


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Milestone II Report

Ampliación de Matemáticas I

3 de octubre de 2022

Autora:

- Rosa Martínez Rubiella

1. Introducción

En este segundo Hito, se añaden a los esquemas temporales explícitos utilizados para resolver el problema de la órbita Kepleriana dos de carácter implícito. Tanto el método de **Crank-Nicolson** como el **Euler Inverso** requieren resolver una ecuación que contiene el paso de tiempo siguiente en dos términos diferentes de la misma. Para ello, se utiliza primeramente la función **fsolve** de la biblioteca Numpy. A continuación, se introduce un método de **Newton-Raphson** y se compara con el que proporciona el paquete de Scipy.

En la última parte del documento se comenta la estrategia **Top-Down** seguida. Es decir, se crean funciones independientes y escalonadas tanto para los esquemas temporales como para la función de fuerza de Kepler, el problema de Cauchy y el esquema de Newton-Raphson. Esto nos permite tener un código más claro y funcional.

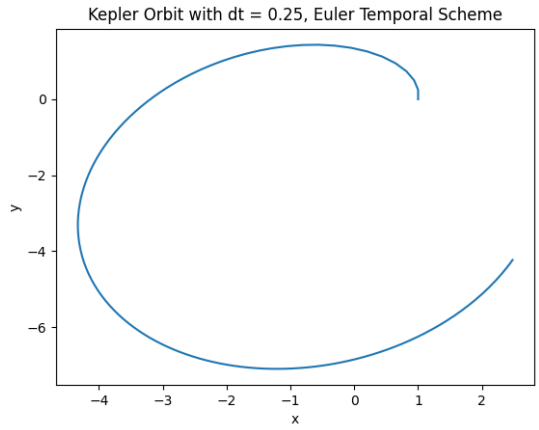
2. Resultados usando fsolve

De forma análoga al documento anterior, se utiliza un tiempo de simulación total de 50 segundos para todos los esquemas temporales. De esta forma se pretende realizar una comparación para diferentes intervalos de integración **dt** entre los diferentes esquemas numéricos. El método de Crank-Nicolson es de orden 2, por lo que se prevén resultados con un error menor a los del Euler. Además, se añade al final del documento una comparación de las energías específicas de estos métodos para cada caso, ya que siempre deberían conservarse. (**Figura 5**)

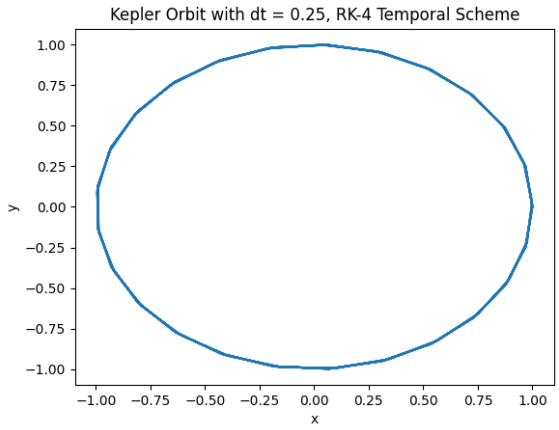
2.1. N = 200

En primer lugar analizamos los resultados para un paso de integración bastante grande ($dt = 0{,}25$), es decir, con un número reducido de puntos de integración. Mientras que la aproximación que realiza el Crank-Nicolson parece bastante similar a la solución analítica (al igual que para el Runge-Kutta), el método de Euler Inverso tiene un comportamiento bastante errático durante los primeros pasos de integración ya que simula una órbita de tamaño mitad a la buscada (pierde energía rápidamente y, a continuación, se conserva con pequeñas fluctuaciones). (**Figura 1**)

