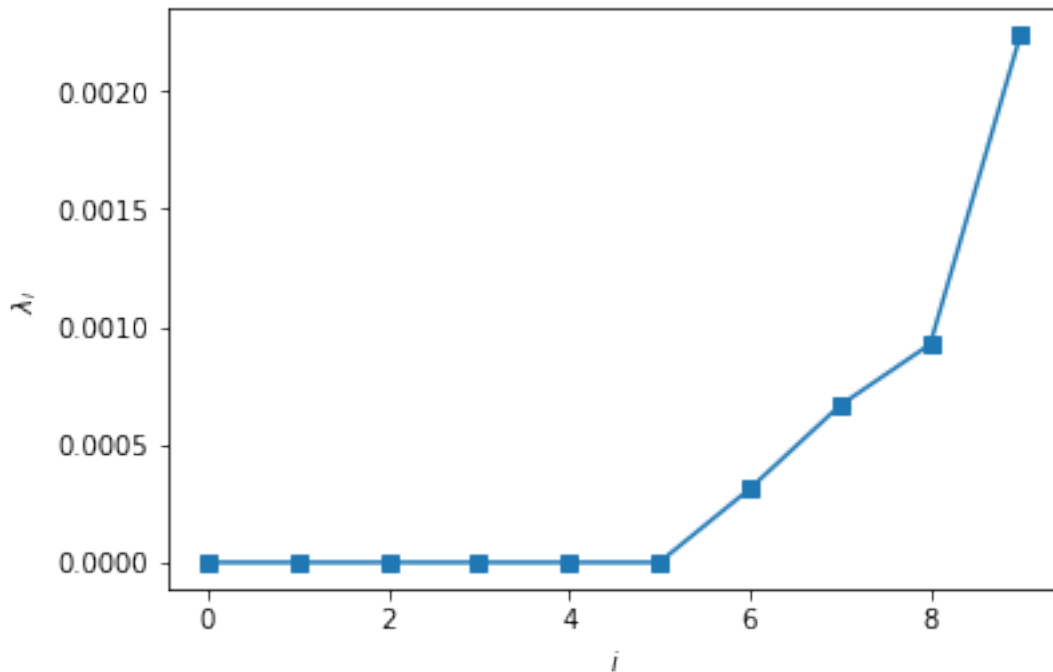
Out[6]: [<matplotlib.lines.Line2D at 0xa132a4490>]
```



We are usually not interested in the whole spectrum. In particular we can focus on the first (smallest) eigenvalues. 6 of them will be equal to zero, they correspond to the rototranslational null modes of the system:

```
In [7]: plt.plot(e_val[:10],marker='s')
        plt.ylabel(r'\lambda_i$')
        plt.xlabel('$i$')
```

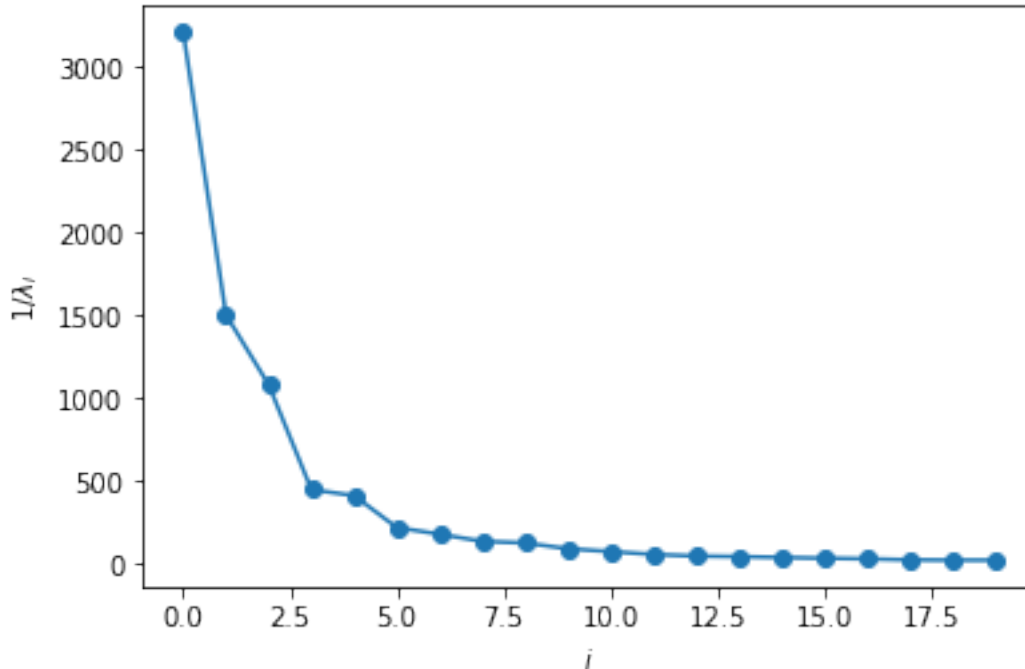
Out[7]: Text(0.5,0, '\$i\$')



After these, we have the eigenvalues representing the non-null normal modes of the system. Their amplitude is given by the inverse of the associated eigenvalue: $\sigma_i = 1/\lambda_i$

