


Universidad Politécnica de Madrid

Ampliación de Matemáticas I

Informe Hito 2

Autor:
Guillermo García del Río

Índice

Índice de figuras

1. Introducción

En este informe se presentan y comentan los resultados obtenidos en el hito 2 propuesto en la asignatura de Ampliación de Matemáticas I. En primer lugar se mostrará el enunciado que marca las pautas del proyecto, seguido de un análisis del problema a resolver, la exposición del código y los resultados obtenidos con él. Para terminar se comentarán y compararán dichos resultados.

2. Enunciado del hito

Los objetivos de este hito quedan establecidos por los siguientes puntos:

1 Escribir una función llamada Euler para integrar un paso. La función $F(U,t)$ del problema de Cauchy debe ser el input.

2 Escribir una función llamada Crank_Nicolson para integrar un paso.

3 Escribir una función llamada RK4 para integrar un paso.

4 Escribir una función llamada Inverse_Euler para integrar un paso.

5 Escribir una función para integrar el problema de Cauchy. El esquema temporal, la condición inicial y la función $F(U,t)$ del problema de Cauchy deben ser los inputs.

6 Escribir una función para expresar la fuerza del problema de Kepler. Poner énfasis en la manera en la que se escribe la función del problema de Cauchy: $F = [\dot{r}, -r/|r|^3]$ donde $r, \dot{r}\ \epsilon\ \mathbb{R}^2$

7 Integrar una órbita Kepler con los esquemas numéricos anteriores y explicar los resultados.

8 Incrementar y disminuir el paso de integración y explicar los resultados.

3. Problema a resolver

Lo primero que hay que hacer es tener claro el problema que se pretende resolver. En este caso se trata de integrar una órbita de Kepler conociendo las condiciones iniciales de la misma. Las ecuaciones que rigen este problema son las siguientes:

$$\ddot{\vec{r}} = -\frac{\vec{r}}{|\vec{r}|^3} \tag{3.0.1}$$

Con las condiciones iniciales de posición y velocidad respectivamente:

$$\begin{aligned} \vec{r}(0) &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \dot{\vec{r}}(0) &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{aligned} \tag{3.0.2}$$

De tal manera que en realidad a lo que nos estamos enfrentando es a un problema de Cauchy de la forma:

$$\begin{aligned} \frac{d\vec{U}}{dt} &= \vec{F}(U,t) \\ \vec{U}(0) &= \vec{U}_0 \end{aligned} \tag{3.0.3}$$

Donde U es el vector de estado formado por las componentes de la posición y la velocidad :

$$\vec{U} = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix} \tag{3.0.4}$$

Y a partir de su derivada se obtiene la función $F(U,t)$ que en este caso será:

$$\vec{F}(U,t) = \frac{d\vec{U}}{dt} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \ddot{x} \\ \ddot{y} \end{pmatrix} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dfrac{-x}{(x^2+y^2)^{3/2}} \\ \dfrac{-y}{(x^2+y^2)^{3/2}} \end{pmatrix} \tag{3.0.5}$$

Como se puede observar, tanto el vector de estado como la $\vec{F}(U)$ variará dependiendo del problema que estemos tratando de resolver. En este caso la $\vec{F}(U)$ no depende del tiempo, por lo que es un sistema autónomo. La manera de resolver este problema dependerá del esquema numérico que se esté utilizando.

4. Exposición del código

De cara a poder entender lo que se está haciendo se va a proceder a exponer el código desarrollado para solucionar el problema propuesto al mismo tiempo que se introducirá la matemática de cada esquema numérico que se ha implementado.

Por otro lado, se ha llevado a cabo una simplificación y abstracción del código, separando cada tipo de función en un módulo distinto de Python. De esta manera, el código queda dividido en los siguientes módulos:

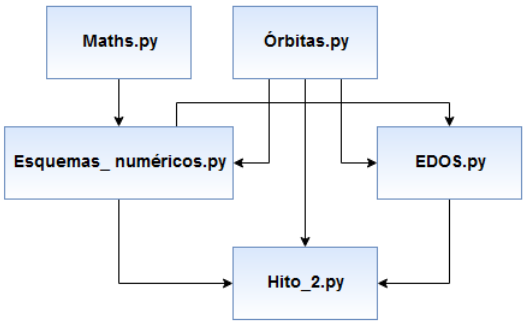


Figura 4.1: Módulos utilizados para el hito 2.

4.1. Módulo de Órbitas

En este módulo se recogen los distintos problemas de órbitas que puede interesar integrar. El código de este módulo es el siguiente:

```
from numpy import array

## ÓRBITAS ##

 #  Inputs:
 #          U : Vector estado en tn
 #          t : tn
 #
 #  Return:
 #
 #          F(U,t) : Función derivada del vector de estado dU/dt = F(U,t)

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])
```

Algoritmo 4.1: Módulo de órbitas

Como se puede observar, de momento solo existe la posibilidad de elegir la órbita de Kepler. Esta función tendrá como imput el vector de estado $\vec{U}$ y el instante de tiempo en el que se encuentre. Esta función devolverá la función derivada del vector de estado $F(U,t)$.

