


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Análisis del Hito 2

Ampliación de Matemáticas I

23 de octubre de 2022

Tiempo de trabajo: 16 horas y 45 minutos.

Autor:

- Rafael Rivero de Nicolás

Índice

1. Introducción

En el presente informe se recogen las reflexiones y conclusiones extraídos de la realización del Hito 3 de la parte de Cálculo Numérico de la asignatura de Ampliación de Matemáticas I de MUSE. En dicho Hito se ha desarrollado un código en Python para extraer el orden de distintos esquemas numéricos comentados en clase y estimar posteriormente el error que se comete con ellos a lo largo del tiempo mediante la extrapolación de Richardson del error.

Se plantean dos Problemas de Cauchy, (1) y (2), que serán introducidos en el programa para analizar las diferencias que introduzca la variación del problema físico que se pretende resolver en los distintos esquemas numéricos. Ambos problemas de Cauchy están gobernados por ecuaciones diferenciales ordinarias lineales de segundo orden; vectorial en el primer caso y escalar de coeficientes constantes en el segundo.

$$\begin{cases} \ddot{\vec{\mathbf{r}}}(t) = -\dfrac{\vec{\mathbf{r}}(t)}{\|\vec{\mathbf{r}}(t)\|^3}, & \vec{\mathbf{r}} \in \mathbb{R}^2, t \in \mathbb{R} \\ \vec{\mathbf{r}}(t=0) = \vec{\mathbf{r}}_o = (1,0) \\ \dot{\vec{\mathbf{r}}}(t=0) = \dot{\vec{\mathbf{r}}}_o = (0,1) \end{cases} \tag{1}$$

$$\begin{cases} \ddot{x}(t) + x(t) = 0, & x, t \in \mathbb{R} \\ x(t=0) = x_o = 0{,}5 \\ \dot{x}(t=0) = \dot{x}_o = 0{,}5 \end{cases} \tag{2}$$

2. Estructura del código

2.1. Módulos necesarios

Las diversas tareas de cálculo han sido factorizadas en distintas funciones, permitiendo así la compartimentación del código, facilitando su lectura y su mantenimiento futuro y consiguiendo una estructura de programación modularizada. Por tanto, para el correcto funcionamiento del código desarrollado, `Milestone_III_Rafa.py`, es necesario importar varios módulos o librerías que se describen brevemente a continuación.

2.1.1. `LB_Math_Functions.py, LB_Temporal_Schemes.py`

Estas librerías fueron creadas para el Hito 2, por lo que son el punto de partida que nos permite resolver un Problema de Cauchy llamando a la función `Cauchy_Problem`, del módulo `LB_Math_Functions.py`, que a su vez selecciona el esquema numérico objetivo, cuyos algoritmos se encuentran recopiladas en `LB_Temporal_Schemes.py`.

2.1.2. LB_Physics_Problems.py

Los problemas de Cauchy planteados pueden escribirse como

$$\frac{\mathrm{d}U}{\mathrm{d}t} = F(U,t), \tag{3}$$

donde U es el vector de estado del problema y la función $F(U,t)$ depende del tipo de ecuación diferencial que se pretenda resolver.

Debido a que el problema que se pretende resolver no es siempre el mismo, en este caso solo se van a analizar dos ecuaciones diferenciales distintas pero podrían ser tantos como se quiera, y para darle más generalidad al código se ha creado un módulo llamado `LB_Physics_Problems.py` en el que se recogen todas las funciones F que se deseen implementar para poder analizar distintos problemas. Este módulo pretende ser un almacén de funciones que presenten problemas físicos que se pueden querer simular.

En este módulo también se encuentra una función que será llamada desde el código principal para elegir el operador adecuado en función del problema que se pretenda resolver, que será un input del programa principal, y los operadores disponibles en ese momento, la cual generará un error (warning) si se introduce un problema el cual no tenga una función asignada.

```
from numpy import array
import warnings

def Problem_Assignment(problem,Physics_Problems_available):

    if problem == Physics_Problems_available[0]:

        return Kepler_Orbits_2N

    elif problem == Physics_Problems_available[1]:

        return Undamped_Armonic_Oscilator

    else:

        warnings.warn('Introduce a valid problem equation to solve\n\t',
   Physics_Problems_available)

        return "ERROR"

def Kepler_Orbits_2N(X, t):

    return array([X[2], X[3], -X[0]/(X[0]**2 + X[1]**2)**(3/2), -X[1]/(X
   [0]**2 + X[1]**2)**(3/2)])


def Undamped_Armonic_Oscilator(X, t):

    return array([X[1], -X[0]])
```

Extracto de código 1: `LB_Physics_Problems.py`.

