Solving the Poisson Equation in Two Dimensional Domain

Group 5: Alwin Anto, Judy Hao, Shan Jiang, Bowen Shi

December 10, 2024

Github Repo: https://github.com/UT-CSE-380-F2024/group-project-team-5

1 Introduction

The Poisson equation is a fundamental partial differential equation (PDE) that models many physical phenomena, including heat distribution, electrostatics, and fluid dynamics. Solving the Poisson equation accurately is crucial for understanding these phenomena in a variety of fields. The objective of this project is to solve the Poisson equation on a rectangular domain using the finite element method (FEM), validate it with an analytical solution, and evaluate the convergence behavior of the numerical solution.

2 Problem Formulation

Consider the Poisson equation on the rectangular domain $\Omega = (0,1) \times (0,1)$,

$$-\Delta u = f, \quad (x,y) \in \Omega.$$

with the boundary conditions

$$\begin{aligned} u &= g_1, \quad (x,y) \in \partial\Omega_0 \\ \frac{\partial u}{\partial \boldsymbol{n}} &= g_2, \quad (x,y) \in \partial\Omega_1 \end{aligned}$$

where $\partial\Omega_0 = \{0\} \times [0,1]$, and $\partial\Omega_1 = \partial\Omega - \partial\Omega_0$. We choose

$$f = 1 + n\cos(\pi y), \quad g_1(y) = 0, \quad g_2(x,y) = 0,$$

where $n \in \mathbb{R}$ is a fixed constant. The problems are formulated as the following:

- Given mesh sizes of $h = \frac{1}{2^n}$, perform triangular subdivisions (with equal-sized triangles). For the mesh size of $\frac{1}{3}$, we sketch the diagram. Design a piecewise linear C^0 finite element, write a program to solve the problem, and report the computational time. Visualize finite element solutions of mesh size.
- Find the analytical solution to the problem, and compute the error between the finite element solution and the analytical solution.

3 Solution Approach

3.1 Finite Element Solution and Algorithm

To derive the weak formulation, we first take the test function $v \in \mathbb{V} := \{v \in \mathbb{H}^1(\Omega) \mid v|_{\partial\Omega_0} = 0\}$, ultiply the left-hand side of equation (1.1) by v, integrate over Ω, and apply the Gauss-Green formula to obtain

$$-\int_{\Omega} \Delta u \cdot v=\int_{\Omega} \nabla u \cdot \nabla v-\int_{\partial \Omega} \frac{\partial u}{\partial \boldsymbol{n}} \cdot v=\int_{\Omega} \nabla u \cdot \nabla v$$

Thus, the variational form is

$$\int_{\Omega} \nabla u \cdot \nabla v=\int_{\Omega} f v$$

The weak solution $u \in \mathbb{V}$ satisfies the above variational form for any $v \in \mathbb{V}$. Similarly, we propose a corresponding functional minimization problem based on the principle of minimum potential energy:

$$J(u)=\min_{v \in \mathbb{V}} J(v)$$

where

$$J(v)=\frac{1}{2} \int_{\Omega}|\nabla v|^{2}-\int_{\Omega} f v$$

This problem is equivalent to the variational form of the original problem. Consider the triangular mesh as shown in Figures 1 (element diameter $\frac{1}{3}$) . For convenience, let h denote the element diameter in the mesh, N the number of nodes, and M the number of elements. Then, $N=\frac{1}{h^{2}}+\frac{2}{h}+1$, and $M=\frac{2}{h^{2}}$.

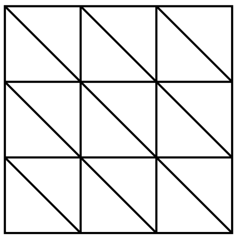


Figure 1: A unit square divided into 9 sub-squares, each subdivided into two triangles.

