


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

MILESTONE III

AMPLIACIÓN DE MATEMÁTICAS I

23 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 3 de la asignatura, el cual parte de los hitos anteriores consistentes en la integración numérica de una órbita kepleriana para el problema de valores iniciales (problema de Cauchy), que se recuerda en la ecuación (1).

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

A partir de ello, se evalúa el error cometido por el esquema numérico a través de la extrapolación de Richardson, recogida en la ecuación (2), según el cual se estima mediante la diferencia entre las soluciones numéricas asociadas a la simulación y a otra semejante pero introduciendo una malla temporal con el doble de instantes. Además, aparece el factor q o ratio de convergencia del método numérico, el cual se resume de manera teórica en la tabla, , aunque se comenta a continuación su cálculo.

$$E = \frac{U^{2N} - U^N}{1 - \frac{1}{2^q}} \quad . \tag{2}$$

Tabla 1: Ratio de convergencia teórica de los métodos numéricos.

Esquema	Euler	Euler Inverso	Crank-Nicolson	Runge Kutta 4
q	1	1	2	4

Relativo al ratio de convergencia, q, definida como el número de iteraciones necesarias por el método para converger a una solución lo suficientemente precisa, se implementa una regresión lineal de los datos refresentados en una gráfica la norma de la diferencia entre esas dos soluciones, $log(|U_{2N} - U_N|)$ y el $log(N)$. En concreto, el tramo decreciente lineal.

De cara a los módulos implementados, partiendo de aquellos utilizados en el hito anterior, se añade:

- MILESTONE_03.py: módulo encargado de hacer las llamadas a los submódulos que contienen el problema a resolver (*Error.py*, *NewtonSolve.py*), así como las gráficas del problema. Las condiciones iniciales vienen dadas en *MILESTONES_AM1.py*, es decir, el tiempo fonal, número de puntos de integracióny las condiciones iniciales.
- Error.py: contiene las funciones asociadas al cálculo del error mediante la extrapolación de Richardlison, así como el ratio de convergencia. Para el error, se requiere de

input el problema a resolver (Kepler), los esquemas numéricos implementados, el vector t de instantes temporales, el orden del esquema definido previamente con el ratio de la convergencia) y las condiciones iniciales, U_0, devolviendo el valor del mismo.

En el ratio, como input se requiere también el problema y esquemas utilizados, vector de tiempos y condiciones iniciales, y el número de puntos a pintar en la gráfica logarítmica, devolviendo $log(|U_{2N} - U_N|)$, $log(E)$, $log(N)$, los valores de la recta de regresión lineal y el orden del esquema

- NewtonSolve.py: encargado de la resolución de ecuaciones por el método de Newton, en vez de utilizar la función de la librería *SciPy*.

A continuación, se presentan los códigos de los módulos anteriores en las figuras 1, 1, 1 y 1.

```
def ErrorNumerico3( tf, N, U0):
    schemes = [ Euler_Scheme, RK4_Scheme, CrankNicolson_Scheme,
       InvEuler_Scheme ]
    GrafSchemes = ['Euler Scheme', 'RK4 Scheme', 'Cranck-Nicolson
       Scheme', 'Inverse Euler Scheme']
    t = linspace(0, tf, N) #[0, N) hasta N-1 pero N puntos
    deltat = round(tf/(N-1),4)
    i = 0
    for Scheme in schemes:
        [logE, logN, logE_lineal, logN_lineal, order, mError,
           logE_total] = ConvergenceRate(Kepler, Scheme, t, U0, 8)
        Error = ErrorNum(Kepler, Scheme, t, U0, order = int(round_(
           abs(mError))))
        ErrorNorm = (Error[:,0]**2+Error[:,1]**2)**(1/2)
```

