Python simulation of orbital motion

TAEWOO LEE (s1711673)

Introduction

In this project, I set the simulation of celestial bodies in the solar system and satellites in a two-dimensional vector space. After I set up the simulation, I simulated to observe the periods of each planet and the energy conservation of total energy over time and to seek the closest approximation of satellite trajectory to Mars.

Methods

My project file consists of 5 files of codes. In this section, I will carefully state my approach and reason for using a method.

First, *'bodies.csv'* is the data list of heavenly bodies for use. The csv means that this data is a value separated by a comma. As you can see inside the file, I write the name, mass, orbital radius, colour, and radius of the planet itself, and they are separated by a comma. Many asks that why use .csv instead of .txt. This is because originally thought that the .csv file is handier to convert the data into the list. Yet, at the very end of the project, I realized that there is no significant difference in function between .csv and .txt, as far as I understand. Yet, using .csv was a good practice to learn python.

Second, *'Planets.py'* is an object class that gets the name, mass, colour, and radius as a constructor. As you may easily notice, these values came from the *'bodies.csv'* I set up previously. I made this class to facilitate the storing of the value of many planets, and this class could work as a blueprint for the flexible addition of other planetary objects. Not only storing the data from the data sheet, but this class also has another crucial function: storing position, velocity, acceleration and kinetic energy data. I used this class as flash memory, or RAM, to store fleeting data of position, velocity, and acceleration for an integrator, which I later explain, to smooth the process of animating celestial objects. In the first two-thirds of the preparation period, I tried to build the vector list to store every position, velocity, and acceleration vector over time. Yet, I found that the result is overly bulky and too much complicated, so I changed my tactics to make a fleeting storage variable. Thanks to the online lecture and aid of my fellow friends.

Third, *'Simulation_final.py'* is a core file that runs the simulation. As a constructor, it receives the number of iterations and a time interval as dt. As aforementioned in the introduction, this file process the orbit animation of planets, the orbital period of planets, the total energy of the system, and the closest distance and time that the Martian probe reaches to the central point of Mars. Let me give you a little bit more detailed look at each process and reason for such decisions.

Reading input data is of utmost importance in this simulation. In the process of reading data, I decided to integrate the data reading and apply each data to objects by calling *'Plantes.py'*. There is a reason I took this approach. I could reduce several processes to save data assigned to each planet into a single function. Furthermore, it made the code more flexible for various planets, so I can freely add more bodies into the datasheet to add to the simulation without any problem.

The orbit animation is the alpha and omega of this simulation. The function works for an orbital animation is comprised of three parts: the Beeman integrator, the iteration process, and the Funcanimation. First, the Beeman integrator is a simplistic integrator to calculate the position and velocity of the next time step. The reason for using this is so simple the integrator is what is given to me for calculation. Yet, because this is a simplistic integrator, the orbital motion is not as equivalent to the celestial orbits in our real-life system. I will comment further in the result and discussion section. Second, the iteration process is of maximum importance in this simulation. This part gives an order of calculation in every repetition. The cause to use this function is that I could simply set up the entire process of iteration in a single module. Hence, I can bring the process wherever I want just by calling a function. Last, the Funcanimation is a part that simply helps the iteration and generates the animation. I used this function because I must generate the animation, yet there is one more useful function; the animate part of the Funcanimation has its repetition function, so I could set up the continuous calculation by calling the iteration process function inside, so the order of calculation is executed continuously if animation is generated. This is very useful since I don't have to make an unstable while and loop for continuous calculations.

In the orbital period part, I had to make a function that temporarily collects the two points of y positions before and after the update of position. The reason is simply a trigonometric fact. In a circle of anti-clockwise motion, the orbit completes when the y value of the positional vector changes from negative to positive. After I collect the position, I made a function which is checking whether the two values are changed from negative to positive and stored the time in a list. The method to store the period is simply appending it to the list. This could've been done since I build the object array in order of inner planet to outer planet and, in fact, the inner planet orbits faster than the outer planet so the period is correctly stored in the right order. However, in this project, I didn't have enough time to elaborate on the output. Hence, I left the period values in a file called *'period.txt'*.

