eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

CIENCIA Y TECNOLOGÍA ESPACIAL

TRABAJO FIN DE MÁSTER

ELABORACIÓN DE UN PROGRAMA PARA SIMULAR Y REPRESENTAR EL PROBLEMA DE N CUERPOS

Estudiante	*Galindo Leandro, Iñigo Jesús*
Director/Directora	*Rojas Palenzuela, Jose Félix*
Departamento	**Física Aplicada**
Curso académico	*2021/2022*

Bilbao, septiembre 2022

Resumen

Castellano

En este trabajo se repasan las leyes de interacción gravitatoria entre 2, 3 y N cuerpos. Tras ello se presenta un método numérico para resolver dichos problemas y se elabora un software programado en Python para calcularlos y representarlos. Por último, se presentan los resultados más relevantes que proporciona el programa y se constata que funciona adecuadamente.

Euskara

Lan honetan 2, 3 eta N gorputzen arteko grabitazio legeak berrikusiko ditugu. Ondoren problema hauek ebazteko zenbakizko metodo bat aurkezten dugu eta Python lengoaian idatzitako programa bat garatzen dugu problema hauen erantzunak kalkulatu eta irudikatzeko. Azkenik, programa honek ematen dizkigun emaitza garrantzitsuenak aurkezten ditugu eta softwarea ondo dabilela egiaztatzen dugu.

English

This thesis reviews the laws of gravitational interaction between 2, 3 and N bodies. Then, a numerical method is described to solve these problems. With these tools a Python code is developed to compute and draw the solutions. Finally, the most relevant results provided by the program are presented and it is shown that it works properly.

Índice general

Introducción

Desde la desaparición de My Solar System como herramienta didáctica de astrodinámica no ha aparecido ninguna alternativa que la sustituya.

En el presente trabajo se hace repaso de las leyes que gobiernan las interacciones gravitatorias clásicas y se presentan los problemas de 2, 3 y N cuerpos así como algunos resultados interesantes que se extraen del análisis de dichos problemas.

Algunos de los problemas que aparecen al analizar dichas leyes no tienen solución analítica por lo que, a continuación, se describen algunos métodos numéricos que pueden ayudar a aproximar un resultado.

Una vez descritas las bases teóricas se describe brevemente el programa que se ha realizado en Python para simular las interacciones gravitatorias y a continuación se presentan los principales resultados que se han obtenido.

De esta forma se repasan e integran gran parte de los conocimientos sobre astrodinámica adquiridos y además se muestra una forma de elaborar un software que sustituya al ya citado My Solar System.

Capítulo 1

Teoría relevante

En el presente capítulo se sientan las bases teóricas que nos permiten desarrollar el software de simulación.

Primero se comienza describiendo brevemente las leyes de interacción gravitatoria clásicas para luego pasar a analizar los problemas de 2, 3 y N cuerpos.

A continuación se describen algunos métodos numéricos y se explica el que finalmente se ha empleado para llevar a cabo las simulaciones.

Por último, se describe la estructura y el funcionamiento interno del software.

1.1. Interacción gravitatoria

1.1.1. Cálculo de variaciones y notación lagrangiana

Se supone una curva $y(x)$ que une los puntos (x_1, y_1) y (x_2, y_2). Si se define una integral

$$S = \int_{x_1}^{x_2} f[y(x), y'(x), x]dx \tag{1.1}$$

El camino para el cual esta integral es estacionaria viene dado por la conocida como ecuación de Euler-Lagrange [1].

$$\frac{\partial f}{\partial y} - \frac{d}{dx}\frac{\partial f}{\partial y'} = 0 \tag{1.2}$$

Ahora se considera una partícula que se mueve libremente en tres dimensiones sujeta a una fuerza conservativa $\vec{F}(\vec{r})$. La energía potencial y cinética de la partícula vienen dadas respectivamente por

$$T = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2 + \dot{z}^2) \tag{1.3}$$

$$U = U(\vec{r}) = U(x, y, z). \tag{1.4}$$

Se define la función lagrangiana como [1]

$$\mathcal{L} = T - U. \tag{1.5}$$

De acuerdo al principio de Hamilton el camino real que sigue una partícula entre los tiempos t_1 y t_2 es tal que la integral de acción es estacionaria cuando se toma a lo largo del camino real [1]. Es decir,

$$S = \int_{t_1}^{t_2} \mathcal{L}dt \tag{1.6}$$

tiene que ser estacionario.

Se puede demostrar que para un sistema con n grados de libertad y n coordenadas generalizadas, y fuerzas que no son de ligadura que se pueden derivar de un potencial con la forma $U(q_1, ..., q_n, t)$, el camino que sigue el sistema viene determinado por [1]

$$\frac{\partial \mathcal{L}}{\partial q_i} = \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} \qquad [i = 1, ..., n]. \tag{1.7}$$

1.1.2. El problema de los dos cuerpos

Ahora se considera un sistema de dos partículas puntuales con masas m_1 y m_2 sometidas únicamente a la su interacción mutua. Si se considera que dicha interacción es gravitatoria se tendrá que el potencial tiene la siguiente forma

$$U(\mathbf{r}_1, \mathbf{r}_2) = U(|\mathbf{r}_1 - \mathbf{r}_2|) = -\frac{Gm_1m_2}{|\mathbf{r}_1 - \mathbf{r}_2|}. \tag{1.8}$$

Donde $\mathbf{r}_1$ y $\mathbf{r}_2$ son las posiciones de las partículas medidas desde un sistema de referencia inercial.

Con el fin de aprovechar que el potencial depende únicamente de $|\mathbf{r}_1 - \mathbf{r}_2|$ se introduce una una nueva variable

$$\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2. \tag{1.9}$$

De esta forma la función lagrangiana del sistema puede definirse como

$$\mathcal{L} = \frac{1}{2}m_1\dot{\mathbf{r}_1}^2 + \frac{1}{2}m_2\dot{\mathbf{r}_2}^2 - U(r) \tag{1.10}$$

Si se aplica la ecuación (1.7) se obtienen las ecuaciones de movimiento de cada partícula.

$$\frac{Gm_1m_2}{r^2} = m_1\ddot{\mathbf{r}_1}^2 \tag{1.11}$$

$$-\frac{Gm_1m_2}{r^2} = m_2\ddot{\mathbf{r}_2}^2 \tag{1.12}$$

Para extraer datos de interés de esta interacción conviene pasar al sistema de referencia del centro de masas. Dicho sistema viene definido por

$$\mathbf{R} = \frac{m_1\mathbf{r}_1 + m_2\mathbf{r}_2}{M} \tag{1.13}$$

Donde M denota la suma de ambas masas.

De acuerdo a la Figura 1.1 los vectores $\mathbf{r}_1$ y $\mathbf{r}_2$ guardan la siguiente relación con el sistema de referencia del centro de masas.

$$\mathbf{r_1} = \mathbf{R} + \frac{m_2}{M}\mathbf{r} \tag{1.14}$$

$$\mathbf{r_2} = \mathbf{R} - \frac{m_1}{M}\mathbf{r} \tag{1.15}$$

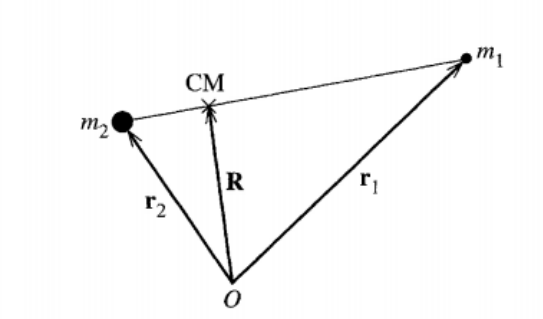

Figura 1.1: Esquema que indica la posición del centro de masas del sistema. Imagen extraída de [1].

De esta forma la energía cinética en el nuevo sistema de referencia es

$$T = \frac{1}{2}(M\dot{\mathbf{R}}^2 + \mu\dot{\mathbf{r}}^2). \tag{1.16}$$

Donde $\mu = \dfrac{m_1m_2}{M}$ es la masa reducida. Sustituyendo en la función lagrangiana se obtiene

$$\mathcal{L} = \frac{1}{2}M\dot{\mathbf{R}}^2 + \left(\frac{1}{2}\mu\dot{\mathbf{r}}^2 - U(r)\right) \tag{1.17}$$

Como se puede ver la función está ahora separada en dos partes. Una depende de la posición del centro de masas y la otra de la posición relativa de los cuerpos. Además, dado que se ha elegido el centro de masas como sistema de referencia se tendrá que $\dot{\mathbf{R}} = 0$. Esto simplifica aún más el problema dado que pasamos de tener que resolver dos partículas de forma simultánea a tener que resolver un problema equivalente que sólo involucra a una.

Teniendo el siguiente lagrangiano

$$\mathcal{L} = \frac{1}{2}\mu\dot{\mathbf{r}}^2 - U(r) = \frac{1}{2}\mu\dot{\mathbf{r}}^2 + \frac{Gm_1m_2}{r} \tag{1.18}$$

si se pasa a coordenadas polares se obtiene

$$\mathcal{L} = \frac{1}{2}\mu(\dot{r}^2 + r^2\dot{\theta}^2) + \frac{Gm_1m_2}{r}. \tag{1.19}$$

Aplicando la ecuación (1.17) se obtiene por un lado la ley de conservación del momento angular

$$\mu r^2\dot{\theta} = \text{cte.} = l \tag{1.20}$$

y por otro lado una ecuación de movimiento

$$\mu\ddot{r} = \mu r\dot{\theta}^2 - \frac{Gm_1m_2}{r^2} \rightarrow \mu\ddot{r} = \frac{l^2}{\mu r^3} - \frac{Gm_1m_2}{r^2} \tag{1.21}$$

$$\mu\ddot{r} = -\frac{d}{dr}\left[-\frac{Gm_1m_2}{r} + \frac{l^2}{2\mu r}\right] \tag{1.22}$$

De esto se deduce que el movimiento radial es el de una partícula sometida a un potencial

$$U_{ef} = -\frac{Gm_1m_2}{r} + \frac{l^2}{2\mu r^2}. \tag{1.23}$$

que se denomina potencial efectivo.

Para conseguir más información acerca de las órbitas se multiplica la ecuación (1.22) a ambos lados por $\dot{r}$ y se obtiene la siguiente ecuación

$$\mu\dot{r}\ddot{r} = -\frac{dr}{dt}\frac{dU_{ef}(r)}{dr} \tag{1.24}$$

$$\frac{d}{dt}\left(\frac{1}{2}\mu\dot{r}^2\right) = -\frac{d}{dt}U_{ef}(r). \tag{1.25}$$

Esta ecuación es la ecuación de conservación de la energía.

$$E = \frac{1}{2}\mu\dot{r}^2 + U_{ef}(r) = \frac{1}{2}\mu\dot{r}^2 + \frac{1}{2}\mu r^2\dot{\theta}^2 + U(r) \tag{1.26}$$

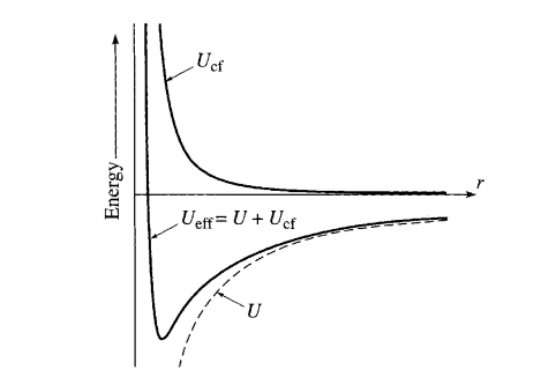

Figura 1.2: Gráfica del potencial efectivo con la energía y la distancia radial en los ejes. Imagen extraída de [1].

Tal y como se puede ver en la Figura 1.2 si la energía total de la órbita coincide con el mínimo del potencial efectivo la distancia radial de la órbita será constante, es decir, la velocidad radial será nula.

Por otro lado, si la energía es superior a dicho mínimo, pero inferior a 0 dicha órbita oscilará entre dos valores de r. Por lo que la velocidad radial será no nula, pero la órbita seguirá estando acotada.

Por último, en los casos en que la $E \geq 0$ la órbita tendrá un valor mínimo de r, pero no un máximo por lo que la órbita no estará acotada.

Esto está relacionado con la excentricidad de las órbitas (ϵ). De hecho, se puede demostrar que la excentricidad y la energía de la órbita están relacionadas mediante la siguiente ecuación [1]

$$E = \frac{(Gm_1m_2)^2\mu}{2l^2}(\epsilon^2 - 1). \tag{1.27}$$

Esto implica que para $\epsilon < 1$ la energía tomará valores negativos y se tendrán órbitas acotadas. En concreto, si $\epsilon = 0$ se tendrá el mínimo de energía correspondiente a la órbita circular. Para excentricidades $\epsilon = 1$ se tiene $E = 0$ y la órbita será parabólica. Como última alternativa se tendría que la excentricidad sea mayor que 1 y que la energía sea superior a 0. Esto daría como resultado una órbita hiperbólica.

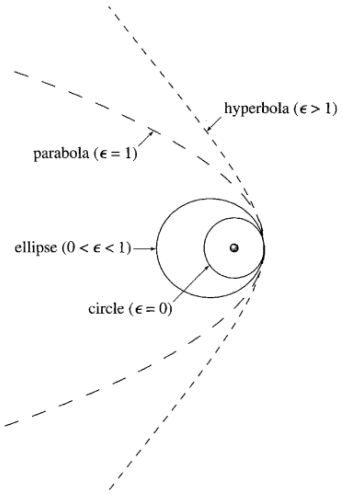

Figura 1.3: Excentricidad y energía de cada tipo de órbita. Imagen extraída de [1].