4.2. Módulo de matemáticas

Ya que en este hito era de especial interés desarrollar el código de Newton-Raphson para resolver el sistema no lineal en los casos de los esquemas numéricos implícitos, se ha creado un módulo en el que se han incluido este método y otras operaciones matemáticas necesarias. Para una mejor comprensión se va a introducir brevemente este método.

El objetivo del método de Newton es construir una secuencia que converja a la solución x mediante la linealización f alrededor de un punto x_i que será una aproximación inicial. Con ello, la secuencia debe proveer un punto x_{i+1} que esté más cerca de la solución que x_i. Esta secuencia tiene la siguiente forma:

$$x_{i+1} - x_i = -(\nabla f(x_i))^{-1} \cdot f(x_i) \tag{4.2.1}$$

Donde $(\nabla f(x_i))^{-1}$ es la inversa de la matriz jacobiana. El código desarrollado para poder aplicar este método es el siguiente:

```
from operator import matmul
from numpy import array, zeros
from numpy.linalg import inv, norm

## OPERACIONES MATEMÁTICAS ##

 #  Inputs:
 #         U : Vector estado en tn
 #         dt: Paso de tiempo
 #         F : Sistema del que se quiere obtener la matriz Jacobiana o que se
    quiere solucionar con Newton-Raphson
 #
 #  Return:
 #
 #         J : Matriz jacobiana
 #         newton : solución del sistema no linear por Newton-Raphson

def jacobiano(F,U):

    dim = len(U)
    Dx = 1e-3
    jacobian = array(zeros((dim,dim)))

    for i in range(dim):

        xj = array(zeros(dim))
        xj[i] = Dx
        jacobian[:,i] = (F(U + xj) - F(U - xj))/(2 * Dx)

    return jacobian

def newton(F, U0):

    dim = len(U0)
    Dx = array(zeros(dim))
    b  = array(zeros(dim))
    U1 = U0

    eps = 1
    iteration = 0
    itmax = 10000

    while eps > 1e-8 and iteration <= itmax:

        J = jacobiano(F,U1)
        b = F(U1)
        Dx = matmul(inv(J),b)
        U  = U1 - Dx
        eps = norm(U - U1)
        U1  = U
        iteration = iteration + 1

        if iteration == itmax:
            print('Máximo número de iteraciones alcanzado')
```

```
54
55      return U
```

Algoritmo 4.2: Módulo de matemáticas

En el módulo, la aproximación inicial x_i se introduce como el $U0$ y las ecuaciones del sistema como F.

4.3. Módulo de esquemas numéricos

De cara a tener una mayor organización del código se ha creado un módulo que incluya los esquemas numéricos empleados para la integración del problema. A pesar de que el código se vaya a mostrar fragmentado en cada apartado de esta sección, se ofrece una visión general del módulo al completo:

```
1  from Maths import newton
2
3  ## ESQUEMAS NUMÉRICOS ##
4
5   #
6   #
7   #   Inputs:
8   #           U : Vector estado en tn
9   #           dt: Paso de tiempo
10  #           t : tn
11  #           F(U,t) : Función derivada del vector de estado dU/dt = F(U,t)
12  #
13  #   Return:
14  #
15  #           U Vector de estado en tn + dt
16
17
18 def Euler(U,dt,t,F):
19
20      return U + dt * F(U,t)
21
22 def RK4(U,dt,t,F):
23
24      k1 = F(U,t)
25      k2 = F(U + k1 * dt/2, t + dt/2)
26      k3 = F(U + k2 * dt/2, t + dt/2)
27      k4 = F(U + k3 * dt, t + dt)
28
29      return U + (dt/6) * (k1 + 2*k2 + 2*k3 + k4)
30
31 def Crank_Nicolson(U, dt, t, F):
32
33      def Ecuación_Crank(Un1):
34          return Un1 - U - dt/2 * (F(U,t) + F(Un1,t))
35
36      return newton(Ecuación_Crank,U)
37
38 def Euler_inverso(U, dt, t, F):
39
40      def Ec_Eulerinverso(Un1):
41          return Un1 -U -dt*F(Un1,t)
42
43      return newton(Ec_Eulerinverso,U)
```

Algoritmo 4.3: Módulo de esquemas numéricos

4.3.1. Método de Euler

El método de Euler es el esquema numérico más sencillo, pues es de primer orden y considera que $F(u,t)$ es constante al evaluar la integral entre t_n y t_{n+1}:

$$\vec{U}^{n+1} = \vec{U}^n + (t_{n+1} - t_n) \cdot \vec{F}^n) \tag{4.3.1}$$

El código implementado para llevar a cabo esta integración es el siguiente:

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

    return U + dt * F(U,t)
```

Algoritmo 4.4: Función para Euler en el módulo de esquemas numéricos

4.3.2. Crank-Nicolson

El esquema numérico de Crank-Nicolson se caracteriza por ser un esquema implícito de un paso y de segundo orden (lo cual ya es una mejora respecto al primer orden de Euler). Se puede expresar de manera general como:

$$U^{n+1} = U^n + \frac{\Delta t_n}{2}(F^{n+1} + F^n) \tag{4.3.2}$$

El código implementado para llevar a cabo este esquema numérico se muestra a continuación:

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

    def Ecuación_Crank(Un1):
        return Un1 - U - dt/2 * (F(U,t) + F(Un1,t))

    return newton(Ecuación_Crank,U)
```

Algoritmo 4.5: Función para Crank Nicolson en el módulo de esquemas numéricos

Se puede observar que el método elegido para resolver el sistema implícito es el de Newton, explicado y expuesto anteriormente.

4.3.3. Método de Runge-Kutta de cuarto orden

Con el método de Runge-Kutta de cuarto orden ya se da un salto en la precisión, ya que pasamos a un equema explícito de cuatro etapas y de cuarto orden. Este esquema puede describirse de manera general de la siguiente manera:

$$\begin{aligned} k_1 &= F(U^n; t_n) \\ k_2 &= F(U^n + \Delta t_n k_1/2; t_n + \Delta t_n/2) \\ k_3 &= F(U^n + \Delta t_n k_2/2; t_n + \Delta t_n/2) \\ k_4 &= F(U^n + \Delta t_n k_3; t_n + \Delta t_n) \\ U^{n+1} &= U^n + \frac{\Delta t_n}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{4.3.3}$$

El código para este esquema numérico se muestra es el siguiente:

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

    k1 = F(U,t)
    k2 = F(U + k1 * dt/2, t + dt/2)
    k3 = F(U + k2 * dt/2, t + dt/2)
    k4 = F(U + k3 * dt, t + dt)

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

Algoritmo 4.6: Función para RK4 en el módulo de esquemas numéricos

4.3.4. Método de Euler inverso

El esquema numérico de Euler inverso se obtiene haciendo que la ecuación diferencial se satisfaga en el instante t_{n+1}. A partir de la construcción del polinomio interpolante de primer grado, derivando con respecto del tiempo y particularizando para el instante se obtiene el esquema numérico de Euler inverso, el cual es de primer orden e implícito.

$$U^{n+1} = U^n + \Delta t_n F^{n+1} \tag{4.3.4}$$

El código desarrollado para este esquema numérico se muestra a continuación:

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

    def Ec_Eulerinverso(Un1):
        return Un1 -U -dt*F(Un1,t)

    return newton(Ec_Eulerinverso,U)
```

Algoritmo 4.7: Función para Euler inverso en el módulo de esquemas numéricos

De la misma manera que ocurría en el caso de Crank Nicolson, se resolverá el sistema implícito mediante el método de Newton.

4.4. Módulo de ecuaciones diferenciales ordinarias

El siguiente paso de abstracción se basa en desarrollar un código que defina una función para el sistema de Cauchy que tenga como inputs el problema a resolver, el tiempo, las condiciones iniciales y el esquema numérico que se quiere utilizar para integrar el problema. Este código se ha incluido en un módulo en el que se irán añadiendo otras ecuaciones diferenciales ordinarias.

```
from numpy import zeros, float64

## Ecuaciones Diferenciales Ordinarias ##

 #   Inputs:
 #
 #         F(U,t) : Función derivada del vector de estado dU/dt = F(U,t)
 #         t : Vector de tiempo
 #         U : Vector estado en tn
 #         Esquema: Esquema temporal empleado para la integración
 #
 #   Return:
 #
```

```
#          U Vector de estado en tn + dt

def Cauchy_Problem(F,t,U0,Esquema):

    n, nv = len(t)-1,len(U0)

    U = zeros((n+1,nv), dtype=float64)

    U[0,:] = U0

    for i in range(n):

        U[i+1,:] = Esquema(U[i,:],t[i+1] - t[i],t[i],F)

    return U
```

Algoritmo 4.8: Módulo de EDOs

4.5. Módulo principal del Hito 2

Por último, se ha creado un módulo específico que cumpla los enunciados de este hito en concreto. Esta tarea se ha visto simplificada gracias a la creación de distintos módulos, ya que tan solo hará falta llamar a las funciones deseadas de los otros módulos. El código de este programa principal es el siguiente:

```

## HITO 2 ##

from numpy import array, zeros, linspace
from Esquemas_numéricos import Euler,RK4, Crank_Nicolson, Euler_inverso
from Órbitas import Kepler
from EDOS import Cauchy_Problem
import matplotlib.pyplot as plt

