simulation_analysis_notebook-PDF version

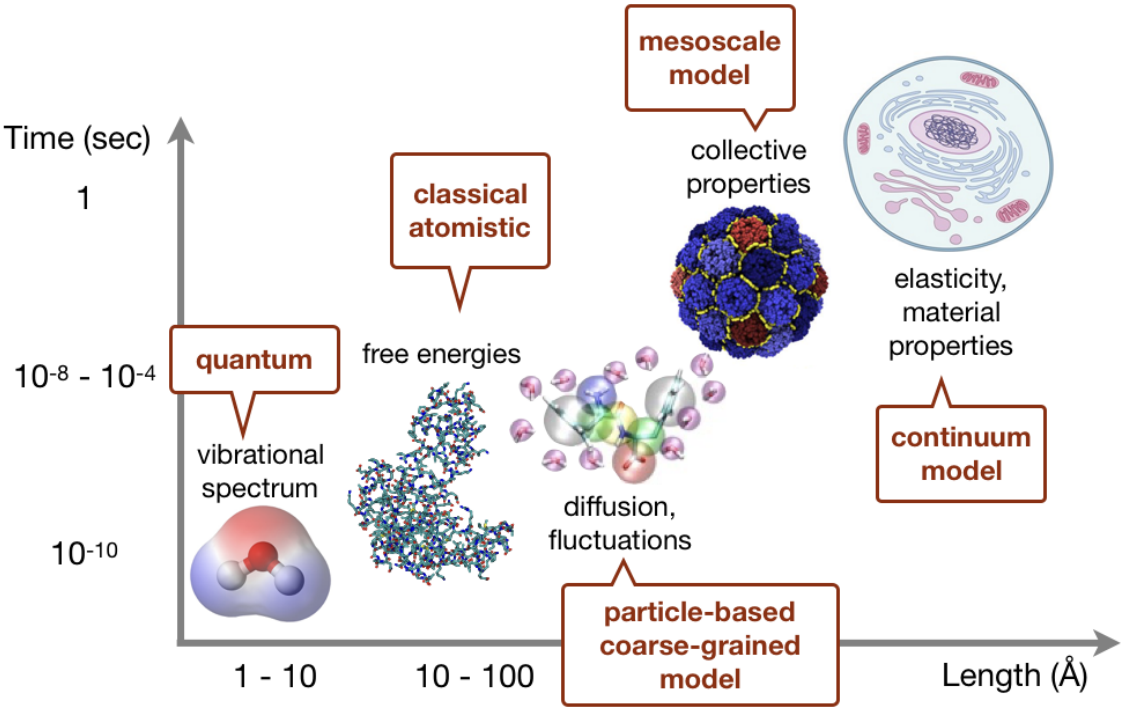
March 26, 2021

[2]:
```
from IPython import display
```

1 Multiscale nature of Nature

[9]:
```
display.Image("./multiscale.png")
```

[9]:

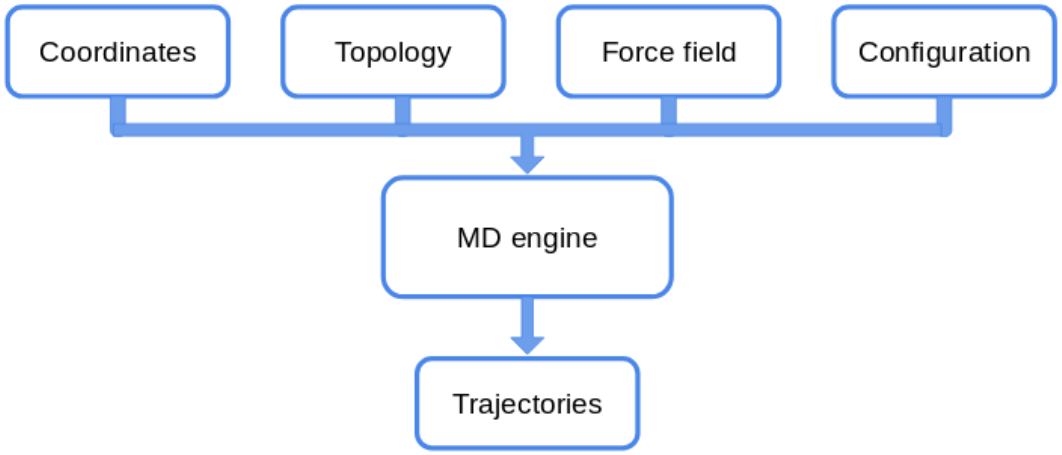


2 MOLECULAR DYNAMICS

- Numerical solution of Newton's equations of motion;
- Forces can be derived from a huge variety of potentials, called force fields;
- Used to observe the time evolution of a system.

[10]:
```
display.Image("./md_machinery.png")
```

[10]:



3 SIMULATION ANALYSIS

Molecular Dynamics (MD) simulations are more and more routinely and effortlessly performed using a plethora of different engines, each optimised for a specific purpose. Examples of these MD engines are GROMACS, NAMD, LAMMPS, DESMOND, AMBER etc.

While setting up and running a MD simulation is still a non-trivial task, more and more importance is devoted to the process of trajectory analysis. With the current impressive hardware and software developments we can *generate* huge amount of trajectory data, so it is crucial to design robust analysis pipelines to **make sense of the data**.

The instruments we are looking for should be: - **reliable**, i.e. widely tested; - **complete**: it is not recommended to analyse data employing a set of tools coming from a bunch of heterogeneous sources; - **suitable for large-scale analysis - suitable for quick and dirty scripting**

4 MDANALYSIS

In this tutorial I will introduce you to **MDAnalysis**, a collaborative python package for molecular dynamics data analysis. Although MDAnalysis is easy to learn and custom, it has a good performance since its routines interface with C codes, especially in the case of computationally expensive tasks. Indeed, one of the main aims of the project is to provide the user with all the data about the system stored in numpy arrays, which can be easily manipulated by the fast python libraries such as *numpy*, *scipy* and *scikit-learn*.

When in doubt, I encourage you to check out the user guide.

For visualization we will use **nglview**, a widget for jupyter notebook. This is very useful for interactive work. Check the github page for the documentation and examples.

[16]:
```
import MDAnalysis as mda
print(mda.__version__)
from pathlib import Path
from os import fspath
import nglview
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import pandas as pd
```

```
1.0.1
```

4.1 Object-oriented software

MDAnalysis is strongly object-oriented. Every object possesses a set of properties, called **attributes** and a set of **procedures** (or **methods**).

Which are the objects in the realm of molecular simulations?

- Atoms
- Groups of Atoms
- Trajectories
- Topology objects
- many more

Each of these elements is an object in MDAnalysis, with its methods and attributes.

4.2 The fundamental object: the Universe

The core object *Universe* contains all the information we possess about the system. Typically we consider a topology and a trajectory file.

Here we will load two sample files, a *PSF* topology file and a *DCD* trajectory file referring to a short (200 nanoseconds, 200 frames) trajectory of the Adenylate Kinase enzyme. MDAnalysis supports several different file formats produced by the currently most used MD engines.

When a universe is initialized, the topology (trajectory) is read by a `TopologyReader` (`TrajectoryReader`) object. Frames in the trajectory are splitted in several `Timestep` objects.

[17]:
```
simdir = Path('./data/')
PSF = simdir / 'adk4AKE.psf'
eqDCD = simdir / 'adk_npt_200.dcd'
PDB = simdir / 'adk_open.pdb'
```

```
cl_PDB = simdir / 'adk_closed.pdb'
```

[18]:
```
universe = mda.Universe(str(PSF),str(eqDCD))
```

[19]:
```
view = nglview.show_mdanalysis(universe)
view
```

```
NGLWidget(max_frame=200)
```

4.3 1. Basic ingredients

- AtomGroups
- Topology
- Trajectory
- Selections

4.4 1.1 The trajectory and the Timesteps

The Trajectory is a collection of Timesteps, the latter representing all the available information we have about a single time frame. One tricky point in MDAnalysis is the fact that it is easier to iterate **implicitly** over the timesteps rather than cycling **explicitly**, as one may expect. Indeed, a trajectory in the Universe possesses an attribute called **`ts`** that defines which Timestep object we are considering. Every time we iterate over this attribute we change our position in time.

Implicit iteration

[6]:
```
# here we are at timestep 0
universe.trajectory.ts
```

[6]:
```
< Timestep 0 with unit cell dimensions [99.683105 99.683105 99.683105 90.
90.       90.       ] >
```

We can easily access the coordinates corresponding to the current timestep

[7]:
```
universe.trajectory.ts.positions
```

[7]:
```
array([[41.149998, 32.739998, 53.61    ],
       [41.63    , 32.47    , 54.45    ],
       [40.43    , 33.41    , 53.850002],
       …,
       [40.72    , 30.03    , 43.239998],
       [40.079998, 29.22    , 43.969997],
       [40.24    , 31.02    , 42.61    ]], dtype=float32)
```

We can switch to the next step:

[8]:
```
universe.trajectory.next()
universe.trajectory.ts
```

[8]:
```
< Timestep 1 with unit cell dimensions [99.79879 99.79879 99.79879 90.      90.
90.     ] >
```

which has its own coordinates vector

[9]:
```
universe.trajectory.ts.positions
```

[9]:
```
array([[41.340004, 32.79    , 52.79    ],
       [40.620003, 33.48    , 52.940002],
       [40.940002, 32.11    , 52.15    ],
       …,
       [47.000004, 27.870003, 44.4     ],
       [46.190002, 26.970001, 44.440002],
       [47.550003, 28.290003, 45.45    ]], dtype=float32)
```

The typical usage of MDAnalysis consists on the iteration of an analysis protocol over all the available frames of a trajectory.

Explicit iteration It is possible to explicitly access the different time steps of the trajectory.

[10]:
```
universe.trajectory[0].positions
```

[10]:
```
array([[41.149998, 32.739998, 53.61    ],
       [41.63    , 32.47    , 54.45    ],
       [40.43    , 33.41    , 53.850002],
       …,
       [40.72    , 30.03    , 43.239998],
       [40.079998, 29.22    , 43.969997],
       [40.24    , 31.02    , 42.61    ]], dtype=float32)
```

[11]:
```
universe.trajectory[1].positions
```

[11]:
```
array([[41.340004, 32.79    , 52.79    ],
       [40.620003, 33.48    , 52.940002],
       [40.940002, 32.11    , 52.15    ],
       …,
       [47.000004, 27.870003, 44.4     ],
       [46.190002, 26.970001, 44.440002],
       [47.550003, 28.290003, 45.45    ]], dtype=float32)
```

4.5 1.2 AtomGroups

The Universe contains a set of atoms that will be employed throughout the analysis. Let's see who they are.

[12]:
```
universe.atoms
```

[12]:
```
<AtomGroup with 3341 atoms>
```

Here we meet the object *AtomGroup*, that allows to lump several atoms in a single structure. This class contains many useful attibutes:

- `natoms`, `n_residues`, `n_fragments`, `n_segments` : they describe how many of these entities are present in the `AtomGroup`;
- `positions`, `velocities`, `forces` , `dimensions`: all the attributes of the `Timestep` object are inherited by this class;
- `bonds`, `angles`, `dihedrals` and other elements of the Topology.

