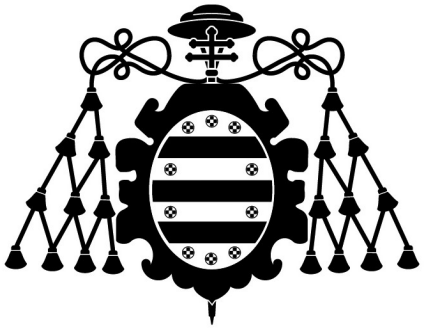
Universidad De Oviedo

Sistema planetario

Introducción a la Física Computacional



PL4
Miguel Durán Vera
David Villa Blanco

6 de mayo de 2021

Índice

1. Resumen

En este informe se explicará la resolución del problema de Sistema Planetario, cuyo objetivo era obtener las trayectorias de los principales cuerpos del Sistema Solar.

En la resolución se tuvo que aplicar la Ley de Gravitación Universal y resolver un sistema de ecuaciones diferenciales con valor inicial mediante el método numérico Runge-Kutta.

2. Objetivos

Los objetivos del problema del sistema planetario consistían en:

- Representar las trayectorias de la Tierra y del Sol, teniendo sólo a éstos en cuenta.
- Añadir a la representación anterior más planetas y observar los cambios.
- Realizar una simulación dinámica de su movimiento.

3. Descripción física del problema

Para calcular las órbitas de los distintos planetas del sistema solar es necesario usar las leyes de la gravitación de Newton y sobretodo la ecuación 1, la cual nos dice la fuerza que un cuerpo de masa M_1 ejerce sobre otro de masa M_2 siendo G la constante de gravitación universal y $\vec{r}_{12}$ el vector que va desde el segundo objeto hasta el primero.

$$\vec{F}_{12} = G \cdot \frac{M_1 M_2}{r_{12}^3} \cdot \vec{r}_{12} \tag{1}$$

Para calcular las distintas órbitas hay que tener en cuenta las fuerzas que ejercen todos los planetas sobre los otros y que éstos a su vez se van moviendo a lo largo del tiempo.

Utilizando la segunda ley de Newton (ecuación 2) para el cuerpo 1 llegamos a la definición de su aceleración en la ecuación 3.

$$\vec{F}_1 = M_1 \cdot \vec{a}_1 \tag{2}$$

$$\vec{a}_1 = G \cdot \frac{M_2}{r_{12}^3} \cdot \vec{r}_{12} \tag{3}$$

4. Descripción matemática del problema

Para llevar a cabo el cálculo de las trayectorias de los planetas a lo largo del tiempo debemos integrar las ecuaciones diferenciales que las describen, debido a que poseemos la información de como estas varían con el tiempo (ecuaciones diferenciales) pero no la definición de la función que depende del tiempo y describe el propio movimiento.

En nuestro caso tenemos las siguientes ecuaciones diferenciales:

$$\frac{d\vec{v}_1}{dt} = \vec{a}_1 = G \cdot \frac{M_2}{r_{12}^3} \cdot \vec{r}_{12} \tag{4}$$

$$\frac{d\vec{x}_1}{dt} = \vec{v}_1 \tag{5}$$

Por lo que necesitamos un método por el cual, a partir de las ecuaciones diferenciales, obtengamos una función que dependa del tiempo y nos describa el movimiento.

5. Métodos numéricos

Para resolver el problema planteado anteriormente se utilizará en python la función integrada en el módulo scipy.integrate solve_ivp con el método RK23, el cuál hace referencia a un método Runge-Kutta cuya precisión se toma como si fuese de orden 2 pero cuyos cálculos son realizados mediante un método de tercer orden.

Nuestro objetivo es, a partir de la función diferencial y de un valor inicial conocido, obtener tanto la función de la velocidad como la de la posición en función del tiempo.

En las clases de teoría se nos introdujo en primera instancia el método de Euler para la resolución de los SEDO (Sistemas de ecuaciones diferenciales ordinarias), el cual, partiendo de una función $F(t) = y$ dependiente de una variable, t, y conociéndose su ecuación diferencial descrita en la ecuación 6, permite aproximar la recta $F(t)$ por la dada en la ecuación 7, y con $(t_1 - t_0) = h$ el paso de cada iteración, podemos expresar y_1 tal y como se describe en la ecuación 8.

$$\frac{dy}{dt} = f(t_0, y_0) \tag{6}$$

$$\frac{(y_1 - y_0)}{(t_1 - t_0)} = f(t_0, y_0) \tag{7}$$

$$y_1 = y_0 + h \cdot f(t_0, y_0) \tag{8}$$

A partir de la ecuación 8, podemos seguir iterando en la misma introduciendo en los valores iniciales los calculados en la iteración anterior para calcular el valor de la próxima iteración. Esto nos dejaría con la ecuación 9.

$$y_n = y_{n-1} + h \cdot f(t_{n-1}, y_{n-1}) \tag{9}$$

Este método de Euler es en realidad el más simple de los métodos de Runge-Kutta. El método de Runge-Kutta de orden s en su forma más general sigue la expresión descrita en la ecuación 10,

$$y_n = y_{n-1} + h \sum_{i=1}^{s} b_i k_i \tag{10}$$

donde cada coeficiente k_i viene descrito por la ecuación 11,

$$k_i = f(t_n + hc_i, y_n + h \sum_{j=1}^{s} a_{ij} k_j) \quad i = 1, ..., s \tag{11}$$

y a_{ij}, b_i y c_i son coeficientes que dependerán del orden del método.

