



Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

# Milestone 4

**Ampliación de Matemáticas I**

**1 de noviembre de 2022**

**Autor:**

Sergio de Ávila Cabral

# Índice

**1. Introducción** **1**

**2. Descripción de módulos nuevos o modificados** **1**

2.1. Milestone_4.py . . . . . 2

2.2. Physics.py . . . . . 4

2.3. R_S.py . . . . . 4

2.4. Plots.py . . . . . 5

**3. Resultados** **6**

# 1. Introducción

Para la realización el *Milestone 4* se define como objetivo principal la integración del oscilador lineal:

$$\ddot{x} + x = 0 \tag{1.0.1}$$

Dicha integración será realizada mediante los esquemas numéricos ya empleados hasta ahora (Euler, Runge-Kutta de orden 4, Crank-Nicolson y Euler inverso) junto con un nuevo esquema, *LeapFrog*. El segundo objetivo es el cálculo de las regiones de estabilidad de dichos esquemas.

Para la realización de estos cálculos se han empleado los módulos ya existentes:

- Physics.py: Incluye la física del problema a resolver. Se encuentra en *Sources/Problems*.
- Temporal_Schemes.py: Se incluyen los diferentes esquemas numéricos. Se localiza en *Sources/ODES*.
- Cauchy_Problem.py: En el se encuentra la definición del problema de Cauchy, cuya localización coincide con la de Temporal_Schemes.py.
- Jab_Newt.py: En el se encuentra la integración numérica de Newton junto con el Jacobiano. Se localiza en el mismo path que las dos anteriores.
- Plots.py: se encarga de graficar las soluciones de los problemas de Cauchy. Se localiza en *Sources/Graf*.

Y se ha definido los siguientes nuevos módulos:

- Milestone_4.py: Actúa como *main* y en él se encuentran importados todos los módulos. A su vez, se definen: las condiciones iniciales del problema, las iteraciones, diccionarios y las llamadas a las funciones que integran el problema objetivo y, por último, llama a la nuevas funciones que calculan la región de estabilidad de los diferentes esquemas numéricos.
- R_S.py: En el se encuentran las dos nuevas funciones que permiten el segundo objetivo principal de este *Milestone*: la función *Stability_Region*, que calcula la región de estabilidad; y la función *Stability_Region*, que entrega a la función anterior el polinomio de estabilidad de cada esquema numérico.

# 2. Descripción de módulos nuevos o modificados

A continuación se hará una descripción de los nuevos módulos y de las modificaciones realizadas a ciertos módulos ya existentes.

## 2.1. Milestone_4.py

Como se ha descrito en la Sección 1, actual como *main*. Este módulo sigue la misma estructura que los *main* de anteriores *Milestones*. En el se pueden apreciar diferentes secciones a lo largo del mismo:

En primer lugar, se exponen las librerías empleadas para el desarrollo de este módulo:

```
from numpy import array, save, zeros, linspace, shape
from ODES.Cauchy_Problem import C_P
from Problems.Physics import Linear_Oscilator
from ODES.Temporal_Schemes import Euler, RK4, Crank_Nicolson,
    Inverse_Euler, LeapFrog
from Stability_Region.S_R import Stability_Region
import Graf.Plots as plt

```

Listing 2.1: Librerías

Tras las librerías empleadas se encuentra la inicialización de las condiciones iniciales. En este caso, se definen como condiciones iniciales $r_0 = 1$ y $v_0 = 0$ para este oscilador libre.

```
#Initial Conditions
r0 = [1]
v0 = [0]
U0 = r0 + v0
T = 100
dt = [0.1, 0.01, 0.001]

```

Listing 2.2: Condiciones iniciales

A continuación, se mantiene el *boolean* que permite elegir si los *plots* se guardan o solo se muestran. Esta función está resultando muy útil puesto que, dada la correcta implementación de las funciones de los anteriores *Milestones*, con la comprobación del correcto funcionamiento de una esquema numérico en las nuevas funciones, se puede graficar todas las figuras si necesidad de controlar el proceso de los cálculos.

```
#Save Plots
Save = False # True Save the plots / False show the plots

```

Listing 2.3: selección guardado

Tras ello se definen las listas que contienen los diferentes esquemas numéricos, donde se puede observar el nuevo esquema numérico,*LeapFrog*.

```
    #Temporal_Schemes to use
    T_S = [Euler, RK4, Crank_Nicolson, Inverse_Euler, LeapFrog]
    T_S_plot = ["Euler", "RK4", "CN", 'Euler_inver', "LeapFrog"]
```

