


Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

MÁSTER UNIVERSITARIO EN SISTEMAS ESPACIALES

MILESTONE 4

Ampliación de Matemáticas I

25 de noviembre de 2022

Autor:
Alberto García Rincón

Índice

1. Introducción

En este trabajo se ha definido un nuevo problema, el oscilador armónico lineal, que se va a integrar mediante los distintos métodos de cálculo numérico que ya se han utilizado en trabajos anteriores y además se ha añadido un nuevo método de integración más: Leap Frog.

El oscilador armónico lineal queda definido por la siguiente ecuación diferencial:

$$\ddot{x} + x = 0 \tag{1}$$

Posteriormente se va a realizar el cálculo y análisis de las regiones de estabilidad de los distintos métodos de cálculo numérico utilizados hasta la fecha.

2. Código de Python

2.1. Diagrama de bloques

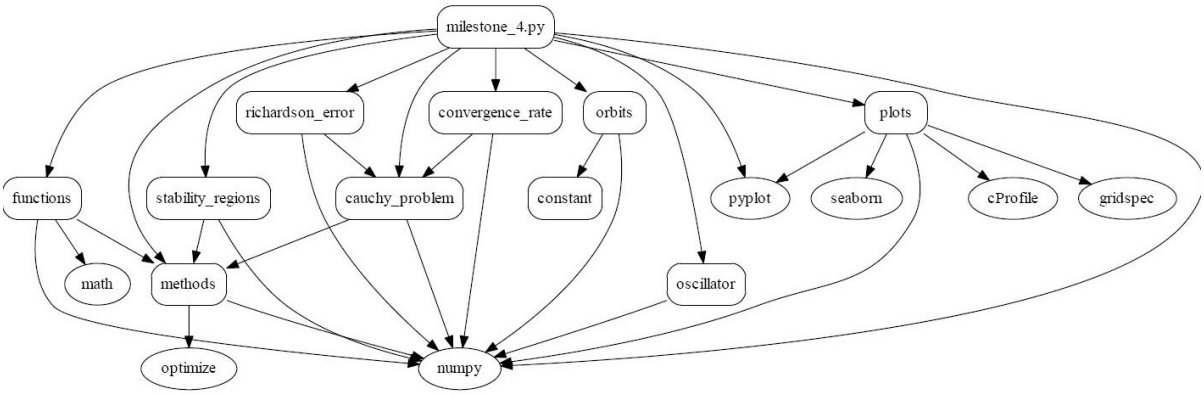

Figura 1: Diagrama archivos Milestone 4.

Para hacer funcionar este programa se debe ejecutar el programa principal denominado $milestone\_4.py$. Este programa se encarga de llamar al resto de funciones definidas en los respectivos archivos.

Se van a comentar los cambios introducidos respecto al trabajo de la semana pasada en el código de los archivos del programa.

2.2. milestone_4.py

En el archivo principal se ha definido un nuevo problema a resolver, en este caso se trata del problema del oscilador armónico. Para ello se han definido unas condiciones iniciales de posición y velocidad, como se muestran en la figura 2. También se ha definido un vector Ut que

contiene varios Δt, se realizarán simulaciones del problema del oscilador con estos incrementos de tiempo.

El vector $mets$ contiene los distintos métodos de resolución que se van a utilizar para resolver el problema.

```
16    #Initialize values
17    Dt = np.array([0.1, 0.01, 0.001])   #[s]
18    tf = 14.                            #[s]
19    #U0 = fun.init_state_vector_orbits(1, 1, 0) #(R[m], V[m/s], Zeta[deg]) -> Orbits
20    U0 = np.array([1, 0])                    #(R[m], V[m/s], Zeta[deg]) -> Oscilator
21    mets = [met.explicit_euler, met.runge_kutta4, met.inverse_euler, met.crank_nicolson, met.leap_frog]
```

Figura 2: Iniciación de valores en el programa principal.

