


Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

MÁSTER UNIVERSITARIO EN SISTEMAS ESPACIALES

MILESTONE 6

Ampliación de Matemáticas I

4 de diciembre de 2022

Autor:
Alberto García Rincón

Índice

1. Introducción

En este trabajo se ha implementado un método de resolución numérica de alto orden y de paso temporal variable. El método elegido ha sido el método ”Runge-Kutta 87”. Se ha definido también una función que define el problema restringido de los tres cuerpos.

A continuación se han obtenido los puntos de Lagrange del sistema Tierra-Luna y se ha realizado un estudio sobre la estabilidad de dichos puntos. Finalmente se han simulado y analizado varias órbitas de un cuerpo que se encontraría en las proximidades de cada punto de Lagrange.

2. Código de Python

2.1. Diagrama de bloques

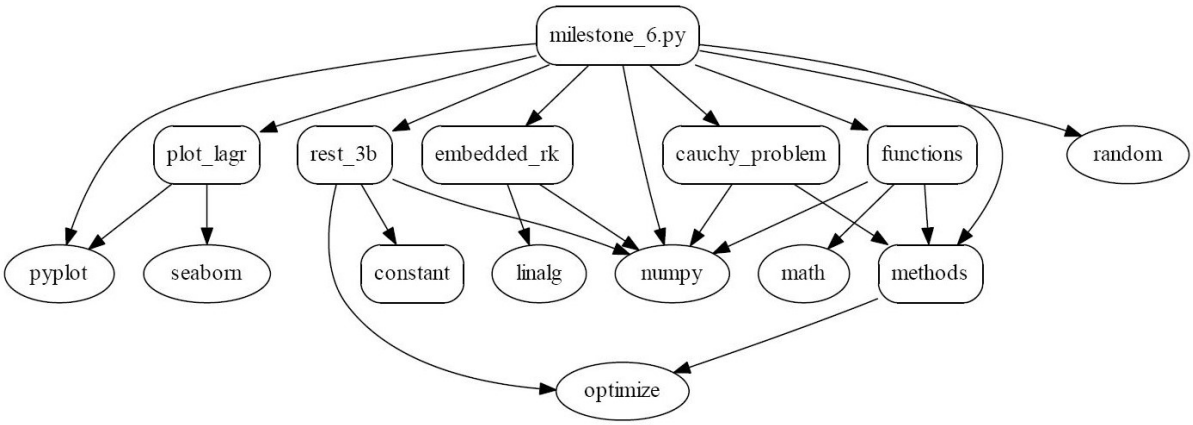


Figura 1: Diagrama módulos Milestone 6.

Como puede observarse en la Figura 1 para hacer funcionar este programa se debe ejecutar el programa principal denominado $milestone\_6.py$. Este programa se encarga de llamar al resto de funciones definidas en los respectivos archivos.

Se van a comentar las nuevas funciones implementadas para la resolución de este problema.

2.2. milestone_6.py

En la Figura 2 se muestra el código que define las condiciones iniciales para realizar las simulaciones.

```
15    ##Initialize values
16    Dt = 0.001      #[s]
17    tf = 100        #[s]
18    N = int(tf/Dt)+1    #Nº steps
19    t = np.linspace(0, tf, N)
20
21    Nl = 5 #Nº lagrange points
22    Nc = 3 #Nº coord
23    Lp_U0 = init_lagr_points(Nl, Nc)
24
25    ##Choose Numerical Method
26    mets = [met.explicit_euler, met.runge_kutta4, met.inverse_euler, met.crank_nicolson, met.leap_frog, embedded_rk]
27    i = 5 #RK Embedded (5)
```

Figura 2: Condiciones iniciales Milestone 6.

En la Figura 3 se muestra el código que llama a la función encargada de determinar las coordenadas de los puntos de Lagrange del sistema Tierra-Luna definido.

```
30    ##Lagrange Points
31    Lp = r3b.lagr_points(Lp_U0, Nl, Nc)
32    plag.plot_lagr(Lp)
```

Figura 3: Código para calcular los puntos de Lagrange.

En la Figura 4 se muestra el código que calcula una órbita en las proximidades del punto de Lagrange, lo hace para cada uno de los puntos de Lagrange calculados anteriormente.

Se define con la función $uniform()$ un punto inicial de la órbita que se encuentra próximo al punto de Lagrange objeto de estudio. Las componentes de velocidad de ese cuerpo se inicializan en cero.

```
36    ##Orbits around Lagrange Points
37    U0 = np.empty([Nl, 2*Nc])
38    UU = np.empty([Nl], dtype=type(U0))
39
40        #Initialize orbits around Langrage Points
41    rand = uniform(0, 1e-5)
42    U0[:,0:Nc] = Lp + rand  #Position Lagr Points
43    U0[:,Nc:2*Nc] = 0       #Vel Lagr Points
44
45    for j in range(Nl):
46        U = cauchy_problem(U0[j,:], t, r3b.rest_3b, mets[i])
47        UU[j] = U
48
49        plag.plot_one_orb_lagr(Lp[j], U, j+1, mets[i])
50
51        #Plots all orbits around Lagrange Points
52    plag.plot_orb_lagr(Lp, UU, mets[i])
```

Figura 4: Código para calcular las órbitas alrededor de los puntos de Lagrange.

En la Figura 5 se muestra el código para calcular y representar los autovalores de cada punto de Lagrange.

```
55      ##Stability Langrage Points
56      Lp0 = np.empty([Nl, 2*Nc])
57      Lp0[:,0:Nc] = Lp    #Position Lagr Points
58      Lp0[:,Nc:2*Nc] = 0  #Vel Lagr Points
59
60      for k in range(len(Lp0)):
61          val, vect = r3b.lagr_points_stab(Lp0[k])
62          plag.plot_stab(val, k+1)
```

Figura 5: Código para calcular la estabilidad de los puntos de Lagrange.

2.3. embedded_rk.py

Las Figuras 6 y 7 muestran el código del método Runge-Kutta embebido que se ha programado.

Cabe destacar la función $step\_size()$ que define el Δt que se va a utilizar en cada paso de la resolución del proceso. Esto es porque al tratarse de un método de paso temporal variable, es decir que el Δt se ”autocalcula” para cada paso del proceso.

