ASTM22

Project 1: The N-body Solar System problem

Author
Nicolai Dimkovski Gottschalk

Project supervisor
David Hobbs

LUNDS UNIVERSITET

Date
February 3, 2025

Contents

1 Abstract

The aim of this project is to simulate the solar system with the addition of two Trojans according to the famous N-body problem. The utilized solutions follows the numerical Runge-Kutta. In conclusion the simulation provides a alike Keplerian rotational profile, drift energy suggesting conservation of energy and almost circular orbits in two dimensional space.

2 Introduction

The Solar System is one of approximately a billion planetary systems contained within the Milky Way. This project attempts to simulate its planetary constituents, namely Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune, along with the addition of two Trojan asteroids. The aim of the simulation is to examine the resulting orbits, the drift energy and rotational profile utilizing the N-body problem. This simulation utilizes N-body solutions following the numerical Runge-Kutta method. Hence, only the mutual interaction of the planets and Trojans will be described by Newtonian physics. To simplify the simulation, the Sun will be set static at origin (0,0,0) with no forces acting upon it, as these forces are negligible. Python was used as programming language and initially the following data was provided for the solar system

	Sun	Jupiter	Trojan 1	Trojan 2
Mass	1	0.001	0	0
x	0	0	-4.503	4.503
y	0	5.2	2.6	2.6
z	0	0	0	0
V_x	0	-2.75674	-1.38	-1.38
V_y	0	0	-2.39	2.39
V_z	0	0	0	0

Table 1: Initial parameters

3 Theory

3.1 The N-body problem and the Runge-Kutta method

In astrophysics, the famous N-body problem follows from Newton's law of gravity as

$$\frac{d^2\vec{x}_i}{dt^2} = -\sum_{j=1;j\neq i} \frac{Gm_j(\vec{x}_i - \vec{x}_j)}{|\vec{x}_i - \vec{x}_j|^3} \tag{1}$$

and it consists of N point masses with respective masses m_i, note G is the gravitational constant. Hence the N-body problem computes the mutual gravitational forces on each body and thereby determining the respective velocity and position vectors. Note that $\vec{x}_{i/j}$ indicates the position vector and can thus be generalized to two- and three-dimensional systems. Solutions to the N-body problem can be derived from numerical integration methods such as the Euler method. The Euler method involves a variable substitution in a general second-order differential equation, which resembles the N-body problem in its mathematical structure.

$$A(x,t)\frac{d^2x}{dt^2} + B(x,t)\frac{dx}{dt} + C(x,t) = 0 \tag{2}$$

Here following the definition of velocity one can substitute $\frac{dx}{dt}$ as $v(t)$ which emphases the transformation to a pair of first order differential equations. Consequently, equation 2 becomes

$$\frac{dv}{dt} = -\frac{B(x,t)}{A(x,t)}v(t) - \frac{C(x,t)}{A(x,t)} \tag{3}$$

and similarly $\frac{d\vec{x}_i}{dt}$ in equation 1 can be substituted by v_i representing the velocity. Furthermore, utilizing a six-dimensional phase space coordinate system $[\vec{x}_i, \vec{v}_i]$, where both $\vec{x}_i$ and $\vec{v}_i$ have three components, the N-body system attains a $6N$ vector description

$$\vec{W} \equiv [\vec{w}_1, ..., \vec{w}_N] \tag{4}$$

which reformulates equation 1, following the velocity substitution, as

$$\frac{dW_l}{dt} = g_l(\vec{W}). \tag{5}$$

Here $g_l(\vec{W})$ is given by the right hand side of equation 1. Furthermore, the Euler method provides the solution in the form

$$W_l^{n+1} = W_l^n + hg_l(W_1^n, ..., W_l^n, ..., W_{6N}^n) = W_l^n + hg_l(\vec{W}^n). \tag{6}$$

which specifies a finite difference of the respective differential equation over an interval (time step)

$$h \equiv \Delta t \equiv t^{n+1} - t^n.$$

The Euler method considers only the two first terms in the Taylor expansion of the exact solution thus yielding cumulative errors. To resolve these cumulative errors one can consider the arbitrariness in the position $(t, \vec{W})$ of g allowing for a refined definition of h. Thereby, one can construct weighted sums with k estimates of g yielding the cancellation of error terms in the Taylor expansion up to $k+1$. This improvement over the Euler method is known as the Runge-kutta method and utilizing the Euler method to determine $\vec{W}(t^n + h/2)$ at the midpoint. Hence, the following Runge-Kutta scheme is found

$$\vec{W}\left(t^n + \frac{h}{2}\right) = \vec{W}_b = \vec{W}(t^n) + \frac{h}{2}\vec{g}(t^n, \vec{W}^n). \tag{7}$$

