


Universidad Politécnica de Madrid

Ampliación de Matemáticas I

Informe Hito 3: Estimacion del error en Soluciones Analiticas.

Autora:
Inés Arauzo Andrés

Índice

Índice de figuras

1. Introducción

A lo largo del presente informe se pretenden comentar los resultados obtenidos durante la realización de los Hitos 4, 5 y 6 correspondientes a la asignatura de Ampliación de Matemáticas I.

2. Definición de los problemas

2.1. Hito 4

El objetivo de este Hito es integrar el problema del oscilador armónico no amortiguado para unas condiciones iniciales dadas. Fisicamente, el planteamiento es el siguiente:

$$\ddot{x} + x = 0 \tag{2.1.1}$$
$$x(0) = 1 \tag{2.1.2}$$
$$\dot{x}(0) = 0. \tag{2.1.3}$$

Numéricamente, se plantea el vector de estado

$$U = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}, \tag{2.1.4}$$

y numéricamente, el problema queda definido como

$$F(U,t) = \frac{dU}{dt} = \frac{d}{dt}\begin{pmatrix} x \\ \dot{x} \end{pmatrix} = \begin{pmatrix} \dot{x} \\ -x \end{pmatrix} \tag{2.1.5}$$

con sus correspondientes condiciones iniciales

$$U_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \tag{2.1.6}$$

Partiendo de esto, se pretende aplicar los esquemas numericos comentados en trabajos anteriores para ver la variación de los resultados con los mismos, así como estudiar las regiones de estavilidad de los diferentes integradores temporales.

2.2. Hito 5

En el Hito 5 se ha integrado el problema de los N cuerpos. Este problema considera N puntos de masa m_i y posición definida por los vectores $r_i, \quad i = 1, 2, ..., N$ en un espacio tridimenisonal que se mueven por la atracción gravitacional que generan entre sí. De este modo,la fuerza que experimenta una particula de masa m_i debido a la presencia de otra masa m_j queda definida como:

$$m_i \frac{d^2 r_i}{di^2} = \sum_{i=1, i \neq j}^{N} m_i m_j \frac{(r_j - r_i)}{\|r_j - r_i\|^3} \tag{2.2.1}$$

Partiendo de la Equation 2.2.1 se obtiene el vector de estado **U**, formado por las posiciones y velocidades de cada partícula, así como el vector **F(U, t)**,

$$U(t) = \begin{pmatrix} x_1 \\ \dot{x}_1 \\ y_1 \\ \dot{y}_1 \\ z_1 \\ \dot{z}_1 \\ . \\ . \\ . \\ x_N \\ \dot{x}_N \\ y_N \\ \dot{y}_N \\ z_N \\ \dot{z}_N \end{pmatrix}, \quad F(U,t) = \begin{pmatrix} \dot{x}_1 \\ \ddot{x}_1 \\ \dot{y}_1 \\ \ddot{y}_1 \\ \dot{z}_1 \\ \ddot{z}_1 \\ . \\ . \\ . \\ \dot{x}_N \\ \ddot{x}_N \\ \dot{y}_N \\ \ddot{y}_N \\ \dot{z}_N \\ \ddot{z}_N \end{pmatrix} \tag{2.2.2}$$

2.3. Hito 6

Finalmente, el Hito 6 calcula los puntos de Lagrange del sistema Tierra-Sol y linealiza la Equation 2.2.1 alrededor de los mismos, para a continuación, calcular las órbitas entorno a ellos mediante un esquema Runge-Kutta de paso adaptativo (*Embedded Runge-Kutta*).

3. Updates del código respecto a Hitos anteriores

De cara a realizar los tres problemas que se han planteado, ha sido necesario desarrollar nuevas funciones y programas que se explican a continuación.

3.1. Mains: *Milestone4.py, Milestone5.py y Milestone6.py*

En primer lugar, se han desarrollado tres programas nuevos desde los que se ejecutan los problemas planteados en subsection 2.1, subsection 2.2 y subsection 2.3, *Milestone4-py, Milestone5-py y Milestone6-py* respectivamente. Estos archivos no llevan a cabo prácticamente ninguna operación,puesto que únicamente llaman al ciertos módulos que a su vez llamarán a otros siguiendo una estructura lógico-matemática. Con ello se ha conseguido que los mains sean archivos muy escuetos y comprensibles en los que la mayor parte de las líneas de código sirve para plotear resultados. En este sentido, considero que aunque queda mucho donde pulir, hay una mejora visible respecto a otros hitos.

No obstante, creo importante destacar que el ploteo en ciertas ocasiones no es una tarea tan simple como parece, consumiendo una gran cantidad de tiempo para obtener unos resultados comprensibles.