You can easily access the possible attributes of the AtomGroup with tab completion:

[14]:
```
universe.atoms.
```

[14]:
```
<bound method Masses.center_of_mass of <AtomGroup with 3341 atoms>>
```

[13]:
```
universe.atoms.velocities
```

```
---------------------------------------------------------------------------
NoDataError                               Traceback (most recent call last)
<ipython-input-13-c281521fb326> in <module>
----> 1 universe.atoms.velocities

//miniconda3/envs/SAT/lib/python3.8/site-packages/MDAnalysis/core/groups.py in␣
 ↪__getattr__(self, attr)
   2290         # special-case timestep info
   2291         if attr in ('velocities', 'forces'):
-> 2292             raise NoDataError('This Timestep has no ' + attr)
   2293         elif attr == 'positions':
   2294             raise NoDataError('This Universe has no coordinates')

NoDataError: This Timestep has no velocities
```

Guess what? The AtomGroup is composed by a series of *Atom* objects!

[18]:
```
universe.atoms[0]
```

[18]:
```
<Atom 1: N of type NH3 of resname MET, resid 1 and segid ADK>
```

From the *Atom* object you can easily access any possible detail about the atom you are interested in.

[14]:
```
print("residue id of atom ", universe.atoms[0], " is ", universe.atoms[0].resid)
print("residue name of atom ", universe.atoms[0], " is ", universe.atoms[0].
 ↪resname)
```

```
print("partial charge of atom ", universe.atoms[0], " is ", universe.atoms[0].
 ↪charge)
print("position of atom ", universe.atoms[0], " is ", universe.atoms[0].
 ↪position)
```

```
residue id of atom  <Atom 1: N of type NH3 of resname MET, resid 1 and segid
ADK>  is  1
residue name of atom  <Atom 1: N of type NH3 of resname MET, resid 1 and segid
ADK>  is  MET
partial charge of atom  <Atom 1: N of type NH3 of resname MET, resid 1 and segid
ADK>  is  -0.30000001192092896
position of atom  <Atom 1: N of type NH3 of resname MET, resid 1 and segid ADK>
is  [41.340004 32.79     52.79    ]
```

4.6 1.3 Atom Selection

The AtomGroup automatically defined by the creation of the Universe contains all the atoms present in the topology. Obviously it is not the only group that you can define.

The user can create your own AtomGroup with the **`select_atoms`** method of a Universe/AtomGroup. The syntax is very similar to that of CHARMM selection commands.

4.6.1 1.3.1 keywords-based selections

Keywords are case-sensitive strings that are recognized by the *SelectionParser*, which outputs a list of atoms ordered according to the sequence of atoms in the Universe.

Useful examples of simple selections are: - simple keywords: `"backbone"`, `"protein"`, `"nucleic"`, `"water"` - topology-based keyworkds: `resname MET`, `name CA`, `"resnum 13:16"` - pattern matching keywords: `"resname HS?"`, `"resname G*"`, `"resnum 2[0-1]"`

[15]:
```
bkb_group = universe.select_atoms("backbone")
print("backbone group\n", bkb_group)
cb_group = universe.select_atoms("name CB")
print("\ncb_group\n",cb_group)
resname_group = universe.select_atoms("resname G*")
print("\nresname_group\n", resname_group)
```

```
backbone group
 <AtomGroup [<Atom 1: N of type NH3 of resname MET, resid 1 and segid ADK>,
<Atom 5: CA of type CT1 of resname MET, resid 1 and segid ADK>, <Atom 18: C of
type C of resname MET, resid 1 and segid ADK>, …, <Atom 3334: C of type CC of
resname GLY, resid 214 and segid ADK>, <Atom 3337: N of type NH1 of resname GLY,
resid 214 and segid ADK>, <Atom 3339: CA of type CT2 of resname GLY, resid 214
and segid ADK>]>

cb_group
```

```
 <AtomGroup [<Atom 7: CB of type CT2 of resname MET, resid 1 and segid ADK>,
<Atom 24: CB of type CT2 of resname ARG, resid 2 and segid ADK>, <Atom 48: CB of
type CT1 of resname ILE, resid 3 and segid ADK>, …, <Atom 3278: CB of type CT2
of resname LYS, resid 211 and segid ADK>, <Atom 3300: CB of type CT1 of resname
ILE, resid 212 and segid ADK>, <Atom 3319: CB of type CT2 of resname LEU, resid
213 and segid ADK>]>

resname_group
 <AtomGroup [<Atom 120: N of type NH1 of resname GLY, resid 7 and segid ADK>,
<Atom 121: HN of type H of resname GLY, resid 7 and segid ADK>, <Atom 122: CA of
type CT2 of resname GLY, resid 7 and segid ADK>, …, <Atom 3339: CA of type CT2
of resname GLY, resid 214 and segid ADK>, <Atom 3340: HA1 of type HB of resname
GLY, resid 214 and segid ADK>, <Atom 3341: HA2 of type HB of resname GLY, resid
214 and segid ADK>]>
```

4.6.2 1.3.2 combining selections

multiple selection can be combined with boolean operators.

Imagine that I want to select all the heavy (not hydrogen) atoms of a protein that do not belong to the main chain.

[40]:
```
side_chain_group = universe.select_atoms("not backbone and not name H*")
print("side_chain_group\n",side_chain_group)
```

```
side_chain_group
 <AtomGroup [<Atom 1: N of type NH3 of resname MET, resid 1 and segid ADK>,
<Atom 2: HT1 of type HC of resname MET, resid 1 and segid ADK>, <Atom 3: HT2 of
type HC of resname MET, resid 1 and segid ADK>, …, <Atom 3339: CA of type CT2
of resname GLY, resid 214 and segid ADK>, <Atom 3340: HA1 of type HB of resname
GLY, resid 214 and segid ADK>, <Atom 3341: HA2 of type HB of resname GLY, resid
214 and segid ADK>]>
```

4.6.3 1.3.3 geometry-based selections

MDAnalysis supports several geometric selections, which allow to keep track of the arrangements of atoms in interesting regions of the structure. Some examples are: - **point x y z distance**: to find all the atoms that lie closer than distance to the (x,y,z) coordinate; - **around distance selection**: to retrieve all atoms closer than distance to another selection; - **prop [abs] property operator value**: to list all the atoms which possess a value of property (x,y or z) that match the condition defined by operator (<,>,==,>=,<=,!=) and value

[66]:
```
around_selection = universe.select_atoms("not name H* and around 2 name CA and␣
 ↪resid 1")
print("around 2A of first c alpha selection\n\n",around_selection.atoms)
print("\naround 2A of first c alpha positions\n\n",around_selection.atoms.
 ↪positions)
```

```
point_selection = universe.select_atoms("point 50 50 50 2.0")
print("\npoint selection positions\n",point_selection.atoms.positions)
prop_selection = universe.select_atoms("prop z > 77")
print("\nprop selection positions\n",prop_selection.atoms.positions)
```

```
around 2A of first c alpha selection

 <AtomGroup [<Atom 1: N of type NH3 of resname MET, resid 1 and segid ADK>,
<Atom 7: CB of type CT2 of resname MET, resid 1 and segid ADK>, <Atom 18: C of
type C of resname MET, resid 1 and segid ADK>]>

around 2A of first c alpha positions

 [[41.340004 32.79     52.79    ]
 [42.070004 33.86     50.64    ]
 [42.95     34.760002 52.65    ]]

point selection positions
 [[49.07     49.020004 48.660004]
 [50.470005 49.020004 50.91    ]]

prop selection positions
 [[45.070004 51.600002 78.15001 ]
 [45.08     50.36     78.380005]
 [45.260002 52.510002 79.00001 ]
 [45.29     57.500004 77.270004]
 [42.870003 56.280003 78.03001 ]
 [43.530003 56.550003 78.75001 ]
 [42.23     57.060005 77.96001 ]
 [42.420002 55.410004 78.26    ]
 [48.920002 52.920002 77.35    ]
 [48.93     51.140003 77.340004]]
```

4.6.4 1.3.4 dynamic selections

While some selections remain constant over the whole trajectory, others (like geometric selections) may change as the molecule evolves.

Can we keep track of the time-evolution in our selections? With the **updating** keyword we force the software to re-calculate the selection when the Timestep is changed.

[11]:
```
# example on our protein
sel_res_128_130 = universe.select_atoms("resid 128:130")
```

Imagine that we want to identify all the atoms closer than 0.45 nanometers to our selection.

[22]:
```
neigh = universe.select_atoms("around 4.5 resid 128:130", updating = True)
print("atom group with ", neigh.n_atoms, " atoms in ", neigh.n_residues ,"␣
 ↪residues")
universe.trajectory.next()
print("atom group with ", neigh.n_atoms, " atoms in ", neigh.n_residues ,"␣
 ↪residues")
```

```
atom group with  56  atoms in  8  residues
atom group with  64  atoms in  6  residues
```

4.7 1.4 AtomGroup's methods

An object AtomGroup possesses a series of useful methods, such as: - *center_of_geometry* - *center_of_mass* - *moment_of_inertia* - *radius_of_gyration*

center of geometry

$$\mathbf{r}_{geom} = \frac{\sum_i \mathbf{r_i}}{N}$$

[20]:
```
calpha = universe.select_atoms("name CA")
calpha.center_of_geometry(pbc=True)
# our trajectory has already been centered and PBC have been removed =>
# it is not necessary to center atoms in the primary unit cell
print("calpha atoms center of geometry = ", calpha.
 ↪center_of_geometry(pbc=False))
print("all atoms center of geometry = ", universe.atoms.
 ↪center_of_geometry(pbc=False))
```

```
calpha atoms center of geometry =  [49.70299051 49.77682223 50.044112  ]
all atoms center of geometry =  [49.81253512 49.85177488 49.84932059]
```

center of mass What happens if each atom is weighted with its own mass?

$$\mathbf{r}_{cm} = \frac{\sum_i m_i \mathbf{r_i}}{\sum_i mi}$$

[68]:
```
print("calpha atoms center of mass = ", calpha.center_of_mass())
print("all atoms center of mass = ", universe.atoms.center_of_mass())
```

```
calpha atoms center of mass =  [49.6814042  49.81233881 50.05850719]
all atoms center of mass =  [49.75132498 49.83777901 49.83428054]
```

moment of inertia tensor A useful quantity if we want to measure the angular momentum.

$$\overline{\mathbf{L}} = \overline{\overline{I}}\overline{\mathbf{w}}$$

[69]:
```
first_res = universe.select_atoms("not name H* and resid 1 and backbone")
print("moment of inertia tensor is")
first_res.moment_of_inertia()
```

```
moment of inertia tensor is
```

[69]:
```
array([[ 80.99809477, -20.35654118,   3.60612371],
       [-20.35654118,  21.9808962 ,  -2.73090073],
       [  3.60612371,  -2.73090073,  95.27641628]])
```

If diagonalized, it can give the principal axes of inertia of the *AtomGroup*.

