Exploring and animating the stable 3-body problems

Yuan Gao & Nicholas Kern

Introduction

As one of the most mysterious and appealing phenomenon, the 3-body systems have attracted scientists for decades. The reason that makes their orbits unstable is that those orbits are sensitive to different initial conditions, from which we can get 18 differential equations (including x1, x2, x3, vx1, vx2, vx3, y1, y2, y3, vy1, vy2, vy3, z1, z2, z3, vz1, vz2, vz3). My goal here is to write a code by using RK4 and adaptive step size to take all these 18 initial conditions in to consideration to make plotting and animation. Furthermore, I have found 4 initial conditions that makes the stars' orbit stable. This article will show the logic behind the code and how those conditions that make the orbit stable are found, with correspondent pictures helping to illuminate and test the ideas.

How to create the code?

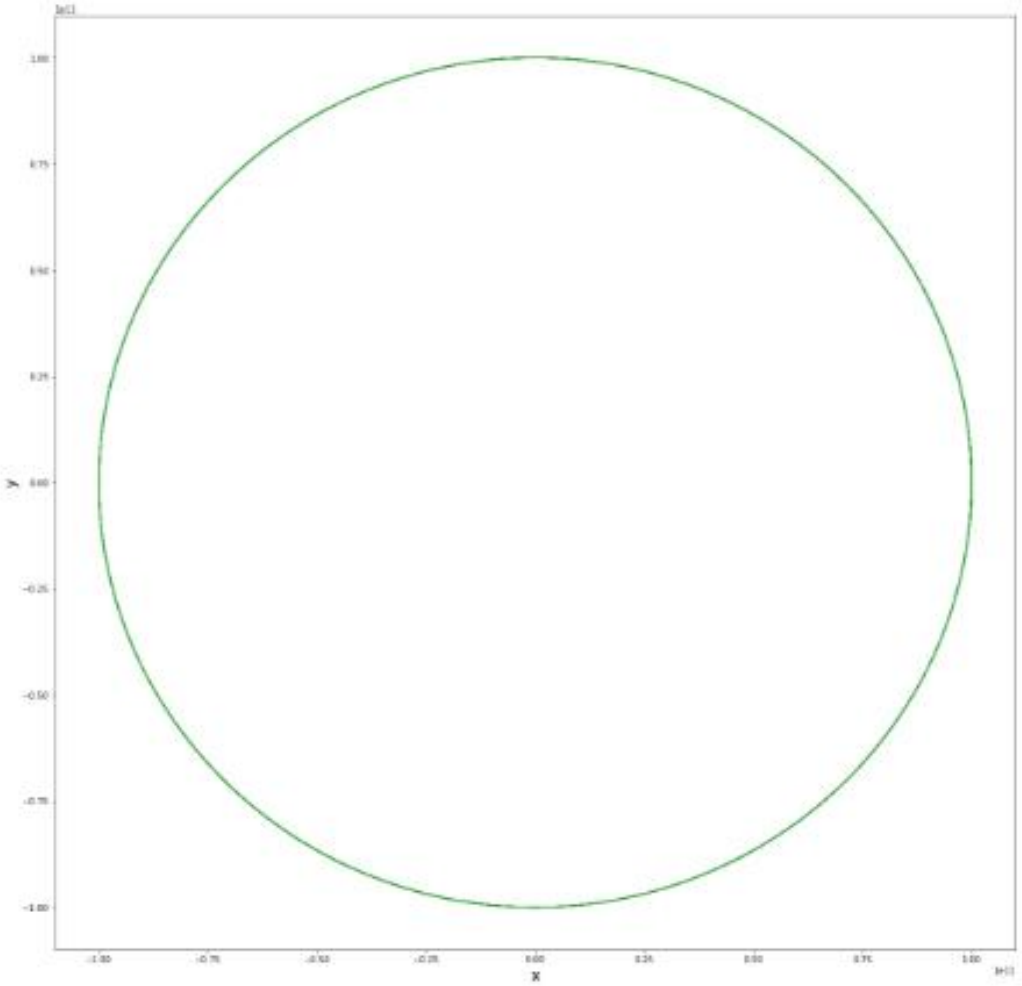
Firstly, since what we need to solve is a second derivative function, we need to make a function to prepare for the RK4 method, which is the first derivative function. Same as the breakout, here we take out each elements from the array containing the initial conditions to get their first derivative function. Then we set up for the RK4 function containing 2 loops to get the adaptive size. One thing worth noticing is that since we are going to plot 3 orbits together, it is impossible for them to always get the same adapt size as each other, which may result in that in certain time period, some have finished their orbit and stay statically and influence the running star wrongly. So I make a condition to take the smallest dt all the time to increase the resolution, the precision. What's more, I set the dt to one km per year. Finally, as soon as we create an array with initial conditions (here I take Jupiter as prototype, setting the orbit radius about 10**11 meters and the mass around 10**27 kilograms), our code will run and plot relatively precise orbits. I also choose the error tolerance to 1 km per year. To make more real stimulation, the rate of the stars, when they come closer, will be bigger, which means the vp.rate is inversly related to the dt. Here is what the code is like (seen appendix). Notice the sequence of the elements in the initial array (rini) is like this: np.array([x1, x2, x3, vx1, vx2, vx3, y1, y2, y3, vy1, vy2, vy3, z1, z2, z3, vz1, vz2, vz3])

