


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

MILESTONE II

AMPLIACIÓN DE MATEMÁTICAS I

8 de octubre de 2022

Autor:
SERGIO LÓPEZ ACEDO
02740571 - Y

Índice

Índice de figuras

1. Introducción

En el presente informe se exponen los contenidos asociados al hito 2 de la asignatura, el cual parte del hito anterior consistente en la integración numérica de una órbita kepleriana para el problema de valores iniciales (problema de Cauchy), que se recuerda en la ecuación (1).

$$\begin{aligned} \ddot{\vec{r}} &= -\frac{\vec{r}}{\|\vec{r}\|^3} \\ \vec{r}(0) &= (1,0) \\ \dot{\vec{r}}(0) &= (0,1) \end{aligned} \quad . \tag{1}$$

Como novedad, se propone una abstracción en cuanto a la estructuración del código, de modo que, siguiendo una aproximación cercana a la programación funcional, se realice una metodología *top-down* del mismo. Para ello, se presenta una enumeración de los archivos utilizados:

- MILESTONES_AM1.py: módulo principal encargado de importar los módulos del hito a utilizar para introducir los inputs del problema, es decir, el tiempo de simulación, t_f, el número de puntos de integración, N y las condiciones iniciales del vector de estado U, U^0.
- MILESTONE_02.py: módulo del hito en cuestión, encargado de importar el resto de módulos de la librería personal que participan en el problema, *CauchyProblem.py*, *Euler_Scheme.py*, *CrankNicolson_Scheme.py*, *RK4_Scheme.py*, *InvEuler_Scheme.py* y *Kepler.py*, que se exponen a continuación.
- AM1_LIBRARY, librería que almacena los módulos *ORBITS.py* (en la categoría *Physics*) y *EDO.py* y *TEMP_SCHEMES.py* (categoría *ODES*).
  - ORBITS.py se encarga de almacenar las funciones que detallen la órbita, en este caso, el problema de Kepler.
  - EDO.py selecciona el problema numérico para nuestro problema, que es un problema de valores iniciales o de Cauchy.
  - TEMP_SCHEMES.py recoge los esquemas numéricos que se pueden utilizar: Euler, Euler inverso, Cranck-Nicolson o Runge-Kutta 4.

2. Código

En esta sección se presentan los detalles del código junto con la explicación asociada a los mismos. Se ordena jerárquicamente, en módulos que se encargan de asociar funciones relativas a la física del problema, *Kepler.py* y los esquemas temporales a utilizar en la integración numérica *TEMP_SCHEMES.py*. Por encima, se encuentra la EDO a resolver,

CauchyProblem que necesita de las funciones de los módulos anteriores para el cálculo completo.

Este módulo, es utilizado por *MILESTONE_ 02.py* para realizar la selección de métodos, discretización temporal y mostrar las gráficas, todo ello a partir de las condiciones iniciales otorgadas en *MILESTONES_ AM1.py*, en el escalón más alto.

En la figura 2.1 se recoge el código asociado al módulo *MILESTONES_ AM1.py*, el cuál se ha estructurado para únicamente tener que llamar a la función *SimulationMIL2* encargada de únicamente necesitar como inputs los valores de tiempo de simulación, t_f, números de instantes temporales, N y el vector de estado en el instante inicial, U^0, definido de acuerdo a:

$$\vec{U}^0 = \begin{pmatrix} x_0 \\ y_0 \\ \dot{x}_0 \\ \dot{y}_0 \end{pmatrix} \tag{2}$$

.

```
import MILESTONE_02
MILESTONE_02.SimulationMIL2(  tf = 20, N = 200,
                        U0 = array( [ 1., 0., 0., 1. ] ) )
```

Figura 2.1: MILESTONES_AM1.py code extract

Entrando al detalle de los últimos módulos, en cuanto a orden jerárquico, en la figura 2.2 se recoge *ORBITS.py* que únicamente necesita el vector de estado U y el tiempo t para devolver el problema debidamente expresado para el problema.

```
def Kepler(U, t):
    x = U[0]; y = U[1]; dxdt = U[2]; dydt = U[3]
    d = ( x**2  +y**2 )**1.5
    return  array( [ dxdt, dydt, -x/d, -y/d ] )
```

Figura 2.2: ORBITS.py code extract

```
def CauchyProblem( F, t, U0, TempScheme):
     N, Nv=  len(t)-1, len(U0)
     U = zeros( (N+1, Nv), dtype=float64 )
     U[0,:] = U0
     for n in range(N):
        U[n+1,:] = TempScheme( U[n, :], t[n+1] - t[n], t[n],  F )
     return U
```

