UNIVERSIDAD NACIONAL DE SAN AGUSTIN
FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA

Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación

Aprobación: 2022/03/01 Código: GUIA-PRLE-001

INFORMACIÓN BÁSICA					
ASIGNATURA:	Física Computacional.				
TÍTULO DE LA PRÁCTICA:	Práctica de Ecuación diferencial de Laplace.				
NÚMERO DE PRÁCTICA:	05	AÑO LECTIVO:	2022-A	NRO. SEMESTRE:	VII
FECHA DE PRESENTACIÓN:	June 11, 2022	HORA DE PRESENTACIÓN:	10:45 pm		
Integrante(s):	Alván Ventura Edsel Yael			NOTA	
DOCENTE(s):	Danny Giancarlo Apaza Veliz.				

Práctica 5
Física Computacional

Escrito por
Alván Ventura, Edsel Yael
ealvan@unsa.edu.pe

Profesor
Apaza Veliz, Danny Giancarlo
dapazav@unsa.edu.pe

June 11, 2022

1 Problema 1

Resuelva aproximadamente la ecuación de Laplace $\nabla^2 u = 0$ en un cuadrado definido por $0 < x < 4$ y $0 < y < 4$, con las condiciones en la frontera.

$$u(x, 0) == 20 \text{ y } u(x, 4) = 300 \text{ para todo } 0 < x < 4$$
$$u(0, y) = 80 \text{ y } u(4, y) = 0 \text{ para todo } 0 < y < 4$$

1.1 Análisis

La ecuación diferencial de Laplace puede ser resuelta numericamente mediante la técnica de diferencias finitas, las diferencias finitas pueden ayudarnos a aproximar mucho a una solución analítica.

Para lograrlo, necesitamos deducir esta formula:

$$u(i, j) = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j-1} + u_{i,j+1}) \tag{1}$$

A partir de las series de taylor, para deducir una aproximación numerica a la segunda derivada de la funcion $u(i, j)$:

1.2 Programación

Luego de la deducción:

1. Se creó una matriz de mxn.
2. Los contornos de la matriz se llenaron con los bordes dados por la función.
3. Luego la parte central de la matriz(no incluyendo los bordes) se lleno de la media de los valores de los valores del borde.
4. Despues de hacer eso, se implementó la técnica numérica para iterar sobre la parte central de la matriz anteriormente dicha.
5. Y por ultimo, se coloco la matriz en un mapa de calor, para ver las condiciones frontera.

Las dimensiones del array son parametros de la función llamada *Laplace*(). Esta función se denota de la siguiente manera:

$$Laplace(xu, xd, yl, yr, f, c, h) \tag{2}$$

Donde:

Parámetros:	**Descrición**	
xu:	Representa la recta $u(x, 4) = 300 = xu$	arriba del cuadrado
xd:	Representa la recta $u(x, 0) = 20 = xd$	abajo del cuadrado
yl:	Representa la recta $u(0, y) = 80 = yl$	izquierda del cuadrado
yr:	Representa la recta $u(4, y) = 300 = yr$	derecha del cuadrado
f:	Representa el numero de filas de la matriz	
c:	Representa el numero de columnas de la matriz	
h:	Representa el numero de iteraciones de la matriz	