Listing 2.4: Esquemas temporales

Se inicializan los diccionarios que se emplearán para el almacenamiento de los datos junto con la lista de los $N$ que se emplearán. En este caso se inicializa también un diccionario para las regiones de estabilidad.

```
#Initiation of Dictionaries
t_dic = { }
U_dic = { }
S_R = { }
N = zeros((len(dt))

```

Listing 2.5: Inicialización de diccionarios

Se calculan las iteraciones necesarias junto con los tiempos que se emplearán en los cálculos de forma análoga al anterior *Milestone*.

```
#Iterations and times
for i in range(len(dt)):
        N[i] = int(T/dt[i])
        t_dic[str(i)] = linspace(0,T, int(N[i]))

```

Listing 2.6: definición tiempos e iteraciones

Por último, para la realización de los diferentes cálculos encomendados se emplea, al igual que en el *Milestone 2*, un doble bucle que recorre todos los quemas numéricos y todos os $\Delta t$ deseados. Tras ello, se implementa un segundo bucle que calcula, mediante la función *Stability_Region*, las diferentes regines de estabilidad. Tras ello, se grafica mediante una nueva función en el módulo *Plots.py*.

```
# # #Simulations
for j in range ( len(T_S_plot) ):
        for x in t_dic: # x is a str and U is creating new keys
                U_dic[x]= C_P(Linear_Oscilator, t_dic[x], U0, T_S[j])
                plt.Plot_CP(U_dic[x], t_dic[x], T, dt[int(x)], T_S_plot[j
   ], Save) # x return the value of the key
        plt.Plot_CP_all(U_dic,t_dic,T, dt, T_S_plot[j], Save)
        print(T_S_plot[j] + " calculado \n")

#Stability Region

for i in range( len(T_S_plot) ):
        S_R[i] = Stability_Region(T_S_plot[i])
        plt.Plot_SR(S_R[i], dt, T_S_plot[i], Save)

```

Listing 2.7: Integración numérica y región de estabilidad

## 2.2. Physics.py

Se define el problema objetivo de este *Milestone* como un oscilador lineal. Este oscilador lineal tiene la siguiente forma:

$$\ddot{x} + x = 0 \tag{2.2.1}$$

Se define el problema de Cauchy para la integración numérica:

$$\frac{dU}{dt} = F(U, t) \tag{2.2.2}$$

Puesto que el vector posición se define como $\vec{r}(x)$ y la aceleración como $\dot{\vec{r}}(\dot{x})$:

$$U = \begin{pmatrix} x \\ \dot{x} \end{pmatrix} \tag{2.2.3}$$

Se obtiene que $F(U, t)$ como:

$$F(U, t) = \begin{pmatrix} \dot{x} \\ -x \end{pmatrix} \tag{2.2.4}$$

Por lo tanto, la programación de este problema resulta de la siguiente forma:

```
1
2 def Linear_Oscilator(U,t):
3
4     return array([U[1], -U[0]])
5
```

Listing 2.8: Definición problema Oscilador lineal

## 2.3. R_S.py

Como se ha descrito en la Sección 1, se implementa este módulo para el cálculo de las regiones de estabilidad. En un primer lugar, se definen los módulos que se van a emplear.

```
1 from numpy import sqrt, linspace, zeros, absolute, array
```

Listing 2.9: librerías para R_S.py

### Función Stability_Region

Se sigue de la función *Stability_Region*, en la cual se definen dos vectores de $N$ componentes, uno para números reales y otro para imaginario. Una vez definidos estas dos series de números, en un primer intento se intento realizar la obtención de $\omega$ mediante un simple bucle, lo cual presenta dos problemas: El primero de ellos que se impide el empleo de la función que se emplea para graficar dicha región, pues requiere un *array* $2-D$ y no $1-D$. Derivado de este problema se deduce un segundo problema: dado que se realiza en un simple bucle, no se garantizan todas las combinaciones posibles para $\omega$.