Next, we introduce the P_1 Lagrange finite element space as

$$\mathbb{V}_{h}=\left\{u \in C(\bar{\Omega}): \left.u\right|_{T_{i}} \in \mathbb{P}_{1}\left(T_{i}\right), \forall T_{i} \in \mathcal{T}_{h}(\Omega)\right\}$$

We then take the finite element trial function space and test function space as

$$\mathbb{V}_{h}\left(0 ; \Omega_{0}\right)=\left\{u \in \mathbb{V}_{h}: u\left(A_{i}\right)=0, \forall A_{i} \in \partial \Omega_{0}\right\}$$

For the variational form (1.2), we obtain the finite element problem

$$\left\{\begin{array}{l}\text { Find } u_{h} \in \mathbb{V}_{h}\left(0 ; \Omega_{0}\right), \text { such that } \\ a\left(u_{h}, v_{h}\right)=F\left(v_{h}\right), \quad \forall v_{h} \in \mathbb{V}_{h}\left(0 ; \Omega_{0}\right),\end{array}\right. \tag{1.3}$$

where

$$a(u, v)=\int_{\Omega} \nabla u \cdot \nabla v, \quad F(v)=\int_{\Omega} f v .$$

Let $\varphi_{i} \in \mathbb{V}_{h}$ satisfy

$$\varphi_{i}\left(A_{j}\right)=\delta_{i j}, \quad i=1,2, \ldots, N$$

Where A_j are the nodes of the mesh. Then, $\{\varphi_i\}_{i=1}^{N}$ forms a basis for $\mathbb{V}_h$, and $\{\varphi_i\}_{A_i \notin \partial\Omega_0}$ forms a basis for $\mathbb{V}_h\left(0;\Omega_0\right)$. Let

$$u_{h}=\sum_{j=1}^{N_{h}} u_{j} \varphi_{j}, \quad v_{h}=\sum_{i=1}^{N_{h}} v_{i} \varphi_{i}$$

where $N_h = \frac{1}{h^2} + \frac{1}{h}$. The discrete problem (1.3) is equivalent to

$$\sum_{j=1}^{N_h} a(\varphi_j, \varphi_i) u_j = F(\varphi_i), \quad i = 1, 2, \ldots, N_h$$

i.e.,

$$\boldsymbol{K} \boldsymbol{u}_h = \boldsymbol{f} \tag{1}$$

When computing the stiffness matrix $\boldsymbol{K}$ and load vector $\boldsymbol{f}$, we calculate the element stiffness matrix $\boldsymbol{K}^e$ and element load vector $\boldsymbol{f}^e$ by scanning the elements. The algorithm is as follows:

Algorithm 1 Algorithm for forming the stiffness matrix and load vector

```
Input: $\boldsymbol{K} = (k(i,j)) := 0$, $\boldsymbol{f} = (f(i)) := 0$
for $e = 1$ to $M$ do
    Compute element stiffness matrix $\boldsymbol{K}^e = (k^e(\alpha, \beta))$
    Compute element load vector $\boldsymbol{f}^e = (f^e(\alpha))$
    for each $\alpha, \beta$ do
        $k(en(\alpha, e), en(\beta, e)) = k(en(\alpha, e), en(\beta, e)) + k^e(\alpha, \beta)$
        $f(en(\alpha, e)) = f(en(\alpha, e)) + f^e(\alpha)$
    end for
end for
Output: $\boldsymbol{K}$, $\boldsymbol{f}$
```

where $en(\alpha, e)$ denotes the assigned number of triangular nodes (global). If the coordinates of the three vertices of element T_e are $\boldsymbol{A}_1^e = (x_1, y_1)^T$, $\boldsymbol{A}_2^e = (x_2, y_2)^T$, $\boldsymbol{A}_3^e = (x_3, y_3)^T$, then the affine transformation $\boldsymbol{x} = L_e(\hat{\boldsymbol{x}}) = \boldsymbol{A}_e \hat{\boldsymbol{x}} + \boldsymbol{a}_e$ maps the reference triangle T_s to T_e, where

$$\boldsymbol{A}_e = (\boldsymbol{A}_2^e - \boldsymbol{A}_1^e, \boldsymbol{A}_3^e - \boldsymbol{A}_1^e), \quad \boldsymbol{a}_e = \boldsymbol{A}_1^e.$$

