DOCUMENTATION OF COSMOSCOPE

POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
College of Computer and Information Sciences
Bachelor of Science in Computer Science
(BSCS 2-2)

Submitted by:

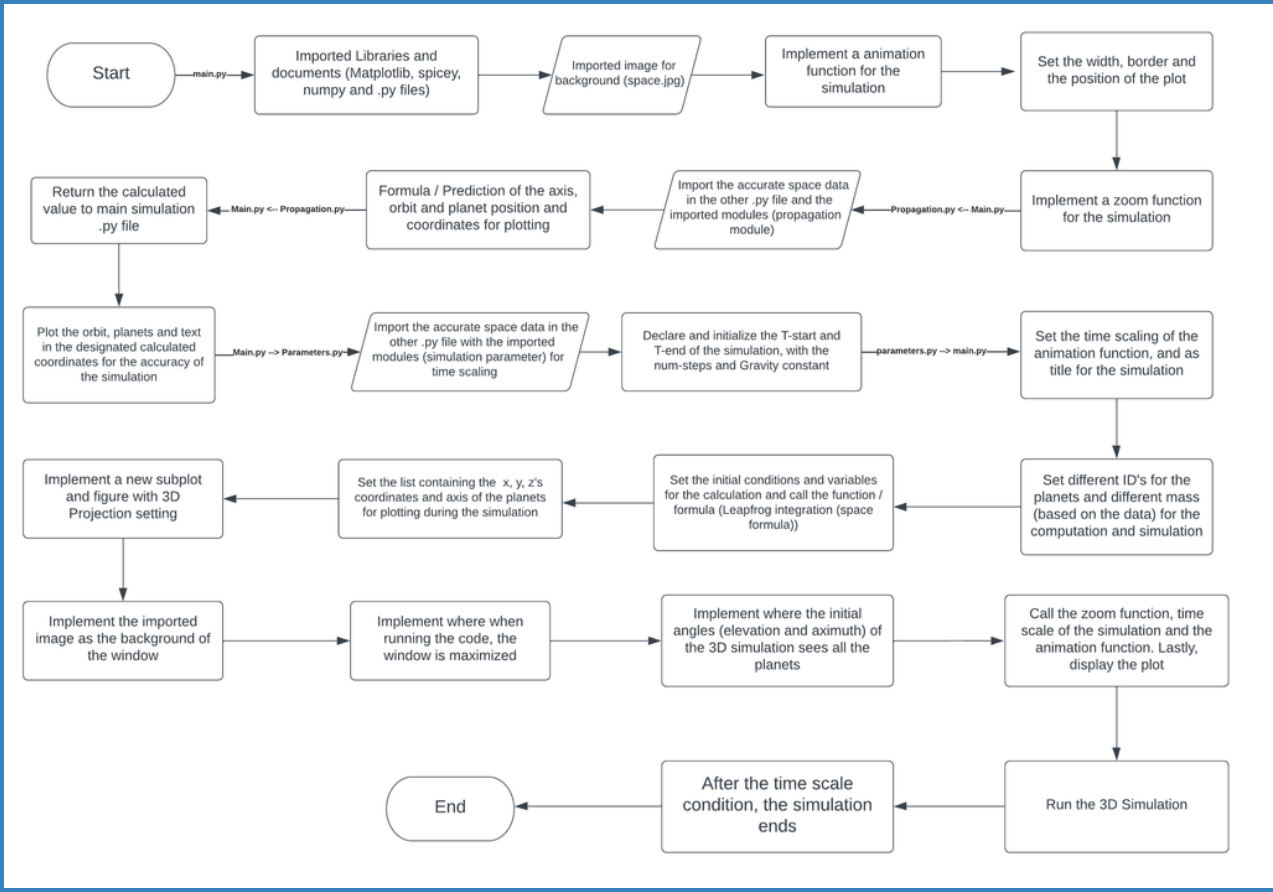
Biabado, Devonere S.
Magbitang, Elisha Jhoyce M.
Masadre, Mark Francis C.
Notario, Shoshannah Mae D.
Pardilla, Dionmelle J.
Quiray, Deseree O.

February 2023

TABLE OF CONTENTS

- Flowchart
- Source Code
- Project Manual

FLOWCHART OF COSMOSCOPE



Solar System 3D Simulation App in Python

This app is a 3D interactive tool designed to simulate the movements of celestial bodies in the Solar System. It is an educational tool that allows users to explore the Solar System and its components, including the Sun, planets, moons, and other celestial bodies. The app also features a detailed 3D representation of the Solar System with accurate orbits and rotations of each planet. The program can be used to study the motion of celestial bodies and to create realistic simulations of the Solar System. With the help of this app, students can gain a better understanding of the Solar System and its components. The app is composed of three Python files: SolarSystemSimulation.py, PropagationModule.py, and SimulationParameters.py.

SolarSystemSimulation.py

- **Header file**

```
from matplotlib import animation
```

imports the **animation** module from the **matplotlib** library. The animation module is used to create animations in matplotlib.

```
import matplotlib.image as mpimg
```

imports a module called **matplotlib.image** which allows users to import and view image files in a Python program. The module is aliased as **"mpimg"** so that it can be referred to in the program with a shorter and easier to remember name.

```
from SimulationParameters import *
```

imports the content of the file **SimulationParameters**. This file could contain a variety of parameters used in a simulation, such as initial conditions, boundary conditions, and other variables. By importing the file, the code can use any of the parameters stored in the file.

```
from PropagationModule import *
```

imports the **PropagationModule.** This module contains functions for calculating the ephemeris (position and velocity) of a body, calculating the gravitational forces of a system of bodies, calculating the kinetic and potential energy of a system of bodies, integrating the motion of the bodies with the leapfrog or Stormer-Verlet integration method, and calculating the coordinates of a body over time.

```
import numpy as np
```

imports **numpy module** (a scientific computing library built on top of the Python programming language) as **np**.

```
import ast
```

imports the **ast (Abstract Syntax Tree)** module. The ast module helps with the analysis and manipulation of Python source code (syntax trees).

```
from mpl_toolkits.mplot3d import Axes3D
```

imports the **Axes3D** module from the **mpl_toolkits** library. Axes3D is a toolkit that provides functions for creating 3D plots with Matplotlib. It provides functions for creating 3D axes, plotting 3D data and customizing 3D plots.

