



THE UNIVERSITY
of EDINBURGH

Planetary Pandemonium - A Solar System Simulation

Exploring orbital dynamics using OOP in Python

Word count: 1,999

**Siddhant Pujni
S2344216**

March 2024

Table of Contents

- 1. Introduction..... 1**
- 2. Background..... 1**
 - 2.1. Beeman Integration..... 1
 - 2.2. Direct Euler..... 2
- 3. Code Design and Implementation.....3**
- 4. Simulation Outcomes..... 6**
 - 4.1. Orbital Periods.....6
 - 4.2. Energy Conservation Comparison.....8
 - 4.3. Planetary alignment..... 9
- 5. Discussion..... 10**
 - 5.1. Orbital Periods.....10
 - 5.2. Energy Conservation Comparison..... 11
 - 5.3. Planetary Alignment..... 11
- 6. Conclusion.....12**
- 7. References..... 12**

1. Introduction

This project uses Python and Object-oriented programming (OOP) to create a 2D simulation of our solar system to better understand aspects of planetary dynamics. The aim is to use this simulation to model the solar system by conducting the following experiments:

1. Calculating orbital periods of simulated planets and comparing to their respective literature values
2. Comparing the performance of *Direct Euler* and *Beeman* integration techniques through energy conservation
3. Checking for instances of planetary alignment up till Jupiter

2. Background

The many body problem is the challenge of evaluating the motion of multiple interacting bodies under the influence of the force of gravity. The ODEs of motion in such a case cannot be solved analytically and hence numerical integration techniques must be used.

2.1. *Beeman Integration*

First we introduce the Beeman technique, which will serve as the primary method for modelling the motion of our planets. It accounts for both the current and previous accelerations when updating the particle positions and velocities ensuring a stable and accurate integration. This is done by the inclusion of a velocity-dependent term in the body's position and velocity as follows:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{6}[4\vec{a}(t) - \vec{a}(t - \Delta t)]\Delta t^2 \quad \{2.1\}$$

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{1}{6}[2\vec{a}(t + \Delta t) + 5\vec{a}(t) - \vec{a}(t - \Delta t)]\Delta t \quad \{2.2\}$$

Where Δt is the small timestep, t is the current time, \vec{r} , \vec{v} , and \vec{a} are the position, velocity, and acceleration vectors respectively.

2.2. Direct Euler

We now look at the Direct Euler method, a simpler technique that updates the state of the system at each time step based on the derivative of that state (Dawkins, 2018). The equations to update position and velocity are below:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t \quad \{2.2\}$$

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t)\Delta t \quad \{2.3\}$$

Direct Euler integration is not computationally intensive to implement, but it may exhibit numerical drift and less accurate long-term behaviour compared to Beeman integration. Conversely, Beeman integration, requiring additional acceleration calculations at each time step, is more computationally intensive but offers improved accuracy and stability in modelling the solar system.

3. Code Design

The provided source code is used as inspiration, following an OOP style with two primary classes: ***Planet*** and ***Solar***.

Instances of the planet class are used to represent celestial bodies for which the position is initialised as a numpy array with the orbital radius as the x-coordinate and zero as the y-coordinate which places the planet on the positive x-axis at a distance equal to its orbital radius. Initial velocity and acceleration are also numpy arrays set using the gravitational force law:

$$F = G \frac{m_1 m_2}{r^2} \quad \{3.1\}$$

where m_i is the mass of the respective bodies and r is the separation. If the ***Planet*** instance is a central body (eg. the sun) then both velocity and acceleration are zero as gravitational effects from all other planets on the Sun are insignificant. Finally, initial velocity is set tangentially to the position and acceleration to incorporate circular motion characteristics. The quantities are then updated using Beeman integration as detailed above. The initial conditions were chosen to aim for a realistic and accurate simulation that captures the gravitational dynamics between celestial bodies as closely as possible.

The **newYear** method tracks a planet's orbit by detecting when its y-coordinate transitions from negative to positive, indicating a crossing of the positive x-axis and a completion of an orbit- it then returns 'True'. Lastly, the **kineticEnergy** method calculates and returns the KE of the planet in joules using *np.dot* to find the dot product of the velocity numpy array with itself.

