Juan Carlos García-Taheño Hijes

1º MUSE ampliación de matemáticas I

Milestone 2:

Justificación de los Apartados:

Después de la clase del jueves 5/10 se han realizado modificaciones al código entregado se realizara el programa milestone2_conCorreciones.py , el cual es un código único y milestone2_modulos.py con sus respectivos módulos para separar el código en partes diferenciadas.

Para milestone2_conCorreciones.py:

1. Condiciones Iniciales y Definición de Función:

Este primer paso es establecer las condiciones iniciales del problema de Kepler, que involucra la posición y velocidad inicial de una partícula en el espacio. Además, se define la función **kepler_force**, que calcula la fuerza de Kepler en función de las posiciones y velocidades actuales de la partícula. Estas condiciones iniciales y la función de fuerza son fundamentales para llevar a cabo la integración numérica y resolver el problema.

```python
from numpy import array, zeros, linspace
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import newton

# Milestone 2: Prototypes to integrate orbits with functions.
#Nota:
#Como se vio en clase, lo optimo seria trabajar con diferentes modulos;
# es decir, crear un modulo con todas las funciones de integracion numerica; Euler, CN, RK, Eulerinv, temporal_schemes
# luego crear otro con los solucionador cauchy_problem.py
# incluso otro con otras funciones por ejemplo non_lineal_sistems.py con funciones como la de newton, la biseccion

#Este codigo sera uno unico


# Condiciones INICIALES

# Defino condiciones iniciales
U0 = array([1.0, 0.0, 0.0, 1.0])  # Condiciones iniciales (x, y, vx, vy)
t0 = 0.0  # tiempo inicial
T = 10  # periodo o tiempo final (2*pi*1=6.28 para una vuelta)

# Crea un vector de tiempo equiespaciado
num_points = 1000  # Numero de puntos deseados
#custom_t = [0.0, 0.1, 0.3, 0.5, 0.7, 1.0, 2.0, 5.0, 7.0] #varia el dt random, no converge el dt es muy amplio
#EL LIMITE EXPLOTA EN EL 0.01 mas grande ya no converge
custom_t = linspace(t0, T, num_points) #10/1000 el dt=0.01
# si num_points 10000 de t0 a T el dt 10/10000=0.001
# si num_points 100 de t0 a T el dt 10/100=0.1, el problema de cauchy no converge
```

Además, se definirá un vector de tiempos para definir los dt que será el custom_t, probando valores me di cuenta que el valor mínimo de dt será de 0.01, también intente crear un vector cuyo dt variase. Desde aquí se puede controlar la precion de los esquemas ya sea aumentando el numero de puntos en el intervalo t0-T o reduciendo T para el mismo numero de puntos y perdiendo parte de la solución.

2. Método de Euler:

El método de Euler es una técnica numérica de integración simple pero menos preciso que algunos otros métodos, como veremos más adelante. Permite avanzar en el tiempo en pequeños pasos y calcular la nueva posición y velocidad de la partícula en cada paso.

Juan Carlos García-Taheño Hijes

1º MUSE ampliación de matemáticas I

```
39      # Funcion de esquema temporal Euler
40      def Euler(U, t1, t2, F):
41          dt = t2 - t1
42          return U + dt * F(U, t1)
```

3. Método de Crank-Nicolson:

El método de Crank-Nicolson es un esquema implícito que mejora la precisión de la integración con respecto al método de Euler. Se busca encontrar una función que minimice la diferencia entre dos estados consecutivos en el tiempo.

```
44      # Funcion de esquema temporal Crank-Nicolson
45      def Crank_Nicolson(U, t1, t2, F):
46          def Residual_CN(X):
47              return X - a - (t2 - t1) / 2 * F(X, t2)
48
49          dt = t2 - t1
50          a = U + (t2 - t1) / 2 * F(U, t1)
51          return newton(Residual_CN, U)
```

Ademas, me di cuenta que para a = 0 resultaria un esquema euler implícito.