- **Source Code**

Reads an image called space_bg2.png and stores it as an image object in the variable img. The mpimg module is used to read the image.

```
# Load the image
img = mpimg.imread('space_bg2.png')
```

Animates a plot for real time results. and sets the face color of an imaginary square to black, clears the plot, sets the axis off, sets the position of the plot, and creates a scatter plot and text for the Sun.

```
def animate_func(i):
    # fig.canvas.mpl_connect('scroll_event', handle_scroll)  # connects 'scroll_event' and handle_scroll function
    # -------------------------------------------------------------------------------
    # PROCEDURE
    # Animating the plot for real time results.
    # -------------------------------------------------------------------------------

    ax.set_facecolor('none')            # Sets the color of imaginary square to black

    ax.clear()
    ax.set_axis_off()

    width = 1000
    border = 1500
    ax.set_position([border / width, 0, (width - border * 2) / width, 1])

    ax.scatter(0, 0, 0, s=150, c="yellow")
    ax.text(0, 0, 0, "Sun", color="white")
```

Plotting the orbits of the planets in our Solar System. It is looping through the time values stored in the timescale array, and if the time is less than t_end, the code creates a 3D plot of the planets' positions at that time and labels them. For each planet, it plots the positions of the previous times up to the current one (i.e. creating a line that connects their orbits) and adds a marker for the current position. It also adds a text label for each planet.

```
if timescale[i] <= t_end:
    fig.suptitle('TIME ' + spice.spiceypy.et2utc(timescale[i], "C", 3), fontsize=16)
    # time_str = 'TIME ' + spice.spiceypy.et2utc(timescale[i], "C", 0)
    # ax.text(2, 2, 2, time_str, transform=ax.transAxes, ha='right', va='top', fontsize=10)

    ax.plot(Lx_mercury[:i + 1], Ly_mercury[:i + 1], Lz_mercury[:i + 1], c="#ffffff", linewidth=1.2)
    ax.scatter(Lx_mercury[i], Ly_mercury[i], Lz_mercury[i], s=1.38, c='#808080', marker='o')
    ax.text(5 + Lx_mercury[i], 5 + Ly_mercury[i], 5 + Lz_mercury[i], "Mercury", color="white", size=5)

    ax.plot(Lx_venus[:i + 1], Ly_venus[:i + 1], Lz_venus[:i + 1], c="#E9ECAA", linewidth=1.2)
    ax.scatter(Lx_venus[i], Ly_venus[i], Lz_venus[i], s=10, c='#E9CA09', marker='o')
    ax.text(5 + Lx_venus[i], 5 + Ly_venus[i], 5 + Lz_venus[i], "Venus", color="white", size=5)

    ax.plot(Lx_earth[:i + 1], Ly_earth[:i + 1], Lz_earth[:i + 1], c="#E9ECAA", linewidth=1.2)
    ax.scatter(Lx_earth[i], Ly_earth[i], Lz_earth[i], s=10.72, c='#0000ff', marker='o')
    ax.text(5 + Lx_earth[i], 5 + Ly_earth[i], 5 + Lz_earth[i], "Earth", color="white", size=5)

    ax.plot(Lx_mars[:i + 1], Ly_mars[:i + 1], Lz_mars[:i + 1], c="#E9ECAA", linewidth=1.2)
    ax.scatter(Lx_mars[i], Ly_mars[i], Lz_mars[i], s=10.5, c='#ff0000', marker='o')
    ax.text(5 + Lx_mars[i], 5 + Ly_mars[i], 5 + Lz_mars[i], "Mars", color="white", size=5)

    ax.plot(Lx_jupiter[:i + 1], Ly_jupiter[:i + 1], Lz_jupiter[:i + 1], c="#EBC0B7", linewidth=1.2)
    ax.scatter(Lx_jupiter[i], Ly_jupiter[i], Lz_jupiter[i], s=70, c='#B3280A', marker='o')
    ax.text(5 + Lx_jupiter[i], 5 + Ly_jupiter[i], 5 + Lz_jupiter[i], "Jupiter", color="white", size=5)
    # labels["Jupiter"].set_visible(False)

    ax.plot(Lx_saturn[:i + 1], Ly_saturn[:i + 1], Lz_saturn[:i + 1], c="#BDB9AD", linewidth=1.2)
    ax.scatter(Lx_saturn[i], Ly_saturn[i], Lz_saturn[i], s=55.4, c='#7C5029', marker='o')
    ax.text(5 + Lx_saturn[i], 5 + Ly_saturn[i], 5 + Lz_saturn[i], "Saturn", color="white", size=5)
```

```
    ax.plot(Lx_uranus[:i + 1], Ly_uranus[:i + 1], Lz_uranus[:i + 1], c="#A2D2DC", linewidth=1.2)
    ax.scatter(Lx_uranus[i], Ly_uranus[i], Lz_uranus[i], s=30, c='#088F8F', marker='o')
    ax.text(5 + Lx_uranus[i], 5 + Ly_uranus[i], 5 + Lz_uranus[i], "Uranus", color="white", size=5)

    ax.plot(Lx_neptune[:i + 1], Ly_neptune[:i + 1], Lz_neptune[:i + 1], c="#BCE9BA", linewidth=1.2)
    ax.scatter(Lx_neptune[i], Ly_neptune[i], Lz_neptune[i], s=30, c='#118A0B', marker='o')
    ax.text(5 + Lx_neptune[i], 5 + Ly_neptune[i], 5 + Lz_neptune[i], "Neptune", color="white", size=5)

    ax.plot(Lx_pluto[:i + 1], Ly_pluto[:i + 1], Lz_pluto[:i + 1], c="#FFC0CB", linewidth=1.2)
    ax.scatter(Lx_pluto[i], Ly_pluto[i], Lz_pluto[i], s=0.38, c='#A52A2A', marker='o')
    ax.text(5 + Lx_pluto[i], 5 + Ly_pluto[i], 5 + Lz_pluto[i], "Pluto", color="white", size=5)
```