## Variables temporales ##

T = 20                          # Duración de la simulación en segundos
dt = 0.001                      # Paso de integración en segundos
n = int(T/dt)                   # Numero de pasos
t  = linspace(0,T,n)            # Vector de instantes separados dt

## Condiciones iniciales ##

U0 = array([1,0,0,1])
U = U0

methods = [Euler,RK4, Crank_Nicolson, Euler_inverso]
lista = ['Euler','RK4','Crank Nicolson','Euler inverso']

for j in range (4):

    U = Cauchy_Problem(Kepler,t,U0,methods[j])
    print(U[len(t)-1,:])

    plt.title(f'Kepler integrado con {lista[j]} con dt = {dt} s y T = {T} s')
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.plot(U[:,0],U[:,1])

    plt.savefig('Plots/'+lista[j]+ ' ' + str(dt)+'.png')

    plt.show()
```

Algoritmo 4.9: Archivo principal del Hito 2

5. Resultados

Con el objetivo de llevar a cabo un buen análisis de los resultados obtenidos, y de forma parecida al hito anterior, se realizarán varias simulaciones variando el valor del paso de integración Δt y para un tiempo de 20 segundos.

Por otro lado, en el caso de los dos esquemas implícitos, también se ofrecerá una comparación entre los resultados obtenidos resolviendo el sistema con el método de Newton desarrollado y con la función de *Scipy*: *fsolve*.

5.1. Variación del paso de integración

Los valores elegidos para este hito son:

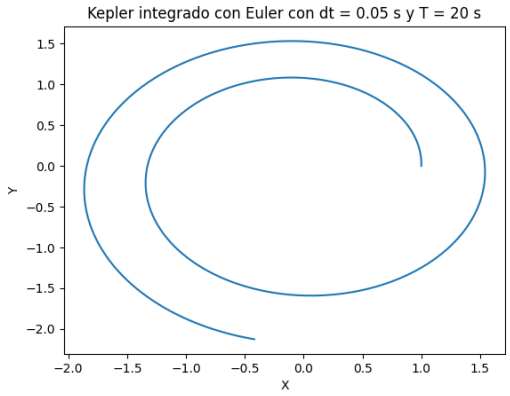
Tabla 5.1: Valores del paso de tiempo empleados

