


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Informe de prácticas

Métodos numéricos y su implementación en Python

MILESTONES

Ampliación de Matemáticas I

12 de junio de 2023

Autor:

García Sánchez, Ignacio

Índice

1. Introducción

Este informe recoge la implementación de los diferentes *milestones* realizados en la asignatura de Ampliación de Matemáticas I y se comentan los resultados obtenidos.

Durante las clases se han impartido conocimientos de programación en Python, la implantación de métodos numéricos para la resolución de ecuaciones diferenciales y los fundamentos de la programación funcional. Los *milestones* propuestos a lo largo del curso han servido como puesta en práctica de los conocimientos adquiridos en clase y comprobar de primera mano los retos y beneficios de la construcción de tus propios códigos para la resolución de problemas y obtención de resultados. Además, ha sentado una base sobre la que se puede seguir trabajando de cara a tener una librería de funciones útiles en la simulación de propagaciones orbitales y demás herramientas para el estudio de misiones espaciales.

En primer lugar se muestra el enunciado de los *milestones* y su fundamento teórico, posteriormente se comenta su implementación en Python y por último, se presentan los resultados obtenidos. Además, en los anexos se puede ver la implantación en Python de las funciones utilizadas.

2. *Milestones*

A continuación se van a presentar los diferentes *milestones* realizados y los fundamentos teóricos utilizados para su resolución. Los *milestones* I y II se han realizado de forma conjunta debido a que la implementación del *milestone* I es la antesala básica del *milestone* II, además de que su realización ya de por sí se hizo conjunta puesto que en el momento en el que comencé en el Máster ya se habían propuesto y realizado en clase.

2.1. *Milestones* I y II

Los *milestones* I y II se corresponden con la implementación de métodos numéricos en Python para la propagación orbital. El movimiento orbital se puede modelizar mediante el movimiento kepleriano a partir de los parámetros orbitales que define Kepler junto con las leyes de Kepler.

$$\begin{aligned} E - e \cdot sin(E) =& \sqrt{\frac{\mu}{a^3}} \cdot (t - \tau) \quad ; \quad r = a \cdot (1 - e \cdot cos(E)) \quad ; \\ \frac{V^2}{2} - \frac{\mu}{r} =& \frac{\mu}{2p} \cdot (e^2 - 1) \quad con \quad p = a(1 - e^2) \end{aligned} \tag{2.1}$$

donde E es la anomalía excentrica, e es la excentricidad, μ es el parámetro gravitacional de la Tierra, a es el semieje mayor de la órbita, t es el tiempo y τ es el t_0, r es el módulo del radio vector, V es el módulo del vector velocidad, p es el parámetro de la órbita.

Por otro lado, es posible modelar el movimiento orbital mediante la siguiente expresión proveniente del movimiento kepleriano:

$$\ddot{\vec{r}} = -\frac{\vec{r}}{|\vec{r}|^3} \tag{2.2}$$

donde $\ddot{\vec{r}}$ es la aceleración sufrida por la acción de la gravedad y $\vec{r}$ es el vector posición.

Para la resolución de los métodos numéricos se ha utilizado la siguiente expresión matricial a la hora de resolver la anterior ecuación diferencial:

$$\begin{Bmatrix} \dot{\vec{r}_x} \\ \dot{\vec{r}_y} \\ \dot{\vec{r}_z} \\ \ddot{\vec{r}_x} \\ \ddot{\vec{r}_y} \\ \ddot{\vec{r}_z} \end{Bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -\frac{1}{|\vec{r}|^3} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{|\vec{r}|^3} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{|\vec{r}|^3} & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \vec{r_x} \\ \vec{r_y} \\ \vec{r_z} \\ \dot{\vec{r}_x} \\ \dot{\vec{r}_y} \\ \dot{\vec{r}_z} \end{Bmatrix} \tag{2.3}$$

A partir de esta expresión y una condiciones iniciales de $\vec{r} = (r_x, r_y, r_z)$ y $\vec{v} = \dot{\vec{r}} = (v_x, v_y, v_z)$ se puede resolver la ecuación diferencial mediante los siguientes metodos numéricos.

Los métodos numéricos utilizados son:

- Método de Euler explícito
- Método de Euler implícito
- Método de Crank-Nicolson
- Método de Runge-Kutta de cuarto orden

El problema a resolver mediante estos métodos numéricos se denomina problema de Cauchy o problema de condiciones iniciales.

2.2. *Milestone* III

En el *milestone* III se plantea la evaluación de los métodos numéricos. Se realiza un evaluación del error numérico derivado de la utilización de los métodos numéricos, que no son más que métodos de aproximación de la solución. Para ello se utilizará las desviaciones de Richardson y se poobtendrá la velocidad de convergencia de los métodos numéricos.

Las desviaciones de Richardson evalúa el error implicito de un método determinado mediante la comparación de dos evaluaciones con distinto paso temporal (distinta malla) dentro del mismo intervalo de tiempo:

$$E = \frac{U^{2N} - U^N}{1 - \frac{1}{2^q}} \tag{2.4}$$

donde E es el error existente, U es la solución obtenida para cada evaluación y q es el orden del método utilizado.

La velocidad de convergencia representa el nivel de convergencia del método, es decir, la capacidad de obtener una solución suficientemente precisa con el menor número de iteraciones, a mayor velocidad de convergencia menor es dicho número de iteraciones.

2.3. *Milestone* IV

El *milestone* IV corresponde con la evaluación de las regiones de estabilidad. Para ello se propone la resolución de la expresión que define un oscilador armónico lineal:

$$\ddot{x} + x = 0 \tag{2.5}$$

Además de los métodos numéricos ya implantados, se procede a utilizar otro más denominado *Leap Frog*.

2.4. *Milestone* V

En el *milestone* V se ha afrontado el problema de N cuerpos. Este problema evalúa el comportamiento de N cuerpos dispuestos en el espacio (bidimensional o tridimensional) únicamente bajo los efectos de sus fuerzas gravitatorias. Este problema es de interés debido al coste computacional que supone el aumento de las variables que intervienen en el cálculo de la propagación de cada cuerpo.

La propagación se determina de la siguiente forma:

$$\ddot{\vec{r_i}} = -\sum_{j=1;j\neq i} \frac{\vec{r_j} - \vec{r_i}}{|\vec{r_j} - \vec{r_i}|^3} \tag{2.6}$$

donde la acelearación del cuerpo i se obtiene a partir de la influencia gravitatoria del resto de cuerpos j.

2.5. *Milestone* VI

El *milestone* VI consiste en la evaluación de los puntos de Lagrange. Para ello se propone un escenario de tres cuerpos en el espacio en el que un cuerpo es significativamente más pequeño, por lo que su influencia sobre el resto es despreciable. Así, los puntos de Lagrange son aquellos puntos del espacio donde la influencia de los dos cuerpos sobre el tercero (de masa despreciable) se compensan, provocando que el cuerpo no sufra aceleración ninguna.

Además, se realiza la implementación de un método numérico embebido a partir del método Runge-Kutta.

3. Implementación

La implementación de los distintos *milestones* propuestos se realiza en Python mediante la modularización y con un enfoque funcional.

Así, tendremos un *script* principal que planteará condiciones iniciales y parámetros del problema, y que llamará a las funciones necesarias para completar su cometido. Estas funciones se recogeran en *scripts* independientes al principal, a los que se los denomina módulos, los cuales recogen funciones del mismo tipo de manera que se obtenga una clasificación y un orden. A su vez, estas funciones se apoyarán en otras funciones más "básicas"para cumplir su objetivo, siguiendo el enfoque funcional.

3.1. *Milestones* I y II

En estos *milestones* se impementas en código Python los diferentes métodos numéricos. Se ha hecho de tal forma que queden recogidos en un mismo módulo denominado *ODEs*. La funciones de cada método numérico necesita la funcion a propagar, el valor de la variable requerido en el tiempo t, el paso de tiempo dt y el valor del tiempo t. Las funciones devuelven el valor de la variable en el tiempo $t + dt$.

También se ha implementado en este módulo el problema de Cauchy que se utiliza para llamar a estas funciones de los métodos numéricos. El problema de Cauchy necesita la función del método numérico, la función a propagar, las condiciones iniciales y el vector tiempo con los instantes temporales a propagar.

En otro módulo denominado *Mecanica Orbital* se ha implantado la ecuación diferencial que define el movimiento kepleriano.

En el *script* principal se imponen las condiciones iniciales y los parámetros necesarios, y se llaman a las funciones de los métodos numéricos mediante un bucle y, posteriormente, mediante la función del problema de Cauchy.

3.2. *Milestone* III

En el módulo *ODEs* se implementa la función de estimación del error de Richardson y la función del cálculo de la velocidad de convergencia.

La función de Richardson necesita la función del método numérico a evaluar, la función a propagar, las condiciones iniciales y, al menos, un vector tiempo con los instantes a propagar. Es posible introducir el segundo vector tiempo con el que comparar resultados para obtener el error, el cual debe tener el mismo dominio que el primero pero con otro paso de tiempo, o puede esr ignorado y calculado en la función imponiendo un paso de tiempo de la mitad del primero. La función utiliza la función del problema de Cauchy para evaluar el método numérico.

La función de velocidad de convergencia realiza un bucle de evaluaciones del método numérico hasta que se obtiene un error menor a una tolerancia dada o se completa el número máximo de iteraciones impuesto.