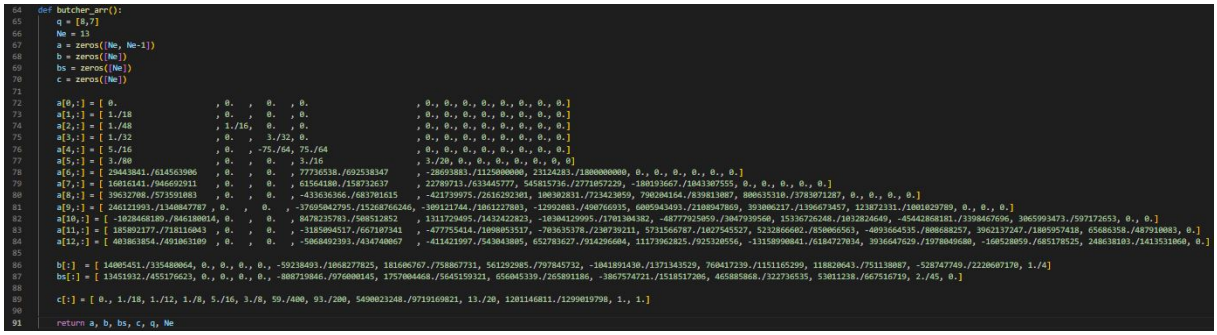
```
5   def embedded_rk(U, dt, t, f):
6       tol = 1e-10
7
8       V1 = rk_scheme(1, U, t, dt, f)
9       V2 = rk_scheme(2, U, t, dt, f)
10
11      a, b, bs, c, q, Ne = butcher_arr()
12
13      h = min(dt, step_size(V1-V2, tol, min(q), dt) )
14
15      N = int(dt/h) + 1
16      h2 = dt/N
17
18      V1 = U
19      V2 = U
20
21      for i in range(N):
22          time = t + i*dt/int(N)
23          V1 = V2
24          V2 = rk_scheme(1, V1, time, h2, f)
25
26      U2 = V2
27
28      return U2
```

Figura 6: Código función método numérico Runge-Kutta embebido.

```
31  def rk_scheme(tag, U1, t, dt, f):
32      a, b, bs, c, q, Ne = butcher_arr()
33      k = zeros([Ne, len(U1)])
34
35      k[0,:] = f(U1, t + c[0]*dt)
36
37      for i in range(1,Ne):
38              Up = U1
39              for j in range(i):
40                  Up = Up + dt*a[i,j]*k[j,:]
41
42              k[i,:] = f(Up, t + c[i]*dt)
43
44      if tag == 1:
45          U2 = U1 + dt*matmul(b,k)
46
47      elif tag == 2:
48          U2 = U1 + dt*matmul(bs,k)
49
50      return U2
51
52
53  def step_size(dU, tol, q, h):
54      normT = norm(dU)
55
56      if normT > tol:
57          ss = h*(tol/normT)**(1/(q+1))
58      else:
59          ss = h
60
61      return ss
```

Figura 7: Código función método numérico Runge-Kutta embebido.

La Figura 8 muestra el array de Butcher que se ha utilizado para el método Runge-Kutta definido anteriormente. Se trata de un método de orden 8, denominado $RK87$.



```
def butcher_arr():
    q = [8,7]
    Ne = 13
    a = zeros([Ne, Ne-1])
    b = zeros([Ne])
    bs = zeros([Ne])
    c = zeros([Ne])

    a[0,:] = [ 0.                , 0.   ,  0.   , 0.                  , 0., 0., 0., 0., 0., 0., 0.]
    a[1,:] = [ 1./18             , 0.   ,  0.   , 0.                  , 0., 0., 0., 0., 0., 0., 0.]
    a[2,:] = [ 1./48             , 1./16,  0.   , 0.                  , 0., 0., 0., 0., 0., 0., 0.]
    a[3,:] = [ 1./32             , 0.   ,  3./32, 0.                  , 0., 0., 0., 0., 0., 0., 0.]
    a[4,:] = [ 5./16             , 0.   , -75./64, 75./64             , 0., 0., 0., 0., 0., 0., 0.]
    a[5,:] = [ 3./80             , 0.   ,  0.   , 3./16               , 3./20, 0., 0., 0., 0., 0., 0]
    a[6,:] = [ 29443841./614563906   , 0.   ,  0.   , 77736538./692538347     , -28693883./1125000000, 23124283./1800000000, 0., 0., 0., 0., 0.]
    a[7,:] = [ 16016141./946692911   , 0.   ,  0.   , 61564180./158732637     , 22789713./633445777, 545815736./2771057229, -180193667./1043307555, 0., 0., 0., 0.]
    a[8,:] = [ 39632708./573591083   , 0.   ,  0.   , -433636366./683701615   , -421739975./2616292301, 100302831./723423059, 790204164./839813087, 800635310./3783071287, 0., 0., 0., 0.]
    a[9,:] = [ 246121993./1340847787 , 0.   ,  0.   , -37695042795./15268766246, -309121744./1061227803, -12992083./490766935, 6005943493./2108947869, 393006217./1396673457, 123872331./1001029789, 0., 0., 0.]
    a[10,:] = [ -1028468189./846180014, 0.   ,  0.   , 8478235783./508512852   , 1311729495./1432422823, -10304129995./1701304382, -48777925059./3047939560, 15336726248./1032824649, -45442868181./3398467696, 3065993473./597172653, 0., 0.]
    a[11,:] = [ 185892177./718116043 , 0.   ,  0.   , -3185094517./667187341   , -477755414./1098053517, -703635378./238739211, 5731566787./1027545527, 5232866602./850066563, -4093664535./808688257, 3962137247./1805957418, 65686358./487910083, 0.]
    a[12,:] = [ 403863854./491063109 , 0.   ,  0.   , -5068492393./434740067   , -411421997./543043805, 652783627./914296604, 11173962825./925322556, -13158990841./6184727034, 3936647629./1978049680, -160528059./685178525, 248638103./1413531060, 0.]

    b[:] = [ 14005451./335480064, 0., 0., 0., -59238493./1068277825, 181606767./758867731, 561292985./797845732, -1041891430./1371343529, 760417239./1151165299, 118820643./751138087, -528747749./2220607170, 1./4]
    bs[:] = [ 13451932./455176623, 0., 0., 0., -808719846./976000145, 1757004468./5645159321, 656045339./265891186, -3867574721./1518517206, 465885868./322736535, 53011238./667516719, 2./45, 0.]

    c[:] = [ 0., 1./18, 1./12, 1./8, 5./16, 3./8, 59./400, 93./200, 5490023248./9719169821, 13./20, 1201146811./1299019798, 1., 1.]

    return a, b, bs, c, q, Ne
```

Figura 8: Código función Butcher_array.

2.4. rest_3b.py

En la Figura 9 se expone el código que define el problema de los 3 cuerpos con movimiento circular restringido. Es un caso particular del problema de los N-Cuerpos que se había definido en el anterior trabajo.

Se trata de la función que se va ha utilizar para en las siguientes simulaciones. Los tres cuerpos que se integran en esta función son: la Tierra, la Luna y el tercer cuerpo es un cuerpo que se encuentra en las proximidades del punto de Lagrange objeto de estudio.

```
#restricted 3 body problem
def rest_3b(U, t):
    mu = MU_em
    r = U[0:3]
    v = U[3:6]

    d_ = np.sqrt( (r[0]+mu)**2 + r[1]**2 + r[2]**2 )
    r_ = np.sqrt( (r[0]-1+mu)**2 + r[1]**2 + r[2]**2)

    dvx_dt = r[0] + 2 * v[1] - (1-mu) * ( r[0] + mu )/d_**3 - mu*(r[0]-1+mu)/r_**3
    dvy_dt = r[1] - 2 * v[0] - (1-mu) * r[1]/d_**3 - mu * r[1]/r_**3
    dvz_dt = - (1-mu)*r[2]/d_**3 - mu*r[2]/r_**3

    return np.array([v[0], v[1], v[2], dvx_dt, dvy_dt, dvz_dt])
```

Figura 9: Código función problema de los 3 cuerpos restringido con movimiento circular.