Un ejemplo de los métodos de Runge-Kutta es el método de orden 4 o RK4, el cual es uno de los métodos de Runge-Kutta más utilizado. Con los valores de los coeficientes a_{ij}, b_i y c_i ya especificados, la ecuación general que describe el método viene dada por la ecuación 12,

$$y_n = y_{n-1} + \frac{1}{6} h(k_1 + 2k_2 + 2k_3 + k_4) \tag{12}$$

donde k_1, k_2, k_3 y k_4, vienen determinandos por las ecuaciones 13, 14, 15 y 16.

$$k_1 = f(t_{n-1}, y_{n-1}) \tag{13}$$

$$k_2 = f(t_{n-1} + \frac{1}{2}h, y_{n-1} + \frac{1}{2}k_1 h) \tag{14}$$

$$k_3 = f(t_{n-1} + \frac{1}{2}h, y_{n-1} + \frac{1}{2}k_2 h) \tag{15}$$

$$k_4 = f(t_{n-1} + h, y_{n-1} + k_3 h) \tag{16}$$

Para la resolución del trabajo, utilizaremos el método Runge-Kutta 3(2), que consiste en un método Runge-Kutta explícito (ya que para calcular k_i sólo se requiere conocer los k_j con $j < i$ que ya hemos evaluado con anterioridad) que controla el error asumiendo una precisión dada por el método de segundo orden pero tomando los pasos mediante las fórmulas del método de tercer orden.

6. Python

6.1. Astropy

Como se explicó anteriormente, para llevar a cabo el método explicado necesitamos unos datos iniciales que nos indiquen la posición y velocidad de todos los planetas al comienzo del programa. Para obtener estos datos se utilizó el módulo *astropy*, el cuál incluye funciones y constantes que facilitan la programación en python de problemas relacionados con la astronomía.

En nuestro caso decidimos utilizar la función 'get_body_barycentric_posvel', la cual nos permite obtener la posición y velocidad de los distintos planetas tomando como eje de coordenadas el baricentro del sistema solar, el cual hace referencia al centro de masa de todos los objetos del sistema solar y se trata del punto alrededor del cuál los planetas orbitan. Debido a que el Sol contiene más del 99 % de toda la masa del sistema solar, este punto se suele encontrar muy cercano a nuestra estrella.

Esta función admite dos parámetros: el objeto del cuál queremos obtener la posición y la velocidad y el momento que nos interesa. Para establecer el primero de estos utilizamos un bucle 'for' que vaya recorriendo una lista 'planets' en la que están todos los planetas que queremos. Y para determinar el tiempo se utiliza el módulo *datetime*, el cuál nos permite obtener la fecha del momento en el que se está utilizando el programa, y está fecha se pasa mediante la función 'Time' de *astropy* al formato que requiere 'get_body_barycentric_posvel'.

Además esta función nos permite cambiar el método por el cuál calcula los datos que nos interesan, y el cuál se denomina efeméride. En nuestro caso se decidió utilizar el método por defecto que da el módulo.

Finalmente los datos se guardan en un 'array' cuya tamaño es de n filas (n=número de cuerpos, en nuestro caso 9 ya que se trata de ocho planetas y el Sol) y 6 columnas (3 para el vector posición (x,y,z) y 3 para el vector velocidad (v_x, v_y, v_z)).

Además hay que tener en cuenta que la función utilizada devuelve los datos en UA (distancia Tierra-Sol) y UA/día, por lo que hay que pasarlo a unidades del S.I. (m y m/s). Para esto se utilizó la función 'to' del módulo *astropy* que permite realizar cambios de unidades.

6.2. Datos iniciales

Además de la información obtenida mediante *astropy* necesitamos más datos entre los que se encuentran las masas de todos los cuerpos en kilogramos y el tiempo que tardan todos los planetas en dar una vuelta completa al Sol en segundos (ya que estas son las unidades del SI).

Para los casos en los que solo se necesitan los datos de ciertos planetas se crean otros 'arrays' a partir de los completos donde estén solo la información de los que nos interesan.

6.3. Funciones definidas

Para realizar todos los cálculos hizo falta definir distintas funciones que facilitasen la programación. Estas las podemos dividir en aquellas que nos permiten realizar los cálculos y en aquellas que tienen como objetivo cambiar el formato para ajustarse a lo necesitado.

6.3.1. Funciones para el cálculo

- r: Esta función tiene como parámetros los valores xyz de la posición de dos cuerpos y nos calcula a través de ellos el vector que los une en forma de 'array'.
- modulo: Esta función tiene como parámetro un vector 'r' con la misma forma que el obtenido con la función anterior y nos devuelve el módulo de este. Por lo que si vemos a 'r' como el vector que une dos cuerpos, esta función nos devuelve la distancia entre ambos.
- acel: Esta función tiene como parámetros un vector 'r' y una masa 'm' y nos permite calcular la aceleración que sufre un cuerpo debido a otro con la ecuación 3, siendo M_2 la masa introducida y r_{12} el modulo del vector 'r' obtenido con la función anterior.
- diffPlanets: Esta función puede cambiar de nombre entre los distintos archivos *.py* utilizados pero en todos tiene el mismo objetivo: calcular las diferenciales en los distintos momentos.

  Para llevar a cabo esto es necesario introducir un 'array' que incluirán la velocidad y la posición de los distintos cuerpos en una fila y que posteriormente se pasará a una forma con 6 columnas y tantas filas como cuerpos nos interesen. Además se creará otro 'array' con la misma forma que el último en el que se irán metiendo los datos obtenidos y se llamará 'diff'.

  Después mediante un bucle 'for' se irá pasando por los distintos planetas y obteniendo sus datos de posición y velocidad. Además, como nos sugiere la ecuación 5, igualaremos la diferencial de la posición al dato de la velocidad que tenemos.

