2D Laplace equation using Finite Difference Method
Anuradha Agarwal

1 Introduction

Laplace's equation is an elliptical second order partial differential equation. We have the following Laplace equation which is used to describe the electric potential u; on $0 < x < 1$ and $0 < y < 1$. We see Laplace's equation in many areas of science.

$$\boxed{\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \text{ where } 0 < \text{x} < 1 \text{ and } 0 < \text{y} < 1}$$

With the following boundary conditions.

$$\begin{array}{llll} y = 0; & x \in (0,1); & u(x,0) = f_1(x) \\ y = 1; & x \in (0,1); & u(x,1) = f_2(x) \\ x = 0; & y \in (0,1); & u(0,y) = g_1(y) \\ x = 1; & y \in (0,1); & u(1,y) = g_2(y) \end{array}$$

The analytical solution is given as:

$$u(x,y) = e^{\pi x}\cos(\pi y)$$

with the following boundary conditions:

$$\begin{array}{llll} y = 0; & x \in (0,1); & u(x,0) = e^{\pi x} \\ y = 1; & x \in (0,1); & u(x,1) = -e^{\pi x} \\ x = 0; & y \in (0,1); & u(0,y) = cos(\pi y) \\ x = 1; & y \in (0,1); & u(1,y) = e^{\pi}\cos(\pi y) \end{array}$$

In this project we will find the numerical solution to the Partial Differential Equation using the Finite Difference Method (FDM). According to Wikipedia, in numerical analysis, finite - difference methods are a class of numerical techniques for solving differential equations by approximating derivatives with finite differences. Here the spatial domain and the time interval are broken down into a number of sections, which is also known as discretization, and the values at those points or coordinates is approximated by solving algebraic equations containing finite differences and values from the points around them. This system of linear equations is solved using the Conjugate Gradient method. According to Wikipedia, the conjugate gradient method is an algorithm for the numerical solution of particular system of linear equations. This is an iterative algorithm that can handle huge matrices. This method is often used when there are sparse matrices which we get to see a lot in the solving of partial differential equations. The Finite Difference Method solves the equation using a grid. The solution is found for the domain grid points using the boundary grid points. The second derivatives at the coordinates can be approximated as

$$\left.\frac{\partial^2 U}{dx^2}\right|_{i,j} \approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{(\Delta x)^2} \quad \text{and} \quad \left.\frac{\partial^2 U}{dy^2}\right|_{i,j} \approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{(\Delta y)^2}$$

2 Code implementation

This section of the project describes the code implementation for the numerical solution of Laplace's equation. The code uses headers `stdio.h, stdlib.h` and `math.h`; and defines π and e. This code solves the numerical solution of Laplace's equation using the Finite Difference Method (FDM). This method converts the PDE into a set of linear algebraic equations. This code solves this system of equations using the Conjugate Gradient (CG) Method. The code writes the approximation matrix into an output file.

In the main we have n and m as the number of steps in x and y directions respectively; hx and hy as step sizes. The domain is of the size $[0,1] * [0,1]$ In this project, $hx = hy$ for the sake of simplicity. tol is the tolerance; $max1$ is the maximum error; W is the matrix with the exact solution (2 - Dimensional); U is the matrix with the approximate solution (2 - Dimensional); F is the error matrix (1 - Dimensional) - to calculate the error. The tolerance is set to $1e-9$ and the maximum number of iterations is set to $1e9$. The function `LAPLACEWCG` gives the approximate solution U; the function `Real_function` gives the analytical solution W; the function `error_matrix` takes matrix U and W to find the error. L - infinity norm is used to find the error.

The code is arranged into the following functions.

