


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Análisis de los Hitos 4, 5 y 6

Ampliación de Matemáticas I

10 de diciembre de 2022

Tiempo de trabajo: 41 horas y 15 minutos.

Autor:

- Rafael Rivero de Nicolás

Índice

1. Introducción

En el presente informe se recogen las reflexiones y conclusiones extraídas de la realización de los Hitos 4, 5 y 6 de la parte de Cálculo Numérico de la asignatura de Ampliación de Matemáticas I del Máster Universitario en Sistemas Espaciales (MUSE) en la Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio (ETSIAE) de la Universidad Politécnica de Madrid (UPM).

En el Hito 4 se pretende resolver el problema de condiciones iniciales dado por la Ecuación 1. En la Sección 3 se analizan las regiones de estabilidad de los distintos esquemas empleados y se justifican así los resultados obtenidos.

$$\begin{cases} \ddot{x}(t) + x(t) = 0, & x, t \in \mathbb{R}, \\ x(t=0) = x_o = 0{,}5, \\ \dot{x}(t=0) = \dot{x}_o = 0{,}5. \end{cases} \tag{1}$$

En el Hito 5 se pretende resolver el Problema de los N cuerpos aprovechando las ventajas asociadas al uso de punteros en Python 3.9.12. En este caso, el número de cuerpos empleados será $N = 3$. La ecuación diferencial que rige este problema es Ecuación 2.

$$\begin{cases} \ddot{\vec{\mathbf{r}}}_i(t) = -\sum\limits_{\substack{j=1 \\ j\neq i}}^{j=N} \dfrac{\vec{\mathbf{r}}_i(t) - \vec{\mathbf{r}}_j(t)}{\left\|\vec{\mathbf{r}}_i(t) - \vec{\mathbf{r}}_j(t)\right\|^3}, & i = 1, 2, \ldots, N, \quad \vec{\mathbf{r}}_i \in \mathbb{R}^2, t \in \mathbb{R} \\ \vec{\mathbf{r}}_i(t=0) = \vec{\mathbf{r}}_{i_o} = (x_{i_o}, y_{i_o}, z_{i_o}), \\ \dot{\vec{\mathbf{r}}}_i(t=0) = \dot{\vec{\mathbf{r}}}_{i_o} = (\dot{x}_{i_o}, \dot{y}_{i_o}, \dot{z}_{i_o}). \end{cases} \tag{2}$$

En el Hito 6 se implementa un esquema Runge Kutta de paso adaptativo (*embedded Runge Kutta*) y se aplica este esquema para el cálculo de las órbitas alrededor de los puntos de Lagrange para el sistema Tierra-Luna, habiéndose linealizado previamente la Ecuación 2 alrededor de estos puntos críticos.

2. Estructura del código

2.1. Programas principales: `Milestone_IV.py`, `Milestone_V.py` y `Milestone_VI.py`

Los programas principales que permiten obtener los resultados correspondientes a cada Hito son, `Milestone_IV.py`, `Milestone_V.py` y `Milestone_VI.py`, respectivamente. Cada uno de estos programas principales es un código compacto gracias a la factoración en funciones y a la modularización.

Como comentario común, existe un parámetro en el Hito 4 y en el Hito 5 que permite la elección del esquema numérico empleado para resolver la ecuación diferencial correspondiente. En el Hito 6 el equivalente sería el nombre (`name`) del esquema multipaso deseado. Como mejora respecto a Hitos anteriores, se ha sustituido la creación de un diccionario con los nombres de los esquemas numéricos a elegir por un diccionario que agrupa las funciones de los esquemas numéricos disponibles. Posteriormente, para la representación gráfica se emplean el atributo `__.name__` de estas funciones para asignar un título a las gráficas realizadas.

2.2. Módulos necesarios

Las diversas tareas de cálculo han sido factorizadas en distintas funciones, permitiendo así la compartimentación del código, facilitando su lectura. Estas funciones han sido agrupadas en módulos, para conseguir ventajas de cara a su mantenimiento futuro y consiguiendo así una estructura de programación modularizada. Por tanto, para el correcto funcionamiento de los códigos código desarrollado, `Milestone_IV_Rafa.py`, `Milestone_V_Rafa.py` y `Milestone_VI_Rafa.py`, es necesario importar varios módulos o librerías que se describen brevemente a continuación.

2.2.1. `LB_Temporal_Schemes.py`

Este módulo reúne todas las funciones asocidas a un esquema numérico que permiten calcular el vector de estado en un instante t_i, U^i, en función del vector de estado en otros instantes de tiempo, U^j, del instante de tiempo t_j y del Δt de la malla en dicho instante.

Ha sido empleado en Hitos anteriores y se le han añadido funciones que permiten el cálculo y la representación de las regiones de estabilidad de los esquemas numéricos Euler, Euler Inverso, Crank-Nicolson y Runge Kutta de 4^{o} orden ya que las regiones de estabilidad son propiedades de cada esquema numérico y se ha considerado mejor mantener esta funcionalidad en el mismo módulo. También se ha añadido una función que permite seleccionar el esquema Runge Kutta multipaso deseado de entre una pequeña colección, además de otra función, mostrada en Extracto de código 1, para realizar el cálculo del vector de estado para cada instante.

```
def Embedded_RK_U_n1(Function, Un, t, h, tag, name="RK65"): # This
   Embedded Runge Kutta computes U_{n+1} according to the selected scheme
    in ERK_selection
    a, b, b2, c, q = ERK_Selection(name)
    k = zeros( [ len(Un), len(b) ] ) # k_i is k[:,i]

    for i, ai in enumerate(a):

        S = zeros( len(Un) )

        for j in range(i): # j in [0, i-1]

            S = S + ai[j]*k[:,j]

        k[:,i] = Function(Un + h*S, t+c[i]*h)

    if tag == "1":
        U_n1 = Un + h * matmul( b, k.transpose() ) # U_{n+1} = U_{n} + dt
    * \Sum_{i=0}^{i=s-1} (b_i * k_i); s == len(a)
        return U_n1, q

    else:
        U_n1_2 = Un + h * matmul( b2, k.transpose() )
        return U_n1_2
```

Extracto de código 1: Función para cualquier esquema Runge Kutta explícito de paso adaptativo.

2.2.2. LB_Physics_Problems.py

En este módulo se han agrupado todas las funciones F que permiten expresar una ecuación diferencial ordinaria de la forma

$$\frac{\mathrm{d}U}{\mathrm{d}t} = F(U,t). \tag{3}$$

De forma que se recogen funciones provenientes de problemas como el oscilador armónico, del problema de Kepler o incluso para el problema de los N cuerpos, función recogida en el Extracto de código 2.

```
def N_Bodies_Function(U, t):

    if int( len(U) ) % 6 == 0:
        Nb, Nc = (int(len(U)/6), 3)
    else:
        print("Error; U must has a dimension of 6*N, with N an integer:
   number of bodies")

    Us = reshape( U, (Nb, Nc * 2) )

    r = Us[:,0:Nc]
    v = Us[:,Nc:]
```

```
    F = zeros(len(U)) # Vector that will be F(U) at the end of the
   function
    F_pointer = reshape( F, (Nb, Nc*2) )

    drdt = F_pointer[:,0:Nc]
    dvdt = F_pointer[:,Nc:]

    drdt[:,:] = v[:,:]

    for i in range(Nb):

        dvdt[i,:] = 0

        for j in range(Nb):

            if j != i:

                d = r[j,:]-r[i,:]

                dvdt[i,:] = dvdt[i,:] + (d)/norm(d)**3

    return F
```

Extracto de código 2: Función para el problema de los N cuerpos empleada en `Milestone_V.py`.

