# Numerical Analysis for PDEs - Report 1
WI4014TU

Shyam Sundar Hemamalini
5071984

28-10-2019

## 1 Proof of the given Theorem

Prove the following Theorem following the steps (a) - (e):

**Theorem 1.** *If a sufficiently smooth u(x, y) has a local maximum at* $(x_0, y_0)$*, then* $\nabla u(x_0, y_0) = 0$ *and the Hessian matrix* $H(x_0, y_0)$ *is negative semi-definite.*

**Definition 1.** *A single-variable function* $f(x)$ *has a local maximum at* $x_0$ *if there exists* $\delta > 0$ *such that* $f(x) \leq f(x_0)$ *for all x,* $|x - x_0| < \delta$*.*

**Definition 2.** *A two-variable function* $u(x, y)$ *has a local maximum at* $(x_0, y_0)$ *if there exists* $\delta > 0$ *such that* $u(x, y) \leq u(x_0, y_0)$ *for all* $(x, y)$*, also given that* $\sqrt{(x - x_0)^2 + (y - y_0)^2} < \delta$*.*

(**a**) Prove for a single-variable function $f(x)$ that, if $f$ has a local maximum at $x_0$ and $f'(x_0)$ exists, then $f'(x_0) = 0$ *(Fermat's Theorem).*

**Solution.** It is given that $f$ is a single-variable function on $x$ and has a local maximum at $x_0$. Then, by definition of a local maximum from *Definition 1*,

$$f(x_0) \geq f(x)$$

where $|x - x_0| \leq \delta$, a small region around $x_0$. Then,

$$f(x_0) - f(x) \geq 0 \tag{1.1}$$

If $f'(x_0)$ exists, then by definition,

$$f'(x_0) = \lim_{x \to x_0^-} \frac{f(x_0) - f(x)}{x_0 - x} \tag{1.2}$$

Substituting (1.1) in (1.2),

$$f'(x_0) \geq 0 \tag{1.3}$$

Similarly, we can express $f'(x_0)$ from the upper bounds of the limit as,

$$f'(x_0) = \lim_{x \to x_0^+} \frac{f(x) - f(x_0)}{x - x_0} \tag{1.4}$$

Substituting (1.1) in (1.4),

$$f'(x_0) \leq 0 \tag{1.5}$$

If $f$ is continuous and differentiable across $\delta$, both (1.3) and (1.5) must satisfy. Hence, at $x = x_0$, we get,

$$f'(x_0) = 0 \tag{1.6}$$

(**b**) Prove that, if a function $f(x)$ has a local maximum at $x_0$ and can be expanded in the Taylor series around $x_0$, then $f''(x_0) \leq 0$ *(Converse of the Second Derivative Test).*

**Solution.** Continuing from subdivision $(a)$, the Taylor series expansion of $f(x_0 + h)$, where $|h| < \delta$, is given by

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \mathcal{O}(h^3) \tag{1.7}$$

Substituting (1.6) in (1.7) and rearranging, we get

$$f(x_0 + h) - f(x_0) = \frac{h^2}{2}f''(x_0) + \mathcal{O}(h^3)$$

From (1.1), we can thus imply

$$\frac{h^2}{2}f''(x_0) \leq 0$$

Since $h$ is non-zero, we can conclude that

$$f''(x_0) \leq 0 \tag{1.8}$$

(**c**) Use Fermat's Theorem to prove that, if a differentiable function $u(x, y)$ has a local maximum at $(x_0, y_0)$, then $\nabla u(x_0, y_0) = 0$. Show that the directional directive $D_{\mathbf{v}}u(x_0, y_0) = 0$ for any $\mathbf{v}$ as well.

**Solution.** Considering the intersection of $u(x, y)$ on a plane $y = y_0$ normal to the y-axis, the function $u$ can be transformed into the intersection function $m$ such that $m$ is a single-variable function of x alone. That is,

$$m(x) = u(x, y_0)$$

Since $m$ is the curve obtained by the intersection of $u$ with a plane parallel to the y-axis,

$$m'(x) = \frac{dm}{dx} = \frac{\partial u}{\partial x} \tag{1.9}$$

Given that $u(x_0, y_0)$ is a local maximum, $m(x_0) = u(x_0, y_0)$ is also a local maximum for $m(x)$. Hence, from Fermat's Theorem,

$$m'(x_0) = 0$$

$$\implies \left.\frac{dm}{dx}\right|_{x=x_0} = 0$$

Hence, from (1.9),

$$\implies \left.\frac{\partial u}{\partial x}\right|_{x=x_0} = 0 \tag{1.10}$$

Similarly, considering a intersection of $u(x, y)$ on a plane $x = x_0$ normal to the x-axis, $u$ can be transformed into another intersection curve $n$ and it can be proved that

$$n'(x_0) = 0$$

and hence,

$$\implies \left.\frac{\partial u}{\partial y}\right|_{y=y_0} = 0 \tag{1.11}$$

From the definition of $\nabla u$,

$$\nabla u = \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix} \tag{1.12}$$

From (1.10), (1.11) and (1.12),

$$\nabla u(x_0, y_0) = \mathbf{0} \tag{1.13}$$

The directional derivative $D_{\mathbf{v}}u(x, y)$ is given by

$$D_{\mathbf{v}}u(x, y) = \nabla u(x, y) \cdot \mathbf{v} \tag{1.14}$$

From (1.13),

$$D_{\mathbf{v}}u(x_0, y_0) = \nabla u(x_0, y_0) \cdot \mathbf{v} = \mathbf{0} \cdot \mathbf{v}$$

$$\implies D_{\mathbf{v}}u(x_0, y_0) = \mathbf{0} \tag{1.15}$$

irrespective of the vector $\mathbf{v}$.

(**d**) Let $H$ be the Hessian matrix and $\mathbf{v} = \langle p, q \rangle$ a unit vector. Show that $\mathbf{v}^T H \mathbf{v}$ equals the second directional derivative of $u$ in the direction of $\mathbf{v}$, i.e. $\mathbf{v}^T H \mathbf{v} = D_{\mathbf{v}}(D_{\mathbf{v}}u)$.

**Solution.** From the definition of a Hessian matrix,

$$H = \begin{bmatrix} \dfrac{\partial^2 u}{\partial x^2} & \dfrac{\partial^2 u}{\partial x \partial y} \\ \dfrac{\partial^2 u}{\partial y \partial x} & \dfrac{\partial^2 u}{\partial y^2} \end{bmatrix}$$

$$\therefore \qquad \mathbf{v}^T H \mathbf{v} = \begin{bmatrix} p & q \end{bmatrix} \begin{bmatrix} \dfrac{\partial^2 u}{\partial x^2} & \dfrac{\partial^2 u}{\partial x \partial y} \\ \dfrac{\partial^2 u}{\partial y \partial x} & \dfrac{\partial^2 u}{\partial y^2} \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix}$$

$$\mathbf{v}^T H\mathbf{v} = \begin{bmatrix} p & q \end{bmatrix} \begin{bmatrix} p\dfrac{\partial^2 u}{\partial x^2} + q\dfrac{\partial^2 u}{\partial x \partial y} \\ p\dfrac{\partial^2 u}{\partial y \partial x} + q\dfrac{\partial^2 u}{\partial y^2} \end{bmatrix}$$

$$\therefore \qquad \mathbf{v}^T H\mathbf{v} = p^2\frac{\partial^2 u}{\partial x^2} + pq\frac{\partial^2 u}{\partial y \partial x} + pq\frac{\partial^2 u}{\partial x \partial y} + q^2\frac{\partial^2 u}{\partial y^2} \tag{1.16}$$

From (1.14),

$$D_{\mathbf{v}}u(x,y) = \nabla u(x,y)\cdot\mathbf{v} = \begin{bmatrix} \dfrac{\partial u}{\partial x} \\ \dfrac{\partial u}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix} = p\frac{\partial u}{\partial x} + q\frac{\partial u}{\partial y}$$

$$D_{\mathbf{v}}(D_{\mathbf{v}}u(x,y)) = \nabla(D_{\mathbf{v}}u(x,y))\cdot\mathbf{v} = \begin{bmatrix} \dfrac{\partial}{\partial x}\left(p\dfrac{\partial u}{\partial x} + q\dfrac{\partial u}{\partial y}\right) \\ \dfrac{\partial}{\partial y}\left(p\dfrac{\partial u}{\partial x} + q\dfrac{\partial u}{\partial y}\right) \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix}$$

$$\therefore \qquad D_{\mathbf{v}}(D_{\mathbf{v}}u(x,y)) = \begin{bmatrix} p\dfrac{\partial^2 u}{\partial x^2} + q\dfrac{\partial^2 u}{\partial x \partial y} \\ p\dfrac{\partial^2 u}{\partial y \partial x} + q\dfrac{\partial^2 u}{\partial y^2} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix}$$

$$\therefore \qquad D_{\mathbf{v}}(D_{\mathbf{v}}u(x,y)) = p^2\frac{\partial^2 u}{\partial x^2} + pq\frac{\partial^2 u}{\partial x \partial y} + pq\frac{\partial^2 u}{\partial y \partial x} + q^2\frac{\partial^2 u}{\partial y^2} \tag{1.17}$$

From (1.16) and (1.17),

$$\mathbf{v}^T H\mathbf{v} = D_{\mathbf{v}}(D_{\mathbf{v}}u(x,y)) \tag{1.18}$$

(**e**) Intersection of any vertical plane going through the point $(x_0, y_0)$ and the graph of $u(x, y)$ is a function of a single variable that has a maximum at $(x_0, y_0)$. To use this fact, parametrize the line in the $(x, y)$-plane going through the point $(x_0, y_0)$ as $x(t) = x_0 + pt$, $y(t) = y_0 + qt$, where $p$ and $q$ are the components of an arbitrary direction vector, and consider the single-variable function $u(x(t), y(t))$, which has a maximum at $t = 0$. Use (b) and (d) to complete the proof of Theorem 1.

