Física Numérica

Tarea #4

D. A. Vázquez Gutiérrez

Escuela Superior de Física y Matemáticas, Instituto Politécnico Nacional,Unidad Profesional .ᴬdolfo López Mateos", Zacatenco, Edificio 9, Col. San Pedro Zacatenco, C.P. 07730 del. Gustavo A. Madero, Ciudad de México, México

email: dvazquezg1600@alumno.ipn.mx

28 de junio de 2024

1. Lanzamiento de martillo.

Partamos del siguiente problema particular :

El récord mundial para hombres en lanzamiento de martillo es de 86.74 m por Yuri Sedykh y se ha mantenido desde 1986. El martillo pesa 7.26 kg, es esférico, y tiene un radio de $R = 6cm$. La fricción en el martillo puede ser considerada proporcional al cuadrado de la velocidad del martillo relativa al aire:

$$F_D = \frac{1}{2}\rho A C_D \nu^2 \tag{1}$$

Donde ρ es la densidad del aire ($1{,}2\frac{Kg}{m^3}$) y $A = \pi R^2$ es la sección transversal del martillo. El martillo puede experimentar, en principio, un *flujo laminar* con coeficiente de rozamiento $C_D = 0{,}5$ o un flujo *flujo inestable oscilante* con $C_D = 0{,}75$.

(a) **Programa**. Escribamos las ecuaciones diferenciales del sistema utilizando las leyes de newton :

$$m\frac{d^2X_x}{dt^2} = -\frac{1}{2}\rho A C_D V_x^2$$

$$m\frac{d^2X_y}{dt^2} = \begin{cases} -\left(\frac{1}{2}\rho A C_D V_y^2 + mg\right) & \text{si } V_y < 0 \\ \\ \frac{1}{2}\rho A C_D V_y^2 - mg & \text{si } V_y > 0 \end{cases}$$

Estas ecuaciones podemos escribirlas en forma de cuatro ecuaciones de primer orden

$$\frac{dV_x}{dt} = -\frac{K}{m}V_x^2 \tag{2}$$

$$\frac{dX_x}{dt} = V_x \tag{3}$$

$$\frac{dV_y}{dt} = \begin{cases} -\left(\frac{K}{m}V_y^2 + mg\right) & \text{si } V_y < 0 \\ \\ \frac{K}{m}V_y^2 - mg & \text{si } V_y > 0 \end{cases} \tag{4}$$

$$\frac{dX_y}{dt} = V_y \tag{5}$$

Donde :

- V velocidad
- X posición
- t tiempo

- x eje x
- y eje y

Con :

$$K = \frac{1}{2}\rho A C_D$$

Empleando estas ecuaciones , creamos el Código 1, con nombre `Velocidad_rozamiento.py`, que se encuentra en A.1, donde para la primera parte especificamos las constantes a utilizar, las condiciones iniciales de nuestros arreglos , como lo son que $X_{xo} = 0$ y $X_{yo} = 2$, así como que el angulo en que se lanza es de $\phi = 45$.

Posteriormente , dado que se nos pedirá mas adelante el ver el comportamiento de la funcion que obtendremos con y sin fricción ,entonces definimos de ambas formas las ecuaciones diferenciales 4 a la 7 .

Luego definimos las funciones que nos darán acceso a los algoritmos de Euler así como al de Runge-Kutta , con los que podremos obtener la función solución a las ecuaciones diferenciales previamente escritas. Inmediatamente después y muy relacionado a esto definimos una función llamada "Mapeo ", la cual nos permite encontrar el valor que cierto valor de t tiene en alguna función que encontremos con los métodos antes comentados.

Uno de los puntos de interes es el encontrar para que valor de V, X_x es máximo, donde esto se da cuando para algun t_f , se cumple que :

$$X_y(t_f) = 0$$

Entonces , $X_x(t_f)$ es la distancia máxima a recorrer. Claramente por la ecuación anterior , y dado que nuestra función fue obtenida de forma puramente analítica, encontrar la raiz de nuestra función también tendrá que ser utilizando métodos numéricos .

El mas útil y popular es el método de Newton Rapson, sin embargo , tiene un detalle; Para poder ser implementado sin problema alguno , este debe de inicializarse en un punto de t relativamente cercano a la raíz, de lo contrario es poco efectivo.

Por ello implementamos este algoritmo para encontrar raíces junto a otro muy utilizado, pero algo mas básico , el Método de Bisección , el cual nos permite auscultar el valor para inicializar Newton Rapson.

Seguimos presentando las funciones f y *distanciamax*, la primera asigna a los arreglos con valores en ceros sus valores respectivos en función de los algoritmos como RungenKutta así como de los valores iniciales de estos , la segunda , *distanciamax*, emplea las ideas de la ultima ecuación planteada para encontrar la distancia máxima .

Por ultimo ,tenemos *VoXdeseada* , la cual es una implementación de la idea general de Bisección , pero implementada a la búsqueda de la Velocidad inicial que detone una distancia máxima que busquemos .

La demás parte del código unicamente es la implementación grafica para obtener las imágenes necesarias posteriormente

(b) **Tiempo y velocidad para el record mundial con distinto valores de fricción**. Nos piden ahora que caculemos las graficas tanto el tiempo de altitud del martillo asi como la trayectoria $X_y = X_y(X_x)$. Añado a esto la velocidad inicial nesesaria para obtener el record de Yuri Sedykh , para los tres regimenes , Sin friccion , Flujo laminar y flujo turbulento oscilante. Los resultados pueden encontrarse rapidamente corriendo el programa Codigo 1.

- Sin Fricción :

  Donde el tiempo de vuelo es:

$$t_{fSF} = 4{,}2685s \tag{6}$$

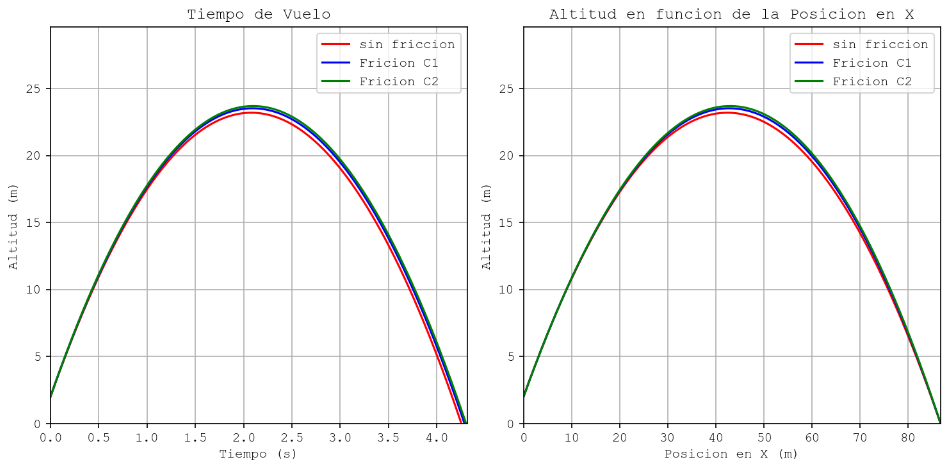

Figura 1: **Altitud**. Usando el código 1,vemos a la altitud (y), en función del tiempo y de la posición

Así como la velocidad para llegar a los 84.74 metros es :

$$V_{SF} = 28{,}822\frac{m}{s} \tag{7}$$

- Flujo Laminar :
  Donde el tiempo de vuelo es:

$$t_{fCD1} = 4{,}30401s \tag{8}$$

Así como la velocidad para llegar a los 84.74 metros es :

$$V_{CD1} = 29{,}19555\frac{m}{s} \tag{9}$$

- Flujo inestable oscilante :
  Donde el tiempo de vuelo es:

$$t_{fCD2} = 4{,}3183s \tag{10}$$

Así como la velocidad para llegar a los 84.74 metros es :

$$V_{CD2} = 29{,}3933\frac{m}{s} \tag{11}$$

(c) **Conclusión**. A simple vista, todas las graficas tienen una forma similar , sin embargo , viendolas detalladamente podemos notar como cuando hay una fuerza de friccion , sea cual sea, a mayor el tiempo o la distancia , la forma pasa de ser curva a se una linea recta, indicando la presencia de una *Velocidad terminal* .

