Name : Kunal Ghosh

Course : M. Tech (Course)

Subject : Computational Heat Transfer and Fluid Flow (ME282)

Assignment No. : 2

SAP No. : 60000076 45

S. R. No. : 05-01-00-10-42-22-1-21061

Answer ①:

$$u'' = -\sin(\beta\pi x) \quad \text{— ①}$$

Using second order finite difference,

$$u_i'' = \frac{u_{i-1} - 2u_i + u_{i+1}}{(\Delta x)^2} \quad \text{— ②}$$

where $u_i = u(x_i)$ — ③

$$x_i = \frac{i}{N}, \quad i = 0, 1, \ldots, N \quad \text{— ④}$$

Using ① and ②

$$u_i'' = -\sin(\beta\pi x_i)$$

or $$\frac{u_{i-1} - 2u_i + u_{i+1}}{(\Delta x)^2} = -\sin(\beta\pi x_i)$$

or $$u_{i-1} - 2u_i + u_{i+1} = -(\Delta x)^2 \sin(\beta\pi x_i)$$

Let $\Delta x = h$ — ⑤

So, $$u_{i-1} - 2u_i + u_{i+1} = -h^2 \sin(\beta\pi x_i) \quad \text{— ⑥}$$

As $u(0) = 0$ (Given)

and $x_0 = 0$

So, $u_0 = 0$

At $i = 1$,

$$u_0 - 2u_1 + u_2 = -h^2 \sin(\beta\pi x_1)$$

or $$-2u_1 + u_2 = -u_0 - h^2 \sin(\beta\pi x_1) \quad \text{— ⑦}$$

Similarly,

$u(1) = 0$ (Given)

$x_N = 1$

So, $u_N = 0$

At, $i = N-1$

$$u_{N-2} - 2u_{N-1} + u_N = -h^2 \sin(\beta\pi x_{N-1})$$

or $$u_{N-2} - 2u_{N-1} = -u_N - h^2 \sin(\beta\pi x_{N-1}) \quad \text{— (8)}$$

Using (6), (7) and (8) we have

$$\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 1 & -2 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & & & & & & \\ & & 1 & -2 & 1 & & \\ \vdots & & & & & & \\ 0 & 0 & 0 & \cdots & 1 & -2 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{i-1} \\ u_i \\ u_{i+1} \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} -u_0 - h^2 \sin(\beta\pi x_0) \\ -h^2 \sin(\beta\pi x_2) \\ \vdots \\ -h^2 \sin(\beta\pi x_{i-1}) \\ -h^2 \sin(\beta\pi x_i) \\ -h^2 \sin(\beta\pi x_{i+1}) \\ \vdots \\ -h^2 \sin(\beta\pi x_{N-2}) \\ -u_N - h^2 \sin(\beta\pi x_{N-1}) \end{bmatrix}$$

As this is a tridiagonal matrix we can solve it using Thomas Algorithm.

Thomas Algorithm :

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & & c_{n-1} & \\ 0 & & a_n & b_n & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$$

So, $$c_i' = \begin{cases} \frac{c_i}{b_i} & \text{if } i = 1 \\ \frac{c_i}{b_i - a_i c_{i-1}'} & \text{if } i = 2, 3, \cdots, (N-1) \end{cases}$$

$$d_i' = \begin{cases} \frac{d_i}{b_i} & \text{if } i = 1 \\ \frac{d_i - a_i d_{i-1}'}{b_i - a_i c_{i-1}'} & \text{if } i = 2, 3, \cdots, N-1 \end{cases}$$

The solution can be obtained by back substitution

$x_n = d'_n$

$x_i = d'_i - c'_i x_{i+1} \quad i = n-1, n-2, \ldots, 1$

Answer (1)(a):

$\beta = 10$

L2 Error: $e_2 = \sqrt{\dfrac{\sum_{i=1}^{N} \left(u_i^{calculated} - u_i^{exact}\right)^2}{N}}$

We observe the rate of convergence is -2.
As the slope the line of L2 vs N graph is -2

Answer (1)(b):

If $\beta = 1$, then, $u^{exact} \simeq u$ calculated
If $\beta = 10$, then u calculated is overestimating the u^{exact} at the maxima and minima.
If $\beta = 100$, u^{exact} is NOT estimated properly by u calculated.
(This is because of Nyquist Theorem)

Exact Solution:

$u'' = -\sin(\beta \pi x)$

$\int u''(x)\, dx = -\int \sin(\beta \pi x)\, dx$

$\Rightarrow u'(x) + C_1 = \dfrac{\cos(\beta \pi x)}{(\beta \pi)}$

$\Rightarrow \int u'(x) + C_1\, dx = \int \dfrac{\cos(\beta \pi x)}{(\beta \pi)}\, dx$

$\Rightarrow u(x) + C_1 x + C_2 = \dfrac{\sin(\beta \pi x)}{(\beta \pi)^2}$

$u(0) = 0$

So, $u(0) + G(0) + C_2 = \frac{\sin(\beta\pi(0))}{(\beta\pi)^2}$

$\Rightarrow \underline{C_2 = 0}$

$u(1) = 0$

So, $u(1) + G(1) + 0 = \frac{\sin(\beta\pi(1))}{(\beta\pi)^2}$

$\Rightarrow C_1 = \frac{\sin(\beta\pi)}{(\beta\pi)^2}$

So,

$$u(x) + C_1 x + C_2 = \frac{\sin(\beta\pi x)}{(\beta\pi)^2}$$

$$\Rightarrow \boxed{u(x) = \left(\frac{\sin(\beta\pi x)}{(\beta\pi)^2}\right) - \left(\frac{\sin(\beta\pi)}{(\beta\pi)^2}\right)x}$$

This is the exact solution.

Question 1 a:
CODE:

```
#include <iostream>
#include <string>
#include <cassert>
#include <cmath>
#include<vector>
#include<fstream>
#include<cassert>

using namespace std;

typedef double real;

double const pi = 4.0*atan(1);

// This function generate 1-dimensional the grid of x between X0 and Xl. (X0 < Xl)
// N = Number of steps
// So, N+1 is the number of the grid points
void generate_1D_grid(vector<real> &x, real X0, real Xl, real N)
{
   for (int i = 0; X0 + ((i/N)*(Xl-X0)) <= Xl;i++)
   {
      x.push_back(X0 + ((i/N)*(Xl-X0)));
   }
}

// This function will be computing the RHS of the equtaion
// u" = − sin(beta*pi*x), where, x = (0, 1)
void compute_rhs(vector<real> &RHS, vector<real> &x, real beta, real h)
{
   // This loop will iterate over different rows of the RHS
   for(int i = 1;i < x.size()-1;i++)
   {
      // Populating the RHS vector with the value of RHS at the respetive grid points
      // WITHOUT implementing the boundary condition
      RHS.push_back(-h*h*sin(beta*pi*x[i]));
   }
}

// This function will compute the exact solution for the ODE: u" = − sin(beta*pi*x)
// At the specified grid points
void compute_exact_sol(vector<real> &u_exact, vector<real> &x, real beta)
{
   // This loop will iterate over the all the grid points for some N
   for (int i = 0; i < x.size();i++)
   {
      // Calculating and storing the value of the exact solution at the respective grid points
      u_exact.push_back((sin(beta*pi*x[i])/(beta*pi*beta*pi)) -
((sin(beta*pi)/(beta*pi*beta*pi))*x[i]));
   }
```

```
}

// This function will compute the L2 error
real compute_L2_error(vector<real> &u_exact, vector<real> &sol)
{
   // Verifying the size of the numerical solution and exact solution vectors
   assert(u_exact.size() == sol.size());

   real sum,L2_error;
   sum = 0;

   // This loop will iterate over each element of the numerical solution and exact solution vectors
   // and adding up the square of the error
   for (int i = 0; i < sol.size();i++)
   {
      sum  = sum + ((u_exact[i]-sol[i])*(u_exact[i]-sol[i]));
   }

   // Taking the average of the sum of the squares
   sum = sum/u_exact.size();

   // Taking the square root of the average of the sum of the squares
   L2_error = pow(sum,0.5);

   return L2_error;
}

// This function will compute the maximum error
real compute_max_error(vector<real> &u_exact, vector<real> &sol)
{
   // Verifying the size of the numerical solution and exact solution vectors
   assert(u_exact.size() == sol.size());

   real max_error;
   max_error = 0;

   // This loop will iterate over each element of the numerical solution and exact solution vectors
   // and calculating the maximum of the error
   for (int i = 0; i < sol.size();i++)
   {
      if (abs(u_exact[i] - sol[i])> max_error)
      {
         max_error = abs(u_exact[i] - sol[i]);
      }
   }
   return max_error;
}

// Thomas_Algorithm_Matix_Solver(vector<vector <real> > &A, vector<real> &B, vector<real>
&X)
// solves a system of equations: AX = B, where A is a Tridiagonal matrix
```

```
void Thomas_Algorithm_Matix_Solver(vector<vector <real> > &A, vector<real> &B,
vector<real> &X)
{
   // Forward Sweep
   for (int row = 1;row < B.size(); row++)
   {
      A[row][row] = A[row][row] - (A[row-1][row]*(A[row][row-1]/A[row-1][row-1]));
      B[row] = B[row] - (B[row-1]*(A[row][row-1]/A[row-1][row-1]));
      A[row][row-1] = 0.0;
   }

   // Backward Sweep
   for (int row = B.size()-1; row >= 1; row--)
   {
      B[row-1] = B[row-1] - (B[row]*(A[row-1][row]/A[row][row]));
      A[row-1][row] = 0.0;
   }

   for (int row = 0;row < B.size(); row++)
   {
      X[row] = B[row]/A[row][row];
   }
}

vector<real> input_parameters(string file_name)
{
   // Vector to read the input parameters from a input file
   vector<real> input;
   string item_name;
   int i = 0;

   // Open the file
   ifstream file;
   file.open(file_name);

   // If the file is NOT opened
   if( !file )
   {
      // Error Message if the file couldn't be opened
      cerr << "Error: Input file could not be opened" << endl;
      exit(1);
   }

   cout<<"Input file "<<file_name<<" is opened."<<endl;

   string line;
   while (getline(file, line))
   {
      // Classifying a string as double if the first character is a numeric
      if(line[0]!= '/')
      {
         // To ensure that we are not reading white space characters
```

```
            if(isdigit(line[0]))
            {
               input.push_back(stod(line));
            }
         }
      }

      // Closing the input file
      file.close();

      cout<<"Input file "<<file_name<<" is closed."<<endl;
      cout<<endl;

      return input;
}

// Function to save a vector<double> to a file of given name
void write_to_file(vector<double> &u, string str)
{
      ofstream file;
      // Open the file
      file.open(str);

      // If the file is NOT opened
      if( !file )
      {
         // Error Message if the file couldn't be opened
         cerr << "Error: file could not be opened" << endl;
         exit(1);
      }

      cout<<"Output file "<< str <<" is opened."<<endl;

      // Writing the vector values to the file in scientific notation
      for(int i=0;i<u.size();i++)
      {
         file<<u[i]<<scientific<<endl;
      }

      // Closing the file
      file.close();

      cout<<"Output file "<< str <<" is closed."<<endl;
      cout<<endl;
}

int main()
{
      real X0,Xl,u0,ul,h, beta;
      int N;
      vector<real> Beta, Boundary_Conditions, Interval, L2_error, max_error, Values_of_N;
```

```
// Opening the input file to read the value of beta
Beta = input_parameters("Value_of_Beta.txt");

// Opening the input file to read the values of N
Values_of_N = input_parameters("Values_of_N.txt");

// Opening the input file to read the values of u(X0) and  u(Xl) (Boundary Conditions)
Boundary_Conditions = input_parameters("Boundary_Conditions.txt");

// Opening the input file to read the Interval
Interval = input_parameters("Interval.txt");

// Interval starts at X0 and ends at Xl
X0 = Interval[0];
Xl = Interval[1];

// Assigning the value of boundary conditions to separate variables
u0 = Boundary_Conditions[0];
ul = Boundary_Conditions[1];

// Assigning the value of beta to a variable
beta = Beta[0];

// This loop will iterate over different values of N
for (int i=0;i < Values_of_N.size();i++)
{
   // Assigning the value of N
   N = Values_of_N[i];

   // Calculating the value of h (step size)
   h = (Xl -X0)/N;

   // Declaring vectors for the RHS, grid location (x) and exact solution of the equation
   vector<real> RHS, x, u_exact;

   // Generating the 1 Dimensional grid
   generate_1D_grid(x,X0,Xl,N);

   // Computing the RHS of the equation (WITHOUT implementing the boundary conditions)
   compute_rhs(RHS,x,beta,h);

   // Implementing the boundary conditions x = X0 and x = Xl
   RHS[0] = RHS[0] - (u0);
   RHS[N -2] = RHS[N-2] - (ul);

   // Declaring the rows and columns for the matix in the LHS
   int rows, cols;
   rows = N-1;
   cols = N-1;

   // Declaring the matrix
   vector<vector <real> > A(rows,vector<real> (cols,0.0));
```

```
    // Assigning the values of first row of A
    A[0][0] = -2.0;
    A[0][1] = 1.0;

    // Assigning the values of last row of A
    A[N-2][N-2] = -2.0;
    A[N-2][N-3] = 1.0;

    // This loop will iterate over all the rows of A except first and last row
    for(int i = 1;i < N-2;i++)
    {
       A[i][i-1] = 1.0;
       A[i][i] = -2.0;
       A[i][i+1] = 1.0;
    }

    // Declaring a vector (sol) to store the value of the numerical solution
    vector<real> sol(N-1);

    // Solving the equation using Thomas Algorithm
    Thomas_Algorithm_Matix_Solver(A,RHS,sol);

    // Assigning the value of u at x = X0, using the boundary condition
    sol.insert(sol.begin(),u0);

    // Assigning the value of u at x = Xl, using the boundary condition
    sol.push_back(ul);

    // Computing the exact solution
    compute_exact_sol(u_exact,x,beta);

    // Computing the L2 error
    L2_error.push_back(compute_L2_error(u_exact,sol));

    // Computing the maximum error
    max_error.push_back(compute_max_error(u_exact,sol));

    // Write the grid to a csv file
    write_to_file(x,"Q_1_a_Grid_Points_N_is_"+to_string(N)+".csv");

    // Write the numerical solution to a csv file
    write_to_file(sol,"Q_1_a_Numerical_Solution_N_is_"+to_string(N)+".csv");

    // Write the exact solution to a csv file
    write_to_file(u_exact,"Q_1_a_Exact_Solution_N_is_"+to_string(N)+".csv");
}

// The order of the L2 error will be same as that of the N in the input file "Values_of_N.txt"
// Write the L2 to a csv file
write_to_file(L2_error,"Q_1_a_L2_Error.csv");
```

```
    // The order of the maximum error will be same as that of the N in the input file
"Values_of_N.txt"
    // Write the maximum error to a csv file
    write_to_file(max_error,"Q_1_a_Max_Error.csv");

    return 0;
}
```

