


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Milestone 3

Ampliación de Matemáticas I

21 de octubre de 2022

Autor:

Sergio de Ávila Cabral

Índice

1. Introducción

La programación del *Milestone 3* se ha realizado siguiendo la metodología adoptada en el anterior *Milestone*, la metodología *Top-Down*. Se han empleado los siguientes módulos existentes:

- 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*.

Y se ha definido los siguientes nuevos módulos:

- Milestone_3.py: Actúa como *main* y se encuentra en la 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 las llamadas a las funciones que determinan el error, la convergencia y el orden de los esquemas empleados.
- Richarson.py: En el se encuentran las dos nuevas funciones, objetivo principal de este *Milestone*: determinación del error mediante *Richarson* y la determinación del orden de los esquemas mediante el ratio de convergencia. Se localiza en *Sources/Error*.

2. Descripción de módulos nuevos o modificados

En este apartado se realizará una descripción de los módulos que han sido programados a raíz de este *Milestone* o hayan sufrido algún tipo de modificación.

2.1. Milestone_3.py

Como se ha descrito en la Sección 1, actual como *main*. En el se pueden apreciar diferentes secciones a lo largo del mismo:

Las librerías empleadas, se encuentran las nuevas librerías *Richarson*, la cual será descrita más adelante, y *alibre_progress*, la cual permite implementar una barra de progreso para

conocer el estado actual de los cálculos y el tiempo que conllevan.

```
from numpy import zeros, linspace, shape
from ODES.Cauchy_Problem import C_P
from Problems.Physics import Kepler
from ODES.Temporal_Schemes import Euler, RK4, Crank_Nicolson,
    Inverse_Euler
from Error.Richarson import Richarson, Convergence_rate
from alive_progress import alive_bar #to see the progress of the
    computations
import Graf.Plots as plt


```

Listing 2.1: Librerías

Tras las librerías empleadas se encuentra la inicialización de las condiciones iniciales. A diferencia de los anteriores *Milestone*, se define el tiempo T y los Δt deseados, que definirán los N que se emplearán.

```
#Initial Conditions
r0 = [1,0]
v0 = [0,1]
U0 = r0 + v0
T = 100
dt = [0.1, 0.01, 0.001]

```

Listing 2.2: Condiciones iniciales

A continuación, se mantiene el *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

```

Listing 2.3: selección guardado

Tras ello se definen las listas que contienen los diferentes esquemas numéricos.

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

```

Listing 2.4: Esquemas temporales

Se inicializan los diccionarios que se emplearán para el almacenamiento de los datos junto con la lista de los N que se emplearán.

```
#Initiation of Dictionaries
Er_dic = { }
t_dic = { }
N = zeros((len(dt)))

```

Listing 2.5: Inicialización de diccionarios

Se calculan las iteraciones necesarias junto con los tiempos que se emplearán en los cálculos. Como se puede observar, se ha realizado un bucle donde se calculan las iteraciones $\left(N = \frac{T}{\Delta t}\right)$ y se continua con un definiendo los tiempos con un equiespaciado.

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

```
#Iterations and times
for i in range(len(dt)):
        N[i] = int(T/dt[i])
        t_dic[str(i)] = linspace(0,T, int(N[i]))

```

Listing 2.6: definición tiempos e iteraciones

Por último, para la realización de los diferentes cálculos encomendados. En primer lugar se observa la asignación de la librería *alive_bar* como *bar*, la cual, como ya se ha mencionado, permite conocer el progreso de los cálculos. Para el cálculo del error, interesa cuando se realiza el error para varios esquemas o para varios tiempos, si se implementa.

Tras el cálculo del error se encuentra el del ratio de convergencia. Adelantando a lo que acontecerá en la descripción de la función *Convergence_rate*, se observa que devuelve tres variables: dos logaritmos y *regress*. Esta última es un objeto que contiene toda la información de la regresión lineal (pendiente, interceptación en el eje y, el coeficiente de determinación, etc).

En ambos bucles se puede observar el empleo de funciones *Plot*, que serán comentadas en la Subsección 2.4. Por último, al igual que en el *Milestone 2*, se definen nuevos *Keys* de forma dinámica. Como se puede apreciar, las *Keys* de ambos diccionarios están sincronizados, simplificando la programación de los *plots*.

```
with alive_bar(len(T_S)) as bar:
        for j in range(len(T_S)):
                for x in t_dic:
                        Er_dic[x] = Richarson(U0, t_dic[x], Kepler, ...
                        T_S[j], C_P, Order[j])
                        plt.Plot_Er(Er_dic[x], t_dic[x], T, ...
                        dt[int(x)], T_S_plot[j], Save)
                plt.Plot_Er_compare(Er_dic, t_dic, T, dt, ...
                T_S_plot[j], Save)
                print(T_S_plot[j] + " calculado \n")
                bar()

for j in range(len(T_S)):
```

```
    for x in t_dic:
        log_Er, log_N, regress = Convergence_rate(U0, t_dic[x], ...
        Kepler, T_S[j], C_P)
        print( '\n' +T_S_plot[j] + ' Calculado \n' )
        plt.Plot_Conv_Rat(log_N, log_Er, T, dt[int(x)], ...
        regress, T_S_plot[j], Save)

```

Listing 2.7: cálculo de error y ratio de convergencia

2.2. Richarson.py

Como se ha descrito en la Sección 1, en este nuevo módulo se integran el cálculo del error y el ratio de convergencia, del cual se obtendrá el orden del esquema empleado. Para este cometido se emplean las librerías que se pueden ver a continuación. Entre ellas, se aprecia la ya mencionada librería *alivre_progress* y el empleo de *linregress* de *Scipy*.

```
from numpy import array, linspace, zeros, size, shape, log10
from numpy.linalg import norm
from alive_progress import alive_bar #to see the progress of the
    computations
from scipy.stats import linregress

```

Listing 2.8: librerías para Richarson.py

Función Richarson

Se sigue de la función *Richarson*, la cual calcula el error cometido por un método numérico sin conocer la solución exacta. En estas líneas se puede observar una primera definición del tiempo, obtenida del libro *How to learn Applied Mathematics through modern FORTRAN*. Tras este, se comprobó que, tal como se comentó en clase, es más sencillo duplicar el número de equiespaciados con un *linspace* donde el tiempo se puede obtener de la última componente de t_1. Tras ello se calculan U_1 y U_2; y se procede al cálculo del error.

```
def Richarson(U0, t1, F, T_S, problem, order):

    N = size(t1) - 1 #Because of the divisons of the time's linspace (
    Python begin at 0)

    ## Definition of t2 according to the book
    # for i in range(N): # 0 to N-1
    #     t2[2*i] = t1[i]
    #     t2[2*i + 1] = ( t1[i] + t1[i+1] ) / 2

    # t2[2*N] = t1[N]

    t2 = linspace(0, t1[-1],2*size(t1))
    Er = zeros( (len(U0), N+1) )
```

```
14
15
16
17      U1 = problem(F, t1, U0, T_S)
18      U2 = problem(F, t2, U0, T_S)
19
20      for i in range(N):
21          Er[:, i] = ( U2[:, 2*i]- U1[:,i] ) / ( 1 - 1 / 2**order )
22
23      return Er
24
```

