1

Implementación de un software para la aproximación de órbitas

Una vez hemos estudiado el método de Laplace para la determinación de órbitas mediante tres observaciones, nos centraremos en desarrollar un programa que sea capaz de realizar estos cálculos y mostrar los resultados de manera visual para hacer más fácil la aproximación. Además, éste nos servirá de ayuda para comprobar la precisión con la que se determina una órbita mediante el método laplaciano.

Antes de hablar sobre el programa y todas sus funcionalidades, centrémonos en las herramientas que se han utilizado para todo el desarrollo.

1.1. Herramientas utilizadas.

Tal y como comentamos en la introducción, todo el código implementado para el funcionamiento de nuestro software está en lenguaje Python. Uno de los motivos por los que se ha tomado la decisión de elegir este lenguaje de programación es el hecho de que durante la carrera nos hemos familiarizado con él dado que en muchas asignaturas hemos requerido su uso. En primer momento se pensó en utilizar C++ por su velocidad y optimización, además de que también se ha utilizado durante la carrera, pero dado que solo necesitábamos hacer unos cálculos sin ninguna carga computacional alta, se decidió tomar Python.

Aún así hay un motivo más importante por el que se ha elegido Python,

concretamente dos, llamados *Numpy* [1] y *Matplotlib* [2]. Mediante estas dos bibliotecas matemáticas el desarrollo del método de determinación y la posterior visualización de resultados se hace mucho más fácil, y su instalación (como la de la mayoría de bibliotecas de Python) es fácil y rápida, por lo que el código podrá ser utilizado fácilmente en diferentes ordenadores.

Junto a estas dos librerías, básicas en Python, hemos utilizado también *seaborn* [3] (basada en *Matplotlib*) para mejorar la visualización de datos, *Astropy* [4] para ayudarnos con las conversiones de ángulos y fechas, *requests* [5] para hacer web scraping (método del que hablaremos más adelante) y *scipy* [1] y *sympy* [6] para ayudarnos con las aproximaciones numéricas.

Por otra parte, hemos utilizado el paquete *Tkinter* [7], el cuál suele venir por defecto en la instalación de Python, para realizar una interfaz y facilitar el uso del software de determinación de órbitas implementado.

Dado que no se conocía por completo el funcionamiento de cada una de estas librerías, se ha utilizado la documentación oficial de cada una de ellas (citada junto al nombre del paquete) para el desarrollo del programa informático por completo.

Durante el desarrollo de esta memoria, hemos utilizado GIT y la plataforma GitHub para mantener las versiones del proyecto y facilitar el acceso al código a quien lo necesite. Respecto a este documento, está desarrollado por completo en LaTeX, elegido porque nos proporciona un formato consistente y elegante. Finalmente, el código ha sido desarrollado utilizando el entorno de desarrollo (IDE) Spyder.

Una vez comentadas las distintas herramientas con las que hemos implementado nuestro software, pasemos a ver el funcionamiento del núcleo de éste.

1.2. Núcleo de la aplicación.

A la hora de desarrollar el código que se encargue de la determinación de órbitas, se ha ido separando el código en función de cuál es su cometido en diferentes archivos y carpetas. Así, la aplicación estará dividida en los directorios `scripts`, que se encargará de la base para la determinación, `utils`

que contendrá código útil para utilizar en diferentes momentos, y `test` que contendrá archivos con los que comprobar el correcto funcionamiento del programa. Dado que finalmente acabaremos utilizando una interfaz gráfica, no será necesario comentar el contenido del directorio `test`. Comentemos a continuación las principales funcionalidades de cada uno de estos archivos.

1.2.1. Elementos orbitales a partir de la posición y velocidad.

Si disponemos de los valores de la posición y velocidad de un cuerpo en un momento determinado, podremos obtener los elementos orbitales de éste tal y como comentamos en **??**. Por tanto, será lo primero en lo que nos centremos a la hora de implementar el código, y más tarde nos centraremos en el método de Laplace.

