Two-Dimensional Boundary Value Problems on Circular and Square Domains with FEM

Sumaia Jahan Brinti

December 2024

1 Introduction

The Finite Element Method (FEM) is a popular method for numerically solving differential equations arising in engineering and mathematical modeling. To solve a problem, the FEM subdivides a large system into smaller, simpler parts called finite elements. This is achieved by a particular space discretization in the space dimensions, which is implemented by the construction of a mesh of the object: the numerical domain for the solution, which has a finite number of points. The finite element method formulation of a boundary value problem finally results in a system of algebraic equations.[5]

Literature Study on 2D Triangulation

A 2D triangulation is a partition of a planar domain into triangles. Key properties include:

- Triangles cover the entire domain without overlaps
- Adjacent triangles share only edges or vertices
- For curved boundaries, small triangles can approximate the domain shape

Tools and Libraries

- **TTL** is a generic triangulation library developed at SINTEF Applied Mathematics. It is generic in the sense that it does not rely on a specific data structure, allowing operation with custom application data structures while benefiting from a variety of generic algorithms in TTL that work directly on any data structure for triangulations. TTL is written in C++ and extensively uses function templates as a generic tool for the application programming interface (API) [3]
- **Qt** Creator is a cross-platform C++, JavaScript, Python and QML integrated development environment (IDE) which simplifies GUI application development. It is part of the SDK for the Qt GUI application development framework and uses the Qt API, which encapsulates host OS GUI function calls.

- **Eigen** is a high-level C++ library of template headers for linear algebra, matrix and vector operations, geometrical transformations, numerical solvers and related algorithms.

One-Dimensional Boundary Value

To demonstrate the differential approach to solving physical problems, we start with a one-dimensional boundary value problem using Poisson's equation as a typical example.

1.1 Domain and Equation

- One-dimensional continuous domain $I = [0, 1]$ with length $L = 1$.
- Poisson's equation in one-dimensional space is given by:

$$-\frac{d}{dx}\left(a\frac{du}{dx}\right) = f \quad \text{in } I. \tag{1}$$

1.2 Analytical Solution

- Assuming constants $a = 1$ and source function $f = 1$, the equation simplifies to:

$$-u'' = 1. \tag{2}$$

- Integration leads to:

$$u(x) = -\frac{x^2}{2} - C_1 x - C_2. \tag{3}$$

- Applying Dirichlet boundary conditions $u(0) = 0.25$, $u(1) = 0$, we find C_1 and C_2.
- The exact solution is:

$$u_{\text{exact}} = -0.5x^2 + 0.25x + 0.25. \tag{4}$$

Two-Dimensional Boundary Value Problem

Boundary value problems (BVPs) are differential equations with specified conditions at multiple points. Here are the main types of boundary conditions:

1.3 Dirichlet Boundary Conditions

- Specify the value of the solution at the boundary.
- Example: $u(0) = 1$, $u(1) = 0$.

1.4 Neumann Boundary Conditions

- Specify the derivative of the solution at the boundary.
- Example: $u'(0) = 2$, $u'(1) = -1$.

1.5 Robin Boundary Conditions

- Combine both Dirichlet and Neumann conditions.
- Involve a linear combination of the solution and its derivative.
- Mathematically expressed as:

$$au + b\frac{\partial u}{\partial n} = g \quad \text{on the boundary.} \tag{5}$$

- Also known as third-type or mixed boundary conditions, they are more general and often more realistic for many physical problems.
- Represented as:

$$au + b\frac{\partial u}{\partial n} = g \quad \text{on } \partial\Omega. \tag{6}$$

- Where:
  - a and b are non-zero constants or functions.
  - u is the unknown solution.
  - $\frac{\partial u}{\partial n}$ is the normal derivative at the boundary.
  - g is a given function on the boundary.