3.2. *resources.Physics*

Este módulo se empleó ya con los Hitos 2 y 3, y contiene la s funciones que pueden usarse como lado forzante de una EDO de orden n.

Para estos proyectos se ha actualizado, añadiendo las funciones del oscilador armónico, el problema de los N cuerpos así como el de los 3 cuerpos restringido, que se muestran a continuación,

```
def Harmonic_Oscilator(U, t):
    amplitude  = 1
    return array([U[1], -amplitude*U[0]])
```

Algoritmo 3.1: Harmonic oscilator function

```
def N_Body_Problem(U,t):
    """The N-body problem is the problem of predicting the
    individual motions of a
        group of mass objects under the influence of their
    gravitational field
        The function F_NBody uses the state vector U as an input
         and returns its derivative F

    Args:
```

```
8         U (array): state vector (position_i, velocity_i)
9         t (array): time partition
10
11     Returns:
12         F(array): derivative of U (velocity_i, acceleration_i)
13     """
14
15     Nb = 4                                    # Number of bodies
16     dim = 3                                   # Dimensions of the
   problem
17
18
19     Us = reshape(U, (Nb, dim, 2))             # reshape U[Nb*dim*2,
   1] into Us[Nb, variable(x,y,z), position(0)/velocity(1)]
20     F  = zeros(len(U))
21     Fs = reshape(F, (Nb, dim, 2))             # reshape F[Nb*dim*2,
   1] into Fs[Nb, variable(x,y,z), velocity(0)/acceleration(1)]
22
23     r  = reshape(Us[:, :, 0],(Nb, dim))       # saves the position of
    every body r[body_index, position]
24     v  = reshape(Us[:, :, 1],(Nb, dim))       # saves the velocity of
    every body v[body_index, velocity]
25
26     drdt = reshape(Fs[:, :, 0], (Nb, dim))
27     dvdt = reshape(Fs[:, :, 1], (Nb, dim))
28
29     dvdt[:, :] = 0
30
31     for i in range(Nb):
32
33         drdt[i, :] = v[i, :]
34
35         for j in range(Nb):
36
37             if j != i:                        # Only aplicable for
   different bodies
38
39                 d = r[j, :] - r[i, :]
40                 dvdt[i, :] = dvdt[i, :] + d[:]/(norm(d)**3)
41
42     return F
```

Algoritmo 3.2: N-body problem function

A lo largo de este código se emplean repetidamente los punteros (mediante los *reshape* con el objetivo de ahorrar memoria y por tanto tiempo de ejecución del código. Además, estos tienen la ventaja añadida de que hacen que el código final sea mucho más legible que si se hiciese de forma matricial directamente.

Sin embargo, cabe destacar la dificultad que se tuvo a la hora de implementar correctamente los *reshape*. Con el desarrollo de esta función quedó clara la necesidad imperante de entender bien tanto la física como la matemática de un problema antes

de pasar a programarlo.

```
def R3BodyProblem(U, t):
    mu =3.0039e-7# 0.0121505856

    x =  U[0]
    y =  U[1]
    vx = U[2]
    vy = U[3]

    d = sqrt( (x + mu)**2 + y**2 )
    r = sqrt( (x - 1 + mu)**2 + y**2 )


    ax = x + 2*vy - (1 - mu)*( x + mu )/d**3 - mu*(x - 1 + mu)/r**3
    ay = y - 2*vx - (1 - mu) * y/d**3 - mu * y/r**3

    return array( [ vx, vy, ax, ay ] )
```

Algoritmo 3.3: Restricted 3-body problem function

Por otra parte, aunque sea una mejora menor, se ha empezado a usar el *docstring*, que facilita la comprensión del código.

3.3. *Lagrange-Points-py*

Para el cálculo de los puntos de Lagrange en el Hito 6 se ha desarrollado un nuevo módulo, que contiene dos funciones, una relativa al cálculo de los puntos y otra que abarca el análisis de estabilidad de los mismos (linealiza al problema empleando el Jacobiano desarrollado para hitos anteriores y obteniendo los autovalores y autovectores de los mismos.

```
def Lagrange_Points(U0, NL):

    LP = zeros([5,2])

    def F(Y):

        X = zeros(4)
        X[0:2] = Y
        X[2:4] = 0
        FX = R3BodyProblem(X, 0)
        return FX[2:4]

    for i in range(NL):
        LP[i,:] = fsolve(F, U0[i,0:2])

    return LP
```

Algoritmo 3.4: Harmonic oscilator function

```
def Lagrange_Points_Stability(U0):

    def F(Y):
        return R3BodyProblem(Y, 0 )

    A = Jacobian(F, U0)
    values, vectors = eig(A)

    return values
```

Algoritmo 3.5: Harmonic oscilator function

Ambas funciones llaman recursivamente a *R3BodyProblem*; es decir, esta función llama a un problema más genérico como es el de los 3 cuerpos para obtener unos puntos concretos.