Q_1_a_Post_Processing

February 27, 2023

1 Import the necessary libraries

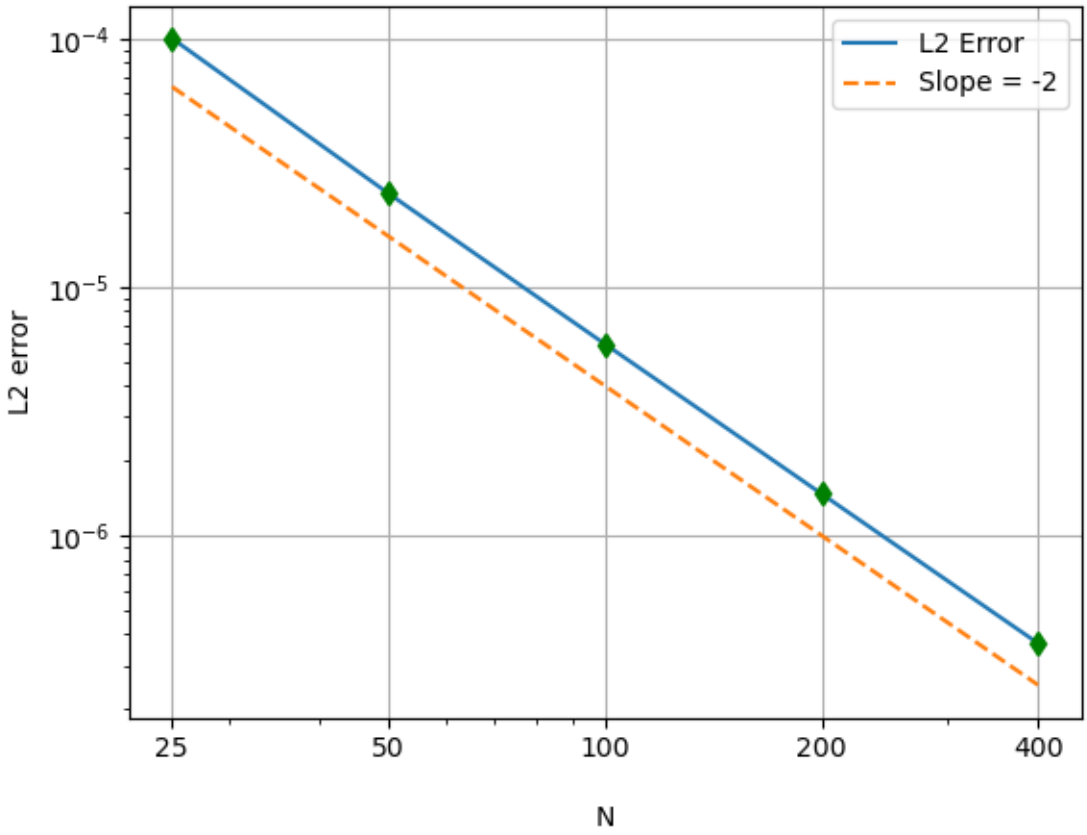
[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

2 Reading the file different values of N

[2]:
```
Input_File = open("Values_of_N.txt", "r")
N_s = []
for line in Input_File:
    if line[0] != "/" and (not line[0].isspace()):
        N_s.append(line)
Input_File.close()
```

[3]:
```
N_num = []
for N in N_s:
    N_num.append(float(N.strip()))
```

[4]:
```
fig = plt.figure()
L2_Error = pd.read_csv("Q_1_a_L2_Error.csv", delimiter = ",",header=None).
 ↪to_numpy()
plt.loglog(N_num,L2_Error)
plt.plot(N_num,(5*np.array(N_num))**(-2),"--")
plt.plot(N_num,L2_Error,"gd")
plt.legend(["L2 Error","Slope = -2"])
plt.xlabel("N")
plt.ylabel("L2 error")
plt.xticks(ticks=N_num,labels=N_s)
plt.grid()
plt.show()
fig.savefig("Q_1_a_L2_Error_vs_N.png",dpi = 500, bbox_inches="tight")
```


