


Universidad Politécnica de Madrid

Ampliación de Matemáticas I

Informe Hito 2

Autora:
Inés Arauzo Andrés

Índice

1. Introducción **1**

2. Definición del problema **1**

3. Código **2**
3.1. Forcing Sides 2
3.2. Math Operators 2
3.3. Temporal Schemes 3
3.3.1. Euler 4
3.3.2. Crank Nicolson 4
3.3.3. Runge-Kutta 4 5
3.3.4. Euler inverso 7
3.4. Cauchy Problem 7
3.5. Main Milestone2 8

4. Resultados **9**
4.1. Euler 9
4.2. Runge Kutta 4 10
4.3. Crank-Nicolson 11
4.4. Euler Inverso 12

Índice de figuras

3.1. Región de estabilidad absoluta del esquema Euler 4
3.2. Región de estabilidad absoluta del esquema Crank Nicoson 5
3.3. Región de estabilidad absoluta del esquema RK4 6
3.4. Región de estabilidad absoluta del esquema Euler Inverso 7
4.1. Orbitas según el esquema Euler 9
4.2. Orbitas según el esquema Runge Kutta 4 10
4.3. Orbitas según el esquema Crank Nicolson 11
4.4. Orbitas según el esquema Euler Inverso 12

1. Introducción

A lo largo del presente informe se pretende comentar y comparar entre sí los resultados obtenidos al programar el Milestone 2. En este, se implementaron los esquemas de integración Euler, Runge-Kutta, Euler Inverso y Crank-Nicolson que se utilizaron para integrar el recorrido de una órbita kepleriana. De este modo, se procede a comentar el problema a resolver, el código utilizado y los resultados obtenidos.

2. Definición del problema

Como se comentaba, el problema que tratamos es la integración de una órbita kepleriana. Este queda definido con el problema de Cauchy que se expone a continuación

$$\frac{d^2\mathbf{r}}{dt^2} = -\frac{\mathbf{r}}{||\mathbf{r}||}^3$$

r(t=0) =(1,0)

dr/dt(t=0)=(0,1) (2.0.1)

Para resolver la EDO de segundo orden, lo convertimos a un sistema:

$$\frac{d\mathbf{U}}{dt} = \mathbf{F}(\mathbf{U}, t)\mathbf{U}(0) = \mathbf{U_0} \tag{2.0.2}$$

Donde t es la variable independiente, U es el vector de estado, y F es el término forzante, procedente de aplicar la transformación de Edo de segundo orden a sistema de EDO´s,

$$\mathbf{U} = \begin{bmatrix} x \\ y \\ u \\ v \end{bmatrix}, \quad \mathbf{F}(\mathbf{U}, \mathbf{t}) = \frac{\mathbf{dU}}{\mathbf{dt}} = \begin{bmatrix} u \\ v \\ \frac{-x}{(x^2+y^2)^{3/2}} \\ \frac{-y}{(x^2+y^2)^{3/2}} \end{bmatrix}, \tag{2.0.3}$$

3. Código

En el código se ha pretendido realizar una abstracción para seguir la jerarquía matemática lógica. De este modo se han desarrollado diferentes módulos , que separan las funciones según su significado físico-matemático.

3.1. Forcing Sides

En este módulo se irán implementando los problemas que sea necesario resolver, esto es, la física que se nos plantea. En este caso, la función Kepler tiene de argumentos $\mathbf{U}$ y t (aunque el t en este caso no es necesario) y devuelve su derivada $\mathbf{F(U, t)}$

```
from numpy import array
    def Kepler(U, t):

        x = U[0]
        y = U[1]
        dxdt = U[2]
        dydt = U[3]
        denom = ( x**2 + y**2 )**1.5
        return array([dxdt, dydt, -x/denom, -y/denom])
        \
```

