UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

NUMERICAL PHYSICS PROJECT REPORT

Planetary orbits in the Solar System

First Year SUTS Masters Degree

Enguerran Vidal - Jonathan Oers

January 11, 2021

Table of Contents

Introduction

Planets, from the ancient Greek *planētēs*, meaning "wanderer", are massive bodies orbiting the nearest star and that must be big enough for its own gravity to not only force it into a spherical shape but also clear away any other object of similar size near its orbit around the said star. Since ancient times, astronomers have carried out observations of the sky to identify our neighbouring planets (from Mercury to Saturn, and first including the Sun and the Moon in the lot).

Over the centuries, astronomers kept working on the observations and data gathered in order to give an accurate description of nature and Earth's surroundings. First came a geocentric model, placing the Earth as the center of the universe and other bodies would orbit our planet. But from the 16Th century, this model was questioned, with Copernicus proposing a heliocentric model of our Solar system in which the planets had a circular orbit around the sun. The model was still not perfect, but following observations made by Kepler in the early 17Th suggested the planets had an elliptical orbit, according to his laws of planetary motion. This was further explained by Newton's work on gravity and how it complimented Kepler's theory.

Uranus was later discovered in 1781 by Herschel with a telescope, and by combining Kepler's and Newton's laws, astronomers tried to predict its trajectory. Yet observations did not concur with the calculations, and the astronomers could only explain these irregularities if a farther planet's gravity was acting upon Uranus. Astronomers began calculations to determine the position of such a planet, and by 1846 Neptune was finally discovered, all according to Le Verrier's calculations, pinpointing the planet's location.

Such methods are still used today to discover large bodies, and to draw their trajectory. With the advance of technology, astronomers started using computers to complete these tasks. Today, we are able to compute the trajectories of the planets and give a 3D representation of the Solar system, and the planets' movement around the Sun according to Kepler's laws of motion.

As students undergoing studies in astrophysics, we have decided to undertake this task. By using Python as our programming language, and libraries such as *Numpy* and *Matplotlib*, we will write a code that will take into account the bodies' position and velocity in space, and by using different numerical methods to solve the differential equations generated by the laws of motion, we can give a 3D representation of the planets' orbits and their variation over time, and analyse how different methods give different results on the representation of Solar system.

Project Presentation and Objectives

2.1 The classical N-Body problem

We consider n celestial bodies of masses m_i with $i = 1, 2, 3, ..., n$ placed in a 3 dimensional inertial reference frame. The mass m_i has a position vector $\vec{q_i} = (x, y, z)$ which first and second derivative yield $\dot{\vec{q_i}} = (\dot{x}, \dot{y}, \dot{z})$ and $\ddot{\vec{q_i}} = (\ddot{x}, \ddot{y}, \ddot{z})$ respectively.
We use Newton's second law ($m\ddot{\vec{q}} = \sum \vec{F}$) where $\sum \vec{F}$ represents the forces acting on each mass. Here the forces are gravitational, given by Newton's law of gravity :

$$\overrightarrow{F_{ij}} = \frac{Gm_i m_j}{\|\vec{q_j} - \vec{q_i}\|^3}(\vec{q_j} - \vec{q_i}) \tag{2.1}$$

(Force felt by m_i from the presence of m_j)

To understand the generalized equations for a n-body problem, we consider at first a 3 body problem with $i = 1, 2, 3$. We get this set of equations :

$$m_1\ddot{\vec{q_1}} = \overrightarrow{F_{12}} + \overrightarrow{F_{13}} = \frac{Gm_1 m_2}{\|\vec{q_2} - \vec{q_1}\|^3}(\vec{q_2} - \vec{q_1}) + \frac{Gm_1 m_3}{\|\vec{q_3} - \vec{q_1}\|^3}(\vec{q_3} - \vec{q_1}) \tag{2.2}$$

$$m_2\ddot{\vec{q_2}} = \overrightarrow{F_{21}} + \overrightarrow{F_{23}} = \frac{Gm_2 m_1}{\|\vec{q_1} - \vec{q_2}\|^3}(\vec{q_1} - \vec{q_2}) + \frac{Gm_2 m_3}{\|\vec{q_3} - \vec{q_2}\|^3}(\vec{q_3} - \vec{q_2}) \tag{2.3}$$

$$m_3\ddot{\vec{q_3}} = \overrightarrow{F_{32}} + \overrightarrow{F_{31}} = \frac{Gm_3 m_1}{\|\vec{q_1} - \vec{q_3}\|^3}(\vec{q_1} - \vec{q_3}) + \frac{Gm_3 m_2}{\|\vec{q_2} - \vec{q_3}\|^3}(\vec{q_2} - \vec{q_3}) \tag{2.4}$$

By generalizing these three equations to a random set of n masses with $i = 1, 2, 3, ..., n$ as expressed at the start, we finally get for each m_i :

$$\ddot{\vec{q_i}} = \sum_{j=1, j\neq i}^{n} \frac{Gm_j}{\|\vec{q_j} - \vec{q_i}\|^3}(\vec{q_j} - \vec{q_i}) \tag{2.5}$$

The goal of this project will be to use this set of equations on nine celestial bodies at first (the Sun, the inner and outer planets), solving them from a set of initial positions and speeds. However, (2.5) is a set of n non-linear second order differential equations which do not have any analytical solutions except for a few known cases as $n = 2$ (2-Body classic Keplerian planetary problem that we will get back to later) or $n = 3$ (3-Body problem with Lagrange points as solutions). We therefore need a way to solve it. Fortunately, we could use an integration scheme.

2.2 The use of Integration Schemes

All dynamic simulations assume to discretize the temporal evolution of the system through small time steps. This time step is usually noted dt. An integration scheme is the numerical method describing how to find the approximate solution for ordinary differential equations (ODE) like we have in (2.5). If we assume our system to be in the form of :

$$\frac{dx}{dt} = f(t, v) \qquad \frac{dv}{dt} = g(t, x) \tag{2.6}$$

Then we can get an approximation of each first equation term in (2.6) depending on the order of said approximation, this gives us multiple ways of getting the values of x and v at the next time step given the previous ones. In this project, we chose five different integration schemes that could provide a range of attributes like calculation speed as well as smaller errors per time step.

First Order Integration Schemes		
Explicit Euler Method	Semi-Implicit Euler Method	Symplectic Euler Method
$v_{n+1} = v_n + dt.g(t_n, x_n)$ $x_{n+1} = x_n + dt.f(t_n, v_n)$	$v_{n+1} = v_n + dt.g(t_n, x_n)$ $x_{n+1} = x_n + dt.f(t_n, v_{n+1})$	$v_{n+1} = v_n + dt.g(t_n, x_{n+1})$ $x_{n+1} = x_n + dt.f(t_n, v_n)$

Higher Order Integration Schemes	
Runge Kutta Method	Heun Method
$v_{n+1} = v_n + dt(k_{v,1} + 2k_{v,2} + 2k_{v,3} + k_{v,4})/6$ $x_{n+1} = x_n + dt(k_{x,1} + 2k_{x,2} + 2k_{x,3} + k_{x,4})/6$ $k_{v,1} = g(t_n, x_n) \quad k_{x,1} = f(t_n, v_n)$ $k_{v,2} = g(t_n + dt/2, x_n + dt.k_{x,1}/2)$ $k_{x,2} = f(t_n + dt/2, v_n + dt.k_{v,1}/2)$ $k_{v,3} = g(t_n + dt/2, x_n + dt.k_{x,2}/2)$ $k_{x,3} = f(t_n + dt/2, v_n + dt.k_{v,2}/2)$ $k_{v,4} = g(t_n + dt, x_n + dt.k_{x,3})$ $k_{x,4} = f(t_n + dt, v_n + dt.k_{v,3})$	$v_{n+1} = v_n + dt(k_{v,1} + k_{v,2})/2$ $x_{n+1} = x_n + dt(k_{x,1} + k_{x,2})/2$ $k_{v,1} = g(t_n, x_n) \quad k_{x,1} = f(t_n, v_n)$ $k_{v,2} = g(t_n + dt, x_n + dt.k_{x,1})$ $k_{x,2} = f(t_n + dt, v_n + dt.k_{v,1})$

2.3 Project Objectives and Requirements

Main Objectives

- ⟶ Make a 3D animation of the Solar System, with the Sun at its center and showing the planets' orbits around it.
- ⟶ Access the total energy of the system and track its conservation tochoose a good integration scheme.
- ⟶ Plot the orbital shifts of the Solar System planets.
- ⟶ Observe the same results for other planetary systems such as $TRAPPIST-1$ and $Kepler-79$.

Code Requirements

- ⟶ Our final code needs to possess an object-oriented architecture, fully using the available *class* system in the Python programming language. This will ensure a greater interactivity with the user instead of random scripts being run. We also need for each subsequent created classes or functions to possess a well documented annotation to facilitate their usage. This will in turn help to avoid as much user-made errors as possible.
- ⟶ The Code needs to be able to have a fully-working Newtonian classical physics engine able to model the interactions between the celestial bodies inputted, as well as integrate our set of equations between each time step by using the diverse range of integration schemes we mentioned in section 2.2. We also need to fully use the *Numpy* library to shorten calculation times. The engine will therefore try to use as many arrays as possible in its variables and calculations.
- ⟶ We need our code to have a way to keep track of the total energy of our system to help in our search for the best integration scheme.
- ⟶ The code engine must be able to do multiple calculation runs, possibly back-to-back.
- ⟶ The code needs to be able to plot our Sun and planets in a 3D graph that animates itself, passing through previously calculated data in order to have a decent FPS count and fully-utilizing the graphical functions and objects from the *Matplotlib* library.
- ⟶ The code must be able to access perihelia and orbital shifts values from the celestial bodies' position and speeds arrays. It also needs to plot these planetary orbits in the previously mentioned 3D animation.
- ⟶ In order to utilize already calculated data at a later date, the code must be able to save them in a .txt file easily identifiable by its name which should be stored in a sub-folder to keeps things clean. It also needs to create the folder if non-existent.
- ⟶ The initial position and speed arrays, masses and names of our studied celestial bodies need to be extracted from an initialization file containing ephemeride data.

Making the Code

3.1 Creation of an object-oriented class tree

A class is a way of bundling the data stored in variables and the functionality of functions. By creating a class, a brand new type of object is created. Python is well-known for its class system which can be implemented with the learning of a minimal amount of new semantics and syntax.

A class-created object has multiple interesting assets for it can contain multiple attributes or specific variables that can be used or called inside or outside its code, some of them can even be class-created objects, creating an inter-dependant object tree. it allows the creation of methods, functions that can change its state as well as return variables and do various operations in the mean time. The Python class system will help this project immensely by giving a better user interaction with the program. Execution scripts can then be done simply by first initializing the right class-created objects and then, using their methods, coming up with the desired output. In this project's case, we can pinpoint a few classes we could create.

Ephemeride_Database

This class' job would be to handle the ephemeride data needed to input the planetary system's initial positions and speeds as well as the celestial bodies' respective names and masses. It would therefore need one method to load ephemeride data from a given .txt file and another method capable of adding an object in the data file by respecting the data format inside the file.

NBody_Engine

This class will handle all the calculations. In order to do that, it needs to have an imprint of the planetary system state as one of its attributes and several methods calculating resulting gravitational forces, vectors and distances. It is this class that will need to possess all the integration schemes we wish to test further in our project.

Planetary_System

This class would be the general actor and interface for the program user, it would need to request calculations, make new saves as well as load old ones. It would also be amongst its methods that the 3D animations and different plots creations will reside. It should also be able to fully initialize an *Ephemeride_Database* object as its initialization database and a *NBody_Engine* object as its calculation engine. it also needs to be the most user-friendly of the three.