L2 Error
Slope = -2
10^{-4}
10^{-5}
10^{-6}
L2 error
25
50
100
200
400
N

Question 1 b:
CODE:

```
#include <iostream>
#include <string>
#include <cassert>
#include <cmath>
#include<vector>
#include<fstream>
#include<cassert>

using namespace std;

typedef double real;

double const pi = 4.0*atan(1);

// This function generate 1-dimensional the grid of x between X0 and Xl. (X0 < Xl)
// N = Number of steps
// So, N+1 is the number of the grid points
void generate_1D_grid(vector<real> &x, real X0, real Xl, real N)
{
   for (int i = 0; X0 + ((i/N)*(Xl-X0)) <= Xl;i++)
   {
      x.push_back(X0 + ((i/N)*(Xl-X0)));
   }
}

// This function will be computing the RHS of the equtaion
// u" = − sin(beta*pi*x), where, x = (0, 1)
void compute_rhs(vector<real> &RHS, vector<real> &x, real beta, real h)
{
   // This loop will iterate over different rows of the RHS
   for(int i = 1;i < x.size()-1;i++)
   {
      // Populating the RHS vector with the value of RHS at the respetive grid points
      // WITHOUT implementing the boundary condition
      RHS.push_back(-h*h*sin(beta*pi*x[i]));
   }
}

// This function will compute the exact solution for the ODE: u" = − sin(beta*pi*x)
// At the specified grid points
void compute_exact_sol(vector<real> &u_exact, vector<real> &x, real beta)
{
   // This loop will iterate over the all the grid points for some N
   for (int i = 0; i < x.size();i++)
   {
      // Calculating and storing the value of the exact solution at the respective grid points
      u_exact.push_back((sin(beta*pi*x[i])/(beta*pi*beta*pi)) -
((sin(beta*pi)/(beta*pi*beta*pi))*x[i]));
   }
```

```
}

// This function will compute the L2 error
real compute_L2_error(vector<real> &u_exact, vector<real> &sol)
{
   // Verifying the size of the numerical solution and exact solution vectors
   assert(u_exact.size() == sol.size());

   real sum,L2_error;
   sum = 0;

   // This loop will iterate over each element of the numerical solution and exact solution vectors
   // and adding up the square of the error
   for (int i = 0; i < sol.size();i++)
   {
      sum  = sum + ((u_exact[i]-sol[i])*(u_exact[i]-sol[i]));
   }

   // Taking the average of the sum of the squares
   sum = sum/u_exact.size();

   // Taking the square root of the average of the sum of the squares
   L2_error = pow(sum,0.5);

   return L2_error;
}

// This function will compute the maximum error
real compute_max_error(vector<real> &u_exact, vector<real> &sol)
{
   // Verifying the size of the numerical solution and exact solution vectors
   assert(u_exact.size() == sol.size());

   real max_error;
   max_error = 0;

   // This loop will iterate over each element of the numerical solution and exact solution vectors
   // and calculating the maximum of the error
   for (int i = 0; i < sol.size();i++)
   {
      if (abs(u_exact[i] - sol[i])> max_error)
      {
         max_error = abs(u_exact[i] - sol[i]);
      }
   }
   return max_error;
}

// Thomas_Algorithm_Matix_Solver(vector<vector <real> > &A, vector<real> &B, vector<real>
&X)
// solves a system of equations: AX = B, where A is a Tridiagonal matrix
```

```
void Thomas_Algorithm_Matix_Solver(vector<vector <real> > &A, vector<real> &B,
vector<real> &X)
{
   // Forward Sweep
   for (int row = 1;row < B.size(); row++)
   {
      A[row][row] = A[row][row] - (A[row-1][row]*(A[row][row-1]/A[row-1][row-1]));
      B[row] = B[row] - (B[row-1]*(A[row][row-1]/A[row-1][row-1]));
      A[row][row-1] = 0.0;
   }

   // Backward Sweep
   for (int row = B.size()-1; row >= 1; row--)
   {
      B[row-1] = B[row-1] - (B[row]*(A[row-1][row]/A[row][row]));
      A[row-1][row] = 0.0;
   }

   for (int row = 0;row < B.size(); row++)
   {
      X[row] = B[row]/A[row][row];
   }
}

vector<real> input_parameters(string file_name)
{
   // Vector to read the input parameters from a input file
   vector<real> input;
   string item_name;
   int i = 0;

   // Open the file
   ifstream file;
   file.open(file_name);

   // If the file is NOT opened
   if( !file )
   {
      // Error Message if the file couldn't be opened
      cerr << "Error: Input file could not be opened" << endl;
      exit(1);
   }

   cout<<"Input file "<<file_name<<" is opened."<<endl;

   string line;
   while (getline(file, line))
   {
      // Classifying a string as double if the first character is a numeric
      if(line[0]!= '/')
      {
         // To ensure that we are not reading white space characters
```

```
            if(isdigit(line[0]))
            {
               input.push_back(stod(line));
            }
         }
      }

      // Closing the input file
      file.close();

      cout<<"Input file "<<file_name<<" is closed."<<endl;
      cout<<endl;

      return input;
}

// Function to save a vector<double> to a file of given name
void write_to_file(vector<double> &u, string str)
{
      ofstream file;
      // Open the file
      file.open(str);

      // If the file is NOT opened
      if( !file )
      {
         // Error Message if the file couldn't be opened
         cerr << "Error: file could not be opened" << endl;
         exit(1);
      }

      cout<<"Output file "<< str <<" is opened."<<endl;

      // Writing the vector values to the file in scientific notation
      for(int i=0;i<u.size();i++)
      {
         file<<u[i]<<scientific<<endl;
      }

      // Closing the file
      file.close();

      cout<<"Output file "<< str <<" is closed."<<endl;
      cout<<endl;
}

int main()
{
      real X0,Xl,u0,ul,h, beta;
      int N;
      vector<real> Value_of_N, Boundary_Conditions, Interval, L2_error, max_error, Values_of_Beta;
```

```
// Opening the input file to read the value of beta
Values_of_Beta = input_parameters("Values_of_Beta.txt");

// Opening the input file to read the values of N
Value_of_N = input_parameters("Value_of_N.txt");

// Opening the input file to read the values of u(X0) and  u(Xl) (Boundary Conditions)
Boundary_Conditions = input_parameters("Boundary_Conditions.txt");

// Opening the input file to read the Interval
Interval = input_parameters("Interval.txt");

// Interval starts at X0 and ends at Xl
X0 = Interval[0];
Xl = Interval[1];

// Assigning the value of boundary conditions to separate variables
u0 = Boundary_Conditions[0];
ul = Boundary_Conditions[1];

// Assigning the value of N
N = Value_of_N[0];

// This loop will iterate over different values of beta
for (int i=0;i < Values_of_Beta.size();i++)
{
   // Assigning the value of beta to a variable
   beta = Values_of_Beta[i];

   // Calculating the value of h (step size)
   h = (Xl -X0)/N;

   // Declaring vectors for the RHS, grid location (x) and exact solution of the equation
   vector<real> RHS, x, u_exact;

   // Generating the 1 Dimensional grid
   generate_1D_grid(x,X0,Xl,N);

   // Computing the RHS of the equation (WITHOUT implementing the boundary conditions)
   compute_rhs(RHS,x,beta,h);

   // Implementing the boundary conditions x = X0 and x = Xl
   RHS[0] = RHS[0] - (u0);
   RHS[N -2] = RHS[N-2] - (ul);

   // Declaring the rows and columns for the matix in the LHS
   int rows, cols;
   rows = N-1;
   cols = N-1;

   // Declaring the matrix
   vector<vector <real> > A(rows,vector<real> (cols,0.0));
```

```
    // Assigning the values of first row of A
    A[0][0] = -2.0;
    A[0][1] = 1.0;

    // Assigning the values of last row of A
    A[N-2][N-2] = -2.0;
    A[N-2][N-3] = 1.0;

    // This loop will iterate over all the rows of A except first and last row
    for(int i = 1;i < N-2;i++)
    {
       A[i][i-1] = 1.0;
       A[i][i] = -2.0;
       A[i][i+1] = 1.0;
    }

    // Declaring a vector (sol) to store the value of the numerical solution
    vector<real> sol(N-1);

    // Solving the equation using Thomas Algorithm
    Thomas_Algorithm_Matix_Solver(A,RHS,sol);

    // Assigning the value of u at x = X0, using the boundary condition
    sol.insert(sol.begin(),u0);

    // Assigning the value of u at x = Xl, using the boundary condition
    sol.push_back(ul);

    // Computing the exact solution
    compute_exact_sol(u_exact,x,beta);

    // Computing the L2 error
    L2_error.push_back(compute_L2_error(u_exact,sol));

    // Computing the maximum error
    max_error.push_back(compute_max_error(u_exact,sol));

    // Write the grid to a csv file
    write_to_file(x,"Q_1_b_Grid_Points_beta_is_"+to_string(beta)+".csv");

    // Write the numerical solution to a csv file
    write_to_file(sol,"Q_1_b_Numerical_Solution_beta_is_"+to_string(beta)+".csv");

    // Write the exact solution to a csv file
    write_to_file(u_exact,"Q_1_b_Exact_Solution_beta_is_"+to_string(beta)+".csv");
}

// The order of the L2 error will be same as that of the N in the input file "Values_of_N.txt"
// Write the L2 to a csv file
write_to_file(L2_error,"Q_1_b_L2_Error.csv");
```

```
    // The order of the maximum error will be same as that of the N in the input file
"Values_of_N.txt"
    // Write the maximum error to a csv file
    write_to_file(max_error,"Q_1_b_Max_Error.csv");

    return 0;
}
```

Q_1_b_Post_Processing

February 27, 2023

1 Import the necessary libraries

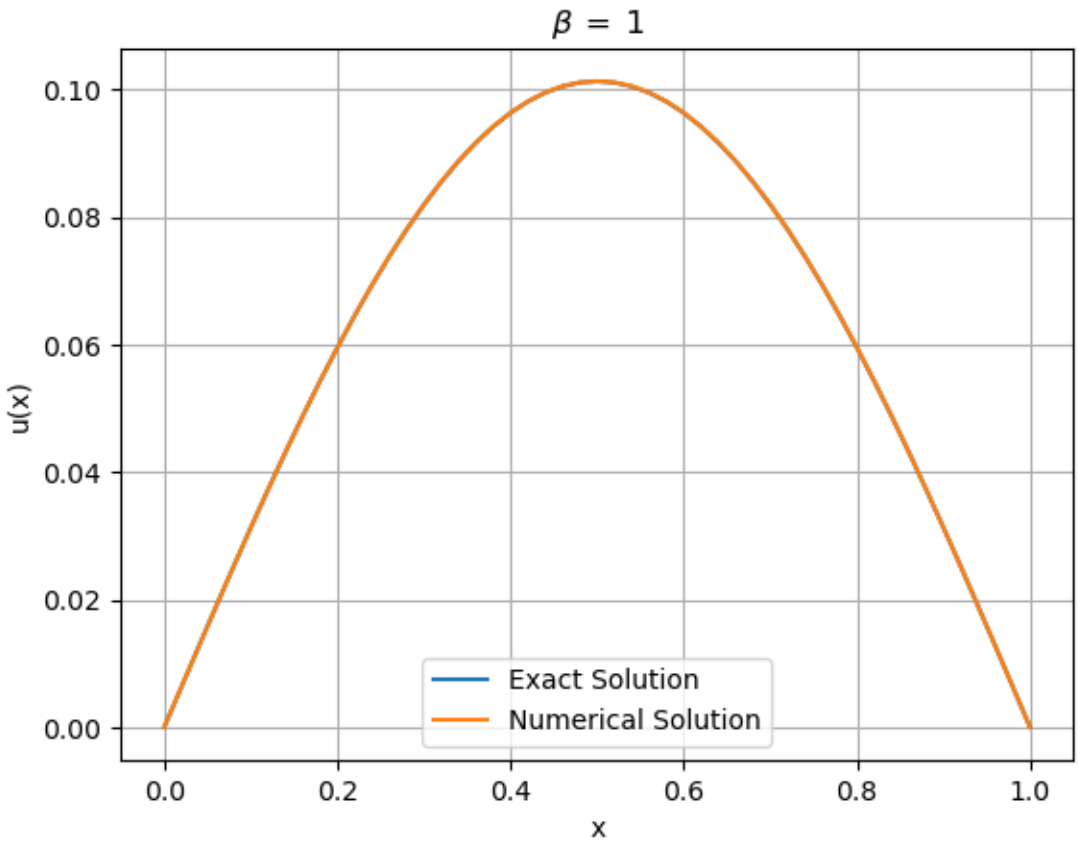
[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