Tras la obtención de $\omega$, se busca obtener todos los $|r|$ de derivan de dichas $\omega$ a través de los diferentes esquemas numéricos, los cuales se obtienen de la siguiente función.

```
def Stability_Region(T_S):
    N = 100
    R = linspace(-5,5,N)
    I = linspace(-5,5,N)
    w = zeros([N, N], dtype = complex)

    # for i in range(N):
    #     w[i] = complex(R[i], I[i])

    for i in range(N):
        for j in range(N):
            w[j,i] = complex(R[i], I[j])
    return absolute(Stability_polynomial(T_S, w))

```

Listing 2.10: Función Stability_Region

### Stability_polynomial

En esta función simplemente se definen los diferentes polinomios de estabilidad para los diferentes esquemas numéricos. Todos estos esquemas numéricos han sido obtenidos siguiendo la memo-técnica explicada en clase, a excepción de Runge-Kutta de orden 4, el cual ha sido obtenido del libro "*Cálculo Numérico en Ecuaciones Diferenciales Ordinarias*".

```
def Stability_polynomial(T_S, w):

    if T_S == 'Euler':
        return 1 + w
    elif T_S == 'RK4':
        return  1 + w + w**2/2 + w**3/6 + w**4/24
    elif T_S == 'CN':
        return  (2+w) / (2-w)
    elif T_S == 'Euler_inver':
        return 1 / (1-w)
    elif T_S == 'LeapFrog':
        return sqrt(1)
```

Listing 2.11: Función Stability_polynomial

## 2.4. Plots.py

Para graficar las regiones de estabilidad es necesario la implementación de una nueva función que empleará una función de la librearía *matplotlib* que permite graficar el contorno deseado de una serie de datos, llamado en esta función "*levels*".

Como se se puede ver a continuación, se implementa la función *Plot_SR*. Como comentario, se ve que se hace necesario separar el caso del *LeapFrog*, dado que el valor devuelto por

la función *Stability_ Region* no es un *array*. El resto de la función es simple, se emplea la función *contour* y *contourf* para que grafique aquellos valores cuyo $|r| = 1$ y resalte el área donde ello ocurre, $levels = [0, 1]$

```
def Plot_SR(r, dt, T_S, Save):
    ax = plt.figure(figsize = (10, 10))
    if T_S == 'LeapFrog':
        Im = linspace(-1,1,100)
        Re = zeros(100)

        ax = plt.plot(Re, Im, color = '#0013ff')

    else:
        N = len(r)
        x = linspace(-5,5,N)
        y = linspace(-5,5,N)
        ax = plt.contour(x,y, r, levels = [0, 1], colors = ['#0013ff'])
        ax = plt.contourf(x,y, r, levels = [0, 1], colors =['#626262'])

    colors = ['r','orange','g']

    for i in range(len(dt)):
        plt.plot([0,0], [dt[i],-dt[i]], 'o', color = colors[i], label = "
   Oscilator's Roots by dt " + str(dt[i]))
    plt.ylabel("Im")
    plt.xlabel("Re")
    plt.legend()
    plt.grid()

    file = 'Stability Region of ' + T_S + ".png"
    Save_plot(Save, file)
```

Listing 2.12: Función para graficar las regiones de estabilidad

# 3. Resultados

Por último, se exponen los resultados obtenidos mediante el código expuesto en este informe. Todos estos resultados han sido obtenidos para un $T = 30$ s.

### Euler

En primer lugar se expone los resultados de la integración mediante el esquema numérico de Euler, Figura 3.1a. Se puede observar que para $\Delta t = 0{,}1$ se produce una amplificación del valor de $x$ con respecto del tiempo. Esto es debido al error que arrastra el método para ese $\Delta t$. Esto es comprobable, pues como se puede apreciar en la región de estabilidad, Figura 3.1b, cuanto menor es $\Delta t$ más estable es la solución del método.

Del mismo modo, se puede comprobar que se produce la amplificación por el error del método, pues en el problema de Caunchy, al igual que ocurría con el problema de Kepler,

mantiene la Energía constante. Como se aprecia, el valor de $x$ aumenta progresivamente, no conservandose la energía.

### Runge-Kutta orden 4

Como se puede observar en la Figura 3.2a, a diferencia del resultado anterior, este error se encuentra acotado dada la estabilidad del método, observable en la Figura 3.2b. Se observa que se encuentra en la región de estabilidad, pero no indefinidamente.

