


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

MÁSTER UNIVERSITARIO EN SISTEMAS ESPACIALES

MILESTONE 2

Ampliación de Matemáticas I

7 de octubre de 2022

Autor:
Alberto García Rincón

Índice

1. Introducción

Como continuación del ejercicio resuelto durante la semana anterior, se ha reestructurado el código referente al programa para resolver el problema de Cauchy aplicado para la resolución de una órbita Kepleriana mediante distintos métodos. Aparte de resolver el problema con el método explícito de Euler y con el método de Runge-Kutta de cuarto orden, que ya se programaron para el anterior ejercicio, se han añadido los métodos de Euler implícito y el método de Crank-Nicolson.

Se procederá a una explicación del código usado para la resolución de este ejercicio y a un breve análisis de los resultados obtenidos.

2. Código de Python

Para este ejercicio se ha procedido a una separación del programa principal o "$main$" (denominado *Milestone_2.py*) donde se llaman a los distintos procesos o funciones que se encuentran en otros archivos individuales.

2.1. *Milestone_2.py*

El programa principal contiene las condiciones iniciales del problema a resolver y va llamando a los distintos subprogramas o funciones para obtener los distintos resultados según el método que se use. Por tanto en el programa "$main$" se tienen todos los datos calculados de los distintos métodos que aquí se implementan. Estos datos se almacenan en una variable, la cual se pasa como argumento a las funciones encargadas de dibujar las gráficas que se encuentran definidas en otro archivo distinto.

```python
#Initiate values
print("Introduzca dt = ")
dt = float(input())
#dt = 0.01       #[s]
tf = 30.        #[s]

R0 = 1.         #[m]
V0 = 1.         #[m/s]
Zeta0 = 0.      #[deg]

N = int(tf/dt)+1    #Nº steps
t = np.linspace(0, tf, N)
U0 = fun.InitStateVector(R0, V0, Zeta0)

#Cauchy solver orbits
U_Euler = cp.CauchyProblem(U0, t, orb.Orbits, met.ExplicitEuler)
U_RK4 = cp.CauchyProblem(U0, t, orb.Orbits, met.RungeKutta4)
U_InvEuler = cp.CauchyProblem(U0, t, orb.Orbits, met.InverseEuler)
U_CN = cp.CauchyProblem(U0, t, orb.Orbits, met.CrankNicolson)
UU = np.array([U_Euler, U_RK4, U_InvEuler, U_CN])

#Energy orbit
E_Euler = orb.OrbitEnergy(U_Euler, t)
E_RK4 = orb.OrbitEnergy(U_RK4, t)
E_InvEuler = orb.OrbitEnergy(U_InvEuler, t)
E_CN = orb.OrbitEnergy(U_CN, t)
E = np.array([E_Euler, E_RK4, E_InvEuler, E_CN])

Met_names = np.array(["Euler", "RK4", "Inverse Euler", "CN"])

#Graphics
fun.plot_Positions(UU, Met_names, dt)
fun.plot_energy(E, Met_names, t)
plt.show()
```

Figura 1: *Milestone_2.py*.

2.2. *CauchyProblem.py*

En otro archivo se tiene programado el problema de Cauchy, al cual se le llama desde el $main$ y se le aporta como argumentos las condiciones iniciales, el tiempo de simulación, la función que define la órbita (la cuál está programada en otro archivo distinto denominado *Orbits.py*) y el método que se usa para la resolución (estos métodos están programados en otro archivo: *Methods.py*).

Este código itera el numero de repeticiones definido como la división entre el tiempo total de la simulación y el intervalo de tiempo de discretización.

```python
def CauchyProblem(U0, t, f, method):
    U = np.array(np.zeros([len(U0), len(t)]))
    U[:,0] = np.transpose(U0)
    for i in range(len(t)-1):
        dt = t[i+1] - t[i]
        U[:,i+1] = method(U[:,i], dt, t, f)

    return U
```

Figura 2: *CauchyProblem.py*.

2.3. *Methods.py*

A continuación se muestra el código de los distintos métodos de cálculo que se utilizan.