En el apartado de resultados se demuestra que el software es capaz de reproducir todo lo que se acaba de enunciar.

1.1.3. El problema de los 3 cuerpos

Se supone un sistema con tres masas m_1, m_2 y m_3 situadas en las posiciones $\mathbf{r}_1$, $\mathbf{r}_2$ y $\mathbf{r}_3$ desde un sistema de referencia inercial. La función lagrangiana de este sistema es

$$\mathcal{L} = \frac{1}{2}\left(m_1\dot{\mathbf{r}_1}^2 + m_2\dot{\mathbf{r}_2}^2 + m_3\dot{\mathbf{r}_3}^2\right) + \left(\frac{Gm_1m_2}{|\mathbf{r}_1 - \mathbf{r}_2|} + \frac{Gm_1m_3}{|\mathbf{r}_1 - \mathbf{r}_3|} + \frac{Gm_3m_2}{|\mathbf{r}_3 - \mathbf{r}_2|}\right). \quad (1.28)$$

Si se aplica la ecuación (1.7) a esta función se obtienen las siguientes ecuaciones de movimiento para cada masa.

$$\ddot{\mathbf{r}}_1 = -\frac{Gm_2}{|\mathbf{r_1} - \mathbf{r_2}|^3}(\mathbf{r_1} - \mathbf{r_2}) - \frac{Gm_3}{|\mathbf{r_1} - \mathbf{r_3}|^3}(\mathbf{r_1} - \mathbf{r_3}) \quad (1.29)$$

$$\ddot{\mathbf{r}}_2 = \frac{Gm_1}{|\mathbf{r_1} - \mathbf{r_2}|^3}(\mathbf{r_1} - \mathbf{r_2}) + \frac{Gm_3}{|\mathbf{r_3} - \mathbf{r_2}|^3}(\mathbf{r_3} - \mathbf{r_2}) \quad (1.30)$$

$$\ddot{\mathbf{r}}_3 = \frac{Gm_1}{|\mathbf{r_1} - \mathbf{r_3}|^3}(\mathbf{r_1} - \mathbf{r_3}) - \frac{Gm_2}{|\mathbf{r_3} - \mathbf{r_2}|^3}(\mathbf{r_3} - \mathbf{r_2}) \quad (1.31)$$

A diferencia del problema de dos cuerpos, no se puede obtener una solución analítica general para estas ecuaciones. Sí se puede, en cambio, obtener una solución particular para ciertos casos como se va a ver a continuación.

Puntos de Lagrange

Se suponen dos masas m_1 y m_2 mucho mayores que la tercera (m_3) que se mueven con una velocidad angular constante ω alrededor de su CM en trayectoria circular.

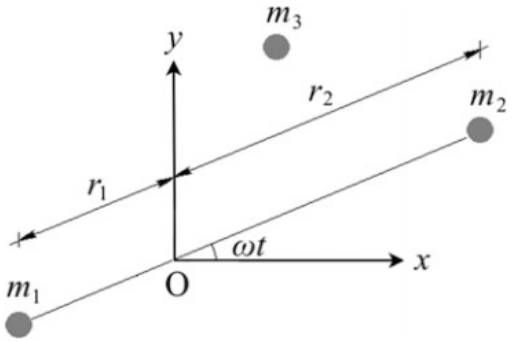

Figura 1.4: Esquema del problema de los 3 cuerpos simplificada. Imagen extraída de [4].

Si se sitúa el origen del sistema de referencia no inercial en el CM las fuerzas que actúan sobre la tercera masa serán

$$\mathbf{F}_3 = \frac{Gm_1m_3}{|\mathbf{r_1} - \mathbf{r_3}|^3}(\mathbf{r_1} - \mathbf{r_3}) - \frac{Gm_2m_3}{|\mathbf{r_3} - \mathbf{r_2}|^3}(\mathbf{r_3} - \mathbf{r_2}) - 2m_3(\omega \times \mathbf{v}) - m_3\omega \times (\omega \times \mathbf{r}). \tag{1.32}$$

Haciendo los siguientes cambios de variables se puede llegar a una expresión para las ecuaciones de movimiento de la masa m_3 [2].

$$M = m_1 + m_2 \tag{1.33}$$
$$x_2 = x_1 + |\mathbf{r}_1 - \mathbf{r}_2| = x_1 + r_{12} \tag{1.34}$$
$$\pi_1 = \frac{m_1}{m_1 + m_2} \tag{1.35}$$
$$\pi_2 = \frac{m_2}{m_1 + m_2} \tag{1.36}$$

$$\ddot{x} - 2\omega\dot{y} - \omega^2 x = -\frac{GM_1}{r_1^3}(x + \pi_2 r_{12}) - \frac{GM_2}{r_2^3}(x - \pi_1 r_{12}) \tag{1.37}$$
$$\ddot{y} + 2\omega\dot{x} - \omega^2 y = -\frac{GM_1}{r_1^3}y - \frac{GM_2}{r_2^3}y \tag{1.38}$$
$$\ddot{z} = -\frac{GM_1}{r_1^3}z - \frac{GM_2}{r_2^3}z \tag{1.39}$$

Como ya se ha mencionado estas ecuaciones no tienen solución analítica, pero se pueden hallar soluciones particulares suponiendo que la masa m_3 se encuentra en

reposo, es decir, cuando todas las componentes de la velocidad y la aceleración sean nulas. Esto implicaría que para un observador inercial la masa m_3 se movería en órbitas circulares alrededor de las otras dos masas [2].

Los puntos en los que m_3 está en reposo se denominan puntos de Lagrange. La posición de dos ellos viene dada por [2]

$$x = \frac{r_{12}}{2} - \pi_2 r_{12} \tag{1.40}$$

$$y = \pm \frac{\sqrt{3}}{2} r_{12}. \tag{1.41}$$

Los otros tres puntos se tienen que obtener resolviendo numéricamente la siguiente ecuación 2

$$\frac{1-\pi_2}{|\xi+\pi_2|^3}(\xi+\pi_2) + \frac{\pi_2}{|\xi+\pi_2-1|^3}(\xi+\pi_2-1) - \xi = 0. \tag{1.42}$$

Donde $\xi = \dfrac{x}{r_{12}}$.

Para el sistema Tierra-Luna los puntos están situados tan y como se puede ver en la Figura 1.5.

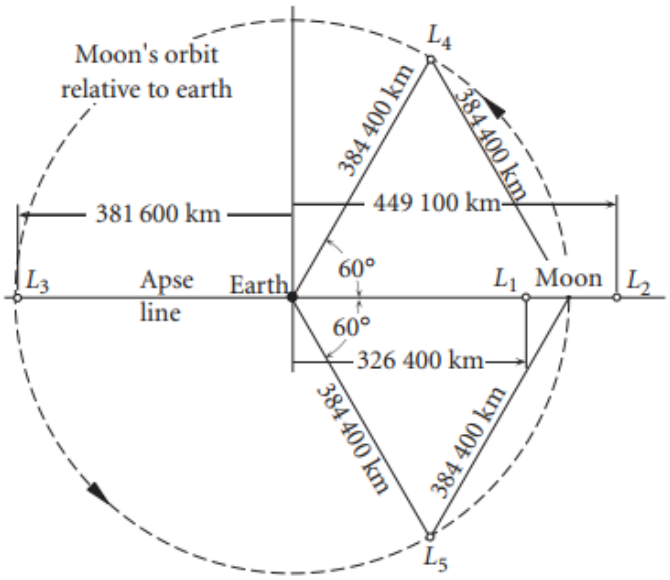

Figura 1.5: Posición de los puntos del Lagrange del sistema Tierra-Luna. Imagen extraída de [2].

Los puntos que se hallan sobre la línea que une la dos masas más grandes son muy interesantes desde el punto de vista de la astrodinámica dado que es posible insertar naves espaciales que mantengan una órbita cuasiperiódica alrededor de

ellos. Dependiendo de las condiciones iniciales a la hora de insertar la nave las órbitas serán de Halo o Lissajous [3].

Simular estas órbitas adecuadamente es otro de los objetivos de este trabajo, pero para ello no se resolverán las ecuaciones de movimiento desde el sistema de referencia no inercial (ecuaciones 1.37, 1.38, 1.39) sino que se resolverán directamente las ecuaciones desde el sistema de referencia inercial y luego se hará un cambio de coordenadas al sistema de referencia no inercial mediante el siguiente cambio de coordendas

$$x' = x\cos(-\omega t) + y\sin(-\omega t) \tag{1.43}$$
$$y' = x\sin(-\omega t) + y\cos(-\omega t) \tag{1.44}$$
$$z' = z \tag{1.45}$$

Donde ω es la velocidad angular a la que orbitan las dos masas más grandes alrededor de su centro de masas y las coordenadas primadas indican que se observa desde el sistema de referencia no inercial.

1.1.4. El problema de N cuerpos

Se supone un sistema con n partículas de masas $m_1, m_2 ... m_n$ sometidas sólo a su interacción gravitatoria mutua. La lagrangiana de dicho sistema será la siguiente

$$\mathcal{L} = \sum_{i=1}^{n} \frac{1}{2} m_i \dot{\mathbf{r}}_i^2 + \sum_{\substack{i,j=1 \\ i\neq j}}^{n} \frac{G m_i m_j}{|\mathbf{r}_i - \mathbf{r}_j|} \tag{1.46}$$

Si se aplica la ecuación (1.7) las ecuaciones de movimiento de cada masa serán

$$\ddot{\mathbf{r}}_i = G \sum_{\substack{j=1 \\ i\neq j}} \frac{m_j}{|\mathbf{r}_i - \mathbf{r}_j|^3} (\mathbf{r}_i - \mathbf{r}_j). \tag{1.47}$$

La forma matricial de este sistema de ecuaciones sería

$$\begin{pmatrix} \ddot{\mathbf{r}}_i \\ \vdots \\ \ddot{\mathbf{r}}_n \end{pmatrix} = \begin{pmatrix} \sum_{j\neq 1}^{n} \frac{Gm_j}{r_{1j}^3} & -\frac{Gm_2}{r_{12}^3} & \cdots & -\frac{Gm_n}{r_{1n}^3} \\ -\frac{Gm_1}{r_{21}^3} & \sum_{j\neq 2}^{n} \frac{Gm_j}{r_{2j}^3} & \cdots & -\frac{Gm_n}{r_{2n}^3} \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{Gm_1}{r_{n1}^3} & \cdots & -\frac{Gm_{n-1}}{r_{n,n-1}^3} & \sum_{j\neq n}^{n} \frac{Gm_j}{r_{nj}^3} \end{pmatrix} \begin{pmatrix} \mathbf{r}_i \\ \vdots \\ \mathbf{r}_n \end{pmatrix}. \tag{1.48}$$

La resolución de este problema es el principal objetivo de este trabajo y del software que se ha creado, pero al igual que el problema de los 3 cuerpos no tiene una solución analítica general. Esto hace necesario resolver el problema a través de métodos numéricos.

1.2. Método de Runge-Kutta de orden 4

Hay multitud de métodos numéricos que permiten integrar las órbitas en un problema de N Cuerpos. Como el método de Gauss-Jackson, el de salto de rana, el de Hermite o el de Runge-Kutta. El software implementa el método de Runge-Kutta de orden 4 con un paso adaptativo para cada cuerpo.

La principal desventaja del método de Runge-Kutta es que se trata de un método no simpléctico, es decir, la energía atribuible a la órbita va a tener un error que va a aumentar con el tiempo. Pero, al mismo, tiempo, para periodos de simulación cortos ofrece unos errores muy bajos comparados con otros métodos debido a que se trata de un método de cuarto orden. Tal y como se puede ver en la Figura 1.6 estos errores se hacen considerables sólo a partir de 30 periodos orbitales.

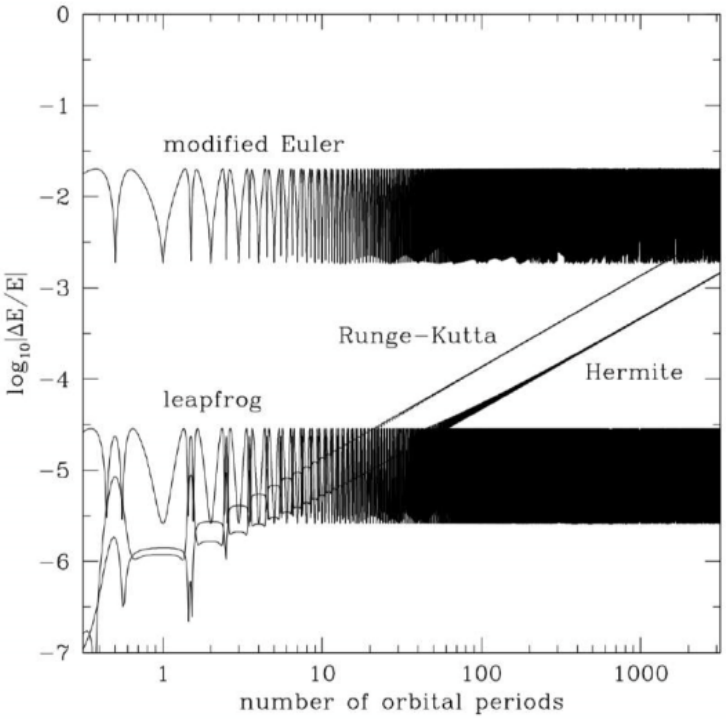

Figura 1.6: Error en la energía de la órbita frente a periodos orbitales para diferentes métodos de integración. Imagen extraída de [5].