La diferencia entonces entre *flujo laminar* y *Flujo turbulento oscilante* no es mas que la rapidez con la que se presenta este 'enrectamiento' de la curva.

De igual forma , a mayor cantidad del coeficiente C_D , mas dificil resulta al martillo moverse, por lo que para llegar a la misma distancia , requiere una mayor cantidad de velocidad inicial e incesantemente tiene mas tiempo de vuelo .

2. Masa y Resorte

Consideramos el sistema de resortes que se muestra en la figura :

(a) **Ecuaciones de modos normales. Caso lineal**. Entonces, las ecuaciones de movimiento de cada partícula son :

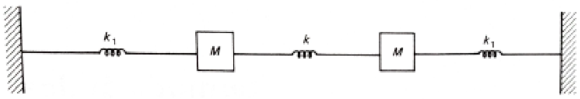

Figura 2: Graficas Altitud flujo turbulento oscilante

$$m\frac{d^2x_1}{dt^2} + (k_1 + k)x_1 = kx_2 \qquad (12)$$

$$m\frac{d^2x_2}{dt^2} + (k_1 + k)x_2 = kx_1 \qquad (13)$$

(b) **Frecuencia de modos normales. Caso lineal** . Para encontrar los modos normales, veamos que tenemos dos casos en los que el sistema de la figura 4 puede comportarse realmente como un movimiento armónico simple, ya que las ecuaciones 12 y 13 realmente no lo son.

- **Movimiento en fase** : Entonces ambas masas se mueven en misma dirección y con misma amplitud, esto hace que el resorte k no perciba ningún tipo de deformidad, luego:

$$m\frac{d^2(x_2 + x_1)}{dt^2} + (k_1)(x_2 + x_1) = 0 \qquad (14)$$

Resolviendo la ecuación diferencial , entonces tenemos:

$$\begin{aligned} x_a &= x_2 + x_1 \\ &= C_1cos(\omega_a t) + C_2sen(\omega_a t) \end{aligned}$$

donde :

$$\omega_a = \sqrt{\frac{k_1}{m}}$$

- **Movimiento en contra fase** : Entonces ambas masas se mueven en direcciones contrarias y con misma amplitud, esto hace que el resorte $k1$ se elongue o comprima , en sus puntos máximos , el doble , luego :

$$m\frac{d^2(x_2 - x_1)}{dt^2} + (k_1 + 2k)(x_2 - x_1) = 0 \qquad (15)$$

Resolviendo la ecuación diferencial , entonces tenemos:

$$\begin{aligned} x_a &= x_2 - x_1 \\ &= D_1cos(\omega_b t) + D_2sen(\omega_b t) \end{aligned}$$

donde :

$$\omega_b = \sqrt{\frac{k_1 + 2k}{m}}$$

(c) **Frecuencia de modos normales. Caso no lineal** . Veamos que en el caso de que los resortes tengan una constante k no lineal , sea esta de la forma

$$F = k(x + 0{,}1x^3)$$

Entonces, podemos encontrar que las ecuaciones diferenciales en modos normales de este caso son :

- **Movimiento en fase** :

$$m\frac{d^2(x_2 + x_1)}{dt^2} + (k_1)(x_2 + x_1 + 0{,}1(x_2^3 + x_1^3)) = 0 \qquad (16)$$

- **Movimiento en contra fase** :

$$\begin{aligned} & m\frac{d^2(x_2 - x_1)}{dt^2} + \\ & (k_1)((x_2 - x_1) + 0{,}1(x_2^3 - x_1^3)) \\ & = 0 \end{aligned} \qquad (17)$$

(d) **Programa**. Ahora queremos encontrar numéricamente la frecuencias normales , tanto para resortes lineales como para resortes no lineales , por esto creamos código 2, con nombre `Masa y Resorte.py`, que se encuentra en la seccion A.2.

Este programa , al menos en la base de plantear una ecuación diferencial y generarla , es bastante similar a la sección 1. El principal problema se genero al intentar encontrar la frecuencia angular ω de los modos normales, esto lo terminamos logrando al utilizar de la función *Newton Rapson*, llevada un poco mas lejos , ya que al ser soluciones de una ecuación de movimiento armónico simple, entonces habría varias raíces de nuestra función por se esta oscilante. Para encontrar nuestra frecuencia angular entonces , debemos de encontrar dos raíces continuas, para to ultilizamos una funcion *while* que ira buscando , primero, la raiz mas cercana a el extremo izquierdo comenzando desde el 0 hasta un *tau* que es el extremo derecho del dominio *tiempo*.

Teniendo dos raices continuas , sea x y x_o , entonces:

$$\pi n = \omega x$$
$$\pi(n-1) = \omega x_o$$

Luego entonces , desarrollando :

$$\omega = \frac{\pi}{x - x_o}$$

Ademas notando que la distancia $x - x_o$ es igual a $T/2$ medio periodo.

Con esto mencionado , podemos encontrar los valores de las frecuencias angulares de los sistemas en fase o en contrafase , siendo o no lineales y tomando $k = 0{,}75$, $k_1 = 0{,}3$,$m = 1{,}2kg$ así como las velocidades iniciales $x_1o = 2m$ y $x_2o = 0m$ entonces:

Cuadro 1: Frecuencias Normales con K lineal

Frecuencias Normales con K lineal		
	Teórica	**Numérica**
Fase	0.5	0.50011
Contrafase	1.22474	1.22477

Entonces, con esto vemos que los errores relativos entre ambos casos , lineal y no lineal son relativamente pequeñas , con un error poco mas del 10 % como puede verse en el cuadro 3.