Para empezar, se ha creado una clase `OrbitalObject` que contendrá los elementos orbitales $(a, e, i, \Omega, \omega)$ que definen la órbita de un objeto junto a su nombre y su período p. En dicha clase se implementarán diferentes funciones para disponer de los ángulos en grados o radianes, así como la sobrecarga del método `__str__` para mostrar adecuadamente los valores de un objeto de esta clase por pantalla. Una vez creada la clase, definimos distintos objetos que guardaremos en `utils/my_constants.py` para un uso futuro, y así de paso comprobamos su correcto funcionamiento.

```
1 Earth = OrbitalObject(name='Earth',
2                       a=9.997843564797363E-01,
3                       e=1.707168344231522E-02,
4                       i=1.982259124359018E-03,
5                       Omega=2.194445465875467E+02,
6                       omega=2.426369002793497E+02,
7                       p=365.256363, degree=True)
```

El parámetro `degree` sirve para "avisar" de que los ángulos están en grados y es necesario almacenarlos en radianes. Todos los valores de las coordenadas astronómicas están tomados de la web de JPL [**?**] para el día 2020-Jul-28 20:00:00.0000.

Ahora que ya disponemos de objetos astronómicos, nos encargaremos de desarrollar un método para poder dibujar su órbita por pantalla. El código para esta funcionalidad se encuentra en el archivo `scripts/orbital_plot.py`

y está basado al completo en **??**. La función `plotOrbit(...)` recibirá como parámetros una lista con las coordenadas astronómicas de distintos objetos y un `bool` que simplemente servirá para centrar o no la gráfica en el Sol, que dibujaremos con un punto amarillo[1]. Se irán tomando los elementos de dicha lista uno a uno, se determinará y rotará la elipse que forman y mediante *Matplotlib* dibujaremos las órbitas en 3D. A continuación podemos ver un pseudocódigo de esta funcionalidad:

```
def plotOrbit(orbitas,sol_centrado=True):
    # Creamos la figura
    ax = crear_figura('3D')

    # Dibujamos el Sol
    ax.anyadir_punto((0,0,0),color='yellow')

    Para cada orb en orbitas:
        # Semieje menor y distancia de los focos
        b = a * raiz_cuadrada(1 - e*e)
        centro = a * e

        # Dibujamos la elipse y la rotamos
        ellipse = (a * cos(x) + centro, b * sin(x), 0)
        ellipse = rotar(i, Omega, omega)
        ax.dibuja(ellipse)

    # Establecemos los limites para los ejes
    Si sol_centrado:
        centrar_sol()
    Sino:
        centrar_orbita()

    mostrar_figura()
```

Dado que ya tenemos una estructura para almacenar las coordenadas astronómicas de un objeto y un método que es capaz de representar la elipse definida por dichos elementos, es momento de ponernos a trabajar en la función que nos calcule los elementos orbitales a partir de la posición y velocidad. El código encargado de esta funcionalidad se encontrará en `scripts/orbital_elements.py`, y simplemente seguirá los pasos descritos en **??** para realizar todos los cálculos a partir de la posición y velocidad

[1] Si no se centra la gráfica en el Sol, se centrará en los límites de la elipse más grande que se dibuje.

pasadas como parámetros. Tras hacer los cálculos pertinentes, devolverá un objeto de la clase `OrbitalObject` con todos los valores obtenidos y como nombre el que se le haya pasado a la función.

Así, ya podemos hacer distintas comprobaciones del funcionamiento del código. Nos basta con ir a la efemérides de JPL [**?**], seleccionar el cuerpo que queramos y anotar su posición y velocidad en un instante t. A continuación, utilizamos la función `getOrbitalElements` pasándole los valores escogidos anteriormente, y podremos dibujar la órbita del objeto que nos devuelva con el método `plotOrbit`. Un ejemplo de este proceso lo podemos ver en el archivo `test/test_1.py`.

Una vez explicado el funcionamiento para la representación de datos, pasemos a la parte importante del trabajo, el método de Laplace.

1.2.2. Implementación del método de Laplace.

El desarrollo informático del método de Laplace está basado en una función a la que, pasada una serie de parámetros que contendrán las observaciones de un cuerpo, llamará a una serie de sub-funciones, cada una correspondiente a uno de los pasos vistos en **??**, y devolverá la posición y velocidad de ese cuerpo en el momento correspondiendo a la segunda observación. Podemos entenderlo mejor con el siguiente pseudocódigo.

```
def Laplace(coordinates, times):
    # Pasamos de ascension recta y declinacion a cartesianas
    # Calculamos las derivadas (aproximadas)
    EC,EC‘,EC‘‘ = aproximaDerivadas(coordinates,times)

    # Tomamos de la web de JPL el vector Tierra-Sol
    SE,SE‘,SE‘‘,R = vectorSE(times[1])

    # Calculamos las distancias $\rho$ y $r$
    rho, r = get_rho_r(EC,EC‘,EC‘‘,SE,R)

    # Obtenemos los vectores de posicion y velocidad
    pos, vel = getPosVel(EC,EC‘,EC‘‘,SE,SE‘,R,r,rho)

    Devuelve pos, vel
```

La primera función, `aproximarDerivadas`, hará exactamente lo que dice

su nombre. Tomando los valores pasados en ascensión recta y declinación, los transformará a ecuaciones cartesianas mediante (**??**) y utilizando interpolación de Lagrange, aproximará la primera y segunda derivada del vector $\overrightarrow{EC}$, como hicimos en **??**.

