FÍSICA COMPUTACIONAL

RUIZ MUÑOZ, JUAN MANUEL

30 noviembre 2023

1 PRÁCTICA 1

Para el primer ejercicio a resolver utilizaré las librerías y paquetes siguientes:

```
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy import linalg, sparse
import time
```

Se define una función como la que nos dice el enunciado:

```
def TridiagonalSolver(d, o, u, r):

    inicial = time.time()

    h_0 = o[0]/d[0] #partiendo de que el primer elemento de la
                                        diagonal principal es no nulo
    p_0 = r[0]/d[0]
    H = [h_0]
    P = [p_0]

    n = len(d) #dimensi n diagonal
    sol = np.zeros(n)

    for i in range(1, n-1):  #bucle para obtener h_i

        H.append(o[i]/(d[i] - u[i-1]*H[i-1]))

    for i in range(1, n): #bucle para obtener p_i
        P.append((r[i] - u[i-1]*P[i-1])/(d[i] - u[i-1]*H[i-1]))
    sol[-1] = P[-1]
    #Al realizar este  ltimo  bucle con append da problemas, decido
                                       cambiar el formato para este
                                        ltimo
    #y a adir las soluciones a un vector de ceros predefinido.
    for i in range(n-1, 0, -1): #bucle para el vector soluci n

        sol[i-1] = P[i-1] - H[i-1]*sol[i]

    final = time.time()

    total = final - inicial
```

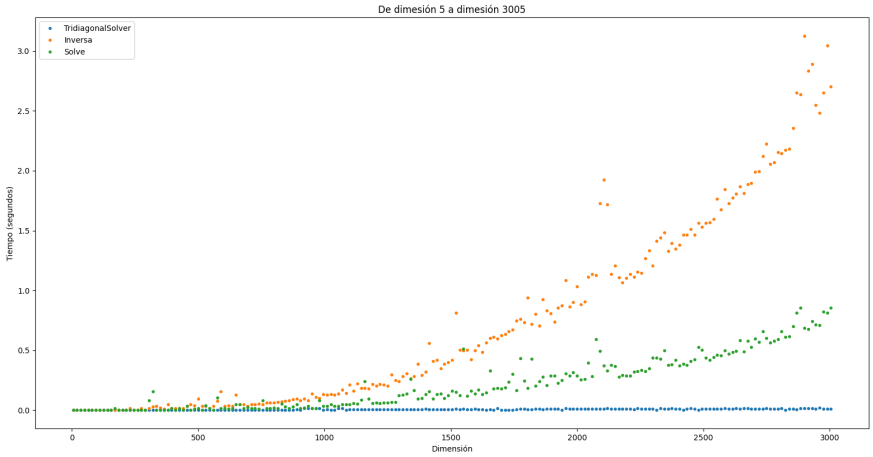
```
    return sol, total
```

Dadas las cuatro entradas d, o, u y r la función nos devuelve la solución del sistema tridiagonal que define matricialmente.
Trabajo vectorialmente puesto que para mí me resulta más sencillo, es decir, las entradas de la función $TridiagonalSolver$ son vectores.
Notar, que la primera salida de dicha función nos devuelve la solucón del sistema tridiagonal a resolver, la salida **sol**. En la segunda, **total**, nos devuelve el tiempo de ejecución, que hemos implementado en la misma para los siguientes apartados.
En cuanto a el resto de apartados, tratamos todos de una vez. Dentro de un mismo bucle, donde iremos resolviendo para distintas dimensiones, colocamos tres formas diferentes de resolver el mismo sistema.

```
for dim in range(dim_min, dim_max + paso, paso):

    #A lo largo de este bucle no se hace uso, como se puede
                                    observar, de las soluciones
                                    calculadas
    #por los diferentes m todos, ya que lo  nico  que nos interesa
                                     en este caso es comparar
                                    tiempos de ejecuci n.
    #Si queremos obtener la soluci n de alg n sistema tridiagonal
                                     solo tenemos que llamar a la
                                     funci n TridiagonalSolver y
                                     definir las
    # 4 entradas en forma vectorial. O bien, hacer uso del m todo
                                    de la inversa o del paquete
                                    solve.
    print(dim)

    A_d = 7*np.ones(dim)
    A_s = np.ones(dim-1)
    A_i = np.ones(dim-1)
    b = 3*np.ones(dim)
    #se obtienen los tiempos resolviendo con TridiagonalSolver.
    solucion, tiempo = TridiagonalSolver(A_d, A_s, A_i, b)
    tiempos.append(tiempo)
    #generamos matriz tridiagonal de dimensi n dim
    B_d = np.diag(A_d, k = 0)
    B_s = np.diag(A_s, k = 1)
    B_i = np.diag(A_i, k = -1)
    B = B_d + B_s + B_i # matriz tridiagonal
    #Se obtienen los tiempos usando la inversa.
    t_i = time.time()
    B_inv = np.linalg.inv(B)
    resolucion = np.dot(B_inv, b)
    t_f = time.time()
    t_t = t_f - t_i

    tiempos_2.append(t_t)
    #Se obtienen los tiempos resolviendo con np.solve()
    t_1s = time.time()
    Solucion_Solver = np.linalg.solve(B, b)
```

```
    t_2s = time.time()
    t_ts = t_2s - t_1s

    tiempos_3.append(t_ts)
    a = dim
```

Definimos una dimensión mínima y otra máxima, que serán los limites inferiores y superiores de nuestro bucle. Además, para dar una mayor rápidez, introducimos el parámetro **paso**.
En este bucle, resolvemos el sistema con los tres métodos (TridiagonalSolver, Solve y inversa) para diferentes dimensiones entre una dimensión mínima. $\mathbf{dim_{min}}$, y una máxima, $\mathbf{dim_{max}}$, que definiremos en el script. Nuestro objetivo principal con esto es observar y comparar los tiempos de ejecución de los tres métodos a medida que la dimensión de los sistemas van aumentando. Como trabajaremos con dimensiones elevadas, del orden de 3000, la densidad de puntos nos es un poco igual, mientras esta nos sea suficiente como para observar lo que queremos. Es por esto último por lo que, para acelerar el proceso computacional introducimos un parámetro de paso con el cuál resolveremos un menor número de veces los sistemas, con los tres métodos.



Gráfica 1: Tiempos de ejecución respecto a la dimensión para los tres métodos.

De la gráfica 1, podemos observar que le método más óptimo, para resolver sistemas tridiagonales de dimensiones elevadas, es la función que hemos creado **TridiagonalSolver**, siendo el menos óptimo el método de la inversa.
Podemos observar también, que para dimensiones pequeñas, es igual el método utilizado pues más o menos los tres tienen tiempos de ejecución similares.

1.1 CÓDIGO ENTERO PRÁCTICA 1

```
# -*- coding: utf-8 -*-
"""
```

```
Created on Thu Sep 21 16:39:10 2023

@author: juanm
"""
#PRE MBULO#

import numpy as np
import matplotlib.pyplot as plt
import math
from scipy import linalg, sparse
import time



#Observando la forma de la relaci n entre los coeficientes del
                                    sistema
#podemos notar que podemos resolver el sistema de forma vectorial
                                    en lugar
#de hacer uso de matrices.
#Por ello, en la funci n TridiagonalSolver le introduciremos como
                                    VECTORES
#cada una de las diagonales de la matriz:
                # d: 'Diagonal principal del sistema (lineal) en
                                                    forma matricial
                                                    a resolver'
                # o: 'diagonal superior del sistema (lineal) en
                                                    forma matricial
                                                    a resolver'
                # u: 'diagonal inferior del sistema (lineal) en
                                                    forma matricial
                                                    a resolver'
                # r: 'vector de t rminos independientes del
                                                    sistema (lineal)
                                                     en forma
                                                    matricial a
                                                    resolver'



def TridiagonalSolver(d, o, u, r):

    inicial = time.time()

    h_0 = o[0]/d[0] #partiendo de que el primer elemento de la
                                        diagonal principal es no nulo
    p_0 = r[0]/d[0]
    H = [h_0]
    P = [p_0]

    n = len(d) #dimensi n diagonal
    sol = np.zeros(n)

    for i in range(1, n-1):  #bucle para obtener h_i

        H.append(o[i]/(d[i] - u[i-1]*H[i-1]))
```

```
    for i in range(1, n): #bucle para obtener p_i
        P.append((r[i] - u[i-1]*P[i-1])/(d[i] - u[i-1]*H[i-1]))
    sol[-1] = P[-1]
    #Al realizar este  ltimo  bucle con append da problemas, decido
                                      cambiar el formato para este
                                       ltimo
    #y a adir las soluciones a un vector de ceros predefinido.
    for i in range(n-1, 0, -1): #bucle para el vector soluci n

        sol[i-1] = P[i-1] - H[i-1]*sol[i]

    final = time.time()

    total = final - inicial

    return sol, total  #la funci n nos devuelve la soluci n, en
                                     su primera salida, y el
                                     tiempo de ejecuci n, en la
                                     segunda.

###Definimos cada una de las matrices

d = np.array([1,2,3,4,5])

o = np.array([1,1,1,1])

u = np.array([1,1,1,1])

r = np.array([4,4,4,4,4])

#si quieres ver c mo funciona la funci n TridiagonalSolver
                                   descomenta el print siguiente:

#print(TridiagonalSolver(d, o, u, r))

###

#Ahora vamos a definir cuatro vectores (A_d, A_s, A_i, b) donde:

                                  # A_d: 'Diagonal principal'
                                  # A_s: 'Diagonal superior'
                                  # A_i: 'Diagonal inferior'
                                  # b:   'T rminos independientes'
                                  # A = A_d + A_s + A_i donde A\vec(x
                                                                    )

                                                                    =

                                                                    \
                                                                    vec
                                                                    (
                                                                    b
                                                                    )


 #El objetivo para esta segunda parte es representar los tiempos de
                                      resoluci n para distintas
```

```
                                    dimensiones.

#Para ello, definimos dichas matrices como unos, principalmente por
                                     comodidad. Iremos variando la
                                    longitud de los mismos.

tiempos = [] #resolviendo con la funci n TridiagonalSolver
tiempos_2 = [] #tiempos resolviendo con la inversa
tiempos_3 = [] #tiempos para solve


#A continuaci n, le decimos al programa la dimensi n m nima y la
                                     m xima para la cu l queremos
                                    resolver.
dim_min = 5
dim_max = 3000
paso = 15 #la variable paso se define con la finalidad de poder
                                    dotar al proceso de una mayor
                                    velocidad, reduciendo el n mero
# de veces que resolvemos, en el bucle, el sistema para diferentes
                                    dimensiones.
#Al final de cuentas no necesitamos una gran densidad de puntos
                                    para observar lo que queremos.
#A mayor paso tendremos un menor tiempo de espera para obtener la
                                     gr fica de tiempos, eso s  ,
                                    tendremos menor densidad de
                                    puntos.

for dim in range(dim_min, dim_max + paso, paso):

    #A lo largo de este bucle no se hace uso, como se puede
                                        observar, de las soluciones
                                        calculadas
    #por los diferentes m todos, ya que lo  nico  que nos interesa
                                         en este caso es comparar
                                        tiempos de ejecuci n.
    #Si queremos obtener la soluci n de alg n sistema tridiagonal
                                         solo tenemos que llamar a la
                                         funci n TridiagonalSolver y
                                         definir las
    # 4 entradas en forma vectorial. O bien, hacer uso del m todo
                                        de la inversa o del paquete
                                        solve.
    print(dim)

    A_d = 7*np.ones(dim)
    A_s = np.ones(dim-1)
    A_i = np.ones(dim-1)
    b = 3*np.ones(dim)
    #se obtienen los tiempos resolviendo con TridiagonalSolver.
    solucion, tiempo = TridiagonalSolver(A_d, A_s, A_i, b)
    tiempos.append(tiempo)
    #generamos matriz tridiagonal de dimensi n dim
    B_d = np.diag(A_d, k = 0)
    B_s = np.diag(A_s, k = 1)
    B_i = np.diag(A_i, k = -1)
    B = B_d + B_s + B_i # matriz tridiagonal
```

```
    #Se obtienen los tiempos usando la inversa.
    t_i = time.time()
    B_inv = np.linalg.inv(B)
    resolucion = np.dot(B_inv, b)
    t_f = time.time()
    t_t = t_f - t_i

    tiempos_2.append(t_t)
    #Se obtienen los tiempos resolviendo con np.solve()
    t_1s = time.time()
    Solucion_Solver = np.linalg.solve(B, b)
    t_2s = time.time()
    t_ts = t_2s - t_1s

    tiempos_3.append(t_ts)
    a = dim
#fig, ax= plt.subplots()

plt.figure('Tiempos')
plt.title(f'De dimesi n {dim_min} a dimesi n {a}')
#ax.scatter(np.linspace(dim_min, dim_max, dim_max - dim_min),
                                    tiempos)
#ax2.scatter(np.linspace(dim_min, dim_max, dim_max - dim_min),
                                    tiempos_2)
plt.plot(np.arange(dim_min, dim_max + paso, paso), tiempos, '.',
                                    label = 'TridiagonalSolver')
plt.plot(np.arange(dim_min, dim_max + paso, paso), tiempos_2, '.',
                                    label = 'Inversa')
plt.plot(np.arange(dim_min, dim_max + paso, paso), tiempos_3, '.',
                                    label = 'Solve')
plt.xlabel('Dimensi n')
plt.ylabel('Tiempo (segundos)')
plt.legend()


plt.show()
```