Figura 1.1: Código asociado a MILESTONE_03.py

```
def ConvergenceRate( Kepler, Scheme, t, U0, m):
    N = len(t)
    N2 = 2*len(t)
    tf = t[N-1]
    logE = zeros(m)
    logN = zeros(m)
    t1 = t #N puntos, va de 0 a N-1 (tf), N puntos
    t2 = linspace(0,tf,N2) #2a malla con doble de puntos
    # recordamos que el 0 cuenta y es [0,N) = [0, N-1]
    U1 = CauchyProblem(Kepler, t1, U0, Scheme)
    for i in range(m):
        U2 = CauchyProblem(Kepler, t2, U0, Scheme)
        logE[i] = log10(norm(U2[-1,:]-U1[-1,:]))
        logN[i] = log10(N)
        if i == (m-1):
            break
        t1 = t2
        U1 = U2
        N = N2
        N2 = 2*N2
        t2 = linspace(0,tf,N2)
    for j in range(m):
            if abs(logE[j]) > 12 :  break #evito tramo final si
               aumenta
    j = min(j, m-1)
    reg = LinearRegression().fit(logN[0:j+1].reshape((-1, 1)),logE
       [0:j+1])
    order = round_(abs(reg.coef_),1)
    mError = reg.coef_
    logN_lineal = logN[0:j+1]
    logE_lineal = reg.predict(logN[0:j+1].reshape((-1, 1)))
    logE_total = logE[:] - log10(1 - 1 / (2**order))
    return logE, logN, logE_lineal, logN_lineal, order, mError,
       logE_total

def ErrorNum( Kepler, Scheme, t, U0, order):
    N = len(t)
    tf = t[N-1]
    E = zeros([N, len(U0)])
    t1 = t #N puntos, va de 0 a N-1 (tf), N puntos
    t2 = linspace(0,tf,2*N) #2a malla con doble de puntos
    # recordamos que el 0 cuenta y es [0,N) = [0, N-1]
    U1 = CauchyProblem(Kepler, t1, U0, Scheme)
    U2 = CauchyProblem(Kepler, t2, U0, Scheme)
    for i in range(0,N):
        E[i,:] = (U2[2*i,:] - U1[i,:])/( 1 - 1./(2**order))
    return E
```

Figura 1.2: Código asociado a Error.py

```
def Jac(F, U):
  N = size(U)
  J= zeros([N,N])
  t = 1e-3
  for i in range(N):
    xj = zeros(N)
    xj[i] = t
    J[:,i] = (F(U + xj) - F(U - xj))/(2*t)
  return J
def factLU(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, L, U]
def solveLU(M,b):
  N=size(b)
  y=zeros(N)
  x=zeros(N)
  [A,L,U] = factLU(M)
  y[0] = b[0]
  for i in range(0,N):
    y[i] = b[i] - dot(A[i,0:i], y[0:i])
  x[N-1] = y[N-1]/A[N-1,N-1]
  for i in range(N-2,-1,-1):
    x[i] = (y[i] - dot(A[i, i+1:N+1], x[i+1:N+1])) / A[i,i]
  return x
```

Figura 1.3: Código asociado a NewtonSolve.py parte (12)

```
def Inv(A):
  N = size(A,1)
  B = zeros([N,N])
  for i in range(0,N):
    one = zeros(N)
    one[i] = 1
    B[:,i] = solveLU(A, one)
  return B
def newton(func, U_0):
  N = size(U_0)
  U = zeros(N)
  U1 = U_0
  error = 1
  stop = 1e-8
  iteration = 0
  while error > stop and iteration < 1000:
    U = U1 - dot(Inv(Jac(func, U1)),func(U1))
    error = norm(U - U1)
    U1 = U
    iteration = iteration +1
  return U
```

Figura 1.4: Código asociado a NewtonSolve.py parte (22)

2. Resultados

A continuación, se presentan los resultados asociados al error y ratio de convergencia para diversos pasos temporales $\Delta t = [0{,}1,\ 0{,}05,\ 0{,}01]$.

2.1. Error numérico

En la figura 2.1 se muestra el error asociado al primer paso temporal, donde destaca la magnitud del error para los métodos de convergencia más lenta (Euler y Euler inverso), al ser de orden unidad. Además, destaca el Euler inverso más aún, aunque se asocia a la implementación propia del método newton en lugar del uso de la función de la librería SciPy. En las figuras 2.2 y 2.3 se aprecia como esta magnitud del error, se reduce, como era de esperar, con la mejora de la precisión de la simulación, así como el efecto del método propio de Newton.