Listing 2.9: Función Richarson

Convergence_rate

Para esta función se hace necesario la definición de una nueva variable de iteración $Nt2$. Esto se debe a la necesidad de refinamiento de la malla, siendo esta realizada en la línea 15. Cabe destacar el empleo de la librería *alive_bar* para el seguimiento de los m puntos calculados. De igual modo, es de gran importancia las variables de salida, con especial importancia *regress*. Esta variable es un objeto que contiene toda la información de obtenida de la regresión lineal realizada mediante *linregress*.

```
1  def Convergence_rate(U0, t1, F, T_S, problem):
2
3      m = 10 #number of points
4      N = size(t1) - 1
5      Nt2 = size(t1)
6      U1= problem(F, t1, U0, T_S)
7
8      log_Er = zeros(m); log_N = zeros(m)
9
10
11     U2 = zeros( shape(U1) )
12     with alive_bar(m) as bar:
13         for i in range(m):
14             Nt2 = 2*Nt2
15             t2 = linspace(0, t1[-1],Nt2) #t[-1] return the last value of
       the vector or list
16             U2 = problem(F, t2, U0, T_S)
17             log_Er[i] = log10(norm(  U2[:, -1]- U1[:,-1] ) )
18             log_N[i] = log10(Nt2)
19             t1 = t2
20             U1 = U2
21             bar()
22
23     regress = linregress(log_N, log_Er) #Its's a object with all the
       information
24
25     return log_Er, log_N, regress
26
27
```

Listing 2.10: Función Convergence_rate

2.3. Jab_Newt.py

A este módulo se le ha agregado tres funciones para realizar la factorización LU y la integración por el método de Newton sin emplear ninguna librería.

Se han definido la función factorización LU, la cual se han empleado dos formas. La primera de ellas resultó ser no funcional en Python. Esta función ha sido extraída del libro *How to learn Applied Mathematics through modern FORTRAN*. Tras ello se decidió comparar con otras metodologías visualizadas por internet, como separar el cálculo de las matrices L y U; y tras este juntar dichas matrices. Dada la no optimización de estas funciones, los resultados poseen un gran error, como se puede apreciar en la Figura 3.8.

```
    def LU(A):
    N = size(A,1)
    A[1:N,0] = A[1:N,1] / A[0,0]

    for k in range (1,N):
        for j in range(k,N):
            A[k,j] = A[k,j] - dot( A[k, 1:k-1], A[1:k-1, j] )

    for i in range(k+1,N):
         A[i,k] =(A[i,k] - dot(A[0:k,k], A[i,0:k])) / (A[k,k])

    return A


```

Listing 2.11: Factorización LU no funcional

```
    def LU(A):
  N = size(A,1)
  U = zeros([N,N])
  L = zeros([N,N])

  U[0,:] = A[0,:]
  for k in range(0,N):
    L[k,k] = 1

  L[1:N,0] = A[1:N,0]/U[0,0]


  for k in range(1,N):

    for j in range(k,N):
      U[k,j] = A[k,j] - dot(L[k,0:k], U[0:k,j])

    for i in range(k+1,N):
      L[i,k] =(A[i,k] - dot(U[0:k,k], L[i,0:k])) / (U[k,k])

  return L@U

def GAUSS(A,b):

    x = zeros(size(b))
```

```
26      y = zeros(size(b))
27
28      A = LU(A)
29      y[0] = b[0]
30
31      for i in range(size(b) ):
32          y[i] = b[i] - dot( A[i,0:i], y[0:i] )
33
34      x[size(b)-1] = y[size(b)-1] / A[size(b)-1,size(b)-1]
35
36      for i in range(size(b)-2, -1, -1):
37          x[i] = ((y[i] - dot(A[i, i+1:size(b)+1], x[i+1:size(b)+1])) / A[i
    ,i])
38
39      return x
40
41  def Inversa(A):
42
43      B = zeros([size(A,1), size(A,1)])
44
45      for i in range(size(A,1)):
46
47          a = zeros(size(A,1))
48          a[i] = 1
49
50          B[:,i] = GAUSS(A, a)
51
52      return B
53
```

Listing 2.12: Factorización e inversa

2.4. Plots.py

El módulo *Plots.py* ha sufrido dos variaciones importantes: la creación de una función *Save_plot* y la creación de tres funciones nuevas: dos para graficar, separada y junto, os errores de los esquemas numéricos y otro para el ratio de convergencia y la regresión lineal.

Save_plot

Se ha procedido ha desarrollar esta función para cumplir con la metodología explicada en clase, siendo llamada esta función por todas las funciones *Plot*, en vez de ser parte de su propio código. No incorpora ninguna novedad en su código.

```
1   def Save_plot(Save, file):
2
3       if not Save:
4           plt.show()
5
6       else:
```

```
        ##Save Plots

        path = os.path.join(os.getcwd(), 'Plots')

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

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

```

Listing 2.13: función Save_plot

En las siguientes funciones se podrá observar como se llama a esta función para proceder, o no, al guardado de las figuras.

Plot_Er y Plot_Er_compare

Sin gran novedad, se desarrolla la función *Plot_Er*, programada de igual modo que las descritas en el *Milestone 2*, a excepción de la imposición de colores para las funciones graficadas.

Para la función *Plot_Er_compare* se puede observar, como novedad, la creación de dos gráficas en el mismo *plot*. De este modo se puede comparar los errores cometidos para diferentes Δt. Se observa una lista *colors*,la cual está concebida para definir los mismos colores en ambas gráficas para un mismo Δt. Destacar que, dado que en el bucle se recorre un *string*, se hace necesario convertirlo a un número entero. Esto es posible a los nombre impuestos a las *Keys* de los diccionarios.

```
def Plot_Er(Er,t, T, dt, T_S, Save):
    fig, ax = plt.subplots( figsize = (10, 10) )
    ax.plot(t, Er[0,:], color = 'b' , label = 'Er_x' )
    ax.plot(t, Er[1,:], color = 'r', label =  'Er_y' )
    ax.set_xlabel('t')
    ax.set_ylabel('Er')
    ax.set_title(T_S + ' dt = ' + str(dt) + ' T = ' + str(T) )
    ax.legend()

    file = T_S + "_dt_" + str(dt) + '_T_' + str(T)  +".png"
    Save_plot(Save, file)

def Plot_Er_compare(Er,t, T, dt, T_S, Save):
    fig, (ax, ay, az) = plt.subplots(3,1,figsize=(15, 15))
    colors = ['r','b','g']
    for i in t:
        Er_n = sqrt(Er[i][0,:]**2 + Er[i][1,:]**2)
        ax.plot(t[i],Er[i][0,:], color = colors[int(i)], label= T_S + "
    dt " + str(dt[int(i)]))
        ay.plot(t[i],Er[i][1,:], color = colors[int(i)], label= T_S + "
    dt " + str(dt[int(i)]))
        az.plot(t[i],Er_n, color = colors[int(i)], label = 'Norm' + " dt
    " + str(dt[int(i)]))
    ax.set_xlabel('t')
```

```
    ax.set_ylabel('Er_x')
    ay.set_xlabel('t')
    ay.set_ylabel('Er_y')
    az.set_xlabel('t')
    az.set_ylabel('Er_norm')
    ax.set_title(T_S + ' Er' + ' T = ' + str(T))
    ax.legend()
    ay.legend()
    az.legend()

    file = T_S + '_comprare_T' + str(T) + ".png"
    Save_plot(Save, file)

