


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Milestones 2

Ampliación de Matemáticas 1

6 de octubre de 2022

Autor:

- Javier Zatón Miguel

Índice

1. Objetivo

El código desarrollado tiene la función de resolver un problema de Cauchy usando una variedad de esquemas matemáticos y funciones, se ha realizado una mejora respecto al Milestones 1, se han definido funciones y una estructura modular para simplificar el código.

2. Desarrollo de código

El conjunto de programas se divide en varios módulos, con un programa principal:

2.1. Cauchy_module:

El problema de Cauchy se ha definido como una función, que requiere de 4 input, La función que proviene del modulo Function (Kepler en particular), las condiciones iniciales **U0**, un vector equiespaciado (linspace_t) y un esquema numérico, este ultimo va a ser importado de otro modulo llamado Schemes_module, el cual va a contener varios esquemas numéricos, entre ellos se encuentran los esquemas usados en Milestones 1 (Euler y RK4).

```
def Cauchy(Function,U0,linspace_t,scheme):

    N = linspace_t.shape[0]-1
    U = zeros((U0.shape[0],N+1))
    U.shape = (U0.shape[0],N+1)
    U[:,0] = U0
    for i in range(N):
        U[:,i+1] = scheme(U[:,i],Function,linspace_t[i+1]-linspace_t[i],linspace_t[i])

    return U
```

Como se observa en la imagen superior, se construye la matriz con las n columnas de soluciones, se supone que **U0** es dato y se inserta como un array columna [N,1], la matriz U se tiene que definir con el mismo numero de filas que el vector de condiciones iniciales y N+1 columnas, contando con que la primera de ellas va a ser el propio vector **U0**.

El bucle **for** consiste en rellenar las columnas con cada solución vectorial en un tiempo particular de cualquier esquema, construyéndose de esta forma la matriz U.

2.2. Functions:

El módulo de funciones únicamente contiene la ecuación de Kepler, ya que para este hito es la única requerida.

```
def Kepler(U, t):

    x = U[0]
    y = U[1]
    vx = U[2]
    vy = U[3]
    d = (x**2+y**2)**1.5

    return  array( [ vx, vy, -x/d, -y/d ] )
```

En la función de Kepler aparece el input de el Vector **U**, este en el primer paso corresponde a las condiciones iniciales, como se explicó en el hito 1 contiene las posiciones y velocidades del cuerpo, la función devuelve **F**, que contiene las velocidades y aceleraciones **(vx, vy, ax, ay)**.

2.3. Scheme_module:

El enunciado pide crear 4 funciones para realizar un paso de integración con Euler, Crank Nicolson, Runge Kutta de orden 4 y Euler Inverso.

Estas funciones se van incluidas en el modulo de esquemas numéricos (Scheme_module).

```
def Euler(U,F,dt,t):

    return U + dt*F(U,t)

def RK4(U,F,dt,t):
    k1=F(U,t)
    k2=F(U + dt*k1/2 , t)
    k3=F(U + dt*k2/2 , t)
    k4=F(U + dt*k3 , t)
    k=(1/6)*(k1 + 2*k2 + 2*k3 + k4)

    return U + dt*k

def Inverse_Euler(U,F,dt,t):

    def iter(x):
        return x - U - F(x,t+dt)*dt
    return fsolve(iter, [U])

def Crank_Nicolson(U,F,dt,t):
    def iter(x):
        return x - U -(F(x,t)+F(U,t))*dt/2
    return fsolve(iter,[U])
```

Euler y RungeKutta de orden 4, son funciones sencillas de definir, por otra parte, Crank Nicolson y Euler inverso son métodos explícitos, por ello es necesario usar un fsolve.

De forma similar, se ha creado un módulo denominado Scheme_module_Newton , donde los sistemas no lineales se resuelven con el método Newton-Raphson, definido en otro módulo aparte, Newton_module.py.

```
def Jacobiano (F, xp):
    N = size(xp)
    dx = 1e-3

    x = zeros(N)
    Jab = zeros([N,N])
    for j in range(N):
        x[j] = dx
        Jab[:,j] = ( F(xp + x ) - F(xp - x) ) /(2*dx)
    return Jab


def Newton(F,x0):
    N = size(x0)
    dx = 2 * x0
    it = 0 #Initialization of iterations
    itmax = 1000 #max iterations
    eps = 1 # initial error

    while (eps > 1e-8) and (it <= itmax):
        it = it + 1
        Jab = Jacobiano(F, x0)
        b = F(x0)
        dx = dot( inv(Jab), b)
        x0 = x0 - dx
        eps = norm(dx)
    return x0
```

Para los resultados se ha usado el módulo Scheme_module, ya que estos son idénticos y el programa principal obtiene los resultados más rápido usando fsolve que Newton. Para cambiar al módulo Scheme_module_Newton solo hay que modificar la linea 3 de Milestones_2.py.