3.4. *resources-Time*$_{schemes} - py$

Finalmente, se ha actualizado el módulo de esquemas temporales, añadiendo el método Leapfrog que se pedía en el Hito 4. Matemáticamente es tal que

$$U^{n+1} = U^{n-1} + 2\Delta t \tag{3.4.1}$$

Y el código, al igual que todos los esquemas del módulo tiene como argumentos U, F, t y dt y como salida la U en en tiempo siguiente

```
def LeapFrog (U, F, t, dt):
     LeapFrog.__name__ = "Leapfrog"
     if t == 0:
         U = U + dt*F(U, t)
     else:
        U_aux = U
        aux = F (U_aux, t)

        L_2 = int(len(U)/2)

        U_aux [L_2 :] = U_aux [L_2 :] + aux[L_2 :]*dt/2
        U_aux [: L_2] = U_aux [: L_2] + aux[: L_2]*dt

        aux = F (U_aux, t)
        U_aux [L_2 :] = U_aux [L_2 :] + aux[L_2 :]*dt/2
        U = U_aux
     return U
```

Algoritmo 3.6: Esquema Leapfrog

Se ha incluido también en este módulo la función del Runge Kutta embebido, que es a grandes rasgos un método basado en los RK clásicos pero que necesita menos

tiempo de cálculo para conseguir la solución numérica utilizando una interpolación de 3 orden.

Este esquema emplea unos sumatorios de coeficientes fijos para obtener la solución a tiempo $n+1$. Estos se pueden organizar en una matriz (matriz de Butcher), y para este caso se han incluido 6 distintas:

- Euler
- Shampine
- RK12
- PrinceDormand
- CashKarp
- Felberg

No se ha incluido el código ya que la mayor parte del mismo son matrices copiadas de internet, por lo que no aporta mucho conocimiento ni a nivel práctico ni a nivel funcional.

3.5. Jerarquía del código

Sabiendo las actualizaciones en el código, se han realizado unos breves esquemas para saber como se esquematizan:

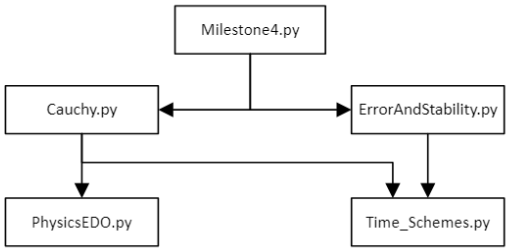


Figura 3.1: Estructura funcional del Milestone 4

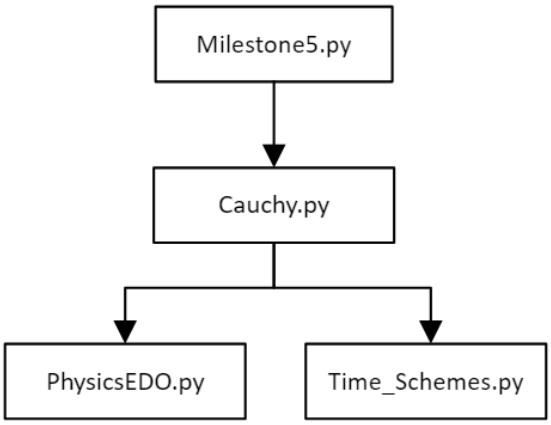


Figura 3.2: Estructura funcional del Milestone 5

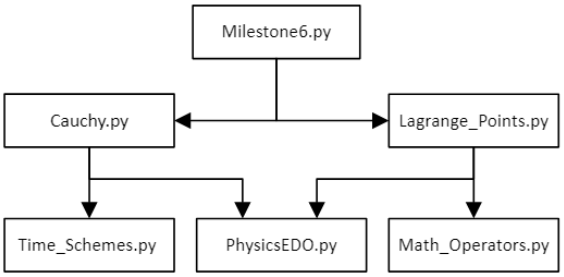


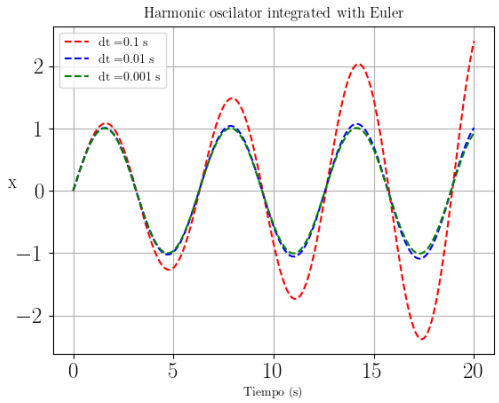
Figura 3.3: Estructura funcional del Milestone 6