$\Delta t(s)$			
0.05	0.01	0.005	0.001

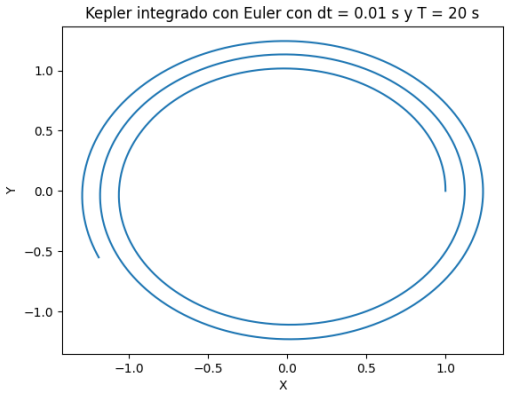
A excepción del esquema numérico de Euler inverso, los resultados que se van a analizar serán muy similares a los obtenidos en el hito 1.

5.1.1. Euler

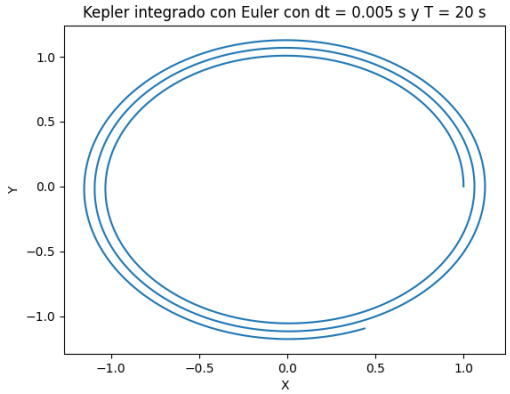
Los resultados obtenidos por el esquema numérico de Euler se presentan en la figura 5.1.



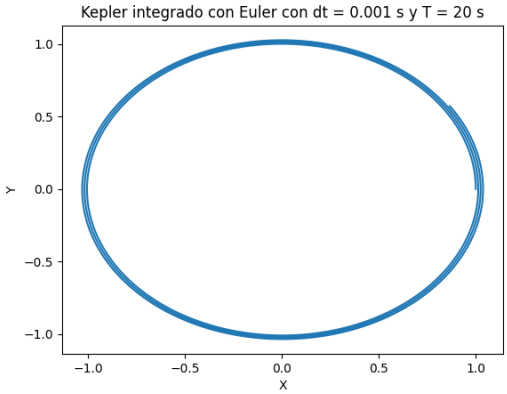
(a) Resultados obtenidos para un $dt = 0{,}05s$



(b) Resultados obtenidos para un $dt = 0{,}01s$



(c) Resultados obtenidos para un $dt = 0{,}005s$



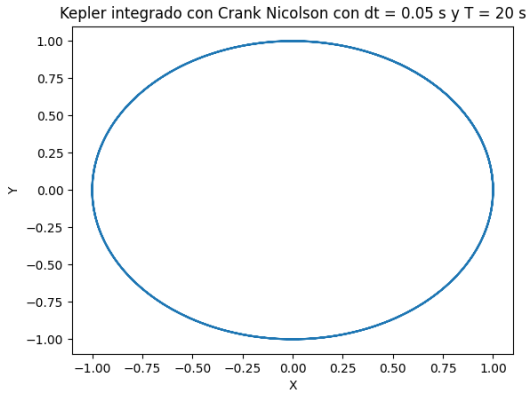
(d) Resultados obtenidos para un $dt = 0{,}001s$

Figura 5.1: Resultados obtenidos con el esquema numérico de Euler

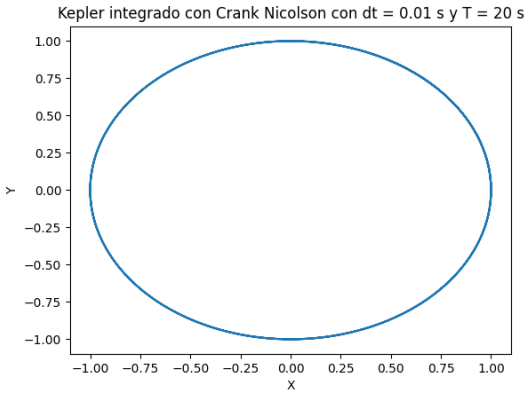
Como se puede observar en la figura 5.1, si bien es cierto que cuanto menor es el paso de integración mejor es la solución, siempre se podrá apreciar una divergencia con respecto a la solución analítica. Esta divergencia es debida a que el radio espectral del esquema numérico de Euler es mayor que uno, por lo que el error se va acumulando.

5.1.2. Crank Nicolson

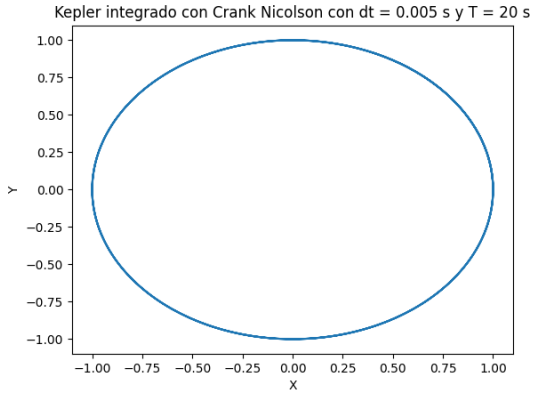
Los resultados obtenidos por el esquema numérico de Crank-Nicolson se presentan en la figura 5.2.



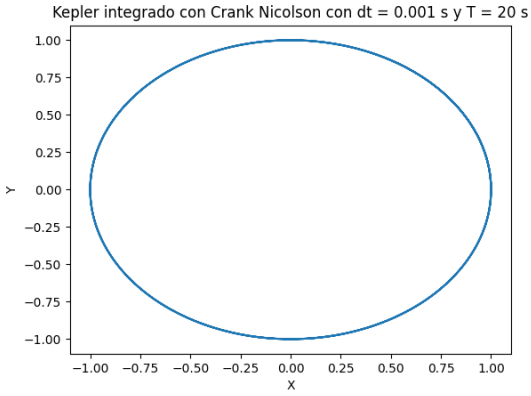
(a) Resultados obtenidos para un $dt = 0{,}05s$



(b) Resultados obtenidos para un $dt = 0{,}01s$



(c) Resultados obtenidos para un $dt = 0{,}005s$



