Group A-03

NUMME Lab - Finite Elements

Sudarson Nanthacoumarane

Objective - Use Finite Element Methods to find the temperature distribution across a domain given the source term and boundary conditions.

1. Problem Statement

The temperature across a domain can be represented in its strong form as:

$$(S) = \begin{cases} div(\overline{q}) - r = 0 \\ \overline{q} = -K.\nabla u \\ u(x) = u_D(x)\ \forall\ x\ \in \partial\Omega_D \\ -\overline{q}.\overline{n} = q_N \quad \forall\ x\ \in \partial\Omega_N \end{cases} \tag{1}$$

2. Solving the problem

We solve the problem using the Ritz Method, or by minimizing energy *J* in a 2-Dimentional space.

$$J(u^h) = \frac{1}{2}\, a(u^{h,} u^h) - b(u^h) \tag{2}$$

$$with\ J_a(u^h) = \frac{1}{2}\, a(u^{h,} u^h) \quad \text{and} \quad J_b(u^h) = b(u^h) \tag{3}$$

$$where\ u^h = N_k(x)\, U_K + N_l(x)\, U_l + N_m(x)\, U_m \tag{4}$$

u^h is the solution in a finite dimentionsal subspace, *N(x)* are interpolation or shape functions, and *U* are the displacement functions at each node. There are 3 shape functions and 3 displacement functions because we are using triangluar elements to solve our problem, resulting in each element consisting of 3 vertexes or nodes. Each of the shape functions take the form:

$$N_i(x) = c_0^i + c_1^i x + c_2^i y \quad i = k, l, m \tag{5}$$

Each of these interpolation functions *Ni* must satisfy the condition that they are equal to 1 at *xi* and 0 everywhere else. For example,

$$\begin{cases} N_k(x_k) = 1 \\ N_k(x_l) = 0 \\ N_k(x_m) = 0 \end{cases} \tag{6}$$

The unknown constants can be found using a linear system of equations. For example, c^k can be found by:

$$\begin{bmatrix} 1 & x_k & y_k \\ 1 & x_l & y_l \\ 1 & x_m & y_m \end{bmatrix} \begin{bmatrix} c_0^k \\ c_1^k \\ c_2^k \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{7}$$

2.1. Elemental Stiffness Matrix

The first term of the energy expression over a single element *n* can be written as:

$$J_a^{(n)} = \frac{1}{2}\, a^{(a)}(u^h, u^h) = \frac{1}{2}\int_{\tau_n} \bar{\bar{\nabla}} u^h \mathrm{K} \bar{\bar{\nabla}} u^h \; d\Omega \tag{8}$$

$$= \frac{1}{2}\,[U_e]^T \int_{\tau_n} [B_e]^T [\mathrm{K}]\; [B_e]\; d\Omega\; [U_e] \tag{9}$$

Where *Be* is obtained by representing flux over the element in terms of matrices.

$$q^h = \mathrm{K}\bar{\bar{\nabla}} u^h = \mathrm{K}[N_e].\,[U_e]^T \tag{10}$$

$$q^h = \begin{bmatrix} c_1^k & c_1^l & c_1^m \\ c_2^k & c_2^l & c_2^m \end{bmatrix} \begin{bmatrix} U_k \\ U_l \\ U_m \end{bmatrix} \quad where\ [B_e] = \begin{bmatrix} c_1^k & c_1^l & c_1^m \\ c_2^k & c_2^l & c_2^m \end{bmatrix} \tag{11}$$

Therefore, elemental energy can be rewritten as:

$$J_a^{(n)} = \frac{1}{2}\,[U_e]^T [K_e]\; [U_e] \quad where\ \ [K_e] = \int_{\tau_n} [B_e]^T [\mathrm{K}]\; [B_e]\; d\Omega \tag{12}$$

2.2. Volumic Force Vector

The second term in the energy expression can be split into two terms:

$$J_b(u^h) = J_{b,v}(u^h) + J_{b,N}(u^h) \tag{13}$$

$$where\ J_{b,v}(u^h) = \int_{\Omega_h} u^h r\; d\Omega \quad \text{and}\ J_{b,N}(u^h) = \int_{\partial\Omega_N} u^h q_N\; d\Gamma \tag{14}$$

Consider the elementary contributions of the first term mentioned above:

$$J_{b,v}^{(n)}(u^h) = \int_{\mathrm{T}_n} [U_e]\; [N_e]^T\; r\; d\Omega \;=\; [U_e][F_e^v] \tag{15}$$

$$where\ [F_e^v] = \int_{\mathrm{T}_e} [N_e]^T r\; d\Omega \tag{16}$$

$$[F_e^v] = \int_{\mathrm{T}_e} N_k(x) r\; d\Omega \;+\; \int_{\mathrm{T}_e} N_l(x) r\; d\Omega \;+\; \int_{\mathrm{T}_e} N_m(x) r\; d\Omega \tag{17}$$

We know that the sum of interpolation functions is equal to 1. Since they are linear, the interpolation functions are also equal to each other.

$$\int_{\mathrm{T}_e} N_k(x)\; d\Omega \;+\; \int_{\mathrm{T}_e} N_l(x)\; d\Omega \;+\; \int_{\mathrm{T}_e} N_m(x)\; d\Omega = 1\; d\Omega = Area\ A_e \tag{18}$$

If source term *r* is a constant, the Volumic Force Vector can be written as:

$$[F_e^v] = \frac{r}{3} A_e \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{19}$$

2.3. Neumann Force Vector

The second contribution to *Jb* represents the Neumann boundary conditions for our problem:

$$J_{b,N}(u^h) = \int\limits_{\partial\Omega_N} u^h q_N \; d\Gamma \tag{20}$$

Across each element, this term is written as:

$$J_{b,N}^{(n)}(u^h) = \int\limits_{\mathrm{T}_e} [U_e]\,[N_e]^T q_N \; d\Gamma \quad = \; [U_e]\,[F_e^N] \tag{21}$$

$$[F_e^N] = \int\limits_{edge} [N_e]^T q_N \; d\Gamma \tag{22}$$

3. Complete tri3thermalDirect.py

3.1. Compute Elemental Stiffness Matrix Ke

We know that the Elemental Stiffness Matrix can be represented as an integral of elemental matrices and vectors across the element.

$$[K_e] = \int\limits_{T_e} [B_e]^T \; .\mathrm{K}.\; [B_e] \;\; d\Omega \tag{23}$$

In the case of isotropic matrial, [K] is a diagonal and:

$$[K_e] = \mathrm{K}\; A_e \; [B_e]^T [B_e] \tag{24}$$

First, seperate functions to compute the area of an element and constants ck, cl, and cm were written to cleanup the code and make these values available anywhere in the script. The *getConstants()* function takes arguements for the coordinates of elemental nodes and finds the coefficients of the interpolation functions by solving the linear system of equations. The *getArea()* function finds area of the triangular element using its coordinates. This function takes advantage of the fact that area of a triangle is half the determinant of its coordinates.

```
# returns constant ck, cl, cm for an element
def getConstants(xyz,N):
    coordinateMatrix = np.ones((N,N))
    c = np.zeros((N,N))
    f = np.eye(N)

    coordinateMatrix[:,1] = xyz[:,0]
    coordinateMatrix[:,2] = xyz[:,1]

    # Find all c values, solution stored row-wise
    inverseMat = np.linalg.inv(coordinateMatrix)
    for i in range (0,N):
        c[i] = np.matmul(inverseMat,f[i])

    return np.transpose(c)

# Find area of element using xyzVerts coordinates
def getArea(xyz):
    xyz[:,2] = 1
    area = 0.5*abs(np.linalg.det(xyz))
    return area
```

The *computeKe()* function uses the above functions to calculate Elemental Stiffness Matrix:

```
def computeKe(xyzVerts, conductivity):
    N = 3
    c = getConstants(xyzVerts,N)

    Be = np.zeros((2,3))
    Be[0,:] = c[1,:]        # Assign c values to Be
    Be[1,:] = c[2,:]

    Ke   =   conductivity   *   getArea(xyzVerts)   *   np.matmul(np.trans-
pose(Be),Be)

    return Ke
```

3.2. Compute Volumic Force Vector

The equation to compute Volumic Force Vector and its function definition are:

$$[F_e^v] = \frac{r}{3} A_e \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{25}$$

```
def computeFve(xyzVerts, sourceTerm, physElt):
    N = 3
    area = getArea(xyzVerts)
    Fve = np.zeros(N)
    one = np.ones(N)

    Fve = sourceTerm(xyzVerts,physElt) * area * one / 3
    return Fve
```

3.3. Compute Neumann Force Vector

In practice, instead of writing restrictions of the interpolation functions on the Neumann Boundary, the approximation is written directly on the edges lying on the boundary as 1D interpolation functions. The integral is therefore simplified into:

$$[F_e^N] = \int_0^{edge} [N_k(\eta), N_l(\eta)]^T q_N \quad d\eta \tag{26}$$

```
def computeFNe(xyzVerts, flux):
    x1 = xyzVerts[0][0]
    x2 = xyzVerts[1][0]
    y1 = xyzVerts[0][1]
    y2 = xyzVerts[1][1]
    Le = np.sqrt( (x2-x1)**2 + (y2-y1)**2)

    FNe = [flux*Le/2, flux*Le/2]
    return FNe
```

3.4. Results

The following results were obtained by considering *u = 0* on the Dirichlet Boundary, *qN = 1* on the Neumann Boundary, and source term *r = 0*. For group A-03, the conductivity was 3 *W/(mm.K)*. These results are shown for a 4x4, 10x10, 20x20, and 40x40 mesh.