Algoritmo 3.1: Forcing$_S$*ides.py*

3.2. Math Operators

En este módulo se ha desarrollado el método Newton-Raphson en formato vectorial para poder resolver con el sistemas implícitos (como Crank-Nicolson o Euler Inverso). Este método consiste en encontrar aproximaciones de los ceros o raíces de una función real, y con ello, una solución aproximada:

$$\mathbf{U_{i+1}} = \mathbf{U_i} - \frac{\mathbf{F(U_i)}}{\nabla \mathbf{F(U_i)}} \tag{3.2.1}$$

Y el código queda como:

```

    def Newton_Raphson(F, U0):

        nv = len(U0)
        DX = array(zeros(nv))
        U_1 = U0
```

```
7
8         tol = 1e-8              # Tolerance
9         i = 0                    # Iteration counter
10        imax = 10000             # Max number of iterations
11        err = 1
12
13        while err > tol and i <= imax:
14            J = Jacobian(F, U_1)
15
16            DX = matmul( inv (J), F(U_1))
17            U  = U_1 - DX
18            err = norm (U - U_1)
19            U_1  = U
20            i = i + 1
21
22        if i == imax:
23            print('You have reached the iteration limit. Problem is
    not converging')
24
25        return U
```

Algoritmo 3.2: Vectorial Newton-Raphson

Para ello, ha sido también necesario crear una función que genere el Jacobiano de un vector:

```
1     def Jacobian(F, U):
2
3         nv = len(U)
4         J = zeros((nv, nv))
5         dx = 1e-3
6
7         for i in range(nv):
8             DX = zeros(nv)
9             DX[i] = dx
10            J[:, i] = (F(U + DX) - F(U))/(dx)
11
12        return J
```

Algoritmo 3.3: Jacobian

Concretamente se ha elegido este esquema porque tiene un buen ratio de convergencia.

3.3. Temporal Schemes

En el módulo de temporal Schemes se han implementado los siguientes esquemas temporales:

3.3.1. Euler

El esquema de Euler es explícito y de primer orden, que se expresa,

$$\mathbf{U^{n+1}} : \mathbf{U^n} + \mathbf{dtF^n}, \tag{3.3.1}$$

Su región de estabilidad es un círculo de radio unidad centrado en $Re(\Delta t) = -1$ (ver 3.1por lo que las soluciones son inestables. De este modo sabemos que la solución es divergente. En el código queda

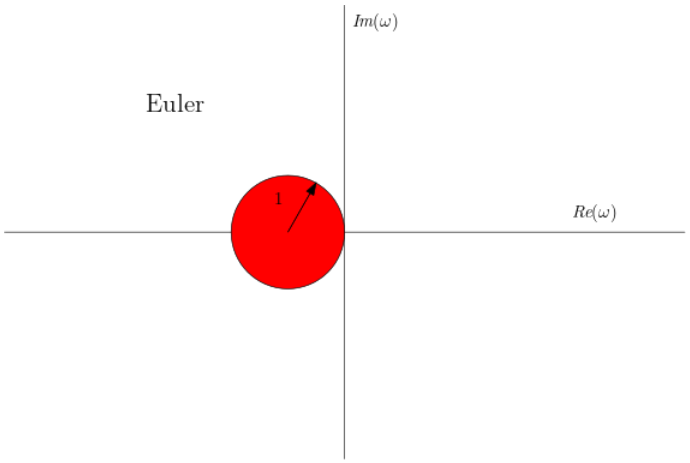


Figura 3.1: Región de estabilidad absoluta del esquema Euler

```
1    def Euler(U, F, t, dt):
2
3        return U + dt * F (U, t)
```

Algoritmo 3.4: Euler

3.3.2. Crank Nicolson

El Crank Nicolson es un esquema implícito monopaso de orden 2, en el cual la región de estabilidad comprende toda la parte positiva del plano imaginario, estando su frontera sobre el imaginario, por lo que las soluciones caen directamente sobre la frontera de la región de estabilidad ($\rho = 1$) (ver fig. 3.2, haciendo que no exista error de amplitud, solo de fase. Matemáticamente el esquema se expresa:

$$\mathbf{U^{n+1}} = \mathbf{U^n} + \frac{\mathbf{dt}}{\mathbf{2}}(\mathbf{F^{n+1}} + \mathbf{F^n}), \tag{3.3.2}$$

y en el código se llama al método de Newton-Raphson para resolver el sistema.

