N-body simulation of a Milky Way-like galaxy

Andrés García-Serra Romero[1]

[1]Master de Astrofísica, Universidad de la Laguna

November 29, 2023

Introduction

For this project we will be performing an N-body simulation from scratch using some tools in hand, a laptop, python and some initial conditions. This will be performed for a galaxy of similar characteristics as the Milky Way and for 1,10,100 and 1000 particles depending on the case.

The code will be based in taking initial conditions from given files and, using different integrating methods, perform the calculation of the position variation of each particle. To do this we will be initially taking in account the total dark matter halo mass inside each particle orbit radius approximating the radial density profile of dark matter to a NFW profile:

$$\rho(r) = \frac{\rho_0}{\frac{r}{R_s}\left(1+\frac{r}{R_s}\right)^2} \tag{1}$$

This will be easy to calculate for any method. Having the position of each particle gives us the radial distance to the center of the galaxy. The enclosed mass at a radius r is given by:

$$M = 4\pi\rho_0 R_s^3 \left[\ln\left(\frac{R_s + R_{\max}}{R_s}\right) - \left(\frac{R_s}{R_s + R_{\max}}\right)\right] \tag{2}$$

The acceleration of each particle will be then calculated as the sum of the one provoked by this central halo mass and, at the end of the project, the one provoked by the different interactions between each particle.

Integration methods

The calculation of the different positions, velocities and accelerations can be performed using different methods. The work project start by using the Euler method which is the simplest and just performs a classical approach of velocity equals the current plus acceleration times time passed:

$$y_{i+1} = y_i + h \cdot f(t_i, y_i) \tag{3}$$

On the other hand, a more precise method normally used in this case is the Runge-Kutta method, which approximates to the fourth order, using the following expressions:

$$y_{i+1} = y_i + \frac{h}{6} \cdot (K_1 + 2 \cdot K_2 + 2 \cdot K_3 + K_4) \tag{4}$$

$$\begin{cases} K_1 = f(t_i, y_i) \\ K_2 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2} \cdot K_1) \\ K_3 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2} \cdot K_2) \\ K_4 = f(t_i + h, y_i + h \cdot K_3) \end{cases} \tag{5}$$

In our case for this method we will be focusing on the spatial part, not on the time. And will be calculating the different velocities and accelerations with the method for each time step.

Code and initial conditions

The code is written in `Python3.8` using `VIM`, and is presented in the different appendixes (see 4,4,4). It is a completely modular code composed of a particle class variable definition holding the mass of each particle and the three spatial components of position and velocity, as well as a universe class variable which is composed by a set of particles and can be called using the different systems (sun only, 10, 100 or 1000 particles) and the different integrating methods (Runge-Kutta or Euler). I really encourage the reader to see the code structure because it has a lot of time in the making.

The modularity and different class definitions of the code leads us with a very straight-forward data analysis in which the different pipelines for the exercises are just a couple of lines calling functions inside the `classes.py`

and plotting the results.

About the initial conditions, it is important to mention that the sets of particles have a total mass of the Milky Way (e.g. for 10 particles each has $10^9 M_\odot$), this means each particle is not gonna be a real star, but from now on we will call them particles or stars as if they were, they are really just representing a group of stars to . This files hold each particle mass and initial position and velocity components.

Softening length

In the case of particles interacting with each other we can be working on very small distances, this will spike up the gravitational acceleration produced by their interaction, giving us asymptotic trajectories. Thus, some particles get very close to others, having a huge acceleration which expels them from the system.

To avoid this effect the Softening Length is introduced, which is simply an scaling distance added to the distance between particles calculation that sets a minimum distance over which interactions are taking in account.

1 Exercise 1

For the first part of the project we will be simply performing a simulation of a Sun-like particle orbiting around the center of the galaxy at 8 kpc. For this simple system we can just calculate the enclosed mass of dark matter using equation 2 explained above and applying an acceleration like the mass was all allocated in the center of the galaxy, approximating the problem to a gravitational Newtonian potential with spherical symmetry. (the code of this section is in appendix B: 4)

The trajectories for the different integration methods are presented in figures 1 and 2. The time of the integration is 18.6×10^8 years and there are 1000 snapshots for that time.

As we can see the Euler method tends to a spiral as the time passes (the figure is not very clear as the two orbits are one on top of another) while the Runge-Kutta method stays circular during all the chosen period.