Solar uses composition to run the simulation of the solar system by reading in orbital and simulation parameters from a *.txt* file and then initialising a list of **Planet** objects based on the data file. The **animate** method updates the positions and velocities by looping over the **Planets** list at each time step which is then reflected in the animation by updating the centres of the patches list to animate.

If **newYear** returns true, **animate** checks if it's the first time for that planet- if so then the current simulation time is stored as the start time of the new year. This is important as the orbital period is calculated using the difference between the current time and the start time of the last orbit, if there is no 'previous' start time the calculation will fail as there is no value to compare to. The data is stored in a list, and then presented in a table using a pandas DataFrame to enhance readability and accessibility.

planetAngle computes the angle between a planet's position vector and the x-axis unit vector via the dot product identity, returning it in degrees. It includes a ValueError check to handle zero magnitude vectors and prevent division errors. Finally, an if statement negates the angle when the position vector y-coordinate is negative because the np.arccos function returns an angle in the range $[0, \pi]$, so this adjustment is necessary for an angle in the range $[-\pi, \pi]$.

Inheritance is then used to create secondary classes **PlanetEuler** and **SolarEuler** which operate in the same way instead now **PlanetEuler** uses Direct Euler integration to update the positions and velocities and a list of **PlanetEuler** objects is created for animation in **SolarEuler**.

Two *.txt* files are created for both Beeman and Direct Euler by **animate** when writing the energy data to file. These files are parsed into two pandas DataFrames using *parse_line* and the *plot_energies* function takes these as an input- using *matplotlib* to plot the evolution of energy over time. Energy is plotted on the y-axis in joules and time in Earth years on the x-axis since data was stored in one Earth year intervals.

Lastly, the run file creates a command line interface for running the simulation of the solar system using either Beeman or Direct Euler. This was done to make the simulation accessible to anyone and no understanding of python would be required to use the program. The file presents a menu of five options allowing the user freedom in what they would like to do.

4. Simulation Outcomes

Figure 1 is a single frame of the Beeman animation- a time step of 0.001 and 1000000 iterations was used. Despite the visual appearance of Mercury colliding with the Sun, it's an illusion caused by the planet sizes and scales necessary for visibility in the simulation.

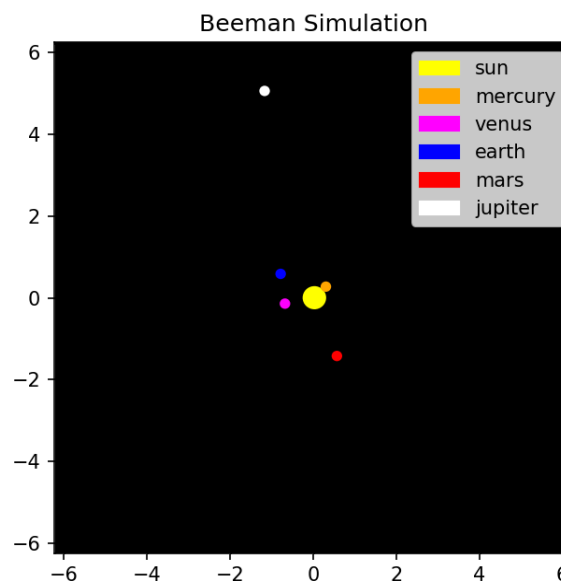
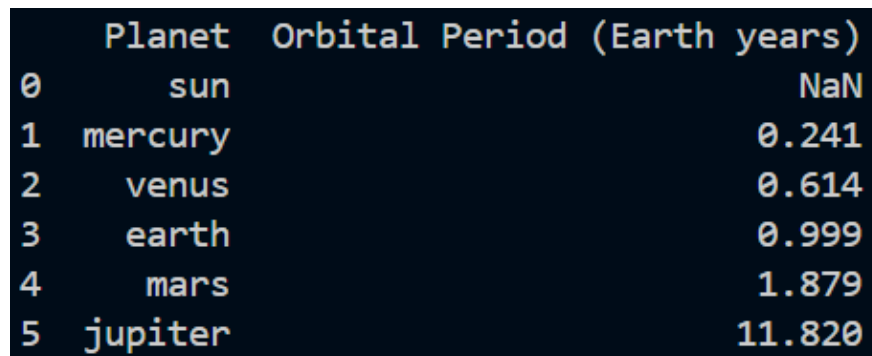


