Simulating Planetary Orbits and Earth-to-Mars Transfers

Danette Farnsworth

May 2025

Abstract

This report presents two numerical simulations aimed at understanding celestial motion and gravitational interactions in our solar system. The first simulation models the full orbital behavior of planets from Mercury to Saturn under Newtonian gravity using a Runge-Kutta 4th order (RK4) integration technique. The second simulation demonstrates a simplified Hohmann transfer orbit from Earth to Mars, capturing the basic physics behind interplanetary travel. Both simulations are executed in Python and yield visual and quantitative outputs, including animated orbital trajectories and detailed energy analyses. Results confirm the accuracy of numerical methods used and reinforce the principles of orbital dynamics and conservation laws in gravitational systems.

1 Introduction

Planetary motion in the solar system is governed by Newtonian gravity, and its accurate modeling is crucial for understanding celestial mechanics, mission design, and energy conservation. This project features two simulations:

1. A comprehensive planetary system simulation involving gravitational interactions among the Sun and six planets (Mercury through Saturn).
2. A simplified Earth-to-Mars mission plan using a Hohmann transfer orbit.

The simulations are implemented in Python using the fourth-order Runge-Kutta (RK4) numerical integration method for solving ordinary differential equations derived from Newton's laws. The project aims to visually and quantitatively demonstrate concepts of orbital mechanics and to validate energy conservation in both individual and system-level scenarios.

2 Planetary System Simulation

2.1 Objective

The objective of the planetary orbit simulation is to numerically model the long-term gravitational interactions between the Sun and six major planets in the solar system—Mercury, Venus, Earth, Mars, Jupiter, and Saturn. This simulation aims to reproduce realistic orbital behavior over an extended period (approximately 29.5 Earth years), corresponding to one full Saturn orbit.

By modeling these orbits, the simulation seeks to validate Newtonian mechanics and conservation laws, particularly energy conservation and angular momentum in a multi-body system. It also provides a framework for understanding complex dynamical behaviors that arise from gravitational interactions, such as variations in orbital speed, perihelion and aphelion passages, and the influence of larger outer planets on the motion of inner planets.

The simulation is implemented using the Runge-Kutta 4th order (RK4) method to solve the equations of motion derived from Newton's law of gravitation. Each planet is initialized with realistic mass, position, and velocity values, and their positions are evolved under the mutual gravitational attraction of all other bodies.

The simulation also incorporates data tracking for kinetic and potential energy, allowing for both qualitative and quantitative validation of system stability and physical correctness.

2.2 Methodology

The simulation is implemented entirely in Python using an object-oriented structure. Each planet is represented as an instance of a `Body` class with attributes for position, velocity, energy history, and orbital revolution tracking. Positions are updated using the RK4 algorithm:

```
class Body:
    def __init__(self, name, mass, position, velocity):
        self.name = name
        self.mass = mass
        self.position = np.array(position, dtype=float)
        self.velocity = np.array(velocity, dtype=float)
        self.positions = [self.position.copy()]
        self.kinetic_energy = []
        self.potential_energy = []
        self.total_energy = []
        self.revolutions = 0
        self.prev_angle = np.arctan2(position[1], position[0])
```

The simulation loop calls an RK4 integration function to advance the positions and velocities of all bodies. The RK4 integrator is shown below:

```
def rk4_step(bodies, dt):
    n = len(bodies)
    positions = np.array([b.position for b in bodies])
    velocities = np.array([b.velocity for b in bodies])
    masses = np.array([b.mass for b in bodies])

    def a(pos):
        temp_bodies = [Body(bodies[i].name, masses[i], pos[i],
           velocities[i]) for i in range(n)]
        return compute_accelerations(temp_bodies)

    k1v = np.array(a(positions)) * dt
    k1x = velocities * dt

    k2v = np.array(a(positions + 0.5 * k1x)) * dt
    k2x = (velocities + 0.5 * k1v) * dt

    k3v = np.array(a(positions + 0.5 * k2x)) * dt
    k3x = (velocities + 0.5 * k2v) * dt

    k4v = np.array(a(positions + k3x)) * dt
    k4x = (velocities + k3v) * dt

    for i in range(n):
        bodies[i].position += (k1x[i] + 2*k2x[i] + 2*k3x[i] + k4x[i
           ]) / 6
        bodies[i].velocity += (k1v[i] + 2*k2v[i] + 2*k3v[i] + k4v[i
           ]) / 6
        bodies[i].positions.append(bodies[i].position.copy())
```