  Para calcular la diferencial de la velocidad habrá que calcular la aceleración que está sufriendo ese cuerpo. Para esto crearemos otro bucle 'for' que vaya pasando por los otros planetas que no sean del que estamos calculando la aceleración y de los cuales obtendremos los datos de posición y masa. A partir de esto crearemos un vector 'R' que nos una ambos cuerpos haciendo uso de la función 'r' definida anteriormente y con el cual, junto al valor de la masa, obtendremos la aceleración mediante la función 'acel'. Finalmente iremos sumando estos valores a la diferencial de la velocidad.

  Con todas las diferenciales obtenidas de este cuerpo las meteremos en la fila correspondiente de 'diff', de manera que al final tengamos un 'array' donde estén todas las diferenciales de todos los planetas, siendo cada columna una diferencial y cada fila un planeta.

  Esta función 'diffPlanets' nos devolverá un array fila con los datos de diff, debido a que este es el formato que requiere la función 'solve_ivp'.

6.3.2. Funciones de formato

- matrizafila: Esta función tiene como parámetro un array de dos dimensiones y nos permite pasarlo a uno con solamente una fila, para lo cual se utiliza la función de *numpy* 'reshape'. Esta función se utilizará para poder utilizar 'solve_ivp', ya que esta no admite 'arrays' multidimensionales.

- filaamatriz: Esta función funciona igual que 'matrizafila' pero al revés, ya que partiendo de un 'array' fila nos devuelve otro con forma nx6, donde n es el tamaño del 'array' introducido entre 6, ya que este se corresponde con lo que ocupan los vectores posición y velocidad. 'n' en nuestro caso se correspondería con el número de cuerpos que se están estudiando.

- matrizacubo: Como la funciones anteriores, esta nos sirve para modificar el formato de un 'array'. En este caso parte de un 'array' bidimensional con forma $(n \cdot 6)\times$t y lo pasa a uno tridimensional con forma $n \times 6 \times t$. 'n' se corresponde con lo mismo a lo explicado en 'filaamatriz' y t con los distintos tiempos donde se está estudiando el movimiento.

- funder: Esta función nos sirve para hacer que la función 'diffPlanets' se pueda utilizar al llamar a solve_ivp. Tiene como parámetros a 't' y a un 'array' 'p'. Lo que hace es llamar a la función 'diffPlanets' metiendo como parámetro a 'p' y acaba devolviendo el valor que nos de ésta.

6.4. solve_ivp

Esta función del módulo *scipy.integrate* permite obtener una función dependiente del tiempo dandole su diferencial y un valor inicial (Initial Value Problem). Esta función tiene muchos parámetros distintos pero los que a nosotros nos interesan son los siguientes:

- fun: es la función que obtiene las distintas diferenciales y que en nuestro caso se trata de 'funder'. Esta no puede devolver un 'array' multidimensional por lo que fue necesario que la nuestra devolviese todos los datos en una fila.

- t_span: consiste en una tupla con dos valores, los cuales se tratan del comienzo y el final del intervalo temporal que nos interesa. En nuestro caso siempre comienza en 0 y el final varía según los planetas que queramos representar, intentando que siempre sea aproximadamente los segundos que tarda el planeta más alejado en dar una vuelta. Como la mayoría de casos es Neptuno, el tiempo final es $6 \cdot 10^9$.

- y0: este parámetro se corresponde con los valores iniciales que se van a dar y que en nuestro caso son obtenidos a partir de los datos de astropy. Estos valores deben de estar en un 'array' unidimensional, por lo que usaremos la función matrizafila.

- method: hace referencia al método que queremos que use para hacer los cálculos. En nuestro caso elegimos el método 'RK23', es decir, un Runge-Kutte de orden 3(2), debido a que nos

interesa que haga muchos cálculos y no tanto que estos tengan mucha precisión. Sin embargo, también se puede utilizar el método 'RK45', el cual es de orden 5(4), pero solamente para los casos con pocos planetas.

- t_eval: este parámetro nos sirve para especificar los distintos tiempos donde queremos que nos evalue los datos. En nuestro caso utilizamos la función 'linspace' la cual nos devuelve un 'array' con nin valores entre t_{in} y t_{fin} o la función 'arange' que devuelve un 'array' con valores entre t_{in} y t_{fin} y con intervalo in.

Finalmente nos devuelve distintos campos y de los cuales cogemos solamente '.y', en el cual están los resultados que nos interesan. Este campo se trata de un 'array' bidimensional, el cuál pasamos a tridimensional con nuestra función 'matrizacubo' con la que acabamos con un 'array' llamado 'trayectorias' y con forma $n \times 6 \times nin$, donde 'n' es el número de planetas y 'nin' el número de tiempos donde solve_ivp ha estudiado la función.

6.5. Creación de la representación

A lo largo del trabajo se realizaron diferentes representaciones de los resultados obtenidos. A pesar de que para cada una se tuvo que seguir un procedimiento distinto, todas tienen en común un bucle ´for´que va recorriendo los datos de todos los cuerpos obtenidos por solve_ivp y que han sido guardados en el 'array' 'trayectorias'. En cada interacción del bucle se obtienen los datos de posición del cuerpo correspondiente y con ellos se realiza la representación deseada.

7. Casos concretos

7.1. Representaciones 2D

Para la representación en 2D se utilizó la función 'plot' del módulo *matplotlib* para representar los datos de x frente a los de y de cada cuerpo. Además se creó una lista con 9 colores diferentes para así poder diferenciar la trayectoria de cada cuerpo. También se mostró la posición final de todos los cuerpos cogiendo el último dato que tenemos de la posición y marcando ese punto con una circunferencia.

