NUMERICAL METHODS FE LAB REPORT

M1_M_ENG_NUMME

REPORT BY:

NAME 1: BENSON DANIEL COSMAN

NAME 2: CHUKWUEMEKA ENEH

GROUP: D-12

ECOLE CENTRALE DE NANTES

1.Introduction

The Finite Element Method (FEM) is a widely utilized computational tool in engineering, building upon earlier techniques like matrix methods and finite difference methods. Its versatility and accuracy make it an indispensable resource for analyzing and designing engineering systems and products. FEM provides engineers with a powerful analytical framework to model complex problems, making it highly effective in solving diverse engineering challenges.

This project focuses on developing a Python-based finite element program capable of solving stationary thermal analysis problems. By simulating heat transfer within a defined domain, stationary thermal analysis aids in understanding and predicting temperature distributions under steady-state conditions.

The program has been designed with the following features:

- **2D Domain Assumption**: The analysis is restricted to two-dimensional domains, simplifying the complexity and making the problem tractable.
- **4-Node Rectangular Elements**: The program uses elements that are linear and rectangular, with four nodes each. The elements are formulated in the physical space, avoiding the need for transformation.
- **Non-Distorted Elements**: Deformed or distorted elements are not supported, as formulations in the physical space are ineffective for such cases.
- **Isotropic Conductivity**: Material conductivity is assumed to be isotropic, meaning the thermal properties are uniform in all directions.

By implementing this program, we aim to provide a robust yet straightforward approach to stationary thermal problem-solving using FEM. The project emphasizes precision, efficiency, and adherence to theoretical principles, making it an excellent learning tool for understanding finite element analysis fundamentals.

2. Finite Element Method

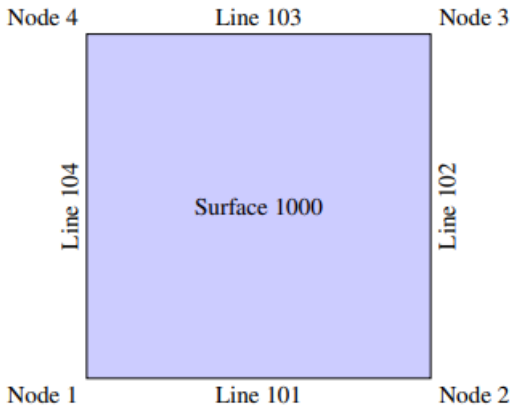
The Finite Element Method (FEM) is a numerical technique widely used to solve complex problems in physics and engineering. It finds application in areas such as structural analysis, heat transfer, fluid dynamics, mass transport, and electromagnetic potential. These problems often require solving partial differential equations (PDEs) with boundary conditions, which can be challenging to address analytically.

FEM transforms these problems into a system of algebraic equations by discretizing the domain into smaller, manageable components called finite elements. Within each element, approximations of the unknown variables are made, simplifying the problem into a smaller set of equations. These localized equations are then systematically assembled to represent the behavior of the entire domain.

By employing variational principles from the calculus of variations, FEM minimizes the associated error to approximate a solution. This approach efficiently breaks down a complex system into a network of interrelated components, making it easier to analyze and solve.

3. Problem

The problem will be specified by referring to geometric entities defined on the mesh. Any entity in the mesh belongs to a geometrical entity than can be a point, a curve or a surface. All the geometries given in the archive obeys the following convention:



If you want to prescribe a boundary condition on the elements lying on the bottom line of the square, you will have to refer to the 1D elements belonging to geometrical line 101. Adding a source term on the domain ends-up to refer to the elements belonging to surface 1000. In these two examples, ids 101 and 1000 are referred as physical ids.

4. Programming

4.1 Computing the Elementary Stiffness Matrix

In the finite element method (FEM), the element stiffness matrix K_e represents an element's resistance to heat conduction. It is derived from the weak form of the heat conduction equation. This weak form involves integrating the product of the material conductivity K, the gradients of the shape functions, and their transpose over the domain of the element:

$$\mathbf{K}_e = \int_{\Omega_e} \mathbf{B}^T K \mathbf{B} \, d\Omega$$

Here:

- K is the isotropic thermal conductivity of the material,
- Ω_e is the domain of the element,
- $\mathbf{B}$ is the matrix containing the spatial derivatives of the shape functions.

The derivatives of the shape functions with respect to x and y, which are determined using the coordinates of the element's nodes, are used to construct the $\mathbf{B}$-matrix. These derivatives are calculated using the coefficients of the shape functions, which are obtained by solving a system of equations involving a matrix $\mathbf{M}$.

The matrix $\mathbf{M}$ is constructed using the nodal coordinates (x_i, y_i) as follows:

$$\mathbf{M} = \begin{bmatrix} 1 & x_1 & y_1 & x_1 y_1 \\ 1 & x_2 & y_2 & x_2 y_2 \\ 1 & x_3 & y_3 & x_3 y_3 \\ 1 & x_4 & y_4 & x_4 y_4 \end{bmatrix}$$

By solving the system $\mathbf{Mx} = \mathbf{e}_i$ for each node, where $\mathbf{e}_i$ represents the standard basis vectors, the coefficients required for the partial derivatives are determined. These coefficients are then used to construct the $\mathbf{B}$-matrix:

$$\mathbf{B} = \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial x} & \frac{\partial N_4}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \frac{\partial N_3}{\partial y} & \frac{\partial N_4}{\partial y} \end{bmatrix}$$

Here, N_i represents the shape function corresponding to node i.