Como apunte, esta función ha sido modificada respecto del NumericalHub para que funcione de acuerdo al criterio establecido en [3] para introducir las condiciones iniciales.

2.2.3. `LB_Math_Functions.py` y `ODE_PDE_Solvers.py`

Previamente el módulo `LB_Math_Functions.py` era importado en el módulo que contiene los esquemas temporales, `LB_Temporal_Schemes.py` y viceversa, lo que no se correspondía con una estructura de código jerarquizada dado que ambos módulos se retroalimentaban entre sí. Además, era común el error representado en la Figura 1.

```
AttributeError: partially initialized module 'LB_Temporal_Schemes' has no
attribute 'RK4' (most likely due to a circular import)
```

Figura 1: *Atribute error.*

Para solucionar este problema, las funciones del antiguo módulo `LB_Math_Functions.py` que dependían del módulo `LB_Temporal_Schemes.py`, que son `Cauchy_Problem_V1()` y `Cauchy_Problem_V2()`, se han recogido en un módulo a parte que está destinado a almacenar las funciones que se emplean para la resolución de ecuaciones diferenciales y que en un futuro podría permitir resolver ecuaciones de contorno o ecuaciones diferenciales en derivadas parciales. Este módulo cuya creación se ha realizado durante el desarrollo del Hito 6 se llama `ODE_PDE_Solvers.py` y también incluye una función llamada `Embedded_RK_Application()` que resuleve el problema de Cauchy con un esquema Ruge Kutta de paso adaptativo.

La función `Cauchy_Problem_V2()` es una variante de la función `Cauchy_Problem_V1()` usada en anteriores Hitos. La diferencia sustancial es que en la nueva versión se introduce directamente la función del esquema temporal como argumento en vez de una cadena de caracteres con su nombre. Este cambio se ha realizado según los comentarios recibidos de los Hitos previos.

2.3. Código `Milestone_IV.py`

El código del Hito 4 es un código simple basado en la estructura del Hito 3: una sección de inputs para que el usuario pueda elegir el esquema numérico empleado, el problema físico que se pretende resolver y parámetros referentes al dominio temporal elegido.

```
# %% Definition

r_0 = 0.5; v_0 = 0.5 # Initial position and velocity, respectively.
Differential_Operator = Undamped_Armonic_Oscilator # [Undamped Armonic
    Oscilator [1D]]

Delta_t = [0.1, 0.08, 0.05] # For Euler and Inverse Euler

tf = 30 # Final time of the simulation

Temporal_schemes_available = {1:ts.Euler,
                              2:ts.Inverse__Euler,
                              3:ts.Crank__Nicolson,
                              4:ts.RK4,
                              5:ts.Leap_Frog}

Temporal_scheme = Temporal_schemes_available[2]

Initial_conditions = hstack((r_0,v_0)); print('Initial State Vector: U_0
    = ', Initial_conditions, '\n\n\n')
U = {}; time_domain = {}

# %% Numeric Simulation
for dt in Delta_t:

    time_domain[dt] = linspace(0, tf, int(tf/dt)+1 )

    U[dt] = Cauchy_Problem_V2(Differential_Operator, Initial_conditions,
    time_domain[dt], Temporal_scheme)


# %% Stability Region call
ts.Absolute_Stability_Region(Temporal_scheme, lst = Delta_t) # User's
    Numeric Jacobian should be used for this purpose due to the use of
    complex values. It can be found in LB_Math_Functions

# %% Plots
# ...
```

Extracto de código 3: Código principal `Milestone_IV.py`.

Como aspecto señalable, destaca el hecho de que a la hora de calcular las regiones de estabilidad de sistemas implícitos complejos, tanto las funciones `fsolve()` como `newton()` han dado problemas. Por lo que este cálculo debe realizarse con la función `Newton_Raphson()` desarrollada por el alumno.

2.4. Código `Milestone_V.py`

Este Hito se ha estructurado de forma general al Hito 4: con una primera zona para establecer los inputs del código y luego una llamada en bucle a la función `Cauchy_Problem_V2()` para resolver el problema de los N cuerpos con distintas mallas temporales elegidas por el usuario. Ver Extracto de código 4.

```
# Imports...

# %% Inputs
''' The code between ---- should be editable by the user '''
''' ------------------------- '''
SELECTED_SCHEME = 4

tf = 50

Delta_t = [0.5, 0.1]    # dt for different simulations

U_0_1 = array([2, 2, 0, 0.5, 0, 0])
U_0_2 = array([-2, -2, 0, -0.5, 0, 0])
U_0_3 = array([0, 0, 0, 0, 0, 0])
''' ------------------------- '''

Temporal_schemes_available = {1:ts.Euler,
                              2:ts.Inverse__Euler,
                              3:ts.Crank__Nicolson,
                              4:ts.RK4,
                              5:ts.Leap_Frog}

Nb = 3; Nc=3 #Number of bodies; Number of coordinates per body

# %% Pre-Computing

U_0 = hstack((U_0_1, U_0_2, U_0_3))
scheme = Temporal_schemes_available[SELECTED_SCHEME]
U = {}; time_domain = {}

# %% Simulations

for dt in Delta_t:

    N = int( tf/dt )

    time_domain[scheme.__name__+'__dt=' + str(dt)] = linspace( 0, tf, N+1
    )

    print('Temporal partition used dt = ', str(dt))
```

```
    U[scheme.__name__+'__dt=' + str(dt)] = Cauchy_Problem_V2( F =
    N_Bodies_Function, U_0 = U_0, time_domain = time_domain[scheme.
    __name__+'__dt=' + str(dt)], Temporal_scheme = scheme )

    print('\n\n\n')

# %% Plots
# ...
```

Extracto de código 4: Código principal `Milestone_V.py`.

Al igual que para el código de los Hitos 3, 4 y 6, la compactación del cálculo en funciones a través de abstracciones funcionales permite que el código principal de cada Hito sea reducido y de esta forma más fácil de leer y de introducir cualquier cambio o actualización sin alterar el funcionamiento en absoluto de todo el motor de cálculo que se ha repartido en los distintos módulos.

2.5. Código `Milestone_VI.py`

El Hito 6 se ha basado en dos etapas:

1. la creación de un esquema Runge Kutta de paso adaptativo y,
2. la implementación de este esquema para el cálculo de órbitas alrededor de los puntos de Lagrange para el sistema Tierra-Luna.

Para la primera de ellas se ha generado un código basado en el repositorio NumericalHub[1] de forma similar a la función `Cauchy_Problem_V2()`, se ha creado específicamente para este Hito la función `Embedded_RK_Application()` que permite realizar de forma iterativa el cálculo del vector de estado en toda la partición temporal de acuerdo a una condición inicial (`U_0`) y demás parámetros: una tolerancia referente al error local de truncamiento, una acotación inferior del Δt, ..., como se observa en el Extracto de código 5. Esta función engloba funciones que acaban llamando a la función `Embedded_RK_U_n1()` recogida en el Extracto de código 1.

```
def Embedded_RK_Application(U_0, F, time_domain, name, tolerance, dt_min)
    : # Global Embedded Runge Kutta application

    dt_min_reached = 0

    U = zeros( [len(U_0), len(time_domain)] )
    U[:,0] = U_0

    for i, t in enumerate(time_domain[1:]): # t = time_domain[i+1]

        dt = t-time_domain[i]
```

[1] https://github.com/jahrWork/NumericalHUB/blob/master/sources/Numerical_Methods/Cauchy_Problem/sources/High_order/Embedded_RKs.f90.

```
        U_n1, dt_min_reached = ERK_Temporal_Application( U[:,i], t, dt, F
    , dt_min_reached, tolerance, dt_min, name)

        U[:,i+1] = U_n1

    return U, dt_min_reached
```

Extracto de código 5: Función de aplicación para el esquema Runge Kutta de paso adaptativo en `Milestone_V.py`.