3.2 Making a Newtonian calculation engine

As mentioned right above, the *NBody_ Engine* class will handle all calculations through a Newtonian straight up approach. We could always use high end numerical methods to speed them up greatly such as the Barnes-Hut method [5]. However this project will not require any approximations since the number of objects is quite low in our case; but we can still speed up the calculation process by the use of *numpy.array* from the *numpy* library. It uses strips of C code to speed up calculation in matrices-like data formats. In this calculation engine, we therefore heave an array for the objects' positions (*self.objects_ X*) and another for their speeds (*self.objects_ V*) as shown below :

$$\vec{R} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ & \vdots & \\ x_i & y_i & z_i \\ & \vdots & \\ x_N & y_N & z_N \end{bmatrix} \quad \vec{V} = \begin{bmatrix} \dot{x}_1 & \dot{y}_1 & \dot{z}_1 \\ \dot{x}_2 & \dot{y}_2 & \dot{z}_2 \\ & \vdots & \\ \dot{x}_i & \dot{y}_i & \dot{z}_i \\ & \vdots & \\ \dot{x}_N & \dot{y}_N & \dot{z}_N \end{bmatrix} \qquad \vec{A} = \begin{bmatrix} a_{x1} & a_{y1} & a_{z1} \\ a_{x2} & a_{y2} & a_{z2} \\ & \vdots & \\ a_{xi} & a_{yi} & a_{zi} \\ & \vdots & \\ a_{xN} & a_{yN} & a_{zN} \end{bmatrix} \longleftarrow \sum_{j=1,j\neq i}^{N} \frac{Gm_j}{\|\vec{q_j}-\vec{q_i}\|^3}(\vec{q_j}-\vec{q_i}) \tag{3.1}$$

with $\vec{q_i} = [x_i, y_i, z_i]$ and $\vec{q_j} = [x_j, y_j, z_j]$. We can get access to a specific object's position or speed by referring to a specific line extracted from its index in the matrix. However to get the acceleration from each object at each time step we do need to build up its matrix bit by bit as shown above which is a little bit more difficult. The objects' acceleration is what will be used in the formulation of $g(t, x)$ mentioned in (2.6). Although, depending on each integration scheme, this formulation changes such as in the Runge-Kutta method ($k_{v,2} = g(t_n + dt/2, x_n + dt.k_{x,1}/2)$). Since the inputted positions need to be different in these cases, the function responsible for the acceleration needs to have a position matrix as one of its inputs. This task is handled by the *acceleration* method in the *NBody_ Engine* class.

To be able to test the different integration schemes freely, we also need an array of methods, each calculating the system state with each time step using a different integration scheme, we will also build a method whose sole purpose is to apply the choice of scheme from the user, this is done by the *compute* method which chooses which sub-method to use from the user's choice.

Figure 3.1: Example of the format used for our ephemeride file

```
2020-08-01-00-00-00
name,mass,x,y,z,xp,yp,zp
Sun,1.0,0.0,0.0,0.0,0.0,0.0,0.0
Earth,3.00348959632e-06,0.6378521924452,-0.7243479827437,-0.3140047136535,0.0130952993725,0.0098600669548,0.0042749866373
Mercury,1.66092974676496e-07,0.2216871020797,0.207043738136,0.0876225602695,-0.025595375398,0.0175954364737,0.0120524505805
Venus,2.4478440210008e-06,0.7031786170452,-0.1490019019144,-0.111535940392,0.0049404993498,0.0178947400705,0.0077391916833
Mars,3.303838555952e-07,1.247283698683,-0.5262674367033,-0.2750412121075,0.0065366656713,0.0126438695982,0.0056230469661
Jupiter,0.000954508993710496,2.0574283157358,-4.3205923673908,-1.9020078692296,0.006833904262,0.003162626707,0.0011892424822
Saturn,0.000285932209569664,4.805400215558,-8.0404171162937,-3.5280265669176,0.004591813035,0.0025380768203,0.0008507310595
Uranus,4.3850948106272e-05,15.7259008587095,11.088034969258,4.6338667047234,-0.0024104646963,0.0026854683842,0.0012100976413
Neptune,5.1660021056704e-05,29.3748816307801,-5.014104735085,-2.7837188632818,0.0005851530628,0.002879208611,0.0011639556301
Pluto,7.396e-09,13.6092469741878,-28.4367417340758,-12.9738110109588,0.0029585295432,0.0009023115967,-0.0006072174576
```

To initialize the objects' positions and speeds at the beginning of the calculations, we will use a .txt file containing ephemeride data, these will be loaded up by the *Ephemeride_Database* and directly given to the *NBody_Engine* class during its initialization. The data shown in figure 3.1 is arranged so that each line gives the object's name, mass, position and speed in our unit system, which, to be more compliant with the problem statement will be : days for units of time, AUs (Astronomical Units) for units of distance and units of mass will be described in $M_{\odot}$ (Solar masses). The data itself is provided by the IMCCE Ephemeride Miriade generator [4] where we took it from the date of *August* 1^{st} 2020.

3.3 Making a session and data saving system

As mentioned above, we wanted the *Planetary_System* class to be as user-friendly as possible. By taking into account the user's needs, such as having direct access to the file containing the computed data (which is also required in later sections) as well as wanting to store the data in case the user wants to utilize it at a later date, a session and data saving system was added to this class.

In order to do so, we had to think about how to retrieve the data and thus, how to save it as well. When handling the code, the user does not have access to the data yet. If the user decides to display anything or run further calculations without having an open session, an error message will be displayed, as the data has yet to be retrieved. As such, the user will have two choices to be able to utilize the data:

- the user can run the calculations, this will open a new session and create a file containing the data. The session will remain open, the user can handle the data, and it will be saved for later utilization.
- the user can load an older session, using data that has already been calculated, stored, and can now be used again. The last saved system state will serve in initializing it for further calculations and can be used immediately for other purposes.

If the user wishes to load older data, the files must be easily recognizable, so we chose to use the date and time when the user first ran the calculations as a file name, with the following format: $yyyy - mm - dd - hh - mm - ss$. If the user ever inputs the file name incorrectly, an error message will be displayed, canceling the loading sequence.

Since we want the code to run the calculations between each time step when using the integration schemes, the data will be saved in the file accordingly. For each time step, a total of twelve parameters for each body (the three coordinates of space and velocity and all six keplerian orbit parameters that will be introduced in the next section) are calculated using *NBody_Engine* and are saved in the save file, with its first line providing information on the number of saves and the ephemeride file used for the objects' definition. Thus, when reloading the data, the first line is skipped, and all thirteen values (including the current time) will be retrieved for further utilization, one time step at a time.

Finally, in order to keep things tidy, all the created data files will be stored in a *logs* folder located in the same directory as the code. If such a folder does not exist, it will automatically be created.

Figure 3.2: Example of the format used in the data save file

```
Base_File Solar_System.txt 1668
0
0.0 0.6378521924452 0.2216871020797 0.7031786170452 1.247283698683 2.0574283157358 4.805400215558 15.7259008587095 29.3748816307801 13.6092469741878
0.0 -0.7243479827437 0.207043738136 -0.1490019019144 -0.5262674367033 -4.3205923673908 -8.0404171162937 11.088034969258 -5.014104735085 -28.4367417340758
0.0 -0.3140047136535 0.0876225602695 -0.111535940392 -0.2750412121075 -1.9020078692296 -3.5280265669176 4.6338667047234 -2.7837188632818 -12.9738110109588
0.0 0.0130952993725 -0.025595375398 0.0049404993498 0.0065366656713 0.006833904262 0.004591813035 -0.0024104646963 0.0005851530628 0.0029585295432
0.0 0.0098600669548 0.0175954364737 0.0178947400705 0.0126438695982 0.0031626267707 0.0025380768203 0.0026854683842 0.002879208611 0.0009023115967
0.0 0.0042749866373 0.0120524505805 0.0077391916833 0.0056230469661 0.0011892424822 0.0008507310595 0.0012100976413 0.0011639556301 -0.0006072174576
0.0 0.999296287241749 0.38709842803729494 0.723329156607765 1 1.5237239820762256 5.208987880044009 9.583637686821607 19.195302551610528 30.235475577051115 39.85626348241374
0.0 0.017439528169169913 0.2056326324738167 0.006779979186074284 0.0934189569752288 0.04888019835986256 0.0510815760784346 0.0456790208823228 0.011359344559583517 0.25255397318002215
0.0 0.4090700870975945 0.4983463975400991 0.4264902879806921 0.43069951504366977 0.4055228575163854 0.3936241426814378 0.4129688816043776 0.3891645499440268 0.409040342118854 4
0.0 6.283116516374416 0.19165773699998673 0.13970492768488127 0.05876055119488786 4 0.056728547706546124 0.10383241170911564 0.0322243972436704 0.06074912005461362 0.7660088189701953
0.0 1.803651165304724 7 1.179942610872377 2.171906344721207 5.812411546533368 0.17130581347640642 1.4843491086213714 2.975051336531835 0.2289489247670424 3 3.230485585387284
0.0 3.588364291796238 5.722723085229964 3.7315530374734385 6.2568579142328264 4.900063581968611 3.63328881505279 3.9310203543957942 5.8065741398322075 1.1870664465466412
```

```
Base_file   Ephemeride file   Number of saves
Time value
X from 1 to N
Y from 1 to N
Z from 1 to N
Vx from 1 to N
Vy from 1 to N
Vz from 1 to N
Semimajor axis from 1 to N
Eccentricity from 1 to N
Inclination from 1 to N
Longitude of the ascending node from 1 to N
Argument of periapsis from 1 to N
True anomaly at epoch from 1 to N
```

3.4 Drawing planetary orbits in 3D

One of the main goals of our project is to be able to plot the orbits and trajectories of the Solar System's planets and update them as the planets move around the Sun. The method that will be responsible for displaying this 3D graph must be handled by the *Planetary_System* class as it commands the two others and has therefore access to their attributes. Here, to plot our data in 3D, we will use *Axes3D* from the *mpl_toolkits.mplot3d* as our graphic artist. This object renders the usual *matplotlib* graphic objects like lines or plots in a 3D manner. They are further described in the Matplotlib 2.0.2 version documentation [3]. Here, we will use the *Axes3D.plot* to plot lines for our orbits and *Axes3D.scatter* to plot the points that will represent our planets and Sun. However, how do we make them move at each time step to create an animation ?

To tackle this hurdle, *matplotlib* comes again to the rescue with its wide array of methods for each that can be used to update these objects' attributes, even after showing the final plot! All it needs is a little time delay to avoid moving too fast through our data (that can be done using the plt.pause() function that stops the program for a desired amount of time) and to use plt.draw() to show the changes when the program is done updating the objects' positions and attributes.

One problem still remains, we do have the position vectors to place our planets, but how do we define their orbits and manage to plot it?

Keplerian Elements

In the next pages, we will assume the trajectory of our planets resemble greatly a standard Keplerian 2-Body problem conic solution. To best define a Keplerian orbit, we need a few parameters that can derive a function easy to plot in 3D. We can use the so-called 6 Keplerian elements that are widely used such as in the TLEs (Two Line Elements) format which eases satellite tracking immensely [6]. They are a set of 6 parameters varying with each specific usage. However, we will here use :

- ⟶ semimajor axis, a
- ⟶ eccentricity, e
- ⟶ inclination, i
- ⟶ longitude of the ascending node, Ω
- ⟶ argument of periapsis, ω
- ⟶ true anomaly at epoch, θ_0

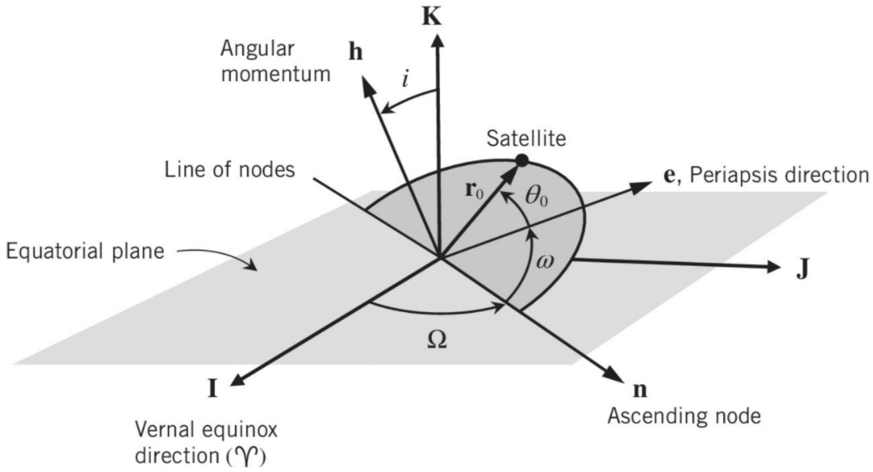


Figure 3.3: Visualization of the Keplerian elements from [2]

The method to obtain them is best described in chapter 3 of "Space Flight Dynamics" [2]. It is used in our code in the *orbital_parameters* function. It returns the 6 previous elements from the position and speed vectors, the gravitational constant and the central body's mass (here the Sun). Let's assume $\vec{r}_0$ and $\vec{v}_0$ are our initial vectors and $\mu = GM_\odot$:

$$\zeta = \frac{v_0^2}{2} - \frac{\mu}{r_0} \qquad a = \frac{-\mu}{2\zeta} \tag{3.2}$$

$$\vec{e} = \frac{1}{\mu}\left[(v_0^2 - \mu/r_0)\vec{r}_0 - (\vec{r}_0\vec{v}_0)\vec{v}_0\right] \qquad and\ e\ norm\ of\ \vec{e} \tag{3.3}$$

$$\cos i = \frac{\vec{K}.\vec{h}}{h} \qquad with\ \vec{h} = \vec{r}_0 \times \vec{v}_0 \tag{3.4}$$

$$\cos\Omega = \frac{\vec{I}.\vec{n}}{n} \qquad \sin\Omega = \frac{\vec{J}.\vec{n}}{n} \qquad with\ \vec{n} = \vec{K} \times \vec{h} \tag{3.5}$$

$$\cos\omega = \frac{\vec{n}.\vec{e}}{ne} \qquad \cos\theta = \frac{\vec{e}.\vec{r_0}}{er_0} \tag{3.6}$$