Setting the 3D axis limits of x, y and z. It is finding the min and max values of the x, y and z coordinates stored in the variable reg_pos and passing these values to the set_xlim3d(), set_ylim3d() and set_zlim3d() functions.

```
    x_data = Coordinates(reg_pos, 4)[0]
    y_data = Coordinates(reg_pos, 4)[1]
    z_data = Coordinates(reg_pos, 4)[2]

    xmin = numpy.min(x_data)
    xmax = numpy.max(x_data)

    ymin = numpy.min(y_data)
    ymax = numpy.max(y_data)

    zmin = numpy.min(z_data)
    zmax = numpy.max(z_data)

    ax.set_xlim3d(xmin, xmax)
    ax.set_ylim3d(ymin, ymax)
    ax.set_zlim3d(zmin, zmax)
```

An else statement that is part of an if-else condition. The code is used to end a simulation and print a message indicating that the simulation ended successfully.

```
else:

    simulation.event_source.stop()
    print("Simulation ended successfully.")
```

Used for zooming in and out of a figure. The on_scroll function is called when the user scrolls up or down on the figure. If the scroll is up, the zoom level of the figure is increased by 10. If the scroll is down, the zoom level of the figure is decreased by 10. The ax.dist variable is used to track the zoom level and the max function is used to ensure that the zoom level does not become negative.

```
# Zoom function
def on_scroll(event):
    ax = event.inaxes
    if ax is not None:
        if event.button == 'up':
            # Zoom in
            ax.dist = max(ax.dist - 10, 0)
        elif event.button == 'down':
            # Zoom out
            ax.dist += 10
```

Creates variables for all of the planets in our solar system. It uses the Ephemeris function to get the positions and velocities of each planet relative to the sun at a given time (t_start). It also sets variables for each planet's mass (m_). The sun is given an id of 0. The other planets are given a unique id from 1-9 in order of the planets from closest to the sun to farthest from the sun.

```
sun_id = 0
pos_sun = Ephemeris(t_start, 'SUN')[0]
vel_sun = Ephemeris(t_start, 'SUN')[1]
m_sun = 1.989e30

# Mercury
mercury_id = 1
pos_mercury = Ephemeris(t_start, 'MERCURY')[0]
vel_mercury = Ephemeris(t_start, 'MERCURY')[1]
m_mercury = 0.330e24

# Mars
mars_id = 2
pos_mars = Ephemeris(t_start, 'MARS BARYCENTER')[0]
vel_mars = Ephemeris(t_start, 'MARS BARYCENTER')[1]
m_mars = 0.642e24

# Earth
earth_id = 3
pos_earth = Ephemeris(t_start, 'EARTH')[0]
vel_earth = Ephemeris(t_start, 'EARTH')[1]
m_earth = 5.97e24

# Venus
venus_id = 4
pos_venus = Ephemeris(t_start, 'VENUS')[0]
vel_venus = Ephemeris(t_start, 'VENUS')[1]
m_venus = 4.87e24

# Jupiter
jupiter_id = 5
pos_jupiter = Ephemeris(t_start, 'JUPITER BARYCENTER')[0]
vel_jupiter = Ephemeris(t_start, 'JUPITER BARYCENTER')[1]
m_jupiter = 1898e24

# Saturn
saturn_id = 6
pos_saturn = Ephemeris(t_start, 'SATURN BARYCENTER')[0]
vel_saturn = Ephemeris(t_start, 'SATURN BARYCENTER')[1]
m_saturn = 568e24

# Uranus
uranus_id = 7
pos_uranus = Ephemeris(t_start, 'URANUS BARYCENTER')[0]
vel_uranus = Ephemeris(t_start, 'URANUS BARYCENTER')[1]
m_uranus = 86.8e24

# Neptune
neptune_id = 8
pos_neptune = Ephemeris(t_start, 'NEPTUNE BARYCENTER')[0]
vel_neptune = Ephemeris(t_start, 'NEPTUNE BARYCENTER')[1]
m_neptune = 102e24

# Pluto
pluto_id = 9
pos_pluto = Ephemeris(t_start, 'PLUTO BARYCENTER')[0]
vel_pluto = Ephemeris(t_start, 'PLUTO BARYCENTER')[1]
m_pluto = 0.322e24
```

Creating a numpy array for the masses of the nine planets plus the sun. Then, it is concatenating two numpy arrays containing the initial positions and velocities of the nine planets plus the sun. This is done by combining the arrays along the 0th axis, thereby creating one larger array containing the initial positions and velocities of all of the planets.

```
# Initial conditions mass, positions and velocities with ephemeris

mass = np.array([m_sun, m_mercury, m_mars, m_earth, m_venus, m_jupiter, m_saturn, m_uranus, m_neptune, m_pluto])

pos_init = np.concatenate(
    (pos_sun, pos_mercury, pos_mars, pos_earth, pos_venus, pos_jupiter, pos_saturn, pos_uranus, pos_neptune, pos_pluto), axis=0)

vel_init = np.concatenate(
    (vel_sun, vel_mercury, vel_mars, vel_earth, vel_venus, vel_jupiter, vel_saturn, vel_uranus, vel_neptune, vel_pluto), axis=0)
```