En el *script* principal se especifican las condiciones iniciales y los parámetros de la simulación y se llaman a las funciones de evaluación del error de Richardson y a las de determinación de la velocidad de convergencia.

3.3. *Milestone* IV

Para el estudio de la estabilidad, se ha implantado una función en el módulo de *ODEs* la únicamente necesita la función del método numérico a evaluar. Además, se ha incluido en este módulo el nuevo método numérico *Leap Frog*.

En el *script* principal se imponen las condiciones iniciales y los parámetros necesarios para la simulación, se define la función del oscilador armónico y se llama a la función de evaluación de la estabilidad.

3.4. *Milestone* V

En el módulo de *Mecanica Orbital* se han implantado las funciones necesarias para el problema de N cuerpos.

Hay una función para la propagación de los N cuerpos a partir de unas condiciones iniciales. El número de cuerpos es un input de la función, cuyo valor mínimo es 2. Las condiciones iniciales también puede ser un input de la función, si no lo es, se determinan unas condiciones aleatorias mediante la llamada a otra función implementada en el mismo módulo. La propagación se realiza mediante un bucle que determina la influencia de cada cuerpo sobre cada uno de ellos.

En el *script* principal se hacen dos simulaciones con número de cuerpos distinto. Se imponen las condiciones iniciales para cada simulación y se hacen las respectivas llamadas a las funciones.

3.5. *Milestone* VI

De nuevo, en el módulo de *Mecanica Orbital* se han implantado las funciones necesarias para el problema restringido de 3 cuerpos y de los puntos de Lagrange. Se ha implantado una función para progapar los 3 cuerpos, una para obtener los puntos de Lagrange del sistema y una para estudiar la estabilidad de dichos puntos.

En el módulo de *ODEs* se ha implantado las funciones relacionadas con el Método numérico embebido. Se ha desarrollado un Runge-Kutta Embebido a partir de la matriz de su Butcher, también implantada como función.

Además se han implantado las funciones auxiliares necesarias, como pueden ser el cálculo del paso variable o la obtención de la Jacobiana para la determinación de la estabilidad de los puntos de Lagrange.

4. Resultados

A continuación se exponen los resultados obtenidos en cada uno de los *milestones*.

4.1. *Milestone* I y II

En las siguientes figuras se van a mostrar los resultados obtenidos mediante los distintos métodos numéricos, propagando 100000 s con un paso de 1 s:

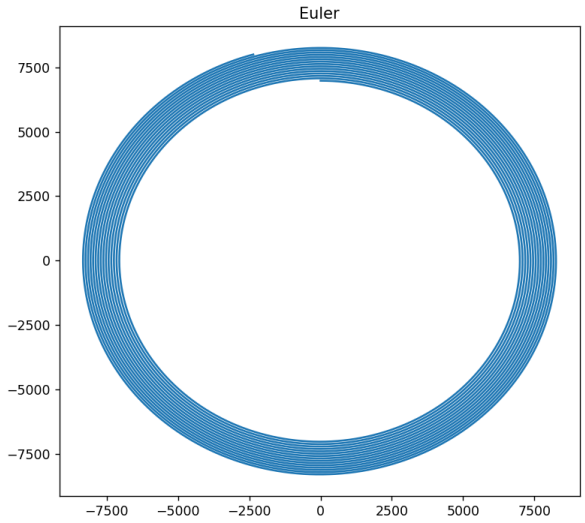

Figura 4.1: Propagación mediante el método de Euler Explícito.

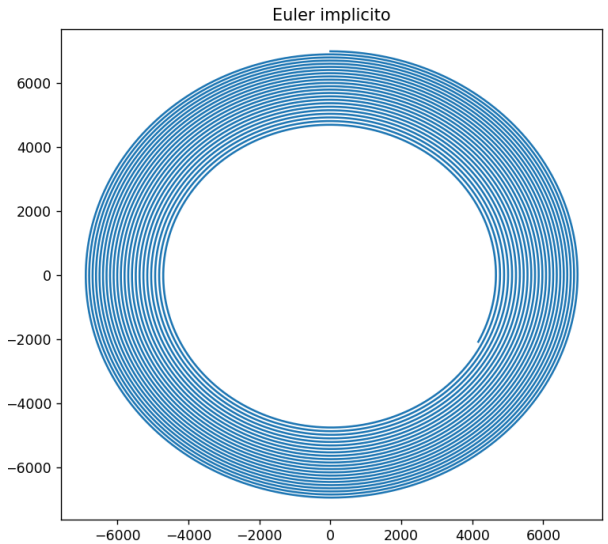

Figura 4.2: Propagación mediante el método de Euler Implícito.

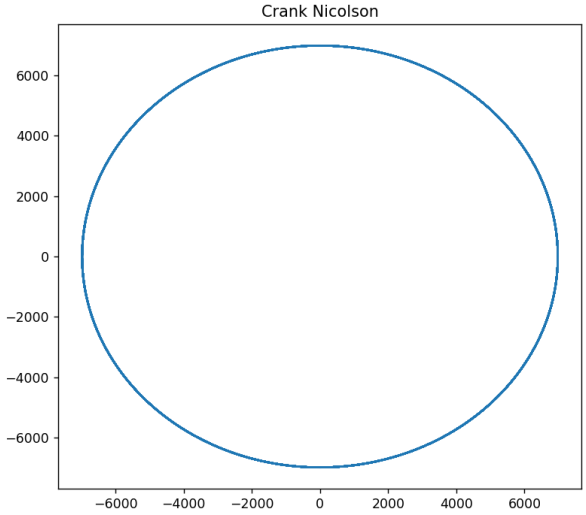

Figura 4.3: Propagación mediante el método de Crank-Nicolson.

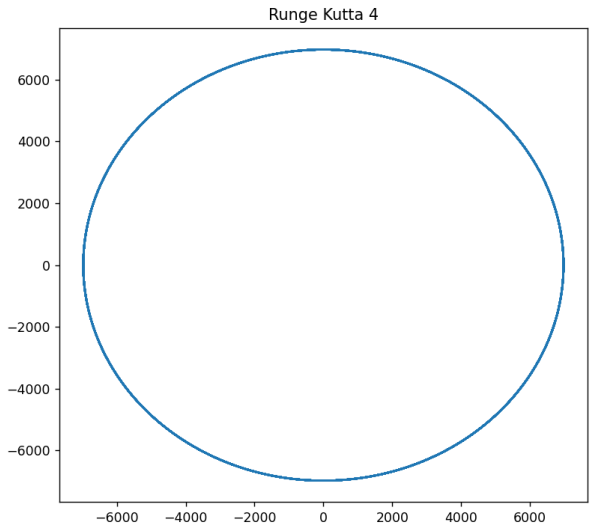

Figura 4.4: Propagación mediante el método de Runge-Kutta.

Se puede observar las diferencias de resultados existentes entre ellos. Por un lado tenemos la propagación de Euler explícito, cuya solución tiende al infinito, mientras que la de Euler implicito tiende a cero y lo hace más rápidamente, lo que significa que se comete mayor error. En cambio, los métodos de Crank-Nicolson y Runge-Kutta cometen un error muy pequeño durante la propagación, que no se puede apreciar en estas figuras, ofreciendo una solución muy cercana a la exacta.

Si disminuimos el paso de tiempo a 0.1, obtenemos otros resultados:

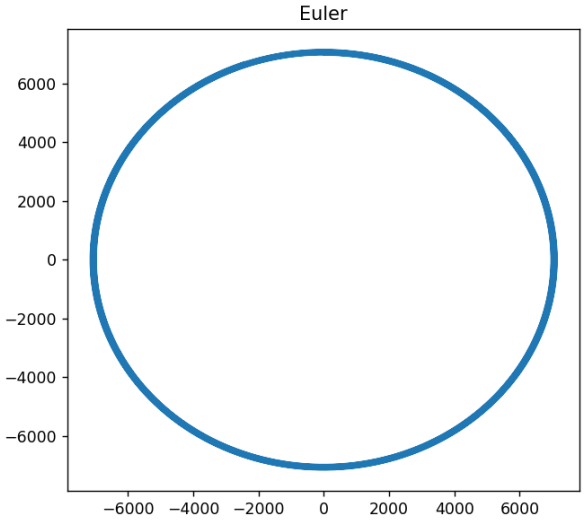

Figura 4.5: Propagación mediante el método de Euler Explícito.

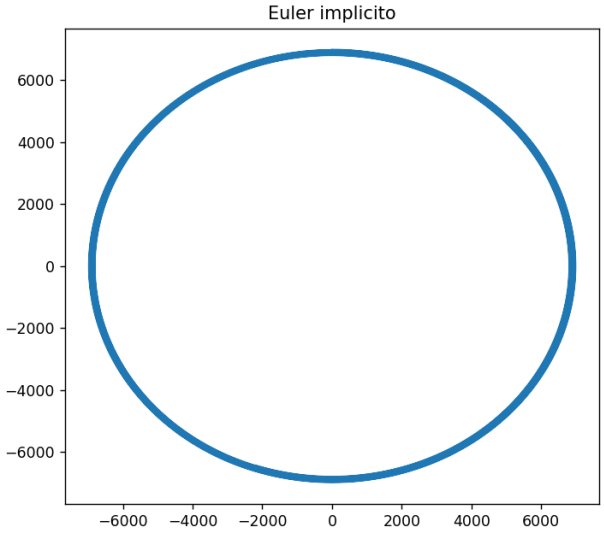

Figura 4.6: Propagación mediante el método de Euler Implícito.