2 Exercise 2

In the second section of the project we will be doing the same process as in the first one but loading initial conditions for 10, 100 and 1000 given particles. We will

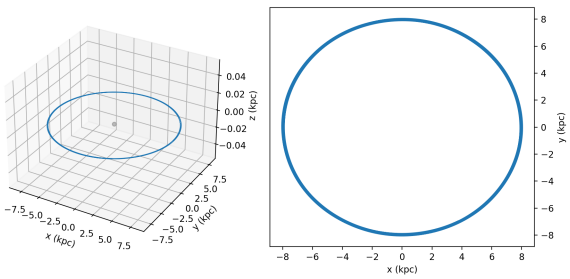


Figure 1: Trajectory of a Sun-like particle orbiting the center of the galaxy following a NFW profile potential. Integrated using Euler method.

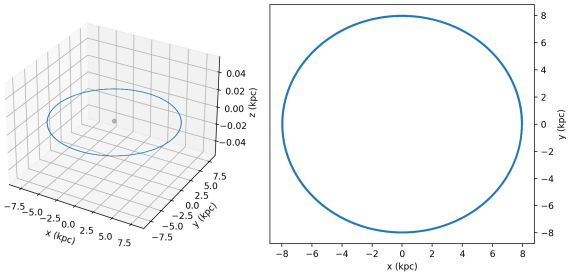


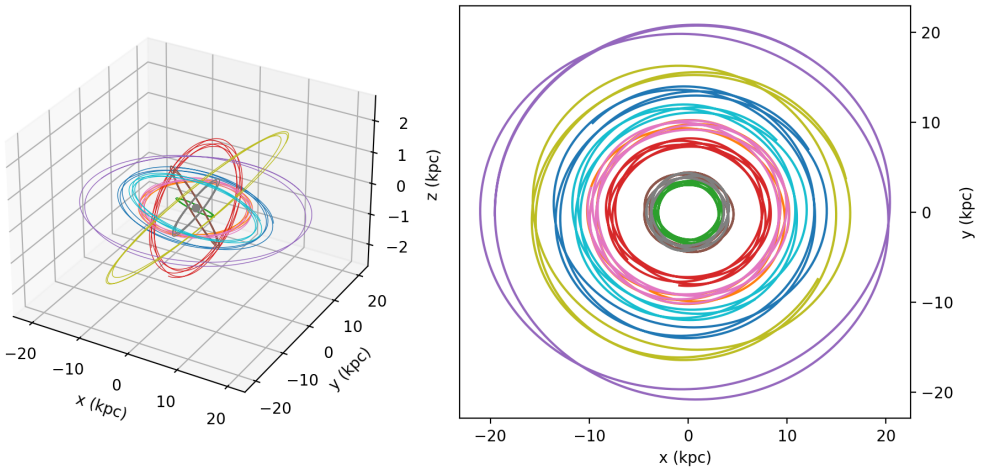
Figure 2: Trajectory of a Sun-like particle orbiting the center of the galaxy following a NFW profile potential. Integrated using Runge-Kutta method of order 4.

be taking in account the total enclosed dark matter mass inside the orbit for each particle and then we will be using Euler and Runge-Kutta again to integrate the different trajectories. (the code of this section is in appendix B: 4)

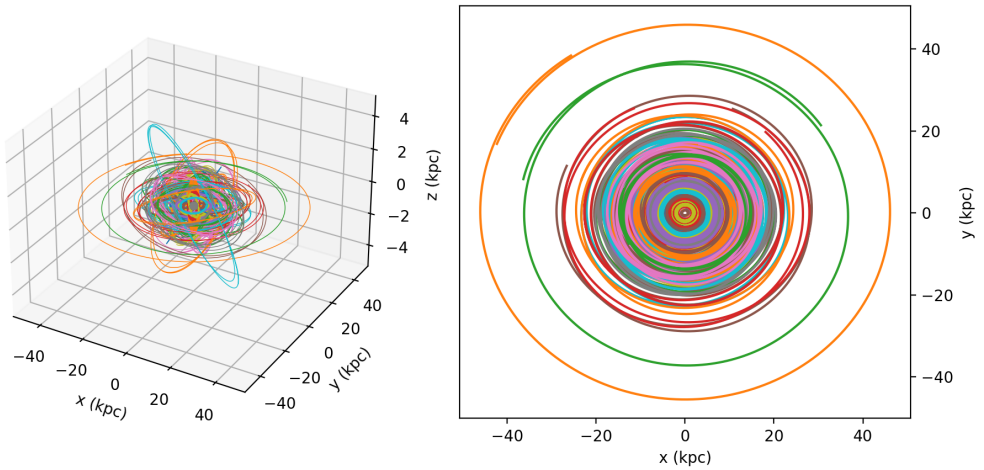
Total integration time and time steps are the same as before. The different subfigures in figure 3 present the cases for 10 (a), 100(b) and 1000(c) particles.