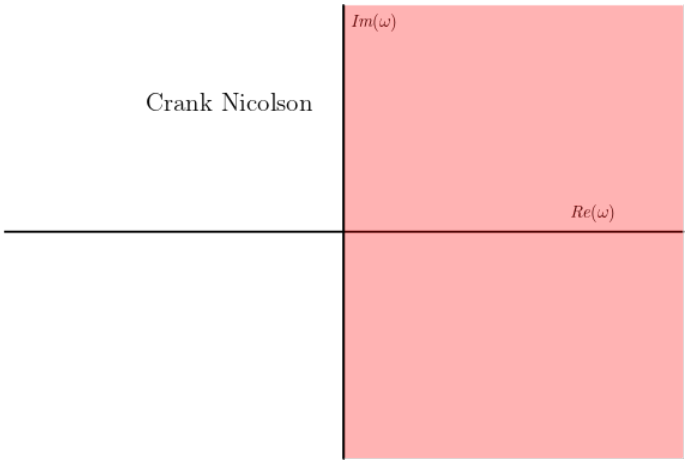


Figura 3.2: Región de estabilidad absoluta del esquema Crank Nicoson

```
        def CN( Un, F,  t, dt):

        def func(Un1):

            return Un1 - Un - dt/2 * ( F (Un1, t + dt) + F (Un, t)
    )

        U = Newton_Raphson(func, Un)

    return U.
```

Algoritmo 3.5: Crank-Nicolson

Para futuros códigos, para la inversa se tratará de desarrollar un código propio mediante factorización LU que se guardará en el módulo de Math Operators.

3.3.3. Runge-Kutta 4

Los métodos Runge-Kutta son métodos multietapa, y en este caso, el utilizado es de cuarto orden. Esto hace que la precisión obtenida sea mucho mejor, ya que la frontera de la región de estabilidad absoluta es tangente al eje imaginario, haciendo que las raíces del polinomio característico de estabilidad estén justo en la frontera (ver fig. 3.3, y por tanto el esquema sea estable y que no se disipe la energía de la órbita. Matemáticamente los esquemas Runge-Kutta se expresan como,

$$\mathbf{U^{n+1}} = \mathbf{U^n} + \mathbf{dt} \sum_{\mathbf{i=1}}^{\mathbf{s}} \mathbf{b_i k_i} \tag{3.3.3}$$

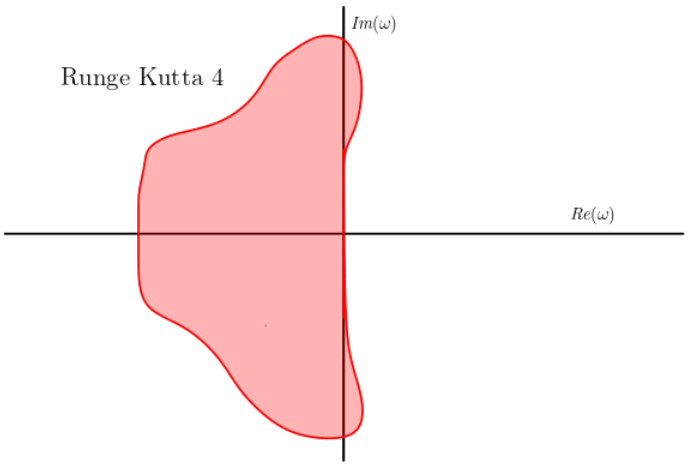


Figura 3.3: Región de estabilidad absoluta del esquema RK4

y en concreto el RK4,

$$\mathbf{k_1} = \mathbf{F}(\mathbf{U^n}, \mathbf{t_n})$$

$$\mathbf{k_2} = \mathbf{F}(\mathbf{U^n} + \mathbf{dtk_1/2}, \mathbf{t_n} + \mathbf{dt/2})$$