Tal y como se puede ver, los tres códigos tienen una parte exactamente igual, ya que en todos ellos parte del problema a resolver era un problema de Cauchy con diferentes funciones. Es esta estructuración ordenada la que nos ha permitido hacer unos archivos main muy compactos y legibles para un usuario ajeno al problema.

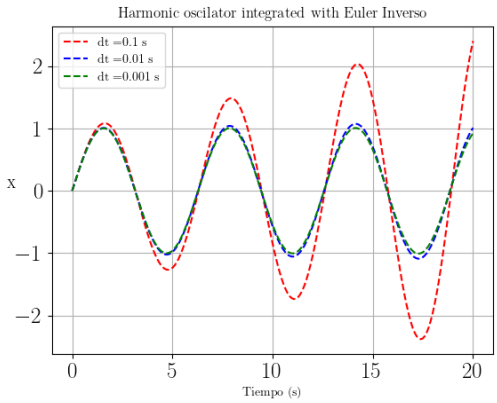
4. Resultados

4.1. Hito 4

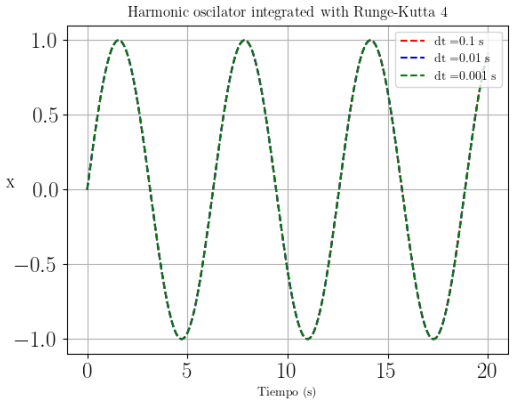
En la Equation 2.1.3 se plantea el problema del oscilador armónico con sus respectivas CI no nulas. La Figure 4.1 muestra la solución numérica a este problema integrada mediante distintos esquemas numéricos



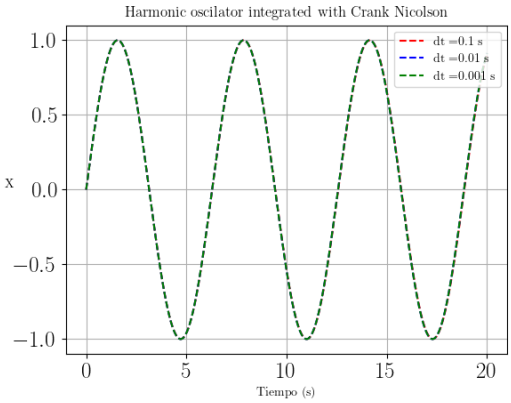
(a) Integrado con Euler



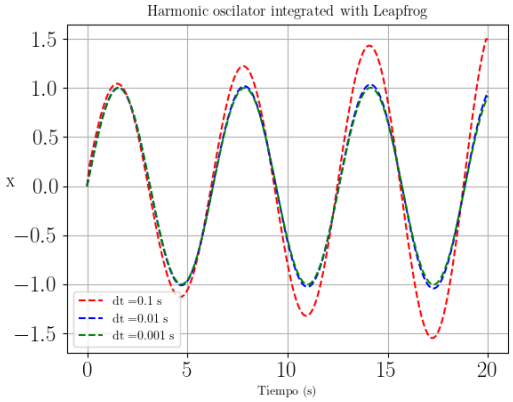
(b) Integrado con Euler Inverso



(c) Integrado con RK4



(d) Integrado con CN



(e) Integrado con Leapfrog

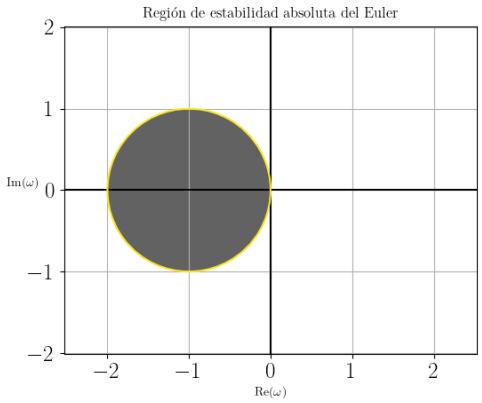
Figura 4.1: Osciladores armonicos integrados con distintos esquemas temporales