Using the leapfrog algorithm to update the position and velocity of an object. It takes three parameters as input: the initial position (pos_init), initial velocity (vel_init), and mass of the object. The code then uses the leapfrog algorithm to calculate the updated position (reg_pos) and velocity (reg_vel) of the object.

```
reg_pos, reg_vel = leapfrog(pos_init, vel_init, mass)
```

Generate the coordinates of the planets in our solar system. The Coordinates function takes in the reg_pos parameter which is the regular position of the planets and the number of the planet. The code then creates the Lx, Ly and Lz coordinates of each planet and assigns them to their respective variables.

```
Lx_mercury = Coordinates(reg_pos, 1)[0]
Ly_mercury = Coordinates(reg_pos, 1)[1]
Lz_mercury = Coordinates(reg_pos, 1)[2]

Lx_mars = Coordinates(reg_pos, 2)[0]
Ly_mars = Coordinates(reg_pos, 2)[1]
Lz_mars = Coordinates(reg_pos, 2)[2]

Lx_earth = Coordinates(reg_pos, 3)[0]
Ly_earth = Coordinates(reg_pos, 3)[1]
Lz_earth = Coordinates(reg_pos, 3)[2]

Lx_venus = Coordinates(reg_pos, 4)[0]
Ly_venus = Coordinates(reg_pos, 4)[1]
Lz_venus = Coordinates(reg_pos, 4)[2]

Lx_jupiter = Coordinates(reg_pos, 5)[0]
Ly_jupiter = Coordinates(reg_pos, 5)[1]
Lz_jupiter = Coordinates(reg_pos, 5)[2]

Lx_saturn = Coordinates(reg_pos, 6)[0]
Ly_saturn = Coordinates(reg_pos, 6)[1]
Lz_saturn = Coordinates(reg_pos, 6)[2]

Lx_uranus = Coordinates(reg_pos, 7)[0]
Ly_uranus = Coordinates(reg_pos, 7)[1]
Lz_uranus = Coordinates(reg_pos, 7)[2]

Lx_neptune = Coordinates(reg_pos, 8)[0]
Ly_neptune = Coordinates(reg_pos, 8)[1]
Lz_neptune = Coordinates(reg_pos, 8)[2]
Lx_pluto = Coordinates(reg_pos, 9)[0]
Ly_pluto = Coordinates(reg_pos, 9)[1]
Lz_pluto = Coordinates(reg_pos, 9)[2]
```

Creating a 3D plot of a solar system simulation with a background image. The background image is set to be transparent with an alpha of 0.3, and the axis is turned off. The figure is also set to be zoomed in.

```
fig = plt.figure("Solar System simulation", dpi=150)
ax = fig.add_subplot(222, projection='3d')  # 3D plot
fig.figimage(img, alpha=0.3, resize='auto')

ax.set_facecolor('none')
ax.set_axis_off()

figManager = plt.get_current_fig_manager()
figManager.window.state('zoomed')
```

Sets various parameters related to a figure created using the matplotlib library in Python. Specifically, it sets the background of the figure to black and sets the axes and text to white.

```
plt.rcParams['axes.facecolor'] = 'black'  # axes in black
plt.rcParams['text.color'] = 'white'  # texts in white
fig.set_facecolor('black')  # black background
```

Sets the initial orientation of the simulation and connects an event handler to the figure canvas. The third line of code generates the time scale of the animation. The fourth line of code creates the animation object and assigns the animate_func function to be called every 100 milliseconds. The fifth line of code tightens the layout of the figure. The last line of code displays the animation.

```
ax.view_init(elev=-80, azim=-50)  # initial orientation of simulation
fig.canvas.mpl_connect('scroll_event', on_scroll)

timescale = np.arange(t_start, t_end + 2 * dt, dt)  # generating time scale

simulation = animation.FuncAnimation(fig, animate_func, interval=100, frames=abs(int(t_end)), blit=False)

plt.tight_layout()

plt.show()
```

PropagationModule.py

- **Header file**

```
import matplotlib.pyplot as plt
```

Imports the matplotlib library, specifically the pyplot module, and assigns it to the variable name "plt". This allows the user to use functions from the pyplot module such as plotting and creating graphs.

```
from matplotlib import *
```

imports all of the modules, functions, and classes from the Matplotlib library. This allows the user to access all of the features of the library in their code.

```
from SimulationParameters import *
```

imports the content of the file **SimulationParameters**. This file could contain a variety of parameters used in a simulation, such as initial conditions, boundary conditions, and other variables. By importing the file, the code can use any of the parameters stored in the file.

- **Source Code**

This code is defining a function called Ephemeris that takes two parameters: ref_time and body. The function uses the spiceypy library to obtain position and velocity information for the body at the given reference time. It then stores this information in two NumPy arrays, pos_matrix and vel_matrix, before returning these two arrays. The returned arrays contain the position and velocity information, respectively, for the body at the given ref_time.

```
def Ephemeris(ref_time,body):
```

```
    x = [row[0] for row in pos] # x element of position matrix
    y = [row[1] for row in pos] # y element of position matrix
    z = [row[2] for row in pos] # x element of position matrix
    acc = np.empty((0,3))
```

This code defines a function called NbodyProblem which takes two parameters, pos and mass.

```
def NbodyProblem(pos, mass):
```

