Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Milestone 2

Ampliación de Matemáticas I

4 de octubre de 2022

Autor:

Sergio de Ávila Cabral

Índice

1. Introducción

Para la programación de este *Milestone* se ha tenido en cuenta la metodlogía *Top-Down*. Para ello se han definido los siguientes módulos, agrupados en diferentes *Path* dentro de la carpeta *Sources*:

- Milestone_2.py: Actua como *main* y se encuentra en la carpeta misma carpeta *Sources*. En el se encuentran importadas todos los módulos siguientes. A su vez, se definen las condiciones iniciales del problema, las iteraciones, diccionarios y la llamada al problema de Cauchy.
- Physics.py: Incluye la física del problema a resolver. Se encuentra en *Sources/Problems*.
- Temporal_Schemes.py: Se incluyen los diferentes esquemas numéricos. Se localiza en *Sources/ODES*.
- Cauchy_Problem.py: En el se encuentra la definición del problema de Cauchy, cuya localización coincide con la de Temporal_Schemes.py.
- Jab_Newt.py: En el se encuentra la integración numérica de Newton junto con el Jacobiano. Se localiza en el mismo path que las dos anteriores.
- Plots.py: se encarga de graficar las soluciones de los problemas de Cauchy. Se localiza en *Sources/Graf*.

Para todas las simulaciones se han realizado un conjunto de tres N iteraciones y tres Δt pasos.

2. Descripción de módulos

2.1. Milestone_2.py

Como se ha descrito en la Sección 1, actual como *main*. En el se pueden apreciar diferentes secciones entre ellas:

- Condiciones iniciales: inicialida las condiciones del problema y el número de iteraciones deseadas.

```
#Initial Conditions
r0 = [1,0]
v0 = [0,1]
U0 = r0 + v0
N = [1000, 10000, 100000]
```

- Save Plots: se trata de un *boolean* que permite elegir si los *plots* se guardan o solo se muestran.

```
#Save Plots
Save = False # TRue Save the plots / False show the plots
```

- Esquemas numéricos: Se indican los esquemas numéricos que se van a emplear.

```
#Temporal_Schemes to use
T_S = [Euler, RK4, Crank_Nicolson, Inverse_Euler]
T_S_plot = ["Euler", "RK4", "CN", 'Euler_inver']
```

- Inicialización de Diccionarios: Inicialización de los diccionario que se emplearán.

```
#Initiation of Dictionaries
U_dic = { }
t_dic = { }
```

- Tiempos: Definición de los tiempos para las diferentes simulaciones.

```
#Times for simulations
for i in range(len(N)):
    t_dic[str(i)] = linspace(0,20,N[i])
```

  Puesto que se definen diferentes *linspace* con diferentes dimensiones, se introducen de forma dinámica en el diccionario, creando nuevas *Keys* que serán empleadas posteriormente.

- Simulaciones: Se realizan las diferentes simulaciones a través del problema de Cauchy.

```
#Simulations
for j in range ( len(T_S_plot) ):
        for x in t_dic: # x is a str and U is creating new keys
                U_dic[x]= C_P(Kepler, t_dic[x], U0, T_S[j])
                print(T_S_plot[j] + " calculado \n")
                Plot_CP(U_dic[x],t_dic[x], T_S_plot[j], Save)
                # x return the value of the key
        Plot_CP_all(U_dic,t_dic, T_S_plot[j], Save)
```

  Al igual que en el anterior diccionario, se definen nuevos *Keys* de forma dinámica. Como se puede apreciar, las *Keys* de ambos diccionarios están sincronizados, simplificando la programación de los *plots*. Del Mismo modo, se puede apreciar que se realiza la llamada a tres funciones:

  - *C_P*: en la cual se introducen los diferentes esquemas numéricos, el problema físico, los tiempos y las condiciones iniciales.
  - *Plot_CP*: a la cual introducen los resultados individualmente junto con el esquema introducido y el *boolean* para guardar o no las imágenes.
  - *Plot_CP_all*: Realiza la misma función que el anterior, pero grafica juntos los resultados de un esquema numérico.

2.2. Temporal_Schemes.py, Cauchy_Problem.py, Jac_Newt.py

En los módulos que tratan los siguientes subapartados, se ha procedido a reducir el numero de variables empleadas como se indición en el *Issue* del *Milestone 1*.

Temporal_Schemes.py

Todos los esquemas numéricos aplicados en el *Milestone 1* se introducen de forma más simplifica: Se introduce como *input* de las funciones la variable *F*, que será el problema físico que se querrá resolver; y se evita el uso redundante de variables al poner la solución en el *return*. Ejemplo de ello es el esquema numérico *Runge-Kutta de cuarto orden*:

```
#Runge-Kutta cuarto orden
def RK4(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/6 * (k1 + 2 * k2 + 2 * k3 + k4)
```

Cauchy_Problem.py

Para realizar el problema de Cauchy se defienen el número de iteraciones *N*, el *array U* y el $\Delta t = t_{i+1} - t_i$.

```
def C_P(F, t, U0,T_S):

    N = len(t)-1 #Because of the divisons of the time's linspace
    U = zeros( (len(U0), N+1 ) )
    U[:,0] = U0

    for i in range(N):
        U[:, i+1] = T_S(U[:,i] , t[i+1]-t[i], t[i], F)

    return U
```

Jac_Newt.py

En este módulo se introducen las funciones *Jacobiano* y *Newton*, con el objetivo de comparar *fsolve* de *Scipy* con este mismo.

En la primera función se puede observar la programación del Jacobiano de una función cualquiera.

```
def Jacobiano (F, xp):
    N = size(xp)
    dx = 1e-3

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

Y tras este, se define el método de Newton.

```
def Newton(F,x0):
    N = size(x0)
    dx = 2 * x0
    it = 0 #Initialization of iterations
    itmax = 1000 #max iterations
    eps = 1 # initial error

    while (eps > 1e-8) and (it <= itmax):
        it = it + 1
        Jab = Jacobiano(F, x0)
        b = F(x0)
        dx = dot( inv(Jab), b)
        x0 = x0 - dx
        eps = norm(dx)
    return x0
```

2.3. Physics.py

Aplicando la metodolgía *Top-Down* explicada en clase, el problema físico se define en un módulo aparte y de manera más simplificada que en el *Milestone 1*.

```
#Kepler
def Kepler(U, t):

    x = U[0]; y = U[1]; x_dot = U[2]; y_dot = U[3]
    d = ( x**2 + y**2 )**1.5

    return array( [ x_dot, y_dot, -x/d, -y/d ] )
```

2.4. Plots.py

En el módulo Plots se definen las funciones *Plot_ CP*, grafica una única función; y *Plot_ CP_ all*, grafica todas las funciones relacionadas con un mismo esquema numérico. Estas funciones están concebidas para seguir la metodología *Top-Down* y simplificar el *main*. En estas funciones se definen los elementos básicos para graficar los resultados y se añade un condicional para guardar los archivos.

```
if not Save:
    plt.show()

else:
    ##Save Plots
    file = T_S + "_dt_" + str(dt) +".png"
    path = os.path.join(os.getcwd(), 'Plots')

    if not os.path.exists(path):
        os.makedirs(path)