Como añadido adicional, esta función devuelve la variable `dt_min_reached`, la cual es un contador que indica el número de pasos en los que no se ha conseguido satisfacer la tolerancia establecida relacionada con la aproximación del error de truncamiento local.

Por otro lado, para calcular las órbitas alrededor de los puntos de Lagrange, primero deben encontrarse los puntos críticos del sistema Tierra-Luna. Para esto se ha empleado la función `Newton_Raphson` creada para anteriores hitos, como se recoge en el Extracto de código 6.

```
U0 = zeros([6, 5]) # 6 coordinates per body and 5 Lagrange points for 2
    main Bodies

U0[:,0] = [ 0.8, 0.6, 0., 0., 0., 0.  ]   # Lagrange points calculation
U0[:,1] = [ 0.8, -0.6, 0., 0., 0., 0.  ]
U0[:,2] = [ -0.1, 0.0, 0., 0., 0., 0.  ]
U0[:,3] = [ 0.1, 0.0, 0., 0., 0., 0.  ]
U0[:,4] = [ 1.1, 0.0, 0., 0., 0., 0.  ]
# Sol = newton(Autonomous_F, U0[:,4]) # No converge bien xD
L1 = mth.Newton_Raphson(Autonomous_F, x_i=U0[:,3])
L2 = mth.Newton_Raphson(Autonomous_F, x_i=U0[:,4])
L3 = mth.Newton_Raphson(Autonomous_F, x_i=U0[:,2])
L4 = mth.Newton_Raphson(Autonomous_F, x_i=U0[:,0])
L5 = mth.Newton_Raphson(Autonomous_F, x_i=U0[:,1])
```

Extracto de código 6: Metodología de cálculo de los puntos de Lagrage en `Milestone_VI.py`.

El código anterior solo se ejecutó en las primeras fases de desarrollo puesto que para ahorrar esfuerzo computacional se apuntaron los puntos de equilibrio y en el código actual se inicializan directamente, de acuerdo al Extracto de código 7.

```
L = {} # Dictionary that will contain Lagrange points
J = {}; lambdas = {} # Dictionaries for Jacobian matrices and their
    eigenvalues

L[1] = array([0.8369151258197125, 0. , 0. , 0. , 0. , 0. ])
L[2] = array([1.1556821654078693, 0. , 0. , 0. , 0. , 0. ])
L[3] = array([-1.0050626458062681, 0. , 0. , 0. , 0. , 0. ])
L[4] = array([0.48784941440000085, 0.8660254037844383 , 0. , 0. , 0. , 0.
    ])
L[5] = array([0.48784941440000085, -0.8660254037844383 , 0. , 0. , 0. ,
    0. ])
```

```
for (key, value) in L.items():

    U0[:,key-1] = value

    J[key] = Numeric_Jacobian(Autonomous_F, U0[:,key-1])

    lambdas[key] = eigvals(J[key])

for key in L:
    print("For Lagrange point L"+str(key)+"::    ")
    for i, value in enumerate(lambdas[key]):
        print("  Re(lambda_"+str(i)+") = "+ str(value.real) )
```

Extracto de código 7: Cálculo de los autovalores del problema linealizado alrededor de los puntos de equilibrio en `Milestone_VI.py`.

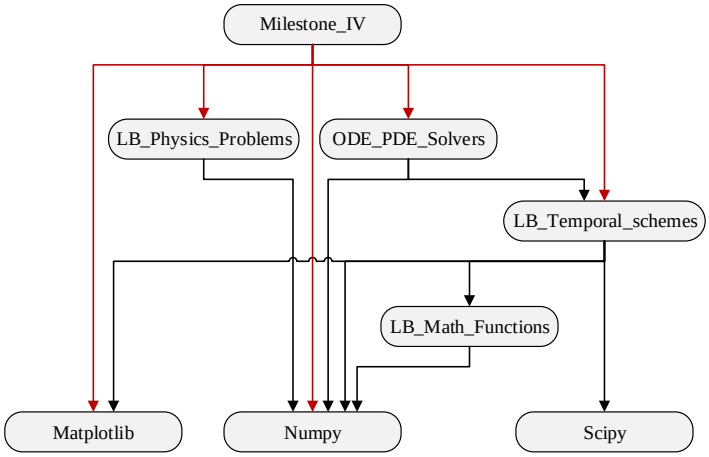
```
# %% Simulation and plotting

tf = 300; nt = 12000; dt = tf/nt

time_domain = linspace(0, tf, nt)

Uper = U0+rand(U0.shape[0], U0.shape[1])*1E-4; Uper[2,:] = 0

for j, v in enumerate(U0.transpose()): # j will be the index of the
    columns

    U = Cauchy_Problem_V2(R3BP_Earth_Moon, Uper[:,j], time_domain) #
    Default RK4

    U2, dt_min_reached = Embedded_RK_Application(Uper[:,j],
    R3BP_Earth_Moon, time_domain, name="RK87", tolerance = 1E-10, dt_min =
     0.001)
    # Plots
    #...
```

Extracto de código 8: Cálculo del vector de estado para el problema linealizado alrededor de los puntos de equilibrio en `Milestone_VI.py`.

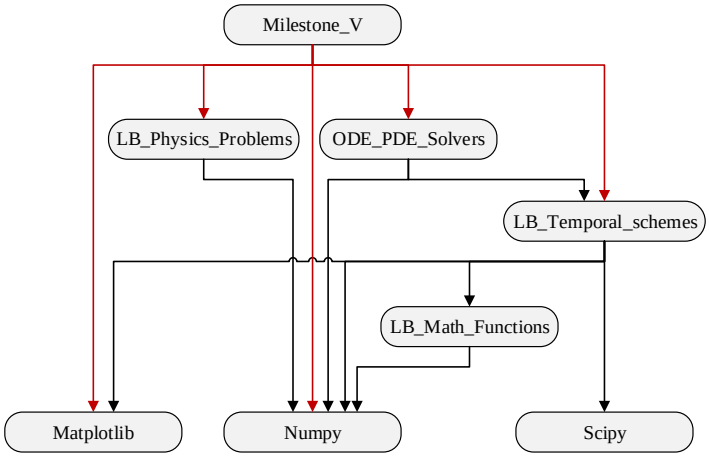
En el Extracto de código 8 se muestra el código encargado de calcular las órbitas alrededor de cada punto de Lagrange para una perturbación inicial reducida en posición y velocidad. Dentro del bucle se incluye el código correspondiente a la representación gráfica que ha sido omitido ya que este contenido no se ha considerado interesante.

2.6. Jerarquía de Módulos

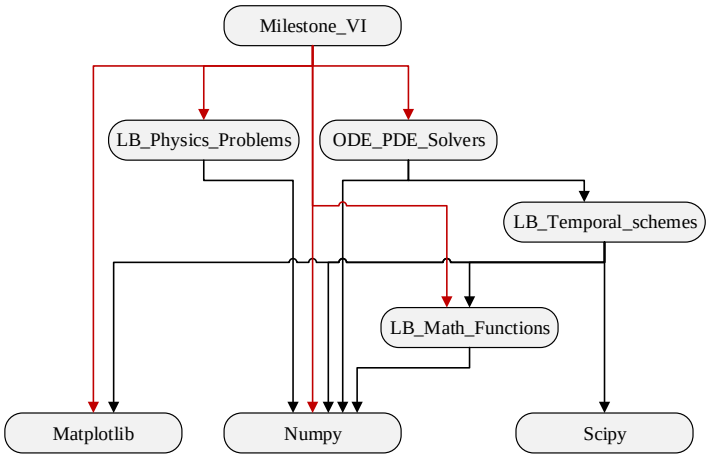
Los módulos básicos desarrollados por el alumno que permiten realizar todos los procesos de cálculo sobre los que se cimientan los tres Hitos comentados en este informe son los módulos explicados previamente: `LB_Math_Functions.py`, `LB_Temporal_Schemes.py`, `ODE_PDE_Solvers.py` y `LB_Physics_Problems.py`.