Los resultados son muy similares a los del Hito 3, en los que se utilizaban estos esquemas para integrar órbitas:

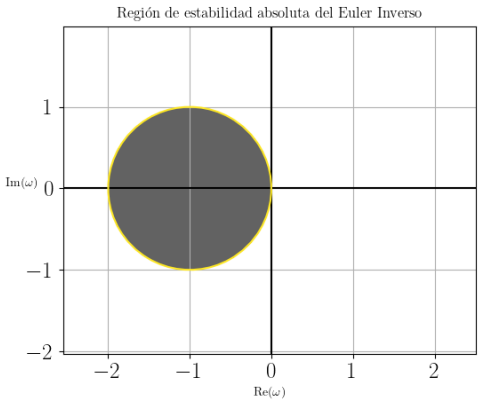
- El esquema Euler diverge totalmente, incluso para mallados finos, puesto que

los autovalores del problema están fuera de la región de estabilidad, tal y como se ve en Figure **??**.

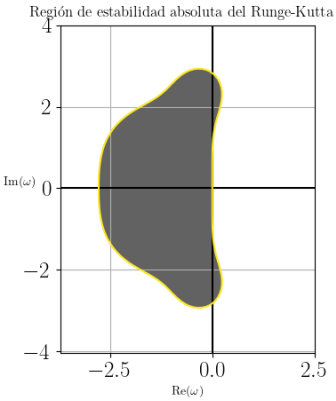
- Análogamente al caso anterior, el esquema Euler inverso también diverge, pero en esta ocasión se debe a que los autovalores del problema caen dentro de la región de estabilidad, en lugar de en la frontera, por lo que para todo Δt disipa energía.
- El esquema Runge-Kutta da un resultado muy bueno para tiempos coherentes (esto es, menores de 1, en cuyo caso comenzaría a disipar energía pues los autovalores se moverían al interior de la región de estabilidad).
- El esquema Crank-Nicolson, ofrece un resultado casi perfecto en módulo debido a que los autovalores del problema caen en el eje imaginario, esto es, en la frontera de estabilidad del esquema.
- Por su parte, el esquema Leapfrog consigue unos resultados muy buenos cuando se introduce un pequeño error, debido a que es un esquema GBS.



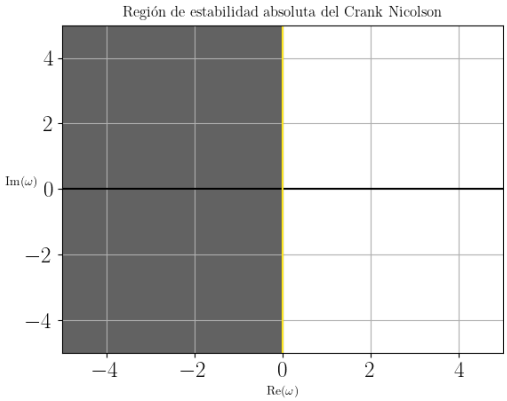
(a) Esquema de Euler



(b) Esquema de Euler Inverso



(c) Esquema RK4



(d) Esquema de CN

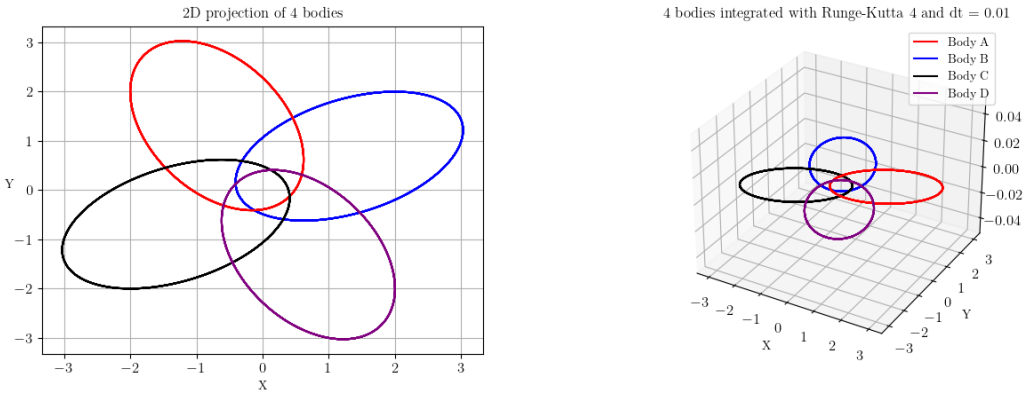
Figura 4.2: Regiónes de estabilidad de distintos esquemas temporales

