


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Milestone 3 Report

Ampliación de Matemáticas 1

10 de noviembre de 2022

Autores:

- Andrés García Serrano

Índice

1. Introducción

En este Milestone se busca realizar un análisis del ratio de convergencia de distintos esquemas y un análisis del error de los esquemas, utilizando el método de Richardson.

1.1. Uso del módulo

Al igual que en los Milestones anteriores, el módulo se puede acceder ejecutando el archivo menu.py situado en la carpeta raíz del proyecto.

La configuración de este módulo contiene los siguientes elementos:

- 1. start: Primer punto en el tiempo a integrar.
- 2. end: Último punto en el tiempo a integrar.
- 3. steps: Número de divisiones temporales.
- 4. X0: Coordenada X inicial.
- 5. Y0: Coordenada Y inicial.
- 6. VX0: Velocidad X inicial.
- 7. VY0: Velocidad Y inicial.
- 8. order: Orden de aproximación para el ratio de convergencia.

2. Análisis de código

Elementos importados

```
from matplotlib import pyplot as plt

from multiprocessing import Pool, cpu_count as ncpus

from numpy import array, empty, float as npfloat, log10, \
linspace, size, zeros, round_
from numpy.linalg import norm
from numpy.typing import ArrayLike

from sklearn.linear_model import LinearRegression

from typing import Callable, Tuple
```

2.1. Computación paralela

El cálculo de la velocidad de convergencia es un algoritmo de coste computacional $O(2^n)$, ya que el número de puntos se duplica por cada paso. Esto combinado con la naturaleza 'single-thread' de Python implica que el tiempo completo de cálculo será $\sum_{n=1}^{s} 2^n$ donde 's' es el orden de análisis de la velocidad de convergencia.

Para mitigar este coste, se paraleliza el cálculo de cada muestra, creando un proceso para cada dimensión de cálculo. Se crean una 'thread-pool' con tantos trabajadores como procesadores lógicos existen en la CPU. De esta manera se maximiza la utilización de la CPU sin sobrecargar el sistema.

Se paraleliza la función de la siguiente manera:

```
def Extract(X: Tuple[int, ArrayLike]) -> ArrayLike:
    return X[1]

class IndexFilter:
    def __init__(self, i):
        self.index = 2 ** i

    def __call__(self, x: Tuple[int, ArrayLike]) -> bool:
        return (x[0] % self.index) == 0

def ParCompute(U0: ArrayLike, t: ArrayLike, F: Callable,\
scheme: Callable, i: int) -> ArrayLike:
    result = enumerate( Cauchy(U0, t, F, scheme) )
    indexed = filter( IndexFilter(i), result )
    return array( list( map( Extract, indexed ) ) )
```

En la función 'ParCompute' se resuelve el problema de Cauchy para cada dimensión dada. De cada dimensión se extraen sólamente los puntos relevantes mediante un filtrado por índice 2^n. Debido a las restricciones de Python se ha utilizado un filtro customizado basado en una clase estática (IndexFilter) y una función de extracción de valores de la enumeración (Extract) estáticas para cumplir con los requisitos de paralelización de Python.

Estas limitaciones demuestran una de los mayores fallos de Python en cuanto al cálculo numérico. Los requisitos de paralelización de Python impiden una paralelización puramente funcional. En concreto, debido a que las funciones lambda de Python capturan variables del entorno, dejan de ser plenamente puras. Esta impureza impide su paralelización de forma 'thread-safe'.

Uno de los puntos de mejora que podría solucionar este problema y crear paralelización funcional es, en vez de denegar la paralelización a las lambdas, comprobar antes de paralelizar si estas lambdas han capturado variables 'inter-thread' o globales. Si esta condición no se cumple, comprobar si las variables capturadas van a moverse ('move by copy' con pérdida de 'ownership' del 'thread' original) junto con la lambda, en cuyo caso puede demostrarse que es 'thread-safe' hacer la paralelización funcional.