Plotting the orbits

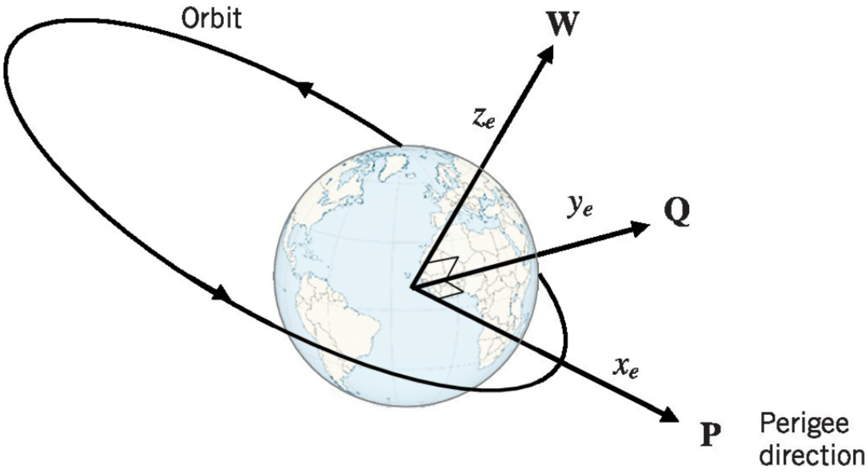


Figure 3.4: Visualization of the perifocal coordinate system from [2]

After obtaining them, we need to decide what type of conic section the object's trajectory will resemble, as described again in chapter 2 of [2]. It all depends on the eccentricity value e. If $e < 1$, we have an ellipse and if $e \geqslant 1$ we have a parabola or hyperbola. For each of these trajectories, the formula will be different. Each time we create a *Theta* list of true anomalies to plot our orbit with. By placing ourselves in the **PQW** reference frame (Perifocal coordinate system) shown in figure A.6 where :

$$\vec{r}_{\mathbf{PQW}} = \begin{bmatrix} r\cos\theta \\ r\sin\theta \\ 0 \end{bmatrix} \quad with\ \ p = a(1-e^2) \tag{3.7}$$

$$\underline{Case\ \ e < 1}: \qquad r = \frac{p}{1 - e\cos\theta} \qquad \theta \in [-\pi, \pi] \tag{3.8}$$

$$\underline{Case\ \ e \geqslant 1}: \qquad r = \frac{p}{1 - e\cos\theta} \qquad \theta \in\ ]-\theta^+_{\text{inf}}, \theta^+_{\text{inf}}[\quad with \quad \theta^+_{\text{inf}} = \arccos\frac{-1}{e} \tag{3.9}$$

Then we retrieve the orbit's points from a change of reference frame using :

$$\vec{r}_{\mathbf{IJK}} = \left[\begin{bmatrix} \cos\omega & \sin\omega & 0 \\ -\sin\omega & \cos\omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos i & \sin i \\ 0 & -\sin i & \cos i \end{bmatrix} \begin{bmatrix} \cos\Omega & \sin\Omega & 0 \\ -\sin\Omega & \cos\Omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \right]^T \vec{r}_{\mathbf{PQW}} \tag{3.10}$$

This gives use access to a list of positions for points of an orbit that we can then plot in 3D as described earlier in this very section. However, these orbits are defined and drawn around the center of the graph but the Sun is also a moving object, this could give us some terrible results if we do not center the reference frame of our system back on the Sun, we can do that by subtracting the Sun's position to everyone's after each *NBody_Engine.compute()* call in *Planetary_System.RUN()*.

3.5 Tracking energy conservation

In order to choose an integration scheme wisely, we must be able to track the effects of their respective errors on the system's total mechanical energy. This can be done by summing the mechanical energy of each planet (we exclude the Sun as our system lies in a heliocentric reference frame). We are in luck to have defined an easy way of gaining each planet's energy from their Keplerian parameters. In fact we can write the total energy of our planets E as :

$$E = \sum_{i=1}^{N-1} m_i \zeta_i \quad with\ each\ \zeta = \frac{-\mu}{2a} = \frac{v^2}{2} - \frac{-\mu}{r} \tag{3.11}$$

We can conclude that the total energy of our system can easily be tracked through the values of the semimajor axis which we can then plot as a function of time. We will only need to compare the behaviours of this variable between each integration scheme we introduced in section 2.2.

3.6 Accessing the orbital perihelion's shifts

The ellipse that a planet's orbit draws around the central body is not necessarily stationary, it will most likely slightly shift over time, with each of the planet's revolution around the central body. In order to observe that phenomenon, we would like to plot how the periapsis shifts over time. This will also be handled by the *Planetary_System* class, as it gives us access to various orbital parameters, but most importantly the argument of periapsis.

This parameter describes the angle of an orbiting body's periapsis (the point where the orbiting body is closest to the central body). This angle is measured between the orbital plane (plane of reference for the central body) and the periapsis, in the direction of motion. Since the calculations of the argument of periapsis for each time step have already been handled, all that is left to do is to retrieve the values, then calculate the actual shift by subtracting the initial value as:

$$\Delta\psi_i = \omega_i - \omega_0 \tag{3.12}$$

and to plot them as a function of time , by using the usual *matplotlib* graphic objects. The function handling tis task is *Planetary_System.apsidal_precession()* which will be able to plot them for each requested planet.

3.7 Creating new planetary systems

Having a complete code modeling the interactions between the Solar system's planets means it could be applied to other planetary systems. this could be done by simply inputting a new ephemeride file containing data specific to this new planetary system. However, creating this file would be hard without having having ephemeride-like data, meaning speeds and positions. We will use the same principle as in section 3.4 to access them from keplerian elements provided on the general Exoplanet Catalog [1].

$$\vec{r}_{\mathbf{PQW}} = \begin{bmatrix} r\cos\theta \\ r\sin\theta \\ 0 \end{bmatrix} \qquad \vec{v}_{\mathbf{PQW}} = \begin{bmatrix} -\frac{\mu}{h}\sin\theta \\ \frac{\mu}{h}(e+\cos\theta) \\ 0 \end{bmatrix} \tag{3.13}$$

$$\vec{r}_{\mathbf{IJK}} = \left[\begin{bmatrix} \cos\omega & \sin\omega & 0 \\ -\sin\omega & \cos\omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos i & \sin i \\ 0 & -\sin i & \cos i \end{bmatrix} \begin{bmatrix} \cos\Omega & \sin\Omega & 0 \\ -\sin\Omega & \cos\Omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \right]^T \vec{r}_{\mathbf{PQW}} \tag{3.14}$$

$$\vec{v}_{\mathbf{IJK}} = \left[\begin{bmatrix} \cos\omega & \sin\omega & 0 \\ -\sin\omega & \cos\omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos i & \sin i \\ 0 & -\sin i & \cos i \end{bmatrix} \begin{bmatrix} \cos\Omega & \sin\Omega & 0 \\ -\sin\Omega & \cos\Omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \right]^T \vec{v}_{\mathbf{PQW}} \tag{3.15}$$

The function *kepler_ to_ cartesian* uses this principle to calculate components of speed and position from which we can create a new ephemeride file using the *add_ object* function in the *Ephemeride_ Database* class. This new file will then be available to begin calculation on an exo-planetary system. However, we must address that null eccentricities and inclinations of 90° have to be avoided since they could glitch the first time step use of the *orbital_ parameters*.

Results and Observations

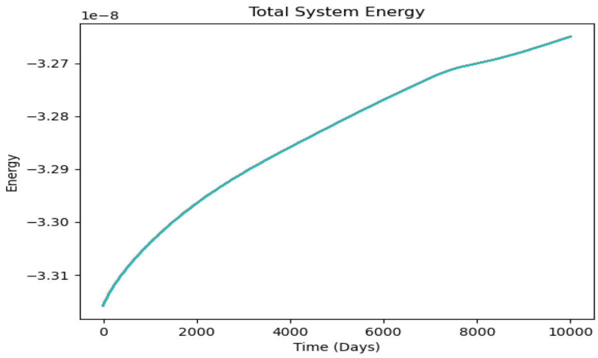
4.1 Influence of the choice of integration scheme on energy conservation

We use the following sequence of commands to display the total energy of our system after using the different integration schemes listed in section 2.2. This will plot each energy curve one by one that we can capture and close to pass to the next.

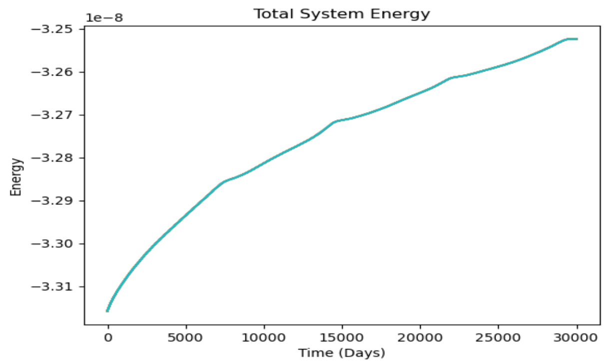
```

# Here goes the main code found in appendix main.py

################################################################################
#--------- SCRIPT ---------#

X=Planetary_System('Solar_System.txt')

X.RUN(0.1,10**4,5,method='Euler_explicit')
X.energy_conservation()

X.new_session('Solar_System.txt')
X.RUN(0.1,10**5,5,method='Euler_semi_implicit')
X.energy_conservation()

X.new_session('Solar_System.txt')
X.RUN(0.1,3*10**4,5,method='Euler_symplectic')
X.energy_conservation()

X.new_session('Solar_System.txt')
X.RUN(0.1,10**4,5,method='Heun')
X.energy_conservation()

X.new_session('Solar_System.txt')
X.RUN(0.1,3*10**4,5,method='Runge_Kutta')
X.energy_conservation()
```

It is important to note that we sometimes have adjusted the values of T the final time in days to observe the behaviour of our system further in time as it did not move much in the first hundreds of iterations and the energy plotted vertically is in $M_{\odot}/AU^2/d^{-2}$ from our problem reference units. The iterative time step dt has been left unchanged throughout all these calculations in order to fairly judge each of them. The resulting energy curves can be observed on figure 4.1. As we can clearly see, the Explicit, Symplectic, Heun and Runge Kutta method make the system total energy slowly rise, the Explicit method makes it rise at a higher rate than the others. However, the result we have gotten for the Semi-Implicit method is quite particular, we can see the energy fluctuating around its initial value, having spikes at a periodic rate. To have a better view, we will zoom on these spikes, getting what we can see on figure 4.2a to find a period of around 7500 days which translates to 20.54 years. When we zoom even more we can see tinier oscillations with a period of around 80-90 days, this is clearly a result of the orbit of Mercury since its period is close to this number, see figure 4.2b.

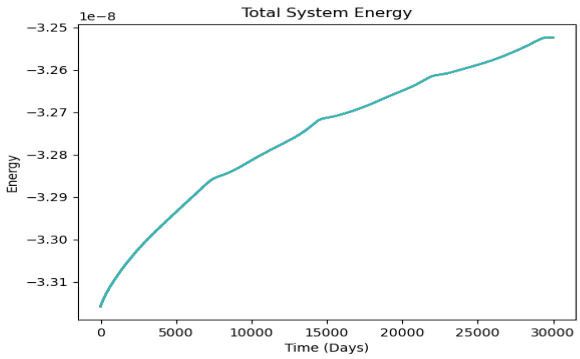
It seems the smarter choice is the Semi-Implicit method, we will use it in the creation of the next results as our integration scheme of choice.