Following four iterations of equation 7, with four respective weights, the fourth order Runge-Kutta method is found to be

$$\vec{W}^{n+1} = W^n + \frac{1}{6}h\vec{f_a} + \frac{1}{3}h\vec{f_b} + \frac{1}{3}h\vec{f_c} + \frac{1}{6}h\vec{f_d} \tag{8}$$

where f_a is the initial iteration and the remaining f are the succeeding iterations following the implementation of the previous computed $\vec{W}$ in equation 7, see example below. [1]

$$\vec{f_a} = \vec{g}(t^n, \vec{W}^n)$$

$$\implies \vec{W_b} = \vec{W}^n + \frac{h}{2}\vec{f_a}$$

$$\implies \vec{f_b} = \vec{g}\left(t^n + \frac{h}{2}, \vec{W_b}\right)$$

3.2 Drift Energy and Rotational Profile

The drift energy is described by the total energy at each time step in a simulation. For the N-body problem the total energy is described by

$$E_{\text{tot}} = \underbrace{\frac{1}{2}m_{\text{tot}}v_{\text{com}}^2}_{(I)} - \underbrace{\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N}\frac{Gm_im_j}{r_{ij}}}_{(II)} + \underbrace{\sum_{i=1}^{N}\frac{1}{2}m_iv_i^2}_{(III)}.$$

here (I) is the kinetic energy of the center of mass of the system, (II) is the gravitational potential energy of the point masses, (III) is the kinetic energy of the point masses. Note that for the Solar system simulation a simplification is made which is assuming $v_{com} = 0$. This implies that the solar system as a

[1] Bodenheimer, P., Laughlin, G. P., Różyczka, M., & Yorke, H. W. (2007). N-Body Particle Methods. I A. Bodenheimer, P., Laughlin, G. P., Różyczka, M., & Yorke, H. W.(Ed.), *Numerical Methods in Astrophysics* (pg. 73– 114). Taylor & Francis.

whole is not translating in space.[2] Consequently, the total energy of the N-body problem, describing the solar system, in the simplified simulation is

$$E_{tot} = -\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N}\frac{Gm_i m_j}{r_{ij}} + \sum_{i=1}^{N}\frac{1}{2}m_i v_i^2. \tag{9}$$

Furthermore, the rotational profile of the orbiting bodies in the solar system is expected to follow a Keplerian profile. The argument follows from macroscopic observations and more formally the rotational velocity can be derived from equating the centripetal force to the Newton's law of gravity, see derivation below.[3]

$$\frac{m_1 v_r^2}{r} = G\frac{m_1 m_2}{r^2}$$

$$\Longrightarrow$$

$$v_r = \sqrt{G\frac{m_2}{r}} \tag{10}$$

4 Method

4.1 Initialization

The following part of the code is depicted in Figure 8 and shows the initialization of parameters which will later be used. Moreover, the velocity and position parameters was extracted utilizing a package called *jplephem.spk* which enables collecting data from bsp files found in NASA Planetary Data System Navigation Node (`https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/`). Note the bsp file needs to be downloaded in the scripts file directory before running the code, and the specific bsp file used in this code is the *de430.bsp*. The planetary masses was normalized in respect to the solar mass and was found on NASA Planetary Fact Sheet - Metric (`https://nssdc.gsfc.nasa.gov/planetary/factsheet/`). This enabled the use of the gravitational constant as $G = 4\pi^2$. Furthermore, the Sun's position has been set static to the origin (0,0,0) thus an additional correction has been added to the phase space elements of the respective orbiting objects to account for this transformation.

[2] Bodenheimer, P., Laughlin, G. P., Różyczka, M., & Yorke, H. W. (2007). N-Body Particle Methods. I A. Bodenheimer, P., Laughlin, G. P., Różyczka, M., & Yorke, H. W.(Ed.), *Numerical Methods in Astrophysics* (pg. 73– 114). Taylor & Francis.

[3] Bensby, T., Andersson E., & Borsato N. (2024). ASTA33 Lab: The Rotation Curve of the Milky Way. Lab manual.

4.2 Acceleration method

This method is shown in Figure 9 and implements equation 1 numerically thereby recalculating the $S - matrix$ which consists of six-dimensional phase space coordinates for the respective orbiting bodies. Note that the choice of excluding nested loops was done due to optimization reasons in Python, which prefers a linear algebraic approach.

4.3 Runge-Kutta method

The Runge-Kutta method is shown Figure 10 and implements equation 8 numerically.

4.4 Drift Energy method

In the following method the total energy, derived from equation 9, is computed at every time step $h = 0.01$ up to the time t_{max}. Moreover, the method produces a plot with the total energies versus time steps. Note that when computing the potential energy matrix, only the upper triangle needs to be considered due to symmetry, excluding the diagonal as it represents self-interaction. This method is illustrated in Figure 11.

4.5 Orbits method

This methods simulates and plots the orbits of the planets and Trojans around the Sun based on their mutual gravitational interaction computed in the acceleration method, utilizing the Runge-Kutta method. The plot utilizes a scale parameter which adjusts the zooming of the plot. Note that Mercury's orbit is filtered in order to exclude diverging trajectory points found for greater elapsed time, as h was set constant here. The method is depicted in Figure 12 and the time step used was $h = 0.01$.

4.5.1 Rotational Velocity method

The following method computes the rotational velocity based on a simplification of the N-body problem where only the x- and y-coordinates is extracted from $S-matrix$. Furthermore, the rotational velocity is then defined as the projection of the two components: $v_r = \sqrt{v_x^2 + v_y^2}$. The argument for allowing this simplification is the visualized circular orbits in the two dimensional xy-plot, constructed by the Orbits method, and thereby the main goal with this method is to get a representation of the system's rotational profile. Note the method is shown in Figure 13.

4.5.2 Animation and Update function

This part of the code is shown in Figure 14 and 15. Its main purpose is to visualize the animation of the mutual gravitational interactions between the planets and Trojans utilizing the acceleration and Runge-Kutta method. Note

the upper time limit was set to 300 years with each iteration of the update function corresponding to a time step of $h = 0.01$ initially. The time step h was made adaptive so that it decreases with increasing elapsed time.

5 Results

5.1 Drift Energy

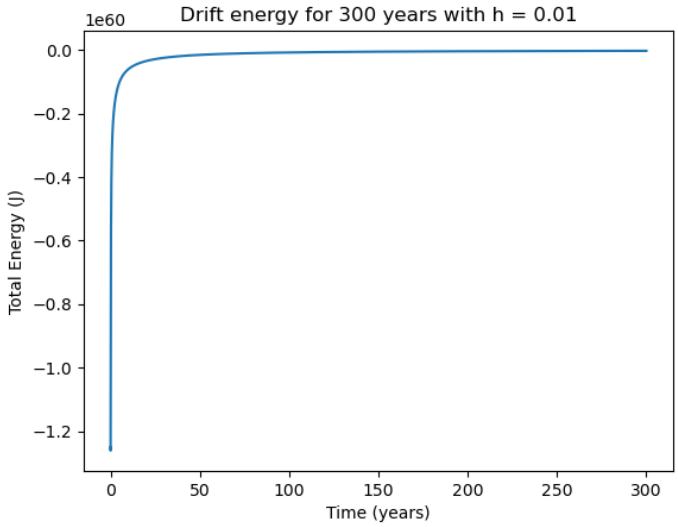


Figure 1: The drift energy in Joules for 300 years of the solar system with two Trojans. Note the computation follows equation 9 and the time step h was set to 0.01.

5.2 Orbits

5.2.1 Solar System orbits

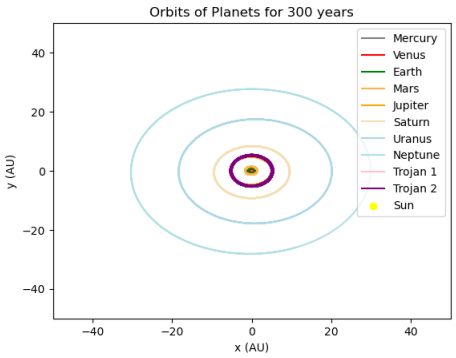


Figure 2: The simulated orbits of the solar system with two Trojans and a orbiting time of 300 years. Here the scale factor was set to 50.

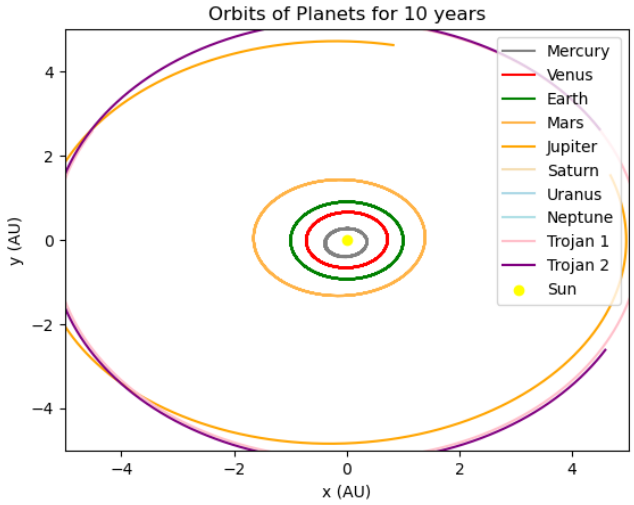


Figure 3: The simulated orbits of the solar system with two Trojans and a orbiting time of 10 years. Here the scale factor was set to 5.

5.2.2 Mercury Orbit

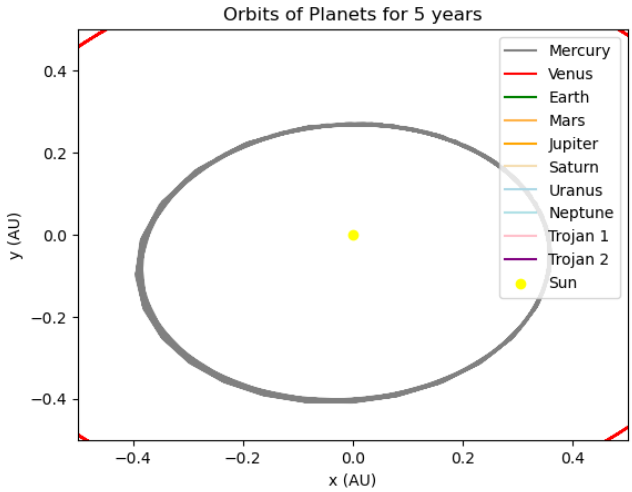


Figure 4: The simulated orbits of Mercury with a orbiting time of 5 years. Here the scale factor was set to 0.5.