In the total energy of the system, I made a function that calculates the total kinetic energy and potential energy of the system and records the sum of total energy. Because I must plot the total energy over time, I put these functions inside of the iteration process function, so every change of total energy is stored in a file called *'energy.txt'*. Furthermore, I made an external program that generates the plot of energy over time, called *'Energy_simulation.py'*. This is because, in the *Spyder*, two figures are not able to run in a single main function. Hence, I had to make a stand-alone program just for running the energy plot.

In the process of calculating the closest point to the Mars and time taken to reach by the satellite, I had to make a function to calculate and store the magnitude of the difference between Mars and the satellite over time. I approached the case by importing the function of calculating and saving the absolute distance between two positions inside of the iteration process function. I also made a function to write and store the distance values inside of a file named *'distance_diff.txt'*. Then I made a stand-alone program, called *'Journey_time_and_the_closest_distance.py'*. to run find the closest distance among a plethora of saved distance data and calculate the time taken. I did this because there was some indexing error when I run the code on the main function. It runs flawlessly when I made a stand-alone program.

Results and Discussion

As a result, I expected the four results from the simulation: the orbital motion animation, the orbital period of planets, the energy conservation over time, and the minimum distance and time taken by the Mars probe.

First, animation for heavenly bodies is generated as follows:

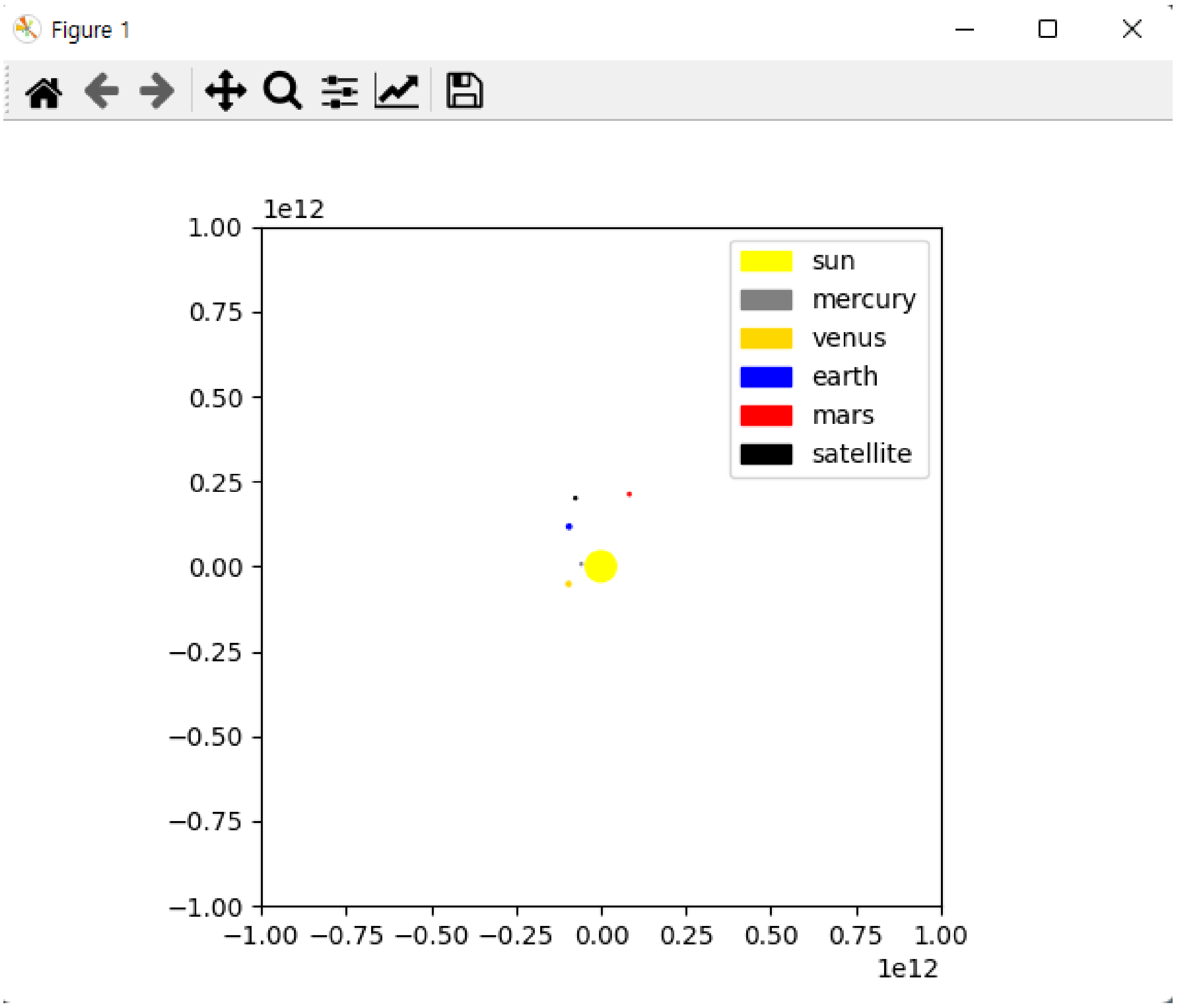
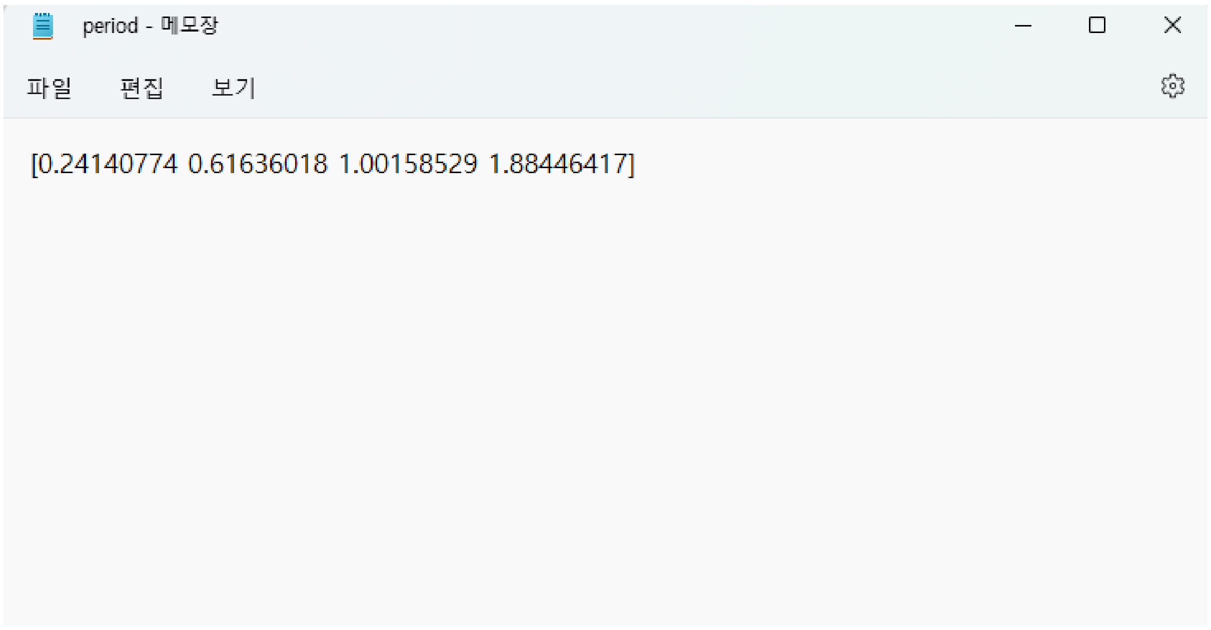


Figure 1

The colour plots inside the legend indicated the name of each object. The size of the Sun is smaller than the actual size and the size of the satellite is bigger than the actual size. This is because of more readability.

My result for an orbit animation is successful. Except sometimes code spits out some enigmatic error and staggered. Yet, this error has been solved by restarting the kernel, so I conclude that they are not critical.