```

Listing 2.14: funciones Plot_Er y Plot_Er_compare

Plot_Conv_Rat

Sin gran variedad con lo descrito hasta ahora, se define esta función para poder graficar el ratio de convergencia y la regresión lineal. Como se comentó anteriormente, la variable *regress* es un objeto que contiene toda la información de la regresión lineal. De este modo, con una sola variable se puede importar la función de regresión linreal ($regress.intercept + regress.slope\dot{x}$), el orden (textitabs(regress.slope)) y el coeficiente de determinación, R^2 ($regress.rvalue$.

```
def Plot_Conv_Rat(x, y, T, dt, regress, T_S, Save):
    fig, ax = plt.subplots(figsize = (10, 10))
    ax.plot(x,y, color = 'b', label = 'Numerical results')
    ax.plot(x, regress.intercept + regress.slope*x, '--', color = 'r',
   label = 'linear regression')
    ax.set_xlabel('Log_N')
    ax.set_ylabel('Log_Er')
    ax.set_title(T_S + ': T = ' + str(T) +', dt = ' + str(dt) + '. R^2 =
   ' + str(round(regress.rvalue**2,3)) + ' & Order = ' + str(round(abs(
   regress.slope),3)))
    ax.legend()

    file = T_S +'_dt_'+ str(dt) +'_conver' + ".png"
    Save_plot(Save, file)

```

Listing 2.15: función Plot_Conv_Rat

3. Resultados

En esta sección se expondrán los resultados obtenidos mediante el código ya expuesto en los anteriores *Milestones* y el detallado en este informe.

3.1. Error de los esquemas numéricos

Euler

Como ya se ha podido ver en anteriores informes, el método de Euler conlleva un gran error acumulativo (Figura 3.1d). En el caso de $\Delta t = 0{,}1$ se aprecia un sincronismo del error hasta $T = 60$, donde se rompe, ocasionando la divergencia de la solución. Para el resto de casos, se aprecia el mismo sincronismo pero sin la rotura del mismo. Para el caso de la Figura 3.1c, se observa una reducción del error a partir $T = 80$. Esto es un ciclo, donde disminuye el error para luego alcanzar un mayor máximo y siendo estos ciclos menores, como se puede apreciar en la 3.1e.

Runge-Kutta orden 4

Como bien se conoce, el esquema de Runge-Kutta de orden 4 obtiene resultados con un bajo error en el proceso de computación. Como se puede observar en las Figuras 3.4, donde el error varía de forma lineal con Δt, estando entre $Er \approx 0{,}04 - 0{,}004$. Como se puede observar en la Figura 3.4,

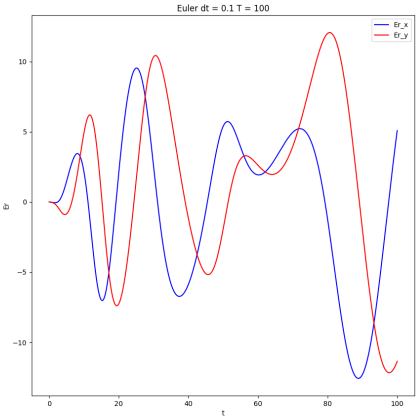
Crank-Nicolson

Crank-Nicolson obtiene resultados con una precisión muy similar al de Runge-Kutta de orden 4, con una divergencia del error similar a la mostrada en las Figuras 3.3. Comparando las Figuras 3.4 y 3.6 se observa que C-N obtiene, para un mismo tiempo, un error menor.

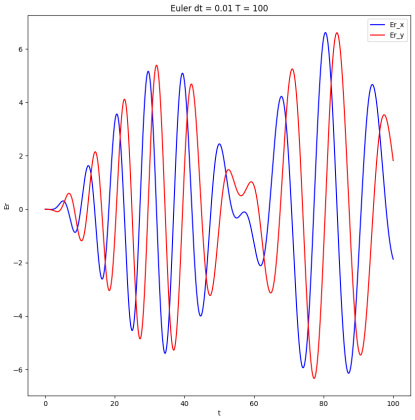
Euler inverso

Por último, se encuentra el esquema implícito Euler inverso. En este caso, para $\Delta t = 0{,}1$ el error diverge para tiempos muy cercanos al inicio, no obteniéndose ningún resultado adecuado debido a ese alto error.

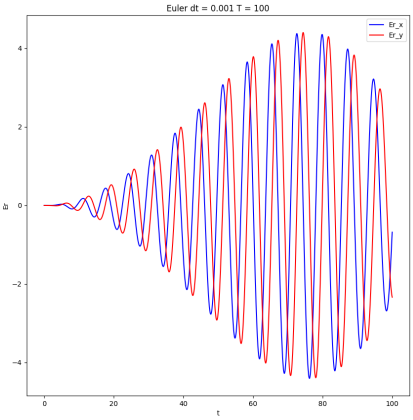
Para $\Delta t = 0{,}01 - 0{,}001$ se obtienen resultados dada la no brusca divergencia del error. Para el caso $\Delta t = 0{,}01$, se observa que, para valores del tiempo entorno a 40 el error se anula. Esto se debe porque para una valor de $\Delta t = 0{,}01$ se produce una espiral hasta el valor 0, lugar donde la solución converge, anulando de dicho modo el error.