- `f1(x)`: This is a C - programming function which outputs the left boundary function. Both the input and output are of the type double. This function is called in the `BDYVAL` function to find the solution of the points on the boundary.
- `f2(x)`: This is a C - programming function which outputs the right boundary function. Both the input and output are of the type double. This function is called in the `BDYVAL` function to find the solution of the points on the boundary.
- `g1(x)`: This is a C - programming function which outputs the upper boundary function. Both the input and output are of the type double. This function is called in the `BDYVAL` function to find the solution of the points on the boundary.
- `g2(x)`: This is a C - programming function which outputs the lower boundary function. Both the input and output are of the type double. This function is called in the `BDYVAL` function to find the solution of the points on the boundary.
- `BDVYAL`: This is a C - programming function which outputs the values at the boundary using the boundary function above. The inputs of the function are option, which is of integer type, and w, which is the parameter at which we want the function to be evaluated and is of the type double. The function returns the value of the function at w with respect to the function. Option 1 evaluates $f_2(w)$; option 2 evaluates $f_1(w)$; option 3 evaluates $g_1(w)$ and option 4 evaluates $g_2(w)$.
- `Real_function`: This is a C - programming function which is of the type void. This function takes in n, m which are the size of the matrix and type integer, hx and hy which are are the step sizes as inputs of type double. It records a matrix W which is the matrix of the exact or analytical solution of size $[n]x[m]$. As mentioned previously the analytical solution is $u(x,y) = e^{\pi x}\cos(\pi x)$.
- `ERROR_METRIC`: This is a C - programming function which is of the type double. This function takes in n, m which are the number of rows and columns of the matrix; a vector A which is of size $m * n$, which is denoted as N and the option for different types of norms as the inputs.

The output of this function is a double depicting the norm of the vector corresponding to the option. This function essentially finds the norm of the vector A of size N ($m * n$ in our case) provided by the input array. This function is used by `LAPLACEWCG`. Here, option 1 returns L1 norm as the Taxi cab norm, option 2 returns L2 norm as the euclidean norm and option 3 returns L - inf norm as the infinity norm. L1 norm is the sum of the absolute values of the vector. L2 norm is the square root of the sum of squares of the components of the vector. L infinity norm is the maximum absolute value of the vector.

- `error_matrix`: This is a C - programming function which is of the type void. This function finds the difference between the analytical solution and the approximation. After finding the difference, this function transforms the matrix which is of size $[n]*[m]$ to a vector of size $n*m$. Inside this function, another function `ERROR_METRIC` is called which finds the Norms of the matrix, with three options : L1 norm, L2 norm, L inf norm. Here the input parameters are n and m which are the size of the matrix, a double matrix U which is the analytical solution, a double matrix W which is the approximate solution. This matrix begins with allocating space for a matrix called G of size $[n] * [m]$, which is the difference of the approximate matrix and the analytical solution.

- `CGUPDATE`: This is a C - programming function which is of the type void. This procedure is used by `LAPLACEWG` to update the iterative solution of the PDE using the CG method. This function takes n, m which are the number of rows and columns of the matrices; 4 matrices of the size $[n] * [m]$ and updates them. This function begins with allocating space for two matrices called RK andPK which are used to update the required matrices.

- `LAPLACEWCG`: This is a C - programming function which is of the type void. This function takes in n, m which are the number of rows and columns, length of the domain a, b, the step size in both the x and y direction hx, hy; error tolerance tol; maximum number of iterations max1 as the inputs. This procedure applied the Finite Difference Method to the Laplace's equation and uses the method of Conjugate Gradient Method to find the problem's solution. Matrix U here is the required approximation. We begin with allocating space for all the matrices and vectors which are used in the function, then initializing all the vectors with either zeroes or ones accordingly. The vectors X and Y, which depict the coordinates, are calculated with respect to the step sizes. The approximation matrix U is initialized with the initial condition; the boundary values and the corner values of the matrix are calculated. `ERROR_METRIC` is called inside this function to find the error err. While the error is greater than the tolerance and the number of iterations don't exceed the maximum number of iterations, the Finite Difference Method is applied. For every iteration, `CGUPDATE` function updates all the matrices.