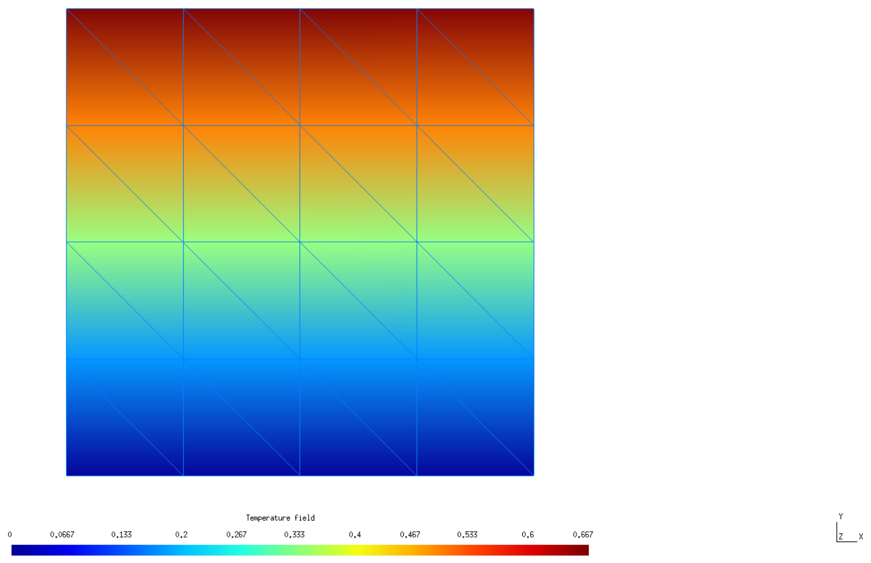

Figure 1 *Temperature distribution on 4x4 square mesh*

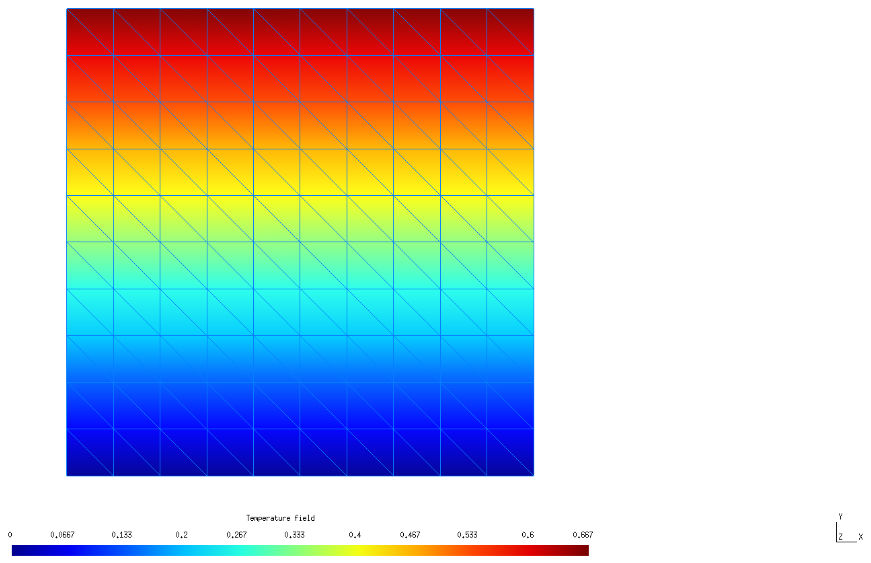

Figure 2 *Temperature distribution on 10x10 square mesh*

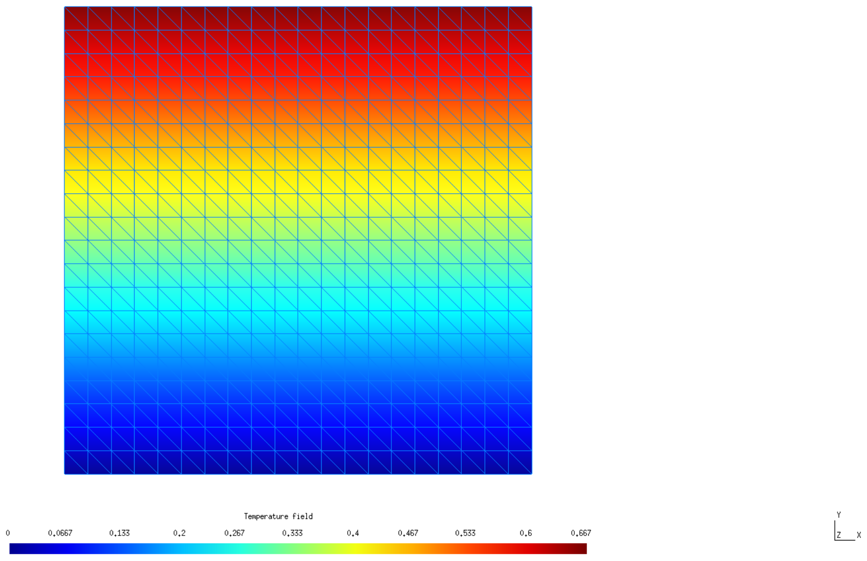

Figure 3 *Temperature distribution on 20x20 square mesh*

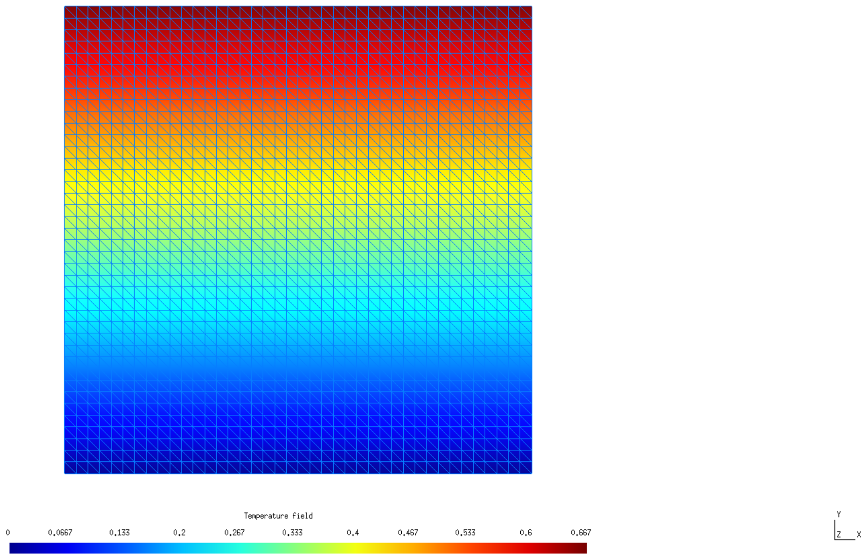

Figure 4 *Temperature distribution on 40x40 square mesh*

4. Analytical Solution

4.1. Solving the problem in 1 Dimension

It can be seen clearly from the FEM results that this problem has the same temperature anywhere along the x-axis. Therefore, the problem can be reduced from a 2D problem into a 1D problem in the domain [-L,L].

$$(S) = \begin{cases} div(\overline{q}) - r = 0 \\ \overline{q} = -K.\nabla u \\ u(x) = u_D(x)\ \forall\ x\ \in \partial\Omega_D \\ -\overline{q}.\overline{n} = q_N \quad \forall\ x\ \in \partial\Omega_N \end{cases} \tag{27}$$

$$\Rightarrow \quad div(-\mathrm{K}.\overline{\nabla} u) - r = 0 \qquad \Rightarrow \quad -\mathrm{K}\ div(\overline{\nabla} u) = r \quad (if\ \mathrm{K}\ is\ a\ constant) \tag{28}$$

$$\Rightarrow \quad \Delta u = \frac{-r}{\mathrm{K}} \qquad or \qquad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{-r}{\mathrm{K}} \tag{29}$$

Since we can simplify our problem into a 1D problem, the derivitave of *x* is equal to zero.

$$\therefore \quad \frac{\partial^2 u}{\partial y^2} = u'' = \frac{-r}{K} \tag{30}$$

$$u' = \frac{-ry}{\mathrm{K}} + A \tag{31}$$

$$u = \frac{-ry^2}{2K} + Ay + B \quad \forall\ y\ \in [-L, L] \tag{32}$$

This is the general solution of the problem. The exact solution can be found by considering boundary conditions.

Neumann Boundary Condition

The Neumann boundary on

$$From\ \ \overline{q} = -\mathrm{K}.\overline{\nabla} u \quad \text{and} \quad -\overline{q}.\overline{n} = q_N \quad \forall\ u\ \in \partial\Omega_N \tag{33}$$

$$\overline{\nabla} u = u' = \frac{q_N}{\mathrm{K}} \qquad \forall\ u\ \in \partial\Omega_N \tag{34}$$

$$\therefore \quad u'(L) = \frac{-rL}{\mathrm{K}} + A = \frac{q_N}{\mathrm{K}} \qquad when\ y = L \tag{35}$$

$$\Rightarrow \quad A = \frac{q_N + rL}{\mathrm{K}} \tag{36}$$

Dirichlet Boundary Conditions

$$u(-L) = \frac{-r(-L)^2}{2\mathrm{K}} + A(-L) + B = 0 \qquad when\ y = -L \tag{37}$$

$$\Rightarrow B = \frac{3}{2}\frac{rL^2}{\mathrm{K}} + \frac{q_N L}{\mathrm{K}} \tag{38}$$

Therefore the exact 1D problem can be written as follows:

$$u(y) = \frac{-ry^2}{2\mathrm{K}} + \frac{(q_N + rL)}{\mathrm{K}}\,y + \frac{3}{2}\frac{rL^2}{\mathrm{K}} + \frac{q_N L}{\mathrm{K}} \tag{39}$$

When source term *r = 0* the equation is simplified into a linear equation of the form:

$$u(y) = \frac{q_N}{\mathrm{K}} (y + L) \tag{40}$$

The exact analytical solution is calculated in the function *uFun()* like so:

```
# Returns exact solution based on Square or Bimat
 def uFun(y,K,qN,r,L,prev = 0, case = 0):
 out = 0
     if case == 0:   # Normal Square
         out = -(r*y**2/(2*K)) + (qN + r*L)*y/K + 3*r*L**2/(2*K) + qN*L/K
     elif case == 1: # Bimat PDE 1
         out =  r*(L**2-y**2)/(2*K) + qN*(y+L)/K
     elif case == 2: # Bimat PDE 2
         out =  -(r*y**2/(2*K)) + (qN + r*L)*y/K + prev
     return out
```