```
In [8]: plt.plot(1/e_val[6:26],marker='o')
plt.ylabel(r'$1/\lambda_i$')
plt.xlabel('$i$')
```

```
Out[8]: Text(0.5,0,'$i$')
```



12.3 3: Mean square fluctuations

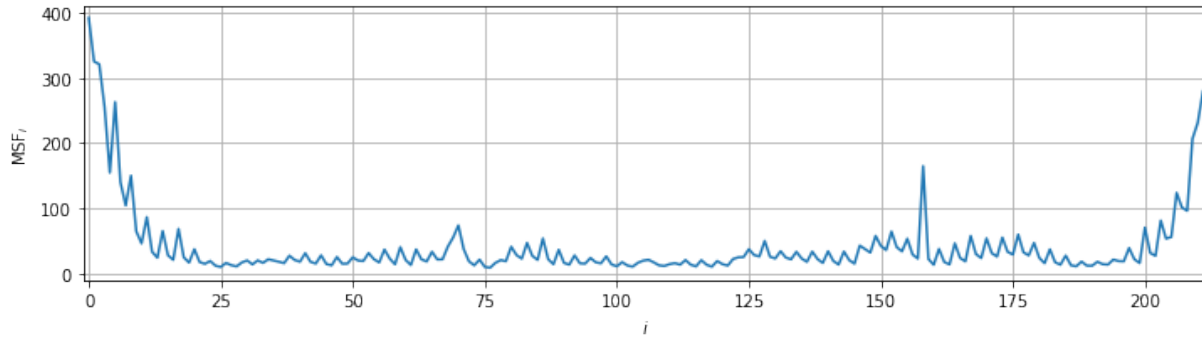
The first information we can obtain from the ENM are the mean square fluctuations (MSF) of its beads. This can be easily computed as:

$$(MSF_i = \langle \delta \mathbf{r}_i^2 \rangle = \sum_{\alpha} 1/\lambda_{\alpha} \sum_{\mu} v_{i,\mu}^{\alpha} v_{i,\mu}^{\alpha})$$

where $i = 1, \dots, N$ is the bead index, $\mu = 0, 1, 2$ indicates the coordinates component (x,y, or z), $\alpha = 1, \dots, 3N$ indicates the mode.