2 Reading the file for different values of β

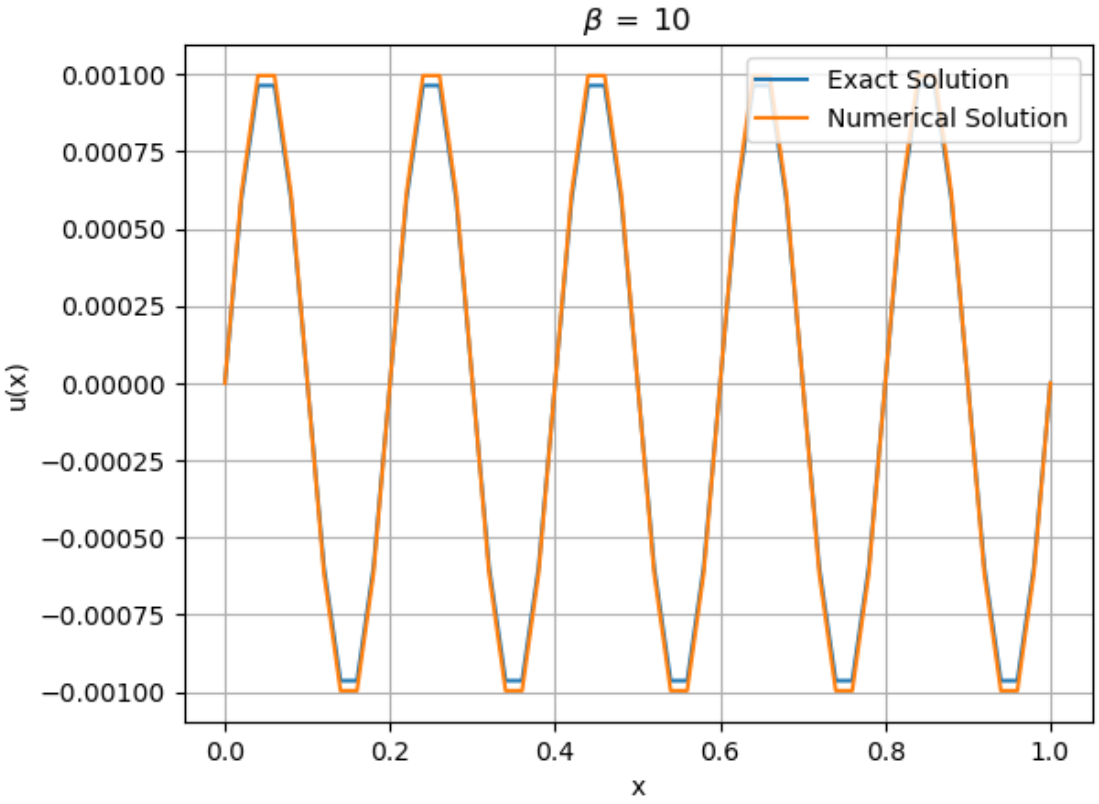
[2]:
```
Input_File = open("Values_of_Beta.txt", "r")
beta_s = []
for line in Input_File:
    if line[0] != "/" and (not line[0].isspace()):
        beta_s.append(line)
Input_File.close()
```

[3]:
```
beta_num = []
for beta in beta_s:
    beta_num.append(float(beta.strip()))
```

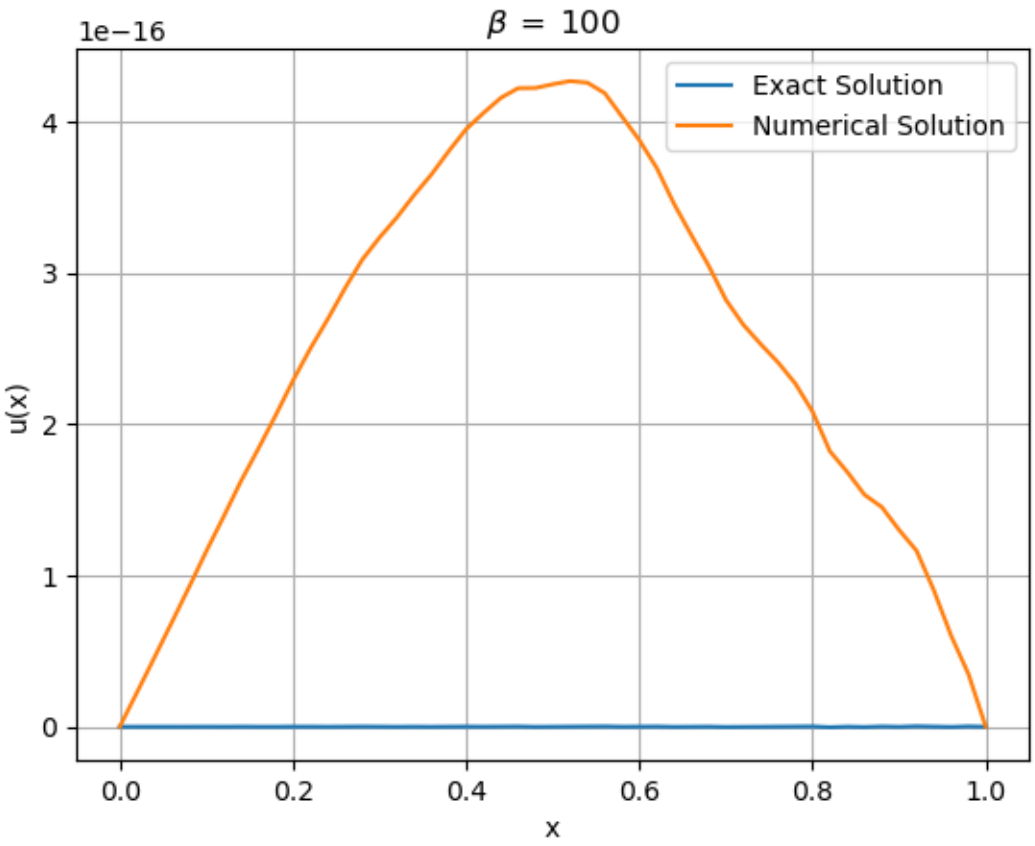
[4]:
```
fig = plt.figure()
Grid_N_1 = pd.read_csv("Q_1_b_Grid_Points_beta_is_1.000000.csv", delimiter =␣
 ↪",",header=None).to_numpy()
U_Exact_N_1 = pd.read_csv("Q_1_b_Exact_Solution_beta_is_1.000000.csv",␣
 ↪delimiter = ",",header=None).to_numpy()
U_Solution_N_1 = pd.read_csv("Q_1_b_Numerical_Solution_beta_is_1.000000.csv",␣
 ↪delimiter = ",",header=None).to_numpy()
plt.plot(Grid_N_1,U_Exact_N_1)
plt.plot(Grid_N_1,U_Solution_N_1)
plt.legend(["Exact Solution","Numerical Solution"])
plt.xlabel("x")
plt.ylabel("u(x)")
plt.title(r"$\beta\:=\:1$")
plt.grid()
plt.show()
fig.savefig("Q_1_b_beta_is_1_u_vs_x.png",dpi = 500, bbox_inches="tight")
```



[5]:
```
fig = plt.figure()
Grid_N_10 = pd.read_csv("Q_1_b_Grid_Points_beta_is_10.000000.csv", delimiter =␣
 ↪",",header=None).to_numpy()
U_Exact_N_10 = pd.read_csv("Q_1_b_Exact_Solution_beta_is_10.000000.csv",␣
 ↪delimiter = ",",header=None).to_numpy()
U_Solution_N_10 = pd.read_csv("Q_1_b_Numerical_Solution_beta_is_10.000000.csv",␣
 ↪delimiter = ",",header=None).to_numpy()
plt.plot(Grid_N_10,U_Exact_N_10)
plt.plot(Grid_N_10,U_Solution_N_10)
plt.legend(["Exact Solution","Numerical Solution"])
plt.xlabel("x")
plt.ylabel("u(x)")
plt.title(r"$\beta\:=\:10$")
plt.grid()
plt.show()
fig.savefig("Q_1_b_beta_is_10_u_vs_x.png",dpi = 500, bbox_inches="tight")
```



[6]:
```
fig = plt.figure()
Grid_N_100 = pd.read_csv("Q_1_b_Grid_Points_beta_is_100.000000.csv", delimiter␣
 ↪= ",",header=None).to_numpy()
U_Exact_N_100 = pd.read_csv("Q_1_b_Exact_Solution_beta_is_100.000000.csv",␣
 ↪delimiter = ",",header=None).to_numpy()
U_Solution_N_100 = pd.read_csv("Q_1_b_Numerical_Solution_beta_is_100.000000.
 ↪csv", delimiter = ",",header=None).to_numpy()
plt.plot(Grid_N_100,U_Exact_N_100)
plt.plot(Grid_N_100,U_Solution_N_100)
plt.legend(["Exact Solution","Numerical Solution"])
plt.xlabel("x")
plt.ylabel("u(x)")
plt.title(r"$\beta\:=\:100$")
plt.grid()
plt.show()
fig.savefig("Q_1_b_beta_is_100_u_vs_x.png",dpi = 500, bbox_inches="tight")
```


$\beta = 100$
1e−16
Exact Solution
Numerical Solution
u(x)
x
0
1
2
3
4
0.0
0.2
0.4
0.6
0.8
1.0

Answer ②:

$$u_{xx} + u_{yy} = -f(x,y) \qquad (x,y) \in (0,1)^2$$

$$u(0,y) = u(1,y) = 0 \qquad y \in [0,1]$$

$$u(x,0) = u(x,1) = 0 \qquad x \in [0,1]$$

$$h = \Delta x = \Delta y = \frac{1}{N}$$

$$x_i = ih, \qquad i = 0, 1, 2, \cdots, N$$

$$y_i = ih,$$

Second Order Finite Difference Approximation :

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

$$\text{or } -u_{i+1,j} + 2u_{i,j} - u_{i-1,j} - u_{i,j+1} + 2u_{i,j} - u_{i,j-1} = h^2 f_{i,j}$$

Let $F_{ij} = h^2 f_{i,j}$

Writing it in matrix form.

$$\begin{bmatrix} 4 & -1 & 0 & & & & & & \\ -1 & 4 & -1 & & & & & & \\ & \ddots & \ddots & \ddots & & & & & \\ & & -1 & 4 & -1 & & & & \\ & & & -1 & 4 & -1 & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & & & & & -1 & 4 & -1 & \\ & & & & & & -1 & 4 & -1 \\ & & & & & & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{1,2} \\ \vdots \\ u_{1,N-1} \\ \vdots \\ u_{i,j} \\ \vdots \\ u_{N-1,N-2} \\ u_{N-1,N-1} \end{bmatrix} = \begin{bmatrix} F_{1,1} \\ F_{1,2} \\ \vdots \\ F_{1,N-1} \\ \vdots \\ F_{i,j} \\ \vdots \\ F_{N-1,N-2} \\ F_{N-1,N-1} \end{bmatrix}$$

N-1 Elements

N-1 Elements

A U F

So, $AU = F$

Answer: (2)(a) :-

$$s_{i,j}^{P,q} = \sin(P\pi x_i)\sin(q\pi y_i)$$

We have to prove $[A][s^{P,q}] = \lambda[s^{P,q}]$

λ is the corresponding eigen value

$$[s^{P,q}] = \begin{bmatrix} \sin\left(\frac{P\pi}{N}\right)\sin\left(\frac{q\pi}{N}\right) \\ \sin\left(\frac{2P\pi}{N}\right)\sin\left(\frac{2q\pi}{N}\right) \\ \vdots \\ \sin\left(\frac{iP\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right) \leftarrow i,j \text{ th Row} \\ \vdots \\ \sin\left(\frac{(N-1)P\pi}{N}\right)\sin\left(\frac{(N-1)q\pi}{N}\right) \end{bmatrix}$$