Importance in Modeling

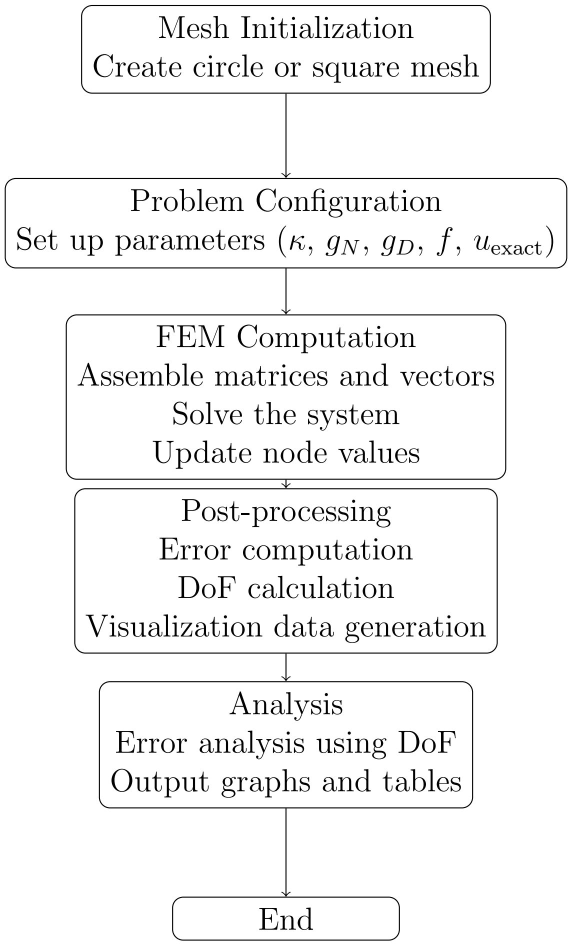
Robin boundary conditions offer more flexibility in modeling real-world phenomena:

- They allow for a more nuanced representation of boundary interactions.
- Can capture physical processes that involve both the value and the flux at boundaries.
- Particularly useful in problems where the boundary interacts with the surrounding environment.

In numerical methods like finite differences or finite elements, Robin conditions require special treatment at the boundary nodes, often involving a combination of the solution value and its derivative approximation.

Workflow and Algorithms

The FEMobject class encapsulates most of the computational logic. The main class orchestrates the overall process and handles result analysis. Mesh generation is problem-dependent (circle for Laplace/Poisson, square for Helmholtz). The solve method integrates various components (matrices, vectors) to produce the solution. Error analysis and visualization are separate processes that use the computed solution. **Click here to visit** Github code link



In the Finite Element Method (FEM) for solving various Boundary Value Problems (BVPs), we need to find the following matrix and vector to compute the system of linear system:

Stiffness Matrix (A)

The elements of the stiffness matrix A are computed as:

$$A_{ij} = \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \, dx, \quad i, j = 1, \ldots, N.$$

Mass Matrix (M)

The entries of the mass matrix M are given by:

$$M_{ij} = \int_\Omega \varphi_j \varphi_i \, dx, \quad i, j = 1, \ldots, N.$$

Load Vector (b)

The load vector b is defined as:

$$b_i = \int_\Omega f \varphi_i \, dx, \quad i = 1, \ldots, N,$$

where f is the given source function.

Robin Matrix (R)

The entries of the Robin matrix R are computed as:

$$R_{ij} = \int_\Gamma \kappa \varphi_j \varphi_i \, ds, \quad i, j = 1, \ldots, N,$$

where κ is a constant that defines the type of boundary conditions.

Robin Vector (r)

The elements of the Robin vector r are given by:

$$r_i = \int_\Gamma \big(\kappa g_D + g_N\big) \varphi_i \, ds, \quad i = 1, \ldots, N,$$

where g_D and g_N specify the inhomogeneous Dirichlet or Neumann boundary conditions, respectively.