```
In [9]: msf=enm_obj.get_MSF()
```

```
In [10]: plt.figure(figsize=(12,3))
plt.plot(msf)
plt.ylabel('MSF$_i$')
plt.xlabel('$i$')
plt.xlim(-1,enm_obj.n_beads)
plt.grid()
```



The MSF usually represent a decent approximation of the B-factors obtained from crystallography

12.4 4: C2-C2 fluctuations

We can see that the MSFs become very large at the terminals. This is because they represent global fluctuations with respect to the equilibrium position, not local fluctuations of distances.

A way of estimating local flexibility is to compute inter-bead distances. In particular C2-C2 fluctuations have been shown to correlate well with SHAPE reactivity (Pinamonti et al. NAR 2015)

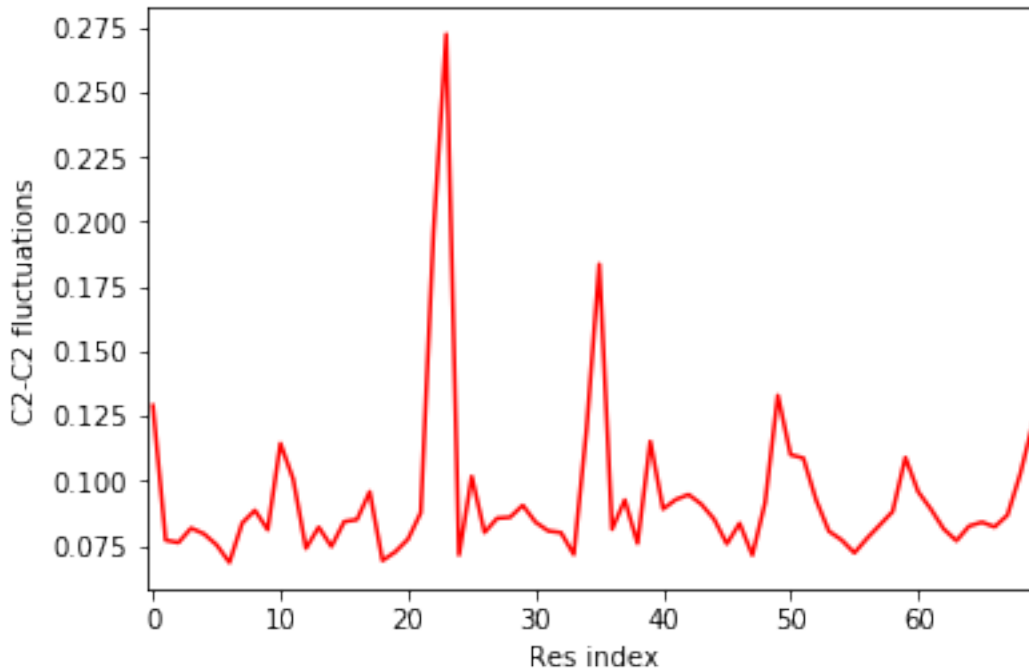
```
In [11]: %time fluc_C2,reslist=enm_AA.c2_fluctuations()
```

```
CPU times: user 11.4 s, sys: 143 ms, total: 11.6 s
```

```
Wall time: 11.9 s
```

```
In [12]: plt.plot(fluc_C2,c='r')
plt.ylabel('C2-C2 fluctuations')
plt.xlabel('Res index')
plt.xlim(-0.5,69.5)
```

```
Out[12]: (-0.5, 69.5)
```

12.5 5: Sparse diagonalization

The dynamics of the ENM is determined by the eigenvectors and eigenvalues of the interaction matrix M_{ij} , and the covariance matrix C_{ij} can be obtained via pseudo-inversion of M_{ij} .

However, we are usually interested only in the eigenvectors corresponding to high amplitude modes, i.e. those with the smallest eigenvalues.

We can exploit the sparsity of M_{ij} to speed up the computation of these modes.

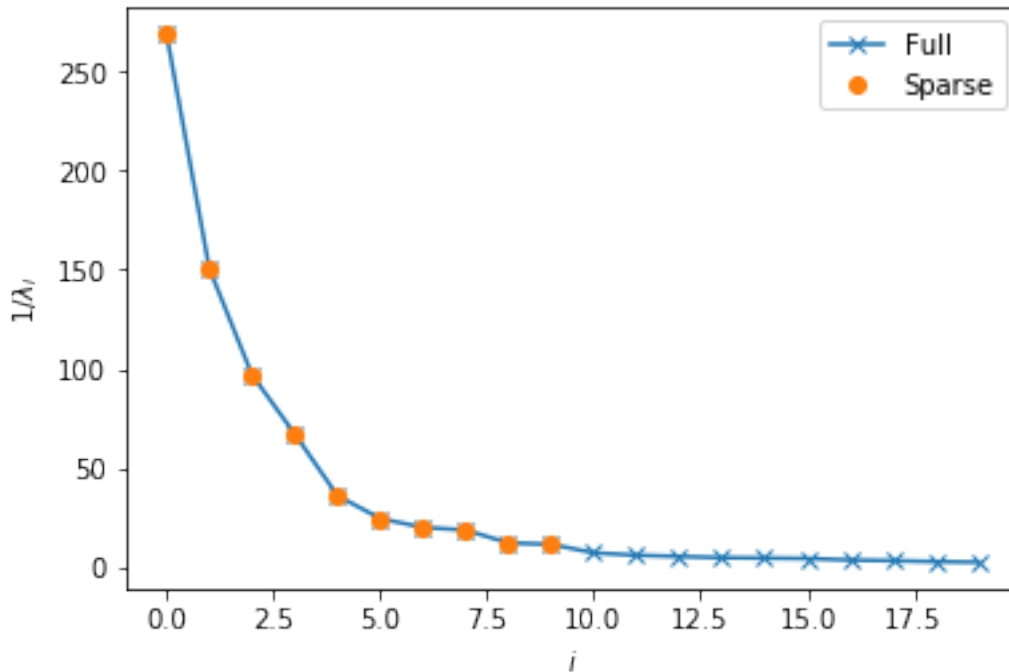
```
In [13]: %time enm_sparse=enm.Enm(fname,sparse=True,sele_atoms="AA",cutoff=0.7)