So, considering the i,j th row of $[A][s^{P,q}]$ vector is

~~$-\sin\left(\frac{(i-1)P\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right) - \sin\left(\frac{iP\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right)$~~

$$-\sin\left(\frac{(i-1)P\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right) - \sin\left(\frac{iP\pi}{N}\right)\sin\left(\frac{(j-1)q\pi}{N}\right)$$

$$+4\sin\left(\frac{iP\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right) - \sin\left(\frac{iP\pi}{N}\right)\sin\left(\frac{(j+1)q\pi}{N}\right)$$

$$-\sin\left(\frac{(i+1)P\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right)$$

$$= -\sin\left(\frac{jq\pi}{N}\right)\left[\sin\left(\frac{(i-1)P\pi}{N}\right) + \sin\left(\frac{(i+1)P\pi}{N}\right)\right]$$

$$-\sin\left(\frac{iP\pi}{N}\right)\left[\sin\left(\frac{(j-1)q\pi}{N}\right) + \sin\left(\frac{(j+1)q\pi}{N}\right)\right]$$

$$+4\sin\left(\frac{iP\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right)$$

As, $\sin(A+B) + \sin(A-B) = 2\sin(A)\cos(B)$

So, this term will be

$$-\sin\left(\frac{jq\pi}{N}\right)\left[2\sin\left(\frac{ip\pi}{N}\right)\cos\left(\frac{p\pi}{N}\right)\right]$$

$$-\sin\left(\frac{ip\pi}{N}\right)\left[2\sin\left(\frac{jq\pi}{N}\right)\cos\left(\frac{q\pi}{N}\right)\right]$$

$$+4\sin\left(\frac{ip\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right)$$

$$= \left[4 - 2\cos\left(\frac{p\pi}{N}\right) - 2\cos\left(\frac{q\pi}{N}\right)\right]\left[\sin\left(\frac{ip\pi}{N}\right)\sin\left(\frac{jq\pi}{N}\right)\right]$$

$$= \underbrace{\left[4 - 2\cos\left(\frac{p\pi}{N}\right) - 2\cos\left(\frac{q\pi}{N}\right)\right]}\left[s_{i,j}^{p,q}\right]$$

This is the eigen value $(\lambda_{p,q})$ (As it is independent of i and j)

$$= \lambda_{p,q}\, s_{i,j}^{p,q}$$

So, $[A]\left[s^{p,q}\right] = \lambda_{p,q}\left[s^{p,q}\right]$

$$\lambda_{p,q} = 4 - 2\cos\left(\frac{p\pi}{N}\right) - 2\cos\left(\frac{q\pi}{N}\right)$$

So, the $(N-1)^2$ eigen vectors of A are given by discrete functions $s^{p,q}$.

Answer (2)(b) :-

Eigen value associated with $s^{p,q}$ is $\lambda_{p,q}$

$$\lambda_{p,q} = 4 - 2\cos\left(\frac{p\pi}{N}\right) - 2\cos\left(\frac{q\pi}{N}\right)$$

$$= 2\left(1 - \cos\left(\frac{p\pi}{N}\right)\right) + 2\left(1 - \cos\left(\frac{q\pi}{N}\right)\right)$$

$$\lambda_{p,q} = 4\left[\sin^2\left(\frac{p\pi}{2N}\right) + \sin^2\left(\frac{q\pi}{2N}\right)\right]$$

Condition Number of A

$$= \frac{\text{absolute value of max. eigen value}}{\text{absolute value of min eigen value}}$$

or $\text{cond}(A) = \dfrac{\lambda_{p,q}|_{max}}{\lambda_{p,q}|_{min}}$

$$= \frac{4\left[\sin^2\left(\left(\frac{N-1}{N}\right)\frac{\pi}{2}\right) + \sin^2\left(\left(\frac{N-1}{N}\right)\frac{\pi}{2}\right)\right]}{4\left[\sin^2\left(\frac{\pi}{2N}\right) + \sin^2\left(\frac{\pi}{2N}\right)\right]}$$

For very large N

$$\sin\left(\frac{\pi}{2N}\right) \simeq \frac{\pi}{2N} \quad \text{(Small Angle Approximation)}$$

Also, $\dfrac{N-1}{N} \simeq 1$

So, $\text{cond}(A) = \dfrac{4\left[\sin^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right)\right]}{4\left[\left(\frac{\pi}{2N}\right)^2 + \left(\frac{\pi}{2N}\right)^2\right]}$

So, $cond(A) = \frac{4\left[1^2 + 1^2\right]}{4\left[\frac{\pi^2}{4N^2} + \frac{\pi^2}{4N^2}\right]} = \frac{8}{4\left[\frac{2\pi^2}{4N^2}\right]}$

$$cond(A) = \frac{4N^2}{\pi^2}$$

This is the condition number of A.

Answer (2)(c)

Given $f = -h^2 \begin{bmatrix} \sin\left(\frac{5\pi}{N}\right)\sin\left(\frac{6\pi}{N}\right) \\ \vdots \\ \sin\left(\frac{5\pi i}{N}\right)\sin\left(\frac{6\pi j}{N}\right) \\ \vdots \\ \sin\left(\frac{5\pi(N-1)}{N}\right)\sin\left(\frac{6\pi(N-1)}{N}\right) \end{bmatrix}$

As $AU = F$,

$A = [P][\lambda][P]^{-1}$

As $F = -h^2[S^{5,6}]$

Let $\bar{F} = P^{-1}F$ and $P^{-1}U = \bar{U}$

So, $[P][\lambda][P]^{-1}[U] = [F]$

or $[P][\lambda][\bar{U}] = [F]$

or $[\lambda][\bar{U}] = [P]^{-1}[F]$

or $[\lambda][\bar{U}] = [\bar{F}]$

So, $U_{i,j} = \frac{\bar{F}_{i,j}}{\lambda_{i,j}}$

As $P\bar{F} = F$

So, the matrix equation is

$$\begin{bmatrix} \uparrow & & \uparrow \\ \sin\left(\frac{i\pi}{N}\right)\sin\left(\frac{j\pi}{N}\right) & \cdots & \sin\left(\frac{iP\pi}{N}\right)\sin\left(\frac{jQ\pi}{N}\right) \\ \downarrow & & \downarrow \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ h^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$= h^2 \begin{bmatrix} -\sin\left(\frac{5\pi}{N}\right)\sin\left(\frac{6\pi}{N}\right) \\ \vdots \\ -\sin\left(\frac{5\pi i}{N}\right)\sin\left(\frac{6\pi j}{N}\right) \\ \vdots \\ -\sin\left(\frac{5\pi(N-1)}{N}\right)\sin\left(\frac{6\pi(N-1)}{N}\right) \end{bmatrix}$$

So, $\bar{F} = \begin{bmatrix} 0 & \cdots & -h^2 & \cdots & 0 \end{bmatrix}^T$

So, $\bar{u} = \frac{\bar{F}}{\lambda}$ or $\bar{u} = \begin{bmatrix} 0 & \cdots & \left(\frac{-h^2}{4\left[\sin^2\left(\frac{5\pi}{2N}\right) + \sin^2\left(\frac{6\pi}{2N}\right)\right]}\right) & \cdots & 0 \end{bmatrix}^T$

As $u = P\bar{u}$

So, $u = -h^2 \begin{bmatrix} \frac{\sin\left(\frac{5\pi}{N}\right)\sin\left(\frac{6\pi}{N}\right)}{4\left[\sin^2\left(\frac{5\pi}{2N}\right) + \sin^2\left(\frac{6\pi}{2N}\right)\right]} \\ \vdots \\ \frac{\sin\left(\frac{5\pi i}{N}\right)\sin\left(\frac{6\pi j}{N}\right)}{4\left[\sin^2\left(\frac{5\pi}{2N}\right) + \sin^2\left(\frac{6\pi}{2N}\right)\right]} \\ \vdots \\ \frac{\sin\left(\frac{5\pi(N-1)}{N}\right)\sin\left(\frac{6\pi(N-1)}{N}\right)}{4\left[\sin^2\left(\frac{5\pi}{2N}\right) + \sin^2\left(\frac{6\pi}{2N}\right)\right]} \end{bmatrix}$

This is the Analytical Solution

As $N \longrightarrow \infty$

$$\sin\left(\frac{5\pi}{2N}\right) \simeq \left(\frac{5\pi}{2N}\right)$$

$$\sin\left(\frac{6\pi}{2N}\right) \simeq \left(\frac{6\pi}{2N}\right)$$

Then i,j the element becomes

$$u_{ij} = \frac{-8\sin\left(\frac{5\pi i}{N}\right)\sin\left(\frac{6\pi j}{N}\right)}{(5\pi)^2 + (6\pi)^2}$$

[~~N in the numerator will be there for each term. Hence, it is just scaled~~]

Using $x_i = \frac{i}{N}$, $y_j = \frac{j}{N}$

$$u(x_i, y_j) = \frac{-\sin(5\pi x)\sin(6\pi y)}{(5\pi)^2 + (6\pi)^2} \quad -(1)$$

In general $u(x,y)$ and $f(x,y)$ are expressed in terms of fourier series, as

$$u(x,y) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} a_{nm}\sin(n\pi x)\sin(m\pi y)$$

$$f(x,y) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \bar{f}_{n,m}\sin(n\pi x)\sin(m\pi y)$$

So, $u_{xx} + u_{yy} = -(n^2\pi^2 + m^2\pi^2)\sum_{n=1}^{\infty}\sum_{m=1}^{\infty} a_{nm}\sin(n\pi x)\sin(m\pi y)$

Comparing coefficients, $a_{nm} = \frac{-\bar{f}_{nm}}{(n\pi)^2 + (m\pi)^2}$

So, $u(x,y) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} a_{nm}\sin(n\pi x)\sin(m\pi y)$

$$= \frac{-\sin(5\pi x)\sin(6\pi y)}{(5\pi)^2 + (6\pi)^2} \quad -(2)$$

So, As $N \longrightarrow \infty$, Equation (1) and (2) looks similar

So, approx. solution matches exact solution.