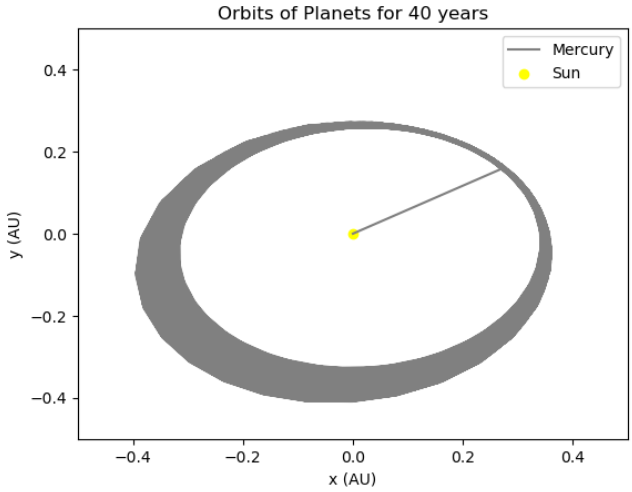


Figure 5: The simulated orbits of Mercury with a orbiting time of 40 years. Here the scale factor was set to 0.5. Note the width of the orbit is due to peculiar changes in the orbiting path.

5.3 Rotational Profile

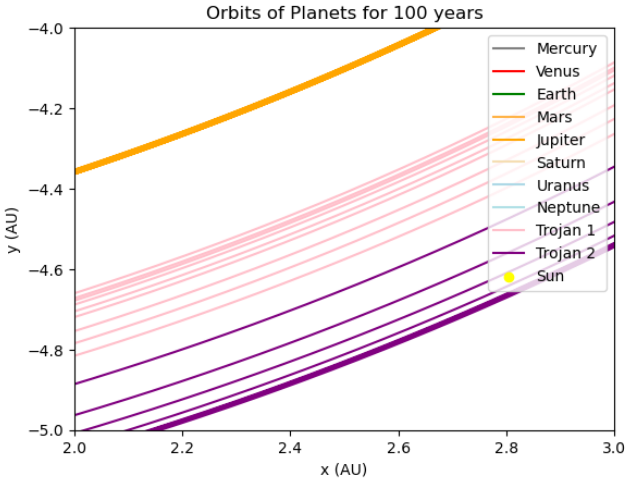


Figure 6: The simulated orbits of the Trojans with a orbiting time of 100 years.

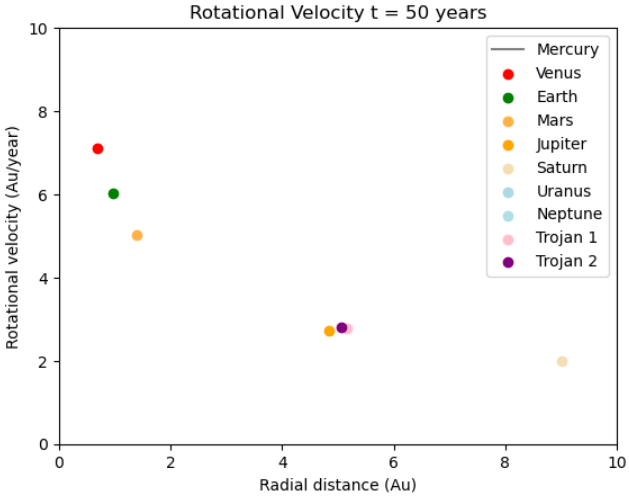


Figure 7: Rotational scatter profile utilizing the 2D simplification for the simulated Solar system over 50 years.

6 Discussion

In the drift energy function the initial spike is seen as a numerical error during the initialization of the simulation. Moreover, at later stages the total energy is seen to remain constant at 0 which suggests conservation of energy is maintained. It is important to remember that this only suggests the energy is conserved locally in the solar system as the velocity of the whole system v_{com} was set to zero. This simplification directly opposes real world observations of the Solar system's orbital motion in the Milky way. Additionally, the orbits shown in Figures 2 and 3 are nearly circular, which is expected to be elliptical in two dimensions when compared to real observations. Note that in the real world, the translation of the solar system as a whole within the Milky Way results in the actual orbital trajectories being helix-shaped, as the planets 'chase' the Sun. Thus, the visualized orbits in the plots represent a significant simplification of real-world observations. In Figure 5, a thickened orbit is observed for Mercury, indicating a change in the orbital path over a longer time span compared to Figure 4. This peculiar change in the orbital path likely arises from numerical errors caused by an insufficiently small time step, significant data load, or perturbations from other planets in the vicinity. The numerical error became particularly noticeable for longer time durations. Around 8 years into the simulation, with a constant time step of h=0.01, Mercury's orbit began to exhibit divergence. To resolve this, adaptive time steps was applied decreasing h with elapsed time. Consequently, for lower time steps h a higher precision from the Runge-Kutta method was obtained and the simulation was more stable without Mercury diverging. Additionally, the two-dimensional simplification of the orbits might contribute to this effect as the z-component of the motion was excluded. It is important to note that, in the real world, Mercury exhibits a peculiar orbital trajectory due to effects described by general relativity. However, this cannot be considered a contributing factor here, as this algorithm only accounts for Newtonian gravitational interactions. In Figure 6, when observing the Trojans' orbits, it can be seen that their respective orbiting trajectories possibly oscillate around a mean orbiting trajectory. Consequently, due to the two-dimensional plotting, information about the z-component is lost, and thus, the actual oscillation might form a helical trajectory orbiting around the Sun. To further investigate this, it is suggested to redo the plotting in three dimensions.