A pesar de los citados problemas se ha elegido este método porque el objetivo del software no es hacer simulaciones a largo plazo y además su implementación es muy sencilla tal y como se verá a continuación.

Para conseguir métodos de orden superior a 2, como el de Runge-Kutta, hay que emplear series de Taylor [6]. Se supone un vector que nos da el estado del sistema en un instante t_n, $y(t_n)$ y una función tal que $\dot{y}(t) = f(y(t), t)$. El estado en un instante de tiempo posterior $t_n + \Delta t$ será [6]

$$y(t_n + \Delta t) = y(t_n) + \Delta t f(y(t_n), t_n) + \frac{\Delta t^2}{2} \frac{df(y(t_n), t_n)}{dt} + ... \tag{1.49}$$

Tal y como se puede comprobar esto exige conocer las derivadas de la función. Esto en ocasiones puede ser algo desconocido, pero hay una solución: aproximarlas por diferencias numéricas. Por ejemplo, para deducir el método de Runge-Kutta de orden 2 se parte de la siguiente función

$$K_1 = f_n = f(y(t_n), t_n) \tag{1.50}$$

El estado del sistema en el instante $t_n + \Delta t$ es

$$y(t_n + \Delta t) \approx \Delta t K_1 \tag{1.51}$$

El gradiente de función se aproxima mediante

$$K_2 = f(y(t_n) + \Delta t K_1, t_n + \Delta t) \tag{1.52}$$

Aplicando la regla del trapecio se obtiene el valor aproximado de y_{n+1}

$$y_{n+1} = y_n + \frac{\Delta t}{2}(K_1 + K_2). \tag{1.53}$$

El método de Runge-Kutta empleado es el de cuarto orden, que mejora la precisión del que acabamos de deducir incluyendo dos cálculos del gradiente en $\Delta t/2$. Los valores de función de dicho método son [6]

$$K_1 = f(y(t_n), t_n) \tag{1.54}$$
$$K_2 = f\left(y(t_n) + \frac{K_1}{2}\Delta t, t_n + \frac{\Delta t}{2}\right) \tag{1.55}$$
$$K_3 = f\left(y(t_n) + \frac{K_2}{2}\Delta t, t_n + \frac{\Delta t}{2}\right) \tag{1.56}$$
$$K_4 = f(y(t_n) + K_3 \Delta t, t_n + \Delta t) \tag{1.57}$$

Aplicando la regla de Simpson estos valores nos permiten aproximar el valor de la función en un instante de tiempo posterior.

$$y_{n+1} = y(t_n) + \frac{\Delta}{6}(K_1 + 2K_2 + 2K_3 + K_4) = y(t_n + \Delta t) + O(\Delta t^5). \tag{1.58}$$

Además, para mejorar la precisión del método se puede implementar un control de paso adaptativo. Para ello es necesario calcular en cada paso el error de truncamiento que se comete. Esto se hace calculado el valor de la función en $t_n + \Delta t$ mediante un paso completo y dos medios pasos [7].

$$y_{n+1} = RK4(y_n, \Delta t) \tag{1.59}$$
$$y'_{n+1} = RK4(RK4(y_n, \Delta t/2)) \tag{1.60}$$

Si se restan estas dos cantidades se obtiene el error de truncamiento

$$\Delta_1 = |y_{n+1} - y'_{n+1}|. \tag{1.61}$$

Si se quiere mantener este resultado dentro de la precisión deseada Δ_0 se habrá de cambiar la magnitud del paso en cada iteración del método. Para ello se calculará cada vez Δ_1 y se cambiará el valor de Δt de acuerdo con la siguiente ecuación [7]

$$h_0 = h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{\alpha} \begin{cases} \alpha = 0{,}2 \text{ si } \Delta_0 > \Delta_1 \\ \alpha = 0{,}25 \text{ si } \Delta_0 \leq \Delta_1 \end{cases} \tag{1.62}$$

Donde h_1 es el paso temporal empleado inicialmente y h_0 el corregido.

1.2.1. Aplicación al problema de N Cuerpos

Todo lo que se acaba de explicar tiene su aplicación concreta al problema de los N Cuerpos en el presente trabajo. Se tienen las ecuaciones de movimiento indicadas en (1.34), pero el método de Runge-Kutta descrito sólo permite resolver ecuaciones de primer orden. Es por esto que lo que habrá que hacer es resolver simultáneamente los dos siguiente sistemas de ecuaciones

$$\dot{\mathbf{r}}_i = \mathbf{v}_i \tag{1.63}$$

$$\dot{\mathbf{v}}_i = G \sum_{\substack{j=1 \\ i \neq j}} \frac{m_j}{|\mathbf{r}_i - \mathbf{r}_j|^3} (\mathbf{r}_i - \mathbf{r}_j). \tag{1.64}$$

La forma matricial más adecuada para escribir esto y para implementarlo es la siguiente.

$$\begin{pmatrix} \dot{x}_1 \\ \vdots \\ \dot{x}_n \\ \dot{y}_1 \\ \vdots \\ \dot{y}_n \\ \dot{z}_1 \\ \vdots \\ \dot{z}_n \\ \dot{v}_{1x} \\ \vdots \\ \dot{v}_{nx} \\ \vdots \\ \dot{v}_{1y} \\ \vdots \\ \dot{v}_{ny} \\ \dot{v}_{1z} \\ \vdots \\ \dot{v}_{nz} \end{pmatrix} = \begin{pmatrix} 0 & \cdots & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & \cdots & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \cdots & \cdots & \vdots & \vdots & \cdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 0 & 0 & \cdots & 1 \\ \sum_{j\neq 1}^{n} \frac{Gm_j}{r_{1j}^3} & -\frac{Gm_2}{r_{12}^3} & \cdots & -\frac{Gm_n}{r_{1n}^3} & 0 & \cdots & \cdots & 0 \\ -\frac{Gm_1}{r_{21}^3} & \sum_{j\neq 2}^{n} \frac{Gm_j}{r_{2j}^3} & \cdots & -\frac{Gm_n}{r_{2n}^3} & 0 & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \cdots & \cdots & \vdots \\ -\frac{Gm_1}{r_{n1}^3} & \cdots & -\frac{Gm_{n-1}}{r_{n,n-1}^3} & \sum_{j\neq n}^{n} \frac{Gm_j}{r_{nj}^3} & 0 & \cdots & \cdots & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ y_1 \\ \vdots \\ y_n \\ z_1 \\ \vdots \\ z_n \\ v_{1x} \\ \vdots \\ v_{nx} \\ \vdots \\ v_{1y} \\ \vdots \\ v_{ny} \\ v_{1z} \\ \vdots \\ v_{nz} \end{pmatrix} \tag{1.65}$$

Donde los vectores verticales tienen dimensión $6n$ siendo n el número de cuerpos y la matriz es una matriz $6n \times 6n$.

Las condiciones iniciales del problema se introducirán con un vector como el de la derecha de la ecuación 1.65. Además se aportará un vector con un paso de tiempo inicial para cada cuerpo, $\mathbf{h}$ y un nivel de tolerancia ϵ.

En cada iteración se multiplica el vector de entrada por el nivel de tolerancias y se selecciona para cada cuerpo el error máximo que se puede admitir $\mathbf{\Delta_0}$. Tras esto, se aplica a cada cuerpo el control de paso que se ha explicado en la ecuación 1.62 y se obtiene el error de truncamiento $\mathbf{\Delta_1}$. Con estas dos cantidades calculadas se puede proceder a calcular el paso temporal adecuado para cada cuerpo $\mathbf{h}'$.

Con el paso temporal ya calculado para cada cuerpo se puede pasar a aplicar el método de Runge-Kutta para resolver el sistema. Para ello se calculan $\mathbf{K}_1$, $\mathbf{K}_2$, $\mathbf{K}_3$ y $\mathbf{K}_4$ como sigue

$$\mathbf{K}_1 = W \cdot \mathbf{r}_n \tag{1.66}$$

$$\mathbf{K}_2 = W \cdot \mathbf{r}_n + \mathbf{h}' \cdot \frac{\mathbf{K}_1}{2} \tag{1.67}$$

$$\mathbf{K}_3 = W \cdot \mathbf{r}_n + \mathbf{h}' \cdot \frac{\mathbf{K}_2}{2} \tag{1.68}$$

$$\mathbf{K}_4 = W \cdot \mathbf{r}_n + \mathbf{h}' \cdot \mathbf{K}_3 \tag{1.69}$$

Donde W es la matriz del lado derecho de la ecuación 1.65 y $\mathbf{r_n}$ es el vector del lado derecho.

Con esas cantidades calculadas se puede calcular $\mathbf{r_{n+1}}$ que actuará como vector de entrada en el siguiente paso.

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \frac{\mathbf{h}'}{6} \cdot (\mathbf{K}_1 + \mathbf{K}_2 + \mathbf{K}_3 + \mathbf{K}_4) \tag{1.70}$$

Repitiendo este proceso de forma iterativa permite resolver el problema de N Cuerpos para el tiempo final que se desee.

1.3. Estructura del software

En este último apartado se indica cómo está estructurado el software de simulación. Por un lado se explica cómo emplear la interfaz gráfica y las opciones que ofrece y por otro se explica como funciona el aspecto computacional.

1.3.1. Interfaz gráfica

La interfaz gráfica y el método que permite mostrar la animación se hallan en el código llamado *n_body_inter.py*. Este código consiste en una clase llamada **n_body_simulator** que permite construir con ayuda de la librería tkinter una interfaz gráfica desde la que lanzar fácilmente las simulaciones numéricas.

Al ejecutar *n_body_inter.py* aparece una ventana como la de la Figura 1.7. Al desplegar la caja de selección hay dos opciones: simulación estándar u órbitas alrededor de L_2.

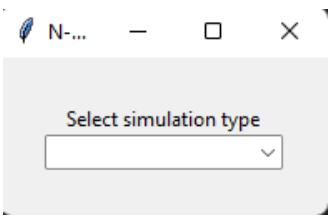

Figura 1.7: Ventana para seleccionar el tipo de simulación.

Si se elige la primera opción se le pedirá al usuario el número de cuerpos que desea simular. Por ejemplo si se quisiera simular todos los planetas del Sistema Solar incluyendo a Plutón habría que escribir "10". De esta forma se desplegarían una serie de cajas en las que habría que introducir las condiciones iniciales de cada cuerpo como se ve en la Figura 1.8.

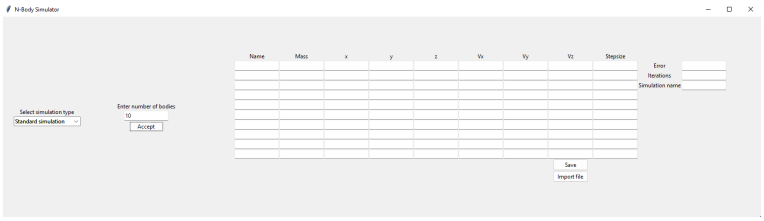

Figura 1.8: Ventana para introducir las condiciones iniciales.

Existe la opción de introducir los datos a mano o importar un archivo .txt dándole al botón Import File. Las primeras 3 líneas de dicho documento contendrán el error que se busca para la simulación, el número de iteraciones y el nombre del GIF que se va a guardar con la animación. Las siguientes líneas contendrán el nombre, masa en kilogramos, posiciones en metros, velocidades en metros segundo y paso temporal en segundos de cada cuerpo. Esto se puede ver en la Figura 1.9.

```
Error 1e-4
Iterations 100000
Name SolarSystem
Sol 2e30 0 0 0 0 0 0 1.0
Mercurio 3.285e23 58e9 0 0 0 -47.85e3 0 2.0
Venus 4.867e24 108.2e9 0 0 0 35e3 0 2.0
Tierra 5.972e24 150e9 0 0 0 30e3 0 2.0
Marte 6.39e23 227.9e9 0 0 0 24.1e3 0 2.0
Jupiter 1.898e27 0 778.5e9 0 -13.1e3 0 0 5.0
Saturno 5.683e26 -1434e9 0 0 0 -9.67e3 0 5.0
Urano 8.681e25 2871e9 0 0 0 6.81e3 0 10.0
Neptuno 1.024e26 -4495e9 0 0 0 -5.477e3 0 10.0
Pluton 1.25e22 6984e9 0 2135e9 0 4.7e3 0 10.0
```

Figura 1.9: Ejemplo del formato del documento .txt.

Una vez se cargan los datos hay que pulsar "Save"para que el software guarde los valores introducidos. Al hacer esto se abrirá un diálogo en el que se podrá elegir la carpeta en la que se quiere guardar el GIF. Una vez hecho esto hay que pulsar el botón "Simulate"que aparecerá y esperar a que aparezca el GIF con la simulación animada.

Por otro lado, si se escoge la opción de órbitas alrededor de L_2 aparece una ventana como la que se ve en la Figura 1.10.

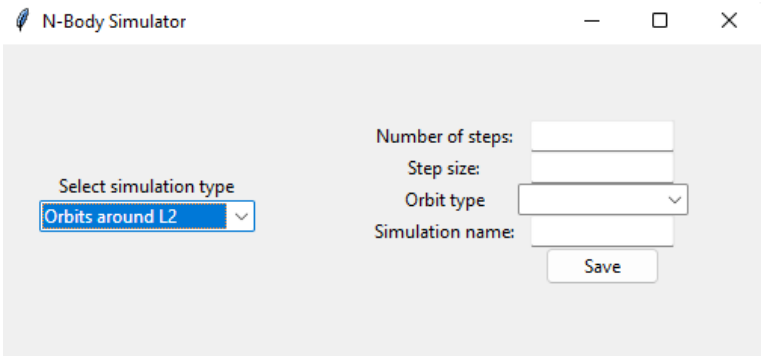

Figura 1.10: Ventana para introducir los datos de las órbitas alrededor de L_2.

