Extragalactic Astronomy

Dynamical Friction in a stable Plummer's Sphere - Numerical Study

Efrem Maconi
814088

Academic Year 2020/2021

Dynamical Friction in a stable Plummer's Sphere Numerical Study

1 Introduction

We present the numerical study of the *dynamical friction process* that consists in the loss of momentum and kinetic energy of a massive body due to gravitational interaction with the surrounding matter space.

In our simulation we study the effect of this process on a perturber within a stable Plummer's sphere, approximated with a finite number of particles. We proceeded as follow:

1. initialization of a Plummer's sphere at equilibrium (remember that not all the Plummer's sphere are at equilibrium, that can be achieved selecting a particular distribution function; for more details see the lecture notes);
2. evolution of the generated configuration and check that is effectively in a stable state;
3. inclusion in the stable configuration, at a given distance from the center and with a given velocity, of a perturber, that consists in a massive particle such that

   $$m_{part} << m_{pert} << M_{sphere},$$

   where m_{part} is the mass of a single particle used to approximate the sphere.
4. evolve the new system and study its evolution: the effect of the perturber on the sphere's particles and, viceversa, the effect of the particles on the perturber (dynamical friction).

We study the dynamical friction process for two perturbers of different mass to verify how this process is related to their mass.

2 About the simulations

The simulations are made through the Joshua E. Barnes' *Tree Code* (for more details see `https://www.ifa.hawaii.edu/~barnes/treecode/treeguide.html`).

The Plummer's sphere at equilibrium is initialized through the *Section 1* of the jupyter notebook *Dynamical Friction In Plummer - Initial Conditions Code (Barnerstreecode).*

The execution of that notebook give as output two files,

`initial_conditions_file.txt`
`system_properties.txt,`

where the first one in used by the *Barnes' Tree Code* and contains the number of particles, the number of dimensions, the initial simulation time and the particles' mass, initial positions and velocities; the second one is used by the jupyter notebook devoted to the analysis.

Once you have runned the the Barnes' Tree Code (see below the command line for the terminal), the *Plummer Sphere Equilibrium (update) - Analysis Code (Barnerstreecode)* notebook is used to check whether the generated Plummer's sphere is at equilibrium or not.

If the Plummer's sphere is at equilibrium (i.e. its properties like the position and velocity distributions do not change in time), the perturber is added to the stable system through the *Section 4* of the *Dynamical Friction [...] - Initial Conditions Code [...]*: there the *.txt* files generated in the first section of the notebook are opened, the perturber's data (positions and velocities) are added and two new files are generated

`initial_conditions_file_perturber.txt`
`system_properties_perturber.txt,`

with the same function of the once generated before.

The new output of the Barners' Tree Code in analyzed in the *Dynamical Friction In Plummer - Analysis Code (Barnerstreecode)*.

In our simulations we considered a Plummer's sphere of mass $M = 1$, a scale radius $r_{scale} = 10$, approximated with 30'000 particles and, in the first case, a perturber of mass $M_{pert} = 0.03$, in the second case a perturber of mass $M_{pert} = 0.01$, both positioned at $t = 0$ on an orbit with $R_{apocenter} = somethin$ and $R_{pericenter} = something$.
Here the command lines used to run the *Barnes' Tree Code* from the terminal:

- to test the Plummer's equilibrium:

```
./treecode in=initial_conditions_file.txt out=%d.data dtime=0.05 eps=0.1 theta=0.5
options=out-phi tstop=320 dtout=10 > system_description.txt ,
```

- to test the dynamical friction effect:

```
./treecode in=initial_conditions_file_perturber.txt out=%d.data dtime=0.05 eps=0.1
theta=0.5 options=out-phi tstop=1600 dtout=10 > system_description_perturber.txt ,
```

where we used the following parameters' values:

- dtime=0.05: it's the *integration time step*; the more particles you use, the less has to be this parameter and, for our simulations, we choose $dtime = eps/2$;
- eps=0.1: it's the *gravitational softening parameter*; we divided the sphere in shells and computed the particles' mean distance in each of them; then we selected the minimum value and take, in our case, $eps = min(meanSeparation)/5$;
- theta=0.5: it's the *opening parameter*; a small value of theta resulats in a better precision but a bigger computational time; since we are using a big number of particles, we decided to set theta=0.5;
- tstop: it's the time at which the simulation ends; to verify the Plummer equilibrium we evolve the system for 10 dynamical times ($tstop = 320$); in the other cases we evolve it for a very long time ($tstop = 1600$);
- out=%d.data: this is the *name.data* file where the system information at each *dtout* are saved; we decided to save the system state not in a unique file, but in different ones in order to decrease the computational time during the analysis (the gain of time is very big!).

3 Equilibrium of the generated Plummer Sphere

We are now going to analyze if the generated Plummer's sphere, where we have initialized the system with the following probability density functions:

- radius pdf:
$$pdf(r)dr = \frac{3r^2}{b^3}\left(1 + \frac{r^2}{b^2}\right)^{-\frac{5}{2}}, \tag{1}$$
- theta pdf:
$$pdf(\theta)d\theta = \frac{\sin\theta}{2}d\theta, \tag{2}$$
- phi pdf:
$$pdf(\phi)d\phi = \frac{d\phi}{2\pi}, \tag{3}$$

- velocity extracted from the distribution function:

$$f(\epsilon)d\epsilon = \left(\psi - \frac{v^2}{2}\right)^{\frac{7}{2}} v^2 dv, \tag{4}$$

where ϵ is the *relative energy* ($\epsilon = \psi - \frac{v^2}{2}$) and ψ the *relative potential* ($\psi = -\phi_{Plum}$),

remains at equilibrium or not: to do that we set the initial conditions, evolve the system and then analyze the results at a given time: if the system is at equilibrium already from $t = 0$ the position and the velocity distributions shouldn't evolve and so be the same at any given time.

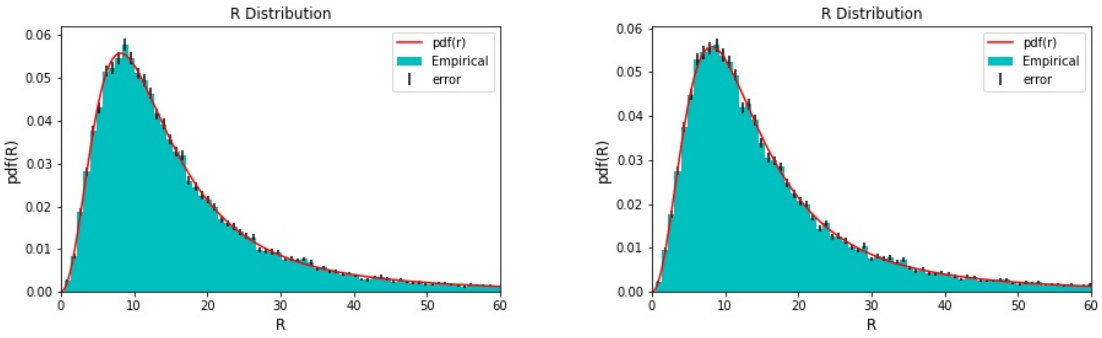
3.1 Position

3.1.1 Position Distributions and Density Function

We know that if the system is at equilibrium already from $t = 0$, the position distributions should be the same at any given time; to check this we compare at $t = 0$ and at another time (in our case, $t = 10 * t_{dyn} = 320$), the empirical distributions with the theorical ones, both qualitatively and quantitatively. For the qualitative test we plot the normalized histograms of the positions and the theorical pdf; for the quantitative test we used a chi square distribution to verify how well the theoretical prediction fitted the empirical one: in each plot we reported the *p-value* computed (for other details, see the relative Section on the notebook devoted to the analysis).

From the theory we know the probability density functions of the particles' positions in spherical coordinates; their pdf have been already reported above.

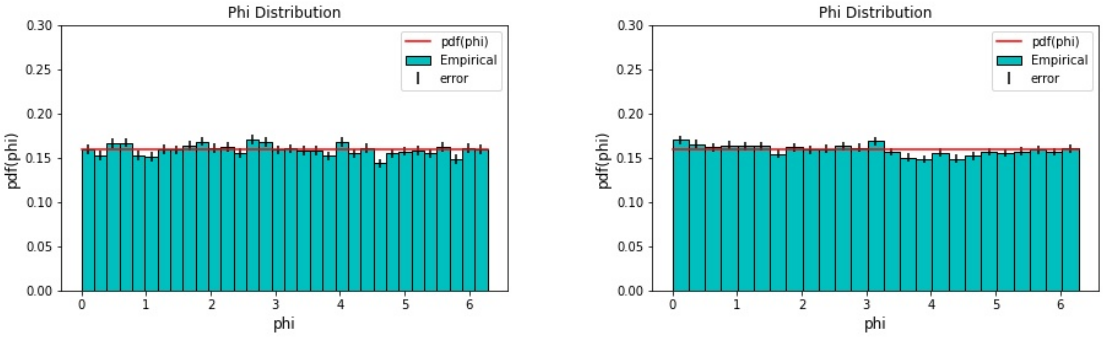
We now report the plots obtained in the simulation.



(a) t=0 (pval=0.49)

(b) t=320 (pval=0.35)

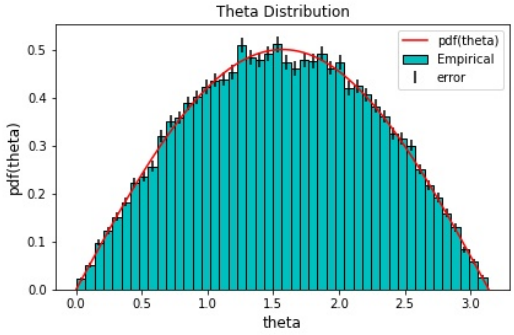
Figure 1: R distribution



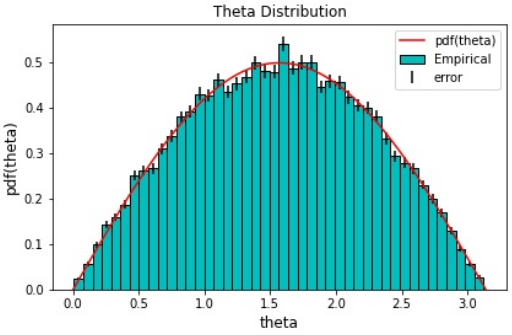
(a) t=0 (pval=0.09)

(b) t=320 (pval=0.05)

Figure 2: ϕ distribution



(a) t=0 (pval=0.61)



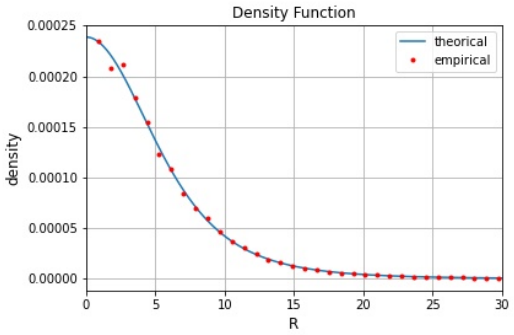
(b) t=320 (pval=0.095)

Figure 3: θ distribution

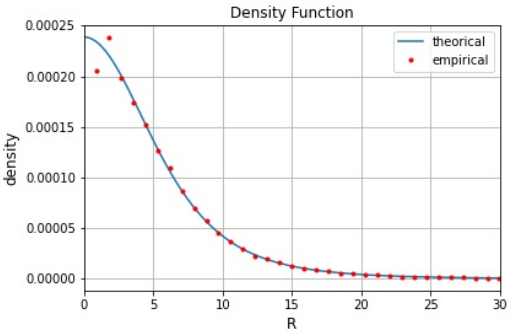
We can conclude, analyzing the plots above, that the position distributions do not change over time.
We then computed the density, at $t = 0$ and $t = 320$, of the generated Plummer's Sphere and compared it qualitatively to the expected one

$$\rho = \frac{3M_{tot}}{4\pi b^3}\left(\frac{1}{1+\frac{r^2}{b^2}}\right)^{\frac{5}{2}}, \tag{5}$$

To compute the empirical density we divided the sphere into shells, calculated the mass embedded in each shell and then divided it by the shell's volume. As can be seen in the plots below, the empirical density is, at least qualitatively, in accordance with the predicted one.



(a) t=0



(b) t=320

Figure 4: Density distribution

3.1.2 Lagrangian Radius

We analyze now the evolution over time of the so called *Lagrangian Radius*, which is the radius that contains a certain percentage of the mass (for more details about their computation, see the notebook).

In the following plot are presented the evolution for the Lagrangian radius that embed the 20%, 40%, 60%, 80% and 90% of the mass; since the system should be at equilibrium, the Lagrangian radius should be constant at any time and this is, indeed, what we have obtained.

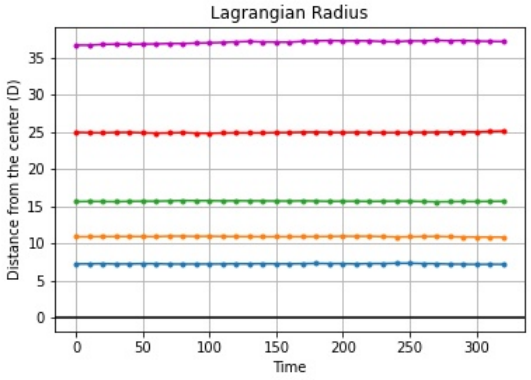


Figure 5: Lagrangian Radius

For a better visualization of the fact that the Lagrangian radius are constant, we focused our attention only on two of them, plotting for each one also a dotted horizontal line that corresponds to the mean value assumed by the Lagrangian radius between $0 < t < 320$.

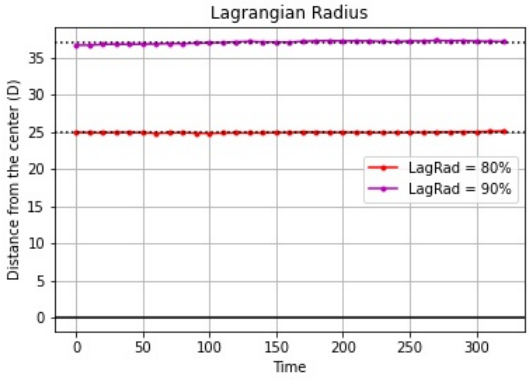


Figure 6: Lagrangian Radius and Mean

3.1.3 Center of Mass

We computed the evolution over time of center of mass' distance respect to the centre of the sphere; in principle, since we are dealing with a spherical symmetric and isolated system, its center of mass should be and remain in $(x, y, z) = (0, 0, 0)$ or, at least, it should be static and do not evolve in time.

If the center of mass is computed considering all the particles' system, we can see that it's not centered in (0,0,0), due to the fact that we are dealing with a finite number of particles and that the small percentage of them which is very far away from the center affects the CoM value; however, the center of mass is more or less static. In the plot of Figure 7 we reported also the system CoM where only particles within the lagrangian radius that embeds the 90% of the mass are considered: it's clearly visible how the rest 10% of particles affects the CoM position; since in both cases the CoM do not evolve, more or less, over time we decided to not change reference frame.

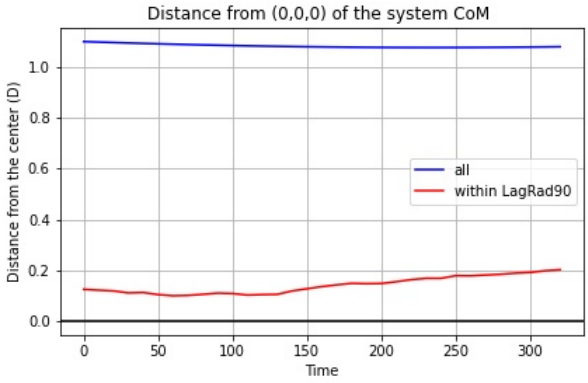


Figure 7: Center of Mass

3.2 Velocity

The initial velocities are distributed as the distribution function

$$f(\epsilon)d\epsilon = \left(\psi - \frac{v^2}{2}\right)^{\frac{7}{2}} v^2 dv, \tag{6}$$

that can be rewritten as

$$f(q)dq = \left(1 - q^2\right)^{\frac{7}{2}} q^2 dq, \tag{7}$$

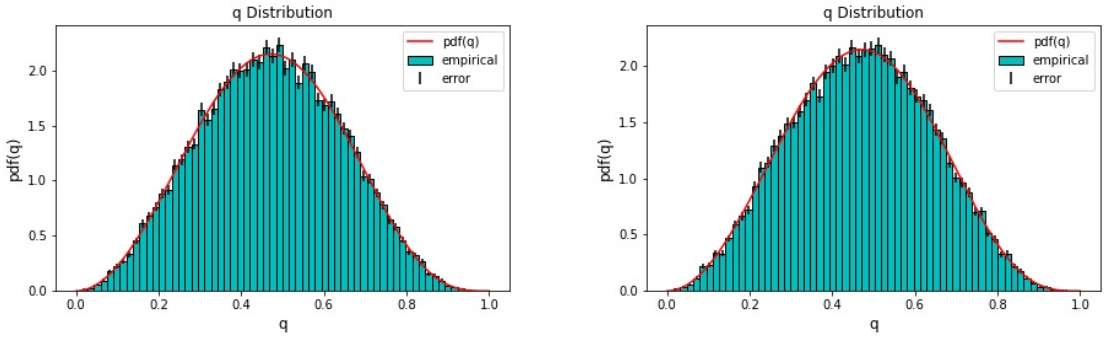
where we have introduce q, which is defined as

$$q = \frac{v}{v_{escape}} = \frac{v}{\sqrt{2\psi}}. \tag{8}$$

To assign a velocity to a particle at $t = 0$, we extract through the *Indirect Monte Carlo method* (since the integral of the q distribution can't be analytically inverted) a number distributed as q and then we find out the modulus of the velocity, $v_{mod} = v_{max} * q$ (where $v_{max} = \sqrt{2 * \psi}$). The velocities have a spherical distribution in the *velocity space* and so, knowing the modulus and generating θ and ϕ with distributions equal to the ones reported above, we can compute the cartesian components of the velocity.

In order to find if the velocity distribution is preserved at a given time different from 0, we do the above process in the inverse way: from the Cartesian components we compute the modulus of the velocity; from the particle's relative potential we find v_{max} and, at the end, we are able to compute q.

In the following plots we reported our results: qualitatively the q distribution is preserved and so it's the velocity one.



(a) t=0

(b) t=320

Figure 8: q distribution

3.3 Energy

We report here the energy plot for our simulation. If the system is at equilibrium, its position and its velocity distributions do not evolve in time and so do the potential and the kinetic energies. In Figure 9 (energies per unit mass) we observe indeed that the energies are constant over time.

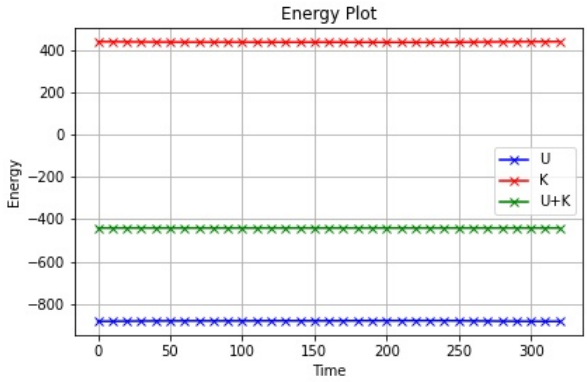


Figure 9: Energies

From the analysis made we can conclude that the Plummer's Sphere under analysis is at equilibrium.

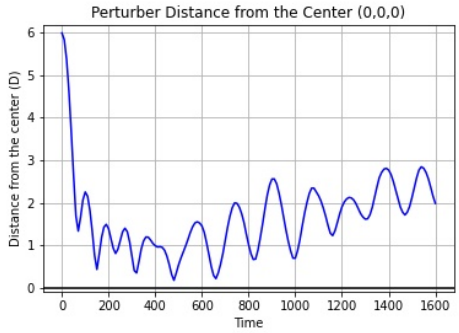
4 Dynamical Friction

After having verified the stability of the generated Plummer's sphere, we are now going to analyze the dynamical friction process over a peturber: our expectation is that, due to the gravitational interaction, the perturber is going to loose over time its kinetic energy (and so its angular momentum) in favour of the system particles that have interacted with it. Moreover, we run the simulation two times changing the perturber mass in order to verify how this affects the dynamical friction strength (we used $M_{pert1} = 3\% M_{sphere}$; $M_{pert2} = 1\% M_{sphere}$). The perturbers have been placed on elliptical orbits, with the apocenter which is 6 internal units distant from the center of the Plummer's sphere (in (0,0,0) at t=0) and the pericenter distant 2 i.u.; the initial velocities have been computed assuming the conservation of energy and angular momentum.

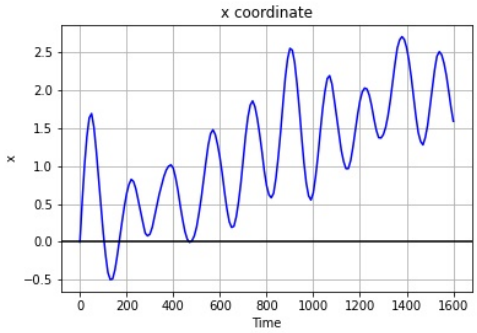
4.1 Change of reference frame

In this subsection we explain why is necessary to change reference frame and we suggest what is the best one for us; all the plots reported here concern the case in which $M_{pert} = 0.03$ because the effects are in that case more visible. The results deduced are then applied to both mass cases.

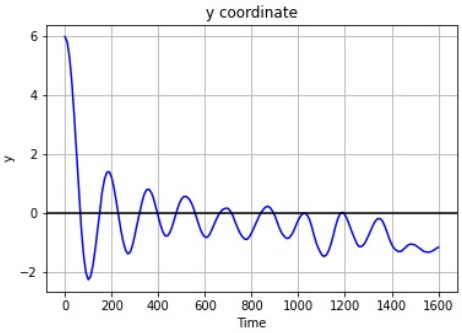
If the expectation stated above (decreasing of the perturber's kinetic energy) is correct, we should observe the decrease of the perturber distance respect to the system center of mass, (note that if the perturber's total energy was conserved, we would have observed an increase in the perturber's distance but, since we have interactions, it's the system energy to be conserved and not the particles or the perturber one). If the center of mass coincide with (0,0,0) or, at least, it's static, we will observe a decrease in its distance and then a stabilization on a closer orbit (obviously only if we are able to observe the dynamical friction process; we will see in the report, why we expect a stabilization on an orbit); however, plotting our results (see Figure **??**), we see that the perturber's distance respect to (0,0,0) decreases significantly at the beginning but then it restarts to increase and, moreover, a global motion in the x coordinate is evident meaning that a check on the center of mass evolution is needed.



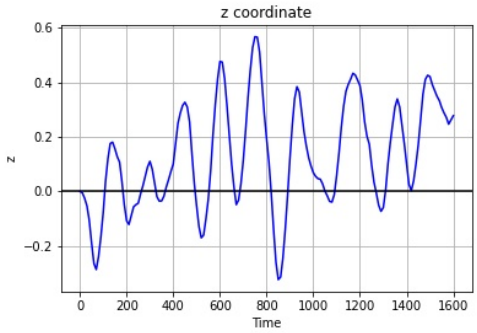
(a) Distance over time



(b) x over time



(c) y over time



(d) z over time

Figure 10: Perturber coordinates (no change of RF)

We computed then the center of mass for each time snapshot, in one case considering all the system bodies (all the particles and the perturber) and in the second case considering only the bodies within the lagrangian radius that embeds the 90% of the mass. In Figure 11 can be observed that in both cases the center of mass is in linear motion and so it's the whole system: this explains the perturbers' coordinates plots reported above and why it's necessary a change in the reference frame.

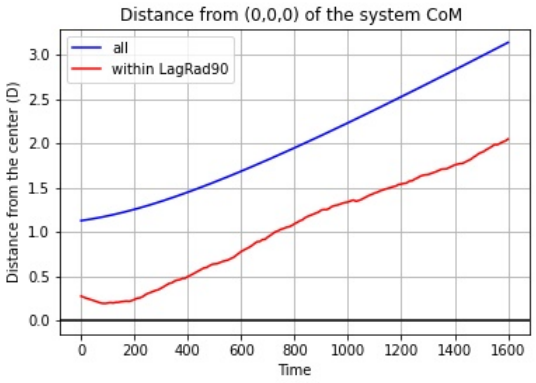
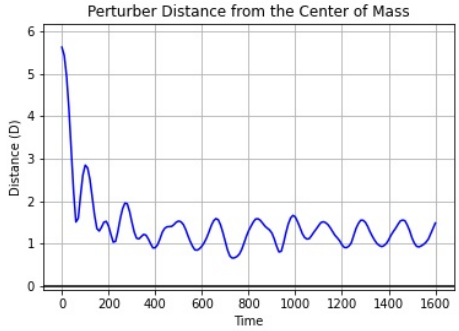


Figure 11: System Center of Mass

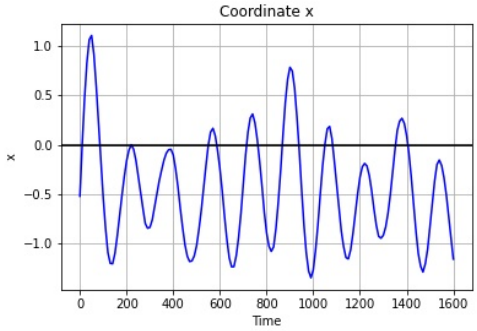
Once the necessity of a change in the reference frame has been established, it's now needed to understand which is the best one: to make the choice we consider the perturber's coordinates evolution in time and compare them in different cases, each one obtained translating the system in a reference frame where the center of mass has been computed considering only bodies within a certain lagrangian radius.