The shape functions $N_i(x, y)$ are polynomials used to approximate field variables within finite elements. They ensure interpolation consistency and satisfy the required continuity within the element. Each shape function is expressed as:

$$N_i(x, y) = a_i + b_i x + c_i y + d_i xy, \quad i = 1,2,3,4$$

The coefficients a_i, b_i, c_i, and d_i are determined based on the nodal values and ensure accurate interpolation of the field variable across the element domain.

```python
import numpy as np
from integrationRule import integrateOnQuadrangle

# TODO
# Computation of the elementary stiffness matrix Ke
# xyzVerts (INPUT): Coordinates of the nodes of the element (4x3 numpy array)
# conductivity (INPUT): Conductivity on the current element (scalar number).
# Ke (OUTPUT): Elementary stiffness matrix (4x4 numpy array).
def compute_coeff(xyzVerts):
    C_s=[]
    x_s=xyzVerts[:,0]
    y_s=xyzVerts[:,1]
    mat_sys=np.array([[1,1,1,1],x_s,y_s,x_s*y_s]).T
    for i in range(len(mat_sys)):
        rhs=np.zeros((4,1))
        rhs[i]=1
        C_matrix= np.linalg.solve(mat_sys,rhs)
        C_s.append(C_matrix)
    C_s_matrix=np.array(C_s)
    return C_s_matrix
```

```python

def computeKe(xyzVerts, conductivity):

    # Ke_xy is a function that should return the quantity that has to be integrated
    # in order to compute Ke: Ke = integral of Ke_xy on the element
    # NOTE: Ke_xy is a matricial quantity !
    # Set a dummy value for the moment
    def Compute_Be(x, y):
        B_e=np.zeros((2,4))
        C_1=compute_coeff(xyzVerts)[:,1]
        C_3=compute_coeff(xyzVerts)[:,3]
        C_2=compute_coeff(xyzVerts)[:,2]
        B_e[0,:]=C_1.flatten()+(C_3.flatten()*y)
        B_e[1,:]=C_2.flatten()+(C_3.flatten()*x)
        return B_e
    K=conductivity
    K=np.eye(2)*K
    Ke_xy = lambda x,y:Compute_Be(x,y).T@ K @ Compute_Be(x,y)


    # Function to compute the integral of Ke_xy
    # Works even if the element is not square
    # Used as a black box here !
    Ke = np.zeros((4, 4)) # Initialize Ke
    Ke = integrateOnQuadrangle(xyzVerts[:,:2], Ke_xy, Ke) # Integrate Ke_xy on the element

    return Ke

```

Figure 1 & 2: Python code for computing Stiffness Matrix

4.2 Elementary Volume Force Vector

The **element volume force vector** $\mathbf{F}_v^e$ represents the effect of internal heat sources distributed within a finite element in the context of the finite element method (FEM). This vector arises from the source term in the governing heat conduction equation, which accounts for the heat generated per unit volume. The weak form of the heat conduction equation leads to the following expression for $\mathbf{F}_v^e$:

$$\mathbf{F}_v^e = \int_{\Omega_e} \mathbf{N}^T \cdot q \, d\Omega$$

Here:

- Ω_e is the area of the finite element,
- $\mathbf{N}$ is the shape function vector that interpolates the nodal values of the field variable over the element,
- q represents the internal heat generation per unit volume.

To define the shape functions $N_i(x, y)$ for a four-node quadrilateral element, a bilinear polynomial form is employed:

$$N_i(x, y) = a_i + b_i x + c_i y + d_i xy, \quad i = 1,2,3,4$$

The coefficients a_i, b_i, c_i, d_i are unique for each node i and are determined by solving a set of equations derived from the nodal coordinates. These equations are formed using a matrix $\mathbf{M}$, which is constructed as follows:

$$\mathbf{M} = \begin{bmatrix} 1 & x_1 & y_1 & x_1 y_1 \\ 1 & x_2 & y_2 & x_2 y_2 \\ 1 & x_3 & y_3 & x_3 y_3 \\ 1 & x_4 & y_4 & x_4 y_4 \end{bmatrix}$$

By solving $\mathbf{M}\mathbf{x} = \mathbf{e}_i$ for each node i, where $\mathbf{e}_i$ is the standard basis vector, the coefficients of the shape functions can be computed. These shape functions are then used to interpolate the heat generation and calculate the contribution of internal sources to the element's volume force vector

```python
# TODO
# Computation of the elementary volume force vector Fve (source term effects)
# xyzVerts (INPUT): Coordinates of the nodes of the element (4x3 numpy array)
# sourceTerm (INPUT): Source term evaluator.
# physElt (INPUT): physical id of the current element (integer)
# Fe (OUTPUT): Elementary force vector (4 components numpy array)
# NOTE: The evaluator is a function of two variables: (xyz, physElt).
# xyz = numpy array containing the coordinates of the evaluation point
# physElt = physical id of the current element (integer)
def computeFve(xyzVerts, sourceTerm, physElt):

    # Similar to the approach used for the stiffness matrix
    # Fve_xy is a function that should return the quantity that has to be integrated
    # in order to compute Fve: Fve = integral of Fve_xy on the element
    # NOTE: Fve_xy is a vectorial quantity !
    # Set a dummy value for the moment
    def compute_Ne(x,y):
        C_0=compute_coeff(xyzVerts)[:,0]
        C_1=compute_coeff(xyzVerts)[:,1]
        C_3=compute_coeff(xyzVerts)[:,3]
        C_2=compute_coeff(xyzVerts)[:,2]
        Ne=C_0+(C_1*x)+(C_2*y)+(C_3*x*y)
        Ne=Ne.flatten()

        return Ne
    def Fve_xy(x,y):
        Ne=compute_Ne(x,y)
        return sourceTerm(np.array([x,y,0.]),physElt)*Ne
    # Fve_xy = lambda x,y: np.ones(4)
    # Function to compute the integral of Fve_xy
    # Works even if the element is not square
    # Used as a black box here !
    Fve = np.zeros(4) # Initialize Fve
    Fve = integrateOnQuadrangle(xyzVerts[:,:2], Fve_xy, Fve) # Integrate

    return Fve
```

Figure 3: Python Code for Implementation of Volume Force Vector

4.3 Neumann Force Vector

In the rect4ThermalDirect.py file, the Neumann force vector was computed using a simplified formula, assuming a constant heat flux q_n. The original integral expression is:

$$\mathbf{F}_e^N = \int_{\partial\Omega_e^N} \mathbf{N}^T \, q_n \, d\mu \qquad (1)$$

This was reduced to a simplified form due to the constant nature of q_n:

$$\mathbf{F}_e^N = \begin{bmatrix} \frac{L_e}{2} \\ \frac{L_e}{2} \end{bmatrix} \qquad (2)$$

Here, L_e represents the length of the Neumann boundary segment. In this formulation, one component was ignored because it contributed zero to the solution. As a result, the formula is explicitly written only for nodes located on the Neumann boundary.