When computing the rotational curve a major simplification was done in discarding the z- component. Consequently, the magnitudes of the respective velocities after 50 years is non-representative of the true velocity. Furthermore, a second simplification was made which was to assume circular orbits following arguments found in Figure 2,3 and 4. Thereby the projection of the v_x and v_y component represents the rotational velocity. These simplifications preserves the structural profile of the rotational curve, meaning the qualitative behavior can still be compared to a Keplerian rotational profile, even though the absolute velocity magnitudes are not exact. Hence, the rotational profile in Figure 6 seems to follow a $R^{-1/2}$ dependency, suggesting a Keplerian profile is followed.

7 References

7.1 Literature

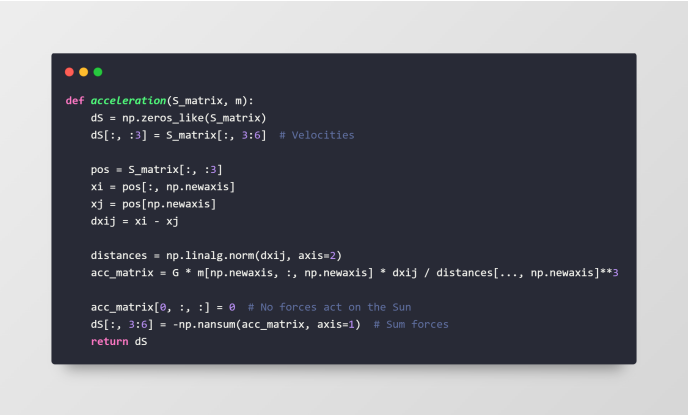
1. Bodenheimer, P., Laughlin, G. P., Różyczka, M., & Yorke, H. W. (2007). N-Body Particle Methods. I A. Bodenheimer, P., Laughlin, G. P., Różyczka, M., & Yorke, H. W.(Ed.), *Numerical Methods in Astrophysics* (pg. 73– 114). Taylor & Francis.

2. Bensby, T., Andersson E., & Borsato N. (2024). ASTA33 Lab: The Rotation Curve of the Milky Way. Lab manual.

8 Appendix A

```
mport matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import tkinter as tk
from jplephem.spk import SPK

#Final version

# Constants
G = 4 * np.pi**2  # Gravitational constant in AU^3 / (yr^2 * Msun)
kmtoAu = 1.0 / 149597870.7  # Conversion factor: km to AU
julian_date = 2460703.5  # Example Julian date
days_per_year = 365.25  # Days in a year

# Masses of celestial bodies (in solar masses)
sunmass = 1.988416 * 10**30  # kg
masses = np.array([
    1,  # Sun
    (3.3 * 10**23) / sunmass,  # Mercury
    (4.87 * 10**24) / sunmass,  # Venus
    (5.97 * 10**24) / sunmass,  # Earth
    (6.42 * 10**23) / sunmass,  # Mars
    (1.898 * 10**27) / sunmass,  # Jupiter
    (5.68 * 10**26) / sunmass,  # Saturn
    (8.68 * 10**25) / sunmass,  # Uranus
    (1.02 * 10**26) / sunmass,  # Neptune
    0,  # Trojan 1
    0   # Trojan 2
])

# Load ephemeris file
eph = SPK.open('de430.bsp')

# Extract the Sun's position and velocity relative to the SSB
sun_position, sun_velocity = eph[0, 10].compute_and_differentiate(julian_date)
sun_position_au = np.array(sun_position) * kmtoAu
sun_velocity_au_year = np.array(sun_velocity) * kmtoAu * days_per_year

# Initialize arrays for positions and velocities
position_rows = []
velocity_rows = []

# Add the Sun at (0,0,0)
position_rows.append([0, 0, 0])
velocity_rows.append([0, 0, 0])

# List of planetary SPK IDs (Mercury to Neptune)
spk_ids = [1, 2, 3, 4, 5, 6, 7, 8]

# Loop through celestial bodies
for i, planet_id in enumerate(spk_ids):
    # Get planet's position and velocity relative to the SSB
    (x, y, z), (vx, vy, vz) = eph[0, planet_id].compute_and_differentiate(julian_date)

    # Convert positions to AU and adjust relative to the Sun
    x = x * kmtoAu - sun_position_au[0]
    y = y * kmtoAu - sun_position_au[1]
    z = z * kmtoAu - sun_position_au[2]

    # Convert velocities to AU/year and adjust relative to the Sun
    vx = vx * kmtoAu * days_per_year - sun_velocity_au_year[0]
    vy = vy * kmtoAu * days_per_year - sun_velocity_au_year[1]
    vz = vz * kmtoAu * days_per_year - sun_velocity_au_year[2]

    # Append to rows
    position_rows.append([x, y, z])
    velocity_rows.append([vx, vy, vz])

# Add Trojan bodies (positions/velocities)
position_rows.append([-4.503, 2.6, 0])  # Trojan 1
velocity_rows.append([-1.38, -2.39, 0])

position_rows.append([4.503, 2.6, 0])  # Trojan 2
velocity_rows.append([-1.38, 2.39, 0])

# Convert to NumPy arrays
positionMatrix = np.array(position_rows)
velocityMatrix = np.array(velocity_rows)

# Combine into the state matrix
S = np.hstack((positionMatrix, velocityMatrix))
initial_state = S.copy()
```