The 3 stable orbits

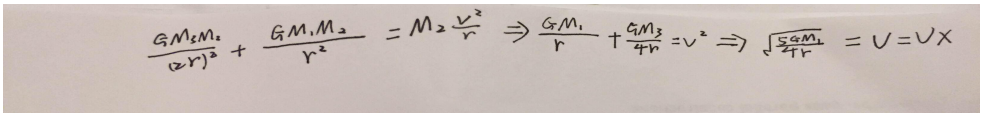
orbit 1

When it comes to plotting the stable 3-body condition, the first thing comes to my mind is that one star should always be static at the central with 3 stars in the same line, so that the force of other 2 stars on the central star would completely in balance. Here I simplify the

model by making the mass of the 3 stars the same, which are e27 each (M1=M2=M3=e27). Then, setting the central star at the origin and other 2 at (0, e11) and (0, -e11) separately, we



could calculate the v, which is same to the vx now, by using the equation GM1M2/R**2=M2*V**2/R, and get that vx equal to np.sqrt(5*G*M2/4/1e11), -np.sqrt(5*G*M2/4/1e11) separately.



So we take in these initial conditions and plot out a circle, which implies the orbits' stability seen (1). We can only see 1 orbit here because the orbits of the 2 side-star overlap while the central one is static. We set the array rini, which means r initial, like this:

M1=1e27

M2=1e27

M3=1e27

rini1=np.array([0,0,0,0,np.sqrt(5*G*M2/4/1e11),np.sqrt(5*G*M2/4/1e11),0,1e11,-1e11,0,0,0, 0,0,0, 0,0,0])

and the time bound and N to 4000000000.0 and 500000 seperately .

orbit 2

Then I am thinking of the similarities of 2-body system and 3-body system: what if 2 stars

firstly are in a stable 2-body system, and then a very light and distant star are orbiting the 2 star system? Will it be stable? The answer is yes because the light and distant star can hardly make influence on the big one. I set the initial position of the heavy central 2-bodies (0, e11) and (0, -e11) seperately. Then I simplify the model my making the 2 huge star the same mass---20e27, so the central of the 2-body orbit will be in the middle of the 2 stars. By calculating the necessary speed of completing the stable 2-body moving, I get their v, which is also vx now, np.sqrt(G*M1/4e11), -np.sqrt(G*M1/4e11).

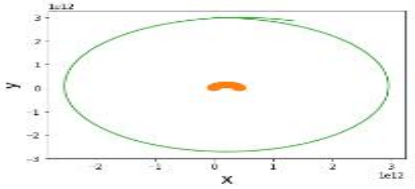
$$\frac{GM_1M_2}{(2r)^2} = M_2\frac{V^2}{r} \Rightarrow V=\sqrt{\frac{GM_1}{4r}} = Vx$$

As for the third, light, and distant star, because of it distance, we can regard the 2 big stars as a bigger star with double mass in the origin, which is their centroid. Then this question becomes another 2-body problem. For counting the speed easily, I still set the location of the third star at (0, 30e11), which guaratee the v equals vx and vy is 0.

$$\frac{G(M_1+M_2)M_3}{(r')^2} = \frac{M_3V^2}{r'} \Rightarrow \sqrt{\frac{2GM_1}{r'}} = V \quad (r'=30e11)$$

Here is the plot:

the reason why there are a orange area is that the small star does have some influence on the 2-bigstar system. I set rini2 as this:



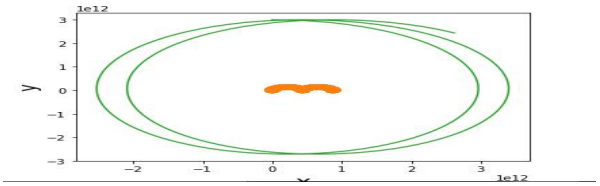
rini2=np.array([0,0,0,
np.sqrt(G*M1/4e11),-np.sqrt(G*M1/4e11),np.sqrt(2*G*M1/30e11),
1e11,-1e11,30e11, 0,0,0, 0,0,0, 0,0,0])

M1=20e27

M2=20e27

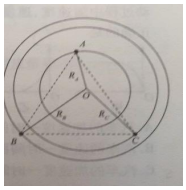
M3=1e27

And the time bound 20000000000.0 and N 500000 since if I increase the time bound, it will be unstable, which shows the unstability of 3-body systems.

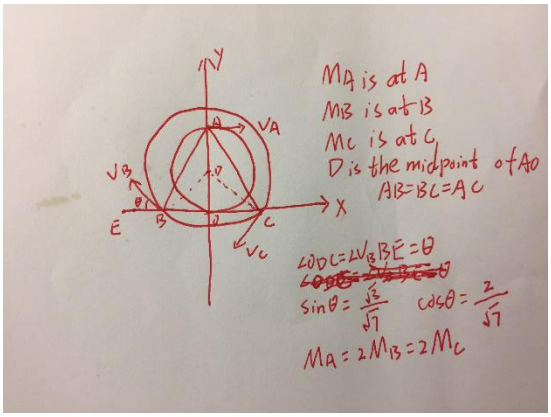


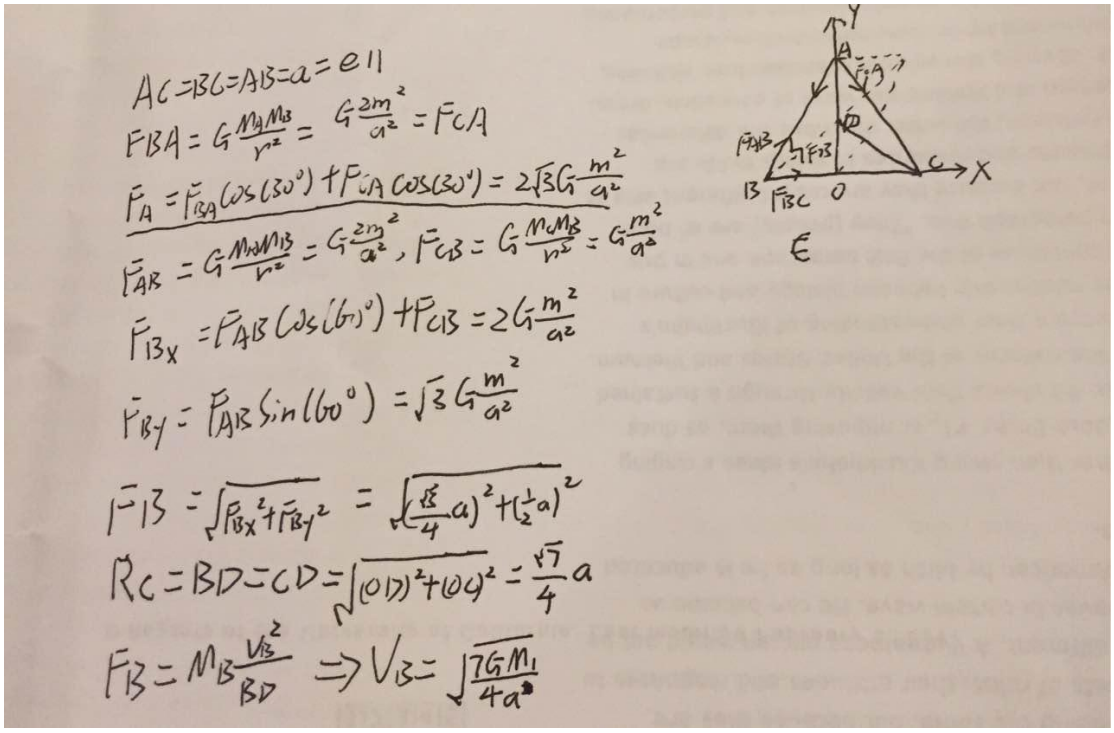
Orbit 3

Now we are going to make use of the vx and vy to make a more complicate stable situation: In this condition, 3 stars are at the vertex of a equilateral triangle, O being the central of their orbits (though the radius of the orbit may be different). To make the condition easier to stimulate, I set the mass of star B and star C the same---M1=15e11 and M2=15e11 respectively, under which circumstance I predict there should only be 2 orbits each because B and C will overlap. The mass of A will be 30e11. The location of A, B, C are (0.5e11, 0), (-0.5e11, 0), (0, 0.5*1e11*np.sqrt(3)), and the radius of orbit A, the length of AD, is 0.25*1e11*np.sqrt(3). The radius of orbit B and C is BD