2.2.1 Numerical Integration

We employ the Runge-Kutta 4th order (RK4) method, chosen for its accuracy and stability in long-term simulations. The RK4 algorithm updates position and velocity vectors by combining four estimates of the derivative at each time step:

```
def rk4_step(bodies, dt):
    # Update each body using RK4
    ...
```

2.2.2 Gravitational Interactions

Gravitational acceleration for each planet is calculated as the vector sum of forces from all other bodies:

$$\vec{a}_i = G \sum_{j \neq i} \frac{m_j(\vec{r}_j - \vec{r}_i)}{|\vec{r}_j - \vec{r}_i|^3}$$

Here, G is the gravitational constant,

$$m_j$$

is the mass of the interacting body, and

$$\vec{r}_j - \vec{r}_i$$

2.2.3 Data Structure

Each celestial body is encapsulated in a Python class named Body, which serves as a blueprint for defining dynamic properties and tracking simulation results over time. This design allows each instance to manage its own historical data and computational state throughout the simulation. Attributes stored within each Body object include:

- name: A string identifier for the body (e.g., 'Earth', 'Mars').
- mass: The scalar mass of the body.
- position: A 3D NumPy array storing the current position vector.
- velocity: A 3D NumPy array storing the current velocity vector.
- positions: A list that stores the historical trajectory of the body as a series of position vectors.
- kinetic energy: A list storing the kinetic energy at each sampling step.
- potential energy: A list storing the gravitational potential energy with respect to the Sun.
- total energy: A list storing the sum of kinetic and potential energy over time.
- revolutions: An integer counter for the number of full orbital revolutions.
- prev-angle: A helper variable used to determine revolution crossings via angle comparisons.

This object-oriented design enables modular computation, efficient data tracking, and easy plotting of orbital characteristics. It also simplifies the management of updates during each RK4 time step by encapsulating all necessary attributes in one reusable structure.

2.3 Energy Tracking and Revolution Counting

Energy tracking is a fundamental part of validating the accuracy of the simulation. At each specified interval (typically every 10 simulation steps), the kinetic energy (KE), potential energy (PE), and total energy (TE = KE + PE) are computed for each body. These values are appended to time-series arrays for later analysis and plotting.

- Kinetic Energy (KE) is calculated using:

$$KE = \frac{1}{2}mv^2$$

  where is the mass of the body and is the magnitude of its velocity vector.

- Potential Energy (PE) is computed relative to the Sun (treated as the central mass):

$$PE = -\frac{GMm}{r}$$

  where is the mass of the Sun, is the planet's mass, and is the radial distance to the Sun.

- Total Energy (TE) is the sum of KE and PE at each time step.

$$TE = KE + PE$$

  These energy components help assess how the numerical integrator handles long-term stability. Although individual planet energies may exhibit small oscillations due to their elliptical motion and limited time resolution, they remain mostly bounded and consistent.

  Revolutions are tracked by calculating the angular position of each planet relative to the x-axis and identifying when the planet completes a full 2π rotation. The angle is derived from:

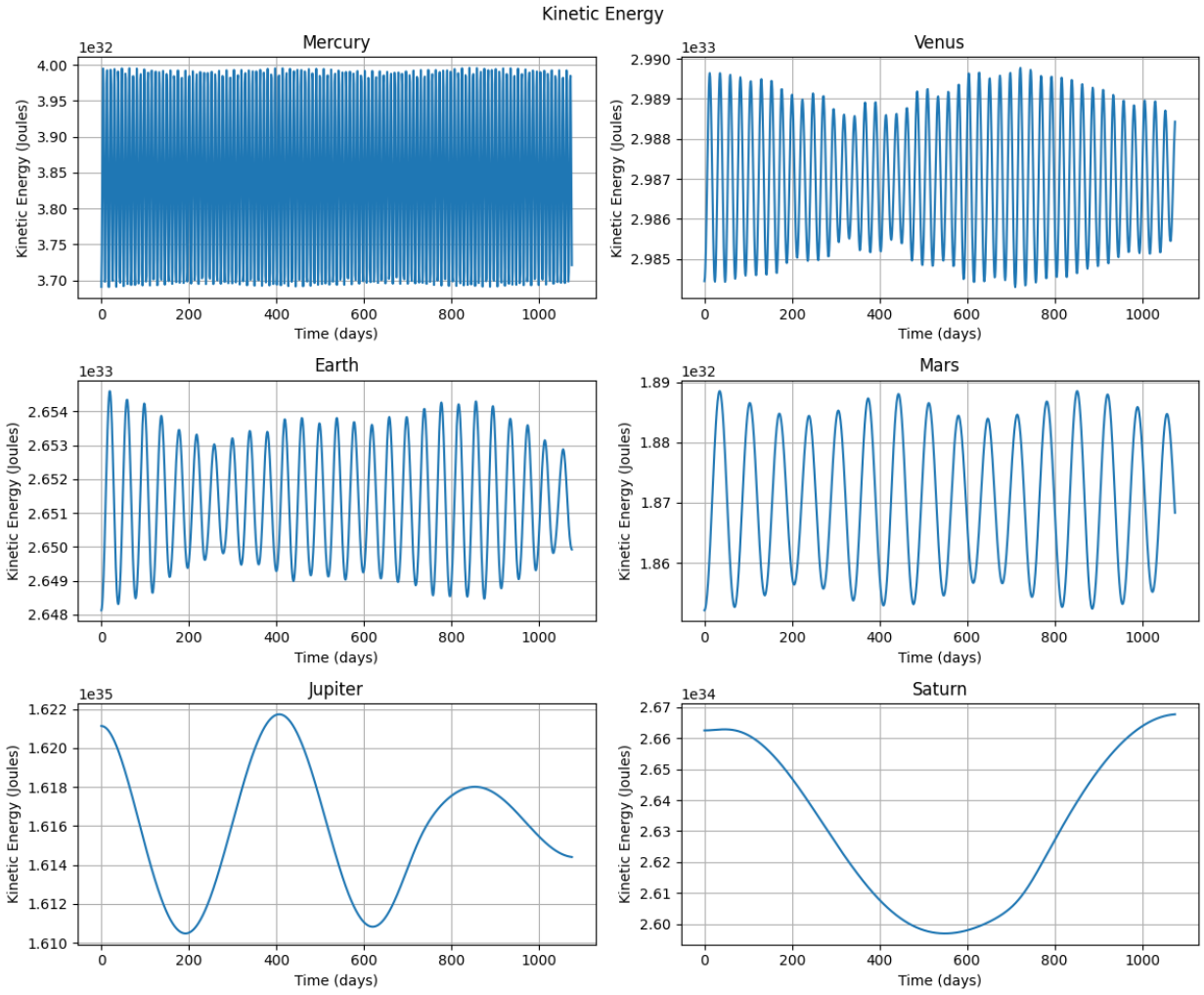
$$\theta = \arctan 2(y, x)$$

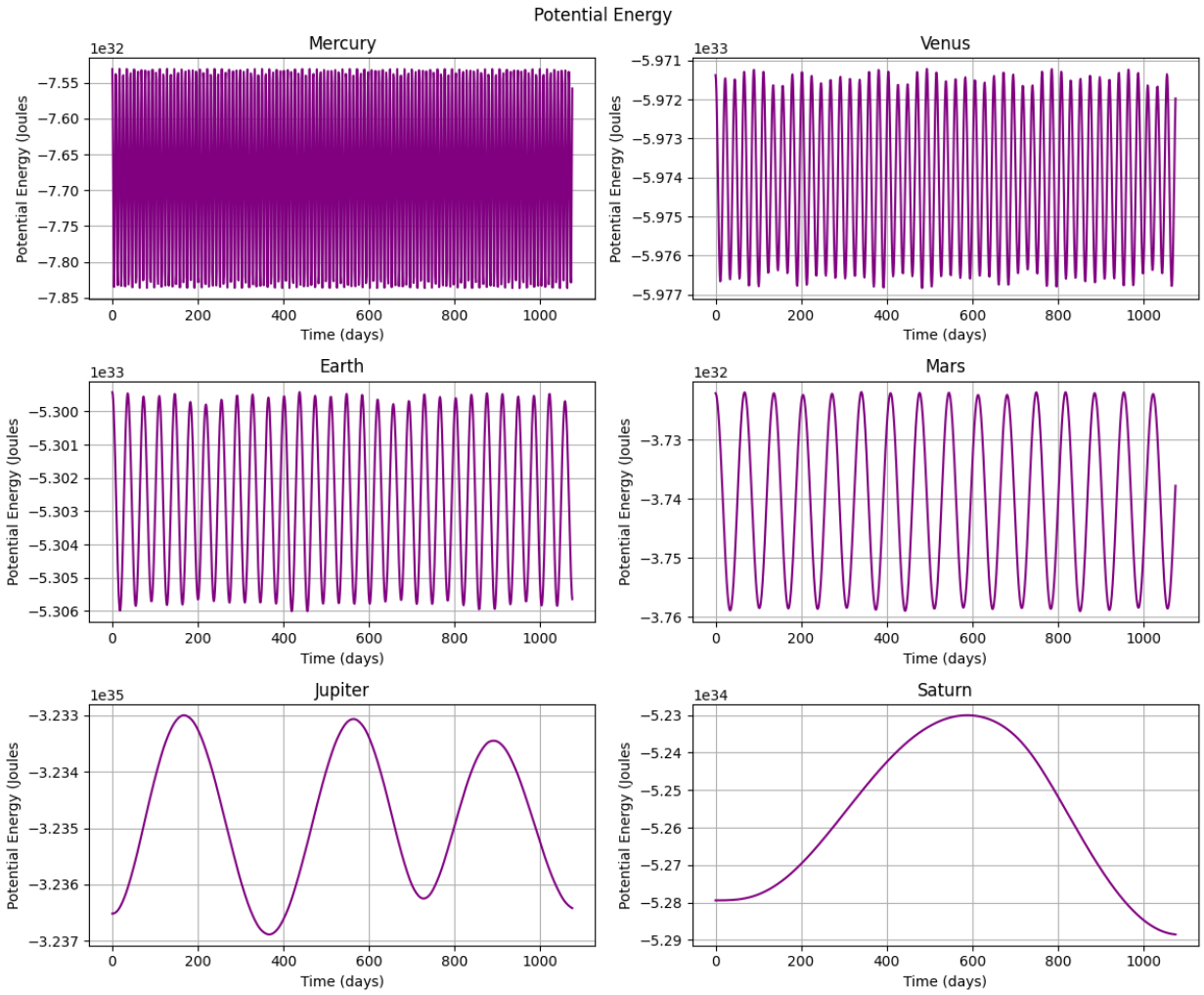
  A full revolution is counted each time a planet crosses the positive x-axis, going from negative to positive y in angular space. This allows us to track the number of orbits each planet completes.