4.2. Hito 5

Para este hito se ha integrado el problema de los N cuerpos planteado en la subsection 2.2 mediante un esquema Runge-Kutta 4. Cabe destacar que en el código se permite al usuario escoger el numero de cuerpos, así como diferentes condiciones iniciales. No obstante, por simplicidad y compacidad del presente informe, se ha optado por incluir solo el problema de los 4 cuerpos con las siguientes condiciones iniciales,

	r_i			v_i		
1	2	2	0	-0.4	0	0
2	-2	2	0	0	-0.4	0
3	-2	-2	0	0.4	0	0
4	2	-2	0	0	0.4	0

Los resultados para estas condiciones iniciales son,



(a) Proyección 2D (b) Visión 3D

Figura 4.3: Problema de los 4 cuerpos integrado con RK4

4.3. Hito 6

Los resultados de las órbitas entorno a los puntos de Lagrange integrados con el Runge Kutta embebido se muestran a continuación,

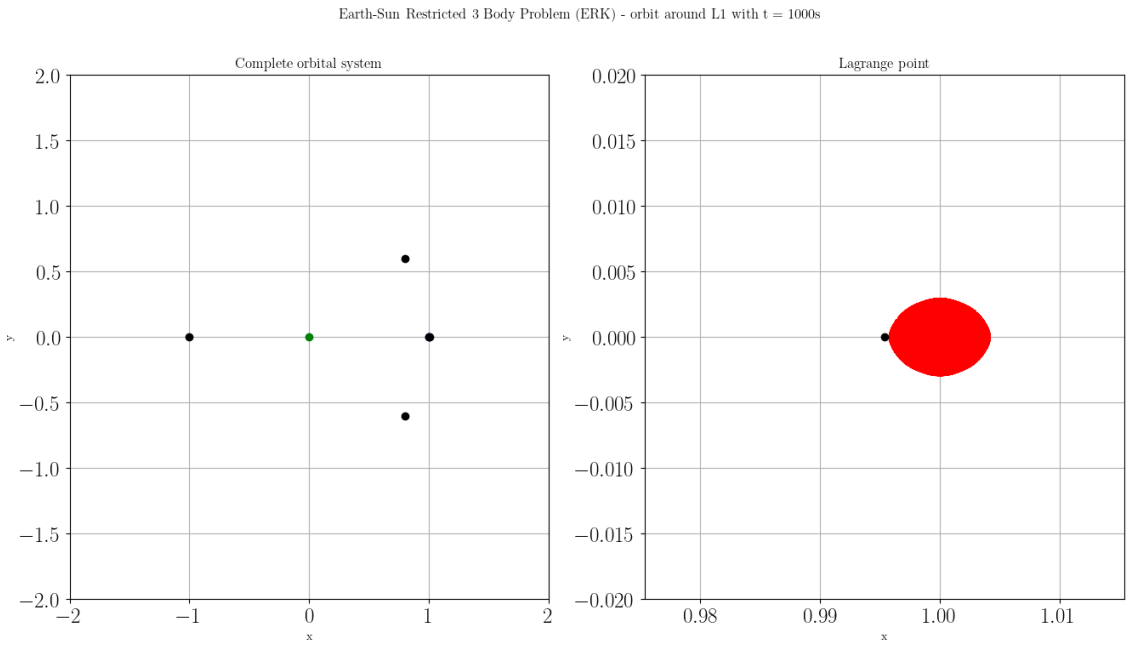


Figura 4.4: Orbitas entorno al primer punto de Lagrange integradas con Runge Kutta embebido