Answer (3):

$$\nabla^2 u + 1000u = (1000 - 200\pi^2)\sin(10\pi x)\cos(10\pi y) \quad \text{— (1)}$$

As, $\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$

Let $u_{i,j} = u(x_i, y_j)$

So, $\nabla^2 u_{i,j} = \left(\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2}\right) + \left(\frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2}\right)$ — (2)

As it is a uniform grid

So, $h = \Delta x = \Delta y$ — (3)

So,

$$\nabla^2 u_{i,j} + 1000u_{i,j} = (1000 - 2000\pi^2)\sin(10\pi x)\cos(10\pi y)$$

or $(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) + (u_{i,j-1} - 2u_{i,j} + u_{i,j+1}) + 1000h^2 u_{i,j}$

$= [(1000 - 200\pi^2)\sin(10\pi x_i)\cos(10\pi y_j)]h^2$ — (4)

As $u(0, y) = 0$

So, $u_{0,j} = 0$

At $i = 1$,

or $(u_{0,j} - 2u_{1,j} + u_{2,j}) + (u_{1,j-1} - 2u_{1,j} + u_{1,j+1}) + 1000h^2 u_{1,j}$

$= [h^2(1000 - 200\pi^2)\sin(10\pi x_1)\cos(10\pi y_j)]$

or $(-2u_{1,j} + u_{2,j}) + (u_{1,j-1} - 2u_{1,j} + u_{1,j+1}) + 1000h^2 u_{1,j}$

$= -u_{0,j} + h^2(1000 - 200\pi^2)\sin(10\pi x_1)\cos(10\pi y_j)$

$= h^2(1000 - 200\pi^2)\sin(10\pi x_1)\cos(10\pi y_j)$ — (5)

Similarly at $i = N-1$

$u(1, y) = 0$

$u_{N,j} = 0$

So,

$$(u_{N-2,j} - 2u_{N-1,j} + u_{Nj}) + (u_{N-1,j-1} - 2u_{N-1,j} + u_{N-1,j+1}) + 1000h^2 u_{N-1,j} = h^2(1000 - 200\pi^2)\sin(10\pi x_{N-1})\cos(10\pi y_j)$$

$$\Rightarrow (u_{N-2,j} - 2u_{N-1,j}) + (u_{N-1,j-1} - 2u_{N-1,j} + u_{N-1,j+1}) + 1000h^2 u_{N-1,j} = h^2(1000 - 200\pi^2)\sin(10\pi x_{N-1})\cos(10\pi y_j) \quad -(6)$$

Similarly at ~~$j=0$~~ $j=1$

$u(x, 0) = \sin(10\pi x)$

$u_{i,0} = \sin(10\pi x_i)$

~~So $u_{i-1} - 2u_i$~~

So, $(u_{i-1,1} - 2u_{i,1} + u_{i+1,1}) + (u_{i,0} - 2u_{i,1} + u_{i,2}) + 1000h^2 u_{i,1} = h^2(1000 - 200\pi^2)\sin(10\pi x_i)\cos(10\pi y_j)$

$$\Rightarrow (u_{i-1,1} - 2u_{i,1} + u_{i+1,1}) + (-2u_{i,1} + u_{i,2}) + 1000h^2 u_{i,1} = h^2(1000 - 200\pi^2)\sin(10\pi x_i)\cos(10\pi y_j) - \sin(10\pi x_i)$$

Similarly at $j = N-1$, $u(x, 1) = \sin(10\pi x)$ $-(7)$

$u_{i,N} = \sin(10\pi x_i)$

$$(u_{i-1,N-1} - 2u_{i,N-1} + u_{i+1,N-1}) + (u_{i,N-2} - 2u_{i,N-1} + u_{i,N}) + 1000h^2 u_{i,N-1} = h^2(1000 - 200\pi^2)\sin(10\pi x_i)\cos(10\pi y_j)$$

$$(u_{i-1,N-1} - 2u_{i,N-1} + u_{i+1,N-1}) + (u_{i,N-2} - 2u_{i,N-1}) + 1000h^2 u_{i,N-1}$$
$$= h^2(1000 - 200\pi^2)\sin(10\pi x_i)\cos(10\pi y_{N-1}) - \sin(10\pi x_i) \quad \text{— (8)}$$

Using (4), (5), (6), (7) & (8), we have

$$\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,N-1} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,N-1} \\ \vdots & & & \\ u_{N-1,1} & u_{N-1,2} & \cdots & u_{N-1,N-1} \end{bmatrix}$$

$$+ \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,N-1} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,N-1} \\ \vdots & & & \\ u_{N-1,1} & u_{N-1,2} & \cdots & u_{N-1,N-1} \end{bmatrix} \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}$$

$$+ 1000h^2 [U]$$

$$* = \begin{bmatrix} R_{1,1} - S_1 & R_{1,2} & \cdots & R_{1,N-2} & R_{1,N-1} - S_1 \\ R_{2,1} - S_2 & R_{2,2} & \cdots & R_{2,N-2} & R_{2,N-1} - S_2 \\ \vdots & & & & \\ R_{N-1,1} - S_{N-1} & R_{N-1,2} & \cdots & R_{N-1,N-2} & R_{N-1,N-1} - S_{N-1} \end{bmatrix}$$

Here,

$R_{ij} = h^2(1000 - 200\pi^2)\sin(10\pi x_i)\cos(10\pi y_j)$

~~$S_i = 10\pi x_i$~~

$S_i = \sin(10\pi x_i)$

Let the RHS be $[F]$

Let $[A] = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & & \\ & 1 & \ddots & & \\ \vdots & & & & \\ 0 & & 1 & -2 & 1 \\ & & 0 & 1 & -2 \end{bmatrix}$ $\quad A = A^T$

$[U] = [U_{ij}]$

So, $[A][U] + [U][A]^T = [F]$

$+ 1000h^2[U]$

So, As $[A] = [P][\Lambda][P]^{-1}$, $[\Lambda] = [\Lambda]^T$

$[P][\Lambda][P]^{-1}[U] + [U]\left[[P]^{-1}\right]^T[\Lambda]^T[P]^T$

$+ 1000h^2[U] = [F]$

or $[\Lambda][P]^{-1}[U] + [P]^{-1}[U]\left[[P]^{-1}\right]^T[\Lambda]^T[P]^T$

$+ 1000h^2[P]^{-1}U = [P]^{-1}[F]$

or $[\Lambda][P]^{-1}[U]\left[P^T\right]^{-1} + [P]^{-1}[U]\left[P^{-1}\right]^T[\Lambda]$

$+ 1000h^2[P]^{-1}[U]\left[P^T\right]^{-1} = [P]^{-1}[F][P]^{-T}$

Let $[V] = [P]^{-1}[U][P]^{-T}$

So, $[\Lambda][V] + [V][\Lambda] + 1000h^2[V] = [P]^{-1}[F][P]^{-T}$

Let $[\tilde{F}] = [P]^{-1}[F][P]^{-T}$

So, $[\Lambda][V] + [V][\Lambda] + 1000h^2[V] = [\tilde{F}]$

As Λ is diagonal matrix.

So,

$$\Lambda_i V_{ji} + V_{ji}\Lambda_j + 1000h^2 V_{ji} = \tilde{F}_{ji}$$

So, $$V_{ji} = \left(\frac{\tilde{F}_{ji}}{\Lambda_i + \Lambda_j + 1000h^2} \right)$$