The element stiffness matrix is

$$\boldsymbol{K}^e = \frac{1}{2 \det \boldsymbol{A}_e} \begin{pmatrix} y_2 - y_3 & x_2 - x_3 \\ y_3 - y_1 & x_3 - x_1 \\ y_1 - y_2 & x_1 - x_2 \end{pmatrix} \begin{pmatrix} y_2 - y_3 & y_3 - y_1 & y_1 - y_2 \\ x_2 - x_3 & x_3 - x_1 & x_1 - x_2 \end{pmatrix}$$

The element load vector is given by

$$\begin{aligned} \boldsymbol{f}_i^e &= \int_{T_e} f(\boldsymbol{x}) \varphi_i(\boldsymbol{x}) dT_e \\ &= |\det \boldsymbol{A}_e| \int_{T_s} f(\boldsymbol{x}) \varphi_i(\hat{\boldsymbol{x}}) dT_s, \quad i = 1, 2, 3. \end{aligned}$$

Also note that

$$\det \boldsymbol{A}_e = 2 |T_e| = h^2$$

Thus, we can calculate the stiffness matrix $\boldsymbol{K}$ and the load vector $\boldsymbol{f}$, and then solve equation (1) to obtain the finite element solution u_h.

We evaluate the load vector using three-point quadrature rule with second-order accuracy. For a reference triangle with vertices $\{(0,0), (0,1), (1,0)\}$, we use the quadrature points $\hat{\boldsymbol{x}}_q := \{(1/6, 1/6), (2/3, 1/6), (1/6, 2/3)\}$ with weights $w_q := \{1/6, 1/6, 1/6\}$. Thus, we have

$$\boldsymbol{f}_i^e \approx |\det \boldsymbol{A}_e| \sum_{q=1}^{3} w_q f(\boldsymbol{A}_e \hat{\boldsymbol{x}}_q + \boldsymbol{a}_e) \varphi_i(\hat{\boldsymbol{x}}_q).$$

3.2 Analytic Solution

Under the chosen conditions of f, g, h, the original problem becomes

$$\begin{cases} -u_{xx} - u_{yy} = 1 + n\cos(\pi y), & (x,y) \in \Omega \\ u(0,y) = 0, & y \in [0,1] \\ u_y(x,0) = u_y(x,1) = 0, & x \in [0,1] \\ u_x(1,y) = 0, & y \in [0,1] \end{cases}$$

Next, we will find the analytical solution of equation (1.5) using the method of separation of variables.
Consider a non-zero solution of the form of separation of variables

$$u(x,y) = X(x)Y(y)$$

Substituting it into the homogeneous equation

$$u_{xx} + u_{yy} = 0, \quad (x,y) \in \Omega,$$

we obtain

$$X''(x)Y(y) + X(x)Y''(y) = 0, \quad (x,y) \in \Omega$$

which implies

$$\frac{X''(x)}{X(x)} = -\frac{Y''(y)}{Y(y)}.$$

In the above equation, the left side is a function of x, and the right side is a function of y; hence they can only be equal to a constant, denoted by λ. Thus, we have

$$\begin{aligned} X''(x) - \lambda X(x) = 0, \quad x \in (0,1) \\ Y''(y) + \lambda Y(y) = 0, \quad y \in (0,1) \end{aligned}$$

Substituting $u(x,y)$ into the homogeneous boundary conditions

$$u_y(x,0) = u_y(x,1) = 0, \quad x \in [0,1]$$

gives us

$$X(x)Y'(0) = X(x)Y'(1) = 0, \quad x \in [0,1]$$

Since $u(x,y) \not\equiv 0$, we have $X(x) \not\equiv 0$, leading to

$$Y'(0) = Y'(1) = 0.$$

Thus, we obtain the Sturm-Liouville boundary value problem

$$\begin{cases} Y''(y) + \lambda Y(y) = 0, \quad y \in (0,1) \\ Y'(0) = Y'(1) = 0 \end{cases}$$