2 PRÁCTICA 2

2.1 Resolvemos ecuación Laplace

Presentamos el código desarrollado para resolver el problema propuesto en el primer apartado del ejercicio 2. Para ello generamos una matriz de dimensiones $dim^2 \times dim^2$ ya que tendremos dim^2 incógnitas, dim^2 filas y dim^2 columnas, conforme he planteado el problema, que es esencialmente, resolver un sistema de ecuaciones lineales, discretizando el espacio por diferencias finitas.

```
    espacio = np.zeros([dim**2, dim**2])
```

Este espacio inicialmente es todo nulo, ceros. Iremos modificando los valores de cada punto a medida que vayamos resolviendo el problema e imponiendo las condiciones de contorno.

Conforme planteamos el sistema a resolver, donde cada una de las ecuaciones corresponderá a un punto en nuestro espacio de potencial, V_{ij}, las ecuaciones que tendremos que tener en cuenta para imponerles las condiciones de contorno serán las $dim^2 - dim$ primeras, empezando por arriba, aunque la primera de todas, la asignaremos directamente por comodidad a la hora de trabajar con nuestro código.
Lo mismo ocurrirá para las $dim^2 - dim$ empezando por abajo y la última de todas, la asignaremos nosotros, de nuevo, manualmente.

```
    espacio[0,0] = -4
espacio[0,1] = 1             #condici n de contorno para el punto
                                   V_00, introduzco a mano.
espacio[0,dim-1] = 1

espacio[dim**2 - 1, dim**2 - 1] = -4
espacio[dim**2 - 1, dim**2 - 2] = 1           #condici n de
                                   contorno para el punto V_dim**
                                   2dim**2.
espacio[dim**2 - 1, dim**2 - dim] = 1
```

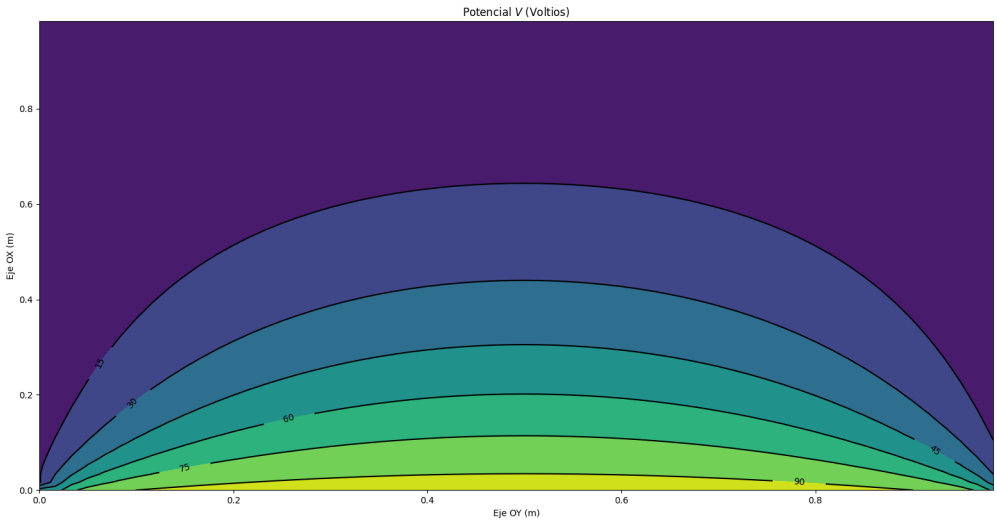
Código 1: Definimos la primera y última fila directamente, a mano.
El resto de condiciones y valores en la matriz a resolver son introducidas en los siguientes bucles:

```
for i in range(1, dim**2 - dim):

    filas_sup = np.zeros(dim**2)
    if i < dim:
        filas_sup[i] = -4
        filas_sup[i+1] = 1   #condiciones de contorno para los
                                   puntos del potencial,
                                   superiores, del mallado.

        filas_sup[i-1] = 1
        filas_sup[i + dim] = 1
        espacio[i,:] = filas_sup
    else:
        filas_sup[i] = -4
        filas_sup[i+1] = 1
        filas_sup[i-1] = 1              #ecuaciones para los puntos
                                      alejados del contorno.
                                      No presentan problemas.

        filas_sup[i + dim] = 1
        filas_sup[i - dim] = 1
        espacio[i,:] = filas_sup


for i in range(dim**2 - dim, dim**2 - 1):

    filas_inf = np.zeros(dim**2)
    filas_inf[i] = -4
    filas_inf[i+1] = 1   #condiciones de contorno para los puntos
                                   del potencial, inferiores,
                                   del mallado.
    filas_inf[i-1] = 1
    filas_inf[i - dim] = 1
    espacio[i,:] = filas_inf
```

Por último, realizamos algunas correcciones debidas a la forma en la que hemos construido la matriz:

```
for i in range(1, dim**2 - 1):
    espacio[i,0] = 0              #condici n de contorno para los
                                        puntos  en contacto con los
                                        margenes izquierdo y derecho.
    espacio[i,dim**2 - 1] = 0     #estos son los puntos de la forma
                                        V_i0 y Vidim**2-1.


for i in range(1, dim):               #corregimos la matriz ya que si
                                     nos damos cuenta en la forma en
                                   la que completo
    espacio[i*dim, i*dim - 1] = 0    #la matriz 'espacio' hemos
                                        introducido al inicio de cada
                                         bloque un 1 extra
#al lado de cada -4. Este debe ser un 0 y eso es lo que corregimos
                                   en este   ltimo   bucle
```

Donde el primer bucle coloca las condiciones de contorno de los margenes izquierdo y derecho, es decir, que el potencial sea nulo en todos estos puntos. El segundo bucle es para correcciones en nuestra matriz a resolver, ya que añadimos un 1 en algunos puntos donde corresponde un 0.



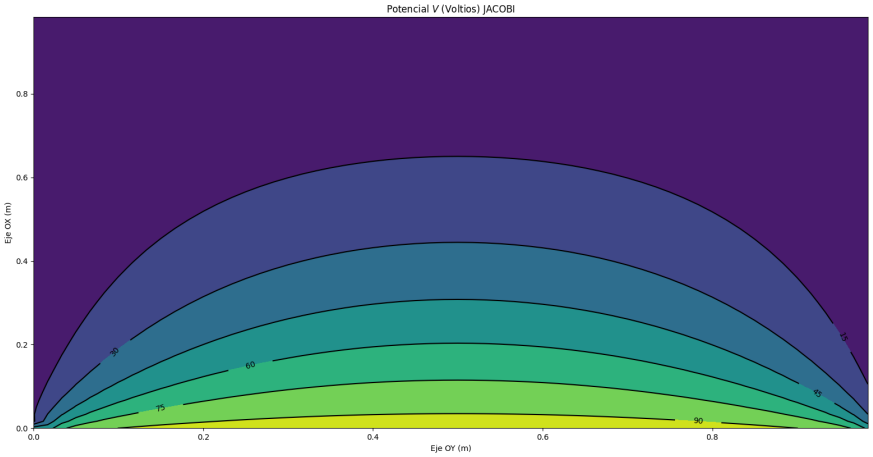
Gráfica 2.1: Representación de potencial(resuelto con solve), $V(x, y)$. Con colores más azulados se representan los valores más bajos mientras que para los más amarillos los más altos.

2.2 Resolviendo con Jacobi

Ahora resolvemos el mismo problema pero con el método de Jacobi. Para ello definimos la siguiente función:

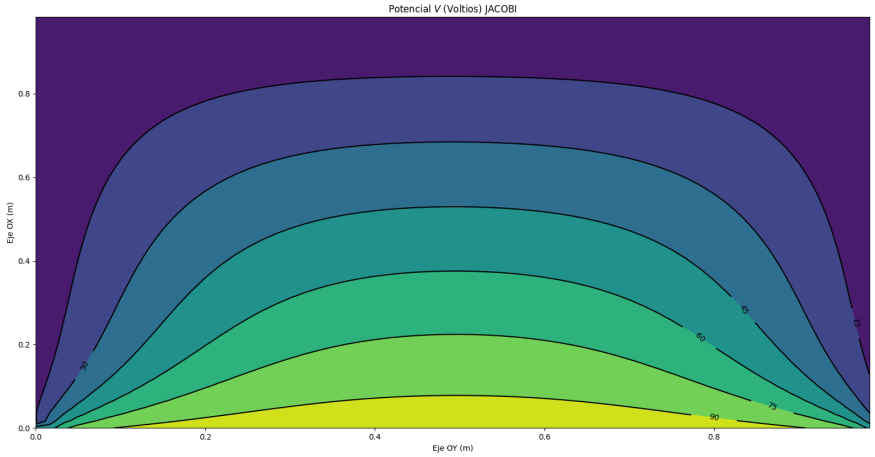
```
def jacobi(A, x_0, dim, b):

    v = -4*np.ones(dim**2)
    D = np.diag(v, k = 0)
    D_inv = np.linalg.inv(D)
    matriz = A - D

    vector_sol =[x_0]
    sol_futuro = np.dot(D_inv,(b - np.dot(matriz,vector_sol[-1])))
    vector_sol.append(sol_futuro)

    while abs(np.linalg.norm(vector_sol[-1]) - np.linalg.norm(
                                        vector_sol[-2])) > 0.01:

        sol_futuro = np.dot(D_inv,(b - np.dot(matriz,vector_sol[-1]
                                            )))
        vector_sol.append(sol_futuro)

    else:

        return vector_sol[-1]
```

Obtenemos, dando una precisión adecuada, una solución muy parecida a la anterior:



Gráfica 2.2: Potencial $V(x, y)$ resuelto por el método de Jacobi.

Ahora bien, si disminuimos la precisión en el método de Jacobi ya notamos la diferencia puesto que el método convergerá antes y la solución no será tan próxima a la que buscamos.



Gráfica 2.2: Potencial $V(x, y)$ resuelto por el método de Jacobi para una precisión menor.