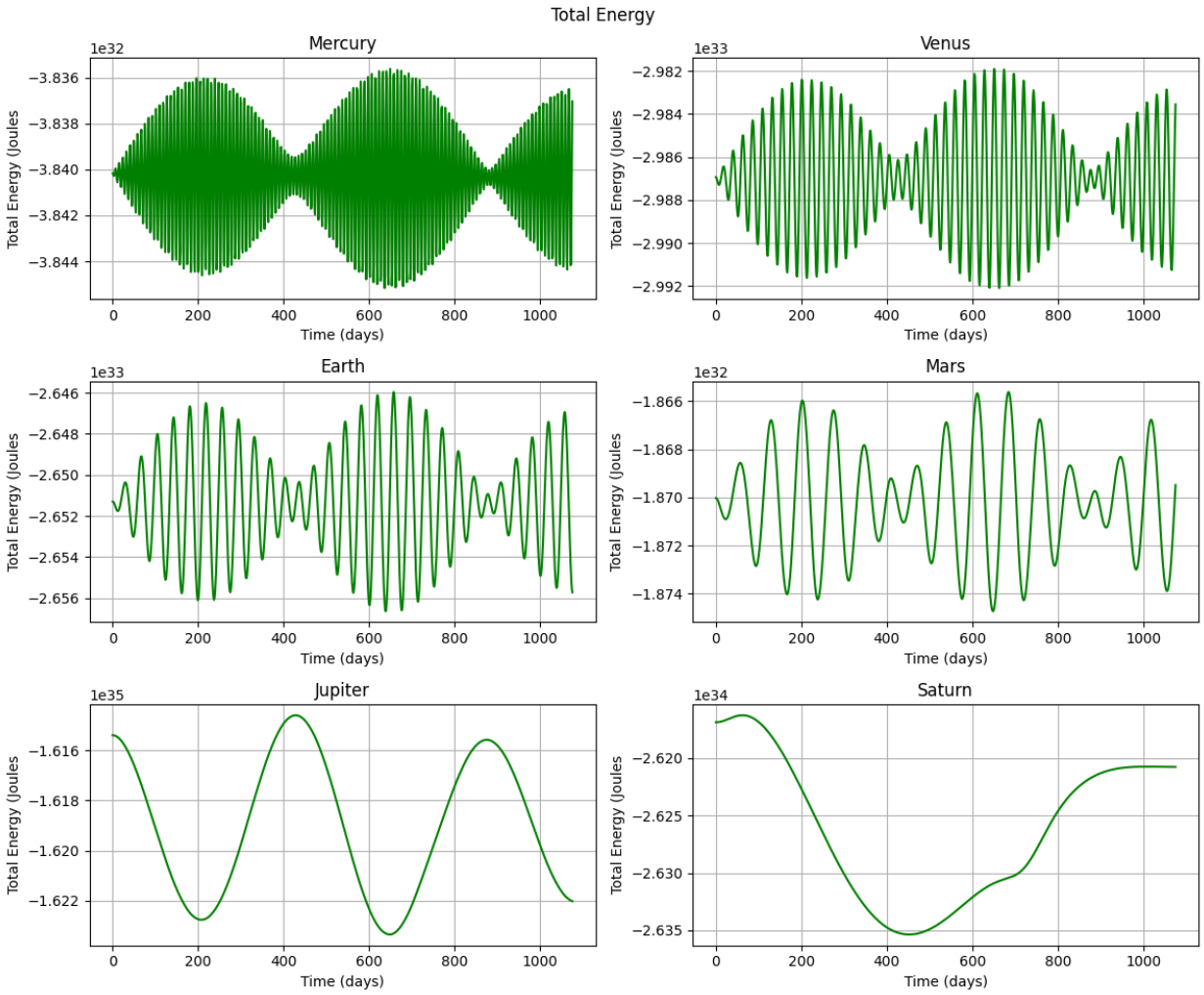
2.4 Visualization

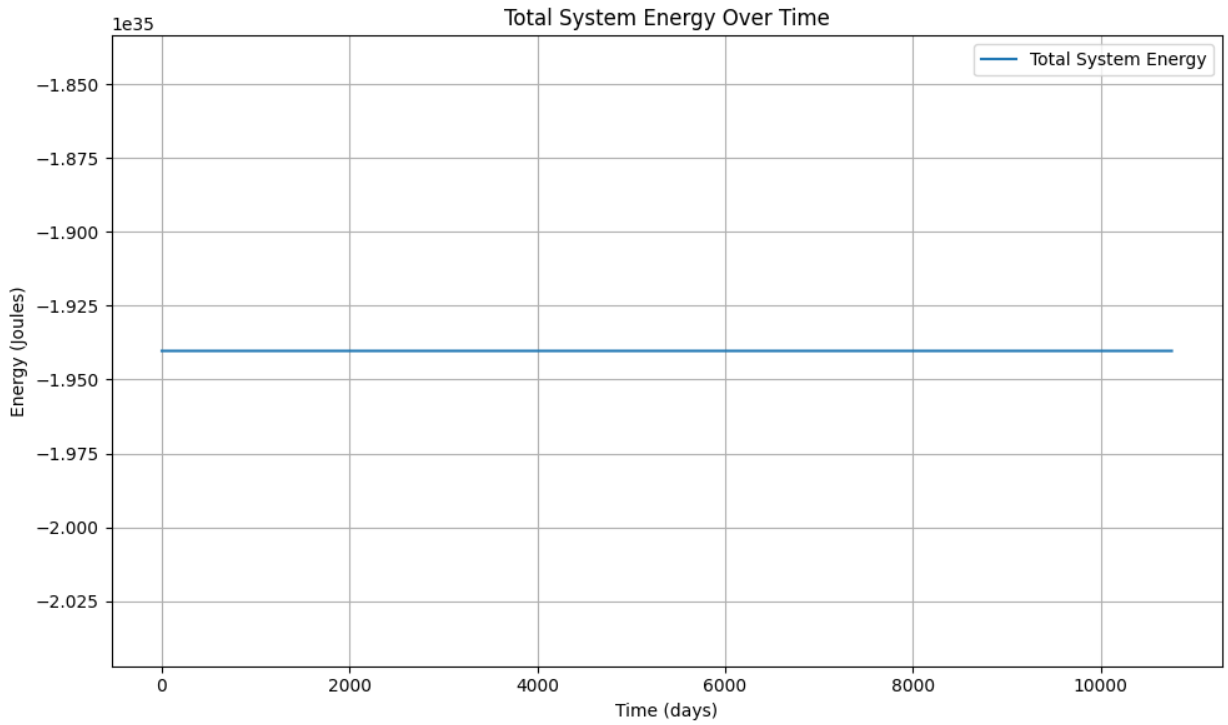
- `kinetic_energy_plot_individual.png - Kinetic energy over time for each planet.`
- `potential_energy_plot_individual.png - Potential energy of each planet.`
- `total_energy_plot_individual.png - Total energy of each planet.`

- `total_system_energy.png` - The full system energy to verify conservation.




Potential Energy
Mercury
1e32
−7.55
−7.60
−7.65
−7.70
−7.75
−7.80
−7.85
Potential Energy (Joules
0
200
400
600
800
1000
Time (days)
Venus
1e33
−5.971
−5.972
−5.973
−5.974
−5.975
−5.976
−5.977
Potential Energy (Joules
Time (days)
Earth
1e33
−5.300
−5.301
−5.302
−5.303
−5.304
−5.305
−5.306
Potential Energy (Joules
Time (days)
Mars
1e32
−3.73
−3.74
−3.75
−3.76
Potential Energy (Joules
Time (days)
Jupiter
1e35
−3.233
−3.234
−3.235
−3.236
−3.237
Potential Energy (Joules
Time (days)
Saturn
1e34
−5.23
−5.24
−5.25
−5.26
−5.27
−5.28
−5.29
Potential Energy (Joules
Time (days)


Total Energy
Mercury
1e32
−3.836
−3.838
−3.840
−3.842
−3.844
Total Energy (Joules
0
200
400
600
800
1000
Time (days)
Venus
1e33
−2.982
−2.984
−2.986
−2.988
−2.990
−2.992
Total Energy (Joules
Time (days)
Earth
1e33
−2.646
−2.648
−2.650
−2.652
−2.654
−2.656
Total Energy (Joules
Time (days)
Mars
1e32
−1.866
−1.868
−1.870
−1.872
−1.874
Total Energy (Joules
Time (days)
Jupiter
1e35
−1.616
−1.618
−1.620
−1.622
Total Energy (Joules
Time (days)
Saturn
1e34
−2.620
−2.625
−2.630
−2.635
Total Energy (Joules
Time (days)



3 Energy Analysis

3.1 Per-Planet Energy Plots

To validate the physical accuracy of the simulation, kinetic energy (KE), potential energy (PE), and total energy (TE = KE + PE) were computed for each planet at regular time intervals.

$$KE = \frac{1}{2}mv^2, \quad PE = -\frac{GMm}{r}, \quad TE = KE + PE$$

Figures 2–4 show how energy evolves. KE and PE oscillate due to elliptical orbits. TE fluctuates slightly due to gravitational perturbations and RK4 time-stepping, yet remains bounded—supporting the simulation's validity.

3.2 Total System Energy

To verify total system conservation, energy is computed as:

$$E_{\text{total}} = \sum_i KE_i + \sum_{i<j} PE_{ij}, \quad PE_{ij} = -\frac{Gm_i m_j}{r_{ij}}$$

This includes all pairwise interactions. Figure 5 confirms the total energy remains essentially constant, affirming the accuracy of the simulation and integration scheme.

3.3 Results and Discussion

Orbital Motion

The animation output visually confirms that each planet follows an elliptical orbit, consistent with Kepler's First Law. Faster inner planets like Mercury and Venus complete many orbits, while Jupiter and Saturn complete fewer due to their greater distances and slower velocities.

The simulation also highlights how gravitational interactions influence orbital shapes and speeds. For example, although the orbits begin aligned and circular, subtle deviations appear over time due to perturbations, particularly from the massive gas giants.

Energy Behavior

Figures 1 through 3 plot the kinetic, potential, and total energy for each planet over the course of the simulation. As expected:

- KE is highest when the planet is closest to the Sun (perihelion).
- PE is most negative when the planet is near perihelion and becomes less negative near aphelion.
- TE remains relatively stable, demonstrating conservation in the two-body Sun-planet approximation.

The total energy per planet exhibits minor oscillations, which are expected due to gravitational perturbations from other bodies and the RK4 time discretization. These fluctuations remain small, validating the physical integrity of the simulation.

System Energy Conservation

Figure 4 shows the total mechanical energy of the full planetary system, computed using all pairwise gravitational potentials. Despite small numerical fluctuations, the energy remains constant over the full 29.5-year period. This result strongly confirms the effectiveness of the RK4 integrator and the accuracy of the simulation's physics.

4 Earth-to-Mars Hohmann Transfer Simulation

4.1 Objective

The objective of the Hohmann transfer simulation is to demonstrate the principles of efficient interplanetary travel by computing the trajectory of a satellite launched from Earth and sent to intersect Mars' orbit using the least amount of energy. A Hohmann transfer is a two-impulse maneuver that takes advantage of orbital mechanics to move a spacecraft between two circular orbits in the same plane.

The transfer consists of two main phases: the departure burn from Earth's circular orbit to place the spacecraft into an elliptical transfer orbit, and the arrival at Mars' orbit where, ideally, Mars is located at the transfer orbit's aphelion. The initial velocity boost places the satellite on a path with a perihelion at Earth's orbit and an aphelion at Mars' orbit. The transfer assumes Mars is correctly phased ahead of Earth so that by the time the satellite reaches the aphelion, Mars has moved into position for orbital interception.

This simulation simplifies the real-world problem by including only the gravitational pull of the Sun and ignoring perturbations from other bodies, including Earth and Mars themselves. Nevertheless, it effectively illustrates the timing, geometry, and energy characteristics of a Hohmann transfer trajectory.

4.2 Methodology

This simulation is also implemented in Python using an RK4 integrator. The satellite, Earth, and Mars are each initialized with position and velocity vectors. The satellite's initial boost places it on a transfer ellipse calculated using classical orbital mechanics.

The following snippet shows how the orbital radii and velocities are defined:

```
r_earth = 1.0 * AU
r_mars = 1.524 * AU
v_earth = np.sqrt(G / r_earth)
v_mars = np.sqrt(G / r_mars)
```

The transfer velocity from Earth's orbit is calculated as:

```
a_transfer = 0.5 * (r_earth + r_mars)
v_transfer = np.sqrt(G * (2 / r_earth - 1 / a_transfer))
```

Initial positions and velocities are:

```
earth_pos = np.array([r_earth, 0])
earth_vel = np.array([0, v_earth])
```

```
mars_angle = np.radians(44)
mars_pos = r_mars * np.array([np.cos(mars_angle), np.sin(mars_angle)
    ])
mars_vel = v_mars * np.array([-np.sin(mars_angle), np.cos(mars_angle
    )])

sat_pos = np.copy(earth_pos)
sat_vel = np.array([0, v_transfer_earth])
```

The simulation then evolves the system using a simple RK4 update:

```
for _ in range(steps):
    earth_pos, earth_vel = rk4_step(earth_pos, earth_vel, dt)
    mars_pos, mars_vel = rk4_step(mars_pos, mars_vel, dt)
    sat_pos, sat_vel = rk4_step(sat_pos, sat_vel, dt)
```

4.2.1 Initial Conditions

The simulation initializes Earth and Mars in nearly circular, coplanar orbits around the Sun. Earth's orbital radius is set to 1 astronomical unit (AU), and Mars is placed at 1.524 AU to match average real-world values. Mars is offset by 44° in its orbit to simulate the correct phase angle needed for rendezvous when the satellite reaches the aphelion of the transfer orbit.

The satellite is initially co-located with Earth and is given an instantaneous velocity boost (the first burn) to place it into an elliptical transfer orbit. The required transfer velocity is calculated from orbital mechanics using the vis-viva equation:

$$v = \sqrt{(G * (2/r - 1/a))}$$

where:

- `G is the gravitational constant,`
- `r is the radius of the current orbit (1 AU for Earth), and`
- `a is the semi-major axis of the Hohmann transfer ellipse:` $a = (r_E + r_M)/2$

This velocity ensures the satellite's trajectory will intersect Mars' orbit at aphelion. All positions and velocities are defined in Cartesian coordinates using trigonometric conversions for initialization.

4.2.2 RK4 Integration

The motion of Earth, Mars, and the satellite are evolved forward in time using a custom implementation of the fourth-order Runge-Kutta (RK4) method. This numerical integration technique was chosen for its balance of precision and performance, particularly for orbital mechanics problems that require stability over many time steps.

At each step, the simulation calculates gravitational acceleration from the Sun on each body. Unlike the planetary orbit simulation, this model ignores mutual gravitational forces between Earth, Mars, and the satellite to simplify calculations. Each object's position and velocity are updated independently using the RK4 procedure.