En la siguiente figura 3, se muestra el código que permite la resolución del problema del oscilador armónico para los Δt definidos en el vector anteriormente citado. De esto de encarga el primer bucle, siendo el segundo el que recorre el vector de los métodos definidos. Finalmente los valores de los resultados se almacenan en el vector de vectores UU, que ha sido inicializado anteriormente.

```
27    #Calculus
28    for j in range(len(Dt)):
29        N = int(tf/Dt[j])+1            #Nº steps
30        t = np.linspace(0, tf, N)
31
32        U = np.zeros([len(mets), len(U0), N])
33        En = np.zeros([len(mets), N])
34        E = np.zeros([len(mets), len(U0)+1, N])
35        CR = np.zeros([len(mets), 2, m])
36
37        for i in range(len(mets)):
38            q = fun.order(mets[i])
39
40            """
41            #Orbits
42            U[i,:] = cp.cauchy_problem(U0, t, orb.orbits, mets[i])       #Cauchy solver orbits
43            En[i,:] = orb.orbit_energy(U[i,:], t)                        #Energy orbit
44            E[i,:] = re.richardson_error(U0, t, orb.orbits, mets[i], q)  #Error Richardson
45            CR[i,:] = cr.convergence_rate(U0, t, orb.orbits, mets[i], m) #Convergence Rate
46            """
47
48            #Oscillator
49            U[i,:] = cp.cauchy_problem(U0, t, osc.oscilator, mets[i])
50            #E[i,:] = re.richardson_error(U0, t, osc.oscilator, mets[i], q)
51            #CR[i,:] = cr.convergence_rate(U0, t, osc.oscilator, mets[i], m)
52
53
54        UU[j] = U
55        TT[j] = t
```

Figura 3: Código de cálculo de los valores del problema del oscilador armónico.

Unas líneas más abajo en el archivo principal. Se definen las condiciones para calcular las regiones de estabilidad de los diferentes métodos de cálculo y el bucle que recorre estos mismos.

Desde este mismo bucle se llama a la función encargada de dibujar las distintas regiones de estabilidad.

```
71      #Stability
72      a = 3
73      n = 100
74      x = np.linspace(-a, a, n)
75      y = np.linspace(-a, a, n)
76      ST = np.empty(len(mets), dtype=np.ndarray)
77
78      for z in range(len(mets)):
79          ST[z] = sr.stability_regions(x, y, mets[z])
80          plo.plot_stability(x, y, ST[z], mets[z].__name__)
```

Figura 4: Código para cálculo de regiones de estabilidad.

2.3. plot.py

En este archivo se definen todas las funciones necesarias para dibujar las distintas gráficas de los resultados obtenidos.

La siguiente figura muestra el código de la función encargada de dibujar la posición y velocidad del oscilador para distintos Δt calculados y para todos los métodos con los que se ha realizado el cálculo.

Se definen dos bucles: el primero para recorrer los métodos de cálculo y el segundo para recorrer los distintos Δt. Esta función permite que siendo llamada una única vez desde el programa principal, dibuje todas las gráficas de una vez.

```
231 #plot oscillator in all Dt
232 def plot_osc_dt(UU, TT, Dt, mets):
233     sns.set()
234     for z in range(len(mets)):
235         plt.figure()
236         plt.tight_layout()
237         plt.subplots_adjust(hspace=0.3, wspace=0.3, top=0.95, bottom=0.05, left=0.05, right=0.95)
238         gs = gridspec.GridSpec(2, 1)
239         ax1 = plt.subplot(gs[0, 0])
240         ax2 = plt.subplot(gs[1, 0])
241
242         for i in range(len(Dt)):
243             ax1.plot(TT[i], UU[i][z,0], label="$\Delta t$ = " + str(Dt[i]) + "[s]")
244             ax2.plot(TT[i], UU[i][z,1], label="$\Delta t$ = " + str(Dt[i]) + "[s]")
245
246         ax1.set_title("Oscillator Position, " + mets[z].__name__)
247         ax2.set_title("Oscillator Velocity, " + mets[z].__name__)
248         ax1.set(xlabel='Time [s]', ylabel='Position [m]')
249         ax2.set(xlabel='Time [s]', ylabel='Velocity [m/s]')
250         ax1.set_xlim([TT[0][0], TT[0][-1]])
251         ax2.set_xlim([TT[0][0], TT[0][-1]])
252         ax1.legend()
253         ax2.legend()
```

Figura 5: Código función dibujar posición y velocidad del oscilador.