2.1.3. LB_Error_and_Convergence_rate_Problems.py

La primera función definida se recoge en el Extracto de código 2, la cual realiza los cálculos que permiten obtener una serie de puntos cuya representación permite obtener el orden de un esquema numérico dado. Esta función realiza el ajuste por mínimos cuadrados y genera una gráfica con los puntos obtenidos y su ajuste lineal. La gráfica generada por la siguiente función permite conocer el orden del esquema empleado para un problema físico dado y para un tiempo final dado.

```
def Convergence_Rate(Differential_operator, Initial_conditions, tf,
    temporal_scheme, M, Adjust = False, Save = False):

    t = {}; X = {}; log_DU = {}; log_N = {} # Dictionaries initialization

    if temporal_scheme == "Euler":
        dt = 0.00001*tf
    elif temporal_scheme == "Inverse Euler":
        dt = 0.001*tf
    elif temporal_scheme == "RK4":
        dt = 0.01*tf
    elif temporal_scheme == "Crank-Nicolson":
        dt = 0.01*tf

    if str(Differential_operator)[10:-23] == "Kepler_Orbits_2N":
        problem = "Kepler Orbits: 2 Bodies [2D]"
    elif str(Differential_operator)[10:-23] == "
    Undamped_Armonic_Oscilator":
        problem = "Undamped Armonic Oscilator [1D]"

    '''----- Computing based on Richardson extrapolation -----'''
    for i in range(M):

        if i == 0:
            t[i] = linspace(0, tf, int(tf/dt)+1)
        else:
            t[i] = linspace(t[0][0], t[0][-1], 2**i*(len(t[0])-1)+1)

        X[i] = mth.Cauchy_Problem( Differential_operator,
    Initial_conditions, t[i], Temporal_scheme = temporal_scheme )

        print(i)

    key_list = list(X.keys())

    for j in range( len(key_list)-1 ):

        log_DU[j] = log10( norm( X[key_list[j+1]][:,-1] - X[key_list[j
    ]][:,-1] ) ) # DU = ||Un - U2n||

        log_N[j] = log10( len( t[key_list[j]] ) )
    '''----------------PLotting----------------'''
    #...
```

Extracto de código 2: Función `Convergence_Rate`. No se incluye el código que permite la representación gráfica por simplicidad.

Los argumentos de entrada son:

- `Differential_operator`: Operador diferencial, denotado como F en 3, correspondiente al Problema de Cuachy seleccionado.
- `Initial_conditions`: Condiciones iniciales del Problema de Cauchy seleccionado.
- `tf`: Tiempo final de simulación en el que se calculará el error estimado.
- `temporal_scheme`: Esquema temporal elegido para la integración numérica.
- `M`: Número de puntos que se pretenden calcular para su posterior representación.
- `Adjust`: Argumento de tipo booleano. De ser verdadero se realizará el ajuste por mínimos cuadrados de los puntos calculados a un modelo lineal. De forma predeterminada es falso.
- `Save`: Argumento de tipo booleano. De ser verdadero hará que la figura generada se guarde automáticamente. De forma predeterminada es falso.

Destaca el hecho de que se ha definido un Δt en función del tiempo de simulación y del esquema empleado que permite que el ajuste a una función lineal se haga en la zona recta de la gráfica. Esto se ha comentado más a fondo en la sección 3.

Mientras, la función `Richardson_Error_Extrapolation`, recogida en el Extracto de código 3 presenta un argumento ligeramente distinto a la función anterior:

- `time_domain`: Malla temporal empleada, no tiene por qué ser equiespaciada o tener una dimensión fija.

Al principio de esta función, mediante una serie de condicionales se asigna un orden al esquema elegido según los resultados que se han obtenido previamente de la función anterior. El orden y el proceso seguido para la realización de estos cálculos se comenta en la Sección 3.

2.2. Programa principal: `Milestone_III_Rafa.py`

El programa principal es un código compacto gracias a la factoración en funciones y a la modularización, representado en Extracto de código 4.

Los parámetros que se pueden elegir en este código son `k`, para seleccionar el esquema numérico empleado y el parámetro `P` para seleccionar el problema físico deseado.

Tras la elección de parámetros se procede a la asignación del operador diferencial y las condiciones iniciales, se llama a las funciones definidas previamente, las cuales se encargan de la representación gráfica de resultados.

```
def Richardson_Error_Extrapolation(Differential_operator,
    Initial_conditions, time_domain, temporal_scheme, Save=False):

    if temporal_scheme == "Euler":
        scheme_order = 1
    elif temporal_scheme == "Inverse Euler":
        scheme_order = 1
    elif temporal_scheme == "RK4":
        scheme_order = 4
    elif temporal_scheme == "Crank-Nicolson":
        scheme_order = 2

    if str(Differential_operator)[10:-23] == "Kepler_Orbits_2N":
        problem = "Kepler Orbits: 2 Bodies [2D]"
    elif str(Differential_operator)[10:-23] == "
    Undamped_Armonic_Oscilator":
        problem = "Undamped Armonic Oscilator [1D]"

    t1 = time_domain; dt = t1[1]-t1[0]; tf = t1[-1]
    t2 = linspace(t1[0], t1[-1], 2*(len(t1)-1)+1)

    X_1 = mth.Cauchy_Problem( Differential_operator, Initial_conditions,
    t1, Temporal_scheme = temporal_scheme )
    X_2 = mth.Cauchy_Problem( Differential_operator, Initial_conditions,
    t2, Temporal_scheme = temporal_scheme )

    Richardson_Error = zeros(len(t1))

    for i in range(0,len(t1)):

        Richardson_Error[i] = norm( X_2[:,2*i] - X_1[:,i] ) / ( 1- ( 1 /
    2**scheme_order ) )

        #...
```