**Solution.** Now that $u(x, y)$ has been parametrized as $u(x(t), y(t))$, where $x(t) = x_0+pt$ and $y(t) = y_0+qt$, the gradient of $u$ can be simplified as

$$\nabla u(x(t), y(t)) = \begin{bmatrix} \dfrac{\partial u}{\partial x} & \dfrac{\partial u}{\partial y} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial u}{\partial t}\dfrac{\partial t}{\partial x} & \dfrac{\partial u}{\partial t}\dfrac{\partial t}{\partial y} \end{bmatrix}$$

$$\therefore \qquad \nabla u(x(t), y(t)) = \begin{bmatrix} \dfrac{1}{p}\dfrac{\partial u}{\partial t} & \dfrac{1}{q}\dfrac{\partial u}{\partial t} \end{bmatrix} \tag{1.19}$$

Therefore, the directional derivative can now be expressed as

$$\implies D_{\mathbf{v}}u = \nabla u \cdot \mathbf{v} = \begin{bmatrix} \frac{1}{p}\frac{\partial u}{\partial t} & \frac{1}{q}\frac{\partial u}{\partial t} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix}$$

$$\therefore \qquad D_{\mathbf{v}}u = p \cdot \frac{1}{p}\frac{\partial u}{\partial t} + q \cdot \frac{1}{q}\frac{\partial u}{\partial t}$$

$$\therefore \qquad D_{\mathbf{v}}u = 2\frac{\partial u}{\partial t} \tag{1.20}$$

$$\implies D_{\mathbf{v}}(D_{\mathbf{v}}u) = D_{\mathbf{v}}\left(2\frac{\partial u}{\partial t}\right) = \nabla\left(2\frac{\partial u}{\partial t}\right) \cdot \mathbf{v}$$

$$= 2\begin{bmatrix} \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial t}\right) & \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial t}\right) \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix}$$

$$= 2\begin{bmatrix} \frac{\partial^2 u}{\partial x \partial t} & \frac{\partial^2 u}{\partial x \partial t} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix}$$

$$= 2\begin{bmatrix} \frac{\partial^2 u}{\partial t^2}\frac{\partial t}{\partial x} & \frac{\partial^2 u}{\partial t^2}\frac{\partial t}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix}$$

$$= 2\begin{bmatrix} \frac{1}{p}\frac{\partial^2 u}{\partial t^2} & \frac{1}{q}\frac{\partial^2 u}{\partial t^2} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix}$$

$$= 2\left(p \cdot \frac{1}{p}\frac{\partial^2 u}{\partial t^2} + q \cdot \frac{1}{q}\frac{\partial^2 u}{\partial t^2}\right)$$

$$= 2\left(\frac{\partial^2 u}{\partial t^2} + \frac{\partial^2 u}{\partial t^2}\right)$$

$$\therefore \qquad D_{\mathbf{v}}(D_{\mathbf{v}}u) = 4\,\frac{\partial^2 u}{\partial t^2} \tag{1.21}$$

From (1.18), we get

$$\mathbf{v}^T H \mathbf{v} = 4\,\frac{\partial^2 u}{\partial t^2} \tag{1.22}$$

From (1.8), at the maximum point where $t = 0$,

$$\left.\frac{\partial^2 u}{\partial t^2}\right|_{t=0} \leq 0 \tag{1.23}$$

Hence, from (1.22) and (1.23), at the maximum point $u(x(t=0), y(t=0))$, we get

$$\mathbf{v}^T H \mathbf{v} \leq 0 \tag{1.24}$$

The above statement is the condition for negative semi-definiteness of a matrix and hence, $H$ is negative semi-definite.

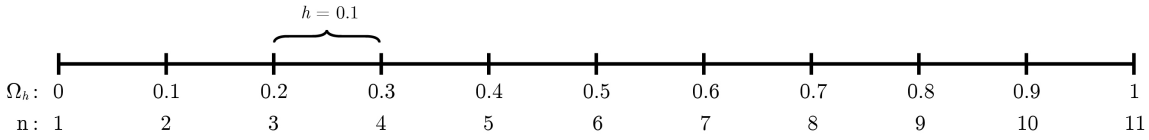
## 2 Properties of the 1D finite-difference Laplacian matrix

*Remark:* The `numpy` and `matplotlib` modules of Python 3 have been imported as:

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

(**a**) Construct on paper the uniform grid $\Omega_h$ on the interval $\Omega = [0, 1]$ with the step size $h = 0.1$. What is the connection between $h$ and the number of grid points, including boundary points?

**Solution.** The grid $\Omega_h$ with the interval divided with the step size of $h = 0.1$ is as shown below:




Including the boundary points in the grid $\Omega_h$, we get 11 grid points, which we can relate to $h$ as

$$n = \frac{1}{h} + 1 \tag{2.1}$$

If the boundary was $l$ instead of 1, the relation would have been

$$n = \frac{l}{h} + 1 \tag{2.2}$$

(**b**) Use `np.linspace()` to construct $\Omega_h$ in Python.

**Solution.** The Python code is as follows:

```
n = 11 # 11 grid points

Omega = np.linspace(0,1,n) # from 0 to 1 with n nodes
print(Omega)
```

On execution, the code gives the following output:

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

(**c**) Discretize on paper the 1D Laplacian operator $\mathcal{L} = -\frac{d^2}{dx^2}$ with zero Dirichlet boundary conditions on your $\Omega_h$ using the central-difference approximation.

**Solution.** As the grid is a uniformly spaced grid, using central difference approximation, the Laplacian operator $\mathcal{L} = -\frac{d^2}{dx^2}$ on $\Omega$ can be discretized as

$$\mathcal{L}(\Omega_i) = \frac{-\Omega_{i+1} + 2\Omega_i - \Omega_{i-1}}{h^2} \tag{2.3}$$

where $i = 2, 3, 4, ..., 10$. If Dirichlet boundary conditions are applied on the end nodes $i = 1$ and $i = 11$, the value on the nodes will be constant and they need not be discretized since they will not be a part of the solution space $\mathbf{u}$.

(**d**) Use `np.diag()` to construct the 1D FD Laplacian matrix L. Use array slicing to print the first, the second and the last rows of L. Use `plt.spy()` to visualize the structure of L with red or green round markers.

**Solution.** The Python code to construct the 1D FD Laplacian matrix L with a solution space of 9 nodes is as follows:

```
n = 11 # number of nodes including boundaries
h = 1 / (n-1)

A = 2.0*np.ones(n-2, dtype=int) # n-2 since first and last nodes are boundaries
B = np.diag(A,0)

P = -1.0*np.ones(n-3, dtype=int) # n-3 since the adjacent diagonals have 1 less element
Q = np.diag(P,1)
R = np.diag(P,-1)

L = (1/h**2)*(B + R + Q)
```

Printing the first, second and the last rows of L can be done using the following code:

```
print("First row: ", L[0]) #first row
print("Second row: ", L[1]) #second row
print("Last row: ", L[n-3]) #last row
```

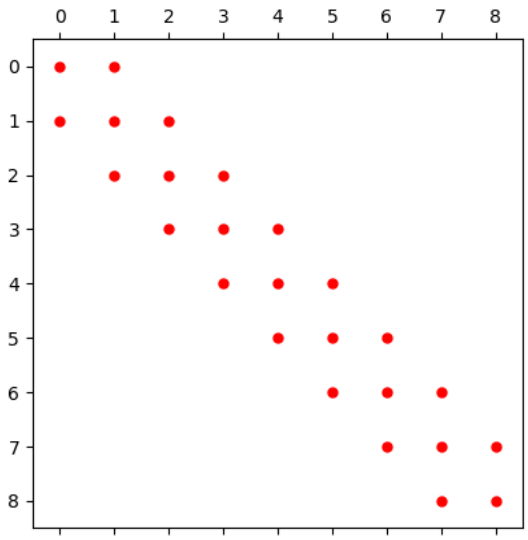
and the output to the above code is as follows:

```
First row: : [ 200 -100   0   0   0   0   0   0   0]
Second row: [-100  200 -100   0   0   0   0   0   0]
Last row: [ 0   0   0   0   0   0   0 -100  200]
```

To plot the spy plot, we use the following command:

```
plt.spy(L, marker=".", color="red") #spy plot
```

and the resulting plot looks as follows:



(e) Compute the first 9 eigenvalues of the operator $\mathcal{L}$ and of the matrix $L$ using their corresponding explicit expressions. Use `np.linalg.eig()` to compute the eigenvalues and eigenvectors of $L$ numerically. Compare the first 9 eigenvalues of $\mathcal{L}$ and $L$ numerically (as a table) and graphically, using `plt.plot()` with $x-$ and $y-$axes representing, respectively, the real and the imaginary parts of the eigenvalues.

**Solution.** The continuous eigenvalues of the operator $\mathcal{L}$ are given by the expression:

$$\tilde{\lambda}_i = \left(\frac{\pi i}{D}\right)^2, \quad i = 1, 2, 3, ... \tag{2.4}$$

The eigenvalues of the discretized Laplacian matrix $L$ can be numerically determined using the following code:

```
eigval, eigvec = np.linalg.eig(L) #calculating eigenvalues and eigenvectors
eigval.sort() #sorting the eigenvalues from lowest to highest
```

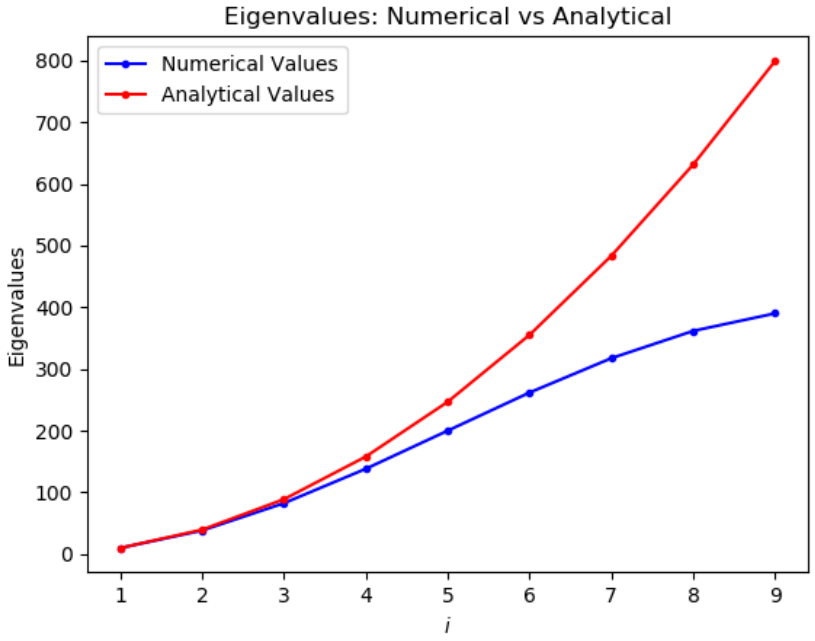
The comparison of the first 9 eigenvalues of $\mathcal{L}$ and $L$ is as follows:

| i | Analytical Eigenvalues $\tilde{\lambda}_i$ | Numerical Eigenvalues $\lambda_i$ |
|---|---|---|
| 1 | 9.8696044 | 9.78869674 |
| 2 | 39.4784176 | 38.19660113 |
| 3 | 88.82643961 | 82.44294954 |
| 4 | 157.91367042 | 138.19660113 |
| 5 | 246.74011003 | 200 |
| 6 | 355.30575844 | 261.80339887 |
| 7 | 483.61061565 | 317.55705046 |
| 8 | 631.65468167 | 361.80339887 |
| 9 | 799.43795649 | 390.21130326 |

