Center Of Mass

Friday, November 28, 2025 8:20 PM

```
import numpy as np
```
→ library for handling arrays and math efficiently

```
import astropy.units as u
```
→ allows one to attach physical units (kpc, km s^{-1}, Myr) to numbers

```
from ReadFile import Read
```
→ function to load a snapshot file and return data

```
class CenterOfMass:
```
↳ The blueprint to calculate the galaxy's COM for the data in a given snapshot.

```
def __init__(self, filename, ptype):
        self.time, self.total, self.data = Read(filename)
```

- __init__ is the constructor

A constructor is a special method that gets called automatically when an object is created from a class.
The __init__ method initializes the newly created instance and is commonly used. It is called immediately after the object is created by __new__ method and is responsible for initializing the attributes of the instance → Source: geeksforgeeks

```
self.index = np.where(self.data['type']==ptype)
```

Finds which rows in data correspond to the particle types I want.
self.index is an array of indices (positions in the array) of particles of the selected type.

```
self.m = self.data['m'] [self.index]
self.x = self.data['x'] [self.index]
self.y = self.data['y'] [self.index]
self.z = self.data['z'] [self.index]
self.vx = self.data['vx'] [self.index]
self.vy = self.data['vy'] [self.index]
self.vz = self.data['vz'] [self.index]
```
each line extracts only the selected particle type from the snapshot.

```
self.vy = self.data['vy'][self.index]
self.vz = self.data['vz'][self.index]
```

from the snapshot.

function to calculate COM using the standard formula

```
def COMdefine(self,a,b,c, m):
    a_com = np.sum(a*m)/np.sum(m)
    b_com = np.sum (b*m)/np.sum(m)
    c_com = np.sum(c*m)/np.sum(m)
    return a_com,b_com,c_com
```

· Computes a weighted average along 3 axes using particle masses.
↳ takes arrays a,b,c (positions or velocities) and m(masses).

```
def COM_P(self,delta=0.1):
```

Computes the center of mass iteratively using the shrinking sphere method. delta = convergence tolerance, the computation stops when COM position changes by less than delta kpc.

```
x_COM, y_COM, z_COM =self.COMdefine(self.x,self.y,self.z,self.m)
r_COM = np.sqrt(xCOM**2 +yCOM**2 + z_COM**2)

x_new = self.x - x_COM
y_new = self.y - y_COM
z_new = self.z - z_COM
r_new = np.sqrt(x_new**2 + y_new**2 + z_new**2)
```

This chunk of code computes the distances of each particle from the COM estimate. Prepares for shrinking the sphere in the iterative process.

```
r_max = np.max(r_new)/2.0
change = 1000.0
```

This sets the initial radius of the sphere (half the farthest particle distance)
change tracks how much COM moves each iteration.

```
while change >delta:
    index2 =np.where(r_new<r_max)
    x2,y2, z2 = self.x[index2] ,self.y [index2],self.z [index2]
    m2 = self.m[index2]
```

Keeps shrinking until the COM converges.
Only considers the particles currently inside the sphere r_max.

```
    x_COM2, y_COM2, z_COM2 =self.COMdefine(x2,y2,z2,m2)
    r_COM2 = np.sqrt(x_COM**2 + y_COM**2 + z_COM**2)
    change = np.abs(r_COM -rCOM_2)
    r_max /=2.0
```

This computes the new COM with the smaller sphere.
It updates change — aka how the COM has moved.
Then, it shrinks the radius by half for the next iteration.

```
    x_new = self.x- x_COM2
    y_new = self.y - y_COM2
    z_new =self.z - z_COM2
    r_new = np.sqrt( x_new **2 + y_new **2+z_new**2)
    x_COM ,y_COM, z_COM= x_COM2, y_COM2, z_COM2
    r_COM = r_COM2
```

This updates distances for the next loop iteration.
It updates the current COM reference.

```
p_COM = np.array([x_COM, y_COM, z_COM])*u.kpc
return np.round(p_COM,2)
```

This returns the final 3D COM position as an array with the units in kpc.