The characteristic values of this problem are given by

$$\lambda_k = (k\pi)^2, \quad k = 0, 1, \cdots,$$

with the corresponding eigenfunctions being

$$Y_k(y) = \cos k\pi y, \quad k = 0, 1, \cdots$$

Returning to the original problem, we express the solution $u(x,y)$ in terms of the eigenfunction series $\{\cos k\pi y\}(k = 0, 1, \cdots)$:

$$u(x,y) = \sum_{k=0}^{\infty} X_k(x)\cos(k\pi y).$$

Substituting these expressions into the original problem and using the completeness of the eigenfunction series $\{\cos k\pi y\}(k = 0, 1, \cdots)$, we obtain that $X_k(x)$ satisfies the following ordinary differential equations:

$$\begin{cases} X_0''(x) = -1, \quad x \in (0,1), \\ X_0(0) = 0, X_0'(1) = 0. \end{cases}$$
$$\begin{cases} X_1''(x) - \pi^2 X_1(x) = -n, \quad x \in (0,1) \\ X_1(0) = 0, X_1'(1) = 0. \end{cases}$$
$$X_k = 0, \quad k \geq 2.$$

Solving these ODEs gives us

$$\begin{aligned} X_0(x) &= -\frac{x^2}{2} + x \\ X_1(x) &= C_1 e^{\pi x} + C_2 e^{-\pi x} + \frac{n}{\pi^2}, \end{aligned}$$

where

$$C_1 = -\frac{ne^{-2\pi}}{\pi^2(e^{-2\pi}+1)}, \quad C_2 = -\frac{n}{\pi^2(e^{-2\pi}+1)}$$

Thus, the analytical solution of equation (1.5) is

$$\begin{aligned} u(x,y) &= \sum_{k=0}^{1} X_k(x)\cos(k\pi y) \\ &= -\frac{x^2}{2} + x + (C_1 e^{\pi x} + C_2 e^{-\pi x} + \frac{n}{\pi^2})\cos(\pi y). \end{aligned}$$

Restricting the analytical solution (1.6) to the grid points of the corresponding triangular mesh yields a vector u, which can be compared with the computed solution u_h. The L^2 error of the finite element solution can be approximated by calculating $\|u - u_h\|_{L^2}$.

4 Implementation

4.1 Instructions for Building and Running

User can use the following commands to build and run the code:

```
$ module load hdf5
$ module load boost
$ mkdir build
$ cd build
$ cmake ..      #to use the cmakelist.txt to generate the make files and some directories
$ make           # compile , link etc
$ ctest            # if you want to run the test suite
$ bin/MeshSolver       # running the program in build. There will be a results folder created i
$ bin/MeshSolver -v   # if you want to run the convergence test
```

Note: NumPy version 1.24.4 and h5py version 3.12.1 are used in this project.

4.2 Code Organization

Below is an overview of the code organization:

- **Main Program:**
  - `main.cpp`: Reads inputs, initializes data structures, calls functions for mesh generation, stiffness matrix assembly, and solving the system, and outputs results.
- **Core Modules:**
  - `GenerateMesh.cpp`: Contains functions to generate a triangular mesh for the domain, given the specified mesh size.
  - `StiffnessMatrix.cpp`: Implements the computation of the stiffness matrix and load vector based on the finite element method.
  - `file_io.cpp`: Handles parsing of input configuration files to extract simulation parameters such as mesh size, n, maximum iterations, and solver tolerance; saves computed results (node coordinates and solution vector) to an HDF5 file.
  - `convergence_analysis.cpp`: Implements functionality for analyzing the convergence of the finite element solution by comparing results across varying mesh resolutions.
- **Python Visualization:**
  - `plot_mesh.py`: A Python script that reads the HDF5 output file, processes the node coordinates and solution data, and generates a 3D surface plot of the finite element solution. It dynamically names the output plot file based on N and n.
- **Build System:**
  - `CMakeLists.txt`: A CMake configuration file to handle project compilation. It ensures proper linking of external libraries like Eigen and HDF5, sets compiler flags, and organizes the build output into a dedicated `bin` directory.
- **Third-party Libraries:**
  - **Eigen Library:** Used for efficient linear algebra operations, including matrix assembly and sparse matrix representations.
  - **HDF5 Library:** Employed to manage input/output operations for storing and accessing datasets efficiently.