    plt.savefig(os.path.join(path, file))
```

Como se puede ver, en estas líneas de código se crea un condicional para el *boolean Save*. Si este es falso, solo enseña las gráficas, pero si es cierto comienza una serie de sentencias para guardar. Esta serie de sentencias crea, en el caso de no haber, una carpeta *"Plots"* en la misma localización que el *main*. De esta forma este modulo es utilizable para cualquier otro *main*.

3. Resultados

A continuación se presenta los resultados obtenidos para los diferentes esquemas numéricos y los diferentes Δt con los que se han realizado los cálculos.

3.1. Resultados Euler

Como se puede apreciar en la Figura 1, hay una fuerte divergencia para $\Delta t \approx 0{,}01$, reduciéndose de forma considerable dicha divergencia para $\Delta t \approx 0{,}0001$.

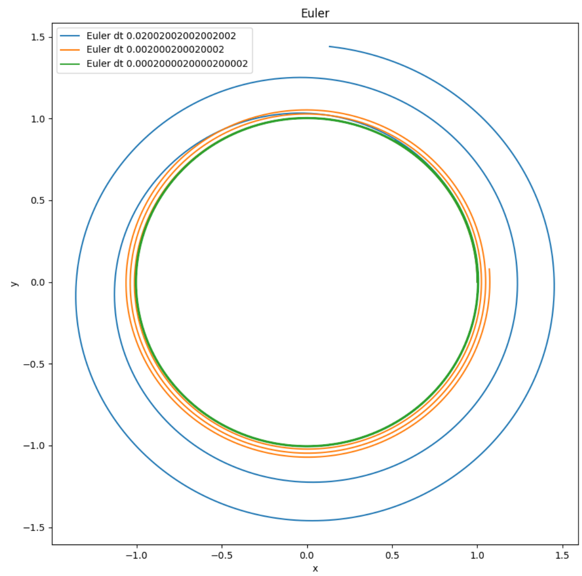

Figura 1: Resultados de Euler para diferentes Δt

3.2. Resultados Runge-Kutta orden 4

Como se puede apreciar en la Figura 2, este esquema numérico tiene una gran precisión para cualquiera de sus Δt con un tiempo de procesamiento muy pequeño.

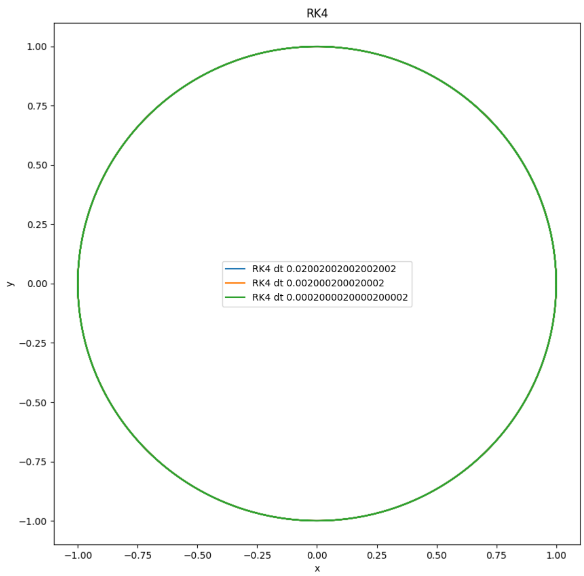

Figura 2: Resultado de RK4 para diferentes Δt

3.3. Resultados Crank-Nicolson

Este esquema produce resultados de gran precisión, como se puede observar en la Figura 3. En comparación con el anterior esquema, resulta de un procesamiento más lento