# Read (1499, 3) coordinates
# Using sparse matrix diagonalization
CPU times: user 3.68 s, sys: 155 ms, total: 3.83 s
Wall time: 3.39 s
```

This is now much faster (x10) than the full-matrix diagonalization.

```
In [14]: plt.plot(1/enm_AA.get_eval()[6:26],label='Full',marker='x')
plt.plot(1/enm_sparse.get_eval()[6:],marker='o',label='Sparse',ls='')
plt.legend()
plt.ylabel(r'$1/\lambda_i$')
plt.xlabel('$i$')
```

```
Out[14]: Text(0.5,0,'$i$')
```



12.5.1 5b: C2-C2 fluctuations with sparse matrices:

The problem here is that all eigenvectors are required to compute these fluctuations (see formula in [2])

Thus, we cannot use the results obtained in the previous section, because computing all $3N$ eigenvectors would be computationally very inefficient. Luckily, there's a nice trick to solve this. Since we are only interested in a subset of the system (i.e. the C2 atoms), we can obtain the "effective" interaction matrix, following the formula proposed in [6], as

$$M_{eff} = M_a - WM_b^{-1}W^T$$

This is the matrix governing the dynamics of the C2 beads. This is convenient because M_a is in general much smaller than M_{tot} ($1/3$ for SBP-ENM, $\sim 1/20$ for AA-ENM) and is therefore much quicker to diagonalize.

The calculation of M_{eff} itself is simply performed using `scipy.sparse.spsolve`.

The computational time is considerably reduced when employing a AA-ENM for molecule of increasing sizes.

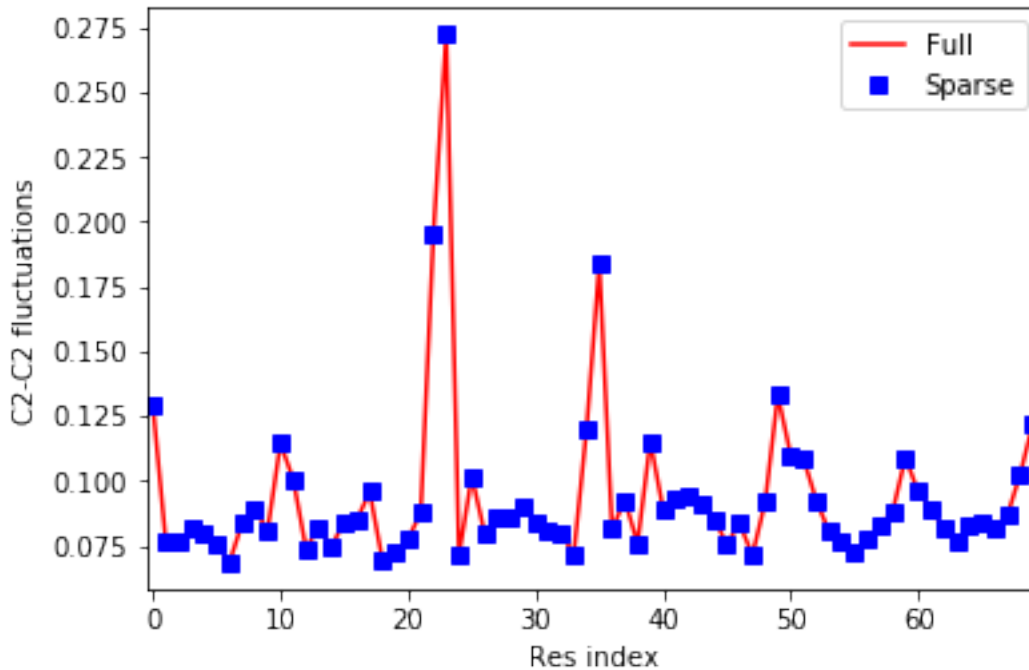
```
In [15]: %time fluc_C2_sparse,reslist=enm_sparse.c2_fluctuations()
```