2.3 CÓDIGO ENTERO PRÁCTICA 2

```
# -*- coding: utf-8 -*-
"""
Created on Tue Sep 26 18:06:41 2023

@author: juanm
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
from copy import copy

#dim = 70  #n mero de puntos por eje, en total tendremos un
                                    mallado de 100 puntos en OX y 100
                                     en OY.
#if dim > 70:
 #   print('Dimensi n alta, tiempo de espera, al menos, 1/2 minuto
                                    ')
dim = 60
cond_sup = -100
cond_inf = 0            #condiciones contorno, t rmino
                                    independiente \vec{b}

#generamos una matriz de dimensiones dim**2xdim**2 ya que tendremos
                                     dim**2 incognitas.
#dim**2 filas y dim**2 columnas, conforme he planteado el problema,
                                     que es esencialmente,
#resolver un sistema de ecuaciones lineales, discretizando el
                                    espacio por diferencias finitas.
espacio = np.zeros([dim**2, dim**2])

#para puntos fuera del contorno
```

```
for i in range(1, dim**2 - dim):

    filas_sup = np.zeros(dim**2)
    if i < dim:
        filas_sup[i] = -4
        filas_sup[i+1] = 1   #condiciones de contorno para los
                                    puntos del potencial,
                                    superiores, del mallado.
        filas_sup[i-1] = 1
        filas_sup[i + dim] = 1
        espacio[i,:] = filas_sup
    else:
        filas_sup[i] = -4
        filas_sup[i+1] = 1
        filas_sup[i-1] = 1              #ecuaciones para los puntos
                                          alejados del contorno.
                                          No presentan problemas.
        filas_sup[i + dim] = 1
        filas_sup[i - dim] = 1
        espacio[i,:] = filas_sup


for i in range(dim**2 - dim, dim**2 - 1):

    filas_inf = np.zeros(dim**2)
    filas_inf[i] = -4
    filas_inf[i+1] = 1   #condiciones de contorno para los puntos
                                del potencial, inferiores,
                                del mallado.
    filas_inf[i-1] = 1
    filas_inf[i - dim] = 1
    espacio[i,:] = filas_inf




espacio[0,0] = -4
espacio[0,1] = 1            #condici n de contorno para el punto
                                  V_00, introduzco a mano.
espacio[0,dim-1] = 1

espacio[dim**2 - 1, dim**2 - 1] = -4
espacio[dim**2 - 1, dim**2 - 2] = 1          #condici n de
                                  contorno para el punto V_dim**
                                  2dim**2.
espacio[dim**2 - 1, dim**2 - dim] = 1


for i in range(1, dim**2 - 1):
    espacio[i,0] = 0            #condici n de contorno para los
                                      puntos  en contacto con los
                                      margenes izquierdo y derecho.
    espacio[i,dim**2 - 1] = 0   #estos son los puntos de la forma
                                      V_i0 y Vidim**2-1.
```

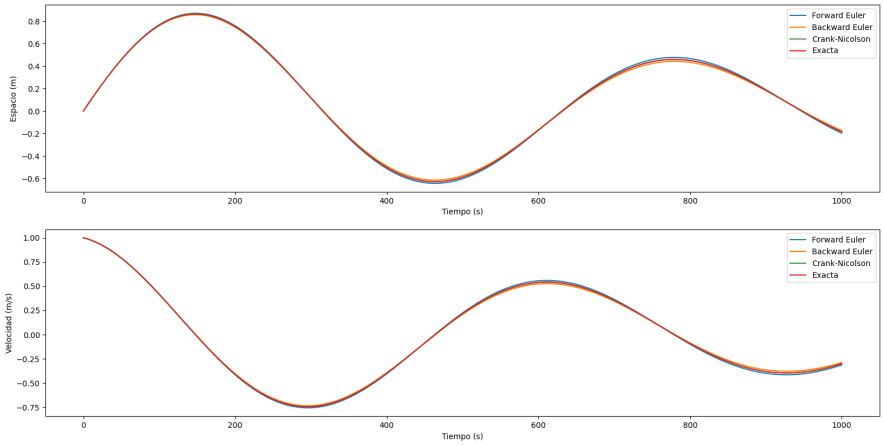
```
for i in range(1, dim):              #corregimos la matriz ya que si
                                    nos damos cuenta en la forma en
                                   la que completo
    espacio[i*dim, i*dim - 1] = 0    #la matriz 'espacio' hemos
                                       introducido al inicio de cada
                                        bloque un 1 extra
                                     #al lado de cada -4. Este debe
                                                                      ser
                                                                      un
                                                                      0
                                                                      y
                                                                      eso
                                                                      es
                                                                      lo
                                                                      que
                                                                      corregimos
                                                                      en
                                                                      este
                                                                       ltimo
                                                                      bucle
#print(espacio)


#definamos el vector de t rminos independientes:
# los primeros dim deberan ser -100, en este problema en particular
                                    .

b = np.zeros(dim**2)

espacio_neumann = copy(espacio)

b_neumann = copy(b)

for i in range(0, dim**2):

    if i < dim:
        b[i] = cond_sup
    else:                           #definimos los t rminos
                                          independientes teniendo en
                                          cuenta las condiciones
                                    #de contorno del problema.
        b[i] = cond_inf
```

```
#print(b)

#
                                    ============================================================

# for i in range(0, dim**2, dim):
#     if i < 2*dim or i > dim**2 - 2*dim:
#         b_neumann[i] += - neumann_izq - neumann_der
#     if i>= 2*dim and i< dim**2 - 2*dim:
#         b_neumann[i] += - 2*neumann_izq - 2*neumann_der
#
                                    ============================================================



def jacobi(A, x_0, dim, b):

    v = -4*np.ones(dim**2)
    D = np.diag(v, k = 0)
    D_inv = np.linalg.inv(D)
    matriz = A - D

    vector_sol =[x_0]
    sol_futuro = np.dot(D_inv,(b - np.dot(matriz,vector_sol[-1])))
    vector_sol.append(sol_futuro)

    while abs(np.linalg.norm(vector_sol[-1]) - np.linalg.norm(
                                        vector_sol[-2])) > 0.01:

        sol_futuro = np.dot(D_inv,(b - np.dot(matriz,vector_sol[-1]
                                            )))
        vector_sol.append(sol_futuro)

    else:

        return vector_sol[-1]

Solucion_Solver = np.linalg.solve(espacio, b)



#mostramos vector soluci n
print(f'La soluci n es: {Solucion_Solver}')

#realizamos ahora el mallado con un mismo n mero de puntos que
                                    nuestro vector soluci n.

L = 1

dx = L/(dim)
dy = L/(dim)

xpuntos = int(L/dx) + 1
ypuntos = int(L/dy) + 1
```

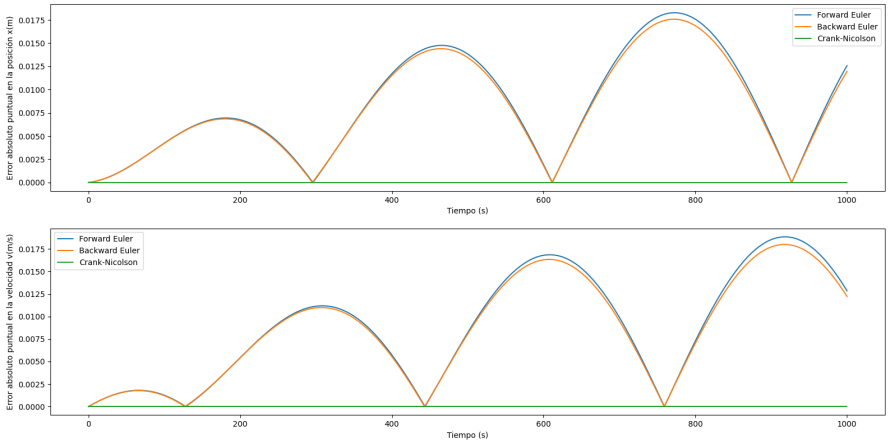
```
x = np.arange(0, L, dx)
y = np.arange(0, L, dy)

def grid(x, y):
    X, Y = np.meshgrid(x, y)
    return X, Y

X, Y = grid(x, y)

plt.figure()
plt.scatter(X, Y)
plt.contourf(X, Y, Solucion_Solver.reshape(X.shape))
C = plt.contour(X, Y, Solucion_Solver.reshape(X.shape), 8, colors='
                                    black')
plt.clabel(C, inline=1, fontsize=10)
plt.title('Potencial $V$ (Voltios)')
plt.xlabel('Eje OY (m)')
plt.ylabel('Eje OX (m)')
plt.show()

x_0 = np.linspace(100, 0, dim**2)

sol_jacobi = jacobi(espacio, x_0, dim, b)


plt.figure()
plt.scatter(X, Y)
plt.contourf(X, Y, sol_jacobi.reshape(X.shape))
C = plt.contour(X, Y, sol_jacobi.reshape(X.shape), 8, colors='black
                                    ')
plt.clabel(C, inline=1, fontsize=10)
plt.title('Potencial $V$ (Voltios) JACOBI')
plt.xlabel('Eje OY (m)')
plt.ylabel('Eje OX (m)')
plt.show()
```

3 PRÁCTICA 3

Nos piden resolver la ecuación para el oscilador armónico amortiguado. Para ello, como bien nos dice la práctica, planteamos un sistema de dos ecuaciones diferenciales de primer orden.
Con ello, obtenemos las siguientes gráficas, tanto para el error como para la solución de $x(t)$ y $v(t)$:



Gráfica 3.1: Solución para cada uno de los métodos.



Gráfica 3.2: Errores para cada uno de los métodos, comparándolos con la solución exacta.

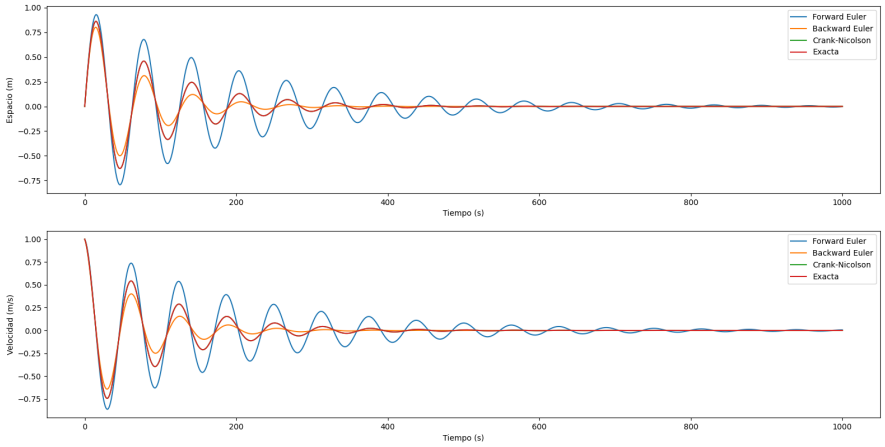
Se han utilizado los siguientes parámetros para las gráficas 3.1 y 3.2:

```
t_f = 1000
w = 1
alpha = 0.2
var_t = 0.01 #PASO TEMPORAL
cte = 1/2
```

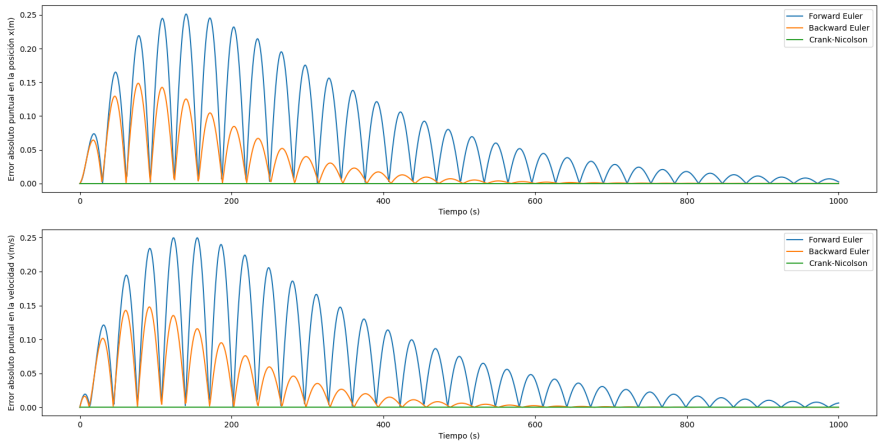
Si prestamos atención a la gráfica 3.1 podemos observar que la línea verde para el método de C-N prácticamente ni se aprecia puesto que se solapa con la solución, en rojo, exacta. Esto podemos verlo, y confirmarlo, en la gráfica 3.2, donde se observa que el error para este método se mantiene nulo, o cerca del cero.