In this code, the variable G is declared as a global variable, meaning that it can be accessed from anywhere in the program. The variable pos is assumed to be a matrix containing the x, y, and z coordinates of a set of points. The next three lines loop over the matrix and store the x, y, and z coordinates in separate lists. Finally, the variable acc is declared as an empty matrix of size (0, 3).

```
    global G
```

```
    pos_list = spice.spiceypy.spkezr(body,ref_time,'J2000','NONE','SUN')[0][0:3]
    pos_matrix = np.array([ [pos_list[0], pos_list[1],pos_list[2]] ])

    vel_list = spice.spiceypy.spkezr(body,ref_time,'J2000','NONE','SUN')[0][3:6]
    vel_matrix = np.array([ [vel_list[0], vel_list[1],vel_list[2]] ])

    return (pos_matrix,vel_matrix)
```

The first part of the code creates three lists, x, y and z, which contain the x, y and z elements of the position matrix respectively. It also creates an empty array called 'acc' which will be used to store the calculations. The second part of the code is a nested loop. The outer loop goes through each element of the x list. The inner loop then goes through each element in the x list, and if the element is not the same as the element from the outer loop, it calculates the acceleration of the outer loop element due to the inner loop element. This is done by calculating the distance between the two elements, and then using the gravitational constant in order to calculate the acceleration. This acceleration is then added to the 'acc' array. Once the loop is finished, the function returns the 'acc' array which contains the accelerations of the elements of the position matrix.

```
for j in range (0,len(x)):
    ax=0
    ay=0
    az=0

    for i in range (0,len(x)):
        if (i!=j):
            dx = x[i]-x[j]
            dy = y[i]-y[j]
            dz = z[i]-z[j]
            norm = np.sqrt(dx**2+dy**2+dz**2)

            ax += (G*mass[i]*1e-9*dx)/norm**3
            ay += (G*mass[i]*1e-9*dy)/norm**3
            az += (G*mass[i]*1e-9*dz)/norm**3

    a = np.array([ax,ay,az])
    acc = np.append(acc,[a],axis=0)

return acc
```

This code defines a function called Energy that takes three parameters: pos, vel, and mass. The function calculates the energy of an object using the formula 0.5*mass*(vel**2) + mass*pos and returns that value.

```
def Energy(pos, vel, mass):
```

This code is setting up variables to calculate energy. The global variable G is declared and then three lists are created, vx, vy, and vz, which are the x, y, and z elements of the velocity matrix. Then, three more lists are created, x, y, and z, which are the x, y, and z elements of the position matrix. Finally, two variables are declared, E_kinetic and E_potential, which will be used to store the kinetic and potential energy.

```
    global G

    vx = [row[0] for row in vel] # x element of velocity matrix
    vy = [row[1] for row in vel] # y element of velocity matrix
    vz = [row[2] for row in vel] # z element of velocity matrix
    x = [row2[0] for row2 in pos] # x element of position matrix
    y = [row2[1] for row2 in pos] # y element of position matrix
    z = [row2[2] for row2 in pos] # z element of position matrix

    E_kinetic = 0
    E_potential = 0
```

This code is calculating the kinetic and potential energy of a set of particles, given the coordinates and velocities of each particle, and the mass of each particle. The outer for loop, j in range (0, len(vx)), is looping through each particle, calculating the norm of the velocity of each particle, and adding the kinetic energy of each particle to the total kinetic energy. The inner for loop, i in range (0, len(x)), is looping through each particle, calculating the distance between the two particles, and adding the potential energy between the two particles to the total potential energy. Once the looping is complete, the function returns the total kinetic and potential energies.

```
for j in range (0,len(vx)):

    norm = np.sqrt(vx[j]**2+vy[j]**2+vz[j]**2)
    E_kinetic += (1/2) * mass[j] * norm**2

    for i in range (0,len(x)):
        if (i!=j):
            dx = x[i]-x[j]
            dy = y[i]-y[j]
            dz = z[i]-z[j]
            norm2 = np.sqrt(dx**2+dy**2+dz**2)

            E_potential += (G*mass[i]*mass[j]*1e-9)/norm2

return (E_kinetic,E_potential)
```

This code defines a function called leapfrog which takes three parameters: pos0 (position), vel0 (velocity), and mass. This function likely performs a calculation involving these parameters to determine the properties of an object over a period of time.

```
def leapfrog(pos0, vel0, mass):
```

This code is setting up the initial conditions for a N-body problem simulation. The global variables G, t_start, t_end, and dt are the gravitational constant, start time, end time, and time step, respectively. The pos_register and vel_register are empty dictionaries which will be used to store the positions and velocities of the bodies in the simulation. The pos and vel variables are set to the initial positions and velocities of the bodies. Finally, the acc variable is set to the result of the NbodyProblem function which calculates the acceleration of each body in the simulation.

```
global G, t_start, t_end, dt

pos_register = {}
vel_register = {}

pos = pos0
vel = vel0
acc = NbodyProblem(pos,mass)
```

This code is a numerical integration method for solving the N-body problem. It is a kick-drift-kick version of the leapfrog algorithm. In this code, the N-Body Problem is solved by starting from an initial position and velocity. The acceleration is calculated from the initial position. The velocity is then updated by adding the acceleration multiplied by the timestep and half. The position is then updated by adding the velocity multiplied by the timestep. The acceleration is then updated again using the new position. The velocity is then updated again by adding the acceleration multiplied by the timestep and half. This is done in a loop from the starting time to the end time, with a given timestep. The final positions and velocities are then saved in the position and velocity registers.

```
for t in np.arange(t_start, t_end, dt): # kick-drift-kick version

    vel = vel + acc * dt * 0.5       # kick
    pos = pos + vel * dt             # drift

    acc = NbodyProblem(pos,mass)     # update acceleration
    vel = vel + acc * dt * 0.5       # kick

    pos_register[t] = pos            # save in the positions register
    vel_register[t] = vel            # save in the velocities register

return (pos_register,vel_register)
```