2.4. Milestones_2:

Este programa importa el resto de modulos para crear las gráficas de los resultados, que en este caso será representar orbitas dadas unas condiciones iniciales, igual que en el Milestones 1.

De esta forma, únicamente se necesita llamar a Cauchy, que necesita unas condiciones iniciales **U0** y un vector temporal (Linspace_t), estos se definen en el programa principal.

Para estudiar distintos casos, se llama a la función de Kepler para cada uno de los esquemas numéricos que se han mencionado, por ejemplo:

```
U0 = array([1.,0.,0.,1.])
linspace_t = linspace(0,10,1001)

Solución = Cauchy(Kepler,U0,linspace_t,Euler)
#plot
Xorb = Solución[0,:]
Yorb = Solución[1,:]
fig, ax = plt.subplots()
ax.plot(Xorb,Yorb)
plt.show()
```

En este ejemplo, se ha determinado que se quiere obtener el calculo de la órbita Kepleriana usando el esquema de Euler, para unas condiciones iniciales.

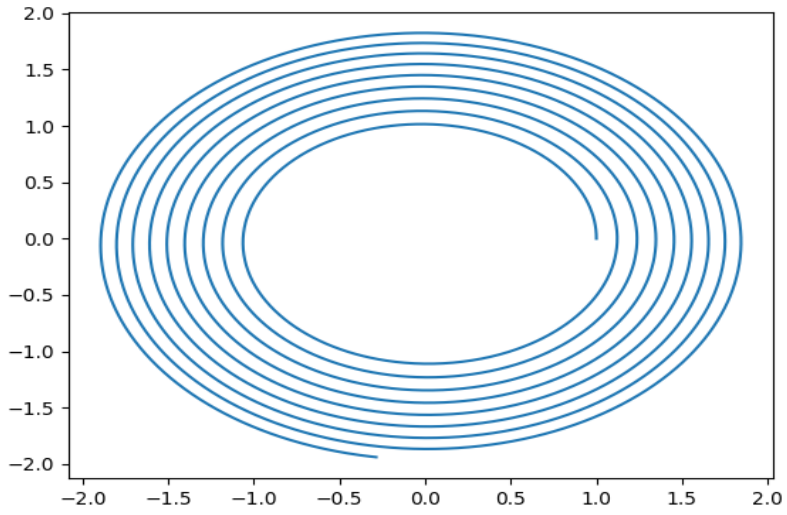
Definiendo linspace(i,f,d), donde i es el tiempo inicial , f el tiempo final y d el numero de particiones equidistantes del vector. El salto $\Delta t = (f - i)/(d - 1)$ y los N pasos como la dimensión del vector menos uno.

A continuación se van a mostrar y discutir los resultados tras usar cada esquema para distintos saltos en el tiempo.

3. Resultados

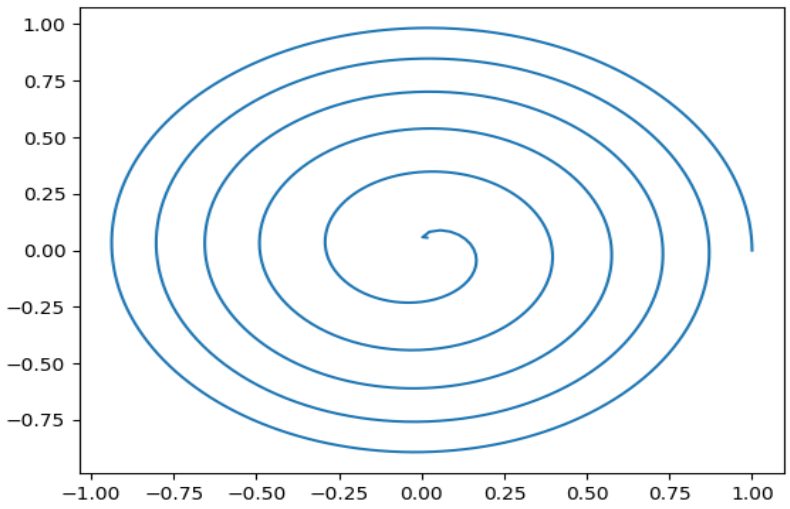
3.1. Δt = 0.01 , N=10000

Órbita Euler caso 1:



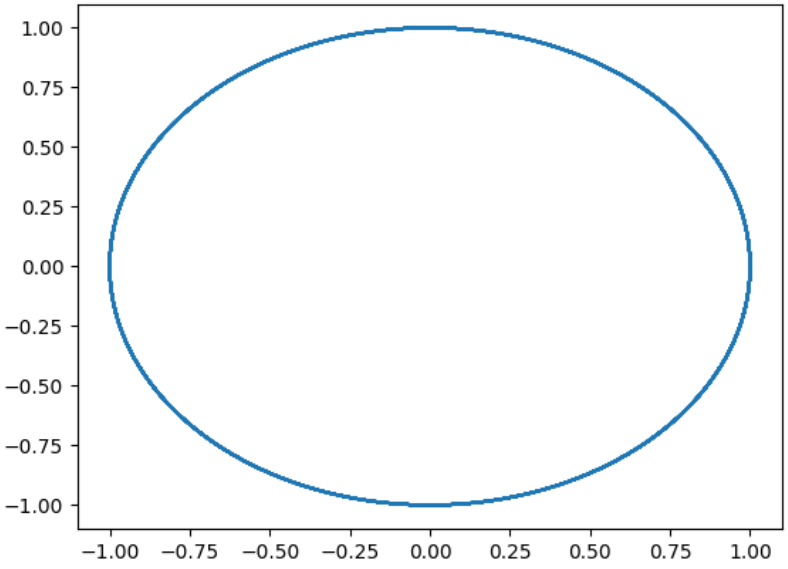
Igual que en Milestones 1, el método de Euler tiene un error considerable y tiende a divergir, aumentando el vector r en cada vuelta.

Órbita Euler inverso caso 1:

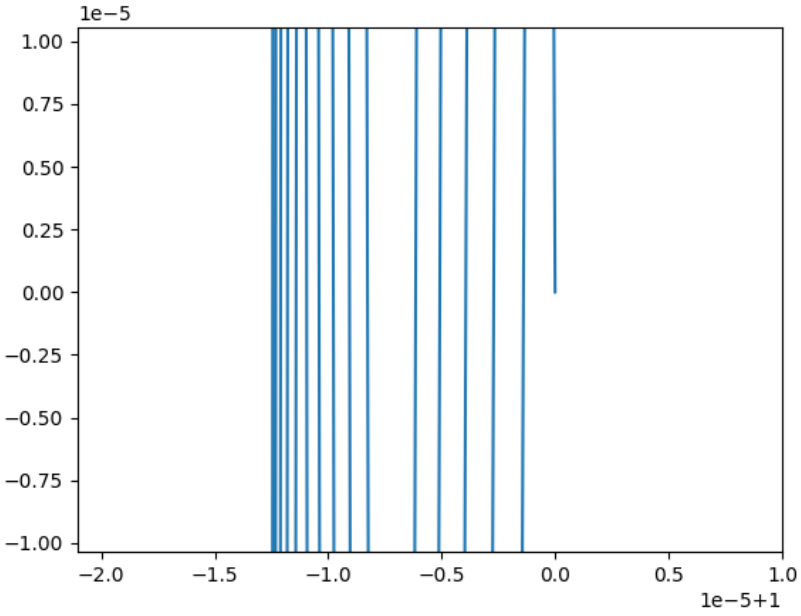


Para el caso explícito, ocurre lo contrario, el error reduce el modulo del vector r en cada vuelta. Aparente el error es del mismo orden de magnitud para Euler y Euler inverso.