4.2. Extracting the FEM Solution

A seperate file *checkFE.py* was written to check the FEM results and compare it to the analytical solution (the above mentioned *uFun()* function is also included in this file). Since the results from the FEM calculation are random and hard to decode, it becomes difficult to find out the temperature at each node. Since this problem can be considered a 1D problem, only one value from each row of nodes is required. The built-in function *numpy.sort()* was employed to sort the values in order of lowest value to highest values, and then the lowest value from each row was considered the temperature at that particular 1D node. It was noticed from *solveFE.py* that the the variable *U* did not store the temperature of the nodes at the Dirichlet Boundary. Hence, a seperate function *getSize()* was written to get the true size of the mesh as the number of nodes on each side of the square mesh.

```
def getApprox1D(U,K,qN,r):
    N = getSize(len(U))
    u = np.zeros(N)
    U = np.sort(U)
    p = np.zeros(N)
    L = 1

    u = U[N-1:N*N-N:N]
    u = np.insert(u,0,0,axis=0)
    x = np.linspace(-L,L,N)

    K1 = K[0]
    p = uFunSquare(np.linspace(-L,L,N),K1,qN,r,L)
    return u, p

def getSize(c):
    a = 1
    b = -1
    c = -c
    d = b**2 - 4*a*c
    size = (-b+math.sqrt(d))/2*a
    return int(size)
```

4.3. Comparing FEM Solution to Analytical Solution

In order to check that our FEM solutions are accurate, it is important to compare them to the analytical solution and plot the error. We expect the errors to reduce as we increase the mesh density, i.e make the mesh finer. It was observed that the FEM solution was very close to the analytical solution when there was no source term present, but the accuracy of the error is more defined if there is a source term.

```
 # Finds exact solution based on square or bimat
 isBimat = 0
 if case == 0:
     newMesh = square
     xex, uex = getExactSoln_Square(K,qN,r)
     name = "square"
 if case == 1:
     newMesh = bimat
     isBimat = 1
     xex, uex = getExactSoln_Bimat(K,qN,r)
     name = "bimat"
 N = len(newMesh)
 e = np.zeros(N)
 a = np.zeros(N)
 h = np.zeros(N)

 # Plot exact solution
 plt.figure()
 plt.plot(xex,uex,label='Exact Solution',linewidth=4,color='black')

 # Find solution and errors for different mesh sizes
 for i in range(0,len(newMesh)):
     meshName = newMesh[i]
     U  =  solveFE(meshName,  conductivities,  BCNs,  BCD_lns,  BCD_nds,
sourceTerm, exportName, useSparse, verboseOutput)
     x, u, v = getApprox1D(U,K,qN,r,isBimat)
     e[i] = max(abs(v-u))
     a[i] = sum(abs(v-u))/len(u)

     print(meshName)
     print('Max Error = %g, Avg Error = %g\n'%(e[i],a[i]))
     plt.plot(x,u, label=newMesh[i])
     h[i] = len(u)

 plt.legend()
 plt.grid()
 plt.xlabel("Position y", fontsize = 12)
 plt.ylabel("Temperature u(y)", fontsize = 12)
 plt.title('FEM Deviations for r = %g'%r)
 figName = "Exact-vs-Approx_r=" + str(r) + "_"+name+".png"
 plt.savefig(figName, bbox_inches='tight', dpi=300)
 cmd = "convert "+figName+" "+figName[:-3]+"eps"     #  Save  as  eps  file
to import into groff document
 os.system(cmd)
```

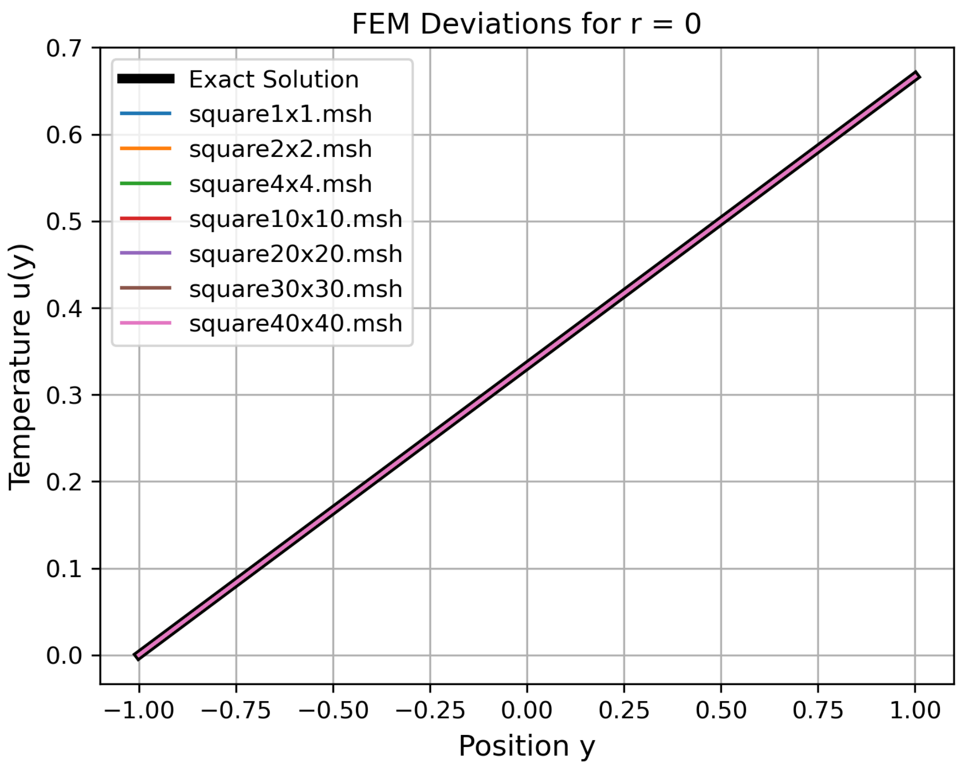

Figure 5 *Exact v.s. Approximate values for square meshes at r=0*

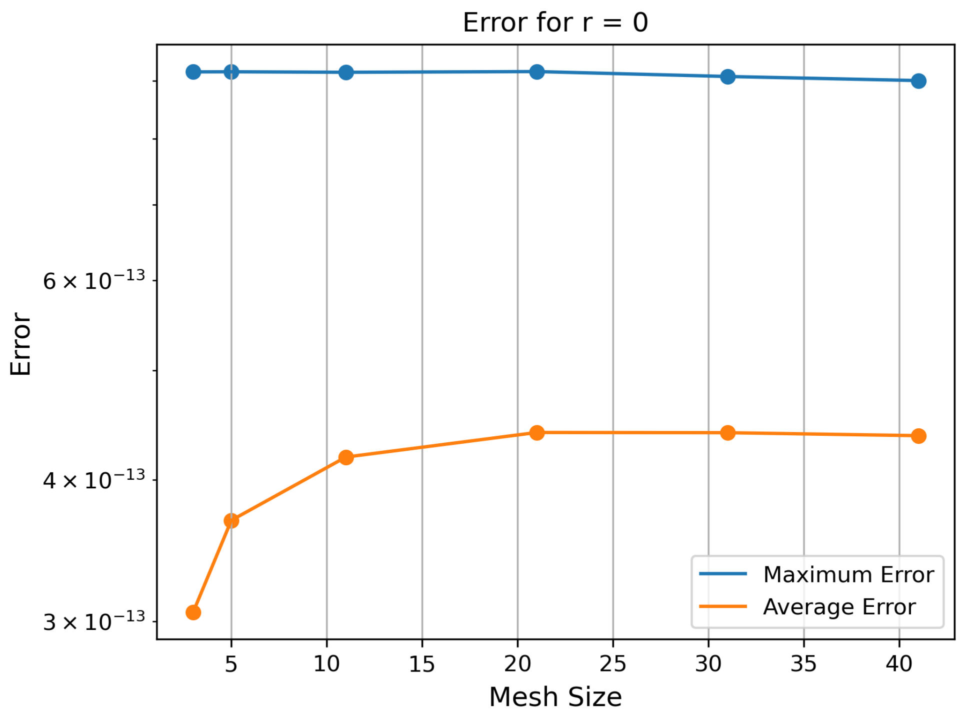

Figure 6 *Error plot for square meshes at r=0*

We can see here that when $r = 0$ the errors are extremely small and that the FEM solutions are difficult to distinguish from the analytical solutions.