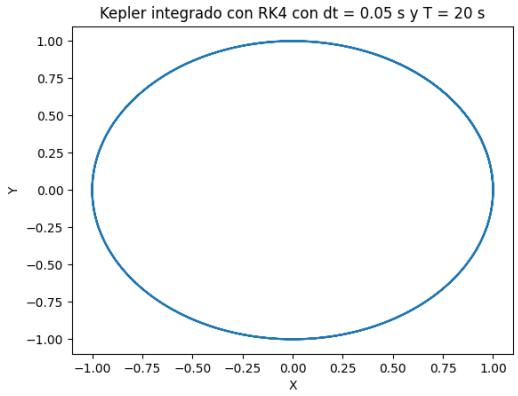
(d) Resultados obtenidos para un $dt = 0{,}001s$

Figura 5.2: Resultados obtenidos con el esquema numérico de Crank-Nicolson

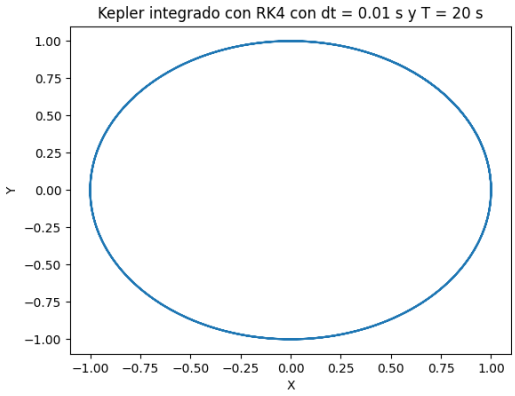
Se puede apreciar que el resultado obtenido con este esquema numérico es bastante mejor que el obtenido con Euler, esto se debe a que es de segundo orden. Por otro lado, la variación del valor de paso de integración no tiene ningún efecto en la solución más allá de que hay más puntos definidos.

5.1.3. Runge Kutta de cuarto orden

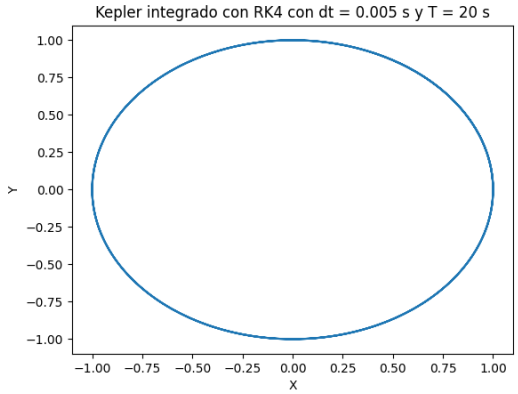
Los resultados obtenidos por el esquema numérico de Runge-Kutta de cuarto orden se presentan en la figura 5.3.



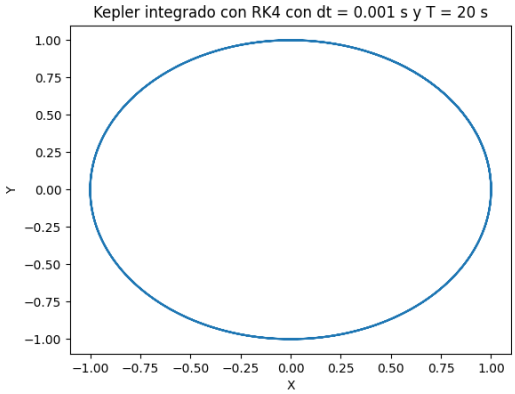
(a) Resultados obtenidos para un $dt = 0{,}05s$



(b) Resultados obtenidos para un $dt = 0{,}01s$



(c) Resultados obtenidos para un $dt = 0{,}005s$



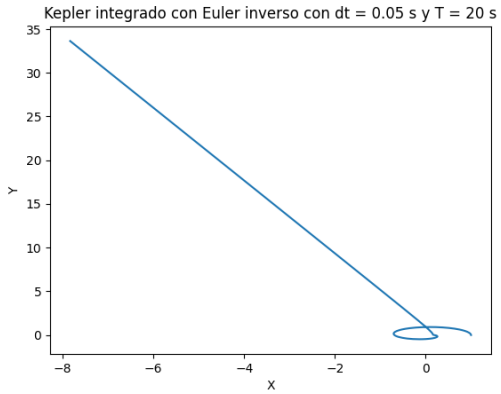
(d) Resultados obtenidos para un $dt = 0{,}001s$

Figura 5.3: Resultados obtenidos con el esquema numérico de Runge-Kutta de cuarto orden

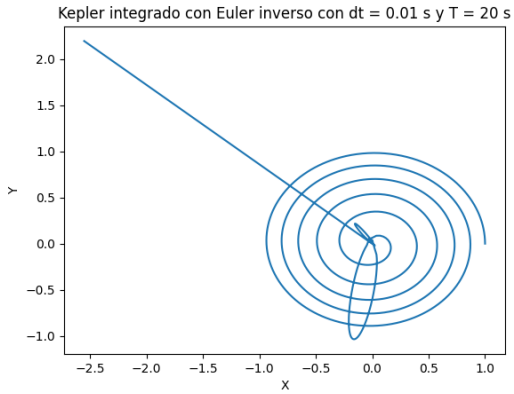
De la misma manera que con Crank-Nicolson, se puede observar que este esquema numérico es bastante mejor que el obtenido con Euler, esto se debe a que es de cuatro etapas y de cuarto orden y por lo tanto mayor precisión. Además, la variación del valor de paso de integración no tiene ningún efecto en la solución más allá de que hay más puntos definidos.