4. Método de RK4 (Runge-Kutta de Cuarto Orden):

El método de RK4 es otro enfoque para la integración numérica que ofrece una mayor precisión en comparación con el método de Euler. Se basa en calcular cuatro pendientes ponderadas en varios puntos en el intervalo de tiempo y combinarlas para obtener una estimación más precisa del siguiente estado.

```
53      # Funcion de esquema temporal RK4
54      def RK4(U, t1, t2, F):
55          dt = t2 - t1
56          k1 = F(U, t1)
57          k2 = F(U + dt * k1 / 2, t1 + (t2 - t1) / 2)
58          k3 = F(U + dt * k2 / 2, t1 + (t2 - t1) / 2)
59          k4 = F(U + dt * k3, t1 + (t2 - t1))
60
61          return U + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6
```

5. Método de Euler Inverso:

El método de Euler Inverso es otro esquema implícito que se utiliza para abordar problemas de integración numérica. Al igual que Crank-Nicolson, se enfoca en encontrar una función que minimice la diferencia entre dos estados en el tiempo.

```
63      # Funcion de esquema temporal Inverse Euler
64      def Inverse_Euler(U, t1, t2, F):
65          dt = t2 - t1
66          def cerosINV(X):
67              return X - U - dt * F(X, t1)
68
69          return newton(func=cerosINV, x0=U)
70
71      # me entro la duda el euler inverso era el euler implicito
72      # def implicit_euler(U, t1, t2, F):
73      #     dt = t2 - t1
74      #     def residual(X):
75      #         return X - U - dt * F(X, t2)
76
77      #     return newton(residual, U)
```

Se añadió un extra del implícito de Euler.

6. Función para Integrar un Problema de Cauchy:

Se crea una función llamada **integrate_cauchy** que generaliza la integración numérica de los problemas de Cauchy. Esta función permite seleccionar un esquema temporal (cualquiera de los métodos mencionados) y resolver el problema de Kepler a lo largo de un intervalo de tiempo especificado. Almacenará los resultados en matrices para su posterior visualización y análisis.

```
82      # Define una funcion para integrar el sistema de ecuaciones
83      def integrate_cauchy(EsquemaTemporal, U0, t, F):   #introduce el esquema temporal con el quieres integrar,
84          #luego las condiciones iniciales, el vector t con el que se definira el dt y
85          #por ultimo que F del problema a resolver
86          num_steps = len(t)
87          states = zeros((len(U0), num_steps))   #states hace referencia a la U los estados
88
89          U = U0  #inicio en U0
90          for step in range(num_steps - 1):
91              t1 = t[step]   #donde estan t1 y t2
92              t2 = t[step + 1]
93
94              states[:, step] = U
95
96              U = EsquemaTemporal(U, t1, t2, F)
97
98          # Asegura que la ultima posicion corresponda al ultimo tiempo
99          states[:, -1] = U
100
101         return t, states
102     ###############################################################################

###############################################################################

#           Soluciones
# Llama a la funcion de integracion con tiempos equiespaciados para cada esquema temporal
t_euler, states_euler = integrate_cauchy(Euler, U0, custom_t, kepler_force)
t_crank_nicolson, states_crank_nicolson = integrate_cauchy(Crank_Nicolson, U0, custom_t, kepler_force)
t_rk4, states_rk4 = integrate_cauchy(RK4, U0, custom_t, kepler_force)
t_inverse_euler, states_inverse_euler = integrate_cauchy(Inverse_Euler, U0, custom_t, kepler_force)
###############################################################################
```

Juan Carlos García-Taheño Hijes

1º MUSE ampliación de matemáticas I

```
117     #           Graficos
118     plt.figure(1, figsize=(12, 8))
119
120     plt.subplot(221)
121     plt.plot(states_euler[0, :], states_euler[1, :], label='Euler')
122     plt.title('Orbita - Euler')
123     plt.axis('equal')
124
125     plt.subplot(222)
126     plt.plot(states_crank_nicolson[0, :], states_crank_nicolson[1, :], label='Crank-Nicolson')
127     plt.title('Orbita - Crank-Nicolson')
128     plt.axis('equal')
129
130     plt.subplot(223)
131     plt.plot(states_rk4[0, :], states_rk4[1, :], label='RK4')
132     plt.title('Orbita - RK4')
133     plt.axis('equal')
134
135     plt.subplot(224)
136     plt.plot(states_inverse_euler[0, :], states_inverse_euler[1, :], label='Inverse Euler')
137     plt.title('Orbita - Inverse Euler')
138     plt.axis('equal')
139
140     plt.tight_layout()
141     plt.show()
```