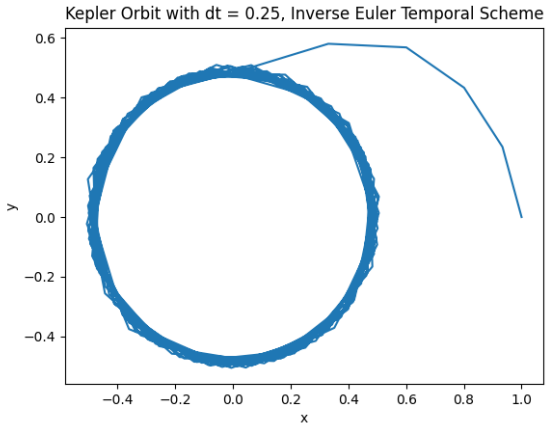
Como ya se comentó en el primer documento y se puede observar en la **Figura 5**, la energía específica de los métodos de Euler no se mantiene constante, si no que diverge mucho en los primeros momentos de la simulación. Sin embargo, tanto el método de Runge-Kutta como el de Crank-Nicolson mantienen dicha energía constante, son conservativos.



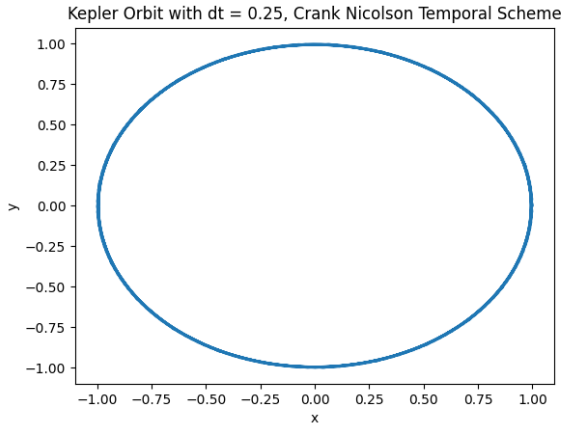
(a) Euler



(b) Runge-Kutta 4



(c) Inverse Euler



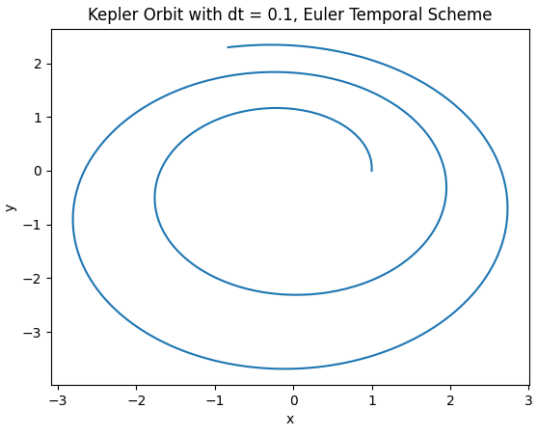
(d) Crank-Nicolson

Figura 1: Kepler Problem, N= 200, t= 50

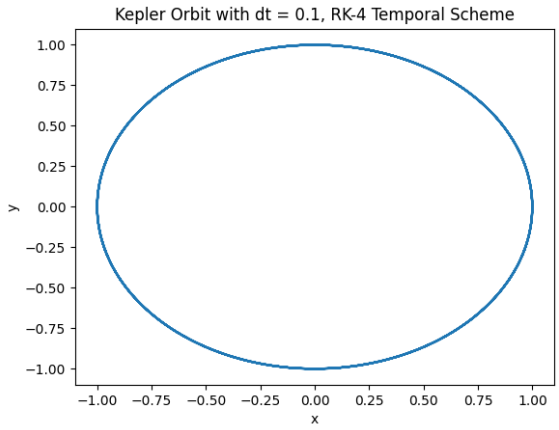
2.2. N = 500

Para el caso de pasos de integración de amplitud $dt = 0{,}1$, el método de Euler Inverso presenta una tendencia a infinito. Esto se produce porque la solución converge a (0,0) en un punto de integración, por lo que aparecen indeterminaciones que tienden a infinito en el siguiente paso (**Figura 2**). Por otro lado, en la **Figura 5** se puede apreciar el pico que produce la energía específica de la órbita en dicho instante y su posterior asíntota a valor nulo.