The Pseudocodes are given below for CircleMesh and Squaremesh

Algorithm 1 circleMesh

```
1: function circleMesh($n, m, r$)
2:     Initialize an empty list for nodes
                        ▷ Step 1: Add three initial nodes forming an equilateral triangle
3:     for $i = 0$ to 2 do
4:         angle $\leftarrow (i \times 2 \times \pi)/3$
5:         radius $\leftarrow r/n$
6:         $(x, y) \leftarrow$ (radius $\times$ cos(angle), radius $\times$ sin(angle))
7:         Create a new node at $(x, y)$
8:         Add the new node to the nodes list
9:     end for
                        ▷ Step 2: Add nodes for each concentric circle
10:    for $k = 1$ to $n - 1$ do
11:        radius $\leftarrow (r \times (k + 1))/n$
12:        for $i = 0$ to $(m \times (k + 1)) - 1$ do
13:            angle $\leftarrow (i \times 2 \times \pi)/(m \times (k + 1))$
14:            $(x, y) \leftarrow$ (radius $\times$ cos(angle), radius $\times$ sin(angle))
15:            Create a new node at $(x, y)$
16:            Add the new node to the nodes list
17:        end for
18:    end for
                        ▷ Step 3: Perform Delaunay triangulation on the nodes list
19:    Perform Delaunay triangulation on the nodes list
20: end function
```

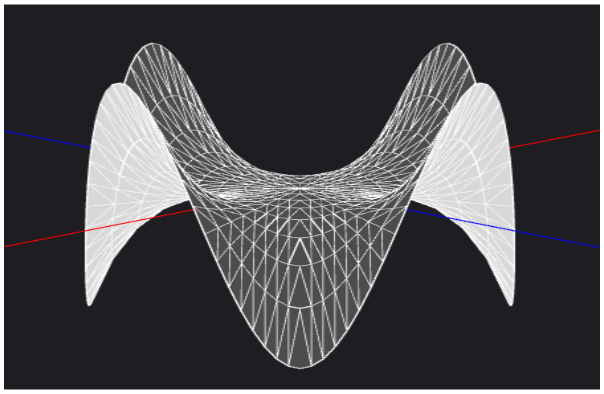
Algorithm 2 SquareMesh

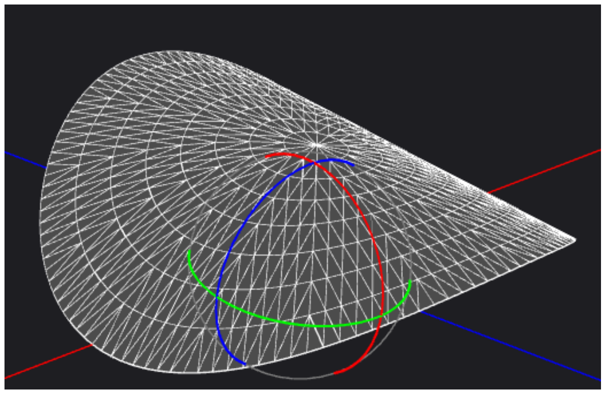
```
1: function SquareMesh($n, d, Op$)
2:     Initialize an empty list for nodes
3:     for $j = 0$ to $n$ do
4:         for $i = 0$ to $n$ do
5:             $(x, y) \leftarrow (Op.x + i \times d/n, Op.y + j \times d/n)$
6:             Create a new node at $(x, y)$
7:             Add the new node to the nodes list
8:         end for
9:     end for
10:    Perform Delaunay triangulation on the nodes list
11: end function
```

