Solar System Simulation / Orbital Dynamics in Python

Ronan Waroquet

December 2022

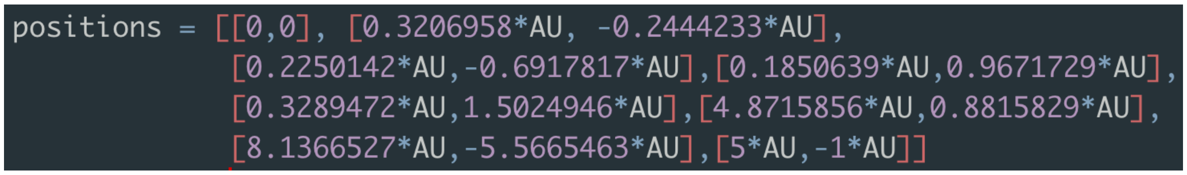
Introduction

The work described below shows my work of Python to create a 2D visualization and dynamic solar system using Newton's Gravitation. In this system, I use Newton's Gravitation law and euler's method to calculate the positions of numerous planets and bodies in an attempt to create an accurate system of 'particles'. The positions and velocities calculated are then used by the Python to simulate a system of bodies over a time period to create an animation to observe these calculate bodies.

Data

In order to make a solar system of an appropriate scale, it is just as important to utilize whatever data is available to make the Python program. The goal was to use euler's method to iterate through a number of iterations, updating the forces each of the planets and bodies experience to determine their positions and velocities. There are a plethora of websites to access planet data. I was able to obtain the x, y, and z positions (in AU) of each planet from nasa's website https://solarsystem.nasa.gov/planets/overview/.

A Python declaration of an array of these positions is shown below.



```
positions = [[0,0], [0.3206958*AU, -0.2444233*AU],
             [0.2250142*AU,-0.6917817*AU],[0.1850639*AU,0.9671729*AU],
             [0.3289472*AU,1.5024946*AU],[4.8715856*AU,0.8815829*AU],
             [8.1366527*AU,-5.5665463*AU],[5*AU,-1*AU]]
```

Figure 1: Positions of Bodies and Planets

In Python, I able to declare these positions, by storing the x and y positions of each planet. Observe that in this array, there are embedded arrays of dimension 1x2. These arrays store the positions. In order to see the effects of orbital dynamics in my code, I also added a comet (the last entry of the array) whose x and y positions are $5AU$ and $-1AU$, respectively, with respect to the Sun. We need to update the positions of these bodies. In order to do this, it is needed to to calculate the forces each individual body experiences to determine their next positions. Utilizing euler's method, we essentially have the following...

$$P(t) = P(t - dt) + \frac{dP(t - dt)}{dt} dt \tag{1}$$

where P is the position of the body and $\frac{dP}{dt}$ is the velocity of the body. In order to update these positions, we need to determine the velocities. The velocities are also determined utilizing euler's method...

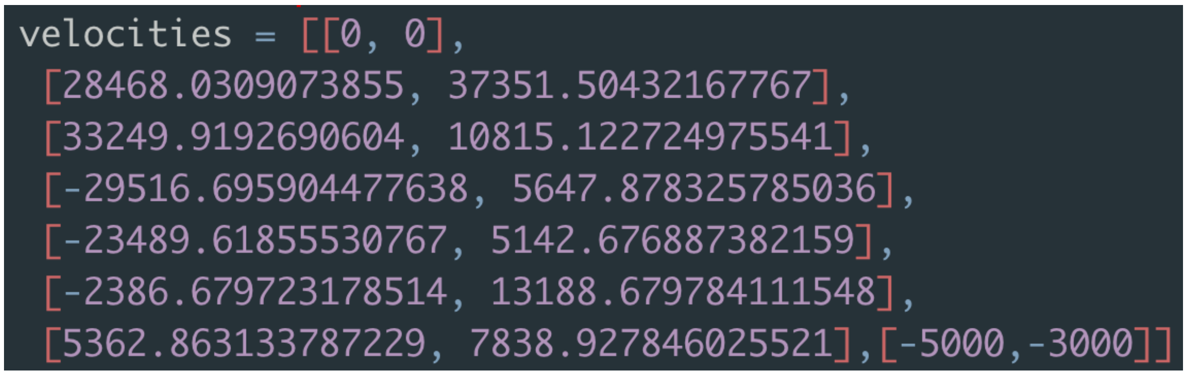
$$V(t) = V(t - dt) + \frac{F(t - dt)}{m} dt \tag{2}$$

where V is the velocity, F is the net force the object is experiencing, and m is the mass of the object. Iterating equations (1) and (2) is sufficient enough to simulate a sample solar system.