This simplification helps focus the computation on relevant contributions, ensuring computational efficiency while accurately capturing the effects of the Neumann boundary condition.

```
90   # TODO
91   # Computation of the elementary Neumann force vector FNe
92   # xyzVerts (INPUT): Coordinates of the nodes of the edge (2x3 numpy array)
93   # flux (INPUT): Value of the prescribed flux on the current edge (scalar number)
94   # FNe (OUTPUT): Elementary force vector (2 components numpy array)
95   def computeFNe(xyzVerts, flux):
96       le=np.linalg.norm(xyzVerts[1]-xyzVerts[0])
97       # Set a dummy value for the moment
98       FNe = np.ones(2)
99
100      return flux*(le/2)*FNe
```

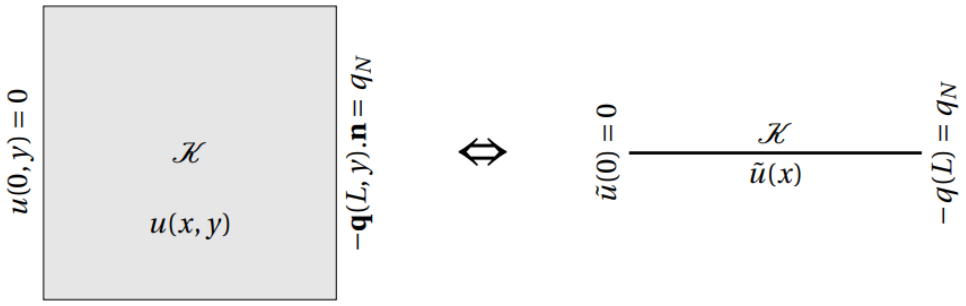
Figure 4: Python Code for Implementation of Neumann Force Vector

To calculate the element length Le, a straightforward mathematical formula was used to determine the distance between two points in a coordinate system. Finally, the result was scaled by an identity-like vector created using the np.ones function, effectively multiplying all components together to compute the desired values.

5 Problem Description

5.1 Problem 1(Stage 3)

Validate your code against simple problems whose analytical solutions are known. As an example, the two problems depicted below are equivalent (i.e. u(x, y) = u˜(x)).



Thus, you can build reference solutions with known features (linear or quadratic for example).

- Assume flux and conductivity as qN=1 and K=1 respectively
- Assume source term value as r=0

```python
#Import finite element solver function
from solveFE import solveFE

#Mesh file
meshName = 'square20x20.msh'

# Setup the problem we want to solve
# Neumann boundary conditions : as a dictionary (~ map) (physicalId: qN value)
BCNs = {102:1}

# Dirichlet boundary conditions for lines : as a dictionary (~ map) (physicalId: uD value)
BCD_lns = {104:0}


# Dirichlet boundary conditions for nodes : as a dictionary (~ map) (physicalId: uD value)
BCD_nds = {}

# Conductivity (isotropic for example) : as a dictionary (~ map) (physicalId: Kfourier)
conductivities = {1000:1}

# Source term (constant for example) : as a lambda function, depending on
# the phydical coordinates and (if necessary) the physical id of the element.
# xyz is assumed to be a numpy array
sourceTerm = lambda xyz, physdom: 0.

exportName = 'stage3.pos'


# Call the resolution routine
# Additional parameters :
# useSparse = True if you experience memory issues
# verboseOutput = False if you want minimal verbosity
useSparse = False
verboseOutput = True
solveFE(meshName, conductivities, BCNs, BCD_lns, BCD_nds, sourceTerm, exportName, useSparse, verboseOutput)
```

Figure 5: Python Code for Solution of Finite Element Analysis (Stage 3)

The output for the code above is as follows:

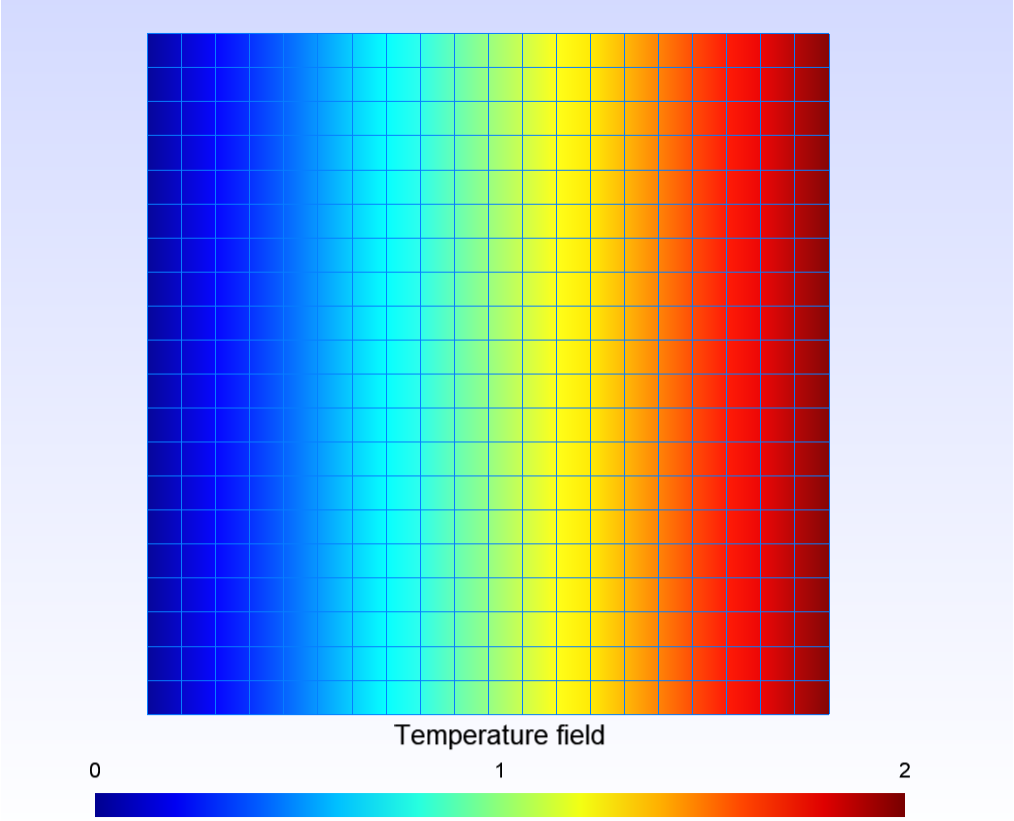


Figure 6: Output of numerical solution

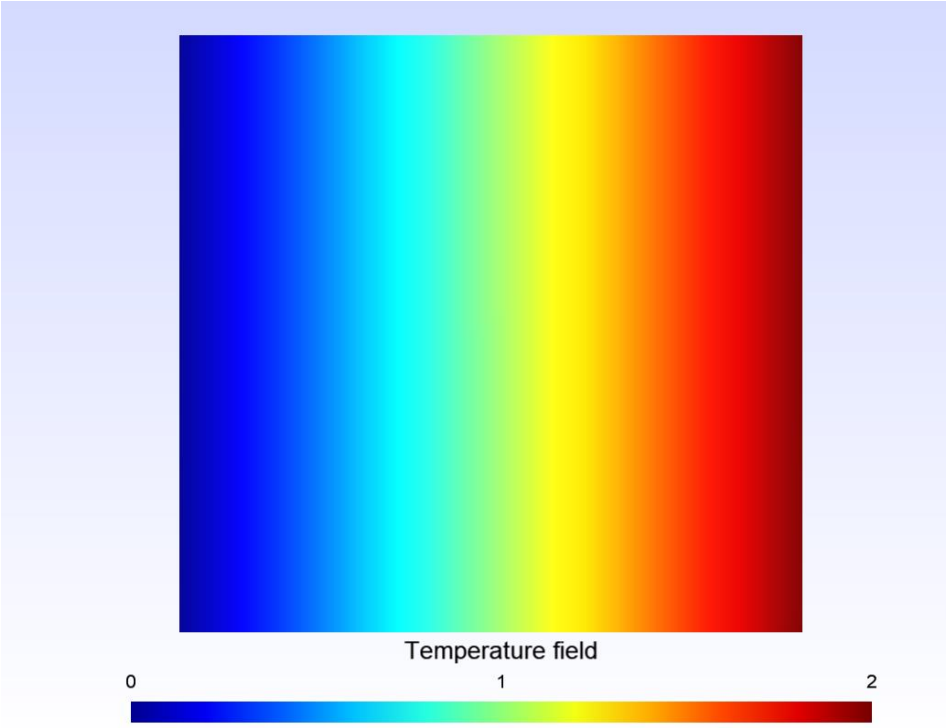


Figure 7: Output of Analytical Solution

5.2 Problem 2(Stage 4:Validation)

1. Validate your code against the following problem: u = 0 on line 101, r = 0, unit flux on line 103, and conductivity equal to X X X. With X X X = groupNumber + Y Y Y and Y Y Y = 0 for group A, 100 for group B and 200 for group C. Example: X X X = 205 for group C-05.

Modifying the boundary conditions as per the given parameters:

- **Group Number**: D-12 , with YYY taken as 300 for group D.
- **XXX**: Assigned the value 312.

```
[mesh]
name = 'square20x20.msh'

[BCs_Neumann]
bc103.physId = 103
bc103.value = 1

[BCs_Dirichlet_line]
bc101.physId = 101
bc101.value = 0

[BCs_Dirichlet_nodes]
#bc1.physId = 1
#bc1.value = 0

[Source]
# Types are: 'constant', 'piecewise_constant'
source1.sourceType = 'constant'
source1.value = 0.0

[Conductivities]
conductivity1000.physId = 1000
conductivity1000.value = 312.0

[export]
exportName = 'Validation_1.pos'

[solver_options]
useSparse = false
verboseOutput = true
```

Figure 8 : Assigning XXX value as 312

The output for the program above is as follows:

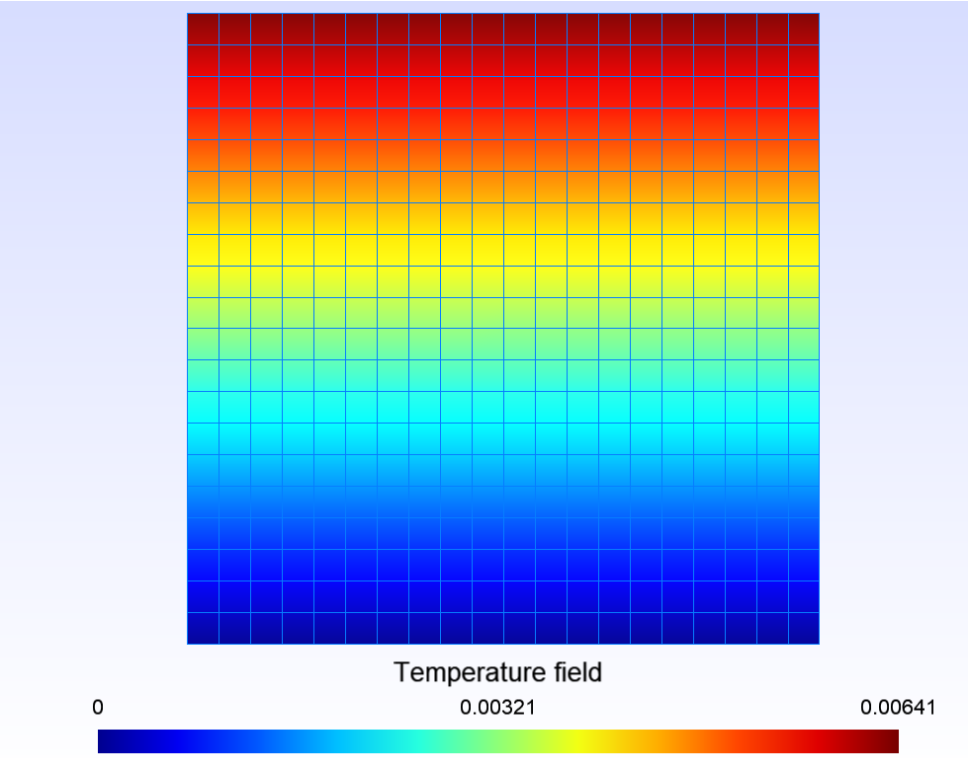


Figure 9: Output for Validation against the given boundary condition with visible mesh