### Crank-Nicolson

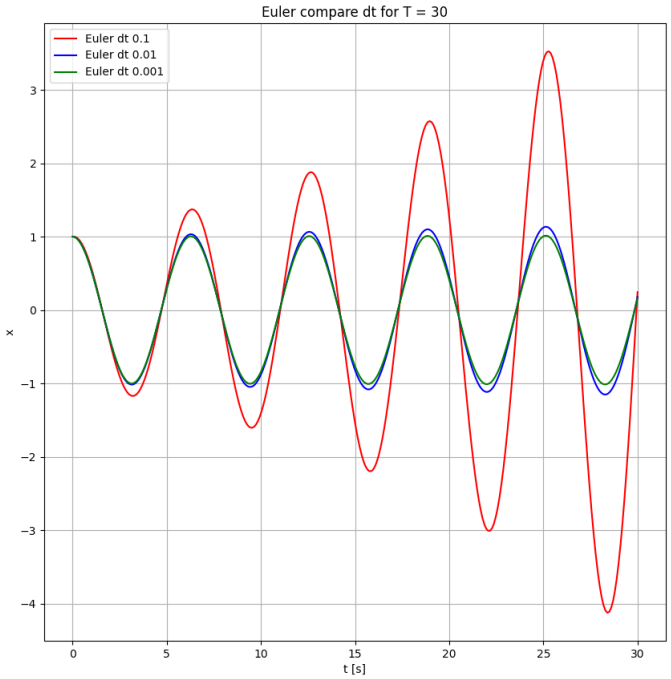
Al igual que en RK4, se obtiene resultados con un error muy acotado (Figura 3.3a). Pero a diferencia de este, como se puede observar en la 3.3b, la región de estabilidad es todo el semiplano negativo de los números reales. Esto le concede grandes propiedades de estabilidad. Las raíces de este oscilador se mantienen en todo momento sobre la frontera de la región de estabilidad.

### Euler Inverso

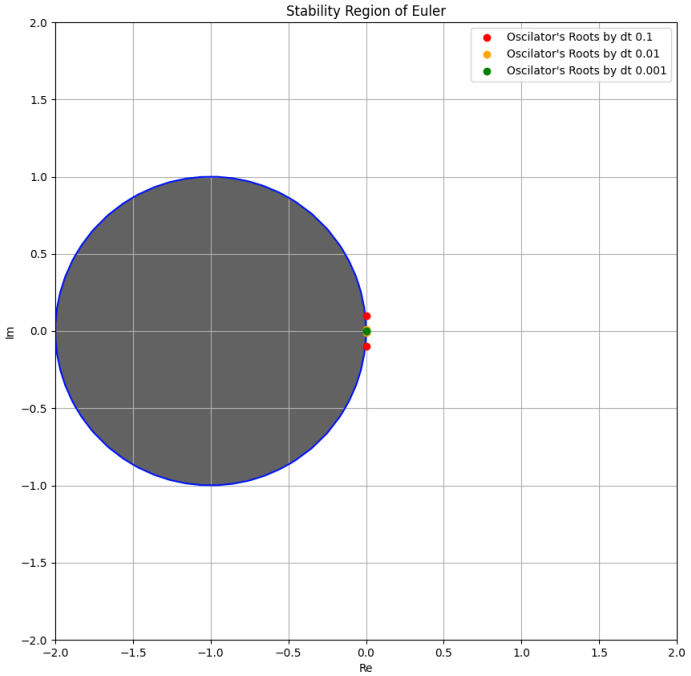
Al igual que en el caso de integración de Euler, la región de estabilidad para Euler Inverso es una circunferencia, pero en este caso en el semieje positivo de los números reales. Como se puede observar en la Figura 3.4a, a medida que se disminuye $\Delta t$, disminuye el error del método. Como se visualiza en la Figura 3.4b, cuanto menor $\Delta t$, más cerca de la frontera se encuentran las raíces del oscilador lineal, otorgando mayor precisión y estabilidad a los resultados.

### LeapFrog

Por último, se encuentra el nuevo método implementado, *LeapFrog*. Como se aprecia en la Figura 3.5a, los resultados tienen una buena estabilidad a partir de $\Delta t = 0{,}01$. Como se puede apreciar en la Figura 3.5b, su región de estabilidad es una recta desde $(0, -1)$ hasta $(0, 1)$. Dado este hecho, se puede deducir que este método no presenta tanta estabilidad como en RK4 o en CN.