5.1.4. Euler inverso

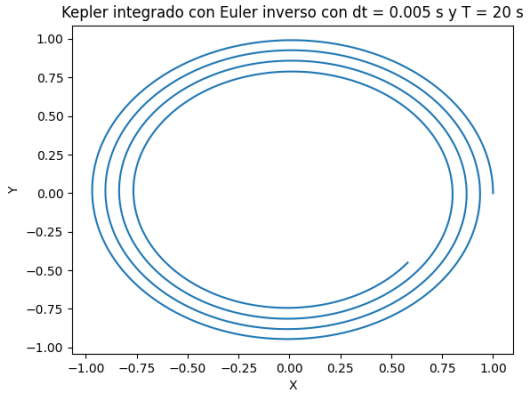
Los resultados obtenidos por el esquema numérico de Euler inverso se presentan en la figura 5.6.



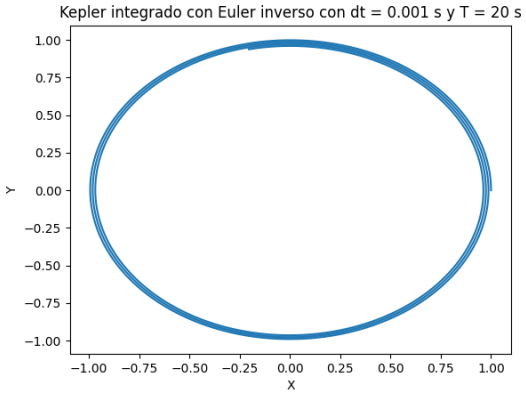
(a) Resultados obtenidos para un $dt = 0{,}05s$



(b) Resultados obtenidos para un $dt = 0{,}01s$



(c) Resultados obtenidos para un $dt = 0{,}005s$



(d) Resultados obtenidos para un $dt = 0{,}001s$

Figura 5.4: Resultados obtenidos con el esquema numérico de Euler inverso

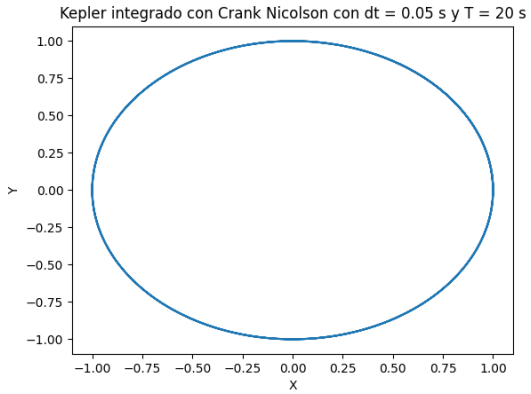
Sobre estos resultados se pueden comentar dos cosas: la primera es que, debido a las características del esquema numérico, al intentar integrar con pasos de integración grandes los resultados obtenidos son erráticos (más adelante se analizará la influencia del método de Newton en estos resultados); por otro lado, y de la misma manera que pasaba con Euler, existirá una divergencia directamente proporcional al paso de integración.

5.2. Diferencias para esquemas implícitos

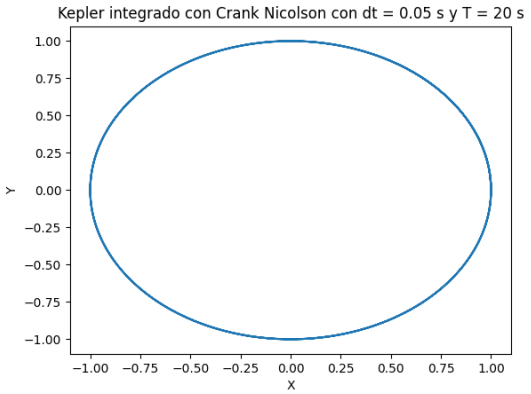
Por último se quieren comparar de forma general con las gráficas obtenidas los distintos tipos de resultados dependiendo de si se utiliza la función de Scipy *fsolve* o el código de Newton-Raphson desarrollado para este hito. Las parámetros temporales elegidos serán un tiempo de simulación de 20 segundos y los Δt extremos, es decir, $0,05s$ y $0,001s$.

5.2.1. Crank Nicolson