and CD, which equally are np.sqrt(7*G*M1/4e11). As shown in the picture, according to the principle of the operation of vector, $F_B=F_{AB}+F_{CB}$, F_B=np.sqrt(7)*G*M1 so V_B=np.sqrt(7*G*M1/4e11). Consequently, VXA= np.sqrt(3*G*M1/4e11), VXB=VXC= -np.sqrt(3)/np.sqrt(7))*np.sqrt(7*G*M1/4e11). VYA=0, VYB=-VYC= (2/np.sqrt(7))*np.sqrt(7*G*M1/4e11).





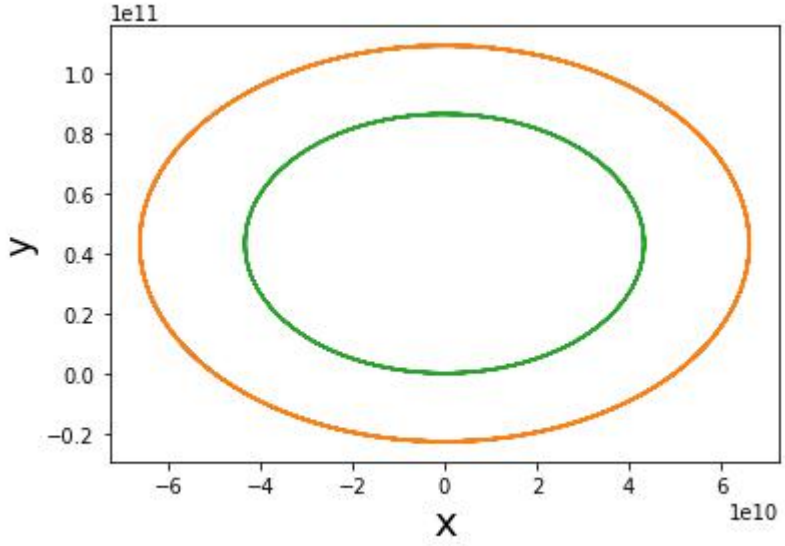
Here is what I set for rini3:

M1=M2=15e11

M3=30e11(star A)

rini3=np.array([0.5*1e11,-0.5*1e11,0,-(np.sqrt(3)/np.sqrt(7))*np.sqrt(7*G*M1/4e11),(np.sqrt(3)/np.sqrt(7))*np.sqrt(7*G*M1/4e11),np.sqrt(3*G*M1/4e11),0,0,0.5*1e11*np.sqrt(3),

-(2/np.sqrt(7))*np.sqrt(7*G*M1/4e11),(2/np.sqrt(7))*np.sqrt(7*G*M1/4e11),0, 0,0,0, 0,0,0])



Taking z into consideration

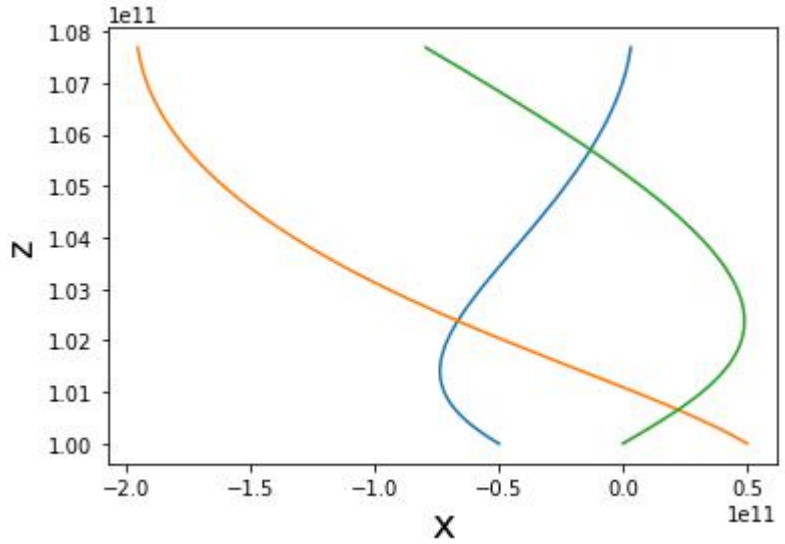
By far we have only talked about 2 demensional orbits, so in order to show the generalizability of the code, I randomly set the initial conditions, which take z axis in to consideration. Here is my initial condition:

rini4=np.array([0.5*1e11,-0.5*1e11,0,

-(np.sqrt(3)/np.sqrt(7))*np.sqrt(7*G*M1/4e11),-(np.sqrt(3)/np.sqrt(7))*np.sqrt(7*G*M1/4e11),np.sqrt(3*G*M1/4e11),

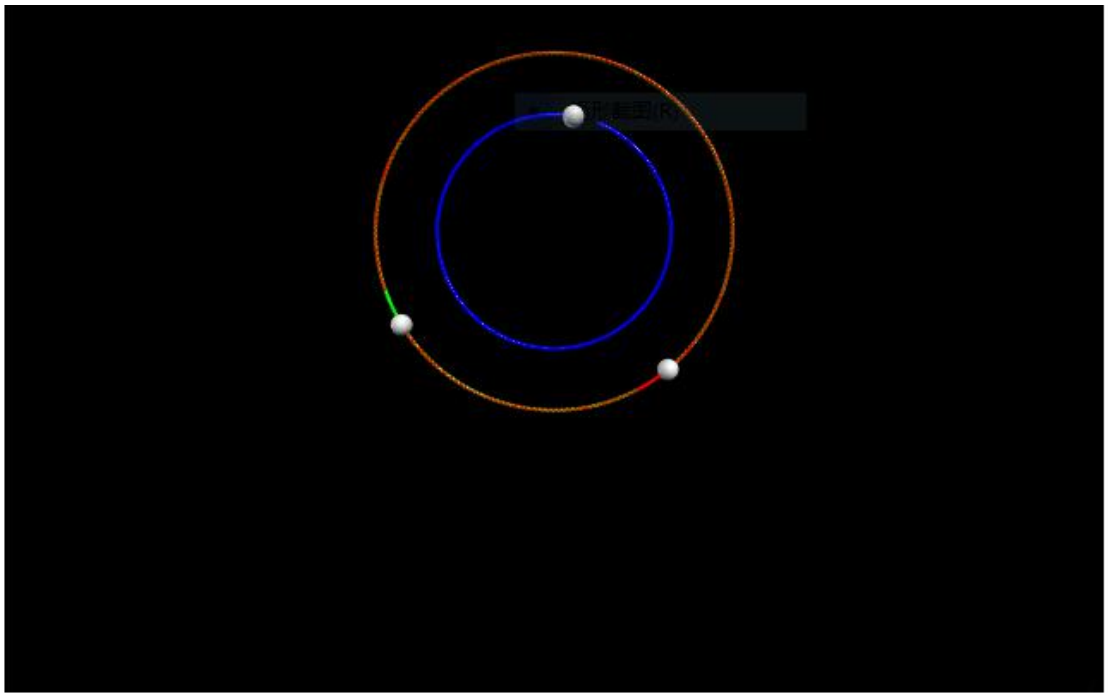
0,0,0.5*1e11*np.sqrt(3), -(2/np.sqrt(7))*np.sqrt(7*G*M1/4e11),(2/np.sqrt(7))*np.sqrt(7*G*M1/4e11),0, 1e11,1e11,1e11,