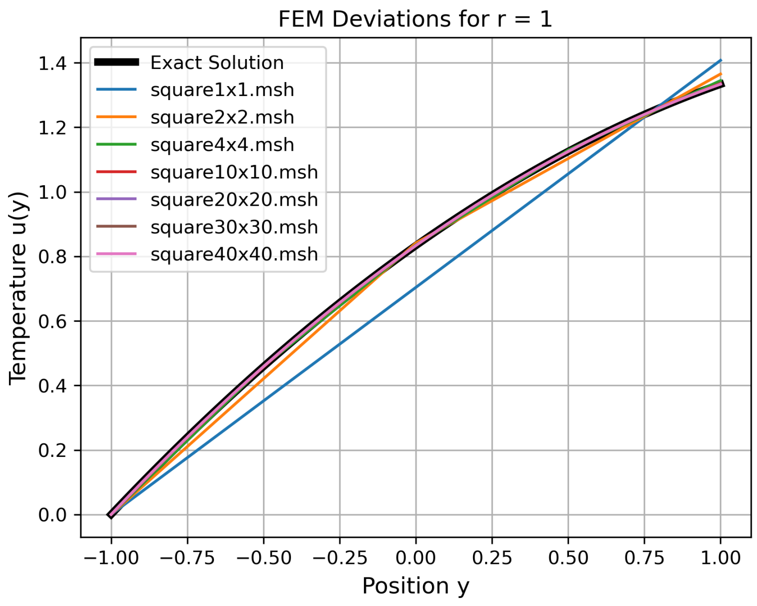

Figure 7 *Exact v.s. Approximate values for square meshes at r=1*

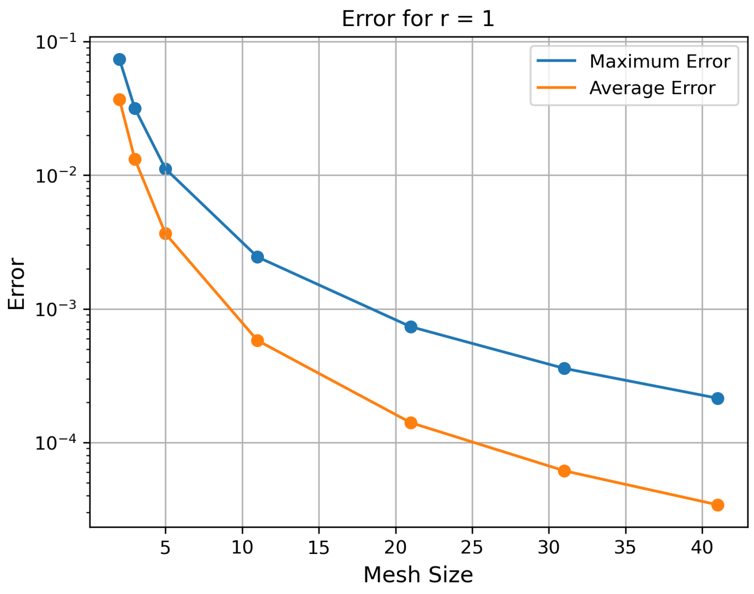

Figure 8 *Error plot for square meshes at r=1*

However, when *r = 1* the accuracy of the FEM solutions reduce significantly, and they can be distinguished from the analytical solution.

5. Comparing solution with Finite Difference (FD) Lab

It is relatively simple to redefine our FE problem as the same one from the FD problem. In order to do this, the Dirichelt Boundary Conditions on all 4 corners of the square were set to *u = 0* , the source term was set as *r = 1* , and the conductivity was set to a unit value. Upon doing this, a contour plot very similar to FD lab was obtained.

It was noted that:

- For the same number of nodes, **Gmsh seems to display a more accurate ouput which closely resembles the exact solution.** Ho wever, the actual error of the FEM solution is not much better than the FD solution. The graphs comparing the errors is shown below.
- **FEM was demonstrated to be significantly faster than FD.** The computing time required for FEM solution was much lower than FD solution for the same number of nodes. To compute the temperature across 5 meshes (N = 4, 10, 20, 30, 40), the FEM solution only took around *1.9 seconds* whereas the FD solution took approximately *6.2 seconds*, i.e. it was more than three times faster.