Veamos ahora qué ocurre si aumentamos el paso temporal.

```
t_f = 1000
w = 1
alpha = 0.2
var_t = 0.1
cte = 1/2
```



Gráfica 3.3: Solución para cada uno de los métodos. Ahora para un paso mayor de tiempo.



Gráfica 3.4: Errores para cada uno de los métodos, comparándolos con la solución exacta. Ahora para un paso de tiempo mayor.

En este caso, podemos observar en la gráfica 3.3 y gráfica 3.4 que las soluciones tienen una mayor discrepancia que en el caso anterior y es que lo que está

ocurriendo es que hemos aumentado el paso temporal, lo que nos provoca que la cota aumente y introduzcamos inestabilidad si no lo compensamos modificando el resto de parámetros del problema. Sabemos que el paso temporal es una cota inferior para la estabilidad del método.
Para tiempos grandes vemos que las soluciones convergen pero esto es solo debido a que se trata de un sistema amortiguado y por la propia naturaleza del problema todas tenderán a una situación estática en $x = 0$.

3.1 CÓDIGO ENTERO PRÁCTICA 3

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct  5 16:42:51 2023

@author: juanm
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
import scipy

t_f = 1000
w = 1
alpha = 0.2
var_t = 0.1
cte = 1/2


A = np.array([[0, 1],[-w**2, - alpha]])

B = np.identity(2) + var_t*A

B_2 = np.identity(2) - var_t*A

B_2inv = np.linalg.inv(B_2)

B_3 = np.identity(2) - cte*var_t*A

B_3inv = np.linalg.inv(B_3)

B_3positiva = np.identity(2) + cte*var_t*A

def exacta(x):
    w = 1
    alpha = 0.2
    A = np.array([[0, 1],[-w**2, - alpha]])
    valor = scipy.linalg.expm(x*A)
    return valor

u_0 = [0, 1]

u = [u_0]
u_2 = [u_0]
u_3 = [u_0]
```

```
u_exacta = [u_0]

tiempo = np.linspace(0, t_f, t_f)

for i in tiempo:

    u_futuro = np.dot(B, u[-1])
    u.append(u_futuro)              #euler-FORWARD

    u_futuro2 = np.dot(B_2inv, u_2[-1])
    u_2.append(u_futuro2)                #Euler backward

    u_futuro3 = np.dot(B_3inv,np.dot(B_3positiva, u_3[-1]))  #Crank
                                        -Nicholson
    u_3.append(u_futuro3)

    u_futuro4 = exacta(i)
    u_exacta.append(u_futuro4)      #EXACTA



x = []
v = []
x_2 = []
v_2 = []
x_3 = []
v_3 = []
x_4 = []
v_4 = []

for i in u:
    x.append(i[0])
    v.append(i[-1])

for i in u_2:
    x_2.append(i[0])
    v_2.append(i[-1])

for i in u_3:
    x_3.append(i[0])
    v_3.append(i[-1])

for i in u_3:
    x_4.append(i[0])
    v_4.append(i[-1])

##### ERROR #####

e_1x = []
e_2x = []
e_3x = []
e_1v = []
e_2v = []
e_3v = []

for i in range(0, len(x_4)):
```

```
    e_1xp = abs(x_4[i] - x[i])
    e_1x.append(e_1xp)
    e_2xp = abs(x_4[i] - x_2[i])
    e_2x.append(e_2xp)
    e_3xp = abs(x_4[i] - x_3[i])
    e_3x.append(e_3xp)
    e_1vp = abs(v_4[i] - v[i])
    e_1v.append(e_1vp)
    e_2vp = abs(v_4[i] - v_2[i])
    e_2v.append(e_2vp)
    e_3vp = abs(v_4[i] - v_3[i])
    e_3v.append(e_3vp)


tiempo_plt = np.linspace(0, t_f, t_f + 1)

plt.figure()

plt.suptitle('M TODOS')

plt.subplot(2,1,1)
plt.plot(tiempo_plt, x, label = 'Forward Euler')
plt.plot(tiempo_plt, x_2, label = 'Backward Euler')
plt.plot(tiempo_plt, x_3, label = 'Crank-Nicolson')
plt.plot(tiempo_plt, x_4, label = 'Exacta')
plt.xlabel('Tiempo (s)')
plt.ylabel('Espacio (m)')
plt.legend(loc = 'best')

plt.subplot(2,1,2)
plt.plot(tiempo_plt, v, label = 'Forward Euler')
plt.plot(tiempo_plt, v_2, label = 'Backward Euler')
plt.plot(tiempo_plt, v_3, label = 'Crank-Nicolson')
plt.plot(tiempo_plt, v_4, label = 'Exacta')
plt.xlabel('Tiempo (s)')
plt.ylabel('Velocidad (m/s)')
plt.legend(loc = 'best')

plt.show()

plt.figure('ERRORES')

plt.suptitle('ERRORES (respecto a la soluci n exacta)')

plt.subplot(2,1,1)
plt.plot(tiempo_plt, e_1x, label = 'Forward Euler')
plt.plot(tiempo_plt, e_2x, label = 'Backward Euler')
plt.plot(tiempo_plt, e_3x, label = 'Crank-Nicolson')
plt.xlabel('Tiempo (s)')
plt.ylabel('Error absoluto puntual en la posici n x(m)')
plt.legend(loc = 'best')

plt.subplot(2,1,2)
plt.plot(tiempo_plt, e_1v, label = 'Forward Euler')
plt.plot(tiempo_plt, e_2v, label = 'Backward Euler')
plt.plot(tiempo_plt, e_3v, label = 'Crank-Nicolson')
plt.xlabel('Tiempo (s)')
```

```
plt.ylabel('Error absoluto puntual en la velocidad v(m/s)')
plt.legend(loc = 'best')
```

4 PRÁCTICA 4

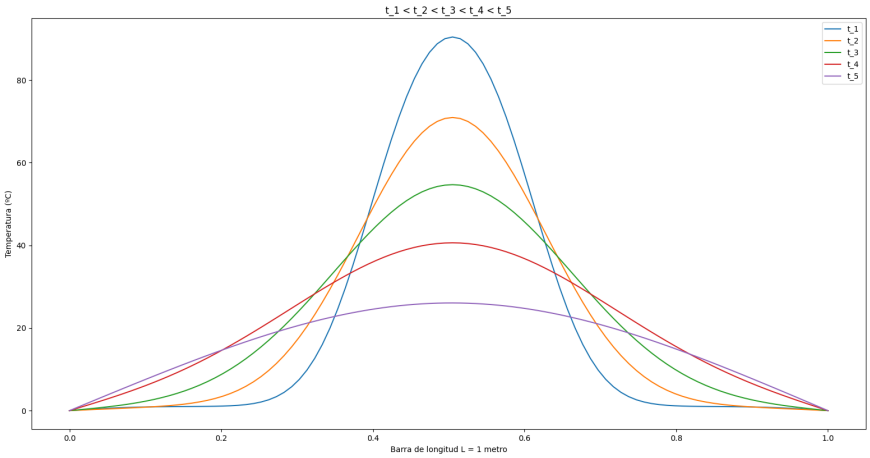
4.1 PRIMER PUNTO - EULER

Se definen una temperatura de 100 constate y un parámetro de difusión $D = 0.01$. En lugar de imponer las condiciones de contorno únicamente en cada uno de los puntos de los extremos de la barra, he decidido para este primer apartado definir un intervalo centrado a una temperatura inicial de 100 y el resto todo a 0. Los únicos puntos que se fijan a una temperatura sí que son los de los extremos, es decir, estos actuarán como focos térmicos a temperatura cero.

```
valor_central = 100
T_0 = np.ones(dim)
T_0[0] = T_0[-1] = cond_cont #condiciones de Dirichlet

izq =int(dim/2 - 10)
der = int(dim/2 + 10)
cond_int = np.arange(izq, der + 1, 1)

for i in cond_int:
    T_0[i] = valor_central
```

Se define una función, para la iteración, con el método de euler.

```
def temperatura(T_0, D, dt, dx, tiempo, dim, cond_cont):

    B1 = np.identity(dim)
    B2 = -2*np.diag(np.ones(dim), k=0) + np.diag(np.ones(dim-1), k=
                                    1) + np.diag(np.ones(dim-1),
                                    k=-1)
    T_t = [T_0]

    for k in tiempo:
            T_new_matrix = np.dot(B1, T_t[-1]) + D*(dt/(dx**2))*np.
                                                dot(B2, T_t[-1])
            T_new_matrix[0] = T_new_matrix[-1] = cond_cont
            #T_new[i] = T_t[-1][i] + D*(dt/(dx**2))*(T_t[-1][i+1] +
                                                T_t[-1][i-1] - 2*T_t
                                                [-1][i])
            T_t.append(T_new_matrix)

    return T_t
```



Gráfica 4.1: Temperatura de la barra L para diferentes instantes de tiempo.

NO HE INTRODUCIDO ANIMACIONES, ES POR ELLO QUE AÑADO LOS GRÁFICOS EN LOS QUE SE COMPARAN LA SOLUCIÓN PARA DIFERENTES TIEMPOS. SI COMPILÁIS EL SCRIPT DE LA PRÁCTICA SE GENERARÁN LA ANIMACIÓN Y LA GRÁFICA QUE APARECE AQUÍ. ESTO LO REALIZO PARA CADA UNA DE LAS PARTES DE ESTE EJERCICIO.

Debido a la diferencia de temperatura inicial podemos observar que el sistema tiende a una situación de equilibrio térmica. Esto se puede explicar debido a que todo sistema física tienen a un estado de menor energía, el sistema tenderá a equilibrar el sistema en contra del gradiente inicial.

Si no tuviéramos dos focos térmicos en los extremos a cero tendríamos, suponiendo que no tenemos interacción con el entorno, el sistema tendría a una temperatura entre la máxima y mínima iniciales. En este caso, como tenemos los focos el sistema seguirá evolucionando hasta que todos los puntos se encuentren a cero. Esto sucederá para un tiempo lo suficiente grande.

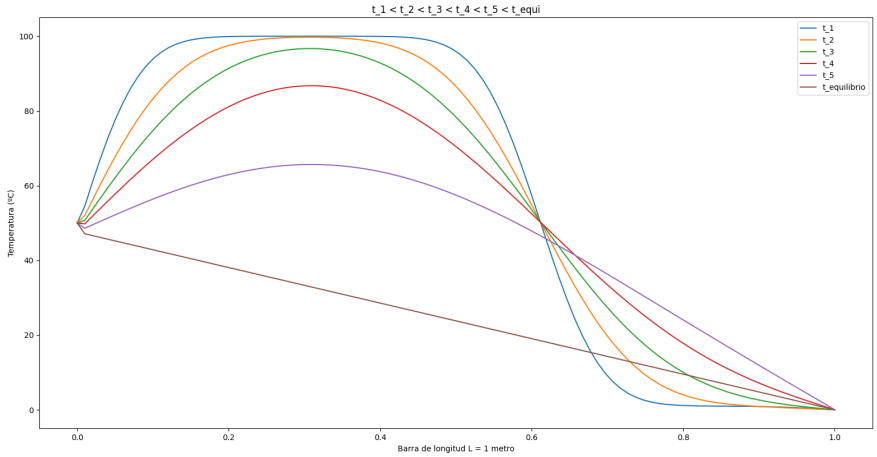
4.2 SEGUNDO PUNTO - CRANK-NICOLSON

En este punto se definen las condiciones iniciales tal cual nos piden. Una temperatura de 50 en el extremo izquierdo y a cero en el derecho. Al igual que en el primer punto, estos dos puntos serán como focos térmicos, con lo que serán fijos en cada instante de tiempo. Sigo definiendo un intervalo de puntos a la izquierda a 100 y el resto a cero, inicialmente. Es lo único que introduzco diferente a lo que se pide.

Ahora se define una función con el método de Crank-Nicolson.