Tras introducir el número de pasos, el tamaño del paso, el tipo de órbita y el nombre de la simulación se pulsa Save y aparece un diálogo donde de nuevo hay que elegir dónde guardar el GIF. Tras esto se pulsa el botón Run y se espera que el GIF aparezca.

1.3.2. Cálculo numérico

Todas funciones que permiten realizar las simulaciones a las que se llama desde la interfaz gráfica están en *n_cuerpos.py*.

Para el caso de la simulación estándar se emplea la función **orbita** cuyos inputs son el error, tamaño del stepsize para cada cuerpo, el número de iteraciones, las condiciones iniciales de cada cuerpo, el número de cuerpos y las masas de cada cuerpo.

Con estos inputs la función calcula en cada iteración el stepsize adecuado con la función **checkStep** y procede a aplicar el método de Runge-Kutta de orden 4 con la función **rk4**. Los resultados de todas las iteracione se almacenan en un solo vector que sirve como input al método **draw_results** del código *n_body_inter.py*.

Para dibujar las órbitas de halo y Lissajous se tiene la función **lissajousOrbit** cuyos inputs son el número iteraciones, el tamaño del paso y el tipo de órbita. Esta función no recalcula el paso temporal dado que es necesario que la resolución de las ecuaciones sea simultánea. Simplemente resuelve el problema de 3 cuerpos iterativamente con la función **rk4** y una vez ha finalizado realiza un cambio de sistema de referencia. De esta forma se obtienen dos animaciones, una que es el problema de los 3 cuerpos desde un sistema de referencia inercial y otra que es la órbita del tercer cuerpo desde un sistema de referencia rotatorio. Esta última será la órbita de Lissajous o de halo.

Capítulo 2

Principales resultados

A continuación se presentan los principales resultados que se han obtenido con el software de simulación. Salvo las condiciones iniciales de la sección referente al problema de los 3 cuerpos, el resto de condiciones iniciales vienen en los documentos .txt anexos.

2.1. Simulación del problema de dos cuerpos

Como primera prueba para el software se planteó que fuera capaz de reproducir adecuadamente las 3 órbitas que se pueden obtener en el problema de dos cuerpos.

2.1.1. Órbita circular

Para la simulación de la órbita circular se tomaron el Sol y un cuerpo arbitrario de masa $5 \cdot 10^{22}$ situado a $75 \cdot 10^9$ metros del Sol. Para que la órbita sea circular la velocidad tendrá que ser por lo tanto

$$v = \sqrt{\frac{GM_\odot}{r}} = 42{,}17 \cdot 10^4 \tag{2.1}$$

El resultado de esta simulación se puede ver en la Figura 2.1.

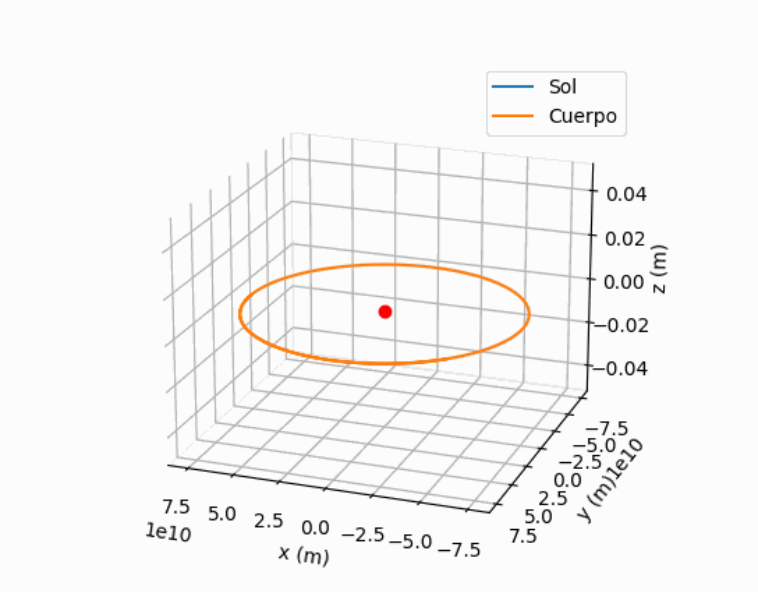

Figura 2.1: Resultado de simular la órbita circular de un cuerpo alrededor del Sol. El tiempo necesario para obtener este resultado fue de 48,47 segundos.

2.1.2. Órbita elíptica

Para simular la órbita elíptica se escogió un cuerpo que tiene una órbita muy excéntrica alrededor del Sol como es el cometa Halley.

Según se puede leer en [8] el 9 de febrero de 1986 el cometa Halley pasó por su perihelio a 87,8 millones de kilómetros del Sol a una velocidad de 54,55 km/s. Introduciendo estas condiciones iniciales se obtuvo la órbita que se puede ver en la Figura 2.2.

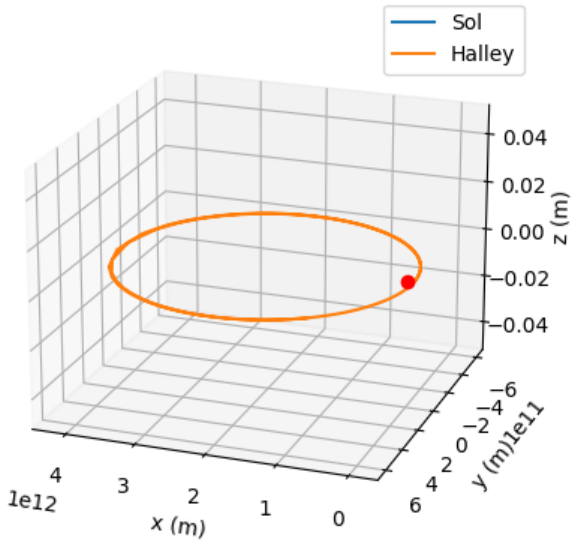

Figura 2.2: Resultados de simular la órbita del cometa Halley alrededor del Sol. El tiempo necesario para obtener este resultado fue de 86,93 segundos.

Tal y como se puede ver la órbita obtenida es muy excéntrica dado que el Sol está muy lejos del centro de la órbita.

Además de esta simulación también se realizó una simulación de la órbita del cometa Halley pero considerando al mismo tiempo todos los planetas del Sistema Solar y Plutón. El resultado de dicha simulación puede verse en la Figura 2.3.

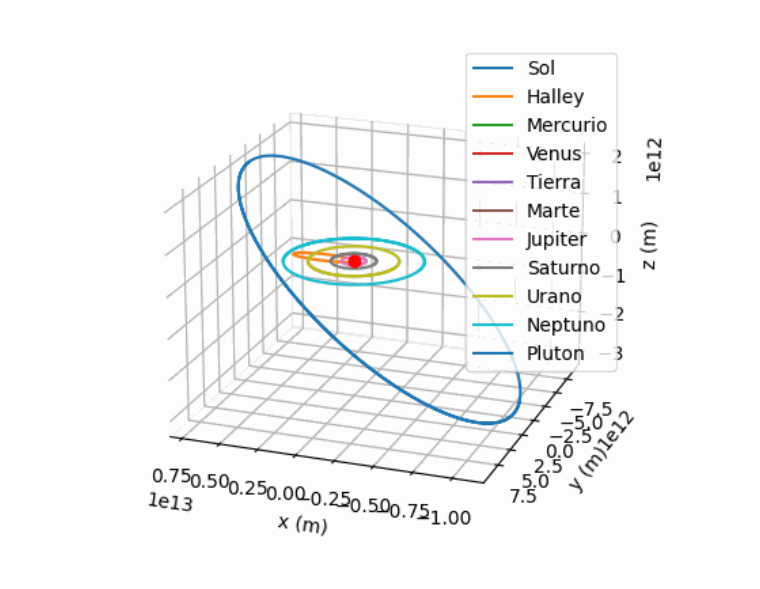

Figura 2.3: Resultado de simular la órbita del cometa Halley junto con el resto de planetas del Sistema Solar alrededor del Sol. El tiempo de simulación fue de 185.35 segundos.

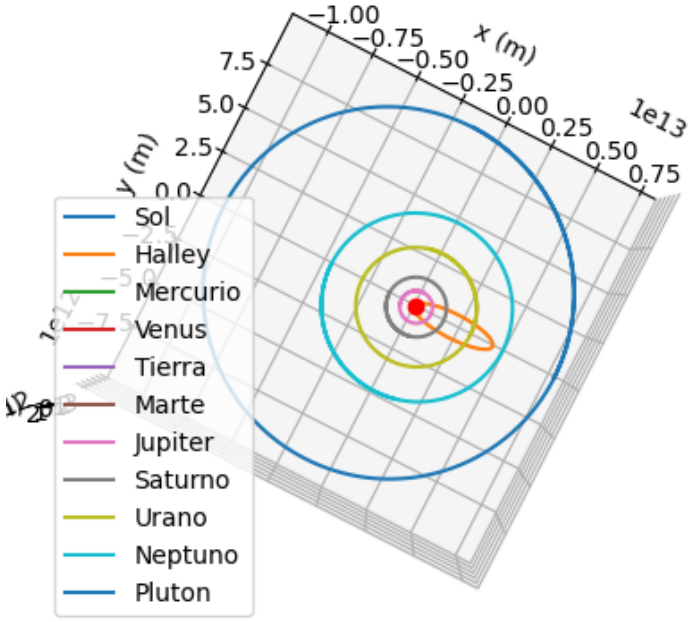

Figura 2.4: Resultado de simular la órbita del cometa Halley junto con el resto de planetas del Sistema Solar alrededor del Sol.

Se puede comprobar que la órbita del cometa Halley sigue siendo elíptica aun a pesar de la influencia del resto de planetas. Esto se debe a que no ha pasado cerca de ningún otro planeta durante el tiempo de simulación y por lo tanto su trayectoria no se ha visto afectada. Si se siguiera corriendo la simulación el probable que se acabara viendo una variación en su trayectoria.

2.1.3. Órbita hiperbólica

Para la órbita hiperbólica se tomó como ejemplo Oumuamua. Según la página Horizons del JPL de la NASA en 2016 este cuerpo se hallaba a 10 unidades astronómica (1500 millones de kilómetros) del Sol y tenía una velocidad de 29,5 km/s. Estas 10 unidades astronómicas se tomaron en la dirección z por encima de la eclíptica y dado que el cuerpo no se dirigía directo al Sol se supuso que su posición en x se hallaba a 200 millones de kilómetros del Sol. De esta forma se obtuvo la Figura

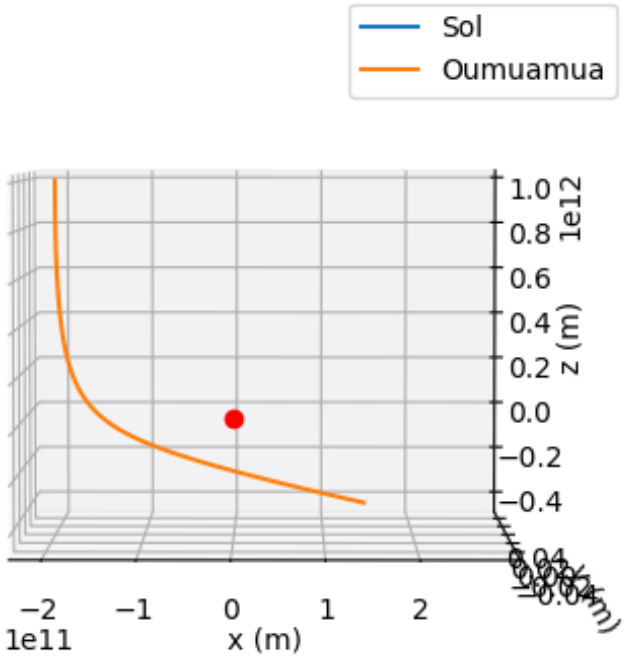

Figura 2.5: Resultado de simular la órbita aproximada de Oumuamua alrededor del Sol. El tiempo que se ha necesitado para esta simulación ha sido de 38,76 segundos.

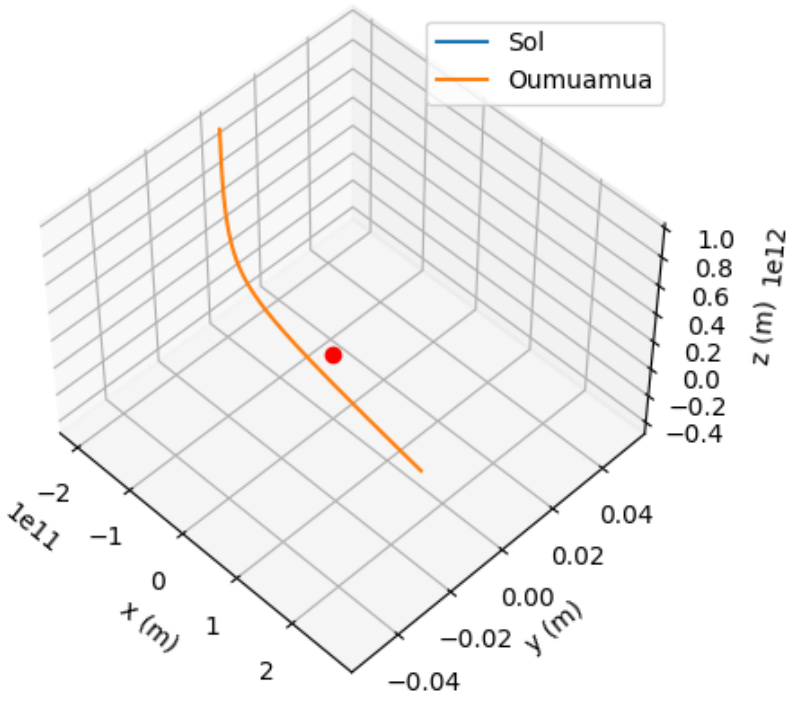

Figura 2.6: Resultado de simular la órbita aproximada de Oumuamua alrededor del Sol.