[70]:
```
p_axes = first_res.principal_axes(pbc=False)
p_axes
```

[70]:
```
array([[ 0.37466394, -0.1347877 ,  0.91731086],
       [ 0.87856613, -0.26448863, -0.39770254],
       [ 0.2962237 ,  0.95492305,  0.01932556]])
```

radius of gyration

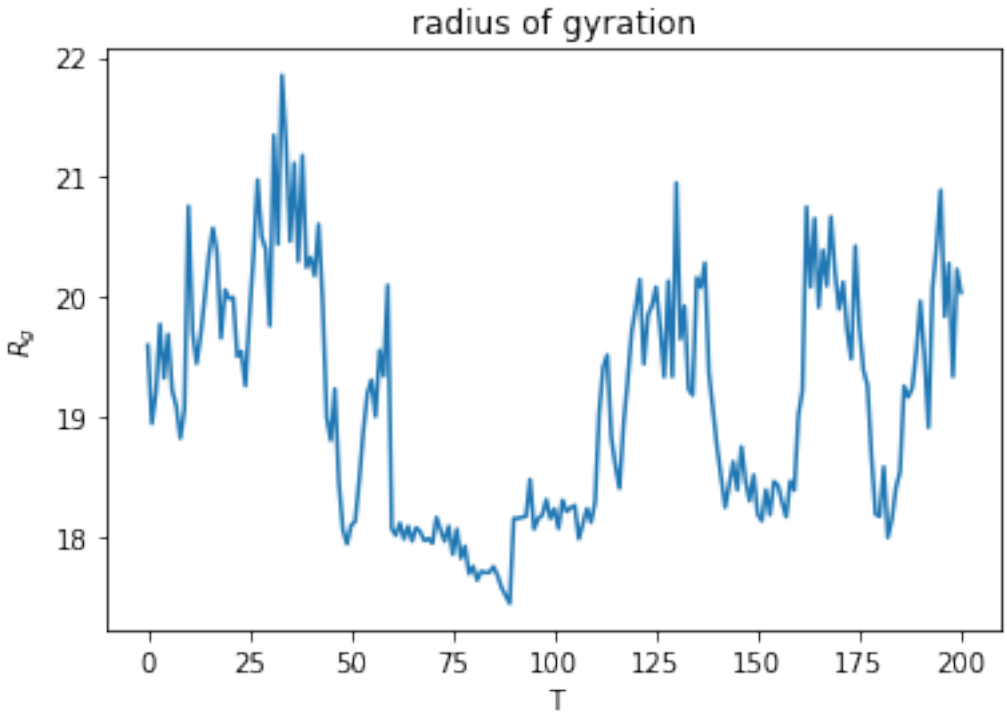
The radius of gyration is a useful measure of globularity of a structure. It is defined as the root mean square deviation of the position of the atoms with respect to the center of mass of the molecule.

$R_g^2 = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{r_i} - \mathbf{r_c})^2$

NB: in this approximation all the masses are assumed to be equal. In the case of heavy atoms of proteins this assumption is pretty much satisfied.

[18]:
```
Rg_vector = []
for ts in universe.trajectory:
    Rg_vector.append(calpha.radius_of_gyration())
```

[19]:
```
plt.plot(Rg_vector)
plt.title("radius of gyration")
plt.ylabel("$R_g$")
plt.xlabel("T")
plt.show()
```



5 2. Basic analysis

Here we will give an overview over the most popular routines that are routinely employed while analyzing MD data:

- Alignments
- RMSD and RMSF
- Native contacts
- Hydrogen bond analysis
- Ramachandran analysis
- PCA

These are only a subset of the Analysis Modules embedded in MDAnalysis.

5.1 2.1 Root mean square deviation

- RMSD: deviations of a subset of the atoms of the structure between two different frames (r_1 and r_2) of the trajectory. One of them is called *reference frame* (for example *r1*)
- useful measure of structure similarity
- basically the first thing to calculate when you have Molecular Dynamics data.

The RMSD between two structures of N atoms is given by:

$$\text{RMSD } (r_1, r_2) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\delta_i^2}$$

where the indices i run over the subset of selected atoms and δ_i is the deviation of atom i in frame r_2 with respect to the position of atom i in the reference frame r_1:

$$\delta_i^2 = |\vec{x}_{(2)i} - \vec{x}_{(1)i}|^2.$$

A rigid rototranslation has to be performed, in which frame r_2 is superimposed over the reference frame in order to remove unimportant translations and rotations (6 degrees of freedom to superimpose two sets of points). In this way the RMSD is minimised.

[6]:
```
from MDAnalysis.analysis import align
from MDAnalysis.analysis.rms import rmsd
```

First, we define the reference conformation by loading the open PDB file.

[21]:
```
ref_pdb = mda.Universe(PSF,str(PDB))
ref_pdb
```

[21]:
```
<Universe with 3341 atoms>
```

[22]:
```
view = nglview.show_mdanalysis(ref_pdb)
view
```

```
NGLWidget()
```

[23]:
```
rmsd(universe.select_atoms('name CA').positions, ref_pdb.select_atoms('name␣
↪CA').positions,superposition=True)
```

[23]:
```
0.9264899111841328
```

[24]:
```
universe.trajectory.ts
```

[24]:
```
< Timestep 0 with unit cell dimensions [99.683105 99.683105 99.683105 90.
90.       90.       ] >
```

Such non negligible difference is due to the fact that the configuration at $t = 0$ was equilibrated.

A more step by step procedure is given in the following cell:

[25]:
```
# centers of mass of the two universes
universe_com = universe.atoms.center_of_mass()
ref_com = ref_pdb.atoms.center_of_mass()
print("universe_com = ",universe_com)
print("ref_com = ",ref_com)
# translation of the universes
```

```
universe_scaled = universe.select_atoms('name CA').positions - universe_com
ref_scaled = ref_pdb.select_atoms('name CA').positions - ref_com
# alignment of the structures and calculation of the RMSD
R, rmsd_value = align.rotation_matrix(universe_scaled, ref_scaled)
print("rmsd = ", rmsd_value)
print("rotation matrix = \n", R)
```

```
universe_com =  [49.75378022 49.83477824 49.8365831 ]
ref_com =  [-3.72799868  9.62400822 14.35029554]
rmsd =  0.9270223919554296
rotation matrix =
 [[ 0.99951237 -0.00990724  0.02961193]
 [-0.00931222 -0.9997533  -0.02016462]
 [ 0.0298044   0.01987903 -0.99935805]]
```

2.1.1 full trajectory alignment

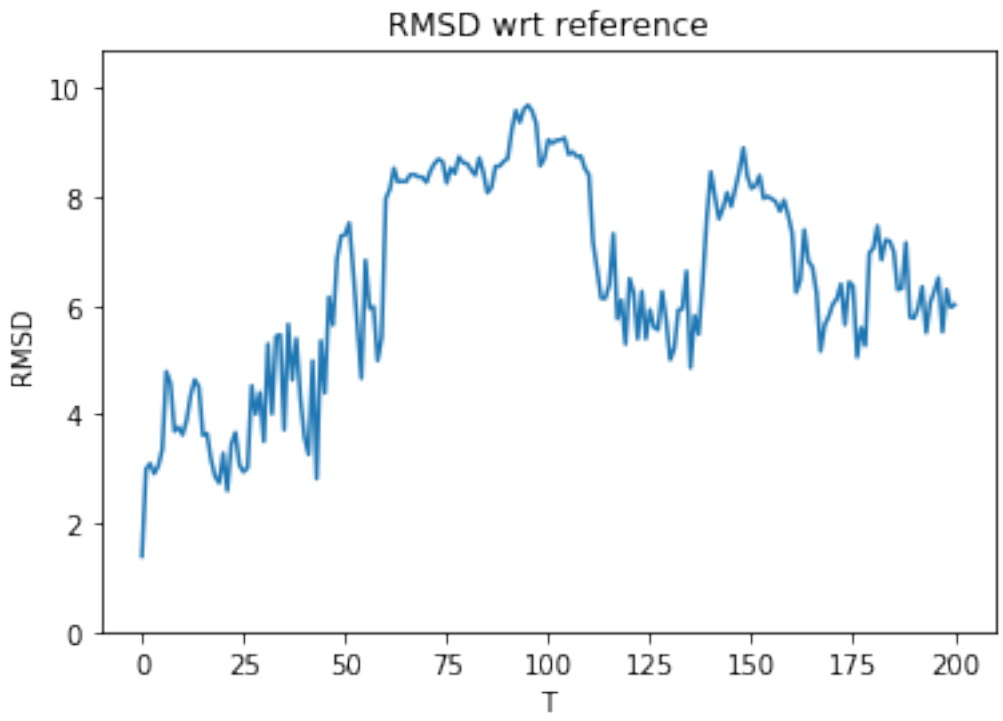
We employ the *AlignTraj* class to align every Frame of a trajectory to a reference conformation.

[26]:
```
alignment = align.AlignTraj(universe, ref_pdb)
alignment.run()
```

[26]:
```
<MDAnalysis.analysis.align.AlignTraj at 0x130342d00>
```

[27]:
```
plt.plot(alignment.rmsd)
plt.ylim(0,max(alignment.rmsd)+1)
plt.xlabel("T")
plt.title("RMSD wrt reference")
plt.ylabel("RMSD")
```

[27]:
```
Text(0, 0.5, 'RMSD')
```



5.1.1 2.1.2 full conformational matrix

We have seen how to: - compute the RMSD between two structures - extract the rototranslation that maximally superimposes two configurations - align a whole trajectory to a reference frame

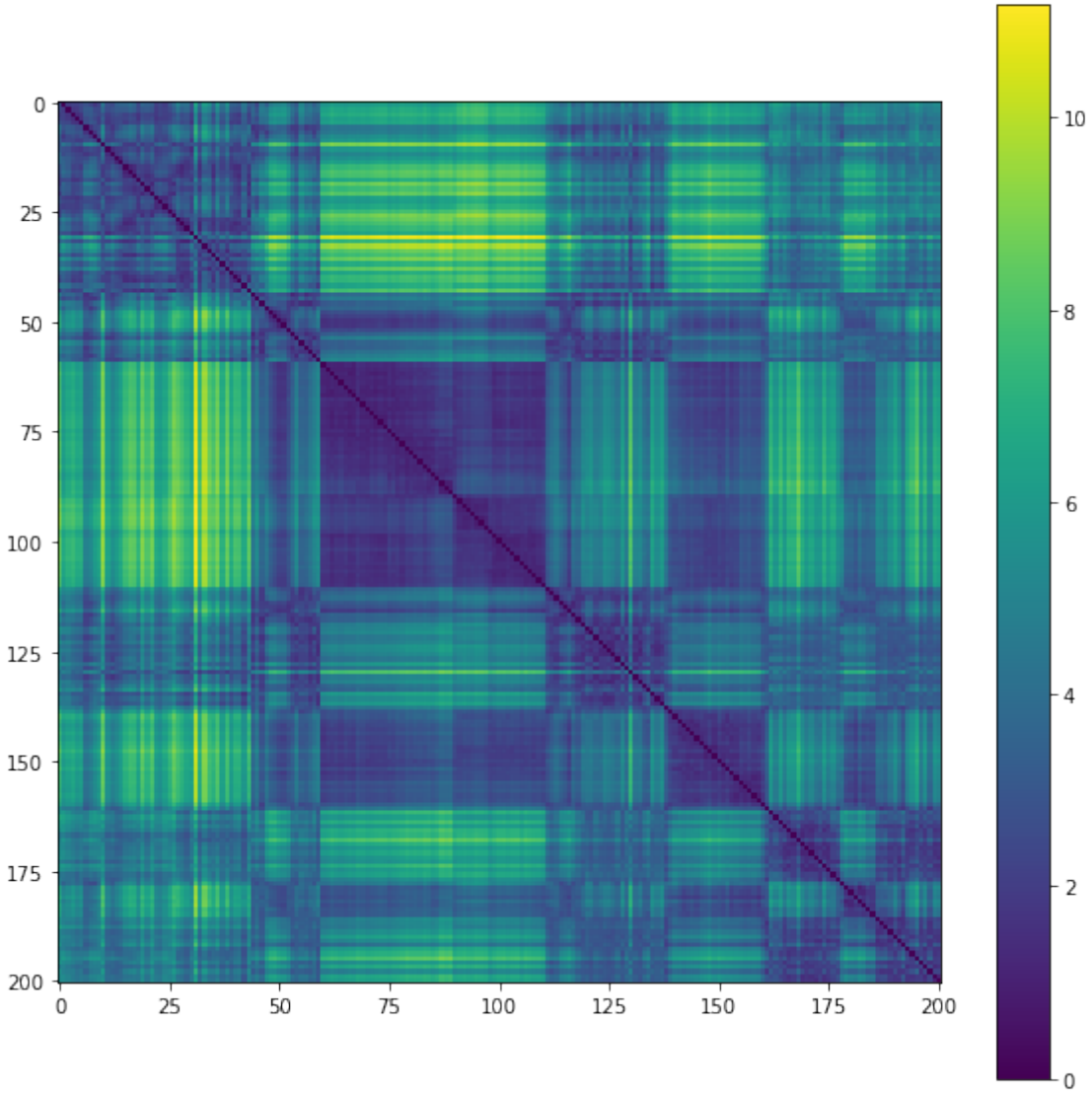
Now we want to build the full conformational matrix over a trajectory, namely the RMSD matrix between all pairs of frames belonging to a trajectory. Of course this task is efficiently implemented in MDAnalysis.