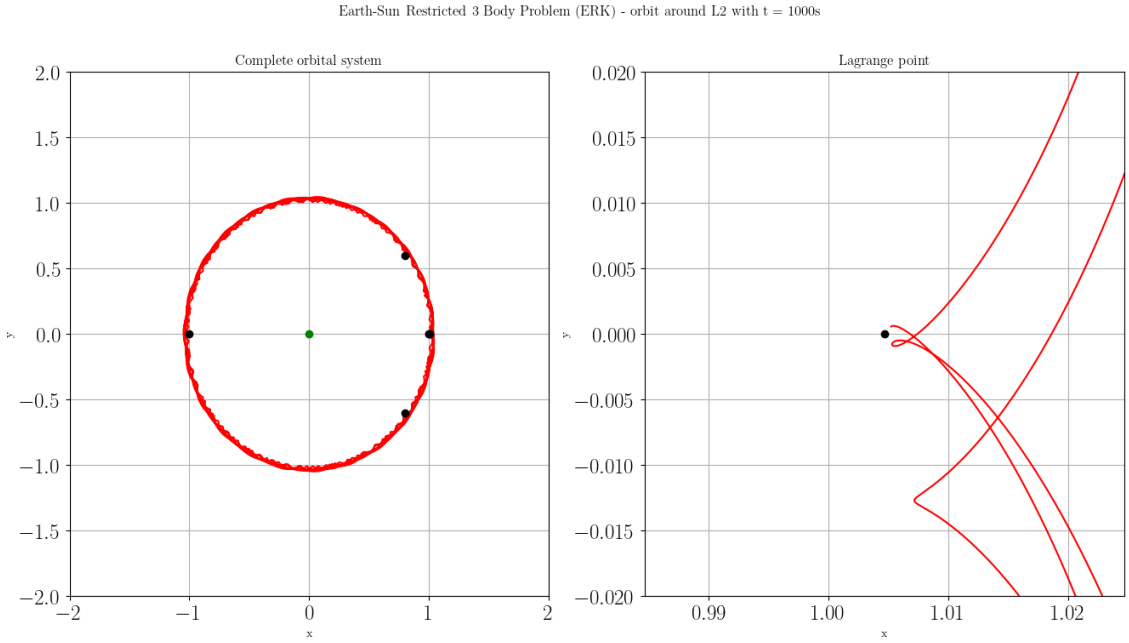


Figura 4.5: Orbitas entorno al segund punto de Lagrange integradas con Runge Kutta embebido

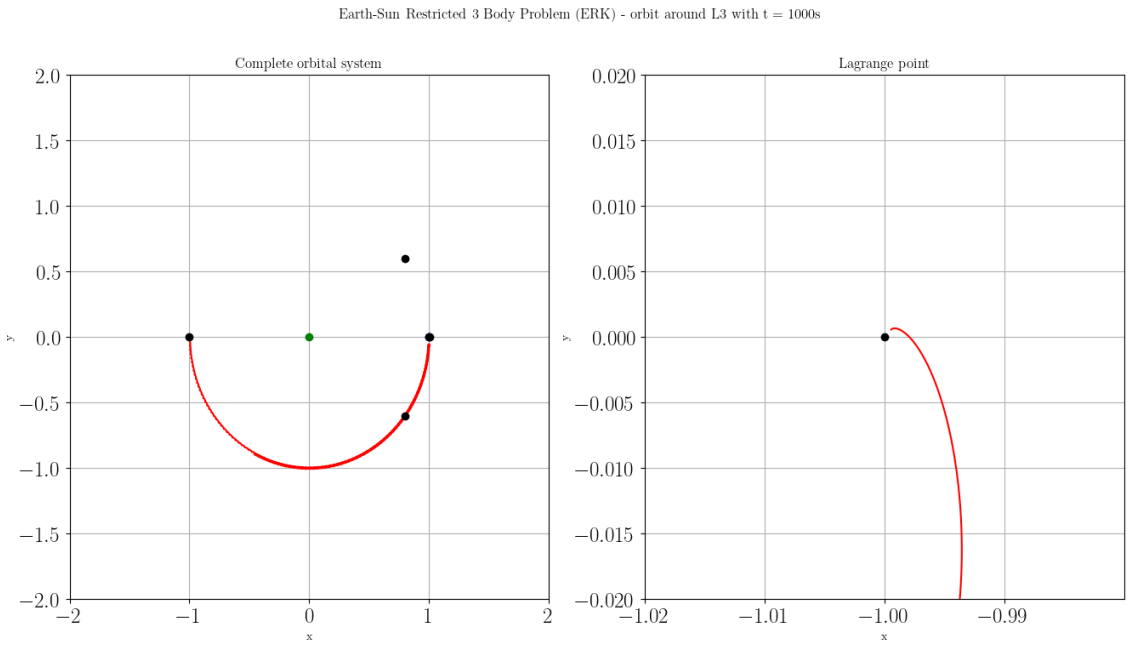


Figura 4.6: Orbitas entorno al primer punto de Lagrange integradas con Runge Kutta embebido