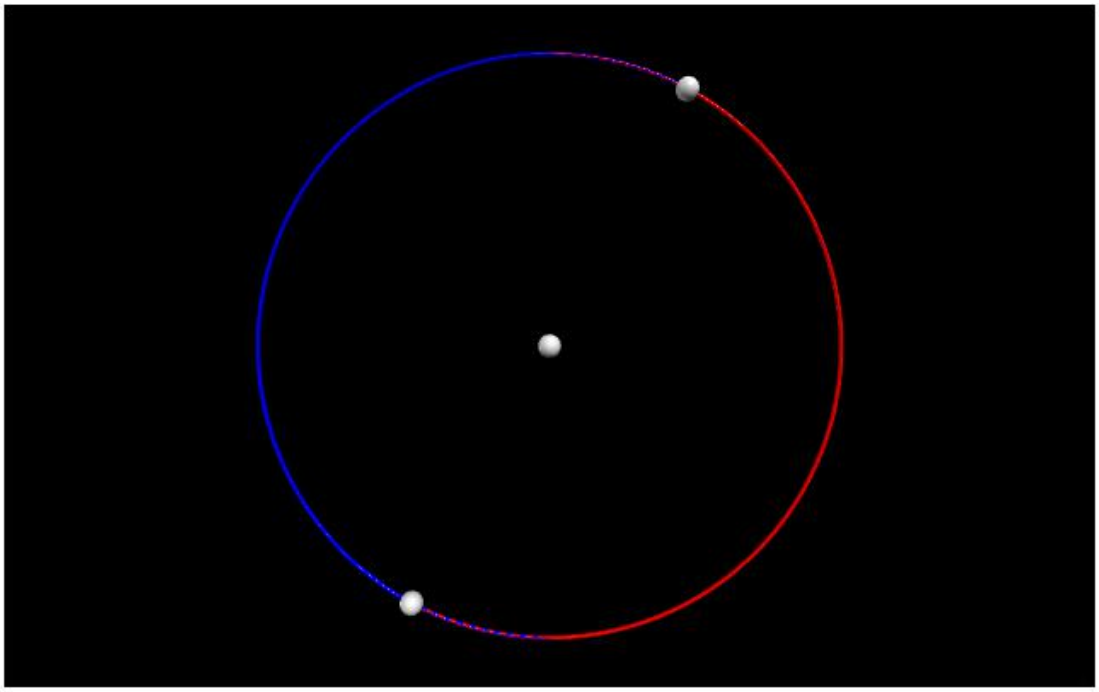
20,20,20])

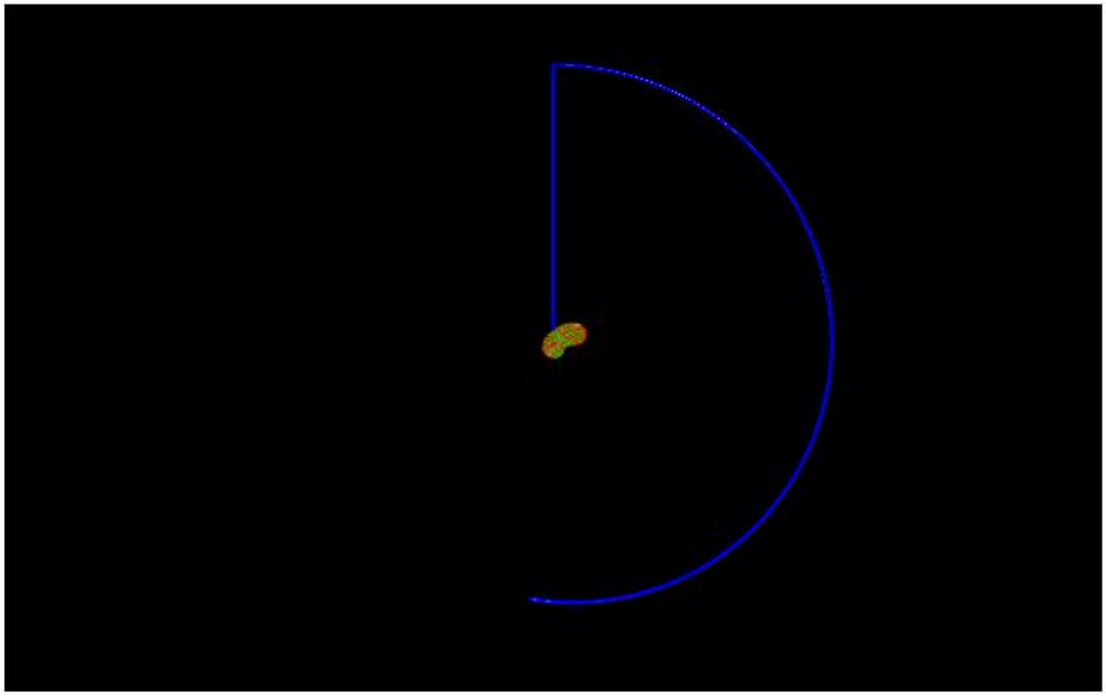


Animation

The animation of the three stable orbits by using vpython are like these:







Those corrospond to the orbits we plotted before.

Conclusion

The program i produced is efficient enough because it takes less than 30 seconds to run. When the code starts to keep running, I turned the tright smaller and error tolerance bigger until it takes less than 10 seconds to run and then I just add them gradually. This code could take all the possible initial states of the 3-body problem, ploting and animating them. To make the program more useful, I have found 3 stale orbits.

Appendix（also in the final.txt i commited）

The code(for the third stable orbit's animation)

```
import vpython as vp
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
G=6.67*1e-11
M1=15*1e27
M2=15*1e27
M3=30*1e27
rini1=np.array([0,0,0, 0,np.sqrt(5*G*M2/4/1e11),-np.sqrt(5*G*M2/4/1e11), 0,1e11,-1e11, 0,0,0, 0,0,0, 0,0,0])
rini2=np.array([0,0,0, np.sqrt(G*M1/4e11),-np.sqrt(G*M1/4e11),np.sqrt(2*G*M1/30e11), 1e11,-1e11,30e11, 0,0,0, 0,0,0,
0,0,0])
rini3=np.array([0.5*1e11,-0.5*1e11,0,
-(np.sqrt(3)/np.sqrt(7))*np.sqrt(7*G*M1/4e11),-(np.sqrt(3)/np.sqrt(7))*np.sqrt(7*G*M1/4e11),np.sqrt(3*G*M1/4e11),
0,0,0.5*1e11*np.sqrt(3), -(2/np.sqrt(7))*np.sqrt(7*G*M1/4e11),(2/np.sqrt(7))*np.sqrt(7*G*M1/4e11),0, 0,0,0, 0,0,0])
ball1 = vp.sphere(pos=vp.vector(rini1[0],rini1[6],rini1[12]), radius=4000000000,make_trail=True, interval=1, retain=200,
```

```
trail_color=vp.color.green)
ball2 = vp.sphere(pos=vp.vector(rini1[1],rini1[7],rini1[13]), radius=4000000000,make_trail=True, interval=1, retain=200,
trail_color=vp.color.red)
ball3 = vp.sphere(pos=vp.vector(rini1[2],rini1[8],rini1[14]), radius=4000000000,make_trail=True, interval=1, retain=200,
trail_color=vp.color.blue)
def f(r,t):
    x1=r[0]
    x2=r[1]
    x3=r[2]
    vx1=r[3]
    vx2=r[4]
    vx3=r[5]
    y1=r[6]
    y2=r[7]
    y3=r[8]
    vy1=r[9]
    vy2=r[10]
    vy3=r[11]
    z1=r[12]
    z2=r[13]
    z3=r[14]
    vz1=r[15]
    vz2=r[16]
    vz3=r[17]
    r12=np.sqrt((x2-x1)**2+(y2-y1)**2+(z2-z1)**2)
    r13=np.sqrt((x3-x1)**2+(y3-y1)**2+(z3-z1)**2)
    r23=np.sqrt((x3-x2)**2+(y3-y2)**2+(z3-z2)**2)
    fx1=vx1
    fy1=vy1
    fz1=vz1
    fx2=vx2
    fy2=vy2
    fz2=vz2
    fx3=vx3
    fy3=vy3
    fz3=vz3
    fvx1=G*M2*(x2-x1)/abs(r12)**3+G*M3*(x3-x1)/abs(r13)**3
    fvy1=G*M2*(y2-y1)/abs(r12)**3+G*M3*(y3-y1)/abs(r13)**3
    fvz1=G*M2*(z2-z1)/abs(r12)**3+G*M3*(z3-z1)/abs(r13)**3
    fvx2=G*M1*(x1-x2)/abs(r12)**3+G*M3*(x3-x2)/abs(r23)**3
    fvy2=G*M1*(y1-y2)/abs(r12)**3+G*M3*(y3-y2)/abs(r23)**3
    fvz2=G*M1*(z1-z2)/abs(r12)**3+G*M3*(z3-z2)/abs(r23)**3
    fvx3=G*M2*(x2-x3)/abs(r23)**3+G*M1*(x1-x3)/abs(r13)**3
    fvy3=G*M2*(y2-y3)/abs(r23)**3+G*M1*(y1-y3)/abs(r13)**3
```

```
    fvz3=G*M2*(z2-z3)/abs(r23)**3+G*M1*(z1-z3)/abs(r13)**3
    return np.array([fx1,fx2,fx3,fvx1,fvx2,fvx3,fy1,fy2,fy3,fvy1,fvy2,fvy3,fz1,fz2,fz3,fvz1,fvz2,fvz3],dtype=np.float)
def RK4(f, r, tleft, tright, dt, err_tol=1000/31556952):
    def rk4_update(r, t, dt):
        k1 = dt * f(r, t)
        k2 = dt * f(r+0.5*k1, t+0.5*dt)
        k3 = dt * f(r+0.5*k2, t+0.5*dt)
        k4 = dt * f(r+k3, t+dt)
        return r + (k1 + 2*k2 + 2*k3 + k4)/6.0
    t=tleft
    x1points=[]
    y1points=[]
    z1points=[]
    x2points=[]
    y2points=[]
    z2points=[]
    x3points=[]
    y3points=[]
    z3points=[]
    while t <= tright:
        x1points.append(r[0])
        x2points.append(r[1])
        x3points.append(r[2])
        y1points.append(r[6])
        y2points.append(r[7])
        y3points.append(r[8])
        z1points.append(r[12])
        z2points.append(r[13])
        z3points.append(r[14])
        while True:
            r1_a=rk4_update(r, t, dt).copy()
            r1=rk4_update(r1_a, t+dt, dt).copy()
            r2=rk4_update(r, t, 2*dt).copy()
            eps_x1 = np.abs(r1[0] - r2[0])/30.0
            eps_y1 = np.abs(r1[6] - r2[6])/30.0
            eps_z1 = np.abs(r1[12] - r2[12])/30.0
            eps_tot1 = np.sqrt(eps_x1**2 + eps_y1**2+eps_z1**2)
            eps_x2 = np.abs(r1[1] - r2[1])/30.0
            eps_y2 = np.abs(r1[7] - r2[7])/30.0
            eps_z2= np.abs(r1[13] - r2[13])/30.0
            eps_tot2 = np.sqrt(eps_x2**2 + eps_y2**2+eps_z2**2)
            eps_x3 = np.abs(r1[2] - r2[2])/30.0
            eps_y3 = np.abs(r1[8] - r2[8])/30.0
            eps_z3 = np.abs(r1[14] - r2[14])/30.0
```

```
                eps_tot3 = np.sqrt(eps_x3**2 + eps_y3**2+eps_z3**2)
                rho = min((dt*err_tol/eps_tot1)**(1./4),(dt*err_tol/eps_tot2)**(1./4),(dt*err_tol/eps_tot3)**(1./4))
                if rho >= 1.0:
                    if rho >= 2.0:
                        rho = 2.0
                    break
                else:
                    if rho < 0.5:
                        rho = 0.5
                    dt *= 0.99*rho

            dt *= 0.99*rho
            r = r1_a.copy()
            vp.rate(20000000/dt)
            t += dt
            ball1.pos=vp.vector(r[0],r[6],r[12])
            ball2.pos=vp.vector(r[1],r[7],r[13])
            ball3.pos=vp.vector(r[2],r[8],r[14])
        x1points = np.array(x1points)
        y1points = np.array(y1points)
        x2points = np.array(x2points)
        y2points = np.array(y2points)
        x3points = np.array(x3points)
        y3points = np.array(y3points)
        return x1points,y1points,z1points,x2points,y2points,z2points,x3points,y3points,z3points
tleft = 0.0
tright = 400000000.0
N = 50000
dt =float(tright- tleft) / N
xp1,yp1,zp1,xp2,yp2,zp2,xp3,yp3,zp3=RK4(f,rini3,tleft,tright,dt,err_tol=1000/31556952)
fig = plt.figure(figsize=(20,20))
ax2 = fig.add_subplot(1, 1, 1)
ax2.set_xlabel('x', fontsize=20)
ax2.set_ylabel('y', fontsize=18)
ax2.set_xlim(-0.5*1e+12,0.5*1e+12)
ax2.set_ylim(-0.5*1e+12,0.5*1e+12)
ax2.plot(xp2, yp2)
ax2.plot(xp1, yp1)
ax2.plot(xp3, yp3)
```