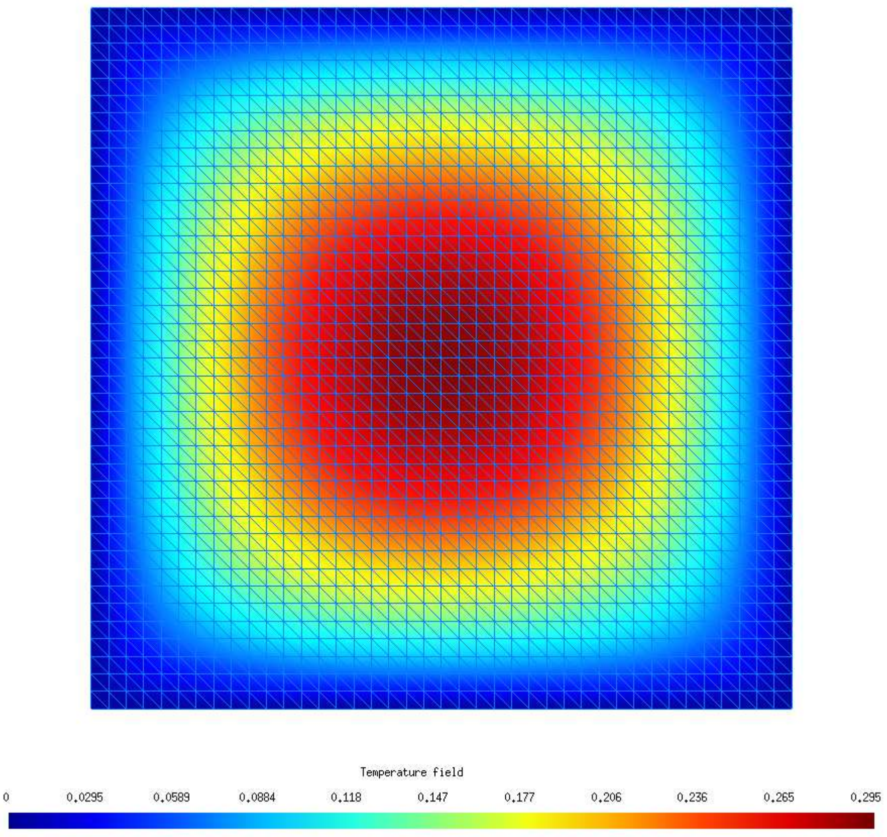

Figure 9 *Gmsh output for 40x40 mesh*

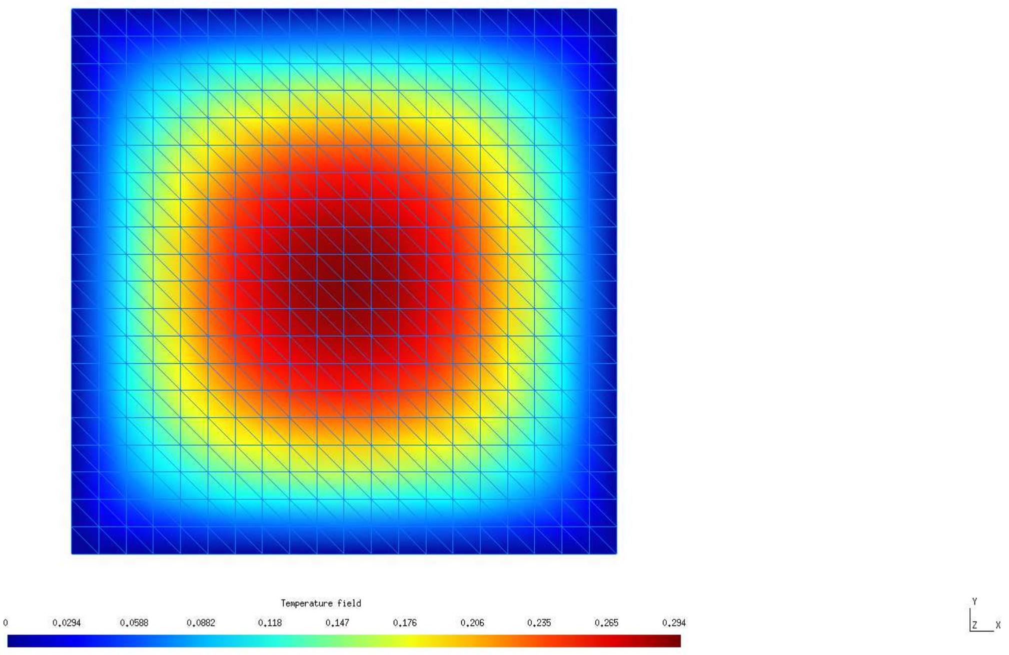

Figure 10 *Gmsh output for 20x20 mesh*

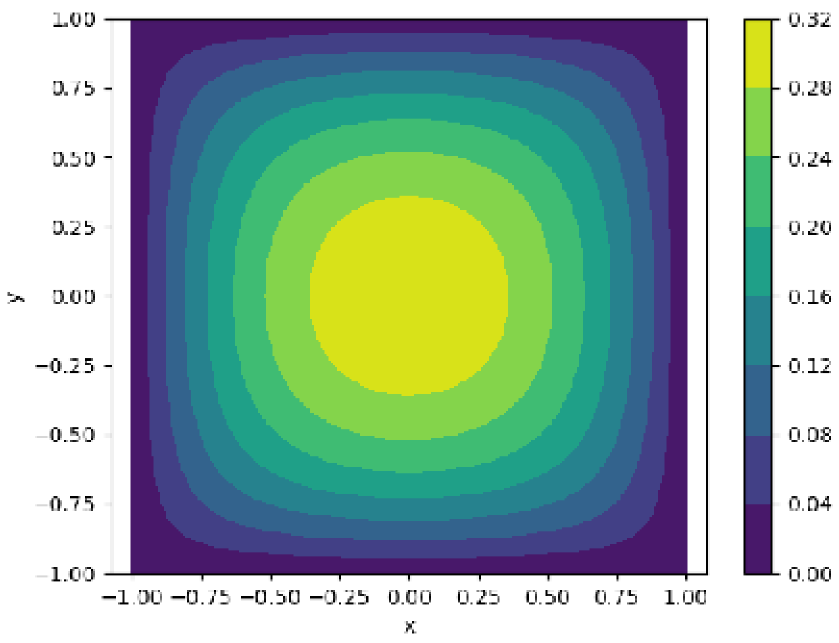

Figure 11 *FD Lab approximation plot for N = 20*

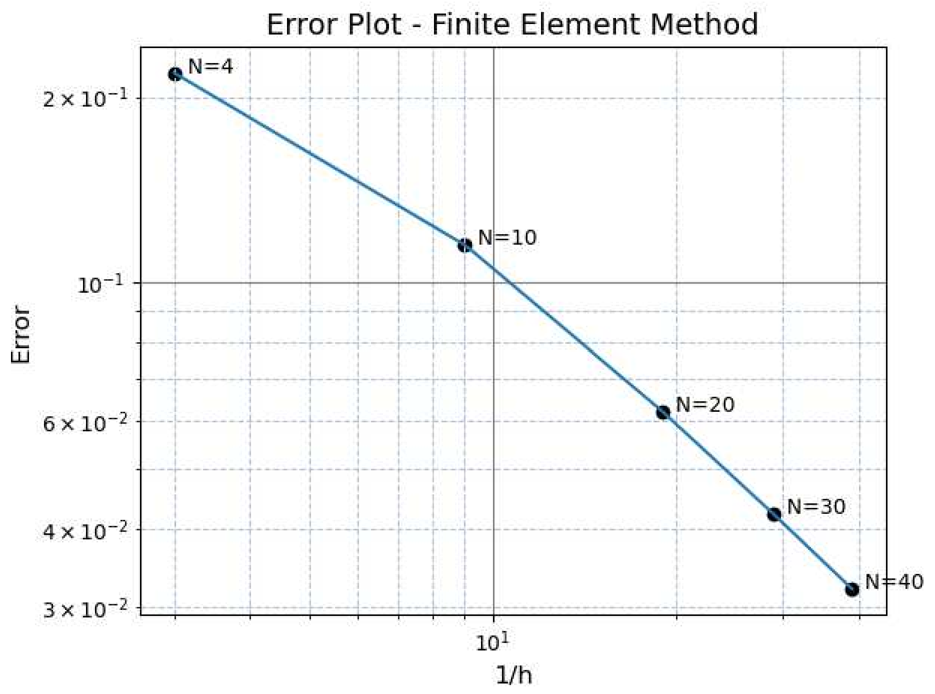

Figure 12 *Logarithmic error plot for FD*

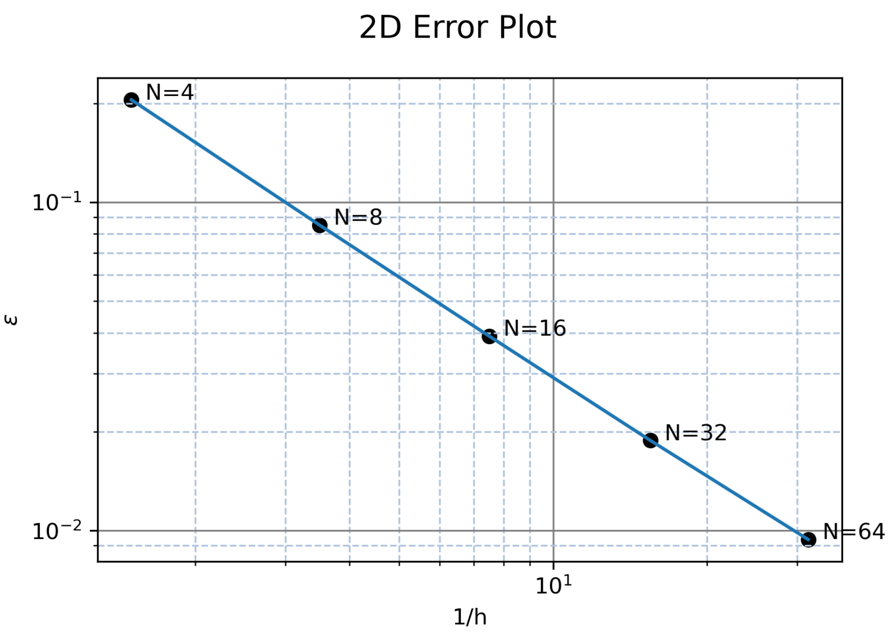

Figure 13 *Logarithmic error plot for FE*

From both of these error plots, we can see that the two methods have about the same accuracies, with FEM being only slightly more accurate.

6. FEM solution for two Isotropic Materials

6.1. Modifying the program

It is relatively simple to solve the problem for a mesh with two isotropic materials. The only change that was required in the program was to add the conductivity of the second element to the *conductivities* variable defined in *testFE.py.* In this case, a pre-defined element ID of 2000 was assigned a conductivity value of 1 W/(mm.K) was assgined.

```
#  Assigns  conductivities  K  =  [3,1,1.38e-3,1510e-3]  if  case  ==  0:
# Normal square(hole)
    conductivities    =    {1000:    K[0]}    elif    case    ==    -1:
# FD Problem
    conductivities    =    {1000:    1}    elif    case    ==    1:
# Bimat and Inclusion
    conductivities  =  {1000:  K[0],  2000:  K[1]}  elif  case  ==  2:
# Circuit
    conductivities = {1000: K[2], 2000: K[3]}
```

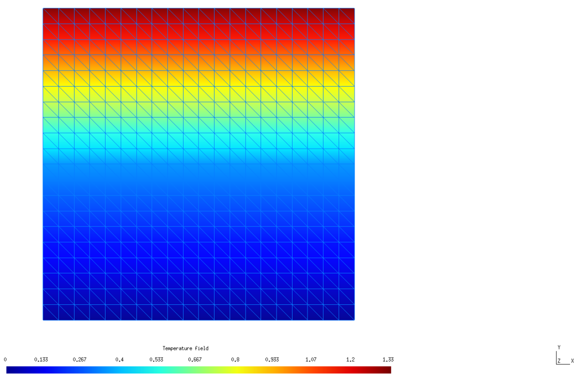

Figure 14 *Temperature gradient for 20x20 Bimaterial square mesh*

6.2. Analytical Solution for two Isotropic Materials

The exact solution across a domain with two isotropic materials must be found by defining two different PDEs - one for each material. If we reduce the problem to a 1D problem like before and split up the domain equally, the first material esists in the domain [-L,0] and the second material exists in a domain [0,L]. Consider the same general solution as a domain with single conductivity, but change the boundary conditions to match this change in boundary.

$$u(y) = \frac{-ry^2}{2\mathrm{K}} + Ay + B \qquad \text{and} \qquad u'(y) = \frac{-ry}{\mathrm{K}} + A \tag{41}$$

First Material [-L,0]

$$A = \frac{q_N}{\mathrm{K}_1} \qquad B = \frac{rL^2}{2\mathrm{K}_1} + \frac{q_N L}{\mathrm{K}_1} \tag{42}$$

$$\therefore \quad u(y) = \frac{r}{2\mathrm{K}_1}(L^2 - y^2) + \frac{q_N}{\mathrm{K}_1}(y + L) \qquad \forall\ y \in\ [-L, 0] \tag{43}$$

Second Material [0,L]

$$A = \frac{rL + q_N}{\mathrm{K}_2} \qquad B = T_b\ \ (Temperature\ at\ Boundary) \tag{44}$$

$$\therefore \quad u(y) = \frac{-ry^2}{2\mathrm{K}_2} + \frac{(rL + q_N)}{\mathrm{K}_2}\, y + T_b \qquad \forall\ y \in\ [0, L] \tag{45}$$

We can compare our FEM solution against the analytical solution, and errors can be plotted:

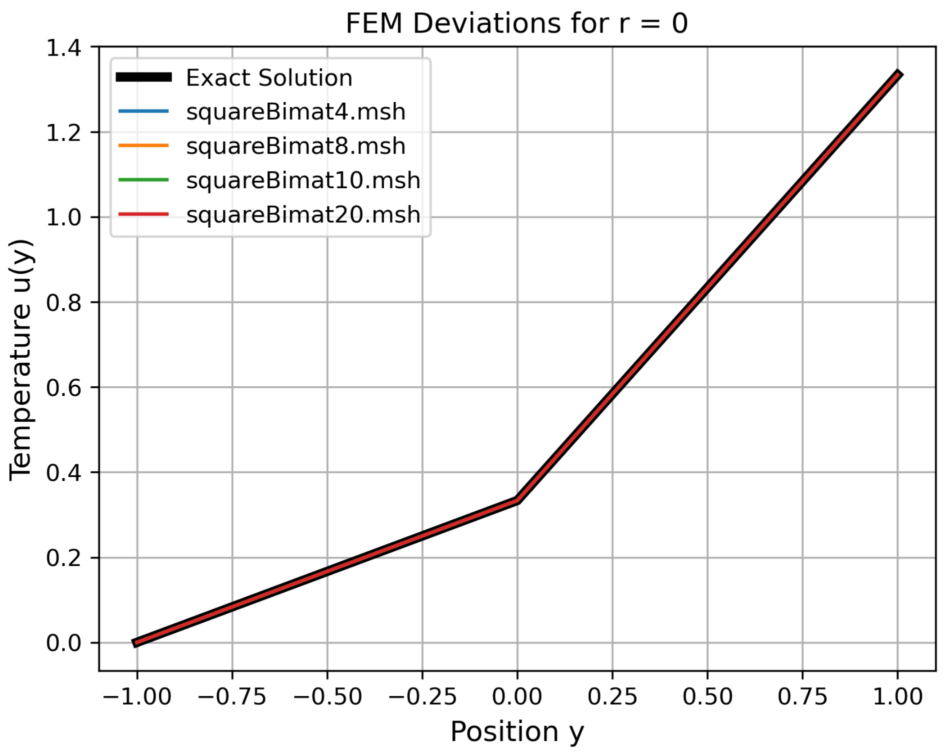

Figure 15 *Exact v.s. Approx values for bimaterial mesh*

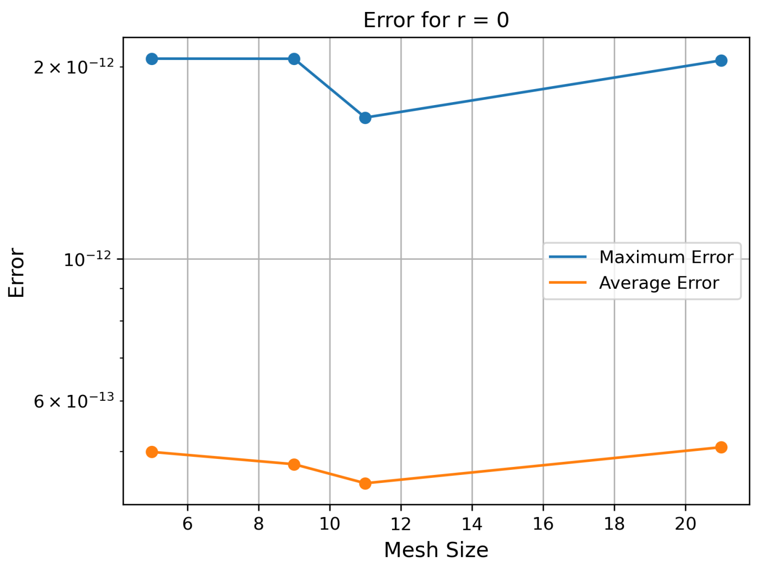

Figure 16 *Error plot of bimaterial mesh*

Again, we see that when source term is equal to zero, the deviations from the analytical solution are extremely small to a point where they can be neglected. In the graph of*P osition y v.s. Temperature u(y)* we see that the there is a **sudden change in slope of the graph** at $y = 0$, i.e. at the boudnary of the two materials. This is where the conductivity changes from 3 W/(mm.K) to 1 W/(mm.K).