4.3 Test-suite

We use the Boost Test framework for unit and regression testing. The testing setup comprises three key components:

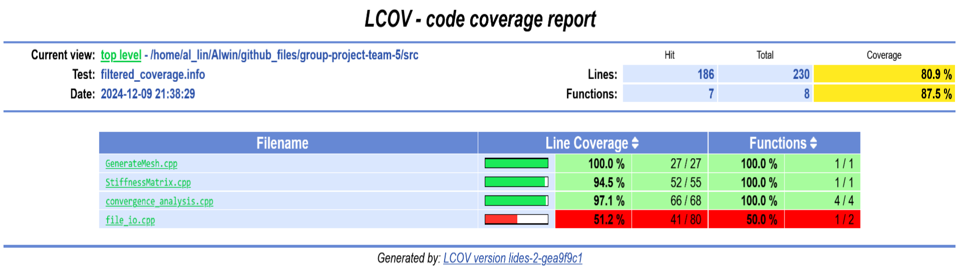
- `test_main.cpp`: Involves two main checks. One is making sure parsed values are as expected. The other check calculates the L^2 norm error of fem solution against the analytical function. Then, this error is compared to the L^2 norm error expected for that same set of inputs and checked if values are within 1%. Any logical errors in the important part of the code would be picked up here.

- `test_computeStiffnessLoad.cpp`: Validates the construction of stiffness matrix by comparing the structure for a basic case.

- `test_convergence_analysis.cpp`: This part makes sure convergence codes function properly by making basic checks on the norm values and whether they are computed correctly.

4.4 Code Coverage

Figure 2 shows the code coverage which is above 75% .The code coverage implementation in TACC was failing due to a version mismatch. However was able to produce the report locally.



LCOV - code coverage report

Current view: top level - /home/al_lin/Alwin/github_files/group-project-team-5/src
Test: filtered_coverage.info
Date: 2024-12-09 21:38:29

	Hit	Total	Coverage
Lines:	186	230	80.9 %
Functions:	7	8	87.5 %

Filename	Line Coverage			Functions	
GenerateMesh.cpp		100.0 %	27 / 27	100.0 %	1 / 1
StiffnessMatrix.cpp		94.5 %	52 / 55	100.0 %	1 / 1
convergence_analysis.cpp		97.1 %	66 / 68	100.0 %	4 / 4
file_io.cpp		51.2 %	41 / 80	50.0 %	1 / 2

Generated by: LCOV version lides-2-gea9f9c1

Figure 2: Code coverage snaphot.