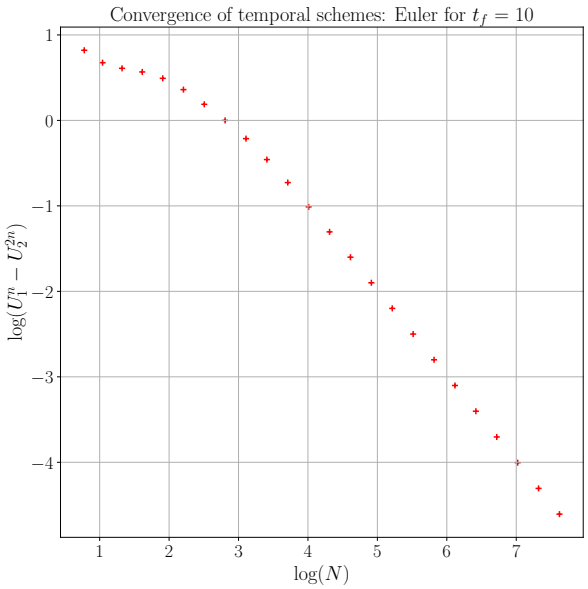
Extracto de código 3: Función `Richardson_Error_Extrapolation`. No se incluye el código que permite la representación gráfica por simplicidad.

```
from numpy import array, linspace, hstack
import LB_Error_and_Convergence_rate as LB_erc # User's Module
import LB_Physics_Problems as ph # User's module

# %% Initialitation

Temporal_schemes_available = {0:"Euler",
                              1:"Inverse Euler",
                              2:"RK4",
                              3:"Crank-Nicolson"}

Physics_Problems_available = {0:"Kepler Orbits: 2 Bodies [2D]",
                              1:"Undamped Armonic Oscilator [1D]"}

'''\\\\\\\\\\\\\\\\\\\\\\\'''
k = 2 # Selection of Numeric Scheme 0-1-2-3
P = 0 # 0-1
tf = 10

if P == 0:
    r_0 = array([1, 0]); v_0 = array([0, 1]) # Initial position and
    velocity, respectively. [Kepler Orbits 2 Bodies [2D]]
elif P == 1:
    r_0 = 0.5; v_0 = 0.5 # Initial position and velocity, respectively. [
    Undamped Armonic Oscilator [1D]]
'''\\\\\\\\\\\\\\\\\\\\\\\'''

Initial_conditions = hstack((r_0,v_0)); print('Initial State Vector: U_0
    = ', Initial_conditions, '\n\n\n')

Differential_operator = ph.Problem_Assignment(Physics_Problems_available[
    P],Physics_Problems_available)

physics_problem = Physics_Problems_available[P]
scheme = Temporal_schemes_available[k]

# %% Convergence Rate

M = 9 # Number of points to compute q

LB_erc.Convergence_Rate(Differential_operator = Differential_operator,
    Initial_conditions = Initial_conditions,
                    tf = tf, temporal_scheme = scheme, M = M, Adjust = True
    , Save = False)

# %% Richardson Extrapolation to compute Error

dt_R = 0.01

time_domain_Richardson = linspace(0, tf, int(tf/dt_R)+1)

LB_erc.Richardson_Error_Extrapolation(Differential_operator,
    Initial_conditions, time_domain_Richardson, scheme, Save = False)
```

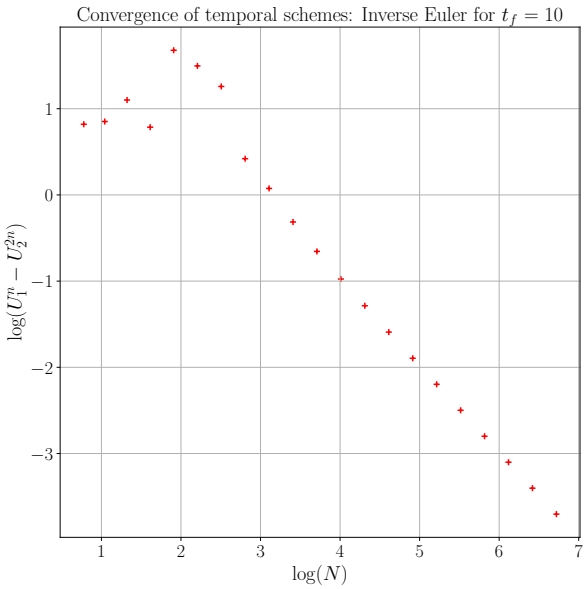
Extracto de código 4: `Milestone_III_Rafa`