Órbita Crank Nicolson caso 1:

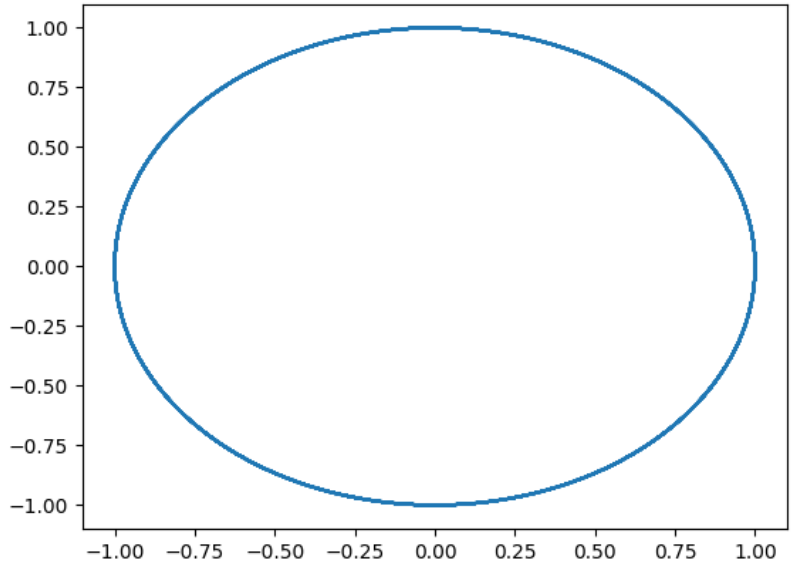


Para Crank Nicolson, la partición usada en el caso 1 es suficiente para que el error sea despreciable.

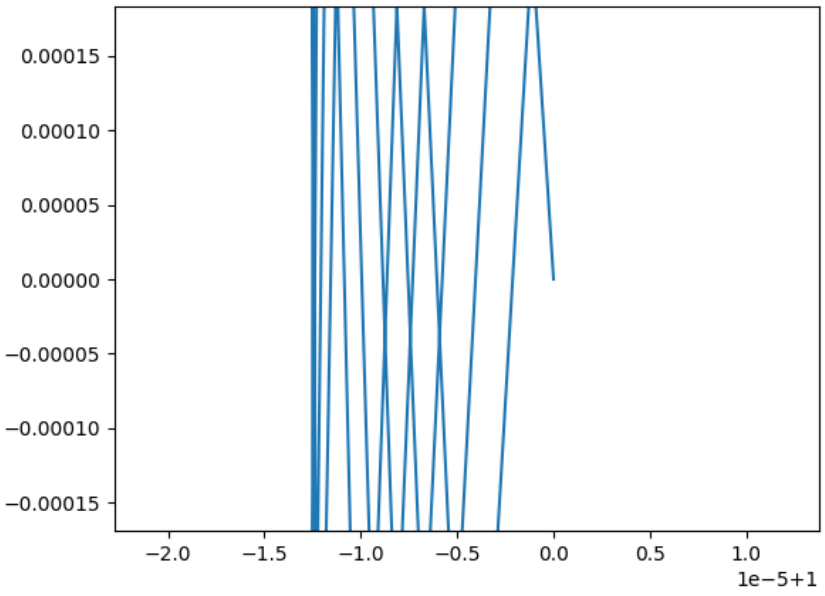


Ampliando en la imagen se puede identificar que el error es del orden de $10^{-}5$.

Órbita RK4 caso 1:



El caso de RK4 es similar a Crank Nicolson, su error no se aprecia a simple vista.

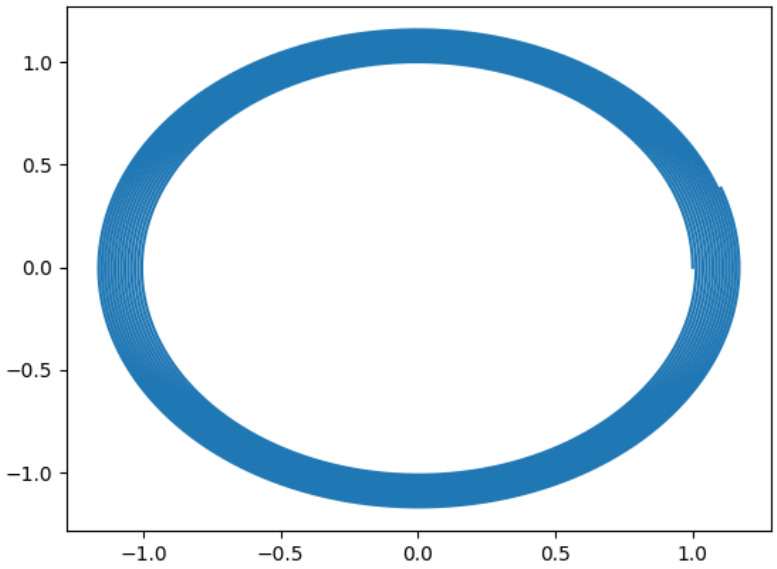


Haciendo zoom como en el caso anterior, se identifica un error total del orden de $10^{-}5$.