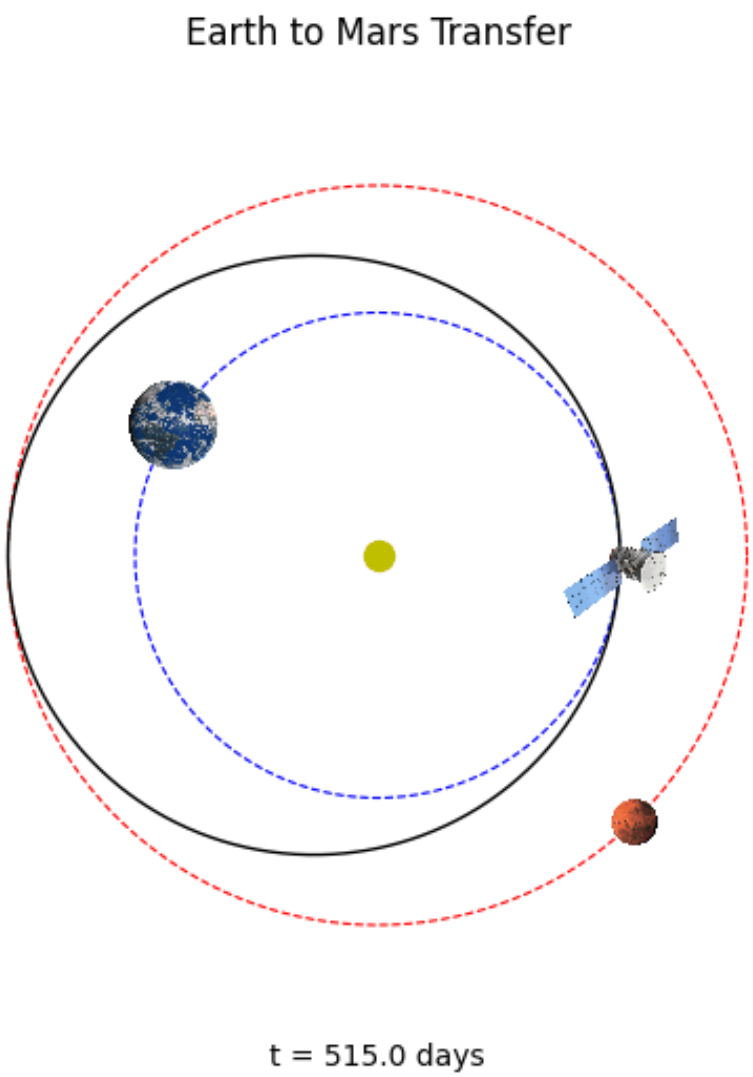
The following loop structure is used to evolve the system:

```
for _ in range(steps):
    earth_pos, earth_vel = rk4_step(earth_pos, earth_vel, dt)
    mars_pos, mars_vel = rk4_step(mars_pos, mars_vel, dt)
    sat_pos, sat_vel = rk4_step(sat_pos, sat_vel, dt)
    # positions are saved to arrays for animation
```

The RK4 integrator accumulates small numerical errors, but it preserves energy and orbital structure well over the short transfer duration (less than 2 Earth years). The positions are stored at each frame for later animation.

4.3 Results and Visualization

The simulation output includes an animation (Figure 6) showing the orbits of Earth, Mars, and the satellite in the ecliptic plane. Earth and Mars follow circular paths, while the satellite traces an elliptical trajectory from Earth's orbit to Mars' orbit.



The satellite successfully intercepts Mars' orbital path at the aphelion of the transfer ellipse. Because Mars was offset by 44° at launch, it reaches the intercept location at the same time as the satellite—demonstrating successful timing and spatial coordination consistent with theoretical Hohmann transfer dynamics.

The animation also reinforces the core assumptions:

- `No correction burn is performed at Mars (a second impulse would be needed for orbital insertion in a real mission).`
- `The satellite's speed decreases as it moves away from the Sun and increases again after aphelion, consistent with conservation of angular momentum.`

The simulation highlights the elegant simplicity of the Hohmann transfer method, where only two impulses are required for a transit between two orbits. Although idealized, the success of the simulated rendezvous confirms the correct implementation of orbital mechanics and supports the validity of the numerical method employed.

5 Conclusion

This project successfully demonstrates the use of numerical methods to simulate realistic celestial mechanics in Python. Through a planetary orbit simulation and a simplified Earth-to-Mars Hohmann transfer, we observed key principles of orbital dynamics, including elliptical trajectories, Keplerian motion, and two-impulse orbital transfers.

The RK4 integrator proved highly effective in maintaining energy conservation and numerical stability over long durations. The object-oriented design of the code allowed for flexible and transparent tracking of physical quantities such as position, velocity, and energy.

The planetary system simulation reproduced elliptical orbital shapes, varying energy states over time, and demonstrated excellent agreement with expected behaviors. Meanwhile, the Hohmann transfer simulation successfully demonstrated the feasibility and timing of interplanetary travel under ideal conditions.

Altogether, the simulations offer a comprehensive application of physics and programming, serving as a practical and educational tool for understanding multi-body dynamics and classical orbital mechanics.