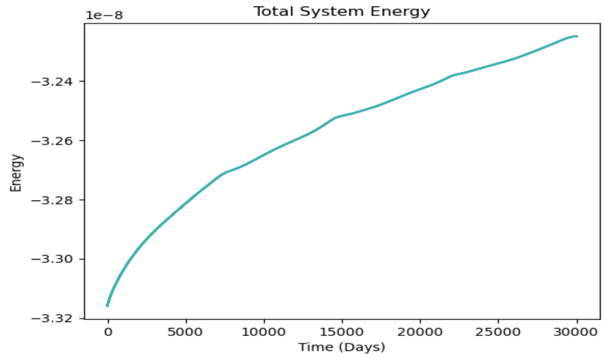
(a) Euler Explicit



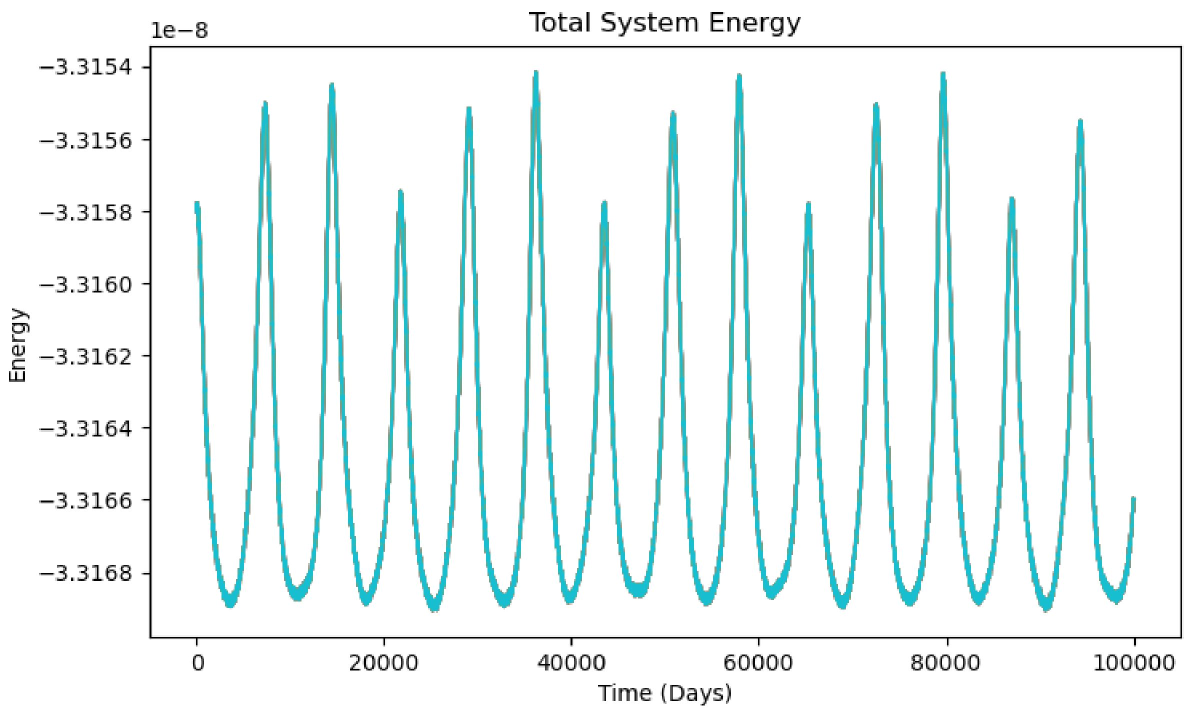
(b) Heun



(c) Runge Kutta

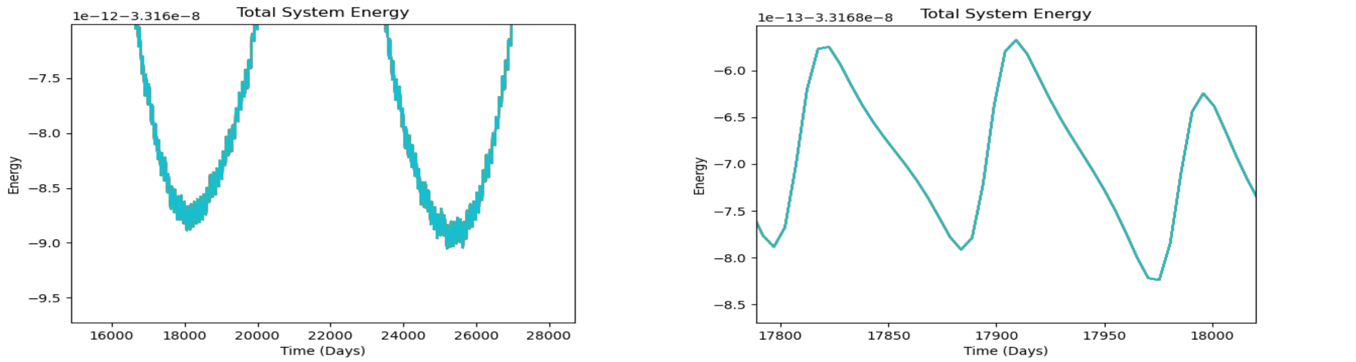


(d) Euler Symplectic



(e) Euler Semi-Implict

Figure 4.1: Energy curves for the different integration schemes, E is in $M_{\odot}/AU^2/d^{-2}$



(a) Big Oscillations with a period of 7500 days

(b) Smaller Oscillations with a period of 80-90 days

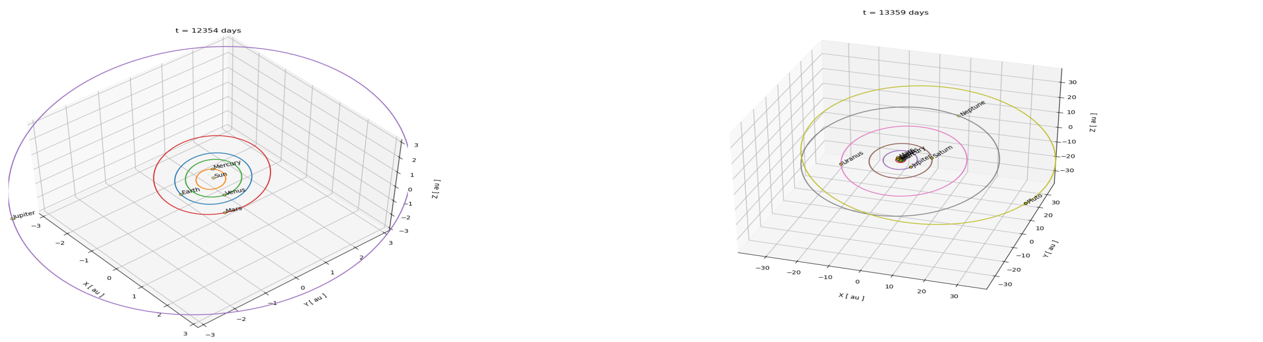
Figure 4.2: Zoom on the Semi-Implicit results

4.2 3D View of Planetary Orbits

Semi-Implicit Results

Given the choices we made in the previous section, we use the following sequence of commands to plot a 3D display of the planets of the Solar System orbiting around the Sun, with the orbits moving around as well. It is important to note that we renamed the data save files that resulted from the script above in the *logs* folder to better differentiate them.

```

# Here goes the main code found in appendix main.py

##############################################################################
#--------- SCRIPT ---------#

X=Planetary_System('Solar_System.txt')

X.load_session("semi_implicit.txt")
X.display_3D()
```



(a) A close-up of the Telluric planets

(b) A broader view of the 8 main planets... and Pluto for nostalgic purposes

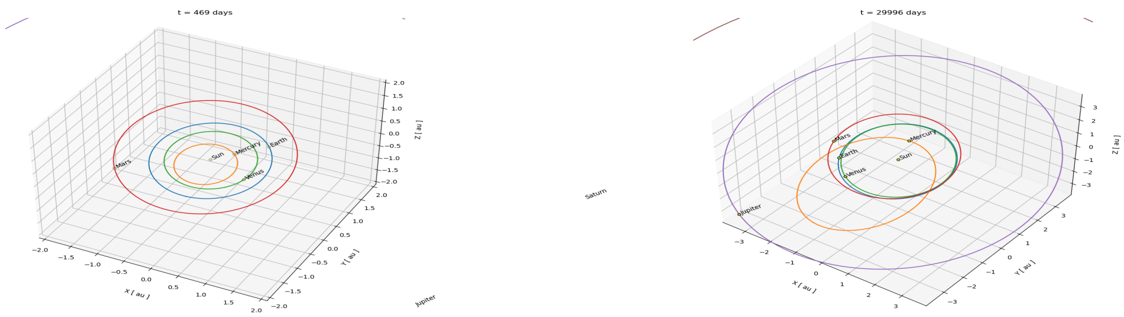
Figure 4.3: View of the 3D graph displayed with the results of the semi-implicit method

After a while, the orbits do not seem to derail from their usual position except for Mercury whose orbit "wobbles" slightly. It seems pretty consistent with our previous results since the total system energy seems constant over long periods of time. But what would happen to the planets trajectories in the case of another one of these integration schemes ?

Symplectic Results

```

# Here goes the main code found in appendix main.py

################################################################################
#--------- SCRIPT ---------#

X=Planetary_System('Solar_System.txt')

X.load_session("symplectic.txt")
X.display_3D()
```

This script gives us the following results:



(a) Starting positions of the Telluric planets

(b) After a while, their orbits go crazy

Figure 4.4: View of the 3D graph displayed with the results of the semi-implicit method

The orbits of the Telluric planets are heavily affected by the error piling on at each time step. The rise in energy of our system may come from the sudden overreach of Mercury's orbit which reaches as far as Jupiter's at the end of the simulation.

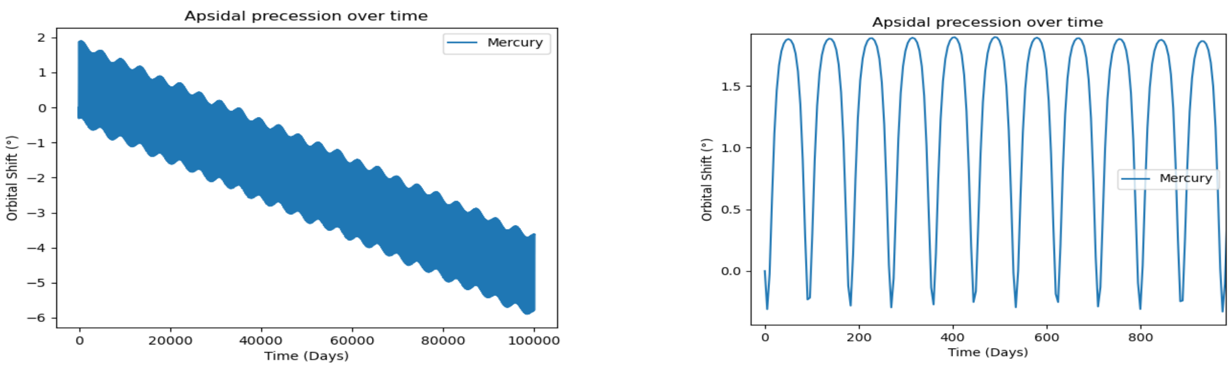
All the previous images of the 3D orbis can be found in better resolution in the appendices.

4.3 Orbital perihelion shifts

We run this given sequence of commands to plot the orbital shifts for the Planet Mercury, to plot them for all the planets present here, we would enter *displayed='all'*

```

# Here goes the main code found in appendix main.py

################################################################################
#--------- SCRIPT ---------#

X=Planetary_System('Solar_System.txt')

X.load_session("semi_implicit.txt")
X.apsidal_precession(displayed=['Mercury'])
```

This returns the figure 4.5a. It shows a downward trend for Mercury's argument of perihelion, big oscillations are visible at this scale with smaller ones present if we zoom on the time scale, this is shown on figure 4.5b.



(a) The orbital Shift of Mercury over time

(b) A zoom on the smallest oscillations

Figure 4.5: View of the orbital shift of Mercury

4.4 Changing the planetary system

After observing the results for the Solar System, we ran the same code on two other planetary systems: $TRAPPIST-1$ and $Kepler-79$. Both are multi-planetary systems, counting four or more planets, but they are both quite different from the Solar System.

For TRAPPIST-1, the scale of the planetary system is much smaller. The star's mass is around a tenth of the Solar mass, and all of the planets' orbits are at most as close as Mercury's orbit, the closest planet completing a full revolution in a mere 1.5 days! For Kepler-79, while the star's size is around the same as the Sun's (its mass is 1.15 times the Solar mass), all four planets still orbit closer to their star than the Earth orbits the Sun. For both systems, all planets have a similar mass, thus a planet's gravitational pull is not that noticeable compared to other planets, as opposed to Jupiter's case.

These differences make both these systems interesting study cases, as we'll see the influence of the different parameters as we run the code. The first problem encountered was the time step dt. As these planets' orbits are much smaller, thus much faster than the ones in the Solar System, if the time step is not adapted for each planetary system, the calculations will not be precise enough to observe the orbit we are looking for. Similarly, the amount of snaps taken and the total number of days can be reduced aswell, in order to reduce the number of calculations, since we will already be able to analyse the important results.

Since we had previously verified that the semi-implicit method was the best choice in terms of the conservation of the system's total energy, we will give the results we obtained by using this method. Of course, other methods can also be used in order to see their effects on smaller planetary systems.

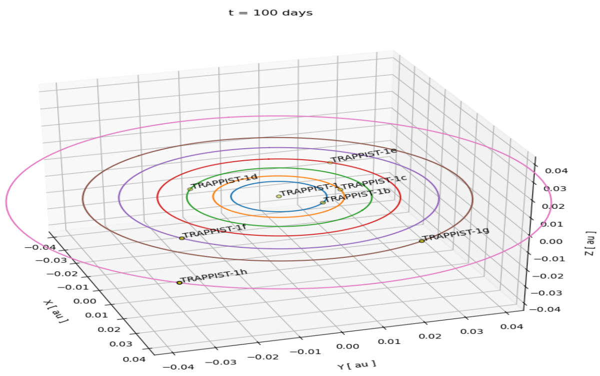
```
# Here goes the main code found in appendix main.py

#################################################################################
#--------- SCRIPT ---------#

X=Planetary_System('TRAPPIST-1_System.txt')
X.RUN(0.001,10**2,0.1,method='Euler_semi_implicit')

X.display_3D()
X.energy_conservation()
```

```
# Here goes the main code found in appendix main.py

#################################################################################
#--------- SCRIPT ---------#

X=Planetary_System('Kepler-79_System.txt')
X.RUN(0.01,10**3,1,method='Euler_semi_implicit')

X.display_3D()
X.energy_conservation()
```

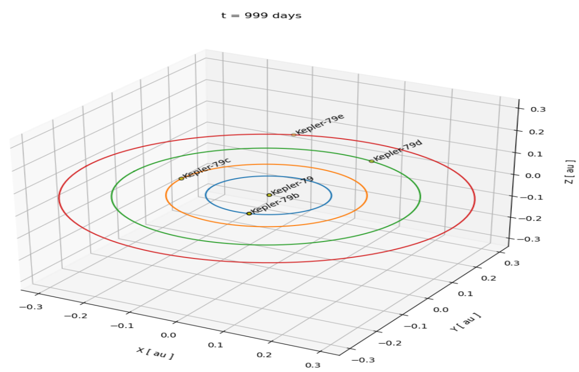
The first two lines of each script run the calculations. The third line will give us the 3D animation for each planetary system, as shown on figures 4.6. The fourth line with plot the variation of the planetary systems' total energy over time, as shown on figures 4.7

Once again, the use of Semi-Implicit method gives us solid results on the overall conservation of energy for both planetary systems, as it did in the Solar system's case.

Further tests can be conducted, using smaller time steps, or calculate the values over a larger amount of time, we could also have a view on how the orbits shift for the planets in these planetary systems. What's most interesting is how we are now able to display and compare different planetary systems. System's larger than our Solar system should also make up interesting case studies, in order to see the different behaviours on an even larger scale.

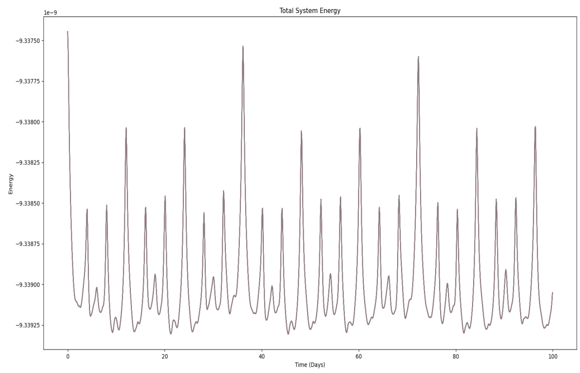


(a) TRAPPIST-1 planetary orbits

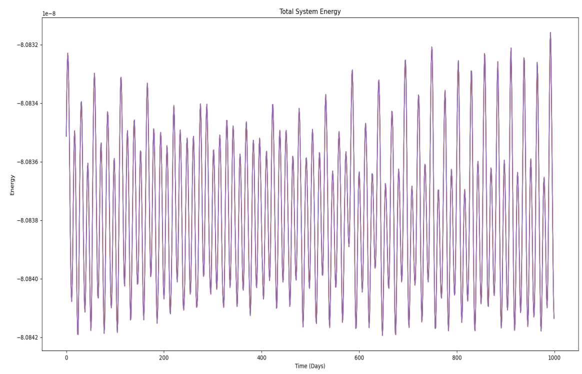


(b) Kepler-79 planetary orbits

Figure 4.6: View of the 3D graph displayed for each planetary system with the results of the semi-implicit method



(a) TRAPPIST-1 planetary system



(b) Kepler-79 planetary system

Figure 4.7: Total energy for different planetary systems

References

[1] *Exoplanet Catalogue*. URL: exoplanet.eu.

[2] Craig A. Kluever. *Space Flight Dynamics*. Aerospace Series. Wiley.

[3] *mplot3d tutorial - Matplotlib 2.0.2 documentation*. URL: https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html.

[4] *PORTAIL SYSTÈME SOLAIRE OBSERVATOIRE VIRTUEL DE L'IMCCE - Miriade*. URL: http://vo.imcce.fr/webservices/miriade/?forms.

[5] *The Barnes-Hut Approximation*. URL: https://jheer.github.io/barnes-hut/.

[6] *TLE/Keplerian Elements Resources - AMSAT*. URL: https://www.amsat.org/keplerian-elements-resources/.

Appendix

Main Program

```
# DATA MANIPULATION AND PLOTTING MODULES
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# FILE HANDLING MODULES
import os
import sys
import time


################################################################################
#--------- SIMULATION CLASS ---------#

class NBody_Engine():
    ''' This class serves as a calculation engine to solve the N-Body problem.'''
    def __init__(self):
        self.objects_name=[]
        self.objects_mass=[]
        self.objects_X=[]
        self.objects_V=[]
        self.objects_type=[]
        self.n_objects=0

    def define_objects(self,objects):
        ''' Defines and creates the arrays previously initiated using the
    inserted bodies' parameters '''
        n=len(objects)
        if self.n_objects==0:
            self.objects_X=np.zeros(shape=(n,3))
            self.objects_V=np.zeros(shape=(n,3))
            for i in range(n):
                self.objects_X[i]=objects[i][1]
                self.objects_V[i]=objects[i][2]
                self.objects_name.append(objects[i][0])
                self.objects_mass.append(objects[i][3])
        else:
            new_X=np.zeros(shape=(self.n_objects+n,3))
            new_V=np.zeros(shape=(self.n_objects+n,3))
            new_X[0:self.n_objects]=self.objects_X[0:self.n_objects]
            new_V[0:self.n_objects]=self.objects_V[0:self.n_objects]
            self.objects_X=new_X
            self.objects_V=new_V
            for i in range(n):
                self.objects_X[self.n_objects+i]=objects[i][1]
                self.objects_V[self.n_objects+i]=objects[i][2]
                self.objects_name.append(objects[i][0])
                self.objects_mass.append(objects[i][3])
                self.objects_type.append(objects[i][4])
        self.n_objects=self.n_objects+n
```

```
    def objvect(self,objects_X,i,j):
        ''' Returns the vector from body i to j '''
        return self.objects_X[j]-self.objects_X[i]

    def objdist(self,objects_X,i,j):
        ''' Returns the distance between bodies i and j '''
        X=self.objvect(objects_X,i,j)
        return np.sqrt(X[0]**2+X[1]**2+X[2]**2)

    def gravconst(self):
        ''' the constant is in au^3/d^2/M_sol '''
        return 2.95912208286*10**(-4)

    def gravconst_SI(self):
        ''' the constant is in m^3/s^2/kg'''
        return 6.67408*10**(-11)

    def solar_mass(self):
        ''' the constant is in kg '''
        return 1.9884*10**(30)

    def astronomical_unit(self):
        ''' the constant is in meters '''
        return 1.49597870*10**(11)

    def acceleration(self,objects_X):
        ''' Returns an array containing the acceleration of the objects '''
        a=np.zeros_like(objects_X)
        n=self.n_objects
        for j in range(n):
            for k in range(n):
                if j!=k:
                    v=self.objvect(objects_X,j,k)
                    d=self.objdist(objects_X,j,k)
                    a[:][j]=a[:][j]+self.gravconst()*(self.objects_mass[k]*v)/(d**3)
        return a

    def compute(self,dt,method='Euler_explicit',focus_back=False):
        ''' Defines which computing method will be used according to the user's request '''
        if method=='Euler_explicit':
            self.compute_euler_explicit(dt)
        if method=='Euler_semi_implicit':
            self.compute_euler_semi_implicit(dt)
        if method=='Euler_symplectic':
            self.compute_euler_symplectic(dt)
        if method=='Heun':
            self.compute_Heun(dt)
        if method=='Runge_Kutta':
            self.compute_Runge_Kutta(dt)
        if focus_back==True:
            # Reference frame change : focus back on central body
            for i in range(self.n_objects):
                self.objects_X[i]=self.objects_X[i]-self.objects_X[0]

    def compute_euler_explicit(self,dt):
```

```
        ''' Calculates the system's next time step state using the explicit Euler method '''
        new_V=self.objects_V+dt*self.acceleration(self.objects_X)
        new_X=self.objects_X+dt*self.objects_V
        self.objects_X=new_X
        self.objects_V=new_V

    def compute_euler_semi_implicit(self,dt):
        ''' Calculates the system's next time step state using the semi-implicit Euler method '''
        new_V=self.objects_V+dt*self.acceleration(self.objects_X)
        new_X=self.objects_X+dt*new_V
        self.objects_X=new_X
        self.objects_V=new_V

    def compute_euler_symplectic(self,dt):
        ''' Calculates the system's next time step state using the symplectic Euler method '''
        new_X=self.objects_X+dt*self.objects_V
        new_V=self.objects_V+dt*self.acceleration(new_X)
        self.objects_X=new_X
        self.objects_V=new_V

    def compute_Heun(self,dt):
        ''' Calculates the system's next time step state using the Heun method '''
        k1_X=self.objects_V*dt
        k1_V=self.acceleration(self.objects_X)*dt
        k2_X=(self.objects_V+k1_V)*dt
        k2_V=self.acceleration(self.objects_X+k1_X)*dt
        new_X=self.objects_X+(k1_X+k2_X)/2
        new_V=self.objects_V+(k1_V+k2_V)/2
        self.objects_X=new_X
        self.objects_V=new_V

    def compute_Runge_Kutta(self,dt):
        ''' Calculates the system's next time step state using the Runge-Kutta method '''
        k1_X=self.objects_V*dt
        k1_V=self.acceleration(self.objects_X)*dt
        k2_X=(self.objects_V+k1_V/2)*dt
        k2_V=self.acceleration(self.objects_X+k1_X/2)*dt
        k3_X=(self.objects_V+k2_V/2)*dt
        k3_V=self.acceleration(self.objects_X+k2_X/2)*dt
        k4_X=(self.objects_V+k3_V)*dt
        k4_V=self.acceleration(self.objects_X+k3_X)*dt
        new_X=self.objects_X+(k1_X+2*k2_X+2*k3_X+k4_X)/6
        new_V=self.objects_V+(k1_V+2*k2_V+2*k3_V+k4_V)/6
        self.objects_X=new_X
        self.objects_V=new_V

################################################################################
#--------- PLANETARY SYSTEM CLASS ---------#

class Planetary_System():
    ''' Main class directing the entire program.'''
    def __init__(self,ephemeride_file):
        self.current_dir=os.path.dirname(os.path.abspath(__file__))
```

```
        self.ephemeride_file=ephemeride_file
        self.engine=NBody_Engine()
        self.database=Ephemeride_Database(self.ephemeride_file)
        objects=self.database.load_data()
        self.engine.define_objects(objects)
        self.time=0
        self.is_new=True
        self.saves_file=None
        self.n_saves=0

    def new_session(self,ephemeride_file):
        ''' Initializes a new session, in order to compute new values, starting from scratch '''
        self.engine=NBody_Engine()
        self.ephemeride_file=ephemeride_file
        self.database=Ephemeride_Database(self.ephemeride_file)
        objects=self.database.load_data()
        self.engine.define_objects(objects)
        self.time=0
        self.is_new=True
        self.saves_file=None
        self.n_saves=0

    def load_session(self,save_file_name):
        ''' Loads a previous session, in order to use older, already computed data '''
        new_path=self.current_dir+"\\logs\\"+save_file_name
        assert os.path.exists(new_path)==True,"ERROR : File does not exists, try a different name."
        self.saves_file=save_file_name
        self.is_new=False
        [x,m]=self.load_save_info()
        if self.ephemeride_file!=x:
            print("WARNING : ephemeride file initialized does not match with the one in save file.")
            print("Replacing "+self.ephemeride_file+'data by '+x+' data .....')
            self.ephemeride_file=x
            self.database=Ephemeride_Database(self.ephemeride_file)
            objects=self.database.load_data()
            self.engine.define_objects(objects)
        with open(new_path,"r") as file:
            file.readline()
            data_needed=[0,1,2,3,4,5,6]
            for i in range(m):
                data=self.load_state(file,data_needed)
            X=np.zeros_like(self.engine.objects_X)
            V=np.zeros_like(self.engine.objects_V)
            X[:,0]=data[1]
            X[:,1]=data[2]
            X[:,2]=data[3]
            V[:,0]=data[4]
            V[:,1]=data[5]
            V[:,2]=data[6]
            self.engine.objects_X=X
            self.engine.objects_V=V
            self.time=data[0]
        self.n_saves=m

```

```
    def save_state(self):
        ''' Saves the computed data in the file containing data from older time steps '''
        self.n_saves=self.n_saves+1
        parameters=np.zeros(shape=(self.engine.n_objects,6))
        for i in range(1,self.engine.n_objects):
            parameters[i]=orbital_parameters(self.engine.objects_X[i],self.engine.objects_V[i],
                                                                    self.engine.gravconst(),self.engine.objects_mass[0])
        new_path=self.current_dir+"\\logs\\"+self.saves_file
        X,Y,Z,Xp,Yp,Zp,a,e,i,Omega,w,theta=[],[],[],[],[],[],[],[],[],[],[],[]
        for j in range(self.engine.n_objects):
            X.append(str(self.engine.objects_X[j,0]))
            Y.append(str(self.engine.objects_X[j,1]))
            Z.append(str(self.engine.objects_X[j,2]))
            Xp.append(str(self.engine.objects_V[j,0]))
            Yp.append(str(self.engine.objects_V[j,1]))
            Zp.append(str(self.engine.objects_V[j,2]))
            a.append(str(parameters[j,0]))
            e.append(str(parameters[j,1]))
            i.append(str(parameters[j,2]))
            Omega.append(str(parameters[j,3]))
            w.append(str(parameters[j,4]))
            theta.append(str(parameters[j,5]))
        with open(new_path,'a') as file:
            file.write(str(self.time)+'\n')
            file.write(' '.join(X)+'\n')
            file.write(' '.join(Y)+'\n')
            file.write(' '.join(Z)+'\n')
            file.write(' '.join(Xp)+'\n')
            file.write(' '.join(Yp)+'\n')
            file.write(' '.join(Zp)+'\n')
            file.write(' '.join(a)+'\n')
            file.write(' '.join(e)+'\n')
            file.write(' '.join(i)+'\n')
            file.write(' '.join(Omega)+'\n')
            file.write(' '.join(w)+'\n')
            file.write(' '.join(theta)+'\n')
            file.close()

    def load_state(self,file,data_needed):
        ''' Loads the computed data for a given time step from the open session file '''
        assert self.is_new==False,"ERROR : The save file has yet to be rendered, try doing calculations first."
        data=[]
        holder=file.readline()
        if 0 in data_needed:
            data.append(float(holder))
        for i in range(1,13):
            holder=file.readline()
            if i in data_needed:
                data.append(str_to_float_list(holder))
        return data

    def load_save_info(self):
        ''' Returns information on the specifics of which file has been loaded
```

```
'''
        assert self.is_new==False,"ERROR : The save file has yet to be rendered, try doing calculations first."
        new_path=self.current_dir+"\\logs\\"+self.saves_file
        with open(new_path,'r') as file:
            initial_line=file.readline()
            file.close()
        initial_line=initial_line.split()
        initial_line.pop(0)
        initial_line[1]=int(initial_line[1])
        return initial_line

    def RUN(self,dt,T,skip,method='Euler_explicit'):
        ''' Runs the calculations, using the ephemeride data initialized, and saves them at each time step '''
        if self.is_new==True:
            logs_path=self.current_dir+"\\logs"
            if os.path.exists(logs_path)==False:
                os.mkdir(logs_path)
            self.saves_file=session_name()
            new_path=self.current_dir+"\\logs\\"+self.saves_file
            with open(new_path,'w') as file:
                file.write('Base_File '+self.ephemeride_file+' '+str(self.n_saves)+'\n')
                file.close()
        self.save_state()
        initial_time=self.time
        last_snap_time=self.time
        print("Beginning Calculations")
        while self.time<initial_time+T:
            self.engine.compute(dt,method=method,focus_back=True)
            self.time=self.time+dt
            if self.time-last_snap_time>=skip:
                self.save_state()
                last_snap_time=self.time
        print("Calculations Finished")
        # Updating the number of snapshots contained inside the file
        file=open(self.current_dir+"\\logs\\"+self.saves_file,"r")
        lines=file.readlines()
        lines[0]='Base_File '+self.ephemeride_file+' '+str(self.n_saves)+'\n'
        file.close()
        file=open(self.current_dir+"\\logs\\"+self.saves_file,"w")
        file.writelines(lines)
        file.close()
        self.is_new=False

    def display_3D(self,labels=True):
        ''' Displays a 3D animation of the planetary system, with the star at the center and the planets' orbit around it using the computed data '''
        assert self.is_new==False,"ERROR : Cannot display System since no calculations have taken place."
        print("Displaying 3D System")
        # Creating the 3D figure
        fig=plt.figure(figsize=(12,12))
        ax=fig.gca(projection='3d')
        ax.set_title('t = 0.0 days')
        ax.set_xlim3d(-50,50)
        ax.set_ylim3d(-50,50)
```

```
        ax.set_zlim3d(-50,50)
        xLabel=ax.set_xlabel('\nX [ au ]',linespacing=3.2)
        yLabel=ax.set_ylabel('\nY [ au ]',linespacing=3.1)
        zLabel=ax.set_zlabel('\nZ [ au ]',linespacing=3.4)
        # Accessing the save file
        [x,m]=self.load_save_info()
        new_path=self.current_dir+"\\logs\\"+self.saves_file
        initial_needed_data=[0,1,2,3,4,5,6,7,8,9,10,11,12]
        with open(new_path,'r') as file:
            file.readline() #Skipping first line
            data=self.load_state(file,initial_needed_data)
            graph=ax.scatter(data[1],data[2],data[3],c='y',edgecolor="k")
            orbits=[]
            for i in range(1,self.engine.n_objects):
                X,Y,Z=find_trajectory(data[7][i],data[8][i],data[9][i],
                                      data[10][i],data[11][i],data[12][i],120)
                orbits.append(ax.plot(X,Y,Z))
            if labels==True:
                Labels=[]
                for i in range(0,self.engine.n_objects):
                    Labels.append(ax.text(data[1][i],data[2][i],data[3][i],self.engine.objects_name[i], (1,1,1)))
            fig.show()
            plt.pause(3)
            for j in range(1,m):
                plt.pause(0.04)
                data=self.load_state(file,initial_needed_data)
                graph._offsets3d=(data[1],data[2],data[3])
                for k in range(1,self.engine.n_objects):
                    X,Y,Z=find_trajectory(data[7][k],data[8][k],data[9][k],
                                          data[10][k],data[11][k],data[12][k],120)
                    line=orbits[k-1][0]
                    line.set_data(X,Y)
                    line.set_3d_properties(Z)
                    orbits[k-1]=[line]
                ax.set_title('t = '+str(round(data[0],))+' days')
                if labels==True:
                    for i in range(0,self.engine.n_objects):
                        Labels[i].set_position((data[1][i],data[2][i]))
                        Labels[i].set_3d_properties(data[3][i],(1,1,1))
                plt.draw()
            file.close()

    def apsidal_precession(self,displayed='all'):
        ''' Calculates and displays the apsidal precession of the bodies requested by the user over time '''
        assert self.is_new==False,"ERROR : The save file has yet to be rendered, try doing calculations first."
        if displayed=='all':
            planets=[]
            for i in range(1,self.engine.n_objects):
                planets.append(i)
        else:
            if type(displayed)==type('n'):
                assert displayed in self.engine.objects_name,displayed+" is not in the Ephemeride file objects list."
                planets=[self.engine.objects_name.index(displayed)]
            if type(displayed)==type(['n']):
```

```
                planets=[]
                for name in displayed:
                    assert name in self.engine.objects_name,name+" is not in the Ephemeride file objects list."
                    planets.append(self.engine.objects_name.index(name))
        if len(planets)>1:
            print(" Displaying apsidal precessions")
        else:
            print(" Displaying apsidal precession")
        # Accessing the save file
        [x,m]=self.load_save_info()
        new_path=self.current_dir+"\\logs\\"+self.saves_file
        initial_needed_data=[0,11]
        n=self.engine.n_objects
        Time=np.zeros(shape=(m,1))
        w0=np.zeros(shape=(1,n))
        w=np.zeros(shape=(m,n))
        with open(new_path,'r') as file:
            file.readline() #Skipping first line
            data=self.load_state(file,initial_needed_data)
            w0=data[1]
            precession=np.zeros(shape=(m,n))
            Time[0]=data[0]
            for i in range(1,m):
                data=self.load_state(file,initial_needed_data)
                w[i]=data[1]
                precession[i]=w[i]-w0
                Time[i]=data[0]
                precession[i]=precession[i]*180/np.pi #from rad to deg
        file.close()
        for j in planets:
            plt.plot(Time,precession[:,j], label=self.engine.objects_name[j])
        plt.title('Apsidal precession over time')
        plt.xlabel('Time (Days)')
        plt.ylabel('Orbital Shift (  )')
        plt.legend()
        plt.show()


    def display_perihelion(self,displayed='all'):
        ''' Calculates and displays the perihelion of the bodies requested by the user over time '''
        assert self.is_new==False,"ERROR : The save file has yet to be rendered, try doing calculations first."
        if displayed=='all':
            planets=[]
            for i in range(1,self.engine.n_objects):
                planets.append(i)
        else:
            if type(displayed)==type('n'):
                assert displayed in self.engine.objects_name,displayed+" is not in the Ephemeride file objects list."
                planets=[self.engine.objects_name.index(displayed)]
            if type(displayed)==type(['n']):
                planets=[]
                for name in displayed:
                    assert name in self.engine.objects_name,name+" is not in the Ephemeride file objects list."
```

```python
                    planets.append(self.engine.objects_name.index(name))
        if len(planets)>1:
            print(" Displaying perihelions")
            plt.title('Perihelions values')
        else:
            print(" Displaying perihelion")
            plt.title('Perihelion values')
        print(planets)
        # Accessing the save file
        [x,m]=self.load_save_info()
        new_path=self.current_dir+"\\logs\\"+self.saves_file
        initial_needed_data=[0,7,8]
        n=self.engine.n_objects
        Times=np.zeros(shape=(m,))
        A=np.zeros(shape=(m,n))
        E=np.zeros(shape=(m,n))
        with open(new_path,'r') as file:
            file.readline() #Skipping first line
            for i in range(m):
                data=self.load_state(file,initial_needed_data)
                Times[i]=data[0]
                A[i]=data[1]
                E[i]=data[2]
            file.close()
        R=A*(1-E)
        for j in planets:
            plt.plot(Times,R[:,j],label=self.engine.objects_name[j])
        plt.legend()
        plt.xlabel("Time (Days)")
        plt.ylabel("Periapsis (AUs)")
        plt.show()

    def energy_conservation(self):
        '''Calculates the mechanical energy of the entire system at each time and plots it over time.'''
        assert self.is_new==False,"ERROR : The save file has yet to be rendered, try doing calculations first."
        [x,m]=self.load_save_info()
        new_path=self.current_dir+"\\logs\\"+self.saves_file
        initial_needed_data=[0,7]
        n=self.engine.n_objects
        Times=np.zeros(shape=(m,))
        E=np.zeros(shape=(m,n))
        with open(new_path,'r') as file:
            file.readline() #Skipping first line
            for i in range(m):
                data=self.load_state(file,initial_needed_data)
                Times[i]=data[0]
                energy=0
                for j in range(1,n):
                    mj=self.engine.objects_mass[j]
                    M=self.engine.objects_mass[0]
                    G=self.engine.gravconst()
                    energy=energy-mj*M*G/(2*data[1][j])
                E[i]=energy
            file.close()
        plt.plot(Times,E)
        plt.xlabel("Time (Days)")
```

```python
        plt.ylabel("Energy")
        plt.title("Total System Energy")
        plt.show()



###############################################################################
#--------- EPHEMERIDE CLASS ---------#

class Ephemeride_Database():
    ''' Initializes and handles the ephemeride data on the planetary system's initial conditions '''
    def __init__(self,ephemeride_file):
        self.current_dir=os.path.dirname(os.path.abspath(__file__))
        self.ephemeride_file=ephemeride_file
        self.path=self.current_dir+"\\ephem\\"
        self.filename=self.current_dir+"\\ephem\\"+ephemeride_file
        assert os.path.exists(self.path)==True,"The ephemerides folder is not present, please create it using the name 'ephem'."
        assert os.path.exists(self.filename)==True,self.ephemeride_file+" is not in the ephemerides folder."
        with open(self.filename,'r') as file:
            objects=file.readlines()
        self.reference_time=objects[0]
        self.labels=objects[1]
        objects.pop(0)
        objects.pop(0)
        n=len(objects)
        self.catalogue=objects

    def add_object(self):
        ''' Adds an object and its initial parameters in the file containing such data on other objects '''
        name=str(input("Object's name :"))
        mass=float(input("Object's mass :"))
        x=float(input("Object's x :"))
        y=float(input("Object's y :"))
        z=float(input("Object's z :"))
        xp=float(input("Object's xp :"))
        yp=float(input("Object's yp :"))
        zp=float(input("Object's zp :"))
        string='\n'+name+','+str(mass)+','+str(x)+','+str(y)+','+str(z)+','
        string=string+str(xp)+','+str(yp)+','+str(zp)
        with open(self.filename,'a') as file:
            file.write(string)
            file.close()

    def load_data(self):
        ''' Loads the ephemeride data from the given file '''
        objects=[]
        n=len(self.catalogue)
        for i in range(n):
            object_i=self.catalogue[i].split(',')
            m=len(object_i)
            name=object_i[0]
            mass=float(object_i[1])
            x=float(object_i[2])
            y=float(object_i[3])
```

```
            z=float(object_i[4])
            xp=float(object_i[5])
            yp=float(object_i[6])
            zp=float(object_i[7])
            object_i=[name,[x,y,z],[xp,yp,zp],mass]
            objects.append(object_i)
        return objects


#################################################################################
#--------- FUNCTIONS ---------#

def quadrant(cos_i,sin_i):
    ''' Returns the real angle by using the four quadrants in trigonometry '''
    if cos_i>=0 and sin_i>=0:  # Quadrant 1
        return np.arccos(cos_i)
    if cos_i<0 and sin_i>=0:  # Quadrant 2
        return np.pi-np.arccos(np.abs(cos_i))
    if cos_i>=0 and sin_i<0:  # Quadrant 4
        return 2*np.pi-np.arccos(cos_i)
    if cos_i<0 and sin_i<0:  # Quadrant 3
        return np.pi+np.arccos(np.abs(cos_i))


def orbital_parameters(R,V,G,M):
    ''' Calculates the orbital parameters used to describe a Keplerian orbit '''
    r=np.linalg.norm(R)
    v=np.linalg.norm(V)
    energy=(v**2)/2-(G*M)/r
    a=-(G*M)/(2*energy)
    E=(1/(G*M))*((v**2-(G*M)/r)*R-np.dot(R,V)*V)
    e=np.linalg.norm(E)
    H=np.cross(R,V)
    h=np.linalg.norm(H)
    K=np.array([0,0,1])
    I=np.array([1,0,0])
    J=np.array([0,1,0])
    i=np.arccos(np.dot(K,H)/h)
    N=np.cross(K,H)
    n=np.linalg.norm(N)
    cos_omega=np.dot(I,N)/n
    sin_omega=np.dot(J,N)/n
    omega=quadrant(cos_omega,sin_omega)
    if E[2]>=0:
        w=np.arccos((np.dot(N,E))/(n*e))
    else:
        w=2*np.pi-np.arccos((np.dot(N,E))/(n*e))
    if np.dot(R,V)>=0:
        theta=np.arccos((np.dot(E,R))/(e*r))
    else:
        theta=2*np.pi-np.arccos((np.dot(E,R))/(e*r))
    return [a,e,i,omega,w,theta]


def find_trajectory(a,e,i,Omega,w,theta,N):
    ''' Computes the trajectory of a body by using its Keplerian orbital
   parameters '''
    rot_Omega=np.array([[np.cos(Omega),np.sin(Omega),0],
```

```
                        [-np.sin(Omega),np.cos(Omega),0],
                        [0,0,1]])
    rot_w=np.array([[np.cos(w),np.sin(w),0],
                    [-np.sin(w),np.cos(w),0],
                    [0,0,1]])
    rot_i=np.array([[1,0,0],
                    [0,np.cos(i),np.sin(i)],
                    [0,-np.sin(i),np.cos(i)]])
    rotation_matrix=np.matmul(rot_w,np.matmul(rot_i,rot_Omega))
    if e<1: #Ellipse case
        Thetas=np.linspace(-np.pi,np.pi,num=N)
        Radiuses=(a*(1-e**2))/(1+e*np.cos(Thetas))
        Xs=[]
        Ys=[]
        Zs=[]
        n=len(Thetas)
        for i in range(n):
            X=np.array([Radiuses[i]*np.cos(Thetas[i]),Radiuses[i]*np.sin(Thetas[i]),0])
            X=np.matmul(np.linalg.inv(rotation_matrix),X)
            Xs.append(X[0])
            Ys.append(X[1])
            Zs.append(X[2])
        return [np.array(Xs),np.array(Ys),np.array(Zs)]
    if e>=1: #Parabola/hyperbola case
        limit=np.arccos(-1/e)
        Thetas=np.linspace(-limit,limit,num=N)
        Radiuses=(a*(1-e**2))/(1+e*np.cos(Thetas))
        Xs=[]
        Ys=[]
        Zs=[]
        n=len(Thetas)
        for i in range(n):
            X=np.array([Radiuses[i]*np.cos(Thetas[i]),Radiuses[i]*np.sin(Thetas[i]),0])
            X=np.matmul(np.linalg.inv(rotation_matrix),X)
            Xs.append(X[0])
            Ys.append(X[1])
            Zs.append(X[2])
        return [np.array(Xs),np.array(Ys),np.array(Zs)]

def str_to_float_list(string):
    L=string.split(' ')
    n=len(L)
    for i in range(n):
        L[i]=float(L[i])
    return L

def kepler_to_cartesian(a,e,i,Omega,w,theta,star_mass,planet_mass):
    ''' Gives the position and speed vectors in a stellarcentric reference frame.
     a needs to be in AUs the main angles in radians, star_mass in units
    of solar mass and planet_mass in units of Jupiter's mass. '''
    jupiter_mass=1.898*10**27 #kg
    sun_mass=1.989*10**30 #kg
    G=2.95912208286*10**(-4) #ua^3/d^2/M_sol
    p=a*(1-e**2)
    r=p/(1+e*np.cos(theta))
    planet_mass=planet_mass*jupiter_mass/sun_mass
```

```python
    mu=G*star_mass
    h=np.sqrt(p*mu)
    R=np.array([r*np.cos(theta),r*np.sin(theta),0])
    V=np.array([-mu*np.sin(theta)/h,mu*(e+np.cos(theta))/h,0])
    # Rotation Matrix
    rot_Omega=np.array([[np.cos(Omega),np.sin(Omega),0],
                        [-np.sin(Omega),np.cos(Omega),0],
                        [0,0,1]])
    rot_w=np.array([[np.cos(w),np.sin(w),0],
                    [-np.sin(w),np.cos(w),0],
                    [0,0,1]])
    rot_i=np.array([[1,0,0],
                    [0,np.cos(i),np.sin(i)],
                    [0,-np.sin(i),np.cos(i)]])
    rotation_matrix=np.matmul(rot_w,np.matmul(rot_i,rot_Omega))
    # Transformation of Position and Speed
    R=np.matmul(np.linalg.inv(rotation_matrix),R)
    V=np.matmul(np.linalg.inv(rotation_matrix),V)
    print("Object's mass (in M_sol) : ",planet_mass)
    print("Object's X,Y,Z (in M_sol) : "+str(R[0])+','+str(R[1])+','+str(R[2]))
    print("Object's Xp,Yp,Zp (in M_sol) : "+str(V[0])+','+str(V[1])+','+str(V[2])
    )

def session_name():
    ''' Names the files with the given date and time format: yyyy-mm-dd-hours-
    mins-secs '''
    t0=time.time()
    struct=time.localtime(t0)
    string=str(struct.tm_year)+'-'
    # MONTHS
    n_months=str(struct.tm_mon)
    if len(n_months)==1:
        n_months='0'+n_months
    string=string+n_months+'-'
    # DAYS
    n_days=str(struct.tm_mday)
    if len(n_months)==1:
        n_days='0'+n_days
    string=string+n_days+'-'
    # HOURS
    n_hours=str(struct.tm_hour)
    if len(n_hours)==1:
        n_hours='0'+n_hours
    string=string+n_hours+'-'
    # MINUTES
    n_mins=str(struct.tm_min)
    if len(n_mins)==1:
        n_mins='0'+n_mins
    string=string+n_mins+'-'
    # SECONDS
    n_secs=str(struct.tm_sec)
    if len(n_secs)==1:
        n_secs='0'+n_secs
    string=string+n_secs+'.txt'
    return string
```

3D Orbits Images

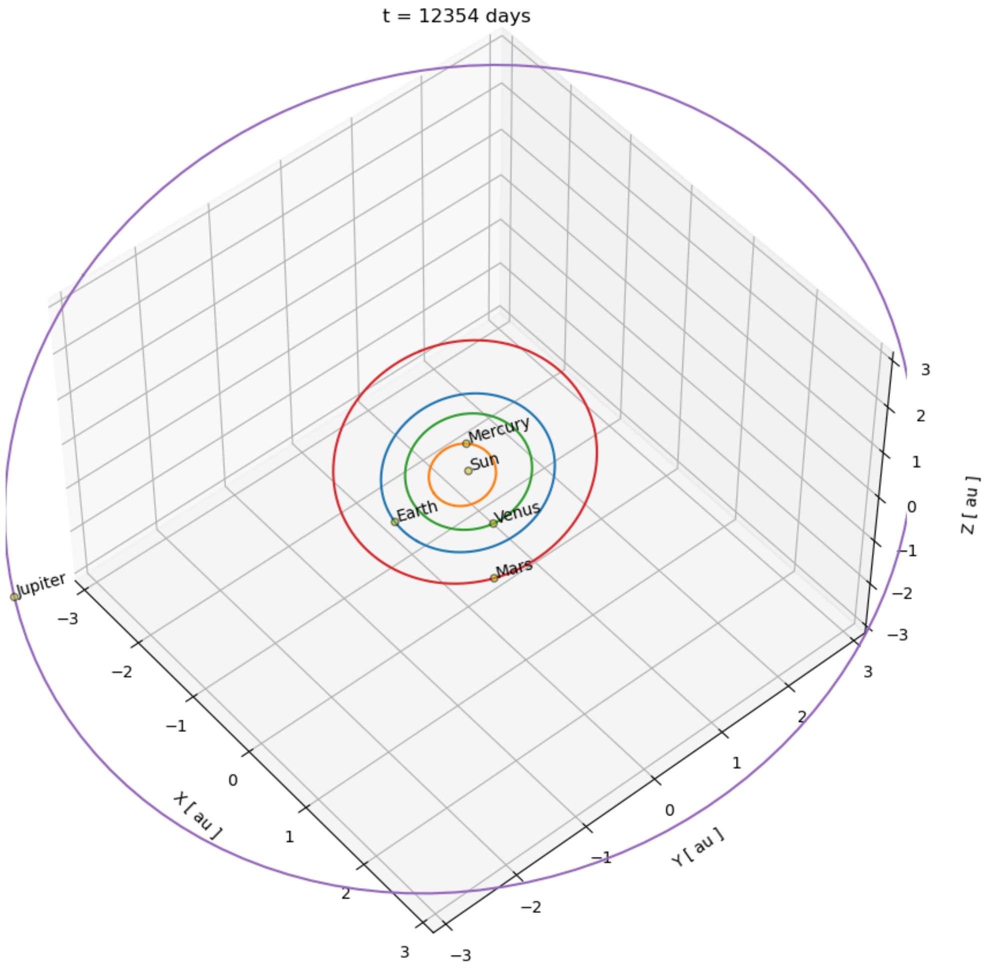


Figure A.1: Figure 4.3(a)

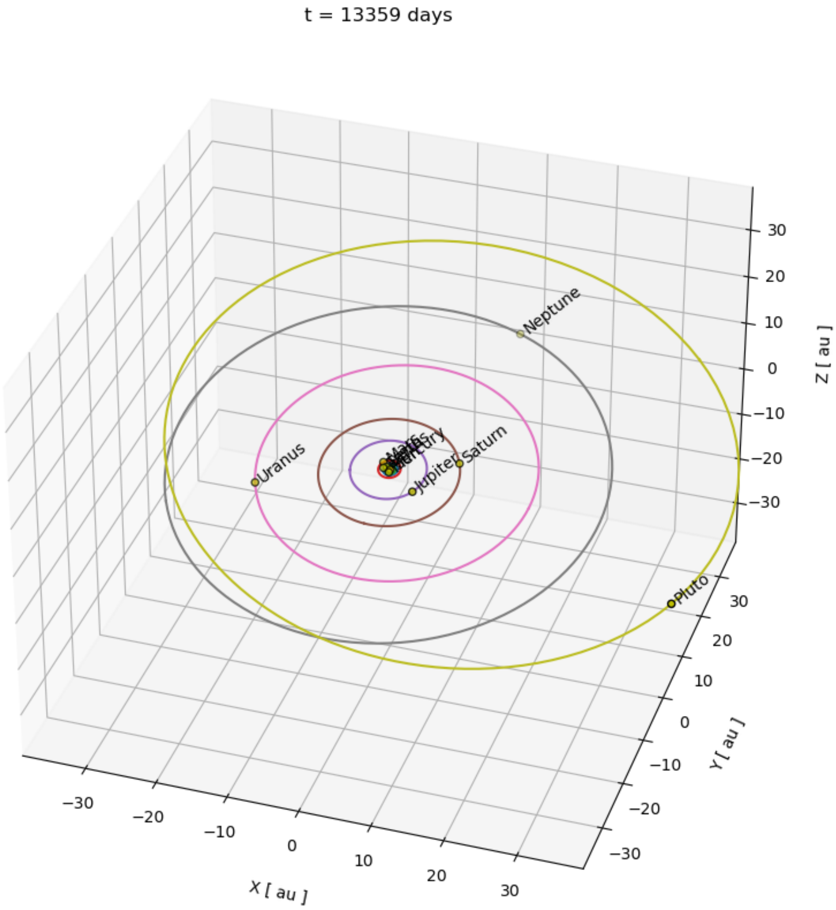


Figure A.2: Figure 4.3(b)

