


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Milestone III Report

Ampliación de Matemáticas I

9 de octubre de 2022

Autora:

- Rosa Martínez Rubiella

1. Introducción

El objetivo del Hito III es la estimación del error de las soluciones numéricas obtenidas en hitos anteriores. Para esto, se implementa un método de extrapolación de **Richardson** para la evaluación del error en el problema de Cauchy, y se utiliza en los diferentes esquemas temporales usados. A continuación, se programa una función que evalúa el **ratio de convergencia** de dichos esquemas temporales. De esta forma se puede obtener el orden de integración de los mismos mediante una aproximación lineal, como se verá a continuación.

Finalmente, se explican de manera breve las actualizaciones en la estructura y contenido del código, así como el razonamiento seguido en las nuevas implementaciones.

2. Extrapolación de Richardson

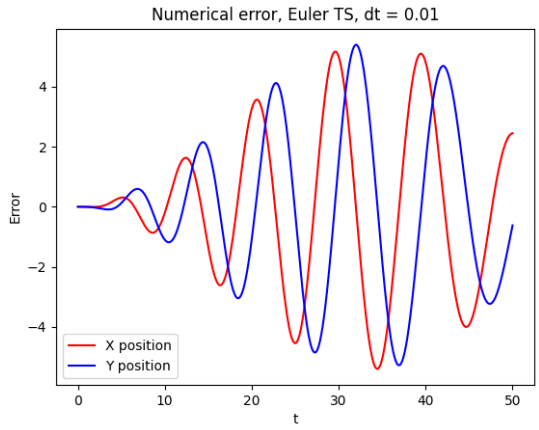
Este método permite evaluar el error que se produce al utilizar un determinado esquema numérico para la resolución de un problema de Cauchy. Para ello, se calcula la solución aproximada en dos mallas temporales diferentes (dentro del mismo intervalo), y se restan en sus puntos coincidentes. En este caso se eligen mallas con un N y 2N puntos de integración, que hace más fácil la evaluación de la solución en dichos puntos. La fórmula resultante queda:

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

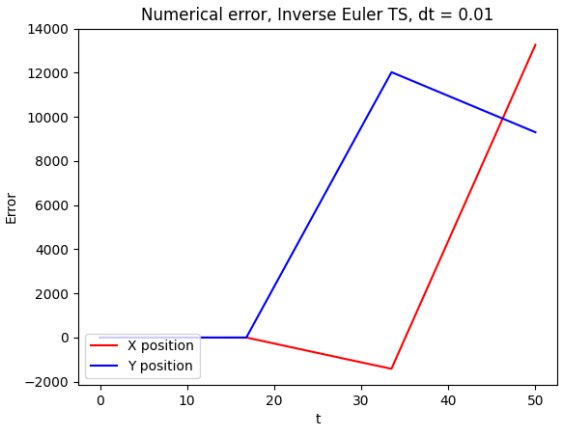
donde q corresponde al orden de integración del esquema utilizado. Dicho orden puede ser obtenido mediante la regresión lineal que posteriormente se muestra o de manera teórica ($q_{Euler} = 1$, $q_{RK-4} = 4$ ... etc). Se realiza una iteración para todos los puntos coincidentes de la malla (N puntos).

Para ejemplificar esto, se realiza una simulación de 50 segundos de duración, con un intervalo de integración de $\mathbf{dt = 0.01}$. Se representa en la **Figura 1** el error en ambas coordenadas de posición para cada esquema temporal. Como se esperaba, tanto el Runge-Kutta-4 como el Crank-Nicolson presentan un error relativamente pequeño, del orden de la centésima aproximadamente. Sin embargo, el Euler presenta errores de orden unidad, mientras que el Euler Inverso diverge totalmente ya que necesita más puntos de integración como se vio en el informe anterior.

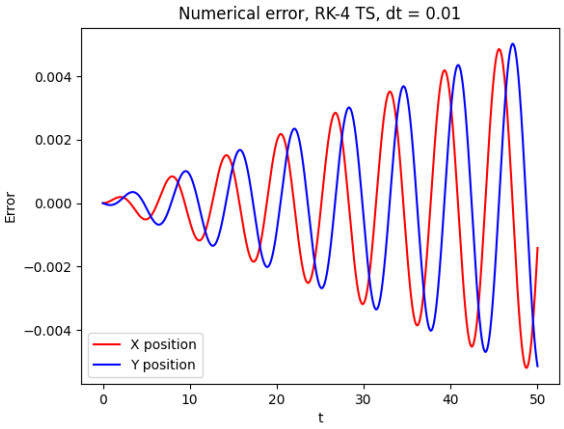
Por otro lado, si en vez de utilizar un método de Newton-Raphson para resolver la ecuación implícita del esquema Euler Inverso, se utiliza la función **fsolve** de Scipy, el error está más acotado. En la **Figura 2** se representa este fenómeno, para diferentes pasos de tiempo.