```
def temperatura_CN(T_0, D, dt, dx, tiempo, dim, cond_cont, cond_CN)
                                        :

    r = D*(dt/(dx**2))/2
```

```
    T_t = [T_0]

    #A = np.zeros([dim,dim])

    diagonal_r = r*np.ones(dim - 1)
    diagonal_0 = -2*r*np.ones(dim)

    A = np.diag(diagonal_r, k = 1) + np.diag(diagonal_r, k = -1) +
                                     np.diag(diagonal_0, k = 0)
    inversa = np.linalg.inv(np.identity(dim) - A)


    for i in tiempo:
        temp_f = np.dot(inversa ,np.dot((np.identity(dim) + A), T_t
                                        [-1]))
        temp_f[0] = cond_CN
        temp_f[-1] = cond_cont
        T_t.append(temp_f)

    return T_t
```



Gráfica 4.2: Temperatura de la barra L para diferentes instantes de tiempo. Se ha implementado el método de Crank - Nicolson.

En el caso de ahora, observamos lo mismo que en el primer punto. El sistema tiende a una temperatura de equilibrio entre la máxima y mínima inicial. A diferencia con el caso anterior, dicha temperatura no será a cero pues tenemos un foco a 50 y otro a cero. La temperatura final estará dentro de dicho intervalo. Cada uno de los puntos de la barra tenderá a una temperatura de equilibrio debido a la asimetría del sistema. En el punto anterior todos los puntos tendían a cero en muestra de la presente simetría del problema.
Computacionalmente podemos encontrar varias diferencias notables. Éstas son poco apreciables en los resultados pero las mencionaremos igualmente ya que si se hacen notar a la hora de la implementación de cada uno de ellos.

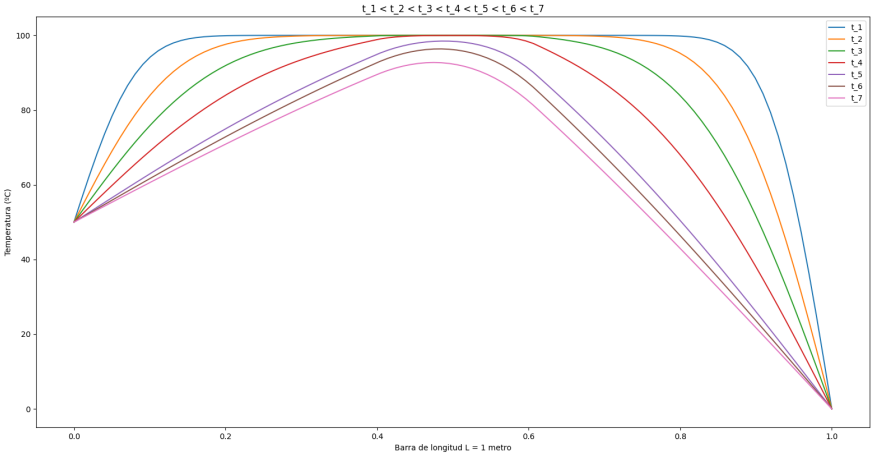
En cuanto a método de Euler tenemos un método explícito, lo que quiere decir que se calcula el siguiente instante de tiempo a partir de valores ya conocidos en el instante 'actual'. Este método es relativamente sencillo de implementar sin embargo, se requiere llevar cuidado con el paso de tiempo pues este debe ser lo suficiente pequeño para garantizar estabilidad. Esto último supone un coste computacional mayor debido a que tendremos que tener más finura en la discretización temporal.
Por otro lado, el método de C-N se trata de un método implícito y su error es menor al de Euler debido al orden de precisión que manejamos en los pasos temporales y espaciales. Este método es más preciso y estable que el de Euler pero por contra la dificultad de su implementación es notablemente superior a la de Euler. Basta observar que nos vemos obligados a calcular la inversa de matrices para la iteración temporal.

4.3 TERCER PUNTO

Aquí se trata de coger el caso anterior y modificar en un intervalo de la barra el coeficiente de difusión.

```
D = 0.01
D_int = 0.001

D0 = D*np.ones(dim)

for i in range(int(0.4*dim), int(0.6*dim) + 1, 1):

    D0[i] = D_int
```

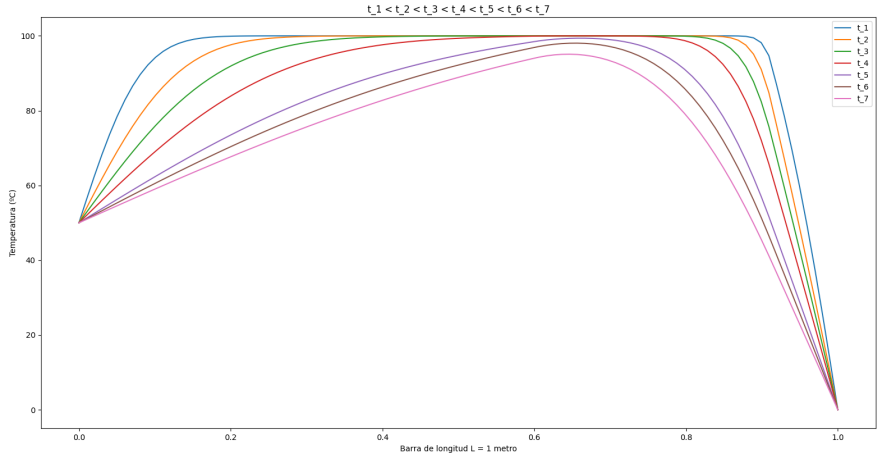
Definimos dos coeficientes. Definimos un vector de $D_{,s}$ y procedemos a intercambiar los valores por el otro coeficiente en el intervalo deseado, en este caso entre el 40% y el 60% de la barra.



Gráfica 4.3: Temperatura de la barra L para diferentes instantes de tiempo. El

coeficiente de difusión toma lo valores de 0.001 entre el 40% y el 60% de la barra y de 0.01 en el resto.

Los instantes de tiempos que he elegido para la representación son muy similares a los de los demás casos. Podemos observar como a la barra le cuesta más disminuir la temperatura. Este resultado era de esperar pues el parámetro de difusión se puede entender cómo la velocidad con la cuál el gradiente térmico disminuye. En otra palabras, la velocidad con la que se trasmite la energía a lo largo de la barra. Si el coeficiente disminuye entonces la barra tardará un tiempo mayor en estabilizarse. Le costará más tiempo que la temperatura baje en dicho intervalo. Veamos cómo cambia si elegimos el intervalo entre el 60% y el 90% de la barra.



Gráfica 4.4: Temperatura de la barra L para diferentes instantes de tiempo. El coeficiente de difusión toma lo valores de 0.001 entre el 60% y el 90% de la barra y de 0.01 en el resto.

4.4 CÓDIGO ENTERO - PRIMER PUNTO

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 10 18:06:47 2023

@author: juanm
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
import scipy
import math
from matplotlib import animation
```

```
L = 1

dim = 100

dx = 0.01

dt = 0.001



D = 0.01
cond_cont = 0
izq =int(dim/2 - 10)
der = int(dim/2 + 10)
cond_int = np.arange(izq, der + 1, 1)


T_1 = np.ones(dim)

valor_central = 100 #KELVIN
T_0 = np.ones(dim) ###condiciones iniciales
T_0[0] = T_0[-1] = cond_cont #condiciones de Dirichlet
T_1[0] = 50
T_1[-1] = 0
for i in cond_int:
    T_0[i] = valor_central

for i in cond_int:
    T_1[i] = valor_central

t_f = 100

tiempo = np.arange(0, t_f, dt)


def temperatura(T_0, D, dt, dx, tiempo, dim, cond_cont):

    B1 = np.identity(dim)
    B2 = -2*np.diag(np.ones(dim), k=0) + np.diag(np.ones(dim-1), k=
                                    1) + np.diag(np.ones(dim-1),
                                    k=-1)
    T_t = [T_0]

    for k in tiempo:
            T_new_matrix = np.dot(B1, T_t[-1]) + D*(dt/(dx**2))*np.
                                                dot(B2, T_t[-1])
            T_new_matrix[0] = T_new_matrix[-1] = cond_cont
            #T_new[i] = T_t[-1][i] + D*(dt/(dx**2))*(T_t[-1][i+1] +
                                                T_t[-1][i-1] - 2*T_t
                                                [-1][i])
            T_t.append(T_new_matrix)

    return T_t



def temperatura_CN(T_0, D, dt, dx, tiempo, dim, cond_cont):
```

```
    r = D*(dt/(dx**2))/2

    T_t = [T_0]

    #A = np.zeros([dim, dim])

    diagonal_r = r*np.ones(dim - 1)
    diagonal_0 = -2*r*np.ones(dim)
    A = np.diag(diagonal_r, k = 1) + np.diag(diagonal_r, k = -1) +
                                        np.diag(diagonal_0, k = 0)
    inversa = np.linalg.inv(np.identity(dim) - A)


    for i in tiempo:
        temp_f = np.dot(inversa ,np.dot((np.identity(dim) + A), T_t
                                            [-1]))
        T_t.append(temp_f)

    return T_t


sol = temperatura(T_0, D, dt, dx, tiempo, dim, cond_cont)

sol_CN = temperatura_CN(T_1, D, dt, dx, tiempo, dim, cond_cont)

fig = plt.figure()
ax = plt.axes(xlim=(0, 1), ylim=(0, valor_central + 10))
line, = ax.plot([], [], lw=2)

# initialization function: plot the background of each frame
def init():
    line.set_data([], [])
    return line,

# animation function.  This is called sequentially
def animate(i):
    x = np.linspace(0, L, dim)
    y = sol[i]
    line.set_data(x, y)
    return line,

# call the animator.  blit=True means only re-draw the parts that
                                    have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(tiempo), interval=10,
                                                                    blit
                                                                    =
                                                                    True
                                                                    )

# save the animation as an mp4.  This requires ffmpeg or mencoder
                                    to be
# installed.  The extra_args ensure that the x264 codec is used, so
                                     that
# the video can be embedded in html5.  You may need to adjust this
                                    for
```

```
# your system: for more information, see
# http://matplotlib.sourceforge.net/api/animation_api.html

#anim.save('basic_animation.mp4', fps=30, extra_args=['-vcodec', '
                                libx264'])
plt.xlabel(f'Barra de longitud L = {1} metro')
plt.ylabel('Temperatura ( C )')
plt.title(f'BARRA CON EXTREMOS A {cond_cont}  C ')
plt.show()



plt.figure()
plt.title('t_1 < t_2 < t_3 < t_4 < t_5')
plt.plot(np.linspace(0, L, dim), sol[200], label='t_1')
plt.plot(np.linspace(0, L, dim), sol[500], label='t_2')
plt.plot(np.linspace(0, L, dim), sol[1000], label='t_3')
plt.plot(np.linspace(0, L, dim), sol[2000], label='t_4')
plt.plot(np.linspace(0, L, dim), sol[5000], label='t_5')
plt.xlabel(f'Barra de longitud L = {1} metro')
plt.ylabel('Temperatura ( C )')
plt.legend()
plt.show()
```

4.5 CÓDIGO ENTERO - SEGUNDO PUNTO

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 12 17:29:36 2023

@author: juanm
"""




import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
import scipy
import math
from matplotlib import animation



