Notes Taken During MDAnalysis Tutorial

- The essential purpose of this package is to read data files created by MD simulation packages into python–standard storage containers (e.g. lists, though usually you'll want to use `numpy` containers)
- First need to load the package with `import MDAnalysis`

- Read in the files with the format (Note: ignore the leading >, it's just so I can distinguish different lines in code):

```
> <variable_name> = MDAnalysis.Universe("<topology_file>","<trajectory_file>")
```

Overview of classes and class members

- the `Universe` class contains all the information which is known about the system from the topology and trajectory files
- Can also read in multiple trajectories at once, like

```
<variable_name> = MDAnalysis.Universe("<topology_file>", ["traj1", "traj2", ...])
```

- The tutorial uses a provided PSF and DCD, so we type `u = MDAnalysis.Universe(PSF, DCD)`, so all further references to the `Universe` will be using `u`
- each particle/atom is read into an `Atom` object, containing the force on the atom, veclocity, index, and position
  - `u.atoms` is the list of all the atoms in the system. It is manipulated just like a regular list
- residues are listen using the `residue` class member, with ResidueGroup being a list of residue objects
  - `residue` class member:

```
> u.atoms[100:130].residues  # [x:y] is an inclusive list slice
    # output
    <ResidueGroup with 3 residues>
```

  - ResidueGroup:

```
> list(u.atoms[100:130].residues
    # output
    [<Residue LEU, 6>, <Residue GLY, 7>, <Residue ALA, 8>]
```

- Chains/segments can accessed in a similar manner using `segment`

Atom selections

- Atom selections uses VMD syntax: `python  > CA = u.select_atoms("protein and name CA")` where we select all C_α in the protein
- range atom selections by index, using `first-last` or `first:last` are inclusive: `python  u.select_atoms("resid 5-100")`
- residue names (resnames) can include wildcards `python  u.select_atoms("resname ASP HS*")  # selects HSE, HSP, and HSD (protonation states of HIS)`
- geometric selection is possible with `around <distance> <selection`: `python  u.select_atoms("resname ASP and around 4.0 resname LYS")` chooses asparagine residues within 4.0 Å of lysine residues
- AtomGroups can be written to a file using `write()`: `python  > NMP = u.select_atoms("protein and resid 30-59")  > NMP.write("<some_file>.pdb")`

AtomGroups

- getting the positions of an atom selection produces a `numpy.ndarray`
  - several quantities can be directly calculated from atom groups:
    * center of mass,using `center_of_mass()`, and centroid, using `center_of_geometry()`
    * total mass, using `total_mass()`
    * total charge, using `total_charge()`
    * radius of gyration, using `radius_of_gyration()`
      · recall that the radius of gyration is

$$R_{gyr.} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} m_i(\mathbf{r}_i - \mathbf{R})^2}$$

        where N is the number of monomer (atoms, residues, etc), $\mathbf{r}_i$ is the position of the ith monomer, and $\mathbf{R}$ is the mean position of all the monomers
    * principal axes, $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, using `principal_axes()`
      · diagonalizes the tensor of interia, which is done manually with `moment_of_inertia()`

$$\Lambda = U^T I U$$

        where $U = (\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ and I is the identity matrix

An example of a calculation

- here's an example of the calculation of the center of mass and center of geometry for each of three domains [*CORE* (residues 1-29, 60-121, 160-214), *NMP* (residues 30-59), *LID* (residues 122-159)] in the provided PSF and DCD

```
> domains = {
    'CORE':u.select_atoms("protein and (resid 1-29 60-121 160-214)"),
    'NMP':u.select_atoms("protein and resid 30-59"),
    'LID':u.select_atoms("protein and resid 122-159")
}
> domain_centroid = dict((name, dom.centroid()) for name,dom in domains.items())
> domain_com = dict((name, dom.center_of_mass()) for name,dom in domains.items()))
```

  which outputs

```
> print domain_centroid
    # output
    {'LID': array([-15.16074944,   2.11599636,  -4.37305355], dtype=float32),
    'CORE': array([ 4.43884087,  2.05389476,  1.63895261], dtype=float32),
    'NMP': array([ -2.99990702, -13.62531662,  -2.93235731], dtype=float32)}
> print domain_com
    # output
    {'LID': array([-15.11337499,   2.12292226,  -4.40910485]),
    'CORE': array([ 4.564116  ,  2.08700105,  1.54992649]),
    'NMP': array([ -3.20330174, -13.60247613,  -3.06221538])}
```