Esta otra función permite dibujar la región de estabilidad del método que se pase como argumento en los valores calculados anteriormente. Se dibuja una gráfica definida por el eje Real y el eje Imaginario.

```
256  #plot stability
257  def plot_stability(x, y, rho, name):
258      sns.set()
259      plt.figure()
260      plt.contour(x, y, np.transpose(rho), np.linspace(0, 1, 11))
261      plt.title("Absolute Stability Region for " + name)
262      plt.xlabel('Re')
263      plt.ylabel('Imag')
264      plt.axis('equal')
265
```

Figura 6: Código función dibujar regiones de estabilidad.

2.4. stability_regions.py

En la siguiente figura se muestra el código de la función que calcula la región de estabilidad pasándole como argumento un método de integración numérica.

```
1   from numpy import zeros, float64, abs
2   from fun.methods import leap_frog
3
4   def stability_regions(x, y, method):
5       N = len(x)
6       rho =  zeros((N, N),  dtype=float64)
7
8       for i in range(N):
9           for j in range(N):
10              w = complex(x[i], y[j])
11              if method == leap_frog:
12                  r = method(1, 1, 1, 0, lambda u, t: w*u)
13              else:
14                  r = method(1, 1, 0, lambda u, t: w*u)
15
16              rho[i, j] = abs(r)
17
18      return rho
```

Figura 7: Código función cálculo regiones de estabilidad.

3. Resultados de los métodos

3.1. Oscilador armónico lineal

Se han realizado 3 simulaciones distintas para cada método con 3 distintos Δt. El análisis de los resultados se muestra a continuación.

3.1.1. Método de Euler explícito

Como se muestra en la siguiente figura para integraciones con incrementos de tiempo altos, el error acumulado se va haciendo notable a partir de los 5 segundos de simulación. Este error se manifiesta en la amplitud tanto de la posición como de la velocidad, aumentando respecto a la solución analítica.

Conforme se va disminuyendo el incremento de tiempo utilizado en la simulación el resultado del método se asemeja más a la resolución analítica y por por lo tanto es una solución más estable.

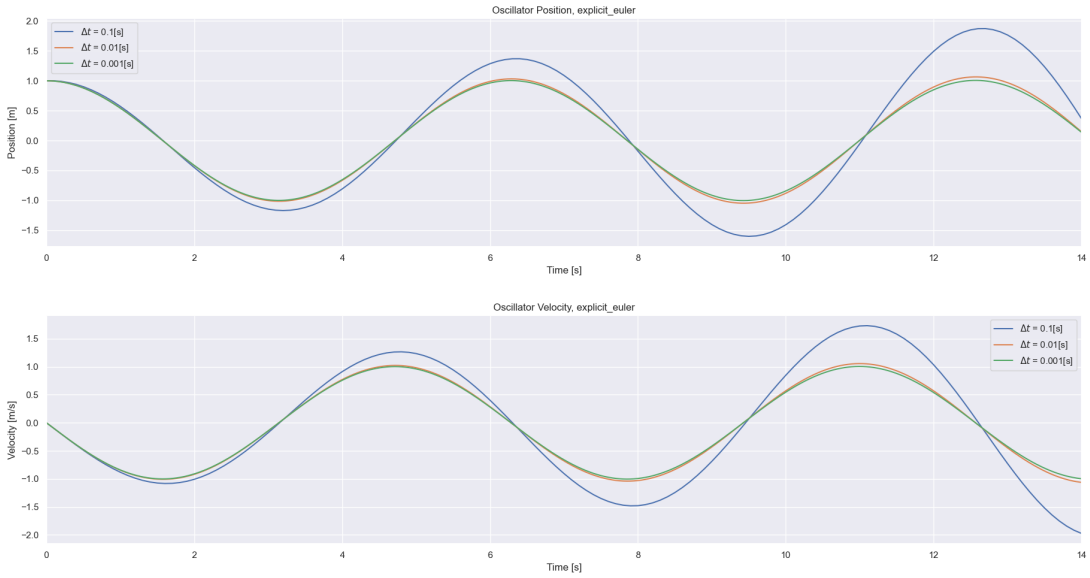

Figura 8: Resolución oscilador con método de Euler y distintos Δt.

3.1.2. Método de Runge-Kutta de orden 4

Este método de integración consigue obtener muy buenos resultados incluso para Δt altos.