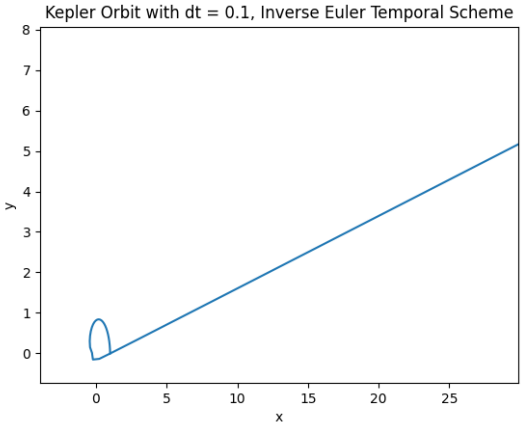
En cuanto al resto de métodos, todos mantienen el comportamiento esperado, ya analizado con anterioridad.



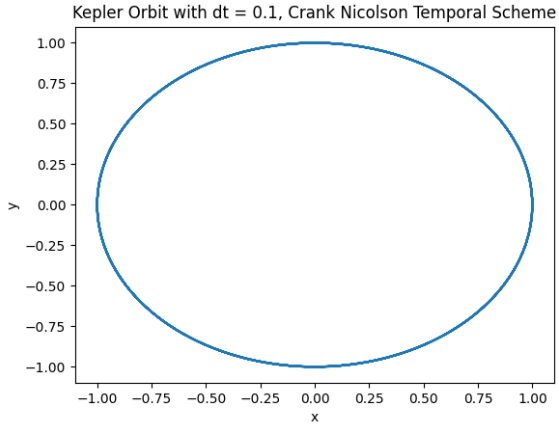
(a) Euler



(b) Runge-Kutta 4



(c) Inverse Euler

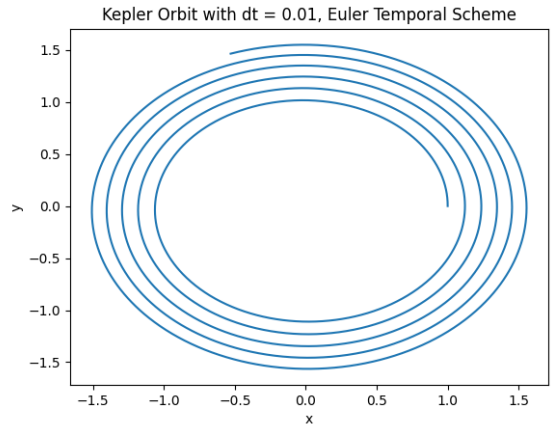


(d) Crank-Nicolson

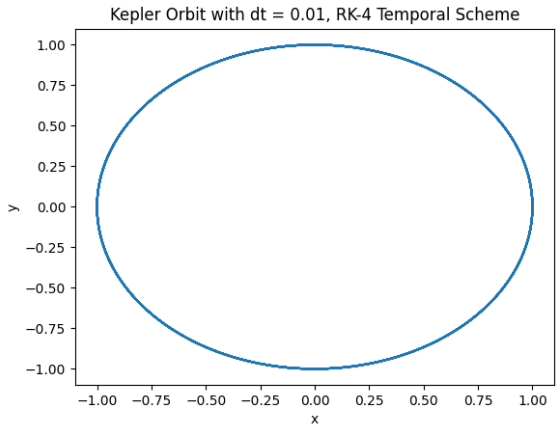
Figura 2: Kepler Problem, N= 500, t= 50

2.3. N = 5000

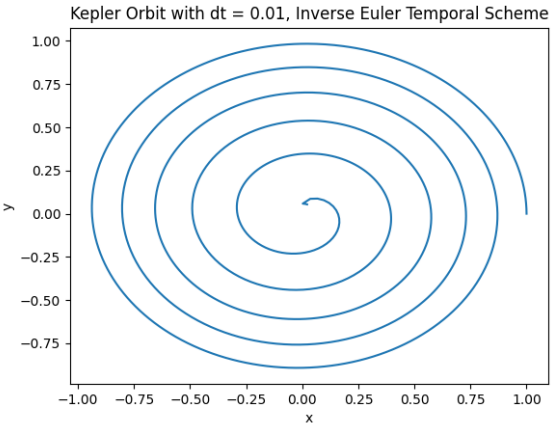
En este caso, mientras que el método de Crank-Nicolson mantiene una muy buena precisión de aproximación respecto a la solución analítica, el Euler-Inverso pierde energía de manera casi-lineal hasta caer bruscamente en las inmediaciones de la posición (0,0). Es esta región la iteración no prospera de manera adecuada, y queda estancada. (**Figura 3**)