Figure 8: The Initialization part of the code



```python
def acceleration(S_matrix, m):
    dS = np.zeros_like(S_matrix)
    dS[:, :3] = S_matrix[:, 3:6]  # Velocities

    pos = S_matrix[:, :3]
    xi = pos[:, np.newaxis]
    xj = pos[np.newaxis]
    dxij = xi - xj

    distances = np.linalg.norm(dxij, axis=2)
    acc_matrix = G * m[np.newaxis, :, np.newaxis] * dxij / distances[..., np.newaxis]**3

    acc_matrix[0, :, :] = 0  # No forces act on the Sun
    dS[:, 3:6] = -np.nansum(acc_matrix, axis=1)  # Sum forces
    return dS
```

Figure 9: The acceleration method.

```python
def RungeKutta(S_matrix, mass,h):
    k1 = acceleration(S, masses) * h
    k2 = acceleration(S + k1 / 2, masses) * h
    k3 = acceleration(S + k2 / 2, masses) * h
    k4 = acceleration(S + k3, masses) * h
    S_matrix += (k1 + 2 * k2 + 2 * k3 + k4) / 6
    return S_matrix
```

Figure 10: The Runge-Kutta method.

```python
def energy_drift(tmax,h,S,mass):
    totEnergy = []
    mass = mass*1.988416 * 10**30 #convert from Msol to kg
    AU_meters = 1.495978707e11  # 1 AU in meters
    secondstot = 365.25 * 24 * 3600  # Seconds in a year
    AUms = AU_meters / secondstot #Au to m/s

    t =0
    while t < tmax:
        kineticEnergy = np.sum(0.5 * mass * np.sum(((S[:, 3:6])*AUms)**2, axis=1))
        positions = S[:, :3]
        ri = positions[:,np.newaxis]
        rj = positions[np.newaxis]
        drij = ri-rj
        r = np.linalg.norm(drij, axis=2)
        np.fill_diagonal(r,np.inf) #avoid self-interaction
        G = 4*np.pi**(2)
        potential_matrix = -G* mass[:,np.newaxis]*mass[np.newaxis, :]/r[..., np.newaxis]
        potential_energy = np.sum(np.triu(potential_matrix, k =1))
        #k=1 indicates that we exclude i=j and triu is the summation of the upper triangle since by symmetry Uij = Uji
        gamma = kineticEnergy+potential_energy
        totEnergy.append(gamma)
        S = RungeKutta(S,mass,h)
        t+=h

    energy_fig = plt.figure()
    xspace = np.arange(0,tmax,h)
    xspace = np.append(np.arange(0, tmax, h), tmax)
    plt.plot(xspace,totEnergy)
    text = "Drift energy for " + str(tmax) + " years with h = " + str(h)
    plt.xlabel("Time (years)")
    plt.ylabel("Total Energy (J)")

    plt.title(text)
    plt.show(block=False)
```

Figure 11: The drift energy method.

```
def Orbits(tmax,h,S_matrix2,mass, _scale):
    t = 0
    steps = int(tmax / h) + 1
    mercuryPos = np.zeros((steps,2))
    venusPos = np.zeros((steps,2))
    earthPos = np.zeros((steps,2))
    marsPos = np.zeros((steps,2))
    jupiterPos = np.zeros((steps,2))
    saturnPos = np.zeros((steps,2))
    uranusPos = np.zeros((steps,2))
    neptunePos = np.zeros((steps,2))
    troj1Pos = np.zeros((steps,2))
    troj2Pos = np.zeros((steps,2))
    index = 0
    #Extracting positions
    while t < tmax:
        print(t)
        S_matrix2 = RungeKutta(S_matrix2,mass,h)
        xlist2 = S_matrix2[:, 0]
        ylist2 = S_matrix2[:, 1]
        mercuryPos[index, 0], mercuryPos[index, 1] = xlist2[1], ylist2[1]
        venusPos[index, 0], venusPos[index, 1] = xlist2[2], ylist2[2]
        earthPos[index, 0], earthPos[index, 1] = xlist2[3], ylist2[3]
        marsPos[index, 0], marsPos[index, 1] = xlist2[4], ylist2[4]
        jupiterPos[index, 0], jupiterPos[index, 1] = xlist2[5], ylist2[5]
        saturnPos[index, 0], saturnPos[index, 1] = xlist2[6], ylist2[6]
        uranusPos[index, 0], uranusPos[index, 1] = xlist2[7], ylist2[7]
        neptunePos[index, 0], neptunePos[index, 1] = xlist2[8], ylist2[8]
        troj1Pos[index, 0], troj1Pos[index, 1] = xlist2[9], ylist2[9]
        troj2Pos[index, 0], troj2Pos[index, 1] = xlist2[10], ylist2[10]
        t+= h
        index+=1

    # Plot orbits
    orbit_fig = plt.figure()

    #Filter out Mercury's positions based on x and y thresholds** DONT CHANGE BELOW 0.5, instability in visualisation
    x_threshold = 0.5
    y_threshold = 0.5

    mercury_filtered = mercuryPos[
        (np.abs(mercuryPos[:, 0]) < x_threshold) &
        (np.abs(mercuryPos[:, 1]) < y_threshold)
    ]



    plt.plot(mercury_filtered[:, 0], mercury_filtered[:, 1], label="Mercury", color="Gray")
    #plt.plot(mercuryPos[:, 0], mercuryPos[:, 1], label="Mercury", color = "Gray")
    plt.plot(venusPos[:, 0], venusPos[:, 1], label="Venus", color = "Red")
    plt.plot(earthPos[:, 0], earthPos[:, 1], label="Earth", color = "Green")
    plt.plot(marsPos[:, 0], marsPos[:, 1], label="Mars", color = "#ffb347")
    plt.plot(jupiterPos[:, 0], jupiterPos[:, 1], label="Jupiter", color = "Orange" )
    plt.plot(saturnPos[:, 0], saturnPos[:, 1], label="Saturn", color ="Wheat" )
    plt.plot(uranusPos[:, 0], uranusPos[:, 1], label="Uranus", color = "LightBlue")
    plt.plot(neptunePos[:, 0], neptunePos[:, 1], label="Neptune", color = "PowderBlue")
    plt.plot(troj1Pos[:, 0], troj1Pos[:, 1], label="Trojan 1", color = "Pink")
    plt.plot(troj2Pos[:, 0], troj2Pos[:, 1], label="Trojan 2", color= "Purple")

    # Plot the Sun at the origin
    plt.scatter(0, 0, color="yellow", label="Sun")

    # Customize the plot
    plt.xlabel("x (AU)")
    plt.ylabel("y (AU)")
    plt.title(f"Orbits of Planets for {tmax} years")
    plt.legend(loc="upper right")
    plt.xlim(-_scale,_scale)
    plt.ylim(-_scale,_scale)
    plt.show()
```