Figure 1: Beeman Simulation used for all experiments

4.1. Orbital Periods

The simulation needs to be run for at least 24 Earth years to obtain a value for Jupiter's orbital period due to the method used. The periods were printed to console as a pandas DataFrame as follows and then transferred to a table with literature values for comparison.



```
   Planet  Orbital Period (Earth years)
0      sun                      NaN
1  mercury                   0.241
2   venus                    0.614
3   earth                    0.999
4    mars                    1.879
5  jupiter                   11.820
```

Figure 2: Console output of Orbital Periods

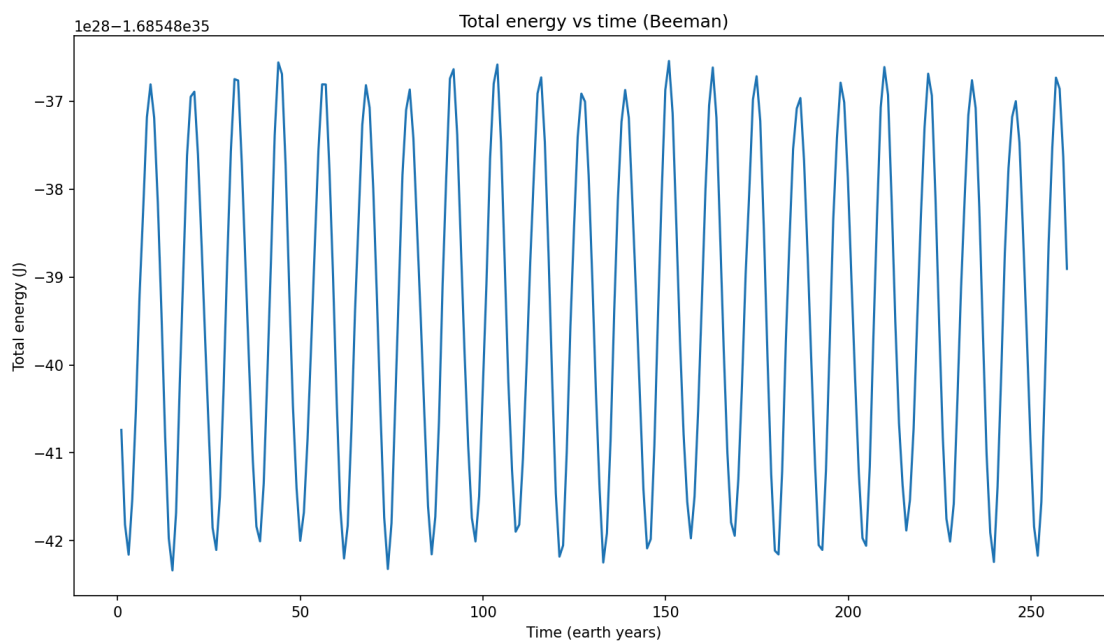
Planet Name	Literature Orbital Period, Earth years (Willaims, 2024)	Simulated Orbital Period, Earth years	Percentage Difference
Mercury	0.24092	0.241	0.033%
Venus	0.61545	0.614	0.236%
Earth	1.0006	0.999	0.160%
Mars	1.8913	1.879	0.650%
Jupiter	11.894	11.820	0.622%

Table 1: Comparison of orbital periods and percentage differences

The comparison between the literature orbital periods and the simulated orbital periods for the planets shows a good overall agreement, with small ($<1\%$) percentage differences observed. Mercury, Venus, and Earth show minimal deviations, showcasing a close match to the expected values. While Mars and Jupiter exhibit much higher deviations, suggesting that accuracy in orbital period decreases with increasing orbital radius or for planets past Earth. This can be tested by introducing the remaining three planets and simulating the system again.