7.1.1. Representando una vuelta de cada planeta

Además, para el caso en el que solo se representa un año de cada planeta, en vez de coger todos los datos de posición del 'array' trayectorias solo se cogen aquellos que se corresponden con la primera vuelta de cada planeta. Esto se realiza utilizando el tiempo que tarda cada uno en orbitar.

7.2. Representaciones 3D

En este caso se utiliza la herramienta del módulo *matplotlib* 'mpl_toolkits.mplot3d' con la función 'Axes3D', la cuál nos permite añadir una tercera dimensión a nuestras representaciones. En este caso cogemos también los datos de posición 'z' de cada cuerpo y con la función 'plot' mostramos la posición en 'x', en 'y' y en 'z'.

7.3. Simulación dinámica en *vpython*

Para crear la simulación en vpython lo primero que se hizo fue crear una esfera para cada cuerpo en su posición inicial correspondiente y cuyo tamaño depende del mismo, siendo los más grandes Júpiter y Saturno y los más pequeños Mercurio y Marte. A pesar de intentar mantener un poco la escala con los planetas, el Sol es más pequeño a lo que le tocaría en relación a los planetas ya que sino taparía la órbita de Mercurio, debido a que las distancias se reducieron con el fin de que se pudiesen ver bien las representaciones de los cuerpos.

Para llevar a cabo el movimiento lo que se hizo fue iniciar un bucle 'for' el cual en cada interacción actualice la posición de todos los cuerpos a la siguiente.

Además para añadir más realismo a la simulación se puso texturas a cada planeta que representan sus superficies y a Saturno y Urano se les añadió discos rodeandolos los cuales representan a sus sistemas de anillos.

8. Resultados

8.1. Sistema Tierra-Sol

Lo primero que se realizó fue un sistema donde solo se tiene en cuenta al Sol y la Tierra y cuyo resultado se puede ver en la Figura 1. En esta podemos comprobar que el movimiento que sigue la Tierra se asemeja bastante a una elipse.

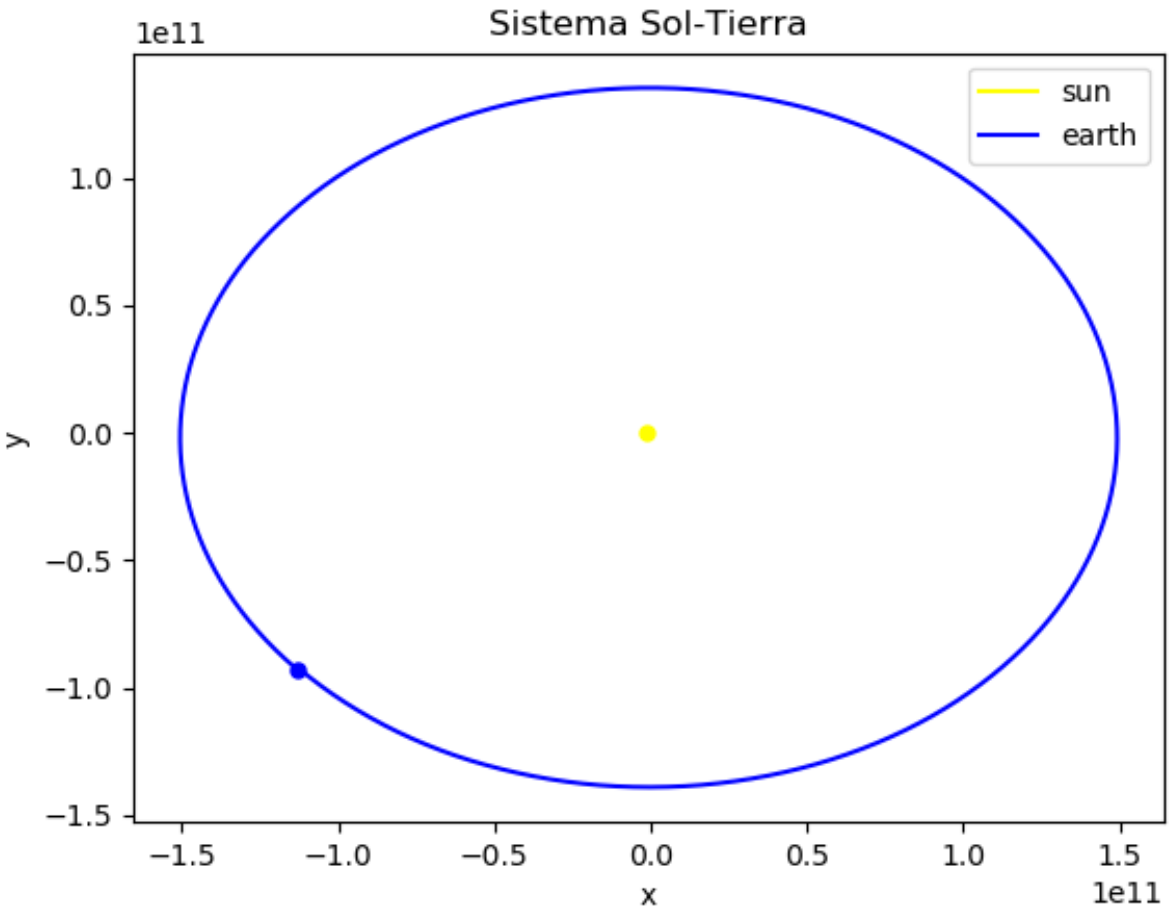


Figura 1: Sistema donde solo se tiene en cuenta al Sol y la Tierra.