As we can see the particles stay in a circular orbit mostly. In fact if we take a look at the scales of the three dimensional plots, the disk of rotation is about 20 kpc in radius and only between 1 and 2 kpc in height. This tells us the rotation remains disk-like.

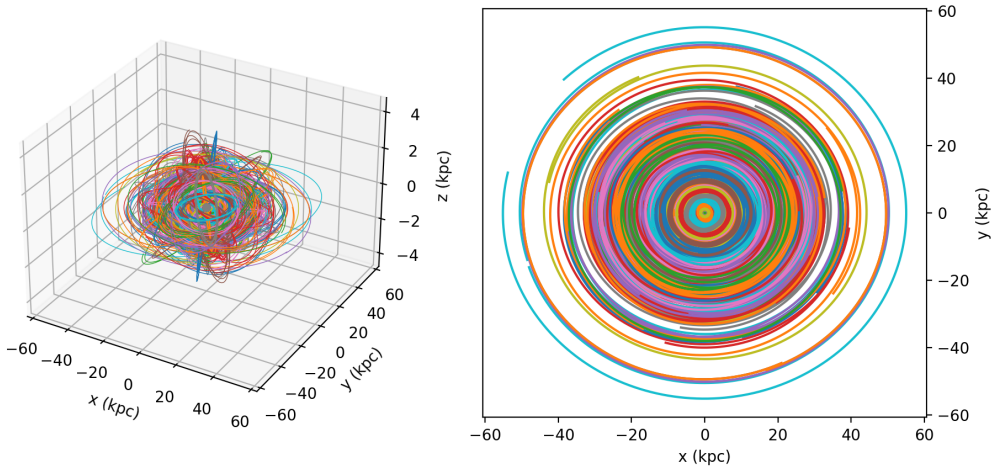
The plots for the Runge-Kutta method are presented in figure 4. The results are very similar to in Euler, almost identical. We can see that for the same scale maybe Runge-Kutta gives a more circular orbit, but in general the trajectories are almost the same.



(a)

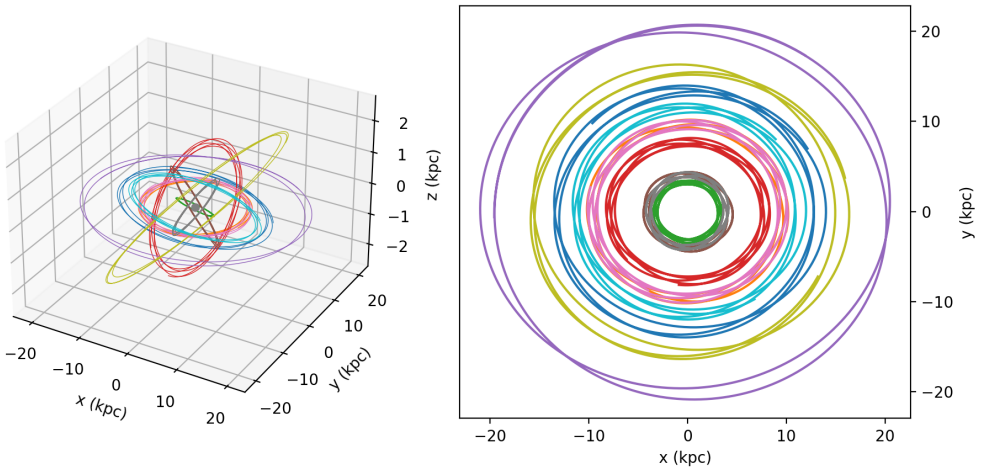


(b)

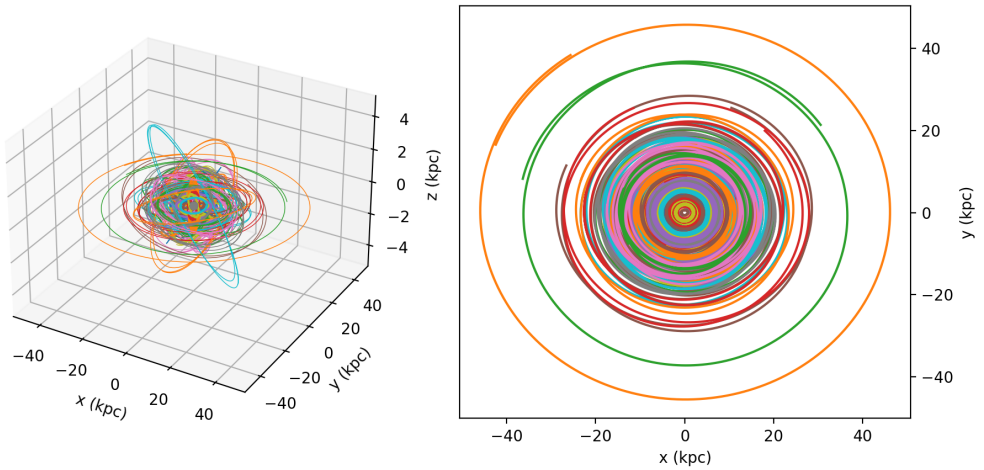


(c)