4.2. Energy Conservation Comparison

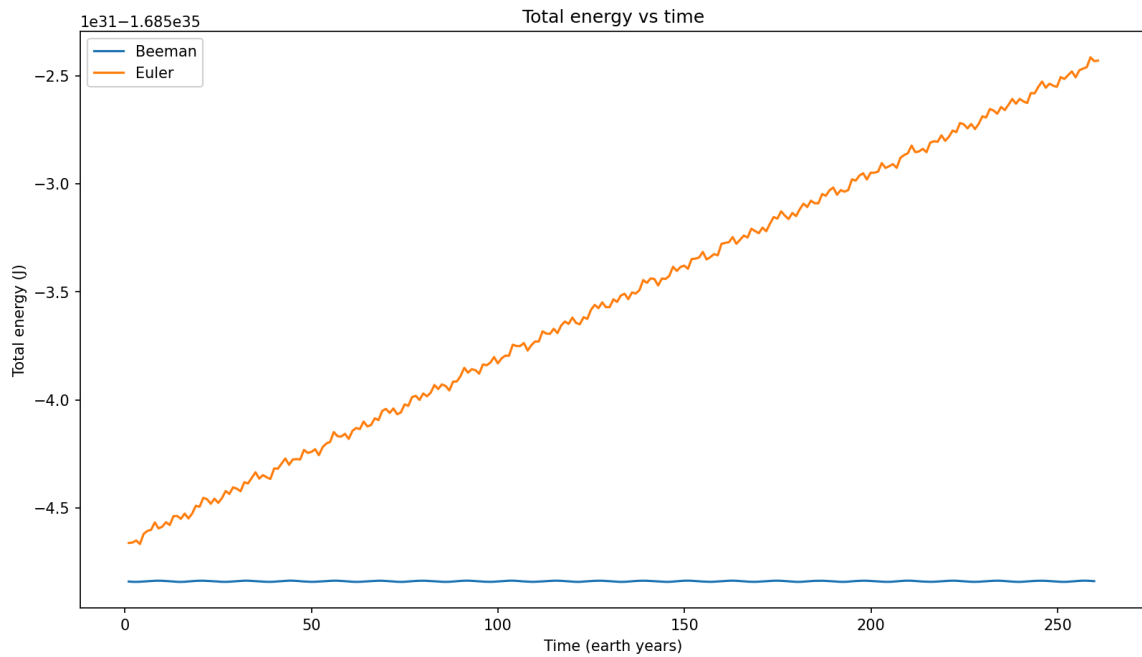
Using option 1 the Beeman Simulation is run for 260 earth years and the following plot of energy with time is created. An extended duration was chosen to capture long term trends, evaluate the stability of the simulation, and detect potential fluctuations in the energy.



Graph 1: Beeman Energy Evolution

The oscillatory pattern showcases periodic variations in the total energy of the system over time, which is to be expected due to the varying positions of the planets as they orbit the Sun. Overall, the energy is conserved to 6 significant figures, $E = 1.68548e35 \text{ J}$, varying very minimally over time even though the changes look significant on the graph due to the scale.

Option 4 was then used to write energy values for both Direct Euler and Beeman to file and compare energy evolution, again running for 260 earth years for the same reasons stated above:



Graph 2: Beeman vs. Euler Energy Evolution over time

Graph 2 illustrates Beeman integration effectively conserving energy over time, compared to Direct Euler, which exhibits divergence. This highlights the error and accuracy of Direct Euler deteriorating and accumulating over longer simulations with extended iterations.