4.5 Valgrind Output

With help of Valgrind, we are able to find out whether the program have memory leak or not. Below is the the clean.out:

```
==110== Memcheck, a memory error detector
==110== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==110== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==110== Command: ./MeshSolver
==110==
Parsed max_iter: 1000
Parsed tolerance: 1e-07
Mesh divisions read from input file: 30
n read from input file: 0
Results successfully saved to ../results/mesh_results.h5
Visualization completed successfully!
==110==
==110== Process terminating with default action of signal 27 (SIGPROF)
==110==    at 0x5052C20: open_nocancel (open64_nocancel.c:45)
==110==    by 0x505D537: write_gmon (gmon.c:370)
==110==    by 0x505DBDF: _mcleanup (gmon.c:444)
==110==    by 0x4FC0AB7: cxa_finalize (cxa_finalize.c:83)
==110==    by 0x10B937: ??? (in /app/build/MeshSolver)
==110==    by 0x400EC77: _dl_fini (dl-fini.c:138)
==110==    by 0x4FC047B: __run_exit_handlers (exit.c:108)
==110==    by 0x4FC060B: exit (exit.c:139)
==110==    by 0x4FAAE13: (below main) (libc-start.c:342)
==110==
==110== HEAP SUMMARY:
==110==     in use at exit: 39,344 bytes in 1 blocks
==110==   total heap usage: 2,636 allocs, 2,635 frees, 8,646,273 bytes allocated
==110==
==110== LEAK SUMMARY:
==110==    definitely lost: 0 bytes in 0 blocks
==110==    indirectly lost: 0 bytes in 0 blocks
==110==      possibly lost: 0 bytes in 0 blocks
==110==    still reachable: 39,344 bytes in 1 blocks
==110==         suppressed: 0 bytes in 0 blocks
==110== Reachable blocks (those to which a pointer was found) are not shown.
==110== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```

```
==110==
==110== For lists of detected and suppressed errors, rerun with: -s
==110== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note: Users are able to run Valgrind inside the docker image.

4.6 Docker

Inside the main directory, we wrote the dockerfile to ensure the cross-platform consistency. We push it as a public image that supports both arm64 and amd64 architecture. The detailed instructions to pull the image and use it are listed inside the README.md file.

4.7 GitHub Actions and Instructions for Docker/Apptainer Build

Furthermore, we make CI with Github Action. The steps inside the yaml file are

- Pull the image and run the container
- Build the object file
- Run the ctest

In the main page of Github repo, we display a workflow status badge that pass the test.

5 Code Verification

5.1 Theoretical Convergence Rate

The theoretical convergence rate of the L_2-norm for the finite element method using P_1 elements is $O(h^2)$, where h denotes the mesh size or the largest triangle's diameter:

$$\|u_{\text{ref}} - u_h\|_{L^2} \leq Ch^2 \|u_{\text{ref}}\|_{H^2(\Omega)},$$

assuming $u_{\text{ref}} \in H^2(\Omega)$, where C is a constant independent of h.

5.2 Discrete Approximation of L_2 Error

To verify the accuracy of our finite element method implementation, we compute the L_2 norm of the error between the numerical solution (u_h) and the analytical solution (u_{ref}). Let denote n_1, n_2, n_3 the nodes of a triangular element, and x_i, y_i represent the coordinates of the i-th node ($i = 1, 2, 3$).The L_2 norm is defined as:

$$L_2 = \sqrt{\sum_{e=1}^{N_{\text{elements}}} L_{2,\text{local}}(e)},$$

where the local contribution is given by:

$$L_{2,\text{local}} = \frac{\det \boldsymbol{A}_e}{3} \left[(u_h(n_1) - u_{\text{ref}}(n_1))^2 + (u_h(n_2) - u_{\text{ref}}(n_2))^2 + (u_h(n_3) - u_{\text{ref}}(n_3))^2 \right].$$

Here, $\det \boldsymbol{A}_e$ is the determinant of the Jacobian matrix representing the element area, computed as:

$$\det \boldsymbol{A}_e = \frac{1}{2} \left| (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1) \right|.$$

This metric evaluates the error of u_h against u_{ref} over the mesh. Smaller L_2 norm values indicate better accuracy and demonstrate convergence as the mesh is refined, thereby validating our implementation.

5.3 Verification Result

Figure 3 shows the verification result for our problem.

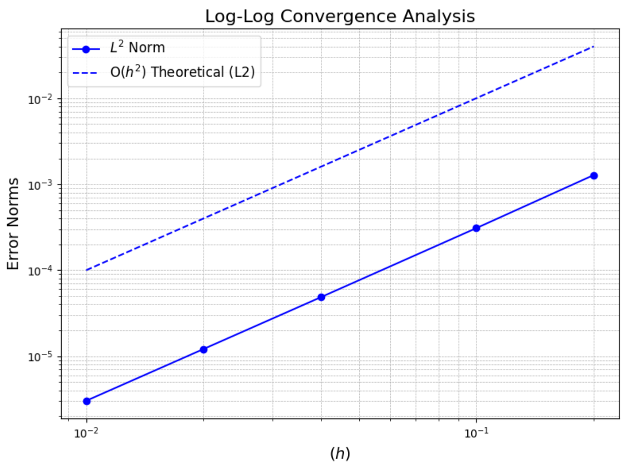


Figure 3: Convergence plot of the L^2-norm.