3 Results and Discussion

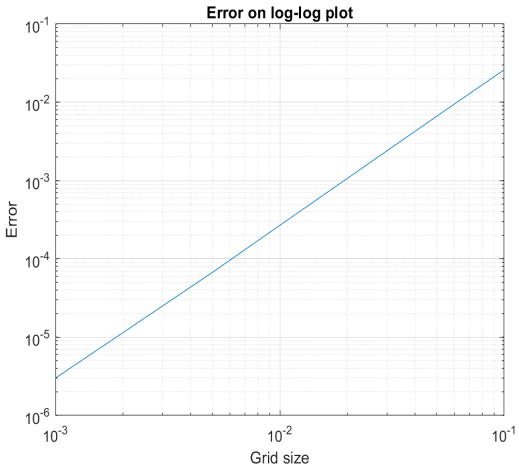
Error metrics for different grid sizes $h_x = h_y = h = \{0.1, 0.05, 0.01, 0.005, 0.001\}$. In the table below we see that as the value of the step size decreases (i.e. the grid gets finer with a larger number of steps), we see that the error is decreasing. This shows that the error is converging. The plot on the left shows the line fit of the errors for the respective step sizes. The equation of the line fit is

$$\boxed{1.9735 * h + 0.88348}$$

The slope here is 1.9735 which is approximately 2 (i.e. Order of the method is 2). L - infinity norm is used to find the errors.

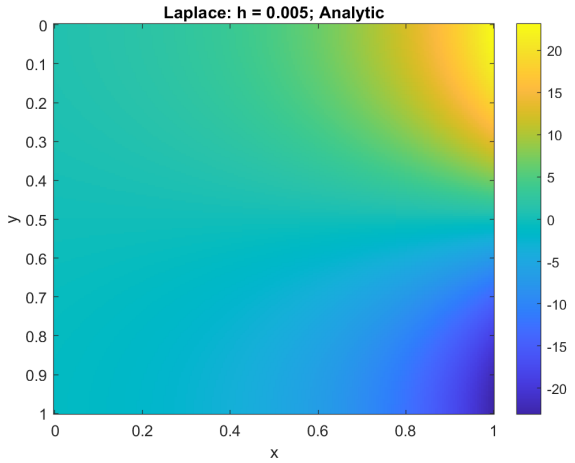
Step Size	Error
0.1	0.026026
0.05	0.006631
0.01	0.000268
0.005	0.000067
0.001	0.000003

(a) Table with the step sizes and their corresponding errors

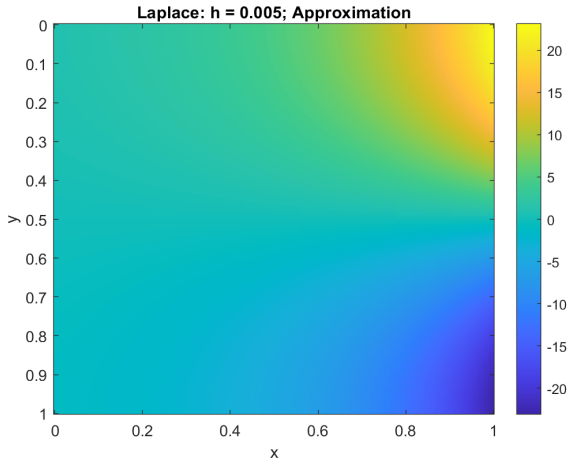


(b) Log log plot with error metrics

Below we can see the contour plot of the finest grid $h = 0.005$. We can see that the analytical solution on the left looks very similar to the analytical solution to the right. As you increase the n number of grid points or decrease the step size, the approximate solution gets closer to the analytical solution.