L = 1

dim = 100

dx = 0.01

dt = 0.001


cond_CN = 50
```

```
D = 0.01
cond_cont = 0
izq =int(dim/2 - 10)
der = int(dim/2 + 10)
cond_int = np.arange(izq, der + 1, 1)
cond_int_CN = np.arange(1, der + 1, 1)

T_1 = np.ones(dim)

valor_central = 100 #KELVIN
T_0 = np.ones(dim) ###condiciones iniciales
T_0[0] = T_0[-1] = cond_cont #condiciones de Dirichlet

T_1[0] = cond_CN
T_1[-1] = 0


for i in cond_int_CN:
    T_1[i] = valor_central

t_f = 100

tiempo = np.arange(0, t_f, dt)




def temperatura_CN(T_0, D, dt, dx, tiempo, dim, cond_cont, cond_CN)
                                    :

    r = D*(dt/(dx**2))/2

    T_t = [T_0]

    #A = np.zeros([dim,dim])

    diagonal_r = r*np.ones(dim - 1)
    diagonal_0 = -2*r*np.ones(dim)

    A = np.diag(diagonal_r, k = 1) + np.diag(diagonal_r, k = -1) +
                                        np.diag(diagonal_0, k = 0)
    inversa = np.linalg.inv(np.identity(dim) - A)


    for i in tiempo:
        temp_f = np.dot(inversa ,np.dot((np.identity(dim) + A), T_t
                                            [-1]))
        temp_f[0] = cond_CN
        temp_f[-1] = cond_cont
        T_t.append(temp_f)

    return T_t


sol_CN = temperatura_CN(T_1, D, dt, dx, tiempo, dim, cond_cont,
                                    cond_CN)
```

```
fig = plt.figure()
ax = plt.axes(xlim=(0, 1), ylim=(0, valor_central + 10))
line, = ax.plot([], [], lw=2)

# initialization function: plot the background of each frame
def init():
    line.set_data([], [])
    return line,

# animation function.  This is called sequentially
def animate(i):
    x = np.linspace(0, L, dim)
    y = sol_CN[i]
    line.set_data(x, y)
    return line,

# call the animator.  blit=True means only re-draw the parts that
                                    have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(tiempo), interval=1, blit
                                                                    =
                                                                    True
                                                                    )

# save the animation as an mp4.  This requires ffmpeg or mencoder
                                    to be
# installed.  The extra_args ensure that the x264 codec is used, so
                                     that
# the video can be embedded in html5.  You may need to adjust this
                                    for
# your system: for more information, see
# http://matplotlib.sourceforge.net/api/animation_api.html

#anim.save('basic_animation.mp4', fps=30, extra_args=['-vcodec', '
                                    libx264'])
plt.xlabel(f'Barra de longitud L = {1} metro')
plt.ylabel('Temperatura ( C )')
plt.title(f'BARRA CON UN EXTREMO A {cond_cont}  C  Y OTRO A {
                                    cond_CN}  C ')
plt.show()


plt.figure()
plt.title('t_1 < t_2 < t_3 < t_4 < t_5 < t_equi')
plt.plot(np.linspace(0, L, dim), sol_CN[200], label='t_1')
plt.plot(np.linspace(0, L, dim), sol_CN[500], label='t_2')
plt.plot(np.linspace(0, L, dim), sol_CN[1000], label='t_3')
plt.plot(np.linspace(0, L, dim), sol_CN[2000], label='t_4')
plt.plot(np.linspace(0, L, dim), sol_CN[5000], label='t_5')
plt.plot(np.linspace(0, L, dim), sol_CN[-1], label='t_equilibrio')
plt.xlabel(f'Barra de longitud L = {1} metro')
plt.ylabel('Temperatura ( C )')
plt.legend()
plt.show()
```

4.6 CÓDIGO ENTERO - TERCER PUNTO

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov  1 10:51:52 2023

@author: juanm
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
import scipy
import math
from matplotlib import animation

L = 1

dim = 100

dx = 0.01

dt = 0.001

T = 100

T0 = T*np.ones(dim)
T0[0] = 50
T0[dim - 1] = 0

t_f = 1000

tiempo = np.arange(0, t_f, dt)

D = 0.01
D_int = 0.001

D0 = D*np.ones(dim)

for i in range(int(0.4*dim), int(0.6*dim) + 1, 1):

    D0[i] = D_int
paso = 50
cantidad = len(tiempo)/(paso)

M_D = np.zeros([dim, dim])

for i in range(0, dim):
    M_D[:,i] = D0

def temperatura(T_0, D, dt, dx, tiempo, dim, paso):

    B1 = np.identity(dim)
    B2 = -2*np.diag(np.ones(dim), k=0) + np.diag(np.ones(dim-1), k=
                                        1) + np.diag(np.ones(dim-1),
                                        k=-1)
    T_t = [T_0]
```

```
    for k in range(0, len(tiempo), paso):
            T_new_matrix = np.dot(B1, T_t[-1]) + D*(dt/(dx**2))*np.
                                                dot(B2, T_t[-1])
            T_new_matrix[0] = 50
            T_new_matrix[-1] = 0
            #T_new[i] = T_t[-1][i] + D*(dt/(dx**2))*(T_t[-1][i+1] +
                                                 T_t[-1][i-1] - 2*T_t
                                                [-1][i])
            T_t.append(T_new_matrix)

    return T_t


sol = temperatura(T0, D0, dt, dx, tiempo, dim, paso)



fig = plt.figure()
ax = plt.axes(xlim=(0, 1), ylim=(0, T + 10))
line, = ax.plot([], [], lw=2)

# initialization function: plot the background of each frame
def init():
    line.set_data([], [])
    return line,

# animation function.  This is called sequentially
def animate(i):
    x = np.linspace(0, L, dim)
    y = sol[i]
    line.set_data(x, y)
    return line,

# call the animator.  blit=True means only re-draw the parts that
                                    have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(tiempo), interval=10,
                                                                    blit
                                                                    =
                                                                    True
                                                                    )

# save the animation as an mp4.  This requires ffmpeg or mencoder
                                    to be
# installed.  The extra_args ensure that the x264 codec is used, so
                                     that
# the video can be embedded in html5.  You may need to adjust this
                                    for
# your system: for more information, see
# http://matplotlib.sourceforge.net/api/animation_api.html

#anim.save('basic_animation.mp4', fps=30, extra_args=['-vcodec', '
                                    libx264'])
plt.xlabel(f'Barra de longitud L = {1} metro')
plt.ylabel('Temperatura ( C )')
plt.title(f'BARRA CON EXTREMOS A  C ')
```

```
plt.show()



plt.figure()
plt.title('t_1 < t_2 < t_3 < t_4 < t_5 < t_6 < t_7')
plt.plot(np.linspace(0, L, dim), sol[200], label='t_1')
plt.plot(np.linspace(0, L, dim), sol[500], label='t_2')
plt.plot(np.linspace(0, L, dim), sol[1000], label='t_3')
plt.plot(np.linspace(0, L, dim), sol[2000], label='t_4')
plt.plot(np.linspace(0, L, dim), sol[5000], label='t_5')
plt.plot(np.linspace(0, L, dim), sol[7000], label='t_6')
plt.plot(np.linspace(0, L, dim), sol[10000], label='t_7')
plt.xlabel(f'Barra de longitud L = {1} metro')
plt.ylabel('Temperatura ( C )')
plt.legend()
plt.show()
```

5 PRÁCTICA 6

5.1 PRIMER PUNTO - EXPLÍCITO

Para este punto se define una función para realizar la iteración temporal. Para simular que la velocidad inicial sea nula se definen los dos instantes iniciales iguales.

```
def inicial(x, valor, n, L):
    func = valor*(np.sin(x*np.pi*n/L) + np.cos(x*np.pi*(n + 1)/L))
    return func
```

```
Cond_iniciales = inicial(vector, valor, n, L)
```

```
def ecuacion(M1, M2, Cond_iniciales, tiempo):

    M0 = [Cond_iniciales, Cond_iniciales]

    for i in tiempo:
        V_futuro = np.dot(M1, M0[-1]) + np.dot(M2, M0[-2])
        M0.append(V_futuro)

    return M0
```

5.2 SEGUNDO PUNTO - IMPLÍCITO

Se definen ahora una función para la iteración mediante un método implícito. A la hora de representar se le dará las misma condiciones iniciales que al primer punto. Para este método hemos tenido que definir matrices inversas, al igual que cuando hemos implementado métodos como el de C-N.

```
def implicito(M1inv, M2, M3, Cond_iniciales, tiempo):

    M0 = [Cond_iniciales, Cond_iniciales]

    for i in tiempo:
        V_futuro = np.dot(M1inv, np.dot(M2, M0[-1]) + np.dot(M3, M0
                                                [-2]))
        M0.append(V_futuro)

    return M0
```

5.3 TERCER PUNTO - TELÉGRAFO

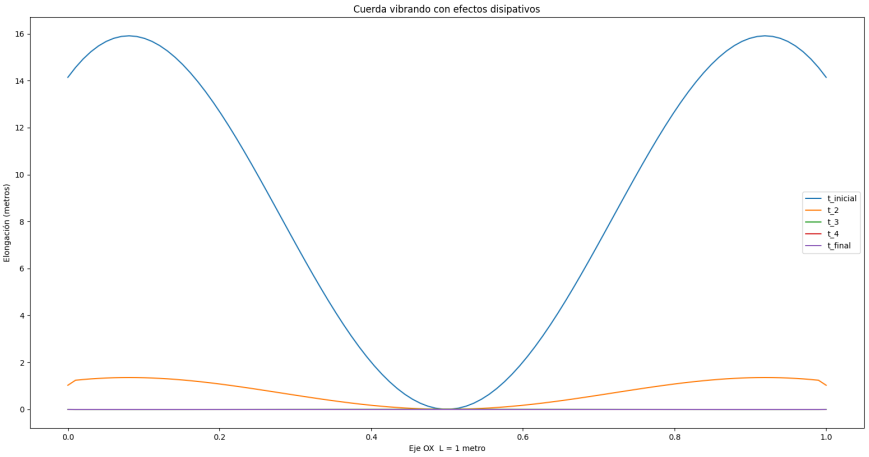
Por último, modificamos el segundo punto para definir una nueva iteración añadiendo un término. Ahora la condición inicial que tomamos cambia y consideramos la que nos dice el apartado $sin(n\pi x)$.

```
def cond_ini_telegrafo(x, n):

    return 10*np.sin(n*np.pi*x)



cond_inicial_telegrafo = cond_ini_telegrafo(vector, n)
```

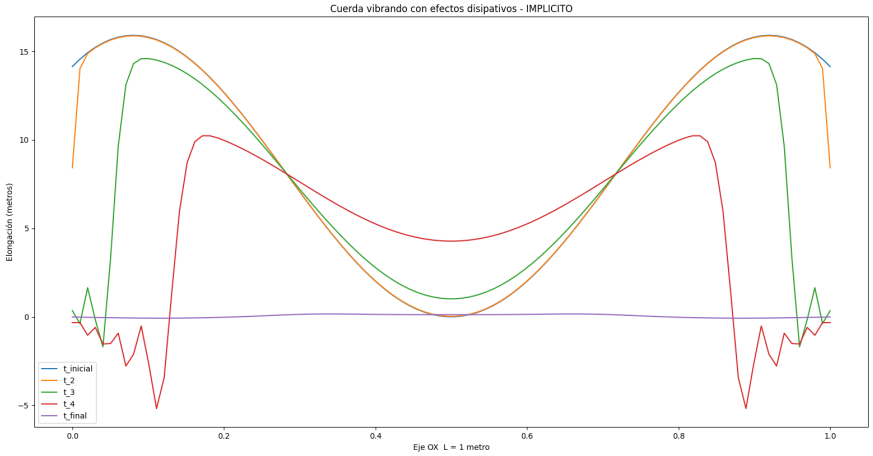
```
def telegrafo(M_inv, M2, M3, Cond_iniciales, tiempo):

    M0 = [Cond_iniciales, Cond_iniciales]

    for i in tiempo:
        V_futuro = np.dot(M_inv, np.dot(M2, M0[-1]) + np.dot(M3, M0
                                                [-2]))
        M0.append(V_futuro)


    return M0
```

5.4 RESULTADOS PARA DISTINTOS VALORES DE N

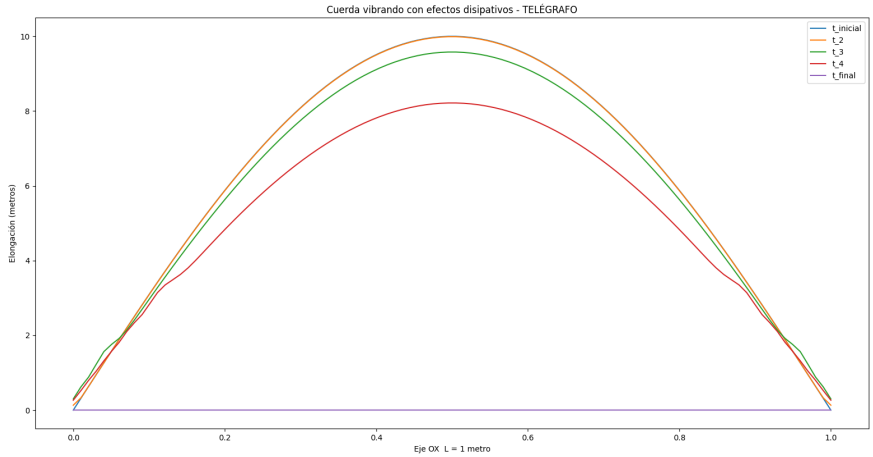
5.4.1 n = 1



Gráfica 5.1: Vibración de la cuerda de longitud $L = 1$ para $n = 1$ con un método explícito.