We can evaluate V using this

Then

$$[U] = [P][V][P]^T$$

As $[V] = [P]^{-1}[U][P]^{-T}$

Question 3:
CODE:

```
#include <iostream>
#include <string>
#include <cassert>
#include <cmath>
#include<algorithm>
#include<iostream>
#include<vector>
#include<fstream>
#include<cmath>
#include<string>
#include<unistd.h>
#include<functional>
#include<cassert>

using namespace std;

typedef double real;

double const pi = 4.0*atan(1.0);

// Generate 1D grid
void generate_1D_grid(vector<real> &x, real X0, real Xl, real N)
{
   for (int i = 0; X0 + ((i/N)*(Xl-X0)) <= Xl;i++)
   {
      x.push_back(X0 + ((i/N)*(Xl-X0)));
      //cout<<X0 + ((i/N)*(Xl-X0))<<endl;
   }
}

// This function calculates the transpose of a given matrix
void matrix_transpose(vector<vector<real>>& A, vector<vector<double>> &A_Transpose)
{
   int rows = A.size();
   int cols = A[0].size();

   for (int row = 0; row < rows; row++)
   {
      for (int col = 0; col < cols; col++)
      {
         A_Transpose[col][row] = A[row][col];
      }
   }
}

// This function calculates the eigen values , eigen vector matrix and inverse of eigen vector matrix
void compute_eigen_value_vector_inverse(vector<vector<real>> &P, vector<vector<real>>
&PINV, vector<real> &eigen_values, vector<real> &x,real Nx,real hx)
{
```

```
    // Set up eigenvalues and matrices
    int i, j;
        for(i = 1 ; i < Nx ; i++) {
                eigen_values.push_back(-4.0*sin(pi*0.5*x[i])*sin(pi*0.5*x[i]));
                for(j = 1 ; j < Nx ; j++) {
                        P[i-1][j-1] = sin(i*pi*x[j]) ;
                        PINV[i-1][j-1] = 2.0*hx*sin(j*pi*x[i]) ;
                }
        }
}

// This function calculate the RHS of the equation
void compute_rhs(vector<vector<real>> &RHS, vector<real> &x, vector<real> &y, real h)
{
    int rows = RHS.size();
    int cols = RHS[0].size();
    real pi = 4.0*atan(1);

    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < cols; col++)
        {
            RHS[row][col] = h*h*(1000.0 -
(200.0*pi*pi))*sin(10.0*pi*x[row+1])*cos(10.0*pi*y[col+1]);
        }
    }
}

// This function computes the product of two given matrix
void matrix_multiply(vector<vector<real>> &A, vector<vector<double>> &B,
vector<vector<double>> &C)
{
    int m = A.size();
    int n = B.size();
    int p = B[0].size();

    // vector<vector<double>> C(m, vector<double>(p, 0));

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    //return C;
}

// This function computes the V matrix
void compute_V(vector<vector <real>> &F_tilde, vector<real> &eigen_values,
vector<vector<real>> &V, real h)
```

```
{
    int rows = F_tilde.size();
    int cols = F_tilde[0].size();

    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < cols; col++)
        {
            V[row][col] = (F_tilde[row][col])/((eigen_values[row] + eigen_values[col])+(1000.0*h*h));
        }
    }
}

// Function to save a vector<double> to a file of given name
void write_to_file(vector<real> &u, string str)
{
    ofstream file;
    // Open the file
    file.open(str);

    // If the file is NOT opened
    if( !file )
    {
        // Error Message if the file couldn't be opened
        cerr << "Error: file could not be opened" << endl;
        exit(1);
    }

    cout<<"Output file "<< str <<" is opened."<<endl;

    // Writing the vector values to the file in scientific notation
    for(int i=0;i<u.size();i++)
    {
        file<<u[i]<<scientific<<endl;
    }

    // Closing the file
    file.close();

    cout<<"Output file "<< str <<" is closed."<<endl;
    cout<<endl;
}

// Function to save a vector<vector <double>> (MATRIX) to a file of given name
void write_to_file(vector<vector <real> > &u, string str)
{
    ofstream file;
    // Open the file
    file.open(str);

    // If the file is NOT opened
    if( !file )
```

```
    {
        // Error Message if the file couldn't be opened
        cerr << "Error: file could not be opened" << endl;
        exit(1);
    }

    cout<<"Output file "<< str <<" is opened."<<endl;

    int rows = u.size();
    int cols = u[0].size();

    // Writing the vector values to the file in scientific notation

    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < cols; col++)
        {
            file<<u[row][col]<<scientific<<",";
        }
        file<<endl;
    }

    // Closing the file
    file.close();

    cout<<"Output file "<< str <<" is closed."<<endl;
    cout<<endl;
}

// This function read the inputs from a file
vector<real> input_parameters(string file_name)
{
    // Vector to read the input parameters from a input file
    vector<real> input;
    string item_name;
    int i = 0;

    // Open the file
    ifstream file;
    file.open(file_name);

    // If the file is NOT opened
    if( !file )
    {
        // Error Message if the file couldn't be opened
        cerr << "Error: Input file could not be opened" << endl;
        exit(1);
    }

    cout<<"Input file "<<file_name<<" is opened."<<endl;

    string line;
```

```
    while (getline(file, line))
    {
        // Classifying a string as double if the first character is a numeric
        if(line[0]!= '/')
        {
            // To ensure that we are not reading white space characters
            if(isdigit(line[0]))
            {
                input.push_back(stod(line));
            }
        }
    }

    // Closing the input file
    file.close();

    cout<<"Input file "<<file_name<<" is closed."<<endl;
    cout<<endl;

    return input;
}

int main()
{
    real X0,Xl,Y0,Yl,h;
    int N;

    // Given the domain of the solution
    X0 = 0;
    Xl = 1;
    Y0 = 0;
    Yl = 1;
    vector<real> L2_error_Vector, max_error_vector,Values_of_N;

    // Reading the values of N from a file
    Values_of_N = input_parameters("Values_of_N.txt");
    for (int g= 0;g < Values_of_N.size() ;g++)
    {
        N = Values_of_N[g];
        h = 1.0/N;

        int rows, cols;
        rows = N-1;
        cols = N-1;

        vector<vector <real> > P(rows,vector<real> (cols,0.0));
        vector<real> x;
        vector<real> y;
        vector<vector <real> > RHS(rows,vector<real> (cols,0.0));
        vector<vector <real> > P_Transpose(rows,vector<real> (cols,0.0));
        vector<vector <real> > P_INV_Transpose(rows,vector<real> (cols,0.0));
        vector<vector <real> > P_INV(rows,vector<real> (cols,0.0));
```

```
vector<vector <real> > temp(rows,vector<real> (cols,0.0));
vector<vector <real> > F_tilde(rows,vector<real> (cols,0.0));
vector<vector <real> > temp1(rows,vector<real> (cols,0.0));
vector<vector <real> > V(rows,vector<real> (cols,0.0));
vector<vector <real> > U(rows,vector<real> (cols,0.0));
vector<vector <real> > U_analytical(rows,vector<real> (cols,0.0));
vector<real> eigen_values;
vector<vector <real>> U_extended(N,vector<real> (N,0.0));

// Generating the Grids
generate_1D_grid(x,X0,Xl,N);
generate_1D_grid(y,Y0,Yl,N);

// Computing the eigen values, eigen vectors and the inverse of the matrix of eigen vectors
compute_eigen_value_vector_inverse(P,P_INV,eigen_values,x,N,h);

// Calculating the Transpose of P such that A = P Lambda P^T
matrix_transpose(P,P_Transpose);

// Calculating the Transpose of P^(-1)
matrix_transpose(P_INV,P_INV_Transpose);

// Calculating the RHS of the given equation
compute_rhs(RHS,x,y,h);

// Applying the boundary conditions
for (int row = 0;row< rows; row++)
{
   RHS[row][0] = RHS[row][0] - sin(10.0*pi*x[row+1]);
   RHS[row][rows-1] = RHS[row][rows-1] - sin(10.0*pi*x[row+1]);
}

// Multiplying: P^(-1) RHS = temp
matrix_multiply(P_INV,RHS,temp);

// Multiplying: F_tilde = temp (P^(-1))^T
matrix_multiply(temp,P_INV_Transpose,F_tilde);

// Calculating V
compute_V(F_tilde, eigen_values,V,h);

// Multiplying: P V = temp1
matrix_multiply(P,V,temp1);

// Multiplying: temp1 P^T = U
matrix_multiply(temp1,P_Transpose,U);

vector<vector <real> > U_Numerical(rows+2,vector<real> (cols+2,0.0));
vector<vector <real> > U_Analytical(rows+2,vector<real> (cols+2,0.0));

// Saving the matrix U (Numerical Solution)
for (int row = 0; row < rows+2; row++)
```

```
{
   for (int col = 0; col < cols+2; col++)
   {
      // Using the boundary condition
      if (col == 0)
      {
         U_Numerical[row][col] = sin(10.0*pi*x[row]);
      }
      // Using the boundary condition
      if (col == rows +1)
      {
         U_Numerical[row][col] = sin(10.0*pi*x[row]);
      }
      // Copying the U previously computed
      if (row < rows && col < cols)
      {
         U_Numerical[row+1][col+1] = U[row][col];
      }
   }
}

// Computing the analytical solution
// Saving the matrix U (Analytical Solution)
for (int row = 0; row < rows+2; row++)
{
   for (int col = 0; col < cols+2; col++)
   {
      U_Analytical[row][col] = sin(10.0*pi*x[row])*cos(10.0*pi*y[col]);
   }
}


// Writing the Numerical Solution to the file
write_to_file(U_Numerical,"Q_3_Numerical_Solution_"+to_string(N)+"_.csv");

// Writing the analytical solution
write_to_file(U_Analytical,"Q_3_Analytical_Solution_"+to_string(N)+"_.csv");

// Write x to a file
write_to_file(x,"Q_3_x_grid_"+to_string(N)+"_.csv");

// Write y to a file
write_to_file(y,"Q_3_y_grid_"+to_string(N)+"_.csv");


// Write a Output file showing values of x,y, Numerical Solution and Analytical Solution
ofstream File("Q_3_Output_"+to_string(N)+"_.dat", ios::out) ;
File.flags( ios::dec | ios::scientific );
File.precision(16) ;
if(!File) {cerr<< "Error: Output file couldnot be opened.\n";}

real Max_Error, L2_Error;
```

```
      Max_Error = L2_Error = 0.0 ;

      File << "TITLE = Flow" << endl << "VARIABLES = X, Y, u, Exact " << endl;
      File << "Zone T = psi I = " << N+1 << " J = " << N+1 << endl ;
      int i,j;

      for(i = 0 ; i < N-1 ; i++) {
         for(j = 0 ; j < N-1 ; j++) {
            if( fabs(U_Numerical[i][j] - U_Analytical[i][j] ) > Max_Error)
            {
               Max_Error = fabs( U_Numerical[i][j] - U_Analytical[i][j] );
            }
            L2_Error += (U_Numerical[i][j] - U_Analytical[i][j])*(U_Numerical[i][j] -
U_Analytical[i][j] )/ ( ( N+1.0 )*(N+1.0) ) ;

            File << x[i] << "\t" << y[j] << "\t" << U_Numerical[i][j] << "\t" << U_Analytical[i][j] <<
endl ;
         }
      }


      L2_Error = sqrt(L2_Error) ;
      File.close() ;
      // Printing the value of N
      cout<<"For N = "<<N<<endl;

      // Printing the value of L2 Error and Maximum Error
      cout << "\n L2 : " << L2_Error << "\t Max : " << Max_Error <<  endl ;
      L2_error_Vector.push_back(L2_Error);
      max_error_vector.push_back(Max_Error);

         }

         // Writing the L2 Errors in a file
   write_to_file(L2_error_Vector,"Q_3_L2_Error.csv");

   // Writing the Max Errors in a file
         write_to_file(max_error_vector,"Q_3_max_error.csv");
}
```

Q_3_Post_Processing

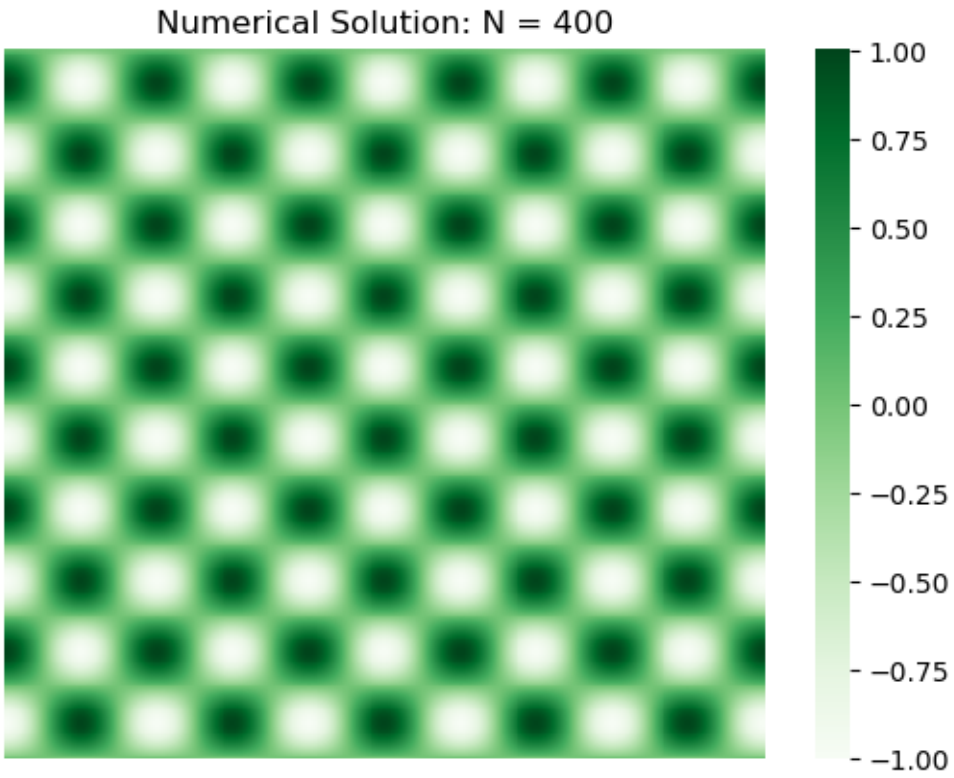
February 27, 2023

[1]:
```
import numpy as np;
import matplotlib.pyplot as plt;
import seaborn as sns;
import pandas as pd;
```