4.3. Planetary alignment

The simulation is run for 100 Earth years using option 5 and a tolerance of 10° and zero planetary alignments have been detected in that time as seen in figure 3.

```
Time = 115.916 earth years. Total energy = -1.685e+35 J.
Time = 116.916 earth years. Total energy = -1.685e+35 J.
Time = 117.915 earth years. Total energy = -1.685e+35 J.
Empty DataFrame
Columns: [Instances of Planetary Alignment (Earth Years)]
Index: []
```

Figure 3: Instances of Planetary Alignment for low tolerance

A high value was not expected as planetary alignments are considered extremely rare. However, to ensure there were no errors with the code, I ran the simulation again using a tolerance angle of 180° for just 25 years, which detected planetary alignments whenever all planets were above the x-axis. The following results were printed and demonstrated that the code is running as expected without errors, and planetary alignments are just extremely rare celestial events.

```
Time = 23.984 earth years. Total energy = -1.685e+35 J.  
Time = 24.983 earth years. Total energy = -1.685e+35 J.  
  Instances of Planetary Alignment (Earth Years)  
0                                     0.001  
1                                     0.002  
2                                     0.003
```

Figure 4: Instances of Planetary Alignment for high tolerance

5. Discussion

5.1. Orbital Periods

The current method of calculating the orbital period relies on the assumption that the time step is small enough and the simulation is accurate enough that the planet's position when it crosses the positive x-axis is close to its position when it started the orbit. If this is not the case, the calculated orbital period may be slightly off. Moreover, the method assumes a perfectly circular orbit and crossing of the x-axis consistently at the same point in each orbit. In reality, however, orbits are elliptical and the point of crossing would vary from orbit to orbit- leading to discrepancies in calculations.

An alternative would be to store the time and position of the planet at each step, and use that to calculate the average orbital period over several orbits. This would account for variations in the orbit and provide a more accurate estimate of the orbital period. Also allowing standard deviation and error on the mean calculations for further evaluation.

5.2. Energy Conservation Comparison

Fluctuations in Beeman integration may arise from simplifications like neglecting perturbations from other celestial bodies or non-uniform gravitational fields. Additionally, numerical errors inherent in the integration methods, including Beeman integration, could contribute to discrepancies. Other integration techniques like Verlet or Leapfrog may offer improved results.

The total energy is read to file only once per Earth year, resulting in large time steps that lead to a jagged and uneven appearance in the graph. Smaller increments for recording energy could yield a smoother curve, providing a more accurate depiction of the energy evolution.

5.3. Planetary Alignment

Although there were no alignments detected within a reasonable value for the tolerance, the simulation was not run for a very long period of time either. In the scale of the solar system, 100 years is a very short time and running the simulation for a significantly longer time period like thousands of years may provide slightly better results and greater instances of planetary alignment.

Furthermore, even with higher tolerance the detected alignments were only found at the beginning because the planets are initialised to all start in a line on the positive x-axis and diverge after that not aligning again.

6. Conclusion

Overall, the code provides a simulation that was effective at providing a reasonable estimation of the orbital mechanics of our solar system. Calculating orbital periods precisely, showcasing energy conservation and divergence with the Beeman and Direct Euler integration techniques, and conducting successful albeit uncommon checks for planetary alignments of the innermost planets.

Unfortunately, I was not able to solve the issue of only showing the simulation for options 1 and 3 and making the other options independent, this would have greatly improved efficiency and reduced computational intensity. Finally, extending the simulation to 3D would allow us to account for the planet's tilt- improving real world application and reliability but increasing complexity which time constraints did not allow for.

7. References

1. Beeman, J. (1976). Some multistep methods for use in molecular dynamics simulations. *Journal of Computational Physics*, 20(2), 130-139.
2. Dawkins, Paul. "Differential Equations - Euler's Method." Tutorial.math.lamar.edu, 3 Dec. 2018, tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx.
3. Smith, Britton. "Project — University of Edinburgh, Computer Simulation Documentation." Ww2.Ph.ed.ac.uk, 2024, www2.ph.ed.ac.uk/compsim/CourseBook2324/project.html#project-task-experiments. Accessed 6 Apr. 2024.
4. Swope, W. C., Andersen, H. C., Berens, P. H., & Wilson, K. R. (1982). A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1), 637-649.
5. Williams, D. R. (2024, April). Planetary Fact Sheet - Metric. Retrieved from NASA: <https://nssdc.gsfc.nasa.gov/planetary/factsheet/>
6. Wisdom, J., & Holman, M. (1991). Symplectic maps for the n-body problem. *The Astronomical Journal*, 102(4), 1528-1538.