We observe that our convergence rate matches the theoretical convergence rate.

6 Results

6.1 Experiment Setting

The project is designed to operate in two distinct modes: **production mode** and **verification mode**. The workflow is controlled by the command-line arguments passed to the program.

- **Production Mode:** By default, the program runs in production mode. In this mode:
  - The finite element method (FEM) solution is computed for a given mesh division and a factor n in the right-hand side of the Poisson equation.
  - The program generates the stiffness matrix and solves the linear system using the Conjugate Gradient (CG) solver.
  - Results, including the node coordinates and solution vector, are saved to an HDF5 file (`mesh_results.h5`).
  - A Python script (`plot_mesh.py`) is invoked to visualize the solution.

  Detailed input and output files for production mode are specified in Section 6.2.

- **Verification Mode:** Verification mode is enabled by passing the `--verification` or `-v` flag when executing the program. In this mode:
  - A convergence analysis is performed by solving the problem for multiple mesh divisions (e.g., $\{5, 10, 25, 50, 100\}$).
  - Errors are logged into a CSV file (`convergence_log.csv`).
  - A Python script (`plot_convergence.py`) generates a log-log plot to evaluate the convergence rate.

Main Workflow: The primary steps in the program are as follows:

1. Parse input parameters from `inputs.txt`.

2. If verification mode is enabled, run convergence analysis and generate the convergence plot.
3. Otherwise, generate the mesh, compute the stiffness matrix, and solve the FEM problem using the CG solver.
4. Save the results in an HDF5 file and visualize them using Python scripts.

6.2 Input and Output in Production Mode

- **Input File:** `inputs.txt` is the file used to specify the problem parameters. An example is

```
[mesh]
mesh_division = 30    # Number of divisions along one side of the domain

[solver]
n = 0                 # Parameter for RHS (f = 1 + n cos(pi*y))
notify_freq = 10      # Frequency of notifications
max_iter = 1000       # Maximum iterations for CG solver
tolerance = 1e-7      # Solver tolerance
verbosity = 1         # Verbosity level (0 for low verbosity, >0 for high verbosity)
```

- **Output Files:**
  - Standard output examples
    High verbosity:

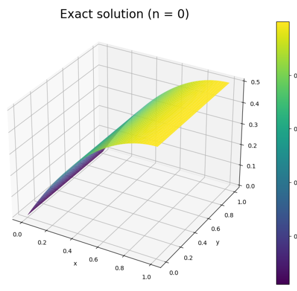
    ```
    $ bin/MeshSolver
    Values read from input file -
    Mesh divisions : 30
    Parameter n of f=1+ncos(pi*y) : 0
    Maximum iterations used by CG solver: 1000
    Tolerance used by the solver: 1e-07
    Results successfully saved to results/mesh_results.h5
    Visualization completed successfully!
    ```

    Low verbosity:

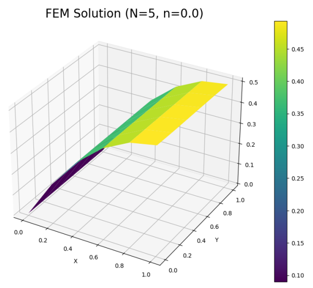
    ```
    $ bin/MeshSolver
    Results successfully saved to results/mesh_results.h5
    Visualization completed successfully!
    ```

  - HDF5 file (`mesh_results.h5`): Stores computed node coordinates and solution values.
  - Visualization plots (`solution_N{N}_n{n}.png`): Generated by the `plot_mesh.py` to visualize the numerical solution for given N (mesh divisions) and n.