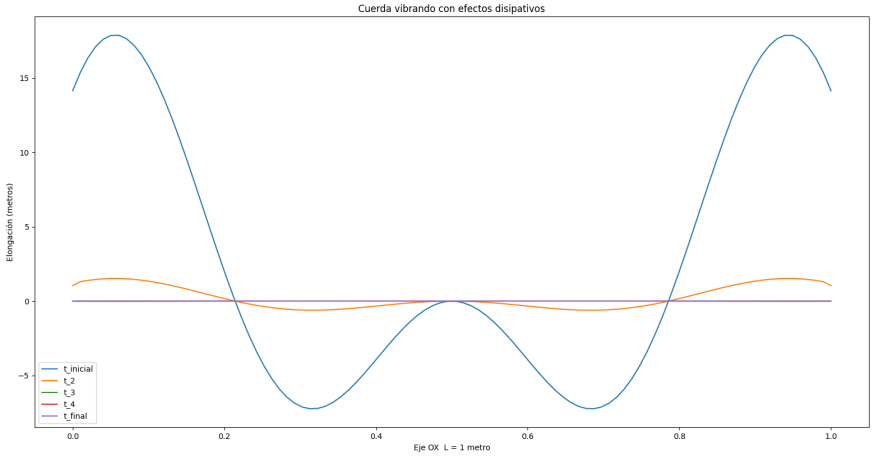


Gráfica 5.2: Vibración de la cuerda de longitud $L = 1$ para $n = 1$ con un método implícito.

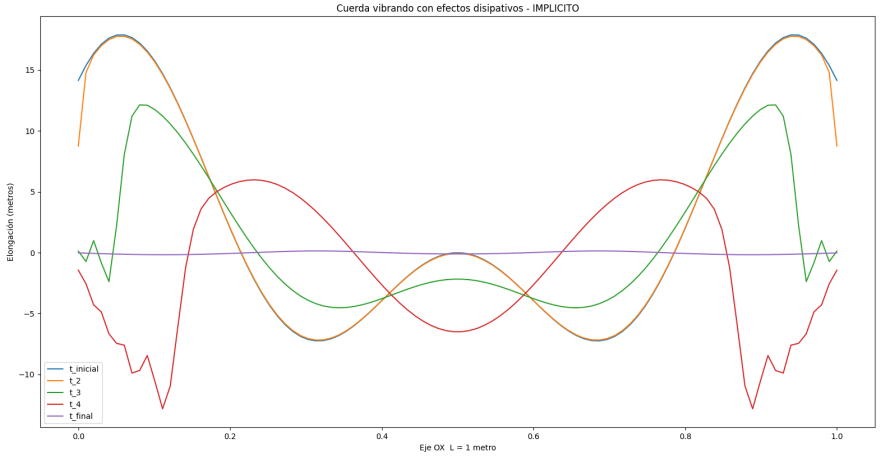


Gráfica 5.3: Vibración de la cuerda de longitud $L = 1$ para $n = 1$ en la ecuación del telégrafo.

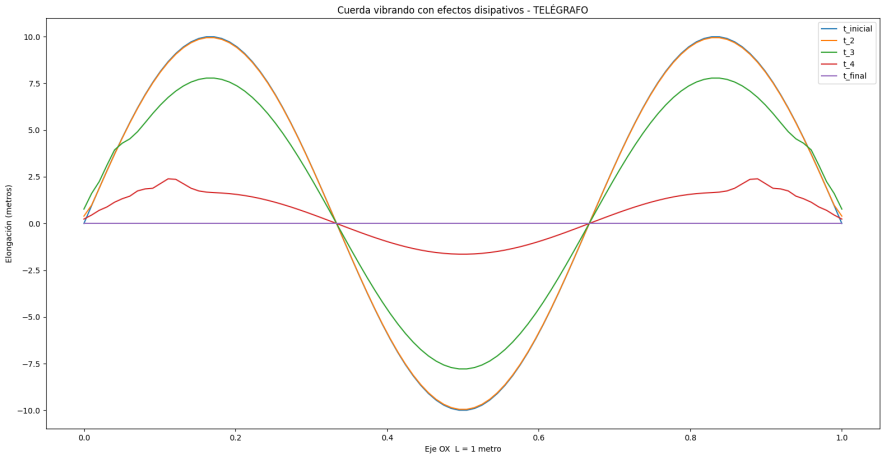
5.4.2 n = 3



Gráfica 5.4: Vibración de la cuerda de longitud $L = 1$ para $n = 3$ con un método explícito.

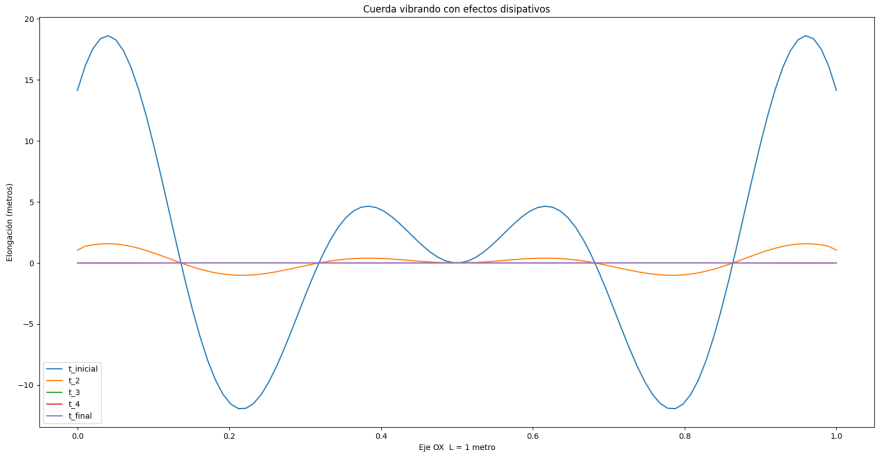


Gráfica 5.5: Vibración de la cuerda de longitud $L = 1$ para $n = 3$ con un método implícito.

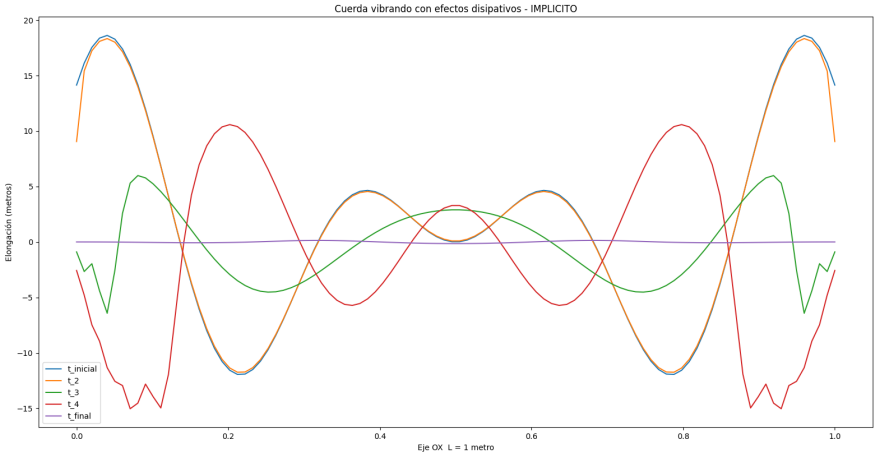


Gráfica 5.6: Vibración de la cuerda de longitud $L = 1$ para $n = 3$ en la ecuación del telégrafo.

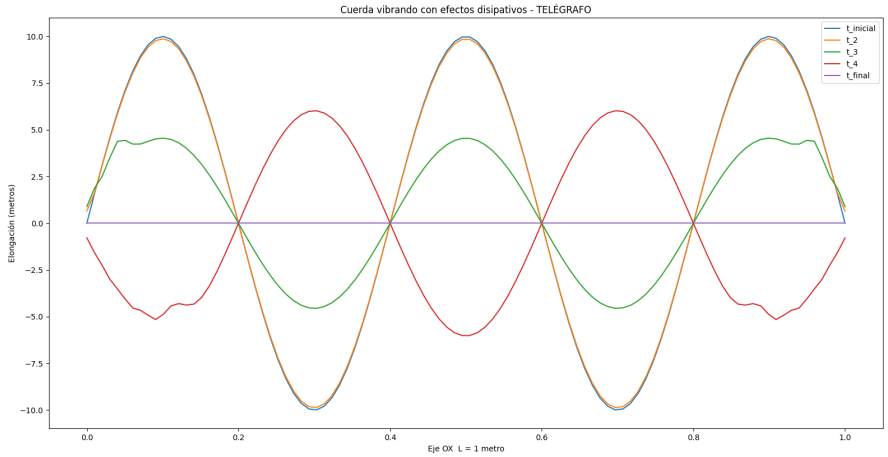
5.4.3 n = 5



Gráfica 5.7: Vibración de la cuerda de longitud $L = 1$ para $n = 5$ con un método explícito.



Gráfica 5.8: Vibración de la cuerda de longitud $L = 1$ para $n = 5$ con un método implícito.



Gráfica 5.9: Vibración de la cuerda de longitud $L = 1$ para $n = 5$ en la ecuación del telégrafo.