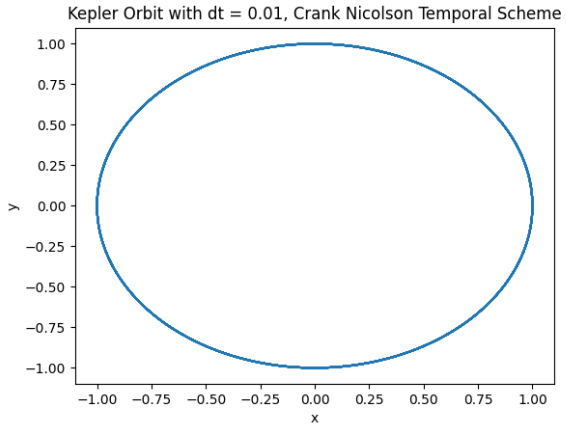
(a) Euler



(b) Runge-Kutta 4



(c) Inverse Euler



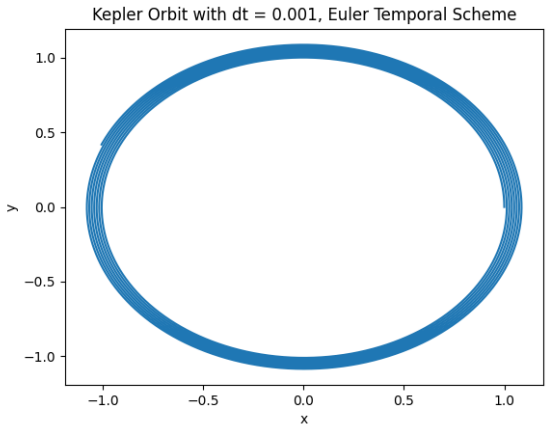
(d) Crank-Nicolson

Figura 3: Kepler Problem, N= 5000, t= 50

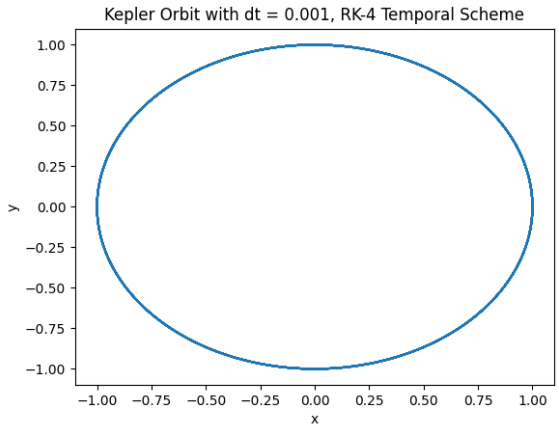
2.4. N = 50000

Por último, se incrementa todavía más el número de puntos para poder alcanza una solución más precisa del método de Euler Inverso. En este caso, el paso de integración es lo suficientemente pequeño como para poder simular una órbita más coherente. Para el número de puntos seleccionado, tanto el Euler explícito como el implícito tienen una solución aproximada parecida, mientras que las soluciones de el Runge-Kutta y Crank-Nicolson se mantienen prácticamente iguales. (**Figura 4**)

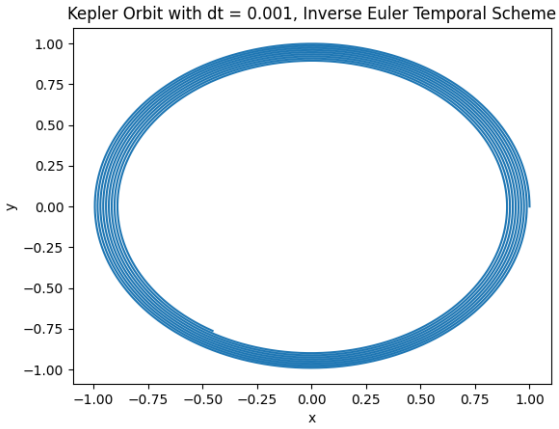
De la **Figura 5** se observa cómo las energías específicas de estas órbitas son mucho más constantes que en casos anteriores (escala del eje menor).