Finding live information on the speeds of the planets was very difficult to find. In order to find the accurate velocities of the planets, I had to calculate them myself. Any information regarding the velocities of planets were shows as they were in their Aphelion or Perihelion positions. My desire was to code a solar system whose positions are that of today's date. The following equation was used to approximate the velocities of each planet.

$$v \approx \sqrt{\frac{GM}{r}} \tag{3}$$

where G is the gravitational constant, M is the mass of the Sun, and r is the distance the object is away from the Sun. After some basic trigonometry, algebra, and simplification, I was able to determine the true velocities of the planets in Python by producing the following...



```
velocities = [[0, 0],
 [28468.0309073855, 37351.50432167767],
 [33249.9192690604, 10815.122724975541],
 [-29516.695904477638, 5647.878325785036],
 [-23489.61855530767, 5142.676887382159],
 [-2386.679723178514, 13188.679784111548],
 [5362.863133787229, 7838.927846025521],[-5000,-3000]]
```

Figure 2: Velocities of Bodies

The construction of these arrays are the same as before. Each embedded array stores the x and y values of the velocities.

Algorithm

Now that we have a basic of idea of what needs done to calculate the positions and velocities of the bodies, we need to construct an algorithm to iterate the process over a long time interval. The most important detail to pay attention to is that we need to calculate the net force each body is experiencing. Therefore, we need to run our algorithm to calculate the pair-wise forces (velocities and positions follow), of the planets. A simple double embedded for loop will take care of it. We want to simulate this whole systme, therefore we need to store all x and y values. Not just the magnitude of them. Observe the code below that is responsible for iterating the procedure.

```
velocities_x, velocities_y = [], []

day_sec = 24*60*60
dt = day_sec*1

num_Iterations = 365*4
num_planets = 8

positions_list = []
velocities_list = []

for i in range(num_Iterations):
    forces = []

    for j in range(num_planets): #calculate at planet j
        fx = 0
        fy = 0

        for k in range(num_planets): #calculate pairs

            if j != k:
                rx = positions[j][0] - positions[k][0]
                ry = positions[j][1] - positions[k][1]

                r = (rx**2 + ry**2)**0.5
                f = -(G * masses[j] * masses[k]) / (r**2)

                fx += f * (rx / r)
                fy += f * (ry / r)

        forces.append([fx, fy])

    for j in range(num_planets):
        velocities[j][0] += forces[j][0] * dt / masses[j]
        velocities[j][1] += forces[j][1] * dt / masses[j]
        positions[j][0] += velocities[j][0] * dt
        positions[j][1] += velocities[j][1] * dt

    positions_list.append([[positions[0][0], positions[0][1]],[positions[1][0],
                                            positions[1][1]], [positions[2][0],
                                            positions[2][1]], [positions[3][0],
                                            positions[3][1]], [positions[4][0],
                                            positions[4][1]], [positions[5][0],
                                            positions[5][1]], [positions[6][0],
                                            positions[6][1]], [positions[7][0],
                                            positions[7][1]]])
```

Without going into detail of the syntax of Python, just observe 3 for loops present in the code. The outer one is responsible for iterating this algorithm however many time we want, and the two embedded for loops are used to calculate the pair-wise forces between the bodies. The final lines of the code append all data into arrays to be used for graphing and simulation. The many brackets used are used to store all x and y values.

Graphing and Simulation

For graphing and simulation, I utilized the Python library matplotlib. To animate our stores positions of the bodies, the package essentially creates individual frames and stitches them together over our pre-defined iteration count and animates it. The code, shown at the bottom of the paper, was used to animate this system (very long and tedious code). After simulating, we are able to produce the following screenshot of the simulation, given below (Figure 3).

If we take a look at the code presented previously, all bodies in the solar system experience net forces from all other bodies. This includes the Sun. The Sun, in this case, is not still. That is, it does experience forces and is allowed to freely roam. Since the sun has increased mass in comparison to the other bodies, it's net force is a lot less. If we, for example, increase the mass of another body, we can get a different motion of the solar system. You can see this with an increased Jupiter mass, for example, in figure (4), below.