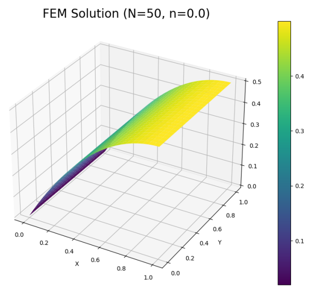
Figure 4 shows visualizations of the analytical solutions and FEM solutions for varying discretizations (N) and right-hand side parameters (n). The results concur with our expectations: as the mesh refines, the finite element solution becomes smoother and more accurate.



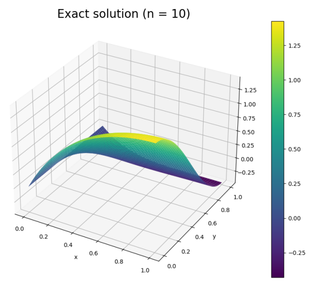
(a) Exact solution (n = 0)



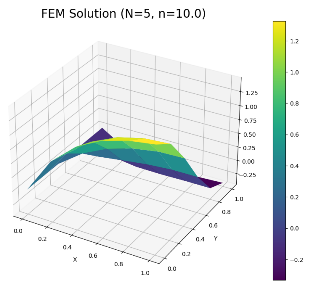
(b) FEM Solution (N = 5, n = 0.0)



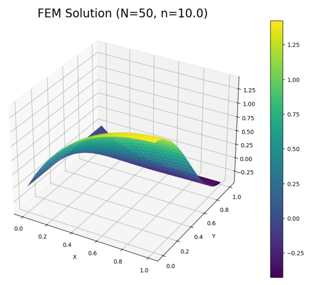
(c) FEM Solution (N = 50, n = 0.0)



(d) Exact solution (n = 10)



(e) FEM Solution (N = 5, n = 10.0)



(f) FEM Solution (N = 50, n = 10.0)

Figure 4: Comparison of exact and FEM Solutions for different values of N and n.

6.3 Timing

Table 1 shows the timings of major functions during the convergence test, where the finite element solutions are computed for 5 different meshsizes. We observe that computing the stiffness matrix and load vector takes the most time (64.73%) per call, while the `Eigen::Map Base` function is called the most (159000 times in total).

Function	**Calls**	**Time (ms)**	**Cumulative Time (s)**	**Percentage Time Per Call (%)**
Compute Stiffness Load	5	44.02	0.22	64.73
`Eigen::Assign Sparse to Sparse`	5	16.01	0.30	23.54
`Eigen::Conjugate Gradient`	5	8.00	0.34	11.77
`Eigen::Map Base`	159000	0.00	0.34	0.00
Generate Mesh	5	0.00	0.34	0.00
Norm Calculation	5	0.00	0.34	0.00

Table 1: Timings for key functions in the implementation. The table lists the number of calls, time per call, cumulative time, and percentage of total time spent per function.

7 Conclusions

This project implements the Finite Element Method (FEM) to solve the Poisson equation on a rectangular domain. We formulate the problem, derive the analytical solution, and develop a modular codebase to compute and visualize the numerical solution. Key steps include mesh generation, stiffness matrix assembly, and result validation through error analysis and convergence testing. The project demonstrates the expected theoretical convergence rates, confirming the accuracy of the implementation. We used a Boost-based test-suite for unit and regression testing, Valgrind for verifying memory safety and detecting potential leaks, and Docker for containerization.

8 Statement of Contributions

Every group member contributes equally to this project. Bowen proposed the topic idea with FEM formulations and wrote the code for MeshGenerate and the stiffness matrix. Shan handled parts with the build system, Docker, Valgrind, and presentation slides. Alwin created the verification mode, timing, code coverage, and test-suite. Judy made input parsing, file_io, result visualization. In addition, Bowen and Judy solved the analytical solution. Alwin and Shan reorganize the code structure together.