Figure 12: The Orbits method.

```
def Rotational(tmax,h,S_matrix3,mass):
    #velocities are in au/year
    #distance in au
    mass = mass*1.988416 * 10**30 #convert from Msol to kg
    planetsmass = mass[1:] #excluding sun mass
    G = 4*np.pi**(2)
    t =0
    while t < tmax:
        S_matrix3 = RungeKutta(S_matrix3,mass,h)
        t+=h

    vxlist2 = S_matrix3[:, 3]
    vylist2 = S_matrix3[:, 4]

    xlist3 = S_matrix3[:, 0]
    ylist3 = S_matrix3[:, 1]
    #Radial velocity in x and y plane, neglecting z component
    mercuryVel = np.sqrt((vxlist2[1]**(2))+(vylist2[1]**(2)))
    venusVel = np.sqrt((vxlist2[2]**(2))+(vylist2[2]**(2)))
    earthVel = np.sqrt((vxlist2[3]**(2))+(vylist2[3]**(2)))
    marsVel = np.sqrt((vxlist2[4]**(2))+(vylist2[4]**(2)))
    jupiterVel = np.sqrt((vxlist2[5]**(2))+(vylist2[5]**(2)))
    saturnVel = np.sqrt((vxlist2[6]**(2))+(vylist2[6]**(2)))
    urnausVel = np.sqrt((vxlist2[7]**(2))+(vylist2[7]**(2)))
    neptuneVel = np.sqrt((vxlist2[8]**(2))+(vylist2[8]**(2)))
    trojan1Vel = np.sqrt((vxlist2[9]**(2))+(vylist2[9]**(2)))
    trojan2Vel = np.sqrt((vxlist2[10]**(2))+(vylist2[10]**(2)))

    #Radial distance
    mercuryR = np.sqrt((xlist3[1]**(2))+(ylist3[1]**(2)))
    venusR = np.sqrt((xlist3[2]**(2))+(ylist3[2]**(2)))
    earthR = np.sqrt((xlist3[3]**(2))+(ylist3[3]**(2)))
    marsR = np.sqrt((xlist3[4]**(2))+(ylist3[4]**(2)))
    jupiterR = np.sqrt((xlist3[5]**(2))+(ylist3[5]**(2)))
    saturnR = np.sqrt((xlist3[6]**(2))+(ylist3[6]**(2)))
    urnausR = np.sqrt((xlist3[7]**(2))+(ylist3[7]**(2)))
    neptuneR = np.sqrt((xlist3[8]**(2))+(ylist3[8]**(2)))
    trojan1R = np.sqrt((xlist3[9]**(2))+(ylist3[9]**(2)))
    trojan2R = np.sqrt((xlist3[10]**(2))+(ylist3[10]**(2)))

    plt.plot(mercuryR, mercuryVel, label="Mercury", color = "Gray")
    plt.scatter(venusR, venusVel, label="Venus", color = "Red")
    plt.scatter(earthR, earthVel, label="Earth", color = "Green")
    plt.scatter(marsR, marsVel, label="Mars", color = "#ffb347")
    plt.scatter(jupiterR, jupiterVel, label="Jupiter", color = "Orange" )
    plt.scatter(saturnR, saturnVel, label="Saturn", color ="Wheat" )
    plt.scatter(urnausR, urnausVel, label="Uranus", color = "LightBlue")
    plt.scatter(neptuneR, neptuneVel, label="Neptune", color = "PowderBlue")
    plt.scatter(trojan1R, trojan1Vel, label="Trojan 1", color = "Pink")
    plt.scatter(trojan2R, trojan2Vel, label="Trojan 2", color= "Purple")

    title = "Rotational Velocity t = " + str(tmax) + " years"
    plt.title(title)
    plt.xlabel("Radial distance (Au)")
    plt.ylabel("Rotational velocity (Au/year)")
    plt.xlim(0,10)
    plt.ylim(0,10)
    plt.legend()
    plt.show()
```