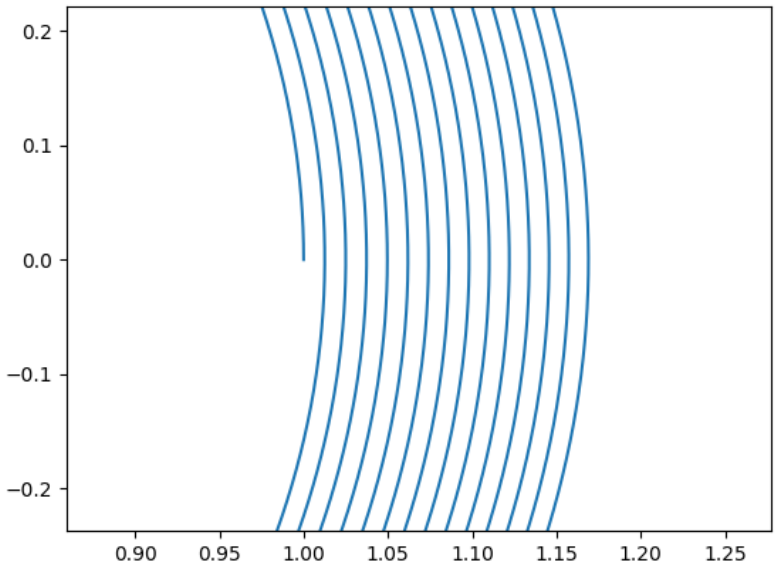
3.2. Δt = 0.001 , N=100000

En el caso 2, se va a reducir la partición temporal a un valor diez veces menor que el resto.

Órbita Euler caso 2:

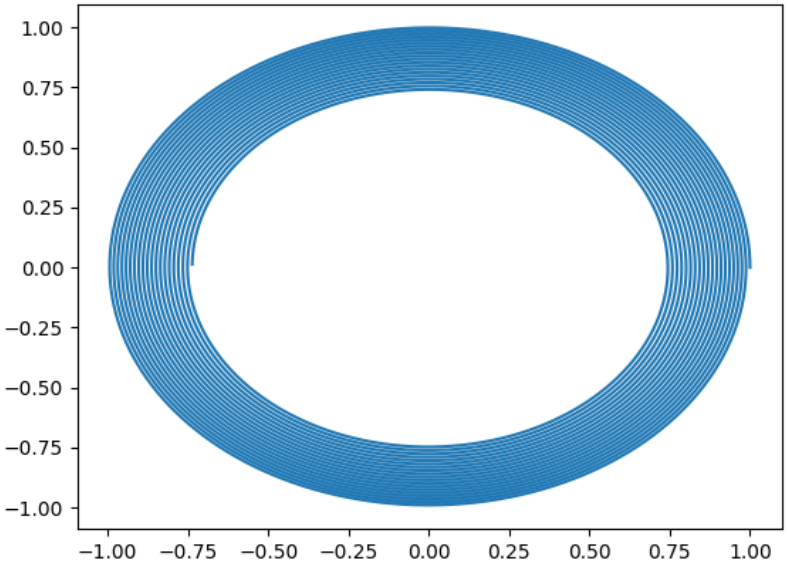


La reducción del valor del salto temporal reduce el error cometido por el esquema aunque sigue siendo considerable después de varias órbitas completadas.