```
CPU times: user 4.24 s, sys: 295 ms, total: 4.53 s
```

```
Wall time: 3.36 s
```

```
In [16]: plt.plot(fluc_C2,c='r',label='Full')
plt.plot(fluc_C2_sparse,c='b',label='Sparse',ls='',marker='s')
plt.ylabel('C2-C2 fluctuations')
plt.xlabel('Res index')
plt.legend()
plt.xlim(-0.5,69.5)
```

```
Out[16]: (-0.5, 69.5)
```



As we can see the final result is extremely accurate, while the computational time is much reduced, already for a relatively small-sized molecule like the one used here (71 nucleotides).

12.6 6: Visualize the eigenmodes

What do the different eigenmodes of our ENM look like?

We can take a look at that using the method `> get_mode_traj(i)`

That will return a `mdtraj.trajectory` object representing the fluctuations defined by the *i*-th eigenvector

$$\mathbf{x}(t) = \mathbf{x}_0 + A\mathbf{v}^i \sin(\omega t)$$

```
In [17]: traj_mode=enm_AA.get_mode_traj(6,amp=5.0,nframes=50)
```

We can take a look at this trajectory using [nglview](#)

```
In [24]: import nglview
         view = nglview.show_mdtraj(traj_mode)
         view
```

```
NGLWidget(count=50)
```

We can also save it as a `pdb` (or any other format) to visualize it later on with a different visualization software