Figure 13: The Rotational method.

```
def update(frame):
    global S, elapsed_time

    #h = 0.01
    #adaptive time steps
    if elapsed_time < 5:
        h = 0.01
    elif elapsed_time < 95:
        h = 0.005
    elif elapsed_time < 300:
        h = 0.002
    if elapsed_time >= 300:
        S[:] = initial_state
        elapsed_time = 0
        print("Simulation reset")

    # Runge-Kutta integration
    S = RungeKutta(S,masses,h)

    elapsed_time += h

    xlist = S[:, 0]
    ylist = S[:, 1]

    # Update positions in the plot
    sun.set_offsets([xlist[0], ylist[0]])
    mercury.set_offsets([xlist[1], ylist[1]])
    venus.set_offsets([xlist[2], ylist[2]])
    earth.set_offsets([xlist[3], ylist[3]])
    mars.set_offsets([xlist[4], ylist[4]])
    jupiter.set_offsets([xlist[5], ylist[5]])
    saturn.set_offsets([xlist[6], ylist[6]])
    uranus.set_offsets([xlist[7], ylist[7]])
    neptune.set_offsets([xlist[8], ylist[8]])
    trojan1.set_offsets([xlist[9], ylist[9]])
    trojan2.set_offsets([xlist[10], ylist[10]])

    # Update elapsed time label
    time_label.set_text(f"Elapsed Time: {elapsed_time:.2f} years (h = {h:.5f})") #after each tick we update the label
    return sun, mercury, venus, earth, mars, jupiter, saturn, uranus, neptune, trojan1, trojan2, time_label
```

Figure 14: The update method.

```python
# Animation update function

elapsed_time = 0
timestep = 1  # Animation interval (ms) (unrelated to calculations)

def update(frame):
    global S, elapsed_time

    h = 0.01

    if elapsed_time >= 300:
        S[:] = initial_state
        elapsed_time = 0
        print("Simulation reset")

    # Runge-Kutta integration
    S = RungeKutta(S,masses,h)

    elapsed_time += h

    xlist = S[:, 0]
    ylist = S[:, 1]

    # Update positions in the plot
    sun.set_offsets([xlist[0], ylist[0]])
    mercury.set_offsets([xlist[1], ylist[1]])
    venus.set_offsets([xlist[2], ylist[2]])
    earth.set_offsets([xlist[3], ylist[3]])
    mars.set_offsets([xlist[4], ylist[4]])
    jupiter.set_offsets([xlist[5], ylist[5]])
    saturn.set_offsets([xlist[6], ylist[6]])
    uranus.set_offsets([xlist[7], ylist[7]])
    neptune.set_offsets([xlist[8], ylist[8]])
    trojan1.set_offsets([xlist[9], ylist[9]])
    trojan2.set_offsets([xlist[10], ylist[10]])

    # Update elapsed time label
    time_label.set_text(f"Elapsed Time: {elapsed_time:.2f} years (h = {h:.5f})") #after each tick we update the label
    return sun, mercury, venus, earth, mars, jupiter, saturn, uranus, neptune, trojan1, trojan2, time_label

# Tkinter setup
root = tk.Tk()
root.title("The Solar System")

fig, ax = plt.subplots()
scale_parameter = 35
ax.set_xlim(-scale_parameter, scale_parameter)
ax.set_ylim(-scale_parameter, scale_parameter)

# Scatter plots for celestial bodies
sun = ax.scatter(S[0, 0], S[0, 1], color="Yellow", label="Sun")
mercury = ax.scatter(S[1, 0], S[1, 1], color="Gray", label="Mercury")
venus = ax.scatter(S[2, 0], S[2, 1], color="Red", label="Venus")
earth = ax.scatter(S[3, 0], S[3, 1], color="Green", label="Earth")
mars = ax.scatter(S[4, 0], S[4, 1], color="#ffb347", label="Mars")
jupiter = ax.scatter(S[5, 0], S[5, 1], color="Orange", label="Jupiter")
saturn = ax.scatter(S[6, 0], S[6, 1], color="wheat", label="Saturn")
uranus = ax.scatter(S[7, 0], S[7, 1], color="LightBlue", label="Uranus")
neptune = ax.scatter(S[8, 0], S[8, 1], color="PowderBlue", label="Neptune")
trojan1 = ax.scatter(S[9, 0], S[9, 1], color="Pink", label="Trojan 1")
trojan2 = ax.scatter(S[10, 0], S[10, 1], color="Purple", label="Trojan 2")

# Add elapsed time label
time_label = ax.text(0.05, 0.95, '', transform=ax.transAxes, fontsize=12, verticalalignment='top', color = 'White')

canvas = FigureCanvasTkAgg(fig, master=root)
canvas_widget = canvas.get_tk_widget()
canvas_widget.pack(side=tk.TOP, fill=tk.BOTH, expand=True)

ani = animation.FuncAnimation(
    fig, update, frames=np.linspace(0, 10, 100), interval=timestep, blit=False
)

plt.legend(loc="upper right")
#plt.grid()
ax.set_facecolor('black')
root.mainloop()
```

Figure 15: The animation part of the code.