[28]:
```
import MDAnalysis.analysis.encore as encore
```

[29]:
```
rmsd_matrix = encore.confdistmatrix.get_distance_matrix(universe,select='name␣
↪CA').as_array()
```

[86]:
```
plt.figure(figsize=((10,10)))
plt.imshow(rmsd_matrix,cmap="viridis")
plt.colorbar()
```

[86]:
```
<matplotlib.colorbar.Colorbar at 0x131e3a760>
```



5.2 2.2 Root mean square fluctuation

- measures the fluctuations of each atom with respect to the equlibrium
- proportional to the temperature (β) factor
- useful to match experimental results

$$RMSF_i = \left[\frac{1}{T} \sum_{t_j=1}^{T} |\mathbf{r}_i(t_j) - \mathbf{r}_i|^2 \right]^{1/2}$$

Usually the reference $\mathbf{r}$ is identified with the average position of the atom over the trajectory. As in the case of RMSD calculations, you can compute RMSF over a subset of atoms. We will stick

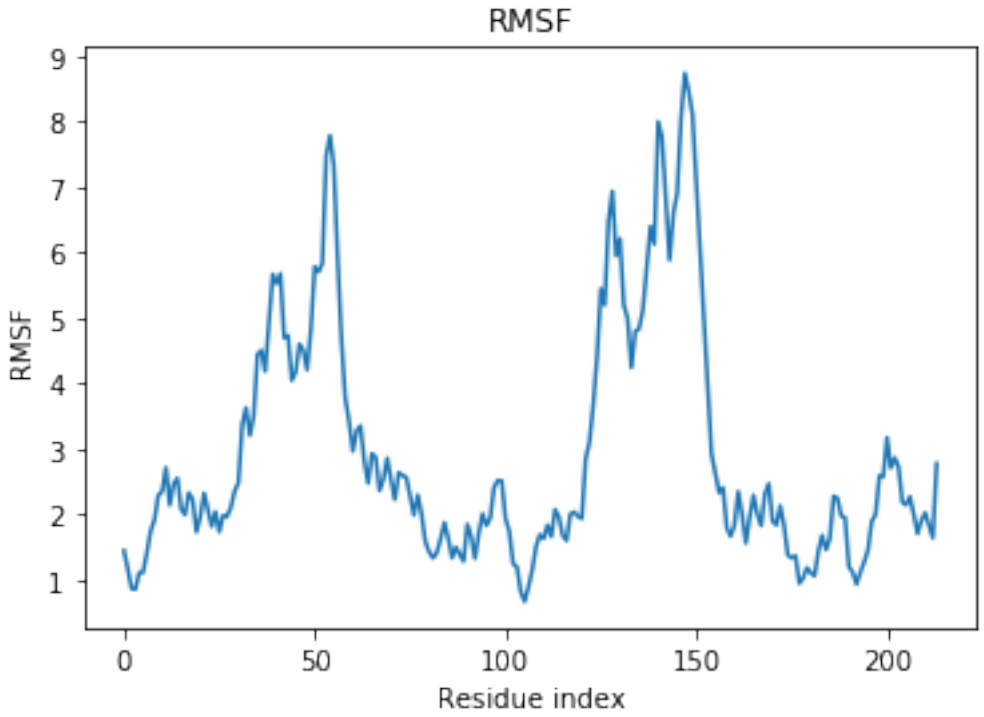
to the C_α atoms once again.

[21]:
```
from MDAnalysis.analysis.rms import RMSF
```

[22]:
```
myRMSF = RMSF(calpha).run()
```

[23]:
```
plt.plot(myRMSF.rmsf)
plt.xlabel('Residue index')
plt.ylabel('RMSF')
plt.title('RMSF')
```

[23]:
```
Text(0.5, 1.0, 'RMSF')
```



Unsurprisingly, the peaks correspond to the mobile domains (NMP and LID) of the enzyme and the minima are located in correspondence of the rigid CORE domain, that acts as a hinge.

5.3 2.3 Native contacts analysis

A native contact is defined as a pair of atoms that are closer than a distance r in the reference conformation of a biomolecule. In proteins, for instance, we call native contacts those that are present in the folded configuration. Understanding the role and the formation of native contacts is a fundamental ingredient in a good analysis pipeline.

MDAnalysis provides a handy and extensible module for Native Contacts analysis. There exist

three options (i.e. methods) to compute the contact matrix: - **hard_cut**: in order to be counted as a contact two atoms should be at a distance equal or lower than their distance in the reference conformation;

- **soft_cut**: a switching function assigns different weights to different distances

$$Q(r, r_0) = \frac{1}{1 + e^{\beta(r - \lambda r_0)}}$$

- **radius_cut**: two atoms form a contact if their distance is lower than a radius defined by the user.

[33]:
```
from MDAnalysis.analysis import contacts
```

[34]:
```
sel_calpha = "(name CA)"
start_calpha = universe.select_atoms(sel_calpha)
ca1 = contacts.Contacts(universe, select=(sel_calpha, sel_calpha),␣
 ↪method="radius_cut",
                        refgroup=(start_calpha, start_calpha),radius=8)
```

[35]:
```
ca1.run()
```

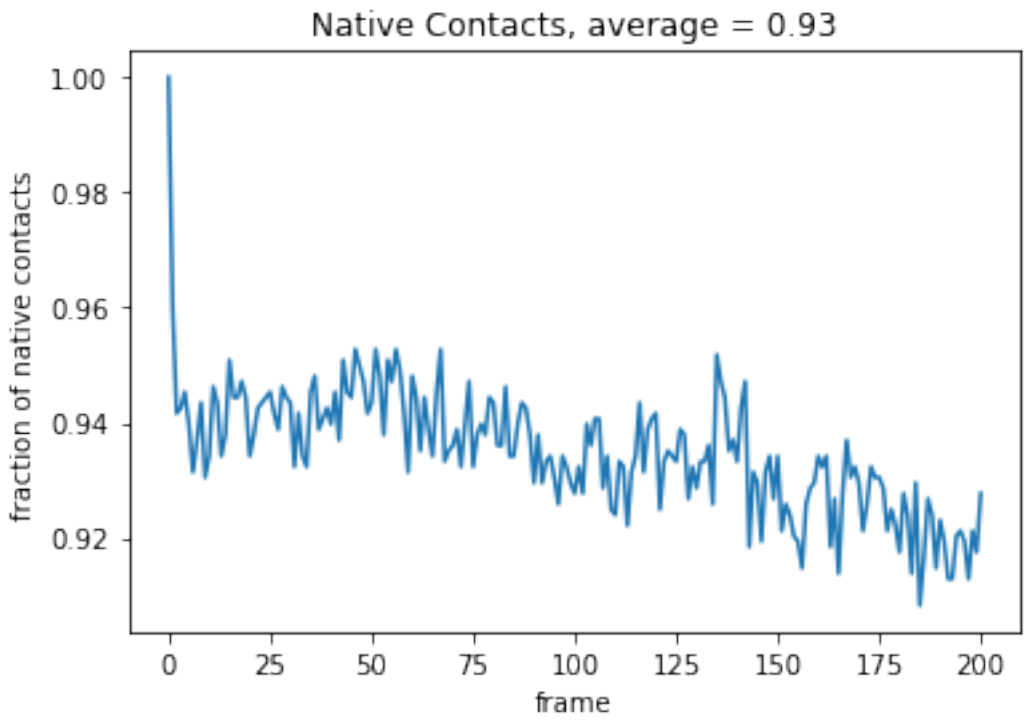
[35]:
```
<MDAnalysis.analysis.contacts.Contacts at 0x130a03040>
```

[36]:
```
universe.trajectory.ts
```

[36]:
```
< Timestep 0 with unit cell dimensions [99.683105 99.683105 99.683105 90.
90.       90.       ] >
```

[37]:
```
average_contacts = np.mean(ca1.timeseries[:, 1])
fig, ax = plt.subplots()
ax.plot(ca1.timeseries[:, 0], ca1.timeseries[:, 1])
ax.set(xlabel='frame', ylabel='fraction of native contacts',
       title='Native Contacts, average = {:.2f}'.format(average_contacts))
fig.show()
```

```
<ipython-input-37-7e27c7031058>:6: UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  fig.show()
```

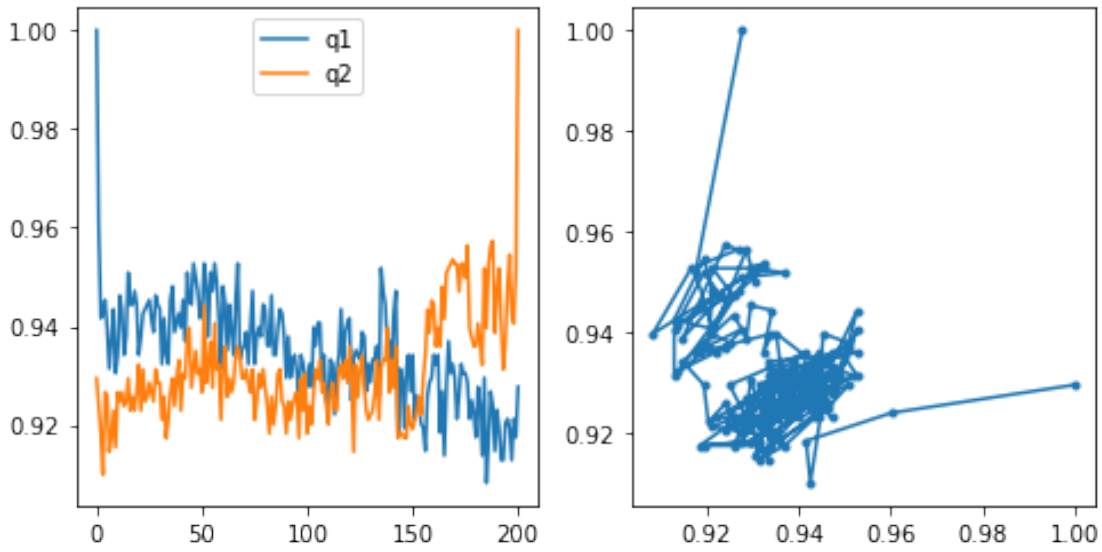


2.3.1 two-dimensional native contacts analysis It is often useful to analyse how the contact vary when changing the reference configuration. It is quite easy to perform this analysis with `contacts` and its built-in `q1q2` function. This function assumes that the two reference configurations coincide with the endpoints of the traejctory, thus showing how the system evolves in time in the space of native contacts.