This code defines a function, EnergyOverTime, which takes three parameters as input: pos_register, vel_register, and mass. This function calculates the energy of a system over time using the given parameters. The pos_register is a list of the system's positions over time, the vel_register is a list of the system's velocities over time, and the mass is the mass of the system. The function will then calculate the energy of the system at each time step, and return the list of energies.

```
def EnergyOverTime(pos_register,vel_register,mass):
```

This code creates four empty lists, _time, _KE, _PE and _TOT. These lists are likely used to store data for a simulation of a system, as evidenced by the global variables G, t_start, t_end and dt. G is likely a gravitational constant, t_start and t_end are likely time intervals, and dt is likely a step size. The _time list likely stores the time intervals, _KE stores the kinetic energy, _PE stores the potential energy, and _TOT stores the total energy of the system.

```
global G, t_start, t_end, dt

_time = []
_KE = []
_PE = []
_TOT = []
```

This code is looping through a set of values (t_start to t_end with a step size of dt) to compute the kinetic, potential, and total energy of a system. It is using the current position and velocity of the system stored in "pos_register" and "vel_register" to calculate the energy. The loop is appending the calculated time, kinetic energy, potential energy, and total energy to the lists _time, _KE, _PE, and _TOT, respectively, and then returning all four lists at the end of the loop.

```
for t in np.arange(t_start, t_end, dt):

    pos = pos_register[t]
    vel = vel_register[t]

    E_kinetic, E_potential = Energy(pos,vel,mass)
    E_total = E_kinetic - E_potential

    _time.append(t)
    _KE.append(E_kinetic)
    _PE.append(E_potential)
    _TOT.append(E_total)

return (_time, _KE, _PE, _TOT)
```

This code defines a function called Coordinates. The function takes two parameters, register and body_id. The function is used to return the coordinates of a particular body with a given register.

```
def Coordinates(register, body_id):
```

This code is creating three empty lists, Lx, Ly, and Lz. The global variables, t_start, t_end, and dt, are used by the code to determine the values that will be stored in each list. The code is used to store the values of three variables over a given time period, indicated by the values of t_start and t_end. The time step size is defined by dt.

```
    global t_start, t_end, dt

    Lx = []
    Ly = []
    Lz = []
```

This code is looping through the values stored in the array "register" between the start and end times, t_start and t_end, with a step size of dt. For each step, the values stored in the matrix located at the index corresponding to the body_id are appended to the lists Lx, Ly, and Lz. The function then returns these three lists, containing the x, y, and z coordinates for the body in question.

```
    for t in np.arange(t_start, t_end, dt):
        matrix = register[t]
        Lx.append(matrix[body_id,0])
        Ly.append(matrix[body_id,1])
        Lz.append(matrix[body_id,2])

    return (Lx,Ly,Lz)
```

SimulationParameters.py

- **Header file**

```
import numpy as np
```

imports **numpy library** which is a library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

```
import spiceypy as spice
```

imports the **SpiceyPy library** which is a Python interface to the SPICE (Spacecraft Planet Instrument C-matrix Events) orbit propagation tool. It provides access to several of the SPICE kernels, allowing users to access various spacecraft trajectory and attitude information.

- **Source Code**

This code is setting up the SPICE Toolkit to compute the position of a planetary body. The first line calls the tkvrsn function to check the version of the toolkit. The next three lines use the furnsh function to load three binary SPICE kernels into the toolkit. The de440.bsp kernel is a planetary ephemeris file that contains the positions of the planets in the solar system. The p10-a.bsp kernel contains the orbits of some interplanetary spacecraft. The naif0012.tls kernel contains constants used in the calculations.

```
spice.tkvrsn('TOOLKIT')
spice.furnsh('de440.bsp')
spice.furnsh('p10-a.bsp')
spice.furnsh('naif0012.tls')
```

This code is using the spiceypy library to convert a string representing a date into a numerical number that can be used in calculations. In this case, the string is "1974 Jan 02 00:00:00.0000", which is converted into a numerical value stored in the variable t_start.

```
t_start = spice.spiceypy.str2et('1974 Jan 02 00:00:00.0000')        # start time of the simulation
```

This code is using the spiceypy library to convert a given string into an ephemeris time (ET). The given string is '1990 Jan 02 00:00:00.0000' which is a date and time. The str2et function will convert this string into a double precision number representing the ephemeris time for that given moment. The result of this code would be a double precision number stored in the variable t_end.

```
t_end = spice.spiceypy.str2et('1990 Jan 02 00:00:00.0000')          # end time of simulation
```

dt = 3*24*3600 is a calculation that results in the number of seconds in 3 days. 3*24 is 72 hours and 72*3600 is 259200 seconds. So, dt = 259200 seconds.

```
dt = 3*24*3600                                                      # timestep
```

This code is defining a constant value for the gravitational constant, G. The value is 6.67430 x 10-11, which is the standard value of the gravitational constant used in physics.

```
G = 6.67430e-11                                                     # Newton's Gravitational Constant
```