```
In [19]: traj_mode.save_pdb('./enm_traj_mode_6.pdb')
```

12.7 References:

[1] Tirion, Monique M. "Large amplitude elastic motions in proteins from a single-parameter, atomic analysis." *Physical review letters* 77.9 (1996): 1905.

[2] Pinamonti, Giovanni, et al. "Elastic network models for RNA: a comparative assessment with molecular dynamics and SHAPE experiments." *Nucleic acids research* 43.15 (2015): 7260-7269.

[3] Setny, Piotr, and Martin Zacharias. "Elastic network models of nucleic acids flexibility." *Journal of chemical theory and computation* 9.12 (2013): 5460-5470.

[4] Zimmermann, Michael T., and Robert L. Jernigan. "Elastic network models capture the motions apparent within ensembles of RNA structures." *RNA* 20.6 (2014): 792-804.

[5] Fuglebakk, Edvin, Nathalie Reuter, and Konrad Hinsén. "Evaluation of protein elastic network models based on an analysis of collective motions." *Journal of chemical theory and computation* 9.12 (2013): 5618-5628.

[6] Zen A. Carnevale V. Lesk A.M. Micheletti C. "Correspondences between low-energy modes in enzymes: dynamics-based alignment of enzymatic functional families." *Protein Sci.* (2008) 17 918 929.

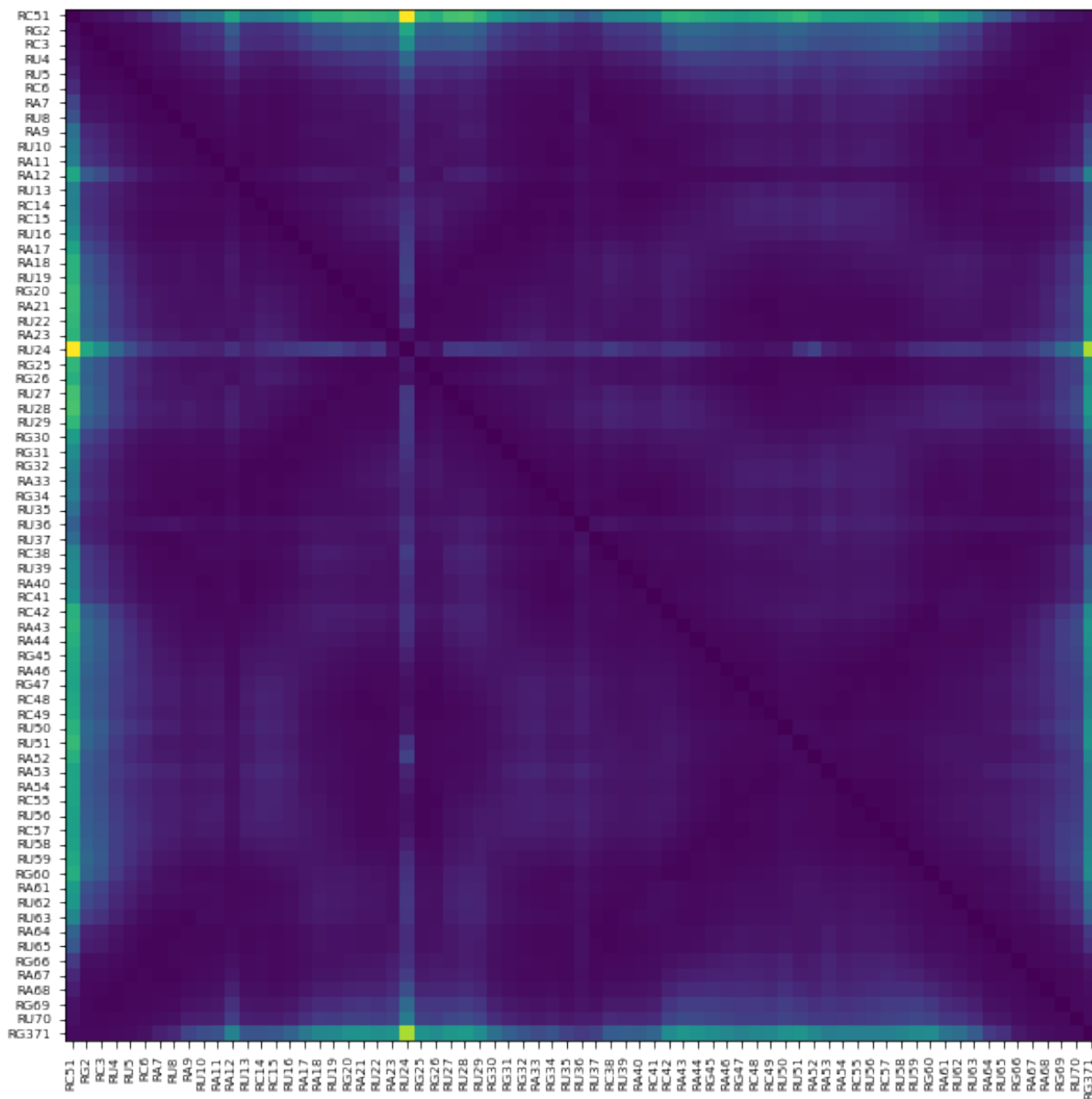
12.8 Extra credits: Distance fluctuation matrix