Cuadro 2: Frecuencias Normales con K no lineal

Frecuencias Normales con K no lineal	
	Numérica
Fase	0.5693274421688148
Contrafase	1.3951705457487051

Cuadro 3: Diferencias entre ω de K lineales y no lineales

Diferencias entre ω de Ks lineales y no lineales		
	Error absoluto	**Error relativo**
Fase	0.06921	0.1384
Contrafase	0.1704	0.122137

(e) **Graficas**. Ahora veamos como evolucionan los movimientos de x1 y x2 para tres casos diferentes , como se ve en las figuras 5, 6 y 7 :

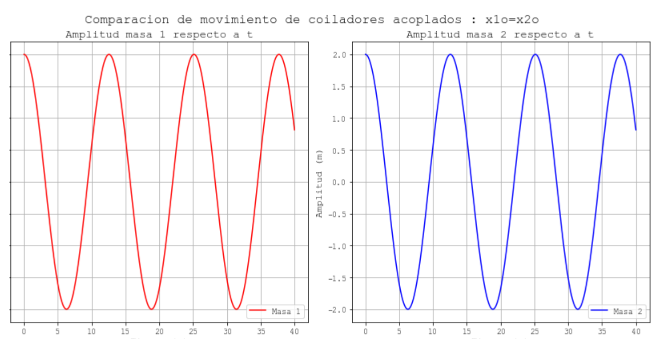

Figura 3: Caso $X1o = X2o$

3. Vibración de una cuerda

Este problema pretende estudiar las oscilaciones de una cuerda. Considere una cuerda de longitud

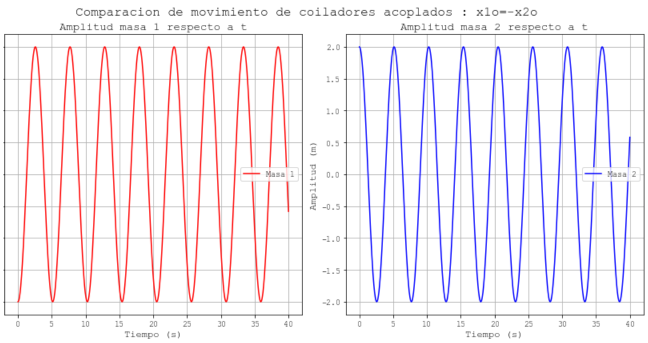

Figura 4: Caso $X1o = -X2o$

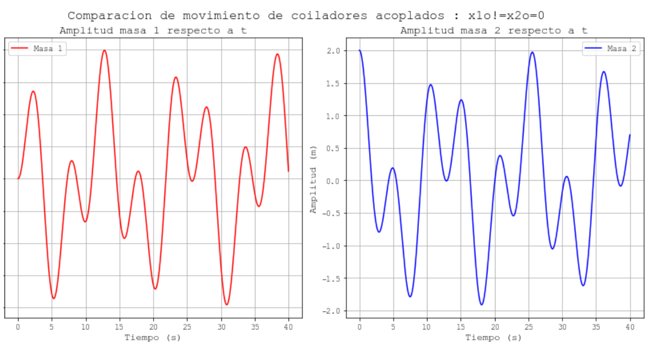

Figura 5: Caso $X2o \neq X1o = 0$

L y densidad $\rho(x)$ por unidad de longitud, atada en ambos extremos y bajo una tensión $T(x)$. Suponga que el desplazamiento relativo de la cuerda respecto a su posición de equilibrio $\frac{y(x,t)}{L}$ es pequeño y que la pendiente de la cuerda $\frac{\partial y}{\partial x}$ tambien es pequeña.

(a) Considere una sección infinitesimal de la cuerda, como se muestra en la figura, note que la diferencia en las componentes de las tensiones en x y $x + \Delta x$ tiene por resultado una fuerza restauradora.

Entonces , podemos notar que el consiente de las fuerzas en el eje y y en el eje x debe ser igual a la pendiente de la cuerda , de tal forma que :

$$\frac{T_{1y}}{T_x} = -\left(\frac{\partial y}{\partial x}\right)_x$$

$$\frac{T_{2y}}{T_x} = \left(\frac{\partial y}{\partial x}\right)_{x+\Delta x}$$

Entonces , la fuerza en el eje y es :

$$T_y = T_{1y} + T_{2y} = T_x\left(\left(\frac{\partial y}{\partial x}\right)_{x+\Delta x} - \left(\frac{\partial y}{\partial x}\right)_x\right)$$

Pero por otro lado sabemos que por la segunda ley de Newton:

$$T_y = ma = \rho(x)dx\frac{\partial^2 y}{\partial t^2}$$

Derivando sobre x ambas partes de la ecuación, tenemos :

$$\frac{dT_x}{dx}\left(\left(\frac{\partial y}{\partial x}\right)_{x+\Delta x} - \left(\frac{\partial y}{\partial x}\right)_x\right) + T_x\left(\frac{\partial^2 y}{\partial x^2}\right) = \rho(x)\frac{\partial^2 y}{\partial t^2}$$

Pero notemos lo siguiente :

$$\left(\frac{\partial y}{\partial x}\right)_{x+\Delta x} - \left(\frac{\partial y}{\partial x}\right)_x =$$

$$\frac{y(x+2\Delta x) - y(x+\Delta x)}{\Delta x} - \frac{y(x+\Delta x) - y(x+)}{\Delta x} =$$

$$\frac{y(x+2\Delta x) - y(x)}{\Delta x} \approx \left(\frac{\partial y}{\partial x}\right)_x$$

Entonces, concluimos:

$$\frac{dT_x}{dx}\left(\frac{\partial y}{\partial x}\right)_x + T_x\left(\frac{\partial^2 y}{\partial x^2}\right) = \rho(x)\frac{\partial^2 y}{\partial t^2} \qquad (18)$$

Que es lo que queríamos encontrar.

(b) Notemos que si la tension en el eje x T_x es constante , entonces:

$$\left(\frac{\partial^2 y}{\partial x^2}\right) = \frac{1}{c^2}\frac{\partial^2 y(x,t)}{\partial t^2}, \quad c = \sqrt{\frac{T_x}{\rho(x)}} \qquad (19)$$

(c) Hay dos condiciones que deben cumplirse para tener una única solución a esta EDP de segundo orden

1) Condiciones Iniciales.
Estas condiciones describen el estado de la onda en el tiempo $t = 0$

- Condición inicial de desplazamiento:

$$u(x, 0) = f(x),$$

donde $f(x)$ es una función que define la velocidad inicial de la onda.

- Condición inicial de velocidad:

$$\frac{\partial u}{\partial t}(x, 0) = g(x),$$

donde $f(x)$ es una función que define la velocidad inicial de la onda.

2) Condiciones Iniciales.
Estas condiciones describen el comportamiento de la onda en los límites espaciales del dominio.

- Condición de frontera de Dirichlet :

$$u(0, t) = h_1(t), u(L, t) = h_2(t)$$

donde $h_1(t)$, y $h_1(t)$son funciones que definen el valor de la onda en los extremos del dominio espacial $x = 0$ y $x = L$.

- Condición de frontera de Neumann :

$$\frac{\partial u}{\partial x}(0, t) = h_3(t), \frac{\partial u}{\partial x}(L, t) = h_4(t)$$

donde $h_3(t)$, y $h_4(t)$son funciones que definen el valor de la derivada espacial de la onda en los extremos del dominio espacial $x = 0$ y $x = L$.

(d) Ahora empleando una malla de pasos de longitud Δt en el tiempo y Δx en el espacio para obtener una solución numérica.