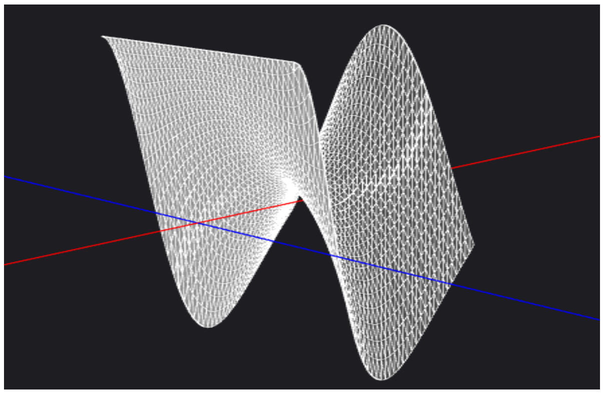
Results

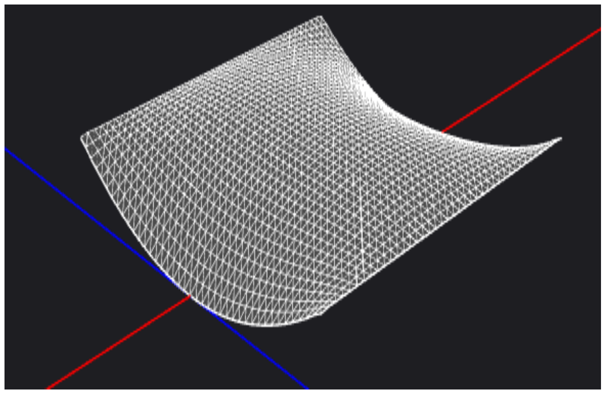
After implementing we got our desired results successfully. We implemented circular mesh for Laplace and Poisson so we got circular output for those. For Laplace, we got the below output. For the figure 1, the Error is 0.0158273 and Dof : 577 . On the other hand, the Error is 0.375146 and the Dof is 1681 for the figure of Square Laplace. For the Circle Poisson problem, we got the below output. For the figure 3, the Error is 0.00184879 and the Dof is 577. On the other hand, the Error for Square Poisson is 0.264921 and the Dof is 1681 . The Helmholtz output is given below. For the circle, the Error is 0.59051 and the Dof is 577. On the other hand, the Error of Square is 0.00246654 and the Dof is 1681 .

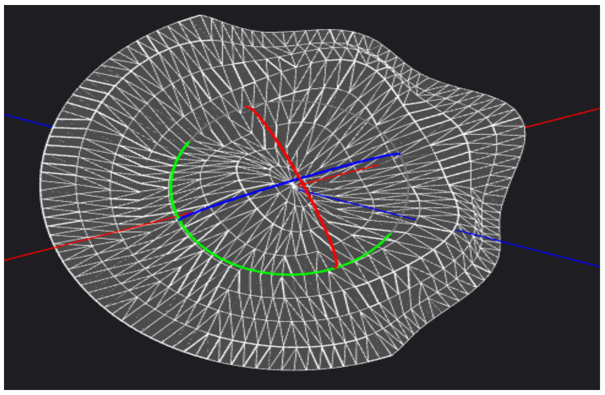
1.6 Laplace's Equation and Poisson's Equation

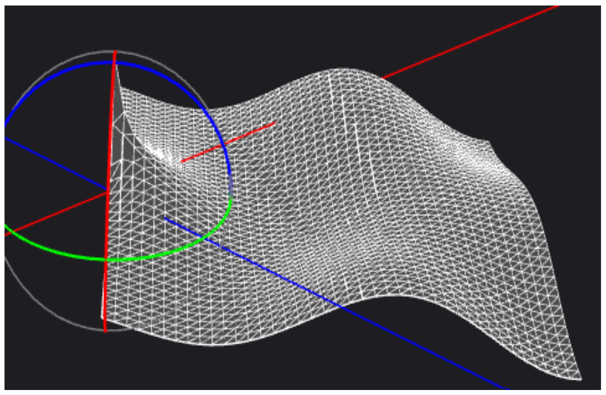

(a) Circle Laplace Equation


(b) Circle Poisson Equation


(a) Square Laplace Equation


(b) Square Poisson Equation


(a) Circle Helmholtz Equation


(b) Square Helmholtz Equation

1.7 Analysis

We got different error and DoF values for Laplace, Poisson and Helmholtz.