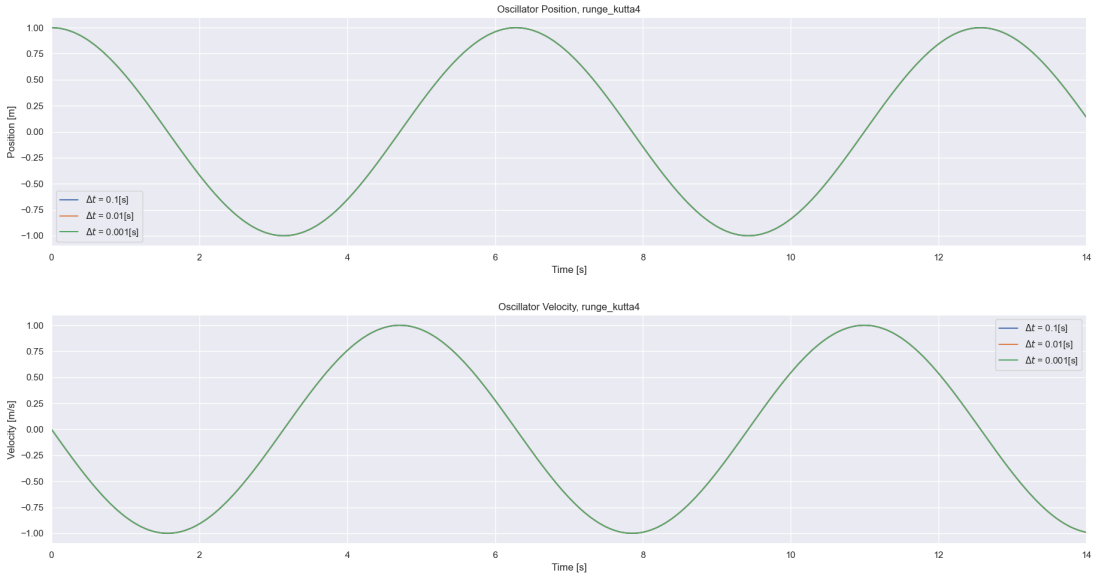

Figura 9: Resolución oscilador con método de Runge-Kutta de orden 4 y distintos Δt.

3.1.3. Método de Euler inverso

Como se muestra en la siguiente figura para integraciones con Δt altos, el error acumulado se va haciendo notable a partir de los 5 segundos de simulación. Este error se manifiesta en la amplitud tanto de la posición como de la velocidad, disminuyendo respecto a la solución analítica.

Conforme se va disminuyendo el Δt utilizado en la simulación, el resultado del método se asemeja más a la resolución analítica y por por lo tanto es una solución más estable.

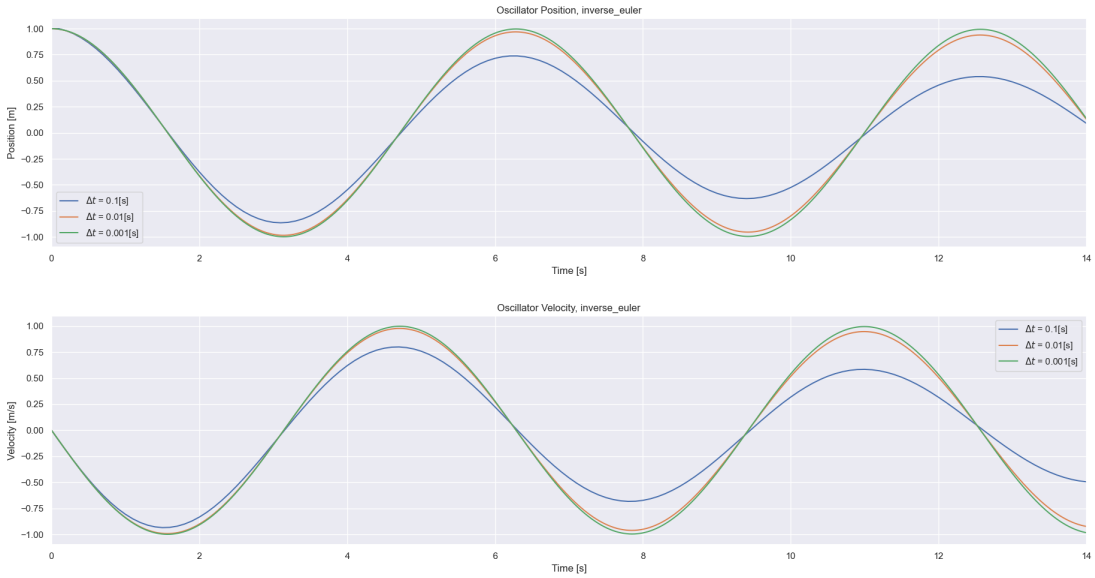

Figura 10: Resolución oscilador con método de Euler inverso y distintos Δt.