[99]:
```
q1q2 = contacts.q1q2(universe, 'name CA', radius=8.0)
q1q2.run()

f, ax = plt.subplots(1, 2, figsize=plt.figaspect(0.5))
ax[0].plot(q1q2.timeseries[:, 0], q1q2.timeseries[:, 1], label='q1')
ax[0].plot(q1q2.timeseries[:, 0], q1q2.timeseries[:, 2], label='q2')
ax[0].legend(loc='best')
ax[1].plot(q1q2.timeseries[:, 1], q1q2.timeseries[:, 2], '.-')
f.show()
```

```
<ipython-input-99-9ee4a03920f2>:9: UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  f.show()
```



5.4 2.4 Hydrogen bond analysis

The Hydrogen Bond Analysis is a fundamental step in many simulation analysis pipelines and MDAnalysis contains a module that is dedicated to this task. The hydrogen bond network of the system is defined as an array of hydrogen bonds, each one characterised by six variables: 1. the frame in which the bond occurs 2. the donor ID 3. the ID of the hydrogen atom attached to the donor 4. the acceptor ID 5. the length of the hydrogen bond 6. the angle of the hydrogen bond (*donor-hydrogen-acceptor*)

[38]:
```
from MDAnalysis.analysis.hydrogenbonds.hbond_analysis import␣
↪HydrogenBondAnalysis as HBA
```

[39]:
```
hbonds = HBA(universe=universe)
hbonds.hydrogens_sel = hbonds.guess_hydrogens("protein")
hbonds.acceptors_sel = hbonds.guess_acceptors("protein")
hbonds.run(start=0,stop=80) # can be extended to the full trajectory
```

[39]:
```
<MDAnalysis.analysis.hydrogenbonds.hbond_analysis.HydrogenBondAnalysis at
0x13174b730>
```

[29]:
```
hbonds.hbonds[0]
```

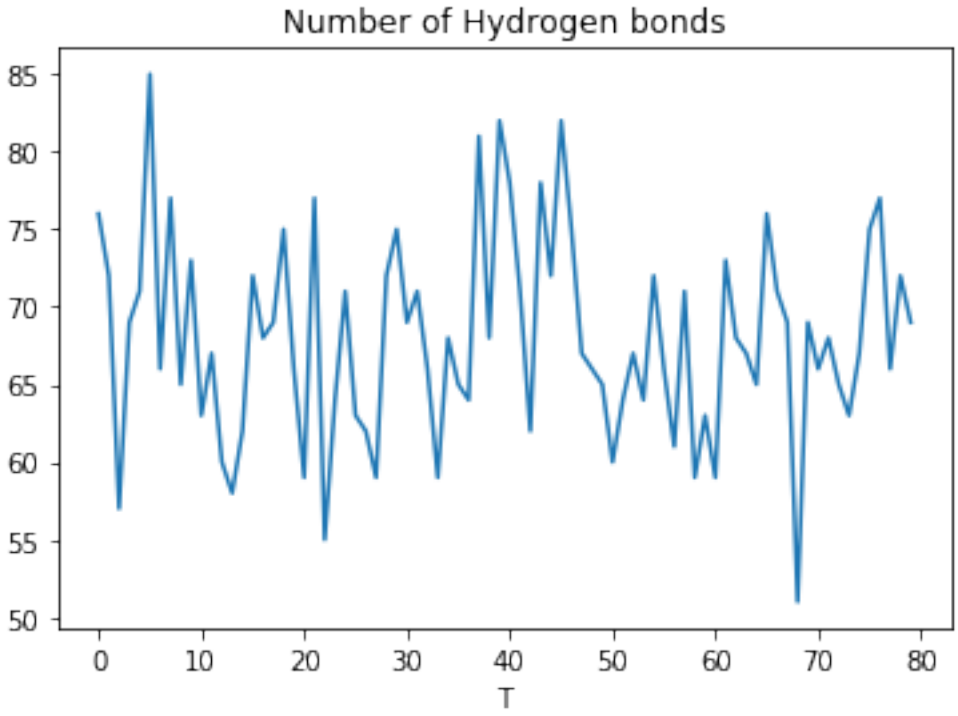
[29]:
```
array([   0.        ,   19.        ,   20.        , 1578.        ,
          2.85410022,  159.29822416])
```

[40]:
```
plt.plot(hbonds.count_by_time())
print(hbonds.count_by_ids())
```

```
plt.title('Number of Hydrogen bonds')
plt.xlabel("T")
```

```
[[2049 2050 2264   73]
 [2009 2010 2264   68]
 [2256 2257 2331   61]
 …
 [1135 1136 1147    1]
 [3045 3048 1681    1]
 [2611 2612 2564    1]]
```

[40]: `Text(0.5, 0, 'T')`



5.5 2.5 Ramachandran analysis

It is possible to project the behaviour of a biomolecule a Molecular Dynamics simulation in terms of fewer observables, called collective variables.

For example, this can be done for any pair of backbone dihedral angles (ψ, ϕ) in a protein:

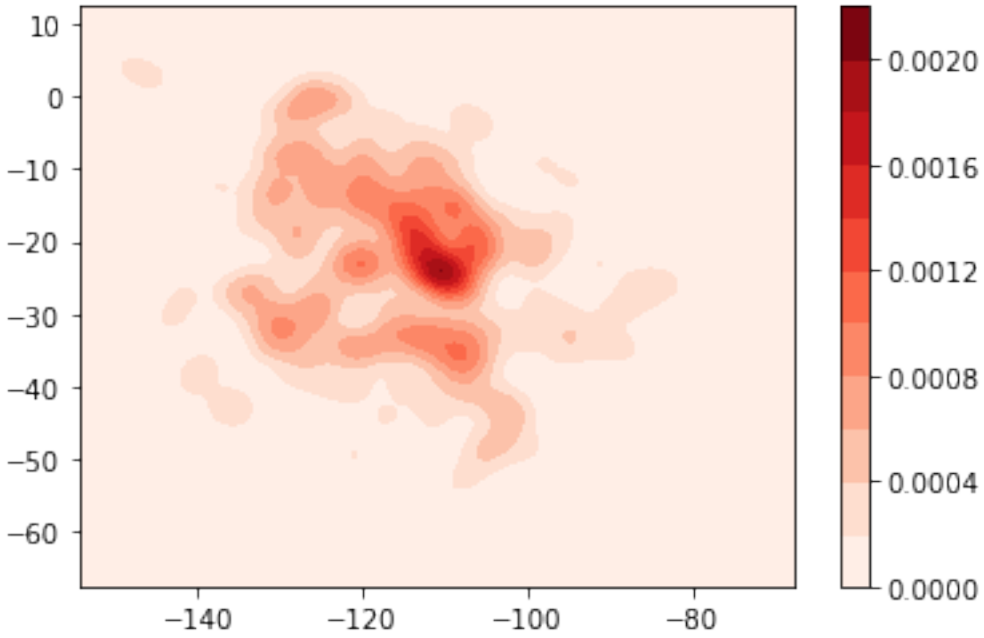
$FE(\psi, \phi) = -k_b T \ \ln(p(\psi, \phi))$

Let's see how the values of the dihedral angles of the catalytic residue of *Adenylate Kinase* are distribute in our short sample simulation.

[41]:
```
from MDAnalysis.analysis.dihedrals import Ramachandran
```

[42]:
```
r = universe.select_atoms("resid 88") # catalytic residue
R = Ramachandran(r).run()
print(R.angles.shape)
# here a single residue is selected, so two dihedrals are retrieved
sns.kdeplot(R.angles[:,0,0], R.angles[:,0,1], cmap="Reds", shade=True,␣
 ↪bw=2,cbar=True)
plt.show()
```

```
(201, 1, 2)
```



5.6 2.6 PCA

The covariance matrix of a sampled trajectory is defined as:

$C_{ij} := \sigma_{ij}^2 = \langle (x_i - \langle x_i \rangle) \rangle \langle (x_j - \langle x_j \rangle) \rangle$

It is real and symmetric, so it is always diagonalizable. The eigenvectors of the diagonal covariance matrix are called *principal components* and form an orthonormal base. Each eigenvalue represents the variance of the sample that is captured by the corresponding eigenvector.

PCA has been widely applied to Molecular Dynamics data for: - reconstructing the Free Energy landscape; - building the Essential Dynamics subspace; - assessing if a structure is equilibrated.

[43]:
```
import MDAnalysis.analysis.pca as pca
```

[44]:
```
pca_analysis = pca.PCA(universe, select='name CA')
pca_analysis.run()
```

[44]:
```
<MDAnalysis.analysis.pca.PCA at 0x13169ab80>
```

Our trajectory has already been aligned (on the first frame). If this is not true we could tell *pca* to align the trajectory.

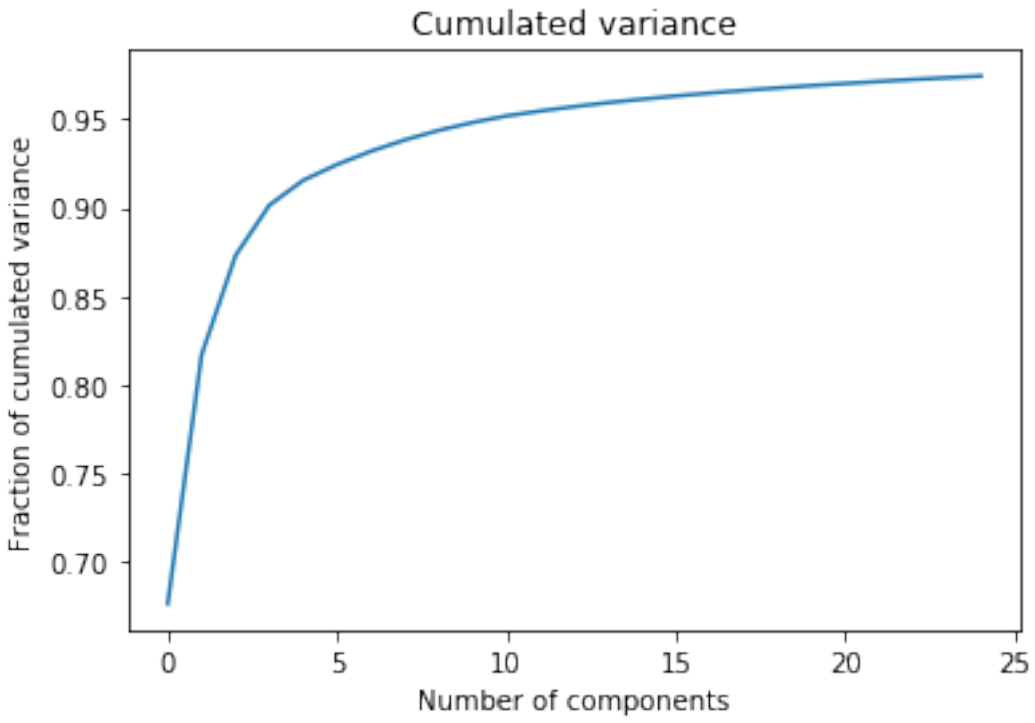
[45]:
```
pca_analysis = pca.PCA(universe, select='name CA',align=True)
pca_analysis.run()
```

[45]:
```
<MDAnalysis.analysis.pca.PCA at 0x1316a0190>
```

each component captures a certain variance, and the components are sorted from larger to smaller variance. We can plot the *cumulated variance* as a function of the number of components to get an idea of how many we need to explain the observed behaviour with a certain "confidence".