7. Temperature fields of meshes with hole/inclusion

/home/sudarson/Study/Sem1/NUMME/NUMMEFELab2020-2021/outputs/temp-squareHole60.eps

squareHole60.msh : r = 0

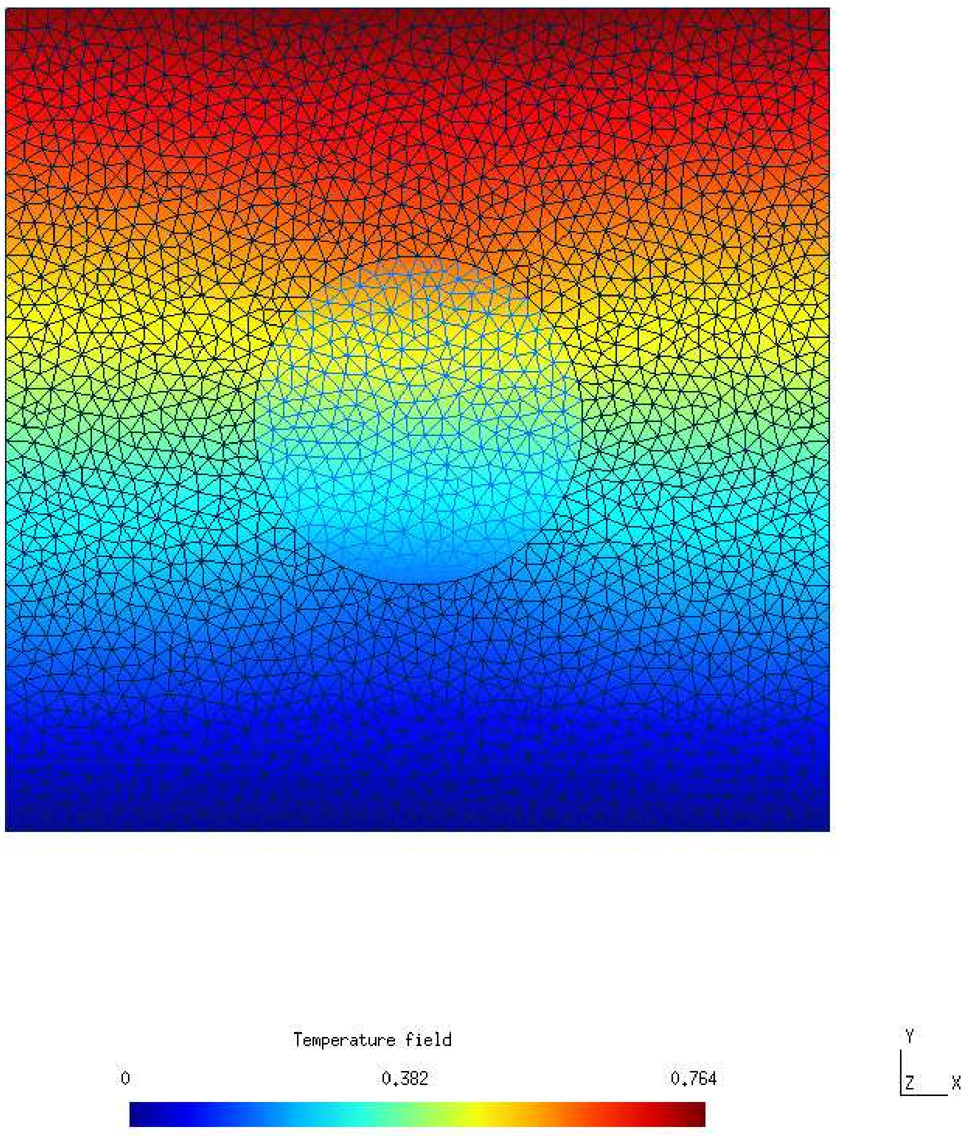

Figure 17 *SquareInclusion40.msh*

8. Solving the Heating Circuit

The conditions for the heating circuit are:

- Size of the plate is 4mm x 2.1mm x 0.05mm, but due to how thin the circuit is, it is modelled as a 2D problem
- Heat source *r* = 1 W/mm^{-2}
- Temperature on the lateral sides = 293K, i.e. Dirichlet Boundary Conditions are fixed at 293K
- Conductivity of Glass = 1.38 10^{-3} W/(mm.K), and Conductivity of Nichrome = 15 10^{-3} W/(mm.K)

8.1. Propose a Converged Solution

In order to measure convergence, the max temperature variation across the mesh was recorded.

- **Circuit 0** - Difference: 292.198 - 538.813 = 246.615 K, **Convergence = 26.0416%**
- **Circuit 1** - Difference: 228.25 - 539.087 = 310.837 K, **Convergence = 12.2018%**
- **Circuit 2** - Difference: 266.342 - 539.252 = 272.91 K, **Convergence = 2.42835%**
- **Circuit 3** - Difference: 273.007 - 539.29 = 266.283 K

From this we can see that convergence of less than 5% accuracy is only reached as we reach **Circuit 2.**