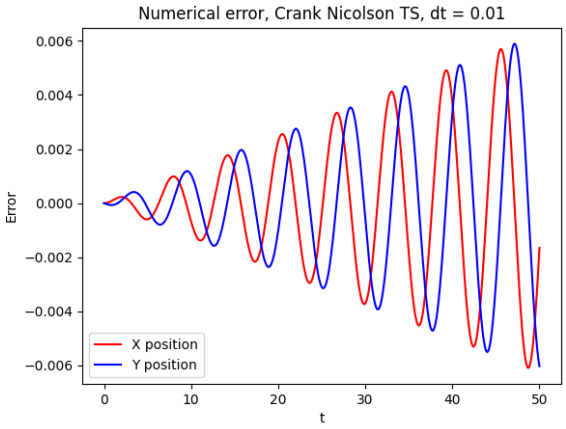
(a) Euler



(b) Inverse Euler

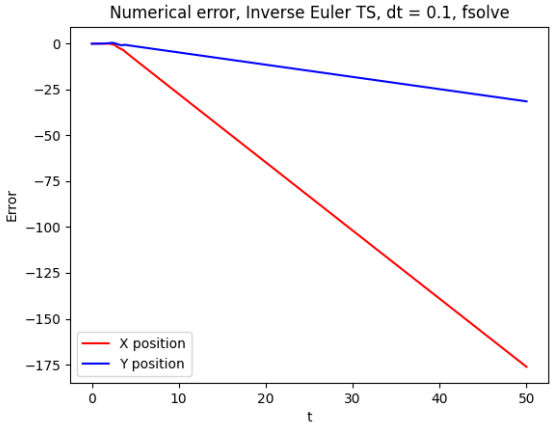


(c) Runge-Kutta-4

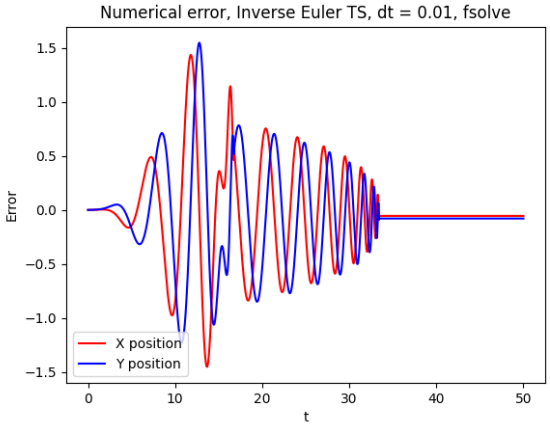


(d) Crank-Nicolson

Figura 1: Richardson extrapolation, Kepler problem



(a) dt = 0.1



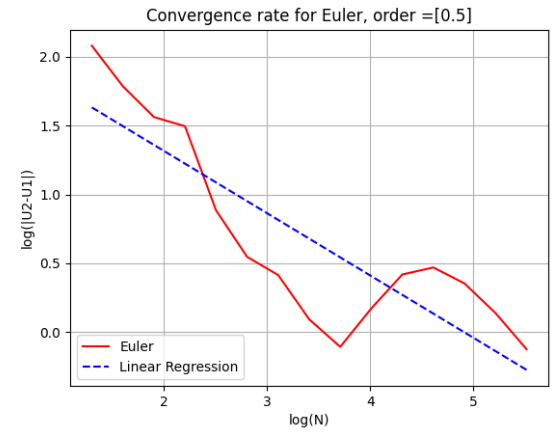
(b) dt = 0.01

Figura 2: Richardson extrapolation, Inverse Euler, fsolve resolution

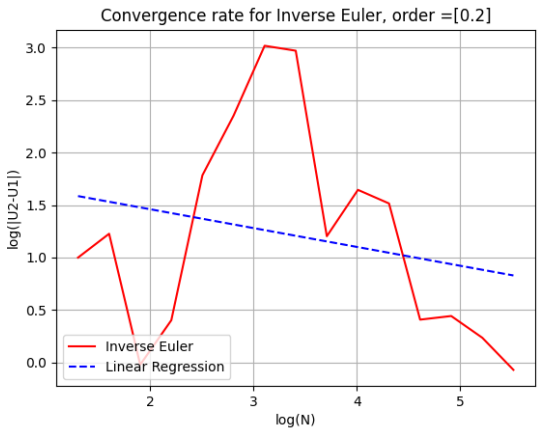
3. Ratio de convergencia

El ratio de convergencia nos permite obtener el orden de convergencia o integración de un esquema numérico. Esto es, a mayor orden, menor número de iteraciones serán necesarias para obtener una solución suficientemente precisa. Se realiza una regresión lineal de la gráfica que representa $\log(||U^{2N} - U^N||)$ (el error), para diferentes números de puntos de integración ($\log(N)$). La pendiente de dicha recta se corresponderá con el orden del esquema temporal. En la **Figura 3** se ha representado esto en los cuatro casos, así como su regresión lineal y el orden obtenido. Este orden se acerca mucho al real en el caso del Runge-Kutta-4 y el Crank-Nicolson, mientras que para el Euler y Euler Inverso sería necesario realizar una simulación más larga para obtener valores cercanos al 1.

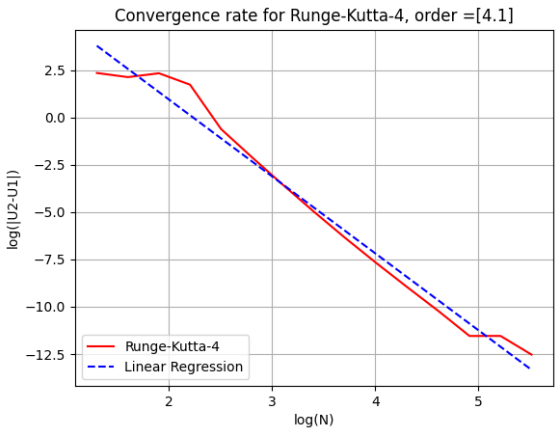
Cabe destacar que errores menores a un orden de 10^{-15} se corresponden a la precisión del propio ordenador, mientras que los errores para muy pocos puntos de integración derivan del propio método de integración.



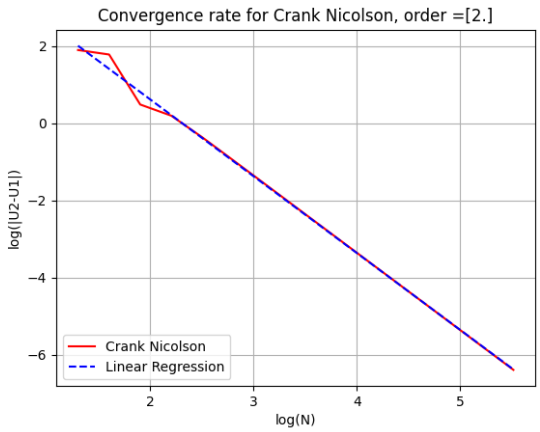
(a) Euler



(b) Inverse Euler



(c) Runge-Kutta-4



(d) Crank-Nicolson

Figura 3: Convergence rate, Kepler problem

4. Code Review

Primero, se ha implementado una factorización LU de matrices, mediante la cual se puede hallar su inversa. Esto se utiliza para realizar la inversa del Jacobiano en el método de Newton-Raphson, así no se depende de otras bibliotecas.

```
def factorization_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]) #Matriz diagonal superior

    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]) #Matriz
   diagonal inferior

  return [L@U, L, U]


def solve_LU(M,b):

  N=size(b)
  y=zeros(N)
  x=zeros(N)

  [A,L,U] = factorization_LU(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


def Inverse(A):

  N = size(A,1)
  B = zeros([N,N])

  for i in range(0,N):
```

```
    one = zeros(N)
    one[i] = 1

    B[:,i] = solve_LU(A, one)

  return B
```

Listing 1: LU factorization and Inverse

Por otro lado, se ha implementado dentro del módulo del problema de Cauchy tanto el método de estimación del error de Richardson como el ratio de convergencia.

```

def Error_Cauchy(F, t, U_0, Temporal_Scheme, order):

    N = size(t)
    E = zeros([N,size(U_0)])

    t1 = t
    t2 = linspace(0, t[N-1], N*2) #Malla refinada

    U2N = Cauchy_Problem(F, t2, U_0, Temporal_Scheme)
    U1N = Cauchy_Problem(F, t1, U_0, Temporal_Scheme)

    for i in range(0,N):
        E[i,:] = (U2N[2*i,:] - U1N[i,:]) / (1 - 1 / (2**order))

    return E


def Convergence_rate(F, t, U_0, Temporal_Scheme):

    N = size(t)

    t1 = t
    tf = t1[N-1]
    U1 = Cauchy_Problem(F, t1, U_0, Temporal_Scheme)

    k = 15 #Numero de evaluaciones para formar la recta, N*2^{k} puntos
   finales

    log_E = zeros(k)
    log_N = zeros(k)

    for i in range(0,k):

        N = 2*N
        t2 = linspace(0, tf, N)

        U2 = Cauchy_Problem(F, t2, U_0, Temporal_Scheme)

        E = norm((U2[int(N-1),:] - U1[int(N/2-1),:]))

        log_E[i] = log10(E)
        log_N[i] = log10(N)

        t1 = t2
```

```
45          U1 = U2
46
47      for j in range(0,k):
48
49          if (abs(log_E[j]) > 12): #Errores menores son de precision del
    ordenador
50
51              break
52
53      j = min(j, k)
54
55      reg = LinearRegression().fit(log_N[0:j+1].reshape((-1,
    1)),log_E[0:j+1]) #Regresi n lineal
56      order = round_(abs(reg.coef_),1) #Pendiente de la recta
57
58      log_N_lineal = log_N[0:j+1] #Almacena los valores de la recta para
    poder plotearla
59      log_E_lineal = reg.predict(log_N[0:j+1].reshape((-1, 1)))
60
61      return [log_N, log_E, order, log_N_lineal, log_E_lineal]
```

Listing 2: Richardson and Convergence Rate

Finalmente se ha creado un módulo desde donde se ejecuta el problema, de forma que solo es necesario llamar a la función del error o la convergencia para plotear las gráficas. Mediante esta estructura de módulos por funcionalidad se tiene el código mucho más escalonado, claro y compacto.