3.1.4. Método de Crank-Nicholson

Con este método de resolución numérico se obtienen buenos resultados incluso para incrementos de tiempo altos. Esto se traduce en que este método es bastante estable en la resolución de problemas numéricos.

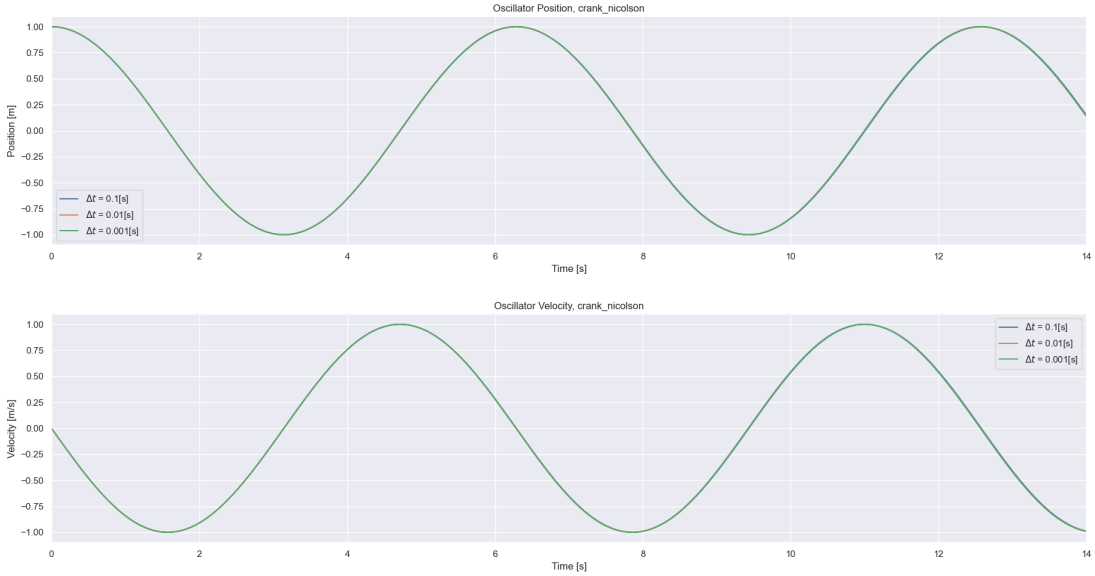

Figura 11: Resolución oscilador con método de Crank-Nicholson y distintos Δt.

3.1.5. Método de Leap-Frog

Con el nuevo método implementado de resolución numérica se obtienen resultados bastante buenos a partir de $\Delta t \leq 0{,}01[s]$. Para incrementos de tiempo mayores se puede apreciar unos escalones entre cada paso temporal, conforme disminuimos dicho incremento de tiempo, esas "discontinuidades"desaparecen o se hacen mucho más pequeñas e imperceptibles.