However, the physics of this animation is quite different from the actual orbit of bodies in the solar system. In real life, the orbit of each body has a certain amount of eccentricity, so the shape of the orbits is not a perfect circle like my simulation but rather an ellipse. After a fair amount of contemplation, I conclude that this is because I assume all bodies are perfect spheres with a given radius and limitation of the simplistic integrator, the Beeman integrator.



```
[0.24140774 0.61636018 1.00158529 1.88446417]
```

Figure 2

This is the period of the orbit. Reading from left to right, it is the orbital period of Mercury, Venus, the Earth, and Mars. These values are given in an Earth year. Compared to the values I searched for on google, my orbital period was equivalent to the actual orbital period. The actual period from Mercury to Mars is 0.24 years, 0.616 years, 1 year, and 1.88 years.

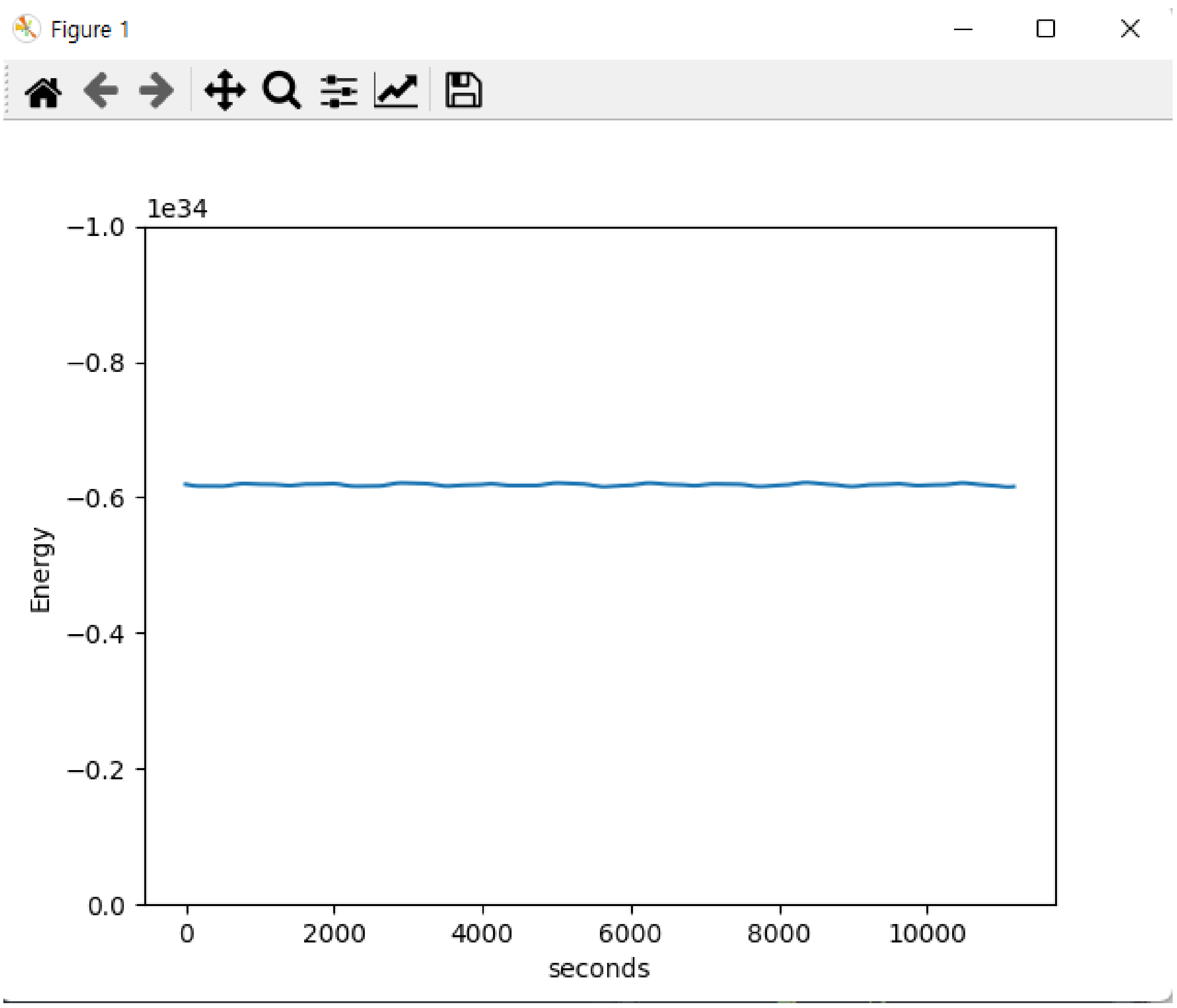
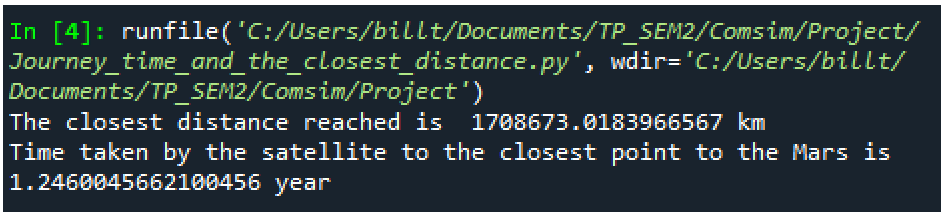