Al igual que con el cometa Halley dado que se está tratando con cuerpos reales se simuló la órbita de este cuerpo considerando en un primer caso todos los planetas y Plutón y en un segundo caso los 6 planetas más cercanos al Sol.

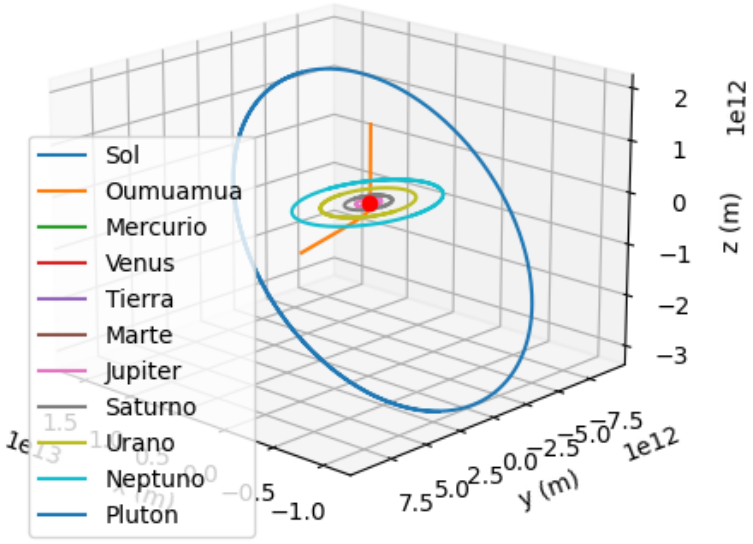

Figura 2.7: Simulación de la órbita de Oumuamua junto con todos los planetas del Sistema Solar y Plutón. El tiempo necesario para esta simulación ha sido de 171.68 segundos.

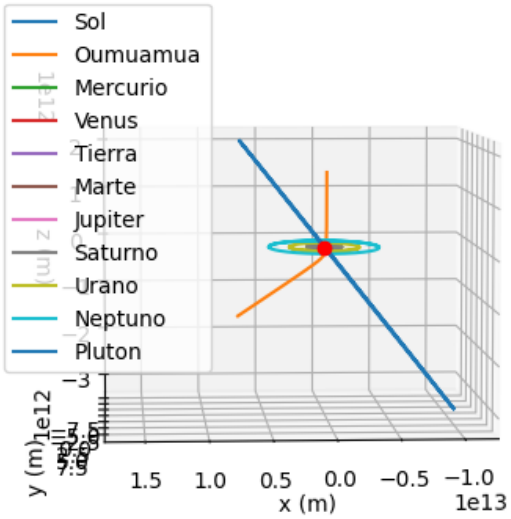

Figura 2.8: Simulación de la órbita de Oumuamua junto con todos los planetas del Sistema Solar y Plutón.

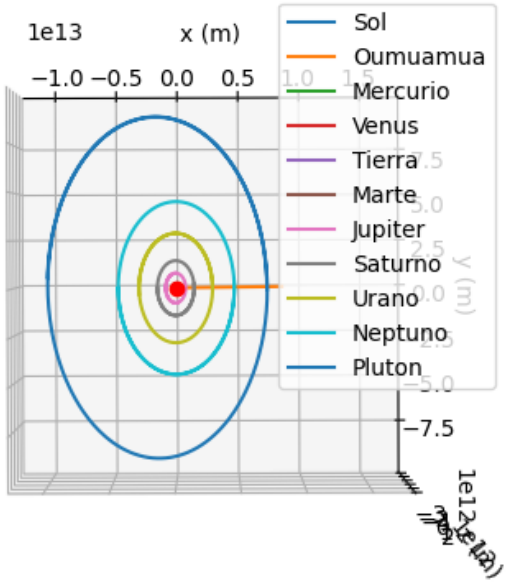

Figura 2.9: Simulación de la órbita de Oumuamua junto con todos los planetas del Sistema Solar y Plutón.

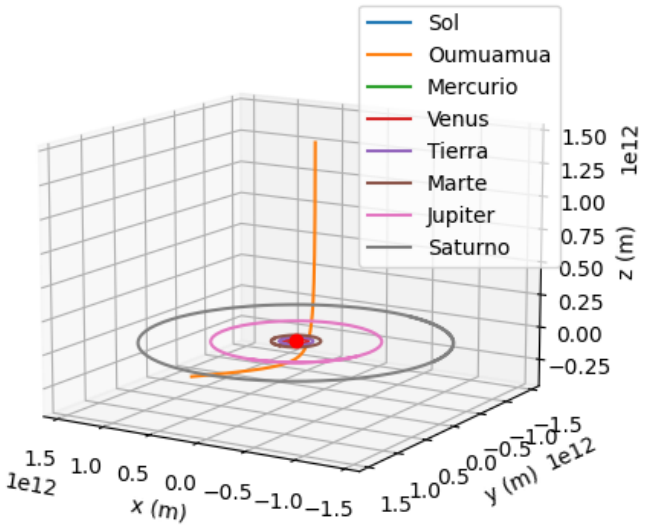

Figura 2.10: Simulación de Oumumua junto con los 6 planetas más cercanos al Sol. El tiempo de simulación fue de 101.52 segundos.

En ambos casos se ve que la órbita sigue siendo hiperbólica a pesar de la interacción con el resto de cuerpos. Esto de nuevo se debe a que Oumuamua no pasa en nuestra simulación lo suficientemente cerca de ningún planeta como para que su trayectoria se vea afectada de forma significativa.

2.1.4. Órbita parabólica

Para obtener la órbita parabólica se supuso que el cuerpo en cuestión tenía la misma posición que en el caso de Oumuamua, pero con una velocidad tal que la órbita fuera parabólica.

La energía de este tipo de órbitas es nula. Esto implica que la velocidad de dichas órbitas viene dada por

$$\frac{v^2}{2} - \frac{GM_\odot}{r} = 0 \rightarrow v = \sqrt{\frac{GM_\odot}{r}}. \tag{2.2}$$

Con las condiciones iniciales que se ha dado a Oumumua necesitaríamos una velocidad en z de -16,17 km/s para conseguir una órbita parabólica. Simulando esto se obtiene la Figura 2.11.

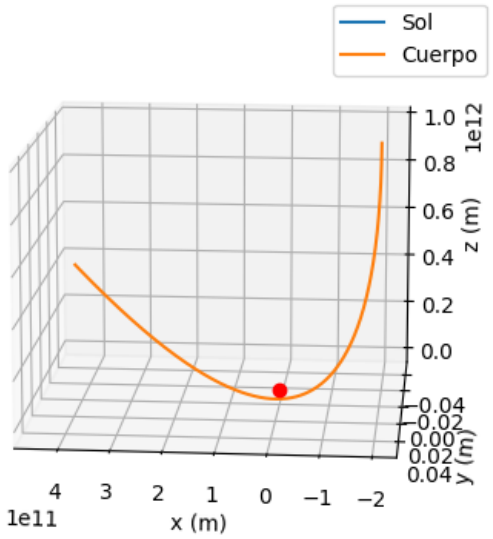

Figura 2.11: Resultado de simular una órbita parabólica de un cuerpo alrededor del Sol. El tiempo necesario para obtener este resultado fue de 46,38 segundos.

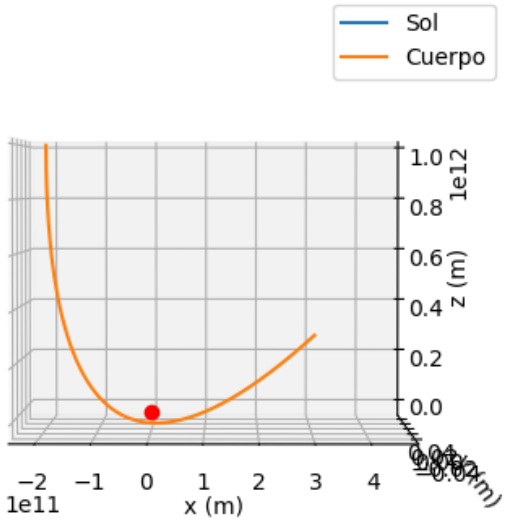

Figura 2.12: Resultado de simular una órbita parabólica de un cuerpo alrededor del Sol.

2.2. Resolución del problema de los 3 cuerpos

Como se mencionó anteriormente el problema de los tres cuerpos no tiene solución analítica y por ello se ha de recurrir a métodos numéricos.

En este casos se ha resuelto el problema de los tres cuerpos suponiendo el sistema Sol-Tierra y una sonda situada en L_2 a 1,5 millones de kilómetros de la Tierra.

Al resolver este problema desde un sistema de referencia inercial se obtienen los resultados que se pueden ver en la Figura 2.13 y 2.14.

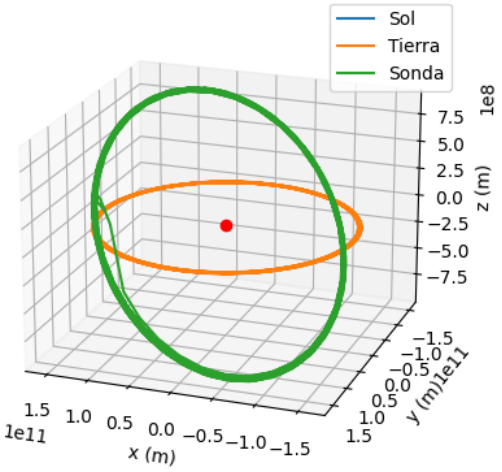

Figura 2.13: Resultado de resolver el problema de los tres cuerpos en sistema de referencia inercial. Las condiciones iniciales de la sonda fueron $\mathbf{r} = (151 \cdot 10^9, 0, 0)$ y $\mathbf{v} = (0, 29{,}9 \cdot 10^3, 165)$. Además se hicicieron 500 mil iteraciones con paso temporal de 4.0. Cada periodo orbital son 37 mil iteraciones por lo que se han simulado alrededor de 13 periodos.

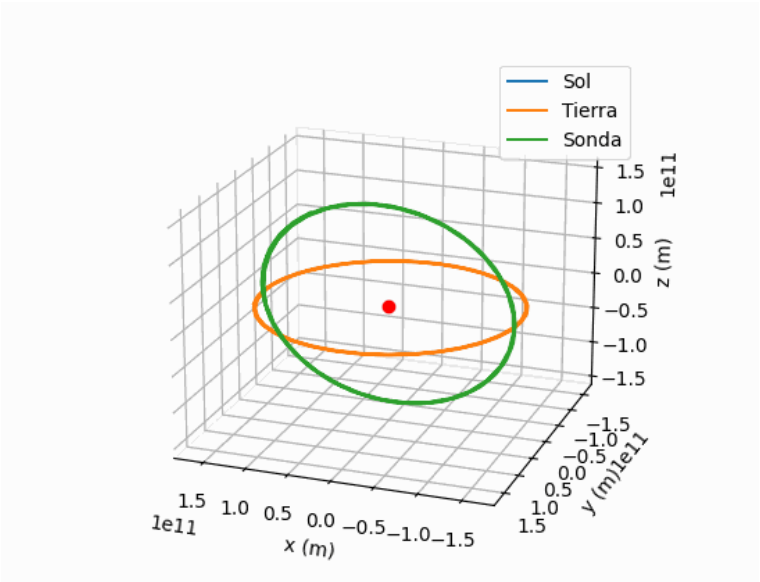

Figura 2.14: Resultado de resolver el problema de los tres cuerpos en sistema de referencia inercial. Las condiciones iniciales de la sonda fueron $\mathbf{r} = (151 \cdot 10^9, 837 \cdot 10^6, 0)$ y $\mathbf{v} = (0{,}5 \cdot 10^3, 29 \cdot 10^3)$. Además se hicicieron 250 mil iteraciones con paso temporal de 4.5. Cada periodo orbital son 17 mil iteraciones por lo que se han simulado cerca de 15 periodos.

2.2.1. Órbitas de halo y Lissajous

Al mismo tiempo que se realizaban las simulaciones arriba representadas se iba almacenando la posición angular de la Tierra. Con dichas posiciones se calculaba

una velocidad angular media que luego nos servía para hacer un cambio de sistema de referencia y pasar a uno que rotaba con la Tierra. De esta forma se podían obtener las órbitas de halo y Lissajous.

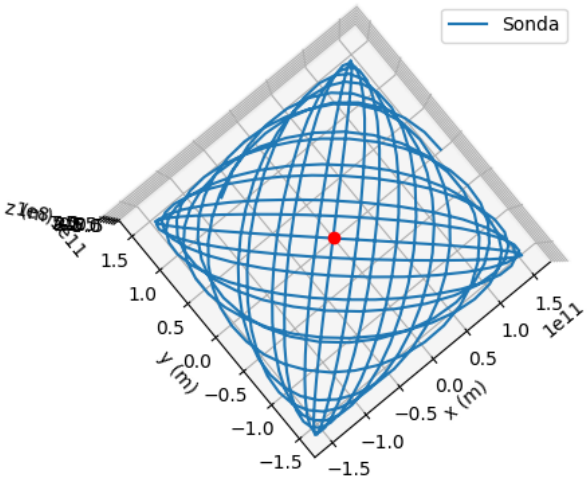

Figura 2.15: Órbita de Lissajous obtenida al hacer el cambio de coordenadas de la simulación de la Figura2.13. El tiempo de simulación fue de 238,08 segundos.