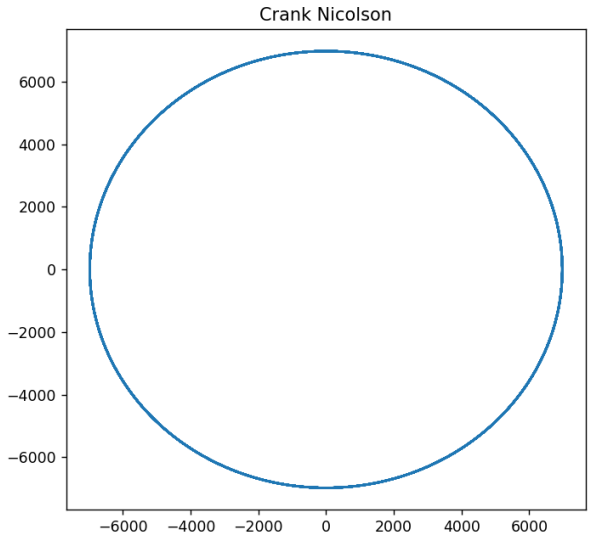

Figura 4.7: Propagación mediante el método de Crank-Nicolson.

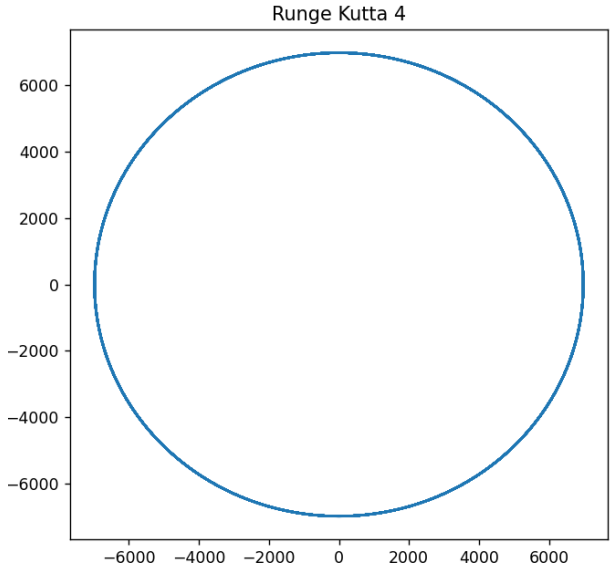

Figura 4.8: Propagación mediante el método de Runge-Kutta.

Como se puede ver, el error cometido en los métodos euler explícito e implícito ha disminuido notablemente gracias a la disminución del error acumulado al disminuir un orden de magnitud el paso de tiempo. Por otro lado, el método de Crank-Nicolson y el Runge-Kutta se mantienen ofreciendo una solución muy cercana a la solución exacta.

4.2. *Milestone* III

A continuación se presentan los resultados obtenidos con cada método numérico estudiado, con dos pasos de tiempo distintos, 1 s y 0.1 s y una propagación de 100000 s.

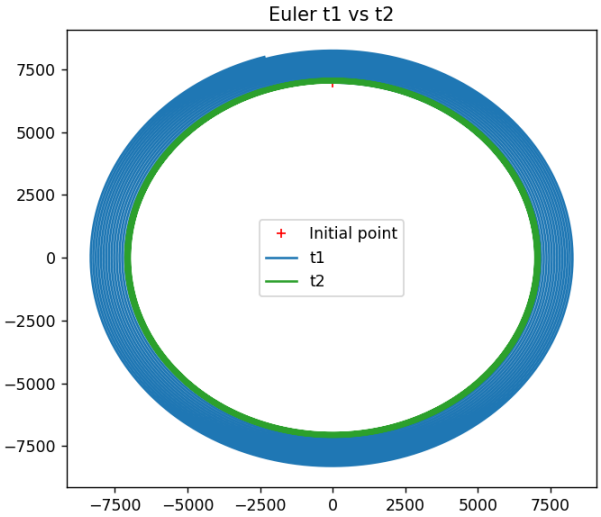

Figura 4.9: Propagación mediante el método de Euler Explícito.

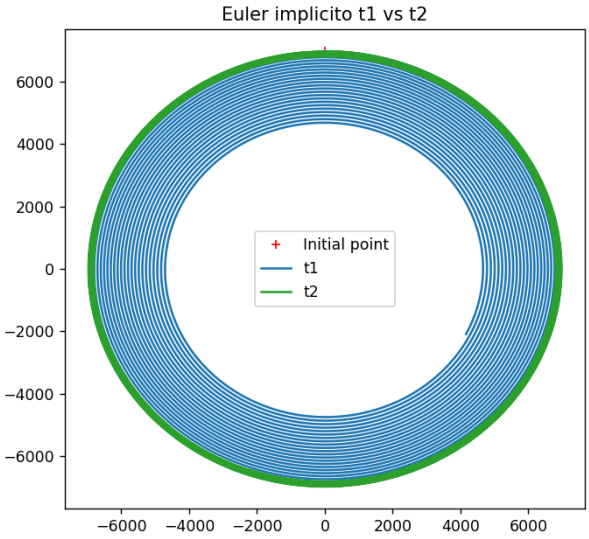

Figura 4.10: Propagación mediante el método de Euler Implícito.

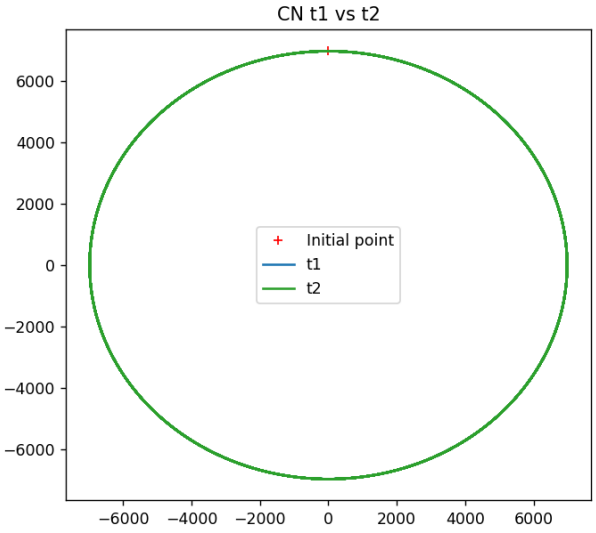

Figura 4.11: Propagación mediante el método de Crank-Nicolson.

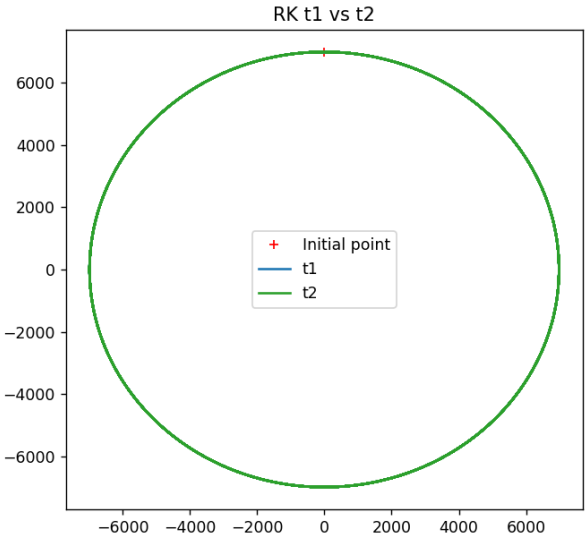

Figura 4.12: Propagación mediante el método de Runge-Kutta.

Como puede observarse, se puede comparar fácilmente las diferencias obtenidas con uno y otro paso de tiempo en los métodos de Euler, mientras que para Crank-Nicolson y Runge-Kutta las soluciones son prácticamente idénticas.

A continuación se muestran los errores cometidos en cada uno de los métodos numéricos utilizados mediante distintas gráficas.

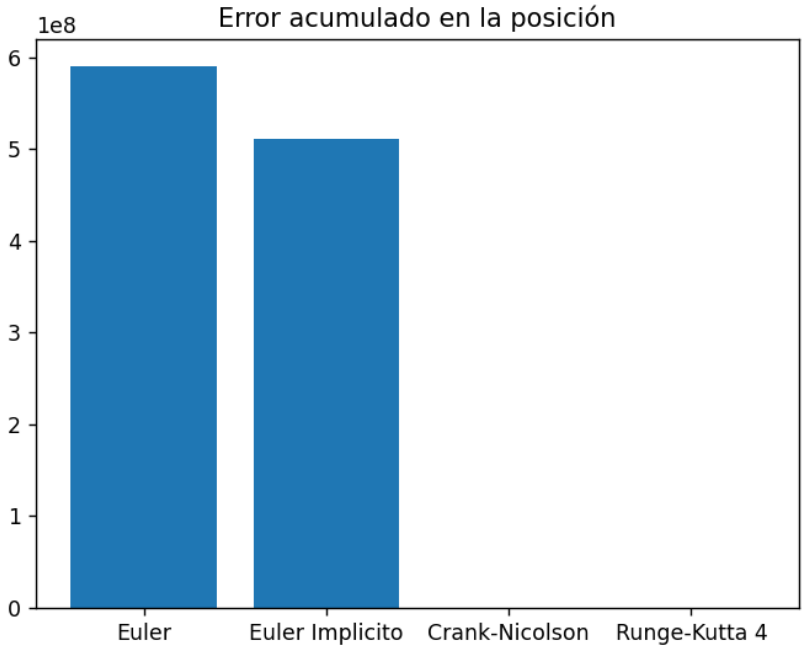

Figura 4.13: Valor del error acumulado después del tiempo de la propagación.