Cabe destacar que para el método de Euler inverso y para el método de Crank-Nicolson, al tratarse de métodos implícitos se ha usado el método de $newton$ de la librería de Python $scipy.optimize$. En el caso del método de Euler inverso ha sido necesario la configuración de la tolerancia (tol) en la resolución e indicar el número máximo de interacciones del método para obtener el resultado ($maxiter$) , de lo contrario la función no conseguía llegar a una solución para determinadas condiciones iniciales ($\Delta t \geq 0{,}1s$).

```python
#Euler
def ExplicitEuler(U, dt, t, f):
    return U + dt*f(U, t)

#RK4
def RungeKutta4(U, dt, t, f):
    k1 = f(U, t)
    k2 = f(U+dt*k1/2, t)
    k3 = f(U+dt*k2/2, t)
    k4 = f(U+dt*k3, t)
    return U + dt*(k1 + 2*k2 + 2*k3 + k4)/6

#Inverse Euler
def InverseEuler(U, dt, t, F):
    def Residual(X):
        return X - U - dt * F(X, t)

    a = scipyop.newton(func = Residual, x0 = U, tol = 10e-5, maxiter = 10000)
    return a

#Crank-Nicolson
def CrankNicolson(U, dt, t, F):
    def Residual_CN(X):
        return  X - a - dt/2 *  F(X, t + dt)

    a = U  +  dt/2 * F(U, t)
    b = scipyop.newton(Residual_CN, U)
    return b
```

Figura 3: *Methods.py*.

2.4. *Orbits.py*

En el siguiente archivo de ha programado la función que define la órbita de Kepler. También se ha programado una función que calcula la energía específica de la órbita, esto nos servirá para analizar el comportamiento de las órbitas calculadas a través de los distintos métodos y compararlos entre sí.

```
4   #Kepler orbit
5   def Orbits(U, t):
6       d = (U[0]**2 + U[1]**2)**1.5
7       rx = -U[0]/d
8       ry = -U[1]/d
9       dxdt = U[2]
10      dydt = U[3]
11      return np.array([dxdt, dydt, rx, ry])
12
13  #Orbit energy
14  def OrbitEnergy(U, t):
15      e = np.array(np.zeros(len(t)))
16
17      for i in range(len(t)):
18          Ek = (U[2,i]**2 + U[3,i]**2)/2  #Kinetic energy
19          Ep = -constant.MU_earth/(U[0,i]**2 + U[1,i]**2)**0.5 #Potential energy
20          e[i] = Ek + Ep
21
22      return e
```

Figura 4: *Orbits.py*.

2.5. *Constant.py*

Se han definido unas constantes en otro archivo. Estas constantes son utilizadas por algunas funciones del programa.

```
1   #constants
2
3   G0 = 9.81           #[m/s2]
4   MU_earth = 3.986e14 #[m3 s-2]
```

Figura 5: *constant.py*.

2.6. *Functions.py*

Por último se ha creado otro archivo con el resto de funciones: función para inicializar el vector de estado; funciones para dibujar la posición de la órbita según ejes X e Y y función para graficar la energía específica en función del tiempo.

```python
#Initiate Vector
def InitStateVector(r, v, zeta):
    zetaRad = math.radians(zeta)
    rx = r*math.cos(zetaRad)
    ry = r*math.sin(zetaRad)
    vx = -v*math.sin(zetaRad)
    vy = v*math.cos(zetaRad)
    return np.array([rx, ry, vx, vy])

#Graphics
def plot_Position(U, title): #plot position vector
    plt.figure()
    plt.plot(U[0,:], U[1,:])
    plt.title(title)
    plt.xlabel('Rx [m]')
    plt.ylabel('Ry [m]')

def plot_Positions(UU, names, dt): #subplot all position vectors
    a = 2
    b = 2
    z = 0
    fig, axs = plt.subplots(a, b)
    fig.suptitle("Orbits Position, dt = " + str(dt) + "[s]")
    fig.tight_layout(pad=1.5)

    for i in range(a):
        for y in range(b):
            axs[i, y].plot(UU[z,0], UU[z,1])
            axs[i, y].set_title(names[z])
            z = z + 1

    for ax in axs.flat:
        ax.set(xlabel='Rx [m]', ylabel='Ry [m]')

def plot_energy(e, names, t): #plot specific energy (same graphic)
    plt.figure()
    for i in range(len(names)):
        plt.plot(t, e[i,:], label=names[i])

    plt.title("Specific Energy")
    plt.xlabel('Time [s]')
    plt.ylabel('Energy')
    plt.legend()
```

Figura 6: *Functions.py*.