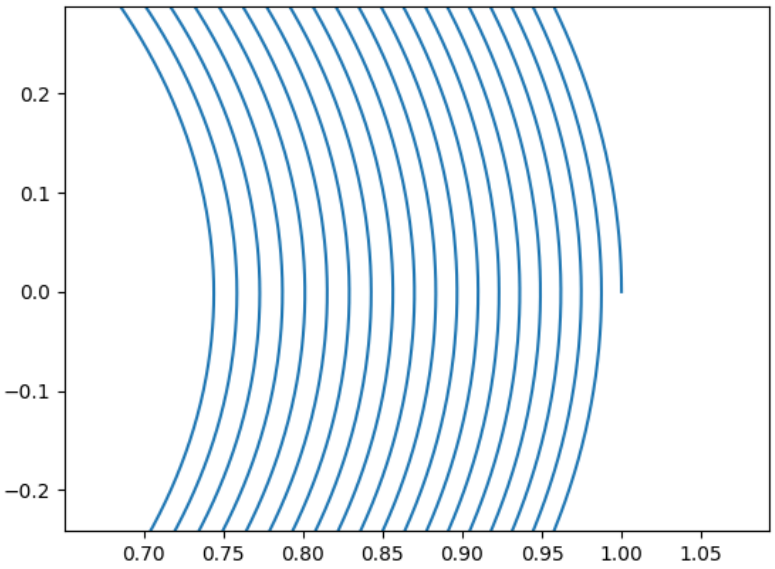


En este caso, el modulo del vector r ha aumentado mas de un 15 %.

Órbita Euler inverso caso 2:

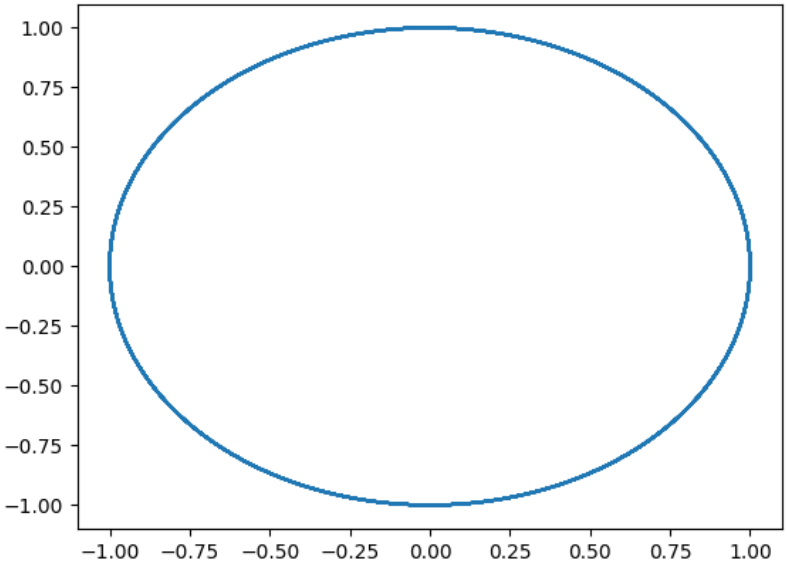


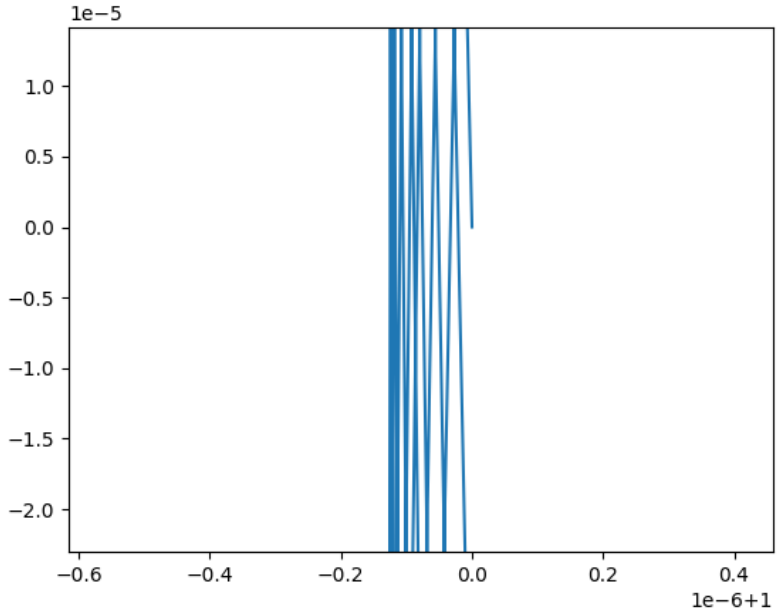
Para Euler inverso sucede lo mismo que el primer caso, la precisión del resultado es mayor, pero para tiempos muy altos este error acumulado sigue siendo considerable.



El modulo del vector r ha sido reducido a un 73 % aproximadamente de su valor inicial.