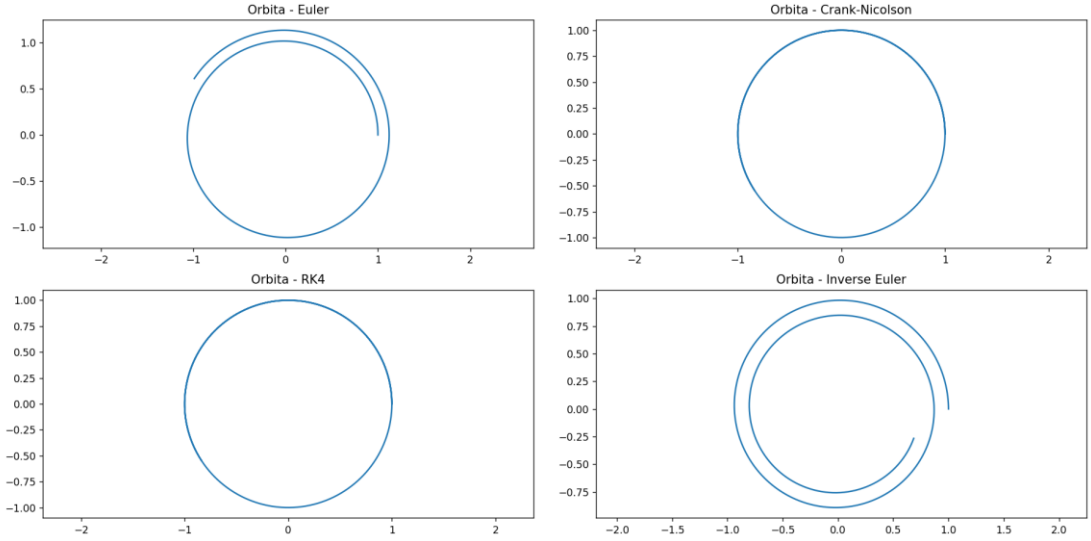

Imagen de la solución para dt=0.01

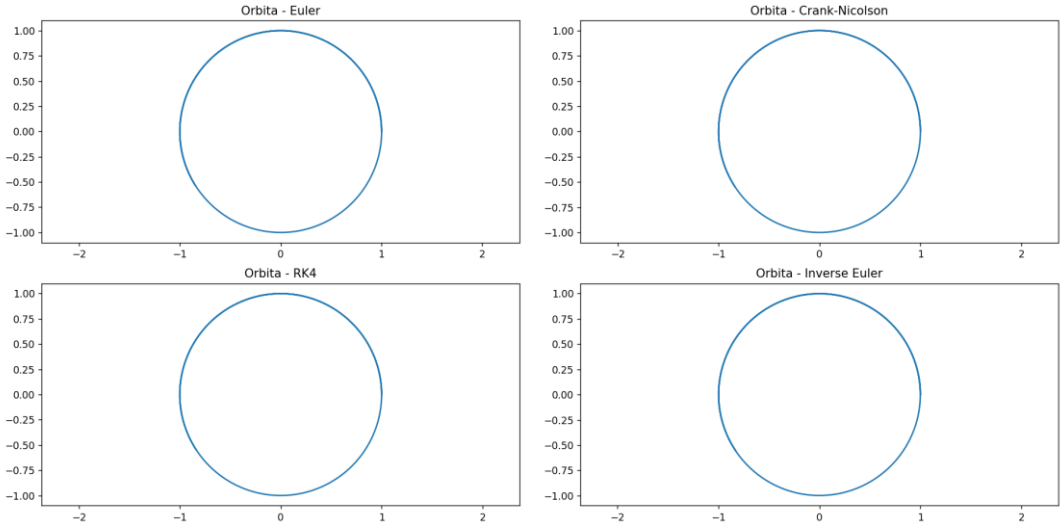

Imagen de la solución para dt=0.0001

Juan Carlos García-Taheño Hijes

1º MUSE ampliación de matemáticas I

7. Explicación de los Resultados al Integrar Kepler:

En los gráficos generados por la integración de Kepler con diferentes esquemas temporales (Euler, Crank-Nicolson, RK4 e Inverse Euler), podemos observar las trayectorias orbitales resultantes de cada método:

- Euler: Este esquema muestra una órbita que se aleja gradualmente del camino correcto debido a su naturaleza explícita. La energía total del sistema no se conserva, lo que resulta en una espiral que se aleja del origen.
- Crank-Nicolson: A diferencia de Euler, el método Crank-Nicolson es implícito y conserva mejor la energía total del sistema. La órbita resultante es mucho más precisa y estable en comparación con Euler.
- RK4: El método de Runge-Kutta de cuarto orden es altamente preciso. La órbita es casi idéntica a la esperada y se considera una mejora significativa sobre los métodos anteriores.
- Inverse Euler: Este esquema también es implícito, pero no del todo preciso.

8. Efecto de Variar el Paso de Tiempo:

Cambiar el valor del paso de tiempo (dt) tiene un impacto directo en la precisión y la eficiencia de la integración numérica.

Un paso de tiempo más pequeño produce resultados más precisos, pero con un mayor costo computacional, ya que se requieren más cálculos para avanzar en el tiempo.

Por otro lado, un paso de tiempo más grande puede generar resultados menos precisos, pero es más eficiente en términos de recursos computacionales. En la práctica, encontrar el equilibrio adecuado entre precisión y eficiencia al ajustar el valor de dt es esencial para obtener resultados confiables en problemas de integración numérica.

Resumen/Conclusión sobre la asignación de variables:

La asignación de variables es una técnica aumenta la legibilidad del código y simplifica la comprensión de las operaciones realizadas.

El costo computacional está mayormente determinado por la complejidad del algoritmo utilizado y la cantidad de operaciones ejecutadas en cada intervalo de tiempo. Además, la asignación de variables puede mejorar la eficiencia del código al evitar cálculos redundantes o innecesarios.

Luego el uso de este método es una práctica recomendable para mejorar la claridad del código; por ejemplo, respecto del hito 1.

Juan Carlos García-Taheño Hijes

1º MUSE ampliación de matemáticas I

Para milestone2_modulos.py:

La estructura de trabajo es la misma pero ahora se han realizados 4 módulos:

El principal: milestone2_modulos.py

Desde aquí se llama al resto de módulos.

```python
from numpy import array, zeros, linspace
import matplotlib.pyplot as plt
from temporal_schemes import Euler, Crank_Nicolson, RK4, Inverse_Euler
from cauchy_problem import integrate_cauchy
from non_linear_systems import custom_newton

#En este codigo se trbaja con 3 modulos, en este:
#Se impoenne las condiciones iniciales y el espaciado del tiempo
#que funcion F a integrar quiza a futuros seria conveniente crear otro modulo para las F
#soluciones
#graficos

###################################
# CONDICIONES INICIALES
U0 = array([1.0, 0.0, 0.0, 1.0])  # Condiciones iniciales (x, y, vx, vy)
t0 = 0.0  # tiempo inicial
T = 10  # periodo o tiempo final (2*pi*1=6.28 para una vuelta)

# Crea un vector de tiempo equiespaciado
num_points = 1000  # Numero de puntos deseados
custom_t = linspace(t0, T, num_points)  # 10/1000 el dt=0.01

###################################
# Funcion para la fuerza de Kepler
def kepler_force(U, t):
    x, y, vx, vy = U[0], U[1], U[2], U[3]
    r = (x ** 2 + y ** 2) ** 0.5
    return array([vx, vy, -x / (r ** 3), -y / (r ** 3)])
```

```python
# Soluciones
t_euler, states_euler = integrate_cauchy(Euler, U0, custom_t, kepler_force)
t_crank_nicolson, states_crank_nicolson = integrate_cauchy(Crank_Nicolson, U0, custom_t, kepler_fc
t_rk4, states_rk4 = integrate_cauchy(RK4, U0, custom_t, kepler_force)
t_inverse_euler, states_inverse_euler = integrate_cauchy(Inverse_Euler, U0, custom_t, kepler_force

###################################
# Graficos
plt.figure(1, figsize=(12, 8))

plt.subplot(221)
plt.plot(states_euler[0, :], states_euler[1, :], label='Euler')
plt.title('Orbita - Euler')
plt.axis('equal')

plt.subplot(222)
plt.plot(states_crank_nicolson[0, :], states_crank_nicolson[1, :], label='Crank-Nicolson')
plt.title('Orbita - Crank-Nicolson')
plt.axis('equal')

plt.subplot(223)
plt.plot(states_rk4[0, :], states_rk4[1, :], label='RK4')
plt.title('Orbita - RK4')
plt.axis('equal')

plt.subplot(224)
plt.plot(states_inverse_euler[0, :], states_inverse_euler[1, :], label='Inverse Euler')
plt.title('Orbita - Inverse Euler')
plt.axis('equal')

plt.tight_layout()
plt.show()
```