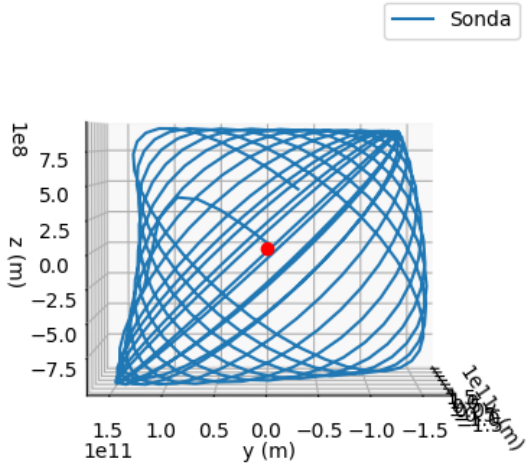

Figura 2.16: Órbita de Lissajous obtenida al hacer el cambio de coordenadas de la simulación de la Figura 2.13.

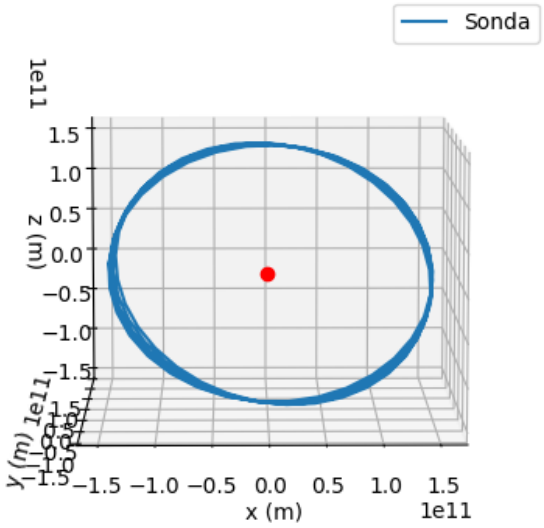

Figura 2.17: Órbita de Lissajous obtenida al hacer el cambio de coordenadas de la simulación de la Figura2.14. El tiempo de simulación fue de 155,90 segundos.

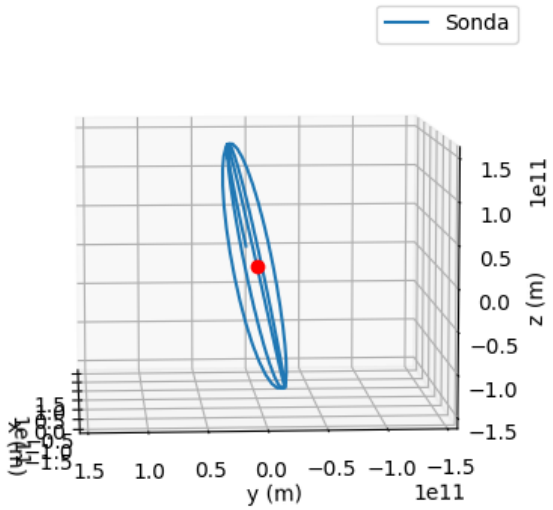

Figura 2.18: Órbita de Lissajous obtenida al hacer el cambio de coordenadas de la simulación de la Figura2.14.

Tal y como se puede comprobar lo que determina que se obtenga una u otra órbita son las condiciones iniciales. En el caso de la órbita de Lissajous la componente de la velocidad más grande es la componente en y mientras que para la órbita de Halo es la componente en z.

2.3. Simulación de la órbita de Ulysses

Otro cuerpo que pareció interesante simular fue la sonda Ulysses dado que tiene una órbita por encima de la eclíptica.

Para tomar esta órbita la sonda primero se impulsó con un *inertial upper stage* que hizo que la sonda tomara 15 km/s de velocidad y tras esto pasó a 378400 km de Júpiter de forma que la periapsis de su órbita se hallaba en dirección contraria al movimiento de Júpiter [9]. Esto provocó que la sonda se frenara y tomara una órbita polar alrededor del Sol tal y como se puede comprobar en [10].

Para simular adecuadamente esta órbita se considero que la sonda tenía una componente de la velocidad en y de 8,6 km/s y una componente en z de 12,29 km/s que en total son 15 km/s. Además se consideró que la posición inicial de Ulysses era $\mathbf{r} = (-378 \cdot 10^6, 778{,}5 \cdot 10^9, 0)$ de forma que se hallará en la misma posición en y que Júpiter pero en un punto en x tal que la periapsis estuviera en dirección opuesta al movimiento del planeta. Esto dio como resultado la simulación que se puede ver en la Figura 2.19.

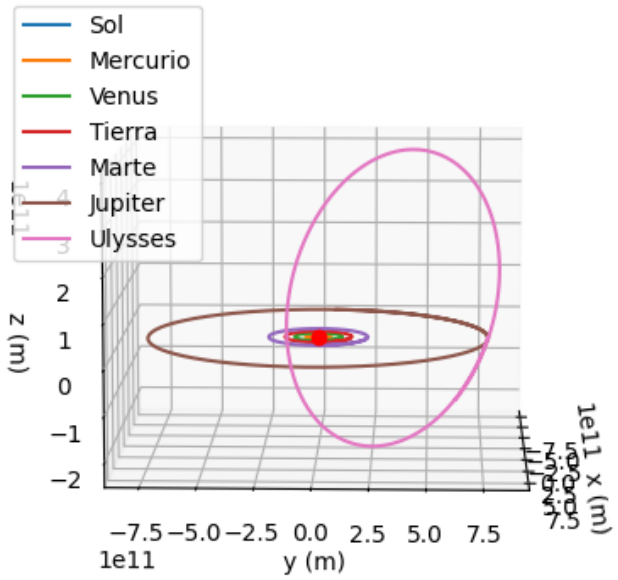

Figura 2.19: Resultado de simular la sonda Ulysses junto con el Sol y los 5 planetas más cercanos al mismo. El tiempo necesario para realizar esta simulación fue de 106,49 segundos.

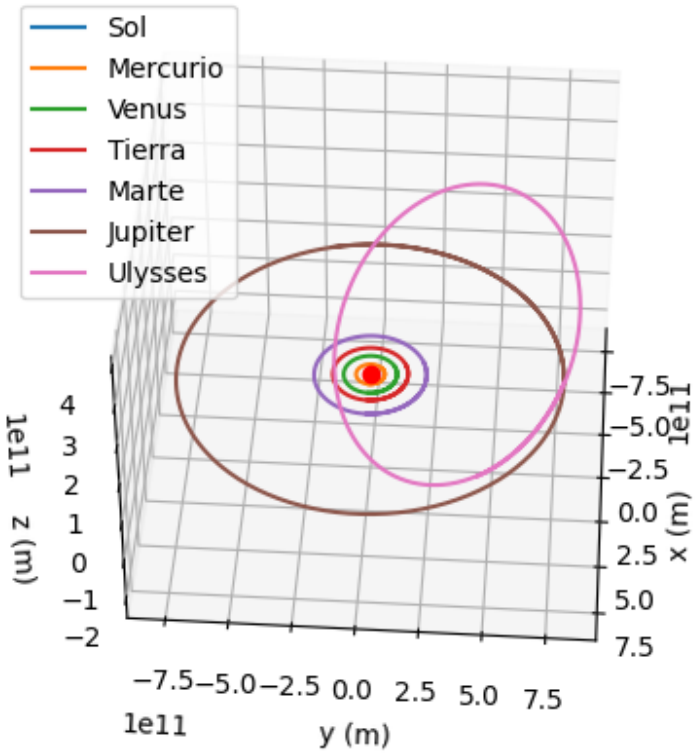

Figura 2.20: Resultado de simular la sonda Ulysses junto con el Sol y los 5 planetas más cercanos al mismo visto desde otro ángulo.

Tal y como se puede comprobar esta órbita es muy similar a la órbita que se ve en [10]. Por lo que la simulación es correcta. También se ha podido comprobar de esta forma que el programa es capaz de reproducir impulsos gravitatorios que unos cuerpos generan sobre otros.

2.4. Simulación masas iguales

Todas las simulaciones que se han presentado hasta ahora tenían una masa mucho mayor que el resto. Esto permitía situar el origen del sistema de referencia en dicha masa para obtener así un sistema de referencia, que aun no siendo el centro de masas real, reprodujera aproximadamente los resultados que se obtendrían en un sistema de referencia inercial.

En el presente caso se han situado 4 masas iguales en en las esquinas de un cuadrado de 600 km de largo con una velocidad de 1 km/s. El origen de coordenadas se ha situado en el centro del cuadrado de forma que, en este caso, coincida con el centro del masas del sistema. El resultado de dicha simulación se puede ver en el Figura 2.21.

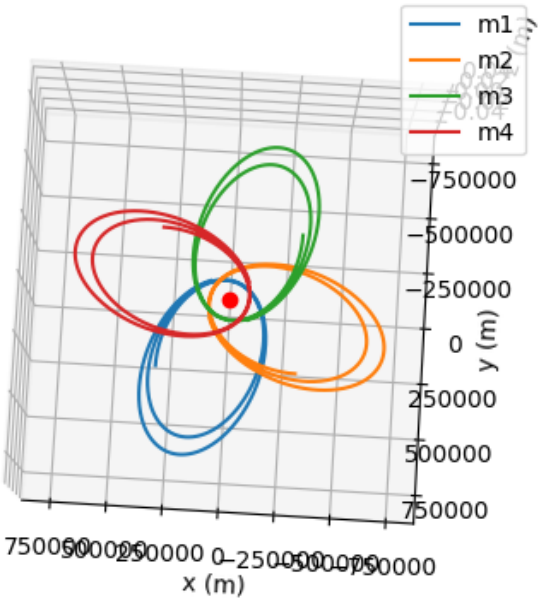

Figura 2.21: Resultado de simular 4 masas iguales sometidas a su interacción gravitatoria. El tiempo de simulación fue de 32.26 segundos.

Tal y como se puede comprobar el software sigue arrojando buenos resultados incluso aunque no haya una masa que sea mucho mayor que el resto.

Conclusiones

A lo largo de este trabajo se han repasado los conceptos más relevantes de la gravitación clásica y se han presentado algunos resultados que se pueden extraer de la misma. Tras constatar que para resolver sistemas más complejos que el de dos cuerpos es necesario recurrir a métodos numéricos, se ha explicado cómo se podría aplicar el método de Runge-Kutta de orden 4. Con dichas herramientas se ha podido construir un software que permite resolver el problema de N Cuerpos cuyo funcionamiento se ha explicado brevemente al final del primer capítulo.

El software ha permitido reproducir adecuadamente todas las órbitas que prevé el desarrollo teórico del problema de dos cuerpos. Además, en los casos elíptico e hiperbólico se han simulado los dos cuerpos empleados inicialmente junto con el resto del Sistema Solar para comprobar si las órbitas predichas se veían afectadas por la presencia del resto de masas. Tal y como se podido ver la magnitud de la masa del Sol hace que dichas órbitas no se vean apenas afectadas.

Tras esto, se ha conseguido simular con éxito el problema de los 3 cuerpos y comprobar que es posible representar las órbitas de halo y Lissajous alrededor del punto L_2 del sistema Sol-Tierra.

Por último, y para poner a prueba las capacidades del programa, se han simulado las órbitas de la sonda Ulysses y un sistema imaginario formado por 4 masas iguales. El primer caso ha permitido asegurar que el software simula los impulsos gravitatorios que en ocasiones se emplean en viajes interplanetarios. El segundo caso, por otro lado, ha servido como prueba de que el programa arroja buenos resultados aunque no haya una masa mucho mayor que el resto de cuerpos.

Estos buenos resultados están condicionados por el sistema de referencia. Siempre se podrán obtener buenos resultados si el sistema de referencia es inercial. En el caso del problema de N Cuerpos dicho sistema de referencia se halla en el centro de masas.

Por ello, una de las posibles mejoras que se le podría hacer al programa es añadir una función que calcule la posición del centro de masas. Con eso habría que recalcular la posición del resto de cuerpos respecto a dicho punto. De esta forma se obtendría la trayectoria de los cuerpos respecto al centro de masas real del sistema y no respecto al aproximado que es lo que se ha hecho en la mayoría de simulaciones.

En conclusión, los resultados permiten decir que se ha conseguido elaborar un

programa que calcula y representa la solución al problema de N Cuerpos, que era el objetivo del trabajo. A pesar de ello, se le podrían añadir algunas mejoras como una función que calcule las trayectorias respecto al centro de masas real del sistema o incluir otros métodos numéricos para la integración de las órbitas.

Bibliografía

[1] Taylor, John R. *Mecánica Clásica.* Reverté, 2013.

[2] Curtis, H. *Orbital Mechanics for Engineering Students.* Elsevier, 2005.

[3] Qian, Y., Xiaodong, Y., Wuxing, J. y Zhang, W. *An improved numerical method for constructing Halo/Lissajous orbits in a full solar system model.* Chinese Journal of Aeronautics, 2018.

[4] Veris de Iaco, A. *Practical astrodynamics.* Springer International Publishing, 2018.

[5] *https://cpb-us-e1.wpmucdn.com/sites.northwestern.edu/dist/2/77/files/2017/01/numerical_methods.compressed-1jnsi3e.pdf*

[6] Scherer, Philipp O.J. *Computational Physics.* Springer, 2013.

[7] Press, W.H., Teukolsky, S.A., Vetterling, W.T. y Flannery, B.P. *Numerical Recipes in C.* Cambridge University Press, 1992.