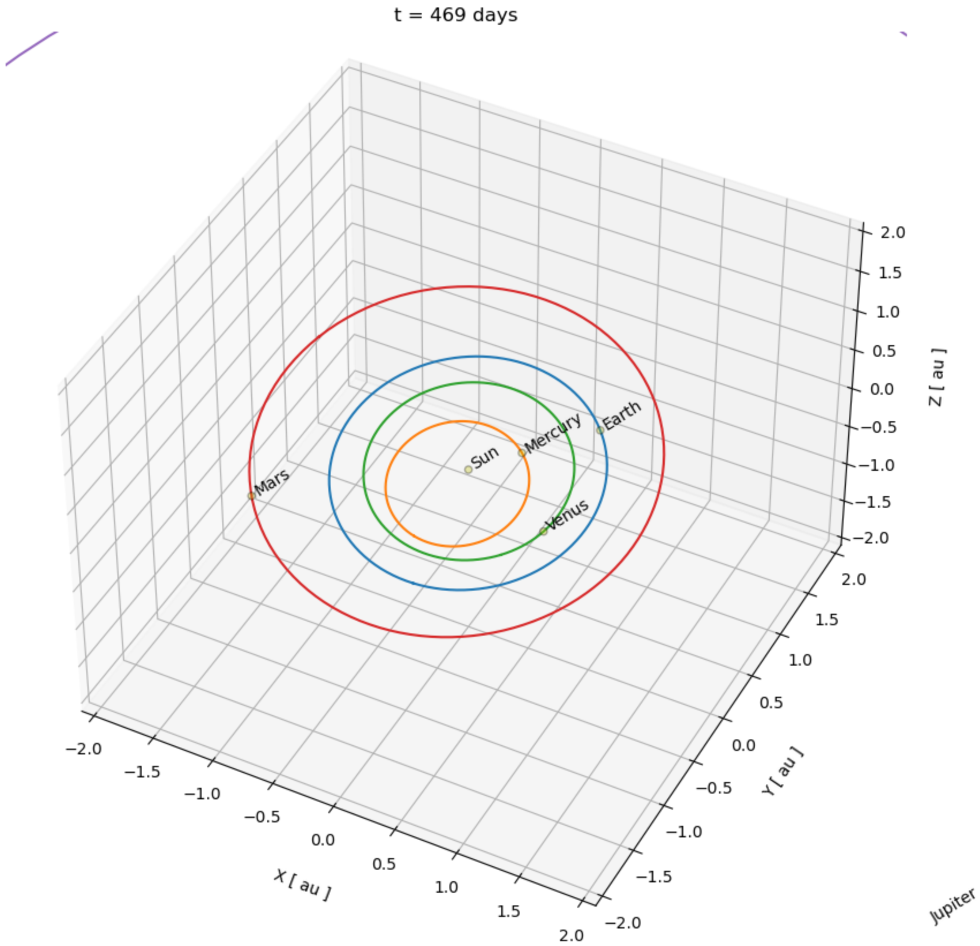


Figure A.3: Figure 4.4(a)

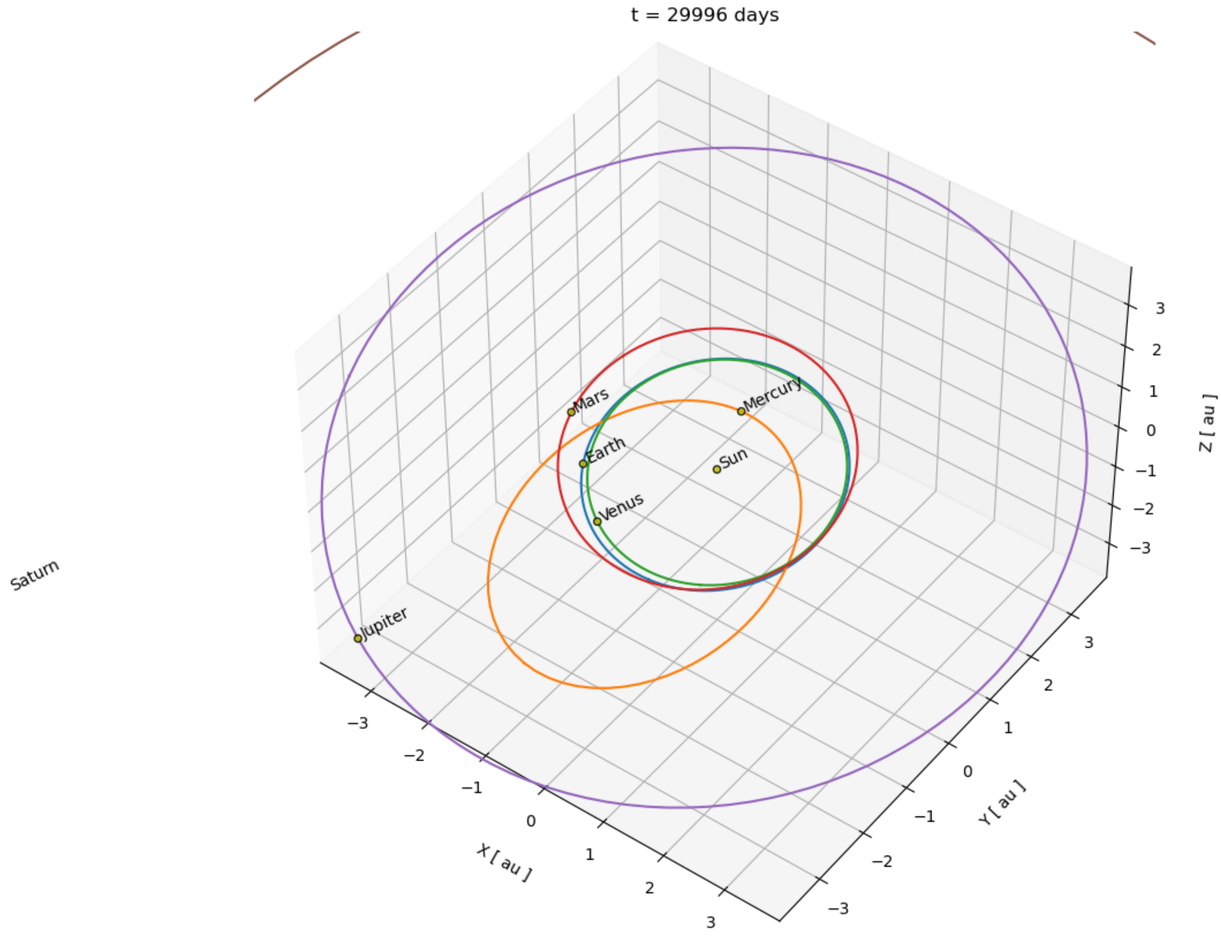


Figure A.4: Figure 4.4(b)

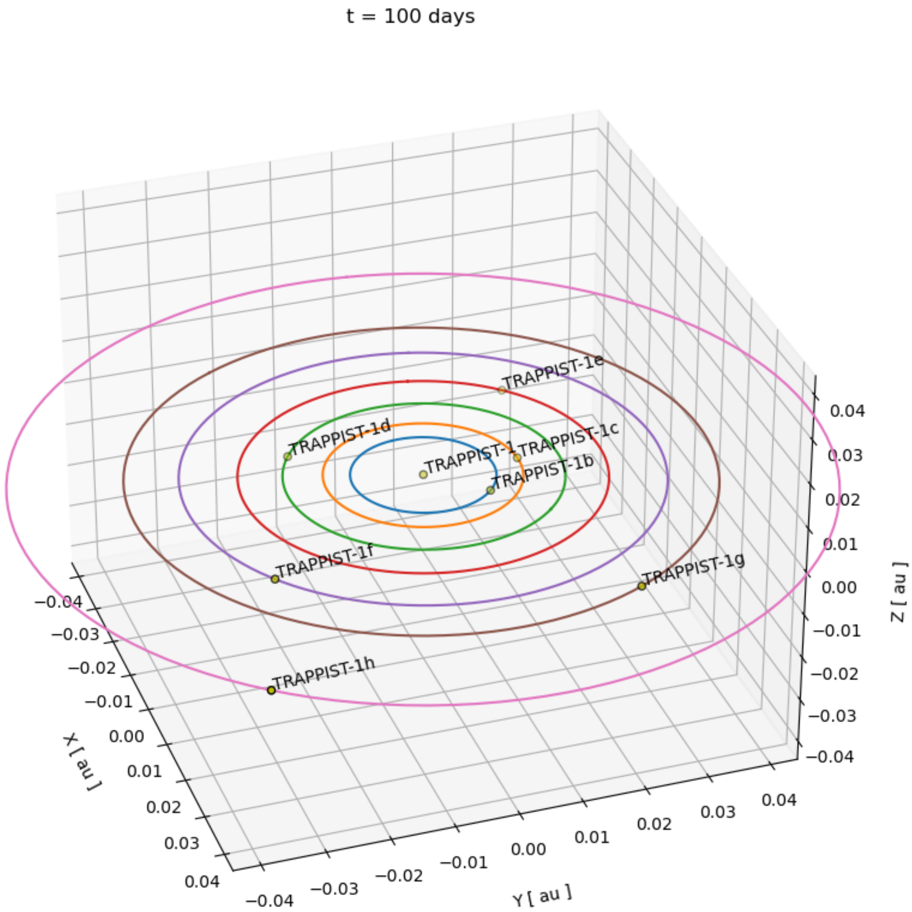


Figure A.5: Figure 4.6(a)

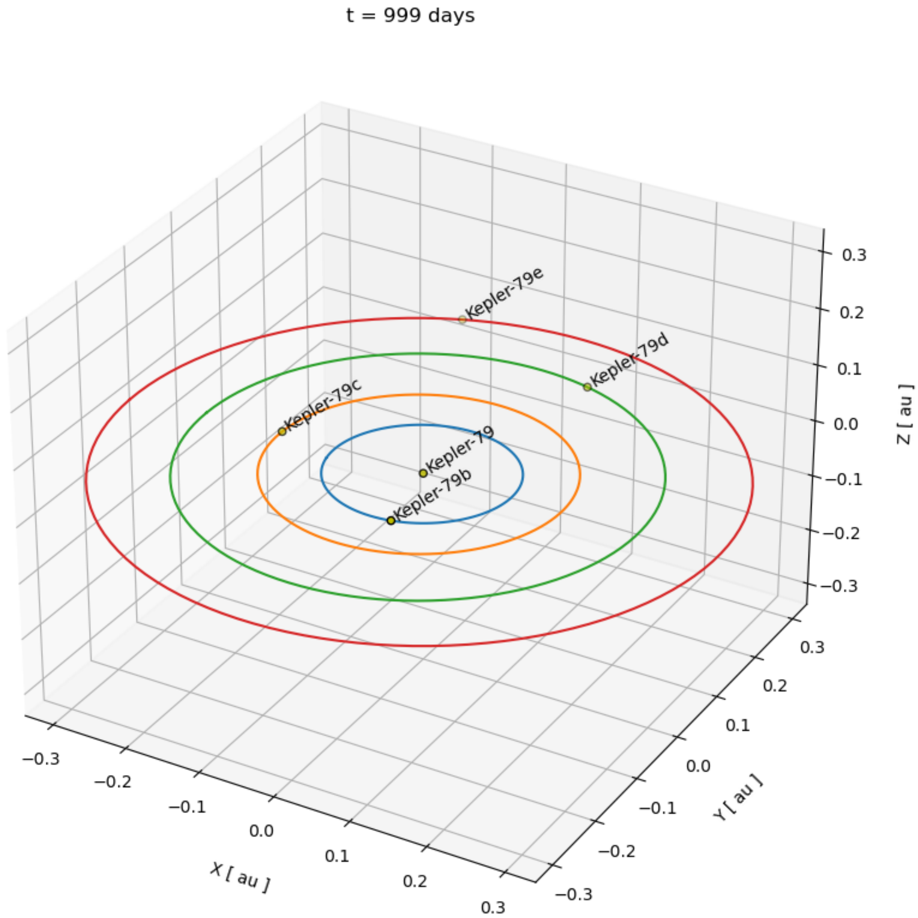


Figure A.6: Figure 4.6(b)