$$\mathbf{k_3} = \mathbf{F}(\mathbf{U^n} + \mathbf{dtk_2/2}, \mathbf{t_n} + \mathbf{dt/2})$$

$$\mathbf{k_4} = \mathbf{F}(\mathbf{U^n} + \mathbf{dtk_3}, \mathbf{t_n} + \mathbf{dt})$$

$$\mathbf{U}^{n+1} = \mathbf{U^n} + \mathbf{dt}(\mathbf{k_1} + \mathbf{2k_2} + \mathbf{2k_3} + \mathbf{k_4})/\mathbf{6}$$

$$\mathbf{U}^{n+1} = \mathbf{U^n} + \mathbf{dt} \sum_{\mathbf{i=1}}^{\mathbf{s}} \mathbf{b_i k_i} \quad (3.3.4)$$

Con todo ello, el código en python queda:

```
def RK4(U, F, t, dt):

    k1 = F(U, t)

    k2 = F(U + dt * k1/2,t + dt/2)

    k3 = F(U + dt * k2/2, t + dt/2)

    k4 = F(U + dt * k3, t + dt)

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

Algoritmo 3.6: Crank-Nicolson

3.3.4. Euler inverso

El esquema Euler inverso es implicito y de primer orden, que matemáticamente se expresa como,

$$U^{n+1} = U^n + \Delta t_n F^{n+1}, \tag{3.3.5}$$

en este caso, tenemos la región de estabilidad inversa al caso del Euler normal, esto es, todo el plano complejo menos un círculo de radio unidad centrado en el 1. debido a esto, las soluciones no caen en la frontera de la región y por tanto el esquema disipa energía (ver fig. 3.4, haciendo que la órbita decaiga.

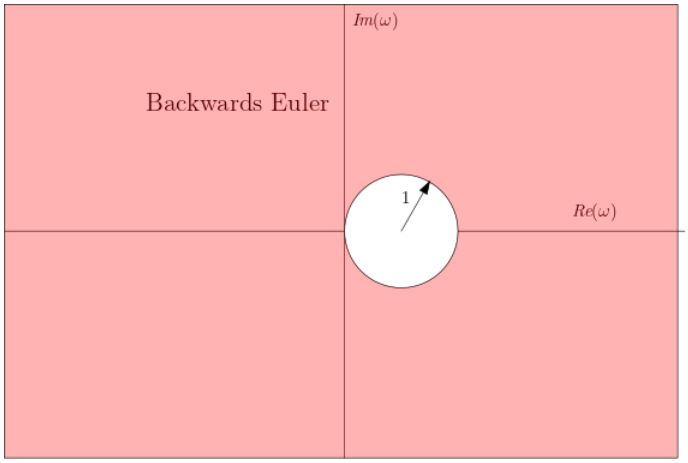


Figura 3.4: Región de estabilidad absoluta del esquema Euler Inverso

El código queda tal que:

```
def BW_Euler(Un, F, t, dt):

    def BW_Euler_eqn(Un1):

        return Un1 - Un - dt * F(Un, t)
    U = Newton_Raphson ( BW_Euler_eqn, Un )

    return U
```

Algoritmo 3.7: BW Euler

Como se puede ver, al tratarse de un esquema implícito ha sido necesario volver a llamar al código de Newton-Raphson para resolver.

3.4. Cauchy Problem

Como se comentaba en la introducción, lo que tratamos de resolver es un problema de Cauchy, cuyos inputs serán unas condiciones iniciales U_0, la física del problema

a resolver F, el esquema temporal a usar (Time Scheme) y el tiempo t,