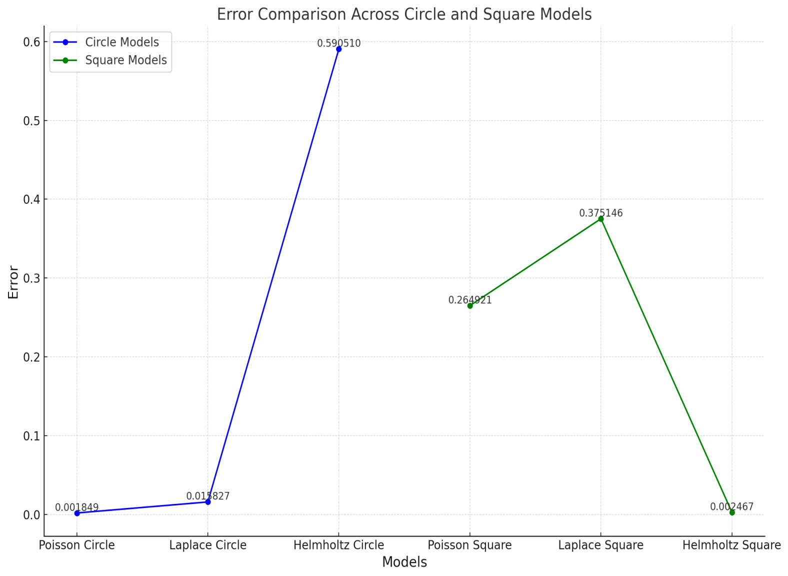


Figure 4: Circle vs Square Equation

Here, we used Python coding to generate a graph where the L2 error is scaled as the y-axis and DoF is scaled as the x-axis. Here, the y-axis followed logarithmic scaling. We plotted Laplace, Poisson and Helmholtz using different colours like blue, red, and green respectively. However, We noticed that when the value of DoF was increased, the error was decreased. This process was the same for all the problems like Laplace, Poisson and Helmholtz.

Increasing DoF improves accuracy (reduces error), but the improvement rate varies between models (circle vs. square) and problem types (Laplace, Poisson, Helmholtz).

Square vs. Circle The square model performs better overall, likely due to simpler boundary conditions and easier meshing.

Problem Complexity: The Poisson equation typically has higher errors due to the presence of the source term ff, which adds complexity to the solution. The Helmholtz equation benefits from its nature and meshes well with both square and circle geometries.

2 Conclusion

Although there are some issues, we have an approximate solution that is very close to being exact. We successfully generated the .obj files and graphs. We also were successful in getting the proper error rate and DoF value. With these values, we analysed the error that satisfies our theoretical knowledge. The graph and table made the analysis easier to understand. Nonetheless, this implementation is a successful demonstration of the Finite Element Method using partial differential equation(PDE)s. We efficiently implemented this method for PDE problem-solving. This solution effectively analyzed the errors

References

[1] Richard Courant. The finite element method. `https://stanoyevitch.net/NOPDEM.Chap13WithSolutions.pdf`.

[2] Tatiana Kravetc. Fem theory and implementation. DTE-3612, Department of Computer Science and Computational Engineering, Faculty of Engineering Science and Technology, UiT - The Arctic University of Norway, `https://uit.instructure.com/courses/34967`.

[3] SINTEF. Ttl, the triangulation template library. `https://www.sintef.no/globalassets/upload/ikt/9011/geometri/ttl/ttl_1.1.0_doc/index.html`.

[4] Wikipedia. Eigen library. `https://en.wikipedia.org/wiki/Eigen_(C++_library)`.

[5] Wikipedia. Finite element method. `https://en.wikipedia.org/wiki/Finite_element_method`.

[6] Wikipedia. Qt creator. `https://en.wikipedia.org/wiki/Qt_Creator`.

[5] [3] [6] [4] [1] [2]