2.2. Ratio de Convergencia

La 'signature' de la función de cálculo de la velocidad de convergencia es la siguiente:

```
def Convergencia(U0: ArrayLike, t: ArrayLike, F: Callable,\
scheme: Callable, samples=5)
-> Tuple[float, ArrayLike, ArrayLike]:
```

Se crean todos los espacios temporales para todos los órdenes de cálculo seleccionados y se realiza la computación en paralelo. Se reduce el número base de puntos de cálculo a 100 para evitar órdenes altos de puntos, reduciendo el coste computacional.

```
# Calculate final time.
tf = (t[1] - t[0]) * 100

# Build the temporal linear spaces.
lin = lambda i: linspace(0, tf, (100 * (2 ** i)) + 1)
time = [lin for i in range(samples+1)]

# Build the multiprocessing pool.
pool = Pool(processes=ncpus())
threads = [pool.apply_async(ParCompute, (U0, time[i], F, scheme, i))\
for i in range(samples+1)]

# Collect the results.
U = [p.get() for p in threads]
```

Se generan los espacios $log_{10}(E)$ y $log_{10}(N)$ donde E es el error entre los resultados de cada orden y N es el logaritmo del número de puntos en cada orden.

Si el espacio 'logE' se reduce por debajo de 10^{-12} se eliminan el resto de puntos para evitar errores de precisión de máquina.

Tras este filtrado, se realiza una regresión linear para hallar la pendiente, que será la velocidad de convergencia.

```
# Build the log E and log N array.
logEfn = lambda U: log10( norm( U[i+1][-1] - U[i][-1] ) )
logE = array([logEfn(i) for i in range(samples)] )
logN = array([log10(100 * (2 ** i)) for i in range(samples)])

# Build the rate of convergence array.
for j in range(samples):
    if abs(logE[j]) > 12:
        break

j = min(j, samples)

logN = logN[0:j+1].reshape((-1,1))
```

```
    reg = LinearRegression().fit(logN, logE[0:j+1])
    q = round_(abs(reg.coef_), 1)

    return q, logE, logN
```

2.3. Error de Richardson

La 'signature' de la función de cálculo de la velocidad de convergencia es la siguiente:

```
def ErrorRichardson(U0:ArrayLike, t:ArrayLike, F:Callable,\
scheme:Callable, order: int)
-> Tuple[ArrayLike,ArrayLike,ArrayLike,ArrayLike]:
```

Se resuelve el problema de Cauchy en dos resoluciones, N y $2N$ y se filtran los resultados de $2N$ que no tienen un punto igual en N.

```
    # Extract base values.
    N = size(t)
    M = size(U0)

    # Build the temporal linear spaces.
    t1 = t
    t2 = linspace(0, t[-1], 2 * N)

    # Calculate the results vectors.
    U1 = Cauchy(U0, t1, F, scheme)
    UF = Cauchy(U0, t2, F, scheme)

    # Filter the unwanted U2 values.
    even = lambda X: X[0] % 2 == 0
    get = lambda X: X[1]
    U2 = array(list(map(get, filter(even, enumerate(UF)))))
```

Se calculan los errores absolutos por dimensión y normalizados y los errores relativos para las coordenadas y velocidad por dimensión y normalizados.

```
    # Richardson constant.
    K = 1.0 - ( 1.0 / (2.0 ** order) )

    # Calculate the raw vectorized delta.
    diff = lambda U: (U[0] - U[1])
    D = array( list( map(diff, zip( U2, U1 ) ) ) )

    # Calculate the vectorized error.
    A = array( list( map( lambda X: X / K, D ) ) )
```