```
from numpy import zeros
def Cauchy(F, Time_Scheme, U0, t):
    N=len(t)-1
    Nv=len(U0)
    U = zeros([N+1, Nv])
    U[0,:] = U0
    for n in range(N):
        U[n+1, :]= Time_Scheme(U[n, :], F, t[n], t[n+1]-t[n])
    return U
```

Algoritmo 3.8: Cauchy

3.5. Main Milestone2

Finalmente, el código ejecutable, llama a la función de Cauchy y plotea los resultados:

```
from numpy import array, linspace
import matplotlib.pyplot as  plt
from Cauchy_Problem import Cauchy
from Time_Schemes import Euler, CN, RK4, BW_Euler
from Forcing_Sides import Kepler



def Cauchy_wrapper(T, N, t_scheme):
    t = linspace(0, T, N)
    U0 = array ([1, 0, 0, 1])
    U = Cauchy(Kepler, t_scheme, U0, t)
    return U



t_schemes = [ Euler, RK4, CN, BW_Euler ]
t_scheme_name = [ "Euler", "RK4", "CN" , "BW_Euler" ]
t_scheme_fig = [ "Euler.png", "RK4.png", "CN.png", "BW_Euler.png" ]
i = 0
for t_scheme in t_schemes:

    U_dta = Cauchy_wrapper(10, 100, t_scheme)
    U_dtb = Cauchy_wrapper(10, 10000, t_scheme)
    U_dtc = Cauchy_wrapper(10, 50, t_scheme)
    plt.figure(figsize=(5, 5), layout='constrained')
    plt.plot(U_dta[:, 0], U_dta[:, 1], label='dt=0.1')
    plt.plot(U_dtb[:, 0], U_dtb[:, 1], label='dt=0.001')
    plt.plot(U_dtc[:, 0], U_dtc[:, 1], label='dt=0.2')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(t_scheme_name[i])  # Add a title to the axes.
    plt.legend()
```