[8] *https://solarsystem.nasa.gov/asteroids-comets-and-meteors/comets/1p-halley/in-depth/*

[9] *https://solarsystem.nasa.gov/missions/ulysses/in-depth/*

[10] *https://sci.esa.int/web/ulysses/-/42904-orbit-of-ulysses*

Anexo 1: códigos

Código de la interfaz gráfica

```python
from tkinter import *
from tkinter import filedialog
from tkinter import ttk
import numpy as np
from n_cuerpos import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter
from time import *

class n_body_simulator:

    #We show the primary frame. Here the user must enter the number of bodies to be simulated

    def __init__(self, root):

        root.title("N-Body Simulator")

        firstframe=ttk.Frame(root, padding="25 25 25 25",borderwidth=2)
        firstframe.grid(column=0, row=0)

        ttk.Label(firstframe,text="Select simulation type").grid(column=0,row=0)
        simuSelectVar=StringVar()

        simuSelect=ttk.Combobox(firstframe,textvariable=simuSelectVar)


        simuSelect["values"]=("Standard simulation","Orbits around L2")
        simuSelect.state(["readonly"])
        simuSelect.grid(column=0,row=1)

        def simuBind(event):


            if event.widget.get()=="Standard simulation":

                thirdframe=ttk.Frame(root,padding="50 50 50 50")
                thirdframe.grid(column=1,row=0)

                self.num_cuerpos_str=StringVar()

                num_cuerpos_entry=ttk.Entry(thirdframe,width=15,textvariable=self.num_cuerpos_str)
                num_cuerpos_entry.grid(column=0,row=1)

                #This button runs n_body_data_entry method that allows the user to comfortably introduce the initial
                #conditions of the system
                ttk.Label(thirdframe, text="Enter number of bodies").grid(column=0,row=0)
                ttk.Button(thirdframe, text="Accept", command=self.n_body_data_entry).grid(column=0, row=2)

            if event.widget.get()=="Orbits around L2":
```

```
50
51
52                  thirdframe=ttk.Frame(root,padding="50 50 50 50")
53                  thirdframe.grid(column=1,row=0)
54
55                  ttk.Label(thirdframe, text="Number of
                    steps:").grid(column=0,row=0)
56                  l2IterStr=StringVar()
57                  l2IterEntry=ttk.Entry(thirdframe,
                    width=15,textvariable=l2IterStr)
58                  l2IterEntry.grid(column=1,row=0)
59
60                  ttk.Label(thirdframe, text="Step
                    size:").grid(column=0,row=1)
61                  l2StepStr=StringVar()
62
                    l2StepEntry=ttk.Entry(thirdframe,width=15,textvar
                    iable=l2StepStr)
63                  l2StepEntry.grid(column=1,row=1)
64
65                  ttk.Label(thirdframe,text="Orbit
                    type").grid(column=0,row=2)
66                  l2TypeStr=StringVar()
67
                    l2TypeEntry=ttk.Combobox(thirdframe,textvariable=
                    l2TypeStr,width=15)
68
69                  l2TypeEntry["values"]=("Lissajous","Halo")
70                  l2TypeEntry.state(["readonly"])
71                  l2TypeEntry.grid(column=1,row=2)
72
73
74
75                  ttk.Label(thirdframe,text="Simulation
                    name:").grid(column=0,row=3)
76                  l2SimNameStr=StringVar()
77
                    l2SimNameEntry=ttk.Entry(thirdframe,width=15,text
                    variable=l2SimNameStr)
78                  l2SimNameEntry.grid(column=1,row=3)
79
80                  def getl2Data():
81                      self.iterations=int(l2IterStr.get())
82                      self.l2Step=float(l2StepStr.get())
83                      self.l2Type=str(l2TypeStr.get())
84                      self.sim_name=str(l2SimNameStr.get())
85                      self.directory=filedialog.askdirectory()
86
87                      ttk.Button(thirdframe, text="Run",
                        command=self.l2OrbitSimulation).grid(column=2
                        ,row=4)
88
89
                    ttk.Button(thirdframe,text="Save",command=getl2Da
                    ta).grid(column=1,row=4)
90          simuSelect.bind("<<ComboboxSelected>>",simuBind)
91
92      def l2OrbitSimulation(self,*args):
93
94          startTime=time()
95
            self.BodyPosi,self.l2OrbitPosi=lissajousOrbit(self.iterat
```

```
            ions, self.l2Step, self.l2Type)
96          self.names=["Sol","Tierra","Sonda"]
97          self.num_cuerpos=3
98          self.draw_results(self.BodyPosi)
99
100         self.names=["Sonda"]
101         self.num_cuerpos=1
102         self.sim_name=self.sim_name+"_"+self.l2Type
103         self.draw_results(self.l2OrbitPosi)
104         endTime=time()
105
106         print("Tiempo de simulación:",endTime-startTime)
107     def n_body_data_entry(self,*args):
108         #This method will let the user enter both the mass and
109         #initial positions and velocities of the bodies
110
111         #We first define the functions that will check that the
            input has the proper format
112         def check_float(inputval):
113             try:
114                 float(inputval)
115
116             except:
117                 print("Invalid input:", inputval)
118             else:
119                 return True
120
121         def check_int(inputval):
122             try:
123                 int(inputval)
124
125             except:
126                 print("Invalid input:", inputval)
127             else:
128                 return True
129
130         #Now we define the rest of the method
131         name_entries=[]
132         masa_entries=[]
133         posi_entries=[]
134         f_time_entries=[]
135
136         self.num_cuerpos=int(self.num_cuerpos_str.get())
137
138         self.vec_ini=np.zeros((self.num_cuerpos*6))
139         self.masas=np.zeros(self.num_cuerpos)
140         self.names=np.empty(self.num_cuerpos,dtype=object)
141         self.f_time=np.zeros(self.num_cuerpos)
142
143         topSecondFrame=ttk.Frame(root,padding="75 75 75 75")
144         topSecondFrame.grid(column=2,row=0)
145         secondframe=ttk.Frame(topSecondFrame,padding="75 75 75
            75")
146         secondframe.grid(column=2, row=0)
147
148         secondframe.pack_forget()
149
150         #This blovk allows the user to name the bodies
151         ttk.Label(secondframe,text="Name").grid(column=0,row=0)
152         for i in range(self.num_cuerpos):
153             name_entry=ttk.Entry(secondframe,width=15)
154             name_entry.grid(column=0,row=1+i)
```

```
155             name_entries.append(name_entry)
156         #This block allows the user to enter the mass
157         ttk.Label(secondframe, text="Mass").grid(column=1,row=0)
158         for i in range(self.num_cuerpos):
159             masa_entry=ttk.Entry(secondframe, width=15,
                validate='focusout',validatecommand=(root.register(ch
                eck_float),'%P'))
160             masa_entry.grid(column=1,row=1+i)
161             masa_entries.append(masa_entry)
162
163
164
165
166         #This block is to introduce initial positions and
            velocities
167
168         ttk.Label(secondframe, text="x").grid(column=2,row=0)
169         ttk.Label(secondframe, text="y").grid(column=3,row=0)
170         ttk.Label(secondframe, text="z").grid(column=4,row=0)
171         ttk.Label(secondframe, text="Vx").grid(column=5,row=0)
172         ttk.Label(secondframe, text="Vy").grid(column=6,row=0)
173         ttk.Label(secondframe, text="Vz").grid(column=7,row=0)
174
175
176         for l in range(6):
177             for k in range(self.num_cuerpos):
178
179                 init_entry=ttk.Entry(secondframe, width=15,
                    validate='focusout',validatecommand=(root.registe
                    r(check_float),'%P'))
180                 init_entry.grid(column=2+l,row=1+k)
181                 posi_entries.append(init_entry)
182
183
184
            ttk.Label(secondframe,text="Stepsize").grid(column=8,row=
            0)
185         for i in range(self.num_cuerpos):
186
                f_time_entry=ttk.Entry(secondframe,width=15,validate=
                'focusout',validatecommand=(root.register(check_float
                ),'%P'))
187             f_time_entry.grid(column=8,row=1+i)
188             f_time_entries.append(f_time_entry)
189
190         #Initial and final time and iteration number
191         error_str=StringVar()
192         ttk.Label(secondframe,text="Error").grid(column=9,row=1)
193
            error_entry=ttk.Entry(secondframe,width=15,validate='focu
            sout',validatecommand=(root.register(check_float),'%P'),t
            extvariable=error_str)
194         error_entry.grid(column=10, row=1)
195
196         iter_entry_str=StringVar()
197
            ttk.Label(secondframe,text="Iterations").grid(column=9,ro
            w=2)
198
            iter_entry=ttk.Entry(secondframe,width=15,validate='focus
            out',validatecommand=(root.register(check_int),'%P'),text
            variable=iter_entry_str)
```

```
199             iter_entry.grid(column=10, row=2)
200
201             sim_name_str=StringVar()
202             ttk.Label(secondframe,text="Simulation
                name").grid(column=9,row=3)
203
                sim_name_entry=ttk.Entry(secondframe,width=15,textvariabl
                e=sim_name_str)
204             sim_name_entry.grid(column=10, row=3)
205
206             #We add two scrollbars just in case
207
208             ttk.Scrollbar(secondframe,orient=VERTICAL,command=l)
209
210             def get_init_data():
211
212                 #DESCRIPTION
213                 #'We save the entries in variables'
214                 for j in range(self.num_cuerpos*6):
215                     self.vec_ini[j]=float(posi_entries[j].get())
216
217                 for i in range(self.num_cuerpos):
218                     self.masas[i]=float(masa_entries[i].get())
219                     self.names[i]=name_entries[i].get()
220                     self.f_time[i]=float(f_time_entries[i].get())
221
222                 self.error=float(error_str.get())
223
224                 self.iterations=int(iter_entry_str.get())
225                 self.sim_name=str(sim_name_str.get())
226                 self.directory=filedialog.askdirectory()
227
228
                    ttk.Button(secondframe,text="Simulate",command=self.c
                    all_simulator).grid(column=8,row=self.num_cuerpos+1)
229
230             def importFile():
231                 fileName=filedialog.askopenfilename()
232
233
234                 with open(fileName) as file:
235                     lines=file.readlines()
236                     i=0
237
238                     for line in lines:
239
240
241                         line=line.split()
242                         if line==[]:
243                             break
244
245                         if i==0:
246                             error_entry.delete(0,'end')
247                             error_entry.insert(0,line[1])
248                         elif i==1:
249                             iter_entry.delete(0,'end')
250                             iter_entry.insert(0,line[1])
251                         elif i==2:
252                             sim_name_entry.delete(0,'end')
253                             sim_name_entry.insert(0,line[1])
254                         else:
255                             name_entries[i-3].delete(0,'end')
```

```
                        name_entries[i-3].insert(0,line[0])

                        masa_entries[i-3].delete(0,'end')
                        masa_entries[i-3].insert(0,line[1])

                        posi_entries[i-3].delete(0,'end')
                        posi_entries[i-3].insert(0,line[2])


                        posi_entries[i+self.num_cuerpos-3].delete
                        (0,'end')

                        posi_entries[i+self.num_cuerpos-3].insert
                        (0,line[3])


                        posi_entries[i+2*self.num_cuerpos-3].dele
                        te(0,'end')

                        posi_entries[i+2*self.num_cuerpos-3].inse
                        rt(0,line[4])


                        posi_entries[i+3*self.num_cuerpos-3].dele
                        te(0,'end')

                        posi_entries[i+3*self.num_cuerpos-3].inse
                        rt(0,line[5])


                        posi_entries[i+4*self.num_cuerpos-3].dele
                        te(0,'end')

                        posi_entries[i+4*self.num_cuerpos-3].inse
                        rt(0,line[6])


                        posi_entries[i+5*self.num_cuerpos-3].dele
                        te(0,'end')

                        posi_entries[i+5*self.num_cuerpos-3].inse
                        rt(0,line[7])

                        f_time_entries[i-3].delete(0,'end')
                        f_time_entries[i-3].insert(0,line[8])

                    i+=1



        ttk.Button(secondframe,text="Save",command=get_init_data)
        .grid(column=7,row=self.num_cuerpos+1)
        ttk.Button(secondframe,text="Import
        file",command=importFile).grid(column=7,row=self.num_cuer
        pos+2)

    def call_simulator(self,*args):

        startTime=time()
        self.rk4_simulation_results=orbita(self.error,
        self.f_time,self.iterations,self.vec_ini,self.num_cuerpos
        ,self.masas)
```

Código para el cálculo numérico

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter



def rcalc(x1,x2,y1,y2,z1,z2):
    return np.sqrt((x1-x2)**2+(y1-y2)**2+(z1-z2)**2)

def generador_matriz(n, masas, posiciones):

    mat_op=np.zeros((6*n,6*n))
    gamma=np.zeros(len(masas))

    #Definimos un vector con las gammas
    for i in range(len(masas)):
        gamma[i]=masas[i]*6.67e-11

    #Generamos la matriz que incluye los elementos de las leyes
    de movimiento
    mat_bloque=np.zeros((n,n))

    for k in range(n):
        for l in range(n):
            if (k-l)!=0:

                mat_bloque[k][l]=gamma[l]/rcalc(posiciones[k],pos
                iciones[l],posiciones[k+n],posiciones[l+n],
                posiciones[k+2*n],posiciones[l+2*n])**3

    for k in range(n):
        suma=0
        for l in range(n):

                suma+=-mat_bloque[k][l]
        mat_bloque[k][k]=suma


    #Generamos el primer bloque
    fila=0
    columna=0
    for i in range(int(3*n) , 3*n+n):

        for j in range(0, n):

            mat_op[i][j]=mat_bloque[fila][columna]
            columna+=1
        fila+=1
        columna=0