[46]:
```
plt.plot(pca_analysis.cumulated_variance[:25])
plt.title("Cumulated variance")
plt.xlabel("Number of components")
plt.ylabel("Fraction of cumulated variance")
```

[46]:
```
Text(0, 0.5, 'Fraction of cumulated variance')
```



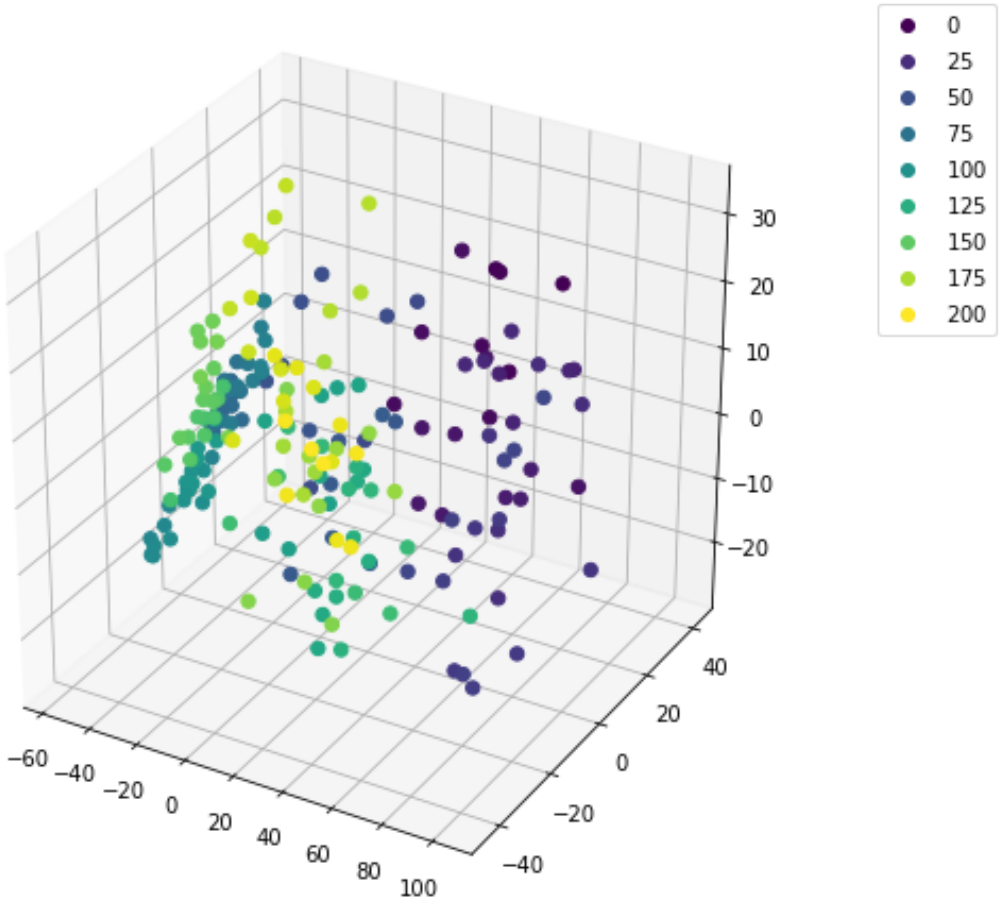
[47]:
```
pca_space = pca_analysis.transform(calpha, 3)
```

[48]:
```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import ListedColormap
```

[49]:
```
fig = plt.figure(figsize=(6,6))
ax = Axes3D(fig)

sc = ax.scatter(pca_space[:,0], pca_space[:,1], pca_space[:,2],
                c=np.arange(pca_space.shape[0]), s=40, marker='o', alpha=1)
# legend
plt.legend(*sc.legend_elements(), bbox_to_anchor=(1.05, 1), loc=2)
```

[49]:
```
<matplotlib.legend.Legend at 0x163dd6dc0>
```



6 3. Analysis building blocks

In our brief but intense journey through the analysis of Molecular Dynamics data we realized how unnecessary it is to re-invent the wheel (i.e. rewrite the analysis functions) every time. The vast majority of the useful functions are already efficiently implemented in MDAnalysis. This is not always the case of course.

The MDAnalysis.analysis.base Class is written exactly for this purpose. It contains a few Template Classes that allow to quickly create your custom analysis Class, while maintaining the MDAnalysis "style".

[50]:
```
import MDAnalysis.analysis.base as base
```

6.0.1 3.1 AnalysisFromFunction

Now we will see `AnalysisFromFunction`, the simplest of these template modules. Given a function working on one or more AtomGroups, this template converts the function in an Analysis that covers the whole trajectory.

Example: we want to calculate the angle between the centers of mass of the three domains of *Adenylate Kinase* over the whole trajectory. We already know from the literature (see for example here) where to place the boundaries between the domains.

First, we have to define a function that computes the angle between the domains.

[51]:
```
def angle_between_domains(sel1,sel2,sel3):
    # compute centers of mass
    com1 = sel1.center_of_mass()
    com2 = sel2.center_of_mass()
    com3 = sel3.center_of_mass()
    # compute vectors
    r12 = com1 - com2
    r32 = com3 - com2
    # compute cosine
    cosine = np.dot(r12, r32) / (np.linalg.norm(r12) * np.linalg.norm(r32))
    # compute angle
    angle = np.arccos(cosine)
    return angle
```

Second, we define these domains.

[52]:
```
nmp = universe.select_atoms("name CA and resid 30:67")
core = universe.select_atoms("name CA and (resid 1:29 or resid 68:117 or resid␣
 ↪161:214)")
lid = universe.select_atoms("name CA and resid 118:160")
```

Third: we check that the function works

[53]:
```
ang = angle_between_domains(nmp,core,lid)
print(ang)
```

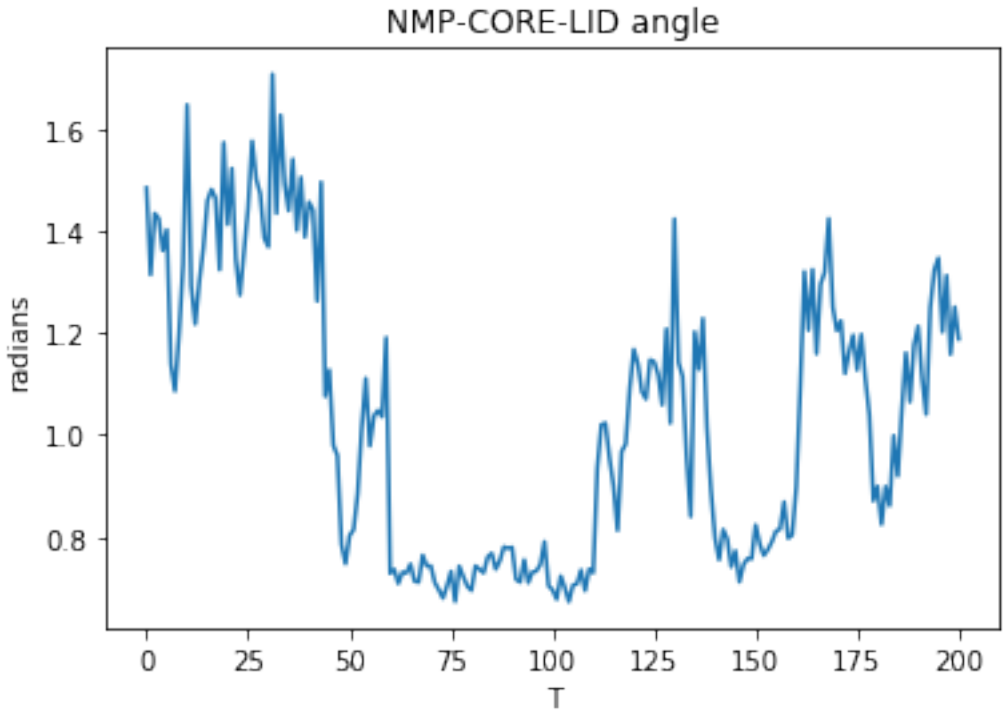
```
1.4842741486516953
```

Fourth: we use `AnalysisFromFunction` to iterate over the full trajectory.

[54]:
```
angle_series = base.AnalysisFromFunction(angle_between_domains,universe.
↪trajectory,nmp,core,lid).run()
```

[55]:
```
plt.plot(angle_series.results)
plt.title("NMP-CORE-LID angle")
plt.ylabel("radians")
plt.xlabel("T")
```

[55]:
```
Text(0.5, 0, 'T')
```



7 4. EXPORTING DATA

MDAnalysis allows you to export your selection into a variety of formats, such as *vmd* or *pymol*. Here we see a simple example of building a vmd script that preserves a selection that we defined in MDAnalysis.

[272]:
```
# define a selection
g = universe.select_atoms('resid 30')
# write down the selection
g.write("mda_30.vmd", name="mda_30")
```

After you load your pdb in vmd you can source your script with `source mda_30.vmd`. Then you will see this selection appearing under `Graphics->Representation->Selection->Singlewords`.

It is possible to write non-standard pdb files in which you may want to store a result you obtained in MDAnalysis. For example, assume that we would like to write a pdb containing only the C_α atoms of a protein. Our aim is to save the value of RMSF extracted in **2.2** directly inside the file. We can use a non-informative (in this case) attribute present in the pdb file, such as the *occupancy* and save the values of RMSF inside this field.

[56]:
```
#add the attribute
universe.add_TopologyAttr("occupancies")
# select the atoms
calpha_to_write = universe.select_atoms('name CA')
print(calpha_to_write.n_atoms)
# overwrite the field "occupancy"
calpha_to_write.atoms.occupancies = myRMSF.rmsf
# write the file
calpha_to_write.write("./CA_fake_occupancies.pdb")
```

```
214
```

```
//miniconda3/envs/SAT/lib/python3.8/site-
packages/MDAnalysis/coordinates/PDB.py:1026: UserWarning: Found no information
for attr: 'altLocs' Using default value of ' '
  warnings.warn("Found no information for attr: '{}'"
//miniconda3/envs/SAT/lib/python3.8/site-
packages/MDAnalysis/coordinates/PDB.py:1026: UserWarning: Found no information
for attr: 'icodes' Using default value of ' '
  warnings.warn("Found no information for attr: '{}'"
//miniconda3/envs/SAT/lib/python3.8/site-
packages/MDAnalysis/coordinates/PDB.py:1026: UserWarning: Found no information
for attr: 'tempfactors' Using default value of '0.0'
  warnings.warn("Found no information for attr: '{}'"
```

Now you can load the resulting file **CA_fake_occupancies.pdb** in VMD. If you choose to color the atoms according to their *occupancy* the color scale will reflect the values of RMSF.