(a) `Milestone_IV.py`.



(b) `Milestone_V.py`.



(c) `Milestone_VI.py`.

Figura 2: Representación de la jerarquía de módulos mediante diagramas de bloques.

Se han agrupado en diagramas de bloques las relaciones jerárquicas entre los distintos módulos para cada uno de los Hitos en la autoreffig: Jerarquia. Estos diagramas de bloques permiten una rápida visualización de las relaciones entre todos los módulos sobre los que se cimientan cada uno de los Hitos.

Como se aprecia en la imagen, los Hitos 4 y 5 tienen una estructura idéntica, pues aunque resuelvan distintos fenómenos físicos su estructura es la misma:

1. emplean una función de `LB_Physics_Problems.py` que modeliza un problema diferencial,
2. inicializan las condiciones iniciales, generalmente haciendo uso de alguna función de Numpy,
3. se selecciona el esquema numérico del módulo `LB_Temporal_Schemes.py` que se quiere emplear,
4. se procede a la simulación numérica a través de las funciones destinadas para este propósito, del módulo `ODE_PDE_Solvers.py`, y,
5. se finaliza con la representación gráfica de resultados empleando el módulo de Matplotlib.

En el Hito 6, la estructura es algo distinta debido a que el esquema numérico empleado para la resolución del problema correspondiente se elige directamente al llamar a la función `Embedded_RK_Application()` del módulo `ODE_PDE_Solvers.py`. Sin embargo, se importa el módulo `LB_Math_Functions.py` para el cálculo del Jacobiano.

Gracias a seguir una metodología similar en la construcción de estos códigos, se han conseguido estandarizar los códigos *main* de forma que se acelera y simplifica el proceso de creación de estos, además de que al estar cada tarea del código perfectamente definida y asignada, cualquier corrección o actualización resulta muy sencilla de implementar.

2.7. Mejoras respecto a Hitos anteriores

Además de la creación de nuevos módulos, como se explica en la Subsección 2.2, que permiten la obtención de una estructura jerarquizada completamente vertical que no existía completamente en Hitos anteriores, se han empleado técnicas de programación avanzadas estudiadas en clase.

- Destaca el uso de punteros en el Hito 5 para el cálculo de las fuerzas ejercidas sobre un cuerpo por el resto de objetos orbitales, lo que se puede apreciar en el Extracto de código 2.
- Se ha actualizado la función `Newton_Raphson()` para que permita resolver ecuaciones vectoriales implícitas de variable compleja.

- También se ha mejorado al descubrir funcionalidades que facilitan los bucles sobre elementos iterables, empleando `ennumerate()`, como en el Extracto de código 5, o con el atributo `.items` en diccionarios , como en el Extracto de código 7. Este tipo de mejoras las considero relevantes pues no solo ahorran lineas de código, sino que falicitan y amenizan el proceso.

- Por último, el hecho de llevar varios meses trabajando siguiendo una filosofía funcional a la hora de programar ha llevado consigo una experiencia que sin duda se ha vuelto cada vez más sencilla. Esta forma de planteamiento del código se ha vuelto más automática cada vez, siendo el paradigma de programación funcional en aquello que se piensa por primera vez a la hora de escribir un código, cosa que no ocurría al principio de la asignatura.

2.8. Aspectos mejorables del código

Como aspectos a mejorar, se encuentran los que generan las siguientes inquietudes:

- Me gustaría que el número de módulos importados desde el programa principal de cada uno de los Hitos fuera menor, aunque los programas son relativamente sencillos requieren de 4 o 5 módulos para funcionar, y cada uno de estos módulos depende a su vez de un par de ellos. Es cierto que Numpy y Matplotlib son necesarios, pero aun así me gustaría reducir el número de módulos utilizados desde cada programa para tener un estructura aun más jerarquizada.

- Otro aspecto que me gustaría solucionar[2] sería que a la hora de resolver ecuaciones implícitas, estas se resolviesen con `newton()` de Scipy o con `Newton_Raphson()` del módulo `LB_Math_Functions.py` en función de qué se tipo de ecuación implícita se quiera resolver. Las ecuaciones implícitas **no vectoriales** reales o complejas se resuelven mucho más rápido con la función `newton()` de Scipy, sin embargo, las ecuaciones implícitas vectoriales no son resolubles. Con `Newton_Raphson()`, no ha habido ningún problema a la hora de resolver ningún tipo de ecuación, pero su velocidad es mucho más reducida.

  Una posible solución a esto sería llamar siempre a la misma función a la hora de resolver ecuaciones implícitas y que esta filtrase qué tipo de ecuación implícita se quiere resolver, si compleja o real o si vectorial o unidimensional. Se propone la aplicación de esta idea mediante un *wrapper* de la función ya existente `Newton_Raphson()` para que, dependiendo de sus argumentos, ejecute el código actual o ejecute la función correspondiente de Scipy para conseguir siempre el resultado correcto con la mayor velocidad y eficiencia posible.

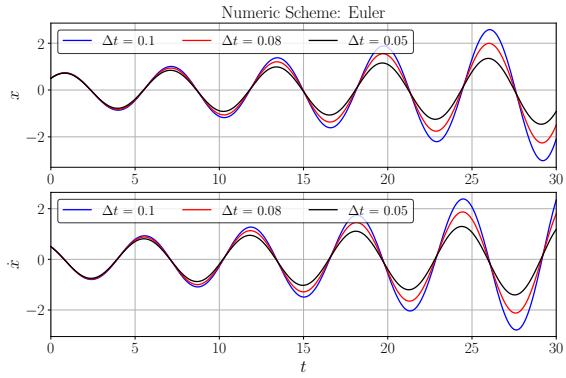
- Se propone también la incorporación de un módulo de *wrappers* que permitan realizar estimaciones del coste computacional (en tiempo y en memoria) del problema resuelto. Además de herramientas de *profiling* para aumentar la fiabilidad de las funciones evitando que den errores por argumentos incorrectos.

[2] Durante la redacción de este párrafo se encontró la función `root()` de Scipy y parece que de momento no genera problemas de incompatibilidad con el código.

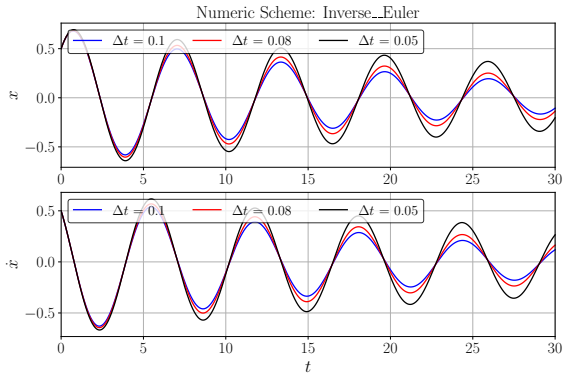
3. Resultados

3.1. Hito 4

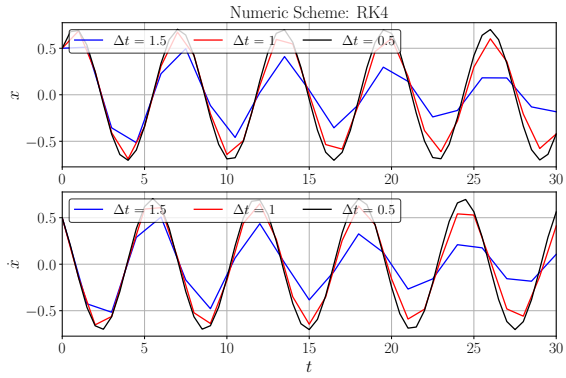
La Ecuación 1 representa un oscilador armónico con condiciones iniciales de posición y velocidad no nulas. En la Figura 3 se representan las soluciones obtenidas numéricamente con distintos esquemas y Δt empleados. El hecho de que el problema planteado esté regido por una ecuación diferencial lineal permite estudiar la estabilidad de las soluciones numéricas independientemente de las condiciones iniciales empleadas.