8.2. Añadimos otro planeta

Después lo que se realizó fue representar el sistema anterior pero donde se añadía otro planeta, para poder ver como este influía en el movimiento de la Tierra. En la figura 2 se puede ver este sistema donde se añadió a Júpiter.

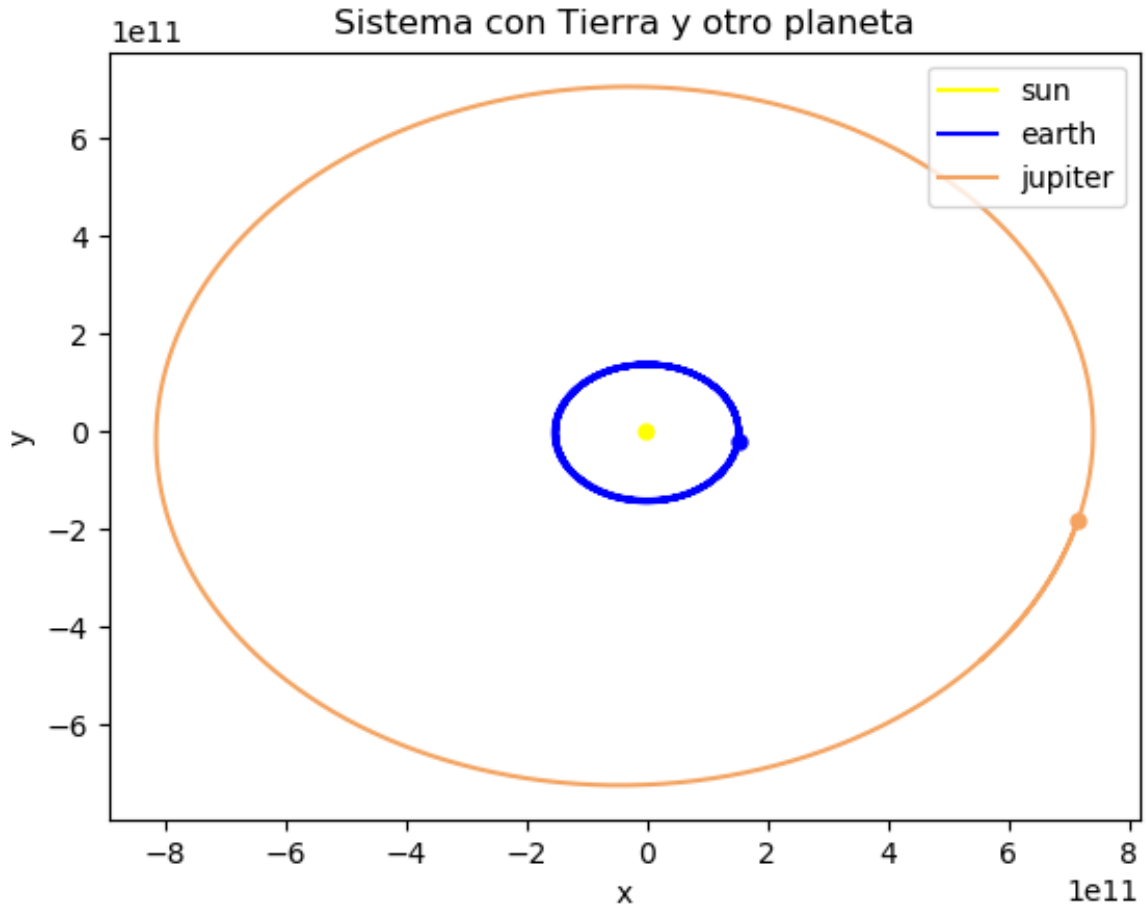


Figura 2: Sistema donde se añade a Júpiter.

8.3. Sistema Solar

El siguiente paso que se llevó a cabo fue añadir ya todos los planetas del sistema solar, tal y como se puede ver en la Figura 3. En esta podemos notar la acumulación del error del método utilizado en los planetas pequeños debido a que estamos representando todas sus órbitas en el período de un año en Neptuno, en el que, por ejemplo, Mercurio da aproximadamente 687 vueltas y la Tierra 165.