8 EXERCISES

8.1 EXERCISE 1: changing the reference in RMSD calculations

The RMSD analysis showed in **2.1.1** every frame of the trajectory is compared to a conformation in the open state. Try to perform the same calculation choosing the closed pdb ***adk_closed.pdb***

as reference.

Does the resulting plot look reasonable?

Try to compute the RMSD with respect to the last frame of the trajectory.

[4]:
```
from MDAnalysis.analysis import align
from MDAnalysis.analysis.rms import rmsd
```

[5]:
```
u_closed = mda.Universe(str(PSF),str(cl_PDB))
```

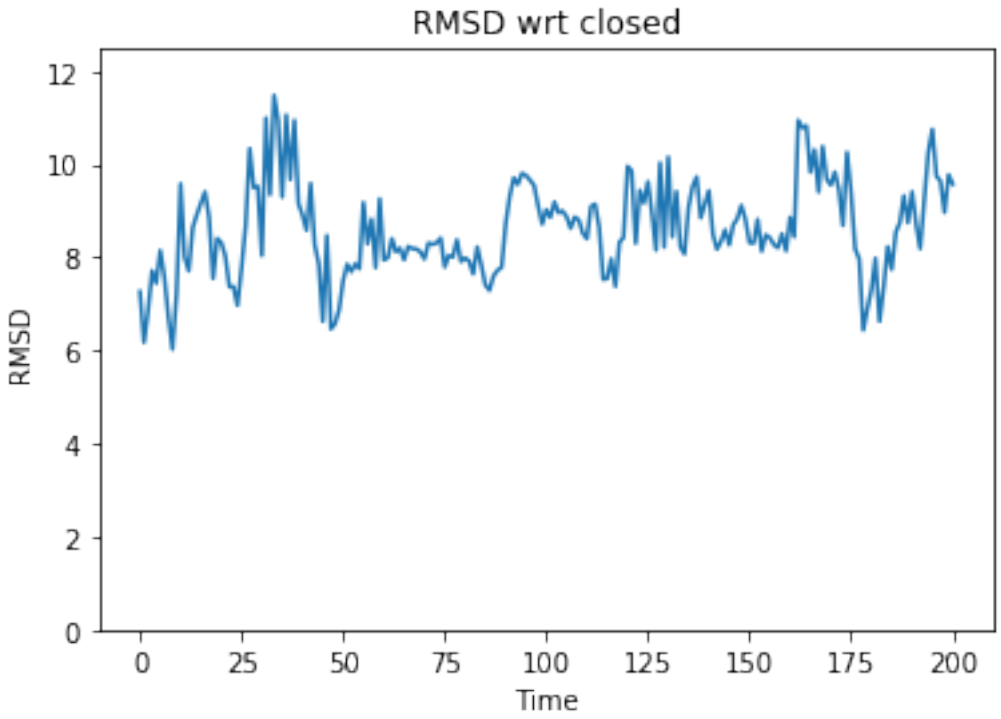
[6]:
```
alignment_closed = align.
↪AlignTraj(universe,reference=u_closed,align=True,select="name CA")
```

[7]:
```
alignment_closed.run()
```

[7]:
```
<MDAnalysis.analysis.align.AlignTraj at 0x126405190>
```

[10]:
```
plt.plot(alignment_closed.rmsd)
plt.xlabel("Time")
plt.ylabel("RMSD")
plt.title("RMSD wrt closed")
plt.ylim(0,max(alignment_closed.rmsd)+1)
```

[10]:
```
(0, 12.477255055109813)
```



[11]:
```
universe.trajectory.ts
```

[11]:
```
< Timestep 0 with unit cell dimensions [99.683105 99.683105 99.683105 90.
90.       90.       ] >
```

[13]:
```
universe.atoms.positions
```

[13]:
```
array([[41.149998, 32.739998, 53.61    ],
       [41.63    , 32.47    , 54.45    ],
       [40.43    , 33.41    , 53.850002],
       …,
       [40.72    , 30.03    , 43.239998],
       [40.079998, 29.22    , 43.969997],
       [40.24    , 31.02    , 42.61    ]], dtype=float32)
```

[14]:
```
universe.trajectory[-1]
```

[14]:
```
< Timestep 200 with unit cell dimensions [99.708336 99.708336 99.708336 90.
90.       90.       ] >
```

[15]:
```
universe.atoms.positions
```

[15]:
```
array([[44.120003, 30.340002, 55.43    ],
       [43.97    , 29.340002, 55.32    ],
       [43.800003, 30.570002, 56.350002],
       …,
       [44.52    , 24.6     , 43.590004],
       [45.29    , 24.250002, 44.52    ],
       [43.300003, 24.540003, 43.710003]], dtype=float32)
```

[16]:
```
u_last = mda.Merge(universe.atoms)
```

[17]:
```
u_last.atoms.positions
```

[17]:
```
array([[44.120003, 30.340002, 55.43    ],
       [43.97    , 29.340002, 55.32    ],
       [43.800003, 30.570002, 56.350002],
       …,
       [44.52    , 24.6     , 43.590004],
       [45.29    , 24.250002, 44.52    ],
       [43.300003, 24.540003, 43.710003]], dtype=float32)
```

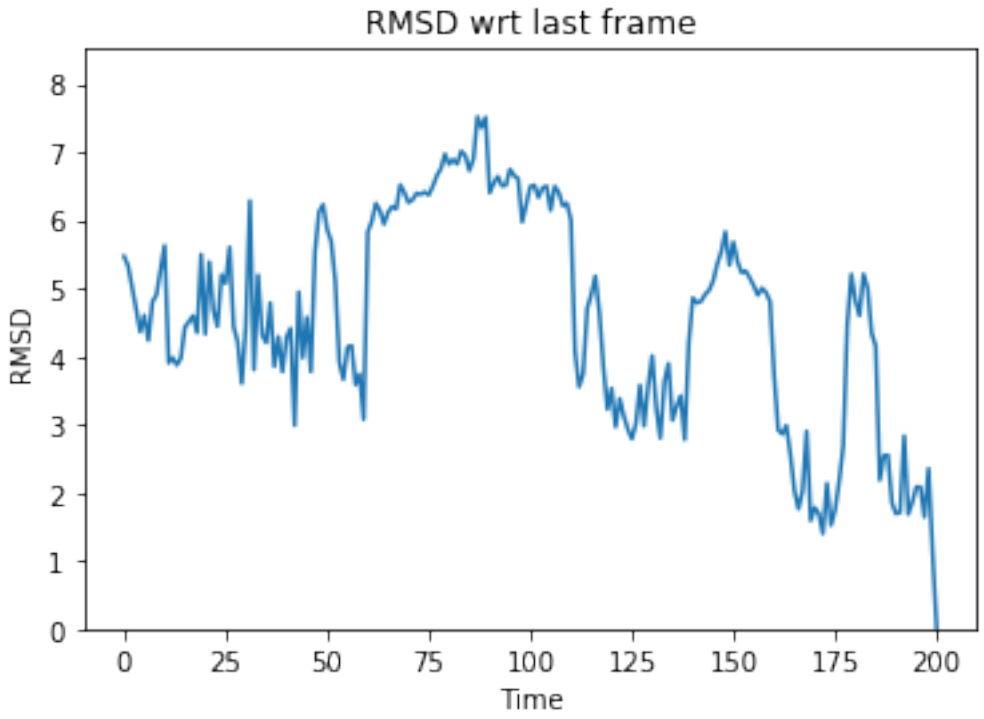
[18]:
```
alignment_last = align.
↪AlignTraj(universe,reference=u_last,align=True,select="name CA")
```

[19]:
```
alignment_last.run()
```

[19]:
```
<MDAnalysis.analysis.align.AlignTraj at 0x1277c12b0>
```

[20]:
```
plt.plot(alignment_last.rmsd)
plt.xlabel("Time")
plt.ylabel("RMSD")
plt.title("RMSD wrt last frame")
plt.ylim(0,max(alignment_last.rmsd)+1)
```

[20]:
```
(0, 8.516451160761587)
```



8.2 EXERCISE 2. bridging MD and experimental data: beta factors

The RMSF extracted in **2.2** can be compared to the experimental atomic beta factors. Do it with MDAnalysis!

Hint: these values are also dubbed *Temperature factors*.

[21]:
```
from MDAnalysis.analysis.rms import RMSF
```

[22]:
```
u_op = mda.Universe(str(PDB))
```

```
//miniconda3/envs/SAT/lib/python3.8/site-
packages/MDAnalysis/topology/PDBParser.py:330: UserWarning: Element information
```

```
is absent or missing for a few atoms. Elements attributes will not be populated.
  warnings.warn("Element information is absent or missing for a few "
```

[23]:
```
heavies = u_op.select_atoms("not name H*")
```

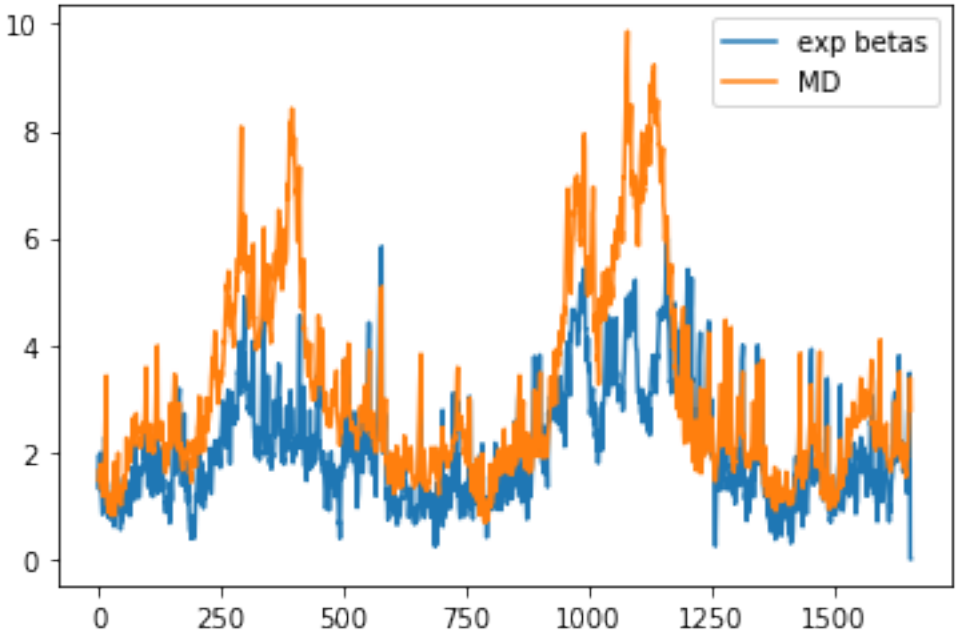
[24]:
```
heavies.tempfactors
```

[24]:
```
array([38.38000107, 26.13999939, 31.72999954, …, 69.52999878,
        0.        ,  0.        ])
```

[25]:
```
heavies_traj = universe.select_atoms("not name H*")
myRMSF = RMSF(heavies_traj).run()
```

[28]:
```
plt.plot(heavies.tempfactors/20,label="exp betas")
plt.plot(myRMSF.rmsf,label="MD")
plt.legend()
```

[28]:
```
<matplotlib.legend.Legend at 0x128a2a1f0>
```



8.3 EXERCISE 3: a new Analysis from function

In **3.1** we calculate the angle between the domains. It is a useful measure, that allows us to discriminate between more and less compact conformations of our enzyme. However, it does not tell anything about the mutual orientation of the mobile domains. In order to estimate this variable, we have to compute the dihedral angle formed by the two mobile domains and the CORE. This is

a quantity that measures the angle between two planes, so it is necessary to define two points on the CORE domains. A quick and easy guess for these points is represented by the residues that lie at the borders between the CORE and two external domains (residue 161 for the LID and residue 29 for the NMP).