Partimos entonces expresando las segunda derivadas en terminos de diferencias finitas . Por lo visto al hacerlo en la ecuacion de Poisson , para unϕ y s arbitrarios, tenemos que :

$$\frac{\partial^2 \phi}{\partial s^2} \approx \frac{\phi(x + \Delta x, y) + \phi(x - \Delta, y) - 2\phi(x, y)}{(\Delta x)^2}$$

Sustituyendo ϕ por y y s por x, t en la ecuación (19) , entonces:

$$\frac{y(x + \Delta x, t) + y(x - \Delta, t) - 2y(x, t)}{(\Delta x)^2} =$$

$$\frac{1}{c^2} \frac{y(x, t + \Delta t) + y(x, t - \Delta t) - 2y(x, t)}{(\Delta t)^2}$$

Ahora discretizando el espacio con pasos igualmente espaciados , entonces:

$$x_i = x_0 + i\Delta, \quad y_j = y_0 + j\Delta, \quad i.j = 0, ..., N_{max} - 1$$

Denotando :

$$\phi_{i,j} = \phi(x_i, y_j)$$

De esta forma llegando a la ecuación de onda por diferencias :

$$\frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2} = \frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{c^2 (\Delta t)^2} \tag{20}$$

Entonces, haciendo un poco de álgebra:

$$y_{i+1,j} + y_{i-1,j} - 2y_{i,j} = \frac{(\Delta x)^2}{(\Delta t)^2} \frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{c^2}$$

Si $\frac{\Delta x}{\Delta t} = c'$ es la velocidad de la malla, entonces:

$$y_{i+1,j}+y_{i-1,j}-2y_{i,j} = \frac{c'^2}{c^2}[y_{i,j+1}+y_{i,j-1}-2y_{i,j}]$$

Y finalmente:

$$\begin{aligned} y_{i+1,j} = \; & 2y_{i,j} - y_{i-1,j} + \\ & \frac{c'^2}{c^2}[y_{i,j+1} + y_{i,j-1} - 2y_{i,j}] \end{aligned} \tag{21}$$

o análogamente

$$\begin{aligned} y_{i,j+1} = \; & 2y_{i,j} - y_{i,j-1} + \\ & \frac{c^2}{c'^2}[y_{i+1,j} + y_{i-1,j} - 2y_{i,j}] \end{aligned} \tag{22}$$

(e) En la ecuación (21), las condiciones iniciales se establecen siempre que un indice j tenga como valor el 0 , entonces , $y_{i,0}$ para un i arbitrario, tomara el valor de la condición inicial de desplazamiento, por otro lado , cuando el indice i valga 0 entonces, $y_{0,j}$ para un j arbitrario , tomara el valor de la condición de frontera de Dirichlet.

(f) La condición de Courant para la estabilidad de la solución es que:

$$\frac{c}{c'} \leq 1 \tag{23}$$

Entonces podemos interpretar de esto que la velocidad de fase de la onda siempre debe ser menor a la velocidad de la malla , ademas esto nos da un impedimento al hacer la malla , ya que entonces siempre que modifiquemos los pasos espaciales también hay que hacerlo con los temporales proporcionalmente para mantener esta relación.

Creamos el siguiente programa, el codigo 3, con nombre `GaussSeideleCOnda2.py`, que se encuentra en la sección A.3, para representar la ecuación de onda con una animación con nombre `animacion1.gif`.

El desarrollo de este programa se hizo análogo al ultimo de la tarea 3, en el cual utilizábamos el método de Gauss Seidel para generar una malla que se fuera mejorando en cada iteración, cosa que hacemos aquí también pero que modificamos para obtener solo algunos fotogramas de esta malla y así poder hacer la animación.

Uno de los puntos mas interesantes y en el que se le invirtió tiempo extra al programa fue de la linea 68 a la linea 76.

Aquí se aplica la ecuación (22), sin embargo , había constantemente problemas al generarse las funciones , ya que aun que no se modificaba en ningún los bordes de la malla después de que fueran inicializados, el programa modificaba erróneamente estos , haciendo que el método quedara truncado y diera resultados bastante raros .

Se noto es que por como esta la ecuacion (22), en la fraja cercana a las condidiciones iniciales, esta no se puede realizar por el paso $y_{i,j-1}$ entonces , se incorporo un paso solo en esa region para evitar ese "bache", teniendo buenos resultados .

En general , como se menciono antes, podemos ver que en `animacion1.gif` , el comportamiento es apropiado para lo que se espera de las condiciones iniciales y de frontera.

En `animacion1.gif` se cumple (23) con $c/c' = 0{,}96$, por otro lado generamos una segunda animación `animacion2.gif`,con $c/c' = 1{,}61$ con en la cual se puede ver como esta cumple con las condiciones iniciales, pero poco después la generación de la función falla y se desborda el valor de esta.

Apéndice

A. Código en Python

A.1. Búsqueda de la velocidad de lanzamiento de martillo necesaria para tener el récord mundial en distintos escenarios

```python
import matplotlib.pyplot as plt
import numpy as np
import math

# --CONSTANTES -----------------
N = 100000   # Numero de pasos
xx0 = 0      # Posicion inicial en x
xy0 = 2      # Posicion inicial en y
v = 20       # Velocidad neta (modificar el codigo si se quiere una velocidad especifica como aqui)
angulo = 45
tau = 20     # Tiempo en segundos de la simulacion
h = tau / float(N-1)  # Paso del tiempo
g = 9.8      # Aceleracion 9.8 m/s**2

#---Obtener K ------
rho = 1.2    # Densidad del aire
R = 6 / 100
A = math.pi * R**2  # Area
m = 7.26     # Masa de la particula
xf = 86.74   # Distancia final
Cd1 = 0.5    # Flujo laminar
Cd2 = 0.75   # Flujo inestable oscilante
Cd = [0, Cd1, Cd2]

k = np.zeros(3)
for i in range(3):
    k[i] = ((rho * A) / 2) * Cd[i]  # Constante

# Entonces 0 es sin friccion, 1 Cd1, 2 Cd2

#-------Arreglos POSICION/ VELOCIDAD

# Generamos un arreglo de Nx2 para almacenar posicion y velocidad
y = np.zeros((N, 2))  # La primera columna sera la posicion y la segunda la velocidad en ese mismo punto
x = np.zeros((N, 2))  # La primera columna sera la posicion y la segunda la velocidad en ese mismo punto

# Generamos tiempos igualmente espaciados
tiempo = np.linspace(0, tau, N)  # El inicio siempre debe ser 0 porque es asi como inicializamos el programa

# Definicion de nuestras ecuaciones diferenciales con friccion
def EDO1(estado, tiempo, i):
    f0 = estado[1]
    f1 = -(k[i] / m) * estado[1]**2
    return np.array([f0, f1])

def EDO2(estado, tiempo, i):
    f0 = estado[1]
    if estado[1] >= 0:
        f1 = (-(k[i] / m) * estado[1]**2) - g
    else:
        f1 = ((k[i] / m) * estado[1]**2) - g
    return np.array([f0, f1])

# Metodo de Runge-Kutta
def RungeKutta(y, t, h, f, i=None):
    if i is not None:
        k1 = h * f(y, t, i)
        k2 = h * f(y + k1 / 2, t + h / 2, i)
        k3 = h * f(y + k2 / 2, t + h / 2, i)
        k4 = h * f(y + k3, t + h, i)
    else:
        k1 = h * f(y, t)
        k2 = h * f(y + k1 / 2, t + h / 2)
        k3 = h * f(y + k2 / 2, t + h / 2)
        k4 = h * f(y + k3, t + h)
    y_p = y + (k1 + 2 * k2 + 2 * k3 + k4) / 6
    return y_p

# Algoritmos utiles
t0 = tau / 2
dt = 3.e-3
err = 0.001
Nmax = 10000  # Parametros
Err0 = 0.015
Err01 = 0.015

def Mapeo(t0, t, x, s, err):
    for j in range(N-1):
        if abs(t[j] - t0) <= err:
            return x[j, s]
    print('\n No se encontro valor de la funcion en t=', t0)
    return 0

def bisection(a, b, err, max_iter, t, x, s):
    if Mapeo(a, t, x, s, err) * Mapeo(b, t, x, s, err) >= 0:
        raise ValueError("La funcion no cambia de signo en el intervalo dado.")

    iter_count = 0
    while (b - a) / 2 > err and iter_count < max_iter:
        c = (a + b) / 2  # Punto medio del intervalo
```

```
        if Mapeo(c, t, x, s, err) == 0:
            return c, iter_count
        elif Mapeo(c, t, x, s, err) * Mapeo(a, t, x, s, err) < 0:
            b = c  # La raiz esta en el subintervalo [a, c]
        else:
            a = c  # La raiz esta en el subintervalo [c, b]
        iter_count += 1

    return (a + b) / 2

def NewtonR(t, dt, x, err, Nmax, s):
    t0 = bisection(0, tau, Err0, 100, t, x, s)
    for it in range(Nmax):
        F = Mapeo(t0, t, x, 0, Err0)
        if F is not None and abs(F) <= err:
            break
        elif F is None:
            print('\n Valor de funcion es None en x=', t0)
            break
        df = (Mapeo(t0 + dt / 2, t, x, 0, Err0) - Mapeo(t0 - dt / 2, t, x, 0, Err0)) / dt  # Central difference
        if df == 0:
            break
        dt = -F / df
        t0 += dt
    if it == Nmax:
        print('\n Newton no encontro raiz para Nmax=', Nmax)
    return t0

# Obtencion de funciones
def f(t, x, y, v, angulo, i):
    y[0, 0] = xy0
    y[0, 1] = np.sin(np.radians(angulo)) * v
    x[0, 0] = xx0
    x[0, 1] = np.cos(np.radians(angulo)) * v
    for j in range(N-1):
        x[j+1] = RungeKutta(x[j], t[j], h, EDO1, i)
        y[j+1] = RungeKutta(y[j], t[j], h, EDO2, i)

def distancia_max(t, x, y):
    t_0 = NewtonR(t, dt, y, err, Nmax, 0)
    x_max = Mapeo(t_0, t, x, 0, Err0)
    return x_max

def Vo_Xdeseada(t, x, y, angulo, f, w, Xdeseada, err, max_iter, i):
    Vo_min = 20
    Vo_max = 200
    Vo_deseada = 0

    iter_count = 0
    while (Vo_max - Vo_min) / 2 > err and iter_count < max_iter:
        Vo_deseada = (Vo_max + Vo_min) / 2  # Punto medio del intervalo
        f(t, x, y, Vo_deseada, angulo, i)
        if distancia_max(t, x, y) == Xdeseada:
            return Vo_deseada, iter_count
        elif Xdeseada - distancia_max(t, x, y) < 0:
            Vo_max = Vo_deseada  # La raiz esta en el subintervalo [a, c]
        else:
            Vo_min = Vo_deseada  # La raiz esta en el subintervalo [c, b]
        iter_count += 1

    return (Vo_min + Vo_max) / 2

def Y_max(y):
    ymax = 0
    for j in range(N):
        if abs(y[j, 1]) < err:
            ymax = y[j, 0]
    return ymax

Velocida_champeon = np.zeros(3)
t_v = np.zeros(3)
x_max = np.zeros(3)
y_max = np.zeros(3)

xxdatos = [np.zeros(N), np.zeros(N), np.zeros(N)]
vxdatos = [np.zeros(N), np.zeros(N), np.zeros(N)]
xydatos = [np.zeros(N), np.zeros(N), np.zeros(N)]
vydatos = [np.zeros(N), np.zeros(N), np.zeros(N)]

for i in range(3):
    if i == 0:
        print('---SIN FRICCION------')
    elif i == 1:
        print('---FRICCION C1------')
    else:
        print('---FRICCION C2------')

    Velocida_champeon[i] = Vo_Xdeseada(tiempo, x, y, angulo, f, distancia_max, xf, err, 60, i)
```

```python

    print('La velocidad necesaria para alcanzar', xf, 'es:', Velocida_champeon[i])

    t_v[i] = NewtonR(tiempo, dt, y, err, Nmax, 0) # Tiempo de vuelo
    print('El tiempo de vuelo es:', t_v[i])

    x_max[i] = distancia_max(tiempo, x, y)
    print('La distancia maxima con esta velocidad es ', x_max[i])

    print('Tenemos un error relativo en la distancia de:', abs((xf - x_max[i]) / xf) * 100, '%')

    y_max[i] = Y_max(y)

    xxdatos[i] = [x[j, 0] for j in range(N)]
    vxdatos[i] = [x[j, 1] for j in range(N)]

    xydatos[i] = [y[j, 0] for j in range(N)]
    vydatos[i] = [y[j, 1] for j in range(N)]

y_Max = np.amax(y_max)

#---------------PARTE DE GRAFICACION ----------------------

# Configuracion de la grafica con mejor resolucion
plt.figure(figsize=(10, 5), dpi=150)  # Tamano de la figura y resolucion DPI

# Ajustes de fuente a Courier New
plt.rc('font', family='Courier New')

# Subgrafico 1
plt.subplot(1, 2, 1)  # Subgrafico de 1 fila y 2 columnas, primer subgrafico
plt.plot(tiempo, xydatos[0], '-r', label='sin friccion')
plt.plot(tiempo, xydatos[1], '-b', label='Fricion C1')
plt.plot(tiempo, xydatos[2], '-g', label='Fricion C2')
plt.xlabel('Tiempo (s)')  # Etiqueta del eje x
plt.ylabel('Altitud (m)')  # Etiqueta del eje y
plt.title('Tiempo de Vuelo')  # Titulo de la grafica
plt.legend(loc='best')  # Mostrar leyenda en la mejor ubicacion
plt.ylim(0, 5/4*y_Max)  # Limitar el eje Y para una mejor visualizacion
plt.xlim(0, np.max(t_v))  # Limitar el eje X al rango de 0 a t_v
plt.grid(True)  # Mostrar cuadricula
plt.tight_layout()  # Ajustar el diseno para que quepa todo

# Subgrafico 2
plt.subplot(1, 2, 2)  # Subgrafico de 1 fila y 2 columnas, segundo subgrafico
plt.plot(xxdatos[0], xydatos[0], '-r', label='sin friccion')
plt.plot(xxdatos[1], xydatos[1], '-b', label='Fricion C1')
plt.plot(xxdatos[2], xydatos[2], '-g', label='Fricion C2')
plt.xlabel('Posicion en X (m)')  # Etiqueta del eje x
plt.ylabel('Altitud (m)')  # Etiqueta del eje y
plt.title('Altitud en funcion de la Posicion en X')  # Titulo de la grafica
plt.legend(loc='best')  # Mostrar leyenda en la mejor ubicacion
plt.ylim(0, 5/4*np.amax(y_max))  # Limitar el eje Y para una mejor visualizacion
plt.xlim(0, np.max(x_max))  # Limitar el eje X al rango de 0 a x_max
plt.grid(True)  # Mostrar cuadricula
plt.tight_layout()  # Ajustar el diseno para que quepa todo

#plt.suptitle('Comparacion de Trayectorias')  # Titulo general

plt.show()
```