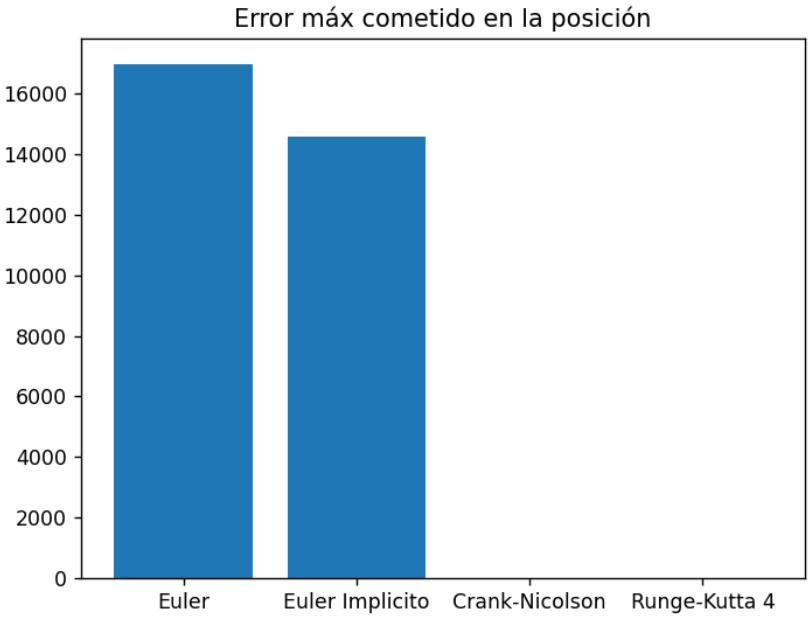

Figura 4.14: Valor máximo del error obtenido durante la propagación.

En estas dos gráficas, se han representado los valores del error acumulado por cada método, lo que nos da una idea de cómo la solución se separa de la solución exacta a medida que trascurre el tiempo; y del error máximo cometido, lo que nos revela qué método ofrece el mayor error. se observa como el error cometido por Euler explícito es el mayor cometido de todos los métodos, muy cercano al obtenido por el implícito. Por otro lado, los errores cometidos por Crank-Nicolson y Runge-Kutta son despreciables en estas figuras.

A continuación se presentan las velocidades de convergencia de los métodos numéricos utilizados. Para representar este valor, se ofrece en el eje x de las gráficas el valor del $los(N)$, siendo N el número de intervalos, y en el eje y el valor de $log(|U^{2N} - U^N|)$.

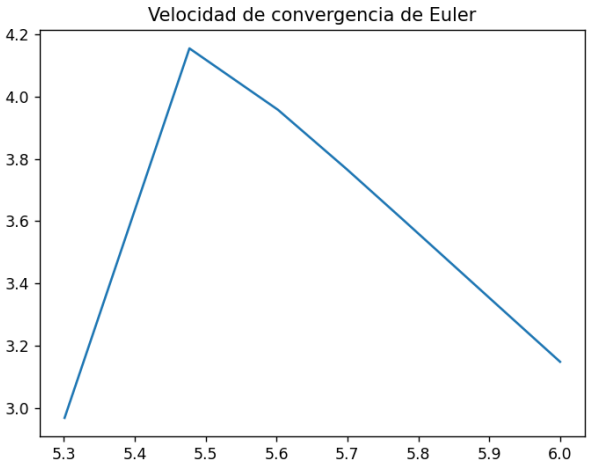

Figura 4.15: Velocidad de convergencia del método de Euler explícito.

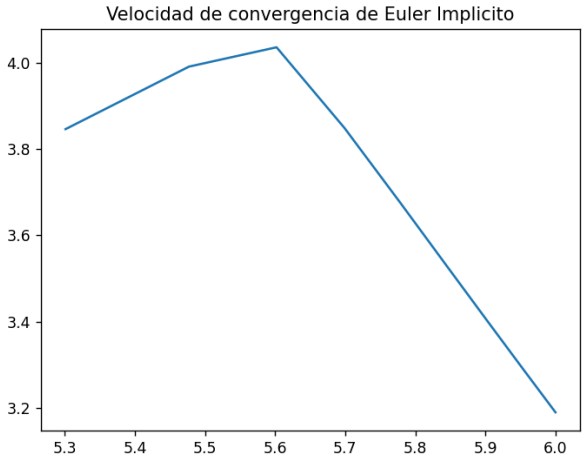

Figura 4.16: Velocidad de convergencia del método de Euler implícito.

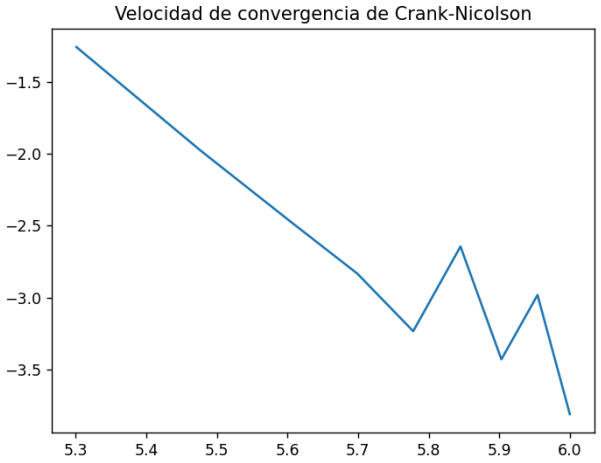

Figura 4.17: Velocidad de convergencia del método de Crank-Nicolson.

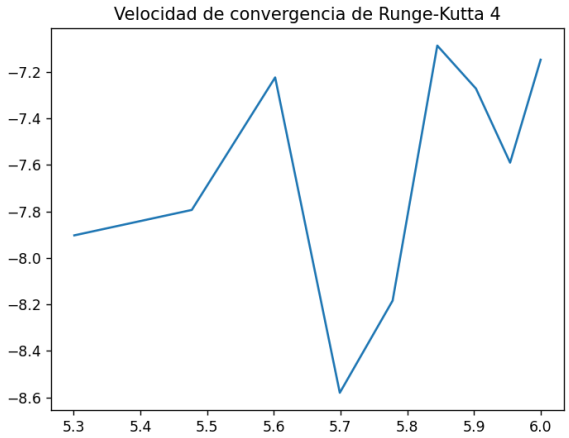

Figura 4.18: Velocidad de convergencia del método de Runge-Kutta.

Mientras que, para los métodos de Euler, estas gráficas revelan que es necesario un gran número de intervalos (es decir, un paso de tiempo muy pequeño) para obtener poco error en la solución, para los métodos de Crank-Nicolson y Runge-Kutta este número no es necesario que sea muy grande puesto que para lo presentado ya se obtienen valores de error despreciables, lo que significa que pueden operar con un error aceptable a pasos de tiempo considerables.

4.3. *Milestone* IV

En las siguientes figuras se van a mostrar los resultados obtenidos mediante los distintos métodos numéricos, propagando 100 s con un paso de 0.1 s:

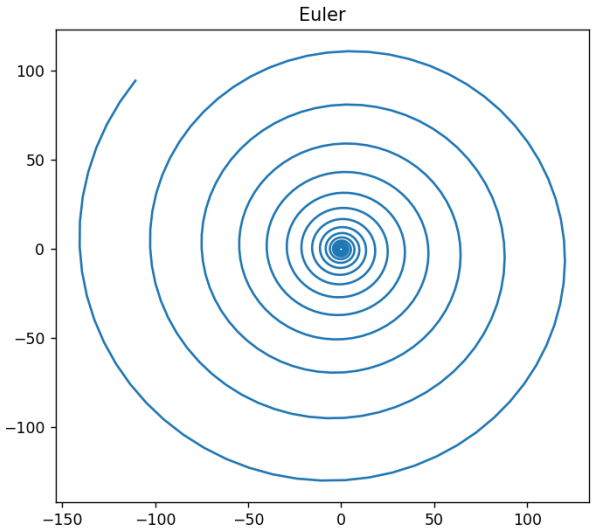

Figura 4.19: Propagación mediante el método de Euler Explícito.

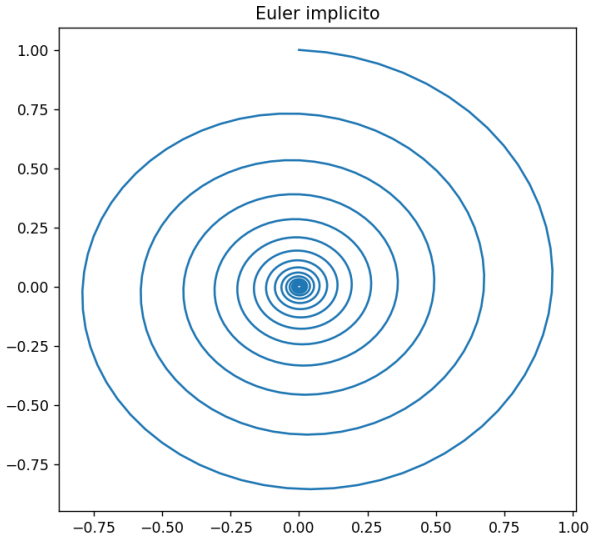

Figura 4.20: Propagación mediante el método de Euler Implícito.

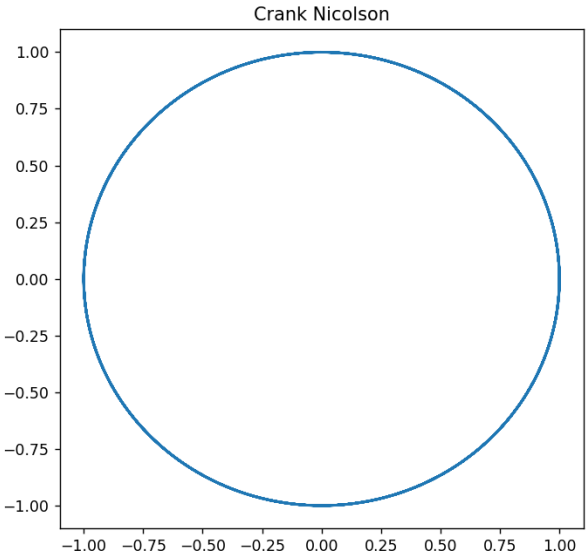

Figura 4.21: Propagación mediante el método de Crank-Nicolson.

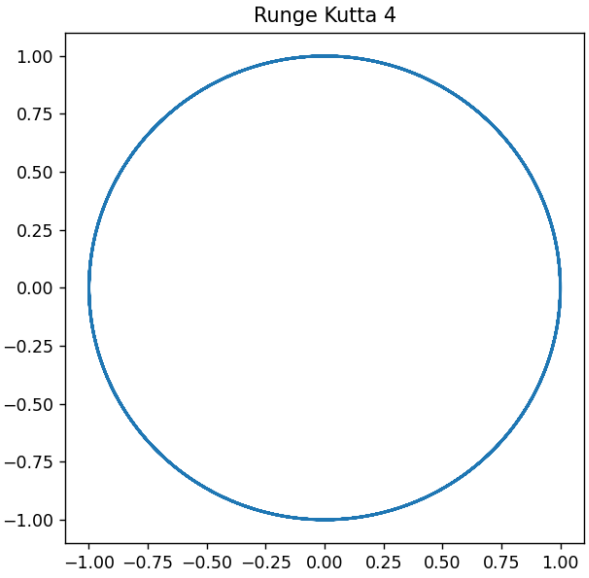

Figura 4.22: Propagación mediante el método de Runge-Kutta.

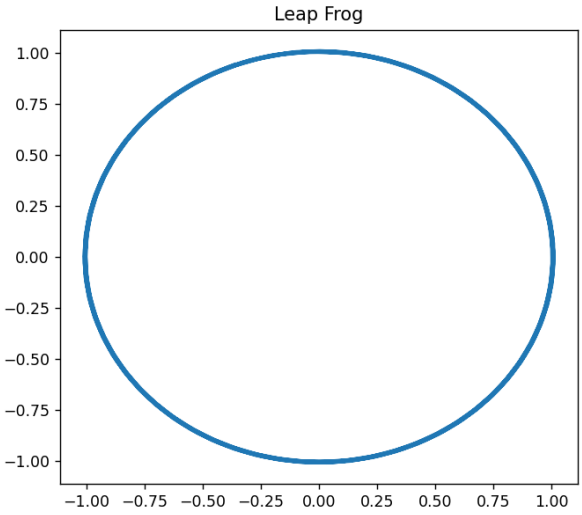

Figura 4.23: Propagación mediante el método de Leap Frog.

Como puede observarse, el comportamiento de los métodos de Euler explícito e implícito es idéntico al observado al evaluar la función de propagación orbital. Con el método de Euler explícito la solución tiende a infinito, mientras que con el implícito la solución tiende a 0. Los métodos de Crank-Nicolson y Runge-Kutta vuelven a ofrecer una solución prácticamente igual a la exacta y el nuevo método introducido de Leap Frog ofrece una muy buena solución, con un error bastante bajo respecto a la solución exacta.

A continuación se muestran las regiones de estabilidad de los diferentes métodos numéricos utilizados. En el problema armónico planteado, los autovalores se corresponden con i y $-i$, por lo que si los puntos (Re,Im) de estas raíces, multiplicado por el paso de tiempo, pertenece a la región de estabilidad, el error cometido por el método numérico estará acotado.

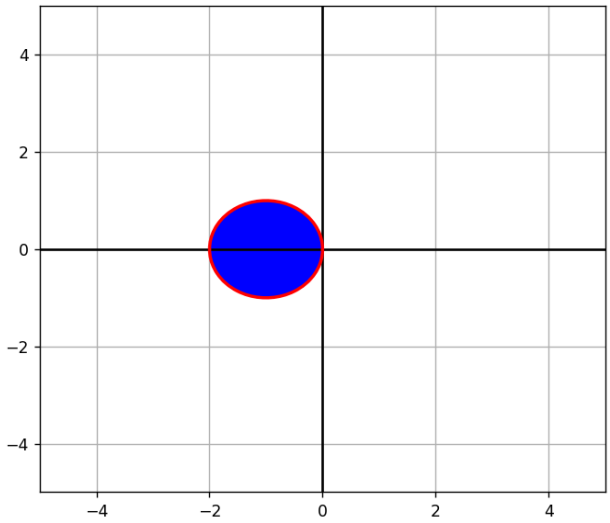

Figura 4.24: Región de estabilidad del método de Euler Explícito.

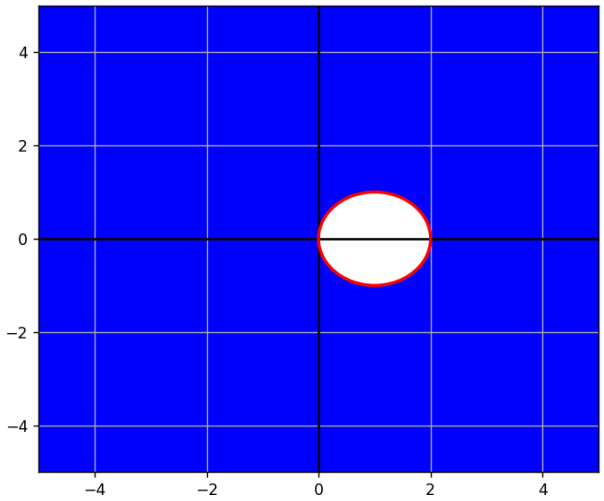

Figura 4.25: Región de estabilidad del método de Euler Implícito.

La región de estabilidad del método de Euler se corresponde con el interior de una circunferencia centrada en (-1,0). Por otro lado, la región de Euler implícito se corresponde con el exterior de una circunferencia centrada en (1,0). Mientras que para el método de Euler explícito el error no está acotado para ningún paso de tiempo, para el implícito todo paso de tiempo presenta un error acotado.

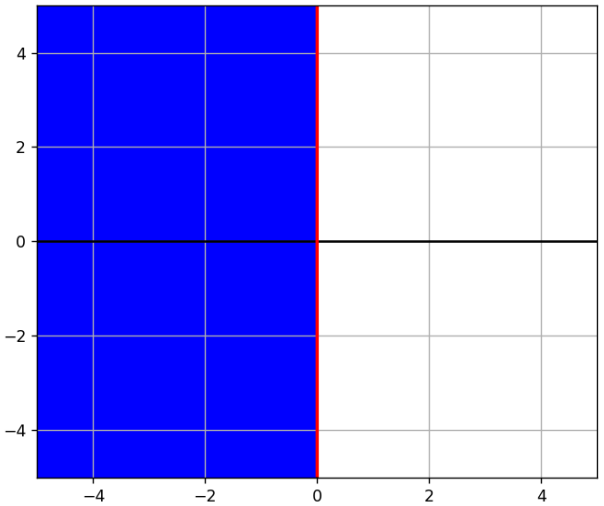

Figura 4.26: Región de estabilidad del método de Crank-Nicolson.

La región de estabilidad de Crank-Nicolson determina una no dependencia del paso de tiempo, puesto que la frontera se corresponde con el eje imaginario.

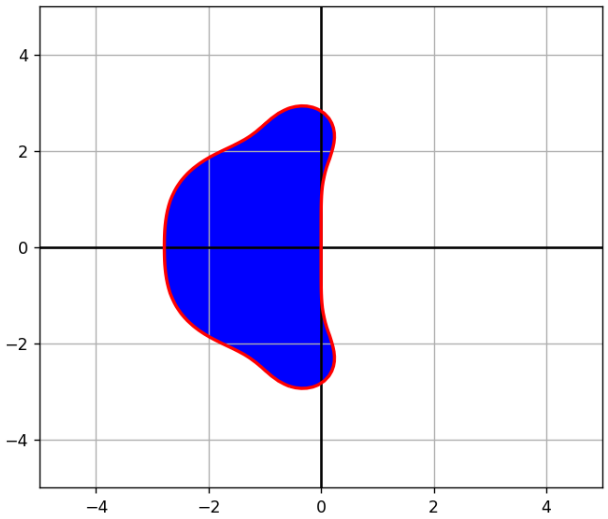

Figura 4.27: Región de estabilidad del método de Runge-Kutta.

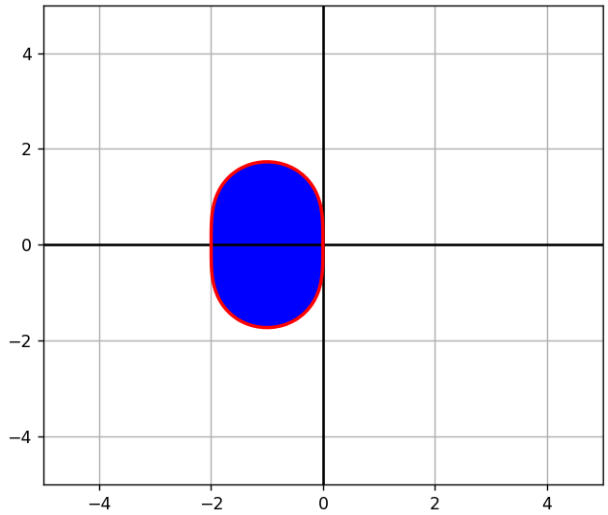

Figura 4.28: Región de estabilidad del método de Leap Frog.

La región de Runge-Kutta es peculiar. Para pasos de tiempo menores al asociado al punto de corte de su frontera con el eje imaginario (0,2.78), el error está acotado, mientras que para pasos de tiempo mayores no lo está. Se puede ver como incluso hay un tramo en el que las soluciones se encuentrar en la frontera de la región, mientra otro tamo se encuentra dentro de ella.

Por último, la región de estabilidad de Leap Frog, se presenta como un óvalo de nuevo centrado en (-1,0), ofreciendo un comportamiento similar al de Euler explícito, exceptuando al tramo en el que la frontera coincide con el eje imaginario. En ese tramo, los pasos de tiempo asociados ofrecen un error acotado.

4.4. *Milestone* V

A continuación, se muestran las propagaciones, mediante los distintos métodos numéricos utilizados, de 4 cuerpos en el problema de N cuerpos, es decir, únicamente bajo la influencia de la presecia de ellos mismos, cada uno sobre los demás. Se ha planetado un caso en el que los cuerpos se encuentran a la misma distancia unos de otros, en torno a un baricentro situado en el centro, y con una velocidad igual en módulo perpendicular a su radio vector.

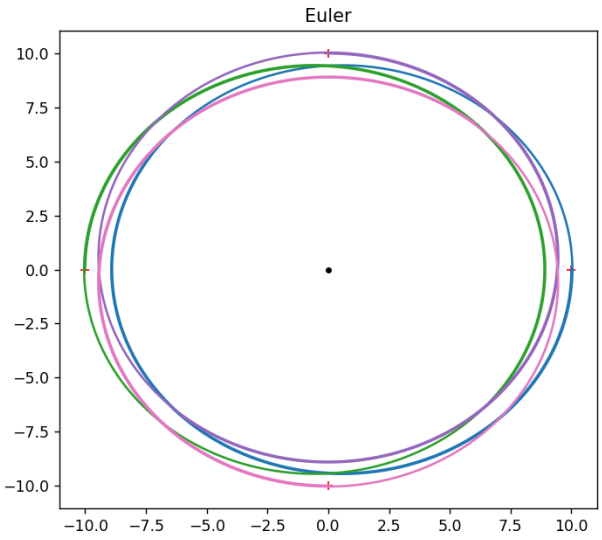

Figura 4.29: Propagación de 4 cuerpos mediante Euler Explícito.

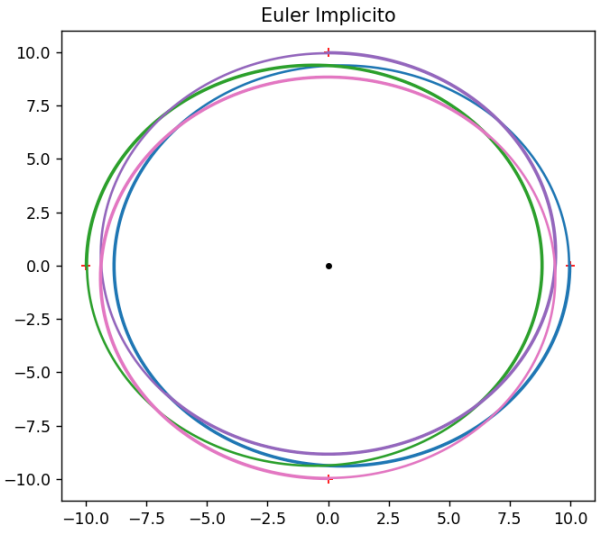

Figura 4.30: Propagación de 4 cuerpos mediante Euler Implícito.

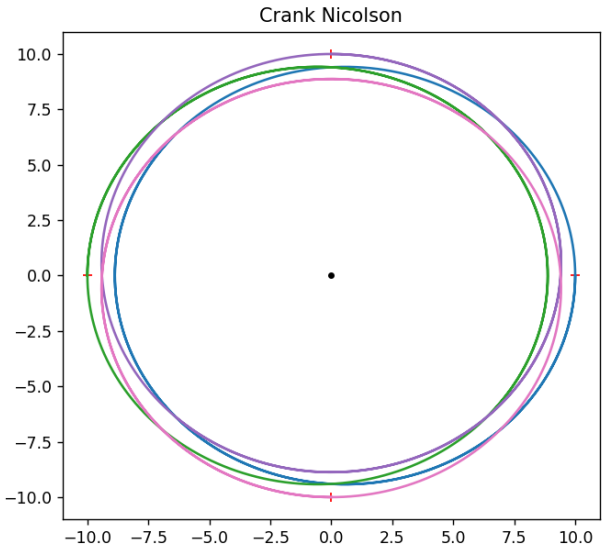

Figura 4.31: Propagación de 4 cuerpos mediante Crank-Nicolson.

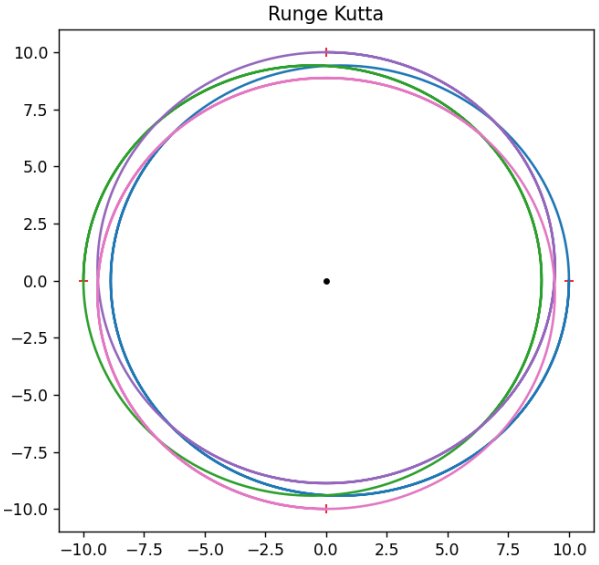

Figura 4.32: Propagación de 4 cuerpos mediante Runge-Kutta.

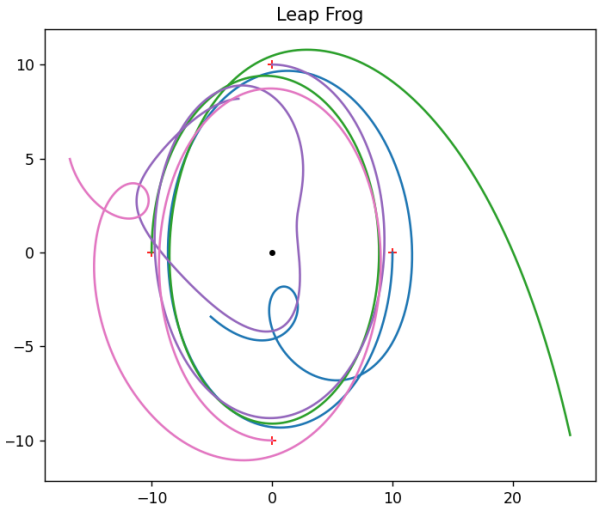

Figura 4.33: Propagación de 4 cuerpos mediante Leap Frog.

Como se puede ver, todos los métodos numéricos ofrecen una solución muy buena a excepción de Leap Frog, cuyo error hace que el sistema se desajuste. El sistema se ha planteado pretendiendo simular la presencia de 4 cuerpos en la misma órbita circular respecto a un centro, y el sistema se mantiene estabel excepto con Leap Frog, donde se puede ver como las diferencias entre soluciones provoca una variación en la influencia entre los cuerpos que desabarata el sistema. Los demás métodos ofrecen una solución suficientemente buena para que la influencia de los cuerpos sea constante, y esto es debido al paso temporal elegido.

Ahora se presenta, únicamente para el método de Euler, la propagación de 10 cuerpos en el problema de N cuerpos.

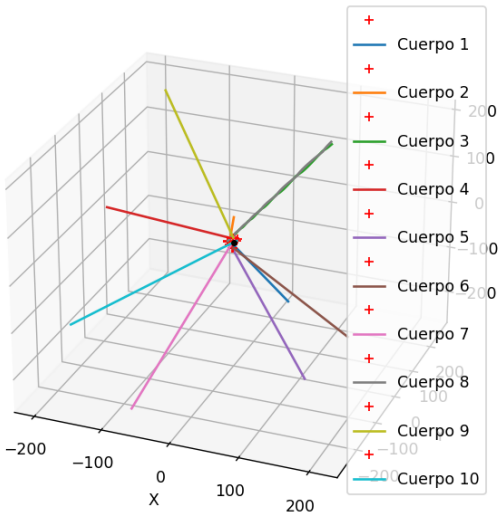

Figura 4.34: Propagación de 10 cuerpos mediante Euler Explícito.