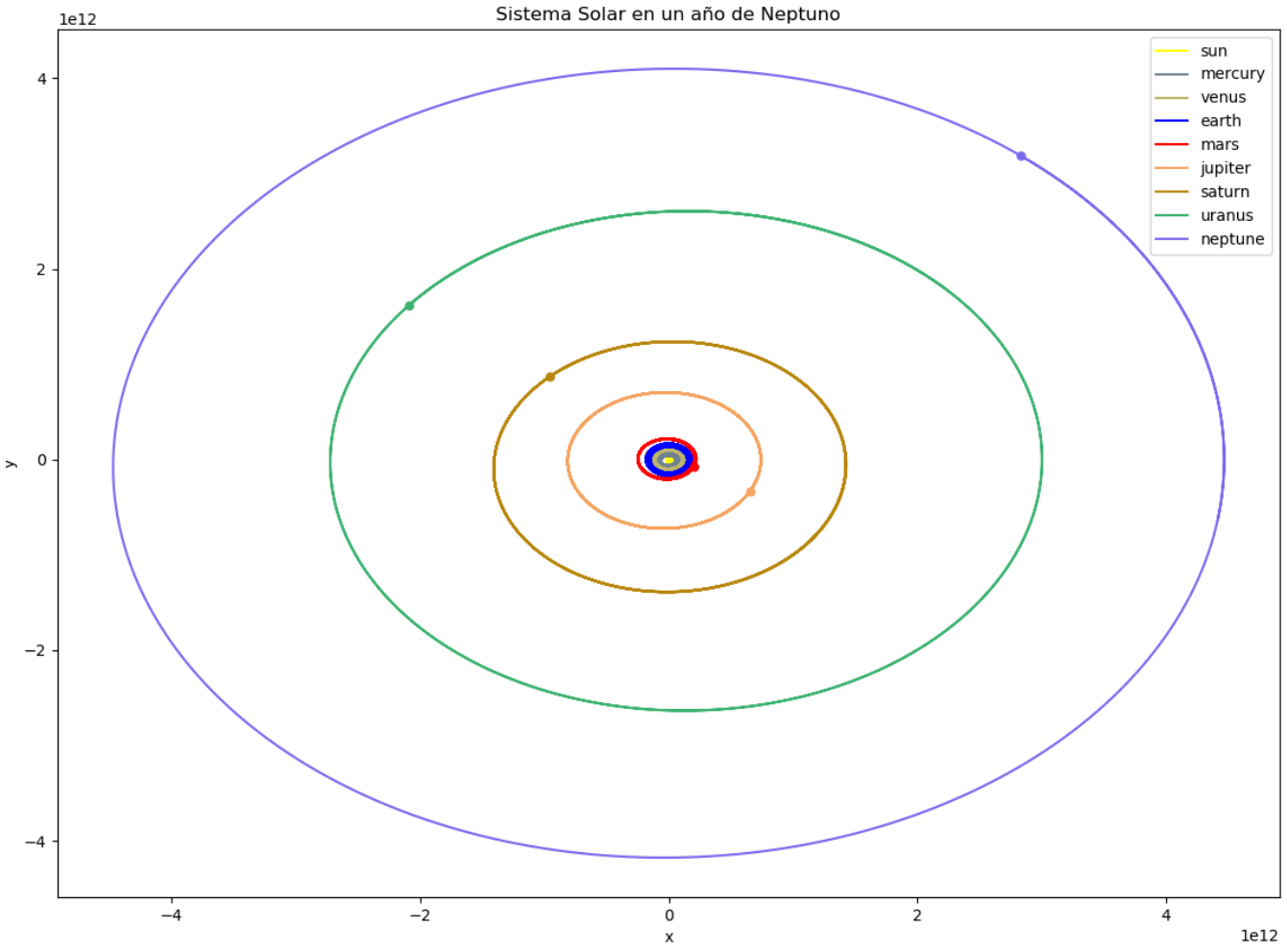


Figura 3: Sistema Solar a lo largo de un año de Neptuno.

Para poder ver mejor lo que sería la trayectoria de los planetas se representó en la figura 4 solamente una órbita en cada planeta.

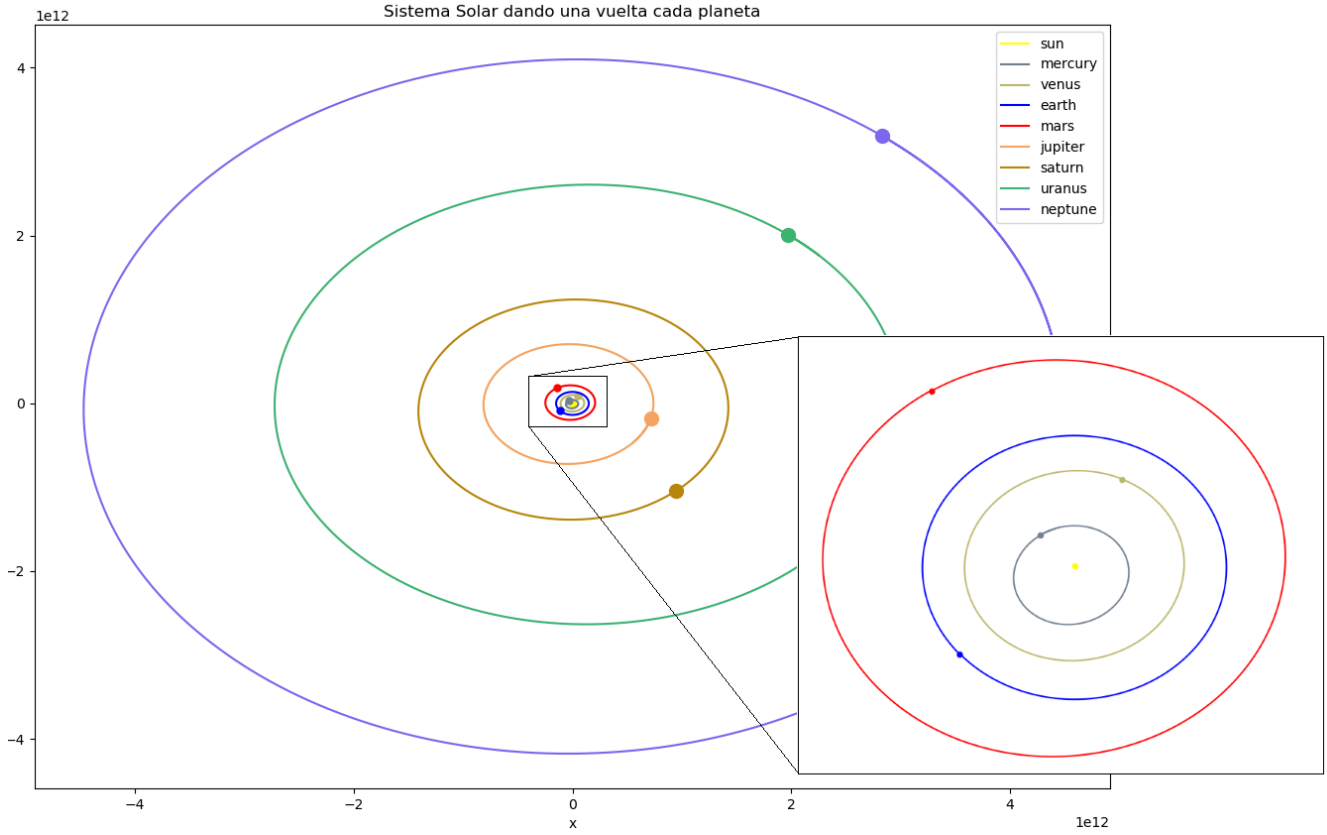


Figura 4: Sistema Solar en el que cada planeta realiza solo una vuelta.