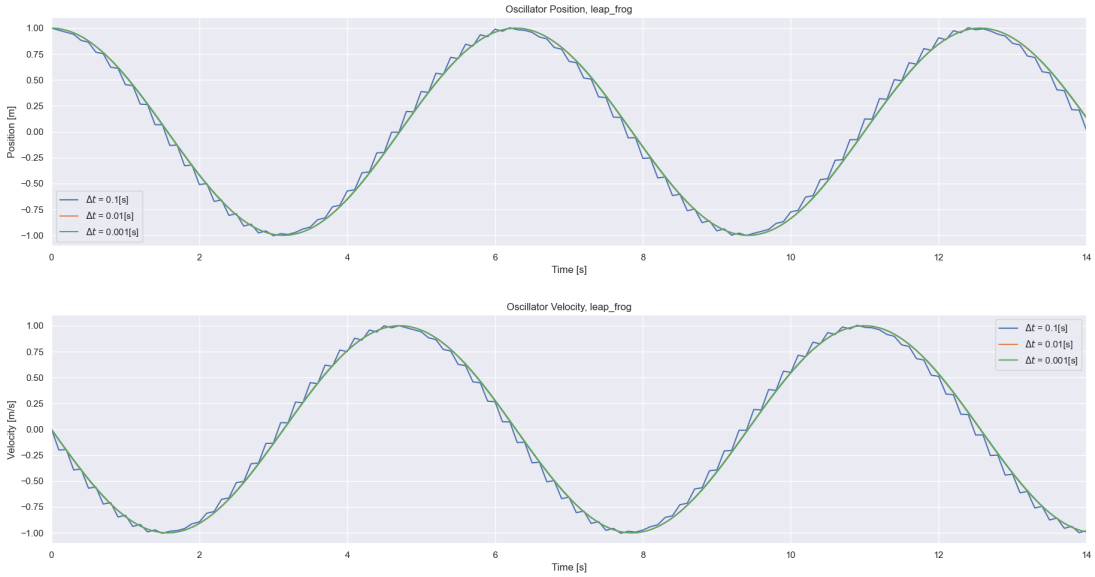

Figura 12: Resolución oscilador con método de Leap-Frog y distintos Δt.

3.2. Regiones de estabilidad

3.2.1. Método de Euler explícito

La región de estabilidad de este método es el interior de una circunferencia cuyo centro es el punto (-1, 0) en el plano complejo.

Como se muestra en la figura siguiente, en estas regiones, en el plano complejo, revelan los valores que puede asumir z (nº complejo) tal que el error en el método se comporte asintóticamente igual al error local de truncado para todo tiempo de la simulación.

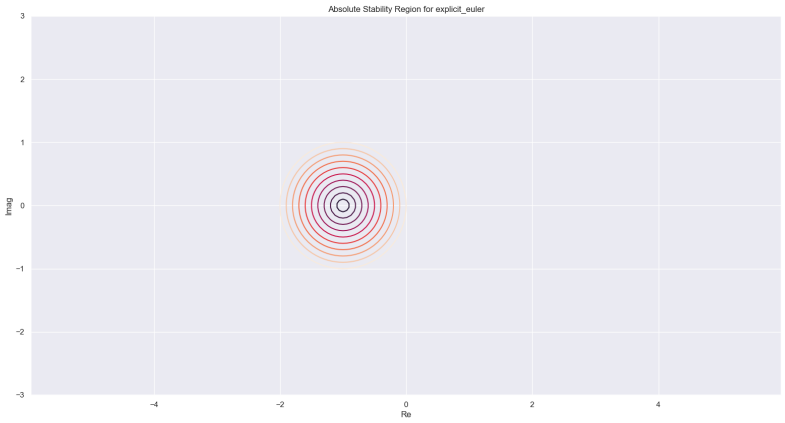

Figura 13: Región de estabilidad método de Euler.

3.2.2. Método de Runge-Kutta de orden 4

En la siguiente figura se muestra la región de estabilidad del método de Runge-Kutta, como puede observarse se trata de una región bastante amplia que hace que este método genere soluciones bastante estables incluso con incrementos de tiempo altos.

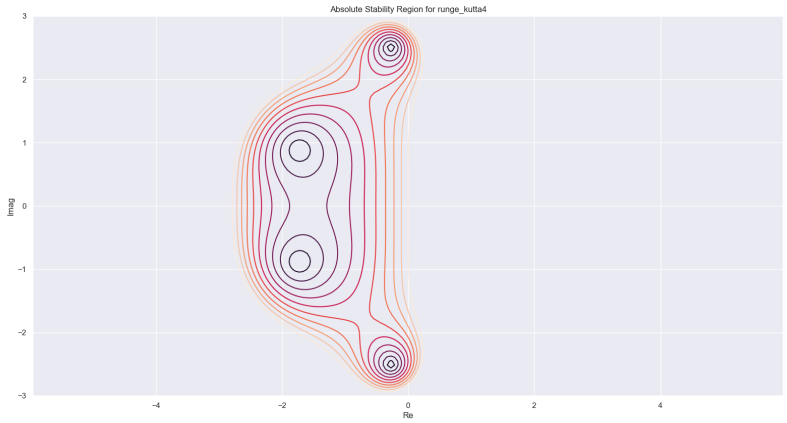

Figura 14: Región de estabilidad método de Runge-Kutta de orden 4.