As all the eigenvalues are real, a comparison of the eigenvalues in a real vs imaginary plot is not presented. The numerical and the analytical eigenvalues are plotted in a eigenvalues vs i plot using the following code:

```
X = np.linspace(0,1,n)
Y = np.linspace(0,1,200)

I = np.linspace(1,n-2,n-2)
Eigval = (np.pi*I)**2 #generating the continuous eigenvalues of operator L

plt.figure()
plt.plot(I,eigval,marker='o',label='Numerical Values')
plt.plot(I,Eigval,marker='o',label='Analytical Values')
plt.title("Eigenvalues: Numerical vs Analytical")
plt.xlabel("$x$")
plt.legend(loc='best')
```

The output for the above code is as follows:




(**f**) Use `plt.plot()` to demonstrate that the numerically computed eigenvectors of $L$ are the sampled eigenfucntions of $\mathcal{L}$. Use `np.insert()` to insert the boundary values into the eigenvectors and a finer grid to plot eigenfunctions.

**Solution.** Inserting the boundary values into the computed eigenvectors can be done using the following code:

```
zeros = np.zeros(n-2)

eigvec = np.insert(eigvec,0, zeros,0) #inserting boundaries to eigenvectors
eigvec = np.insert(eigvec,n-1,zeros,0)
```

The continuous eigenfunctions of the operator $\mathcal{L}$ are given by the expression:

$$\tilde{v_i}(x) = sin\left(\frac{\pi i x}{D}\right), \quad i = 1, 2... \tag{2.5}$$

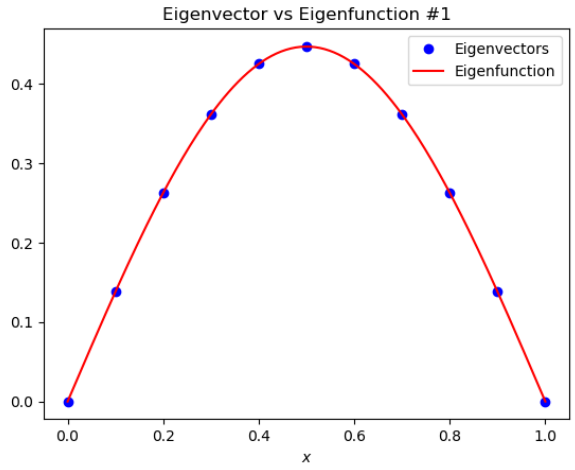
Here, the domain length $D$ is 1. Also, on normalizing the eigenfunction, the normalizing factor was found to be $\sqrt{2h}$. Hence, the normalized eigenfunction is as follows:

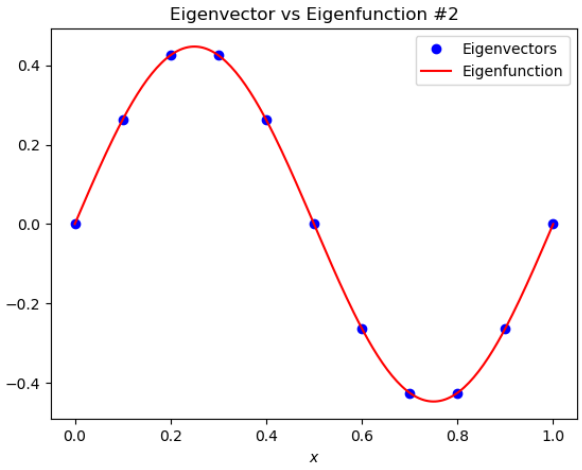
$$\tilde{v_i}(x) = \sqrt{2h}\, sin\left(\pi i x\right), \quad i = 1, 2... \tag{2.6}$$

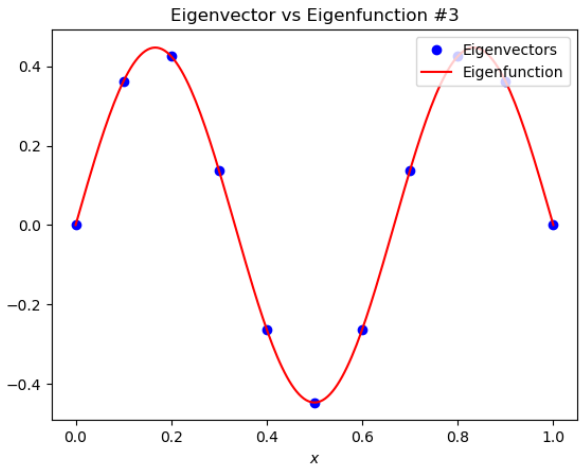
The first 9 sampled eigenfunctions are then calculated for a grid of $n = 200$ and plotted against the 9 eigenvectors using the following code:

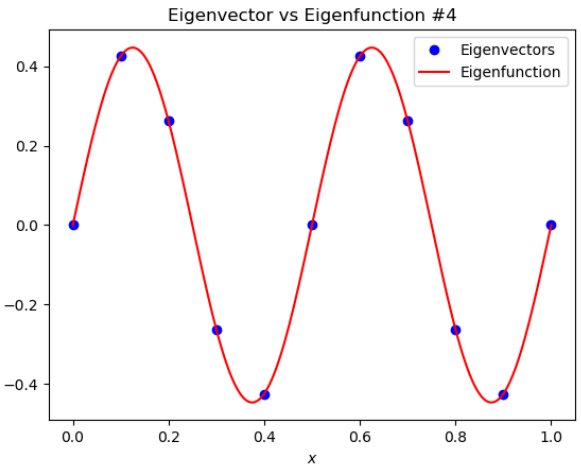
```
X = np.linspace(0,1,n)
Y = np.linspace(0,1,200)

for i in range(0,9):

    Eigfunc = ((2*h)**0.5)*np.sin(np.pi*Y*(i+1)) #generating the (i+1)th eigenfunction

    plt.figure()
    plt.plot(X,eigvec[:,i],'bo-',label='Eigenvectors')
    plt.plot(Y,Eigfunc,'r',label='Eigenfunction')
    plt.title("Eigenvector vs Eigenfunction #%i" %(i+1))
    plt.xlabel("$x$")
    plt.legend(loc='upper right')
```

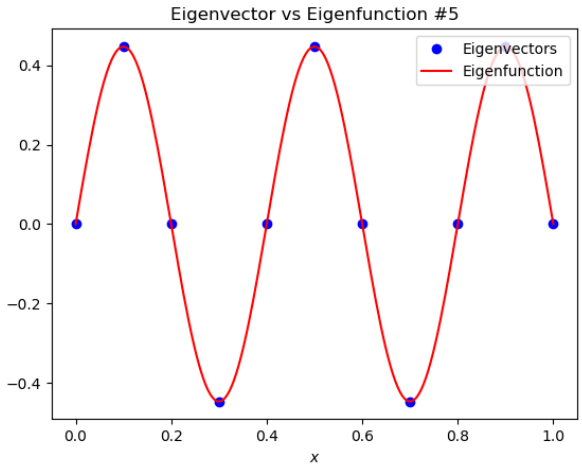
The plots of the 9 normalized eigenvectors compared with the corresponding eigenfunctions generated by the above code is as below:

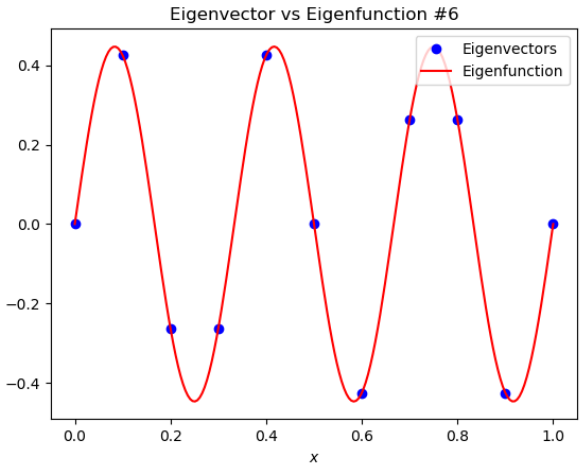


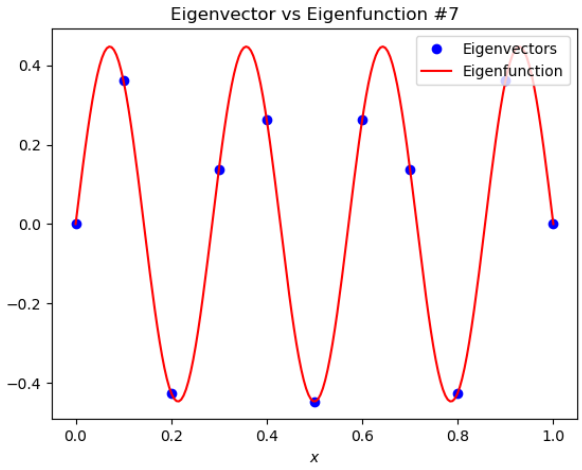



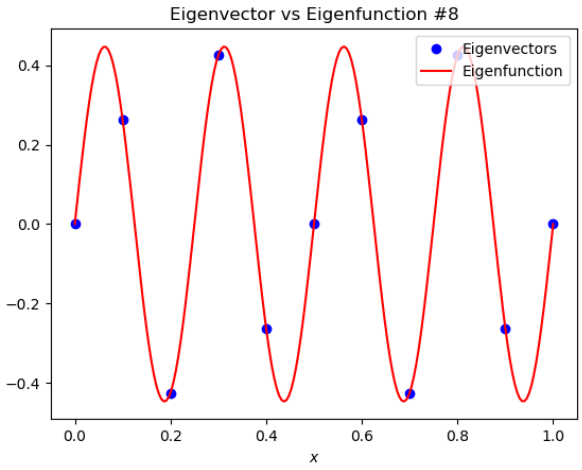



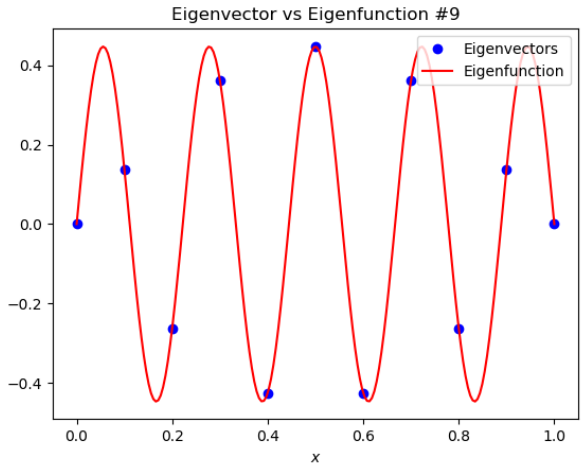






From the plots, it can be seen that the eigenvectors match exactly with the corresponding eigenfunctions. Hence, it can be proved that the eigenvectors of the discretised Laplacian matrix $L$ are the corresponding sampled eigenfunctions of the Laplacian operator $\mathcal{L}$.

# 3 Solution of the 1D Poisson Equation

Consider the following boundary-value problem:

$$\begin{aligned} -\frac{d^2u}{dx^2} &= f_i, \quad x \in (0,1); \\ u(0) &= 1, \quad u(1) = 2; \end{aligned} \tag{3.1}$$

with

$$f_1(x) = 1, \quad f_2(x) = e^x, \quad x \in [0,1]. \tag{3.2}$$

(**a**) Find the exact solutions $u_1(x)$ and $u_2(x)$ of (3.1) corresponding to $f_1(x)$ and $f_2(x)$ given in (3.2).

**Solution.** Considering the first source function $f_1(x) = 1$, the 1D Poisson equation is

$$-\frac{d^2u_1}{dx^2} = 1 \tag{3.3}$$

$$\frac{d^2u_1}{dx^2} = -1$$

Integrating twice, we get

$$\iint d^2u_1 = \iint -dx^2$$

$$\int du_1 = \int -x + c_1 \, dx$$

$$u_1 = \frac{-x^2}{2} + c_1x + c_2 \tag{3.4}$$

Substituting the values given in (3.1) for the boundaries $x = 0$ and $x = 1$, we get the exact solution as

$$u_1 = \frac{-x^2}{2} + \frac{3x}{2} + 1 \tag{3.5}$$

Similarly, for the second source function $f_2(x) = e^x$, we get

$$-\frac{d^2u_2}{dx^2} = e^x \tag{3.6}$$

$$\iint d^2u_2 = \iint -e^x \, dx^2$$

$$\therefore \quad u_2 = -e^x + c_1x + c_2 \tag{3.7}$$

Applying the boundary values given in (3.1), we get the exact solution as

$$u_2 = -e^x + ex + 2 \tag{3.8}$$

(**b**) Discretize the problem (3.1) on a uniform grid with the step size $h = 0.2$ using the FDM. Derive the corresponding linear algebraic problem $L\mathbf{u} = \mathbf{f}$. Print the entries of $L$ and $\mathbf{f}$ for the two RHS functions (3.2).

**Solution.** Given the length of domain $l = 1$ and $h = 0.2$, the number of nodes for discretisation is determined from the relation

$$n = \frac{l}{h} + 1 = \frac{1}{0.2} + 1$$

$$\therefore \quad n = 6 \tag{3.9}$$

As the grid is assumed to be uniform, using the central difference approximation for the second derivative transforms (3.1) as

$$-\left(\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}\right) = f_i, \quad i = 2, 3, ..., n-1 \tag{3.10}$$

$$\implies \quad \frac{2u_i - u_{i+1} - u_{i-1}}{h^2} = f_i, \quad i = 2, 3, ..., n-1 \tag{3.11}$$

Since $i = 1$ and $i = n$ denote the boundary points, they do not form a part of the solution space. Hence, the solution matrix $\mathbf{u}$ is given by

$$\mathbf{u} = \begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_{n-1} \end{bmatrix}_{(n-2)} \tag{3.12}$$

The coefficient matrix for $\mathbf{u}$ is the Laplacian matrix $L$ given by

$$L = \begin{bmatrix} 2 & -1 & \dots & 0 & 0 \\ -1 & 2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 2 & -1 \\ 0 & 0 & \dots & -1 & 2 \end{bmatrix}_{(n-2)\times(n-2)} \tag{3.13}$$

The RHS matrix $\mathbf{f}$ is given by

$$\mathbf{f} = \begin{bmatrix} f_2 + {}^{u_1}/_{h^2} \\ f_3 \\ \vdots \\ f_{n-1} + {}^{u_n}/_{h^2} \end{bmatrix}_{(n-2)} \tag{3.14}$$

Note that the first and last entries in the RHS matrix $\mathbf{f}$ has the boundary values since they are not part of the coefficient matrix and are constants, given the Dirichlet boundary condition.

Hence, the linear algebraic problem $L\mathbf{u} = \mathbf{f}$ is given by

$$\begin{bmatrix} 2 & -1 & \dots & 0 \\ -1 & 2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 2 \end{bmatrix} \cdot \begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} f_2 + {}^{u_1}/_{h^2} \\ f_3 \\ \vdots \\ f_{n-1} + {}^{u_n}/_{h^2} \end{bmatrix} \tag{3.15}$$

The coefficient matrix $L$ and the RHS matrix $\mathbf{f}$ for both the functions given in (3.2) can be coded as below.

```
h = 0.2
n = int(1/h + 1) #defining the number of grid points

#defining the coefficient matrix
L = np.diag(2*np.ones(n-2)) + np.diag(-1*np.ones(n-3),1) + np.diag(-1*np.ones(n-3),-1)
L = L / h**2
print(L)

bc = 1/h**2

#defining the RHS matrix for function 1
f1 = np.ones(n-4)
f1 = np.insert(f1,0,1+bc) #inserting boundary values
f1 = np.insert(f1,n-3,1+2*bc)

print(f1)

#defining the RHS matrix for function 2
f2 = np.ones(n-4)

for i in range(n-4):
    f2[i] = np.exp(h*(i+2))

f2 = np.insert(f2,0,(np.exp(h)+bc)) #inserting boundary values
f2 = np.insert(f2,n-3,(np.exp(1-h)+2*bc))

print(f2)
```

The output for the above code is

```
L:
[[ 50. -25.   0.   0.]
 [-25.  50. -25.   0.]
 [  0. -25.  50. -25.]
 [  0.   0. -25.  50.]]
f1:  [26.  1.  1. 51.]
f2:  [26.22140276  1.4918247   1.8221188  52.22554093]
```

(**c**) Use `np.linalg.solve()` to find the numerical solutions $\mathbf{u}_1$ and $\mathbf{u}_2$ of the two problems. Compare the exact and numerical solutions on the grid, including the boundary points, both graphically(plot) and numerically(table). Use `np.linalg.norm()` to compute the global errors. Explain your results.

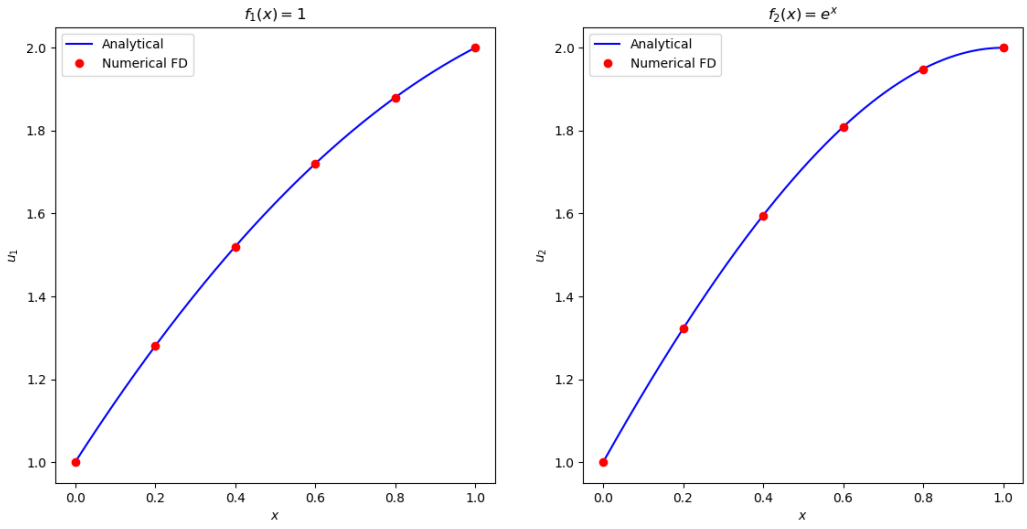
**Solution.** Solving the system of equations from (3.15) numerically and determining the exact solution can be done using `np.linalg.solve()` as shown below.

```
u1 = np.linalg.solve(L,f1) #solving for RHS matrix f1
u1 = np.insert(u1,0,1) #inserting boundary values in solution
u1 = np.insert(u1,n-1,2)

u2 = np.linalg.solve(L,f2) #solving for RHS matrix f2
u2 = np.insert(u2,0,1) #inserting boundary values in solution
u2 = np.insert(u2,n-1,2)

X = np.linspace(0,1,n) #defining the grid

U1 = -((X**2)/2) + 3*X/2 + 1 #exact solution for 1st case
U2 = -1*np.exp(X) + np.e*X + 2 #exact solution for 2nd case
```

The numerical solution is then compared with the exact solution and the values for both across every grid point is as shown in the table below.

| x | Analytical Solution $\tilde{\mathbf{u}_1}$ | Numerical Solution $\mathbf{u}_1$ | Analytical Solution $\tilde{\mathbf{u}_2}$ | Numerical Solution $\mathbf{u}_2$ |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 0.2 | 1.28 | 1.28 | 1.32225361 | 1.32184691 |
| 0.4 | 1.52 | 1.52 | 1.59548803 | 1.59483771 |
| 0.6 | 1.72 | 1.72 | 1.8088503 | 1.80815552 |
| 0.8 | 1.88 | 1.88 | 1.94908453 | 1.94858858 |
| 1 | 2 | 2 | 2 | 2 |

A graphical comparison of the above set of values is as shown below.



The global error of the two functions can be evaluated using `np.linalg.norm` as shown in the code below.

```
error1 = np.linalg.norm(U1-u1) #determining global error for f1
print("Norm of u1 = ",error1)

error2 = np.linalg.norm(U2-u2) #determining global error for f2
print("Norm of u2 = ",error2)
```

The output for the above code is as follows.

```
Norm of u1 =  3.1401849173675503e-16
Norm of u2 =  0.001147612812602845
```

The global error of $f_1$ is of very small order of magnitude because, considering the exact solution of $f_1$, from (3.5),

$$u_1 = \frac{-x^2}{2} + \frac{3x}{2} + 1$$

The Taylor expansion for the above equation is

$$u_1(x+h) = u_1 + h \cdot u_1' + \frac{h^2}{2} \cdot u_1'' + \frac{h^3}{6} \cdot u_1''' + \mathcal{O}(h^4) \tag{3.16}$$

Differentiating $u_1$, it can be derived that

$$\frac{d^2 u_1}{dx^2} = -1 \tag{3.17}$$

Thus, it can be inferred that the third and the higher order derivatives of $u_1$ are zero. Hence, the Taylor expansion reduces to the discretised equation of order $\mathcal{O}(h^2)$ and the discretised equation pertains to the exact solution.

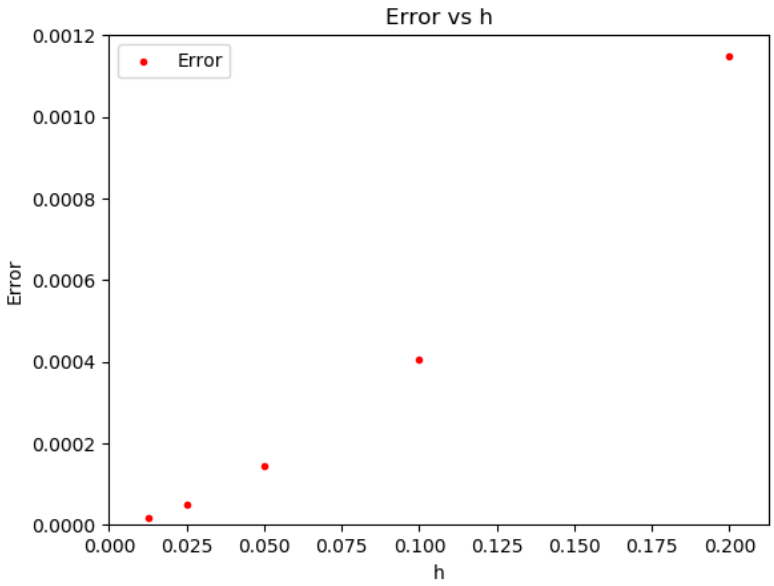
However, for $f_2$, it can be seen that the errors are of order $\mathcal{O}(h^3)$, which can be evaluated to $h^3 = 0.008$.

(**d**) Refine the uniform grid by changing the grid step as $h^k = 0.2/2^k$, where $k = 0, 1, 2, 3, 4$, and recomputing the numerical solution $\mathbf{u}_2$ for each $h_k$. Study the behavior of the global error with respect to $h$. Explain your results. Bonus: use `scipy.optimize.curve_fit()` to fit the global error data with the curve $Ch^\alpha$, and determine the convergence order $\alpha$ as well as the constant $C$.

**Solution.** Refining the grid according to the relation $h^k = 0.2/2^k$, where $k = 0, 1, 2, 3, 4$, and recomputing the numerical solution $\mathbf{u}_2$ for each $h_k$, we get the following results for the global error.

| k | h | Global error $L_2\ norm(\mathbf{u}_2)$ |
|---|---|---|
| 0 | 0.2 | 1.147613e-03 |
| 1 | 0.1 | 4.067147e-04 |
| 2 | 0.05 | 1.438574e-04 |
| 3 | 0.025 | 5.086621e-05 |
| 4 | 0.0125 | 1.798434e-05 |

The global errors are plotted and the variation is as shown below.




From the values of global error obtained, it can be seen that the global error is decreasing with decreasing h.

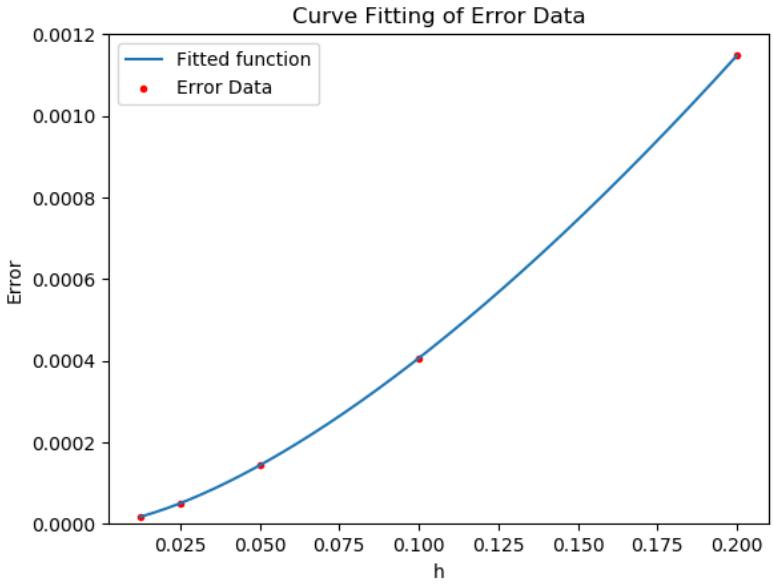
Assuming that the global error convergence is of the form $Ch^{\alpha}$, the convergence $\alpha$ and the constant $C$ can be determined using `scipy.optimize.curve_fit()` as shown in the code below.

```
from scipy import optimize

K = np.linspace(0,4,5)

H = 0.2 / 2**(K)

y_data = error
x_data = H

def test_func(x, c, alpha):
    return c * (x**alpha)

parameters, covariance = optimize.curve_fit(test_func, x_data, y_data, p0=[0.01,1.5])

print("C, Alpha: ", parameters)
```

The output to the above code is as below.

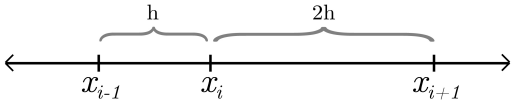
```
C, Alpha:  [0.01277392 1.49722444]
```

Plotting the fitted function with the error data, we get the following plot.




(**e**) Repeat calculation (b) - (d) on a non-uniform grid obtained by joining any two neighbouring intervals of the uniform grid. Explain your results.

**Solution.** Assuming two neighbouring intervals are joined at position $x_i$, the spacing at that grid position will be as shown in the diagram below.




From the Taylor series expansion at $u_{i-1}$, we get

$$u_{i-1} = u_i + (-h)u_i' + \frac{(-h)^2}{2}u_i'' + \mathcal{O}(h^3) \tag{3.18}$$

Similarly, the Taylor series expansion at $u_{i+1}$ is

$$u_{i+1} = u_i + (2h)u_i' + \frac{(2h)^2}{2}u_i'' + \mathcal{O}(h^3) \tag{3.19}$$

Manipulating (3.18) and (3.19), we get

$$u_{i+1} + 2u_{i-1} = 3u_i + 3(h)^2 u_i'' + \mathcal{O}(h^3) \tag{3.20}$$

$$\therefore \quad u_i'' = \frac{u_{i+1} - 3u_i + 2u_{i-1}}{3h^2} + \mathcal{O}(h) \tag{3.21}$$

Substituting in the Poisson equation, we get

$$\frac{1}{h^2}\left(u_i - \frac{u_{i+1}}{3} - \frac{2u_{i-1}}{3}\right) = f_i \tag{3.22}$$

Similarly, the discretised Poisson equation corresponding to $u_{i+1}$ will be

$$\frac{1}{h^2}\left(u_{i+1} - \frac{u_i}{3} - \frac{2u_{i+2}}{3}\right) = f_{i+1} \tag{3.23}$$

Hence, the coefficient matrix $L$ varies as below.

$$\frac{1}{h^2}\begin{bmatrix} \ddots & \ddots & \ddots & & & & & \\ \dots & -1 & 2 & -1 & \dots & & & \\ & \dots & -2/3 & 1 & -1/3 & \dots & & \\ & & \dots & -1/3 & 1 & -2/3 & \dots & \\ & & & & -1 & 2 & -1 & \dots \\ & & & & & \ddots & \ddots & \ddots \end{bmatrix} \cdot \begin{bmatrix} \vdots \\ u_{i-1} \\ u_i \\ u_{i+1} \\ u_{i+2} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ f_{i-1} \\ f_i \\ f_{i+1} \\ f_{i+2} \\ \vdots \end{bmatrix} \tag{3.24}$$

A special case is when the non-uniformity in the grid occurs at one of the boundaries. In such a case, the RHS matrix $\mathbf{f}$ varies. The boundary values which are added to $\mathbf{f}$, see (3.14), are also changed according to the equations (3.22) and (3.23).

Considering $h = 0.2$, there are 4 inner grid points. Hence, there can be 4 different ways to join neighbouring intervals. Hence, there can be 4 different coefficient matrices depending on the skipping position. The coefficient matrix and the RHS matrix for both cases in (2) for every possible skipping position of a grid point is as below.

```
Skipping position: 1
L:
 [[ 25.         -16.66666667   0.        ]
 [-25.          50.         -25.        ]
 [  0.         -25.          50.        ]]
f1:  [ 9.33333333  1.         51.        ]
f2:  [ 9.82515803  1.8221188  52.22554093]

Skipping position: 2
L:
 [[ 25.          -8.33333333   0.        ]
 [ -8.33333333  25.         -16.66666667]
 [  0.         -25.          50.        ]]
f1:  [17.66666667  1.         51.        ]
f2:  [17.88806942  1.8221188  52.22554093]

Skipping position: 3
L:
 [[ 50.         -25.           0.        ]
 [-16.66666667  25.          -8.33333333]
 [  0.          -8.33333333  25.        ]]
f1:  [26.          1.         34.33333333]
f2:  [26.22140276  1.4918247  35.55887426]
```

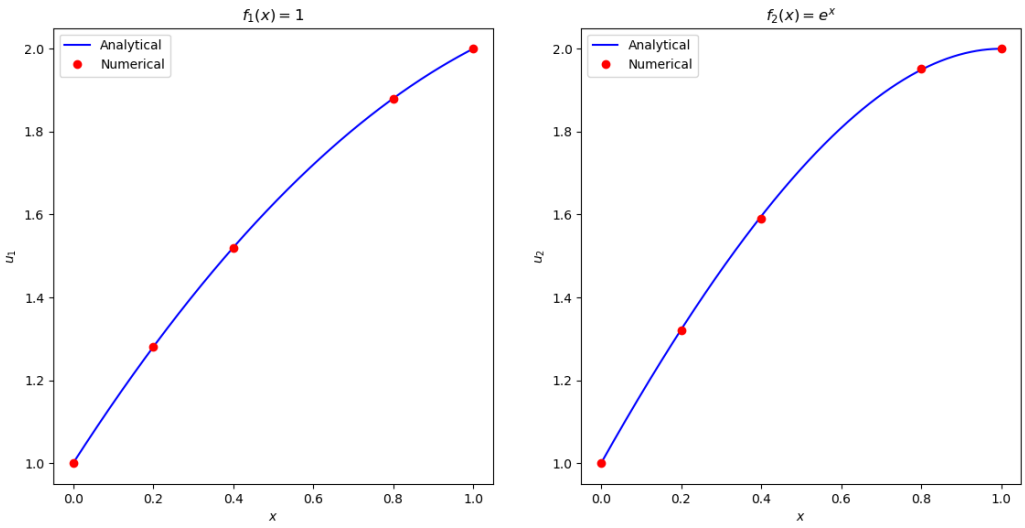
```
Skipping position: 4
L:
 [[ 50.         -25.           0.        ]
 [-25.          50.         -25.        ]
 [  0.         -16.66666667  25.        ]]
f1:  [26.          1.         17.66666667]
f2:  [26.22140276  1.4918247  18.48878547]
```

The system of equations are solved using `np.linalg.solve` and the solution matrix **u** obtained is as found in the tables below. In the tables, $x^*$ denotes the skipped grid point.

| $x$ | Exact Solution | Numerical Solution for $f_1$ | | | |
|---|---|---|---|---|---|
| | | $x^* = 0.2$ | $x^* = 0.4$ | $x^* = 0.6$ | $x^* = 0.8$ |
| 0.2 | 1.28 | | 1.28 | 1.28 | 1.28 |
| 0.4 | 1.52 | 1.52 | | 1.52 | 1.52 |
| 0.6 | 1.72 | 1.72 | 1.72 | | 1.72 |
| 0.8 | 1.88 | 1.88 | 1.88 | 1.88 | |

| $x$ | Exact Solution | Numerical Solution for $f_2$ | | | |
|---|---|---|---|---|---|
| | | $x^* = 0.2$ | $x^* = 0.4$ | $x^* = 0.6$ | $x^* = 0.8$ |
| 0.2 | 1.322 | | 1.320 | 1.3195 | 1.3186 |
| 0.4 | 1.595 | 1.6013 | | 1.5901 | 1.5884 |
| 0.6 | 1.809 | 1.8125 | 1.8139 | | 1.7984 |
| 0.8 | 1.949 | 1.9507 | 1.9515 | 1.9524 | |

A graphical representation of the solution for $x^* = 0.6$ is shown in the plot below.



```
#X -> grid coordinates
#position -> skipping position

u1 = np.linalg.solve(L,f1) #solving for RHS matrix f1

u1 = np.insert(u1,0,1) #inserting boundary values
u1 = np.insert(u1,n-1,2)

U1e = -((X**2)/2) + 3*X/2 + 1 #exact solution for f1
error1 = np.linalg.norm(u1-U1e) #error norm for f1

u2 = np.linalg.solve(L,f2)

u2 = np.insert(u2,0,1)
u2 = np.insert(u2,n-1,2)

U2e = -1*np.exp(X) + np.e*X + 2 #exact solution for f2
error2 = np.linalg.norm(u2-U2e) #error norm for f2
```

The global error for each case is determined using `np.linalg.norm` and can be found in the table below. It can be easily noted that the global error is more or less independent of the skipping position.

| $x^*$ | Global error $L_2\ norm(\mathbf{u}_1)$ | Global error $L_2\ norm(\mathbf{u}_2)$ |
|---|---|---|
| 0.2 | 3.1402e-16 | 0.00707 |
| 0.4 | 0 | 0.00598 |
| 0.6 | 3.846e-16 | 0.00687167 |
| 0.8 | 3.1402e-16 | 0.01309079 |

The variation of the global error with decreasing grid spacing can be determined the same way as (**d**). The following table shows the variation of the global error with grid spacing $h$ and $x^*$ for different skipping positions.

| $h$ | Error norm $L_2\ norm(\mathbf{u}_2)$ | | |
|---|---|---|---|
| | $x^* = h$ | $x^* = L/2$ | $x^* = L - h$ |
| 0.2 | 7.076e-3 | 6.872e-3 | 1.309e-2 |
| 0.1 | 1.314e-3 | 1.259e-3 | 3.719e-3 |
| 0.05 | 2.048e-4 | 2.51e-4 | 8.54e-4 |
| 0.025 | 3.381e-5 | 5.76e-5 | 1.864e-4 |
| 0.0125 | 1.189e-5 | 1.736e-5 | 4.237e-5 |

The convergence $\alpha$ for each case is also determined and is as follows.

```
x* = 1, Alpha =  2.4438373444298795
x* = n/2, Alpha =  2.4382798405057167
x* = n-1, Alpha =  1.857384400030101
```

# 4 2D Poisson Equation with FDM

Consider the following boundary value problem:

$$\begin{aligned}
&-\Delta u = f, \quad (x,y) \in \Omega(0,2)\times(0,1),\\
&u(x,y) = sin(2\pi y), \quad x = 0, y \in (0,1)\\
&u(x,y) = sin(2\pi y), \quad x = 2, y \in (0,1)\\
&u(x,y) = sin(0.5\pi x), \quad x \in (0,2), y = 0\\
&u(x,y) = 0, \quad x \in (0,2), y = 1\\
&f(x,y) = 20sin(\pi y)sin(1.5\pi x + \pi), \quad (x,y) \in \overline{\Omega}
\end{aligned} \tag{4.1}$$

where $\overline{\Omega} = [0,2]\times[0,1]$ is the rectangle with corners (0,0), (2,0), (2,1) and (0,1).

For this problem, we assume the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse as sp
import scipy.sparse.linalg as la
```

The goal is to make a compact code optimized for memory and speed. It can be re-used in subsequent assignments.

(**a**) Derive (on paper) the FD discretization of the problem on a doubly-uniform grid with step $h$.

**Solution.** Since $u$ is a function of two variables, the Laplacian operator in 2D, $\Delta$, can be expressed and discretised as below.

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{4.2}$$

$$\therefore \quad -\Delta u = -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{4.3}$$

Therefore, assuming the grid is uniform along $x-$ and $y-$axes with a grid spacing $h$, the given differential equation $-\Delta u = f$ can be discretised using central difference approximation as below.

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f$$

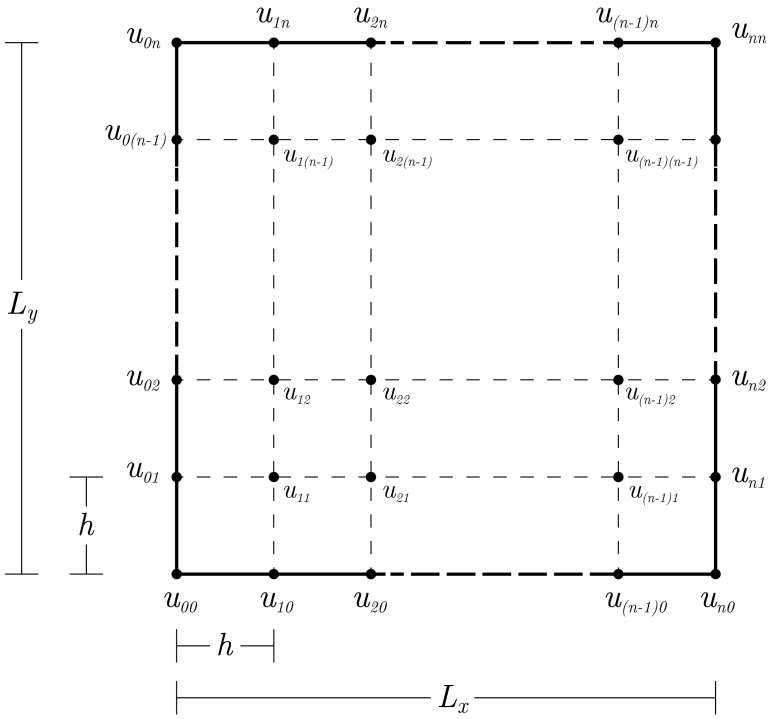
$$\implies \quad -\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}\right) = f_{i,j}$$

$$\therefore \quad \frac{4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}}{h^2} = f_{i,j} \tag{4.4}$$

The above equation (4.4) is the FD discretisation of a Laplacian in 2D doubly uniform grid of spacing $h$.

(**b**) Show (on paper) that on the lexicographic grid, the 2D FD Laplacian matrix $L$ can be obtained as $L = I_y \otimes L_{xx} + L_{yy} \otimes I_x$, where $L_{xx}$ and $L_{yy}$ are matrices of 1D FD Laplacians in $x$- and $y$-direction, respectively, and $I_x$, $I_y$ are identity matrices of proper size. Also show that $L_{xx} = D_x^T D_x$ and $L_{yy} = D_y^T D_y$, where $D_x$ and $D_y$ are matrices representing one-sided FD approximations of the first derivatives.

**Solution.** A typical 2D doubly-uniform grid with $xy$ naming scheme is as below.




As the boundaries have a fixed value, as given by the Dirichlet boundary conditions in (4.1), the solution space will only consist of the following nodal points.

$$\mathbf{u} = u_{ij}, \quad i \in [1, n_x - 1],\ j \in [1, n_y - 1]$$

Arranging the solution space $\mathbf{u}$ in lexicographic order, we will arrive at the following 1D solution matrix $\mathbf{u}$.

$$\mathbf{u} = \begin{bmatrix} u_{11} \\ u_{21} \\ \vdots \\ u_{(n_x-1)1} \\ u_{21} \\ u_{22} \\ \vdots \\ u_{(n_x-1)2} \\ \vdots \\ u_{1(n-1)} \\ u_{2(n-1)} \\ u_{(n_x-1)(n_y-1)} \end{bmatrix}_{(n_x-1)\times(n_y-1)} \tag{4.5}$$

From (4.4), we can write the discretised Poisson equation for some of the grid points as below.

$$\begin{aligned}
4u_{11} - u_{21} - u_{01} - u_{12} - u_{10} &= h^2 \cdot f_{11} \\
4u_{21} - u_{31} - u_{11} - u_{22} - u_{20} &= h^2 \cdot f_{21} \\
4u_{12} - u_{22} - u_{02} - u_{13} - u_{11} &= h^2 \cdot f_{12} \\
4u_{(n-1)(n-1)} - u_{n(n-1)} - u_{(n-2)(n-1)} - u_{(n-1)n} - u_{(n-1)(n-2)} &= h^2 \cdot f_{(n-1)(n-1)}
\end{aligned}$$

The solution space will contain $(n_x-1)\cdot(n_y-1)$ number of variables since the boundaries are not part of the solution space. Hence, the coefficient matrix will be a square matrix of size $(n_x-1)\cdot(n_y-1)\times(n_x-1)\cdot(n_y-1)$.

The 2D Laplacian matrix $L$ can be expressed in matrix form as below.

$$L = \frac{1}{h^2}\begin{bmatrix}
4 & -1 & . & -1 & . & . & . & . & . \\
-1 & 4 & -1 & . & -1 & . & . & . & . \\
. & -1 & 4 & . & . & -1 & . & . & . \\
-1 & . & . & 4 & -1 & . & -1 & . & . \\
. & -1 & . & -1 & 4 & -1 & . & -1 & . \\
. & . & -1 & . & -1 & 4 & . & . & -1 \\
. & . & . & -1 & . & . & 4 & -1 & . \\
. & . & . & . & -1 & . & -1 & 4 & -1 \\
. & . & . & . & . & -1 & . & -1 & 4
\end{bmatrix}_{(n_x-1)\cdot(n_y-1)\times(n_x-1)\cdot(n_y-1)} \tag{4.6}$$

We know that the 1D Laplacian matrix $L_{xx}$ can be represented as:

$$L_{xx} = \frac{1}{h^2}\begin{bmatrix}
2 & -1 & . & . & . & . & . & . \\
-1 & 2 & -1 & . & . & . & . & . \\
. & -1 & 2 & -1 & . & . & . & . \\
. & . & -1 & 2 & -1 & . & . & . \\
. & . & . & -1 & 2 & -1 & . & . \\
 & & & & \ddots & \ddots & \ddots & \\
. & . & . & . & . & -1 & 2 & -1 \\
. & . & . & . & . & . & -1 & 2
\end{bmatrix}_{(n_x-1)\times(n_x-1)} \tag{4.7}$$

Hence, the Kronecker product of $I_y$ and $L_{xx}$ is given by:

$$I_y \otimes L_{xx} = \begin{bmatrix}
1 & . & . & . & . & . \\
. & 1 & . & . & . & . \\
. & . & 1 & . & . & . \\
 & & & \ddots & & \\
. & . & . & . & 1 & . \\
. & . & . & . & . & 1
\end{bmatrix} \otimes L_{xx}$$

$$\implies \quad I_y \otimes L_{xx} = \begin{bmatrix} L_{xx} & . & . & . & . & . \\ . & L_{xx} & . & . & . & . \\ . & . & L_{xx} & . & . & . \\ & & & \ddots & & \\ . & . & . & . & L_{xx} & . \\ . & . & . & . & . & L_{xx} \end{bmatrix}_{(n_y-1)\times(n_y-1)} \tag{4.8}$$

Also, the Kronecker product of $L_{yy}$ and $I_x$ is given by

$$L_{yy} \otimes I_x = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & . & . & . & . & . & . \\ -1 & 2 & -1 & . & . & . & . & . \\ . & -1 & 2 & -1 & . & . & . & . \\ . & . & -1 & 2 & -1 & . & . & . \\ . & . & . & -1 & 2 & -1 & . & . \\ & & & & \ddots & \ddots & \ddots & \\ . & . & . & . & . & -1 & 2 & -1 \\ . & . & . & . & . & . & -1 & 2 \end{bmatrix} \otimes I_x$$

$$\implies \quad L_{yy} \otimes I_x = \frac{1}{h^2} \begin{bmatrix} 2I_x & -I_x & . & . & . & . & . & . \\ -I_x & 2I_x & -I_x & . & . & . & . & . \\ . & -I_x & 2I_x & -I_x & . & . & . & . \\ . & . & -I_x & 2I_x & -I_x & . & . & . \\ . & . & . & -I_x & 2I_x & -I_x & . & . \\ & & & & \ddots & \ddots & \ddots & \\ . & . & . & . & . & -I_x & 2I_x & -I_x \\ . & . & . & . & . & . & -I_x & 2I_x \end{bmatrix}_{(n_y-1)\times(n_y-1)} \tag{4.9}$$

From (4.8) and (4.9), we get the Kronecker sum as

$$I_y \otimes L_{xx} + L_{yy} \otimes I_x = \begin{bmatrix} L_{xx} + \frac{2}{h^2} I_x & -\frac{1}{h^2} I_x & . & . & . \\ -\frac{1}{h^2} I_x & L_{xx} + \frac{2}{h^2} I_x & -\frac{1}{h^2} I_x & . & . \\ & \ddots & \ddots & \ddots & \\ . & . & -\frac{1}{h^2} I_x & L_{xx} + \frac{2}{h^2} I_x & -\frac{1}{h^2} I_x \\ . & . & . & -\frac{1}{h^2} I_x & L_{xx} + \frac{2}{h^2} I_x \end{bmatrix}_{(n_y-1)\times(n_y-1)} \tag{4.10}$$

The above matrix has $L_{xx} + 2I_x$ as the main diagonal element and $I_x$ as the adjacent diagonal element. Evaluating $L_{xx} + 2I_x$, we get:

$$L_{xx} + \frac{2}{h^2} I_x = \frac{1}{h^2} \begin{bmatrix} 4 & -1 & . & . & . & . & . & . \\ -1 & 4 & -1 & . & . & . & . & . \\ . & -1 & 4 & -1 & . & . & . & . \\ . & . & -1 & 4 & -1 & . & . & . \\ . & . & . & -1 & 4 & -1 & . & . \\ & & & & \ddots & \ddots & \ddots & \\ . & . & . & . & . & -1 & 4 & -1 \\ . & . & . & . & . & . & -1 & 4 \end{bmatrix}_{(n_x-1)\times(n_x-1)} \tag{4.11}$$

As each element of the the Kronecker sum matrix in (4.10) is of size $(n_x-1)\times(n_x-1)$, it can be expanded to form of matrix of size $(n_x-1)\cdot(n_y-1)\times(n_x-1)\cdot(n_y-1)$ which will be of the form:

$$I_y\otimes L_{xx}+L_{yy}\otimes I_x = \frac{1}{h^2}\left[\begin{array}{ccccc|ccccc|c} 4 & -1 & . & . & . & -1 & . & . & . & . & . \\ -1 & 4 & -1 & . & . & . & -1 & . & . & . & . \\ & \ddots & \ddots & \ddots & & & & \ddots & & & . \\ . & . & -1 & 4 & -1 & . & . & . & -1 & . & . \\ . & . & . & -1 & 4 & . & . & . & . & -1 & . \\ \hline -1 & . & . & . & . & 4 & -1 & . & . & . & . \\ . & -1 & . & . & . & -1 & 4 & -1 & . & . & . \\ & & \ddots & & & & \ddots & \ddots & \ddots & & . \\ . & . & . & -1 & . & . & . & -1 & 4 & -1 & . \\ . & . & . & . & -1 & . & . & . & -1 & 4 & . \\ \hline . & . & . & . & . & \ddots & . & . & . & . & \ddots \end{array}\right]_{(n_x-1)\cdot(n_y-1)\times(n_x-1)\cdot(n_y-1)}$$

Hence, from (4.6),

$$L = I_y \otimes L_{xx} + L_{yy} \otimes I_x \tag{4.12}$$

Furthermore, we know that the backward first-order difference matrix in the $x$-direction $D_x$ can be expressed as:

$$D_x = \frac{1}{h}\begin{bmatrix} 1 & . & . & . & . & . \\ -1 & 1 & . & . & . & . \\ . & -1 & 1 & . & . & . \\ . & . & -1 & 1 & . & . \\ & & & \ddots & \ddots & \\ . & . & . & . & -1 & 1 \\ . & . & . & . & . & -1 \end{bmatrix}_{(n_x)\times(n_x-1)} \tag{4.13}$$

Hence,

$$D_x^T D_x = \frac{1}{h}\begin{bmatrix} 1 & -1 & . & . & . & . & . \\ . & 1 & -1 & . & . & . & . \\ . & . & 1 & -1 & . & . & . \\ . & . & . & 1 & -1 & . & . \\ & & & & \ddots & \ddots & \\ . & . & . & . & . & 1 & -1 \end{bmatrix} \cdot \frac{1}{h}\begin{bmatrix} 1 & . & . & . & . & . \\ -1 & 1 & . & . & . & . \\ . & -1 & 1 & . & . & . \\ . & . & -1 & 1 & . & . \\ & & & \ddots & \ddots & \\ . & . & . & . & -1 & 1 \\ . & . & . & . & . & -1 \end{bmatrix} \tag{4.14}$$

$$\therefore \quad D_x^T D_x = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & . & . & . & . & . & . \\ -1 & 2 & -1 & . & . & . & . & . \\ . & -1 & 2 & 1 & . & . & . & . \\ . & . & -1 & 2 & -1 & . & . & . \\ . & . & . & -1 & 2 & -1 & . & . \\ & & & & \ddots & \ddots & \ddots & \\ . & . & . & . & . & -1 & 2 & -1 \\ . & . & . & . & . & . & -1 & 2 \end{bmatrix}_{(n_x-1)\times(n_x-1)} \tag{4.15}$$

Hence, from (4.7) and (4.15), it is proved that

$$L_{xx} = D_x^T D_x \tag{4.16}$$

We can also prove that $L_{yy} = D_y^T D_y$ using the same procedure.

(**c**) Use `sp.diags()` to create sparse matrices $D_x$ and $D_y$. Employing `A.transpose()` to transpose a sparse matrix `A` and `A.dot(B)` to find the product of sparse matrices `A` and `B`, create 1D FD Laplacian matrices $L_{xx}$ and $L_{yy}$. Use `sp.eye()` to create sparse identity matrices $I_x$ and $I_y$, and the Kronecker product function `sp.kron()` to create the sparse 2D FD Laplacian matrix $L$. Visualize L with `plt.spy()` for $h = 0.2$.

**Solution.** The sparse matrices $D_x$ and $D_y$ can be created using the code below.