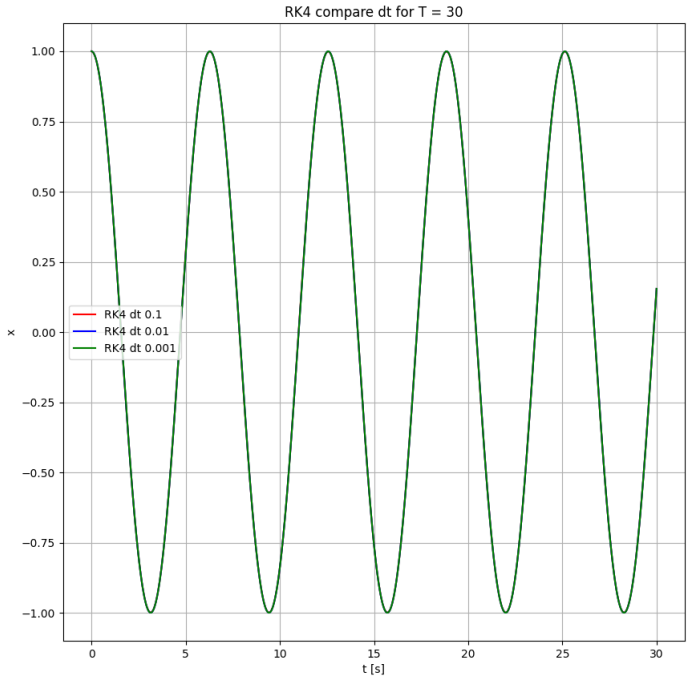



(a) Integración Euler para diferentes $\Delta t$

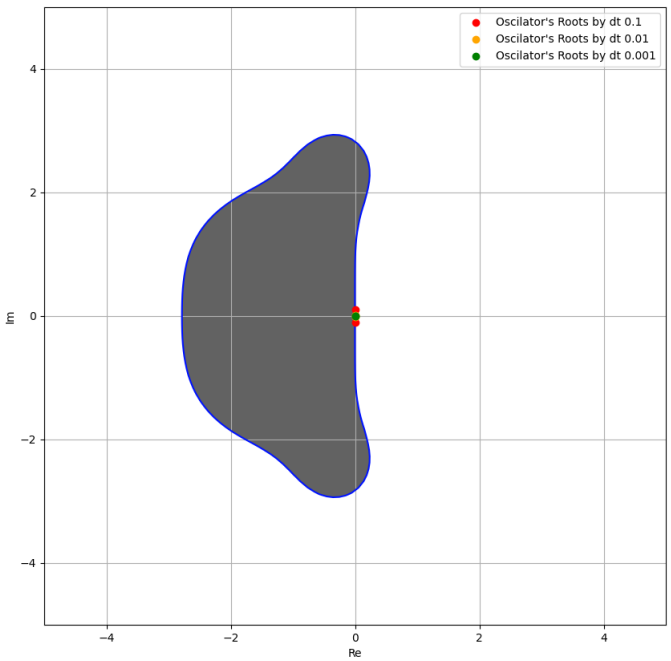



(b) Región de estabilidad de Euler

Figura 3.1: Resultados Euler

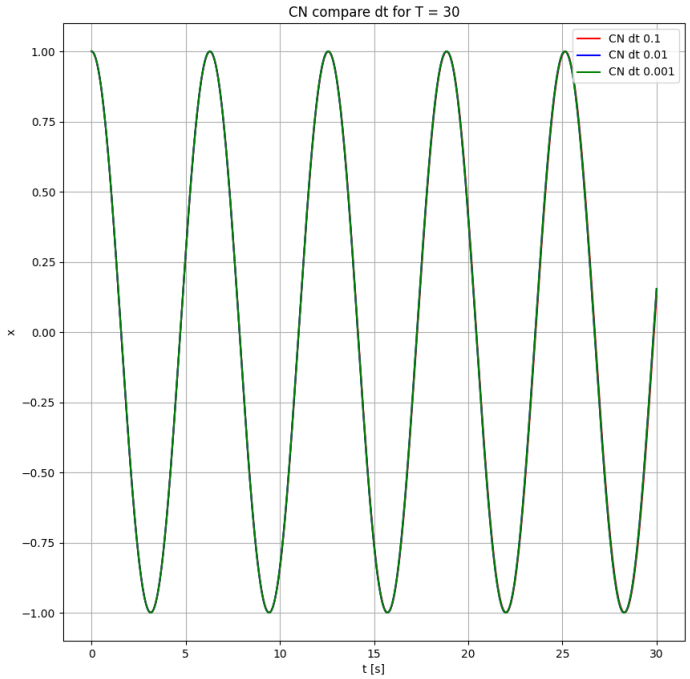



(a) Integración RK4 para diferentes $\Delta t$

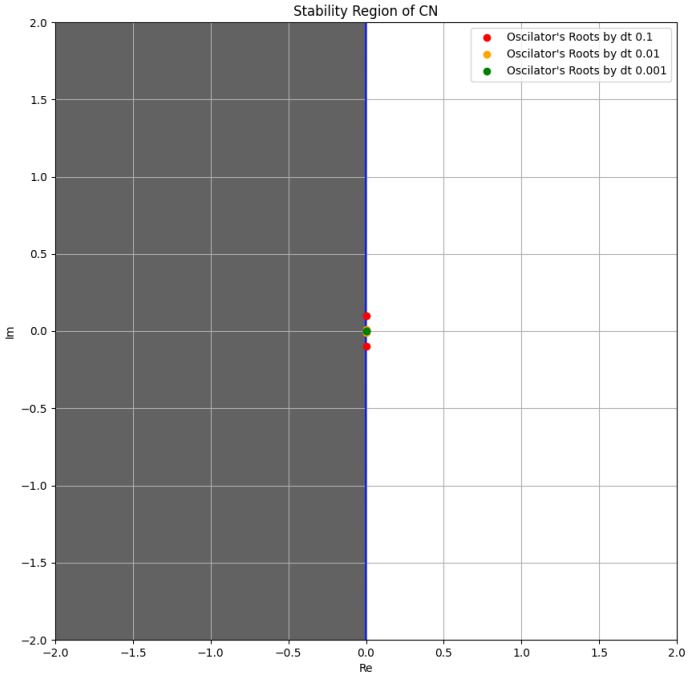



