Simulation of the Inner Planets of the Solar System and a Probe Launched from Earth to Mars

M. W. Giles

Introduction

This project simulates, in two dimensions, the orbits of the inner planets (Mercury, Venus, Earth and Mars) around the Sun. The simulation can be run with or without a graphical animation. It runs for a time which is specified at input by a combination of timestep and number of iterations. At the end of a run the orbital periods of each body in the simulation are printed to the terminal, in units of Earth years. The experiment compares these calculated values to known values.

The simulation calculates the total energy of the system, a combination of the kinetic and potential energies of each body and saves the value of this to a text file at regular intervals. I also determine whether energy is conserved by my simulation.

The project also simulates a satellite which can be launched from any location with a specified starting velocity. In this case I have sent the Viking Probe from 100 km above the surface of Earth to fly past Mars. Specifying Mars as the target and Viking as the probe, the simulation returns the distance and time of closest approach between the target and probe within the running timeframe. Part of my experiment is to determine whether the values I generate from my simulation could be obtained by a real probe.

Methods

The core project code is split into two components: a *CelestialBody* class and a *Simulation* class. The *CelestialBody* class represents any planet, sun or satellite being simulated. I chose to represent bodies with a class so that new ones can be easily added and all bodies can be processed by other classes using common functionality. The class is initialised with the body's mass, name, colour, nominal size, initial velocity and initial position. The colour can be chosen by the user and is used for the circle which represents that body in the animation. The nominal size indicates how large the body should appear in the simulation; I would expect this to range from 0.1 to 2. The *CelestialBody* class also stores a current and previous acceleration for the body, which are initialised to the zero vector, and an orbital period, which is initialised to None and calculated later. All attributes are stored as private variables and relevant getters and setters are provided, I chose this to adhere to good and secure programming practice, making the program more robust and harder for a user to break.

As the simulation is representing a many-body problem, we cannot calculate positions analytically and so use the three-step Beeman integration scheme. The function to update the acceleration of a *CelestialBody* takes as input a new acceleration and a Boolean to say whether this update is during the first iteration. The function sets the previous acceleration to what is the current acceleration, and sets the current acceleration to the value passed in. If the function is called during the first iteration, we have no previous acceleration, so this can be approximated to the current acceleration.

The function to update the velocity of a body follows the algorithm specified by the Beeman integration scheme exactly, taking the timestep and the next value for acceleration as input.

The function to update the position of a body takes as input the timestep and iteration number. The new position is calculated again following the Beeman algorithm. This function also checks whether the period for the body has been calculated, and if it hasn't and if the y value for the position changes from positive to negative (a full circle is completed assuming the body starts on the x axis), then the orbital period is calculated.

Finally, there is a function that calculates the kinetic energy of the body based upon its speed. I chose to contain as many body specific calculations as possible inside *CelestialBody*; this makes expansion of the code which uses bodies for different purposes much easier.

The other class is for the simulation. It stores the number of iterations for which the simulation should run, the timestep of each iteration, and a list of *CelestialBody* objects. The attributes to create each *CelestialBody* are read in from a file. *Simulation* also stores values for the distance and time of closest approach between the target and probe, which are initialised to None.

The first line of the data input file for the project contains the timestep and number of iterations, separated by a space. The *read* function takes this data from the first line and processes all subsequent lines as bodies consisting of space separated initialisation values for a *CelestialBody*.

The function *calc_acceleration* returns the vector acceleration of the body passed as input. It sums the components of acceleration from every body other than itself, which are calculated using Newton's Law of Gravitation (Britannica, n.d.).

The most important function in the class is *step*, which is all the code that must be run in an iteration of the simulation. Its main purpose is updating the acceleration, position and velocity of each body. It first iterates through each body and updates its acceleration. Then iterates through each body and updates its position. These must be performed in separate for loops as the acceleration is dependent upon the velocity. Finally, it iterates through each body a third time and calculates (but doesn't update) the acceleration of the body, and then updates the velocity, passing it this next acceleration. The *step* function also calculates the distance between the target and probe and updates the distance and time of closest approach if the new value is lower.

Simulation also has functions which calculate the kinetic, potential and total energy of the system as a whole. The *without_animation* function runs the simulation, calling *step* the specified number of times and writing the total energy to a file at regular intervals. I added this function so that data could be collected from the simulation without having to wait for an animation to run. The *animate* function does the same, but every iteration updates the positions of the circles which represent each *CelestialBody*. I chose to animate on the fly so that it wouldn't be necessary to store data for the position of every body in every iteration as this would take extra time to compute and be extremely space inefficient, especially when there can be a large number of iterations.