La Figura 10 define la función que calcula los puntos de Lagrange dados unos valores iniciales aproximados. Esta función hace uso de la función definida en 9.

```python
#Lagrange points
def lagr_points(U0, Nl, Nc):
    Lp = np.zeros([Nl, Nc])

    def f(y):
        x = np.zeros(6)
        x[0:3] = y
        x[3:6] = 0
        fx = rest_3b(x, 0)
        return fx[3:6]

    for i in range(Nl):
        Lp[i,:] = fsolve(f, U0[i, 0:3])

    return Lp
```

Figura 10: Código función cálculo puntos de Lagrange.

La Figura 11 define el código necesario para el cálculo de los autovalores de las regiones de estabilidad de los puntos de Lagrange. La función $sist_matrix()$ calcula el Jacobiado asociado a cada punto que se le pasa como argumento. Con el comando $eig(A)$ obtenemos los autovalores y los autovectores de la matriz Jacobiana.

```python
#Lagrange points stability
def sist_matrix(U0, t, f):
    eps = 1e-6
    N = len(U0)
    A = np.empty([N, N])

    for i in range(N):
        delta = np.zeros(N)
        delta[i] = eps

        A[:,i] = (f(U0+delta, t) - f(U0-delta, t)) / (2*eps)

    return A

def lagr_points_stab(U0):
    def f(y, t):
        return rest_3b(y, 0)

    A = sist_matrix(U0, 0, f)

    return np.linalg.eig(A)
```

Figura 11: Código funciones cálculo autovalores y autovectores de los puntos de Lagrange.

2.5. plot_lagr.py

La Figura 12 muestra la función que dibuja en 3 dimensiones, puesto que los cálculos se han realizado considerando las 3 coordenadas geométricas, los puntos de Lagrange en el sistema Tierra-Luna; también representadas.

```
5   #Plot Lagrange Points
6   def plot_lagr(Lps):
7       E = [0, 0, 0]
8       M = [1, 0, 0]
9
10      sns.set()
11      plt.figure(figsize=(10,9))
12      ax = plt.axes(projection='3d')
13      col = ['c', 'r', 'g', 'y', 'm']
14
15      ax.plot(E[0], E[1], E[2], color='darkblue', marker='o', markersize=25, label='Earth')
16      ax.plot(M[0], M[1], M[2], color='grey', marker='o', markersize=15, label='Moon')
17
18      for i in range(len(Lps)):
19          Lp = Lps[i]
20          ax.plot(Lp[0], Lp[1], Lp[2], color=col[i], marker='o', markersize=6, label='L' + str(i+1))
21
22      ax.set_title('Lagrange Points in 3D')
23      ax.set_xlabel('X [m]')
24      ax.set_ylabel('Y [m]')
25      ax.set_zlabel('Z [m]')
26      ax.set_zlim([-0.1, 0.1])
27      ax.legend(loc='upper left')
```

Figura 12: Código función representar en 3D los puntos de Lagrange.

La Figura 13 muestra la función que representa la órbita calculada en las proximidades del punto de Lagrange correspondiente.

```
30  #Plot perturbed orbit around lagrange point (INDIVIDUAL)
31  def plot_one_orb_lagr(Lp, U, n, m): #(LagrangePoint, Values, method)
32      sns.set()
33      plt.figure(figsize=(10,9))
34      ax = plt.axes(projection='3d')
35      ax.plot(Lp[0], Lp[1], Lp[2], marker='o', markersize=5, label='L' + str(n))
36      ax.plot3D(U[0,:], U[1,:], U[2,:], label='Orbit')
37
38      ax.set_title('Orbit around Lagrange Point L' + str(n) +' in 3D, ' + m.__name__ + ' method')
39      ax.set_xlabel('X [m]')
40      ax.set_ylabel('Y [m]')
41      ax.set_zlabel('Z [m]')
42      ax.set_zlim([-1, 1])
43      ax.legend()
```

Figura 13: Código función representar en 3D la órbita perturbada alrededor del punto de Lagrange.

3. Resultados y análisis

Se ha llevado a cabo una serie de simulaciones para determinar los puntos de Lagrange del sistema Tierra-Luna, así como la estabilidad de estos mismos y el análisis de una órbita en las proximidades de cada uno de los puntos determinados anteriormente.

3.1. Determinación de los puntos de Lagrange

Tabla 1: Puntos de Lagrange Tierra-Luna.

Coord.	X	Y	Z
L1	0.836915	0	0
L2	1.155682	0	0
L3	-1.005063	0	0
L4	0.487849	0.866025	0
L5	0.487849	-0.866025	0

La Tabla 1 muestra los puntos de Lagrange obtenidos con el programa en coordenadas en 3 dimensiones. Dado que los dos cuerpos a estudiar, en este caso la Tierra y la Luna, forman un plano que coincide con el plano XY del sistema de referencia propuesto, la componente en eje Z queda anulada y su valor es cero para todos los puntos obtenidos.