En cada uno de los casos hemos utilizado diferentes valores de n. Para las gráficas de los dos primeros puntos hemos utilizado condiciones iniciales tales que:

```
def inicial(x, valor, n, L):
    func = valor*(np.sin(x*np.pi*n/L) + np.cos(x*np.pi*(n + 1)/L))
    return func
```

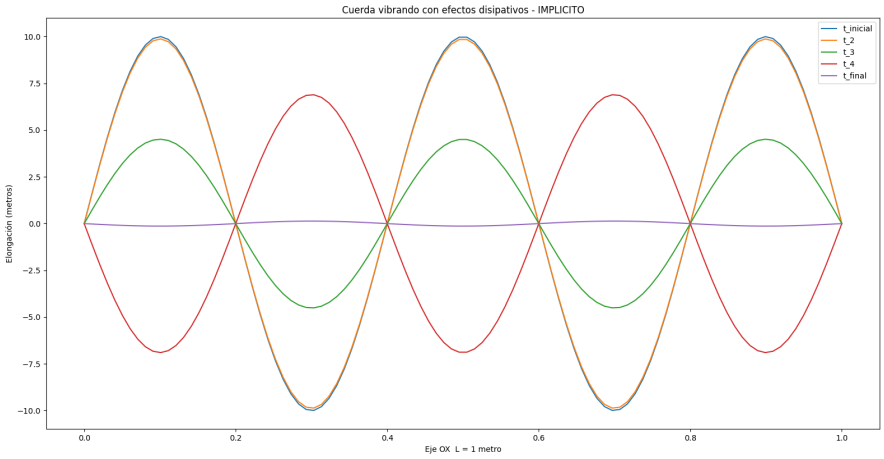
Mientras que para las gráficas de la ecuación del telégrafo nos hemos ceñido a las condiciones que nos piden. Hemos aprovechado y hemos definido ambas funciones en función de n para así estudiar en los tres puntos cómo cambian las soluciones.

```
def cond_ini_telegrafo(x, n):

    return 10*np.sin(n*np.pi*x)
```

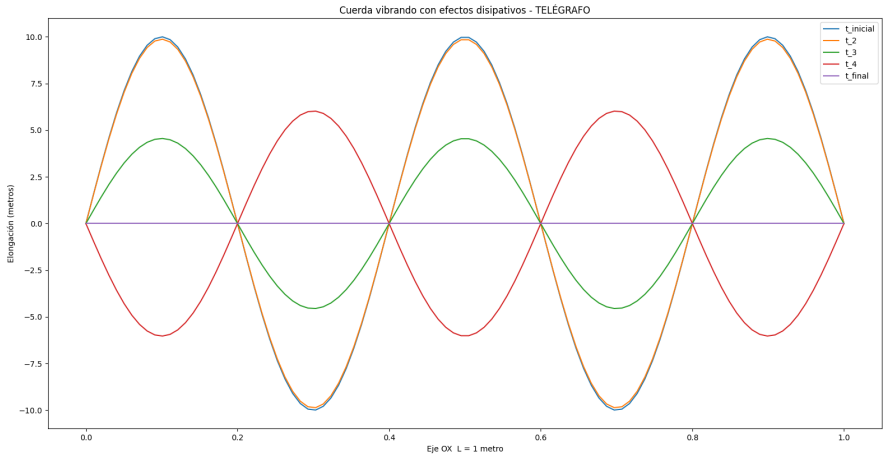
En las del segundo y tercer puntos podemos observar que para tiempos avanzados tenemos la propagación de ruido. Tras indagar en el código, llego a la conclusión de que el problema está en que los extremos de la 'cuerda' quedan libres. Se ha solucionado fijando, en cada iteración temporal, los extremos en cero.

```
V_futuro[0] = V_futuro[-1] = 0
```

Notar también, que para el caso explícito e implícito se utilizan unas condiciones iniciales tales que estás son proporcionales a una combinación lineal de senos y cosenos, con lo que sería necesario buscar la forma de fijar los extremos (ya que en cero es imposible) dadas estás condiciones iniciales, para evitar el ruido mencionado. He tomado la decisión de utilizar como condición inicial un seno, como en el caso del telégrafo, para mostrar como se soluciona el problema.



Gráfica 5.10: Vibración de la cuerda de longitud $L = 1$ para $n = 5$ con un método implícito. Ahora para una condición inicial proporcional a $sin(2\pi nx)$.



Gráfica 5.11: Vibración de la cuerda de longitud $L = 1$ para $n = 5$ con un método implícito. Ahora para una condición inicial proporcional a $sin(2\pi nx)$.

En cuanto al comportamiento de éstas respecto a la variación del valor de n podemos observar cómo las soluciones presentan más nodos al oscilar, modos normales.

Hacer notar por último, que para los resultados del primer punto, la implementación de un método explícito, vemos que no observamos el ruido como en los otros dos puntos. Este tipo de método a pesar de presentar un mayor coste computacional, pues se debe de dar un paso temporal lo suficiente pequeño para evitar inestabilidades, son más sencillos de trabajar y con ello es menos probable que nos encontremos con problemas como los comentados en los otros puntos.

TODOS LOS PUNTOS AL COMPILARSE SE CARGARÁN CON LAS RESPECTIVAS GRÁFICAS MÁS SU ANIMACIÓN.

5.5 CÓDIGO ENTERO PRÁCTICA 6

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 19 17:44:34 2023

@author: juanm
"""


import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
import scipy
import math
from matplotlib import animation


dim = 100
L = 1
deltax = 1/50
c = 1 #velocidad de la luz (m/s)

deltat = 0.5*(deltax**2)/(c**2)

deltat2 = (0.5)*(deltax**2)/(c**2)

rho = 0.01

kappa = 0.001

X_min = 0
X_max = 1

T = 40

espacio = np.arange(X_min, X_max, deltax)
tiempo = np.arange(0, T, deltat)
tiempo2 = np.arange(0, T, deltat2)

A = (2/(rho + deltat))*(rho - (c**2)*rho*(deltat**2)/(deltax**2) +
                                    kappa*(deltat**2))

B = (rho/(rho + deltat))*(deltat*c/deltax)**2

C = -rho/(rho + deltat)

parametros = [A, B, C]

#generamos las matrices para la recursividad temporal

M1 = np.diag(A*np.ones(dim), k = 0) + np.diag(B*np.ones(dim - 1), k
                                     = 1) + np.diag(B*np.ones(dim - 1
                                    ), k = -1)

M2 = C*np.identity(dim)
```

```
vector = np.linspace(0, 1, dim)

valor = 10*np.sqrt(2/L)

n = 5
#func = valor*(np.sin(x*np.pi*n/L) + np.cos(x*np.pi*(n + 1)/L))
def inicial(x, valor, n, L):
    func = valor*(np.sin(x*np.pi*n/L) + np.cos(x*np.pi*(n + 1)/L))
    return func

Cond_iniciales = inicial(vector, valor, n, L)

def cond_ini_telegrafo(x, n):

    return 10*np.sin(n*np.pi*x)



cond_inicial_telegrafo = cond_ini_telegrafo(vector, n)

def ecuacion(M1, M2, Cond_iniciales, tiempo):

    M0 = [Cond_iniciales, Cond_iniciales]

    for i in tiempo:
        V_futuro = np.dot(M1, M0[-1]) + np.dot(M2, M0[-2])
        M0.append(V_futuro)

    return M0

A2 = (2*rho*(c*deltat)**2 + 2*kappa*deltat*deltax**2 + rho*deltax**
                                    2)/(((deltax*deltat)**2)*rho)
B2 = -(c**2)/(deltax**2)
C2 = (2*rho + 2*kappa*deltat)/(rho*deltat**2)
D2 = -1/(deltat**2)

M_imp = np.diag(A2*np.ones(dim), k=0) + np.diag(B2*np.ones(dim-1),
                                    k=1) + np.diag(B2*np.ones(dim-1),
                                     k=-1)

M_imp_inversa = np.linalg.inv(M_imp)

M_C2 = C2*np.identity(dim)

M_D2 = D2*np.identity(dim)

def implicito(M1inv, M2, M3, Cond_iniciales, tiempo):

    M0 = [Cond_iniciales, Cond_iniciales]

    for i in tiempo:
        V_futuro = np.dot(M1inv, np.dot(M2, M0[-1]) + np.dot(M3, M0
                                            [-2]))
        M0.append(V_futuro)

    return M0
```

```
A3 = (deltax**2 + deltat*deltax**2 + 2*(deltat*deltax)**2 + 2*
                                    deltat**2)/(deltat*deltax)**2
B3 = -1/(deltax**2)
C3 = (2 + deltat)/deltat**2
D3 = -1/deltat**2

M_1tlg = np.diag(A3*np.ones(dim), k=0) + np.diag(B3*np.ones(dim-1),
                                     k=1) + np.diag(B3*np.ones(dim-1)
                                    , k=-1)
M_tlg_inv = np.linalg.inv(M_1tlg)

M_C3 = C3*np.identity(dim)
M_D3 = D3*np.identity(dim)

def telegrafo(M_inv, M2, M3, Cond_iniciales, tiempo):

    M0 = [Cond_iniciales, Cond_iniciales]

    for i in tiempo:
        V_futuro = np.dot(M_inv, np.dot(M2, M0[-1]) + np.dot(M3, M0
                                            [-2]))
        M0.append(V_futuro)


    return M0


sol = ecuacion(M1, M2, Cond_iniciales, tiempo)

sol_imp = implicito(M_imp_inversa, M_C2, M_D2, Cond_iniciales,
                                    tiempo2)

sol_tlg = telegrafo(M_tlg_inv, M_C3, M_D3, cond_inicial_telegrafo,
                                    tiempo2)

fig = plt.figure()
ax = plt.axes(xlim=(0, 1), ylim=( - valor - 1, valor + 1))
line, = ax.plot([], [], lw=2)

# initialization function: plot the background of each frame
def init():
    line.set_data([], [])
    return line,

# animation function.  This is called sequentially
def animate(i):
    x = np.linspace(0, L, dim)
    y = sol[i]
    line.set_data(x, y)
    return line,

# call the animator.  blit=True means only re-draw the parts that
                                    have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(tiempo), interval=100,
                                                                blit
                                                                =
```

```
                                                                    True
                                                                    )

# save the animation as an mp4.  This requires ffmpeg or mencoder
                                    to be
# installed.  The extra_args ensure that the x264 codec is used, so
                                    that
# the video can be embedded in html5.  You may need to adjust this
                                    for
# your system: for more information, see
# http://matplotlib.sourceforge.net/api/animation_api.html

#anim.save('basic_animation.mp4', fps=30, extra_args=['-vcodec', '
                                    libx264'])
plt.xlabel(f'Eje OX  L = {1} metro')
plt.ylabel('Elongaci n (metros)')
plt.title(f'Cuerda vibrando con efectos disipativos')
plt.show()


plt.figure()
plt.plot(np.linspace(0, L, dim), sol[0], label = 't_inicial')
plt.plot(np.linspace(0, L, dim), sol[100], label = 't_2')
plt.plot(np.linspace(0, L, dim), sol[700], label = 't_3')
plt.plot(np.linspace(0, L, dim), sol[1500], label = 't_4')
plt.plot(np.linspace(0, L, dim), sol[-1], label = 't_final')
plt.xlabel(f'Eje OX  L = {1} metro')
plt.ylabel('Elongaci n (metros)')
plt.title(f'Cuerda vibrando con efectos disipativos')
plt.legend()
plt.show()

plt.figure()
plt.plot(np.linspace(0, L, dim), sol_imp[0], label = 't_inicial')
plt.plot(np.linspace(0, L, dim), sol_imp[100], label = 't_2')
plt.plot(np.linspace(0, L, dim), sol_imp[700], label = 't_3')
plt.plot(np.linspace(0, L, dim), sol_imp[1500], label = 't_4')
plt.plot(np.linspace(0, L, dim), sol_imp[-1], label = 't_final')
plt.xlabel(f'Eje OX  L = {1} metro')
plt.ylabel('Elongaci n (metros)')
plt.title(f'Cuerda vibrando con efectos disipativos - IMPLICITO')
plt.legend()
plt.show()


plt.figure()
plt.plot(np.linspace(0, L, dim), sol_tlg[0], label = 't_inicial')
plt.plot(np.linspace(0, L, dim), sol_tlg[100], label = 't_2')
plt.plot(np.linspace(0, L, dim), sol_tlg[700], label = 't_3')
plt.plot(np.linspace(0, L, dim), sol_tlg[1500], label = 't_4')
plt.plot(np.linspace(0, L, dim), sol_tlg[-1], label = 't_final')
plt.xlabel(f'Eje OX  L = {1} metro')
plt.ylabel('Elongaci n (metros)')
plt.title(f'Cuerda vibrando con efectos disipativos - TEL GRAFO')
plt.legend()
plt.show()
```