All of the functions described for *Simulation* are private and not accessible by the user. This adds protection to the code and less likely to be broken and ensures that I can be more confident it will always run as I have intended it to – this is important for it to follow the Beeman scheme correctly and produce accurate results. The final function is *run* which is called by the user to start the simulation. It takes as arguments a scale on which to show the animation, the file in which the energies should be stored, the indexes of the target and probe bodies in the file and a Boolean to tell the simulation to animate or not. It calls either *animate* or *without_animation* and prints the orbital periods of every body at the end. If the simulation is

to be animated, it creates a circle object for each *CelestialBody* with the initial values accessed using the body's getters. I chose for the target and probe to be specified when calling *run* so that the same *Simulation* can be re-executed analysing different satellites each time.

To generate results for the experiments I created a separate file and class to generate some graphs. One function graphs the energy against time (in Earth days) from data stored in a text file. The other runs the simulation with a variety of initial x velocities (iterating over an inputted range in specified step sizes) for the Viking Probe and plots its closest approach distance to Mars against x velocity and time of closest approach to Mars against x velocity. The initial y velocity is fixed at 30,000 ms^{-1}, approximately the same as Earth's initial y velocity. I chose to program this generically so that it could be used for any combination of target and probe.

Results and Discussion

First, I will discuss the results produced for the orbital periods. These were generated using a timestep of 43200 seconds or half a day on Earth, this is sufficiently smaller than the orbital periods and so I expected reasonable accuracy.

Planet	Calculated Orbital Period (Earth Years)	Known Orbital Period (Earth Years)	Percentage Error
Mercury	0.239561944	0.2408467	0.533433145
Venus	0.613278576	0.61519726	0.311881051
Earth	0.999315537	1	0.06844627
Mars	1.882272416	1.8808476	0.07575394

Figure 1: A table showing calculated orbital periods of the planets relative to Earth's and how accurate these are

It is clear that the simulation produces accurate periods with errors less than a percent, for all planets. It seems that the planets farther away from the Sun (Earth and Mars), thus with longer orbital periods, produced values even closer to the true ones with errors of less than a tenth of a percent. As the inner planets have approximately circular orbits, Earth's has a tiny eccentricity of 0.0167 (Nelson, 2021), our method of calculating initial positions and velocities, which assumed uniform circular motion around a stationary body, is reasonable. If we attempted to calculate the initial values for a body with a very eccentric orbit, say a comet, I would expect a greater percentage error in the results. The assumption that the Sun is stationary is valid as it is so massive compared to the inner planets.

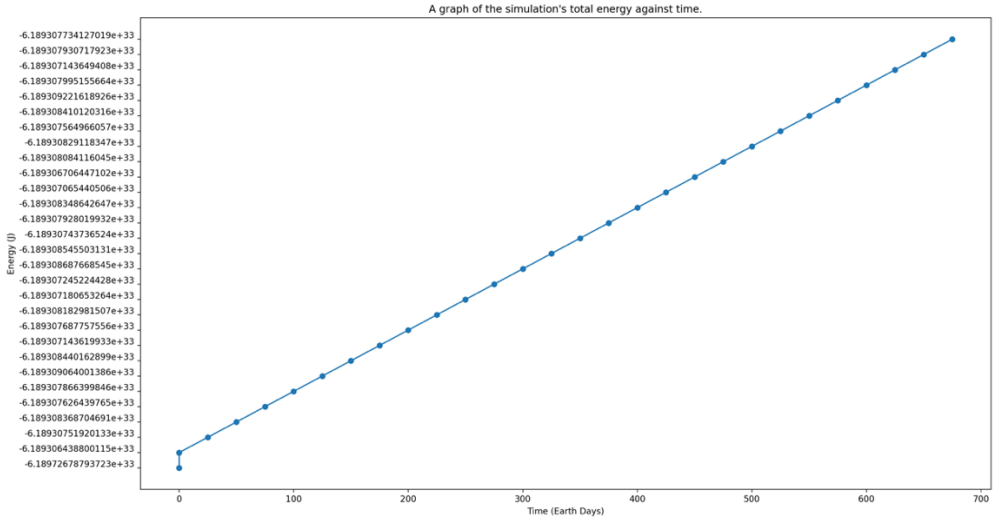

Figure 2: A graph showing total energy of the simulation over a period of 275 days

Running the simulation with the time timestep of 43200 seconds, for 1400 intervals, produced values for the total energy of the system as plotted above. We see an increase in energy over time, which is not what we would expect in a physical system where energy ought to be conserved. However, upon closer analysis, comparing the value for total energy at the start of the 275 days of simulation to that at the end, we see that there has only been a 0.00677% change. I would say that this is reasonably negligible and conclude that my simulation does conserve energy. In fact, I would not expect complete conservation as the Beeman algorithm has an error term for higher orders of the timestep which I omitted from the code (Omelyan, 1998).

Finally, I attempted to determine the optimum initial velocity for a satellite launched 100 km above the surface of the Earth, also beginning on the x axis. I assumed that its initial velocity in the y direction would be approximately the same as the Earth, so I fixed it at 30,000 ms^{-1}. To find the best value for the x component I began by varying it in steps of 1000 ms^{-1} from 8000 ms^{-1} to 20,000 ms^{-1} and plotted the distances and times of closest approach against this. The distances are given relative to the smallest value.

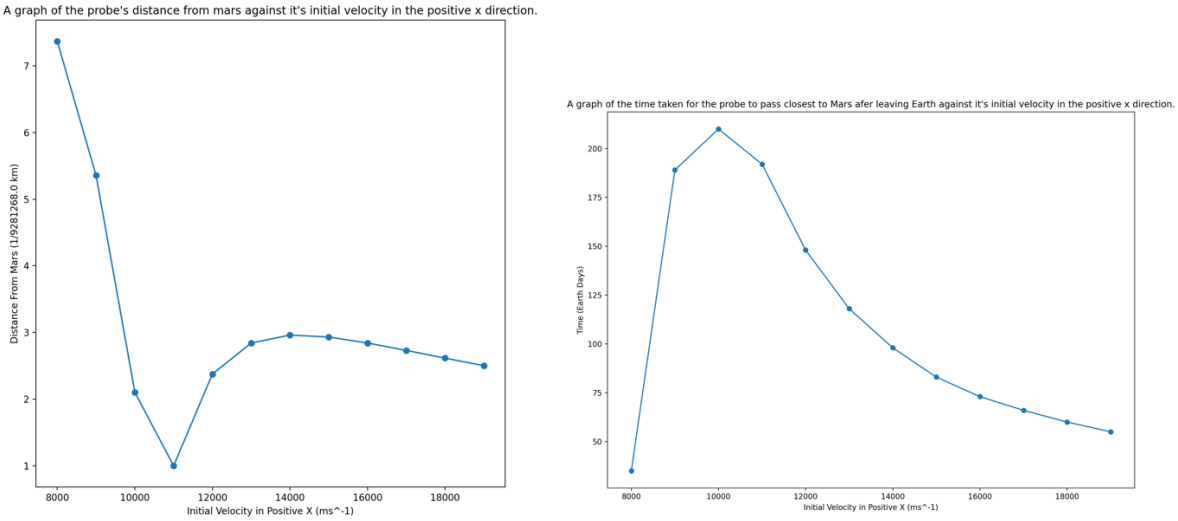

Figure 3: Graphs of distance and time of closest approach for x velocities from 8000 to 20000 ms^{-1}

From these graphs we see that the distance of closest approach is smallest somewhere between 10,000 ms^{-1} and 11,000 ms^{-1} and that the time taken is greatest in this region. For higher initial x velocities, we still get relatively close and also in a much shorter time, however, these seem unrealistic to achieve. In order to accurately determine where this point was, I plotted again using the range I just specified.

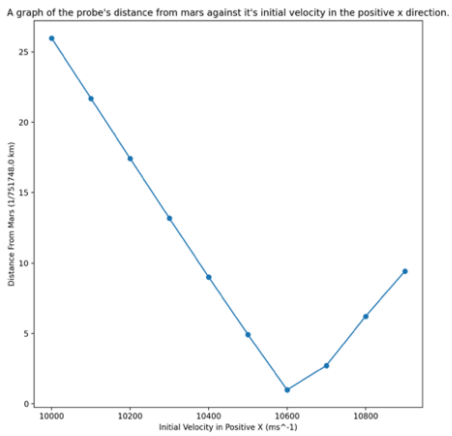

Figure 4: Distance of closest approach for x velocities from 10000 to 11000 ms^{-1}

For even greater precision, I produced one further set of plots as shown below, this time between 10,600 ms^{-1} and 10,700 ms^{-1}.

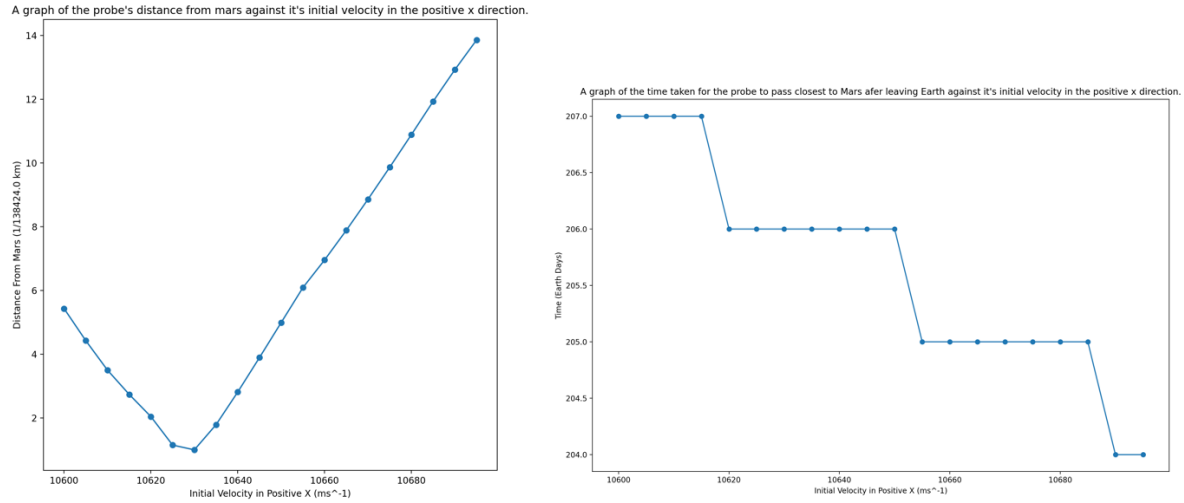

Figure 5: Graphs of distance and time of closest approach for x velocities from 10600 to 10700 ms^{-1}

We see from the y-axis title that the minimum distance to Mars reached by the probe is 138,424 km above its surface for an initial x velocity of 10,630 ms^{-1}. This separation is just 0.00022% of the distance from Earth to Mars (NASA, 2020), even when they are at their closest, so this is an extremely near flypast. Therefore, I can deduce that I have been able to find an initial velocity for the Viking Probe which sends it close to Mars. To determine how realistic this is I will first look at the time. With this optimum initial velocity, it takes 206 Earth days (to the nearest day) for the satellite to travel from Earth to Mars. NASA estimate the journey should take around seven months or 213 days (NASA, 2020). Therefore, although slightly faster, our calculated timeframe is entirely possible for the journey to be made in.

The magnitude of the optimum initial velocity is given by $\sqrt{10630^2 + 30000^2}$ ms^{-1} which is 31,828 ms^{-1}. For NASA's Perseverance mission, the spacecraft departs Earth at around 11,000 ms^{-1} (NASA, 2020); although this is approximately three times lower than the figure I calculated, the order of magnitude is very much correct and so a realistic value. Also, my simulated Viking Probe starts 100 km above the surface of the Earth and so could have gained some further speed since departing Earth.

Conclusions

The final implementation creates a simulation which accurately simulates the inner planets of the solar system. The orbital periods are calculated to a high degree of accuracy, and this could be further improved by simulating with a smaller timestep. There is negligible change in the total energy of the system, therefore, I would say that my simulation conserves energy. The small increase in energy that is present could be decreased by making the implementation of the Beeman algorithm more precise, by including terms of higher order timestep.

The simulation of the Viking Probe travelling from 100 km above Earth to Mars produced a realistic time for the journey and a close flypast. The speed was somewhat higher than expected; perhaps this could be optimised by varying not only the initial x component of velocity, but also the component in the y direction. Under graphical observation of the probe, it does not return to Earth within eight years for this initial velocity, however, it does remain within the inner planets so it is not impossible for it to eventually return to Earth.

I am satisfied with the graphical animation as it shows clearly the orbits of the inner planets, and is easy to follow with colour coding, relative sizes and a legend. If I were to expand the code further, I would consider animating and simulating in three dimensions. It would also be useful to include the outer planets and determine whether the accuracy of my project holds for these also. Overall, I am very pleased with how the project turned out and believe that it is capable of providing useful and usable data regarding motion for the inner solar system, within a small margin of error.

Bibliography

Britannica. (n.d.). *Newton's law of gravity*. Retrieved April 7, 2021, from Britannica: https://www.britannica.com/science/gravity-physics/Interaction-between-celestial-bodies

NASA. (2020). *Cruise*. Retrieved April 8, 2021, from Mars 2020 Mission Perseverence Rover: https://mars.nasa.gov/mars2020/timeline/cruise/

NASA. (2020, October 1). *Mars in our Night Sky: Mars Close Approach to Earth*. Retrieved April 8, 2021, from Mars Exploration Program: https://mars.nasa.gov/all-about-mars/night-sky/close-approach/

NASA. (n.d.). *Planet Compare*. Retrieved April 6, 2021, from NASA Science: Solar System Exploration: https://solarsystem.nasa.gov/planet-compare/

Nelson, B. (2021, January 25). *Everything You Need to Know About Earth's Orbit and Climate Change*. Retrieved April 7, 2021, from Treehugger: https://www.treehugger.com/everything-you-need-to-know-about-earths-orbit-and-climate-cha-4864100

Omelyan, I. P. (1998). Numerical integration of the equations of motion for rigid polyatomics: The matrix method. *omputer Physics Communications, 109*(2), 171-183.