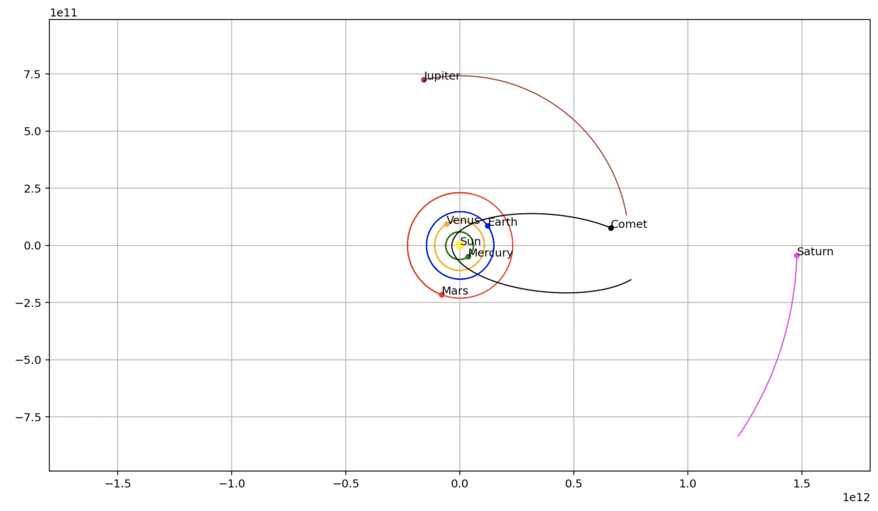


Figure 3: Solar System Simulation

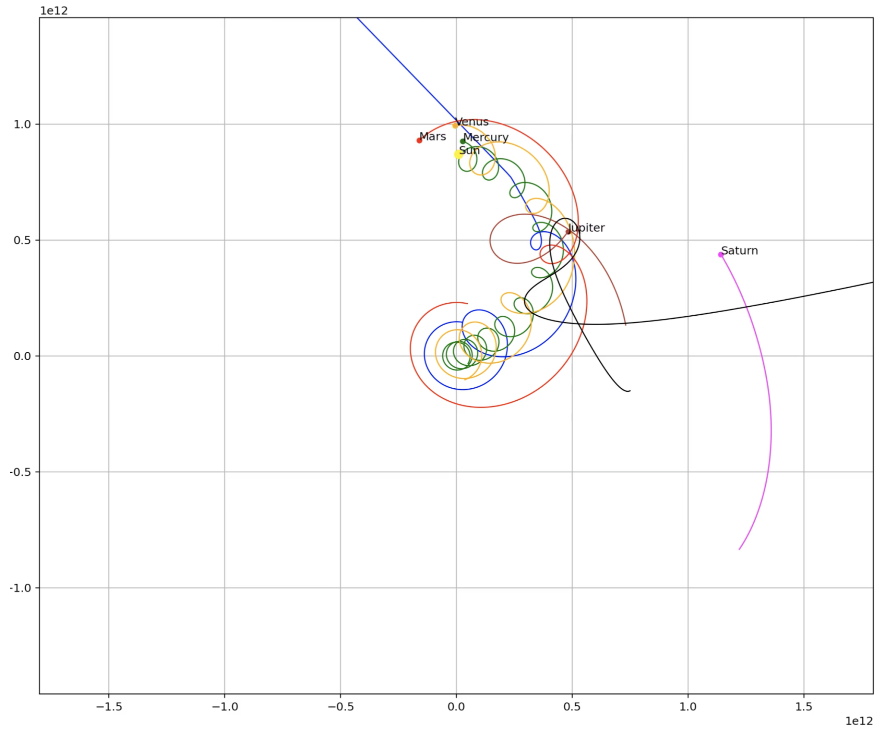


Figure 4: Solar System With Increased Jupiter Mass

Conclusion and Limitations

The code itself is very limited. That is, we do not utilize any other frictional forces or energy associated with this system. This solar system is in an ideal vacuum without the effect of other interstellar bodies, such as asteroids and satellites. This is our primary limitation to this model. For small times, t, our utilization of the euler's method, by nature, will accumulate errors over a period of time. So in addition to the expense of other forces in a realistic system, we experience errors over a long period of time. Since the expanse of space is so large, these errors will not be present in the simulations as we deal with a very small time interval of several years. A more realistic and accurate approach to this modeling problem is to use an alternative iterative algorithm or solve the system analytically for all time t. Alternative methods could be the runge-kutta method or velocity-verlet. However, the resultant code is sufficient to portray Newton's Gravitation of a small system of bodies.

Full Code

```
from matplotlib import animation
import numpy as np
import matplotlib.pyplot as plt

G = 6.67e-11
Ms = 2.0e30
Mmer = 0.330e24
Mven = 4.87e24
Me = 5.972e24
Mm = 0.642e24
Mj = 1898e24
Msat = 568e24
Mcomet = 10000
AU = 1.5e11
num_planets = 7

positions = [[0,0], [0.3206958*AU, -0.2444233*AU],
             [0.2250142*AU,-0.6917817*AU],[0.1850639*AU,0.9671729*AU],
             [0.3289472*AU,1.5024946*AU],[4.8715856*AU,0.8815829*AU],
             [8.1366527*AU,-5.5665463*AU],[5*AU,-1*AU]]

velocities = [[0, 0],
 [28468.0309073855, 37351.50432167767],
 [33249.9192690604, 10815.122724975541],
 [-29516.695904477638, 5647.878325785036],
 [-23489.61855530767, 5142.676887382159],
 [-2386.679723178514, 13188.679784111548],
 [5362.863133787229, 7838.927846025521],[-5000,-3000]]
masses = [Ms, Mmer, Mven, Me, Mm, Mj, Msat, Mcomet]

velocities_x, velocities_y = [], []

day_sec = 24*60*60
dt = day_sec*1

num_Iterations = 365*4
num_planets = 8

positions_list = []
velocities_list = []

for i in range(num_Iterations):
    forces = []

    for j in range(num_planets): #calculate at planet j
        fx = 0
        fy = 0

        for k in range(num_planets): #calculate pairs

            if j != k:
                rx = positions[j][0] - positions[k][0]
                ry = positions[j][1] - positions[k][1]

                r = (rx**2 + ry**2)**0.5
                f = -(G * masses[j] * masses[k]) / (r**2)

                fx += f * (rx / r)
                fy += f * (ry / r)

        forces.append([fx, fy])

    for j in range(num_planets):
        velocities[j][0] += forces[j][0] * dt / masses[j]
        velocities[j][1] += forces[j][1] * dt / masses[j]
        positions[j][0] += velocities[j][0] * dt
        positions[j][1] += velocities[j][1] * dt

    positions_list.append([[positions[0][0], positions[0][1]],[positions[1][0],
                                             positions[1][1]], [positions[2][0],
                                             positions[2][1]], [positions[3][0],
                                             positions[3][1]], [positions[4][0],
                                             positions[4][1]], [positions[5][0],
                                             positions[5][1]], [positions[6][0],
                                             positions[6][1]], [positions[7][0],
                                             positions[7][1]]])
```

The code below is used to animate the simulation. No further calculations are made.

```


fig, ax = plt.subplots(figsize=(10,10))
ax.set_aspect('equal')
ax.grid()

line_e,     = ax.plot([],[],'-g',lw=1,c='blue')
point_e,    = ax.plot([AU], [0], marker="o"
                      , markersize=4
                      , markeredgecolor="blue"
                      , markerfacecolor="blue")
text_e      = ax.text(AU,0,'Earth')

line_merc,     = ax.plot([],[],'-b',lw=1,c='green')
point_merc,    = ax.plot([AU], [0], marker="o"
                      , markersize=4
                      , markeredgecolor="green"
                      , markerfacecolor="green")
text_merc      = ax.text(AU,0,'Mercury')

line_ven,     = ax.plot([],[],'-b',lw=1,c='orange')
point_ven,    = ax.plot([AU], [0], marker="o"
                      , markersize=4
                      , markeredgecolor="orange"
                      , markerfacecolor="orange")
text_ven      = ax.text(AU,0,'Venus')

line_mars,     = ax.plot([],[],'-b',lw=1,c='red')
point_mars,    = ax.plot([AU], [0], marker="o"
                      , markersize=4
                      , markeredgecolor="red"
                      , markerfacecolor="red")
text_mars      = ax.text(AU,0,'Mars')

line_jup,     = ax.plot([],[],'-b',lw=1,c='brown')
point_jup,    = ax.plot([AU], [0], marker="o"
                      , markersize=4
                      , markeredgecolor="brown"
                      , markerfacecolor="brown")
text_jup      = ax.text(AU,0,'Jupiter')

line_sat,     = ax.plot([],[],'-b',lw=1,c='magenta')
point_sat,    = ax.plot([8.1366527*AU], [-5.5665463*AU], marker="o"
                      , markersize=4
                      , markeredgecolor="magenta"
                      , markerfacecolor="magenta")
text_sat      = ax.text(8.1366527*AU,-5.5665463*AU,'Saturn')

line_comet,     = ax.plot([],[],'-b',lw=1,c='black')
point_comet,    = ax.plot([AU], [0], marker="o"
                      , markersize=4
                      , markeredgecolor="black"
                      , markerfacecolor="black")
text_comet      = ax.text(AU,0,'Comet')


point_s,    = ax.plot([0], [0], marker="o"
                      , markersize=7
                      , markeredgecolor="yellow"
                      , markerfacecolor="yellow")
text_s      = ax.text(0,0,'Sun')

exdata,eydata = [],[]                   # earth track
```

```
sxdata,sydata = [],[]                    # sun track
mercxdata, mercydata = [],[]
venxdata, venydata = [],[]
marsxdata, marsydata = [],[]
jupxdata,jupydata = [],[]
satxdata,satydata = [],[]
cometxdata,cometydata = [],[]


print(len(positions_list))

def update(i):
    exdata.append(positions_list[i][3][0])
    eydata.append(positions_list[i][3][1])

    mercxdata.append(positions_list[i][1][0]) #i'th frame, planet 1, x
    mercydata.append(positions_list[i][1][1]) #i'th frame, planet 1, y

    venxdata.append(positions_list[i][2][0]) #i'th frame, planet 1, x
    venydata.append(positions_list[i][2][1]) #i'th frame, planet 1, y

    marsxdata.append(positions_list[i][4][0])
    marsydata.append(positions_list[i][4][1])

    jupxdata.append(positions_list[i][5][0])
    jupydata.append(positions_list[i][5][1])

    satxdata.append(positions_list[i][6][0])
    satydata.append(positions_list[i][6][1])

    cometxdata.append(positions_list[i][7][0])
    cometydata.append(positions_list[i][7][1])

    line_merc.set_data(mercxdata,mercydata)
    point_merc.set_data(positions_list[i][1][0],positions_list[i][1][1])
    text_merc.set_position((positions_list[i][1][0],positions_list[i][1][1]))

    line_ven.set_data(venxdata,venydata)
    point_ven.set_data(positions_list[i][2][0],positions_list[i][2][1])
    text_ven.set_position((positions_list[i][2][0],positions_list[i][2][1]))

    line_e.set_data(exdata,eydata)
    point_e.set_data(positions_list[i][3][0],positions_list[i][3][1])
    text_e.set_position((positions_list[i][3][0],positions_list[i][3][1]))

    line_mars.set_data(marsxdata,marsydata)
    point_mars.set_data(positions_list[i][4][0],positions_list[i][4][1])
    text_mars.set_position((positions_list[i][4][0],positions_list[i][4][1]))
    line_jup.set_data(jupxdata,jupydata)
    point_jup.set_data(positions_list[i][5][0],positions_list[i][5][1])
    text_jup.set_position((positions_list[i][5][0],positions_list[i][5][1]))

    line_sat.set_data(satxdata,satydata)
    point_sat.set_data(positions_list[i][6][0],positions_list[i][6][1])
    text_sat.set_position((positions_list[i][6][0],positions_list[i][6][1]))

    line_comet.set_data(cometxdata,cometydata)
    point_comet.set_data(positions_list[i][7][0],positions_list[i][7][1])
    text_comet.set_position((positions_list[i][7][0],positions_list[i][7][1]))

    point_s.set_data(positions_list[i][0][0],positions_list[i][0][1])
    text_s.set_position((positions_list[i][0][0],positions_list[i][0][1]))
    ax.axis('equal')
    ax.set_xlim(-12*AU,12*AU)
    ax.set_ylim(-12*AU,12*AU)

    return line_e,point_s,point_e,text_e,text_s,line_merc,point_merc,text_merc,
                                            line_ven,point_ven,text_ven,line_mars,
                                            point_mars,text_mars,line_jup,point_jup
                                            ,text_jup,line_sat,point_sat,text_sat,
                                            line_comet,point_comet,text_comet



anim = animation.FuncAnimation(fig
                                ,func=update
                                ,frames=len(positions_list)
                                ,interval=1
                                ,blit=True)

plt.show()

writervideo = animation.FFMpegWriter(fps=60)
anim.save('Solar_System_2.mp4', writer=writervideo)
plt.close()
```