Código 1: Programa que a traves de metodos numericos , obtine la funciones solucion de ecuaciones diferenciales , sus graficas, asi como puntos especiales como distancias maximas de recorrido , asi como la velocidad para lograr algun valor de x especifico . NOMBRE : "Problema1 Rozamiento.py"

A.2. Búsqueda de frecuencias normales en un sistema de dos masas acopladas con resortes

```python
# -*- coding: utf-8 -*-
"""
Created on Thu May 16 11:49:50 2024
```

```python

@author: dangv
"""

import matplotlib.pyplot as plt
from pylab import *
import math

N = 100000   # Numero de pasos
x10 = 2        # Posicion inicial en masa 1 k lineal
v10 = 0    # Velocidad inicial en en masa 1 k lineal
x20 = 0        # Posicion inicial en masa 2 k lineal
v20 = 0   # Velocidad inicial en masa 2 k lineal
x30 = 2        # Posicion inicial en masa 1 k no lineal
v30 = 0   # Velocidad inicial masa 1 k no lineal
x40 = 0        # Posicion inicial masa 2 k no lineal
v40 = 0   # Velocidad inicial en masa 2 k no lineal
k1=0.3 #Constante resortes laterales
k=0.75 #Constante resorte intermedio
tau =40  # Tiempo en segundos de la simulacion
h = tau / float(N-1)   # Paso del tiempo
m = 1.2          # Masa de la particula


# Generamos un arreglo de Nx2 para almacenar posicion y velocidad
x1 = zeros([N,2])   #la primera columna sera la posicion y la segunda la velocidad en ese mismo punto
x2 = zeros([N,2])   #la primera columna sera la posicion y la segunda la velocidad en ese mismo punto
#notemos que hablamso de una matrix de Nx2
x3 = zeros([N,2])
x4 = zeros([N,2])

# tomamos los valores del estado inicial
x1[0,0] = x10
x1[0,1] = v10

x2[0,0] = x20
x2[0,1] = v20

x3[0,0] = x30
x3[0,1] = v30

x4[0,0] = x40
x4[0,1] = v40


#Tomamos el cambio de variable para encontrar soluciones armonicas simples.
xa=x1+x2   #Estado armonico 1 fase lineal
xb=x2-x1  #Estado armonico 2 no fase lineal

xc=x3+x4   #Estado armonico 1 fase nolineal
xd=x4-x3  #Estado armonico 2 no fase no lineal



# Generamos tiempos igualmente espaciados
tiempo = linspace(0, tau, N)   #el inicio siempre debe ser 0 por que es asi como inicializamos el programa

#------------Definicon de nuestras ecuaciones diferenciales con constante k lineal  ------------------------

def EDO1(estado1, tiempo):
    f0 = estado1[1]  #la primera derivada de la posicion no es mas que la misma velocidad
    f1 = -(k1/m)*estado1[0]
    # se sigue la ecuacion 1  , que iguala la derivada de la velocidad ( la aceleracion ), con la misma posicion .
    #se hace notar ademas que el sistema esta "amortiguado " al tener la presencia de la gravedad en este mismo .

    return array([f0, f1])

def EDO2(estado1, tiempo):
    f0 = estado1[1]  #la primera derivada de la posicion no es mas que la misma velocidad
    f1 = -((k1+2*k)/m)*estado1[0]
    # se sigue la ecuacion 1  , que iguala la derivada de la velocidad ( la aceleracion ), con la misma posicion .
    #se hace notar ademas que el sistema esta "amortiguado " al tener la presencia de la gravedad en este mismo .

    return array([f0, f1])

#------------Definicon de nuestras ecuaciones diferenciales con constante
```

```
     k  NO lineal ------------------------

def EDO3(estado1, tiempo):
    f0 = estado1[1]  #la primera derivada de la posicion no es mas que la misma velocidad
    f1 = -(k1/m)*(estado1[0]+0.1*estado1[0]**3)
    # se sigue la ecuacion 1  , que iguala la derivada de la velocidad ( la aceleracion ), con la misma posicion .
    #se hace notar ademas que el sistema esta "amortiguado " al tener la presencia de la gravedad en este mismo .

    return array([f0, f1])

def EDO4(estado1, tiempo):
    f0 = estado1[1]  #la primera derivada de la posicion no es mas que la misma velocidad
    f1 = -((k1+2*k)/m)*(estado1[0]+0.1*estado1[0]**3)
    # se sigue la ecuacion 1  , que iguala la derivada de la velocidad ( la aceleracion ), con la misma posicion .
    #se hace notar ademas que el sistema esta "amortiguado " al tener la presencia de la gravedad en este mismo .

    return array([f0, f1])


#---------------- Metodos para obtener las funciones de las ecuaciones diferenciales ---------

# Metodo de Euler para  resolver numericamente la EDO
def Euler1(y, t, h, f):
    y_s = y + h * f(y, t)  # Calculamos el valor siguiente de y
    return y_s

def Euler2(y1,t,h,f):
    y_m = y1+h*((f(y1,t)+f(Euler1(y1,t,h,f),t))/2)  #Calulamos un valor mas puro siguiente de y
    return y_m

def RungeKutta(y, t, h, f):
    k1 = h * f(y, t)
    k2 = h * f(Euler2(y,t,h/2,f), t+h/2 )
    k3= h*f(y+k2/2,t+h/2)
    k4=h*f(y+k3,t+h)
    y_p = y + 1/6*(k1+2*k2+2*k3+k4)
    return y_p

#----------------------- Algoritmos Utiles --------------------------------

t0 = tau / 2
dt = 3.e-3
err = 0.001
Nmax = 100 # Parametros
Err0=0.0015
Err01=0.015

def Mapeo(t0, t, x, s, err):
    for j in range(N-1):
        #print('Iteracion',j,'diferencia:',abs(t[j] - t0))
        if abs(t[j] - t0) <= err:
            F = x[j, s]
            return F
    print('\n No se encontro valor de la funcion en t=', t0)
    return 0

#Usaremos el metodo de la biseccion para tener una aproximacion a la raiz y ya despues purificamos con newton rapson
def bisection( a, b, err, max_iter,t,x,s):
    """
    Implementacion del metodo de biseccion para encontrar una raiz de una funcion.

    Args:
        f: Funcion cuya raiz se busca.
        a: Extremo izquierdo del intervalo inicial.
        b: Extremo derecho del intervalo inicial.
        tol: Tolerancia para el error absoluto entre iteraciones consecutivas.
        max_iter: Numero maximo de iteraciones permitidas.

    Returns:
        float: Aproximacion de la raiz encontrada.
        int: Numero de iteraciones realizadas.

    # Verificar si la raiz esta dentro del intervalo [a, b]
    #como esta funcion oscila no tiene sentido agregar esto
    if Mapeo(a, t, x, s, err) * Mapeo(b, t
```

```
, x, s, err)  >= 0:
        print(Mapeo(a, t, x, s, err) * Mapeo(b, t, x, s, err))
        raise ValueError("La funcion no cambia de signo en el intervalo dado.")
"""
    # Inicializar variables
    iter_count = 0
    while (b - a) / 2 > err and iter_count < max_iter:
        c = (a + b) / 2  # Punto medio del intervalo
        if Mapeo(c, t, x, s, err) == 0:
            return c, iter_count
        elif Mapeo(c, t, x, s, err)* Mapeo(a, t, x, s, err)< 0:
            b = c  # La raiz esta en el subintervalo [a, c]
        else:
            a = c  # La raiz esta en el subintervalo [c, b]
        iter_count += 1

    return (a + b) / 2

def NewtonR(t, dt, x, Err0, Nmax,s,a,b):
    t0=bisection( a, b, Err01,100,t,x,s)
    for it in range(0, Nmax + 1):
        F = Mapeo(t0, t, x, 0 ,Err0)
        if F is not None and abs(F) <= err:
            break
        elif F is None:
            print('\n Valor de funcion es None en x=', t0)
            break
       # print('Iteracion=', it, 'x=', t0, 'f(x)=', F)
        df = (Mapeo(t0 + dt / 2, t, x, 0, Err0) - Mapeo(t0 - dt / 2, t, x, 0, Err0)) / dt  # Central difference
        if df == 0:
            # Division por cero en df. No se puede continuar.
            break
        dt = -F / df
        t0 += dt  # Nueva propuesta
    if it == Nmax + 1:
        #print('\n Newton no encontro raiz para Nmax=', Nmax)
        return None
    return t0
#El algoritmo de newton es preciso , pero nesesita  un valor inicial cercano a la raiz para ser realmente efectivo , por eso usaremos en conjunto biseccion y newton rapson
#----------------------Obtencion de funciones ----------------------

def f(t, xa, xb):
    for j in range(N-1):
        xa[j+1] = RungeKutta(xa[j], t[j], h, EDO1)
        xb[j+1] = RungeKutta(xb[j], t[j], h, EDO2)

def n(t, xa,xb):
    for j in range(N-1):
        xa[j+1] = RungeKutta(xa[j], t[j], h, EDO3)
        xb[j+1] = RungeKutta(xb[j], t[j], h, EDO4)

f(tiempo,xa,xb)
n(tiempo,xc,xd)


#Utilizaremos las funciones de biseccion y Newton Rapson  para encontrar
#donde la distancia entre raices, "Un medio periodo "
#Obtenemos las posicones originales en funcion de las del cambio de variable

def FrecAng(t, dt, x, err,err1, Nmax,s):
    r0=0
    r1=NewtonR(t, dt, x, err, Nmax,s,r0,tau)
    iter_count=0
    while abs(r1-r0)>err:
       iter_count=+1
       r0=r1
       r1=NewtonR(t, dt, x, err, Nmax,s,0,r0)
       #print('ro',iter_count,'=',r0)
#entonces r0 la raiz mas cercana al cero
    r1=0
    r2=NewtonR(t, dt, x, err, Nmax,s,r0+err1,tau)
    iter_count=0
    while abs(r2-r1)>err:
       iter_count=+1
       r1=r2
       r2=NewtonR(t, dt, x, err, Nmax,s,r0+err1,r1) #el anadir el error en la cota inferior de la busqueda de la raiz evita que vuelva a determinarse en la  raiz
       #print('r1',iter_count,'=',r1)

    SP=r1-r0 #Semiperiodo
    FA=np.pi/SP
```

```python

    return FA

#Los convertimos a datos las variables -------------------------------
x1=(xa+xb)/2
x2=(xa-xb)/2


xadatos = [xa[j, 0] for j in range(N)]
vadatos = [xa[j, 1] for j in range(N)]

xbdatos = [xb[j, 0] for j in range(N)]
vbdatos = [xb[j, 1] for j in range(N)]

x1datos = [x1[j, 0] for j in range(N)]
v1datos = [x1[j, 1] for j in range(N)]

x2datos = [x2[j, 0] for j in range(N)]
v2datos = [x2[j, 1] for j in range(N)]

x3datos = [x3[j, 0] for j in range(N)]
v3datos = [x3[j, 1] for j in range(N)]

x4datos = [x4[j, 0] for j in range(N)]
v4datos = [x4[j, 1] for j in range(N)]


#GRAFICACION ---------------------------------------------------------------

# Definimos el tamano de la figura
plt.figure(figsize=(12, 6))

# Ajustes de fuente a Courier New
plt.rc('font', family='Courier New')

# Subgrafico 1
plt.subplot(1, 2, 1)  # Subgrafico de 1 fila y 2 columnas, primer subgrafico
plt.plot(tiempo, x1datos, '-r', label='Masa 1')
plt.xlabel('Tiempo (s)', fontsize=12)  # Etiqueta del eje x
plt.ylabel('Amplitud (m)', fontsize=12)  # Etiqueta del eje y
plt.title('Amplitud masa 1 respecto a t', fontsize=14)  # Titulo de la grafica
plt.legend(loc='best', fontsize=10)  # Mostrar leyenda en la mejor ubicacion
plt.grid(True)  # Mostrar cuadricula

# Subgrafico 2
plt.subplot(1, 2, 2)  # Subgrafico de 1 fila y 2 columnas, segundo subgrafico
plt.plot(tiempo, x2datos, '-b', label='Masa 2')
plt.xlabel('Tiempo (s)', fontsize=12)  # Etiqueta del eje x
plt.ylabel('Amplitud (m)', fontsize=12)  # Etiqueta del eje y
plt.title('Amplitud masa 2 respecto a t', fontsize=14)  # Titulo de la grafica
plt.legend(loc='best', fontsize=10)  # Mostrar leyenda en la mejor ubicacion
plt.grid(True)  # Mostrar cuadricula

# Ajustar el diseno para que quepa todo
plt.tight_layout(rect=[0, 0, 1, 0.96])

# Titulo general
plt.suptitle('Comparacion de movimiento de coiladores acoplados : x1o!=x2o=0', fontsize=16, fontfamily='Courier New')

# Mostrar la grafica
plt.show()



#Comparacion Valores Numericos con teoricos -------------


wat=math.sqrt(k1/m)
wbt=math.sqrt((k1+2*k)/m)

#Notamos que en los cambios de variable , para cuando X1=X2 o X1=-X2, entonces los cambios de
#variable , una de las dos se hacen cero , entonces las frecuencias angulares de alguna de ellas se hace cero
#Para evitar conflictos y modificar la funcion mapeo , entonces aplicamos unas funciones if
#SIN EMBARGO, lo siguiente no tiene tento sentido fue los casos especiales, ya que
#en los casos especiales, las ocndiciones iniciales harian que se haga cero alguna
#alguno de los cambios de variable

if x10==0 & x20==0 :
    wan=0
    wbn=0
elif x10==-x20:
    wan=0
    wbn=FrecAng(tiempo,dt,xb,Err0,Err01,Nmax,0)
elif x10==x20:
    wan=FrecAng(tiempo,dt,xa,Err0,Err01,
```

```
    Nmax,0)
    wbn=0
else :
    wbn=FrecAng(tiempo,dt,xb,Err0,Err01,Nmax,0)
    wan=FrecAng(tiempo,dt,xa,Err0,Err01,Nmax,0)

if x30==0 & x40==0 :
    wcn=0
    wdn=0
elif x30==-x40:
    wcn=0
    wdn=FrecAng(tiempo,dt,xd,Err0,Err01,Nmax,0)
elif x30==x40:
    wcn=FrecAng(tiempo,dt,xc,Err0,Err01,Nmax,0)
    wdn=0
else :
    wdn=FrecAng(tiempo,dt,xd,Err0,Err01,Nmax,0)
    wcn=FrecAng(tiempo,dt,xc,Err0,Err01,Nmax,0)




print('\n Frecuencias Normales con K lineal :\n')

print('fase: Teorica/Numerica',wat,'/',wan)
print('contrafase: Teorica/Numerica',wbt,'/',wbn)


print('\n Frecuencias Normales con K no lineal :\n')

print('fase:Numerica',wcn)
print('contrafase : Numerica',wdn)

print('\n Veamos entonces las diferencias entre las frecuencias de los modos normales: \n ')

if wan==0 or wdn==0:
    print('fase:Numerica  Error absoluto ',abs(wan-wcn))
    print('Contrafase :Numerica  Error absoluto ',abs(wbn-wdn))
else:
    print('fase:Numerica  Error absoluto ',abs(wan-wcn),'Error relativo :',abs((wan-wcn)/wan),'\n')
    print('Contrafase :Numerica  Error absoluto ',abs(wbn-wdn),'Error relativo :',abs((wbn-wdn)/wdn))
```