Figura 2.3: EDO.py code extract

En cuanto a *EDO.py*, en la figura 2.3, se realiza un bucle por el cual se llama al esquema temporal correspondiente para obtener el valor del vector de estado. En lo referente a los esquemas temporales apreciados en 2.4, dando como inputs los vectores de estado, paso temporal, tiempo y función F que se corresponde a la matriz que devuelve la función *Kepler* anteriormente comentada, devuelve el valor de U en el instante posterior. Es por ello que en *CauchyProblem* (función), se importa dicha función F de *Kepler* y el esquema temporal escogido, se hace el cálculo completo.

```
1  def Euler_Scheme(U, dt, t, F):
2      return U + dt * F(U, t)
3
4  def RK4_Scheme(U, dt, t, F):
5      k1 = F(U,t)
6      k2 = F(U + k1 * dt/2, t + dt/2)
7      k3 = F(U + k2 * dt/2, t + dt/2)
8      k4 = F(U + k3 * dt, t + dt)
9      return U + (dt/6) * (k1 + 2*k2 + 2*k3 + k4)
10
11 def InvEuler_Scheme(U, dt, t, F):
12     def InvEuler_Eq(X):
13           return X - U - dt * F(X, t)
14     return newton(func = InvEuler_Eq, x0 = U )
15
16 def CrankNicolson_Scheme(U, dt, t, F ):
17     def CN_Eq(X):
18          return  X - a - dt/2 *  F(X, t + dt)
19     a = U  +  dt/2 * F( U, t)
20     return newton( CN_Eq, U )