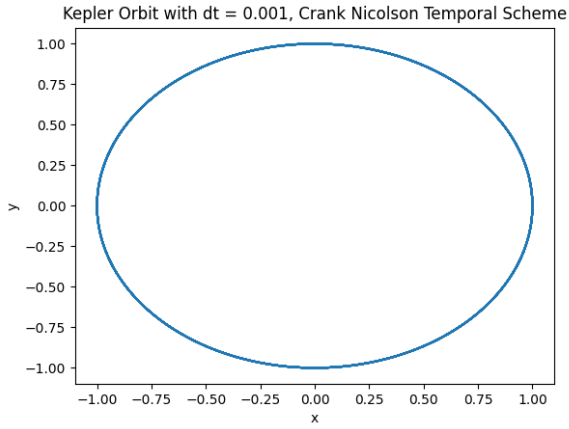
(a) Euler



(b) Runge-Kutta 4

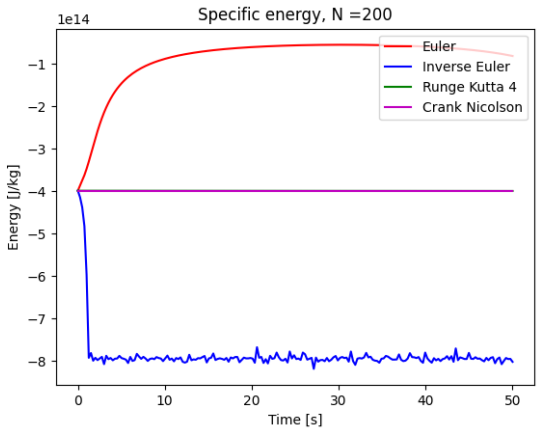


(c) Inverse Euler

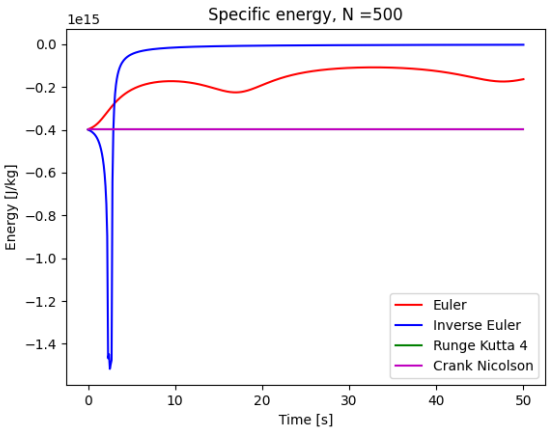


(d) Crank-Nicolson

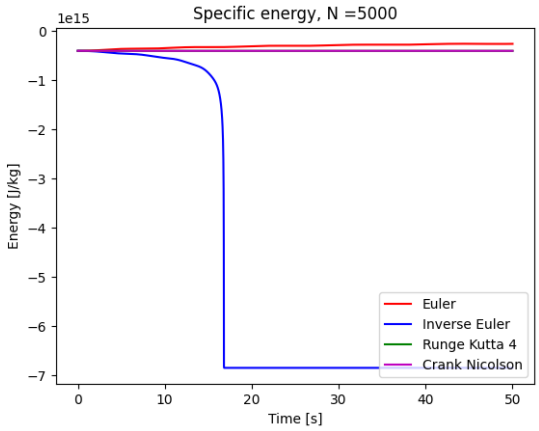
Figura 4: Kepler Problem, N= 50000, t= 50