```
X = 2 #domain lengths
Y = 1
h = 0.2 #grid spacing

nx = int(X/h) #number of grid points
ny = int(Y/h)

maindia_x = (1/h)*np.ones(nx) #main diagonal of Dx
belowdia_x = (-1/h) * np.ones(nx) #adjacent diagonal of Dx

Dx_diagonals = np.array([maindia_x, belowdia_x])

Dx = sp.diags(Dx_diagonals,[0,-1], shape=[nx,nx-1]) #creating sparse array Dx

maindia_y = (1/h) * np.ones(ny) #main diagonal of Dy
belowdia_y = (-1/h) * np.ones(ny) #adjacent diagonal of Dy

Dy_diagonals = np.array([maindia_y, belowdia_y])

Dy = sp.diags(Dy_diagonals,[0,-1], shape=[ny,ny-1]) #creating sparse matrix Dy
```

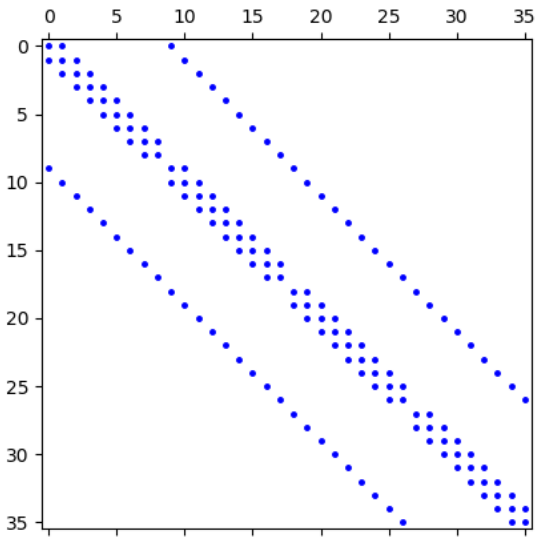
The 1D FD Laplacian matrices $L_{xx}$ and $L_{yy}$ can be determined from $D_x$ and $D_y$ using the sparse dot product `A.dot(B)` as below.

```
Dx_T = Dx.transpose()
Lxx = Dx_T.dot(Dx) #sparse product to create Lxx

Dy_T = Dy.transpose()
Lyy = Dy_T.dot(Dy) #sparse product to create Lyy
```

The 2D Laplacian matrix $L$ can be determined using (4.12), using `sp.eye()` to create the identity matrices and `sp.kron()`, as below.

```
Ix = sp.eye(nx-1) #Identity matrices
Iy = sp.eye(ny-1)

L = sp.kron(Iy,Lxx) + sp.kron(Lyy,Ix) #Equation (4.12)
```

Visualising $L$ with `plt.spy()`, we get the following plot.




(**d**) Create doubly uniform 2D grid on the rectangle with the grid-step $h$. To this end, study the array routine `np.mgrid[]` and use it to create two 2D arrays, `x` and `y`, containing the $x$- and $y$-coordinates of the grid points, including boundary points.

**Solution.** The array routine `np.mgrid[]` is typically used to create grids of varying dimensions depending on the arguments passed to it. It returns an array of arrays, based on the number of dimensions required. In our case, as our grid is 2D with $n_x+1$ and $n_y+1$ grid points in the $x$- and $y$-coordinate axes respectively, the function returns 2 2D arrays, each depicting the $x$-coordinates and $y$-coordinates of the required grid.

The 2D grid can be created using the following code.

```
x,y = h*np.mgrid[0:nx+1,0:ny+1] #create a grid of [nx+1,ny+1] nodes
```

(**e**) Use the `x` and `y` arrays created in (d) and `np.sin()` to compute the grid values of the source function $f(x,y)$ as a two-dimensional array `f`. Use `plt.imshow()` and `plt.colorbar()` to visualize `f`. Change the direction the the $y$ axis in the plot, so that the zero of the coordinate system is in the lower left corner.

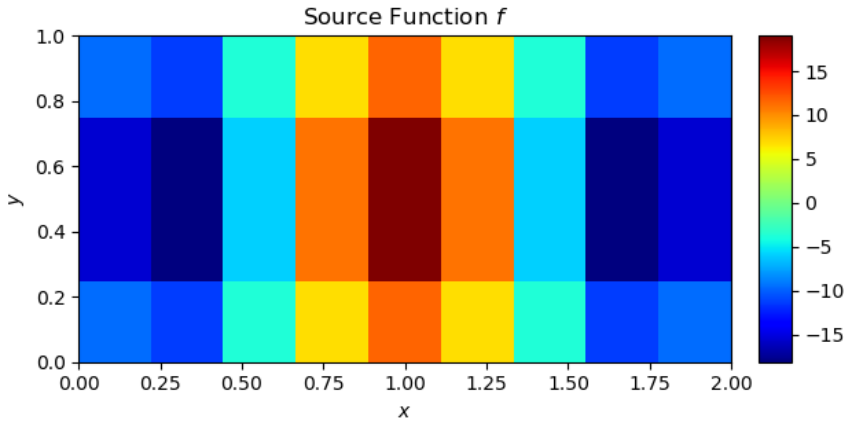
**Solution.** The source function as given in (4.1) can be computed using the grid as follows.

```
f = np.zeros((nx-1,ny-1)) #initializing f

for i in range(0,nx-1):
    for j in range(0,ny-1):
        f[i][j] = 20 * np.sin(np.pi*y[i+1][j+1])* \
                        np.sin(1.5*np.pi*x[i+1][j+1] + np.pi) #source function from (4.1)
```

Using `plt.imshow()` and `plt.colorbar` to visualize the magnitude of the source function across the grid, $f$ can be visualized using the below code. The origin of the plot is set to the lower right corner using the `.T` function to transpose the input data and setting the origin as `lower`.

```
plt.figure()
ax = plt.gca() #get current axes on figure
im = ax.imshow(f.T,origin='lower',cmap='jet') #show image with origin and colormap
plt.title("Source Function $f$")
plt.ylabel("$y$")
plt.xlabel("$x$")
divider = make_axes_locatable(ax) #margin for the plot
cax = divider.append_axes("right", size="5%", pad=0.2) #set location for colorbar
plt.colorbar(im, cax=cax)
```

The visualisation obtained for $h = 0.2$ is as seen in the image below.




(**f**) Use 1D grids for $x$ and $y$ coordinates to create the boundary value vectors and modify your source-function 2D array `f` as required to make the 2D RHS array.

**Solution.** The boundary value vectors can be determined from (4.1) using the grid $x$- and $y$-coordinates as below. The values at the boundaries are then inserted into the source matrix **f** as below.

```
boundary_x0 = np.zeros(ny+1) #initialising boundary 1D vectors
boundary_xn = np.zeros(ny+1)
boundary_y0 = np.zeros(nx+1)
boundary_yn = np.zeros(nx+1)
```

```
for i in range(0,nx+1):
    boundary_y0[i] = np.sin(0.5*np.pi*x[i][0]) #from (4.1), boundary at y=0

for i in range(0,ny+1):
    boundary_x0[i] = np.sin(2*np.pi*y[0][i]) #from (4.1), boundary at x=0
    boundary_xn[i] = np.sin(2*np.pi*y[nx][i]) #from (4.1), boundary at x=2

for i in range(0,ny-1):
    f[0][i] = f[0][i] + boundary_x0[i+1] / (h**2) #inserting boundary at x=0
    f[nx-2][i] = f[nx-2][i] + boundary_xn[i+1] / (h**2) #inserting boundary at x=2

for i in range(0,nx-1):
    f[i][0] = f[i][0] + boundary_y0[i+1] / (h**2) #inserting boundary at y=0
```

The resultant source function is the RHS matrix **f** of the algebraic equation, albeit in 2D form.

(**g**) Use `np.reshape()` to reshape your 2D RHS array into a lexicographically ordered vector, using `order = 'F'` option.

**Solution.** The 2D RHS matrix **f** can be converted into 1D using `np.reshape()`. As we are employing a lexicographic order to the solution, the conditional argument `order = 'F'` is passed to the function. The argument `order='F'` reshapes the array in a Fortran-like order with the first index changing from the initial value to the final value first, followed by increments in the second index and so on. The code to reshape **f** into a 1D array is as below.

```
F = np.reshape(f,-1,order='F') #create 1D RHS matrix F from 2D matrix f
```

Here, the additional argument `-1` indicates that the result is to be 1D.

(**h**) Use `la.spsolve()` to solve the algebraic problem. Then, use `np.reshape()` to reshape the solution vector into the appropriate 2D array and visualize it using `plt.imshow()`. Present your results for $h = 0.02$. You should also be able to run your code with $h = 0.002$.

**Solution.** The algebraic equation $L\mathbf{u} = \mathbf{f}$ is solved using the sparse matrix $L$ and the 1D RHS matrix **f**, using `la.spsolve()` to obtain the 1D solution matrix **u** in lexicographic order.

As the solution matrix **u** is in 1D form, it is reshaped using `np.reshape()` to the 2D array of size `[nx-1,ny-1]`. The reshaped solution matrix **u** can be visualised in the same way as in (**e**) using `plt.imshow()`. The results below are determined for $h = 0.02$. As the entire program is related to the value of `h` declared at the beginning of the code, changing the value of `h` will execute the program without any errors in code.