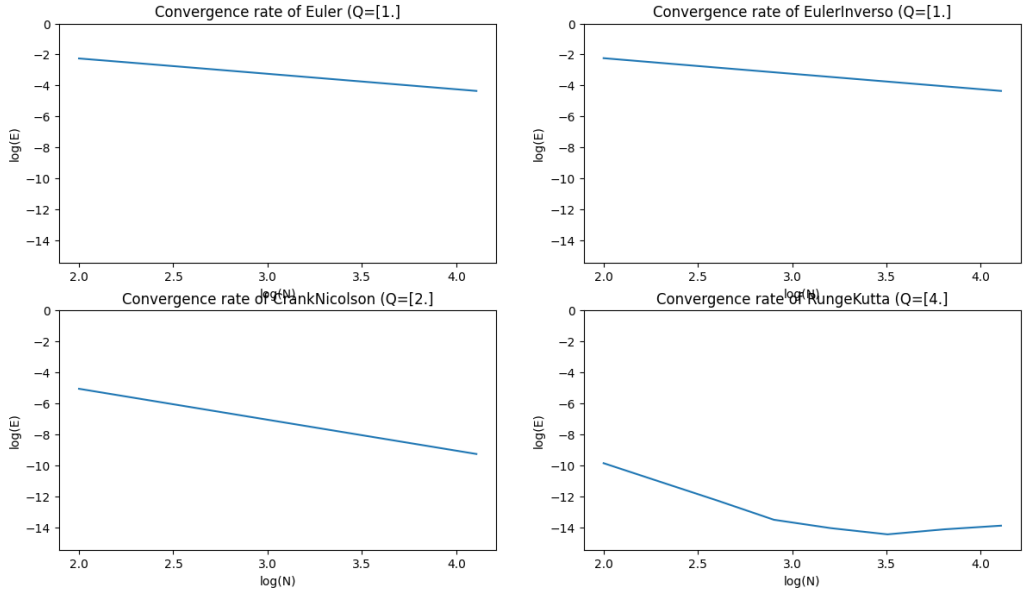
```
    # Calculate the scalar normalized error.
    E = array( list( map( lambda X: norm(X), A ) ) )

    # Calculate the vectorized relative error.
    dp = map( lambda X: norm(X[:2]), U2 )
    dv = map( lambda X: norm(X[2:]), U2 )

    ep = map( lambda X: norm(X[:2]), A )
    ev = map( lambda X: norm(X[2:]), A )

    V = empty([N, 2], dtype=npfloat)

    divide = lambda X: X[0] / X[1]

    V[:,0] = array( list( map( divide, zip(ep, dp) ) ) )
    V[:,1] = array( list( map( divide, zip(ev, dv) ) ) )

    # Calculate the scalar normalized relative error.
    R = array( list( map( lambda X: norm(X), V ) ) )

    return A, E, V, R
```

3. Resultados

3.1. Velocidad de convergencia

Como se puede ver en las figuras, el análisis de la velocidad de convergencia es altamente inestable para precisiones bajas, convergiendo a su valor real a partir de un número de puntos de orden $2^5 \sim 2^8$. Algunos esquemas vuelven a presentar inestabilidades de origen desconocido a órdenes de análisis 2^{12} 2^{15}.

En la convergencia de Runge-Kutta se observa un límite en la pendiente $log_{10}(E)/log_{10}(N)$ para los órdenes de precisión $O \sim 6$ que diverge. Aún así, se hayan resultados correctos.