Translation in the reference frame where the center of mass has been computed considering:

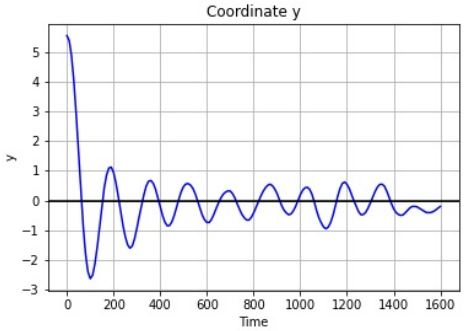
1. all the particles:



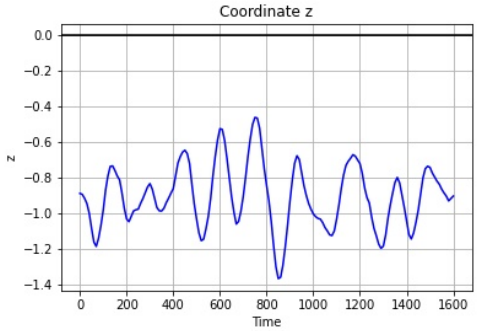
(a) Distance over time



(b) x over time



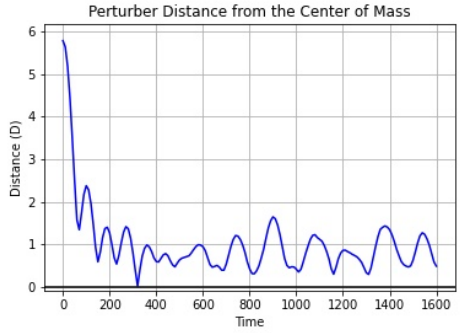
(c) y over time



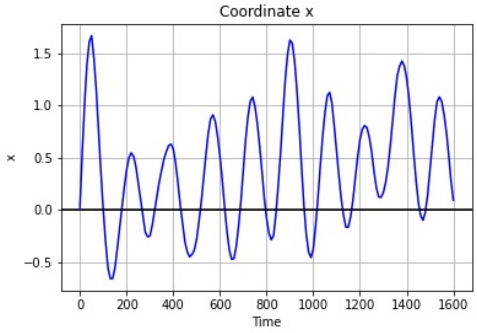
(d) z over time

Figure 12: Perturber coordinates (all particles considered)

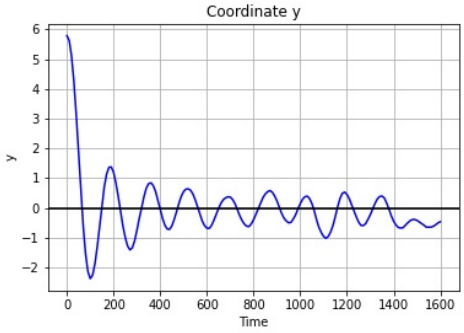
2. only particles within the lagrangian radius that embeds the 80% of the mass:



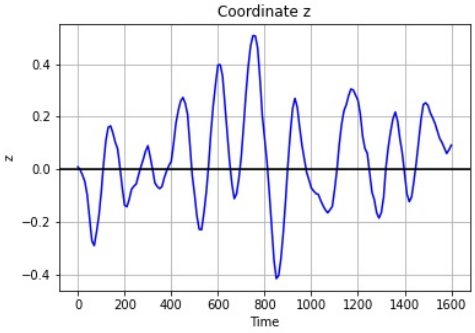
(a) Distance over time



(b) x over time



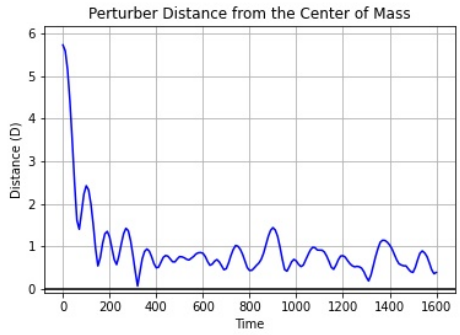
(c) y over time



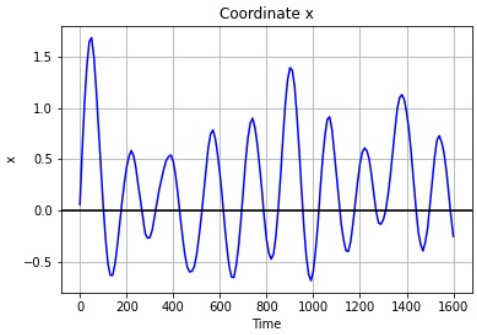
(d) z over time

Figure 13: Perturber coordinates (only particles within LagRad80 considered)

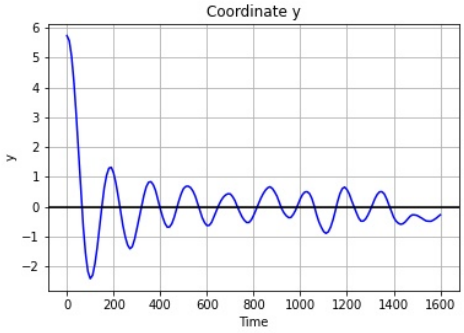
3. only particles within the lagrangian radius that embeds the 90% of the mass:



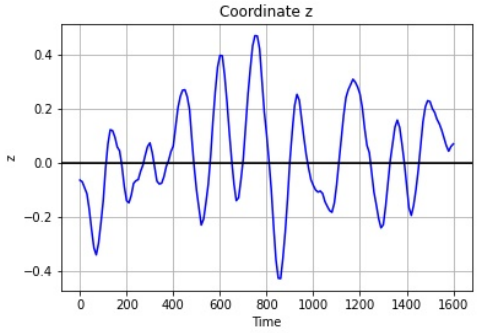
(a) Distance over time



(b) x over time



(c) y over time



(d) z over time

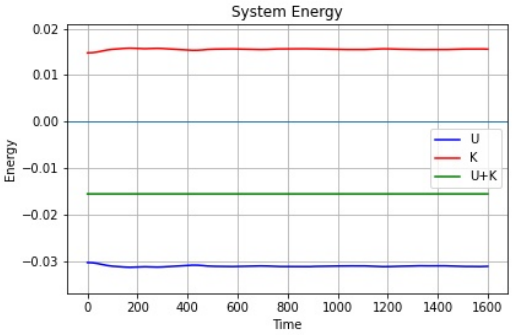
Figure 14: Perturber coordinates (only particles within LagRad90 considered)

From the plots above can be understood why is important to study not only the evolution of the perturber's distance from the CoM, but also the evolution of its coordinates; if we choose the correct reference frame, the perturber's orbit has to be on a plane that include also the CoM and so we should observe the pertruber's coordinates oscillate around (0,0,0) (N.B. we have translated the whole system s.t. the center of mass is now in (0,0,0)). In *case 1* the center of mass computation is affected by the far away particles and this is particularly evident analyzing the evolution of the perturber's z coordinate: it does not oscillate around 0, meaning that we are not in the smartest reference frame. The same can be stated for the *case 2*, where this effect is present but less visible (e.g. not all the x oscillations intersect x=0): since in this case we considered only the particles within LagRad80, we can say that we have neglected too much particles. We conclude that the best change in the reference frame is the translation in the center of mass computed considering only the particles within the LagRad90.
The results presented from now on (for both the perturbers) are referred to a system translated in the center of mass computed as in *case 3*; obviously, we translated not only the particles' positions but also their velocities (for the computation see the notebook devoted to the analysis).

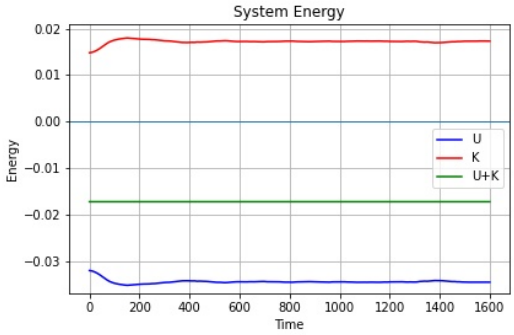
4.2 Conservation of Energy, Lagrangian Radius and Sphere contraction

We proceeded, as already explained, generating a Plummer's sphere and verifing that it's at equilibrium; then we re-evolve it from t=0 having include a perturber and we are now going to study how the system evolved.

We computed the kinetic and the potential energy of the whole system (particles+perturber) and we verified the conservation of the total energy. We report the plots obtained.



(a) $M_{pert} = 0.01$

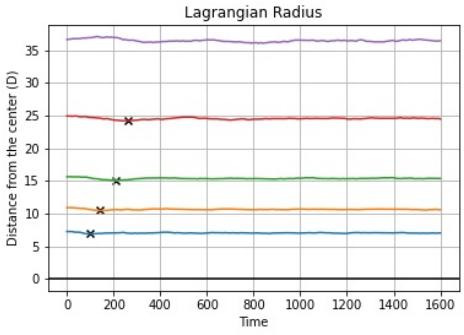


(b) $M_{pert} = 0.03$

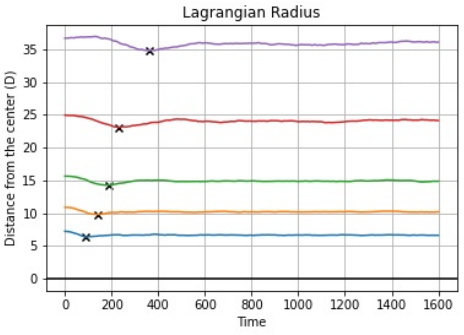
Figure 15: Energy plots

From the plots just reported we can state, at least qualitatively, that the total energy of the system is conserved; however, the kinetic and the potential energies are not constant in time already from t=0 but they evolve until they reach a constant value: we considered this effect as the particles' response to the perturber presence.

Including a perturber in the stable sphere, brings the system to reach a new equilibrium: the system is now more bound. To see and explain how this contraction is related to the new particle added to the system (i.e. the perturber) we report the lagrangian radius, computed considering only the original Plummer's sphere particles. We can see that over time the lagrangian radius contract, reach a minimum and then re-expand to a constant value which is less then the initial one. The bigger is the perturber's mass, the more the contraction is evindent.



(a) $M_{pert} = 0.01$



(b) $M_{pert} = 0.03$

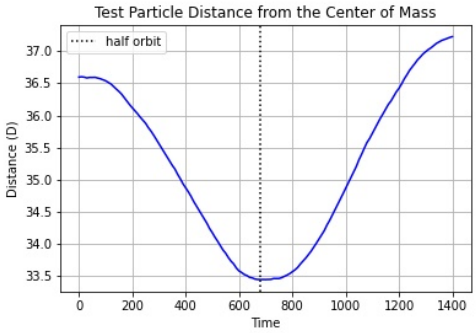
Figure 16: Lagrangian Radius (20,40,60,80,90)%

The perturbers' masses are only a small percentage of the Plummer's sphere's mass but have a measurable effect on the system. To have a corroboration of their influence we have performed the following test: we initialized a Plummer's sphere of mass $M = 1.03$ (N.B. $M = 1 + M_{pert}$) at equilibrium, where we included a test particle with an initial distance, respect to (0,0,0), equal to the initial value of the lagrangian radius that embeds the 90% of the mass in the case of $M_{pert} = 0.03$ (Figure 16b) and an initial velocity equal to the circular velocity that should have a particle of the same mass, placed at the same radius, but that belongs to a sphere of mass $M = 1$.

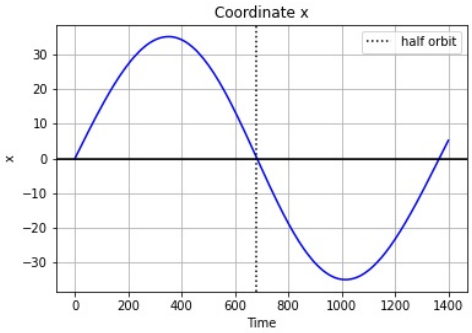
To do that the computational step followed are: initiliazation of a Plummer's sphere of mass $M = 1$, with 30'000 equal massive particles ($m = M/Npart$); computation of the circular velocity of a particle placed at the distance described above; initialization of a new system of mass $M = 1.03$ approximated with 30'900 particles (in this way each particle still have a mass equal to m); inclusion of the test particle of mass m with the coordinates (0,r,0) and the velocities (v,0,0).

If increasing the mass of the sphere by an amount equal to the 3% of its total mass is negligible, we'll shouldn't see any appreciable change in the test particle's dynamics (this because its velocity is equal to the circular velocity in a sphere of mass 1 at that radius and so if setting $M = 1$ is the

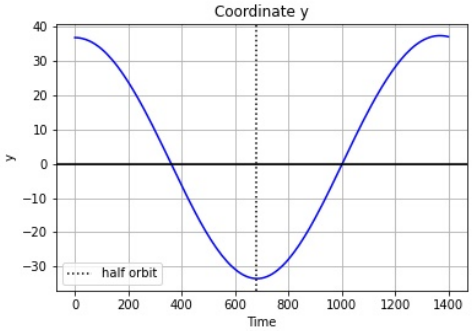
nearly the same of setting $M = 1.03$, the test particle should evolve on a circular orbit); otherwise, if that change is not negligible, the particle's dynamics is affected. In the plots of Figure **??** we observe an appreciable variation in the particle's orbit (e.g. its distance respect to the center of mass is not constant as in a circular motion); we corroborated our hypothesis that the inclusion of a perturber affects the whole system.



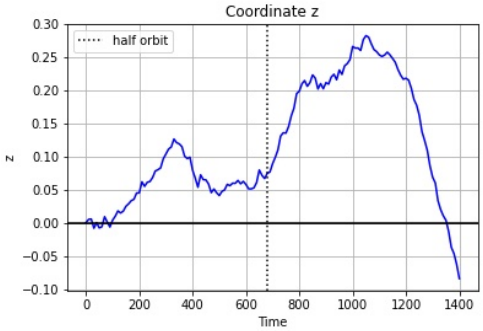
(a) Distance over time



(b) x over time



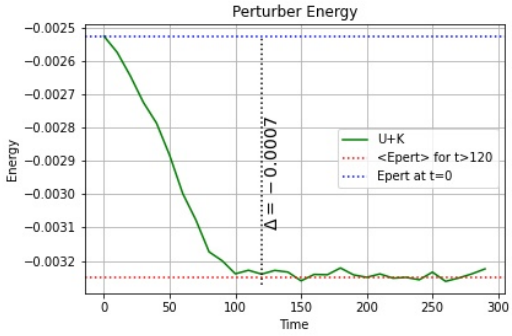
(c) y over time



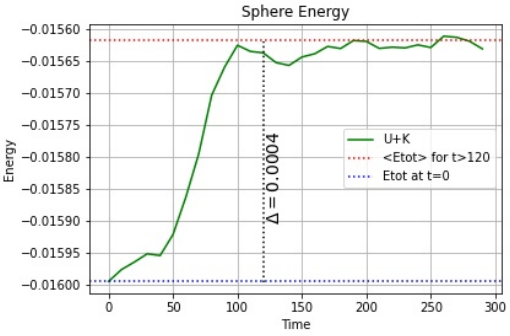
(d) z over time

Figure 17: Test Particle Coordinates

After having demonstrated that the system contraction is due to the presence of the perturber we report two plots, related to the case in which the perturber's mass is 0.03, where we graphically visualize the decrease of the perturber's energy and the increase of the particles' energy over time.



(a) Perturber



(b) Particles

Figure 18: Change in Energy

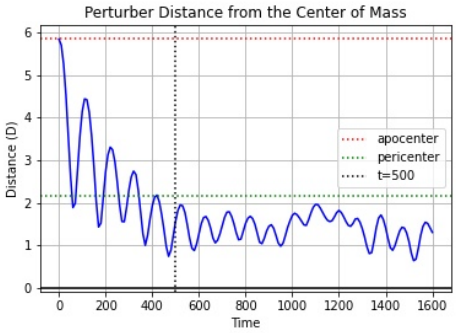
The perturber, due to the interaction with the system particles (dynamical friction), looses kinetic energy and so it becomes more bound (indeed its total energy decreases); the particles, instead, acquire kinetic energy and this results in an increasing of their total energy. The conservation of energy implies that the energy lost by one of the subsystems is gained by the other

one but, in the plots above, we see that the particles acquire only half of the energy lost by the perturber; however this is due to the fact that for the computation of the the potential energy of the particles, at each snapshot we sum the potential energies per each particle (computed by the Barners' Tree Code, excluding the perturber) and then we divide it by two to avoid double counting: the problem is that the potential energy of each particle, as computed by the tree code, includes its interaction with the perturber and so we can't disentangled this value unless we computed it in a different way. In essence, the plot b of Figure 18, it's the particles' total energy plus half of the potential energy due to their interaction with the perturber.
We can conclude that the total energy of the system is conserved and that the dynamical friction process subtracts kinetic energy from the perturber, energy then gained by the system particles.

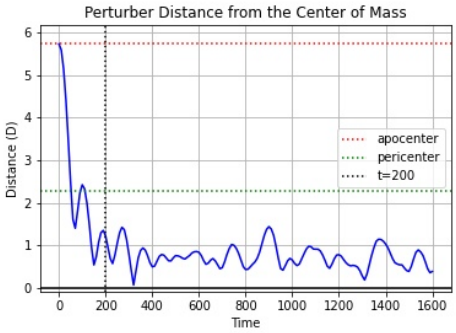
5 Perturber's Positions and Velocities over time

We now study the evolution over time of the perturbers' velocities and distances respect to the center of mass; at $t = 0$ the perturbers were initialized on elliptical orbits (as already explained above) and they would have continued on them if the dynamical friction effect was not present or sufficiently strong but this is not the case as can be seen in the plots of Figure 19. In the perturbers' distance plots of Figure 19 it's clearly visible the dynamical friction effect that causes the lost of the perturbers' kinetic energy, bringing them down toward the center of mass (if the effect under study was not present, the perturbers' distance would have been embedded between the apocenter and the pericenter distances, highlighted in the plots by the horizontal dotted lines).

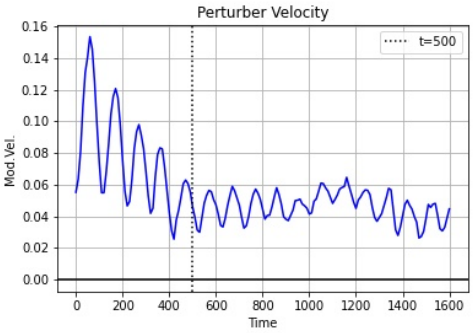
We know that the dynamical friction strength is proportional to the perturber's mass and we verified this using two different masses: in Figure 19 the distance decreasing when $M_{pert} = 0.01$ is slower respect to the case of $M_{pert} = 0.03$; observing the plots, we see that after a certain time (highlighted with a vertical dotted line and found qualitatively) the perturber's distance do not evolve any more significantly, meaning that the dynamical friction isn't still effective. We state that the dynamical friction in our simulation is not any more effective because at those distances from the CoM (the ones reached after $t = 500$ for $M_{pert} = 0.01$ and $t = 200$ for $M_{pert} = 0.03$) the perturber will interact with a very small percentage of the particles used to approximate the sphere; we verified this overlapping, in Figure 20, the Lagrangian radius within which the perturber is doing its orbit on the distance's plots (we plot the lagrangian radius' mean value assumed after $t = 500$ and $t = 200$ respectively). We found that the dynamical friction becomes ineffective because the pertuber will interact only with a number of particles that is between 9 and 270 for the $M_{pert} = 0.01$ case and between 3 and 75 for the $M_{pert} = 0.03$ case.



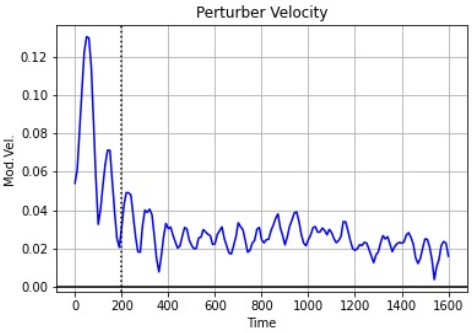
(a) $M_{pert} = 0.01$



(b) $M_{pert} = 0.03$

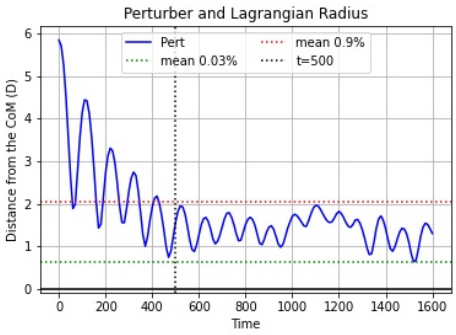


(c) $M_{pert} = 0.01$

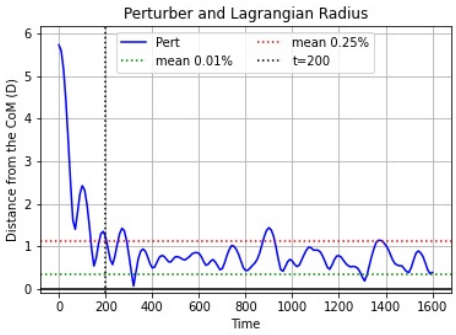


(d) $M_{pert} = 0.03$

Figure 19: Perturber's velocity and distance from the CoM



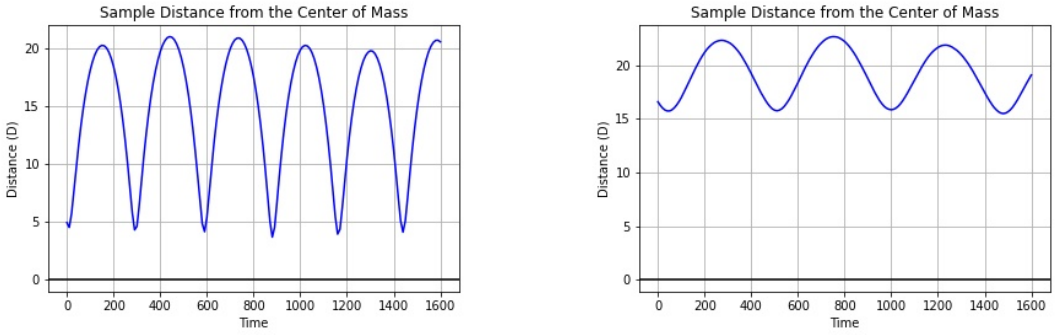
(a) $M_{pert} = 0.01$



(b) $M_{pert} = 0.03$

Figure 20: Perturber and Lagrangian Radius

The dynamical friction effect is effective on the bodies whose mass is much bigger respect to the particles' mass; in Figure 21 we report the evolution in time of the distances of two sampled particles: as can be seen, their orbits do not change in time.

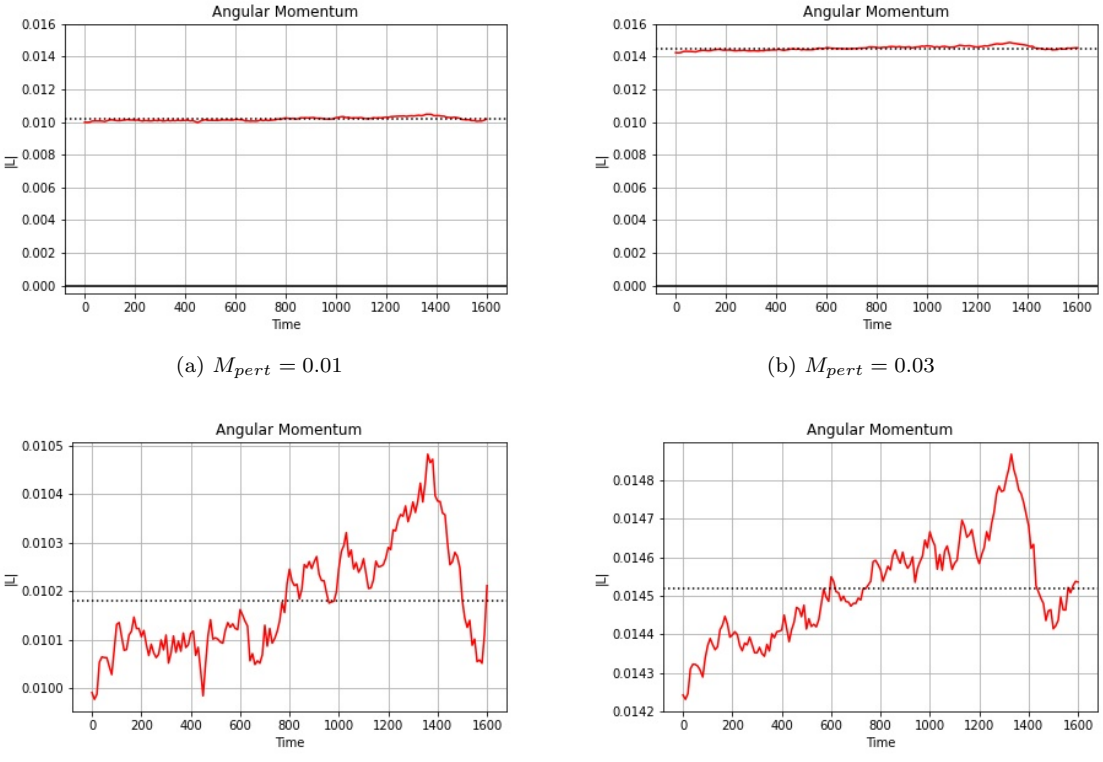


(a) $M_{pert} = 0.01$ (b) $M_{pert} = 0.03$

Figure 21: Sample distance from the CoM

6 Angular Momentum

The total angular momentum of the system is pretty well conserved over time, as can be seen in Figure 22.



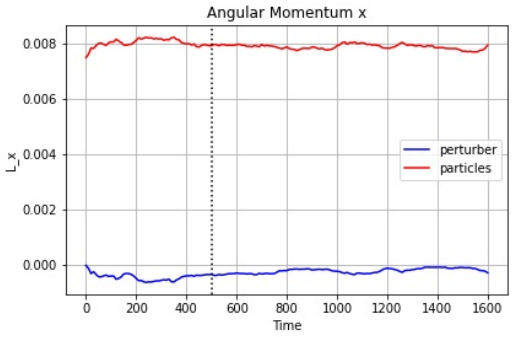
(a) $M_{pert} = 0.01$ (b) $M_{pert} = 0.03$

(c) $M_{pert} = 0.01$ (d) $M_{pert} = 0.03$

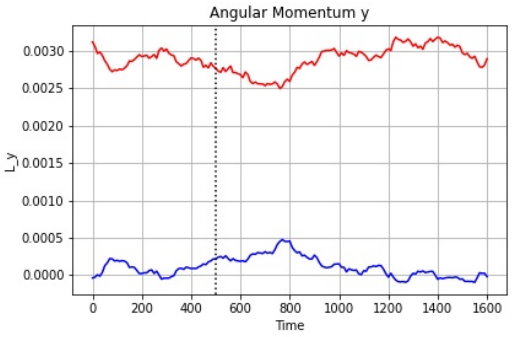
Figure 22: System Angular Momentum over time

The total angular momentum of the system is conserved; moreover the angular momentum lost (or gained) by one subsystem is gained (or lost) by the other one as it is clearly visible in the plots of Figure 23 and Figure 24, where we reported L_x, L_y and L_z of the particles that made up the Plummer's sphere and of the perturber. The perturber's orbit is mostly on the (x, y) plane and so its angular momentum is more or less all on the z direction (L_x and L_y of the perturber are nearly equal to zero); for the particles' L components we simply sum, at each snapshot, the components of each particle (in principle, since the system is isotropic, we should have obtained

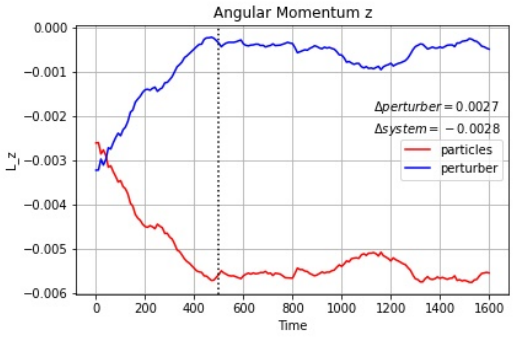
for the particles $L_x = 0$, $L_y = 0$ and $L_z = 0$, but this does not happen because we approximated the sphere with a finite number of particles).



(a) L_x

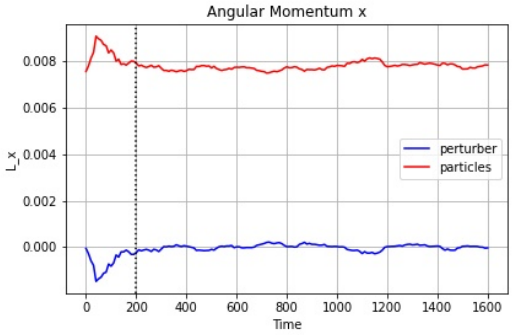


(b) L_y

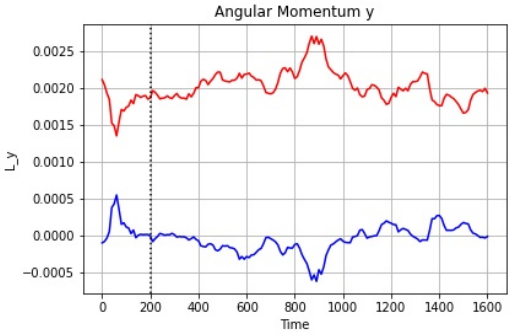


(c) L_z

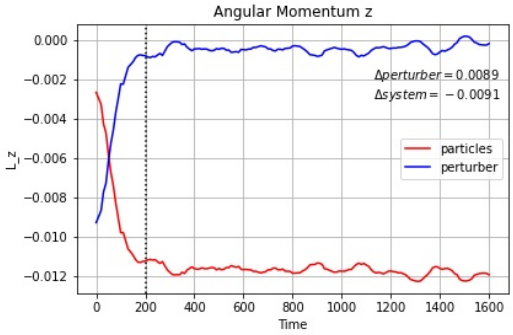
Figure 23: Subsystem L components over time, $M_{pert} = 0.01$



(a) L_x



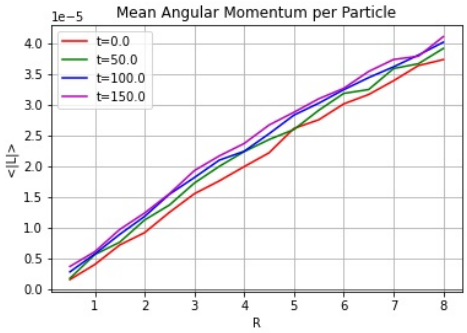
(b) L_y



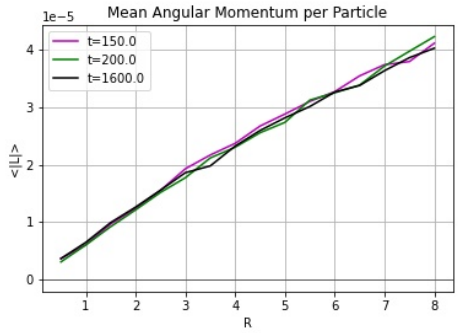
(c) L_z

Figure 24: Subsystem L components over time, $M_{pert} = 0.03$

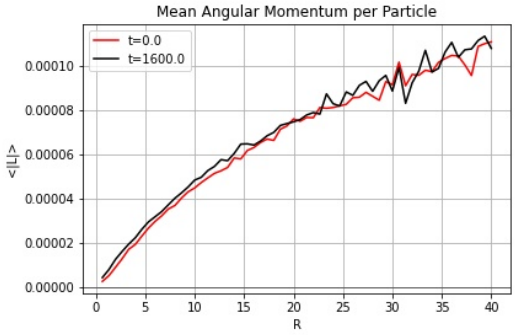
For the case of $M_{pert} = 0.03$ we study the particles' angular momentum in function of their distance from the CoM: we considered a radius interval between r_{min} and r_{max}, we divided it into equal length bins and then, for each bin, computed the absolute value of the angular momentum (i.e. $|l| = \sqrt{l_x^2 + l_y^2 + l_z^2}$) for every particle which belongs to that bin and, finally, mediated them. The process just described has been iterated for different snapshots and the results are reported in Figure 25.