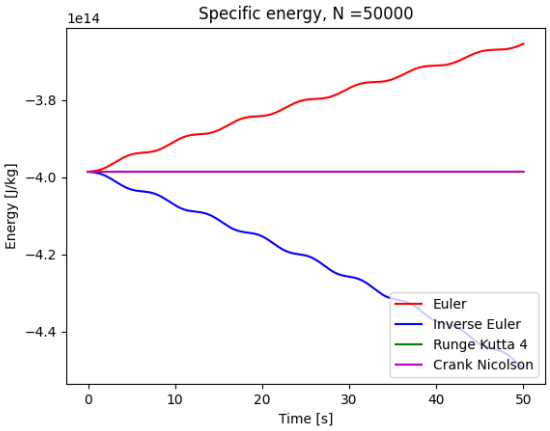
(a) N=200



(b) N=500



(c) N=5000



(d) N=50000

Figura 5: Specific energy, Kepler orbit approximation

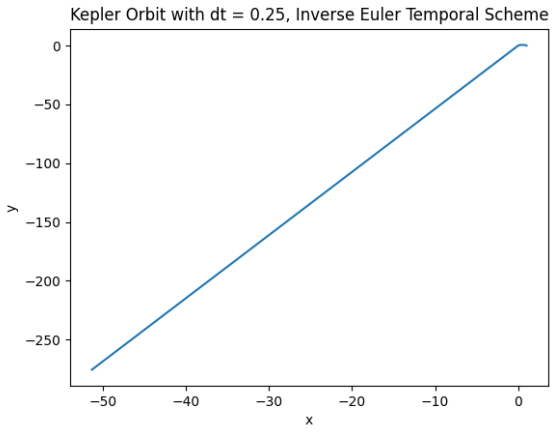
3. Resultados usando Newton-Raphson

En esta sección se realiza una comparación entre la solución proporcionada por el código en el caso de los esquemas implícitos (Euler Inverso y Crank-Nicolson) haciendo uso del método de Newton-Raphson proporcionado por Scipy, el auto-implementado y la función fsolve. Se estudia el caso de N=200 puntos y N= 50000 puntos, para no saturar el documento con gráficas.

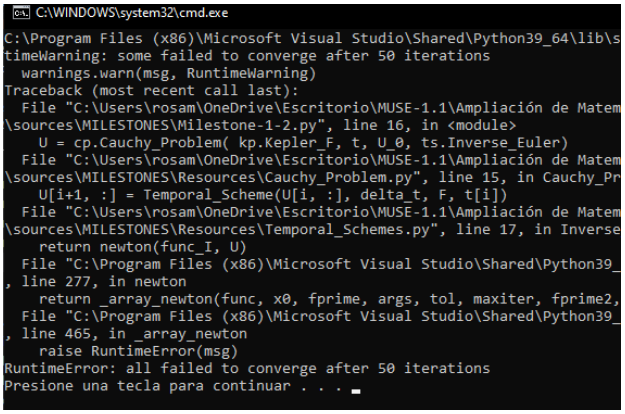
3.1. N=200

Para el método de Newton auto-implementado con el esquema de Euler Inverso, el comportamiento de la solución diverge muy rápidamente con un número de puntos tan pequeño. Sin embargo, el Newton de Scipy ni siquiera encuentra una solución, ya que proporciona un error de no convergencia.

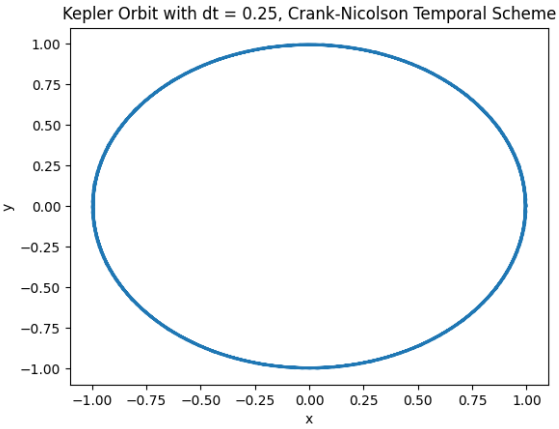
En el caso del esquema temporal del Crank-Nicolson, ambas soluciones son bastante precisas, al igual que lo era con la función fsolve. (**Figura 6**)



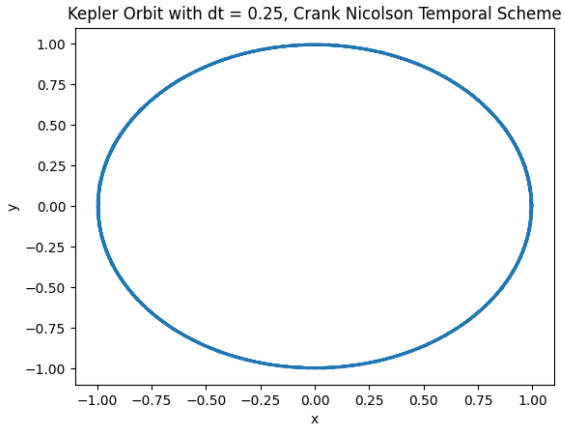
(a) Inverse Euler, own Newton



(b) Inverse Euler, Scipy Newton



(c) Crank-Nicolson, own Newton

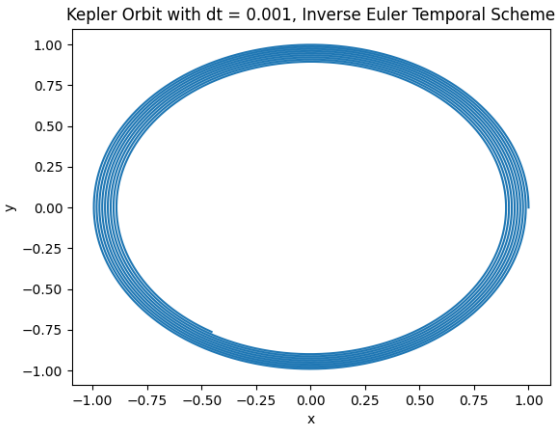


(d) Crank-Nicolson, Scipy Newton

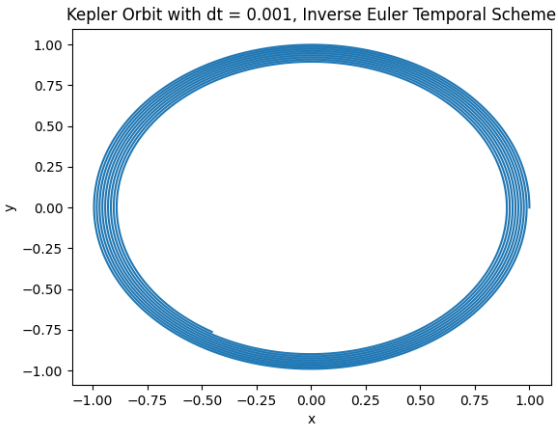
Figura 6: N = 200, t = 50, Newton-Raphson comparison

3.2. N=50000

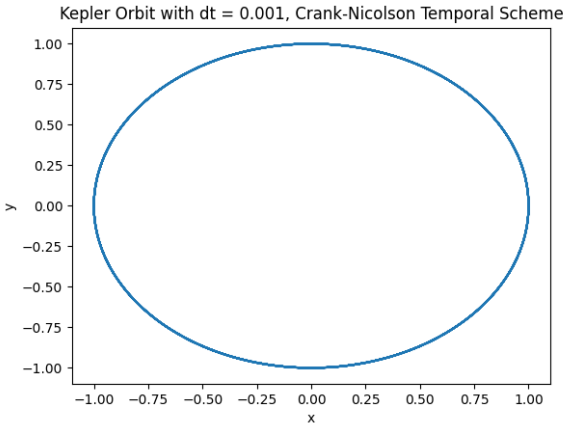
Con 50000 puntos de integración, el método de Euler Inverso comienza a aproximarse a la solución analítica. Podemos observar como ambas soluciones son prácticamente similares en los dos casos, al igual que para la función fsolve anteriormente utilizada. (**Figura 7**)



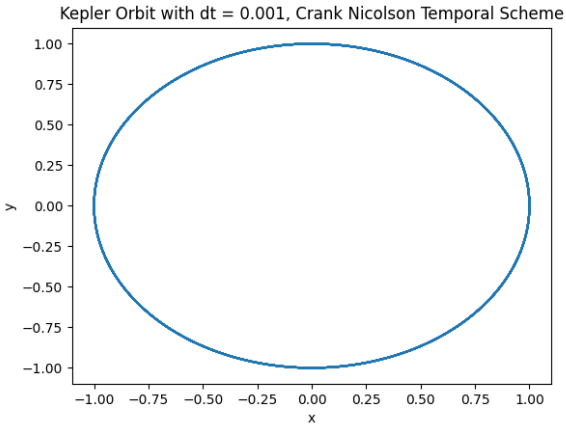
(a) Inverse Euler, own Newton



(b) Inverse Euler, Scipy Newton



(c) Crank-Nicolson, own Newton



(d) Crank-Nicolson, Scipy Newton

Figura 7: N = 50000, t = 50, Newton-Raphson comparison

4. Code Review

Para este Hito, se ha reestructurado el código, utilizando una estructura Top-Down, donde cada módulo solo ve las funciones de las que depende directamente. De esta forma, es mucho más sencillo y claro poder implementar cualquier problema o buscar cualquier error en el código. Por tanto, ahora se tiene una carpeta de *Resources* donde se pueden encontrar los módulos de **Esquemas temporales**, **Problema de Cauchy**, **Métodos Numéricos** (donde se inclute el Newton-Raphson) y **Problema de Kepler**.

Como ejemplo, se adjunta un diagrama de flujo del caso de integrar el problema con un método de Euler Inverso (**Figura 8**).

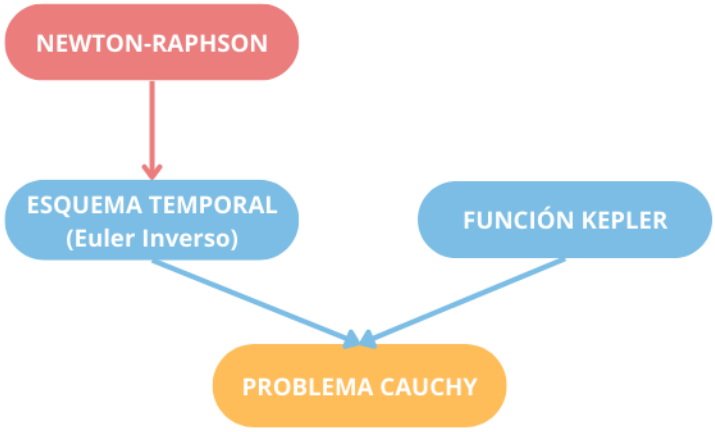


Figura 8: Diagram of the code implementation

En cuanto al método de Newton-Raphson, para su implementación se ha seguido el ejemplo del libro *How to learn Applied Mathematics through modern Fortran*, calculando primero el jacobiano e implementándolo dentro del método iterativo (**Figura 9**).

```
# NEWTON-RAPHSON ALGORITHM : find roots of a real function

from numpy import size, array, zeros, dot
from numpy.linalg import inv, norm

def Jacobian(F, U):
    N = size(U)
    J= array(zeros([N,N]))
    t = 1e-3

    for i in range(N):
        xj = array(zeros(N))
        xj[i] = t
        J[:,i] = (F(U + xj) - F(U - xj))/(2*t)
    return J


def newton(func, U_0):
    N = size(U_0)
    U = array(zeros(N))
    U1 = U_0
    error = 1
    stop = 1e-8
    iteration = 0

    while error > stop and iteration < 1000:
        U = U1 - dot(inv(Jacobian(func, U1)),func(U1))
        error = norm(U - U1)
        U1 = U
        iteration = iteration +1
    return U
```

Figura 9: Newton-Raphson Method Code