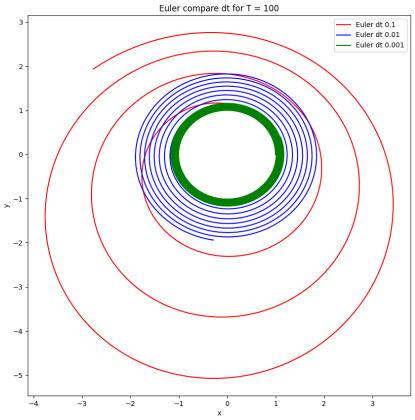
(a) $\Delta t = 0{,}1$



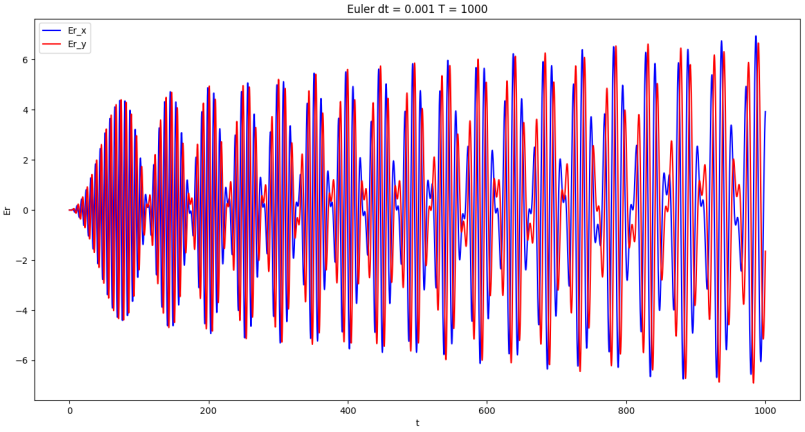
(b) $\Delta t = 0{,}01$



(c) $\Delta t = 0{,}001$



(d) Solución Esquema de Euler



(e) $\Delta t = 0{,}0001; T = 1000$

Figura 3.1: Error y solución esquema de Euler

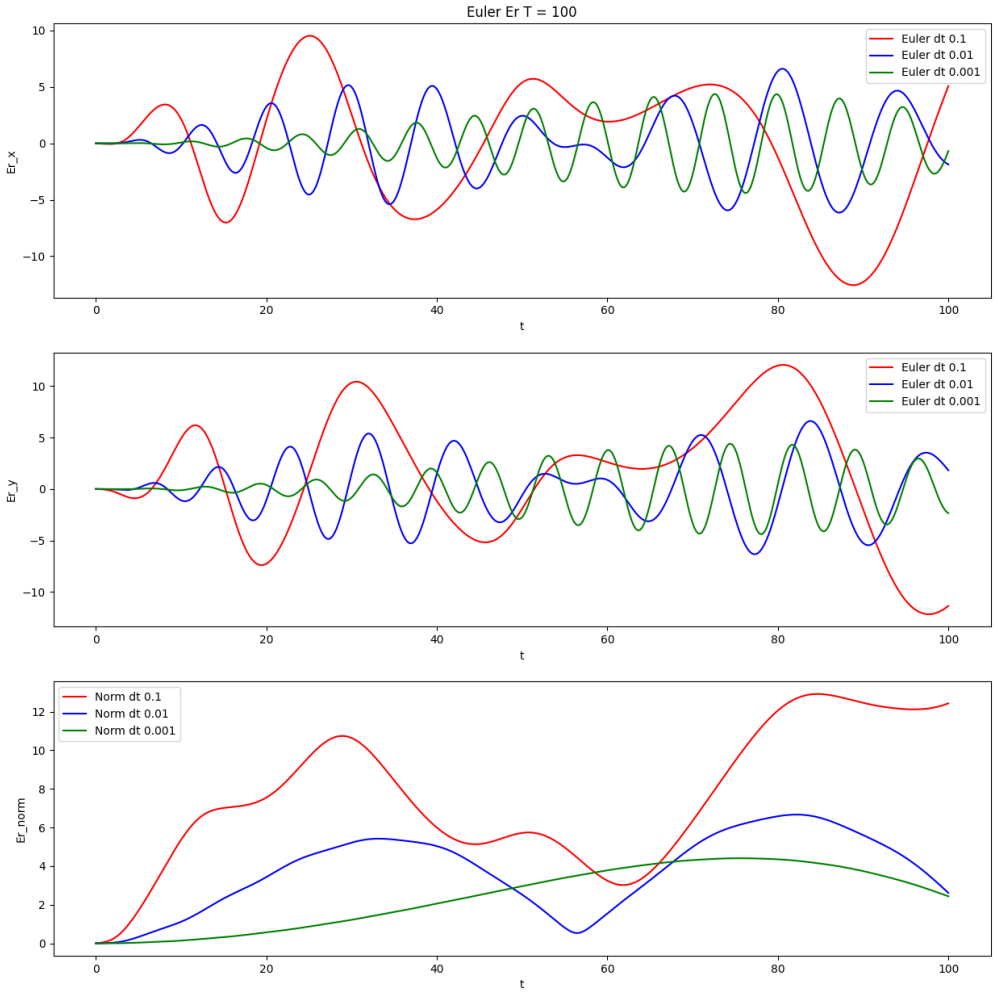
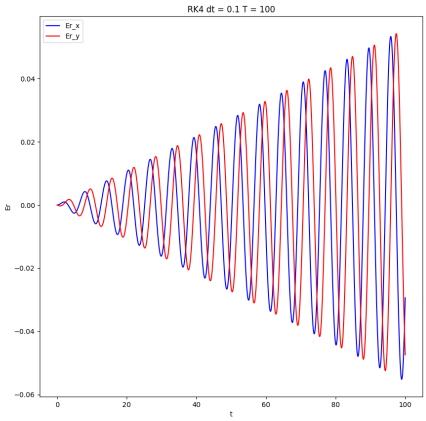
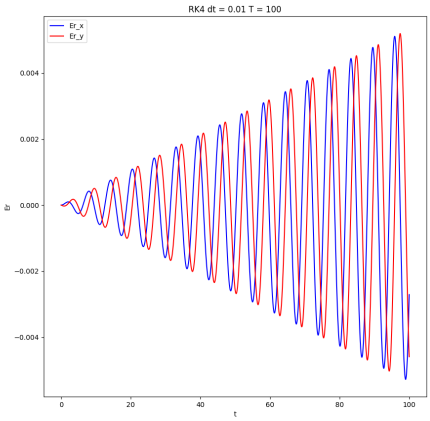


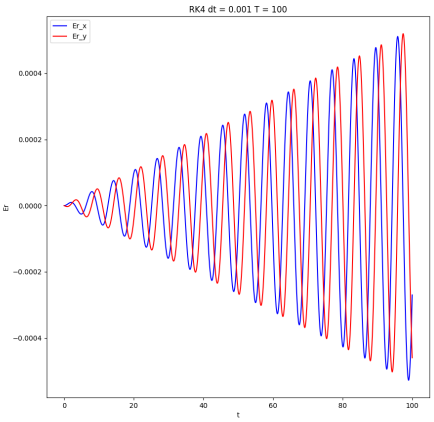
Figura 3.2: Comparación errores Euler para los diferentes Δt



(a) $\Delta t = 0{,}1$



(b) $\Delta t = 0{,}01$



(c) $\Delta t = 0{,}001$

Figura 3.3: Error y solución esquema RK4

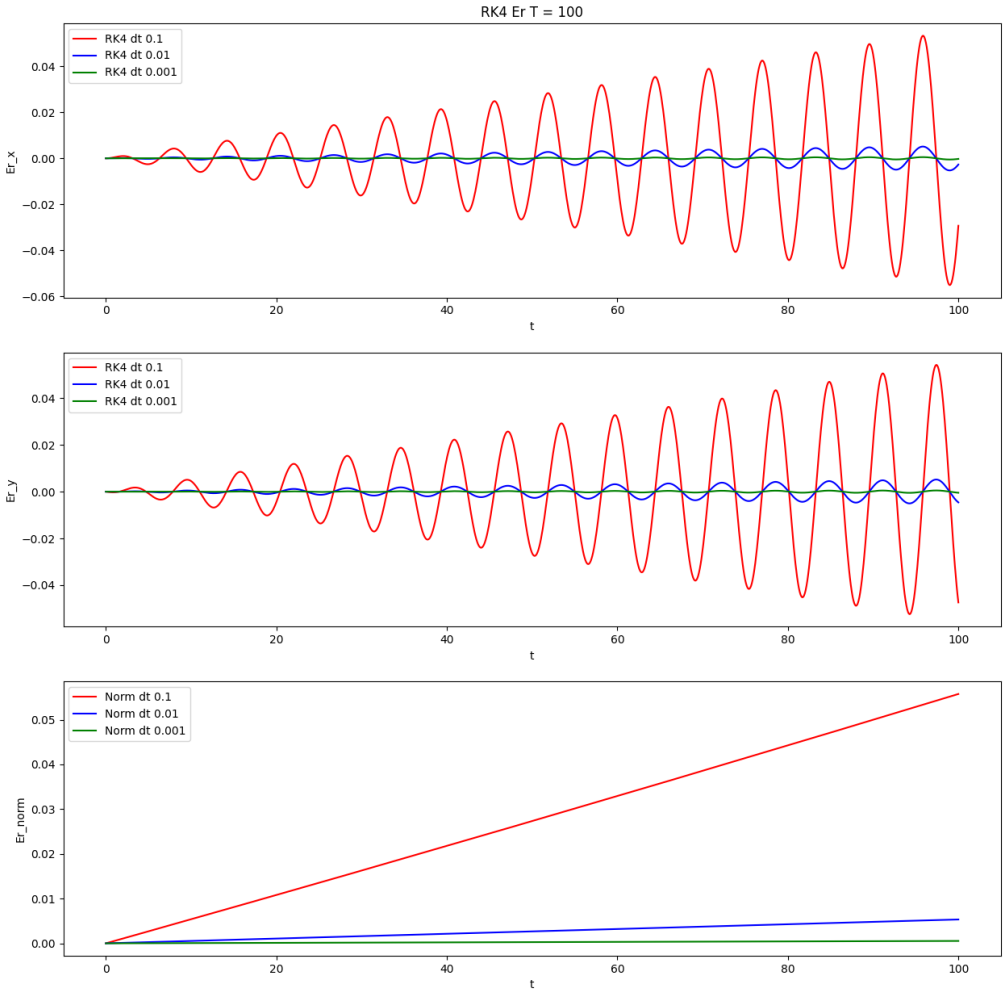
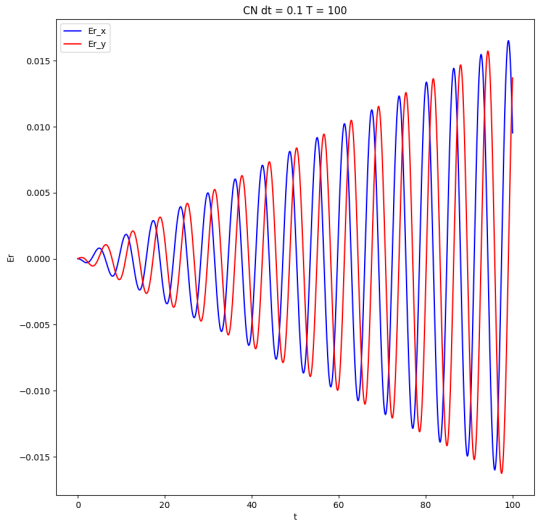
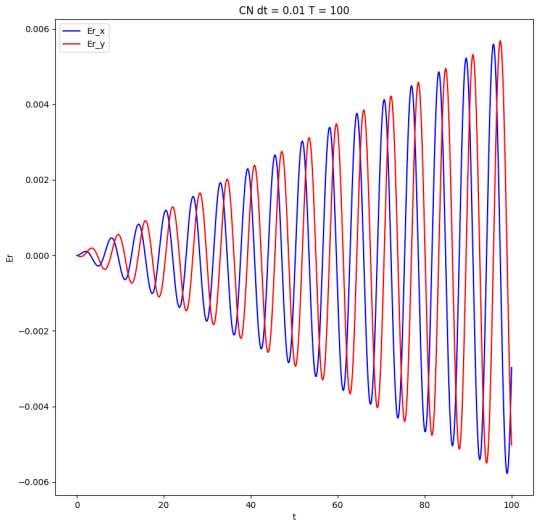