3. Resultados de los métodos

Se ha fijado un tiempo total de simulación de 30 segundos se han realizado varias simulaciones con distintos Δt para comprobar que el código funciona correctamente. Aquí solo se muestran algunas de esas simulaciones.

3.1. Simulaciones con: $\Delta t = 0{,}1s$

Los métodos de Runge-Kutta y Crank-Nicolson no acumulan apenas error respecto a la resolución analítica durante el tiempo de simulación. Esto queda reflejado en la energía específica obtenida con estos métodos se mantiene constante durante toda la simulación.

Sin embargo el método de Euler explícito va aumentando su energía, lo que se traduce en que el objeto que simulamos en la órbita se aleja cada vez más, formando una espiral.

En el método de Euler inverso ocurre que la energía disminuye durante los primeros 3 segundos, lo que quiere decir que el objeto se acerca cada vez más al origen de coordenadas, hasta que converge en dicho punto y comienza aumentar la energía obteniéndose un resultado no válido.

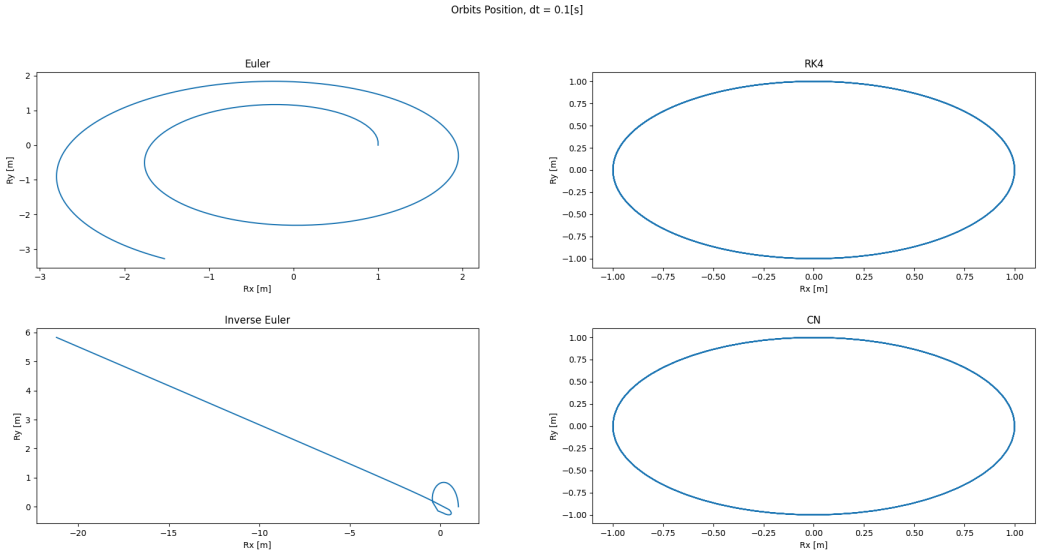

Figura 7: Resolución órbita de Kepler por varios métodos durante $30s$.

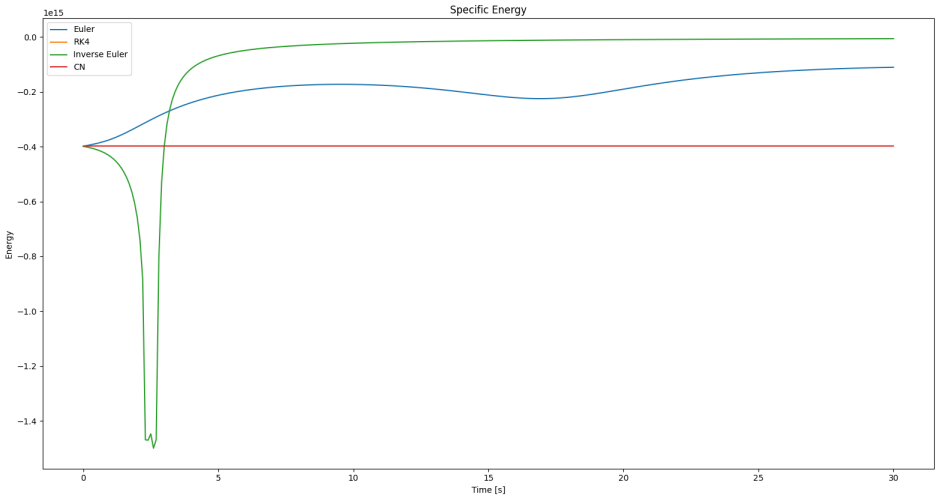

Figura 8: Energía específica con $\Delta t = 0{,}1s$.