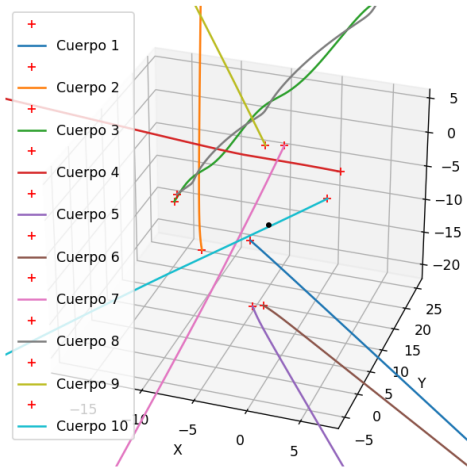

Figura 4.35: Propagación de 10 cuerpos mediante Euler Explícito (zoom).

En este caso, se puede observar la influencia que existe entre ellos únicamente entre los cuerpos 3 y 8, los cuales se propagan dibujando una hélice. El comportamiento observado se debe a las condiciones iniciales aportadas en cuanto a la velocidad inicial, que hace que se separen unos de otros sin influir mucho la presencia de los demás cuerpos.

Procediendo al cambio de condiciones iniciales, se presenta otro caso a continuación.

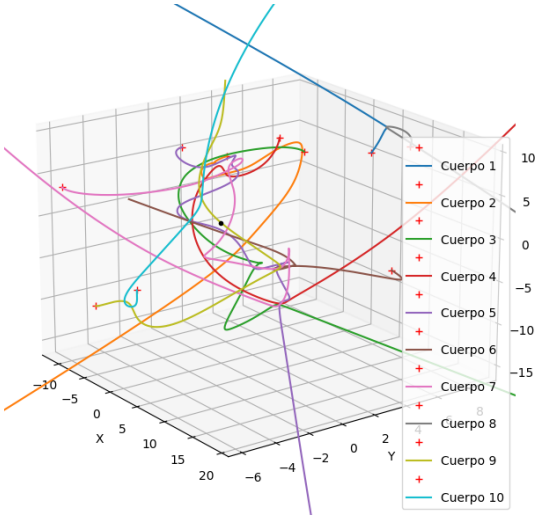

Figura 4.36: Propagación de 10 cuerpos mediante Euler Explícito con otras condiciones iniciales.

Ahora sí que es posible observar como los cuerpos se mueve de forma irregular por la presencia de los demás cuerpos, divergiendo finalmente hacia el infinito, como ocurria en el caso anterior. Cuando se obtiene una velocidad que separa al cuerpo de los demás cuerpos, la influencia que ejercen estos es cada vez menor por lo que la velocidad tiende a ser constante y se alejan unos de otros hacia el infinito.

4.5. *Milestone* VI

5. Conclusiones

Como conclusión final de la realización de estos *milestones* se puede decir que ha sido muy satisfactoria.

En mi caso, he trabajado durante los últimos años creando códigos en Python para los proyectos en los que he estado involucrado. He realizado códigos de propagación orbital durante mi etapa en el departamento de I+D donde estudiabamos la utilización de una constelación de satélites LEO para comunicación global y regional, y he realizados códigos de pre-procesado, procesado y post-procesado de datos en los proyectos de análisis estructural. Por lo que la programación en Python no era desconocida para mi. Aun así, mis conocimientos provenían de un autoaprendizaje realizado a medida que debía afrontar los retos que se me presentaban en los diferentes proyectos en los que he estado involucrado, y no he tenido una base como esta en la que apoyarme. Así, estas clases me han servido como aprendizaje de una forma de programación (la programación funcional), que yo en parte ya la ponía en práctica, pero que he podido entender mejor y perfeccionar, lo cual me ha sido muy útil. Además, he adquirido conocimientos nuevos de Python que desconocía, características de su programación que no aprovechaba y toda una comunidad de programación en la que apoyarme que me va a venir muy bien durante toda mi carrera profesional.

Así pues, como conclusión de las prácticas realizadas puedo decir que estoy muy contento de cómo se plantea la asignatura y los conocimientos que se dan.