(a) Analytical solution



(b) Approximate solution

Below we see the table with the number of iterations for the tested h values. We see that the number of iterations increases as the value of h decreases.

h	Iterations
0.1	36
0.05	82
0.01	440
0.005	895
0.001	5136

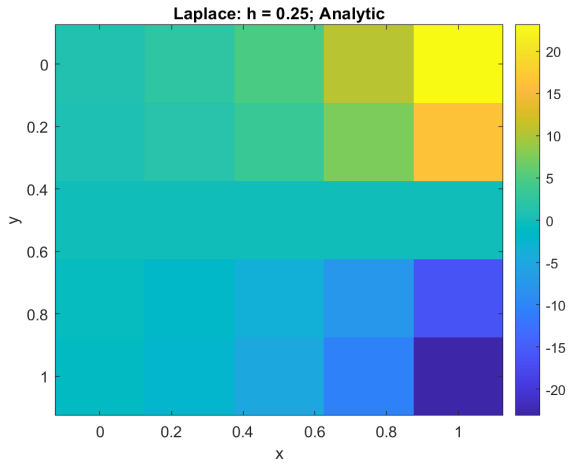
Figure 3: Iterations

4 Conclusion

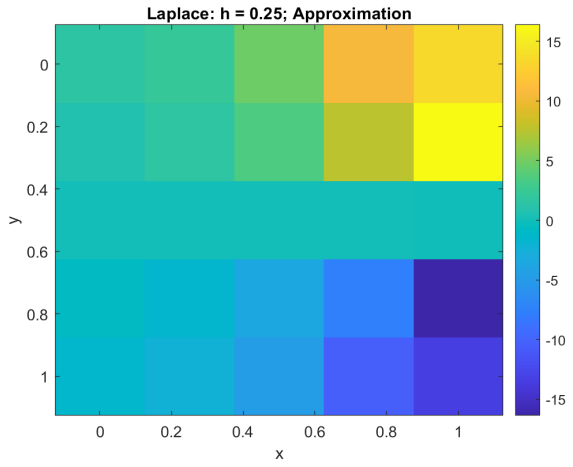
In conclusion, we see that the error converges at order 2, demonstrating the accuracy of the code done for the project. We see that as the number of steps increases, the error goes down even faster. For future projects we can calculate the Laplace's equation in 3D and accelerate the code to make it faster and more efficient.

5 Extra figures

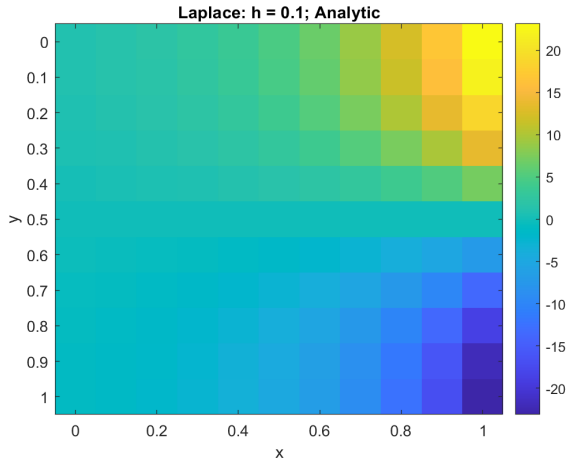
In the figures below we can see the comparison between the analytical solution and the approximate solution for different h sizes. We see that as the grid gets finer the analytical solution and the approximate solution look more similar to each other.