A continuación se muestra la implementación:

```python
import matplotlib.pyplot as plt
import numpy as np
#LAPLACE ->
#Ua= Yl
#Ub= Yr
#Uc= Xd
#Ud= Xu
#N = a
#M = b
#H = 1000 iterations
#Error: 0.0001 error
# def mean() hara el trabajo de media

def fillMatrix(arr,xu,xd,yl,yr):
    arr[0,:] = xu
    arr[arr.shape[0]-1,:] = xd
    arr[0:,0] = yl
    arr[:,arr.shape[1]-1] = yr
    arr[1,:arr.shape[1]-2]
    arr[1:arr.shape[0]-1,1:arr.shape[1]-1] = (xu+xd+yl+yr)/4

def calculate(arr,times,error):
    convergencia = 0
```

```
    contador = 0
    # tmp = arr
    while(contador < times and convergencia == 0):
        tmp = arr
        for i in range(1,arr.shape[0]-1):
            for j in range(1,arr.shape[1]-1):
                arr[i,j] = 0.25*(arr[i+1][j]+arr[i-1][j]+arr[i][j+1]+arr[i][j-1])
        # if(np.linalg.norm(arr-tmp,np.inf)/np.linalg.norm(arr,np.inf) < error):
        #     convergencia = 1
        contador+=1
    # if(convergencia == 1):
    getCalorMap(arr)
    print(f"contador:{contador}\n[arr-tmp]={np.linalg.norm( tmp)}\n[arr]={np.linalg.no
    # print(arr[:int(arr.shape[0]/2),:int(arr.shape[1]/2)])

def getCalorMap(arr):
    plt.imshow(arr,cmap='hot',interpolation='nearest')
    plt.show()

def Laplace(arriba,abajo,izq,dere, filas, cols,h,error):
    malla = np.zeros((filas ,cols))#creando malla
    fillMatrix(malla,arriba,abajo,izq,dere)#llenando malla
    calculate(malla,h,error)

def main():
    filas = 100
    cols = 100
    x_upper = 0
    x_down = 80
    l_left = 20
    l_right= 300
    times = 500
    error=0.001
    Laplace(x_upper, x_down, l_left, l_right, filas, cols, times, error)
    # print(malla)

if __name__ == "__main__":
    main()
```

Como se puede ver, los datos del primer ejercicio estan puestos en la siguiente pieza de código:

```
def main():
    filas = 100
    cols = 100
    x_upper = 0
    x_down = 80
    l_left = 20
    l_right= 300
```

```
    times = 500
    error=0.001
    Laplace(x_upper, x_down, l_left, l_right, filas, cols, times, error)
```

1.3 Resultados

Los resultados son los siguientes: En la anterior imagen se puede ver, los puntos mas

calientes, donde la función alcanza sus valores máximos(color rojizo anaranjado), y la parte negra representa los valores mínimos de la función.

2 Problema 2

Use y modifique el código del ejercicio 1 y calcule las aproximaciones de la función $u(x, y)$ en el cuadrado cuadrado definido por $0 < x < 4$ y $0 < y < 4$, con las condiciones en la frontera.

$$u(x,0) = 120 \text{ y } u(x,4) = 0 \text{ para todo } 0 < x < 4 \; u(0,y) = 120 \text{ y } u(4,y) = 0 \text{ para todo } 0 < y < 4$$

2.1 Análisis

En la siguiente tabla se especifica que valores se tomaron para el problema 2, a partir de la implementación del problema 1:

2.2 Programación

La implementación es la misma que el anterior problema, la diferencia radica en:

```
def main():
    filas = 100
    cols = 100
    x_upper = 0
    x_down = 80
    l_left = 20
```

Parámetros:	**Descrición**	
xu:	Será $u(x,4) = 0 = xu$	arriba del cuadrado
xd:	Será la recta $u(x,0) = 120 = xd$	abajo del cuadrado
yl:	Será la recta $u(0,y) = 120 = yl$	izquierda del cuadrado
yr:	Será la recta $u(4,y) = 0 = yr$	derecha del cuadrado
f:	100 filas para la matriz	
c:	100 columnas para la matriz	
h:	1000 iteraciones de la matriz	

```
    l_right= 300
    times = 500
    error=0.001
    Laplace(x_upper, x_down, l_left, l_right, filas, cols, times, error)
```

2.3 Resultados

El resultado es el siguiente:

Como se puede apreciar en la anterior figura,

3 Problema 3

Implementar un código computacional para la solución del siguiente potencial:

$$V(x,y) = \frac{4}{\pi} \sum_{n=1,3,5...}^{\infty} \frac{1}{n} \left(\frac{V_2 sinh(\frac{n\pi}{b}x) + V_1 sinh(\frac{n\pi}{b}(a-x))}{sinh(\frac{n\pi}{b}a)}\right) sin(\frac{n\pi}{b}y) \tag{3}$$

3.1 Análisis

La anterior función es la solucion de una ecuación diferencial, debido a esto, debemos implementar gráficamente la función pero análiticamente(debido a que la función es análitica).