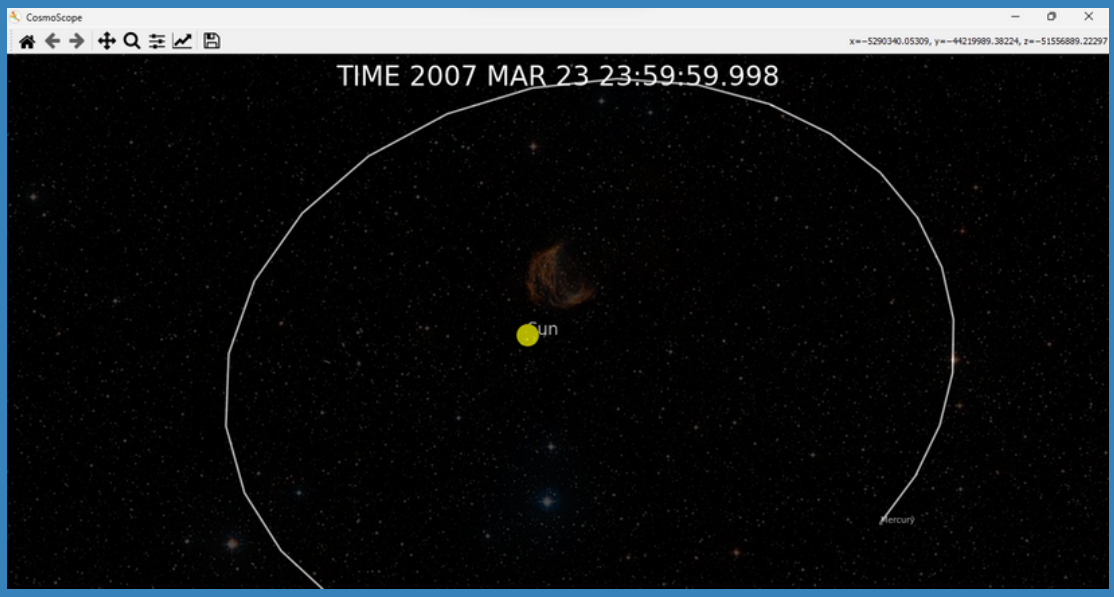
This code takes the value of t_end (the end time of a simulation) and divides it by the value of dt (the time step size). The result is then rounded up to the nearest integer using the ceil() function from the NumPy library and stored in the variable num_ts. num_ts is therefore equal to the number of time steps required to complete the simulation.

```
num_ts = int(np.ceil(t_end/dt))                                     # number of timesteps
```