Figure 3

This is the energy against the time plot. The energy given in the plot is the total energy of the system.

It seems a bit wobbly to say perfectly constant, yet with further iterations, the energy line would be smoothed out into a clean line. Therefore, the total energy of the system seems conserved in my simulation as much as in real life.



```
In [4]: runfile('C:/Users/billt/Documents/TP_SEM2/Comsim/Project/
Journey_time_and_the_closest_distance.py', wdir='C:/Users/billt/
Documents/TP_SEM2/Comsim/Project')
The closest distance reached is  1708673.0183966567 km
Time taken by the satellite to the closest point to the Mars is
1.2460045662100456 year
```

Figure 4

This shows the minimum distance between the satellite and Mars and its time. According to the internet, Mars’s sphere of influence is 5e5 km. It seems my closest distance is out of the influence sphere. Therefore, the satellite is failed to orbit around Mars. Furthermore, in comparison with the actual travel time taken by the recent Mars probe, the Perseverance, the time taken by my satellite to reach the closest approach is longer than the Perseverance. Moreover, my satellite never returns to earth but orbits around the sun in between the orbit of Earth and Mars. According to NASA, the time taken by the Perseverance to reach Mars was about 7 months, yet my satellite takes 1.24 years to reach. As I found the closest distance this far, I could brute-forced the initial velocity value of satellite launch to seek more feasible trajectories. Yet, I could use more highly acclaimed and accurate dynamics called *the Hohmann transfer orbit* to calculate the orbit between the Earth and the target planet, for example, Mars. Furthermore, because the mass and size of the satellite is varying throughout the launch, the approximation that I made, which is that the mass and size of the satellite are consistent throughout the simulation, could cause a fair amount of uncertainty. Yet, my guess could work only if I launch the payload via a mass driver, so I can convey a satellite like *From the Earth to the Moon (1865),* written by *Jules Verne* instead of a conventional rocket system.

Conclusions

Most of the code works flawlessly and shows expected results, such as correct orbital period and conserved total energy of the system. Yet, this simulation is not perfect and complete. If I had more time, I would've made another external program to generate the outcome of the orbital period more elaborate. Furthermore, the satellite trajectory was a failure. Hence, if I had more time, I would've worked on the function that only works for the satellite by using the *Hohmann transfer orbit.* In addition, the bodies in the systems don't have any collision physics. Hence, I would like to add a collision feature in a near future, so that I could test the asteroid collision to planets.

References

Computer Simulation Course Book. (n.d.). Retrieved from ComSimLearn2122.

Google Search Engine. (n.d.). Retrieved from Google.

NASA. (n.d.). *MARS 2020 mission perseverance*. Retrieved from MARS 2020 mission perseverance: https://mars.nasa.gov/mars2020/

NASA. (n.d.). *Planetary Fact Sheet - Metric*. Retrieved from NASA: https://nssdc.gsfc.nasa.gov/planetary/factsheet/

Wikipedia. (n.d.). *Hohmann transfer orbit*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Hohmann_transfer_orbit