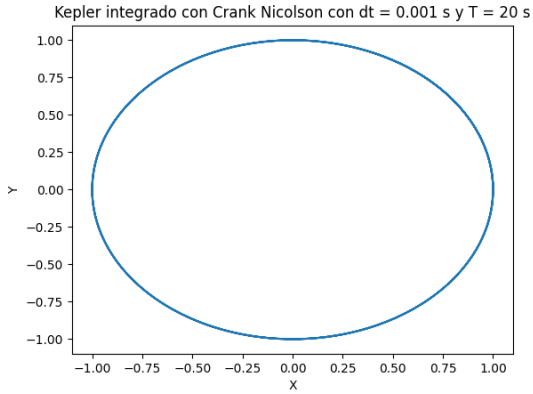
En primer lugar se compararán los resultados obtenidos por Crank Nicolson.



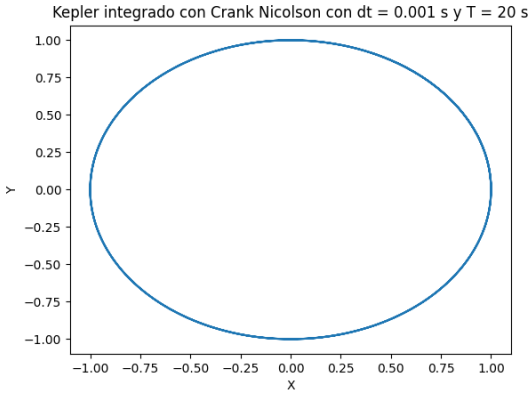
(a) Resultados con Newton para un $dt = 0{,}05s$



(b) Resultados con fsolve para un $dt = 0{,}05s$



(c) Resultados con Newton para un $dt = 0{,}001s$



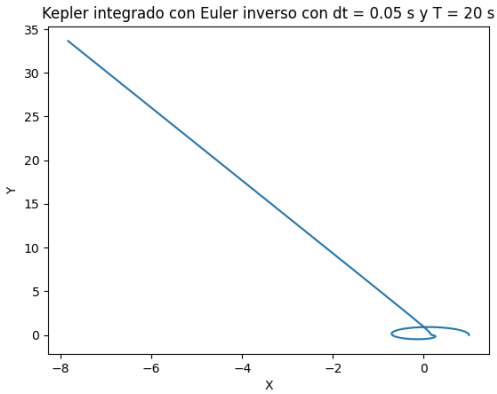
(d) Resultados con fsolve para un $dt = 0{,}001s$

Figura 5.5: Comparación de resultados para Crank-Nicolson

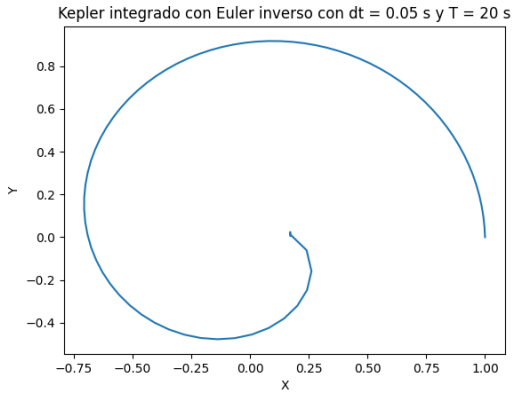
Como se puede comprobar en 5.5, el método de Crank Nicolson apenas experimenta diferencias dependiendo de cómo se resuelva el sistema implícito.

5.2.2. Euler inverso

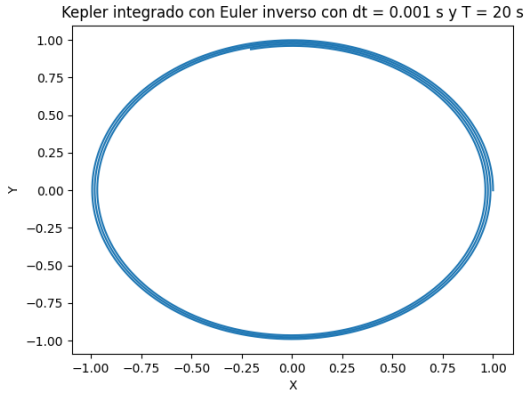
Por último, en el caso de Euler inverso:



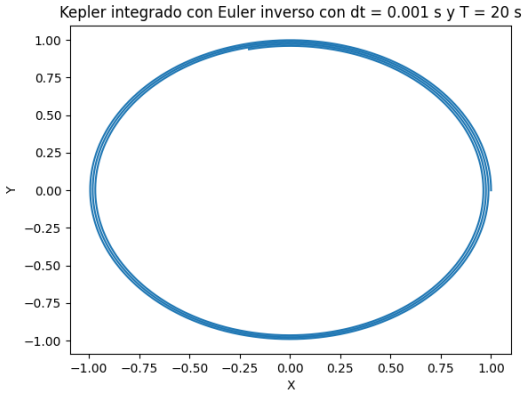
(a) Resultados con Newton para un $dt = 0{,}05s$



(b) Resultados con fsolve para un $dt = 0{,}05s$



(c) Resultados con Newton para un $dt = 0{,}001s$



(d) Resultados con fsolve para un $dt = 0{,}001s$

Figura 5.6: Comparación de resultados para Euler inverso

A diferencia del caso de Crank Nicolson, en este caso sí que varían los resultados dependiendo del método empleado para resolver el sistema implícito para pasos de tiempo grandes, mientras que para pasos más pequeños los resultados son muy similares. Esta mala respuesta al método de Newton-Raphson está ligada al bajo orden del esquema y a la fuerte dependencia del orden del paso de tiempo.