(b) Región de estabilidad de Rk4

Figura 3.2: Resultados Runge-Kutta orden 4

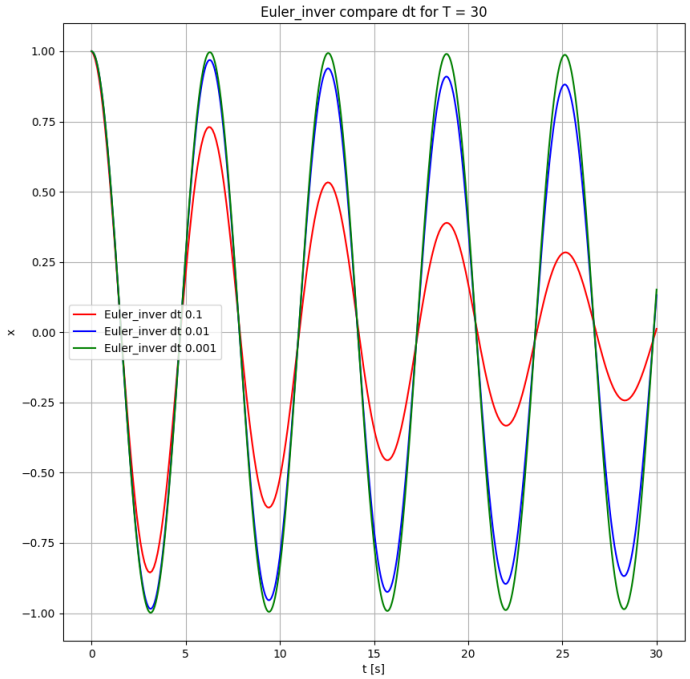



(a) Integración CN para diferentes $\Delta t$

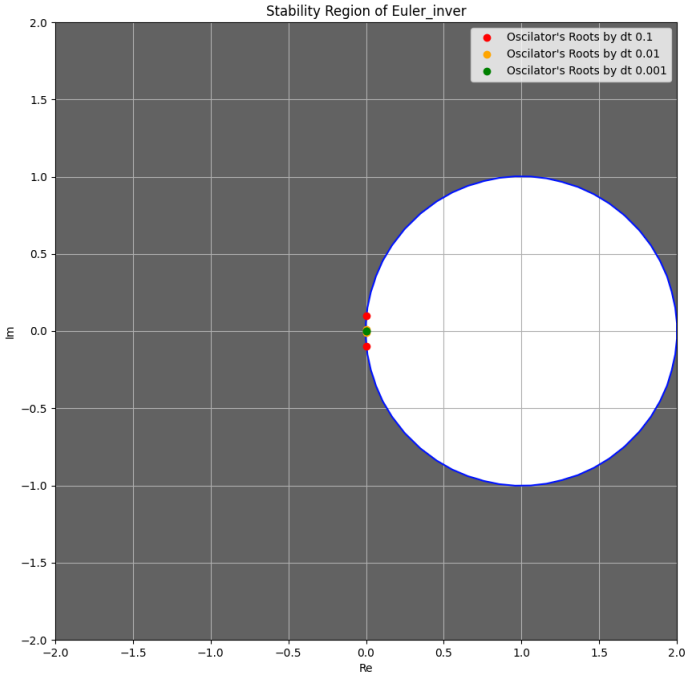



(b) Región de estabilidad de CN

Figura 3.3: Resultados Crank-Nicolson

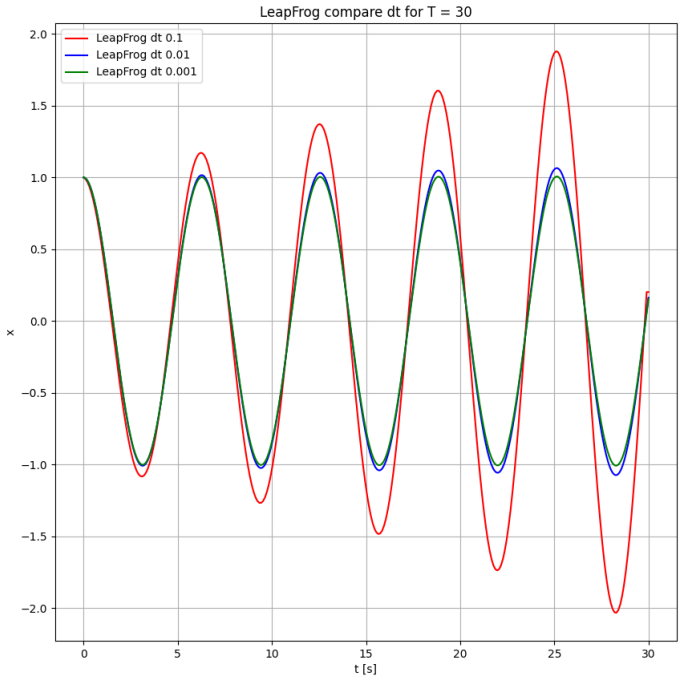