Write a function `dihedral_between_domains` that computes this angle and the corresponding `AnalysisFromFunction` class.

Hint: `MDAnalysis.lib.mdamath.dihedral` method can be quite useful when computing a dihedral angle.

[32]:
```
from MDAnalysis.lib.mdamath import dihedral as compute_dihedral
```

[33]:
```
import MDAnalysis.analysis.base as base
```

[29]:
```
def dihedral_between_domains(sel1,sel2,sel3,sel4):
    # compute centers of mass
    com1 = sel1.center_of_mass()
    com2 = sel2.center_of_mass()
    com3 = sel3.center_of_mass()
    com4 = sel4.center_of_mass()
    # compute vectors
    r12 = com2 - com1
    r23 = com3 - com2
    r34 = com4 - com3
    # compute dihedral
    dihedral = compute_dihedral(r12,r23,r34)
    return dihedral
```

[30]:
```
nmp = universe.select_atoms("name CA and resid 30:67")
core_nmp = universe.select_atoms("resid 29")
core_lid = universe.select_atoms("resid 161")
lid = universe.select_atoms("name CA and resid 118:160")
```

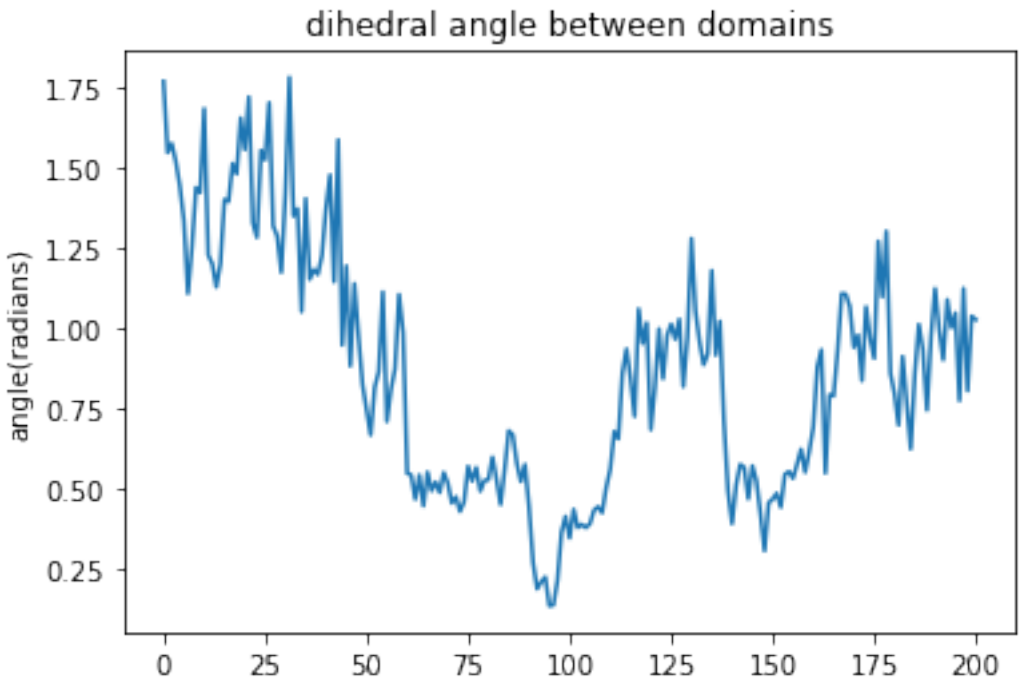
[34]:
```
dihedral_between_domains(nmp,core_nmp,core_lid,lid)
```

[34]:
```
1.7706047784276293
```

[35]:
```
dihedral_analysis = base.AnalysisFromFunction(dihedral_between_domains,universe.
 ↪trajectory,nmp,core_nmp,core_lid,lid).run()
```

[37]:
```
plt.plot(dihedral_analysis.results)
plt.title("dihedral angle between domains")
plt.ylabel("angle(radians)")
```

[37]:
```
Text(0, 0.5, 'angle(radians)')
```



8.4 EXERCISE 4: more than Adenylate Kinase

The Collaborative Computational Project for Biomolecular Simulation (CCPBioSim) created SlimMD, a small but very well curated database of MD trajectories, including:

- *DNA* fragments
- *Sars Cov2* proteins
- *Protein-membrane* systems

Choose the system that you like the most, download the trajectory and use MDAnalysis to extract information about its time-evolution.

9 EXTRAS

9.1 How to load many snapshots in a single trajectory?

[38]:
```
list_of_snapshots = [str(PDB),str(cl_PDB)]
u_mixed = mda.Universe(str(PSF),list_of_snapshots)
```

```
//miniconda3/envs/SAT/lib/python3.8/site-
packages/MDAnalysis/coordinates/base.py:865: UserWarning: Reader has no dt
information, set to 1.0 ps
  warnings.warn("Reader has no dt information, set to 1.0 ps")
```

[39]:
```
u_mixed.trajectory
```

[39]:
```
<ChainReader containing adk_open.pdb, adk_closed.pdb with 2 frames of 3341
atoms>
```

9.2 Use van der Waals radii to estimate contacts

[229]:
```
mda.topology.tables.vdwradii
```

[229]:
```
{'H': 1.1,
 'HE': 1.4,
 'LI': 1.82,
 'BE': 1.53,
 'B': 1.92,
 'C': 1.7,
 'N': 1.55,
 'O': 1.52,
 'F': 1.47,
 'NE': 1.54,
 'NA': 2.27,
 'MG': 1.73,
 'AL': 1.84,
 'SI': 2.1,
 'P': 1.8,
 'S': 1.8,
 'CL': 1.75,
 'AR': 1.88,
 'K': 2.75,
 'CA': 2.31,
 'NI': 1.63,
 'CU': 1.4,
 'ZN': 1.39,
 'GA': 1.87,
 'GE': 2.11,
 'AA': 1.85,
 'SE': 1.9,
 'BR': 1.85,
 'KR': 2.02,
 'RR': 3.03,
 'SR': 2.49,
 'PD': 1.63,
 'AG': 1.72,
 'CD': 1.58,
 'IN': 1.93,
 'SN': 2.17,
 'SB': 2.06,
```

```
 'TE': 2.06,
 'I': 1.98,
 'XE': 2.16,
 'CS': 3.43,
 'BA': 2.68,
 'PT': 1.75,
 'AU': 1.66,
 'HH': 1.55,
 'TL': 1.96,
 'PB': 2.02,
 'BI': 2.07,
 'PO': 1.97,
 'AT': 2.02,
 'RN': 2.2,
 'FR': 3.48,
 'RA': 2.83,
 'U': 1.86}
```

[40]:
```
from MDAnalysis.analysis import contacts
```

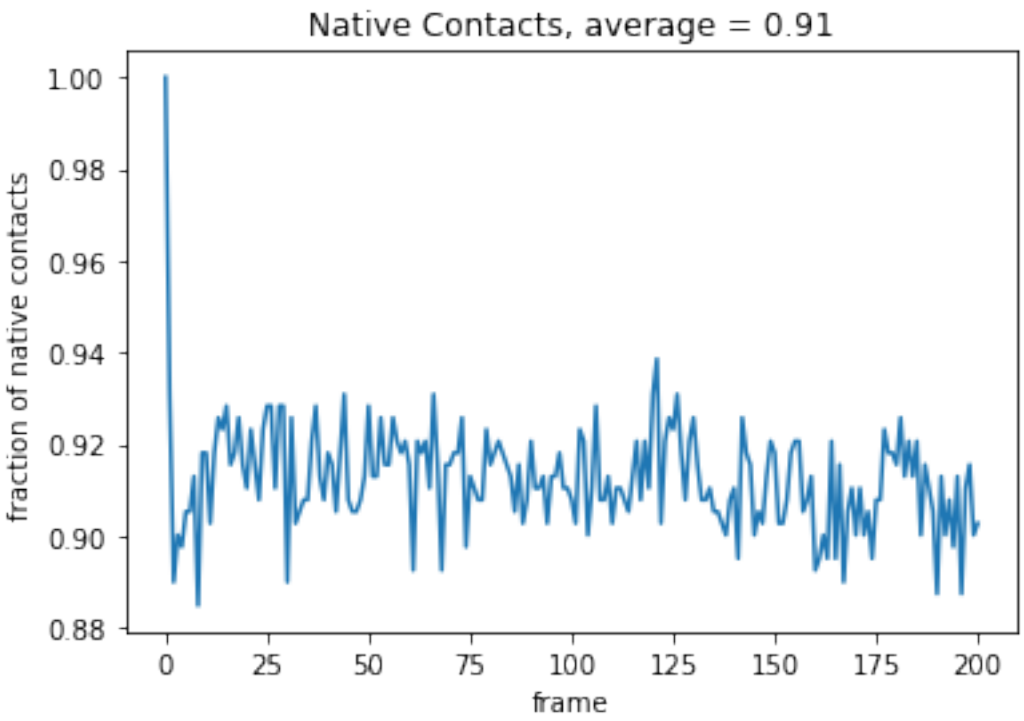
[47]:
```
sel_calpha = "(name CA)"
start_calpha = universe.select_atoms(sel_calpha)
ca1 = contacts.Contacts(universe, select=(sel_calpha, sel_calpha),␣
 ↪method="radius_cut",
                         refgroup=(start_calpha, start_calpha),radius=mda.
 ↪topology.tables.vdwradii["C"]*3)
```

[48]:
```
ca1.run()
```

[48]:
```
<MDAnalysis.analysis.contacts.Contacts at 0x129737af0>
```

[49]:
```
average_contacts = np.mean(ca1.timeseries[:, 1])
fig, ax = plt.subplots()
ax.plot(ca1.timeseries[:, 0], ca1.timeseries[:, 1])
ax.set(xlabel='frame', ylabel='fraction of native contacts',
       title='Native Contacts, average = {:.2f}'.format(average_contacts))
fig.show()
```

```
<ipython-input-49-7e27c7031058>:6: UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  fig.show()
```


Native Contacts, average = 0.91
fraction of native contacts
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0
25
50
75
100
125
150
175
200
frame