3. Evolución del Código

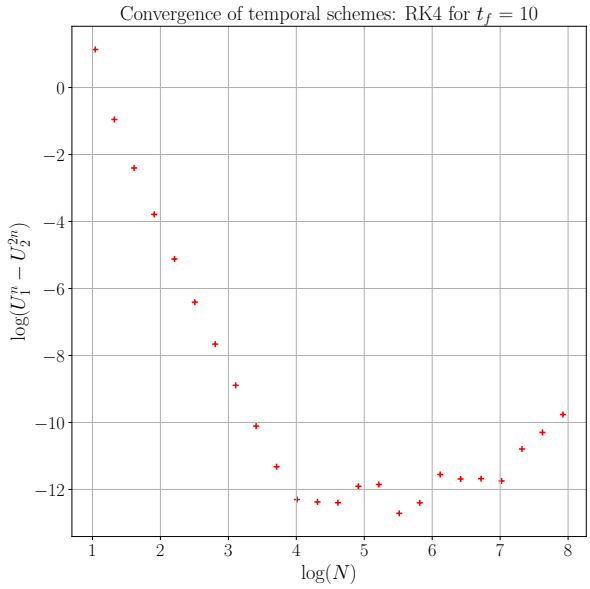
Primero se realizó todo el código sin compartimentar, en el mismo archivo `.py`. Una vez se hubo verificado el funcionamiento del programa para distintos esquemas y para distintos problemas físicos se empezó a compartimentar en módulos.



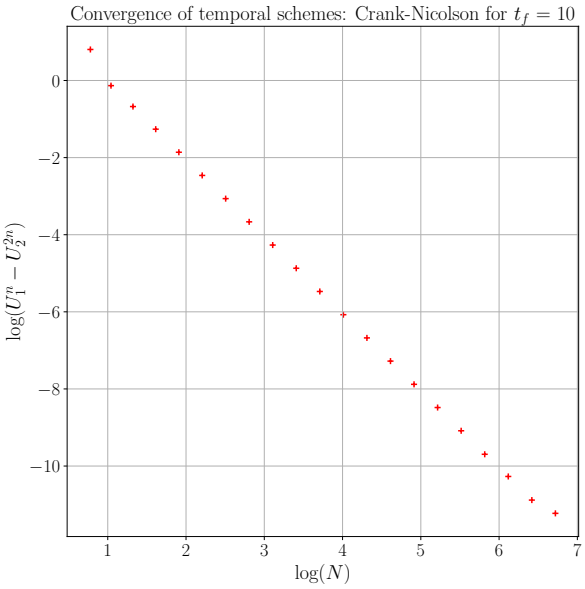
(a) **Esquema Euler**



(b) **Esquema Euler Inverso**



(c) **Esquema Runge Kutta de 4º orden**



(d) **Esquema Crank-Nicolson**

Figura 1: Barrido de la estimación del error para distintos esquemas numéricos aplicados al problema (1) y $t_f = 10$.

Previamente a la realización del ajuste que permite conocer el orden de los distintos esquemas numéricos planteados se ha realizado un barrido que ha generado las gráficas recogidas en la Figura 1 para poder identificar la zona lineal de las gráficas y poder así

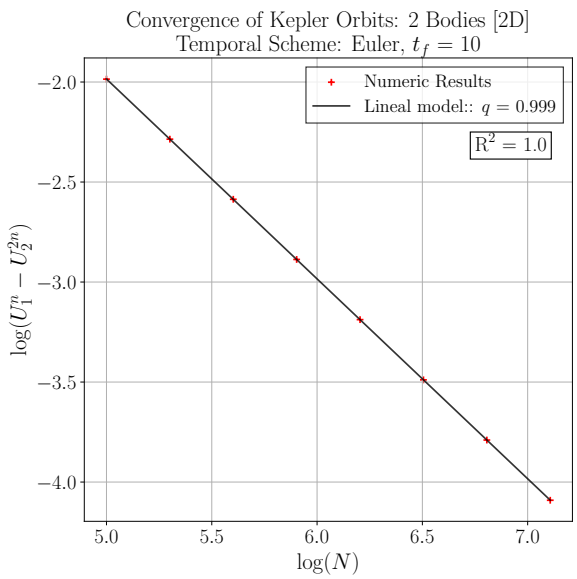
ajustar el Δt empleado en las simulaciones para asegurarse de que el ajuste por mínimos cuadrados a una función lineal se ha realizado en la zona no problemática de las gráficas. Este proceso de barrido ha llevado un total de más de 10 horas para la obtención de estas 4 gráficas.

Las mejoras sustanciales respecto del Hito 2 han sido la inclusión del módulo que permite almacenar cuantas ecuaciones diferenciales se quieran, `LB_Physics_Problems.py`, sin que estas ocupen espacio del código principal y el hecho de permitir que el usuario elija de un diccionario con las opciones (tanto del esquema numérico como de la ecuación diferencial) la key correspondiente y poder probar múltiples combinaciones solamente cambiando un par de números, haciendo el programa más ágil.