8.3.1. Representación 3D

Además se representó el sistema solar en tres dimensiones, para así poder tener también en cuenta el eje z, como se ve en la Figura 5.

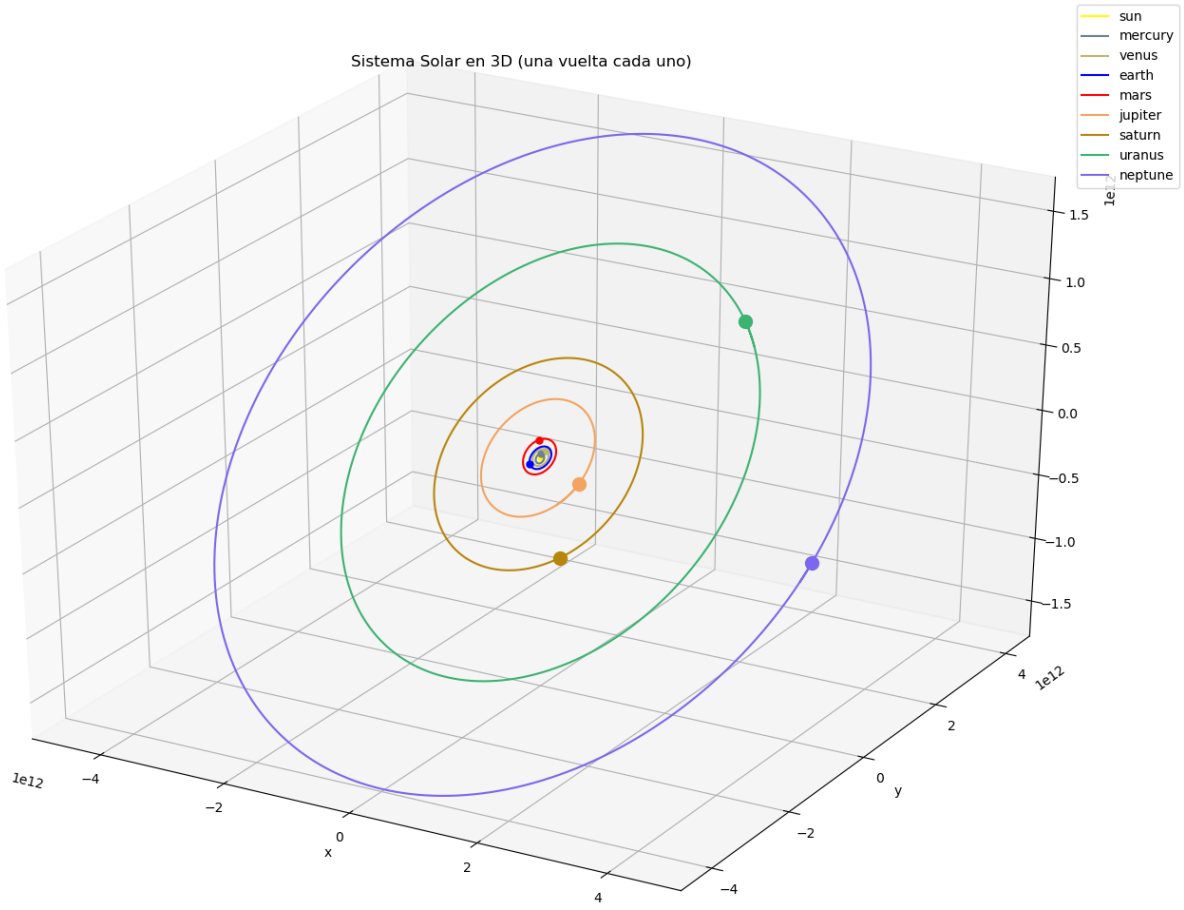


Figura 5: Representación 3D del Sistema Solar.

8.4. Representación en *vpython*

Finalmente se realizó una simulación dinámica del movimiento de los planetas mediante el módulo *vpython*. En la figura 6 solo se ve una muestra de la representación ya que en el informe no podemos mostrar el movimiento de los cuerpos.

Además, como se explicó anteriormente cuando se comentó como se realizó el código de *vpython*, el tamaño de Sol es mucho más pequeño de lo que le tocaría en comparación con los planetas debido a que las distancias se tuvieron que disminuir con el fin de que se pudiesen ver los cuerpos y sino la representación de nuestra estrella llegaría hasta la órbita de Mercurio.