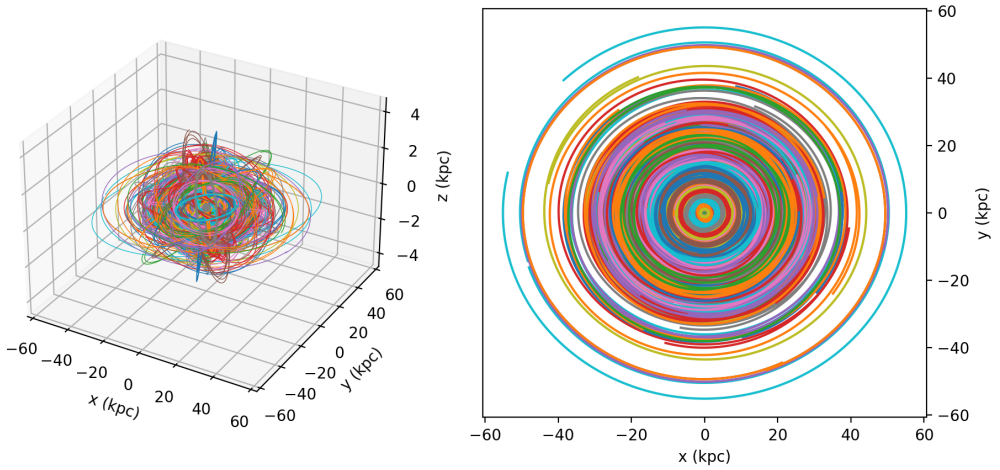
Figure 3: Orbital integration of 10 (a), 100 (b), and 1000 (c) particles using the Euler method and no interaction between them.



(a)



(b)



(c)

Figure 4: Orbital integration of 10 (a), 100 (b), and 1000 (c) particles using the Runge-Kutta method and no interaction between them.

2.1 Computation time vs number of particles

We can also plot the time the system required to integrate the orbits for each number of particles. This plot is shown in figure 5.

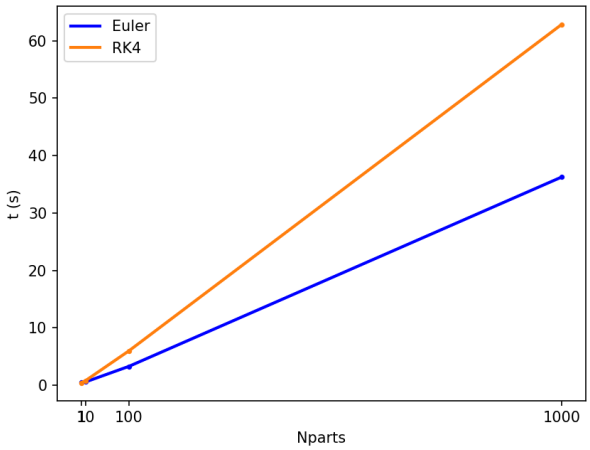


Figure 5: Computation time for each set of particles, Euler in blue, Runge-Kutta in orange. No interaction between particles applied.

As can be seen, the relation between the computation time and the number of particles involved in the simulation is linear (O(n)).

3 Exercise 3

In this last section of the project we will be performing the same simulation but adding the acceleration caused by each other particle involved. As described in the project guidelines, this mass is very relevant in the trajectory of the particles, being its close distance. As distance gets smaller, less mass is required for a particle to give similar accelerations to the ones induced by the dark matter halo of the galaxy.

To perform this we will simply add to the current acceleration of the system a component for each other particle in the system. The current code is below in appendix 4.

In this case, we will be using Runge-Kutta integration and sets of 10 and 100 particles, as 1000 particles requires too much computation time in the different test we've done.

To avoid asymptotic trajectories we will be applying a softening length of 1 kpc to our interactions. The different subfigures in figure 7 show the trajectories for 10 and 100 particles in the system de.