```
34      plt.show()
35      plt.savefig(t_scheme_fig[i])
36      i=i+1
```

Algoritmo 3.9: Milestone2 wrapper.py

4. Resultados

Se ha llevado a cabo un análisis comparativo del efecto del paso del tiempo Δt sobre la eficacia y eficiencia de los esquemas temporales. Para ello, se han empleado tres pasos de tiempo diferentes, $\Delta t = 0{,}1, 0{,}2, 0{,}001$, que se presentan a continuación.

4.1. Euler

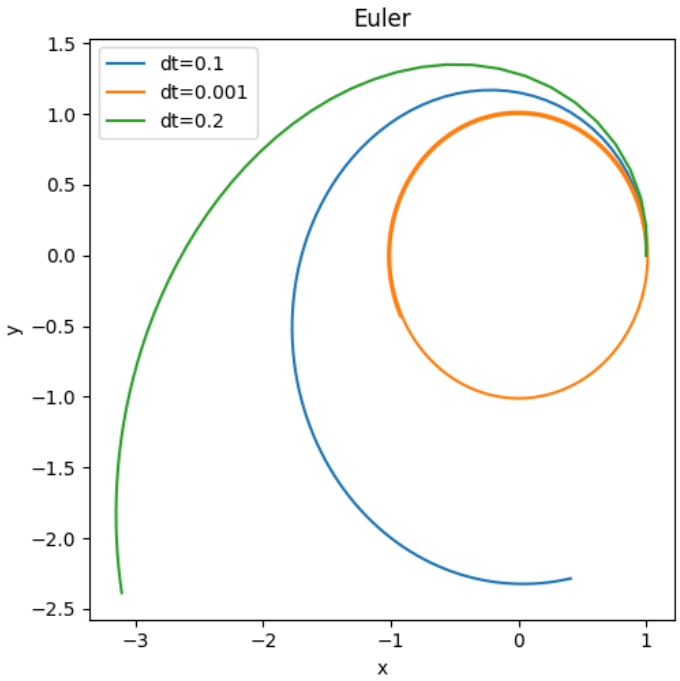


Figura 4.1: Orbitas según el esquema Euler

En la Fig.4.1 se aprecia como $\forall \Delta t$ El problema diverge. Para que este esquema funcione, se tendría que dar $\Delta t \longrightarrow 0$

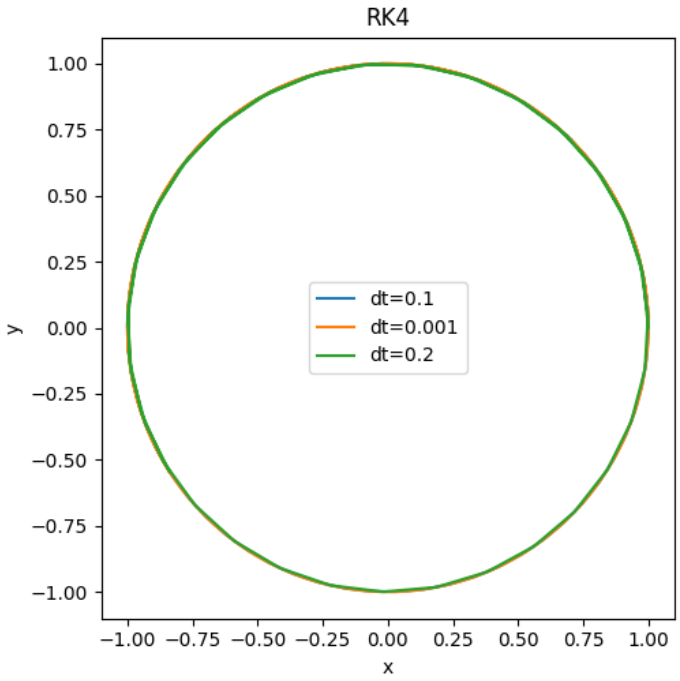
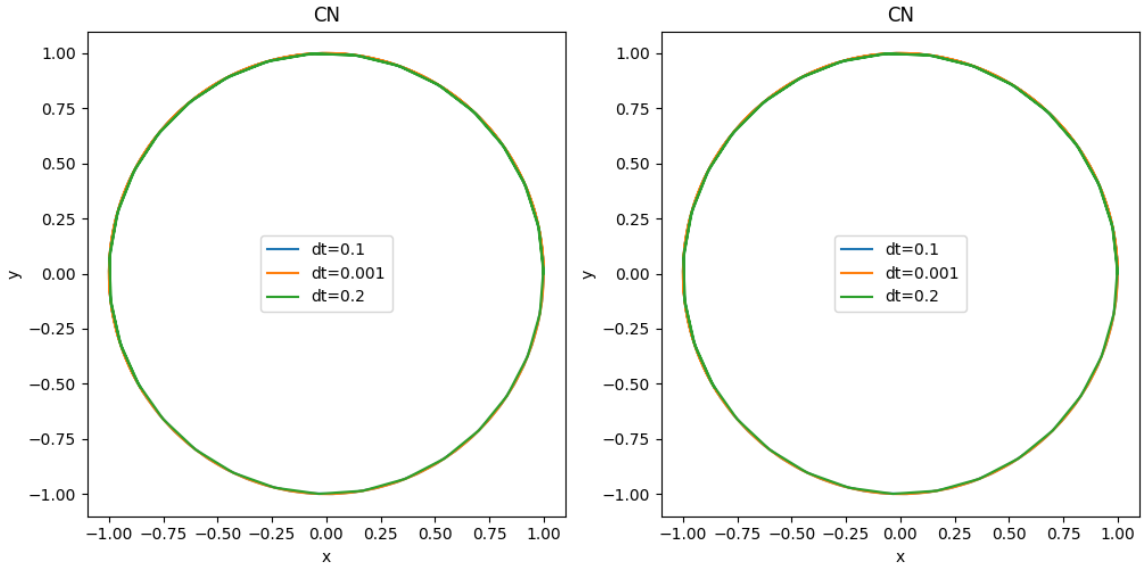


Figura 4.2: Orbitas según el esquema Runge Kutta 4

4.2. Runge Kutta 4

En la Fig.4.2 Se aprecia que para los valores de Δt dados, las órbitas obtenidas son idénticas. Esto se debe a que el esquema RK4 es un esquema de orden 4, mientras el de Euler es de orden 1 (de hecho el Euler es el RK1). Ello hace que el error cometido en este caso sea mucho menor.

4.3. Crank-Nicolson

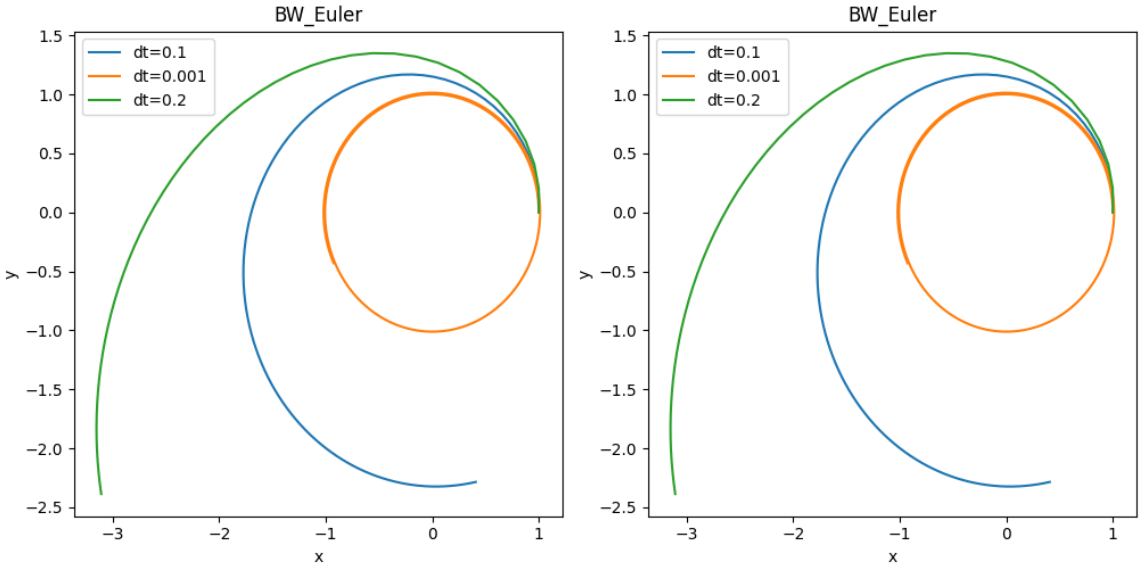


(a) Newton Raphson (b) SciPy

Figura 4.3: Orbitas según el esquema Crank Nicolson

En la Fig.4.3 se deduce lo mismo que en el caso anterior: este esquema es mucho más estable, ya que en este caso se trata de un esquema implícito de orden 2, y de nuevo el error cometido es mucho menor, de hecho, tal y como se comentó anteriormente, al estar la frontera en el eje imaginario, no existe error de amplitud. Además, tras compararlo con el resultado obtenido utilizando la funcion Fsolve de SciPy en vez de llamar al Newton, no se ha obtenido evidencia de ningun tipo de diferencia.

4.4. Euler Inverso



(a) Newton Raphson (b) SciPy

Figura 4.4: Orbitas según el esquema Euler Inverso

En la Fig.4.4 se aprecia como $\forall \Delta t$ el esquema euler inverso hace que la órbita decaiga, dado que disipa energía. Para que este esquema funcione, se tendría que dar $\Delta t \longrightarrow 0$, para que las soluciones estuviesen en el origen del plano complejo (porque son imaginarias puras) y así tocasen la frontera del esquema.

En este caso, la comparativa entre el Newton Raphson y el fsolve no da problemas para pasos de tiempo pequeños, pero para grandes Δt, se aprecia que la órbita calculada mediante NR decae más rápido.