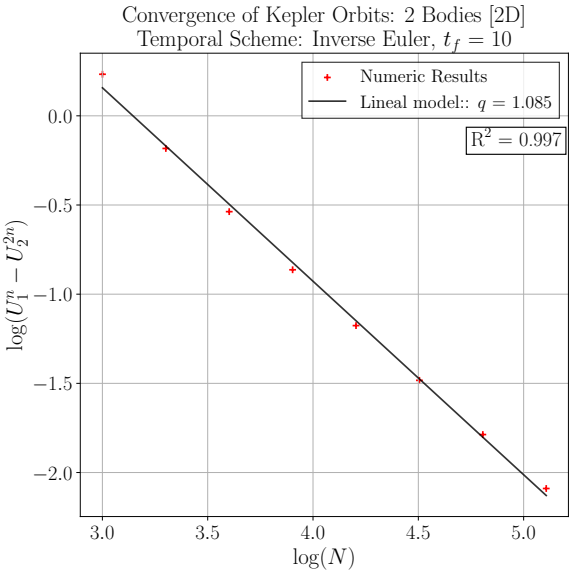
Para futuros códigos se reutilizarán estas mejoras y se propone compartimentar la representación gráfica generando un módulo cuyas funciones se encarguen de la generación de figuras y así no tener que repetir continuadamente el bloque de código correspondiente.

4. Resultados

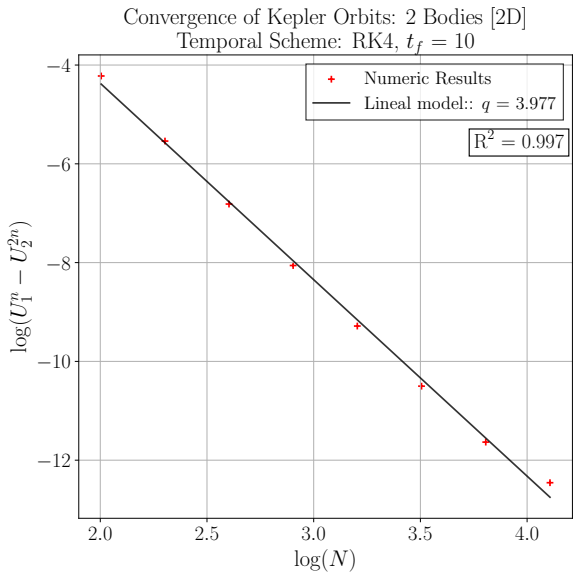
En la Figura 2 se representa la aproximación realizada por mínimos cuadrados que permite obtener el orden del esquema correspondiente para el problema 1 y $t_f = 10$. La pendiente de la curva es la esperada, presentando valores realmente próximos a los vistos en las clases de teoría de la asignatura. Destaca el gran tiempo de simulación requerido para los esquemas Euler Inverso y Crank-Nicolson, requiriendo del orden de minutos. El tiempo de cálculo aumenta mucho más rápido que en los esquemas no implícitos, en los que los cálculos tardan del orden de segundos.