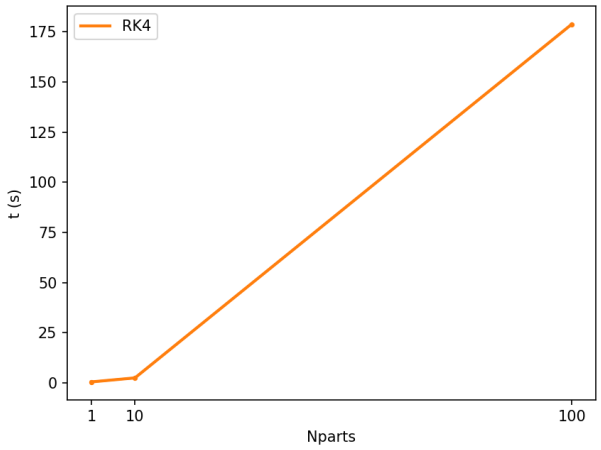


Figure 6: Computation time for each set of particles using Runge-Kutta integrating method. Interaction between particles applied.

As we can see, particles loose their almost perfect circular (elliptical) orbits and start behaving in a more chaotic way, leaving us with a more realistic system. On top of that, we can see that in this case the disk-like structure remains in scale between a radius of 20-30 kpc and a height of 4-6kpc, which is wider than before, as expected with the interactions involved.

3.1 Computation time

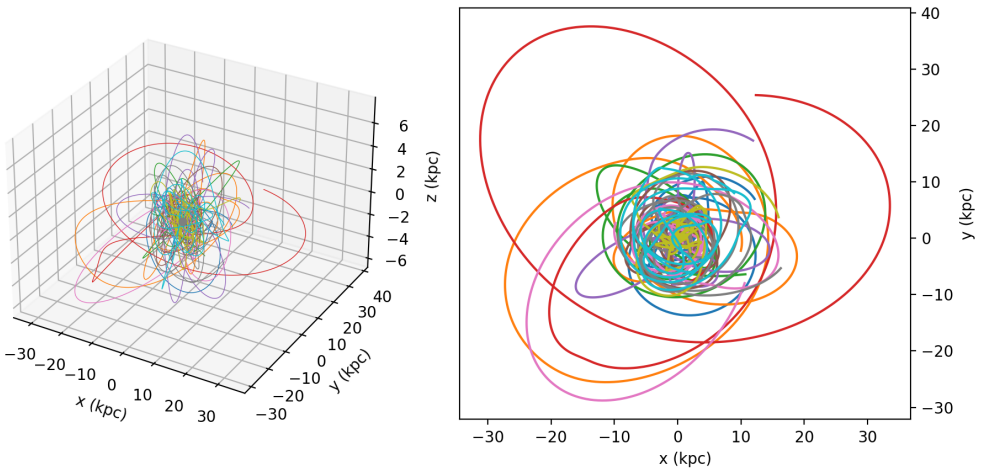
Finally, we can also take a look into the computation time, as we did for the last section. In Figure 6.

We can see that in this case the 100 particles system spikes up integration time almost by a factor of 3 the integration time it took without particle interaction in a 1000 particles system. Checking the data we can confirm this is a power law of second order behavior ($O(n^2)$).

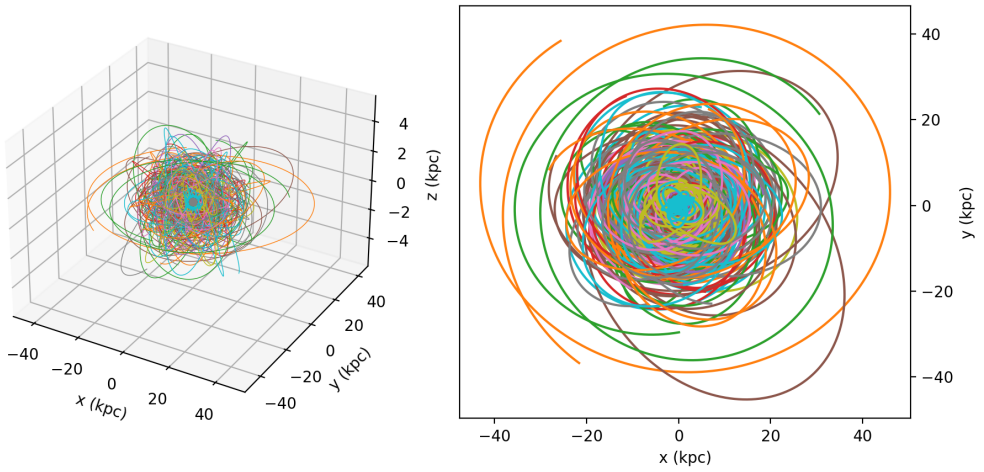
4 Conclusions

To sum up, we could have improved the code in some ways performance-wise so it could be applied to larger systems like for the one of 1000 particles in last case. Some of these improving methods are commented in the project guidelines and involve tree algorithms or just taking into account closer neighbours and computing farther neighbours as a whole gravitational potential.