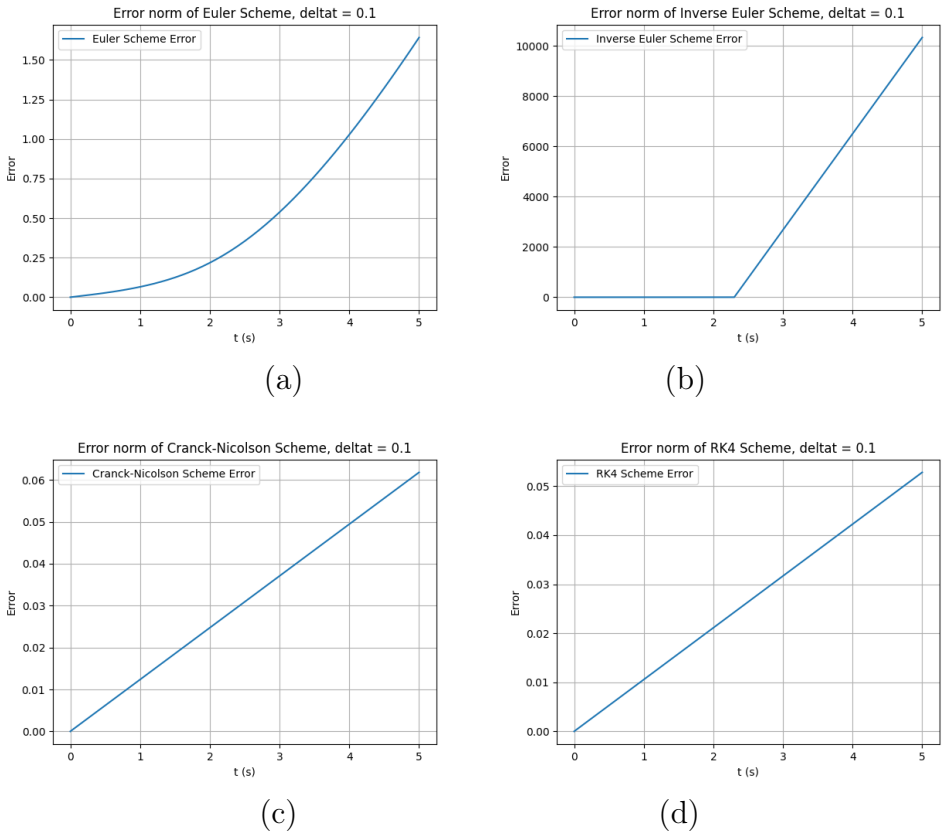
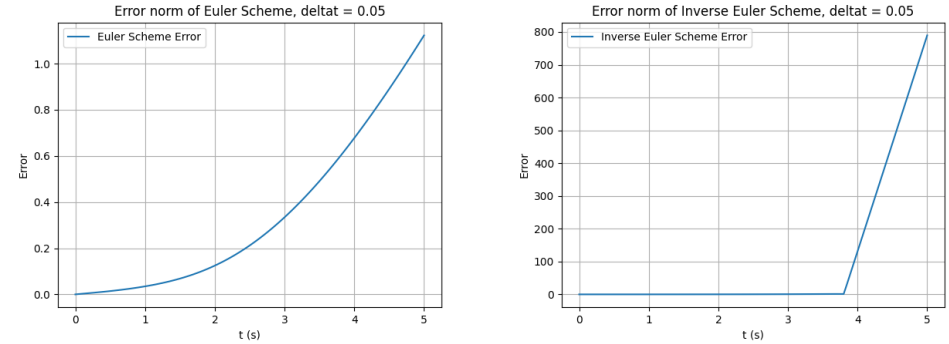
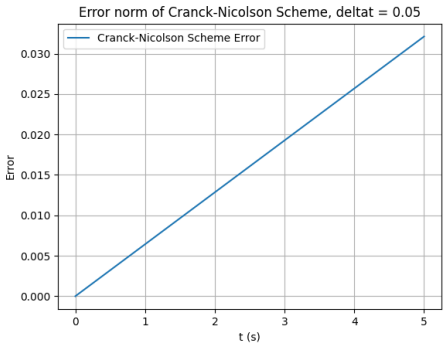


Figura 2.1: Error numérico para $\Delta t = 0{,}1$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4.