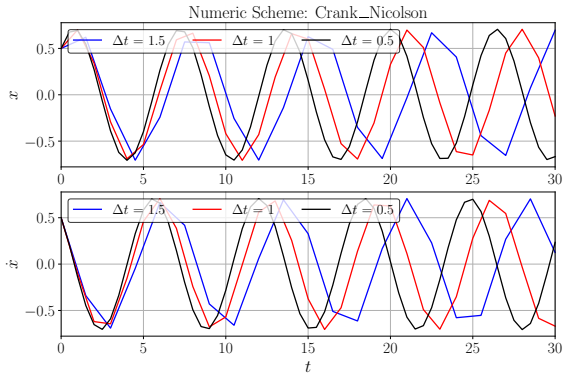
(a) **Esquema Euler.**



(b) **Esquema Euler Inverso.**



(c) **Esquema Runge Kutta de 4º orden.**



(d) **Esquema Crank-Nicolson.**

Figura 3: Resultados de la Ecuación 1 con distintos esquemas numéricos.

Los resultados obtenidos son acordes a los resultados obtenidos de los Hitos previos: el esquema Euler y el Euler inverso no conservan la energía del sistema, ni siquiera con un mallado temporal mucho más refinado que los esquemas Runge Kutta de 4º orden o Crank-Nicolson. El esquema Runge Kutta proporciona un resultado muy similar al real, incluso con Δt elevados, excepto para Δt =1.5, en el cual el esquema numérico disipa la energía del oscilador armónico. Esto es una consecuencia de que dicho Δt cae en la región de estabilidad del esquema numérico empleado representado en la Figura 4, por lo que si bien la solución obtenida es estable (se mantiene siempre a una distancia acotada de la solución analítica) no es una buena solución. En el esquema Crank-Nicolson, por el contrario, como los autovalores del sistema lineal están completamente en la frontera de estabilidad, no se

produce disipación de energía e independientemente del Δt la solución obtenida no se aleja de la analítica. Sin embargo, se aprecia perfectamente en la Figura 4d como existe un error de fase entre las soluciones que depende del Δt como se vio en clase.

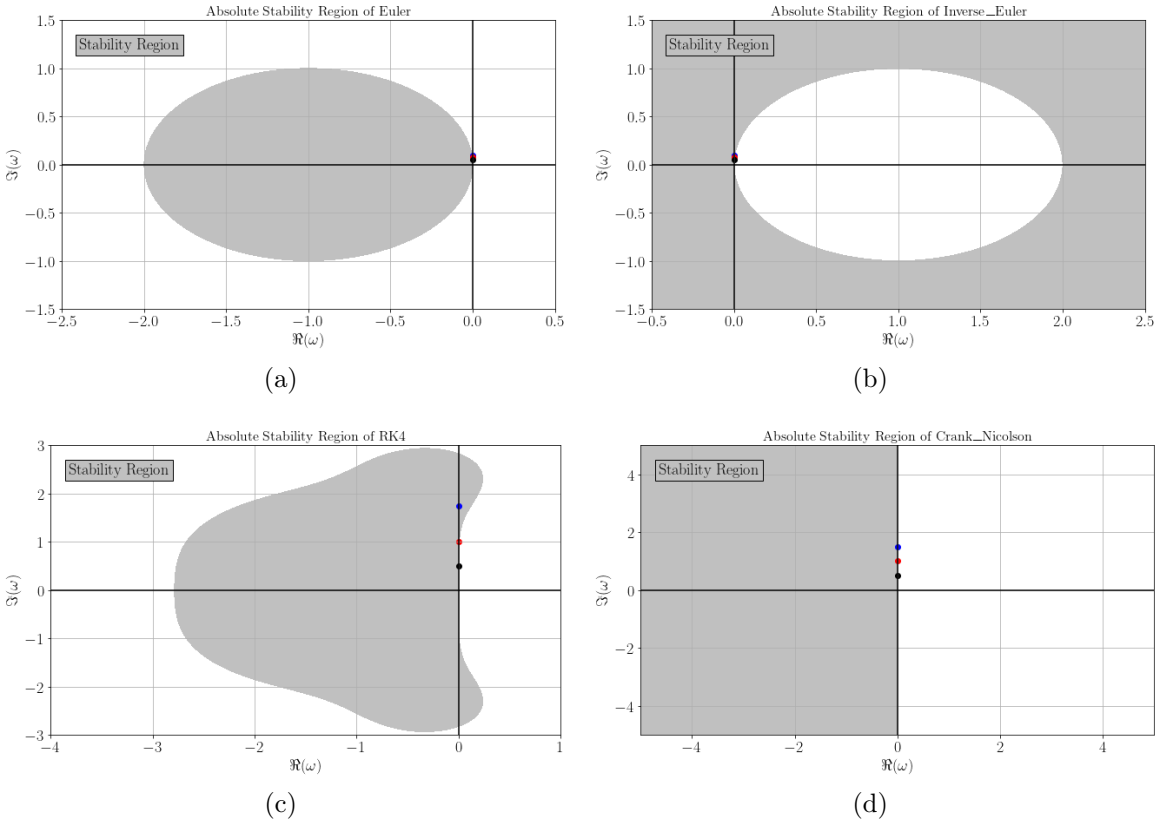


Figura 4: Regiones de estabilidad de distintos esquemas numéricos empleados.

Tras la clase en la que se explicaron los esquemas GBS y cómo estos incorporan un filtrado que permite eliminar soluciones espúreas resultantes del esquema Leap frog se ha aprovechado este Hito para observarlas. En la Figura 5 se recogen las soluciones numéricas frente al oscilador armónico introduciendo un pequeño error en el primer cálculo previo al empleo del esquema Leap Frog, lo que facilita la aparición de estas soluciones y exagera su efecto, como se ve en las Figuras 5a, 5b y 5c. En estas imágenes se ve cómo existe un término armónico que hace oscilar la solución numérica y que su efecto es mucho más acusado cuanto más refinados sea el mallado temporal con el que ha realizado la simulación. En la 5d se recogen las soluciones de este esquema cuando no se exagera la parte espúrea de la solución y se aprecia como los distintos Δt empleados proporcionan todos ellos soluciones estables que se mantienen próximas a la solución analítica.