COSMOSCOPE PROJECT MANUAL

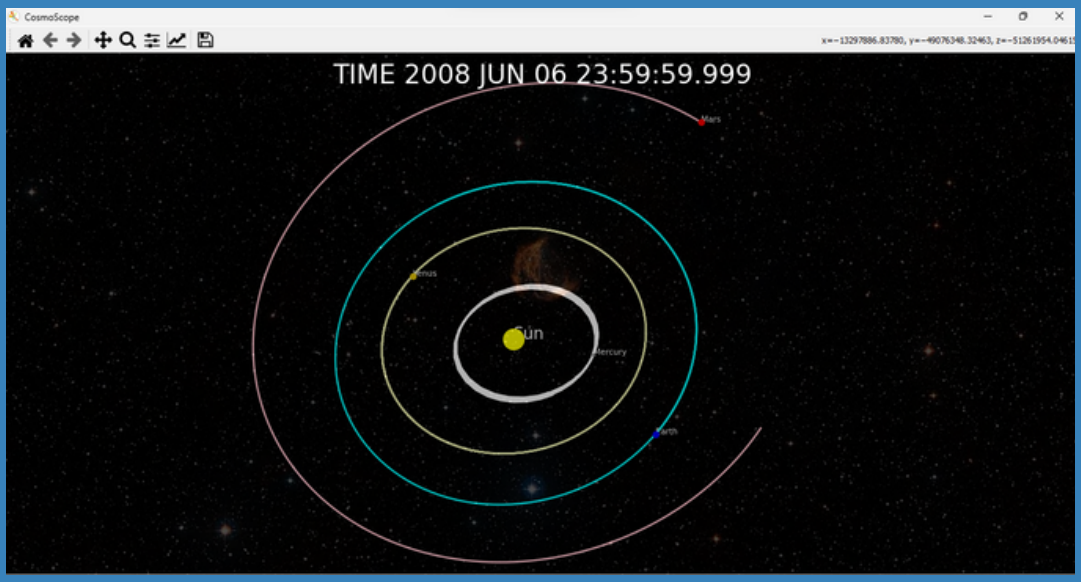
FIND THE 'COSMOSCOPE' EXECUTABLE FILE

DOUBLE CLICK AND A SIMULATION WINDOW WILL POP UP



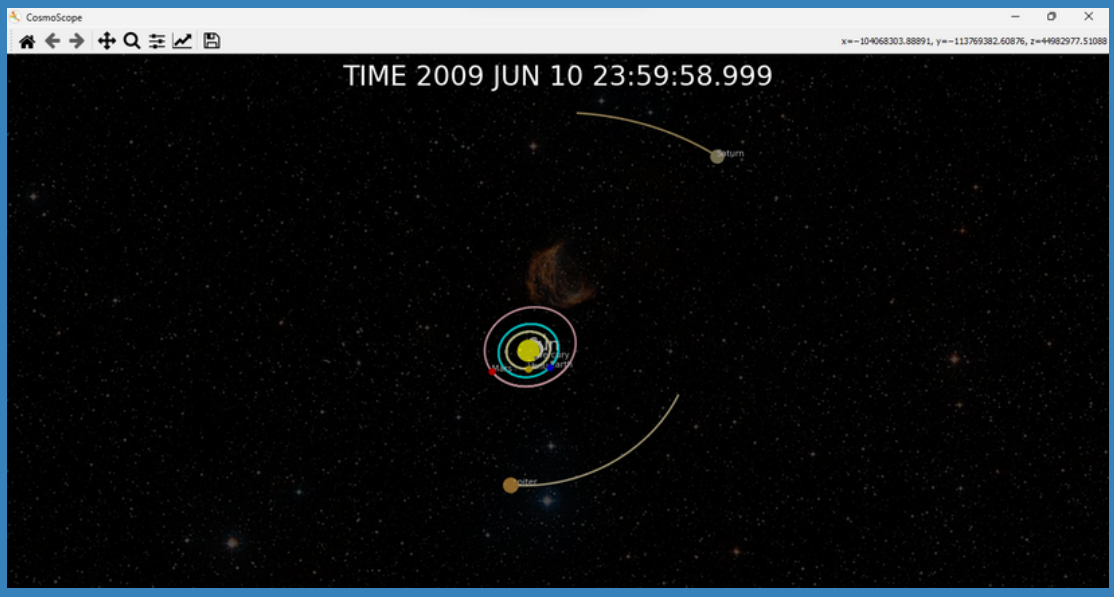
COSMOSCOPE PROJECT MANUAL

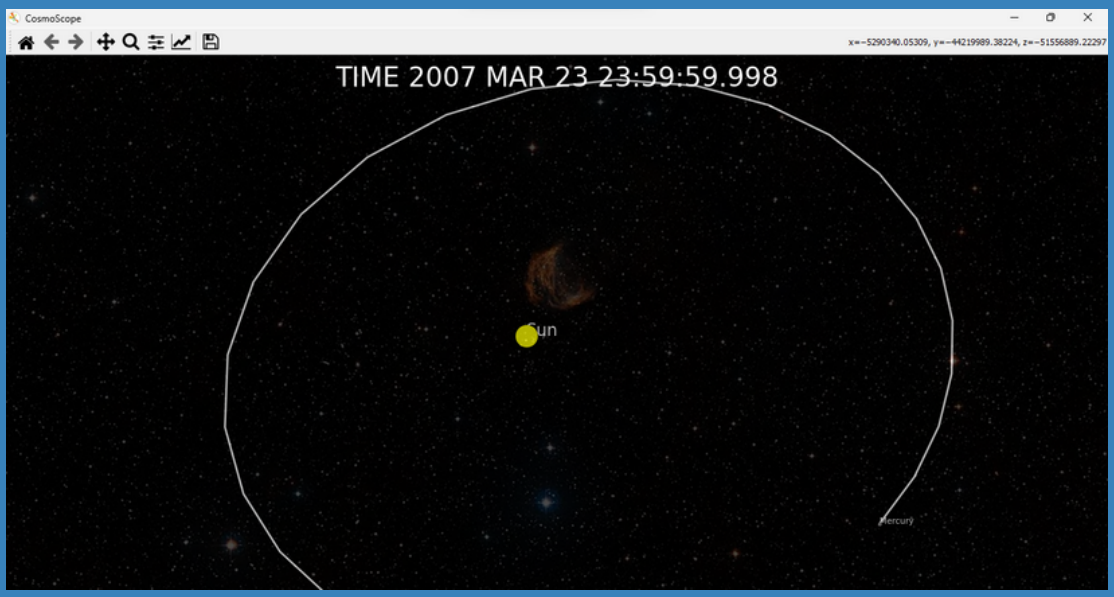
THE SIMULATION STARTS AND THE USER COULD ZOOM OUT THE SIMULATION TO SEE THE OTHER PLANETS



COSMOSCOPE PROJECT MANUAL

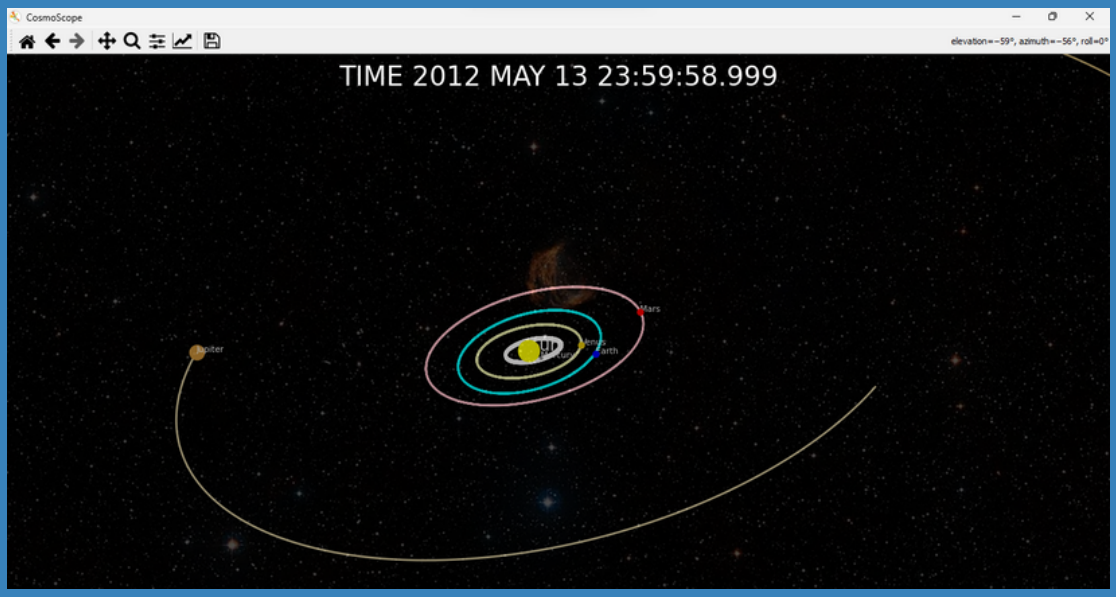
THE USER COULD ALSO ZOOM IN THE SIMULATION IF THEY ARE TOO FAR OUT THE SOLAR SYSTEM





COSMOSCOPE PROJECT MANUAL

SINCE IT IS A 3D SIMULATION, THE USER COULD TILT THE FIGURE TO SEE THE SYSTEM IN PARTICULAR ANGLES



AFTER SOME TIME, THE SIMULATION WOULD END AND THE USER WOULD HAVE THE CHANCE TO RUN IT AGAIN, BY CLICKING THE EXECUTABLE FILE 'COSMOSCOPE'.