```
In [20]: # default is beads_name="C2"
         %time fluc_mat,res_list=enm_sparse.get_dist_fluc_mat()
```

CPU times: user 22.2 s, sys: 383 ms, total: 22.6 s

Wall time: 22.5 s

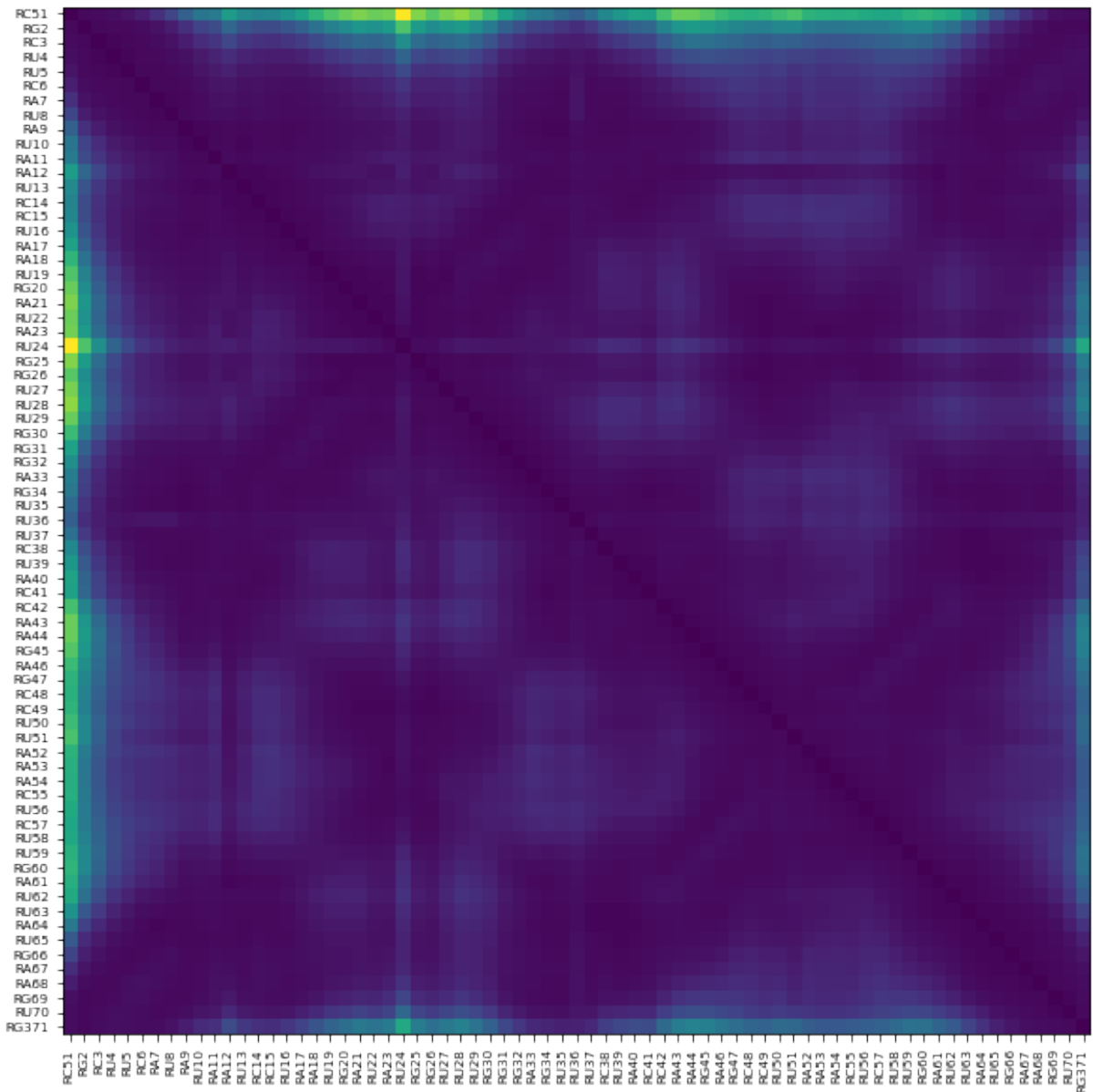
```
In [21]: plt.figure(figsize=(10,10))
         plt.imshow(fluc_mat,)
         tt=plt.xticks(np.arange(len(res_list)),res_list,rotation=90,fontsize=7)
         tt=plt.yticks(np.arange(len(res_list)),res_list,fontsize=7)
```



```
In [22]: # one can calculate fluctuations of different atoms
         %time fluc_mat,res_list=enm_sparse.get_dist_fluc_mat(beads_name="C1\')
```

CPU times: user 26.7 s, sys: 615 ms, total: 27.3 s
 Wall time: 27.8 s

```
In [23]: plt.figure(figsize=(10,10))
         plt.imshow(fluc_mat,)
         tt=plt.xticks(np.arange(len(res_list)),res_list,rotation=90,fontsize=7)
         tt=plt.yticks(np.arange(len(res_list)),res_list,fontsize=7)
```



13 eSCORE, a scoring function for RNA structure prediction

With barnaba it is possible to score decoys according to the eSCORE function. Essentially, we train a simple statistical potential using the interactions observed in a reference structure, as described in example 2. The distribution from the crystal structure of the large ribosomal subunit is then used to assign a score to each decoy in a long MD trajectory of a tetraloop.

We show below how to *train* the model:

```
In [1]: from barnaba import escore
        pdb = "../test/data/1S72.pdb"
        Escore = escore.Escore([pdb])

# KDE computed. Bandwidth= 0.25 using 10655 base-pairs
```

And how to score samples from a trajectory

```
In [2]: traj = "../test/data/UUCG.xtc"
        top = "../test/data/UUCG.pdb"
        scores = Escore.score(traj,topology=top)