Juan Carlos García-Taheño Hijes

1º MUSE ampliación de matemáticas I
Módulo: cauchy_problem.py

```python
from numpy import zeros

# Define una funcion para integrar el sistema de ecuaciones
def integrate_cauchy(EsquemaTemporal, U0, t, F):
    num_steps = len(t)
    states = zeros((len(U0), num_steps))

    U = U0
    for step in range(num_steps - 1):
        t1 = t[step]
        t2 = t[step + 1]

        states[:, step] = U

        U = EsquemaTemporal(U, t1, t2, F)

    # Asegura que la ultima posicion corresponda al ultimo tiempo
    states[:, -1] = U

    return t, states

```

Módulo temporal_schemes.py

```python
from numpy import array
from non_linear_systems import custom_newton

# modulo propio de los esquemas temporales
#diseñados para que vayan en 1 salto de tiempo
# Funcion de esquema temporal Euler
def Euler(U, t1, t2, F):
    dt = t2 - t1
    return U + dt * F(U, t1)

# Funcion de esquema temporal Crank-Nicolson
def Crank_Nicolson(U, t1, t2, F):
    def Residual_CN(X):
        return X - a - (t2 - t1) / 2 * F(X, t2)

    dt = t2 - t1
    a = U + (t2 - t1) / 2 * F(U, t1)   #nota si a=0 esto pasaría a ser un euler implicito
    return custom_newton(Residual_CN, U)

# Funcion de esquema temporal RK4
def RK4(U, t1, t2, F):
    dt = t2 - t1
    k1 = F(U, t1)
    k2 = F(U + dt * k1 / 2, t1 + (t2 - t1) / 2)
    k3 = F(U + dt * k2 / 2, t1 + (t2 - t1) / 2)
    k4 = F(U + dt * k3, t1 + (t2 - t1))

    return U + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6

# Funcion de esquema temporal Inverse Euler
def Inverse_Euler(U, t1, t2, F):
    dt = t2 - t1
    def cerosINV(X):
        return X - U - dt * F(X, t1)   #aqui me lie un poco con t1 y t2 en la funcion F

    return custom_newton(func=cerosINV, x0=U)
```

Juan Carlos García-Taheño Hijes

1º MUSE ampliación de matemáticas I

Módulo non_linear_systems.py

```python
# Funcion personalizada de Newton
#Con esta funcion se reolveran CN y Euler inverso

def custom_newton(func, x0, tol=1e-6, max_iter=100, delta=1e-6):
    def func_derivative(x):
        return (func(x + delta) - func(x - delta)) / (2 * delta)

    x = x0
    for _ in range(max_iter):
        delta_x = func(x) / func_derivative(x)
        x = x - delta_x
        if (abs(delta_x) < tol).any():
            return x
    return x

```

La estructura principal sigue siendo la misma, pero ahora permite una mayor compactación del código y una diferenciación de funciones según su uso.

Quizá una de las posibles mejoras es añadir otro módulo de funciones a integrar, como en este ejercicio solo se ha usado la de Kepler, no me he molestado en poner más o en el módulo de non_linear_systems.py se podría añadir más métodos y no solo el de newton, pero como conclusión esta estructura permite una mayor facilidad de entender que estas programando.