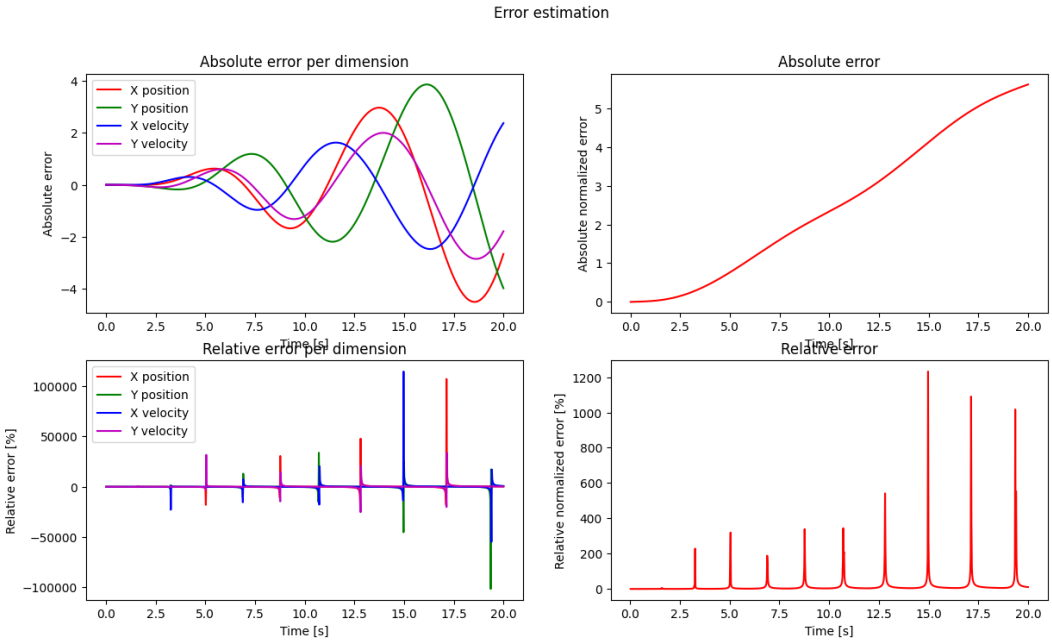
3.2. Error de Richardson

El error de los esquemas Euler y Euler inverso crece de forma acelerada por lo que no son esquemas viables para los problemas de estudio.

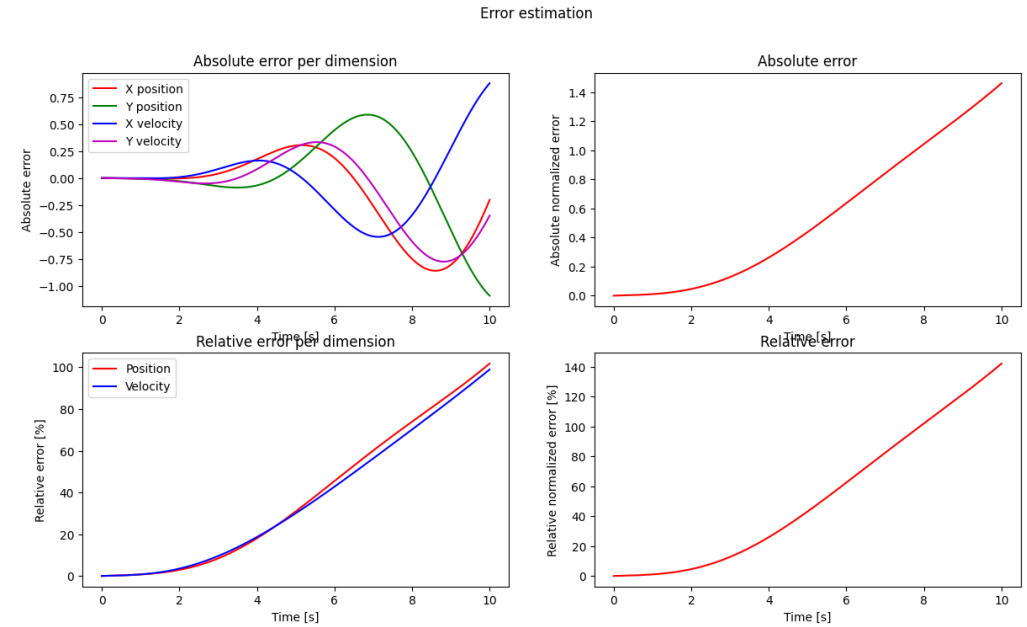
El error del esquema Crank-Nicolson es el menor de los sistemas estudiados, aunque su mayor coste computacional provoca que se tenga que evaluar su uso frente al esquema Runge-Kutta 4.