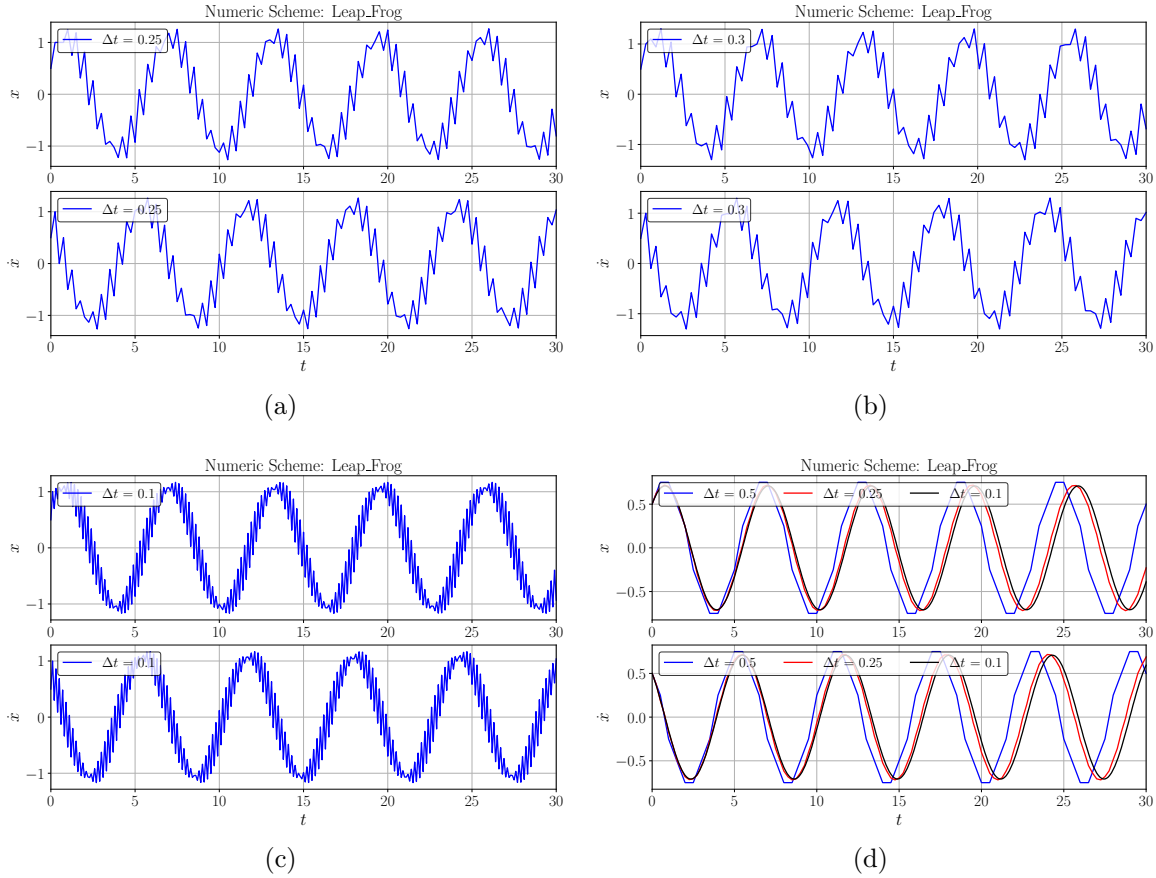
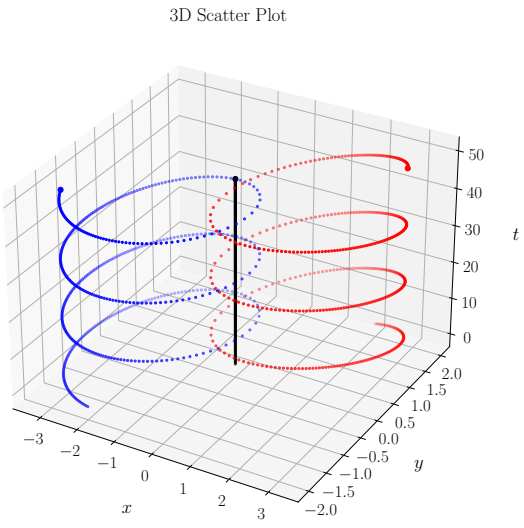


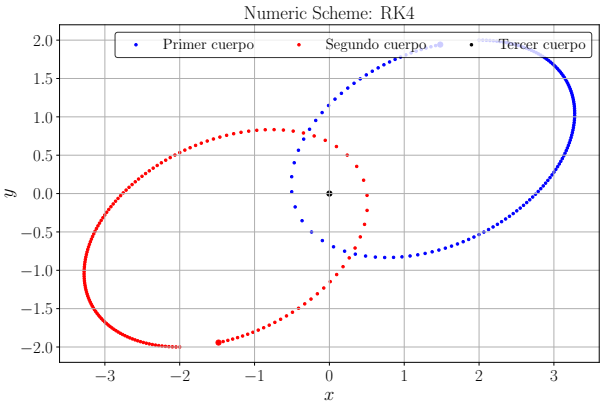
Figura 5: Soluciones numéricas con el esquema Leap Frog.

3.2. Hito 5

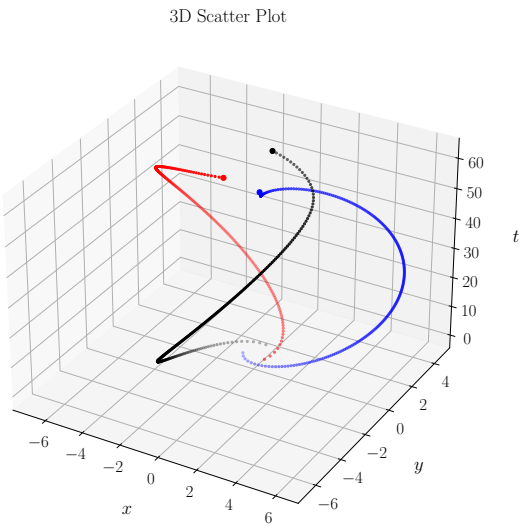
En este Hito se ha resuelto el problema de los 3 cuerpos, todos ellos con la misma masa, para distintas condiciones iniciales. Dos simulaciones han sido realizadas, cada una con condiciones iniciales distintas. Para la correcta visualización de los resultados se han limitado los resultados mostrados a problemas planos, aunque el código permite también la resolución de problemas tridimensionales.