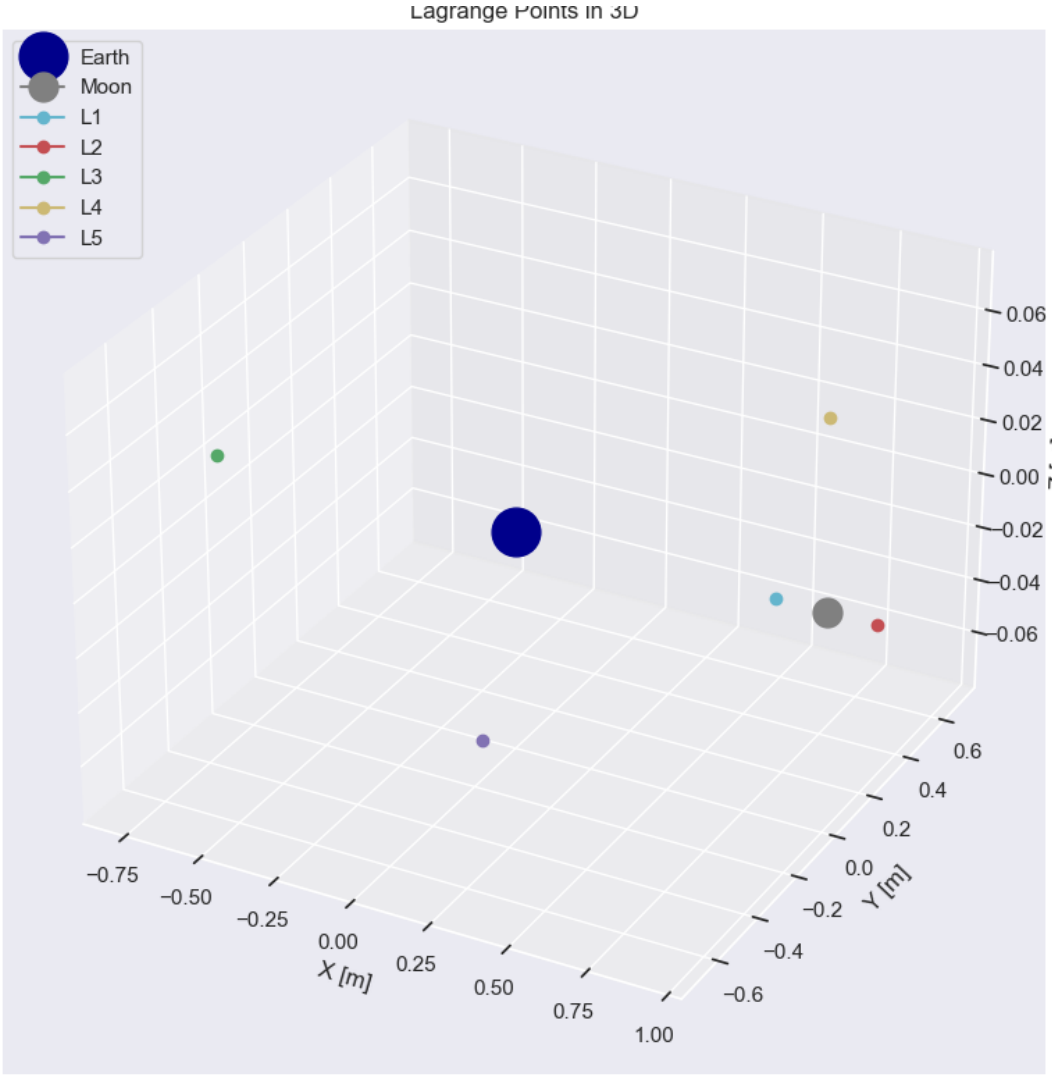


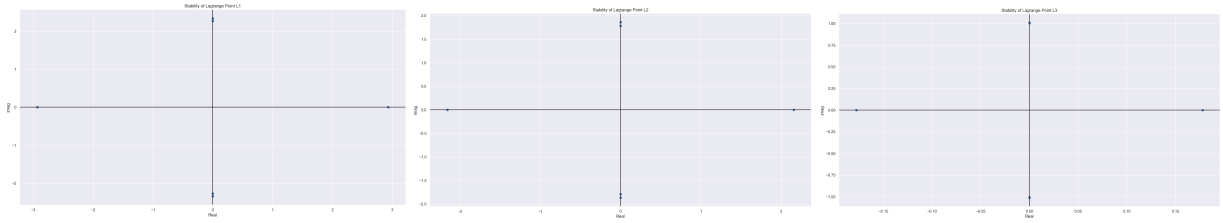
Figura 14: Representación en 3D de los puntos de Lagrange Tierra-Luna.

En la Figura 14 se representan los puntos de Lagrange del sistema Tierra-Luna, así como estos dos cuerpos destacando sobre el resto.

3.2. Estabilidad de los puntos de Lagrange

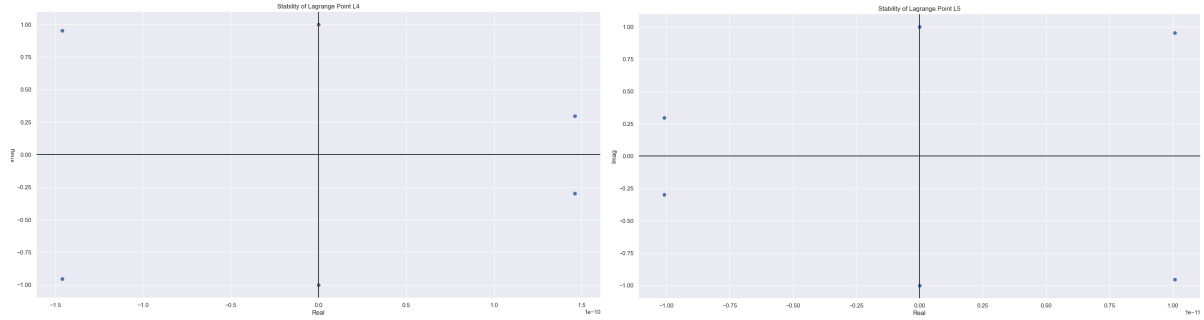
Se han representado en las Figuras 15 y 16 los autovalores de los distintos puntos de Lagrange en el plano complejo. Puede observarse como los puntos $L1, L2, L3$ tienen al menos un autovalor en la parte positiva Real, indicando que es un punto inestable.

Los puntos $L4, L5$ también tienen no uno, sino dos autovalores en la región positiva del eje Real, pero como se puede apreciar según la escala, están muy próximos al eje Imaginario. Además este valor se podría asociar a un error de cálculo del propio ordenador. Esto indica que son puntos estables, indicando que para un objeto que se encuentre en las proximidades de este punto su órbita se mantendrá estable en dicho punto y no divergirá.



(a) Autovalores L1 (b) Autovalores L2 (c) Autovalores L3

Figura 15: Autovalores puntos de Lagrange inestables.



(a) Autovalores L4 (b) Autovalores L5

Figura 16: Autovalores puntos de Lagrange estables.

3.3. Órbitas alrededor de los puntos de Lagrange

En estas simulaciones se ha fijado el tiempo de simulación en 100 segundos y un $\Delta t = 0.001$ segundos. Se han llevado ha cabo varias simulaciones utilizando varios métodos de resolución numérica.

3.3.1. Runge-Kutta de alto orden. *RK87*

Se ha realizado una primera simulación utilizando el esquema numérico RK87 que se ha definido anteriormente.

La Figura 17 es una representación en 3D del sistema orbital Tierra-Luna, con sus puntos de Lagrange (representados como puntos) y de las órbitas en las proximidades de estos. Las condiciones iniciales de posición para estos cuerpos se han determinado según una función definida en la Figura 4, las condiciones de velocidad iniciales son nulas.