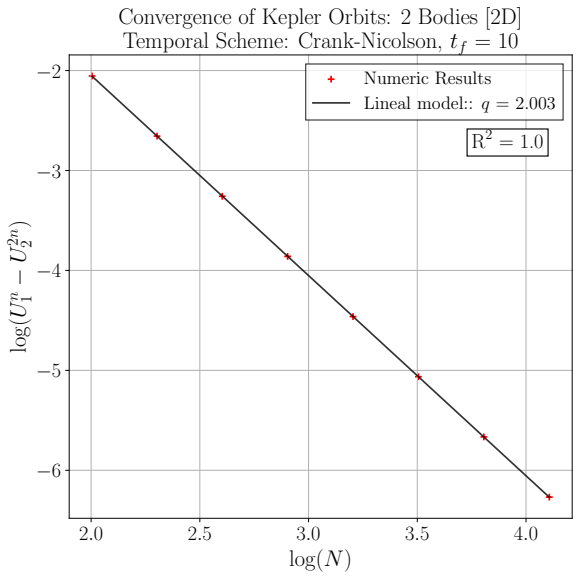
(a) **Esquema Euler**



(b) **Esquema Euler Inverso**



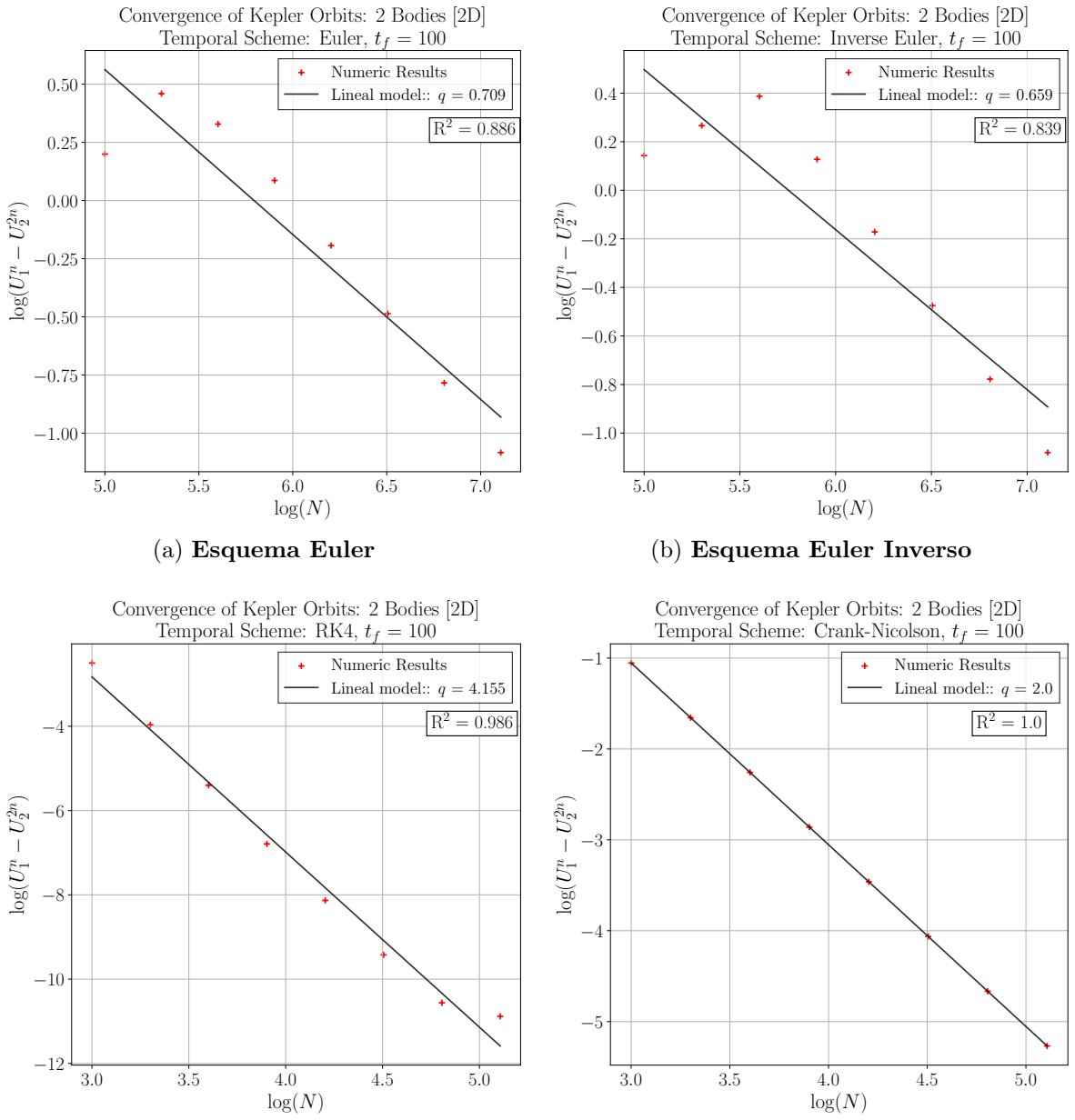
(c) **Esquema Runge Kutta de 4º orden**



(d) **Esquema Crank-Nicolson**

Figura 2: Obtención del orden del esquema numérico para el problema (1) y $t_f = 10$.

En la Figura 3 se ha representado el resultado obtenido con $t_f = 100$. Para Runge Kutta de 4º orden y Crank-Nicolson se obtienen valores de q muy cercanos a los teóricos, sin embargo, los esquemas Euler y Euler Inverso no convergen lo suficientemente rápido y a pesar de que para este último esquema el tiempo de simulación fue de 6 horas y 13 minutos ni siquiera se llega a la zona lineal de la gráfica, lo que desvirtúa la aproximación realizada.



(a) **Esquema Euler**

(b) **Esquema Euler Inverso**

(c) **Esquema Runge Kutta de 4º orden**

(d) **Esquema Crank-Nicolson**

Figura 3: Obtención del orden del esquema numérico para el problema (1) y $t_f = 100$.

Mientras, en el esquema Runge Kutta de 4º orden se llega a la zona problemática a la derecha del tramo lineal dado que se alcanzan errores del orden del error de redondeo.

También es curioso el hecho de que aunque el Δt empleado depende del tiempo final introducido en el programa principal de forma que N sea independiente de t_f,

$$\Delta t = a t_f, \quad N = \frac{t_f}{\Delta t} = f(a),$$

y dependa por tanto solo del esquema numérico seleccionado, el tiempo asociado al proceso de cálculo sí ha crecido considerablemente al aumentar t_f. La explicación posiblemente se deba a que al aumentar t_f el mallado temporal es menos exhaustivo, lo que resulta en un mayor número de iteraciones para la resolución de los esquemas implícitos, ya que la condición inicial estará cada vez más lejana a la solución real y cada iteración se debe invertir la matriz Jacobiana, lo que es muy costoso computacionalmente.