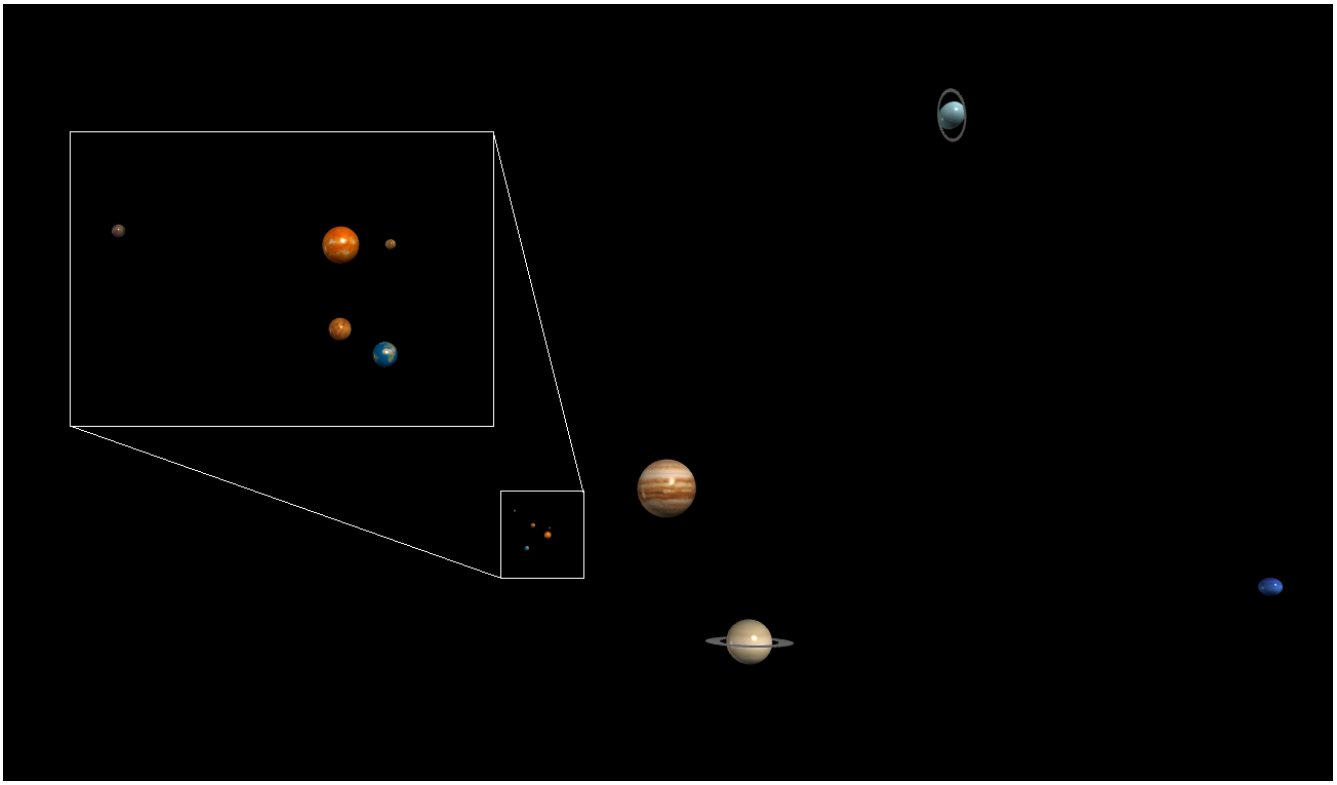


Figura 6: Representación en *vpython* del Sistema Solar.

9. Conclusiones

A modo de conclusión se puede apreciar que, para períodos de tiempo relativamente grandes (aproximadamente 165 años terrestres, que es el tiempo que tarda Neptuno en dar una vuelta completa alrededor del sol), los planetas con órbitas y periodos mucho menores, (como Mercurio, Venus...) sufren una notable desviación respecto a su órbita inicial. Esto puede achacarse a la acumulación de errores debido a la bestialidad de cálculos que se deben de realizar cuando tenemos en cuenta todos los planetas.

Como cada uno de los planetas influye sobre cada uno del resto, estos errores (y los cálculos a realizar) crecen exponencialmente, lo cual explica que los resultados para intervalos de tiempo relativamente grandes, pero con pocos planetas interviniendo, sean mucho menos fluctuantes que los resultados para el sistema solar 'en su totalidad'.

Otra consideración a tener en cuenta es que, aparte de los fallos en las órbitas provocados por la acumulación de errores, las trayectorias tampoco se ajustarían fielmente al sistema solar real pues no todos los cuerpos que lo componen son representados en esta simulación, tales como los diferentes satélites de los planetas, planetas enanos como Plutón, asteroides...

Por tanto, debemos concluir que esta simulación es una mera aproximación que estima, con un cierto rango de error, la evolución en el movimiento de una cantidad de cuerpos especificados, pero que en ningún momento busca ser una simulación con un alto grado de precisión del sistema solar real.

10. Bibliografía

[1] *scipy.integrate.solve_ivp*. Scipy.org. Consultado: 5-5-2021.

[2] *Método de Euler*. Wikipedia.com. Consultado: 5-5-2021.

[3] *Método de Runge-Kutta*. Wikipedia.com. Consultado: 5-5-2021.

[4] L.F. Shampine P. Bogacki. "A 3(2) pair of Runge - Kutta formulas". En: (1989).

[5] Julio M. Fernandez Díaz y Rosario Díaz Crespo. "Introducción a la Física Computacional: Problemas de Física resueltos numéricamente". En: (2021).

11. Apéndice

Listado de programas:

- Tierra-Sol.py: programa en el que se representa la trayectoria de la Tierra alrededor del Sol.
- 3planetas.py: permite añadir un planeta al sistema Tierra-Sol.
- solarsystem.py: están incluidos todos los planetas del Sistema Solar y sus trayectorias se representan en 2D.
- solarsystemoneyeareach.py: lo mismo que el anterior pero los planetas solo recorren una órbita.
- solarsystem3D.py: representamos las trayectorias en 3D.
- solarsystemVpythonrings.py: simulación dinámica del Sistema Solar a través de vpython.

Link a repositorio de *Github*®:

`https://github.com/DavidVB10/Solar_system`