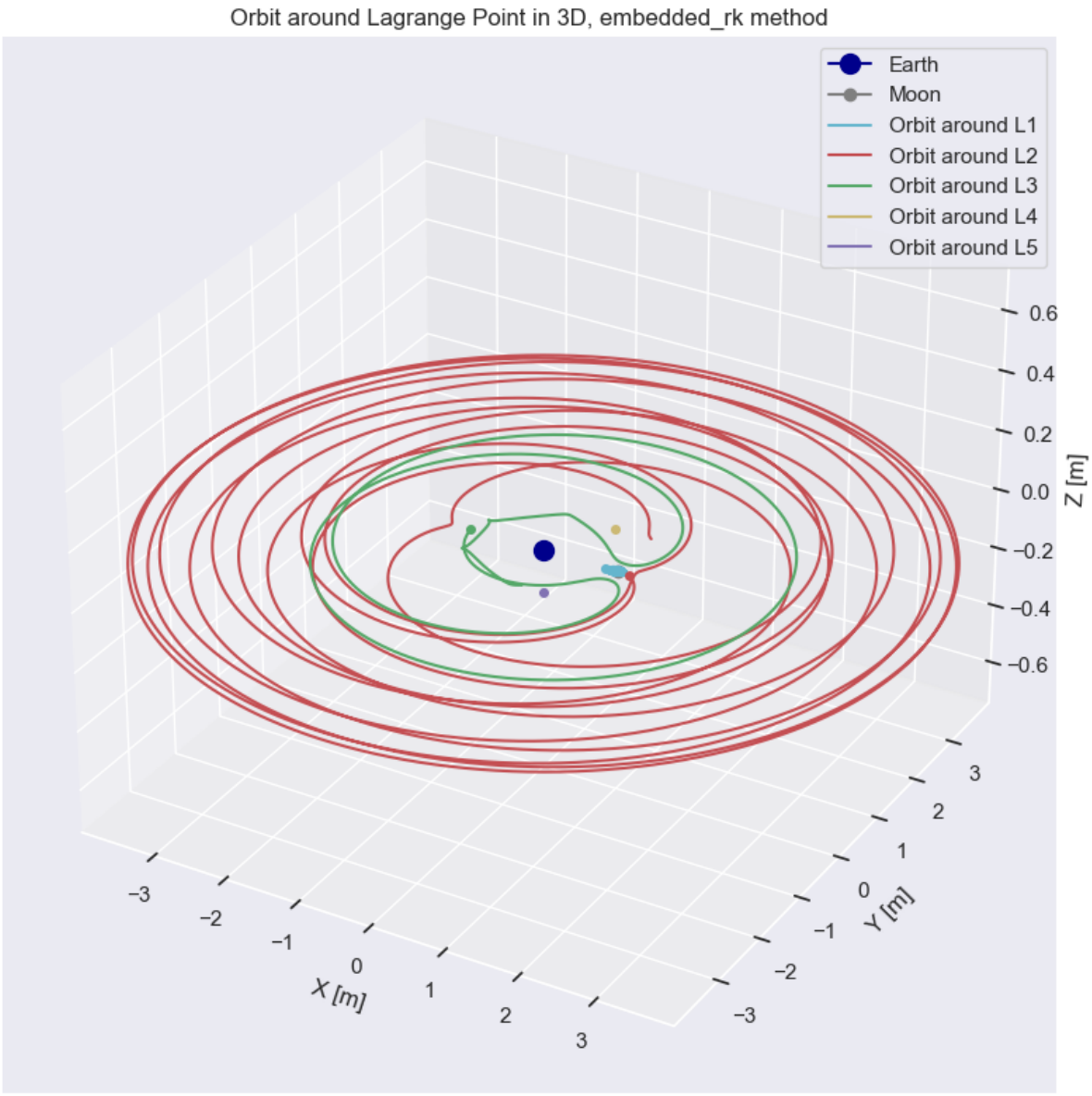


Figura 17: Representación en 3D de las órbitas sobre los puntos de Lagrange en sistema Tierra-Luna, RK87, $t = 100s$.

Las siguientes Figuras representan cada una de las órbitas de los cuerpos alrededor de los puntos de Lagrange de manera individual.

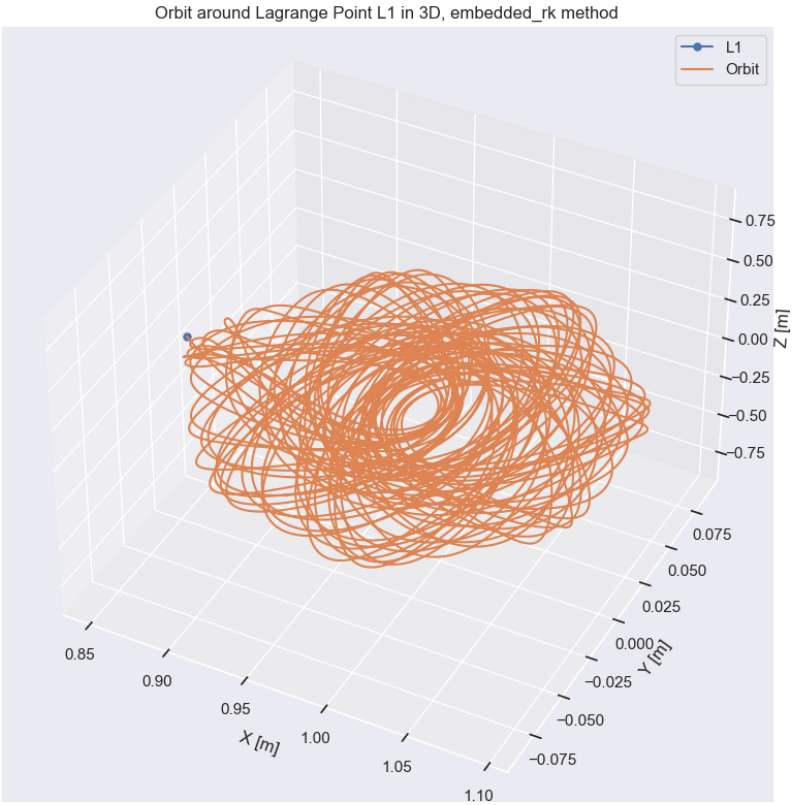


Figura 18: Órbita en 3D sobre L1, RK87, $t = 100s$.

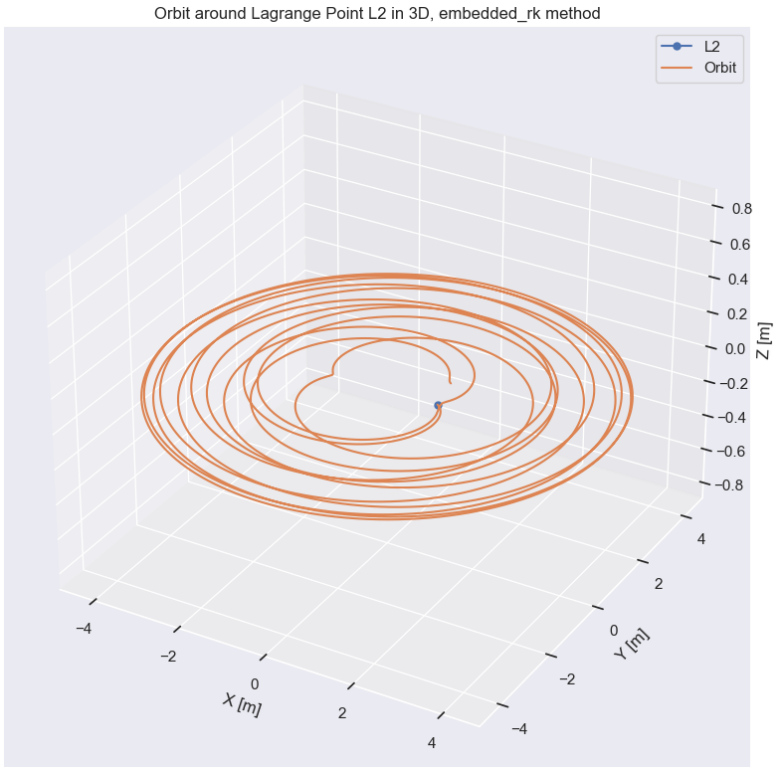


Figura 19: Órbita en 3D sobre L2, RK87, $t = 100s$.