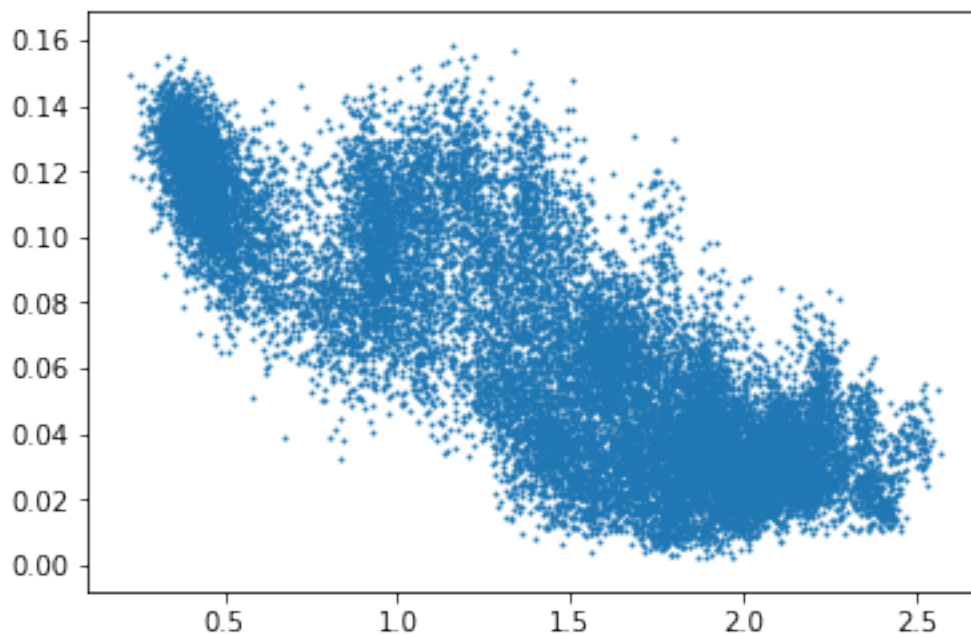
# Loaded sample ../test/data/UUCG.xtc
```

We visualize the data by plotting the score versus the eRMSD from native

```
In [14]: import barnaba as bb
         import matplotlib.pyplot as plt
         native="uucg2.pdb"
         ermsd = bb.ermsd(native,traj,topology=top)

         plt.scatter(ermsd,scores,s=1)
         plt.show()

# Loaded reference uucg2.pdb
# Loaded target ../test/data/UUCG.xtc
```



We can see that some high-scoring decoys have a large eRMSD from native:

```
In [13]: import numpy as np
         print "%#8s   %6s %6s" % ("Index", "eSCORE", "eRMSD")
         for j in np.argsort(scores)[-5:-1]:
             print " %8d   %6.4f %6.4f" % (j, scores[j], ermsd[j])
```

```
#  Index  eSCORE  eRMSD
    3849   0.1583  1.1591
    19518  0.1567  1.3370
    16169  0.1551  1.2246
    18536  0.1548  0.3352
```

We now pick two structure for comparison

```
In [6]: import mdtraj as md

         # load trajectory
         tt = md.load(traj, top=top)
         # save low ermsd
         tt[3849].save("best_score_0.pdb")
         tt[18536].save("best_score_1.pdb")

         # align to native and write aligned PDB to disk
         rmsd1 = bb.rmsd(native, 'best_score_0.pdb', out='best_score_0a.pdb')
         rmsd2 = bb.rmsd(native, 'best_score_1.pdb', out='best_score_1a.pdb')
         print rmsd1, rmsd2

# found 93 atoms in common
# found 93 atoms in common
[0.22360237] [0.11369839]
```

```
In [9]: import py3Dmol

         pdb_e = open('best_score_0a.pdb', 'r').read()
         pdb_n = open('best_score_1a.pdb', 'r').read()

         p = py3Dmol.view(width=900, height=600, viewergrid=(1,2))
         p.addModel(pdb_n, 'pdb', viewer=(0,0))
         p.addModel(pdb_e, 'pdb', viewer=(0,1))

         p.setStyle({'stick':{}})
         p.setBackgroundColor('Oxeeeeeee')
         p.zoomTo()
```