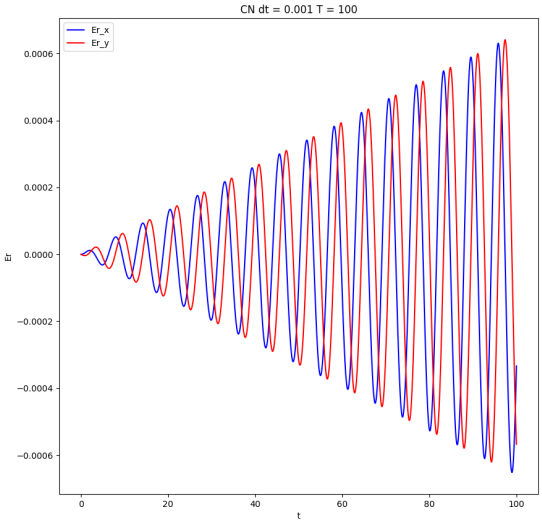
Figura 3.4: Comparación errores RK4 para los diferentes Δt



(a) $\Delta t = 0{,}1$



(b) $\Delta t = 0{,}01$



(c) $\Delta t = 0{,}001$

Figura 3.5: Error y soluciones esquema C-N

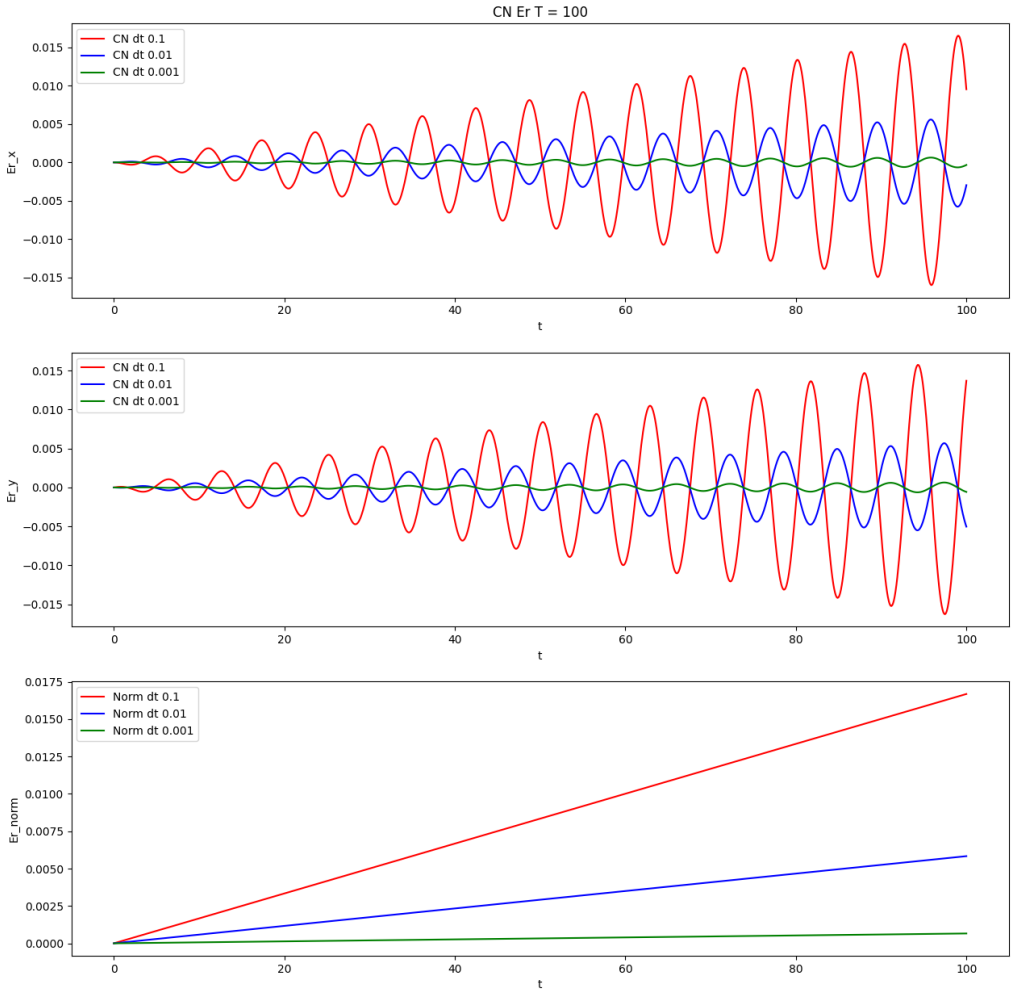
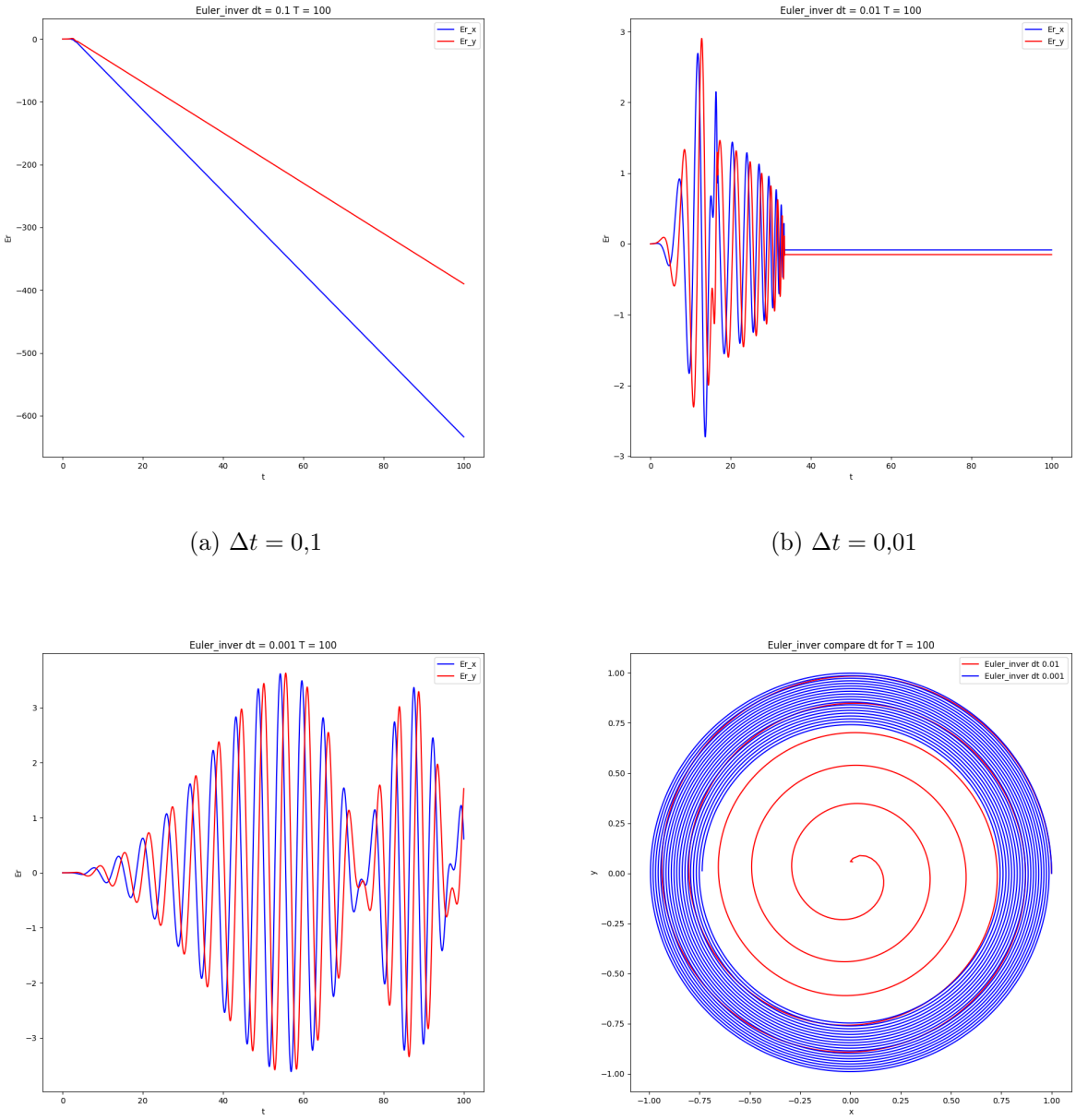


Figura 3.6: Comparación errores C-N para los diferentes Δt

Como se puede apreciar en la Figura3.7d,



(a) $\Delta t = 0{,}1$

(b) $\Delta t = 0{,}01$

(c) $\Delta t = 0{,}001$

(d) Solución Euler inverso

Figura 3.7: Error y soluciones esquema Euler inverso

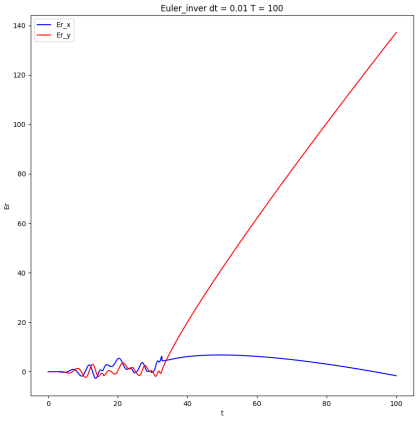


Figura 3.8: Error de Euler inverso integrado mediante la integración de Newton

3.2. Ratio de convergencia

Por último, se detallan los resultados de los ratios de convergencia y la estimación del orden de los diferentes esquemas numéricos empleados. Dicha estimación se ha realizado mediante una regresión linea. También se podría haber realizado con la función *scipy optimize curve fit*.

Las siguientes convergencias han sido calculadas para $T = 10$ segundos y $\Delta t = 0{,}1 - 0{,}001$, a excepción de Runge-Kutta de orden 4. Del mismo modo, recordar que el valor del número de iteraciones (N) es inversamente proporcional a Δt, pues:

$$N = \frac{T}{\Delta t}$$

Cabe destacar el gran tiempo de procesamiento de los datos mostrados a continuación, pues se debe refinar la malla las veces necesarios hasta la convergencia del esquema numérico.

Euler

Comenzando con el esquema de Euler, se observa que para un $\Delta t = 0{,}001$ ya se obtiene una buena convergencia, con un orden aproximado a 1.

Runge-Kutta de orden 4

En el caso de la convergencia para Runge-Kutta, se observa que para el valor preestablecido de $\Delta t = 0{,}1$ ya se ha realizado la convergencia, lo por cual, se necesita un número menor de iteraciones, tal y como se puede apreciar en las Figuras 3.11a y 3.11b. Para los casos mostrados en las Figuras 3.11c y 3.11d,se observan error de precisión de máquina, pues son menores de 10^{-12}. Con este esquema y el esquema de Crank-Nicolson se obtienen los resultados con menor error.

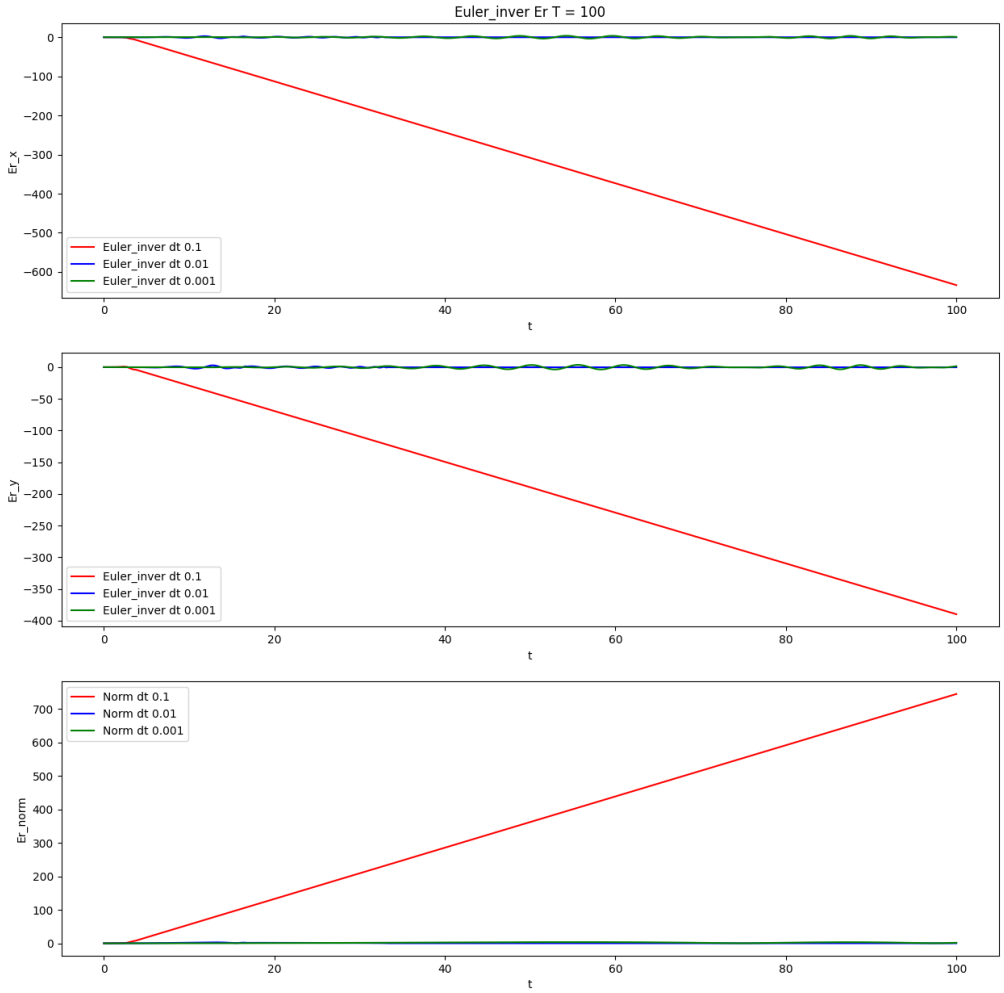


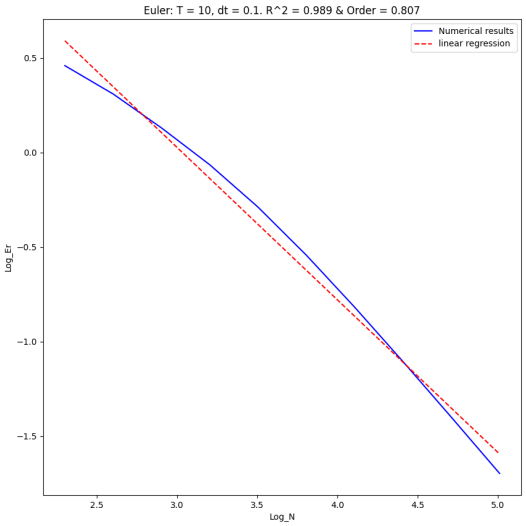
Figura 3.9: Comparación errores Euler inverso para los diferentes Δt

Crank-Nicolson

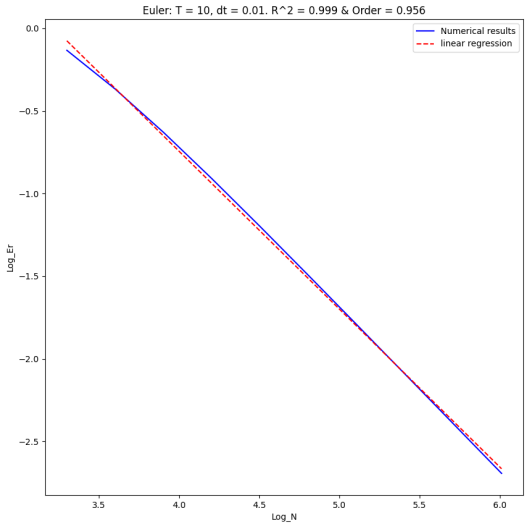
En el caso de este esquema numérico, los resultados convergen para $\Delta t = 0{,}01$ ya se obtiene una buena convergencia. En comparación con Runge-Kutta de orden 4, para una precisión similar, tiene una convergencia más lenta.

Euler inverso

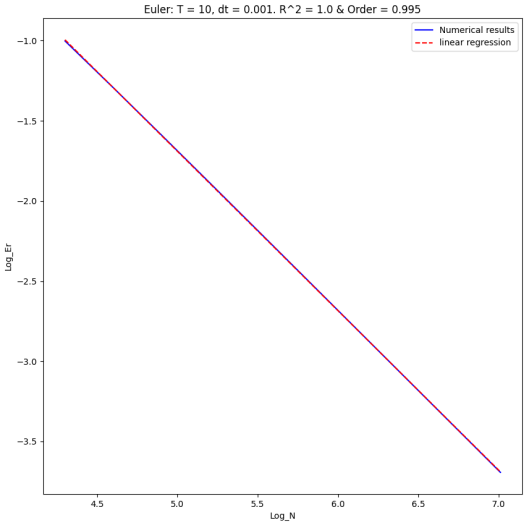
Por último, en el esquema de Euler inverso, se puede apreciar, del mismo que en el Esquema de Euler, una convergencia a partir de valores de $\Delta t = 0{,}001$.



(a) $\Delta t = 0{,}1$

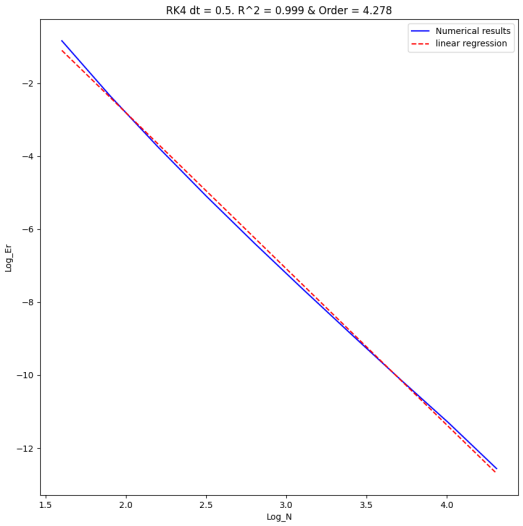


(b) $\Delta t = 0{,}01$

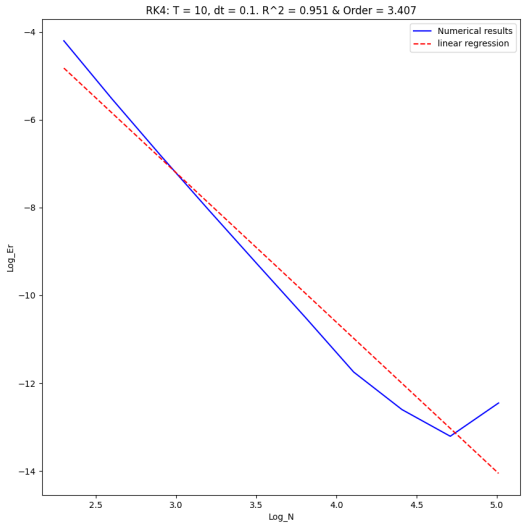


(c) $\Delta t = 0{,}001$

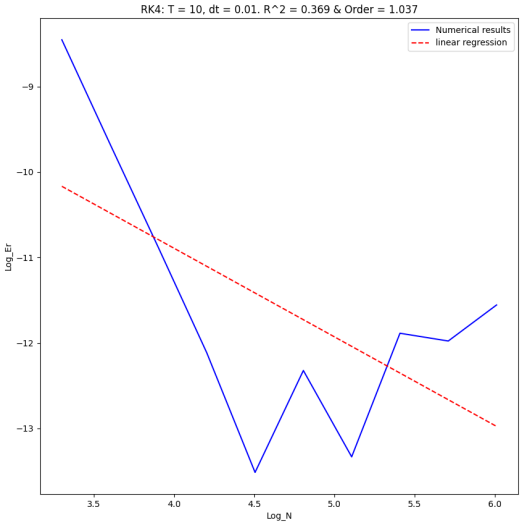
Figura 3.10: Ratio de convergencia de Euler para diferentes mallas