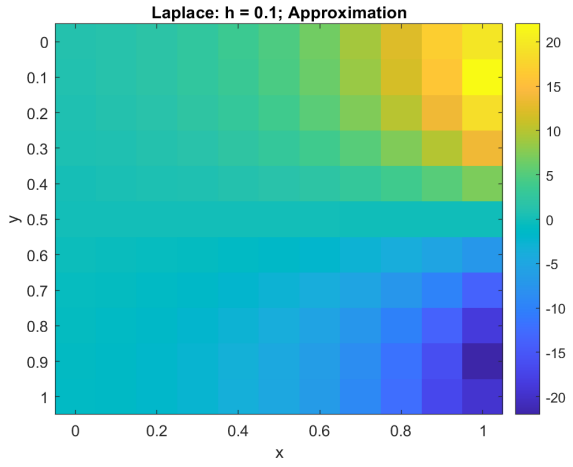
(a) Analytical solution; h = 0.25



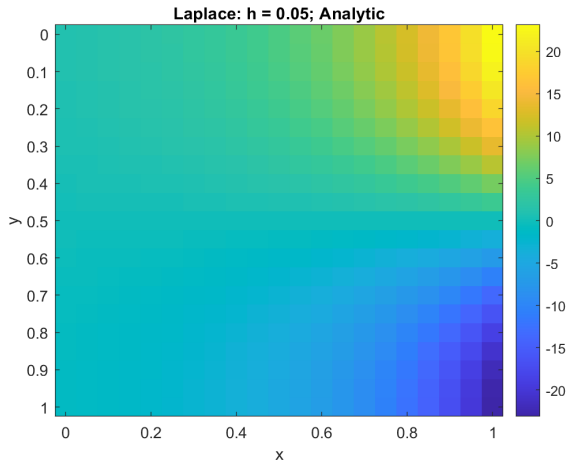
(b) Approximate solution; h = 0.25



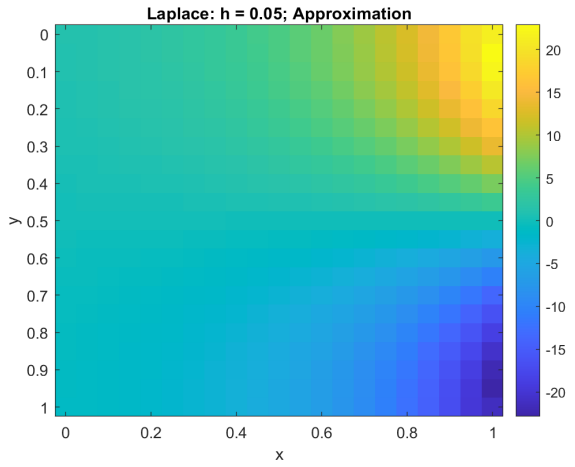
(a) Analytical solution; h = 0.1



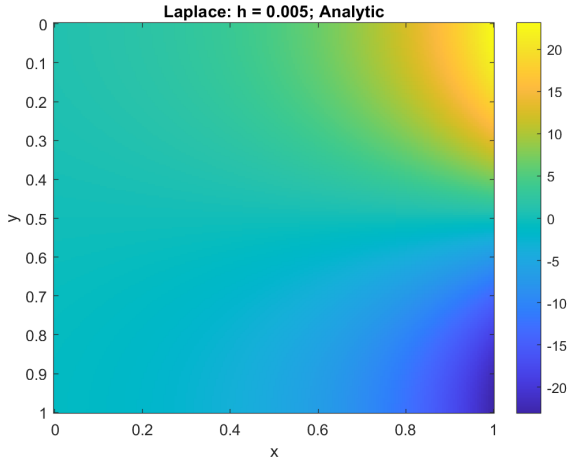
(b) Approximate solution; h = 0.1



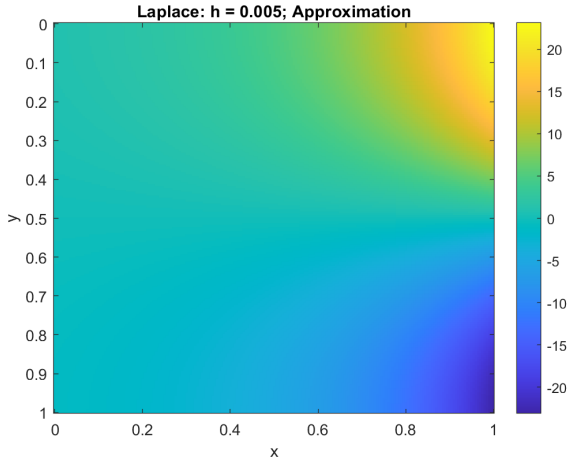
(a) Analytical solution; h = 0.05



(b) Approximate solution; h = 0.05



(a) Analytical solution; ; h = 0.005



(b) Approximate solution; h = 0.005