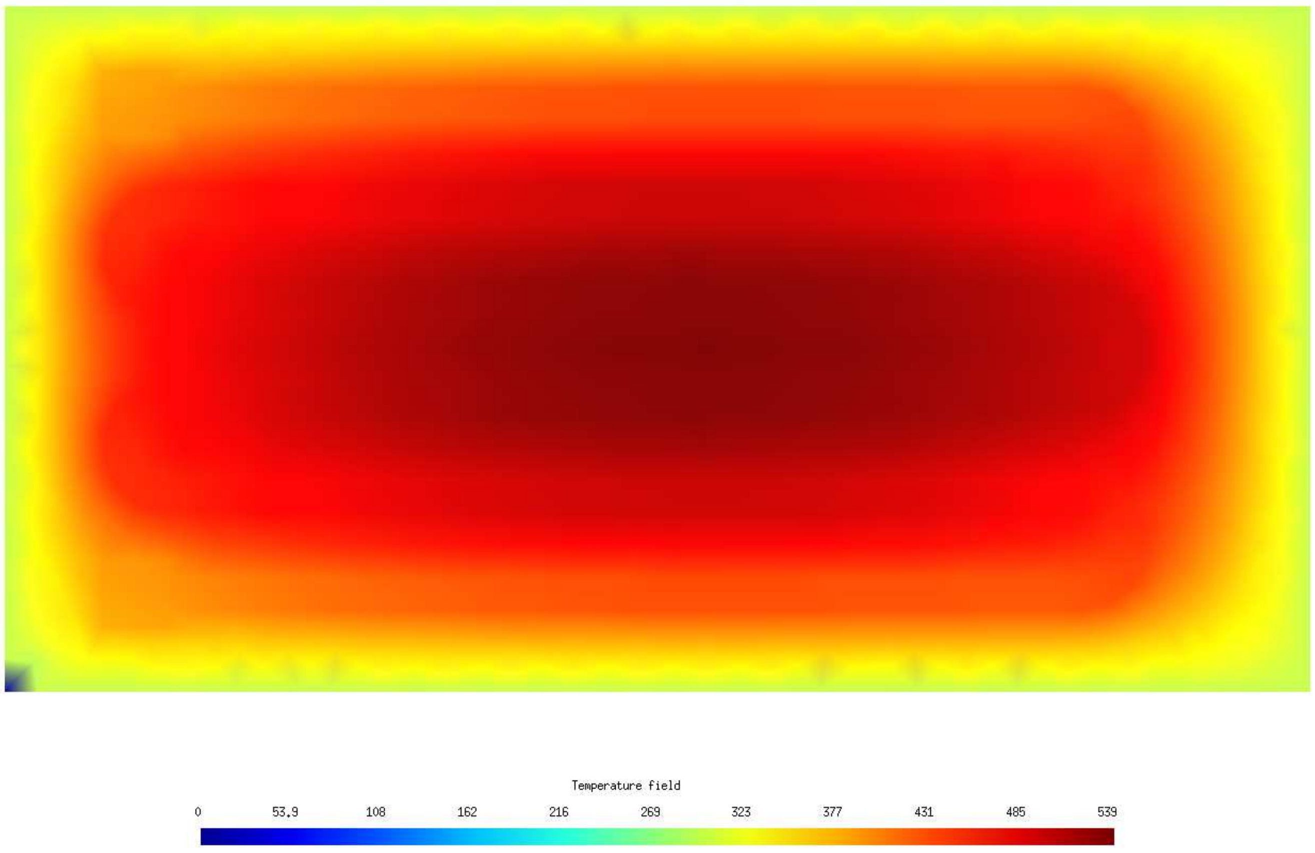

Figure 18 *Gmsh Temperature field for circuit0.msh*

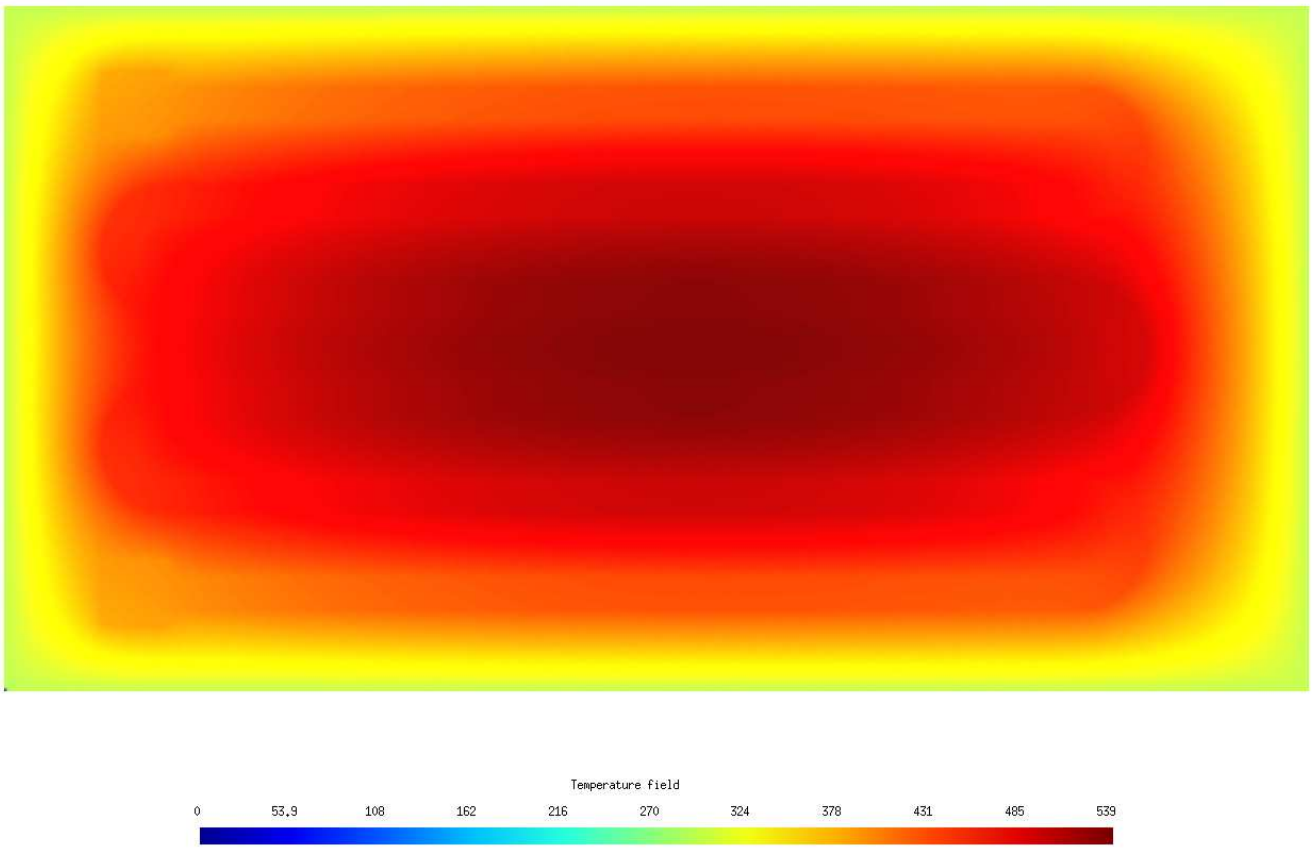

Figure 19 *Gmsh Temperatre field for circuit3.msh*

We see a cleaner boundary in Circuit 3 as compared to Circuit 0 because the mesh is much finer.

8.2. Range of Temperature Variation

Temperature variation for each circuit mesh was mentioned in the previous section. The variation has been plotted below. Here, we can clearly see that the variation reduces as we increase the density of the mesh, and why the FEM solution converges at *Circuit 2*.