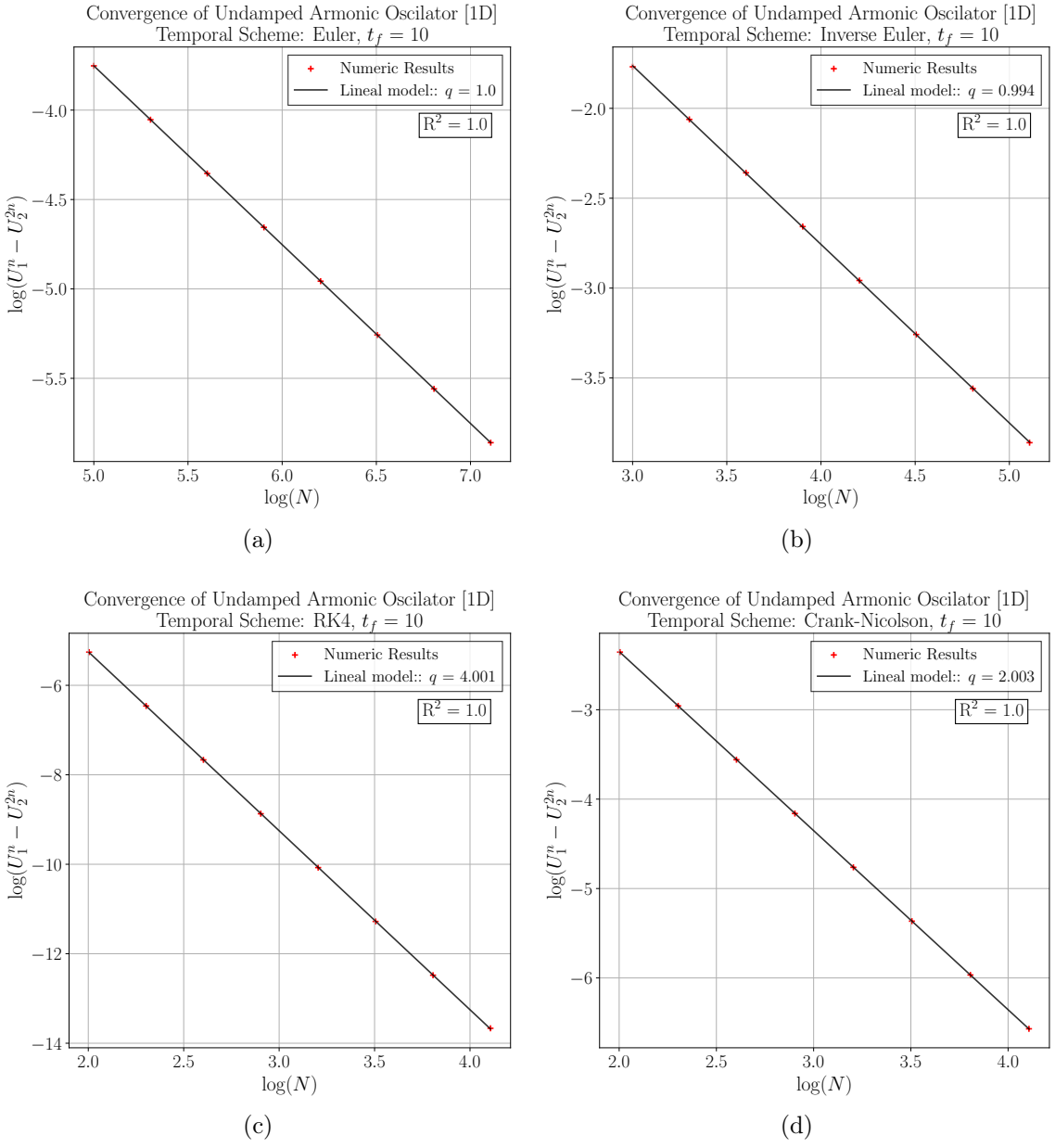


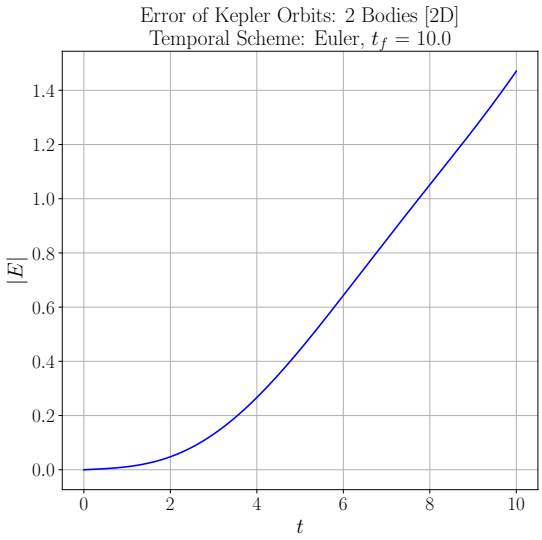
Figura 4: Obtención del orden del esquema numérico para el problema (2) y $t_f = 10$.

En la Figura 4 se ha introducido un Problema de Cauchy o de Valores Iniciales distinto, lo que no ha introducido cambios sustanciales en los órdenes de los esquemas estudiados respecto a la Figura 2.

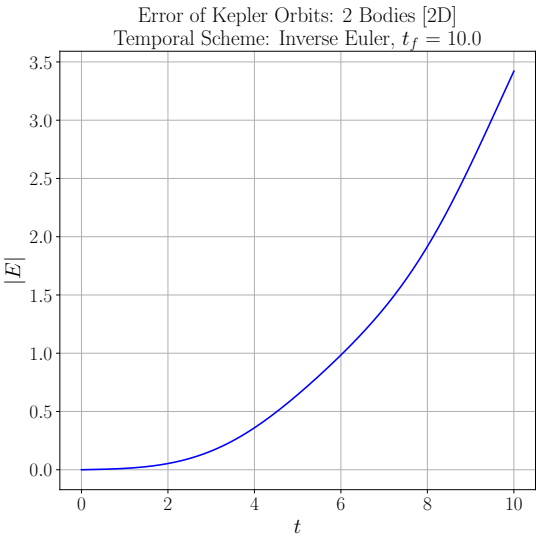
En la Figura 4 se recoge la evolución temporal de la norma del error estimado mediante la extrapolación de Richardson,

$$|E| = \frac{U_1^i - U_2^{2i}}{1 - 2^{-q}}, \tag{4}$$

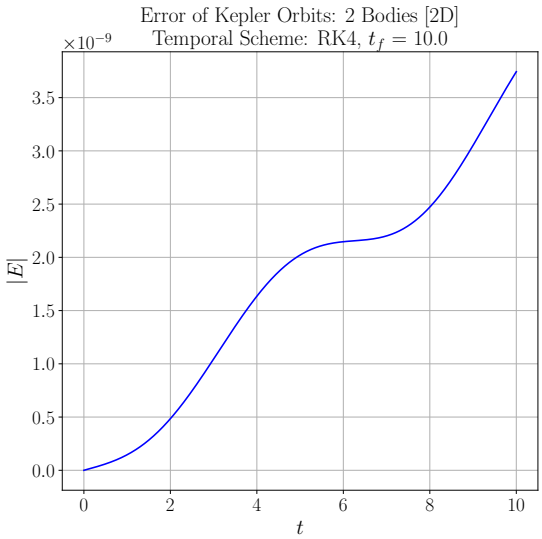
y se aprecia que a rasgos generales este error va creciendo según avanza el tiempo final de simulación, aunque el error como tal es muchos órdenes de magnitud inferior en los esquemas de mayor orden.



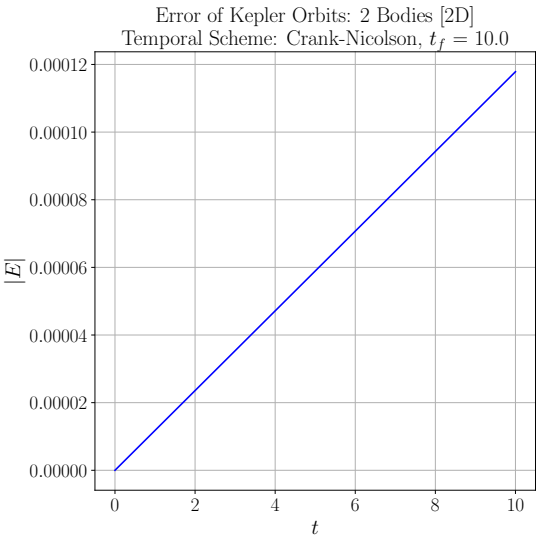
(a) **Esquema Euler**



(b) **Esquema Euler Inverso**



(c) **Esquema Runge Kutta de 4º orden**



(d) **Esquema Crank-Nicolson**

Figura 5: Módulo del error obtenido por extrapolación de Richardson en función del tiempo para distintos esquemas numéricos aplicados al problema (1) y $t_f = 10$.

5. Conclusiones

Se ha definido en este informe el código desarrollado para el Hito 3 y se ha obtenido el orden de distintos esquemas numéricos, obteniendo resultados muy similares a los teóricos. Se ha comprobado que el cambiar el tiempo final de la simulación puede afectar a la zona de la gráfica representada, que en caso de no corresponderse con el tramo lineal provocará una alteración de la obtención del orden del esquema, además de al tiempo de cálculo necesario previo a la representación gráfica.

Destaca el hecho de que los esquemas con un orden elevado necesitan un rango de valores de N mucho menor que los esquemas de orden 1, Euler y Euler Inverso, que abarcan valores varios órdenes de magnitud superiores.

En concreto, el esquema Euler inverso, al necesitar un valor elevado de N y ser además un esquema temporal implícito, los tiempos de cálculo han sido mucho más superiores al resto de esquemas para un mismo tiempo final. En concreto, la obtención de la Figura 3b ha tenido un tiempo asociado de computación de 6 horas y 13 minutos, mientras que la Figura 3c se ha obtenido en menos de 2 minutos.

También destaca el hecho de los valores obtenidos en el eje vertical para los esquemas de orden superior a 1, Runge Kutta de 4^{o} orden y Crank-Nicolson, son muy inferiores a los obtenidos con esquemas de órdenes más reducidos. En la Figura 1 se aprecia como el esquema de mayor orden, Runge Kutta, alcanza la región en la que el error de redondeo juega un papel importante distorsionando el valor representado de $\log(U_1^n - U_2^{2n})$ para $N > 4$.

Para concluir, el hecho de haber obtenido un método que permita obtener el orden del esquema numérico en función del tiempo de simulación y el problema planteado es una herramienta de gran utilidad ya que esto permite estimar el error cometido en cada instante de tiempo. Esto es una gran ventaja a la hora de resolver problemas en los que no se disponga de solución analítica ya que a pesar de esto se puede realizar un análisis que permita confiar o no en la solución numérica.

Referencias

[1] Hernández, Juan Antonio, *Cálculo Numérico en Ecuaciones Diferenciales Ordinarias*, 2018.

[2] Hernández, Juan Antonio & Escoto, F. Javier *How to learn Applied Mathematics through modern FORTRAN*, 2017.