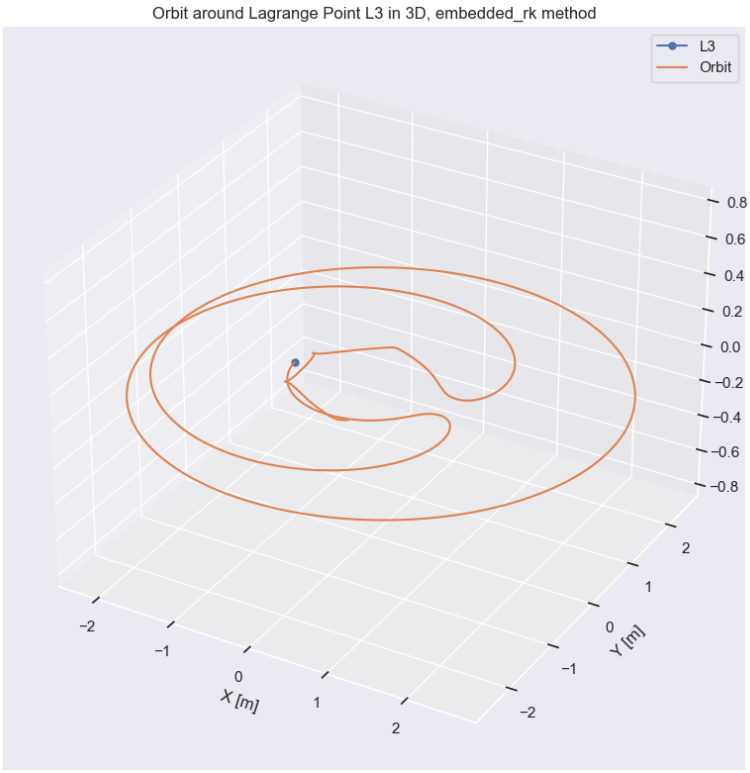


Figura 20: Órbita en 3D sobre L3, RK87, $t = 100s$.

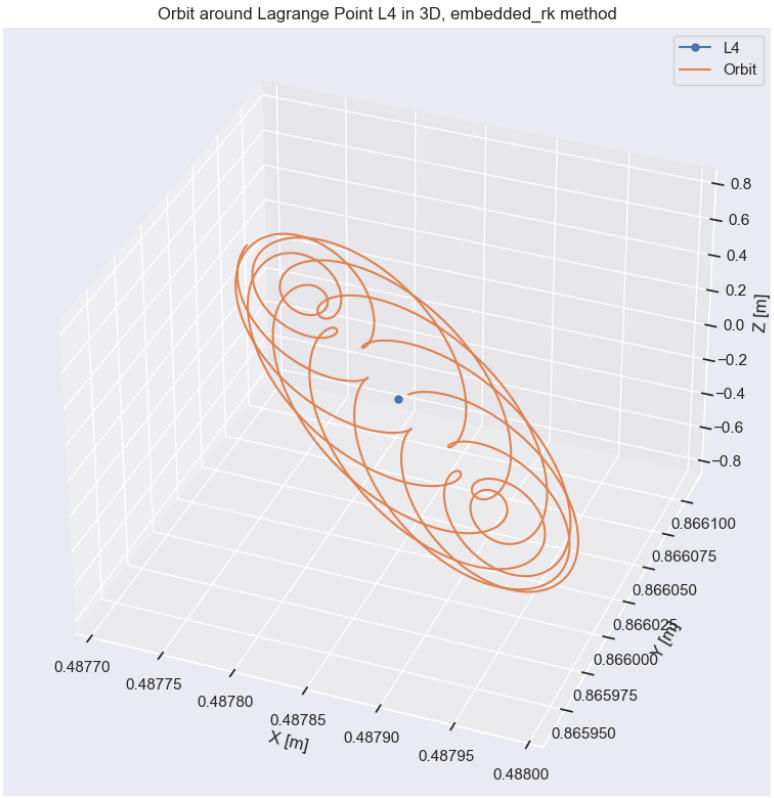


Figura 21: Órbita en 3D sobre L4, RK87, $t = 100s$.

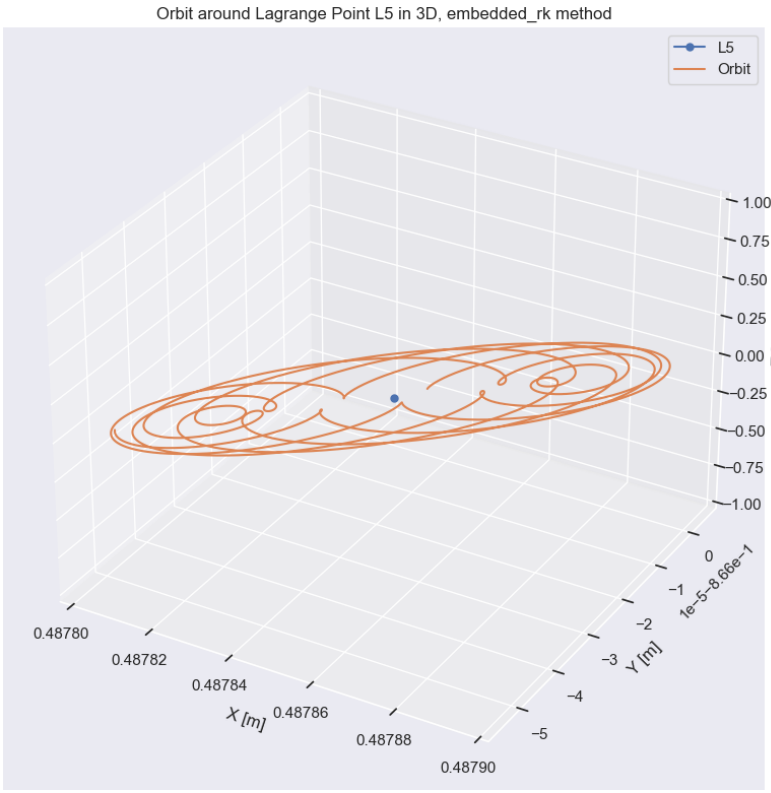


Figura 22: Órbita en 3D sobre L5, RK87, $t = 100s$.

3.3.2. Runge-Kutta de orden 4

A modo de comparación, respecto al nuevo esquema numérico implementado, se ha utilizado para la resolución de las órbitas el método de Runge-Kutta de orden 4 ya implementado en trabajos anteriores.

A grandes rasgos para las mismas condiciones de simulación (mismos Δt y tiempo total) se obtienen unos resultados un poco menos precisos con este método; se puede apreciar en la escala de los ejes de las gráficas obtenidas.

También es cierto que el tiempo de cómputo es mucho menor con este método. Serviría para la obtención de una primera aproximación de los resultados y luego obtener los resultados más precisos con un método como el RK87.