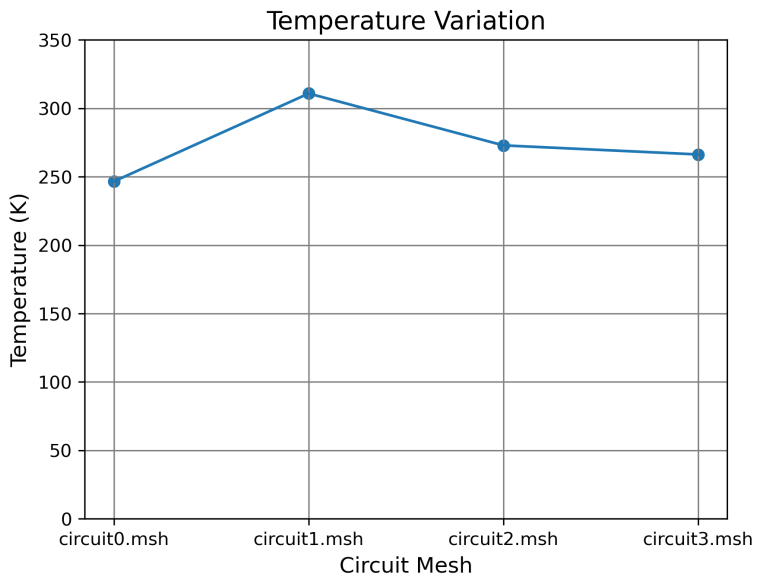

Figure 20 *Temperature Variation across circuits*

9. User Scripts

testFE.py

This is the main python script that calls all other functions. Since this script needs to execute different functions based on different meshes and boundary conditions, there are a set of "control variables" at the beginning of the script which let the user test for different conditions quickly.

```
import os
import time
import matplotlib.pyplot as plt
import numpy as np

#Import finite element solver function
from solveFE import solveFE
from checkFE import *

np.set_printoptions(linewidth=300)
np.set_printoptions(precision=3, suppress=True)

start = time.time()

# List to store all the mesh files
meshes = [
            "square1x1.msh",            #0
            "square2x2.msh",            #1
            "square4x4.msh",            #2
            "square10x10.msh",          #3
            "square20x20.msh",          #4
            "square30x30.msh",          #5
            "square40x40.msh",          #6

            "squareHole20.msh",         #7
            "squareHole30.msh",         #8
            "squareHole40.msh",         #9
            "squareHole60.msh",         #10

            "squareBimat4.msh",         #11
            "squareBimat8.msh",         #12
            "squareBimat10.msh",        #13
            "squareBimat20.msh",        #14

            "squareInclusion20.msh",    #15
            "squareInclusion40.msh",    #16

            "circuit0.msh",             #17
            "circuit1.msh",             #18
            "circuit2.msh",             #19
            "circuit3.msh",             #20
          ]
```

```
#--------------------------------------------------------#
#            These values control the program            #
#--------------------------------------------------------#

meshNumber = 2
openMesh = 0
displayU = 0
isFD = 0
plotErrors = 1
showPlot = 1
useSparse = True           # Reduce memory consumption
verboseOutput = False      # Reduce console output

#--------------------------------------------------------#

# Stores the Meshes in seperate lists
square = meshes[:7]
squareHole = meshes[7:11]
bimat = meshes[11:15]
inclusion = meshes[15:17]
circuit = meshes[17:21]

meshName = meshes[meshNumber]

# Sets the case based on chosen mesh
if isFD == 1:
    case = -1
elif meshNumber <=10 and isFD == 0:
    case = 0
elif meshNumber > 10 and meshNumber <= 16:
    case = 1
elif meshNumber > 16 and meshNumber <= 20:
    case = 2


# Assigns Neumann Boundary Conditions
if case == -1:
    BCNs = {103: 0}
else:
    BCNs = {103: 1}
    qN = 1


# Assigns Dirichlet Boundary Conditions
if case == -1:
    BCD_lns = {101: 0, 102: 0, 103: 0, 104: 0}
elif case == 2:
    BCD_lns = {101: 293, 102: 293, 103: 293, 104: 293}
else:
    BCD_lns = {101: 0}
BCD_nds = {1: 0}


# Assings conductivities
```

```
 K = [3,1,1.38e-3,15e-3]
 if case == 0:                            # Normal square or square hole
     conductivities = {1000: K[0]}
 elif case == -1:                                   # FD Problem
     conductivities = {1000: 1}
 elif case == 1:                                    # Bimat and Inclusion
     conductivities = {1000: K[0], 2000: K[1]}
 elif case == 2:                                    # Circuit
     conductivities = {1000: K[2], 2000: K[3]}


 # Assigns source term
 if case == -1 or case == 2:
     r = 1
 else:
     r = 0
 sourceTerm = lambda xyz, physdom: r


 # Name of temperature file to be stored
 name = meshName[:-4]
 exportName = ’outputs/temp’ + ’-’ + name + ’.pos’

 # Calls function to calculate temperature and export to .pos
 if plotErrors == 0:
     U = solveFE(meshName, conductivities, BCNs, BCD_lns, BCD_nds, sourceTerm, ex-
portName, useSparse, verboseOutput)

     if case == -1:
         FD_getMaxError(U)

     # Open .pos file in Gmsh
     if openMesh == 1:
         cmd = "gmsh " + exportName + " &"
         os.system(cmd)


 # Plots errors based on the case
 # e stores the maximum error
 # a stores the average error
 # h = 1/h or the number of nodes
 else:

     # Plot the error for the Finite DIFFERENCE problem
     if case == -1:
         newMesh = square
         newMesh = np.delete(newMesh,0)
         newMesh = np.delete(newMesh,0)
         N = len(newMesh)
         e = np.zeros(N)
         h = np.zeros(N)

         for i in range(0,N):
             meshName = newMesh[i]
```

```
            U  =  solveFE(meshName,  conductivities,  BCNs,  BCD_lns,  BCD_nds,
sourceTerm, exportName, useSparse, verboseOutput)
            e[i], h[i] = FD_getMaxError(U)

        errFig = plt.figure()
        ax = errFig.add_subplot(111)
        plt.loglog(h,e)
        ax.scatter(h,e,color='black')
        j = 0
        for xy in zip(h, e):
            ax.annotate('  N=%i' % (h[j]+1), xy=xy, textcoords='data')
            j += 1

        plt.xlabel('1/h', fontsize=12)
        plt.ylabel('Error', fontsize=12)
        plt.title("Error Plot - Finite Element Method", fontsize=14)
        plt.grid(b=True, which='major', color='grey', linestyle='-')
        plt.grid(b=True, which='minor', color='lightsteelblue', linestyle='--')
        plt.tight_layout()
        figName = "FD_Error.png"
        plt.savefig(figName, bbox_inches='tight',dpi=300)
        end = time.time()
        cmd = "convert "+figName+" "+figName[:-3]+"eps"     # Save as eps file to
import into groff document
        os.system(cmd)
        if showPlot:
            plt.show()

    # Plot error for CIRCUIT
    elif case == 2:
        newMesh = circuit
        N = len(newMesh)
        diff = np.zeros(N)

        for i in range(0,N):
            U  =  solveFE(newMesh[i],  conductivities,  BCNs,  BCD_lns,  BCD_nds,
sourceTerm, exportName, useSparse, verboseOutput)

            diff[i] = getTempDiff(U)
            print(diff[i])

            if i > 0:
                conv = abs((diff[i]-diff[i-1])/diff[i-1])
                print("Convergence = %g%%"%(conv*100))

        plt.figure()
        plt.plot(newMesh,diff)
        plt.scatter(newMesh,diff)
        plt.ylim([0, 350])
        plt.xlabel('Circuit Mesh', fontsize=12)
        plt.ylabel('Temperature (K)', fontsize=12)
        plt.title("Temperature Variation", fontsize=14)
        plt.grid(b=True, which='major', color='grey', linestyle='-')
        plt.grid(b=True, which='minor', color='lightsteelblue', linestyle='--')
```

```
         figName = "Circuit_Temp_Variation.png"
         plt.savefig(figName, bbox_inches='tight', dpi=300)
         cmd = "convert "+figName+" "+figName[:-3]+"eps"     # Save as eps file to
import into groff document
         os.system(cmd)
         if showPlot:
             plt.show()

     # Plot error for SQUARE and BIMAT
     else:
         # Finds exact solution based on square or bimat
         isBimat = 0
         if case == 0:
             newMesh = square
             xex, uex = getExactSoln_Square(K,qN,r)
             name = "square"
         if case == 1:
             newMesh = bimat
             isBimat = 1
             xex, uex = getExactSoln_Bimat(K,qN,r)
             name = "bimat"
         print("isBimat = ",isBimat)
         N = len(newMesh)

         # Stores maximum and average error
         e = np.zeros(N)
         a = np.zeros(N)
         h = np.zeros(N)

         # Plot exact solution
         plt.figure()
         plt.plot(xex,uex,label='Exact Solution',linewidth=4,color='black')

         # Find solution and errors for different mesh sizes
         for i in range(0,len(newMesh)):
             meshName = newMesh[i]
             U   =   solveFE(meshName,   conductivities,   BCNs,   BCD_lns,   BCD_nds,
sourceTerm, exportName, useSparse, verboseOutput)
             x, u, v = getApprox1D(U,K,qN,r,isBimat)
             e[i] = max(abs(v-u))
             a[i] = sum(abs(v-u))/len(u)

             print(meshName)
             print('Max Error = %g, Avg Error = %g\n'%(e[i],a[i]))
             plt.plot(x,u, label=newMesh[i])
             h[i] = len(u)

         plt.legend()
         plt.grid()
         plt.xlabel("Position y", fontsize = 12)
         plt.ylabel("Temperature u(y)", fontsize = 12)
         plt.title('FEM Deviations for r = %g'%r)
         figName = "Exact-vs-Approx_r=" + str(r) + "_"+name+".png"
         plt.savefig(figName, bbox_inches='tight', dpi=300)
         cmd = "convert "+figName+" "+figName[:-3]+"eps"     # Save as eps file to
import into groff document
         os.system(cmd)

         if showPlot:
             plt.show()

         if case == 0 and r == 0:
             newMesh.pop(0)
             e = np.delete(e,0)
             a = np.delete(a,0)
             h = np.delete(h,0)
         plt.figure()
         for i in range(0,len(newMesh)):
             newMesh[i] = newMesh[i][6:]
             newMesh[i] = newMesh[i][:-4]
         plt.plot(h,e, label='Maximum Error')
         plt.plot(h,a, label='Average Error')
         plt.scatter(h,e)
         plt.scatter(h,a)
         plt.yscale("log")
         plt.xlabel("Mesh Size", fontsize=12)
         plt.ylabel("Error", fontsize=12)
         plt.title('Error for r = %g'%r)
         plt.legend()
         plt.grid()
         figName = "ErrorPlot_r="+str(r)+"_"+name+".png"
         plt.savefig(figName,bbox_inches='tight',dpi=300)
         cmd = "convert "+figName+" "+figName[:-3]+"eps"     # Save as eps file to
import into groff document
         os.system(cmd)

         end = time.time()

         if showPlot:
             plt.show()

 try:
     end
 except NameError:
     end = time.time()
 print('\n\nTime Elapsed = %g\n\n'%(end-start))
```

checkFE.py

This script containts user functions used to extract approximate and exact values of U for all the different cases in the FE Lab problem.

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

# This function sorts the values of U
# and extracts 1D values from it
def getApprox1D(U,K,qN,r,isBitmat = 0):
    N = getSize(len(U))
    u = np.zeros(N)     # Stores the 1D values
    U = np.sort(U)      # Sort U values
    p = np.zeros(N)     # Exact values for 1D nodes
    L = 1               # Size of the boundary from -1 to 1

    u = U[N-1:N*N-N:N]  # Extract 1D values
    u = np.insert(u,0,0,axis=0)    # Add BCD
    x = np.linspace(-L,L,N)

    # Two different PDEs need to be used in the case of two isotropic materials
    if isBitmat == 1:
        K1 = K[0]
        K2 = K[1]
        N = int(N/2)+1
        p1 = uFun(np.linspace(-L,0,N),K1,qN,r,L,case=1)
        p2 = uFun(np.linspace(0,L,N), K2,qN,r,L,prev=p1[-1],case=2)
        p2 = np.delete(p2,0)
        p   = np.concatenate((p1,p2),axis=0)

    # Use standard PDE for single material
    else:
        K1 = K[0]
        p = uFun(np.linspace(-L,L,N),K1,qN,r,L,case=0)

    return x, u, p

# Returns exact FD values for a given NxN mesh size
# Taken from FD Lab script
def FDExact(L,n):
    U = np.zeros((n,n))
    N = 5
    X = np.linspace(-L,L,n)
    Y = np.linspace(-L,L,n)
    for p in range(0,n):
        for q in range(0,n):
            x = X[p]
            y = Y[q]
            out = 0
            for i in range(1,2*N,2):
                for j in range(1,2*N,2):
                    out        +=        (64/(i*j*(i**2+j**2)*math.pi**4))        *
math.sin(i*math.pi*(x+1)/2) * math.sin(j*math.pi*(y+1)/2)
```

```
                    U[p][q] = out

    return U

# Compares the FEM solution to the Analytical solution
def FD_getMaxError(U):
    N = getSize(len(U))
    U = np.sort(U)
    L = 1

    Uex = FDExact(L,N)      # Stores exact temperature values
    Uex = Uex.flatten()
    U = np.sort(U)          # Sorts U values like in getAprrox1D()
    Uex = np.sort(Uex)

    max = 0
    for i in range(0,N*N):
        temp = abs(U[i] - Uex[i])
        if temp > max:
            max = temp
    print('N = %i, Error = %g\n'%(N,max))
    return max, N

# Returns size of the mesh given the number of elements
def getSize(c):
    a = 1
    b = -1
    c = -c
    d = b**2 - 4*a*c
    size = (-b+math.sqrt(d))/2*a
    return int(size)

# Returns exact solution based on Square or Bimat
def uFun(y,K,qN,r,L,prev = 0, case = 0):
    out = 0
    if case == 0:   # Normal Square
        out = -(r*y**2/(2*K)) + (qN + r*L)*y/K + 3*r*L**2/(2*K) + qN*L/K
    elif case == 1: # Bimat PDE 1
        out =  r*(L**2-y**2)/(2*K) + qN*(y+L)/K
    elif case == 2: # Bimat PDE 2
        out =  -(r*y**2/(2*K)) + (qN + r*L)*y/K + prev
    return out

# Returns high precision values for Square mesh
def getExactSoln_Square(K,qN,r):
    n = 100
    L = 1
    K1 = K[0]
    xex = np.linspace(-L,L,n)
    uex = uFun(xex,K1,qN,r,L,case=0)
    return xex, uex

# Returns high precision values for Bimat mesh
def getExactSoln_Bimat(K,qN,r):
    n = 100
    L = 1
    K1 = K[0]
    K2 = K[1]

    x1 = np.linspace(-L,0,n)
    x2 = np.linspace(0,L,n)
    u1 = uFun(x1,K1,qN,r,L,case=1)
    u2 = uFun(x2,K2,qN,r,L,prev=u1[-1],case=2)

    uex = np.concatenate((u1,u2),axis=0)
    xex = np.concatenate((x1,x2),axis=0)
    return xex, uex

# Returns max temperature variation
def getTempDiff(U):
    U = np.sort(U)
    p = U[0]
    q = U[len(U)-1]
    print("Difference: %g - %g = %g" %(p,q,abs(p-q)))
    return abs(p-q)
```

Table of Contents