```

Figura 2.4: TEMP_SCHEMES.py code extract

Por último, el módulo *MILESTONE_ 02.py*, se aprecia en la figura 2.5.

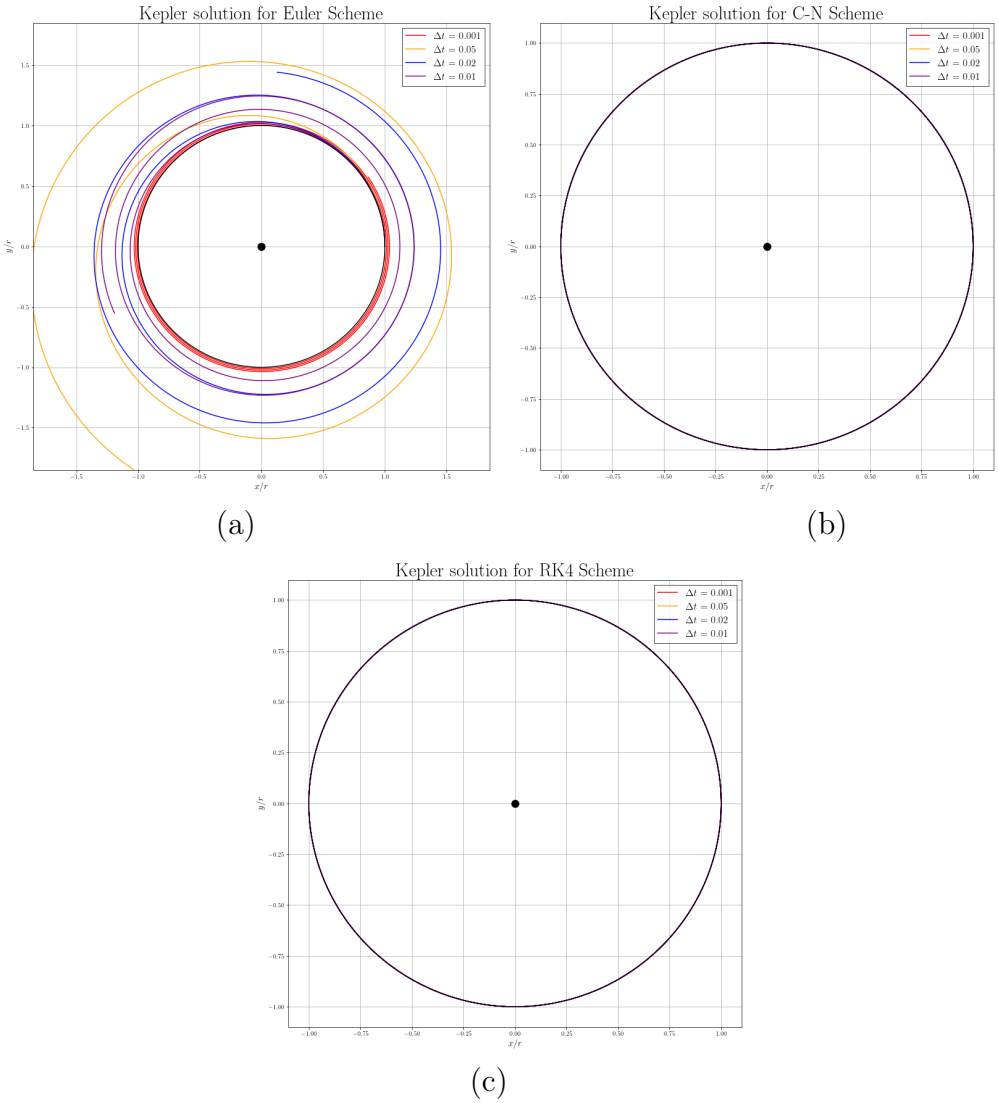
```
from AM1_LIBRARY.ODES.EDO import CauchyProblem
from AM1_LIBRARY.ODES.TEMP_SCHEMES import Euler_Scheme
from AM1_LIBRARY.ODES.TEMP_SCHEMES import CrankNicolson_Scheme
from AM1_LIBRARY.ODES.TEMP_SCHEMES import RK4_Scheme, InvEuler_Scheme
from AM1_LIBRARY.PHYSICS.ORBITS import Kepler

def SimulationMIL2(tf, N, U0):
    t = linspace(0, tf, N)
    schemes = [ Euler_Scheme, RK4_Scheme, CrankNicolson_Scheme,
    InvEuler_Scheme ]
    LegendScheme = ["Euler Scheme", "RK4 Scheme", "Cranck Nicolson Scheme",
    "InvEuler Scheme"]
    for indic in range(len(schemes)): #(a,b] en in range, enumerate
       method = schemes[indic]
       U =  CauchyProblem( Kepler, t, U0, method)
       fig, ax = plt.subplots(1,1, figsize=(11,11),
       constrained_layout='true')
       ax.set_xlim(-1.85,1.85)
       ax.set_ylim(-1.85,1.85)
       ax.set_title("Kepler for "+ LegendScheme[indic], fontsize=30)
       ax.grid()
       ax.set_xlabel(r'$x/r$',fontsize=15)
       ax.set_ylabel(r'$y/r$',fontsize=15)
       plt.plot(U[:,0] , U[:,1])
       plt.show( )
```