  - These dictionaries can be manipulated with any math commands you could regularly use. for example, to get the distance between the center of masses of two domains, you could do

```
> print norm(domain_com['CORE'] - domain_com['NMP'])
```

  - many more examples of the types of calculations you can do is found here

---

Trajectory Analysis

- the trajectory initially read in is accessed from the `trajectory` member, as in `u.trajectory`, and actions done on the entire trajectory are performed using a standard loop iterating over the trajectory:

```
    # "".format() formats a string like printf
  > for ts in u.trajectory:
  ...   print "Frame: {0:5d}, Time: {1:8.3f} ps".format(ts.frame, u.trajectory.time)
  ...   print "Rgyr: {0:5d} A".format(u.atoms.radius_of_gyration())

      Frame:     0, Time:     0.000 ps
      Rgyr: 16.669 A
```

Time

- the `time` member contains all the about the system for **only** the current timestep
- again, to access `time`, one would iterate over the entire trajectory (and usually use `numpy` containers),

```
> Time = []  # make an empty list
> protein = u.select_atoms("protein")  # select all the atoms of the protein
> for ts in u.trajectory:
    Rgyr.append((u.trajectory.time, protein.radius_of_gyration()))
    # appends a tuple (time, rgyr) to the list
> Rgyr = numpy.array(Rgyr)  # create a numpy array from the list
```

- this can be simplified using python's weird syntax,

```
> protein = u.select_atoms("protein")
> data = numpy.array([u.trajectory.time, protein.radius_of_gyration()) for ts in u.trajectory])
  # [] creates a list out of the data, with the in--line for loop
  # an aside: I usually use this to read in lines of data from a file,
  # data = [line.split() for line in open("<file>",r)]
  # creates a list of lists, where each line is a list, broken by whitespace
```

- this array can then be plotted using a package like `matplotlib`

Trajectory Iterator

- Traditional python list indexing syntax can be used to jump to specific frames.
  - `u.trajectory[50]` accesses the 50th frame of the trajectory
  - trajectory lists can also be sliced, e.g,
    ```
    > for ts in u.trajectory[9:-10:5]
      ... print ts.frame
    ```
    prints every 5 frames, starting at the 10th frame and ending at the frame 10 timesteps before the end of the trajectory
  - Note that DCD and XTC both support index access and list slicing

Writing coordinates

- as mentioned above, the `write` command easily outputs single frames to a file,

```
> protein = u.select_atoms("protein")  # select only the protein, i.e. no waters
> protein.write("protein.gro")  # write to a gro file.
                                # file format is automatically chosen by the extension provided
```

- Typical way to write trajectories:
  1. get a trajectory writer with `MDAnalysis.Writer()`, specifying the atoms the frame(s) will contain
  2. use `write()` to write a new timestep to the trajectory
  3. close with `close()`. the tutorial suggests instead using python's `with` statement to allow the trajectory to be closed automatically,

     ```
     with open("<traj file>") as outfile
         ...
     ```

- many more examples are shown here

Some provided analysis functions

- RMSD is calculated using `MDAnalysis.analysis.rms.rmsd()` (as an aside, `MDAnalysis.analysis` is a stupidly redundant name)

  ```
  > import MDAnalysis.analysis.rms  # import the analysis module
  > backbone = u.select_atoms("backbone")  # selecting only the backbone
  > A = backbone.positions  # get the coordinates of the backbone atoms at the first timestep
  > u.trajectory[-1]  # ffwd to the last frame
  > B = bb.positions  # get the coordinates of the same atoms at the last frame
  > MDAnalysis.analysis.rms.rmsd(A, B)  # calculate the RMSD between the two frames
    # output
    <some number>
  ```

- Superpositions are calculated with `MDAnalysis.analysis.align`. I won't replicate the example, shown here

Example analysis of a membrane–protein interactions

- notes taken from this tutorial

Calculating the number of lipid atoms within a distance of the protein over time