6. Anexos

6.1. Anexo I: Códigos de las funciones

```
""" Euler explicito """

def Euler(f,xn,dt,t,**arg):
    Euler.__name__ = "Euler"

    return xn + dt*f(xn,t,**arg), 1
```

```
""" Euler implicito """

def Euler_implicito(f,xn,dt,t,**arg):
    Euler_implicito.__name__ = "Euler Implicito"

    def f_xn1(X):
        return X - xn - dt*f(X,t,**arg)      # f

    return newton(f_xn1,xn,maxiter=200), 1
```

```
""" Crank - Nicolson """

def CN(f,xn,dt,t,**arg):
    CN.__name__ = "Crank-Nicolson"

    def f_xn_xn1(X):
        return  X - (xn + (dt/2)*f(xn,t,**arg)) - (dt/2)*f(X,t+dt,**arg)

    return newton(f_xn_xn1,xn,maxiter=200), 2
```

```
""" Salto de rana (leap frog) """

def LF(f,xn,dt,t,**arg):
    LF.__name__ = "Leap-Frog"

    return xn + dt*f(xn + (dt/2)*f(xn,t),t,**arg), 2
```

```
""" Runge-Kutta (RK) """

# Grado 4
def RK4(f,xn,dt,t,**arg):
    RK4.__name__ = "Runge-Kutta 4"

    return xn + (dt/6)*(f(xn,t,**arg) + 2*f(xn + f(xn,t,**arg)/2, t+dt/2,**arg) + 2*f(xn + f(xn + f(xn,t,**arg)/2, t+dt/2,**arg)/2, t+dt/2,**arg) + f(xn + f(xn + f(xn + f(xn,t,**arg)/2, t+dt/2,**arg)/2, t+dt/2,**arg), t+dt,**arg)), 4
```

```
def Richardson(f_et,f,xo,t1,t2=[]):

    if not t2.any(): t2 = linspace(t1[0],t1[-1],2*len(t1))   # valor por defecto de t2

    C = (len(t2)/len(t1)) # C = dt1/dt2 , como ambos tienen mismo to y tf y aplicando dt = (tf-to)/n siendo n = numero de pasos -> C = n1/n2

    m = 1
    while m*C != int(m*C): m+=1

    U1, q = Cauchy(f_et,f,xo,t1)
    U2, q2 = Cauchy(f_et,f,xo,t2)

    return (U2[:,::int(m*C)] - U1[:,::m])/(1 - 1/C**q)
```

```
def Cauchy(f_et,f,xo,t,output_q=True,**arg):

    x = zeros((len(xo),len(t)))

    x[:,0:1] = xo[:]

    for i in range(1,len(t)):

       x[:,i:i+1], q = f_et(f,x[:,i-1:i],t[i]-t[i-1],t[i],**arg)

    if output_q:
        return x, q
    else:
        return x
```

```
def Convergence_rate(f_et,f,xo,t,imax=[]):

    if imax:

        log_N = []
        log_E = []

        for i in range(1,imax):

            U1 = Cauchy(f_et,f,xo,linspace(t[0],t[-1],i*len(t)),output_q=False)
            U2 = Cauchy(f_et,f,xo,linspace(t[0],t[-1],(i+1)*len(t)),output_q=False)

            log_N.append(log10((i+1)*len(t)))
            log_E.append(log10(norm((U2[0:2,-1] - U1[0:2,-1]))))
            print("  Iteración " + str(i) + " de la convergencia.")
    else:
        log_N = [log10(2*len(t))]
        U1 = Cauchy(f_et,f,xo,linspace(t[0],t[-1],len(t)),output_q=False)
        U2 = Cauchy(f_et,f,xo,linspace(t[0],t[-1],2*len(t)),output_q=False)
        E2 = log10(norm((U2[0:2,-1] - U1[0:2,-1])))
        log_E = [E2]
        E1 = 0; i = 1

        while (E2 - E1) > 1e-2: # a cierta tolerancia

            i+=1
            U1 = Cauchy(f_et,f,xo,linspace(t[0],t[-1],i*len(t)),output_q=False)
            U2 = Cauchy(f_et,f,xo,linspace(t[0],t[-1],(i+1)*len(t)),output_q=False)

            E1 = E2
            E2 = log10(norm((U2[0:2,-1] - U1[0:2,-1])))
            log_N.append(log10((i+1)*len(t)))
            log_E.append(E2)
            print("  Iteración " + str(i) + " de la convergencia.")

    return log_N, log_E
```

```
def Nbodies(Ub=[],t=[],Nb=2):

    if not Ub.any(): Ub = random_Ub(Nb)

    Nf, Nc = shape(Ub)

    if Nc == 1:  # vector de estado
        Nc = Nf/Nb
        formato = "Vector"
    else:
        formato = "Matriz"

    Us = reshape(Ub,(Nb,2,int(Nc/2)))        # [Nb x 2 x Nc/2] reorganizacion para separar componentes de velocidad de componentes de posicion
    r  = reshape(Us[:,0,:],(Nb,int(Nc/2)))   # [Nb x Nc/2] Posicion : [Nb x 2] si tenemos rx y ry ó [Nb x 3] si tenemos rx, ry y rz
    v  = reshape(Us[:,1,:],(Nb,int(Nc/2)))   # [Nb x Nc/2] Velocidad : [Nb x 2] si tenemos vx y vy ó [Nb x 3] si tenemos vx, vy y vz

    Ub1 = zeros((len(Ub),1))

    Us1 = reshape(Ub1,(Nb,2,int(Nc/2)))
    drdt = reshape(Us1[:,0,:],(Nb,int(Nc/2)))
    dvdt = reshape(Us1[:,1,:],(Nb,int(Nc/2)))

    for i in range(Nb):

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

        for j in range(Nb):

            if j != i:
                r_d = r[j,:] - r[i,:]
                dvdt[i,:] = dvdt[i,:] + r_d[:]/(norm(r_d)**3)

    if formato == "Vector":
        return Ub1
    elif formato == "Matriz":
        return reshape(Ub1,(Nb,Nc))
```

```
def Kepler(x,t,GM=mu):

    if len(x) == 4: # x = [rx,ry,vx,vy]

        return array([x[2], x[3], -GM*(x[0])/((x[0]**2 + x[1]**2)**1.5), -GM*(x[1])/((x[0]**2 + x[1]**2)**1.5)])

    elif len(x) == 6: # x = [rx,ry,rz,vx,vy,vz]

        return array([x[3], x[4], x[5], -GM*(x[0])/((x[0]**2 + x[1]**2 + x[2]**2)**1.5), -GM*(x[1])/((x[0]**2 + x[1]**2 + x[2]**2)**1.5), -GM*(x[2])/((x[0]**2 + x[1]**2 + x[2]**2)**1.5)])
```

```
def random_Ub(Nb,dim=3,kp=1,kv=0.01):

    # Inicializacion del vector de estado
    U = zeros((Nb*2*dim,1))
    Us = reshape(U,(Nb,2,dim))

    for i in range(0,Nb):
        phi = uniform(0,2*pi)
        theta = uniform(0,2*pi)
        pos = array([cos(theta)*sin(phi),sin(theta)*sin(phi),cos(phi)])
        for j in range(dim):
            Us[i,0,j] = pos[j]*kp    # Espacio de -1*k a 1*k
            Us[i,1,j] = uniform(-1,1)*kp*kv # Velocidad log10(1/kv) órdenes de magnitud menor que la posición

    return U
```

```
def Stability_region(fet):

    x = linspace(-5,5,100)
    y = linspace(-5,5,100)
    SR = zeros((100, 100))

    for i in range(0,100):
      for j in range(0,100):

        r, q = fet(lambda u, t: complex(x[i], y[j])*u, 1, 1, 0)
        SR[i, j] = abs(r)

    return SR
```

```
def solarsystem():

    # SUN, MERCURY, VENUS, EARTH, MOON, MARS, JUPITER, SATURN, URANUS, NEPTUNE, PLUTO
    M = array([1988500., 0.330, 4.87, 5.97, 0.073, 0.642, 1898., 568., 86.8, 102., 0.0130])*1e+24

    U0 = zeros((11*2*3,1))
    U_0 = reshape(U0, (11, 2, 3) ) # punteros
    r0 = reshape(U_0[:, 0, :], (11, 3))
    v0 = reshape(U_0[:, 1, :], (11, 3))

    with open('Initialposition_solarsystem.txt') as data:
        substrings = data.read().split()
        p = [float64(substring) for substring in substrings]

    p = reshape(p, (11, 3))
    r0[:,:] = p[:,:]

    with open('Initialvelocity_solarsystem.txt') as data:
        substrings = data.read().split()
        v = [float64(substring) for substring in substrings]

    v = reshape(v, (11, 3))
    v0[:,:] = v[:,:]

    return U0,M
```

```
""" Características orbitales """


def v_circular(r,GM=mu):
    if len(r) > 1:
        nr = 0
        for i in range(0,len(r)):
            nr =+ r[i]**2
        nr = (nr)**0.5
    else:
        nr = r

    return (GM/nr)**0.5

def v_orbital(r,a=0,GM=mu):
    if len(r) > 1:
        nr = 0
        for i in range(0,len(r)):
            nr =+ r[i]**2
        nr = (nr)**0.5
    else:
        nr = r

    if a == nr or a == 0:
        return v_circular(r,GM)
    else:
        return (2*GM*(1/r + 1/(2*a)))**0.5
```

```
""" Problema restringido de 3 cuerpos """
def bodies3(U,t):
    r = U[0:2]
    drdt = U[2:4]

    r1 = ((r[0] + mu)**2 + r[1]**2)**0.5
    r2 = ((r[0] + mu - 1)**2 + r[1]**2)**0.5

    dvdt1 = -(1-mu)*(r[0] + mu)/(r1**3) - mu*(r[0] + mu - 1)/(r2**3)
    dvdt2 = -(1-mu)*r[1]/(r1**3) - mu*r[1]/(r2**3)

    return array([ drdt[0], drdt[1], r[0] + 2*drdt[1] + dvdt1, r[1] - 2*drdt[0] + dvdt2])
```

```
""" Puntos de Lagrange """
def Lagrange(U0,t):

    def f(r):
        x = zeros(4)
        x[0:2] = r
        U = bodies3(x,t)
        return U[2:4]

    Lp = zeros([5,2])

    for i in range(5):
        Lp[i,:] = newton(f, U0[i,0:2])

    return Lp
```

```
""" Estabilidad de los puntos de Lagrange """

def Stb_Lagrange(U0,t):

    def Jacobian (F, xp):

        N = size(xp)
        dx = 1e-10

        Jac = zeros([N,N])

        for j in range(N):
            x = zeros(N)
            x[j] = dx
            Jac[:,j] = ( F(xp + x, t) - F(xp - x, t) ) /(2*dx)

        return Jac

    A = Jacobian(bodies3, U0)
    values, vectors = eig(A)

    return values
```

```
""" Runge-Kutta Embebido """
def RKE(f,xn,dt,t):
    RKE.__name__ = "RK45"

    eps = 1e-10

    V1 = RK45("First", f,xn,dt,t)
    V2 = RK45("Second", f,xn,dt,t)

    (a, b, bs, c, q, Ne) = Butcher_array()

    h = min(dt, Step_size(V1 - V2, eps, min(q), dt))

    N = int( dt / h ) + 1
    h = dt / N

    V1 = xn; V2 = xn

    for i in range(N):
        time = t + i * dt / N
        V1 = V2

        V2 = RK45("First",f,V1,h,time)

    U2 = V2

    ierr = 0

    return U2, [4,5]
```

```
""" Step size """
def Step_size(dU, eps, q, dt):

    if( norm(dU) > eps ):
        size =  dt *( eps / norm(dU) )**( 1 / ( q + 1) ) # Step_size
    else:
        size = dt # Step_size

    return size
```

```
""" RK para embebido """
def RK45(tag,f,xn,dt,t):

    (a, b, bs, c, q, Ne) = Butcher_array()

    N = len(xn)
    k = zeros( [Ne, N] )

    aux = f( xn, t + c[0]*dt )

    k[0:1,:] = reshape(aux,(1,len(xn)))


    if tag == "First":

        for i in range(1,Ne):
            Up = xn

            for j in range(i):

                aux = dt * a[i,j]*k[j,:]
                Up[:,0:1] = Up[:,0:1] + reshape(aux,(len(xn),1))


            k[i:i+1,:] =  reshape(f( Up, t + c[i]*dt ),(1,len(xn)))

        U2 = xn + reshape(dt * matmul(b,k),(len(xn),1))

    elif tag == "Second":
```

```
elif tag == "Second":

    for i in range(1,Ne):
        Up = xn

        for j in range(i):
            aux = dt * a[i,j]*k[j,:]
            Up[:,0:1] = Up[:,0:1] + reshape(aux,(len(xn),1))

        k[i:i+1,:] =  reshape(f( Up, t + c[i]*dt ),(1,len(xn)))

    U2 = xn + reshape(dt * matmul(bs,k),(len(xn),1))

return U2
```

```
""" Bucher Array """

def Butcher_array():
    q = [5,4]
    Ne = 7

    a = zeros( [Ne, Ne-1] )
    b = zeros( [Ne] )
    bs = zeros( [Ne] )
    c = zeros( [Ne] )

    c[:] = [ 0., 1./5, 3./10, 4./5, 8./9, 1., 1. ]

    a[0,:] = [          0.,           0.,           0.,         0.,           0.,     0. ]
    a[1,:] = [      1./5  ,           0.,           0.,         0.,           0.,     0. ]
    a[2,:]= [      3./40 ,        9./40,           0.,         0.,           0.,     0. ]
    a[3,:] = [     44./45 ,      -56./15,        32./9,         0.,           0.,     0. ]
    a[4,:] = [ 19372./6561, -25360./2187,  64448./6561,  -212./729,           0.,     0. ]
    a[5,:] = [  9017./3168,    -355./33 ,  46732./5247,    49./176, -5103./18656,     0. ]
    a[6,:]= [    35./384 ,           0.,    500./1113,   125./192, -2187./6784 , 11./84 ]

    b[:]  = [ 35./384   , 0.,   500./1113,  125./192,  -2187./6784  ,  11./84  ,     0.]
    bs[:] = [5179./57600, 0., 7571./16695,  393./640, -92097./339200, 187./2100, 1./40 ]

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