In our case, using matrix nomenclature instead of vectors for positions and velocities would've been better performance wise but the realization on this came too far in the code process.



(a)



(b)

Figure 7: Orbital integration of 10 (a) and 100 (b) particles using the Runge-Kutta method and interaction between them.

Appendix A: Class definition and orbit integration code

```
#IMPORTING PACKAGES:
import os
import random
import numpy as np
import matplotlib.pyplot as plt
from scipy import pi
import h5py
import sys
import random
from matplotlib.offsetbox import TextArea, DrawingArea, OffsetImage, AnnotationBbox
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from mpl_toolkits.mplot3d import Axes3D


#General variables:
LOAD_DIR = 'ics/'
SAVEDIR = 'figures/'

#Natural to cgs convertion units:
U_T = 3600*24*365 #yrs to s
U_VEL = 1e5 #[km/s] to [cm/s]
U_DIST = 3.085678e21 #[kpc] to [cm]
U_MASS = 1.989e33 #[Msol] to [g]
G = 6.67e-8 #cgs units

#NFW parameters:
RHOo = 5932371.*U_MASS*(U_DIST**(-3))
Rs = 20.*U_DIST

TIMESTEP = 9.8e8*U_T

# Particle class:
class particle:
    m = 0
    xpos = 0
    ypos = 0
    zpos = 0
    vx = 0
    vy = 0
    vz = 0

    def __init__(self,mass,x,y,z,vx,vy,vz):
        self.m = mass
        self.xpos = x
        self.ypos = y
        self.zpos = z
        self.vx = vx
        self.vy = vy
        self.vz = vz

    def pos(self):
        return print('[x,y,z]= '+str(np.array([self.x,self.y,self.z])))

    def vel(self):
        return print('[vx,vy,vz]= '+str(np.array([self.vx,self.vy,self.vz])))



# Universe class:
class universe:
    part = np.array([])
    frames = np.array([])
    oldpart = np.array([])
    acc_int = np.array([])      # Acceleration tensor.
    tbin = 0.001*(18.6e8*U_T)   # Time bin for orbit integration.
    method = ''                 # Integration method ('Euler' for Euler and 'RK4' for Runge-Kutta)
    sys = ''                    # System type ('sun' for sun-SagA / 'loadN' to load ics from loadN.
    txt)
```

```
    inter = ''                  # Boolean for grav interaction between stars. ('yes'/'no')
    softlength = 1*U_DIST       # System softening length

    def __init__(self,method,sys,inter):          #Definition example: u = Universe('Euler','load10','yes')
        self.sys = sys
        self.method = method
        self.inter = inter

        # Creating the actual Universe:
        if self.sys=='sun':
            self.add(particle(1*U_MASS, 8*U_DIST,0,0, 0,127*U_VEL,0))
        if self.sys=='load10':
            self.load(10)
        if self.sys=='load100':
            self.load(100)
        if self.sys=='load1000':
            self.load(1000)

    def load(self,N):
        data = np.loadtxt(LOAD_DIR+'disk'+str(N)+'.txt')
        for i in range(len(data)):
            self.add(particle(data[i,0]*U_MASS,data[i,1]*U_DIST,data[i,2]*U_DIST,data[i,3]*U_DIST,data[i,4]*U_VEL,data[i,5]*U_VEL,data[i,6]*U_VEL))

    def show(self):
        print(self.part)

    def add(self,newpart):
        self.part = np.append(self.part,newpart)

    def plot_trace(self):       #Plots the n-element inside the frames vector of particles through time

        fig = plt.figure(figsize=(10,5))
        ax1 = fig.add_subplot(121, projection='3d')
        ax2 = fig.add_subplot(122)

        for i in range(len(self.part)):
            x = [self.frames[j,i].xpos/U_DIST for j in range(len(self.frames[:,i]))]
            y = [self.frames[j,i].ypos/U_DIST for j in range(len(self.frames[:,i]))]
            z = [self.frames[j,i].zpos/U_DIST for j in range(len(self.frames[:,i]))]
                #plotting
            ax1.scatter3D(0,0,0,c='grey',marker='o',alpha=0.5)
            ax1.plot3D(x,y,z,lw=0.5)
            ax2.plot(x,y,ms=0.5)

        #formatting plot
        ax1.set_xlabel('x (kpc)')
        ax1.set_ylabel('y (kpc)')
        ax1.set_zlabel('z (kpc)')
        ax2.set_xlabel('x (kpc)')
        ax2.set_ylabel('y (kpc)')
        ax2.yaxis.set_label_position('right')
        ax2.yaxis.tick_right()
        fig.tight_layout(pad=3.0)
        fig.subplots_adjust(hspace=4)
        plt.savefig(SAVEDIR+str(self.sys)+'_'+str(self.method)+'_'+str(self.inter)+'int'+'.png',dpi=200)
        #plt.show()

    def nextframe(self):
        if (self.method=='Euler'):
            for i in range(len(self.oldpart)):
                mass = self.oldpart[i].m
                newpartx = self.oldpart[i].xpos + self.oldpart[i].vx * self.tbin
                newparty = self.oldpart[i].ypos + self.oldpart[i].vy * self.tbin
                newpartz = self.oldpart[i].zpos + self.oldpart[i].vz * self.tbin
                r = np.sqrt(newpartx**2 + newparty**2 + newpartz**2)
                mass_r = 4*np.pi*RHOo*(Rs**3)*(np.log((Rs+r)/Rs)-(r/(Rs+r)))
                newpartvx = self.oldpart[i].vx - (G*mass_r/(r**3)) * newpartx * self.tbin
                newpartvy = self.oldpart[i].vy - (G*mass_r/(r**3)) * newparty * self.tbin
```

```
                newpartvz = self.oldpart[i].vz - (G*mass_r/(r**3)) * newpartz * self.tbin
                self.add(particle(mass,newpartx,newparty,newpartz,newpartvx,newpartvy,newpartvz))

        if (self.method=='RK4'):
            for i in range(len(self.oldpart)):
                [newpartx,newparty,newpartz,newpartvx,newpartvy,newpartvz] = self.rk4(i)
                self.add(particle(self.oldpart[i].m,newpartx,newparty,newpartz,newpartvx,newpartvy,
    newpartvz))

    def mass_r(self,x,y,z):
        r = np.sqrt(x**2 + y**2 + z**2)
        mass = 4*np.pi*RHOo*(Rs**3)*(np.log((Rs+r)/Rs)-(r/(Rs+r)))
        return mass,r

    def acc(self,x,y,z,vx,vy,vz,i):
        if self.inter=='no':
            m,r = self.mass_r(x,y,z)
            acc = [vx,vy,vz,-(G*m/(r**3))*x,-(G*m/(r**3))*y,-(G*m/(r**3))*z]
            return acc
        if self.inter=='yes':
            m,r = self.mass_r(x,y,z)
            accx = 0.
            accy = 0.
            accz = 0.
            for j in range(len(self.oldpart)):
                if j!=i:
                    dx = self.oldpart[j].xpos-x
                    dy = self.oldpart[j].ypos-y
                    dz = self.oldpart[j].zpos-z
                    intr = np.sqrt(dx**2+dy**2+dz**2+self.softlength**2)
                    accx += (self.oldpart[j].m)*(G*dx*(intr**(-3)))
                    accy += (self.oldpart[j].m)*(G*dy*(intr**(-3)))
                    accz += (self.oldpart[j].m)*(G*dz*(intr**(-3)))
            acc = [vx,vy,vz,accx-(G*m/(r**3))*x,accy-(G*m/(r**3))*y,accz-(G*m/(r**3))*z]
            return acc


    def rk4(self,i):
        h = self.tbin
        x = self.oldpart[i].xpos ;  y = self.oldpart[i].ypos ;  z = self.oldpart[i].zpos
        vx = self.oldpart[i].vx  ;  vy = self.oldpart[i].vy  ;  vz = self.oldpart[i].vz
        k1=self.acc(x,y,z,vx,vy,vz,i)
        k2=self.acc(x+k1[0]*h/2, y+k1[1]*h/2, z+k1[2]*h/2, vx+k1[3]*h/2, vy+k1[4]*h/2, vz+k1[5]*h/2,
    i)
        k3=self.acc(x+k2[0]*h/2, y+k2[1]*h/2, z+k2[2]*h/2, vx+k2[3]*h/2, vy+k2[4]*h/2, vz+k2[5]*h/2,
    i)
        k4=self.acc(x+k3[0]*h, y+k3[1]*h, z+k3[2]*h, vx+k3[3]*h, vy+k3[4]*h, vz+k3[5]*h,i)

        newx = x + (h/6.)*(k1[0] + 2*k2[0] + 2*k3[0] + k4[0])
        newy = y + (h/6.)*(k1[1] + 2*k2[1] + 2*k3[1] + k4[1])
        newz = z + (h/6.)*(k1[2] + 2*k2[2] + 2*k3[2] + k4[2])
        newvx = vx + (h/6.)*(k1[3] + 2*k2[3] + 2*k3[3] + k4[3])
        newvy = vy + (h/6.)*(k1[4] + 2*k2[4] + 2*k3[4] + k4[4])
        newvz = vz + (h/6.)*(k1[5] + 2*k2[5] + 2*k3[5] + k4[5])

        return newx,newy,newz,newvx,newvy,newvz

    def whole(self,t):
        totalt = t
        i = 0
        self.oldpart = self.part
        self.frames = np.append(self.frames,self.oldpart)
        self.part = np.array([])
        self.nextframe()
        i+=self.tbin
        while i < totalt:
            self.oldpart = self.part
            self.frames = np.vstack((self.frames,self.oldpart))
            self.part = np.array([])
            self.nextframe()
            i+=self.tbin
```

Appendix B: Pipeline for exercises 1 and 2

```
#IMPORTING PACKAGES:
import os
import random
import numpy as np
import matplotlib.pyplot as plt
from scipy import pi
import h5py
import sys
from classes import universe,particle
import time



#PROGRAM VARIABLES:
OUTDIR = 'output/'
ICSDIR = 'ics/'
FIGDIR = 'figures/'
U_T = 365*24*3600



#EXERCICES 1 & 2:

    #Arrays for diferent Nparts, methods and plot styles:
n = [1,10,100,1000]
sys = ['sun','load10','load100','load1000']
method = ['Euler','RK4']
c = ['blue','tab:orange']
ticks = [1,10,100,1000]

    #Computing different universes for each case:
for j in range(len(method)):
    t = np.array([])
    nt = np.array([])
    for i in range(len(sys)):
        ti = time.time()
        u = universe(method[j],sys[i],'no')
        u.whole(18.6e8*U_T)
        u.plot_trace()
        tf = time.time()
        t = np.append(t, (tf-ti))
        nt = np.append(nt, n[i])
    np.savetxt(OUTDIR+'timevsN_'+method[j]+'_noint.txt',np.transpose([t,nt]),header='Time(s)\tNparts
    ')

    #Plotting computing time vs Npart (no interaction between stars):
data = np.array([[],[]])
plt.figure(dpi=150)
for j in range(len(method)):
    data = np.loadtxt(OUTDIR+'timevsN_'+method[j]+'_noint.txt', skiprows=1)
    plt.plot(data[:,1],data[:,0],color=c[j],ls='-',lw=2,label=method[j])
    plt.plot(data[:,1],data[:,0],color=c[j],marker='.',ms=5)
plt.xticks(ticks)
plt.xlabel('Nparts')
plt.ylabel('t (s)')
plt.legend()
plt.savefig(FIGDIR+'timevsN_noint.png')
```

Appendix C: Pipeline for exercise 3

```
#IMPORTING PACKAGES:
import os
import random
import numpy as np
import matplotlib.pyplot as plt
from scipy import pi
import h5py
import sys
from classes import universe,particle
import time



#PROGRAM VARIABLES:
OUTDIR = 'output/'
ICSDIR = 'ics/'
FIGDIR = 'figures/'
U_T = 365*24*3600




#EXERCICE 3:

    #Arrays for diferent Nparts, methods and plot styles:
n = [10,100]
sys = ['load10','load100']
c = ['blue','tab:orange']
ticks = [1,10,100]
j=1

#    #Computing different universes for each case:
#t = np.array([])
#nt = np.array([])
#for i in range(len(sys)):
#    ti = time.time()
#    u = universe('RK4',sys[i],'yes')
#    u.whole(18.6e8*U_T)
#    u.plot_trace()
#    tf = time.time()
#    t = np.append(t, (tf-ti))
#    nt = np.append(nt, n[i])
#np.savetxt(OUTDIR+'timevsN_'+'RK4'+'_yesint.txt',np.transpose([t,nt]),header='Time(s)\tNparts')

    #Plotting computing time vs Npart (no interaction between stars):
data = np.array([[],[]])
plt.figure(dpi=150)
data = np.loadtxt(OUTDIR+'timevsN_'+'RK4'+'_yesint.txt', skiprows=1)
plt.plot(data[:,1],data[:,0],color=c[j],ls='-',lw=2,label='RK4')
plt.plot(data[:,1],data[:,0],color=c[j],marker='.',ms=5)
plt.xticks(ticks)
plt.xlabel('Nparts')
plt.ylabel('t (s)')
plt.legend()
plt.savefig(FIGDIR+'timevsN_yesint.png')
plt.show()
```