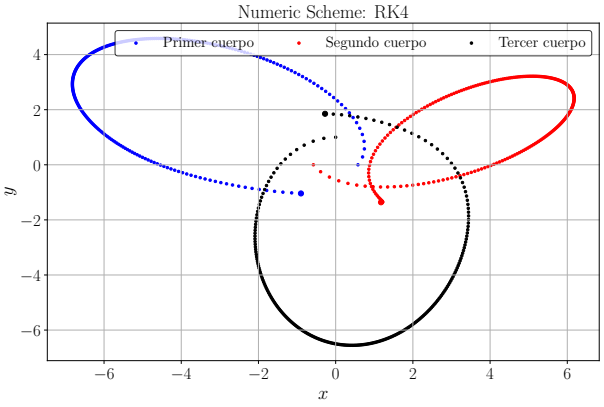
(a) **Simulación 1 en 3D.**



(b) **Simulación 1 en 2D.**



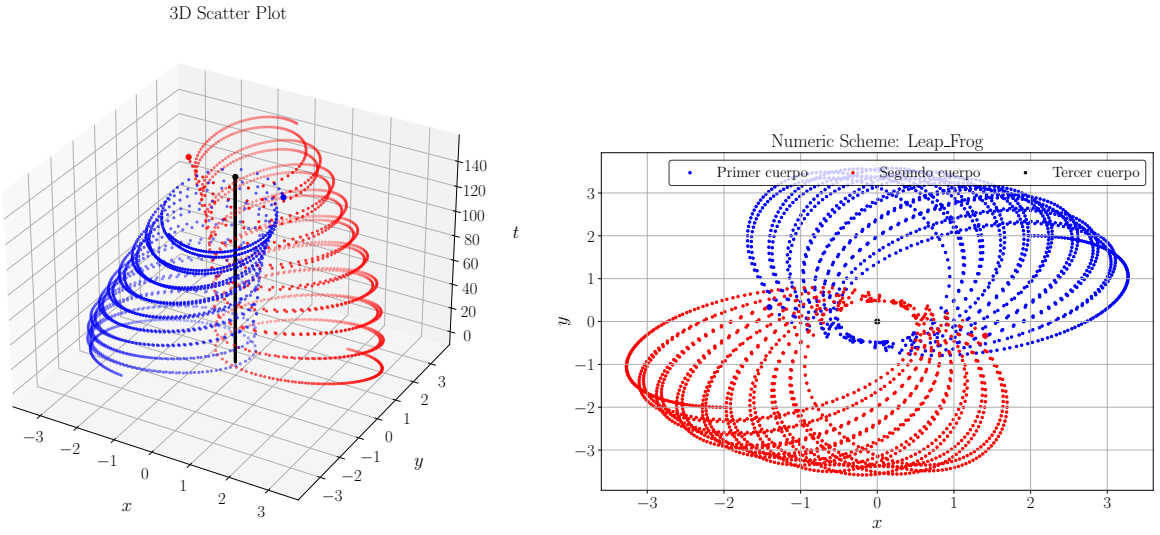
(c) **Simulación 2 en 3D.**



(d) **Simulación 2 en 2D.**

Figura 6: Trayectorias para el problema de los 3 cuerpos integrado numéricamente mediante un esquema Runge Kutta de 4º orden.

En la Figura 6 recogen los resultados obtenidos. Al haberse empleado una partición temporal equiespaciada, se aprecia cómo en las zonas en las que los cuerpos se mueven más rápido los puntos representados se encuentran más alejados que en las zonas correspondientes a una velocidad menor.



(a) **Simulación 1 en 3D.** (b) **Simulación 1 en 2D.**

Figura 7: Trayectorias para el problema de los 3 cuerpos integrado numéricamente mediante un esquema Leap Frog.

Las mismas condiciones iniciales simuladas con un esquema Leap Frog, representadas en la Figura 7, generan trayectorias diferentes a las recogidas en las Figuras 6a y 6b. Esta diferencia podría atribuirse al mayor orden de aproximación del esquema Runge Kutta y al hecho de que al no filtrarse las soluciones espúreas del Leap Frog estas pueden producir alteraciones significativas en la solución numérica obtenida.

3.3. Hito 6

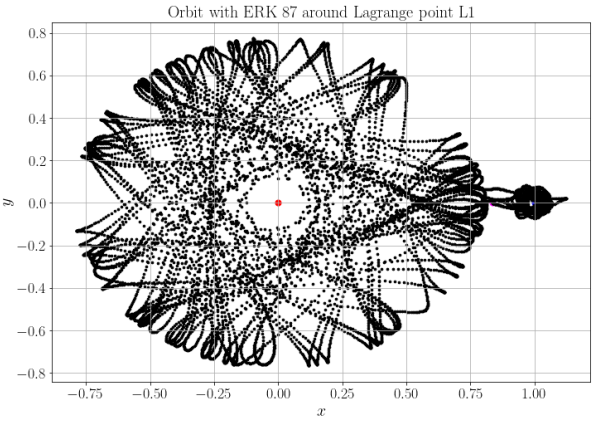
En la Figura 8 se muestra el mensaje que sale por pantalla al ejecutar el código del Extracto de código 7, en el que se recoge la parte real de los autovalores del Jacobiano resultante de linealizar el sistema de los tres cuerpos alrededor de los distintos puntos de equilibrio.

```
For Lagrange point L1::
  Re(λ_0) = -2.9320560087038396
  Re(λ_1) = 2.9320560087038383
  Re(λ_2) = -2.7755575615628914e-17
  Re(λ_3) = -2.7755575615628914e-17
  Re(λ_4) = 0.0
  Re(λ_5) = 0.0
For Lagrange point L2::
  Re(λ_0) = -2.1586744830522546
  Re(λ_1) = 2.1586744830522537
  Re(λ_2) = -3.3306690738754696e-16
  Re(λ_3) = -3.3306690738754696e-16
  Re(λ_4) = 0.0
  Re(λ_5) = 0.0
For Lagrange point L3::
  Re(λ_0) = 7.025630077706069e-17
  Re(λ_1) = 7.025630077706069e-17
  Re(λ_2) = -0.17787544337210673
  Re(λ_3) = 0.17787544337210653
  Re(λ_4) = 0.0
  Re(λ_5) = 0.0
For Lagrange point L4::
  Re(λ_0) = 2.679695223786749e-06
  Re(λ_1) = 2.679695223786749e-06
  Re(λ_2) = -2.67969522396716e-06
  Re(λ_3) = -2.67969522396716e-06
  Re(λ_4) = 1.1102230246251565e-16
  Re(λ_5) = 1.1102230246251565e-16
For Lagrange point L5::
  Re(λ_0) = -2.679695223786749e-06
  Re(λ_1) = -2.679695223786749e-06
  Re(λ_2) = 2.67969522396716e-06
  Re(λ_3) = 2.67969522396716e-06
  Re(λ_4) = -1.1102230246251565e-16
  Re(λ_5) = -1.1102230246251565e-16
```

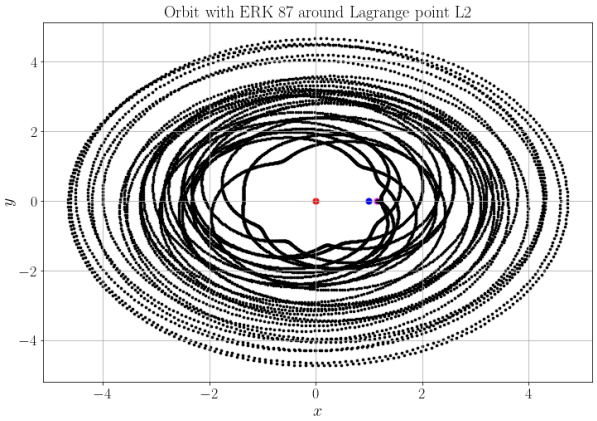
Figura 8: Parte real de los autovalores de la matriz Jacobiana del problema de los 3 cuerpos linealizada alrededor de los 5 puntos de Lagrange.

En la Figura 9 se recogen las órbitas simuladas alrededor de los puntos de Lagrange inestables y en la Figura 10 se muestran las órbitas estables. En estas gráficas la Tierra y la Luna se han pintado como masas puntuales de color azul y rojo, mientras que cada punto de Lagrange se ha pintado de magenta, aunque a veces estos puntos quedan camuflados por los puntos en los que se ha calculado la trayectoria del tercer cuerpo.

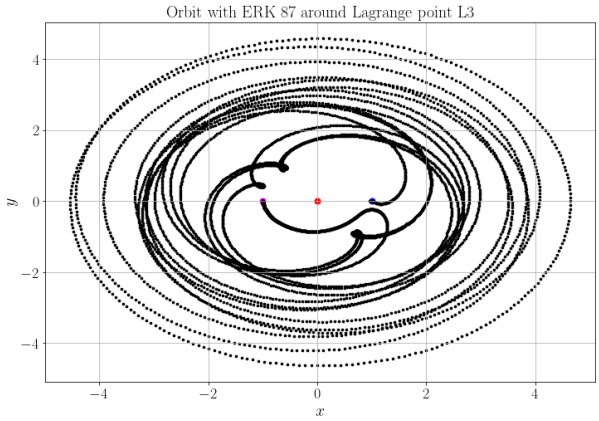
Los puntos inestables presentan todos ellos al menos un autovalor con parte real positiva de orden unidad. Esto es lo que los hace inestables, pues la inestabilidad del sistema linealizado implica inestabilidad del sistema completo. Por otra parte, en los puntos estables L4 y L5 la parte real positiva de los autovalores es de orden 10^{-6}. Este valor puede ser debido a la imprecisión del modelo empleado, ya que estos puntos son conocidos por ser estables y, por tanto, la parte real de sus autovalores nunca podría ser positiva. Como se aprecia en la Figura 10 las órbitas alrededor de estos puntos son similares y se mantienen a una distancia acotada del punto de equilibrio, verificando la definición de estabilidad.