Figura 2.5: MILESTONE_02.py code extract

3. Resultados

En esta sección, se exponen los resultados obtenidos para el segundo hito. La única novedad es la incorporación del esquema tipo Euler Inverso, ya que los esquemas Euler, Cranck-Nicolson y Runge-Kutta de cuarto orden no cambian y se comentaron en el hito anterior, por lo que solamente se recordarán sus variaciones, al residir la mayor importancia de este trabajo en la reestructuración del código ya mencionada.

Con ello, se presenta la variación para diversos pasos temporales $\Delta t = [0{,}2,\ 0{,}1,\ 0{,}05,\ 0{,}001]$ para los esquemas ya conocidos del hito anterior en la figura 3.1.



(a) (b)

(c)

Figura 3.1: Solución del problema de Kepler con un $t_f = 20$ y pasos temporales comentados para esquema de (a) Euler, (b) Cranck-Nicolson y (c) RK4.

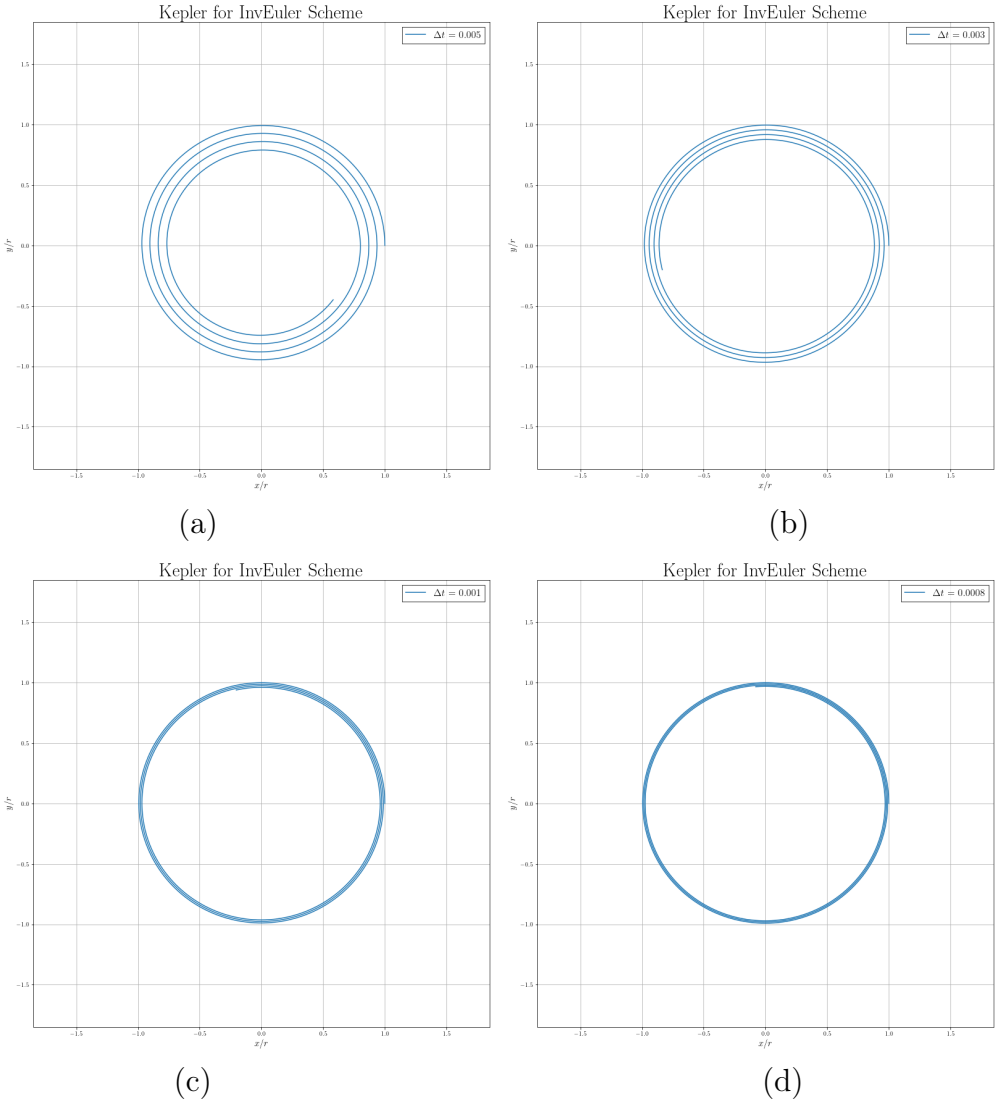
Como se comentó, el esquema de menor orden, Euler, acumula un mayor error y sensibilidad al paso temporal, por lo que aumentando el número de puntos temporales, aumenta su precisión. En RK4 y Cranck-Nicolson no se aprecia efecto con el paso temporal.

3.1. Esquema de Euler Inverso

Para este esquema, el método se corresponde con la ecuación (3)

$$U^{n+1} = U^n + \Delta t^n F^{n+1} \quad . \tag{3}$$

En cuanto a los resultados, se presentan para valores de $\Delta t = [0{,}005,\ 0{,}003,\ 0{,}001,\ 0{,}0008]$, valores menores a los anteriores ya que si no, no convergía el método de newton utilizado para la resolución de la ecuación anterior.



(a) (b)

(c) (d)

Figura 3.2: Solución del problema de Kepler con esquema tipo Euler Inverso para un $t_f = 20$ y pasos temporales (a) $\Delta t = 0{,}005$, (b) $\Delta t = 0{,}003$, (c) $\Delta t = 0{,}001$ y (d) $\Delta t = 0{,}0008$.

En la figura 3.2 se presentan los resultados anteriores, observando como a medida que disminuye el paso temporal, mejora la aproximación de la órbita (teniendo en cuenta que los valores deben ser más precisos que para esquemas anteriores para segurar la convergencia del método Newton), degenerando la órbita a una elipse que disminuye de semiejes.

Referencias

[1] HERNANDEZ, Juan and TAMARIT, Rapado, *Advanced Programming for Numerical Calculations: Climbing Python & Fortran*, 2022.

[2] HERNANDEZ, Juan and ZAMECNIK, Mario A *Calculo Numerico en Ecuaciones Diferenciales Ordinarias*, 2022.

[3] HERNANDEZ, Juan, *Interpolación polinomica de alto orden. Metodos espectrales. Aplicacion a problemas de contorno y de condiciones iniciales*, 2020.