(a)



(b)



(c)

Figure 25: Mean $|L|$ per particle vs. R, $M_{pert} = 0.03$

In the plot (c) of Figure 25 we can see the mean angular momentum per particle in function of R at $t = 0$ and $t = 1600$; we observe a significantly change in $< |L| >$ between $r = 0$ and $r = 20$ and a unvaried mean for bigger distances: this can be explained by the fact that the perturber is at distances less then 6 and so its angular momentum is mostly gained by nearest particles; for this reason we focused our attention on the particles between 0 and 8. The dynamical friction, in the case of $M_{pert} = 0.03$ is effective until $t = 200$, meaning that after 200 i.u.time we will not see anymore an appreciably lost in the perturber's angular momentum; this is clear analyzing the plots (a) and (b) of Figure 25: in (a) the particles continues to gain angular momentum up to $t = 150/200$ but then, as can be seen in (b), their angular momentum do not change any more and stabilizing itself at the value for $t = 150/200$.

7 Conclusion

We wanted to study numerically the dynamical friction effect on a perturber in motion within a distribution of mass; to do that, we initialized a Plummer's sphere and verified its stability; then we included a perturber and re-evolved the system.

We observed that the perturber, due to the gravitational interaction with the system's particles, decreased its velocity (and so its angular momentum and kinetic energy) and its distance from the center of mass; moreover, we verified the correlation between the dynamical friction strength (or efficiency) and the perturber's mass running the code for two different masses.

We observed also that the perturber's distance from the CoM decreases significantly at the beginning but then this effect is no more present, probably due to the small percentage of particles at those radius: to bring the perturber closer to the center of mass one should run the simulation increasing the number of particles.

Concluding, in this report we have initialized a system which is stable; we've verified the conservation of the system's energy and of the system's angular momentum and we have studied the dynamical friction effect on a perturber.

In [1]:
```
### Plummer Sphere Equilibrium (update) - Analysis code ###
```

In [2]:
```
# About the code #

# The code is divided into five main Sections:
# 1 - Setting of useful parameters
# 2 - Definition and filling of the time array
# 3 - Energy section
# 4 - Position section
# 5 - Stability Section
#
# If you want to check the initial conditions (even if it's recommended to do that in the Initial
# Condition notebook) you need only to run the Stability Section, choosing 'snapshot_to_check=0'.
# In all the other cases, it is mandatory to run Sections 1 and 2 while the execution's order of
# the other sections is at the user's discretion (for example, you can run section 1 and 2 and then
# run only the Position Section).
```

In [3]:
```
##########################################################################################
```

In [4]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline
```

In [5]:
```
## Section 1 - Setting of useful parameters (Nparticles,TotMass,ScaleRadiusSphere,Nsnapshots) ##
```

In [6]:
```
## Number of particles, total mass and scale radius of the sphere
Nparticles,M,r_scale = np.loadtxt('system_properties.txt')
Nparticles = int(Nparticles)
m = M/Nparticles # mass of a single particle

Ndimensions = 3

Nlines_in_snapshot = 3+4*Nparticles # Nparticles(1row)+Ndimensions(1)+time(1)+mass_particles(N)+
                                    # +positions(N)+velocities(N)+potential_energy(N)
```

In [7]:
```
## Reading the name of the generated files (output of 'treecode') and order them chronologically
# (each one is the system description at a given snapshot)

import glob, os
import re # regular expression

path_data = '/home/efrem/Astronomia_extragalattica/2-NBodySimulations/6-Dinamical_Friction_(alg:treecodeBarn
data_files = []
os.chdir(path_data)
for file in glob.glob("*.data"): # selection of the directory's files that end with '.data'
    data_files.append(file)

# 'data_files' containes the name of the output 'treecode' files; each file contains the state
# of the system at a given time; since the files in 'data_files' are not in chronologival order,
# here a code to do that

def chronological_order(list):
    search_num = re.compile('(^\d+)')
    number_to_name = []
    for i in list:
        l = search_num.search(i)
        number = int(l.group(1))
        number_to_name.append(number)
    number_to_name = sorted(number_to_name)
    # add .data
    for i in range(len(number_to_name)):
        number_to_name[i]=('{}.data'.format(number_to_name[i]))
    return (number_to_name)

data_files = chronological_order(data_files)
```

In [8]:
```
# Number of Snapshots
Nsnapshots = len(data_files)
Nsnapshots = int(Nsnapshots)
```

In [9]:
```
## Selection of the snapshots to analyze
# Here you can choose the number of snapshots that you want to analyze. The code is very performant,
# so we suggest to analyze all of them ('step=1'). If, however, you want to select some of them you
# can modify the 'step' values or you can manually select the ones needed (commented part)

sn = 0
step = 1 # if 1, analyze all the snapshots
selected_snapshot = []
for i in range(int(Nsnapshots/step)):
    selected_snapshot.append(step*i)

# Manual selection
#selected_snapshot = [0,Nsnapshots-1] # this is an example, where we analyze the first and the last snaphots

print('The total number of snapshots is:,',Nsnapshots)
print('\nThe selected snapshots are: ',selected_snapshot)
print('\nYou want to analyze {} snapshots'.format(len(selected_snapshot)))
```

```
The total number of snapshots is:, 33

The selected snapshots are:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

You want to analyze 33 snapshots
```

In [10]:
```
##########################################################################################
```

In [11]:
```
## Section 2 - Definition and filling of the time array ##
```

In [12]:
```
t = np.empty([len(selected_snapshot)])
for i_snapshot in range(Nsnapshots):
    file_name = data_files[i_snapshot]
    initial_line_time = 2
    t[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_time,max_rows=1)
```

In [13]:
```
##########################################################################################
```

In [14]:
```
## Section 3 - Energy section
```

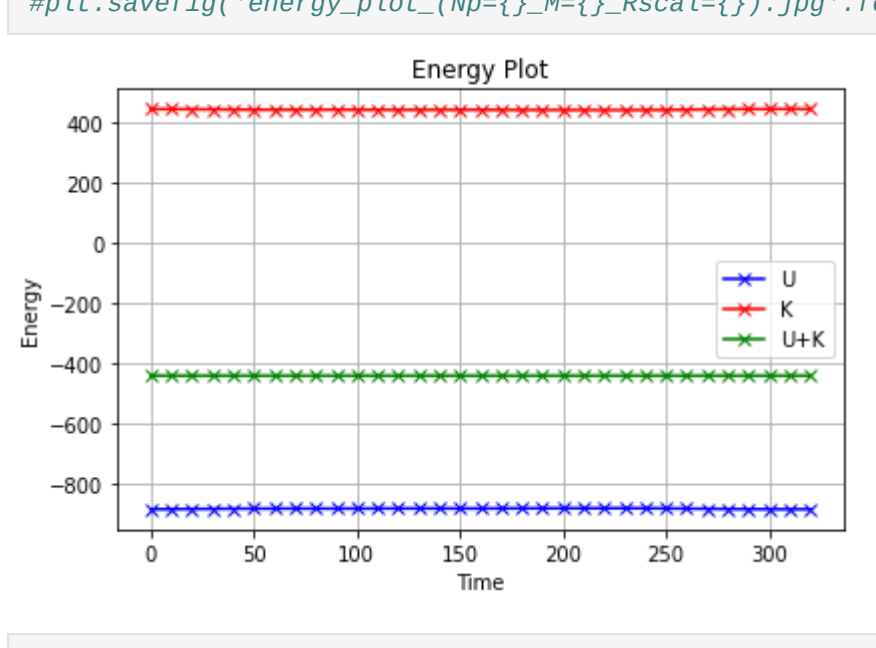
In [15]:
```
# In this Section are computed, for selected snapshots, the total kinetic and the total potential energies p
# unit mass.
# The kinetic energy, the potential energy and the total energy (sum of the previous ones) are then
# plotted respect to time.
# The total energy, in principle, should be preserved at every time and should be equal to the total
# energy at t=0; for this reason is presented the graph of '(Etot(t)-Etot(t=0))/Etot(t=0) vs. time'
# that gives an idea of the conservation of energy deviation (the non perfect conservation of energy
# is due to computational approximations)
```

In [16]:
```
velocities = np.empty([len(selected_snapshot),Nparticles,Ndimensions])

initial_line_velocity = 3+2*Nparticles
for i_snapshot in range(len(selected_snapshot)):
    file_name = data_files[selected_snapshot[i_snapshot]]
    velocities[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_velocity,max_rows=Nparticles)

vx = np.empty([len(selected_snapshot),Nparticles])
vy = np.empty([len(selected_snapshot),Nparticles])
vz = np.empty([len(selected_snapshot),Nparticles])
modV = np.empty([len(selected_snapshot),Nparticles])

for i_snapshot in range(len(selected_snapshot)):
    for i_particle in range(Nparticles):
        vx[i_snapshot,i_particle] = velocities[i_snapshot][i_particle][0]
        vy[i_snapshot,i_particle] = velocities[i_snapshot][i_particle][1]
        vz[i_snapshot,i_particle] = velocities[i_snapshot][i_particle][2]
modV = pow(vx**2+vy**2+vz**2,0.5)

del velocities # free up memory
```

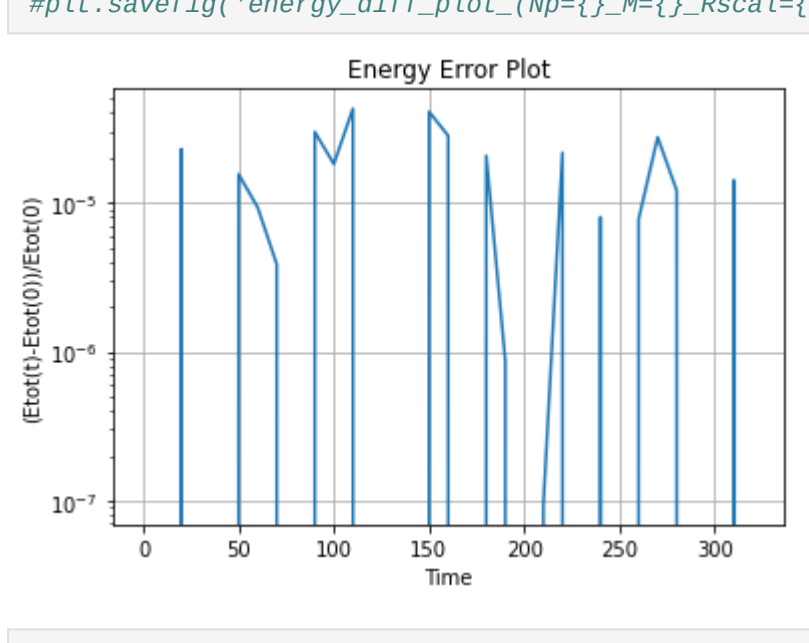
In [17]:
```
potential_energy = np.empty([len(selected_snapshot),Nparticles])

initial_line_potential = 3+3*Nparticles
for i_snapshot in range(len(selected_snapshot)):
    file_name = data_files[selected_snapshot[i_snapshot]]
    potential_energy[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_potential,
                                                 max_rows=Nparticles)
```

In [18]:
```
# Potential and Kinet energy per unit mass
Utot = np.sum(potential_energy,1)*0.5 # to avoid double counting
Ktot = np.sum(vx**2+vy**2+vz**2,1)*0.5
```

In [19]:
```
## Plot of the ptential energy (U), kinetic energy(K), total energy(U+K)
plt.plot(t,Utot,c='b',label='U',marker='x')
plt.plot(t,Ktot,c='r',label='K',marker='x')
plt.plot(t,Ktot+Utot,c='g',label='U+K',marker='x')

plt.title('Energy Plot')
plt.xlabel('Time')
plt.ylabel('Energy')

plt.grid()
plt.legend(loc='best')
plt.tight_layout()
#plt.savefig('energy_plot_(Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale))
```



In [20]:
```
## Plot of '(TotalEnergy(t)-TotalEnergy(t=0))/TotalEnergy(t=0)  vs. Time'
diff = ((Utot+Ktot)-(Utot[0]+Ktot[0]))/(Utot[0]+Ktot[0])
plt.plot(t[selected_snapshot],diff)

plt.title('Energy Error Plot')
plt.xlabel('Time')
plt.ylabel('(Etot(t)-Etot(0))/Etot(0)')

plt.yscale('log')
plt.grid()
#plt.savefig('energy_diff_plot_(Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale))
```



In [21]:
```
##########################################################################################
```

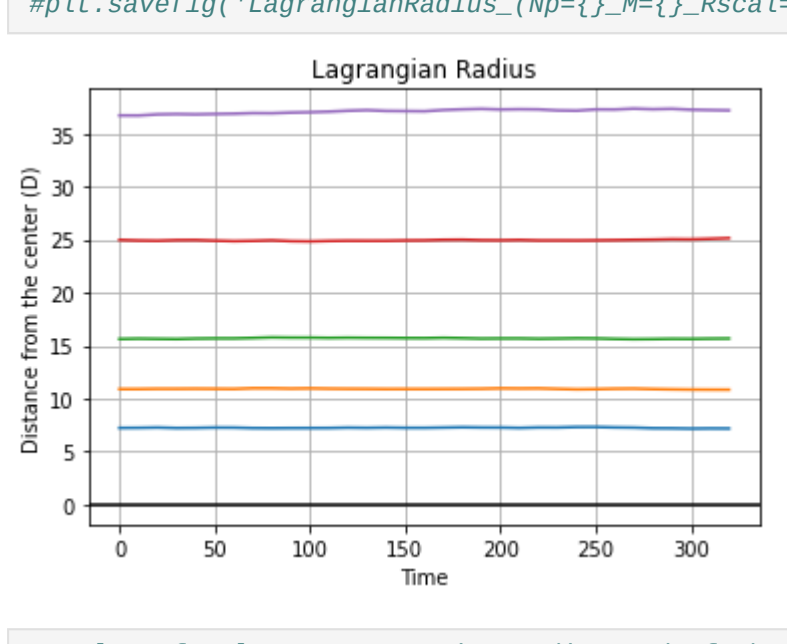
In [22]:
```
## Section 4 - Position section ##
```

In [23]:
```
# Subsection 1 - Lagrangian Radius #
```

In [24]:
```
# Positions
positions = np.empty([len(selected_snapshot),Nparticles,Ndimensions])

for i_snapshot in range(len(selected_snapshot)):
    file_name = data_files[selected_snapshot[i_snapshot]]
    initial_line_position = 3+Nparticles
    positions[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_position,max_rows=Nparticles)

x = np.empty([len(selected_snapshot),Nparticles])
y = np.empty([len(selected_snapshot),Nparticles])
z = np.empty([len(selected_snapshot),Nparticles])
dist = np.empty([len(selected_snapshot),Nparticles]) # particles' distance from (0,0,0) at each time

for i_snapshot in range(len(selected_snapshot)):
    for i_particle in range(Nparticles):
        x[i_snapshot,i_particle] = positions[i_snapshot][i_particle][0]
        y[i_snapshot,i_particle] = positions[i_snapshot][i_particle][1]
        z[i_snapshot,i_particle] = positions[i_snapshot][i_particle][2]
dist = pow(x**2+y**2+z**2,0.5)

del positions # free up memory
```

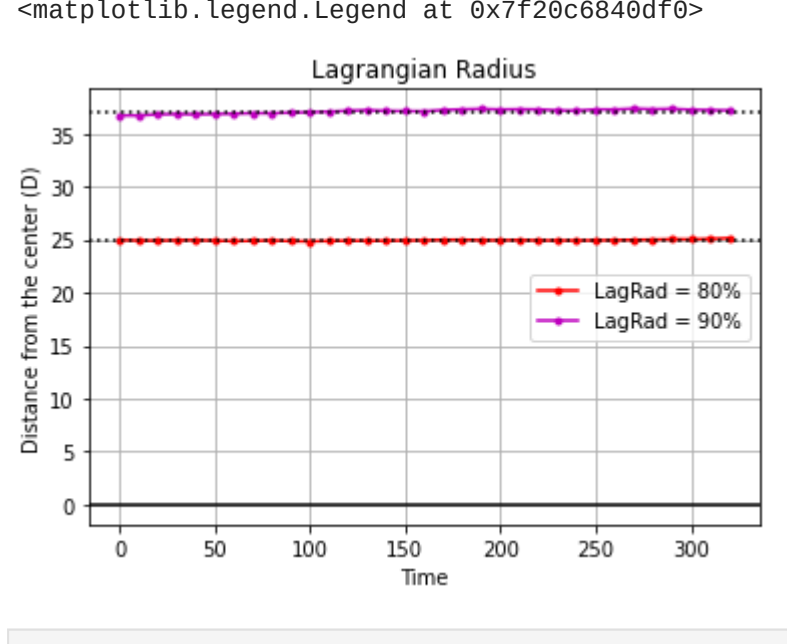
In [25]:
```
## Lagrangian Radius of the sphere (we do not consider the perturber)

p = [20,40,60,80,90] # lagrangian radius that contain that percentage of mass

# definition of the lagrangian radius
for perc in p:
     globals()['LagRad'+str(perc)] = np.empty([len(selected_snapshot)])

sorted_distance = np.sort(dist)

# filling of the lagrangian radius
for i_snapshot in range(len(selected_snapshot)):
    for perc in p:
        (globals()['LagRad'+str(perc)])[i_snapshot] = (sorted_distance
                                                       [i_snapshot][int(perc/100*Nparticles)-1])
```

In [26]:
```
## Plot of the Lagrangian radius
for perc in p:
    plt.plot(t,globals()['LagRad'+str(perc)],label='LagRad = {}%'.format(perc))

plt.axhline(0,0,1,c='k',ls='-')

plt.title('Lagrangian Radius')
plt.xlabel('Time')
plt.ylabel('Distance from the center (D)')

plt.grid()
#plt.legend(loc=4)
#plt.savefig('LagrangianRadius_(Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale))
```



In [27]:
```
## Plot of only two Lagrangian radius and of the corrispondent mean (only two for a better visualization)
plt.plot(t,LagRad80,label='LagRad = 80%',marker='.',c='r')
plt.plot(t,LagRad90,label='LagRad = 90%',marker='.',c='m')

mean80 = sum(LagRad80)/len(LagRad80)
mean90 = sum(LagRad90)/len(LagRad90)
plt.axhline(mean80,0,1,c='k',ls=':')
plt.axhline(mean90,0,1,c='k',ls=':')
plt.axhline(0,0,1,c='k',ls='-')

plt.title('Lagrangian Radius')
plt.xlabel('Time')
plt.ylabel('Distance from the center (D)')

plt.grid()
plt.legend(loc='best')
#plt.savefig('LagrangianRadiusMean_(Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale))
```

Out[27]:
```
<matplotlib.legend.Legend at 0x7f20c6840df0>
```



In [28]:
```
##########################################################################################
```

In [29]:
```
## Center of mass
```

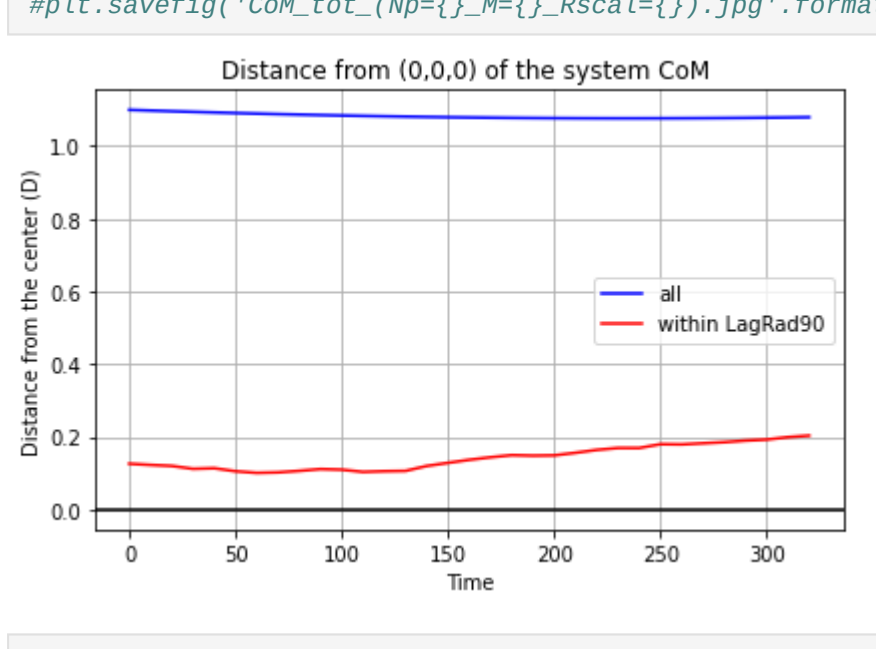
In [30]:
```
# N = len(selected_snapshot), R = distance vector (only particles with r<=R are considered in the
# computation) (ex. R = LagRad80). If you want to consider all particles:
# R = np.ones([len(selected_snapshot)])*np.max((np.sort(dist)[-1]))

def CoM(N,Nbodies,R,x,y,z,dist,mass_bodies):
    M_tot = np.sum(mass_bodies)
    CoM_x = np.empty([N])
    CoM_y = np.empty([N])
    CoM_z = np.empty([N])
    for i_snapshot in range(N):
        c_x = 0
        c_y = 0
        c_z = 0
        for i_particle in range(Nbodies):
            if dist[i_snapshot][i_particle] <= (R[i_snapshot]):
                c_x = c_x + x[i_snapshot,i_particle]*mass_bodies[i_particle]
                c_y = c_y + y[i_snapshot,i_particle]*mass_bodies[i_particle]
                c_z = c_z + z[i_snapshot,i_particle]*mass_bodies[i_particle]
        CoM_x[i_snapshot] = c_x
        CoM_y[i_snapshot] = c_y
        CoM_z[i_snapshot] = c_z
    CoM_x = CoM_x/M_tot
    CoM_y = CoM_y/M_tot
    CoM_z = CoM_z/M_tot
    dist_CoM = pow(CoM_x**2+CoM_y**2+CoM_z**2,0.5)
    return CoM_x,CoM_y,CoM_z,dist_CoM
```

In [31]:
```
mass_bodies = np.ones(Nparticles)*m
```

In [32]:
```
# CoM where all the particles are considered
R = np.ones([len(selected_snapshot)])*np.max((np.sort(dist)[-1]))
CoM_x,CoM_y,CoM_z,dist_CoM = CoM(len(selected_snapshot),Nparticles,R,x,y,z,dist,mass_bodies)
del R
```

In [33]:
```
## CoM of the sphere (only particles within LagRad90)
R = LagRad90
CoM_x2,CoM_y2,CoM_z2,dist_CoM2 = CoM(len(selected_snapshot),Nparticles,R,x,y,z,dist,mass_bodies)
del R
```

In [34]:
```
plt.plot(t,dist_CoM,label='all',c='b')
plt.plot(t,dist_CoM2,label='within LagRad90',c='r')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('Distance from (0,0,0) of the system CoM ')
plt.xlabel('Time')
plt.ylabel('Distance from the center (D)')

plt.grid()
plt.legend()
plt.tight_layout()
#plt.savefig('CoM_tot_(Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale))
```



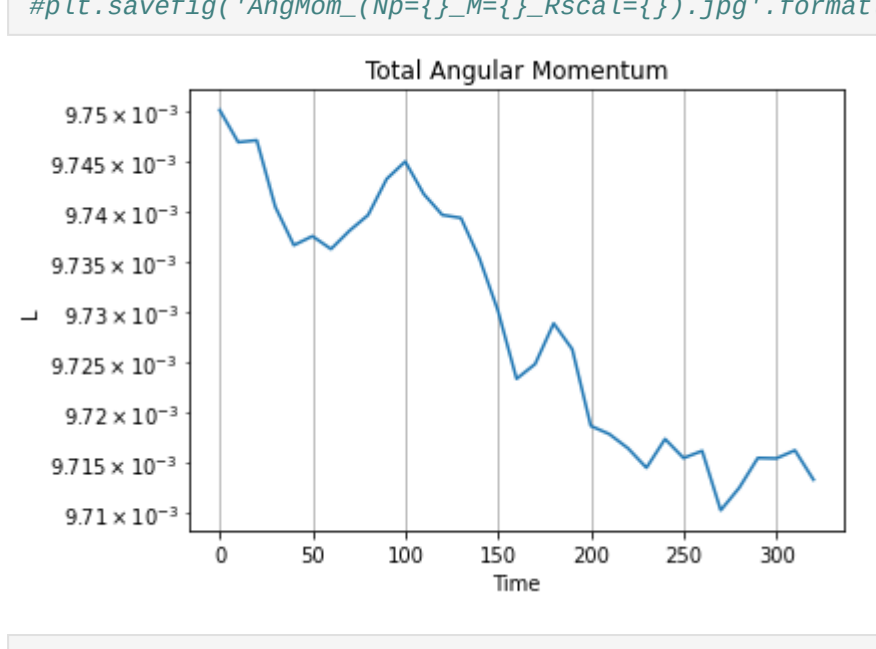
In [35]:
```
##########################################################################################
```

In [36]:
```
## Angular momentum of the system
```

In [37]:
```
def angularMomentum(m,x,y,z,vx,vy,vz):
    Lx = (y*vz-z*vy)*m
    Ly = (z*vx-x*vz)*m
    Lz = (x*vy-y*vx)*m
    return Lx,Ly,Lz
```

In [38]:
```
Lx,Ly,Lz = angularMomentum(mass_bodies,x,y,z,vx,vy,vz)
Lmod_tot = pow(np.sum(Lx,1)**2+np.sum(Ly,1)**2+np.sum(Lz,1)**2,0.5)
```

In [39]:
```
plt.plot(t,Lmod_tot)

plt.title('Total Angular Momentum')
plt.xlabel('Time')
plt.ylabel('L')

plt.yscale('log')
plt.grid()
plt.tight_layout()
#plt.savefig('AngMom_(Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale))
```



In [40]:
```
##########################################################################################
```

In [41]:
```
 ## Section 5 - Stability Section ##
```

In [42]:
```
# This Section can be used either to check if the initial conditions generated are correct
# (the distribution of R, theta and phi, the density function and the q distribution (through
# which you check the velocity one)) or to check at a selected snapshot if the positions and the
# velocity distributions have changed over time or if they preserved themselves.
# To check if the intial conditions are correct choose the 'snapshot_to_check=0'.
#
# This analysis is made both qualitatively (histogram) and quantitatively (chisquare test) for
# the R,theta and phi distributions; for the velocity distribution and for the density one it's
# made only qualitatively.
#
# This Section is divided into three subsections:
#
# -Subsection 1: Check of the Position distribution (R,theta,phi).
#
# -Subsection 2: Check of the Density Function. We compute the density function by dividing the sphere
#                into shells, calculating the mass embedded in each shell (n_particles*m) and then
#                dividing it by the shell's volume
#
# -Subsection 3: Check of the q distribution. At a snapshot different from the first one (t=0), to check
#                if the velocity distribution is correct we derive the q values from the particles'
#                velocity and
#                potential energy at the selected snapshot and make a normalized histogram; we then plot the
#                theorical pdf of q and make a qualitatively comparison.
#                For a quantitatively evaluation we need to use the chi-square, whose computation is made
#                through the cdf of the q distribution through which one can find the expected frequencies;
#                in this case, the cdf must be calculated numerically: this can be done but we decided to
#                evaluate the q distribution only qualitatively.
#                If the snapshot to analyze is the first one (t=0), we compute q using the potential energy
#                computed through the Plummer's potential and not using the 'real' one (this because the
#                system evolution has not been already computed and so our data file do not contain the
#                potential energy (which in principle can be computed)).
#
# About the histograms and the statistical choices:
# - we decided to use the 'auto' number of bins selection: if the histogram is not too satisfactory
#   or if the chi-square test is not passed, try to give manually an appropiate bin selection
# - we used a chi square distribution test to verify how well the theoretical distribution fitted
#   the empirical ones. To use the chi-square distribution we considered only bins with a number
#   of counts greater than 9.
# - we plot on the normalized histograms the error bars. We assume a Poissonian distribution for
#   the counts and so the error associated to each bin is the square root of the number of counts in
#   that bin; for the normalized histogram's error bars we rescale the error dividing it by the area
#   of the non-normalized histogram (it's maybe a non correct procedure)
```

In [43]:
```
from scipy.stats import chisquare
```

In [44]:
```
snapshot_to_check = Nsnapshots-1
```

In [45]:
```
if snapshot_to_check == 0:
    Nparticles,M,r_scale = np.loadtxt('system_properties.txt')
    Nparticles = int(Nparticles)
    m = M/Nparticles
    Nlines_in_snapshot = 3+4*Nparticles
    file_name = "initial_conditions_file.txt"
else: file_name = data_files[snapshot_to_check]
```

In [46]:
```
# Subsection 1 - Check of the Position distribution #
```

In [47]:
```
positions = np.empty([Nparticles,Ndimensions])
x1 = np.empty([Nparticles])
y1 = np.empty([Nparticles])
z1 = np.empty([Nparticles])

radius = np.empty([Nparticles])
theta = np.empty([Nparticles])
phi = np.empty([Nparticles])

initial_line_position = 3+Nparticles

positions = np.genfromtxt(file_name,skip_header=initial_line_position,max_rows=Nparticles)

for i_particle in range(Nparticles):
    x1[i_particle] = positions[i_particle][0]
    y1[i_particle] = positions[i_particle][1]
    z1[i_particle] = positions[i_particle][2]


radius = (x1**2+y1**2+z1**2)**(1/2)
theta  = np.arccos(z1/radius)
phi    = np.arctan2(y1,x1)
phi    = phi+np.pi

del positions # free up memory
```

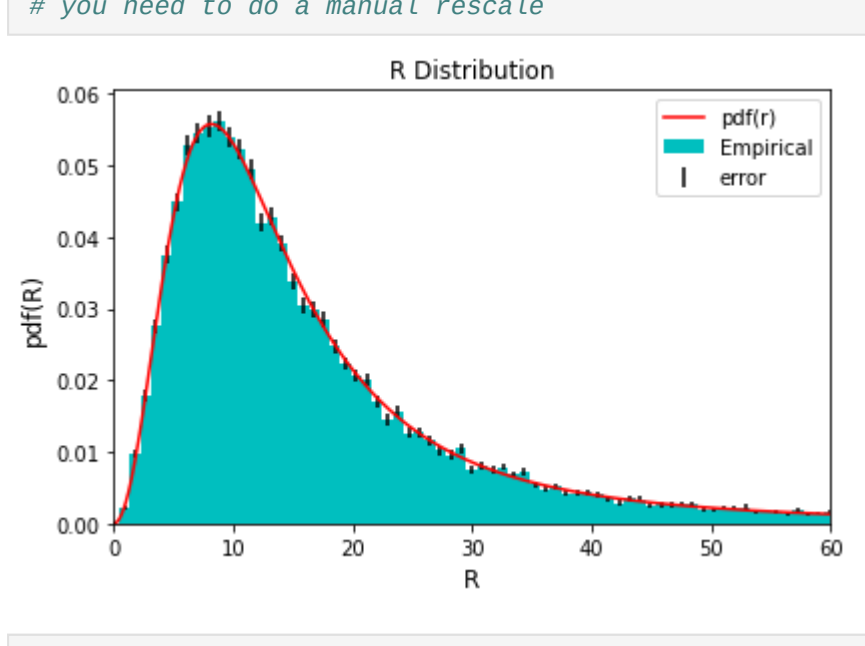
In [48]:
```
## Radius Distribution
```

In [49]:
```
## pdf and cdf of the radial positions
def pdf_r(r,b):
    return (3/b**3)*(r**2)*(1+(r/b)**2)**(-5/2)
def cdf_r(b,r):
    return b**(-3)*r**3/(1+(r/b)**2)**(3/2)
```

In [50]:
```
# Chisquare Section #

empirical,edges = np.histogram(radius,bins='auto')
counts = np.copy(empirical) # used in the histogram section
theorical = []
for i_bin in range(len(empirical)):
    if empirical[i_bin]>9:
        prob = cdf_r(r_scale,edges[i_bin+1]) - cdf_r(r_scale,edges[i_bin])
        theorical.append(prob*Nparticles)
empirical = empirical[empirical>9]

sta, pvalue = chisquare(empirical,theorical)
if pvalue>=0.05: print('pvalue: {}, test superato'.format(pvalue))
else: print('pvalue: {}, test non superato'.format(pvalue))
```

```
pvalue: 0.34801585825474207, test superato
```

In [51]:
```
# Histogram Section #

# Generation of the normalized histogram and of the error bars
area = sum(np.diff(edges)*counts)
error = np.sqrt(counts) # Poissonian errors
bincenters = 0.5*(edges[1:]+edges[:-1])

normalized_counts_r, bin_edges_r, hist_r = plt.hist(radius,bins='auto',range=None,density=True,
                                                     color='c',label='Empirical')
error = np.divide(error, area) # the error bar must be rescaled since we normalized the histogram
plt.errorbar(bincenters,normalized_counts_r,yerr=error,fmt='none',label='error',color='k')

## generation of the arrays containing some pdf's points
left_range = 0
right_range = r_scale*6
radius_pdf = np.empty(1000)
ascissa = np.linspace(left_range,right_range,1000)
radius_pdf = pdf_r(ascissa,r_scale)

plt.plot(ascissa,radius_pdf,color='r',label='pdf(r)')
plt.xlim(left_range,right_range)

plt.title('R Distribution')
plt.xlabel('R',fontsize=12)
plt.ylabel('pdf(R)',fontsize=12)

plt.legend()
plt.tight_layout()
#plt.savefig('R_histo_(t={}_Np={}_M={}_Rscal={})_(pval={:3.3}).jpg'.format(t[snapshot_to_check],Nparticles,M

# N.B. if you change the scale radius or if you want to see the whole histogram,
# you need to do a manual rescale
```



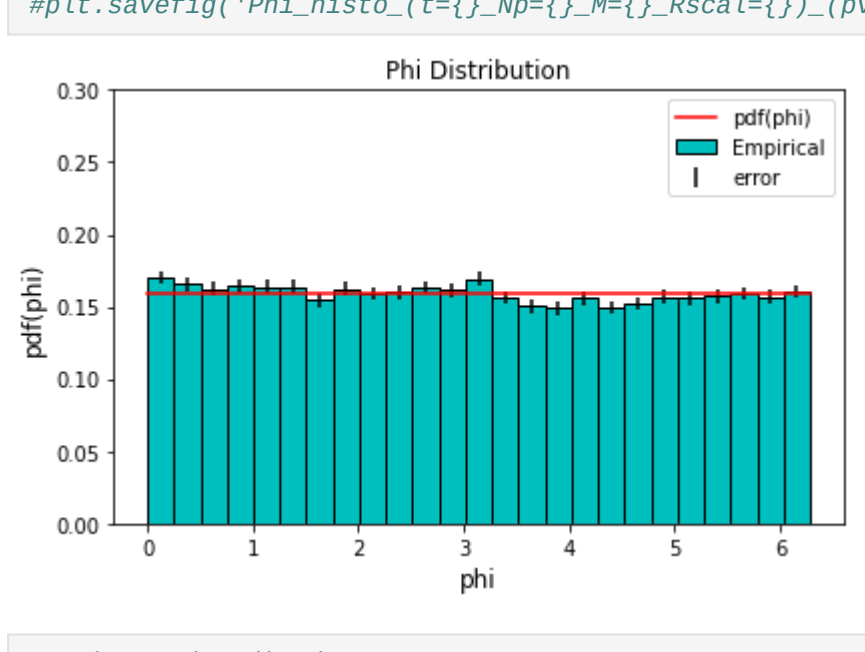
In [52]:
```
## Phi Distribution
```

In [53]:
```
## pdf and cdf of the phi angle
def pdf_p():
    return (2*math.pi)**(-1)
def cdf_p(p):
    return p*(2*math.pi)**(-1)
```

In [54]:
```
# Chisquare Section #

empirical,edges = np.histogram(phi,bins=25)
counts = np.copy(empirical) # used in the histogram Section
theorical = []
for i_bin in range(len(empirical)):
    if empirical[i_bin]>9:
        prob = cdf_p(edges[i_bin+1]) - cdf_p(edges[i_bin])
        theorical.append(prob*Nparticles)
empirical = empirical[empirical>9]

sta, pvalue = chisquare(empirical,theorical)
if pvalue>=0.05: print('pvalue: {}, test superato'.format(pvalue))
else: print('pvalue: {}, test non superato'.format(pvalue))
```

```
pvalue: 0.05065754231720089, test superato
```

In [55]:
```
# Histogram Section #

# Generation of the normalized histogram and of the error bars
area = sum(np.diff(edges)*counts)
error = np.sqrt(counts) # Poissonian errors
bincenters = 0.5*(edges[1:]+edges[:-1])

normalized_counts_p, Bin_edges_p, hist_p = plt.hist(phi,bins=25,range=None,density=True,
                                                     color='c',edgecolor='black',label='Empirical')
error = error/area # the error bar must be rescaled since we normalized the histogram
plt.errorbar(bincenters,normalized_counts_p,yerr=error,fmt='none',label='error',color='k')

## generation of the arrays containing some pdf's points
phi_pdf = np.empty(10)
ascissa = np.linspace(0,2*math.pi,10)
phi_pdf = np.ones(10)*pdf_p()

plt.plot(ascissa,phi_pdf,color='r',label='pdf(phi)')
plt.ylim(0,0.3)

plt.title('Phi Distribution')
plt.xlabel('phi',fontsize=12)
plt.ylabel('pdf(phi)',fontsize=12)

plt.legend()
plt.tight_layout()
#plt.savefig('Phi_histo_(t={}_Np={}_M={}_Rscal={})_(pval={:3.3}).jpg'.format(t[snapshot_to_check],Nparticles
```



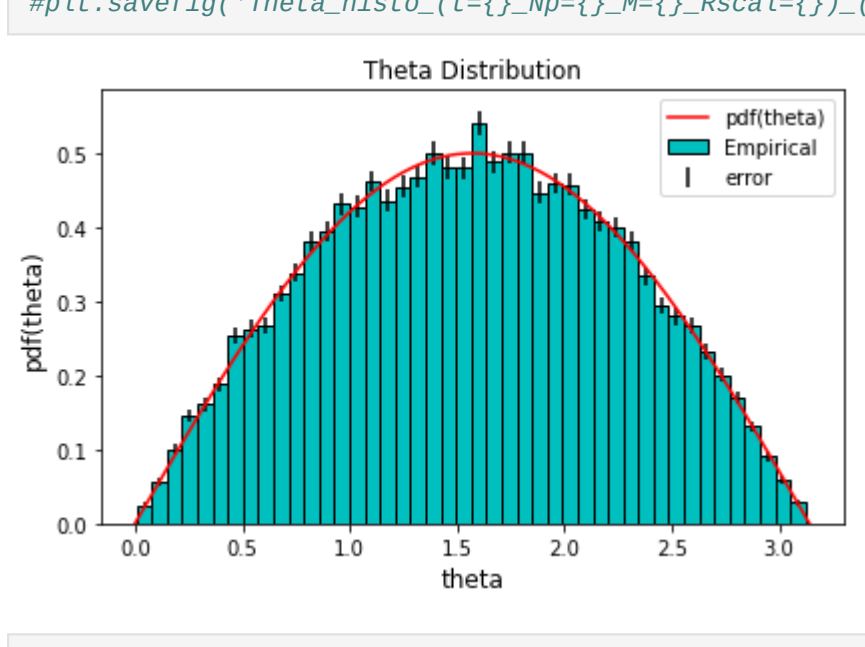
In [56]:
```
## Theta Distribution
```

In [57]:
```
## pdf and cdf of the phi angle
def pdf_t(t):
    return 0.5*np.sin(t)
def cdf_t(t):
    return 0.5*(1-np.cos(t))
```

In [58]:
```
# Chisquare Section #

empirical,edges = np.histogram(theta,bins=44)
counts = np.copy(empirical) # used in the Histogram Section
theorical = []
for i_bin in range(len(empirical)):
    if empirical[i_bin]>9:
        prob = cdf_t(edges[i_bin+1]) - cdf_t(edges[i_bin])
        theorical.append(prob*Nparticles)
empirical = empirical[empirical>9]

sta, pvalue = chisquare(empirical,theorical)
if pvalue>=0.05: print('pvalue: {}, test superato'.format(pvalue))
else: print('pvalue: {}, test non superato'.format(pvalue))
```

```
pvalue: 0.0952939083548818, test superato
```

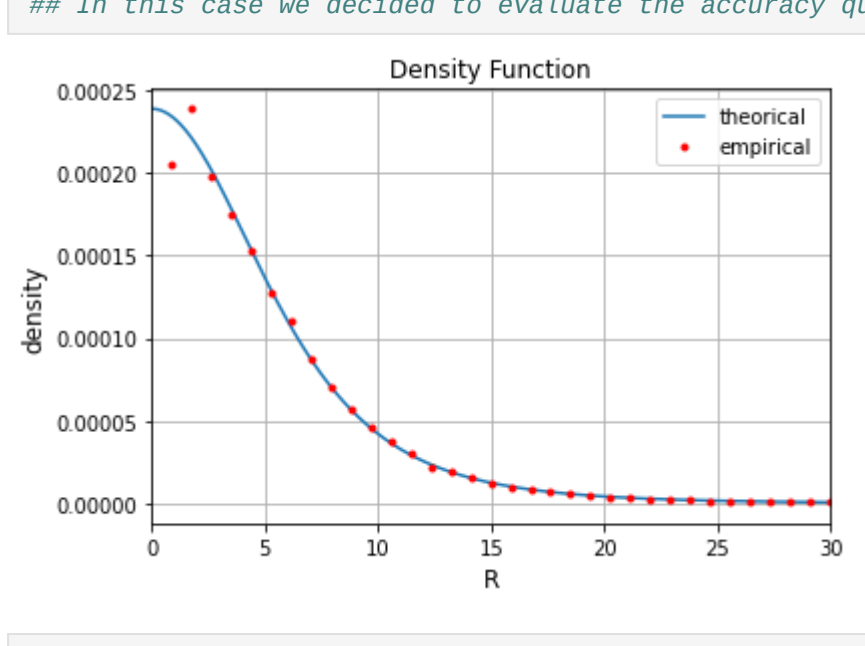
In [59]:
```
# Histogram Section #

# Generation of the normalized histogram and of the error bars
area = sum(np.diff(edges)*counts)
error = np.sqrt(counts) # Poissonian errors
bincenters = 0.5*(edges[1:]+edges[:-1])

normalized_counts_t, Bin_edges_t, hist_t = plt.hist(theta,bins=44,range=None,density=True,
                                                     color='c',edgecolor='black',label='Empirical')
error = error/area # the error bar must be rescaled since we normalized the histogram
plt.errorbar(bincenters,normalized_counts_t,yerr=error,fmt='none',label='error',color='k')

## generation of the arrays containing some pdf's points
theta_pdf = np.empty(1000)
ascissa = np.linspace(0,math.pi,1000)
theta_pdf = pdf_t(ascissa)

plt.plot(ascissa,theta_pdf,color='r',label='pdf(theta)')

plt.title('Theta Distribution')
plt.xlabel('theta',fontsize=12)
plt.ylabel('pdf(theta)',fontsize=12)

plt.legend()
plt.tight_layout()
#plt.savefig('Theta_histo_(t={}_Np={}_M={}_Rscal={})_(pval={:3.3}).jpg'.format(t[snapshot_to_check],Npartic
```



In [60]:
```
# Subsection 2 - Check of the Density Function #
```

In [61]:
```
# density function
def rho(r,b,M):
    return M/(4/3*math.pi*b**3)/(1+(r/b)**2)**(5/2)
```

In [62]:
```
## generation of the theorical density
left_range = 0
right_range = 3*r_scale
density = np.empty(1000)
ascissa = np.linspace(left_range,right_range,1000)
density = rho(ascissa,r_scale,M)
plt.plot(ascissa,density,label='theorical')

## computation of the empirical density
counts, bin_edges = np.histogram(radius, bins='auto')
system_density = []
for i in range(len(counts)):
    system_density.append(counts[i]*m/(4/3*math.pi*(bin_edges[i+1]**3-bin_edges[i]**3)))

center = bin_edges[1::]-(bin_edges[1::]-bin_edges[:-1])/2

plt.plot(center,system_density,'r.',label='empirical')
plt.xlim(left_range,right_range)

plt.title('Density Function')
plt.xlabel('R',fontsize=12)
plt.ylabel('density',fontsize=12)

plt.legend()
plt.grid()
plt.tight_layout()
#plt.savefig('density_function_(t={}_Np={}_M={}_Rscal={}).jpg'.format(t[snapshot_to_check],Nparticles,M,r_sc

## N.B. in order to compute the error associated to the empirical density, one should use
## the error propagation
## In this case we decided to evaluate the accuracy qualitatively
```



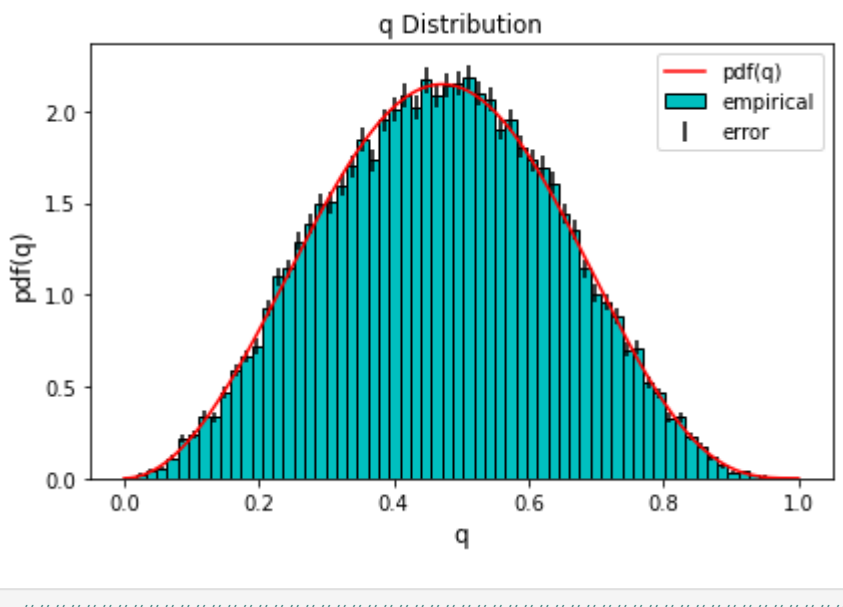
In [63]:
```
# Subsection 3 - Check of the q distribution
```

In [64]:
```
q = np.empty([Nparticles])
initial_line_position = 3+Nparticles
initial_line_velocity = 3+2*Nparticles
initial_line_potential = 3+3*Nparticles

velocities = np.empty([Nparticles,Ndimensions])

velocities = np.genfromtxt(file_name,skip_header=initial_line_velocity,max_rows=Nparticles)

vx1 = np.empty([Nparticles])
vy1 = np.empty([Nparticles])
vz1 = np.empty([Nparticles])
modV1 = np.empty([Nparticles])

for i_particle in range(Nparticles):
    vx1[i_particle] = velocities[i_particle][0]
    vy1[i_particle] = velocities[i_particle][1]
    vz1[i_particle] = velocities[i_particle][2]
modV1 = pow(vx1**2+vy1**2+vz1**2,0.5)

del velocities # free up memory
```

In [65]:
```
rel_pot = np.empty([Nparticles])

if snapshot_to_check!=0:
    rel_pot = -1*(np.genfromtxt(file_name,skip_header=initial_line_potential,max_rows=Nparticles))
else:
    rel_pot = M/(radius**2+r_scale**2)**(1/2)

v_max = (2*rel_pot)**(1/2)
q = modV1/v_max
```

In [66]:
```
Norm = 23.285605868167888 # normalization constant of the distribution function computed
                          #  numerically (you can find the code at the end of this notebook)
def pdf_q(q,Norm):
    return Norm*(1-q**2)**(7/2)*q**2
q_max = (2)**(1/2)/3  # abscissa where the distribution 'dist_func' has a maximum
f_max = pdf_q(q_max,Norm)
```

In [67]:
```
# Histogram Section #

counts, edges = np.histogram(q,bins='auto')
area = sum(np.diff(edges)*counts)
error = np.sqrt(counts) # Poissonian errors
bincenters = 0.5*(edges[1:]+edges[:-1])

normalized_counts_q, Bin_edges_q, hist_q = plt.hist(q,bins='auto',range=None,density=True,
                                                     color='c',edgecolor='black',label='empirical')

error = error/area # the error bar must be rescaled since we normalized the histogram
plt.errorbar(bincenters,normalized_counts_q,yerr=error,fmt='none',label='error',color='k')

## generation of the arrays containing some pdf's points
q_pdf = np.empty(1000)
ascissa = np.linspace(0,1,1000)
q_pdf = pdf_q(ascissa,Norm)

plt.plot(ascissa,q_pdf,color='r',label='pdf(q)')

plt.title('q Distribution')
plt.xlabel('q',fontsize=12)
plt.ylabel('pdf(q)',fontsize=12)

plt.legend()
plt.tight_layout()
#plt.savefig('q_histo_(t={}_Np={}_M={}_Rscal={}).jpg'.format(t[snapshot_to_check],Nparticles,M,r_scale,pvalu
```



In [68]:
```
####################################################################################################
```

In [ ]:
```
"""
## Computation of the normalization constant, N, of the distribution function f(q)=N*(1-q**2)**(7/2)*q**2
def dist_func(q):
    return (1-q**2)**(7/2)*q**2
q_max = (2)**(1/2)/3  # computing: derivative(distribution_function)=0
f_max = dist_func(q_max)
points = 1e08
points = int(points)
accepted = 0 # number of points accepted
for i in range(points):
    k1 = np.random.random() # generation of x between [0,1], interval of definition of f(q)
    k2 = np.random.random()*f_max # generation of y between [0,f_max], in the square that contains f(q)
    if k2<=dist_func(k1):
        accepted = accepted+1
f_area = (accepted/points)*(f_max*1) # fraction_accepted*area_square
f_normalization = f_area**(-1)
print('Normalization constant of the distribution function (points={}): {}'.format(points,f_normalization))
# Normalization constant of the distribution function (points=1e08): 23.285605868167888
"""
```

In [ ]:
```
'''
import time
start_time = time.time()
main()
print("--- %s seconds ---" % (time.time() - start_time))
'''
```

```
In [1]: ### Dynamical Friction in a stable Plummer's Sphere - Initial Conditions code for 'treecode' ###
```

```
In [2]: ###########################################################################################################
```

```
In [3]: ## Section 1 ##

        # Creation of the initial condition file without the perturber
        # A stable Plummer's sphere is initialized and the particles' positions and velocities are
        # saved in a file.txt that can be used to verify, using the Barners' tree code and the notebook
        # devoted to the Plummer stability, if the generated system is really at equilibirum
```

```
In [4]: import math
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [5]: ## Definition of the parameters of the system (in internal units)
        Nparticles = 30000
        Ndimensions = 3
        time = 0   # initial time simulation
        r_scale = 10 # scale radius of the Plummer's sphere
        M = 1 # total mass of the system
        m = M/Nparticles # mass of each particle (equal mass particles)

        ## distribution function (not normalized)
        def dist_func(q):
            return (1-q**2)**(7/2)*q**2

        q_max = (2)**(1/2)/3  # abscissa where the distribution 'dist_func' has a maximum
        f_max = dist_func(q_max)

        ## Positions setting function
        def positions(b):
            # position of the particle in spherical coordinates
            k = np.random.random() # generation of a random number uniformly distributed in [0,1)
            R = b*k**(1/3)/(1-k**(2/3))**(1/2)
            k = np.random.random()
            t = math.acos(1-(2*k))
            k = np.random.random()
            f = 2*(math.pi)*k
            # position of the particle in cartesian coordinates
            x = R*(math.sin(t))*(math.cos(f))
            y = R*(math.sin(t))*(math.sin(f))
            z = R*(math.cos(t))
            return x,y,z,R,t,f

        ## Velocity setting function
        def velocities(a,b):
            rel_pot = M/(a**2+b**2)**(1/2) # relative potential of the particle
            v_max = (2*rel_pot)**(1/2)
            leave = 'False' # used to exit the 'while' cycle
            while leave=='False':
                q = np.random.random()
                q1 = np.random.random()*f_max
                if q1<=dist_func(q):
                    vmod = q*v_max # modulo of the velocity
                    # spherical distribution in the velocities: theta and phi distributed in such a way
                    k = np.random.random()
                    t = math.acos(1-(2*k))
                    k = np.random.random()
                    f = 2*(math.pi)*k
                    # velocity of the particle in cartesian coordinates
                    vx = vmod*(math.sin(t))*(math.cos(f))
                    vy = vmod*(math.sin(t))*(math.sin(f))
                    vz = vmod*(math.cos(t))
                    leave = 'True'
            return vx,vy,vz,q

        ## Creation and filling of the position and velocity arrays
        x = np.empty(Nparticles)
        y = np.empty(Nparticles)
        z = np.empty(Nparticles)
        vx = np.empty(Nparticles)
        vy = np.empty(Nparticles)
        vz = np.empty(Nparticles)

        ## Radius, theta, phi and q arrays are used in the next section to check the initial conditions
        radius = np.empty(Nparticles)
        theta = np.empty(Nparticles)
        phi = np.empty(Nparticles)
        q = np.empty(Nparticles)

        for i in range(Nparticles):
            x[i],y[i],z[i],radius[i],theta[i],phi[i] = positions(r_scale)
            vx[i],vy[i],vz[i],q[i] = velocities(radius[i],r_scale)
```

```
In [6]: ## Saving the initial conditions in 'initial_conditions_file.txt' that then will be
        ## runned by the c++ code
        with open('initial_conditions_file.txt','w') as file:
            file.write("{}\n{}\n{}\n".format(Nparticles,Ndimensions,time))
            for i in range(Nparticles):
                file.write("{}\n".format(m))
            for i in range(Nparticles):
                file.write("{:25.15e}{:25.15e}{:25.15e}\n".format(x[i],y[i],z[i]))
            for i in range(Nparticles):
                file.write("{:25.15e}{:25.15e}{:25.15e}\n".format(vx[i],vy[i],vz[i]))
        ## Saving some system's parameters used during the analysis section
        with open('system_properties.txt','w') as file:
            file.write("{}\n{}\n{}\n".format(Nparticles,M,r_scale))
```

```
In [7]: ###########################################################################################################
```

```
In [8]: ## Section 2 ##

        # Check of the Initial Conditions
```

```
In [9]: # Before evolving the system, it's recommended to check the initial conditions: this can be done
        # directly in the  notebook devoted to the study of the Plummer stability (choosing the variable
        # 'snapshot_to_check = 0'); however, for computational costs, we recommend to do it here (indeed
        # here we have already the R, theta, phi and q arrays; in the analysis code they have to be recomputed)
```

```
In [10]: from scipy.stats import chisquare
```

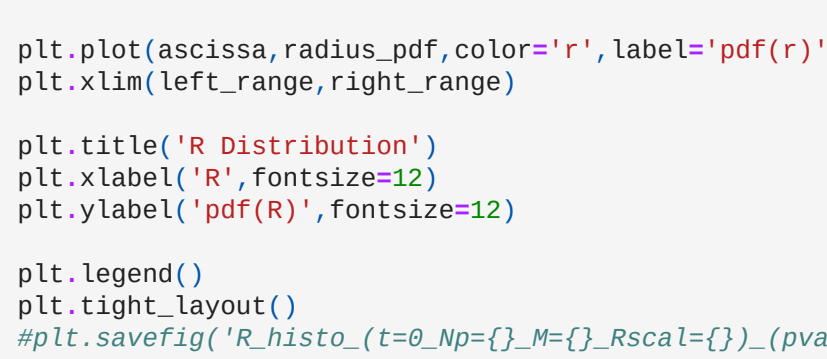
```
In [11]: ## Radius Distribution
```

```
In [12]: ## pdf and cdf of the radial positions
         def pdf_r(r,b):
             return (3/b**3)*(r**2)*(1+(r/b)**2)**(-5/2)
         def cdf_r(b,r):
             return b**(-3)*r**3/(1+(r/b)**2)**(3/2)
```

```
In [13]: # Chisquare Section #
         empirical,edges = np.histogram(radius,bins='auto')
         counts = np.copy(empirical) # used in the histogram section
         theorical = []
         for i_bin in range(len(empirical)):
             if empirical[i_bin]>9:
                 prob = cdf_r(r_scale,edges[i_bin+1]) - cdf_r(r_scale,edges[i_bin])
                 theorical.append(prob*Nparticles)
         empirical = empirical[empirical>9]

         sta, pvalue = chisquare(empirical,theorical)
         if pvalue>=0.05: print('pvalue: {}, test superato'.format(pvalue))
         else: print('pvalue: {}, test non superato'.format(pvalue))
```

```
pvalue: 0.09148713717263036, test superato
```

```
In [14]: # Histogram Section #

         # Generation of the normalized histogram and of the error bars
         area = sum(np.diff(edges)*counts)
         error = np.sqrt(counts) # Poissonian errors
         bincenters = 0.5*(edges[1:]+edges[:-1])

         normalized_counts_r, bin_edges_r, hist_r = plt.hist(radius,bins='auto',range=None,density=True,
                                                             color='c',label='Empirical')

         error = np.divide(error, area) # the error bar must be rescaled since we normalized the histogram
         plt.errorbar(bincenters,normalized_counts_r,yerr=error,fmt='none',label='error',color='k')

         ## generation of the arrays containing some pdf's points
         left_range = 0
         right_range = r_scale*6
         radius_pdf = np.empty(1000)
         ascissa = np.linspace(left_range,right_range,1000)
         radius_pdf = pdf_r(ascissa,r_scale)

         plt.plot(ascissa,radius_pdf,color='r',label='pdf(r)')
         plt.xlim(left_range,right_range)

         plt.title('R Distribution')
         plt.xlabel('R',fontsize=12)
         plt.ylabel('pdf(R)',fontsize=12)

         plt.legend()
         plt.tight_layout()
         #plt.savefig('R_histo_(t=0_Np={}_M={}_Rscal={})_(pval={:3.3}).jpg'.format(Nparticles,M,r_scale,pvalue))
         # N.B. if you change the scale radius or if you want to see the whole histogram, you need to do a manual res
```



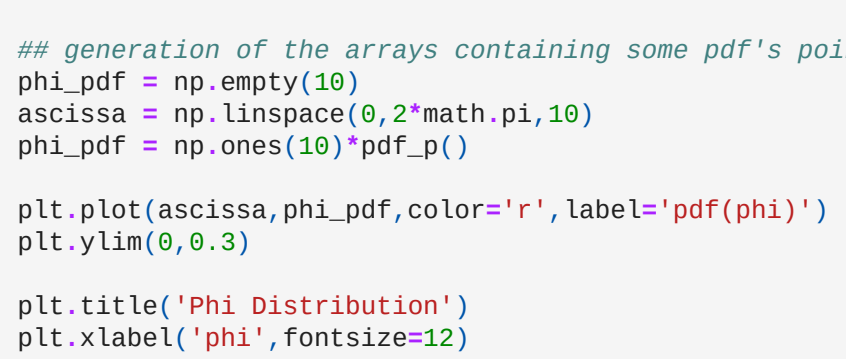

```
In [15]: ## Phi Distribution
```

```
In [16]: ## pdf and cdf of the phi angle
         def pdf_p():
             return (2*math.pi)**(-1)
         def cdf_p(p):
             return p*(2*math.pi)**(-1)
```

```
In [17]: # Chisquare Section #
         empirical,edges = np.histogram(phi,bins='auto')
         counts = np.copy(empirical) # used in the Histogram Section
         theorical = []
         for i_bin in range(len(empirical)):
             if empirical[i_bin]>9:
                 prob = cdf_p(edges[i_bin+1]) - cdf_p(edges[i_bin])
                 theorical.append(prob*Nparticles)
         empirical = empirical[empirical>9]

         sta, pvalue = chisquare(empirical,theorical)
         if pvalue>=0.05: print('pvalue: {}, test superato'.format(pvalue))
         else: print('pvalue: {}, test non superato'.format(pvalue))
```

```
pvalue: 0.22424478378585275, test superato
```

```
In [18]: # Histogram Section #

         # Generation of the normalized histogram and of the error bars
         area = sum(np.diff(edges)*counts)
         error = np.sqrt(counts) # Poissonian errors
         bincenters = 0.5*(edges[1:]+edges[:-1])

         normalized_counts_p, Bin_edges_p, hist_p = plt.hist(phi,bins='auto',range=None,
                                                             density=True,color='c',edgecolor='black',label='Empirica
         error = error/area # the error bar must be rescaled since we normalized the histogram
         plt.errorbar(bincenters,normalized_counts_p,yerr=error,fmt='none',label='error',color='k')

         ## generation of the arrays containing some pdf's points
         phi_pdf = np.empty(10)
         ascissa = np.linspace(0,2*math.pi,10)
         phi_pdf = np.ones(10)*pdf_p()

         plt.plot(ascissa,phi_pdf,color='r',label='pdf(phi)')
         plt.ylim(0,0.3)

         plt.title('Phi Distribution')
         plt.xlabel('phi',fontsize=12)
         plt.ylabel('pdf(phi)',fontsize=12)

         plt.legend()
         plt.tight_layout()
         #plt.savefig('Phi_histo_(t=0_Np={}_M={}_Rscal={})_(pval={:3.3}).jpg'.format(Nparticles,M,r_scale,pvalue))
```



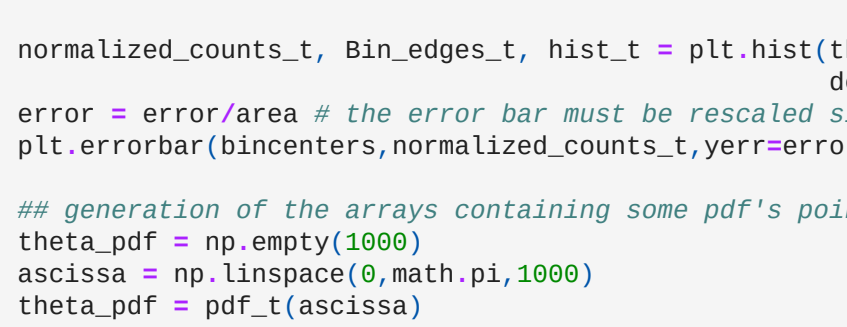

```
In [19]: ## Theta Distribution
```

```
In [20]: ## pdf and cdf of the phi angle
         def pdf_t(t):
             return 0.5*np.sin(t)
         def cdf_t(t):
             return 0.5*(1-np.cos(t))
```

```
In [21]: # Chisquare Section #
         empirical,edges = np.histogram(theta,bins='auto')
         counts = np.copy(empirical) # used in the Histogram Section
         theorical = []
         for i_bin in range(len(empirical)):
             if empirical[i_bin]>9:
                 prob = cdf_t(edges[i_bin+1]) - cdf_t(edges[i_bin])
                 theorical.append(prob*Nparticles)
         empirical = empirical[empirical>9]

         sta, pvalue = chisquare(empirical,theorical)
         if pvalue>=0.05: print('pvalue: {}, test superato'.format(pvalue))
         else: print('pvalue: {}, test non superato'.format(pvalue))
```

```
pvalue: 0.7487495881079602, test superato
```

```
In [22]: # Histogram Section #

         # Generation of the normalized histogram and of the error bars
         area = sum(np.diff(edges)*counts)
         error = np.sqrt(counts) # Poissonian errors
         bincenters = 0.5*(edges[1:]+edges[:-1])

         normalized_counts_t, Bin_edges_t, hist_t = plt.hist(theta,bins='auto',range=None,
                                                             density=True,color='c',edgecolor='black',label='Empirica
         error = error/area # the error bar must be rescaled since we normalized the histogram
         plt.errorbar(bincenters,normalized_counts_t,yerr=error,fmt='none',label='error',color='k')

         ## generation of the arrays containing some pdf's points
         theta_pdf = np.empty(1000)
         ascissa = np.linspace(0,math.pi,1000)
         theta_pdf = pdf_t(ascissa)

         plt.plot(ascissa,theta_pdf,color='r',label='pdf(theta)')

         plt.title('Theta Distribution')
         plt.xlabel('theta',fontsize=12)
         plt.ylabel('pdf(theta)',fontsize=12)

         plt.legend()
         plt.tight_layout()
         #plt.savefig('Theta_histo_(t=0_Np={}_M={}_Rscal={})_(pval={:3.3}).jpg'.format(Nparticles,M,r_scale,pvalue))
```




```
In [23]: ## Density Distribution
```

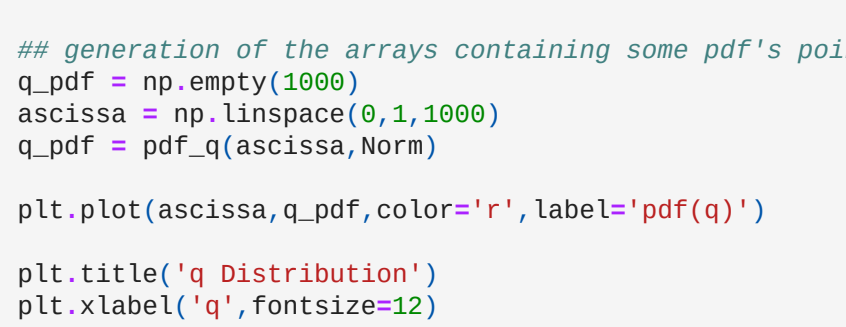
```
In [24]: # density function
         def rho(r,b,M):
             return M/(4/3*math.pi*b**3)/(1+(r/b)**2)**(5/2)
```

```
In [25]: ## generation of the theorical density
         left_range = 0
         right_range = r_scale*3
         density = np.empty(1000)
         ascissa = np.linspace(left_range,right_range,1000)
         density = rho(ascissa,r_scale,M)
         plt.plot(ascissa,density,label='theorical')

         ## computation of the empirical density
         counts, bin_edges = np.histogram(radius, bins='auto') # if you are not satisfied, choose 'bins' manually
         system_density = []
         for i in range(len(counts)):
             system_density.append(counts[i]*m/(4/3*math.pi*(bin_edges[i+1]**3-bin_edges[i]**3)))
         center = bin_edges[1::]-(bin_edges[1::]-bin_edges[:-1])/2
         plt.plot(center,system_density,'r.',label='empirical')

         plt.xlim(left_range,right_range)

         plt.title('Density Function')
         plt.xlabel('R',fontsize=12)
         plt.ylabel('density',fontsize=12)

         plt.legend()
         plt.grid()
         plt.tight_layout()
         #plt.savefig('density_function_(t=0_Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale))
         ## N.B. in order to compute the error associated to the empirical density, one should use the
         ## error propagation
         ## In this case we decided to evaluate the accuracy qualitatively
```


```
In [26]: ## q Distribution
```

```
In [27]: Norm = 23.285605868167888 # normalization constant of the distribution function computed
                                   # numerically (you can find the code at the end of this notebook)
         def pdf_q(q,Norm):
             return Norm*(1-q**2)**(7/2)*q**2
         q_max = (2)**(1/2)/3  # abscissa where the distribution 'dist_func' has a maximum
         f_max = pdf_q(q_max,Norm)
```

```
In [28]: # Histogram Section #

         counts, edges = np.histogram(q,bins='auto')
         area = sum(np.diff(edges)*counts)
         error = np.sqrt(counts) # Poissonian errors
         bincenters = 0.5*(edges[1:]+edges[:-1])

         normalized_counts_q, Bin_edges_q, hist_q = plt.hist(q,bins='auto',range=None,
                                                             density=True,color='c',edgecolor='black',label='empirica

         error = error/area # the error bar must be rescaled since we normalized the histogram
         plt.errorbar(bincenters,normalized_counts_q,yerr=error,fmt='none',label='error',color='k')

         ## generation of the arrays containing some pdf's points
         q_pdf = np.empty(1000)
         ascissa = np.linspace(0,1,1000)
         q_pdf = pdf_q(ascissa,Norm)

         plt.plot(ascissa,q_pdf,color='r',label='pdf(q)')

         plt.title('q Distribution')
         plt.xlabel('q',fontsize=12)
         plt.ylabel('pdf(q)',fontsize=12)

         plt.legend()
         plt.tight_layout()
         #plt.savefig('q_histo_(t=0_Np={}_M={}_Rscal={}).jpg'.format(Nparticles,M,r_scale,pvalue))
```




```
In [29]: ###########################################################################################################
```

```
In [30]: ## Section 3 ##

         # Instruction to compile the 'treecode' from the terminal and hints to set the parameters ##
```

```
In [31]: # When you have runned 'Section 1', your system initial conditions will be saved in
         # 'initial_conditions_file.txt'; then, before running the 'notebook for the Plummer's Sphere Stability,
         #  you need to run from the terminal the c++ executable file. You can write (for example):
         #
         #  ./treecode in=initial_conditions_file.txt out=%d.data dtime=0.05 eps=0.1 theta=0.1 options=out-phi
         #                                                            tstop=300 dtout=10 > system_description.txt
         #
         # All the parameters are expressed in internal units; it's useful to compute the dynamical time of the syste

         t_dynamical = r_scale/(M/r_scale)**(1/2)

         # Here you can find the description of some of the parameters passed through the terminal
         # (see for more details the documentation of the 'Barnes tree code' - http://www.ifa.hawaii.edu/
         #                                                            faculty/barnes/treecode/treeguide.html):
         #
         # - dtime: it is the integration time step; for example choose: dtime=t_dynamical/(10^a);
         #   if you decrease eps, you should also decrease dtime
         #   To be correct, 't_dynamical' isn't computed in the correct way; however, it's of the same order
         #   of the real one.

         dtime = t_dynamical/(10**2)

         # - eps: it is the smoothing length used in the gravitational force calculation.
         #  Choose a value that is a fraction of the typical separation between particles:
         #                                                                  10^(-b)*(Vol/Nparticles)**(1/3)
         #  Here we divided the sphere in shells, computed the particles' mean distance in each shell
         #  and selected the smallest one

         counts, bin_edges = np.histogram(radius, bins='auto')
         MeanSeparation = []
         for i in range(len(counts)):
             if counts[i]!=0:
                 MeanSeparation.append( ((4/3*math.pi*(bin_edges[i+1]**3-bin_edges[i]**3))/counts[i])**(1/3))
         MinMeanSeparation = min(MeanSeparation)
         eps = MinMeanSeparation/5

         # - theta: it is the opening angle used to adjust the accuracy of the force calculation.
         #   The less is theta the more accurate is the results, albeit at greater computational expense.

         theta = 0.5

         # - options=out-phi: it saves in the output_data.txt also the potential energy

         # - tstop: it is the time at which the N-Body integration terminates. Set it, for example,
         #          as tstop=10*t_dynamical

         tstop = 10*t_dynamical

         # - dtout: it is the time interval between output files. To insure that outputs are performed
         #   when expected, dtout should be a multiple of dtime. The number of snapshots depends on 'dtout':
         #   the less its value is, the more number of outputs you'll have. A greater number of snapshots
         #   will be useful during the analysis because you'll have more information about the system but
         #   the computational cost of the analysis will be increased.

         dtout = dtime*10

         Nsnapshots = int(tstop/dtout)
```

```
In [32]: print('\ndtime: ',dtime)
         print('\nMean separation: ',MinMeanSeparation,'    eps: ',eps)
         print('\ntheta: ', theta)
         print('\ntstop: ', tstop)
         print('\ndtout: ', dtout)
         print('\nNsnaphots: ', Nsnapshots)
```

```
dtime:  0.31622776601683794

Mean separation:  0.5328963549948176     eps:  0.10657927099896351

theta:  0.5

tstop:  316.2277660168379

dtout:  3.1622776601683795

Nsnaphots:  99
```

```
In [33]: ###########################################################################################################
```

```
In [34]: ## Section 4 ##

         # Section used to include the perturber in the previously generated system. This Section can be run
         # independently from the previous ones; you need to include in the same directory the .txt files
         # generated in Section 1
```

```
In [35]: import math
         import numpy as np
```

```
In [36]: def plummer_pot(M,r,a):
             return -M/(r**2+a**2)**(1/2)
```

```
In [37]: ## Number of particles, total mass and scale radius of the already generated Plummer's sphere
         Nparticles,M,r_scale = np.loadtxt('system_properties.txt')
         Nparticles = int(Nparticles)

         ## Setting of the perturber's properties
         M_pert = 0.01
         r_apocenter = 6
         r_pericenter = 2

         # potential and velocities at the apo- and peri- center (obtained assuming conservation of the
         # energy and of the angular momentum)
         pot_apocenter = plummer_pot(M,r_apocenter,r_scale)
         pot_pericenter = plummer_pot(M,r_pericenter,r_scale)
         v_apocenter = np.sqrt( 2*(pot_apocenter-pot_pericenter) /((r_apocenter/r_pericenter)**2-1) )
         v_pericenter = np.sqrt( 2*(pot_pericenter-pot_apocenter) /((r_pericenter/r_apocenter)**2-1) )

         # position and velocity of the perturber
         pos_pert = [0,r_apocenter,0]
         v_pert = [v_apocenter,0,0]
```

```
In [38]: with open('initial_conditions_file.txt','r') as file:
             data = file.readlines()

         Nbodies = Nparticles+1
         data[0] = ("{}\n".format(Nbodies))
         data.insert(3,"{}\n".format(M_pert))
         data.insert(3+Nbodies,"{:25.15e}{:25.15e}{:25.15e}\n".format(pos_pert[0],pos_pert[1],pos_pert[2]))
         data.insert(3+2*Nbodies,"{:25.15e}{:25.15e}{:25.15e}\n".format(v_pert[0],v_pert[1],v_pert[2]))
         with open('initial_conditions_file_perturber_1.txt','w') as file:
             for i in range(len(data)):
                 file.write(data[i])
```

```
In [39]: ## Saving some system's parameters used during the analysis section
         with open('system_properties_perturber.txt','w') as file:
             file.write("{}\n{}\n{}\n{}\n".format(Nbodies,r_scale,M,M_pert))
```

```
In [40]: ###########################################################################################################
```

```
In [ ]: """
        ## Computation of the normalization constant, N, of the distribution function f(q)=N*(1-q**2)**(7/2)*q**2
        # We implemented manually the code but, for a better precision, use the python's functions
        def dist_func(q):
            return (1-q**2)**(7/2)*q**2
        q_max = (2)**(1/2)/3  # computing: derivative(distribution_function)=0
        f_max = dist_func(q_max)
        points = 1e08
        points = int(points)
        accepted = 0 # number of points accepted
        for i in range(points):
            k1 = np.random.random() # generation of x between [0,1], interval of definition of f(q)
            k2 = np.random.random()*f_max # generation of y between [0,f_max], in the square that contains f(q)
            if k2<=dist_func(k1):
                accepted = accepted+1
        f_area = (accepted/points)*(f_max*1) # fraction_accepted*area_square
        f_normalization = f_area**(-1)
        print('Normalization constant of the distribution function (points={}): {}'.format(points,f_normalization))
        # Normalization constant of the distribution function (points=1e08): 23.285605868167888
        """
```

In [1]:
```
### Dynamical Friction in a Plummer Sphere - Analysis Code ###
```

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

In [3]:
```
## Section 1 ##

# - Setting of useful parameters (Nparticles,TotMass,ScaleRadiusSphere,Nsnapshots)
# - Read and order the name of the generated files
# - Selection of the snapshots to analyze
```

In [4]:
```
## Number of bodies (particles+perturber), scale radius, Plummer's sphere mass, perturber's mass
Nbodies,r_scale,M,M_pert = np.loadtxt('system_properties_perturber.txt')
Nbodies = int(Nbodies)
Nparticles = Nbodies-1

m = M/Nparticles # mass of a single particle
M_tot = M+M_pert

Ndimensions = 3
Nlines_in_snapshot = 3+4*Nbodies # Nbodies(1row)+Ndimensions(1)+time(1)+mass_bodies(N)+
                                 # +positions(N)+velocities(N)+potential_energy(N)
```

In [5]:
```
## Reading the name of the generated files (output of 'treecode') and order them chronologically (each one
## is the system description at a given snapshot)

import glob, os
import re # regular expression

path_data = '/home/efrem/Astronomia_extragalattica/2-NBodySimulations/6-Dinamical_Friction_(alg:treecodeBarn
data_files = []
os.chdir(path_data)
for file in glob.glob("*.data"): # selection of the directory's files that end with '.data'
    data_files.append(file)

# 'data_files' contains the output 'treecode' files name; each file contains the state of the system at
# a given time; since the files in 'data_files' are not in chronologically ordered, here a code to do that

def chronological_order(list):
    search_num = re.compile('(^\d+)') # regular expresion: '^' start from the string beginning,
                                      #'\d' select the first number of the string,  '+' select all
                                      #the numbers of the string
    number_to_name = []
    for i in list:
        l = search_num.search(i)
        number = int(l.group(1))
        number_to_name.append(number)
    number_to_name = sorted(number_to_name)
    # add .data
    for i in range(len(number_to_name)):
        number_to_name[i]=('{}.data'.format(number_to_name[i]))
    return (number_to_name)

data_files = chronological_order(data_files)
```

In [6]:
```
# Number of Snapshots
Nsnapshots = len(data_files)
Nsnapshots = int(Nsnapshots)
```

In [7]:
```
##########################################################################################################
```

In [8]:
```
## Selection of the snapshots to analyze
# Here you can choose the number of snapshots that you want to analyze. The code is very performant,
# so we suggest to analyze all of them ('step=1'). If, however, you want to select some of them you
# can modify the 'step' values or you can manually select the ones needed (commented part)

sn = 0
step = 1 # if 1, analyze all the snapshots
selected_snapshot = []
for i in range(int(Nsnapshots/step)):
    selected_snapshot.append(step*i)

# Manual selection
#selected_snapshot = [0,Nsnapshots-1] # this is an example, where we analyze the first and the last snaphots

print('The total number of snapshots is:,',Nsnapshots)
print('\nThe selected snapshots are: ',selected_snapshot)
print('\nYou want to analyze {} snapshots'.format(len(selected_snapshot)))
```

```
The total number of snapshots is:, 161

The selected snapshots are:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 10
2, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123,
124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 14
5, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160]

You want to analyze 161 snapshots
```

In [9]:
```
##########################################################################################################
```

In [10]:
```
## Section 2 ##

# - Definition and filling of the time array
```

In [11]:
```
t = np.empty([len(selected_snapshot)])
for i_snapshot in range(len(selected_snapshot)):
    file_name = data_files[selected_snapshot[i_snapshot]]
    initial_line_time = 2
    t[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_time,max_rows=1)
```

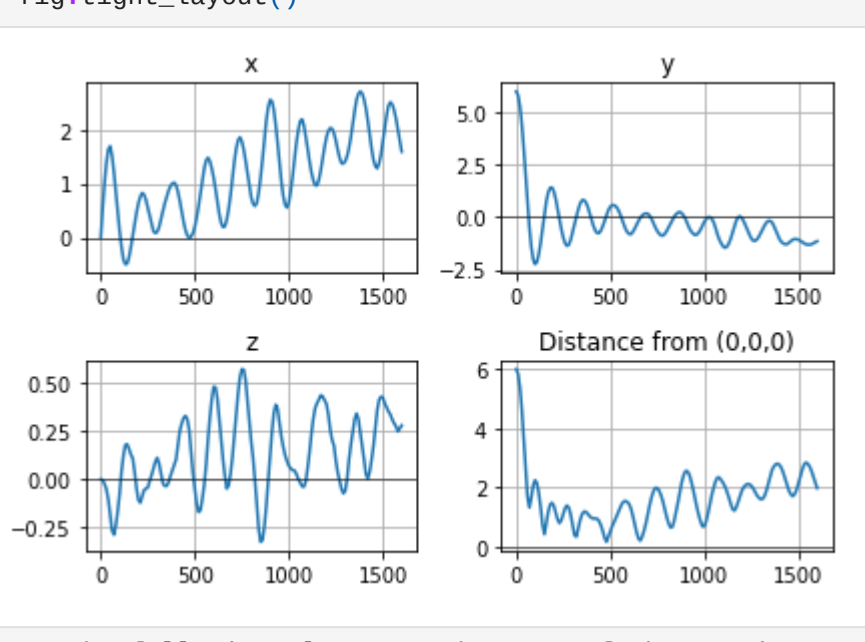
In [12]:
```
##########################################################################################################
```

In [13]:
```
## Section 3 ##

# - Positions of the bodies in the system
# - Plot of the evolution of the perturber's coordinates and of its distance from (0,0,0)
```

In [14]:
```
positions = np.empty([len(selected_snapshot),Nbodies,Ndimensions])

for i_snapshot in range(len(selected_snapshot)):
    file_name = data_files[selected_snapshot[i_snapshot]]
    initial_line_position = 3+Nbodies
    positions[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_position,max_rows=Nbodies)

x = np.empty([len(selected_snapshot),Nbodies])
y = np.empty([len(selected_snapshot),Nbodies])
z = np.empty([len(selected_snapshot),Nbodies])
dist = np.empty([len(selected_snapshot),Nbodies]) # particles' distance from (0,0,0) at each time

for i_snapshot in range(len(selected_snapshot)):
    for i_particle in range(Nbodies):
        x[i_snapshot,i_particle] = positions[i_snapshot][i_particle][0]
        y[i_snapshot,i_particle] = positions[i_snapshot][i_particle][1]
        z[i_snapshot,i_particle] = positions[i_snapshot][i_particle][2]
dist = pow(x**2+y**2+z**2,0.5)

del positions # free up memory
```

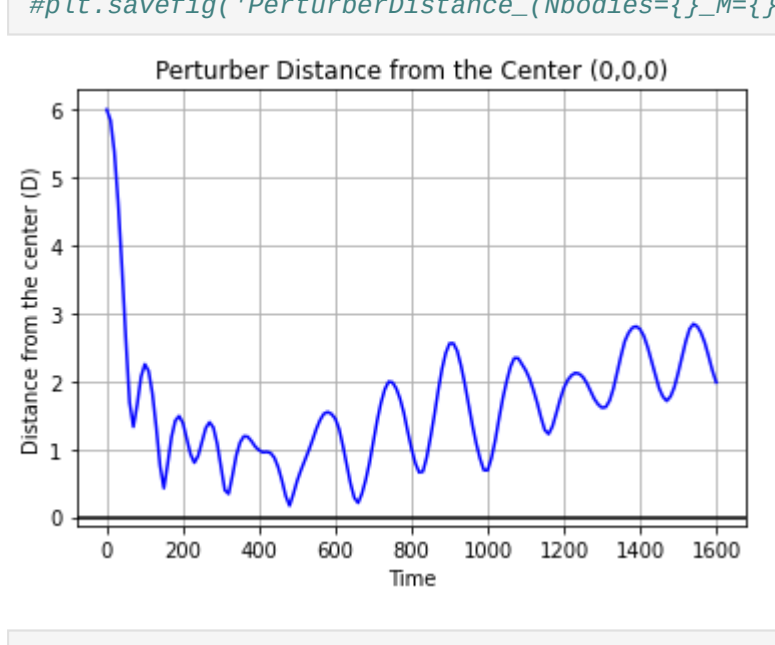
In [15]:
```
## Perturber position and distance respect to the center (0,0,0) at each selected snapshot

# We write the initial_condition_file_perturber.txt such that the perturber is considered as the
# 'first particle' because, in this way, we know what are the lines that we need to read from the output
# data to have information  about it
x_pert = np.empty([len(selected_snapshot)])
y_pert = np.empty([len(selected_snapshot)])
z_pert = np.empty([len(selected_snapshot)])
dist_pert = np.empty([len(selected_snapshot)])

for i_snapshot in range(len(selected_snapshot)):
    x_pert[i_snapshot] = x[i_snapshot,0]
    y_pert[i_snapshot] = y[i_snapshot,0]
    z_pert[i_snapshot] = z[i_snapshot,0]

dist_pert = pow((x_pert**2+y_pert**2+z_pert**2),0.5)
```

In [16]:
```
## Plot of the evolution of the (x,y,z) perturber's coordinates and of its distande respect to (0,0,0)
# This plot is made only here in the notebook in order to have in a single plot an idea of what is
# happening to the perturber's position (we note that a description of the system from the center of mass
# will be better since, for example, the x coordinate, while oscillating, has a global motion)

fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(t, x_pert)
axs[0, 0].grid()
axs[0, 0].axhline(0,0,1,c='k',ls='-',lw=0.5)
axs[0, 0].set_title("x")

axs[1, 0].plot(t, z_pert)
axs[1, 0].grid()
axs[1, 0].axhline(0,0,1,c='k',ls='-',lw=0.5)
axs[1, 0].set_title("z")


axs[0, 1].plot(t, y_pert)
axs[0, 1].grid()
axs[0, 1].axhline(0,0,1,c='k',ls='-',lw=0.5)
axs[0, 1].set_title("y")

axs[1, 1].plot(t,dist_pert)
axs[1, 1].grid()
axs[1, 1].axhline(0,0,1,c='k',ls='-',lw=0.5)
axs[1, 1].set_title("Distance from (0,0,0)")

fig.tight_layout()
```

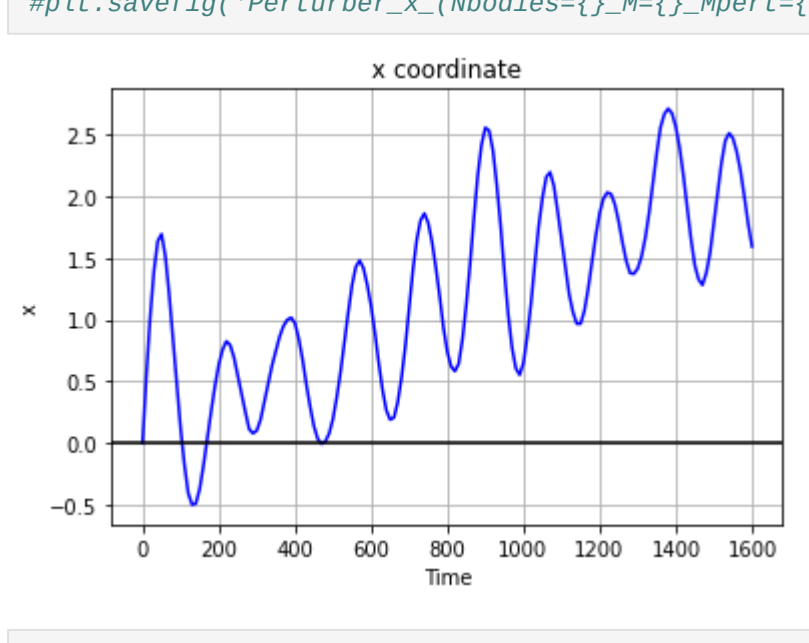


In [17]:
```
## The following plots are the same of the previous one (in a sense it's a repetition but the aims are
## different: the plots above are used to have an overview, the plots below are bigger and so clearer
## and are saved)
```

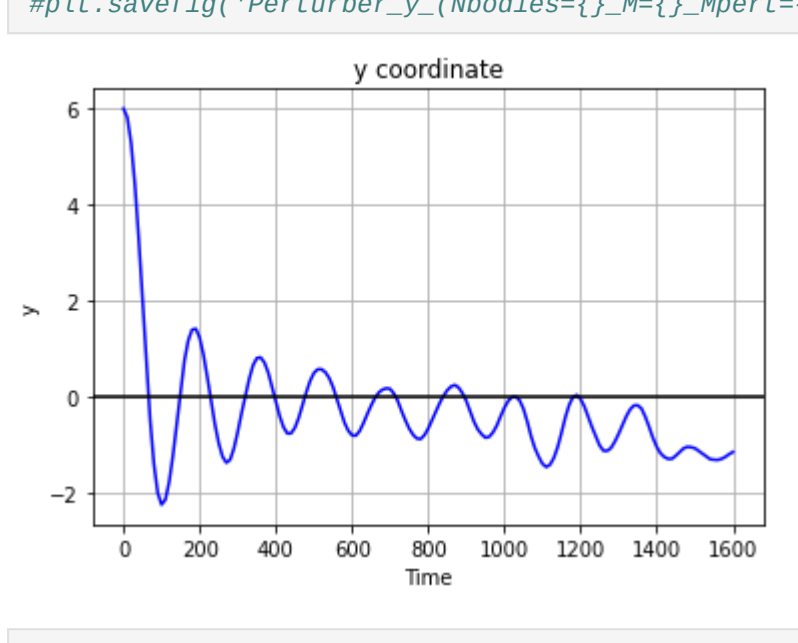
In [18]:
```
## Plot of the Perturber distance respect to (0,0,0) over time
plt.plot(t,dist_pert,color='b')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('Perturber Distance from the Center (0,0,0)')
plt.xlabel('Time')
plt.ylabel('Distance from the center (D)')

plt.grid()
#plt.savefig('PerturberDistance_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



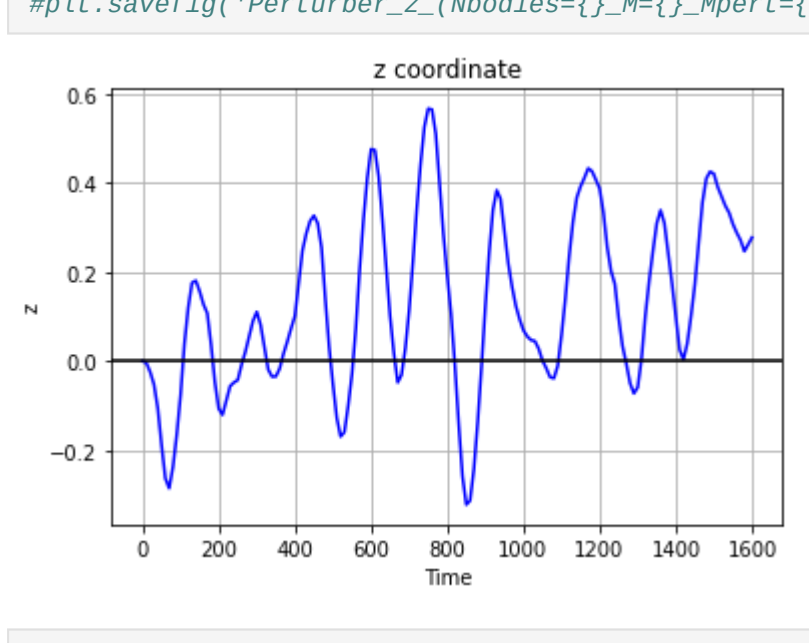
In [19]:
```
# Plot of the evolution of the x coordinate of the perturber over time
plt.plot(t,x_pert,c='b')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('x coordinate')
plt.xlabel('Time')
plt.ylabel('x')

plt.grid()
#plt.savefig('Perturber_x_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [20]:
```
# Plot of the evolution of the y coordinate of the perturber over time
plt.plot(t,y_pert,c='b')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('y coordinate')
plt.xlabel('Time')
plt.ylabel('y')

plt.grid()
#plt.savefig('Perturber_y_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [21]:
```
# Plot of the evolution of the z coordinate of the perturber over time
plt.plot(t,z_pert,c='b')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('z coordinate')
plt.xlabel('Time')
plt.ylabel('z')

plt.grid()
#plt.savefig('Perturber_z_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [22]:
```
# Before analyzing the system properties, we need to change the reference frame from (0,0,0) to the
# center of mass
```

In [23]:
```
##########################################################################################################
```

In [24]:
```
## Section 4 ##

# - Computation of the System Center of Mass evolution (CoM where all the bodies are considered, CoM where
#   only bodies within LagRad80 are considerd, CoM where only particles (perturber excluded) within
#   LagRad80 are considered)
```

In [25]:
```
## Center of mass
```

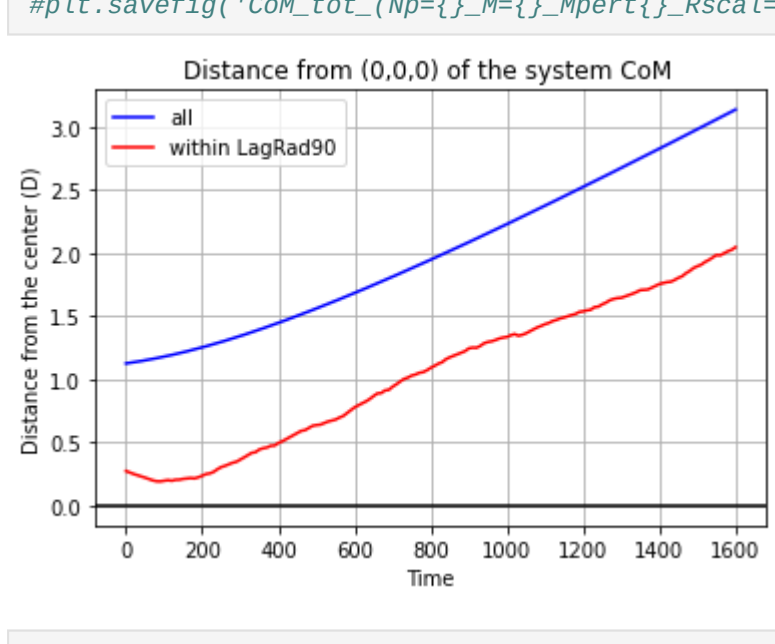
In [26]:
```
# N = len(selected_snapshot), R = distance vector (only particles with r<=R are considered in the
# computation) (ex. R = LagRad80). If you want to consider all particles:
# R = np.ones([len(selected_snapshot)])*np.max((np.sort(dist)[-1]))

def CoM(N,Nbodies,R,x,y,z,dist,mass_bodies):
    M_tot = np.sum(mass_bodies)
    CoM_x = np.empty([N])
    CoM_y = np.empty([N])
    CoM_z = np.empty([N])
    for i_snapshot in range(N):
        c_x = 0
        c_y = 0
        c_z = 0
        for i_particle in range(Nbodies):
            if dist[i_snapshot][i_particle] <= (R[i_snapshot]):
                c_x = c_x + x[i_snapshot,i_particle]*mass_bodies[i_particle]
                c_y = c_y + y[i_snapshot,i_particle]*mass_bodies[i_particle]
                c_z = c_z + z[i_snapshot,i_particle]*mass_bodies[i_particle]
        CoM_x[i_snapshot] = c_x
        CoM_y[i_snapshot] = c_y
        CoM_z[i_snapshot] = c_z
    CoM_x = CoM_x/M_tot
    CoM_y = CoM_y/M_tot
    CoM_z = CoM_z/M_tot
    dist_CoM = pow(CoM_x**2+CoM_y**2+CoM_z**2,0.5)
    return CoM_x,CoM_y,CoM_z,dist_CoM
```

In [27]:
```
mass_bodies = np.ones(Nparticles)*m
mass_bodies = np.insert(mass_bodies,0,M_pert)
```

In [28]:
```
# CoM where all the bodies are considered (particles and perturber)
R = np.ones([len(selected_snapshot)])*np.max((np.sort(dist)[-1]))
CoM_x,CoM_y,CoM_z,dist_CoM = CoM(len(selected_snapshot),Nbodies,R,x,y,z,dist,mass_bodies)
del R
```

In [29]:
```
## CoM of the system (only bodies within LagRad90)

LagRad90 = np.empty([len(selected_snapshot)])
sorted_distance = np.sort(np.delete(dist,0,1)) # the disturber distance is not considered
for i_snapshot in range(len(selected_snapshot)):
    LagRad90[i_snapshot] = sorted_distance[i_snapshot][int(0.9*Nparticles)-1]

R = LagRad90
CoM_x2,CoM_y2,CoM_z2,dist_CoM2 = CoM(len(selected_snapshot),Nbodies,R,x,y,z,dist,mass_bodies)
del R
```

In [30]:
```
plt.plot(t,dist_CoM,c='b',label='all')
plt.plot(t,dist_CoM2,c='r',label='within LagRad90')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('Distance from (0,0,0) of the system CoM ')
plt.xlabel('Time')
plt.ylabel('Distance from the center (D)')

plt.legend()
plt.grid()
#plt.savefig('CoM_tot_(Np={}_M={}_Mpert{}_Rscal={}).jpg'.format(Nparticles,M,M_pert,r_scale))
```



In [31]:
```
##########################################################################################################
```

In [32]:
```
## Section 5 ##

# - Change of reference frame in positions and velocities.
#   From (0,0,0) to the Center of mass of the system (only bodies within LagRad90 are considered)
# - Plot of the evolution of the new and old CoM (to verify if we succeded in our change of R.F.)
```

In [33]:
```
## Positions and velocities translation
```

In [34]:
```
R = LagRad90
CoM_x,CoM_y,CoM_z,dist_CoM = CoM(len(selected_snapshot),Nbodies,R,x,y,z,dist,mass_bodies)
del R

CoM_x_old = CoM_x
CoM_y_old = CoM_y
CoM_z_old = CoM_z
dist_CoM_old = dist_CoM
```

In [35]:
```
## Before changing the position reference frame (translating the system in the CoM computed considering
## the bodies within LagRad90), we change the velocity reference frame (we compute at each snapshot the
## velocity of the CoM (bodies within LagRad90))
```

In [36]:
```
# Definition of the velocities respect to the original reference frame
velocities = np.empty([len(selected_snapshot),Nbodies,Ndimensions])

for i_snapshot in range(len(selected_snapshot)):
    file_name = data_files[selected_snapshot[i_snapshot]]
    initial_line_velocity = 3+2*Nbodies
    velocities[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_velocity,max_rows=Nbodies)

vx = np.empty([len(selected_snapshot),Nbodies])
vy = np.empty([len(selected_snapshot),Nbodies])
vz = np.empty([len(selected_snapshot),Nbodies])
modV = np.empty([len(selected_snapshot),Nbodies])

for i_snapshot in range(len(selected_snapshot)):
    for i_particle in range(Nbodies):
        vx[i_snapshot,i_particle] = velocities[i_snapshot][i_particle][0]
        vy[i_snapshot,i_particle] = velocities[i_snapshot][i_particle][1]
        vz[i_snapshot,i_particle] = velocities[i_snapshot][i_particle][2]
modV = pow(vx**2+vy**2+vz**2,0.5)

del velocities # free up memory
```

In [37]:
```
## Velocity of the center of mass that embeds the 90% of the mass
CoM_vx = np.empty([len(selected_snapshot)])
CoM_vy = np.empty([len(selected_snapshot)])
CoM_vz = np.empty([len(selected_snapshot)])
modV_CoM = np.empty([len(selected_snapshot)])

for i in range(len(selected_snapshot)):
    boolArray = dist[i]<=LagRad90[i] # selection at each snapshot of the particles needed
    a = vx[i][boolArray]
    b = vy[i][boolArray]
    c = vz[i][boolArray]
    e = mass_bodies[boolArray]
    CoM_vx[i] = np.sum(a*e)/np.sum(e)
    CoM_vy[i] = np.sum(b*e)/np.sum(e)
    CoM_vz[i] = np.sum(c*e)/np.sum(e)
```

In [38]:
```
## Computation of the new velocity arrays
for i_snapshot in range(len(selected_snapshot)):
    vx[i_snapshot] = vx[i_snapshot]-CoM_vx[i_snapshot]
    vy[i_snapshot] = vy[i_snapshot]-CoM_vy[i_snapshot]
    vz[i_snapshot] = vz[i_snapshot]-CoM_vz[i_snapshot]
modV = pow(vx**2+vy**2+vz**2,0.5)
```

In [39]:
```
## Perturber velocity
vx_pert = np.empty([len(selected_snapshot)])
vy_pert = np.empty([len(selected_snapshot)])
vz_pert = np.empty([len(selected_snapshot)])
mod_Vpert = np.empty([len(selected_snapshot)])

for i_snapshot in range(len(selected_snapshot)):
    vx_pert[i_snapshot] = vx[i_snapshot,0]
    vy_pert[i_snapshot] = vy[i_snapshot,0]
    vz_pert[i_snapshot] = vz[i_snapshot,0]

mod_Vpert = pow((vx_pert**2+vy_pert**2+vz_pert**2),0.5)
```

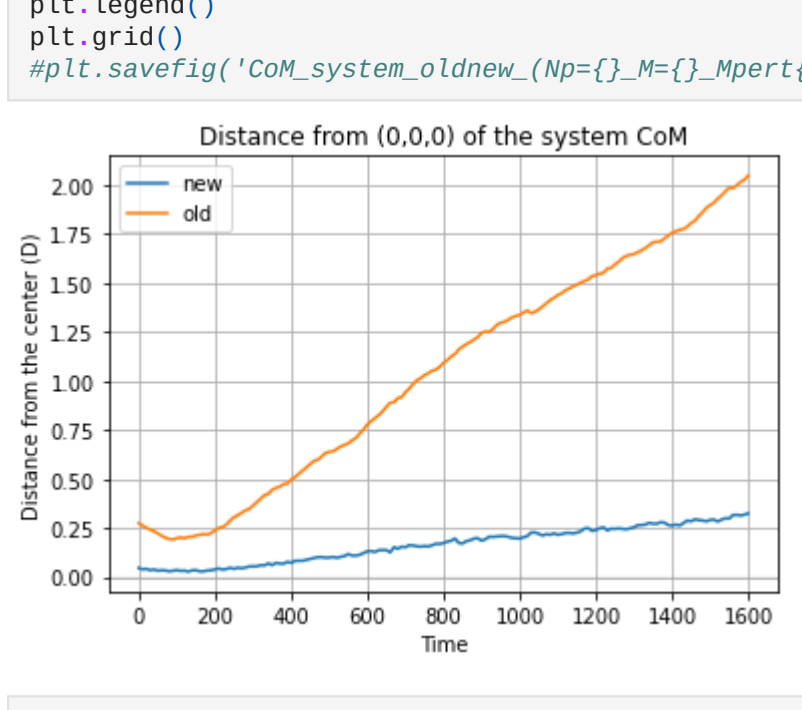
In [40]:
```
## Computation of the new position arrays
for i_snapshot in range(len(selected_snapshot)):
    x[i_snapshot] = x[i_snapshot]-CoM_x[i_snapshot]
    y[i_snapshot] = y[i_snapshot]-CoM_y[i_snapshot]
    z[i_snapshot] = z[i_snapshot]-CoM_z[i_snapshot]
dist = pow(x**2+y**2+z**2,0.5)
```

In [41]:
```
## Perturber position and distance respect to the center of mass of the system at each selected snapshot

# n.b. x_pert etc. already defined above
for i_snapshot in range(len(selected_snapshot)):
    x_pert[i_snapshot] = x[i_snapshot,0]
    y_pert[i_snapshot] = y[i_snapshot,0]
    z_pert[i_snapshot] = z[i_snapshot,0]
dist_pert = pow((x_pert**2+y_pert**2+z_pert**2),0.5)
```

In [42]:
```
## Definition of the new coordinates of the apocenter and pericenter of the perturber orbit
r_apocenter = 6
r_pericenter = 2
apocenter = np.array([0,r_apocenter,0])
pericenter = np.array([0,-r_pericenter,0])

apocenter = apocenter - np.array([CoM_x[0],CoM_y[0],CoM_z[0]])
pericenter = pericenter - np.array([CoM_x[0],CoM_y[0],CoM_z[0]])
```

In [43]:
```
## Computation of the new center of mass

LagRad90 = np.empty([len(selected_snapshot)])
sorted_distance = np.sort(np.delete(dist,0,1)) # the disturber distance is not considered
for i_snapshot in range(len(selected_snapshot)):
    LagRad90[i_snapshot] = sorted_distance[i_snapshot][int(0.9*Nparticles)-1]

R = LagRad90

CoM_x,CoM_y,CoM_z,dist_CoM = CoM(len(selected_snapshot),Nbodies,R,x,y,z,dist,mass_bodies)
del R
```

In [44]:
```
plt.plot(t,dist_CoM,label='new')
plt.plot(t,dist_CoM_old,label='old')

plt.axhline(0,0,1,c='k',ls='-',lw=0.1)

plt.title('Distance from (0,0,0) of the system CoM')
plt.xlabel('Time')
plt.ylabel('Distance from the center (D)')

plt.legend()
plt.grid()
#plt.savefig('CoM_system_oldnew_(Np={}_M={}_Mpert{}_Rscal={}).jpg'.format(Nparticles,M,M_pert,r_scale))
```



In [45]:
```
##########################################################################################################
```

In [46]:
```
## Section 6 ##

# Study of the evolution of the system (Plummer Sphere + Perturber) properties:
# - Energy
# - Lagrangian Radius (perturber not included)
# - Perturber distance and velocity evolution
# - Angular momentum (in function of time and R)
# - Density evolution
# - Radial Distribution of particles
```

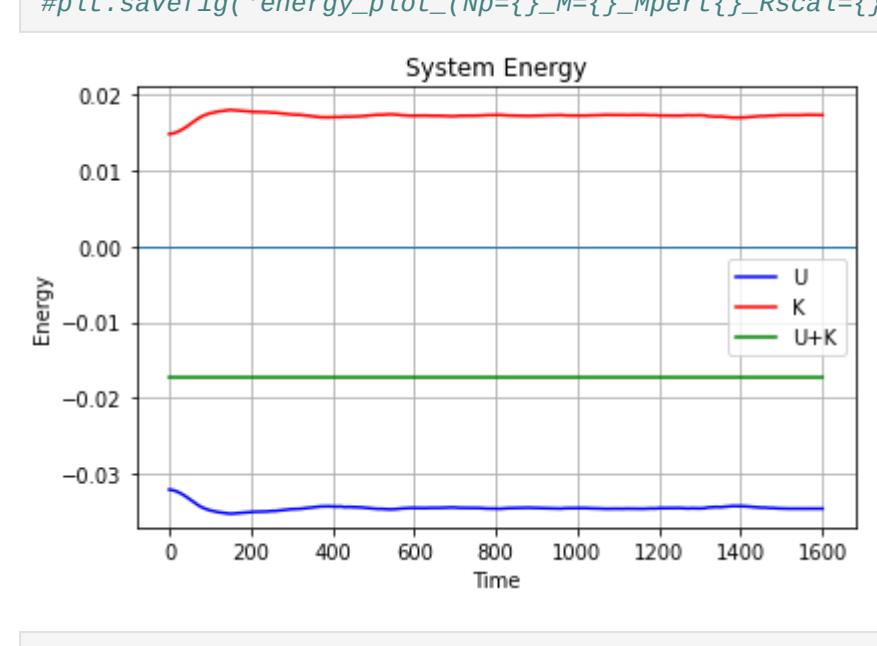
In [47]:
```
#### o ####
```

In [48]:
```
## Energy
```

In [49]:
```
potential_energy = np.empty([len(selected_snapshot),Nbodies])

initial_line_potential = 3+3*Nbodies
for i_snapshot in range(len(selected_snapshot)):
    file_name = data_files[selected_snapshot[i_snapshot]]
    potential_energy[i_snapshot] = np.genfromtxt(file_name,skip_header=initial_line_potential,
                                                  max_rows=Nbodies)
```

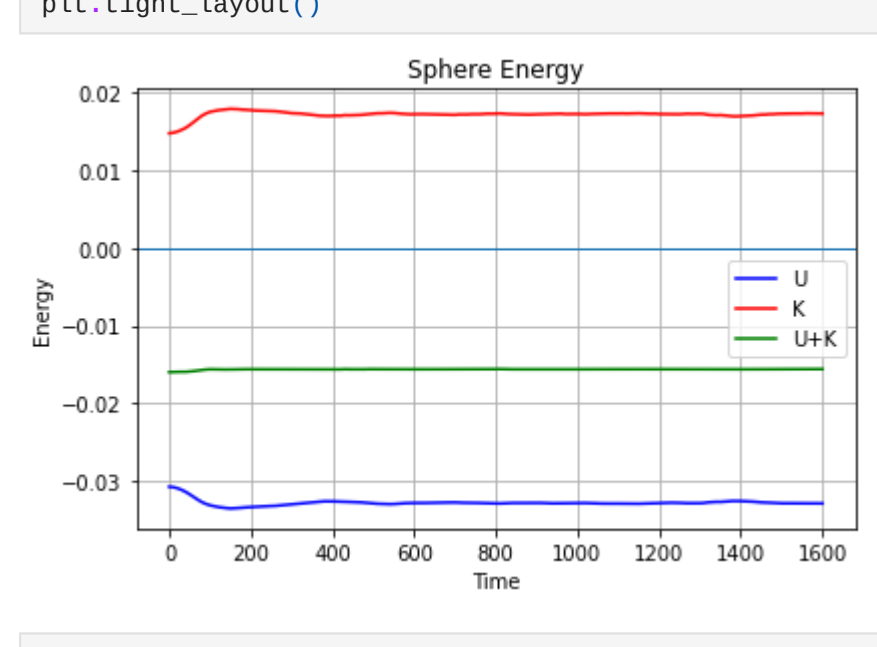
In [50]:
```
## Potential and Kinet energy of the system
Utot = np.sum(mass_bodies*potential_energy,1)*0.5 # to avoid double counting
Ktot = np.sum((vx**2+vy**2+vz**2)*mass_bodies,1)*0.5
Etot = Utot+Ktot
```

In [51]:
```
## Plot of the potential energy (U), kinetic energy(K), total energy(U+K)
plt.plot(t,Utot,c='b',label='U')
plt.plot(t,Ktot,c='r',label='K')
plt.plot(t,Etot,c='g',label='U+K')
plt.ylim([-0.037,0.021])

plt.axhline(0,0,1,lw=1)

plt.title('System Energy')
plt.xlabel('Time')
plt.ylabel('Energy')

plt.legend(loc='best')
plt.grid()
plt.tight_layout()
#plt.savefig('energy_plot_(Np={}_M={}_Mpert{}_Rscal={}).jpg'.format(Nparticles,M,M_pert,r_scale))
```



In [52]:
```
## Potential and Kinetic energy of the sphere
Utot_sphere = np.sum(mass_bodies[1:]*np.delete(potential_energy,0,1),1)*0.5 # to avoid double counting
Ktot_sphere = np.sum((np.delete(vx,0,1)**2+np.delete(vy,0,1)**2+np.delete(vz,0,1)**2)
                     *mass_bodies[1:],1)*0.5
Etot_sphere = Utot_sphere+Ktot_sphere
```

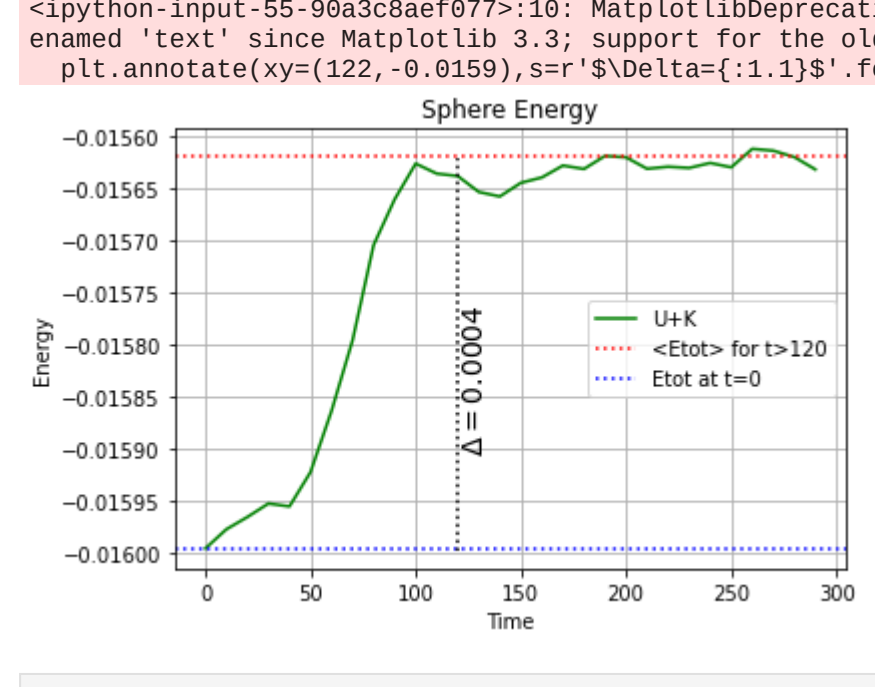
In [53]:
```
## Plot of the potential energy (U), kinetic energy(K), total energy(U+K)
plt.plot(t,Utot_sphere,c='b',label='U')
plt.plot(t,Ktot_sphere,c='r',label='K')
plt.plot(t,Etot_sphere,c='g',label='U+K')

plt.axhline(0,0,1,lw=1)

plt.title('Sphere Energy')
plt.xlabel('Time')
plt.ylabel('Energy')

plt.grid()
plt.legend(loc='best')
plt.tight_layout()
```



In [54]:
```
## The two plots below are made only dor M_pert=0.03
```

In [55]:
```
## Plot of the evolution of the sphere's total energy

mean_sphere = np.mean(Etot_sphere[t>=120])
deltaE_sphere = mean_sphere-Etot_sphere[0]

plt.plot(t[t<=300],Etot_sphere[t<=300],c='g',label='U+K')
plt.axhline(mean_sphere,0,1,ls=':',c='r',label='<Etot> for t>120')
plt.axhline(Etot_sphere[0],0,1,ls=':',c='b',label='Etot at t=0')
plt.axvline(120,0.04,mean_sphere+0.95,ls=':',c='k')
plt.annotate(xy=(122,-0.0159),s=r'$\Delta={:1.1}$'.format(deltaE_sphere),rotation=90,fontsize=14)

plt.title('Sphere Energy')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.grid()
plt.legend(loc='best')
plt.tight_layout()
#plt.savefig('Sphere_energy_plot_(Np={}_M={}_Mpert{}_Rscal={}).jpg'.format(Nparticles,M,M_pert,r_scale))
```

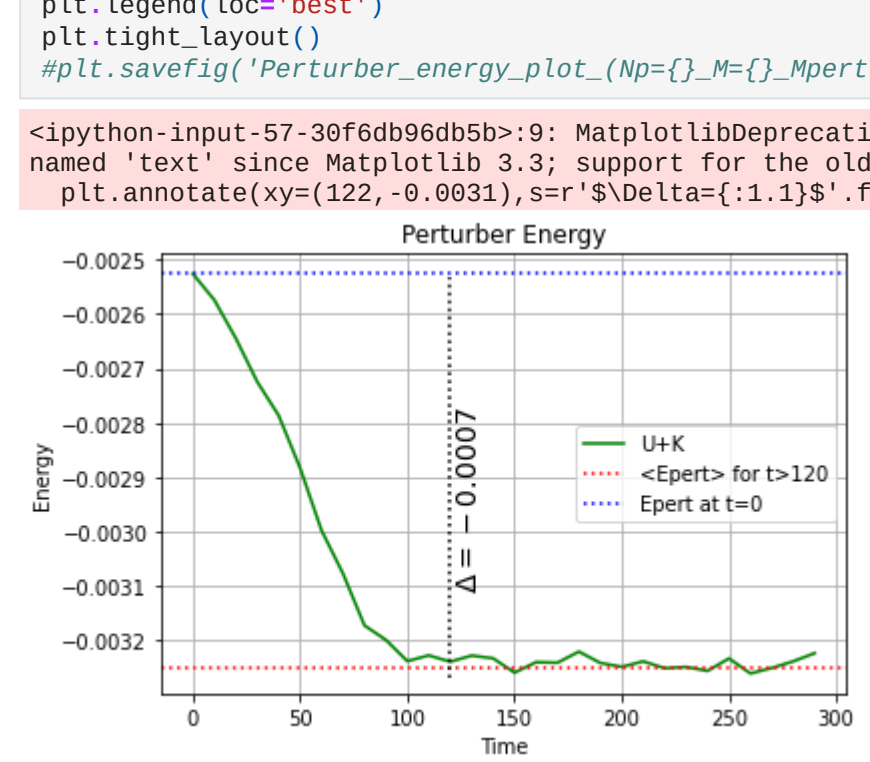
```
<ipython-input-55-90a3c8aef077>:10: MatplotlibDeprecationWarning: The 's' parameter of annotate() has been r
enamed 'text' since Matplotlib 3.3; support for the old name will be dropped two minor releases later.
  plt.annotate(xy=(122,-0.0159),s=r'$\Delta={:1.1}$'.format(deltaE_sphere),rotation=90,fontsize=14)
```



In [56]:
```
## Kinetic and Potential energy of the perturber
U_pert = np.empty([len(selected_snapshot)])
for i in range(len(selected_snapshot)):
    U_pert[i] = potential_energy[i][0]
U_pert = U_pert*M_pert

K_pert = 0.5*M_pert*(vx_pert**2+vy_pert**2+vz_pert**2)
Etot_pert = U_pert+K_pert
```

In [57]:
```
## Plot of the evolution of the perturber's total energy
mean_pert = np.mean(Etot_pert[t>=120])
deltaE_pert = mean_pert-Etot_pert[0]

plt.plot(t[t<=300],Etot_pert[t<=300],c='g',label='U+K')
plt.axhline(mean_pert,0,1,ls=':',c='r',label='<Epert> for t>120')
plt.axhline(Etot_pert[0],0,1,ls=':',c='b',label='Epert at t=0')
plt.axvline(120,0.035,mean_pert+0.95,ls=':',c='k')
plt.annotate(xy=(122,-0.0031),s=r'$\Delta={:1.1}$'.format(deltaE_pert),rotation=90,fontsize=14)

plt.title('Perturber Energy')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.grid()
plt.legend(loc='best')
plt.tight_layout()
#plt.savefig('Perturber_energy_plot_(Np={}_M={}_Mpert{}_Rscal={}).jpg'.format(Nparticles,M,M_pert,r_scale))
```

```
<ipython-input-57-30f6db96db5b>:9: MatplotlibDeprecationWarning: The 's' parameter of annotate() has been re
named 'text' since Matplotlib 3.3; support for the old name will be dropped two minor releases later.
  plt.annotate(xy=(122,-0.0031),s=r'$\Delta={:1.1}$'.format(deltaE_pert),rotation=90,fontsize=14)
```

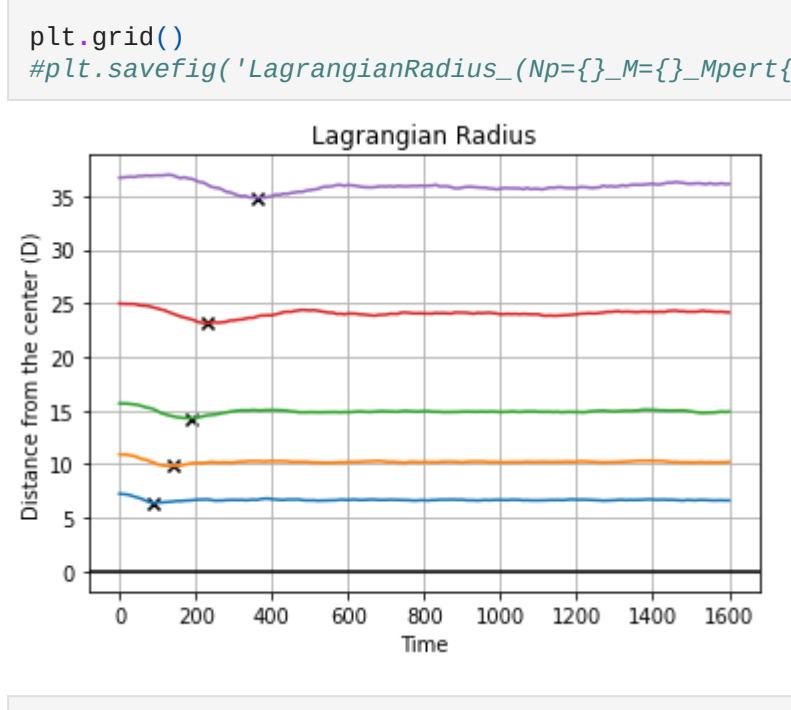


In [58]:
```
#### o ####
```

In [59]:
```
## Lagrangian Radius of the sphere (we do not consider the perturber)
```

In [60]:
```
p = [20,40,60,80,90] # lagrangian radius that contain that percentage of mass

# definition of the lagrangian radius
for perc in p:
     globals()['LagRad'+str(perc)] = np.empty([len(selected_snapshot)])

sorted_distance = np.sort(np.delete(dist,0,1)) # the disturber distance is not considered

# filling of the lagrangian radius
for i_snapshot in range(len(selected_snapshot)):
    for perc in p:
        (globals()['LagRad'+str(perc)])[i_snapshot] = (sorted_distance
                                                       [i_snapshot][int(perc/100*Nparticles)-1])

# Since we observe a contraction of the sphere, we want to find for each LagRad the minimum distance from th
# center and when it happens

for perc in p:
    globals()['Lag'+str(perc)+'Min'] = []
    globals()['Lag'+str(perc)+'Min'].append(np.min(globals()['LagRad'+str(perc)]))
    index = globals()['LagRad'+str(perc)]==globals()['Lag'+str(perc)+'Min']
    globals()['Lag'+str(perc)+'Min'].insert(0,(t[index])[0])
```

In [61]:
```
## Plot of the Lagrangian radius
for perc in p:
    plt.plot(t,globals()['LagRad'+str(perc)],label='{}%'.format(perc))
    a = (globals()['Lag'+str(perc)+'Min'])[0]
    b = (globals()['Lag'+str(perc)+'Min'])[1]
    plt.scatter(a,b,marker='x',color='k')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('Lagrangian Radius')
plt.xlabel('Time')
plt.ylabel('Distance from the center (D)')

plt.grid()
#plt.savefig('LagrangianRadius_(Np={}_M={}_Mpert{}_Rscal={}).jpg'.format(Nparticles,M,M_pert,r_scale))
```



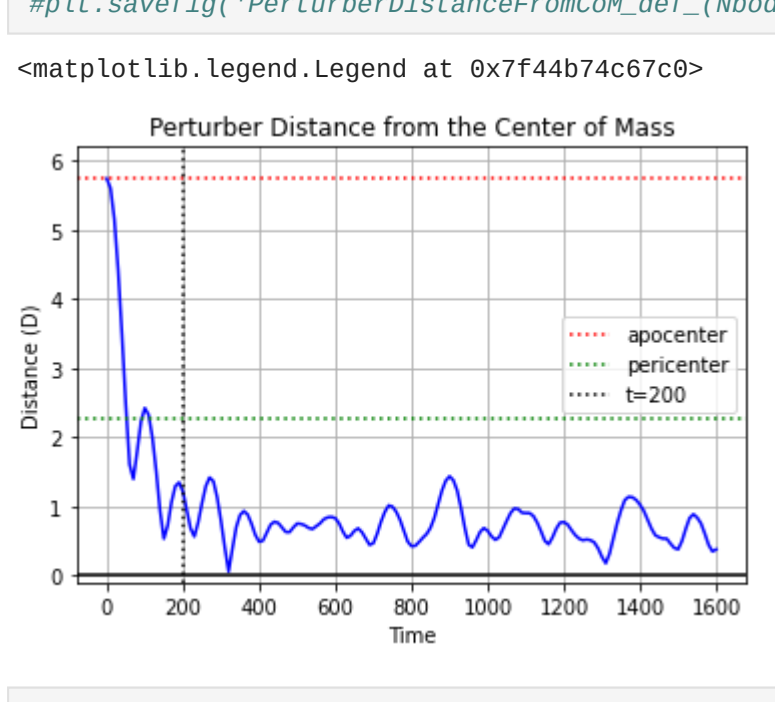
In [62]:
```
#### o ####
```

In [63]:
```
## Perturber distance respect to the CoM
```

In [64]:
```
dist_apo = pow(np.sum((apocenter**2)),0.5)
dist_per = pow(np.sum((pericenter**2)),0.5)
# The perturber orbit, without dynamical friction, will be within [dist_pericenter,dist_apocenter]
```

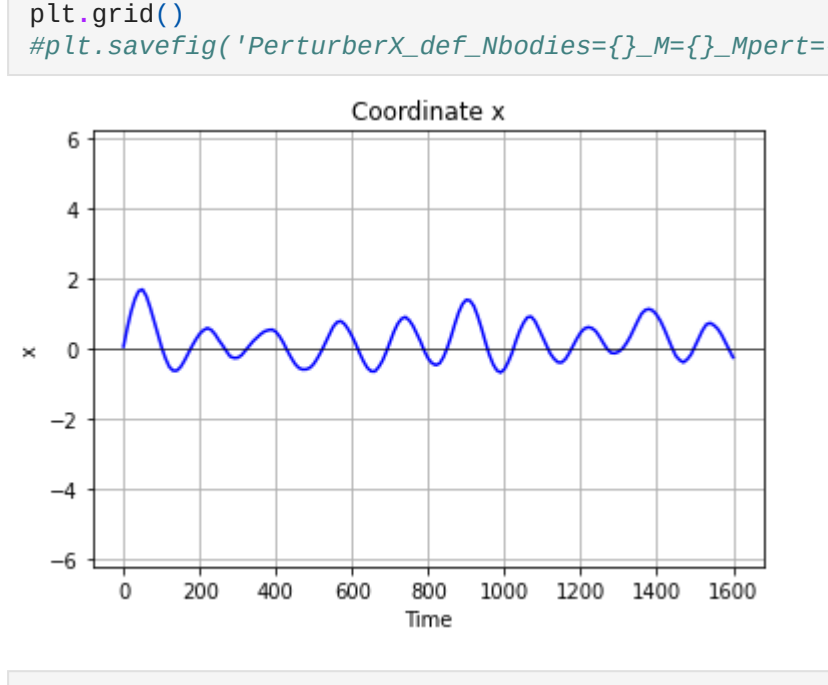
In [65]:
```
plt.plot(t,dist_pert,color='b')
plt.ylim([-0.1,6.2])

plt.axhline(0,0,1,c='k',ls='-')
plt.axhline(dist_apo,0,1,c='r',ls=':',label='apocenter')
plt.axhline(dist_per,0,1,c='g',ls=':',label='pericenter')
#plt.axvline(500,0,1,c='k',ls=':',label='t=500') # for M_pert=0.01
plt.axvline(200,0,1,c='k',ls=':',label='t=200') # for M_pert=0.03


plt.title('Perturber Distance from the Center of Mass')
plt.xlabel('Time')
plt.ylabel('Distance (D)')


plt.grid()
plt.legend()

#plt.savefig('PerturberDistanceFromCoM_def_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,
```

Out[65]:
```
<matplotlib.legend.Legend at 0x7f44b74c67c0>
```

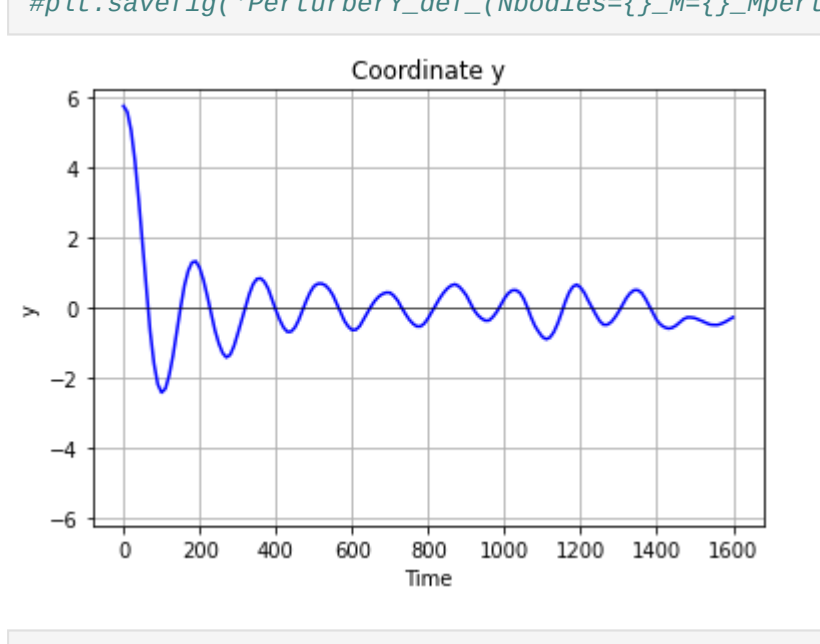


In [66]:
```
plt.plot(t,x_pert,c='b')
plt.ylim([-6.2,6.2])

plt.axhline(0,0,1,lw=0.5,c='k')

plt.title('Coordinate x')
plt.xlabel('Time')
plt.ylabel('x')
```

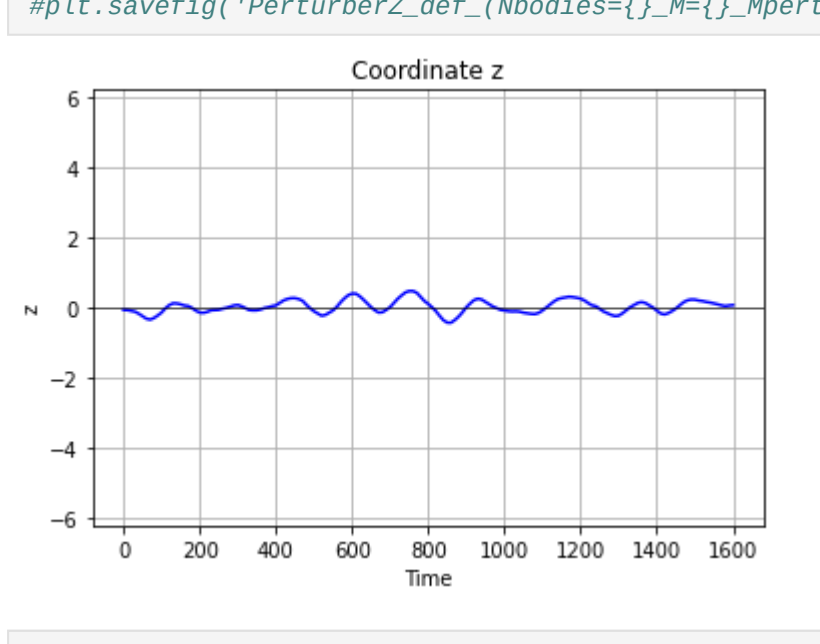
```
plt.grid()
#plt.savefig('PerturberX_def_Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [67]:
```
plt.plot(t,y_pert,c='b')
plt.ylim([-6.2,6.2])

plt.axhline(0,0,1,lw='0.5',c='k')

plt.title('Coordinate y')
plt.xlabel('Time')
plt.ylabel('y')

plt.grid()
#plt.savefig('PerturberY_def_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



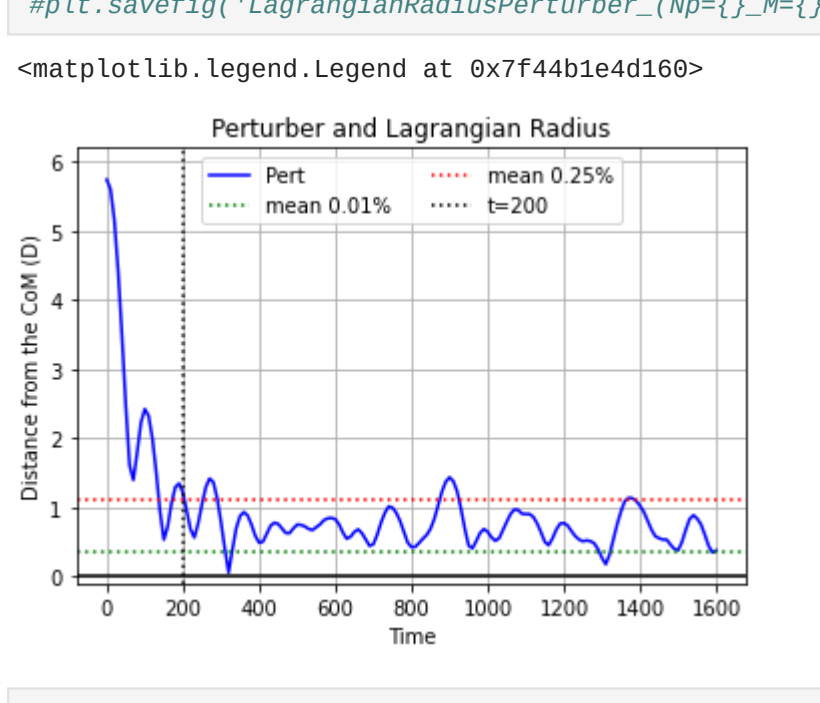
In [68]:
```
plt.plot(t,z_pert,c='b')

plt.axhline(0,0,1,lw='0.5',c='k')

plt.ylim([-6.2,6.2])

plt.title('Coordinate z')
plt.xlabel('Time')
plt.ylabel('z')

plt.grid()
#plt.savefig('PerturberZ_def_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [69]:
```
## Finding the Lagrangian Radius that include the final orbit of the perturber

# lagrangian radius that contain that percentage of mass
#p = [0.03,0.9] # for M_pert=0.01
p = [0.01,0.25] # for M_pert=0.03

# definition of the lagrangian radius
for perc in p:
     globals()['LagRad'+str(perc)] = np.empty([len(selected_snapshot)])

sorted_distance = np.sort(np.delete(dist,0,1)) # the disturber distance is not considered

# filling of the lagrangian radius
for i_snapshot in range(len(selected_snapshot)):
    for perc in p:
        (globals()['LagRad'+str(perc)])[i_snapshot] = (sorted_distance
                                                       [i_snapshot][int(perc/100*Nparticles)-1])
```

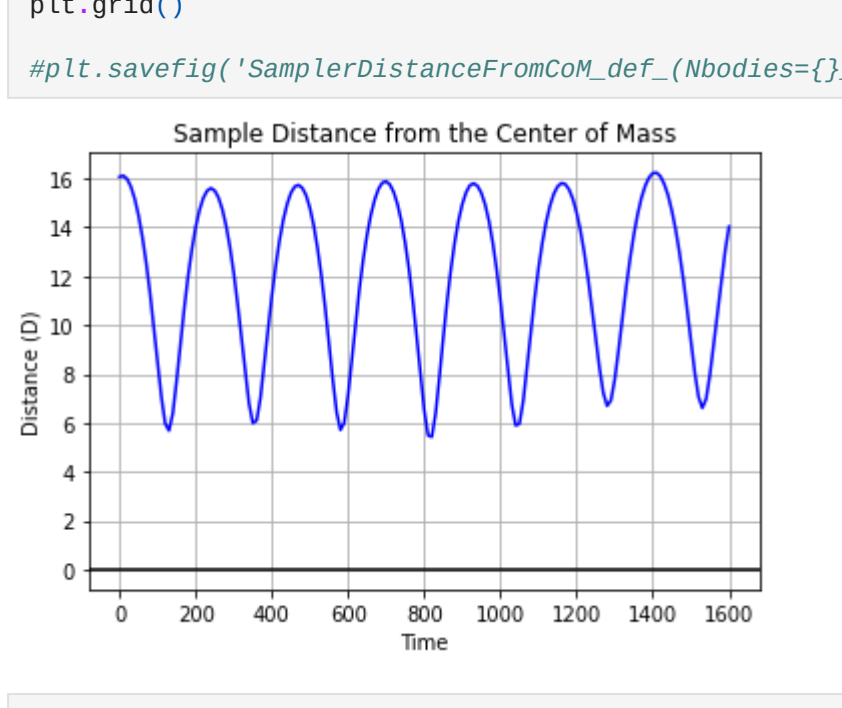
In [70]:
```
## Plot of the Lagrangian radius
plt.plot(t,dist_pert,color='b',label='Pert')
plt.ylim([-0.1,6.2])

c = []
for perc in p:
    c.append(np.mean( (globals()['LagRad'+str(perc)])[t>200]  ))
plt.axhline(c[0],0,1,ls=':',c='g',label='mean {}%'.format(p[0]))
plt.axhline(c[1],0,1,ls=':',c='r',label='mean {}%'.format(p[1]))
plt.axhline(0,0,1,c='k',ls='-')
#plt.axvline(500,0,1,c='k',ls=':',label='t=500') # for M_pert=0.01
plt.axvline(200,0,1,c='k',ls=':',label='t=200') # for M_pert=0.03

plt.title('Perturber and Lagrangian Radius')
plt.xlabel('Time')
plt.ylabel('Distance from the CoM (D)')

plt.grid()
plt.legend(ncol=2,loc=9)

#plt.savefig('LagrangianRadiusPerturber_(Np={}_M={}_Mpert{}_Rscal={}).jpg'.format(Nparticles,M,M_pert,r_scal
```

Out[70]: <matplotlib.legend.Legend at 0x7f44b1e4d160>



In [71]:
```
## Selection of a random particle anstudy of its distance evolution over time
index = np.random.randint(0, Nparticles-1) # random selection

x_sample = np.empty([len(selected_snapshot)])
y_sample = np.empty([len(selected_snapshot)])
z_sample = np.empty([len(selected_snapshot)])
dist_sample = np.empty([len(selected_snapshot)])

for i_snapshot in range(len(selected_snapshot)):
    x_sample[i_snapshot] = x[i_snapshot,index]
    y_sample[i_snapshot] = y[i_snapshot,index]
    z_sample[i_snapshot] = z[i_snapshot,index]
dist_sample = pow((x_sample**2+y_sample**2+z_sample**2),0.5)
```

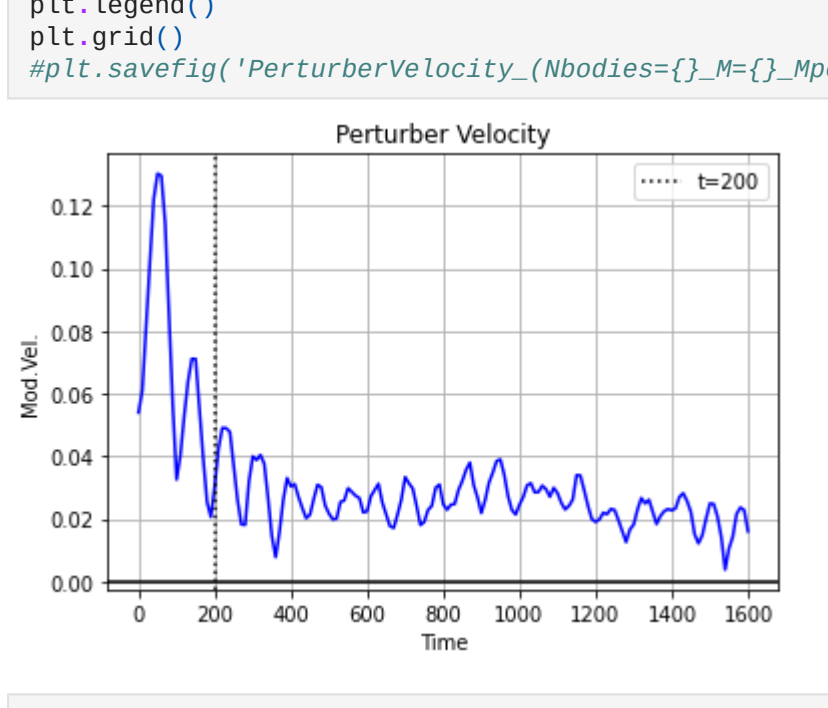
In [72]:
```
plt.plot(t,dist_sample,color='b')

plt.axhline(0,0,1,c='k',ls='-')

plt.title('Sample Distance from the Center of Mass')
plt.xlabel('Time')
plt.ylabel('Distance (D)')

plt.grid()

#plt.savefig('SamplerDistanceFromCoM_def_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_
```



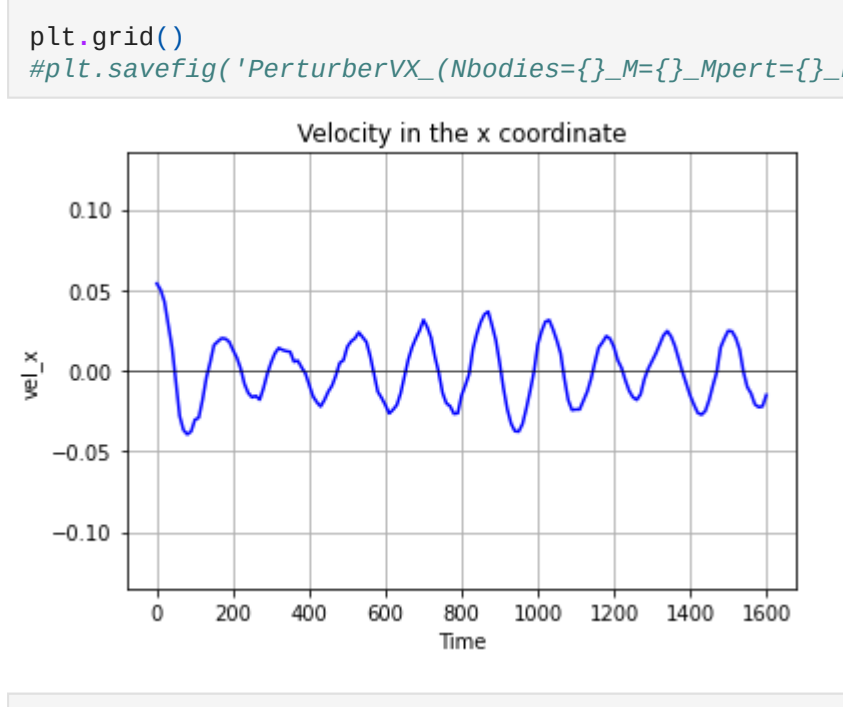
In [73]:
```
#### o ####
```

In [74]:
```
## Perturber velocity
```

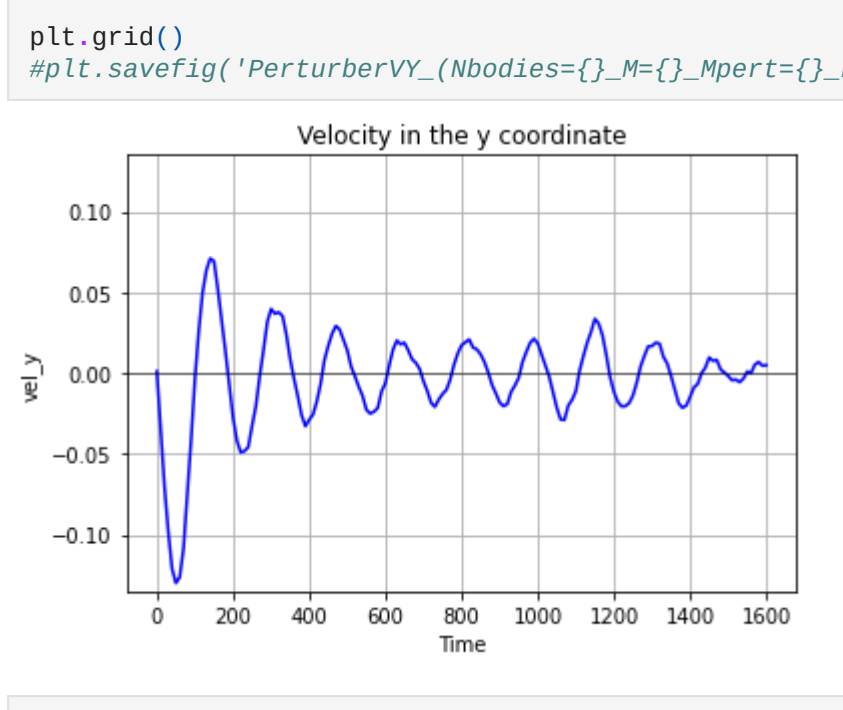
In [75]:
```
plt.plot(t,mod_Vpert,color='b')

#plt.axvline(500,0,1,c='k',ls=':',label='t=500') # for M_pert=0.01
plt.axvline(200,0,1,c='k',ls=':',label='t=200') # for M_pert=0.03
plt.axhline(0,0,1,c='k',ls='-')

plt.title('Perturber Velocity')
plt.xlabel('Time')
plt.ylabel('Mod.Vel.')

plt.legend()
plt.grid()
#plt.savefig('PerturberVelocity_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



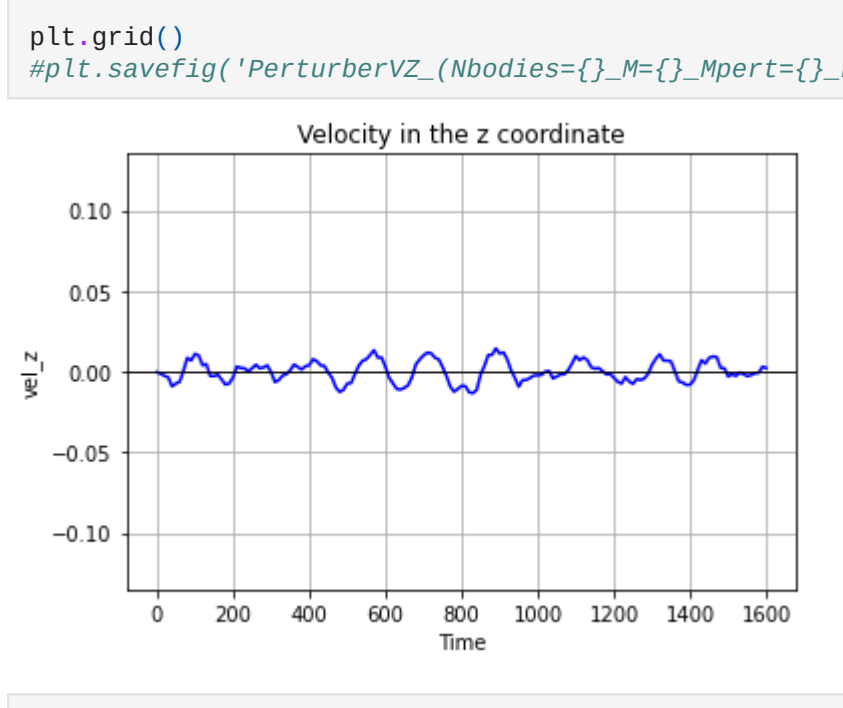
In [76]:
```
plt.plot(t,vx_pert,c='b')
plt.ylim([-0.135,0.135])

plt.axhline(0,0,1,lw=0.5,c='k')

plt.title('Velocity in the x coordinate')
plt.xlabel('Time')
plt.ylabel('vel_x')

plt.grid()
#plt.savefig('PerturberVX_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [77]:
```
plt.plot(t,vy_pert,c='b')
plt.ylim([-0.135,0.135])

plt.axhline(0,0,1,lw=0.5,c='k')

plt.title('Velocity in the y coordinate')
plt.xlabel('Time')
plt.ylabel('vel_y')

plt.grid()
#plt.savefig('PerturberVY_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [78]:
```
plt.plot(t,vz_pert,c='b')
plt.ylim([-0.135,0.135])

plt.axhline(0,0,1,c='k',ls='-',lw=0.5)
plt.axhline(0,0,1,lw=0.6,c='k')

plt.title('Velocity in the z coordinate')
plt.xlabel('Time')
plt.ylabel('vel_z')

plt.grid()
#plt.savefig('PerturberVZ_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



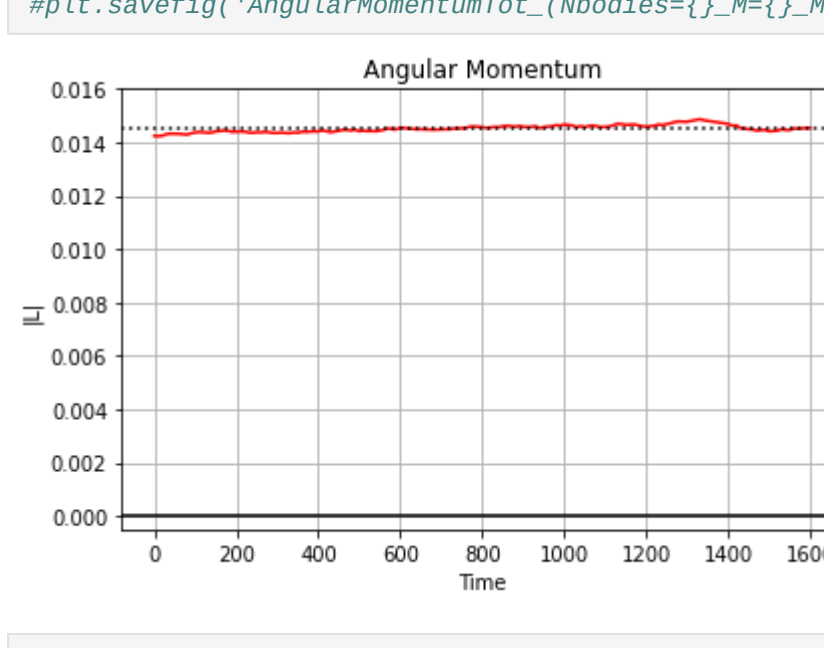
In [79]:
```
#### o ####
```

In [80]:
```
## Angular Momentum
```

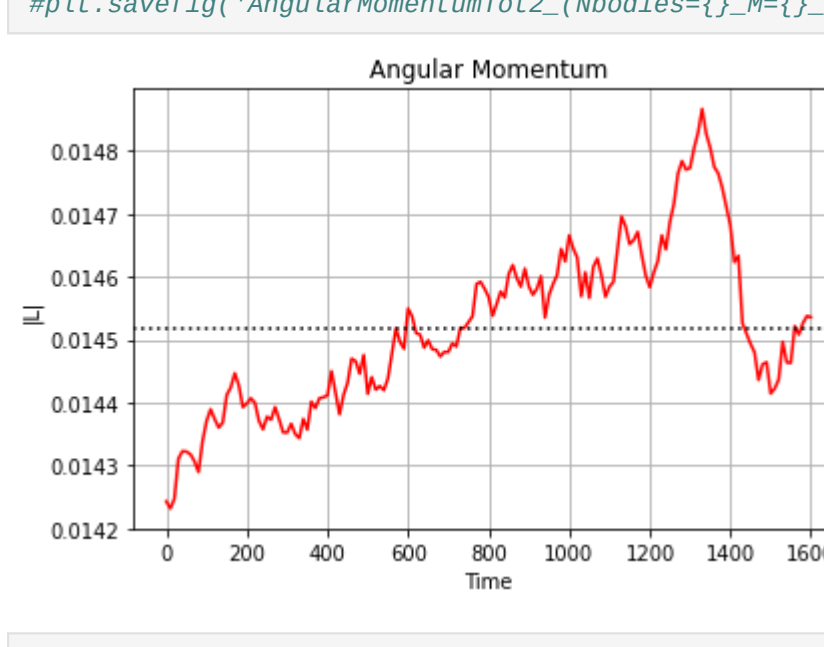
In [81]:
```
def angularMomentum(m,x,y,z,vx,vy,vz):
    Lx = (y*vz-z*vy)*m
    Ly = (z*vx-x*vz)*m
    Lz = (x*vy-y*vx)*m
    return Lx,Ly,Lz
```

In [82]:
```
## Conservation of the angular momentum of the system (all bodies considered)
# To prove the conservation of angular momentum we consider all the particles's system because some
# of them have very elliptical orbits and so they can came to small R, subtract angular momentum to
# the perturber and then continue on their elliptical orbits, returning far away from the CoM. If we
# exclude some particles, we will not take care of the angular momentum that they can have gained from
# the perturber

Lx,Ly,Lz = angularMomentum(mass_bodies,x,y,z,vx,vy,vz)
Lmod_tot = pow(np.sum(Lx,1)**2+np.sum(Ly,1)**2+np.sum(Lz,1)**2,0.5)
Lmod_mean = np.mean(Lmod_tot)
```

In [83]:
```
plt.plot(t,Lmod_tot,color='r')
plt.ylim([-0.0005,0.016])

plt.axhline(Lmod_mean,0,1,c='k',ls=':')
plt.axhline(0,0,1,c='k',ls='-')

plt.title('Angular Momentum')
plt.xlabel('Time')
plt.ylabel('|L|')

plt.tight_layout()
plt.grid()
#plt.savefig('AngularMomentumTot_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [84]:
```
plt.plot(t,Lmod_tot,color='r')

plt.axhline(Lmod_mean,0,1,c='k',ls=':')

plt.title('Angular Momentum')
plt.xlabel('Time')
plt.ylabel('|L|')

plt.tight_layout()
plt.grid()
#plt.savefig('AngularMomentumTot2_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```

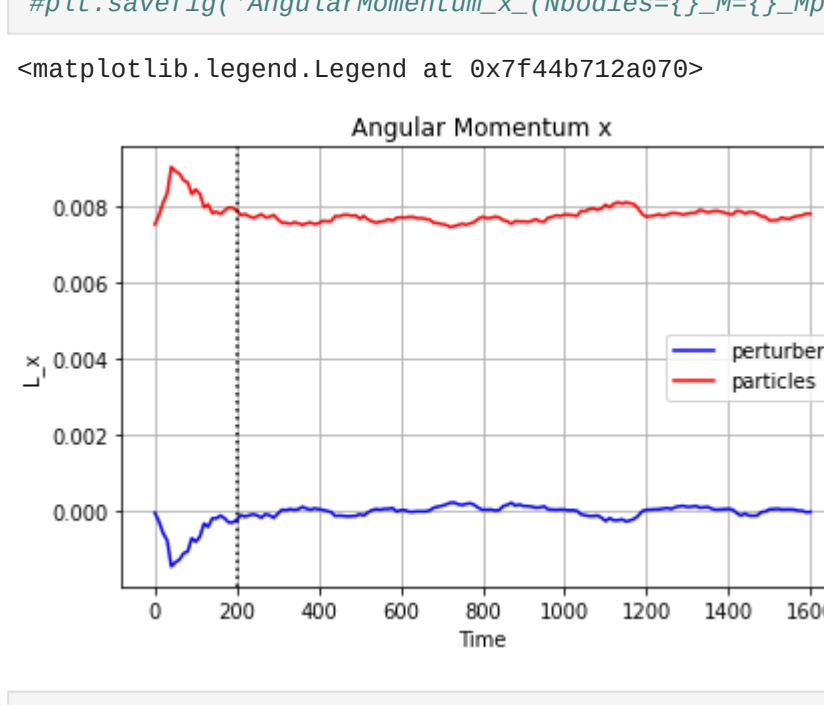


In [85]:
```
## Perturber angular momentum in time
Lx_pert,Ly_pert,Lz_pert = angularMomentum(M_pert,x_pert,y_pert,z_pert,vx_pert,vy_pert,vz_pert)
Lmod_pert = pow(Lx_pert**2+Ly_pert**2+Lz_pert**2,0.5)
```

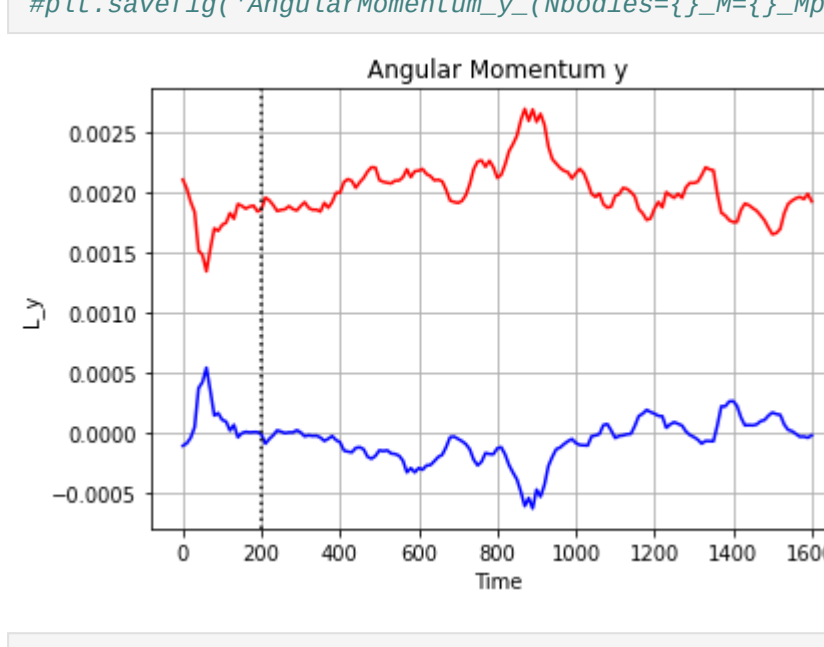
In [86]:
```
## Particles angular momentum in time
Lx,Ly,Lz = angularMomentum(mass_bodies[1:],np.delete(x,0,1),np.delete(y,0,1),np.delete(z,0,1),
                           np.delete(vx,0,1),np.delete(vy,0,1),np.delete(vz,0,1)) # we delete the perturbers
                                                                                  # data from the arrays
Lmod_tot = pow(np.sum(Lx,1)**2+np.sum(Ly,1)**2+np.sum(Lz,1)**2,0.5)
```

In [87]:
```
a = np.sum(Lx,1)
plt.plot(t,Lx_pert,color='b',label='perturber')
plt.plot(t,a,color='r',label='particles')

#plt.axvline(500,0,1,c='k',ls=':') # for M_pert=0.01
plt.axvline(200,0,1,c='k',ls=':') # for M_pert=0.03

plt.title('Angular Momentum x')
plt.xlabel('Time')
plt.ylabel('L_x')

plt.tight_layout()
plt.grid()
plt.legend()
#plt.savefig('AngularMomentum_x_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```

Out[87]: <matplotlib.legend.Legend at 0x7f44b712a070>

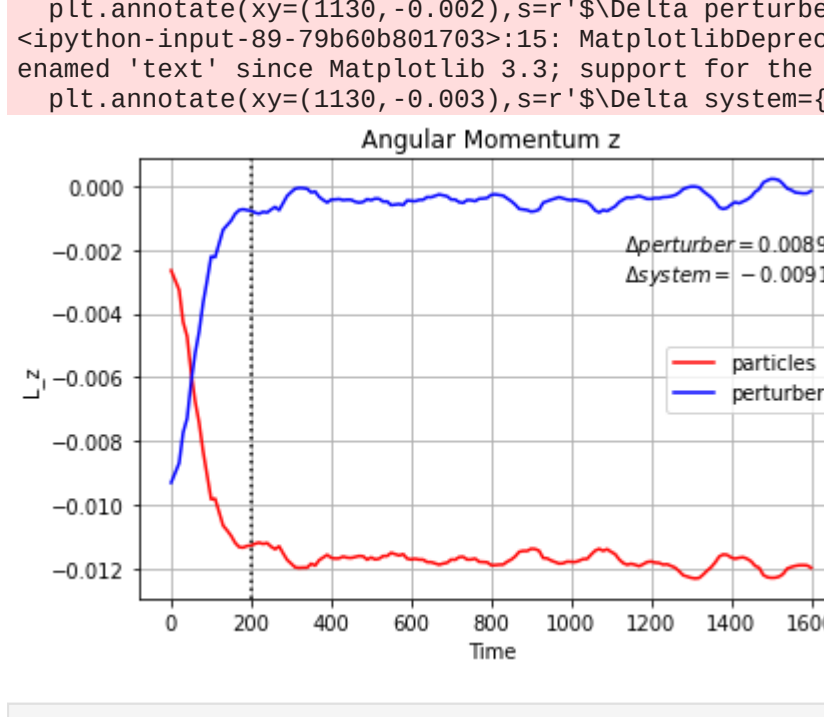


In [88]:
```
a = np.sum(Ly,1)
plt.plot(t,Ly_pert,color='b',label='perturber')
plt.plot(t,a,color='r',label='particles')

#plt.axvline(500,0,1,c='k',ls=':') # for M_pert=0.01
plt.axvline(200,0,1,c='k',ls=':') # for M_pert=0.03

plt.title('Angular Momentum y')
plt.xlabel('Time')
plt.ylabel('L_y')

plt.tight_layout()
plt.grid()
#plt.savefig('AngularMomentum_y_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```



In [89]:
```
a = np.sum(Lz,1)
plt.plot(t,a,color='r',label='particles')
plt.plot(t,Lz_pert,color='b',label='perturber')

delta_pert = np.mean(Lz_pert[t>220])-Lz_pert[0]
delta_syst = np.mean(a[t>220])-a[0]

# for M_pert=0.01
#plt.annotate(xy=(1130,-0.0019),s=r'$\Delta perturber={:1.2}$'.format(delta_pert),rotation=0,fontsize=10)
#plt.annotate(xy=(1130,-0.0024),s=r'$\Delta system={:1.2}$'.format(delta_syst),rotation=0,fontsize=10)
#plt.axvline(500,0,1,c='k',ls=':')

# for M_pert=0.03
plt.annotate(xy=(1130,-0.002),s=r'$\Delta perturber={:1.2}$'.format(delta_pert),rotation=0,fontsize=10)
plt.annotate(xy=(1130,-0.003),s=r'$\Delta system={:1.2}$'.format(delta_syst),rotation=0,fontsize=10)
plt.axvline(200,0,1,c='k',ls=':')

plt.title('Angular Momentum z')
plt.xlabel('Time')
plt.ylabel('L_z')

plt.tight_layout()
plt.legend()
plt.grid()
#plt.savefig('AngularMomentum_z_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```

```
<ipython-input-89-79b60b801703>:14: MatplotlibDeprecationWarning: The 's' parameter of annotate() has been r
enamed 'text' since Matplotlib 3.3; support for the old name will be dropped two minor releases later.
  plt.annotate(xy=(1130,-0.002),s=r'$\Delta perturber={:1.2}$'.format(delta_pert),rotation=0,fontsize=10)
<ipython-input-89-79b60b801703>:15: MatplotlibDeprecationWarning: The 's' parameter of annotate() has been r
enamed 'text' since Matplotlib 3.3; support for the old name will be dropped two minor releases later.
  plt.annotate(xy=(1130,-0.003),s=r'$\Delta system={:1.2}$'.format(delta_syst),rotation=0,fontsize=10)
```



In [90]:
```
## Angular momentum in function of R
```

In [91]:
```
def takeSecond(elem):
    return elem[1]


# a and b are joint one to one into a list which is sorted in ascending oreder respect to a
def orderAs(a,b):
    d = sorted(zip(a,b))
    f = []
    for i in range(len(d)):
        f.append( takeSecond(d[i]) )
    return np.array(f)
```

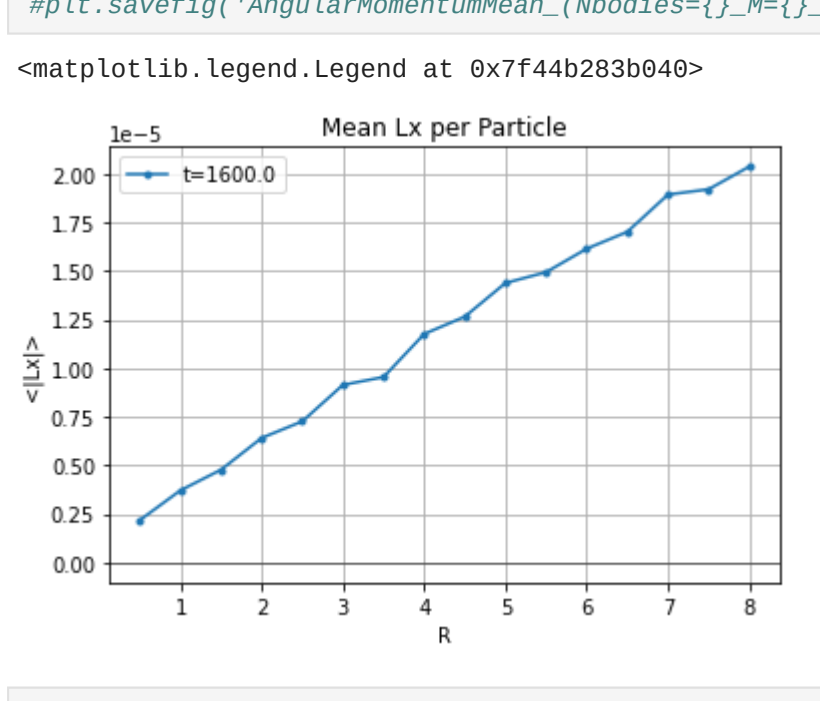
In [92]:
```
snapshot_to_check = Nsnapshots-1

# The perturber is not considered
# The arrays are ordered as the particles distance from the CoM
x_order = orderAs(dist[snapshot_to_check][1:],x[snapshot_to_check][1:])
y_order = orderAs(dist[snapshot_to_check][1:],y[snapshot_to_check][1:])
z_order = orderAs(dist[snapshot_to_check][1:],z[snapshot_to_check][1:])
vx_order = orderAs(dist[snapshot_to_check][1:],vx[snapshot_to_check][1:])
vy_order = orderAs(dist[snapshot_to_check][1:],vy[snapshot_to_check][1:])
vz_order = orderAs(dist[snapshot_to_check][1:],vz[snapshot_to_check][1:])
```

In [93]:
```
Lx,Ly,Lz = angularMomentum(mass_bodies[1:],x_order,y_order,z_order,vx_order,vy_order,vz_order)
Lmod_tot = pow(Lx**2+Ly**2+Lz**2,0.5)
```

In [94]:
```
withinR = [0,8]  #LagRad90[snapshot_to_check]*2 # al the particles within this value are considered
Npoints = 16
delta = (withinR[1]-withinR[0])/Npoints
r = [withinR[0]]
L_binX = [] # it will contain the mean angular momentum per particle in the generated bins
L_binY = []
L_binZ = []
L_binM = []
sorted_distance = np.sort(np.delete(dist,0,1))

for i in range(Npoints):
    r.append(withinR[0]+(i+1)*delta)
    boolArray = (r[i]<=sorted_distance[snapshot_to_check]) & (sorted_distance[snapshot_to_check]<=r[i+1])
    A = abs(Lx[boolArray])
    B = abs(Ly[boolArray])
    C = abs(Lz[boolArray])
    L_binX.append( np.mean(A) )
    L_binY.append( np.mean(B) )
    L_binZ.append( np.mean(C) )
    D = np.mean( (A**2+B**2+C**2)**0.5 )
    L_binM.append(D)
```

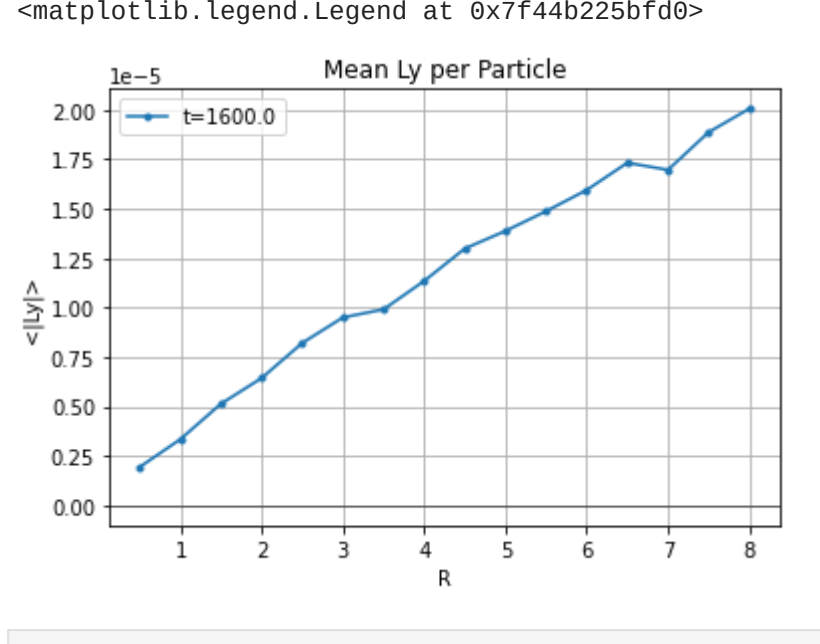
In [95]:
```
plt.plot(r[1:],L_binX,marker='.',label='t={:5.5}'.format(t[snapshot_to_check]))

plt.axhline(0,0,1,c='k',ls='-',lw=0.5)

plt.title('Mean Lx per Particle')
plt.xlabel('R')
plt.ylabel('<|Lx|>')

plt.grid()
plt.legend()
#plt.savefig('AngularMomentumMean_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```

Out[95]: <matplotlib.legend.Legend at 0x7f44b283b040>



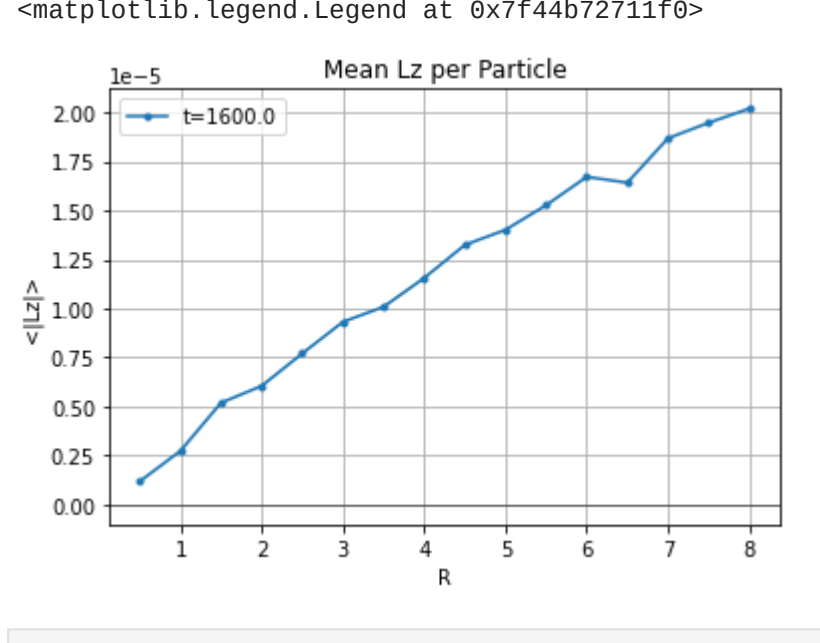
In [96]:
```
plt.plot(r[1:],L_binY,marker='.',label='t={:5.5}'.format(t[snapshot_to_check]))

plt.axhline(0,0,1,c='k',ls='-',lw=0.5)

plt.title('Mean Ly per Particle')
plt.xlabel('R')
plt.ylabel('<|Ly|>')

plt.grid()
plt.legend()
#plt.savefig('AngularMomentumMean_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```

Out[96]: <matplotlib.legend.Legend at 0x7f44b225bfd0>



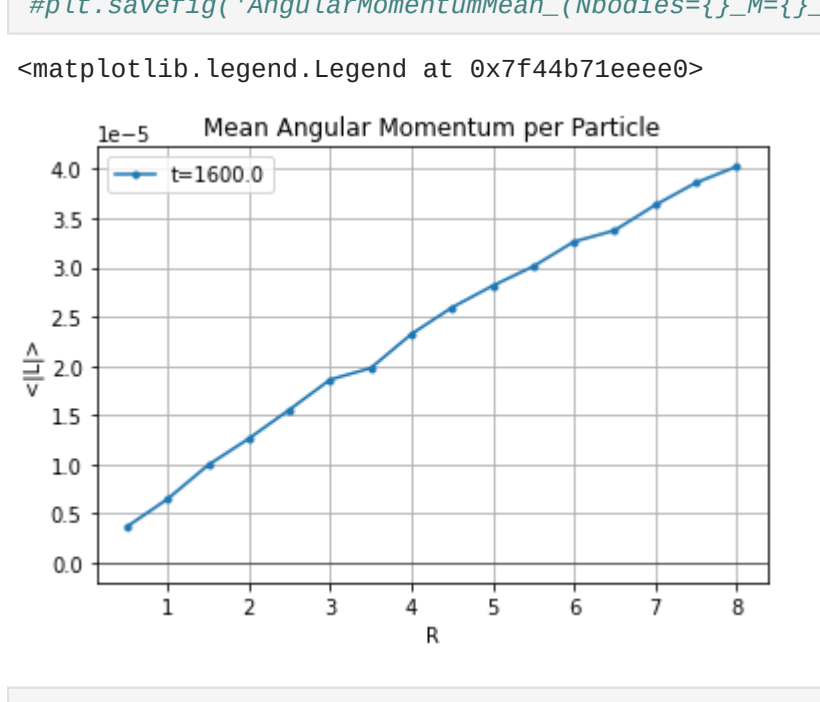
In [97]:
```
plt.plot(r[1:],L_binZ,marker='.',label='t={:5.5}'.format(t[snapshot_to_check]))

plt.axhline(0,0,1,c='k',ls='-',lw=0.5)

plt.title('Mean Lz per Particle')
plt.xlabel('R')
plt.ylabel('<|Lz|>')

plt.grid()
plt.legend()
#plt.savefig('AngularMomentumMean_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```

Out[97]: <matplotlib.legend.Legend at 0x7f44b72711f0>



In [98]:
```
plt.plot(r[1:],L_binM,marker='.',label='t={:5.5}'.format(t[snapshot_to_check]))

plt.axhline(0,0,1,c='k',ls='-',lw=0.5)

plt.title('Mean Angular Momentum per Particle')
plt.xlabel('R')
plt.ylabel('<|L|>')

plt.grid()
plt.legend()
#plt.savefig('AngularMomentumMean_(Nbodies={}_M={}_Mpert={}_Rscal={}).jpg'.format(Nbodies,M,M_pert,r_scale))
```

Out[98]: <matplotlib.legend.Legend at 0x7f44b71eeee0>



In [99]:
```
##########################################################################################################
```