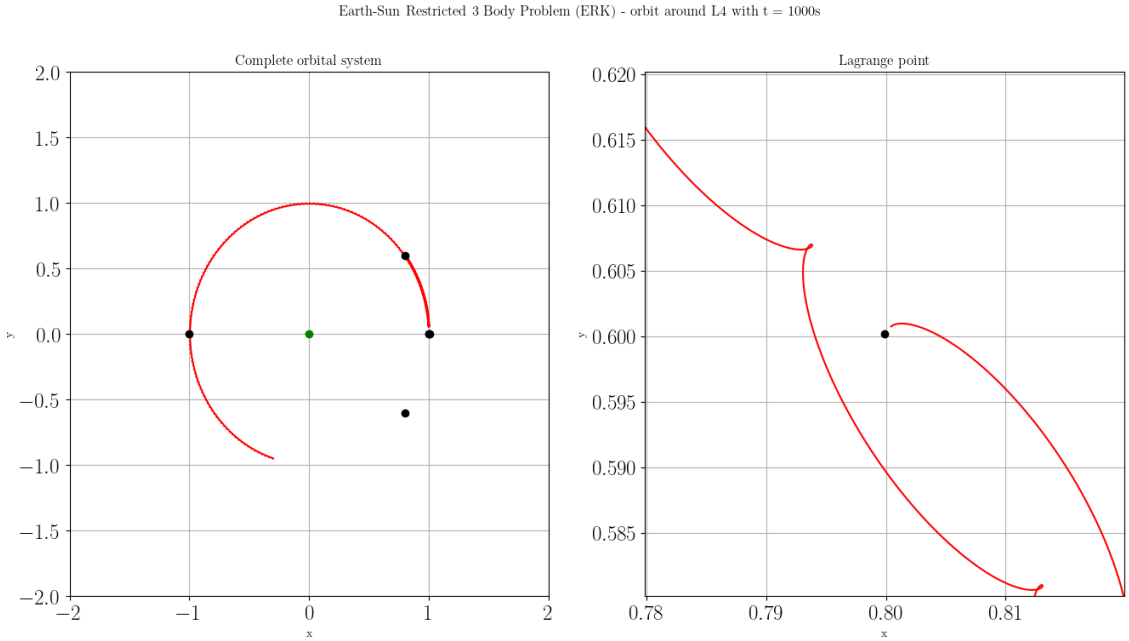


Figura 4.7: Orbitas entorno al primer punto de Lagrange integradas con Runge Kutta embebido

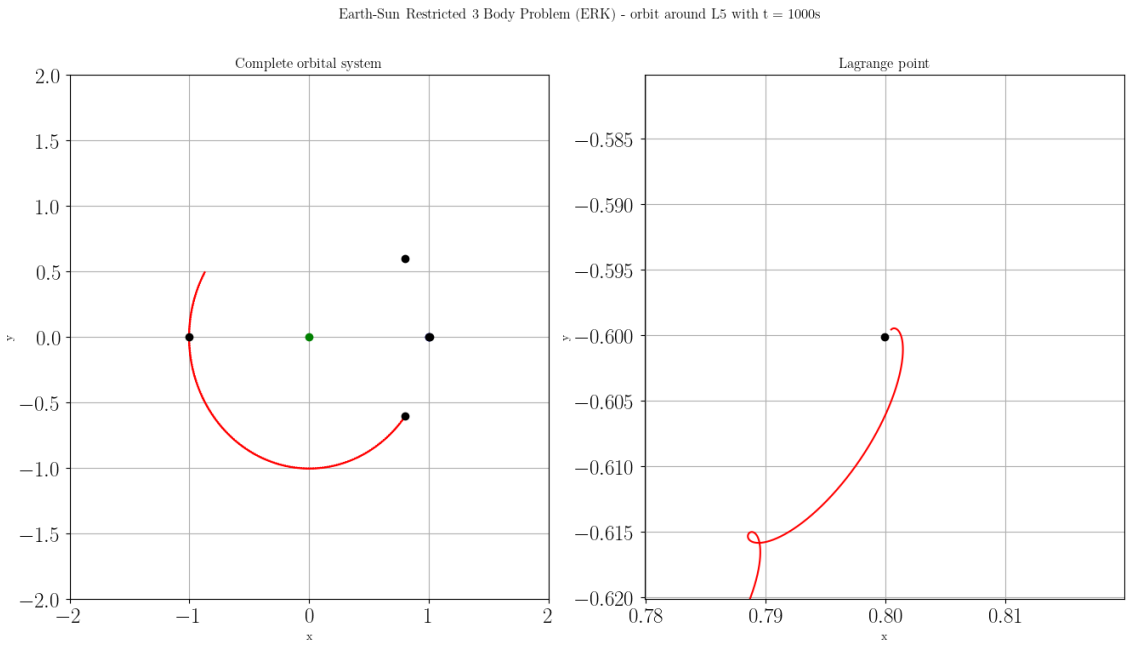


Figura 4.8: Orbitas entorno al primer punto de Lagrange integradas con Runge Kutta embebido

5. Conclusiones