3.2.3. Método de Euler inverso

En este caso la región de estabilidad es el caso contrario que para el método de Euler explícito. Esta se sitúa en el exterior de una esfera centrada en el punto (1, 0) del plano complejo.

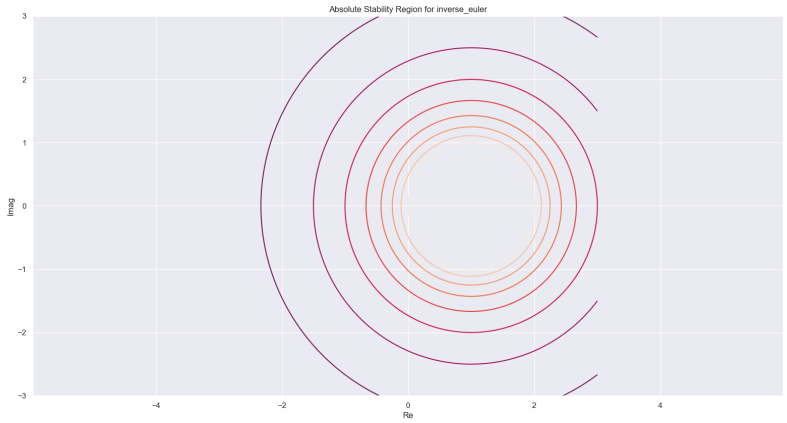

Figura 15: Región de estabilidad método de Euler inverso.

3.2.4. Método de Crank-Nicholson

En la siguiente figura se muestra la región de estabilidad del método de Crank-Nicholson, la cual está perfectamente acotada en la parte negativa del eje Real del plano complejo. Esto hace

que este método genere soluciones bastante estables incluso para incrementos de tiempo altos.

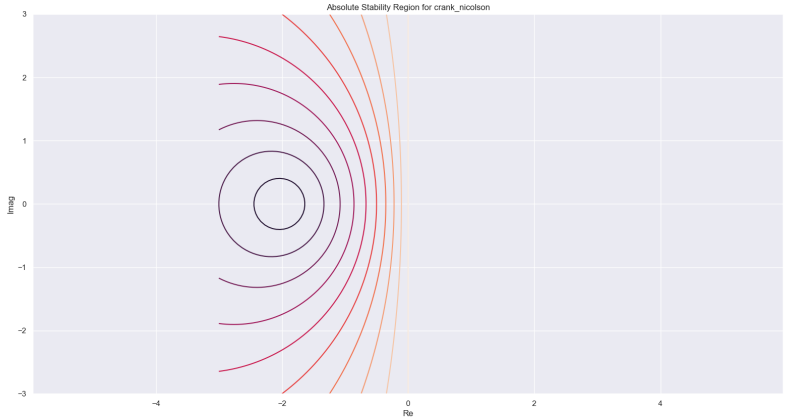

Figura 16: Región de estabilidad método de Crank-Nicholson.

3.2.5. Método de Leap-Frog

En la siguiente figura se muestra la región de estabilidad absoluta del método de Leap-Frog.

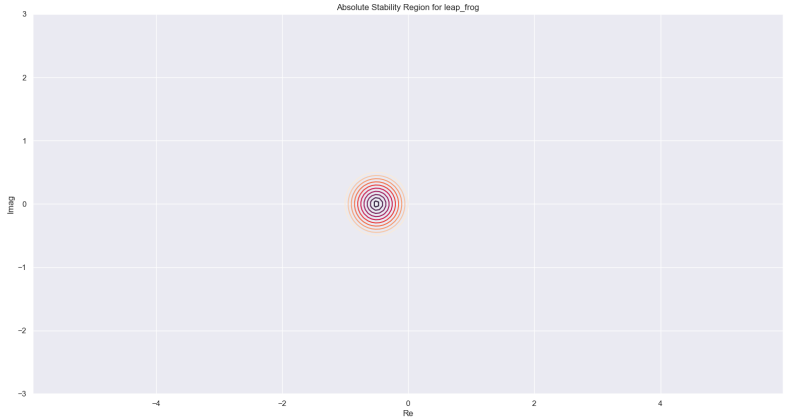

Figura 17: Región de estabilidad método de Leap-Frog.