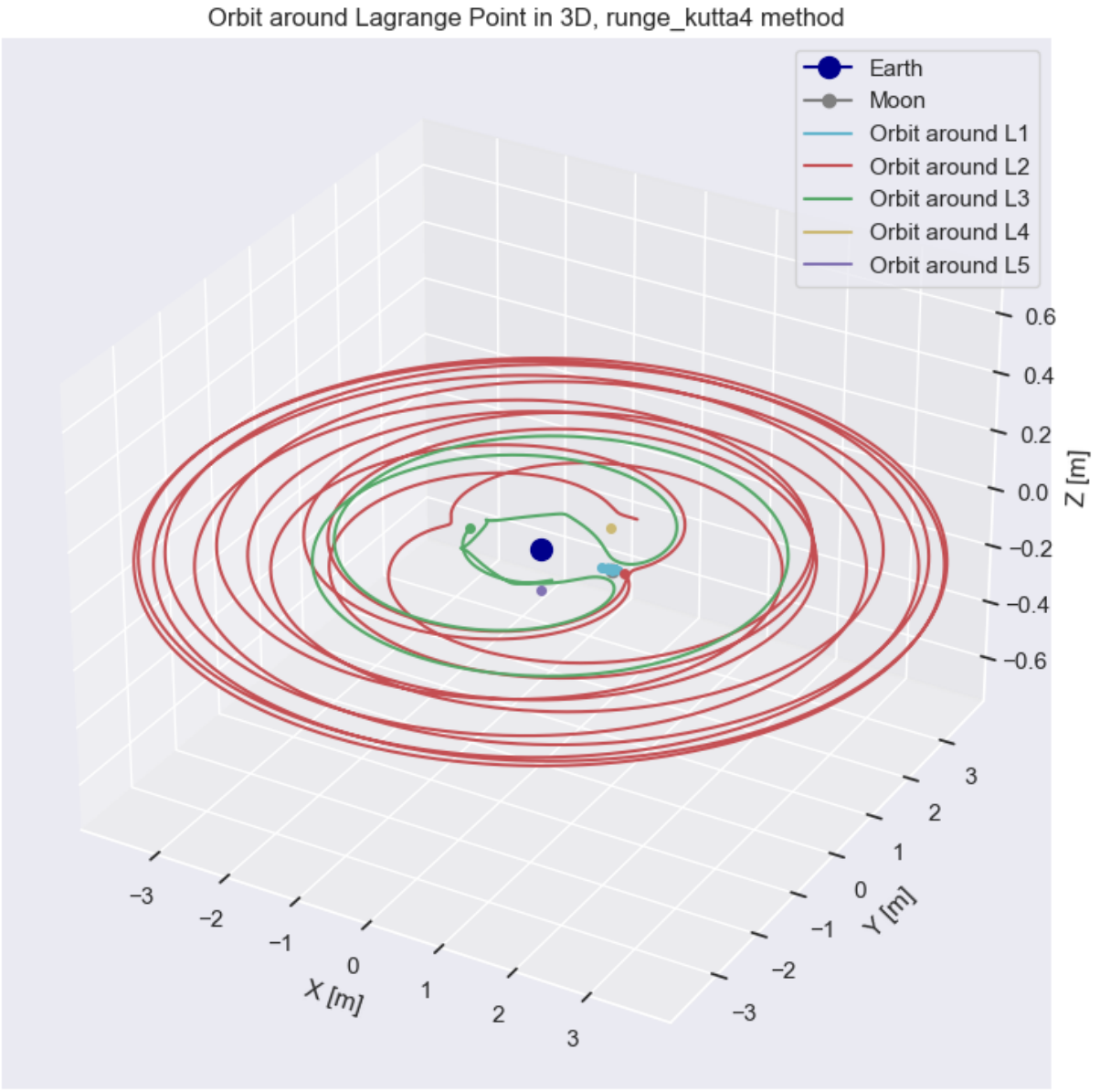


Figura 23: Representación en 3D de las órbitas sobre los puntos de Lagrange en sistema Tierra-Luna, RK4, $t = 100s$.

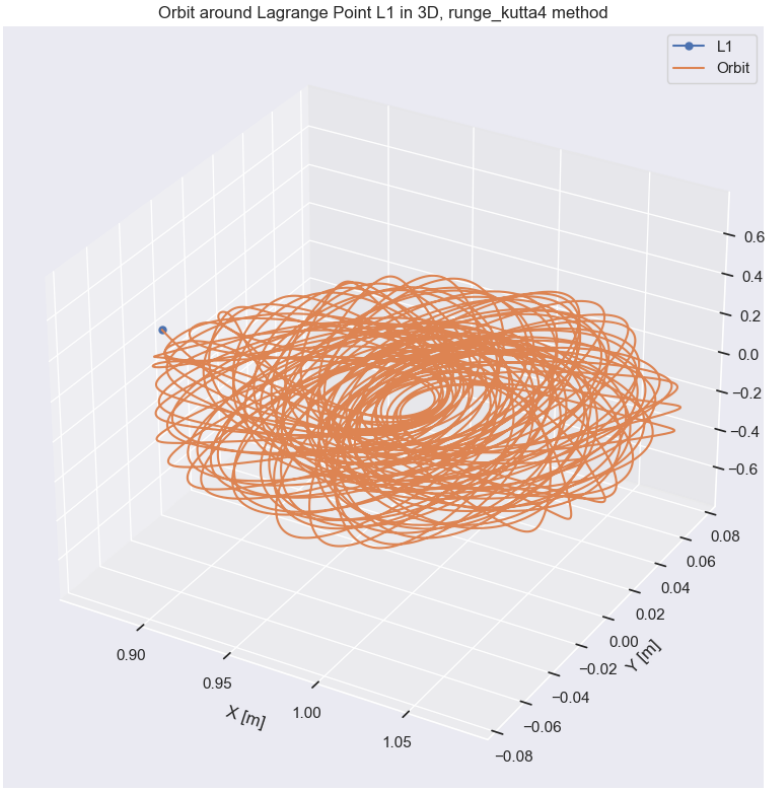


Figura 24: Órbita en 3D sobre L1, RK4, $t = 100s$.

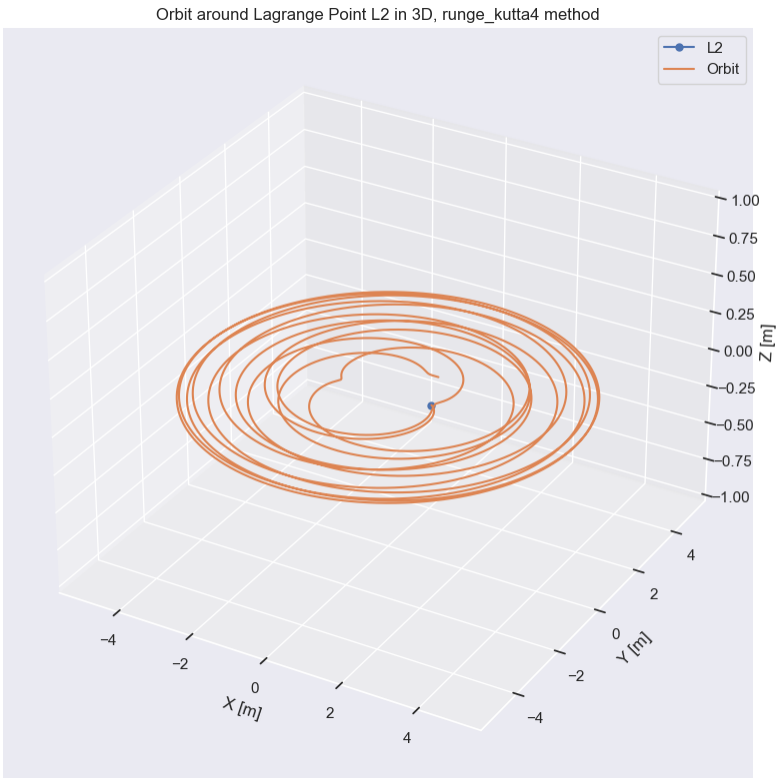


Figura 25: Órbita en 3D sobre L2, RK4, $t = 100s$.

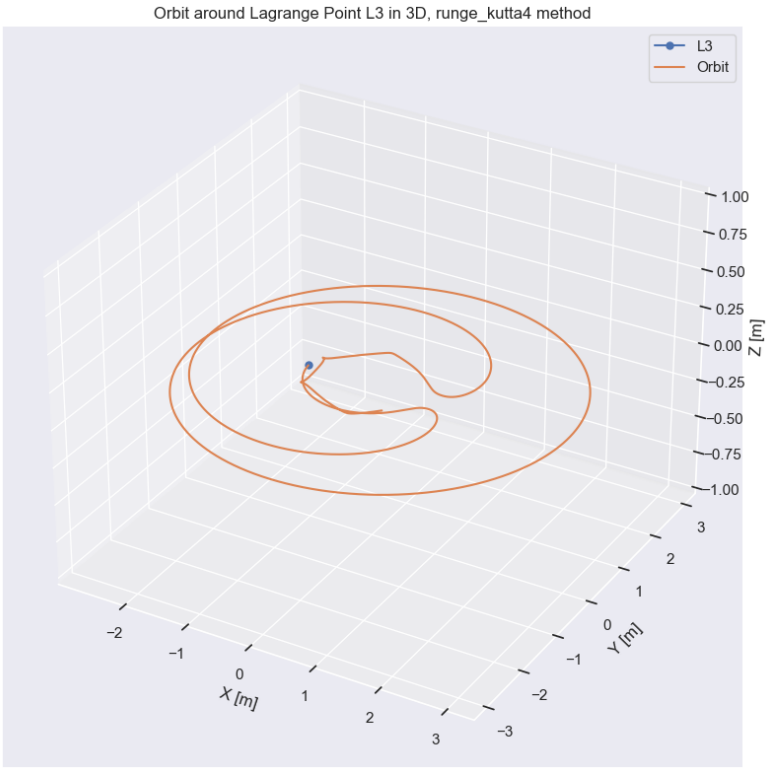


Figura 26: Órbita en 3D sobre L3, RK4, $t = 100s$.

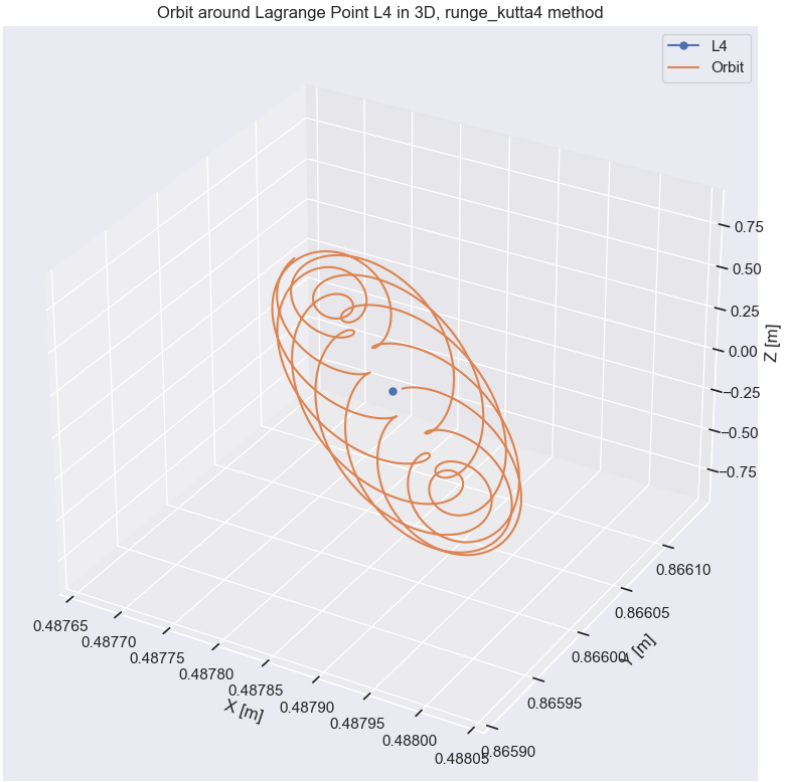


Figura 27: Órbita en 3D sobre L4, RK4, $t = 100s$.

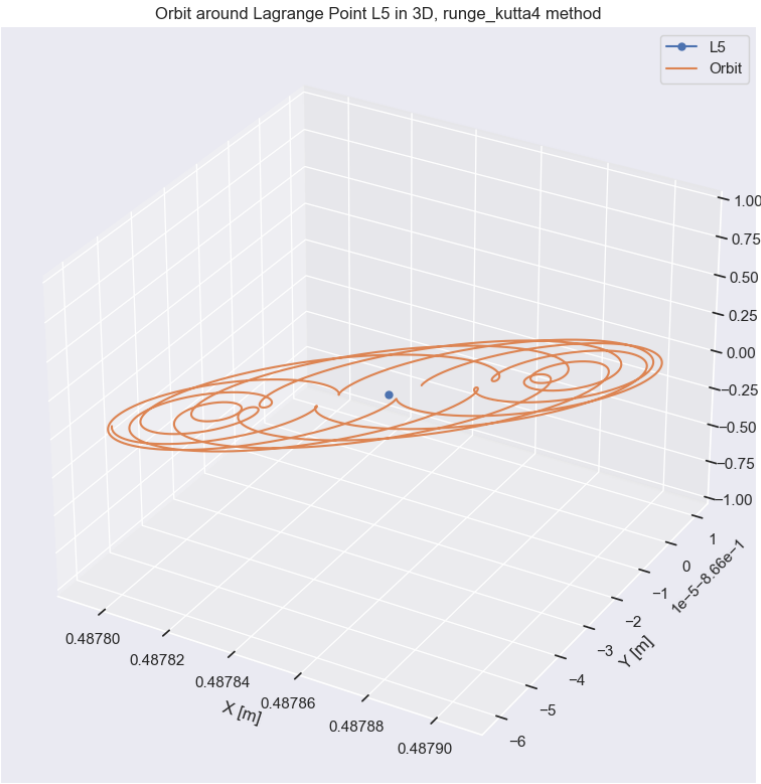


Figura 28: Órbita en 3D sobre L5, RK4, $t = 100s$.

Simulaciones con otros métodos de resolución numérica Se han simulado las órbitas con otros métodos ya implementados en anteriores trabajos, como lo son el Método de Euler, Crank-Nicolson y Leap-Frog.

Los resultados no se muestran en este informe puesto que algunos de ellos (Euler, Euler inverso y Leap-Frog), incluso con Δt pequeños, el resultado que se obtiene es una divergencia de los cuerpos a los pocos segundos de simulación. Con el método de Crank-Nicolson se pueden obtener buenos resultados pero aún así divergen cuando se simula durante tiempos mayores a 100 segundos.

4. Conclusiones

Cabe destacar el tiempo de procesado que se necesita para realizar las simulaciones utilizando un método como es el *RK87* de alto orden en comparación con los métodos que ya se habían implementado en los anteriores trabajos, para las mismas condiciones de tiempo de simulación y de paso temporal Δt. Por el contrario se obtienen unos resultados mucho más precisos con el método RK87 que con el resto de métodos implementados.

Una buena extensión de este trabajo sería la de poder realizar el cálculo de las órbitas de cada punto de Lagrange de forma paralela. De esta forma se aprovecharía mejor la potencia del PC y se reduciría el tiempo de computo.