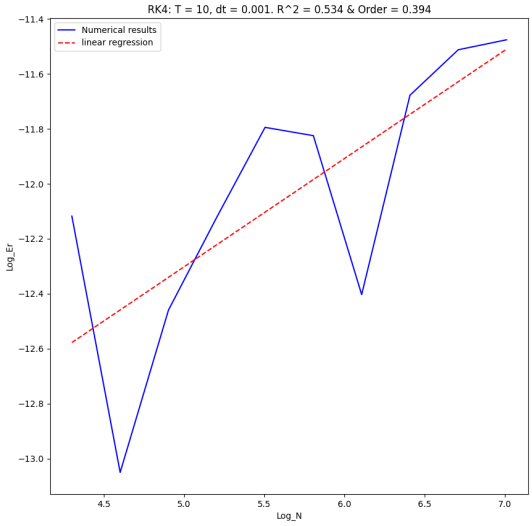
(a) $\Delta t = 0{,}1$



(b) $\Delta t = 0{,}1$

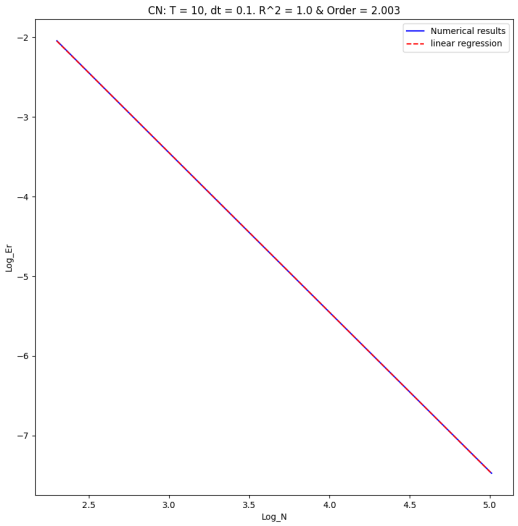


(c) $\Delta t = 0{,}01$

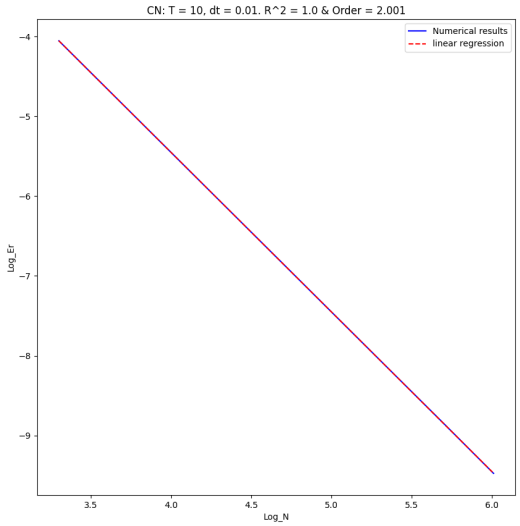


(d) $\Delta t = 0{,}001$

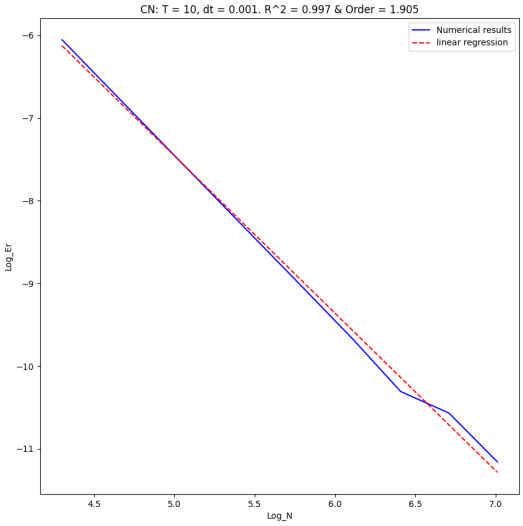
Figura 3.11: Ratio de convergencia de RK4 para diferentes mallas



(a) $\Delta t = 0{,}1$

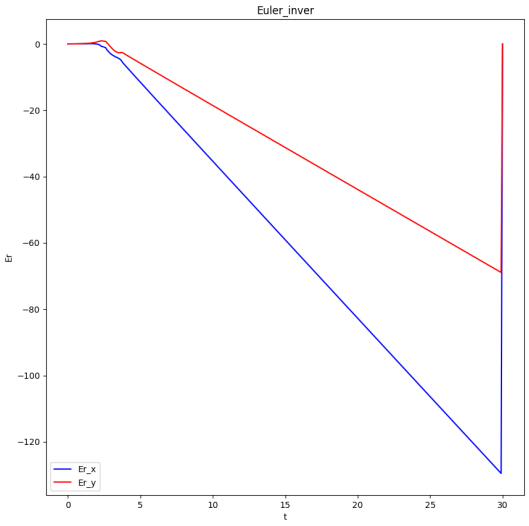


(b) $\Delta t = 0{,}01$

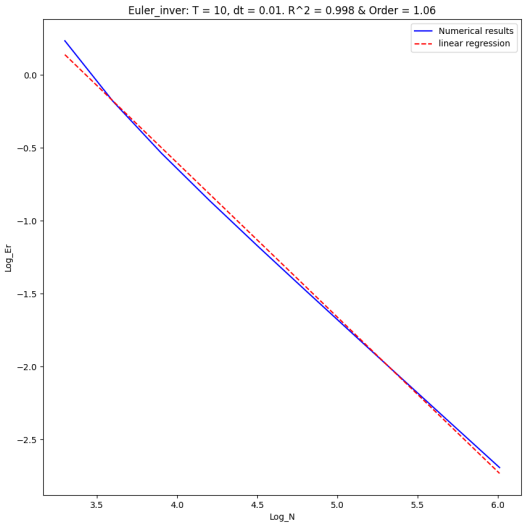


(c) $\Delta t = 0{,}001$

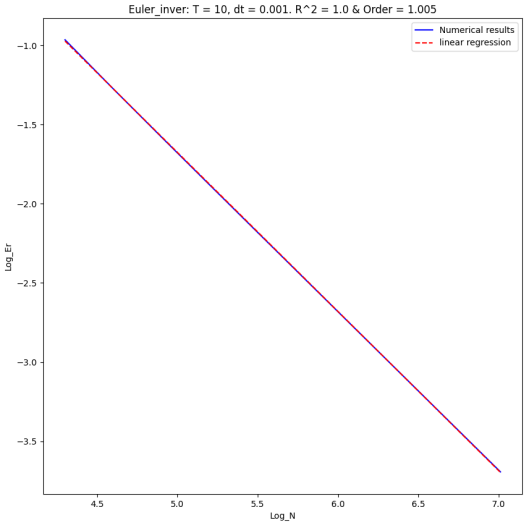
Figura 3.12: Ratio de convergencia de C-N para diferentes mallas



(a) $\Delta t = 0{,}1$



(b) $\Delta t = 0{,}01$



(c) $\Delta t = 0{,}001$

Figura 3.13: Ratio de convergencia de Euler inverso para diferentes mallas