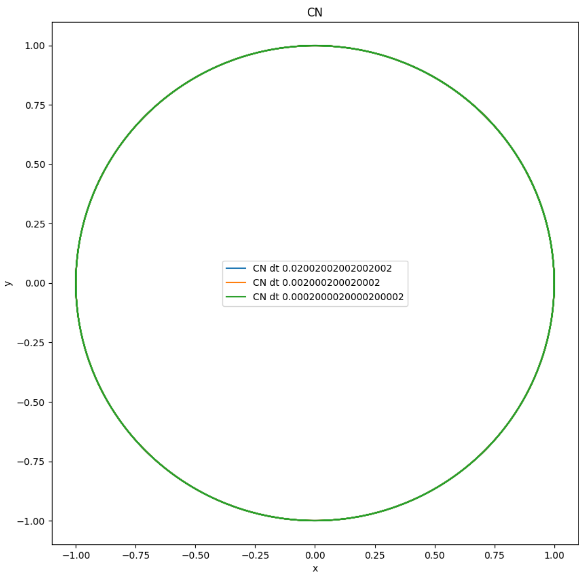

Figura 3: Resultados de Crank-Nicolson para diferentes Δt

En el caso de emplear como función una integración numérica de Newton, el tiempo de procesamiento se ve muy incrementado, de forma notable a partir de $\Delta t \approx 0{,}0001$, produciendo resultados análogos, como los mostrados en la Figura 4.

3.4. Resultados Euler inverso

Por último, se muestran los resultados del esquema numérico de Euler Inverso. Este esquema devuelve resultados análogos a los mostrados en la Sección 3.1, con divergencia en función de la Δt, pero de sentido inverso, como se puede apreciar en la espiral. Este resultado en la Figura 5.

Del mismo modo que en la Sección 3.3, se procede a realizar una integración numérica de Newton, obteniéndose resultados análogos, a excepción del resultado de $\Delta t \approx 0{,}01$. Como se puede apreciar en la Figura 6, hay una fuerte divergencia del resultado numérico con respecto al resultado exacto para $\Delta t \approx 0{,}01$.

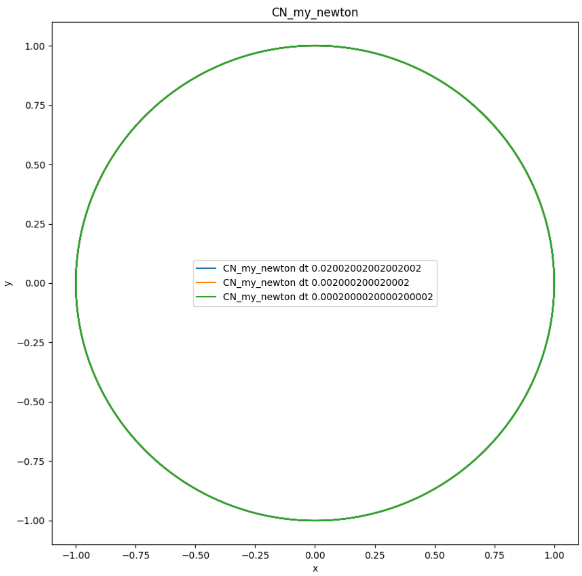

Figura 4: Resultados de Crank-Nicolson con función de Newton

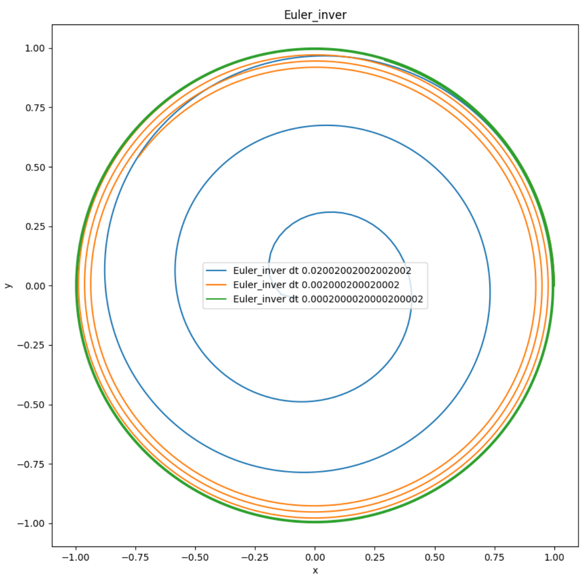

Figura 5: Resultados de Euler inverso para diferentes Δt

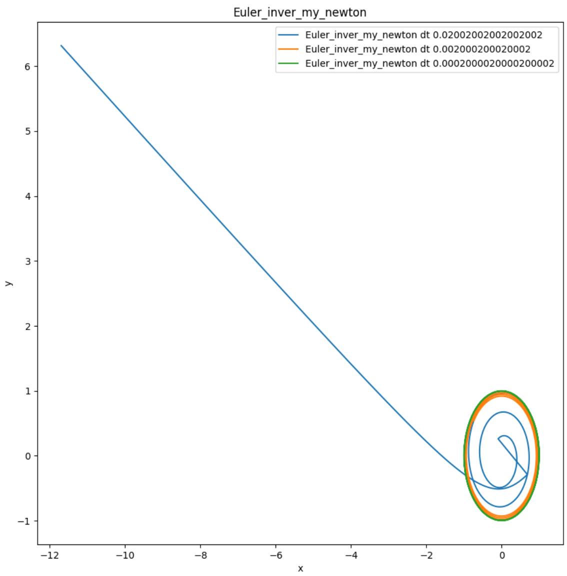

Figura 6: Resultados de Euler inverso con función de Newton