Antes de continuar con el funcionamiento de estas funciones, recordemos las ecuaciones (**??**):

$$\begin{cases} x = \rho\lambda - X \\ y = \rho\mu - Y \\ z = \rho\nu - Z \end{cases}$$

Tal y como vimos, estas ecuaciones representaban las relaciones entre los tres cuerpos que nos interesan, C, E y S. Pero para poder resolverlas, necesitamos conocer el valor del vector (X, Y, Z). Dado que sería muy cansado tener que tomar la posición (y la velocidad) del Sol de la efemérides cada vez que quisiéramos hacer una aproximación, nos encargaremos de que nuestro programa tome ese valor automáticamente. Y de aquí surge otro problema, no podemos almacenar junto al programa la posición del Sol respecto a la Tierra en todo momento, pues aumentaría el tamaño de este considerablemente. De ahí surge la necesidad de realizar web scraping.

Web Scraping

La técnica de web scraping es un proceso mediante el cuál podemos obtener información de Internet de manera automatizada. Es una herramienta potente con la que incluso podemos rellenar formularios de manera automática y obtener en formato HTML la página resultante [8]. Para nuestro caso no tendremos que profundizar tanto en las funcionalidades del web scraping, ya que la web de JPL [**?**] nos facilita la obtención de la información del cuerpo que queramos a partir de la URL.

De esta manera, definimos en `util/utilities.py` una función llamada `getVectorsFromEphemeris` que reutilizaremos a lo largo de la implementación. A dicho método le pasaremos como parámetros los nombres de los cuerpos que A y B que formarán el vector $\overrightarrow{AB}$, el momento en el que queremos tomar el valor del vector y el plano de referencia utilizado (ICRF o eclíptica). Con estos valores formará una `string` que se corresponderá con la URL que nos devolverá la información requerida. Tras ello, con los paquetes *requests* y *csv* extraeremos la información de la web a la vez que gestionamos los distintos errores que pueden ocurrir. Tras esto, si todo ha

salido bien, se devolverán los vectores de posición y velocidad requeridos y una variable `True`, y si ha habido algún error, se devolverán vectores de ceros y una variable `False`.

Utilizaremos esta función al llamar al método `vectorSE`, obteniendo la posición y velocidad del Sol visto desde la Tierra utilizando el plano de referencia ICRF. También se obtendrá el valor de la segunda derivada mediante (**??**) y la distancia Tierra-Sol, devolviendo finalmente estos cuatro elementos.

Determinación de ρ y r : método de Newton.

El código desarrollado para determinar r y ρ es el más complejo. Para el cálculo de estos dos valores utilizaremos las secciones **??** y **??**.

La función consistirá en ir calculando los valores de D, D_1, N y ψ para así llegar a las cantidades m y M de la ecuación (**??**):

$$\sin^4 \phi = M \sin(\phi + m)$$

Una vez tengamos estos valores, iremos utilizando el método de separación de raíces en $[0, \pi]$ para encontrar los intervalos con una raíz en ellos, y aplicando en su valor intermedio el método de Newton, tal y como vimos en **??**, obtenemos así todas las raíces.

```
1  def approximate_phi(M,m,tol=1E-09,max_tries=64,plot=False):
2      f = sin(x)^4 - M * sin(x+m)
3
4      # Aplicamos el metodo de separacion de raices de Bolzano
5      Desde i = 0 hasta max_tries:
6          alpha_i= (i*pi) / max_tries
7          alpha_{i+1}= ((i+1) * pi) / max_tries
8
9          # Si encontramos una raiz en el intervalo ...
10         Si signo(f(alpha_i)) != signo(f(alpha_{i+1})):
11             # ... tomamos el valor intermedio ...
12             x_0=(alpha_i + alpha_{i+1}) / 2
13
14             # ... aplicamos el metodo de Newton ...
15             phi,n_iters=newton(f, x_0, tol)
16
17             # ... y anyadimos la raiz al resto
18             phi_values.anyade(phi)
```

```
19
20      Si plot:
21          dibuja(sin(x)^4)
22          dibuja(M * sin(x+m))
23          dibuja(phi_values)
24
25      Devuelve phi_values
```