    #Generamos el segundo bloque
    fila=0
    columna=0
    for i in range(3*n+n, 3*n+2*n):

        for j in range(n, 2*n):

            mat_op[i][j]=mat_bloque[fila][columna]
            columna+=1
        fila+=1
```

```
59          columna=0
60
61      #Generamos el tercer bloque
62      fila=0
63      columna=0
64      for i in range(3*n+2*n, 3*n+3*n):
65
66          for j in range(2*n, 3*n):
67
68              mat_op[i][j]=mat_bloque[fila][columna]
69              columna+=1
70          fila+=1
71          columna=0
72
73      #Generamos la diagonal
74
75      for i in range(0, int(3*n)):
76          for j in range(int(n*3), 6*n):
77              if j-i==int(n*3):
78                  mat_op[i][j]=1
79
80
81      return mat_op
82
83  ##########################################################################
    #####
84  def rk4(mat_ini, vec_ini, paso):
85
86      k1=paso*(mat_ini.dot(vec_ini))
87      k2=paso*(mat_ini.dot(vec_ini)+paso*k1/2)
88      k3=paso*(mat_ini.dot(vec_ini)+paso*k2/2)
89      k4=paso*(mat_ini.dot(vec_ini)+k3*paso)
90
91      vec_fin=vec_ini+(1/6)*(k1+2*k2+2*k3+k4)
92
93      return vec_fin
94
95
96
97
98  def checkStep(mat_ini,vec_ini,paso,error):
99
100     #We calculate fn1 with two steps
101
102     fn=rk4(mat_ini,vec_ini,paso/2)
103     fn1=rk4(mat_ini,fn,paso/2)
104
105     #We calculate fn1 with one step
106
107     fn1_2=rk4(mat_ini,vec_ini,paso)
108
109     #We take a different stepsize for each body
110
111     #We check the errors and modify the stepsizes of each body
112     loopError=np.abs(fn1-fn1_2)
113
114     numCuerpos=len(vec_ini)//6
115     bodyLoopError=np.zeros(6)
116
117     bodyLocalError=np.zeros(6)
118
119     for i in range(numCuerpos):
```

```
120
121             bodyLoopError[0]=loopError[i]
122             bodyLoopError[1]=loopError[i+numCuerpos]
123             bodyLoopError[2]=loopError[i+2*numCuerpos]
124             bodyLoopError[3]=loopError[i+3*numCuerpos]
125             bodyLoopError[4]=loopError[i+4*numCuerpos]
126             bodyLoopError[5]=loopError[i+5*numCuerpos]
127
128             bodyLocalError[0]=error[i]
129             bodyLocalError[1]=error[i+numCuerpos]
130             bodyLocalError[2]=error[i+2*numCuerpos]
131             bodyLocalError[3]=error[i+3*numCuerpos]
132             bodyLocalError[4]=error[i+4*numCuerpos]
133             bodyLocalError[5]=error[i+5*numCuerpos]
134
135             for j in range(len(bodyLoopError)):
136                 if bodyLoopError[j]<1e-6:
137                     bodyLoopError[j]=1e-6
138                 if bodyLocalError[j]<1e-6:
139                     bodyLocalError[j]=1e-6
140
141             maxBodyError=np.max(bodyLoopError)
142             maxLocalError=np.max(bodyLocalError)
143
144             if maxBodyError>=maxLocalError:
145                 stepScale=np.abs(maxLocalError/maxBodyError)**0.25
146             elif maxBodyError<maxLocalError:
147                 stepScale=np.abs(maxLocalError/maxBodyError)**0.2
148
149             #Now we can scale the step of the body
150
151
152
153             paso[i]=paso[i]*stepScale
154             paso[numCuerpos+i]= paso[numCuerpos+i]*stepScale
155             paso[2*numCuerpos+i]=paso[2*numCuerpos+i]*stepScale
156             paso[3*numCuerpos+i]=paso[3*numCuerpos+i]*stepScale
157             paso[4*numCuerpos+i]=paso[4*numCuerpos+i]*stepScale
158             paso[5*numCuerpos+i]=paso[5*numCuerpos+i]*stepScale
159
160         return paso
161
162
163     def orbita(error,t_fin, iteraciones, vec_actual, n_cuerpos,
    masas):
164         posiciones=np.zeros((iteraciones+1, n_cuerpos*3))
165         velocidades=np.zeros((iteraciones+1,n_cuerpos*3))
166
167         #We assign the stepsizes
168         delta_t=np.zeros(n_cuerpos*6)
169         for i in range(n_cuerpos):
170
171             delta_t[i]=t_fin[i]
172             delta_t[n_cuerpos+i]=t_fin[i]
173             delta_t[2*n_cuerpos+i]=t_fin[i]
174
175             delta_t[3*n_cuerpos+i]=t_fin[i]
176             delta_t[4*n_cuerpos+i]=t_fin[i]
177             delta_t[5*n_cuerpos+i]=t_fin[i]
178
179         a=0
180         time=0
```

```
181
182         while a<=iteraciones:
183         #Almacenamos las posiciones
184
185
186             for k in range(0,n_cuerpos*3):
187                 posiciones[a][k]=vec_actual[k]
188                 velocidades[a][k]=vec_actual[k+int(n_cuerpos*3)]
189
190
191             mat_op=generador_matriz(n_cuerpos, masas=masas,
                posiciones=vec_actual[0:n_cuerpos*3])
192
193             localError=np.abs(error*vec_actual)
194             delta_t=checkStep(mat_op, vec_actual, delta_t,
                localError)
195
196             vec_actual=rk4(mat_op,vec_actual,delta_t)
197
198
199             a=a+1
200             print(a)
201             time+=delta_t
202
203
204         return posiciones
205
206
207     #ORBITAS DE LISSAJOUS/HALO
208
209
210     def lissajousOrbit(iteraciones,step, orbitType):
211         n_cuerpos=3
212
213         posiciones=np.zeros((iteraciones+1, n_cuerpos*3))
214         velocidades=np.zeros((iteraciones+1,n_cuerpos*3))
215
216         posiLissajous=np.zeros((iteraciones+1,3))
217
218
219         #To get the Lissajous/Halo orbit in the vicinity of a
            Lagrange point we must
220         #first solve the 3 body problem of the Sun-Earth-Probe
            system. Then
221         #we can ccompute Earth's orbital angular velocity with
            respect to the Sun
222         #and from there we can solve the equations of motion of the
            non-inertial
223         #reference frame that will give us the Lissajous/Halo orbit.
224
225         #First we initialize the vector with the bodies' positions
            and speed
226
227         if orbitType=="Halo":
228
229             #vec3BodyL2=np.array([0,384e6, 476.333e6, 0,0,0,
                0,0,-4.478e6, 0,0,37.02,0,1000,-1000+154,0,0,0])
230             vec3BodyL2=np.array([0,150e9,151.5e9, 0,0,837e6, 0,0,0,
                0,0,0, 0,30e3,5e3,0,0,29e3])
231         elif orbitType=="Lissajous":
232
233             vec3BodyL2=np.array([0,150e9,151.5e9, 0,0,0, 0,0,0,
```

```
            0,0,0, 0,30e3,29.9e3,0,0,0.165e3])
234
235
236         #vec3BodyL2=np.array([0,384e6, 464.183e6, 0,0,13.912e6,
            0,0,4.519e6, 0,0,57.83,0,1000,1018.18,0,0,16.3])
237
238
239     #masas=np.array([5.972e24,7.35e22, 6000 ])
240     masas=np.array([2e30,5.972e24,6000])
241     #This vector gives the initial conditions of our system. We
        now simulate the
242     #system the number of iterations given by the user.
243     delta_t=step
244
245
246     a=0
247
248     thetaTemp=0.0
249     thetaArray=np.zeros(iteraciones+1)
250     while a<=iteraciones:
251
252
253         for k in range(0,n_cuerpos*3):
254             posiciones[a][k]=vec3BodyL2[k]
255             velocidades[a][k]=vec3BodyL2[k+int(n_cuerpos*3)]
256
257         #localError=np.abs(error*vec3BodyL2)
258         #delta_t=checkStep(mat_op, vec3BodyL2, delta_t,
            localError)
259
260
261         mat_op=generador_matriz(3, masas, vec3BodyL2[:9])
262         #vec3BodyL2=rk4(mat_op,vec3BodyL2,delta_t)
263
264         #We introduce this vector in the R3BCP function
265
266         vec3BodyL2=rk4(mat_op,vec3BodyL2,delta_t)
267
268         #We calculate the current angular position between the
            IRF and the NIRF
269
270
            rMoon=np.array([vec3BodyL2[1],vec3BodyL2[4],vec3BodyL2[7]
            ])
271
272         if rMoon[0]>=0 and rMoon[1]>=0:
273             theta=np.arctan(rMoon[1]/rMoon[0])
274         elif rMoon[0]<=0 and rMoon[1]>=0:
275             theta=np.pi-np.arctan(-rMoon[1]/rMoon[0])
276         elif rMoon[0]<=0 and rMoon[1]<=0:
277             theta=np.arctan(rMoon[1]/rMoon[0])+np.pi
278         elif rMoon[0]>=0 and rMoon[1]<=0:
279             theta=2*np.pi-np.arctan(-rMoon[1]/rMoon[0])
280
281
282
283         thetaArray[a]=(theta-thetaTemp)/delta_t
284         print(thetaArray)
285
286         thetaTemp=theta
287         a+=1
288         print(a)
```

Anexo 2: archivos .txt de para el input

Órbita circular

```
Error 1e-4
Iterations 100000
Name orbitaCircular
Sol 2e30 0 0 0 0 0 0 1.0
Cuerpo 5e22      75e9 0 0 0 4.217e4 0 1.0
```

Halley-Sol

```
Error 1e-4
Iterations 250000
Name orbitaEliptica
Sol 2e30 0 0 0 0 0 0 1.0
Halley 2.2e14 -87.8e9 0 0 0 -54.55e3 0 1.0
```

Halley-Sistema Solar

```
Error 1e-4
Iterations 100000
Name halleySolarSystem
Sol 2e30 0 0 0 0 0 0 1.0
Halley 2.2e14 -87.8e9 0 0 0 -54.55e3 0 1.0
Mercurio 3.285e23 58e9 0 0 0 47.85e3 0 2.0
Venus 4.867e24 108.2e9 0 0 0 35e3 0 2.0
Tierra 5.972e24 150e9 0 0 0 30e3 0 2.0
Marte 6.39e23 227.9e9 0 0 0 24.1e3 0 2.0
Jupiter 1.898e27 0 778.5e9 0 -13.1e3 0 0 5.0
Saturno 5.683e26 -1434e9 0 0 0 -9.67e3 0 5.0
Urano 8.681e25 2871e9 0 0 0 6.81e3 0 10.0
Neptuno 1.024e26 -4495e9 0 0 0 -5.477e3 0 10.0
Pluton 1.25e22 6984e9 0 2135e9 0 4.7e3 0 10.0
```

Oumuamua-Sol

```
Error 1e-4
Iterations 50000
Name orbitaHiperbolica
Sol 2e30 0 0 0 0 0 0 1.0
Oumuamua 4e4 -200e9 0 1500e9 0 0 -38.3e3 1.0
```

Oumuamua-Sistema Solar

```
Error 1e-4
Iterations 100000
Name oumuamuaSolarSystem
Sol 2e30 0 0 0 0 0 0 1.0
Oumuamua 4e4 -200e9 0 1500e9 0 0 -29.5e3 1.0
Mercurio 3.285e23 58e9 0 0 0 47.85e3 0 2.0
Venus 4.867e24 108.2e9 0 0 0 35e3 0 2.0
Tierra 5.972e24 150e9 0 0 0 30e3 0 2.0
Marte 6.39e23 227.9e9 0 0 0 24.1e3 0 2.0
Jupiter 1.898e27 0 778.5e9 0 -13.1e3 0 0 5.0
Saturno 5.683e26 -1434e9 0 0 0 -9.67e3 0 5.0
Urano 8.681e25 2871e9 0 0 0 6.81e3 0 10.0
Neptuno 1.024e26 -4495e9 0 0 0 -5.477e3 0 10.0
Pluton 1.25e22 6984e9 0 2135e9 0 4.7e3 0 10.0
```

Órbita parabólica

```
Error 1e-4
Iterations 85000
Name orbitaParabolica
Sol 2e30 0 0 0 0 0 0 1.0
Cuerpo 4e4 -200e9 0 1000e9 0 0 -16.17e3 1.0
```

Ulysses

```
Error 1e-4
Iterations 100000
Name ulysses
Sol 2e30 0 0 0 0 0 0 1.0
Mercurio 3.285e23 58e9 0 0 0 47.85e3 0 2.0
Venus 4.867e24 108.2e9 0 0 0 35e3 0 2.0
Tierra 5.972e24 150e9 0 0 0 30e3 0 2.0
Marte 6.39e23 227.9e9 0 0 0 24.1e3 0 2.0
Jupiter 1.898e27 0 778.5e9 0 -13.1e3 0 0 5.0
Ulysses 367 -378.4e6 778.5e9 0 0 8.6e3 12.29e3 5.0
```

Masas iguales

```
Error 1e-4
Iterations 10000
Name masas
m1 1e22 300e3 300e3 0 0 -1e3 0 1.0
m2 1e22 -300e3 300e3 0 1e3 0 0 1.0
m3 1e22 -300e3 -300e3 0 0 1e3 0 1.0
m4 1e22 300e3 -300e3 0 -1e3 0 0 1.0
```