Resultados no filtrados

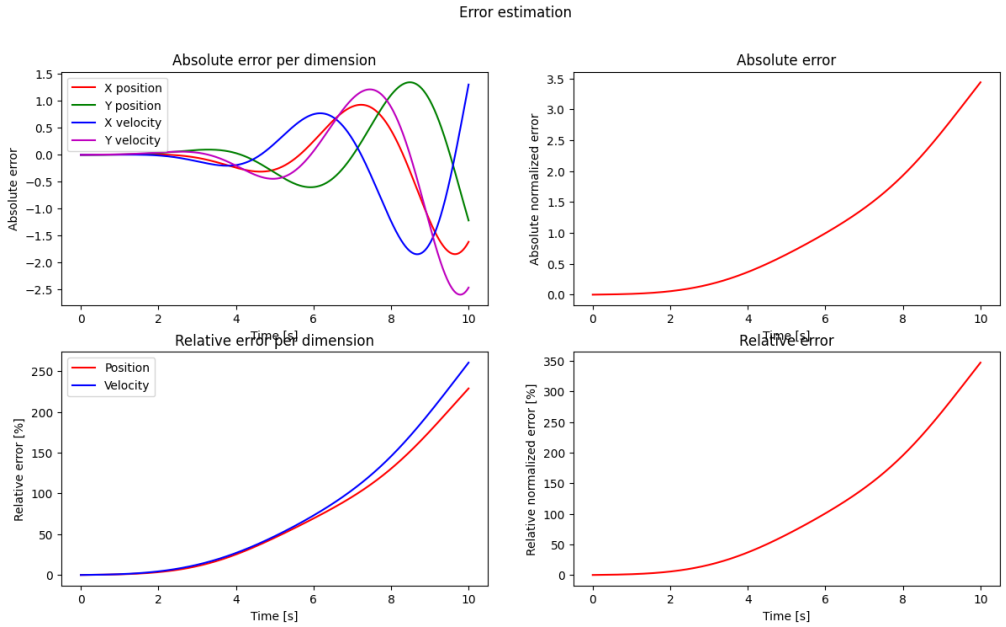
Si se toman los resultados directamente sin hacer ninguna transformación, se pueden encontrar singularidades cada vez que la posición X o la posición Y son cercanas a 0. Es por esto que los errores relativos se realizan con las coordenadas normalizadas, ya que el único sitio donde se encontraría la singularidad es en la posición $(0, 0)$, la cual es una singularidad física y el estudio de esta no es necesario.


Error estimation
Absolute error per dimension
X position
Y position
X velocity
Y velocity
Absolute error
Time [s]
Absolute error
Absolute normalized error
Time [s]
Relative error per dimension
Relative error [%]
Time [s]
Relative error
Relative normalized error [%]
Time [s]

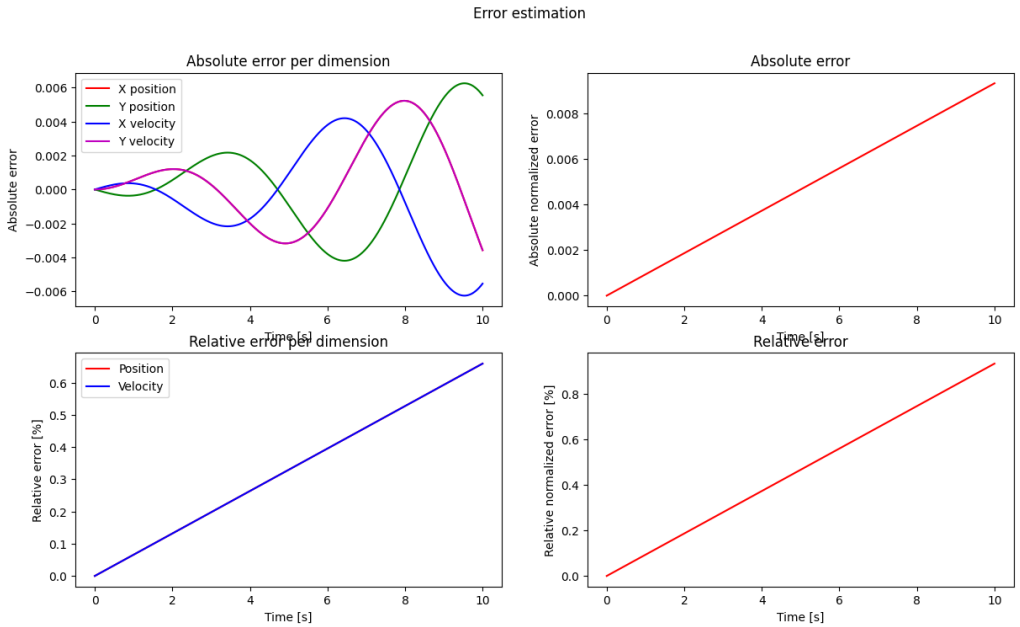
Euler


Error estimation
Absolute error per dimension
X position
Y position
X velocity
Y velocity
Absolute error
Time [s]
Absolute error
Absolute normalized error
Time [s]
Relative error per dimension
Position
Velocity
Relative error [%]
Time [s]
Relative error
Relative normalized error [%]
Time [s]

Euler inverso



Crank-Nicolson



Runge-Kutta 4