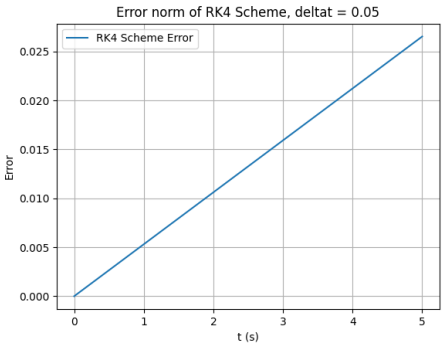


(a)

(b)

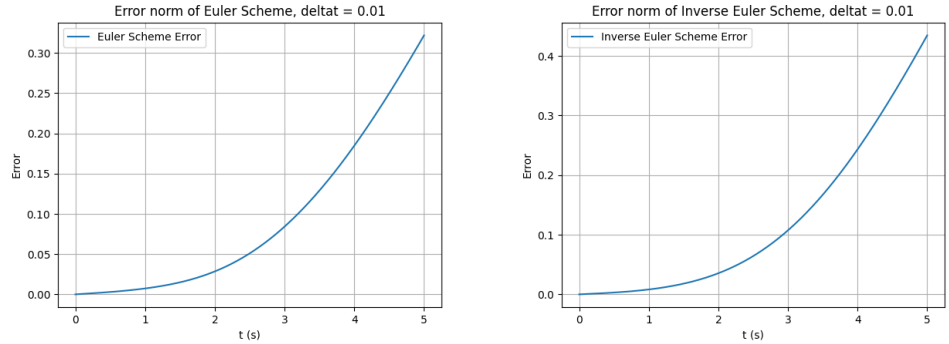


(c)



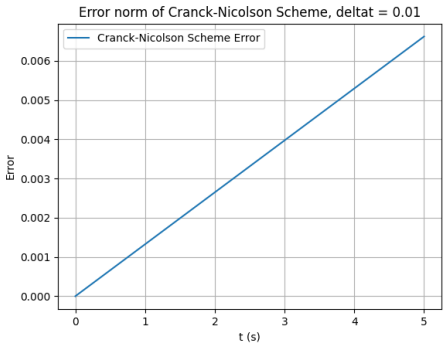
(d)

Figura 2.2: Error numérico para $\Delta t = 0{,}05$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4.

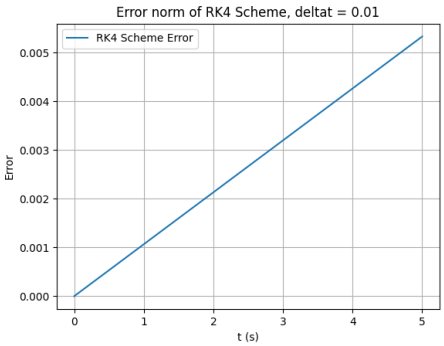


(a)

(b)



(c)



(d)

Figura 2.3: Error numérico para $\Delta t = 0{,}01$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4.

2.2. Ratio de convergencia

Como se ha comentado, el ratio de convergencia hace referencia al número de iteraciones necesarias para que el esquema converja a una solución lo suficientemente precisa y por lo tanto, menor esfuerzo computacional.

En primer lugar, se representa el logaritmo del error cometido, así como el de la norma $U_{2N} - U_N$, que representan la misma curva trasladada una cantidad $1/(1-2^q)$.

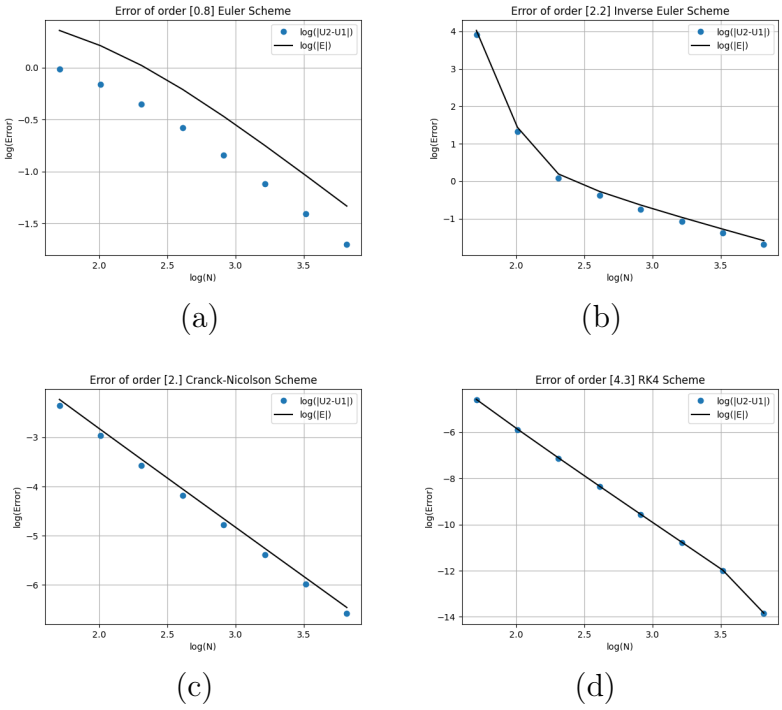


Figura 2.4: Logaritmo del error numérico para $\Delta t = 0{,}1$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4 frente a la norma de $U_{2N} - U_N$.

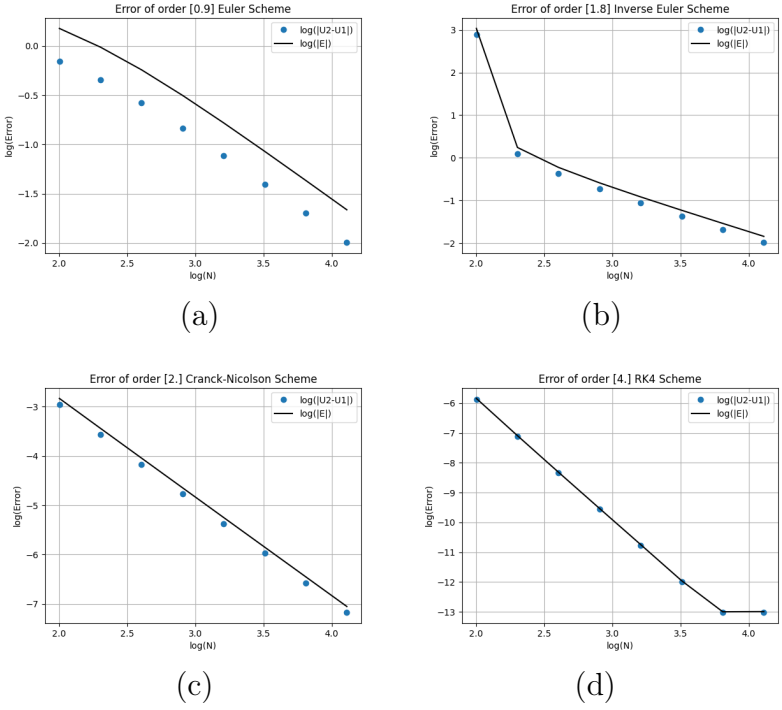


Figura 2.5: Logaritmo del error numérico para $\Delta t = 0{,}05$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4 frente a la norma de $U_{2N} - U_N$

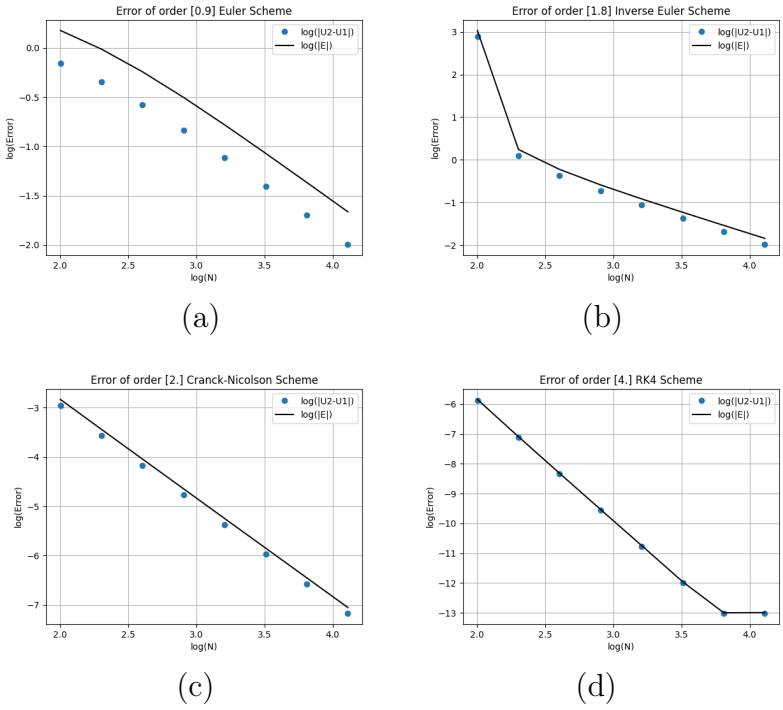


Figura 2.6: Logaritmo del error numérico para $\Delta t = 0{,}01$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4 frente a la norma de $U_{2N} - U_N$

En las figuras 2.4, 2.5 y 2.6 se aprecia cómo el método de Euler presenta mayor discrepancia entre el valor real del error y el de la norma, mientras que en el resto se ajusta más. Mencionar que todos aquellos términos del orden de 10^{12} se corresponden a la precisión del propio ordenador, y el resto a los del propio esquema, por lo que a un determinado número de puntos en el mallado, habrá variaciones innatas en el ajuste de la regresión lineal (como picos en aquellos métodos buenos en lo que respecta a convergencia, RK4 o Crank-Nicholson).

Finalmente, se presentan juntos los valores de la regresión con el del error, para apreciar la similitud, o no, del mismo, en las figuras 2.7, 2.8 y 2.9.

Comentar que solo se tiene en cuenta el tramo lineal para realizar el ajuste.

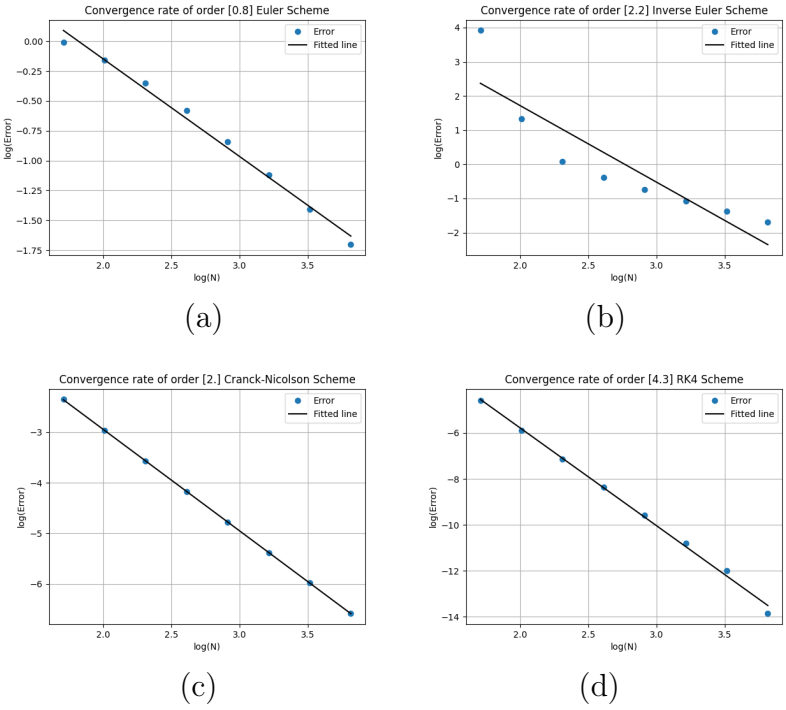


Figura 2.7: Logaritmo del error numérico para $\Delta t = 0{,}1$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4 frente al ajuste realizado.

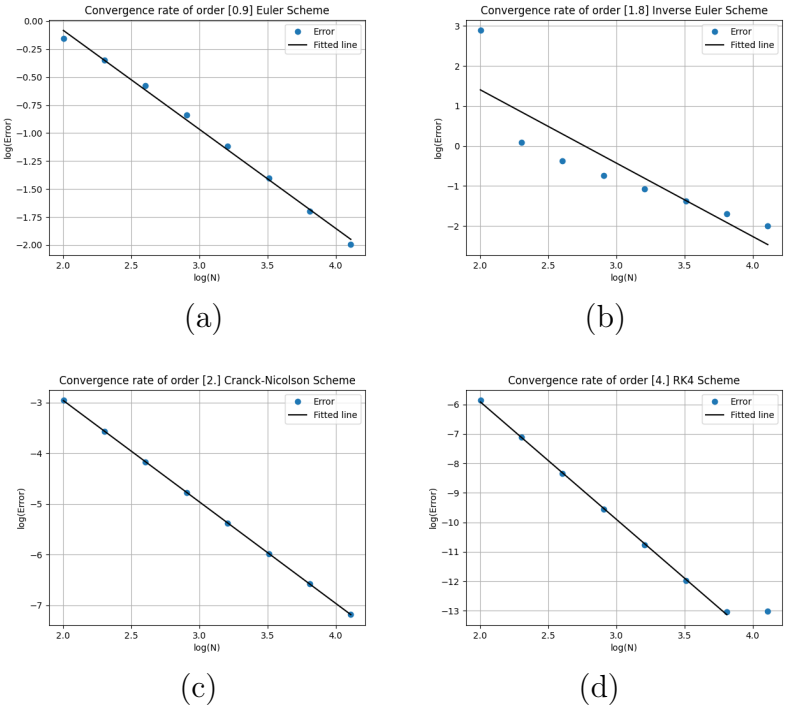
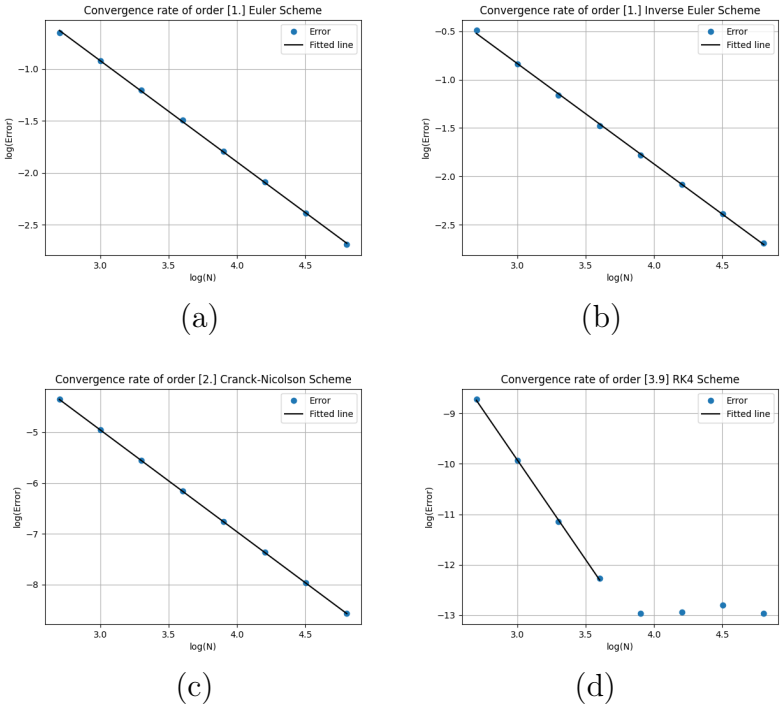


Figura 2.8: Logaritmo del error numérico para $\Delta t = 0{,}05$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4 frente al ajuste realizado.



(a) (b)

(c) (d)

Figura 2.9: Logaritmo del error numérico para $\Delta t = 0{,}01$ con esquema (a) Euler, (b) Euler Inverso, (c) Crank-Nicholson y (d) Runge Kutta 4 frente al ajuste realizado.

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.