Simulation and several Experiments in the Solar System

Author: Yuxuan Qian Date: 05/04/2023

Student number: s2286473

This project aims to simulate the motion of the 6 inner planets of the solar system in two dimensions and to predict launch conditions for satellites to successfully reach planets.

In the package that I uploaded, there are 4 Python files and one text file. Planets.py contains all methods that update data for a single planet. Solar.py contains all methods that update data for whole solar systems. Experiments.py includes code for 5 experiments. And a code running files run.py. The parameters_solar.txt stores all the parameters of the solar system, such as the number of iterations, time-step in earth year, gravitational constant G in unit AU^3/(M⊙year^2), and name, mass, orbital radius, and color of each planet.

Planets.py

This file includes 2 classes, Planet_Beeman(object) uses Beeman algorithm to update data of planets, and Planet_Euler(Planet_Beeman) uses Euler algorithm.

In the first class, the initialise() method creates an initial state of the planet. The initial position is just [orbital radius, 0]. The initial velocity is calculated by $\sqrt{\frac{\boldsymbol{GM}}{\boldsymbol{r}}}$, where M is the mass of the Sun and r is the orbital radius, and it's tangent to the position.

updatePos() update position by $\underset{r}{\rightarrow}(t+\Delta t) = \underset{r}{\rightarrow}(t) + \underset{v}{\rightarrow}(t)\Delta t + 16\left[4\underset{a}{\rightarrow}(t) - \underset{a}{\rightarrow}(t-\Delta t)\right]\Delta t^2$, where $\underset{r}{\rightarrow}, \underset{v}{\rightarrow}, \underset{a}{\rightarrow}$ represent the position, velocity, and acceleration vectors for the motion. updateVel() update velocity by $\underset{v}{\rightarrow}(t+\Delta t) = \underset{v}{\rightarrow}(t) + 16(2\underset{a}{\rightarrow}(t+\Delta t) + 5\ \underset{a}{\rightarrow}(t) - \underset{a}{\rightarrow}(t-\Delta t)$, where $\underset{a}{\rightarrow}(t)$ is updated acceleration, so we need to obtain a new acceleration first. updateAcc() returns acceleration influenced by another planet, which is calculated by $a = G\frac{M}{\left|\overrightarrow{r(t)}\right|^3}\underset{r}{\rightarrow}$, where M is the mass of another planet.

newYear() method checks whether the planet travels a round and returns True or False. If the y-coordinate of the old position is negative and the new position is positive, it completes a circle. kineticEnergy() method returns K.E calculated by $\frac{m}{2}\underset{v}{\rightarrow}\cdot\underset{v}{\rightarrow}$.

By using heritage, Planet_Euler(Planet_Beeman) is a class that updates data by Euler algorithm with all other methods same with Planet_Beeman(). So the position is updated by $\underset{r}{\rightarrow}(t+\Delta t) = \underset{r}{\rightarrow}(t) + \underset{v}{\rightarrow}(t+\Delta t)\Delta t$, and velocity is updated by $\underset{v}{\rightarrow}(t+\Delta t) = \underset{v}{\rightarrow}(t) + \underset{a}{\rightarrow}\Delta t$.

Solar.py

This file includes a class Solar(), which calculates the total energy and simulates the motion of 6 inner planets of the solar system for 100 Earth years.

In __init__ (algorithm), the parameter 'algorithm' refers to which algorithm, Beeman or Euler, that will be used in the simulation, and I default it to Beeman. I fetch all the data needed from the parameter file and construct each planet as an object of the corresponding class of that algorithm, and initialise each of them.

To make an animation, I plot the initial positions of all planets first, then plot positions for each time-step by using FuncAnimation() with function animate(i), where parameter i is an increment that represents the number of iterations. In animate(i), use updatePos() and updateVel() to obtain an updated position for each planet and add it to patches. Then check whether each planet completes a circle and print the time in Earth year, time = time-step*i. Also, print the total energy of the simulation for each Earth year. Finally, return all patches. I also add some code to compare the distance between the satellite and Mars to obtain a minimum value, this part only works when there is a satellite, and skip the test for the new year and total energy.

Total energy during the simulation is the sum of kinetic energy plus potential energy. Each iteration calculates the K.E of one planet by using kineticEnergy() method in Planet_Beeman(object) class and calculates each P.E of other planets influenced by this planet using the equation P.E = $-G\frac{Mm}{r}$, where M is the mass of this planet, m is mass of other planet and r is the distance between them. Adding all of the K.E and P.E together gives the total energy of all planets.

Experiments.py

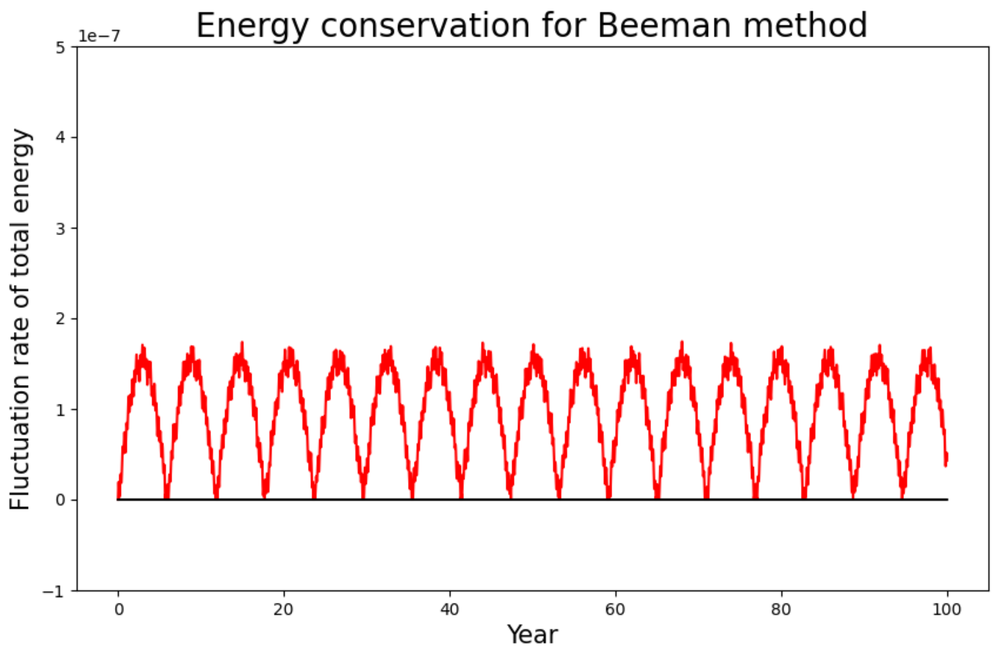
In the __init__() method, I fetch updated data for each time step just like the way in simulation but store all the data I need in the following experiment in separate lists. So I don’t need to rerun the same code for each experiment. In specific, I store the positions of all planets for all iterations in ‘self.r’, periods for all planets for all iterations in ‘self.period’, and total energy by using both Beeman and Euler algorithms for all iterations in ‘self.totalE_B’ and ‘self.totalE_E’ respectively.

The first experiment compares the actual and simulated periods. By Kepler's Third Law, the actual period of a planet is calculated by $2\pi r^{\frac{3}{2}}\sqrt{GM}$, where M is the mass of the Sun and r is the orbital radius. Since the orbital radius varies insignificantly, so I just take r to be the initial orbital radius for each planet and assume the actual period is a constant. And using the data from the self.period to calculate the errors and percentage errors for each iteration.

The average error between actual and simulated is 0.17 days for Mercury, 0.19 days for Venus, 0.22 days for Earth, 0.44 days for Mars, and 12.67 days for Jupiter. And the percentage error is 0.199%, 0.083%, 0.060%, 0.064%, and 0.293% respectively. So the errors are small compares to the actual periods, which means the simulation is highly accurate.

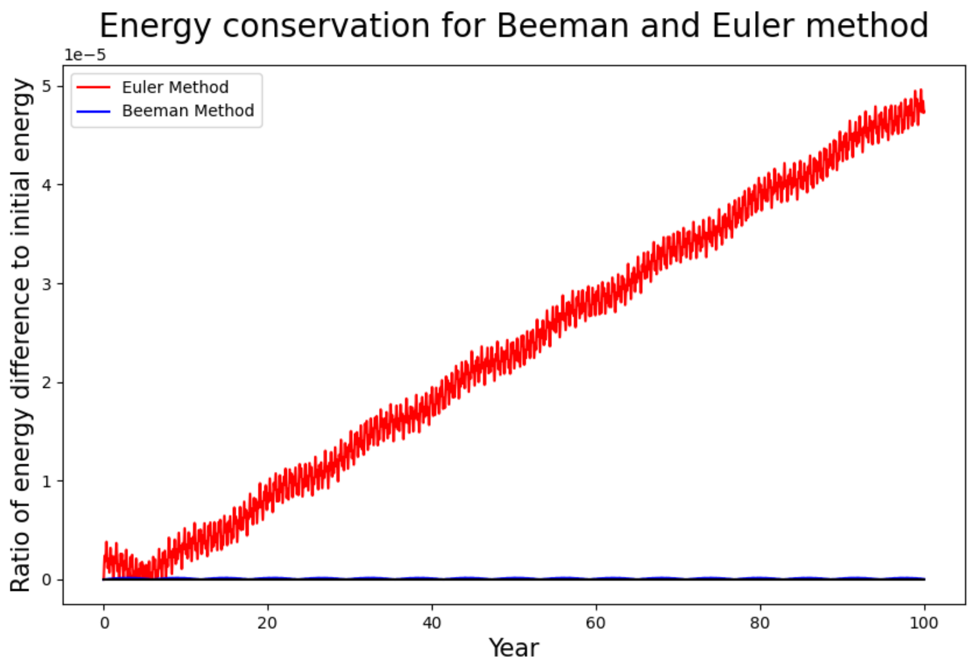
In addition, If we narrow down the time step, the error should be much smaller. i.e. if I take the time step to be 0.0001 years, the percentage error is 0.033%, 0.040%, 0.0453%, 0.063% and 0.293% respectively. So the influence by time step is more obvious for inner planets.

The second experiment checked whether total energy is conserved by the Beeman algorithm. Using the data from self.totalE_B to calculate percentage errors of total energy at each time-step based on initial total energy. And plot it graphically.



From the graph above, the x-axis represents years, the y-axis represents the difference between energy at each year and initial energy. A more explicit expression is given by $y_i = |E_i - E_0|/E_0$, where i is the number of iterations, and E is the total energy. The y_i I obtained is oscillating between 0 and 1.8×10^{-7}, which means energy fluctuated within 0.000018%. And this may caused by the time step I choose. If I narrow down the time step, the rate would be even smaller. I also test for 0.0001 time-step, and the percentage error is less than 0.000002%. Therefore I conclude that energy is conserved for the Beeman method.

The next experiment is energy conservation for the Euler algorithm and compare with the Beeman algorithm. The method is similar to the last experiment, I just plot percentage errors for both algorithms on the same graph.



The red line is the percentage error of total energy based on initial total energy by using the Euler algorithm, and the blue line is that for the Beeman algorithm. This graph shows the total energy using the Euler algorithm increasing significantly compare to the Beeman method. i.e. At year 100, the percentage error of the Euler method is approximately 250 times larger than that of the Beeman algorithm. So I can simply conclude that the Beeman method gives a more accurate simulation than the Euler method.

In the fourth experiment, I want to find out how close to Mars my satellite gets and whether it gets back to Earth.

I treated the satellite as a planet and put it into the solar system. Then simulate the animation of the system by using the run() method in the class of Solar(), whenever you see the satellite getting close to Mars, you can close the animation, and results will be shown.

To obtain the minimum distance, I compare the distance between Mars and the satellite for every two iterations and store the smaller one. The method is in animation() in Solar.(object).

The initial position of the satellite is [Earth position + 0.001, 0] since it launched on the surface of the Earth. According to the data from 'Mars 2020 Mission, Perseverance Rover' by NASA at <Mars 2020 Perseverance Rover - NASA Mars>. I take the mass of the satellite to be 1025 kg, which is 1.71×10^{-22} of mass of Earth, and the speed is 24600 mph (11 km/s). For this speed, I got the closest distance when velocity is [11, 0], and the result is 5.1211×10^{6} km at 0.541 years(197.5 days). As the x-component of velocity decreases, the closest distance that the satellite can approach increases, i.e. For v = [10.9, 1.48], the closest distance is 36.2537 M km.

The NASA Perseverance probe took 203 days, which is very close to my simulated value with a 2.7% error. This may caused by the time step I choose. If I take the timestep to be 0.0001, the estimated time will be 201.8 days.

To check whether the satellite returns, I calculate the distance between the satellite and Earth, if it is less or equal to 0.001 AU, it means the satellite returned to Earth. In my experiment, the satellite never returns to Earth in the first 100 years, but it is close to Earth with 0.045 AU in the 86th year.

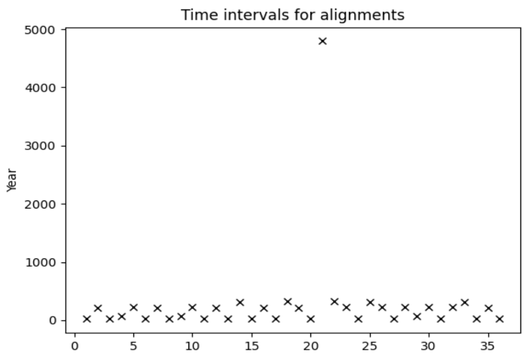
The last experiment tests planetary alignment for the inner 6 planets (out to and including Jupiter). I have taken the instance of planetary alignment to be when planets align themselves to within ±5°. I take a baseline of alignment to be the line between Sun and Mercury, then check the angle for the line between other planets to Sun with the baseline. The angle is calculated by $\cos^{-1}\frac{\underset{r1}{\rightarrow}\cdot\underset{r2}{\rightarrow}}{\left|\underset{r1}{\rightarrow}\right|\left|\underset{r2}{\rightarrow}\right|}$, where $\underset{r1}{\rightarrow}$ is the baseline and $\underset{r2}{\rightarrow}$ is another line. Alignments happen if and only if the angle for all planets within (-5°, +5°) or (175°, 185°). And I exclude the alignments for consecutive time-step to avoid double counting.

In the results, 2 alignments happen within 100 years, one is the beginning and another one happens in the 22.359th year. But this method only tests alignments for 100 years. To solve this, there is another method Alignment2() which takes an input parameter, years, so it can show the number of alignments within that many years.

The second method is similar to the first one, I calculate the time interval between every two alignments in addition. When you input a large number, it takes time, since this method test alignment for each time step.

So I test for 10000 years in advance and observed 37 alignments. The average interval is 272.43 years and is 143.17 years if excluding the prominent one.

From the graph on the right, the time intervals between every two alignments seem randomly distributed. But there might be a pattern if we observe more alignments.



In conclusion, this project has successfully simulated the motion of the inner planets in the solar system and predicted launch conditions for satellites to reach other planets. And the simulated periods is very close to the actual periods. The Beeman algorithm conserves total energy pretty well. Overall, this project has contributed to a deeper understanding of the dynamics of the solar system. The ability to accurately simulate the motion of planets in two dimensions and predict launch conditions for satellites has numerous practical applications, including space exploration and satellite communication.