Además, podremos mostrar si queremos una gráfica por pantalla de la intersección de las funciones junto a las raíces aproximadas, como vemos en el siguiente ejemplo:

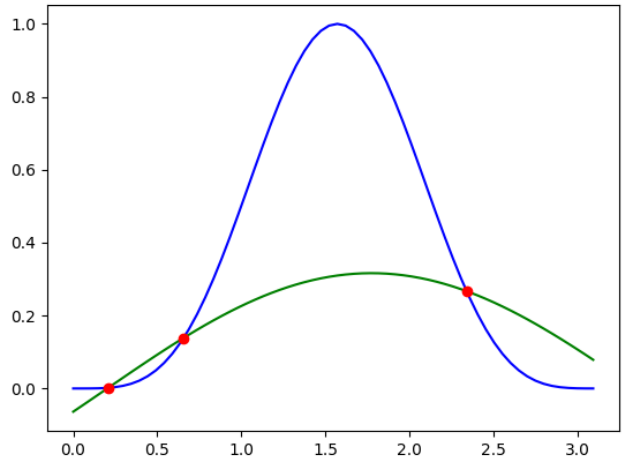

Fig. 1.1: Ejemplo de la gráfica que muestra la función `approximate_phi` junto a los valores de ϕ aproximados.

Tras obtener todos los posibles valores de ϕ en el intervalo $[0, \pi]$, comprobaremos cuál de ellos es igual a $\pi - \psi$ para discernir entre cuál de los dos restantes es la solución del problema físico, como hicimos en la parte final de **??**. En el caso de que la solución sea única, se devolverá el valor dicho valor, y si la solución es doble se avisará de ello por pantalla, dando la opción al usuario de elegir entre uno de los dos valores como solución del problema. Se deja para una futura implementación la posibilidad de añadir una cuarta observación para determinar en el caso de una solución doble cuál de las dos es la correcta.

Finalmente, con este valor de ϕ obtendremos las distancias r y ρ usando (**??**), y devolveremos estos valores.

Posición y velocidad del cuerpo.

Una vez hemos obtenido los vectores $\overrightarrow{EC}$ y $\overrightarrow{SE}$ y sus derivadas junto a las distancias R, r y ρ, solo tendremos que utilizar (**??**) y (**??**) para obtener la posición y velocidad del cuerpo observado.

Nótese que los valores obtenidos están respecto al plano ICRF, por lo que usaremos la función `ICRS_to_ecliptic` para pasarlos al plano de la eclíptica, tal y como se ha visto en **??**.

1.2.3. Estudio del error.

Para obtener una estimación de cómo de buena ha sido la aproximación del objeto mediante el método de Laplace, se implementarán una serie de funciones para estudiar la diferencia de estas aproximaciones con el valor real.

El método `getApproximationError` tomará como parámetros la posición y velocidad aproximada mediante el método laplaciano de un cuerpo en un instante t, así como el nombre del cuerpo que se ha decidido aproximar. Mediante dicho nombre, llamaremos a la función `getVectorsFromEphemeris` comentada anteriormente para obtener el valor real de la posición y velocidad del cuerpo en el instante t pasado como parámetro. Obtenidos los valores reales, bastará con comprobar la diferencia con los valores aproximados y sus normas, que la función almacenará en una variable `string`[2] y devolverá al terminar.

1.3. Desarrollo de una interfaz gráfica de usuario.

[2]Se ha escogido una variable del tipo `string` para facilitar la impresión por pantalla para la interfaz gráfica.

Bibliografía

[1] Oliphant, T.E. et al., (2020), Numpy (1.18.5) and Scipy (1.4.1) [library documentation]. Obtenido de link.

[2] Hunter, J.D. & Droettboom, M, (2020), Matplotlib (3.0.3) [library documentation]. Obtenido de link.

[3] Waskom, M., (2020), Seaborn (0.9.1) Color Palette [library documentation]. Obtenido de link.

[4] The Astropy Developers, (2020) Astropy (3.2.3) [library documentation]. Obtenido de link.

[5] Reitz, K., (2020), Requests (2.9.1) [library documentation]. Obtenido de link.

[6] SymPy development team, (2020), Sympy (1.6.2) [library documentation]. Obtenido de link.

[7] Lundh, F., (2020), Tkinter (3.5.1-1) [library documentation]. Obtenido de link.

[8] Alonso Burgos, S., (2020), *"Teoría"de Web Scraping* [Archivo de vídeo]. Recuperado de [vídeo oculto].