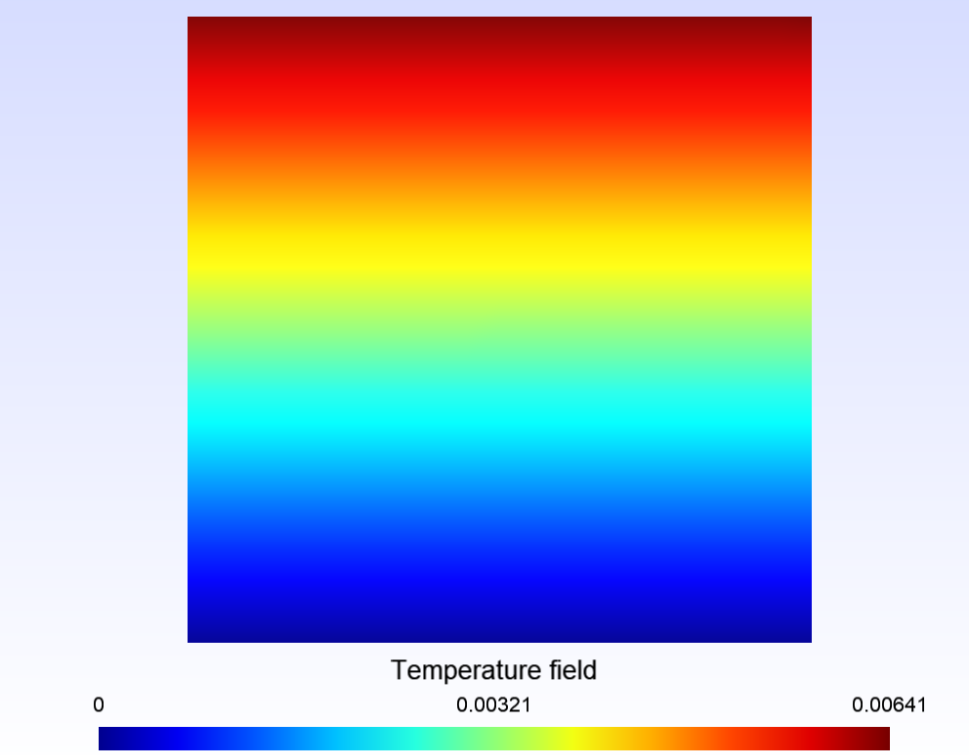
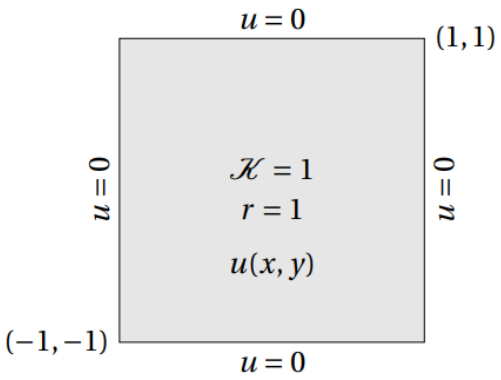


Figure 10: Output for Validation against the given boundary condition without visible mesh

2. Validate your code against the following problem:



The boundary conditions shown in the problem above is defined in a validation2stage4.toml file, and then run in the testFE_cli.py script. The definition of the Boundary conditions and other parameters is shown in the screenshot below

```
[mesh]
name = 'square20x20.msh'

[BCs_Neumann]


[BCs_Dirichlet_line]
bc101.physId = 101
bc101.value = 0
bc102.physId = 102
bc102.value = 0
bc103.physId = 103
bc103.value = 0
bc104.physId = 104
bc104.value = 0

[BCs_Dirichlet_nodes]


[Source]
# Types are: 'constant', 'piecewise_constant'
source1.sourceType = 'constant'
source1.value = 1.0

[Conductivities]
conductivity1000.physId = 1000
conductivity1000.value = 1.0

[export]
exportName = 'Validation_2.pos'

[solver_options]
useSparse = false
verboseOutput = true
```

Figure 11: Assigning the Boundary condition and other parameters

The result produced for the code above is as follows:

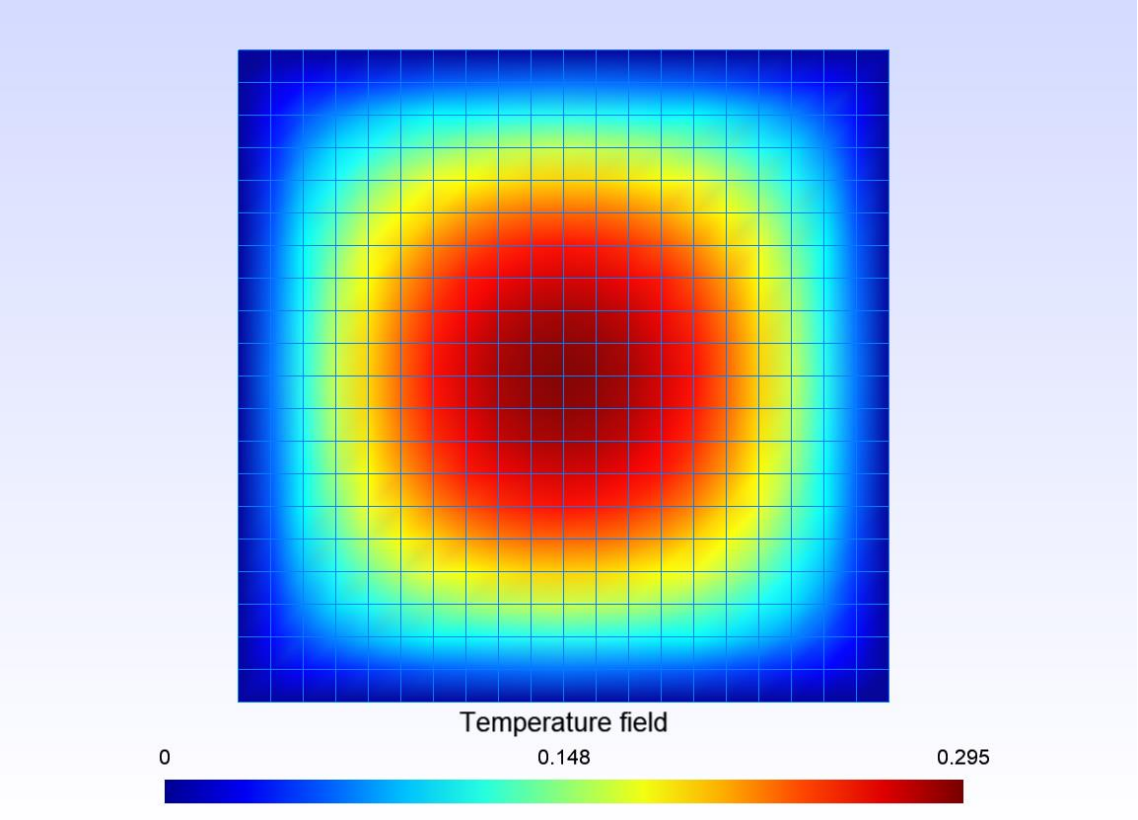


Figure 12: Output for Numerical Solution with visible mesh

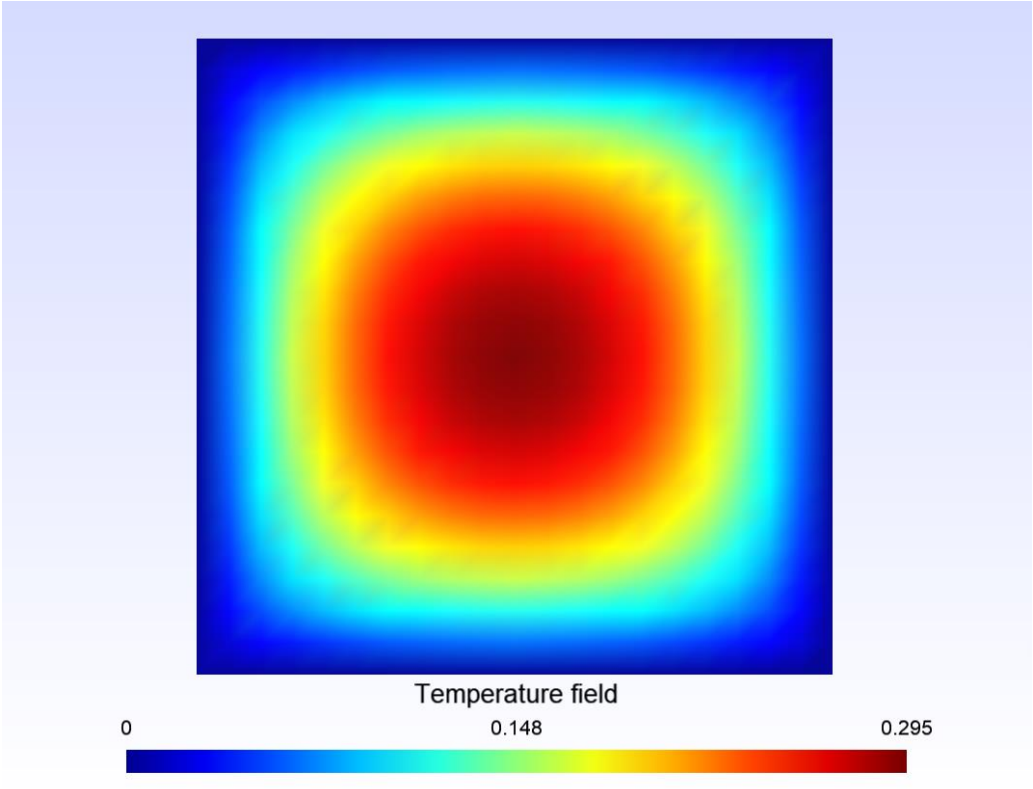


Figure 13: Output for Numerical Solution without visible mesh

The exact solution for the problem is given by:

$$u(x,y) = \sum_{i=1,\text{odd}}^{\infty} \sum_{j=1,\text{odd}}^{\infty} \frac{64}{\pi^4(i^2+j^2)ij} \sin\left(\frac{i\pi(x+1)}{2}\right) \sin\left(\frac{j\pi(y+1)}{2}\right)$$

- The above equation is represented in the code below:

```
import numpy as np
import matplotlib.pyplot as plt
def solnexact(x,y):
    exact_soln=0
    for i in range(1,100,2):
        for j in range(1,100,2):
            soln=(64/(np.pi**4*(i**2+j**2)*i*j))*np.sin((i*np.pi*(x+1))/2)*np.sin((j*np.pi*(y+1))/2)
            exact_soln+=soln
    retu
                (function) def solnexact(
                    x: Any,
                    y: Any
                ) -> (Any | Literal[0])
x_values
y_values
X,Y=np.m
U_exact=solnexact(x=X,y=Y)
plt.contourf(X,Y,U_exact, cmap='coolwarm')
plt.colorbar(label='u(x,y)')
plt.show()
```