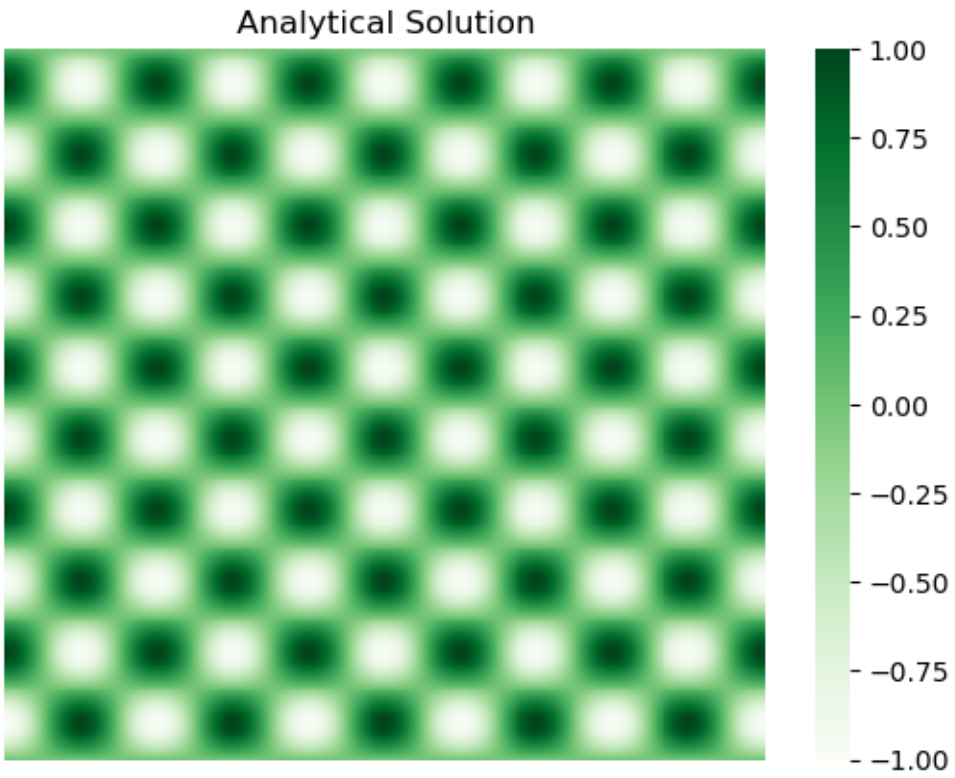
[2]:
```
Numerical = pd.read_csv("Q_3_Numerical_Solution_400_.csv").to_numpy()
```

[3]:
```
Analytical = pd.read_csv("Q_3_Analytical_Solution_400_.csv").to_numpy()
```

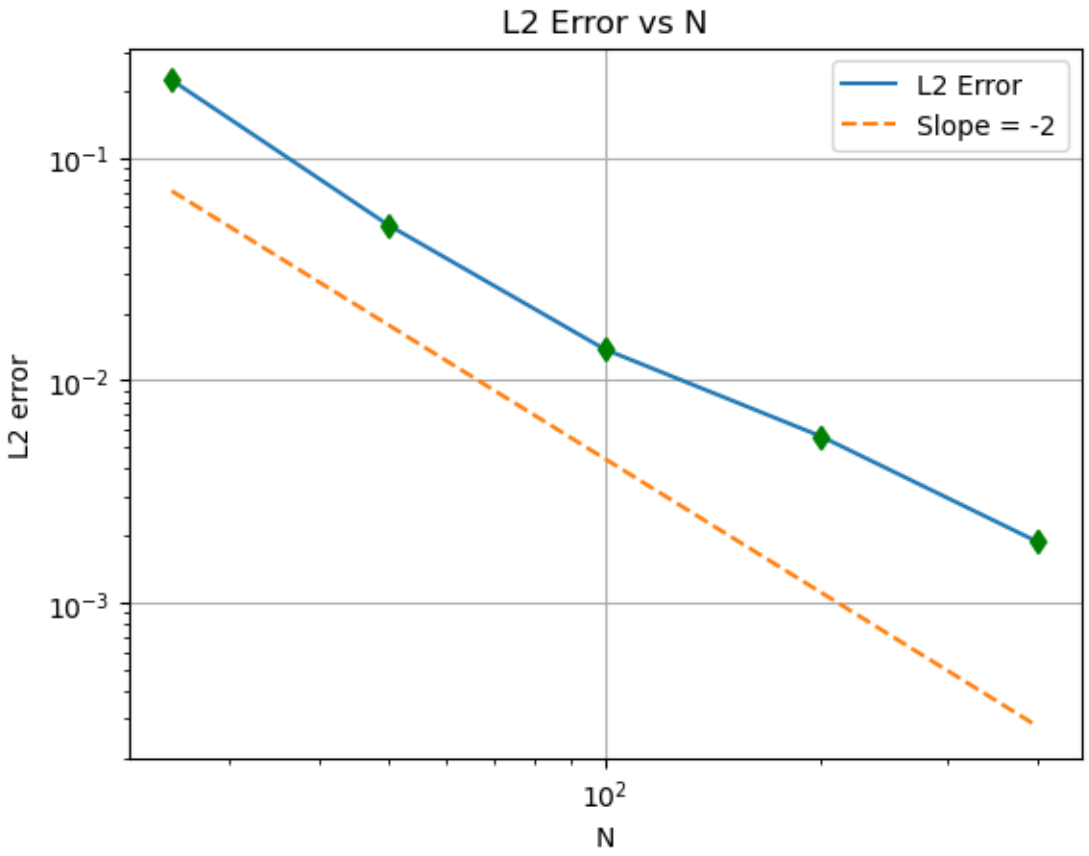
[4]:
```
ax = sns.heatmap(Numerical,cmap='Greens')
x_ticks = np.linspace(0,1,10)
y_ticks = np.linspace(0,1,10)
plt.xticks([],[])
plt.yticks([],[])
plt.title("Numerical Solution: N = 400")
plt.show()
```



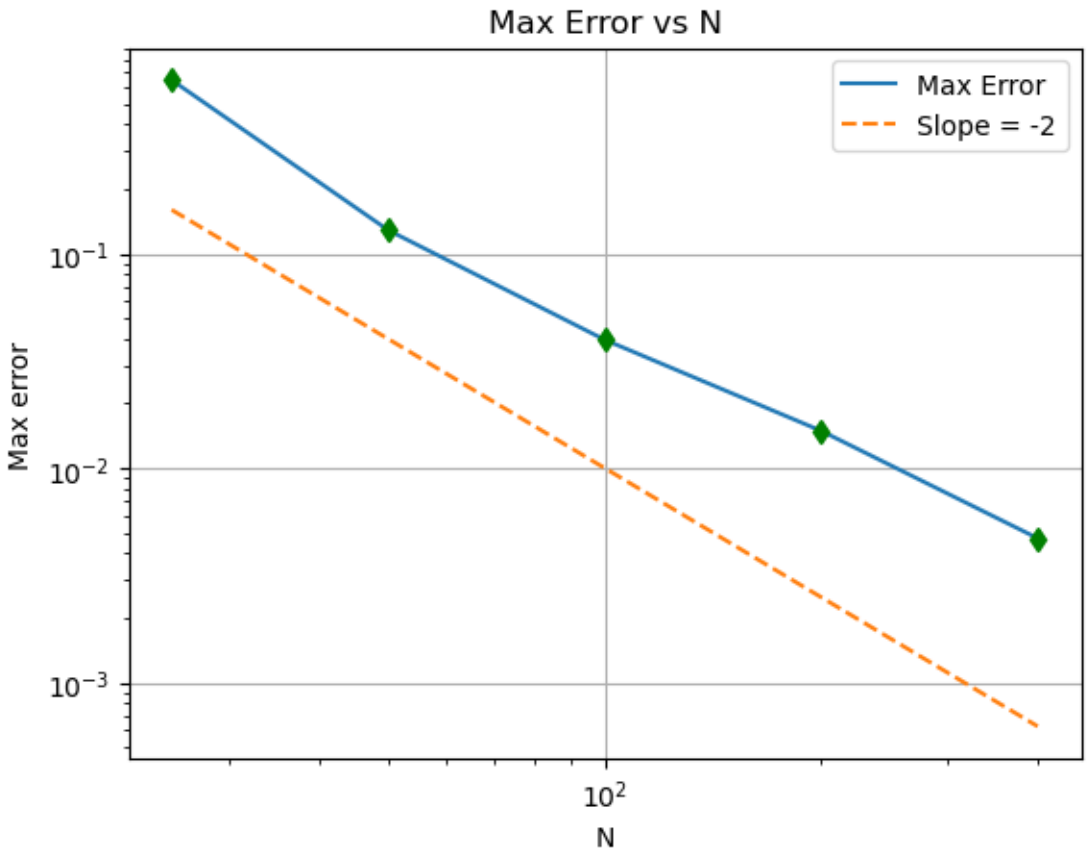
[5]:
```
ax = sns.heatmap(Analytical,cmap='Greens')
x_ticks = np.linspace(0,1,10)
y_ticks = np.linspace(0,1,10)
plt.xticks([],[])
plt.yticks([],[])
plt.title("Analytical Solution")
plt.show()
```



[6]:
```
fig = plt.figure()
N_num = [25.0,50.0,100.0,200.0,400.0];
L2_Error = pd.read_csv("Q_3_L2_Error.csv", delimiter = ",",header=None).
 ↪to_numpy()
plt.loglog(N_num,L2_Error)
plt.plot(N_num,(0.15*np.array(N_num))**(-2),"--")
plt.plot(N_num,L2_Error,"gd")
plt.legend(["L2 Error","Slope = -2"])
plt.xlabel("N")
plt.ylabel("L2 error")
#plt.xticks(ticks=N_num,labels=N_s)
plt.grid()
plt.title("L2 Error vs N")
plt.show()
fig.savefig("Q_1_a_L2_Error_vs_N.png",dpi = 500, bbox_inches="tight")
```



[7]:
```
fig = plt.figure()
N_num = [25.0,50.0,100.0,200.0,400.0];
Max_Error = pd.read_csv("Q_3_max_error.csv", delimiter = ",",header=None).
 ↪to_numpy()
plt.loglog(N_num,Max_Error)
plt.plot(N_num,(0.1*np.array(N_num))**(-2),"--")
plt.plot(N_num,Max_Error,"gd")
plt.legend(["Max Error","Slope = -2"])
plt.xlabel("N")
plt.ylabel("Max error")
#plt.xticks(ticks=N_num,labels=N_s)
plt.grid()
plt.title("Max Error vs N")
plt.show()
fig.savefig("Q_1_a_Max_Error_vs_N.png",dpi = 500, bbox_inches="tight")
```



1 So, the rate of convergence for both L2 Error and Max Error is approximately -2. As, it is almost parallel to a line with slope -2.