3.2. Simulaciones con: $\Delta t = 0{,}001s$

Al disminuir el intervalo de tiempo para realizar el cálculo numérico se obtienen resultados con menor error. Los métodos de Runge-Kutta de orden 4 y de Crank-Nicolson mejoran aún su precisión. El método de Euler explícito reduce el error, así como el método inverso de Euler hace lo mismo.

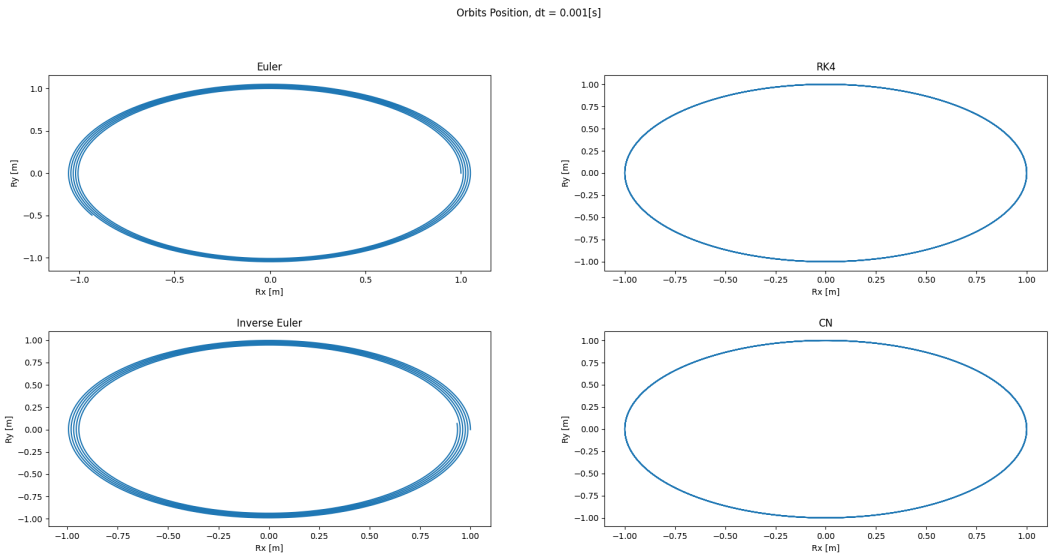

Figura 9: Resolución órbita de Kepler por varios métodos durante $30s$.

Respecto al análisis de la energía específica de la órbita cabe destacar que con el método de Euler explícito se aumenta su valor casi de forma lineal según va pasando el tiempo de

simulación. Ocurre lo contrario y de manera simétrica con el método inverso de Euler.

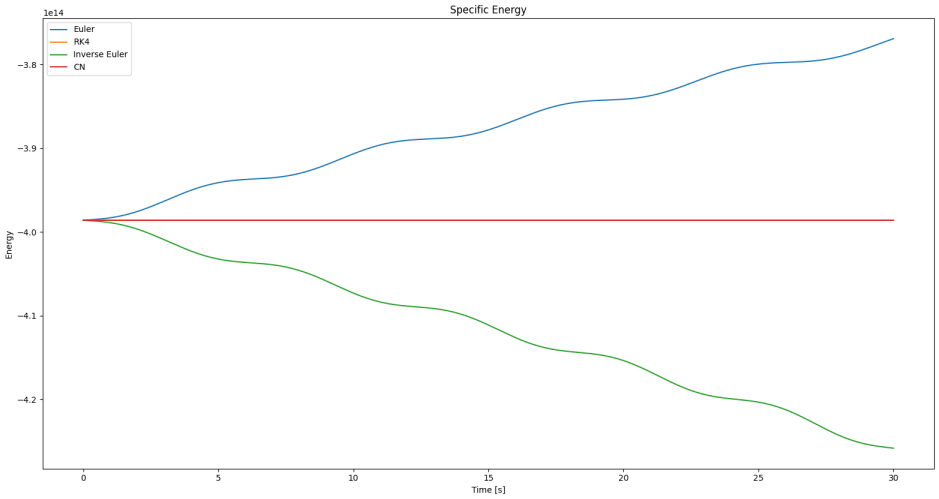

Figura 10: Energía específica con $\Delta t = 0{,}001s$.