Código 2: Programa que a traves de metodos numericos , obtine la funciones solucion de ecuaciones diferenciales , sus graficas, asi como las frecuencias angulares ω para algun valor de t especifico . NOMBRE : "Masa y Resorte.py"

A.3. Animación de cuerda creada aparir de la ecuación diferencial de onda

```
# -*- coding: utf-8 -*-
"""
Created on Fri May  3 15:45:37 2024

@author: dangv
"""
#Biblioteca para poder manejar archivos (Se usa en la parte de crear carpetas para ingresar imagenes de generacion de la animacion)
import os

#Biblioteca de matematicas, vectores y graficos

import numpy as np
import matplotlib.pyplot as plt
import math

#Biblioteca para animaciones
from matplotlib import animation as anim
from PIL import Image
from glob import glob

#Variables generales

Lx = 2  # Maximo de distancia espacial
Lt = 20  # Maximo de tiempo
Nx = 1000  # Numero de particiones del eje x
Nt = 8000  # Numero de particiones del eje t
dx = Lx / Nx
dt = Lt / Nt
F = 5  # Fuerza de tension de la cuerda en x
M = 3  # Densidad de masa lineal
cf = math.sqrt(F / M)  # Velocidad de fase
cm = dx / dt
CNT = cf / cm

print('Velocidad de Fase:', cf, '\nVelocidad de Malla:', cm, '\
```

```python
nCondicion de Courant:', CNT)

Z = 60  # Iteraciones de Gauss-Seidel
x = np.linspace(0, Lx, Nx)
t = np.linspace(0, Lt, Nt)

y = np.ones((Nx, Nt))  # Inicializa la matriz y con valores "basura"

# Definimos las ecuaciones de las condiciones de frontera -----


def Condicion_Inicial(x):
    return math.sin((3 * math.pi) * x / Lx)  #Funion de condicion inicial IMPORTANTE que la condicion  inicial y las condiciones de frontera concuerden

def Condicion_Frontera_L0(t):
    return 0

def Condicion_Frontera_Lx(t):
    return 0

# Inicializacion de las condiciones de frontera e iniciales  ---------


for j in range(Nt):
    y[0, j] = Condicion_Frontera_L0(t[j])
    y[Nx-1, j] = Condicion_Frontera_Lx(t[j])

for i in range(Nx):
    y[i, 0] = Condicion_Inicial(x[i])

print('Generando funcion...')

# Iteracion de la EDP
for u in range(Z):
    for j in range(Nt - 1):
        for i in range(1, Nx - 1):  # Corrige los limites para no sobrescribir las fronteras
            if j == 0:
                y[i, 1] = y[i, 0] +  0.5*(CNT**2) * (y[i + 1, 0] + y[i - 1, 0] - 2 * y[i, 0]) #Pequeno paso  que es util matematicamente  por la falta de informacion en y[i,j-1], cuando j==0
            else:
                y[i, j + 1] = 2 * y[i, j] - y[i, j - 1] + (CNT**2) * (y[i + 1, j] + y[i - 1, j] - 2 * y[i, j])
    print((u + 1) / Z * 100, '%')

print('Graficando...')

# Crear la carpeta "imagenes_grafica" si no existe
output_dir = "imagenes_grafica"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Crear las figuras para cada instante de tiempo
n = 100  # Numero de divisiones
s = Nt // n
for i in range(n):
    d = s * i
    y1 = y[:, d]  # Obtener la columna correspondiente del tiempo t = d * dt

    plt.figure(figsize=(12, 6), dpi=300)  # dpi=300 aumenta la calidad de la grafica
    plt.plot(x, y1, color='b', label=f'Tiempo t={t[d]:.2f}s')

    # Anadir etiquetas y titulo
    plt.xlabel('Distancia X (m)', fontsize=12)
    plt.ylabel('Amplitud', fontsize=12)
    plt.title('Funcion de Onda', fontsize=20)
    plt.legend()

    # Ajustes para mejorar la visualizacion
    plt.ylim(-1.1, 1.1)  # Limitar el eje Y para una mejor visualizacion de la funcion seno
    plt.xlim(0, Lx)  # Limitar el eje X al rango de 0 a Lx

    # Guardar la figura
    plt.savefig(f'{output_dir}/figura_{i:03d}.png')  # Guardar con un nombre unico para cada frame

    # Cerrar la figura para evitar sobrecarga de memoria
    plt.close()

print('Imagenes generadas exitosamente\nGenerando GIF...')

# Animacion ------------------

# Obtener la lista de archivos que coinciden con el patron
files = sorted(glob(f'{output_dir}/figura_
```

```
    *.png'))

# Verificar si se encontraron archivos
if not files:
    print("No se encontraron archivos de imagen para animar.")
else:
    # Crear la figura y los ejes
    fig, ax = plt.subplots(figsize=(12, 6))
    ax.axis('off')  # Ocultar los ejes

    # Lista para almacenar los objetos de imagen
    ims = []

    # Cargar cada imagen y anadirla a la lista de frames
    for fname in files:
        im = ax.imshow(Image.open(fname), animated=True)
        ims.append([im])

    # Crear la animacion
    ani = anim.ArtistAnimation(fig, ims, interval=100, repeat_delay=1000, blit=True)

    # Guardar la animacion como un archivo GIF
    ani.save('animacion.gif', writer='pillow')

    plt.close()  # Cerrar la figura despues de guardar

print('GIF generado exitosamente')
```

Código 3: Programa que genera valores de funcion de onda apartir de condiciones de frontera e iniciales.NOMBRE : ”GaussSeidelEcONDA2.py”