Figure 14: Python Code for analytical solution

The result produced for the code above is as follows

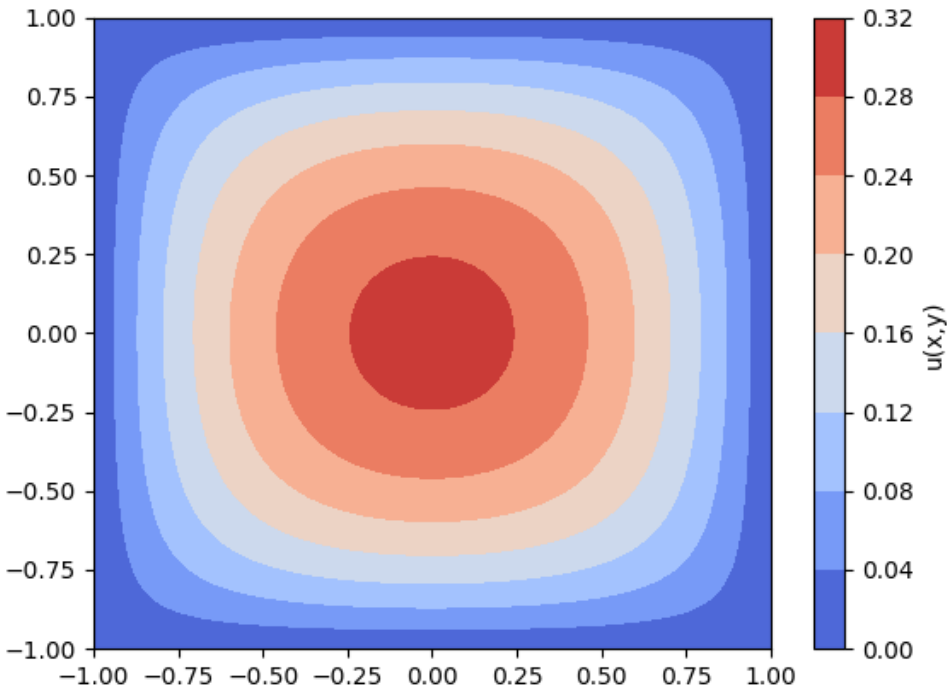


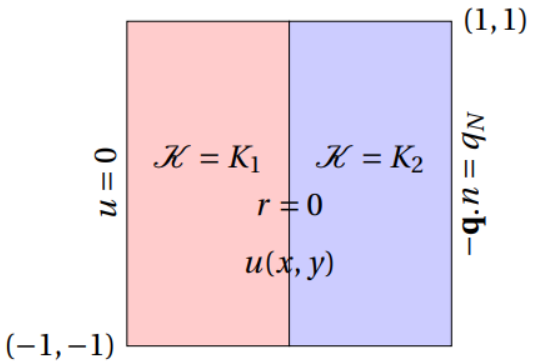
Figure 15: Output For analytical solution

Observation: We can understand that there is a high correlation between the numerical and analytical solutions, considering the temperature distribution pattern is the same for both solutions, having the minimum at the outer surface and maximum at the center of the surface. For the numerical solution, the temperature gradient value ranges from 0 to 0.295, while for the analytical exact solution, it ranges from 0 to 0.32.

5.3 Problem 3(Bonus Question)

1. Modify your program to be able to treat a domain with two isotropic materials (see squareBimat.msh);

2. Check your implementation against analytical a 1D problem consisting of two materials, such as:



The analytical solution for $u(x)$ is piecewise-defined, depending on the domain:

$$u(x) = \begin{cases} \frac{q_N}{K_1}(x+1), & x \in [-1,0] \\ \frac{q_N}{K_2}x + \frac{q_N}{K_1}, & x \in [0,1] \end{cases}$$

Explanation:

1. **Domain Division**:
   - The solution is divided into two segments: $x \in [-1,0]$ and $x \in [0,1]$, reflecting a change in material properties or boundary conditions at $x = 0$.
2. **Parameters**:
   - q_N: The heat flux or Neumann boundary condition.
   - K_1 and K_2: Thermal conductivities for the respective domains.
3. **Solution Components**:
   - In $x \in [-1,0]$: The solution is linear, scaled by $\frac{q_N}{K_1}$, with a slope proportional to the conductivity.
   - In $x \in [0,1]$: The solution retains linearity but incorporates both $\frac{q_N}{K_2}x$ and an offset term $\frac{q_N}{K_1}$, ensuring continuity at $x = 0$.

This piecewise analytical solution demonstrates how thermal conductivity and boundary conditions influence the temperature profile across the domain.

- For numerical Programming,
- assume qN=5
- K1=1
- K2=10

```
meshName = 'squareBimat20x20.msh'

# Setup the problem we want to solve
# Neumann boundary conditions : as a dictionary (~ map) (physicalId: qN value)
BCNs = {101:5}

# Dirichlet boundary conditions for lines : as a dictionary (~ map) (physicalId: uD value)
BCD_lns = {103:0}


# Dirichlet boundary conditions for nodes : as a dictionary (~ map) (physicalId: uD value)
BCD_nds = {}

# Conductivity (isotropic for example) : as a dictionary (~ map) (physicalId: Kfourier)
conductivities = {1000:10, 2000:1}

# Source term (constant for example) : as a lambda function, depending on
# the phydical coordinates and (if necessary) the physical id of the element.
# xyz is assumed to be a numpy array
sourceTerm = lambda xyz, physdom: 0.

exportName = 'bonusquestion.pos'
```

Figure 16: Python Code for Numerical Solution

The Output for the code above is as follows:

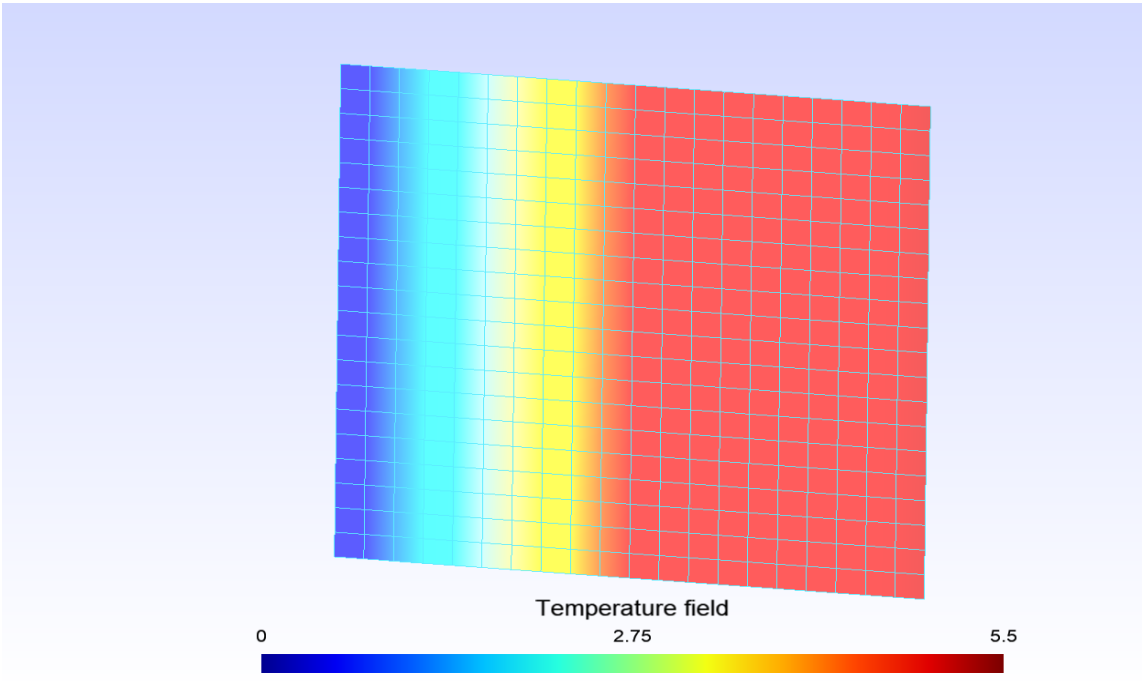


Figure 17: Output for Numerical solution

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

def exactsoln_bonus(x,K1,K2,qN):
    if -1<=x and x<=0:
        exact_soln=(qN/K1)*(x+1)
    else:
        exact_soln=(qN*x/K2)+(qN/K1)

    return exact_soln

x_value=np.linspace(-1,1,100)
y_value=np.linspace(-1,2,100)
X,Y=np.meshgrid(x_value,y_value)
U=np.zeros_like(X)
for i in range(U.shape[0]):
    U[i]=exactsoln_bonus(x=x_value[i],K1=1,K2=10,qN=5)
U=np.transpose(U)

plt.contourf(X,Y,U, cmap='plasma')
plt.show
```

Figure 18: Python Code for analytical solution

The output for the code above is as follows:

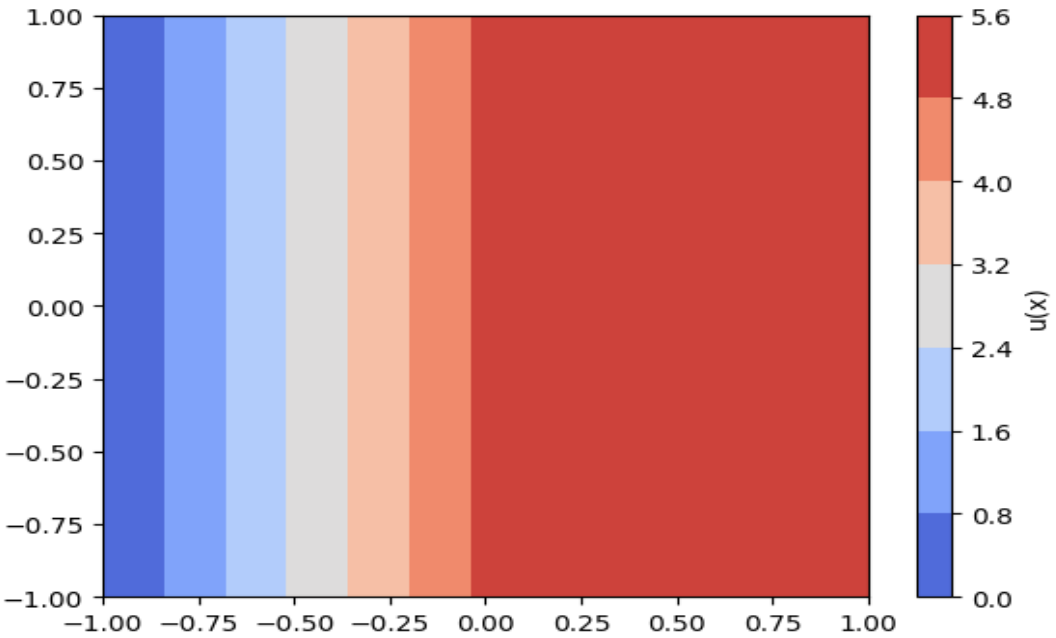


Figure 19: Output for analytical solution

Observation: We can understand that there is a high correlation between the numerical and analytical solutions. considering the temperature distribution pattern is the same for both solutions, having the minimum towards the left side of the surface and maximum at the right side of the

surface. For the numerical solution, the temperature gradient value ranges from 0 to 5.5, while for the analytical exact solution, it ranges from 0 to 5.6.

6 Conclusion

This project successfully employed linear quadrilateral elements to solve stationary thermal problems using a 2D finite element method (FEM) implemented in Python. The formulation involved constructing stiffness matrices and force vectors to account for internal heat sources and boundary flux conditions. By solving the system of equations, the code determined temperature distributions across the domain. Validation against analytical solutions confirmed the accuracy of the implementation, highlighting the reliability of the chosen methodology.

The use of Gmsh for mesh generation facilitated the creation of complex geometries, while the modular structure of the code streamlined tasks such as mesh parsing, numerical integration, and result visualization. This modularity not only made the implementation efficient but also easier to extend and adapt for different scenarios. The good agreement between numerical and analytical results underscored the robustness of the FEM approach in modeling heat conduction problems.

By producing precise and verifiable solutions, the project deepened the understanding of finite element methods for thermal analysis. Additionally, it enhanced Python programming skills through the integration of various computational tools and libraries. Overall, this work demonstrated how numerical techniques can provide accurate and insightful solutions to real-world engineering problems, further strengthening the utility and adaptability of FEM for thermal applications.