(a) Integración Euler inverso para diferentes $\Delta t$

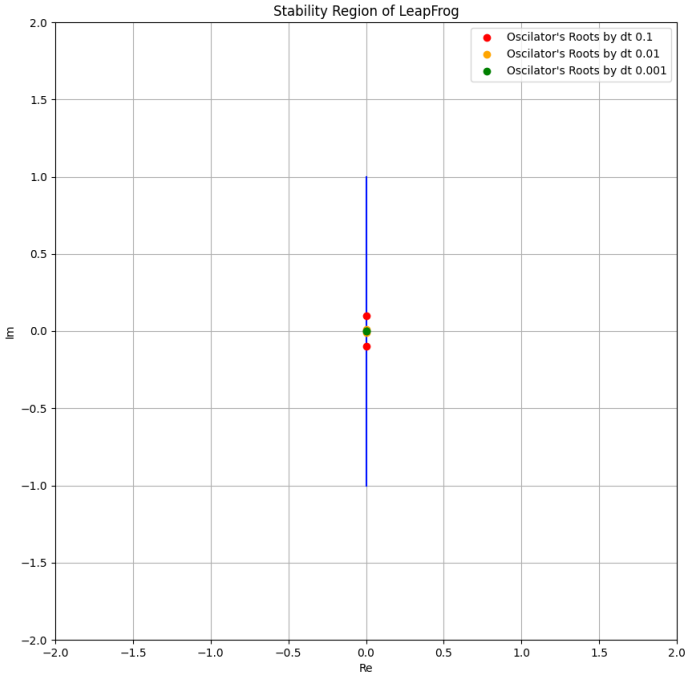



(b) Región de estabilidad de Euler inverso

Figura 3.4: Resultados Euler inverso




(a) Integración LeapFrog para diferentes $\Delta t$




(b) Región de estabilidad de LeapFrog

Figura 3.5: Resultados LeapFrog