(a) **Órbita alrededor de L1.**

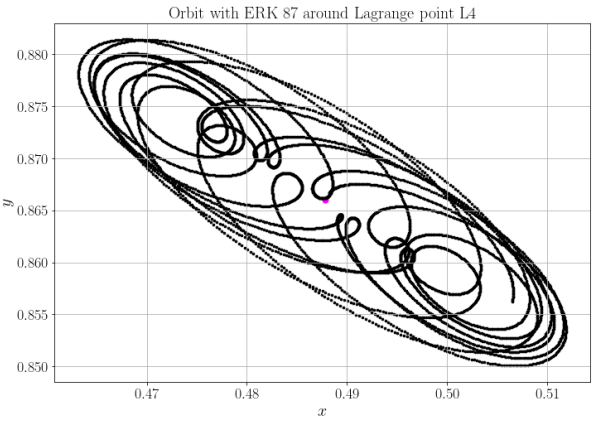


(b) **Órbita alrededor de L2.**

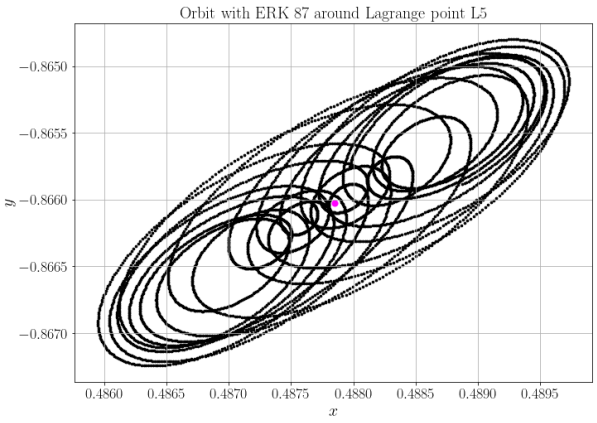


(c) **Órbita alrededor de L3.**

Figura 9: Órbitas alrededor de los puntos de Lagrange inestables del sistema Tierra-Luna: L1, L2 y L3. Simulaciones realizadas con $\Delta t = 0{,}025$.



(a) **Órbita alrededor de L4.**



(b) **Órbita alrededor de L5.**

Figura 10: Órbitas alrededor de los puntos de Lagrange estables del sistema Tierra-Luna: L4, y L5. Simulaciones realizadas con $\Delta t = 0{,}025$.

4. Conclusiones

1. Del Hito 4, se extrae la conclusión de que el hecho de que los autovalores asociados al problema lineal planteado estén dentro de la región de estabilidad del esquema numérico empleado no asegura que la solución numérica tienda a la solución correcta, ya que estabilidad no implica estabilidad asintótica. Como se ha comentado en clase en reiteradas ocasiones, en problemas no disipativos en los que se conserva la energía, como la mecánica orbital, es necesaria la utilización de esquemas numéricos que permitan que los autovalores de la matriz linealizada del problema diferencial objetivo se ubiquen en la frontera de la región de estabilidad del esquema numérico para así conservar la física del problema y que el esquema numérico no introduzca alteraciones en la energía del sistema.

   El estudio de ramas de las matemáticas como las ecuaciones en diferencias destaca en este Hito pues permite conocer por qué se generan soluciones espúreas para ciertas condiciones iniciales cuando se emplea el esquema Leap Frog. Por esto mismo me gustaría resaltar la importancia de las ecuaciones en diferencias, que resultan de gran utilidad a la hora de modelizar una ecuación diferencial e implementar así un método numérico de simulación.

2. Respecto al Hito 5, se han realizado varias simulaciones para el problema de los 3 cuerpos con distintas condiciones iniciales que pueden consultarse en el código `Milestone_V.py` y la conclusión más destacable es la importancia de confiar en las funciones anteriormente programadas ya que este Hito depende de ellas. El haber seguido una filosofía de programación funcional ha permitido la reutilización de funciones programadas con anterioridad, consiguiendo así un repertorio de código versátil que será reutilizado en el futuro. Además, el haber comenzado a escribir funciones más simples y haberlas ido verificando y *testeando* una a una permite construir códigos más complejos sobre una base sólida en la que se confía y que funciona.

   A parte de lo anterior, técnicas de programación más avanzadas como el uso de punteros permiten no solo resolver problemas complejos de manera eficiente y sin duplicar la memoria empleada, sino evitar y prevenir problemas resultantes de no comprender correctamente su funcionamiento y sus propiedades.

3. En la que concierne al Hito 6, me gustaría destacar la utilidad de todas las funciones construidas con anterioridad. El hecho de haber factorizado las tareas y haber seguido el paradigma de programación funcional ha permitido hacer uso de multitud de estas funciones creadas hace meses para la resolución de problemas actuales, como por ejemplo el cálculo de la matriz Jacobiana que se ha usado en este Hito. Además la implementación del Hito ha sido relativamente rápida pues sus principales pilares habían sido programados con anterioridad, acelerando mucho una tarea que habría sido mucho más exigente si se hubiese tratado de realizar directamente desde el principio.

En la Subsección 2.8 se han enumerado varias caminos de mejora del código para futuro que no se han podido llevar a cabo por desconocimiento o por falta de tiempo. Estas mejoras propuestas harían que el software desarrollado tuviera unas prestaciones más elevadas y lo haría una base más robusta para seguir implementando herramientas a lo largo del tiempo.

Gracias a la estructura del código, estas modificaciones tan solo alterarían el interior de las funciones que se desean actualizar, sin afectar en nada al resto de código generado y sin requerir un proceso de integración posterior a la actualización.

Como conclusiones generales sobre lo aprendido en la asignatura, como ya se ha comentado en la Subsección 2.7, la programación funcional es una forma de plantear el código en la que se ha mejorado durante el desarrollo de la asignatura. Lo que al principio había que pensar (qué jerarquía seguía el código, en qué módulos agrupar cada función, étc.) se ha ido realizando de forma más inconsciente según iba aumentando la familiaridad con una programación enfocada en *qué hacer* en vez de en el *cómo hacerlo*. Además de la facilidad para realizar estas abstracciones antes de comenzar a programar, se ha ido aumentando el nivel de complejidad de estas. Este conocimiento, subyacente a la asignatura, ha trascendido más allá de los Hitos pues se ha recurrido a él en otros trabajos de diferentes asignaturas del máster, dando muy buenos resultados.

Por todo lo anterior, las principales ventajas que se le atribuyen al paradigma de programación funcional tras haberlo puesto en práctica durante la asignatura, son: la capacidad de reutilización de funciones programadas anteriormente en las que se confía, la capacidad de realizar modificaciones y actualizaciones en funciones particulares sin afectar a su interacción con el resto de funciones, la claridad y brevedad del código compactado dentro de cada función y la estructura mental que se refuerza con cada abstracción que permite desarrollar nuevo código cada vez más complejo.

Referencias

[1] Hernández, Juan Antonio, *Cálculo Numérico en Ecuaciones Diferenciales Ordinarias*, 2018.

[2] Hernández, Juan Antonio & Escoto, F. Javier, *How to learn Applied Mathematics through modern FORTRAN*, 2017.

[3] Hernández, Juan Antonio & Rapado, Miguel, *Advanced Programming for Numerical Calcualtions*, 2022.