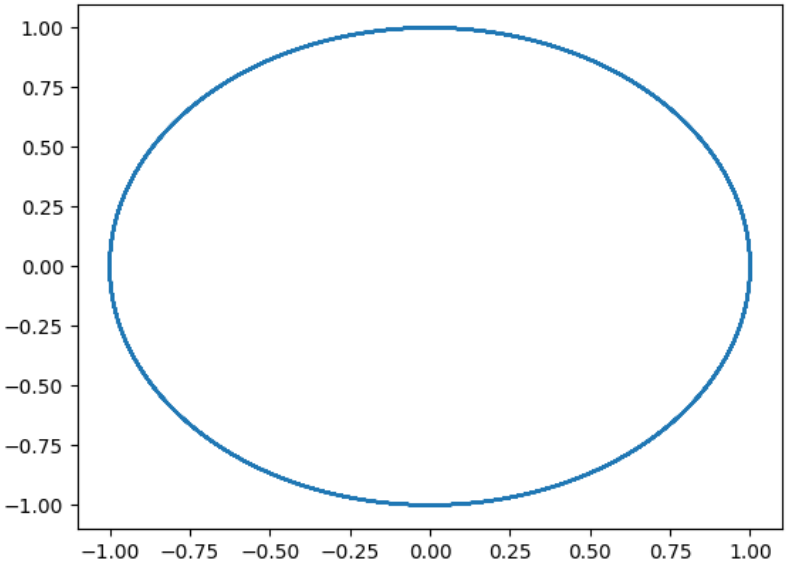
Órbita Crank Nicolson caso 2:

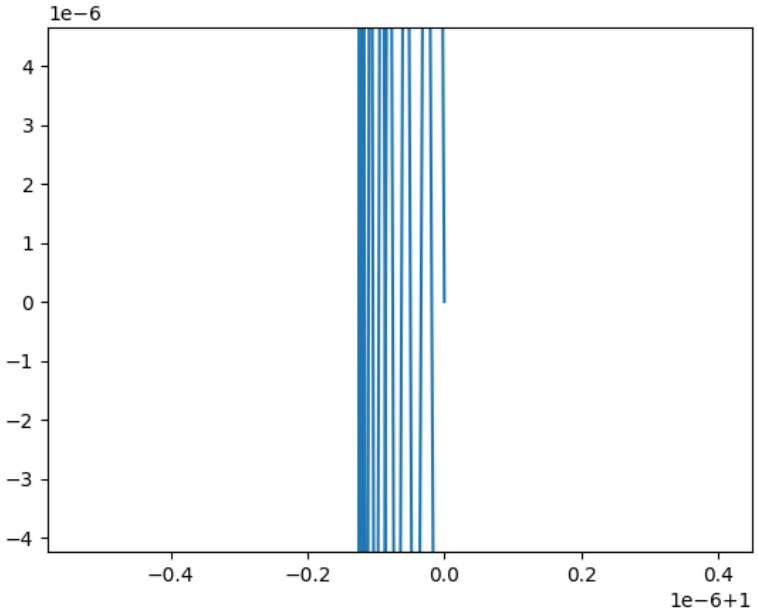




Para Crank Nicolson, el cambio es apenas notable, si se presta atención a la segunda imagen, se puede ver que el error cometido es de un orden mas pequeño ($10^{-}6$).

Órbita RK4 caso 2:



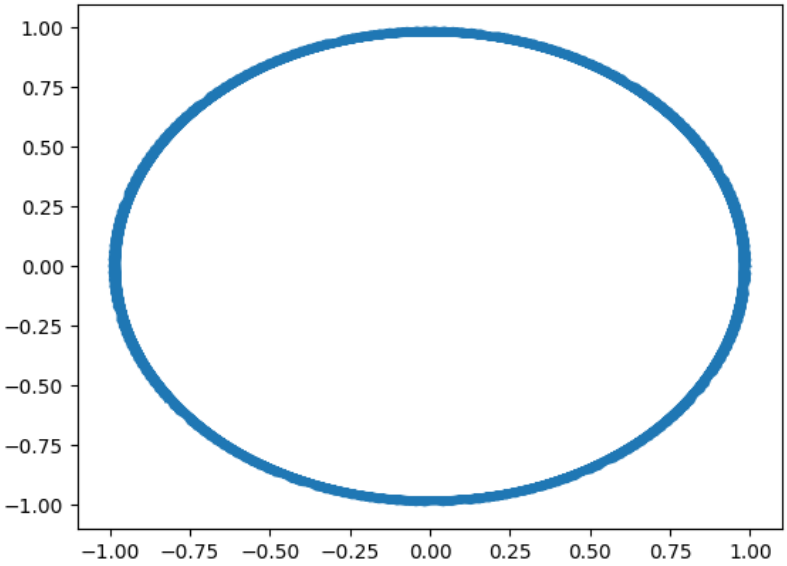


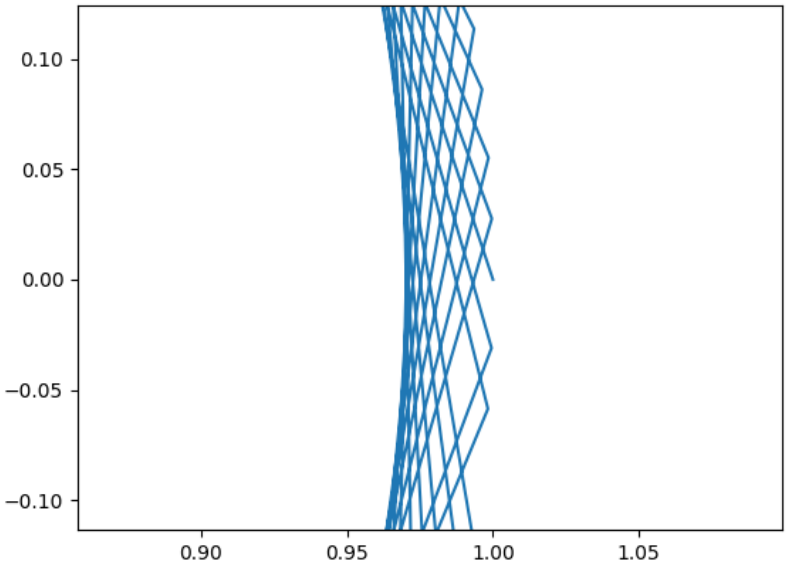
Con el caso de RK4 sucede algo similar, el error se reduce en un orden.

3.3. $\Delta t = 0.5$, N=200

Este ultimo caso usa un Δt considerablemente grande, lo que empeora los resultados, para este caso solo se representan los esquemas Crank Nicolson y RK4.

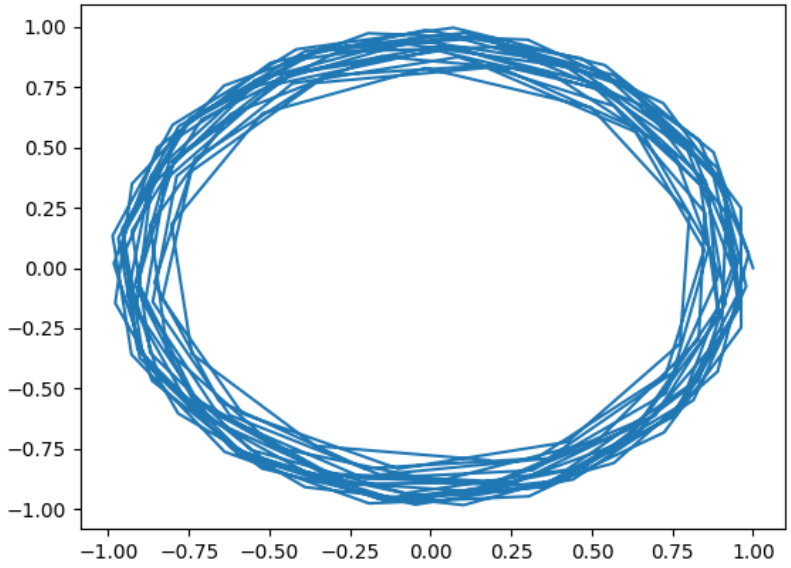
Órbita Crank Nicolson caso 3:





Con este esquema se comete un error total de un 5 %, mejor resultado que el Euler con $\Delta t = 0{,}001$.

Órbita RK4 caso 3:



Para RK4 se puede observar a simple vista que los resultados para bajos N empeoran de forma considerable.

4. Conclusión

Comparando los cuatro esquemas para distintos Δt se ha llegado a la conlcusión de que Euler explícito no aporta una mejora a los resultados para el problema de Cauchy de Kepler.

Crank Nicolson y Runge Kutta, para valores de Δt no muy pequeños, consiguen resultados con un error despreciable.