Para lograr esto, debemos tener en cuenta los parámetros de la función:

Variables y constantes	**Descrición**.
V_1	Potencial 1.
V_2	Potencial 2.
a	Distancia entre las dos laminas.
b	Anchura de las dos láminas.
$sinh()$	Se refiere a la funcion hiperbólica de $sin()$.
x,y	son los parámetros de la función en el plano cartesiano.

3.2 Programación

Para la implementación del algoritmo se siguio de la siguiente manera:

1. Se creo la matriz de mxn.
2. Luego se le indico los limites del plano cartesiano(x,y), en los que actuaría la función.
3. Después, se recorrió la matriz en toda su extensión con dos *for* anidados(uno para fila y el otro para las columnas).
4. Mientras se hacia el paso anterior, se puso adentro de los dos *for*, se uso la función analítica(1) a implementar con los valores (x, y) del plano cartesiano.
5. Todos estos valores se guardaban respectivamente en la matriz.
6. Luego se puso la matriz en un mapa de calor(se uso la librería *matplotlib*).

A continuación, se muestra la implementación:

```
from math import pi,sinh,sin
import numpy as np
import matplotlib.pyplot as plt
INFINIT0 = 100

#constantes
cts = {
    "V2":0,
    "V1":1,
    "a":1,
    "b":1,
}
#z=
#    z+4*(V2*sinh((N)*pi*x/b)+V1*sinh((N)*pi*(a-x)/b)).*
```

```python
#    sin((N)*pi*y/b)/(sinh((N)*pi*a/b)*(N)*pi)

def funcion(x,y):
    const = 4/pi
    a = cts["a"]
    b = cts["b"]
    V1 = cts["V1"]
    V2 = cts["V2"]
    general2=0
    for n in range(1,INFINIT0,2):
        # print(f"((n:{n}*pi)/b:{b})*(a:{a}-x:{x})")
        c1 = 1/n
        part1 = V2*sinh(((n*pi)/b)*x)
        part2 = V1*sinh(((n*pi)/b)*(a-x))
        denomi = sinh(((n*pi)/b)*a)
        part3 = sin(((n*pi)/b)*y)
        general2 += c1*((part1+part2)/denomi)*part3
    return const*general2

def fillMatrix(arr,xu,xd,yl,yr):
    arr[0,:] = xu
    arr[arr.shape[0]-1,:] = xd
    arr[0:,0] = yl
    arr[:,arr.shape[1]-1] = yr
    arr[1,:arr.shape[1]-2]
    #(xu+xd+yl+yr)/4
    arr[1:arr.shape[0]-1,1:arr.shape[1]-1] = 0

def calculate(arr):
    a = cts["a"]
    b = cts["b"]

    x=np.linspace(0,a,arr.shape[0])
    y=np.linspace(0,b,arr.shape[1])

    for i in range(0,arr.shape[0]-1):
        for j in range(0,arr.shape[1]-1):
            arr[i][j] = funcion(x[i],x[j])

def getCalorMap(arr):
    plt.imshow(arr,cmap='hot',interpolation='nearest')
    plt.show()

def analiticalFuncion(nx,ny):
    malla = np.zeros((nx,ny))
    # fillMatrix(malla,xu,0,0,yr)
    calculate(malla)
    getCalorMap(malla)

def main():
```

```
    filas = 50
    cols = 50
    analiticalFuncion(filas ,cols)


if __name__ == "__main__":
    main()
```

3.3 Resultados

Los resultados para la función analítica son: