Institute of Lightweight Design and Structural Biomechanics

Numerical Simulation and Scientific Computing II - Exercise 4

Group members:
Mario Hiti, 01327428
Markus Hickel, 01026326
Markus Renoldner, 01621777

Submission: June 10, 2021

Contents

1 Introduction

A FEM program is to be developed and implemented to solve plane, steady state heat conduction problems. The example follows closely the text book by Zienkiewic [1].
Chapter 5 gives all required information and the derived equations as well as pre-computed variables.
Steady state heat conduction (see [1], Chpt. 5) is described by the partial differential equation,

$$-\frac{\partial}{\partial x_i} q_i + Q = 0 \tag{1}$$

together with the constitutive law,

$$q_i = -k_{ij} \frac{\partial}{\partial x_j} T \tag{2}$$

where q_i is the heat flux vector, Q the power density, T the temperature, and k_{ij} the second order conductivity tensor. As boundary conditions, only heat flux across the boundary and prescribed temperatures at the boundary, respectively, will be considered.
We obtain

$$\frac{\partial}{\partial x_i}\left(k_{ij} \frac{\partial}{\partial x_j} T\right) + Q = 0 \tag{3}$$

For plane heat conduction problems $i, j = 1, 2$, which implies $\frac{\partial}{\partial x_3} T = 0$ and $q_3 = 0$. Nevertheless, three-dimensional situations with a certain thickness are considered in terms of physics (at least in the mind set of an engineer).

Isotropic material properties are treated, i.e. $k_{11} = k_{22}$ and $k_{12} = k_{21} = 0$, which are temperature independent but location (element) dependent.

2 Software Architecture

We designed the software to follow a data oriented approach and rely less on object oriented paradigms. Therefore our program consists of only two modules:

- **mesh.py**: contains all data related to the mesh
- **magicsolver.py**: Our solver for linear systems of equations with variables on both sides of the equal sign

2.1 Important components

2.1.1 Mesh class

This class holds all data that describes a the mesh, its nodes and how they are connected. Data is either stored in vectors of size N or in matrices of size N x N with $N = nodes_x \cdot nodes_y$.

Algorithm 1: Data structure

```
class Mesh:
    # Matrices
    index_mat = np.array    # represents the index position of every node
    adj_mat = np.array  # adjacency matrix
    stiff_mat = np.array # global stiffness matrix

    # Nodal data
    nodal_temps = []
    nodal_forces = []
    nodal_coords_x = np.array
    nodal_coords_y = np.array

    # face data
    face_ids = np.array # number of every face [1,2,3,...]
    face_k = np.array   # k value for a given face_id
    face_center_x = np.array    # x coordinate of the center of every face
    face_center_y = np.array    # y coordinate of the center of every face
    face_flux_x = np.array  # x component of the flux
    face_flux_y = np.array  # y component of the flu
    face_gradient_x = np.array # x component of the gradient
    face_gradient_y = np.array # y component of the gradient

    def __init__(self, variation=None):
        # Fill data according to a give variation

    def solve(self):
        # Solve the LSE and calculate missing results
```

During initialization basic data such as nodal positions, thermal conductivities, matrices are generated. Then a dedicated solver (see Section 2.2) is applied to solve the linear system of equations. Finally some post-processing steps such as calculation of the flux are performed.
A mesh class can be instanciated and calculated as follows:

Algorithm 2: API Example

```
# Instanciate a mesh according to variation "V0"
# possible variations are: V0, V1, V2, V3, V4a, V4b
mesh = Mesh(V0)
# calculate unknowns by solving the LSE
mesh.solve()

# Plot functions
mesh.plot_flux()
mesh.plot_gradient()
mesh.plot_conservation_of_flow()
```

Note that for *V0* addidional sanity checks according to Section 5.5 of the assignment are performed.

2.1.2 Adjacency Matrix Generator

In order to determine which nodes on the mesh are directly connected to each other and therefore are directly exchanging heat energy, we will generate a so-called adjacency matrix. For a squared domain of n nodes, for each node we will define whether any other node is a neighbour or not. Therefore, the adjacency matrix $\boldsymbol{A}$ is a symmetric (n, n) matrix.

In the assignment, the mesh was defined as in Figure 1.

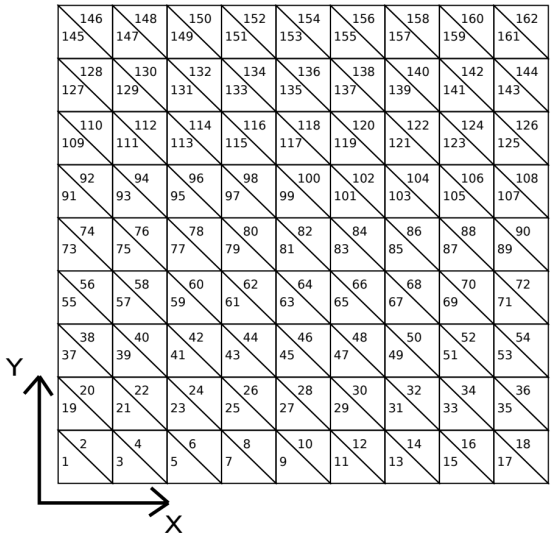

Fig. 1: Mesh for Task 1

For $n = 10$, we get a 10 by 10 adjacency matrix, that has 0 for each combination of the row and column where the two nodes with the respective node numbers are not neighbours, and e.g. a 1 in case they are neighbours. So, to recap: the actual squared domain consisting of n nodes has a side length of $\sqrt{n}$, while the adjacency matrix is of size n by n.

$\boldsymbol{A}$ would look like this:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{4}$$

The implementation in Python3 can be seen below.

At first, the matrix is being filled ignoring the boundary nodes. The boundary nodes are special, because they do not have all 6 neighbours, like center nodes. For the center nodes, the filling process results in several diagonals.

Using two extra loops, all non-existing neighbours for boundary nodes are being deleted again.

Algorithm 3: adjacency generator

```
1  def generate_adjacency(nnodes):
2
3      if int(nnodes**0.5)*int(nnodes**0.5)!=nnodes: return
4
5      n = int(nnodes**0.5)
6      A = numpy.zeros([nnodes,nnodes])
7
8      #fill matrix neglecting boundary nodes
9      hshift = 1 #horizontal shift
10     vshift = n #vertical shift
11     dshift = n - 1 #diagonal shift from bottom right to upper left
12     dshift2 = n + 1 #diagonal shift other direction (not used!)
13     for i in range(nnodes-hshift): #horizontal
14         A[i,i+hshift] = 1
15         A[i+hshift,i] = 1
16     for i in range(nnodes-vshift): #vertical
17         A[i,i+vshift] = 1
18         A[i+vshift,i] = 1
19     for i in range(nnodes-dshift): #diagonal
20         A[i,i+dshift] = 1
21         A[i+dshift,i] = 1
22
23     #correct boundary nodes
24     hskip = n
25     dskip = n
26     for i in range(n,nnodes-hshift,hskip): #horizontal
27         A[i-hshift,i] = 0
28         A[i,i-hshift] = 0
29     #vertikal muss nicht korrigert werden
30     for i in range(n-1,nnodes,dskip): #diagonal
31         A[i-dshift,i] = 0
32         A[i,i-dshift] = 0
33
34     return A
```

2.1.3 Matrix-Solver

The total linear system of equations consists of sub-matrices and sub-vectors with known and unknown entries, respectively,

$$\begin{bmatrix} & & \\ & \boldsymbol{H} & \\ & & \end{bmatrix} \begin{bmatrix} T_1 \\ \vdots \\ T_n \end{bmatrix} = \begin{bmatrix} P_1 \\ \vdots \\ P_n \end{bmatrix} \tag{5}$$

In this equation, $\boldsymbol{H}$ is the so called stiffness matrix from continuum mechanics, which is also a sparse, symmetric matrix with the same non-zero elements as the adjancency matrix $\boldsymbol{A}$.
T_i and P_i are the temperature and reaction force vectors.
For some nodes we know the value of T_i and for all other nodes we know the value of P_i. This requires a matrix solver, that takes that into account.

We implemented our version of this type of matrix solver below, see 2.2.

At first, the unknown and known values and there indices of T_i and P_i are being parsed.

Then, the unknown values of T_i are being computed, by creating a submatrix from $\boldsymbol{H}$ (which is called A in the code). Before solving for the unknown values of T_i, the right hand side needs to be corrected using the known values of P_i. Then we used numpy's linalg.solve method.

Next, the unknown values of P_i are being computed simply by using numpy's linalg.dot.

After each of the two computation steps, the vectors T and P are being filled with the new values.

2.2 Stiffness Matrix Assembly

The stiffness matrix is calculated according to [1, Section 5]. First the element stiffness matrix is calculated for each face (see. Alg. 4)

Algorithm 4: element stiffness matrix

```
def generate_element_stiffness_mat(self, face):
    k = self.get_face_k(face)
    # get the 3 nodes that make up the face
    nodes = self.get_face_nodes(face)

    # get absolute coordinates of every node
    x = [self.nodal_coords_x[node] for node in nodes]
    y = [self.nodal_coords_y[node] for node in nodes]

    # Zienkiewicz p. 120
    b = [
        y[1] - y[2],
        y[2] - y[0],
        y[0] - y[1]
    ]
    c = [
        x[2] - x[1],
        x[0] - x[2],
        x[1] - x[0]
    ]
    area = (x[0] * b[0] + x[1]*b[1] + x[2]*b[2]) / 2

    # calculate sub matrix
    H_xx = np.array([
        [b[0]*b[0], b[0]*b[1], b[0]*b[2]],
        [b[1]*b[0], b[1]*b[1], b[1]*b[2]],
        [b[2]*b[0], b[2]*b[1], b[2]*b[2]],
    ])
    H_yy = np.array([
        [c[0]*c[0], c[0]*c[1], c[0]*c[2]],
        [c[1]*c[0], c[1]*c[1], c[1]*c[2]],
        [c[2]*c[0], c[2]*c[1], c[2]*c[2]],
    ])
    H = np.zeros((3,3))
    H += k*self.hz / (4*area) * (H_xx + H_yy)
    return np.array(H)
```

Finally the element stiffness matrix (3x3) is added to the global stiffness matrix (NxN) componentwise.

Algorithm 5: magicsolver

```
def magicsolver(A,T,P):
    n = len(A)
    T_known = []
    T_known_i = []
    T_unknown = []
    T_unknown_i = []
    P_known = []
    P_known_i = []
    P_unknown = []
    P_unknown_i = []

    #find unkown values
    for i in range(n):
        if T[i] == None:
            T_unknown.append(T[i])
            T_unknown_i.append(i)
        else:
            T_known.append(T[i])
            T_known_i.append(i)
    for i in range(n):
        if P[i] == None:
            P_unknown.append(P[i])
            P_unknown_i.append(i)
        else:
            P_known.append(P[i])
            P_known_i.append(i)

    #compute T_unknown
    A_sub = A[P_known_i[0]:P_known_i[-1]+1, P_known_i[0]:P_known_i[-1]+1]

    for Pi in range(len(P_known)):
        for Ti in range(len(T_known)):
            P_known[Pi] = P_known[Pi] - A[P_known_i[Pi],T_known_i[Ti]] * T_known
                [Ti]

    T_unknown = numpy.linalg.solve(A_sub, P_known)

    j = 0
    for i in range(n):
        if T[i] == None:
            T[i] = T_unknown[j]
            j+=1

    #compute P_unknown
    for i in range(len(P_unknown)):
        P_unknown[i] = numpy.dot(A[P_unknown_i[i]],T)

    j = 0
    for i in range(n):
        if P[i] == None:
            P[i] = P_unknown[j]
            j+=1

    return T,P
```

3 Results

In the following sections we present our results for different meshes and thermal conductivities. The plots show temperature heatmaps with temperature gradients and flux flows indicated as arrows. To compare the fluxes in elements attached to the boundary $y = L$ with the applied Neumann BCs, we provide plots showing the difference between the calculated output flow, defined input flow and the relative error.
The error is computed by integrating both functions and calculating the relative difference.

$$Q_{in} = \int_0^x q(y = 0)dx \tag{6}$$

$$Q_{out} = \int_0^x q(y = L)dx \tag{7}$$

$$error = \frac{|Q_{out} - Q_{in}|}{Q_{in}} \tag{8}$$

Please note that the labelling of the x-Axis is missing in some cases. The correct dimensions are *x [m]* for all x-Axis

3.1 Variation 0

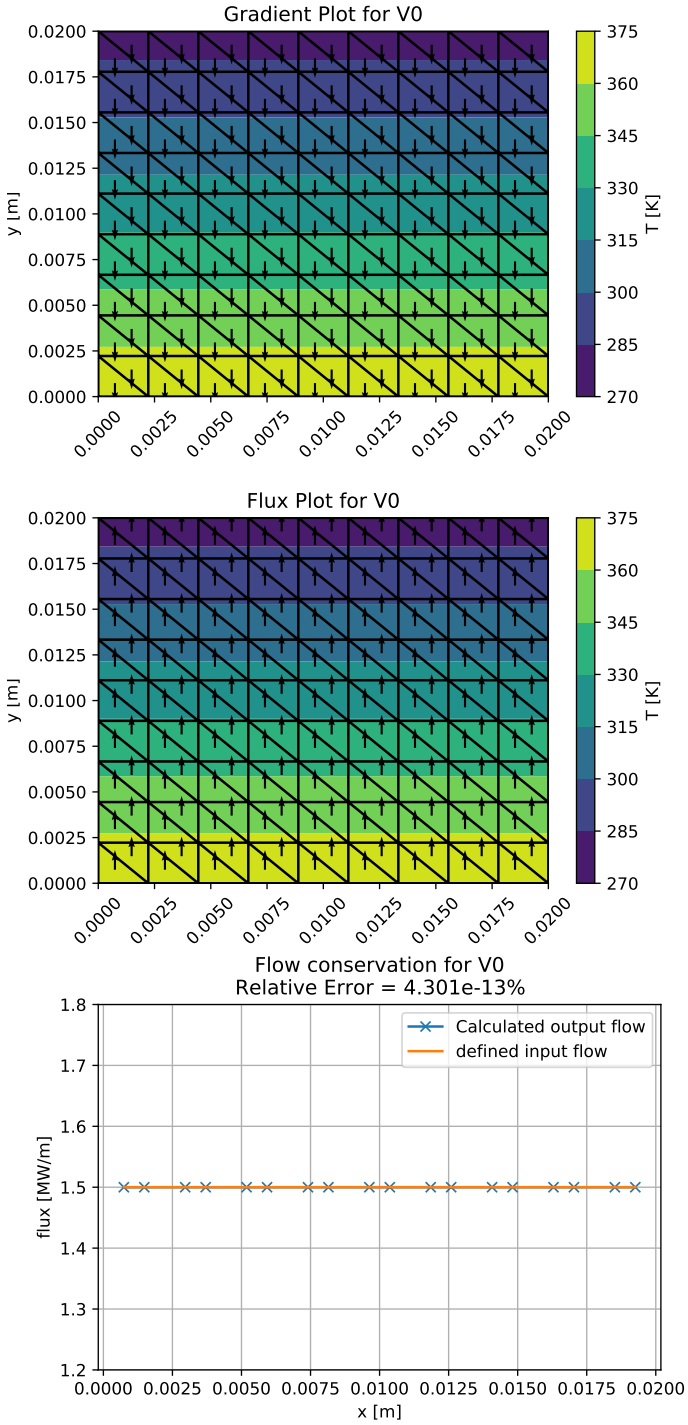

Fig. 2: For the regular mesh we observe the expected constant temperature gradient and flux flow. The x-component of both is zero while the flux is opposite to the gradient. Also the flux attached to boundary is equal to the applied Neumann BC values (input = output). The error is negligible.

3.2 Variation 1

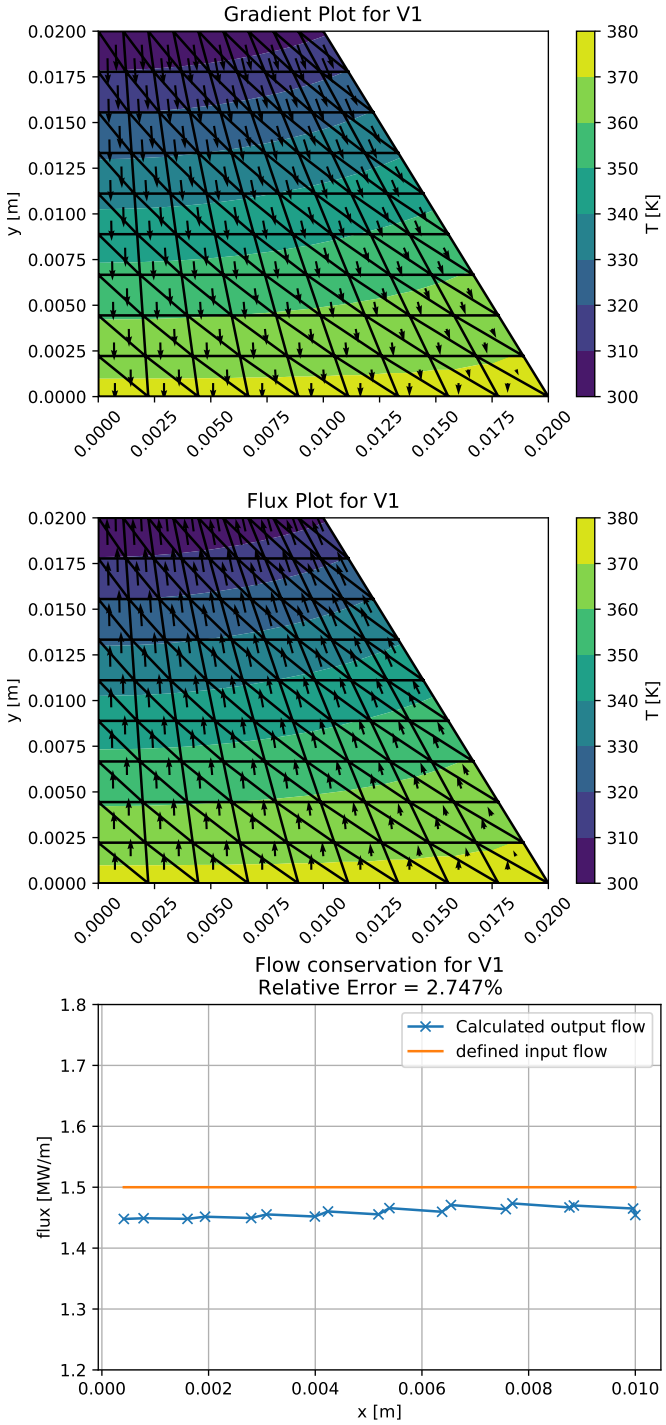

Fig. 3: For the trapezoidal mesh we observe a higher flux flow and temperature gradient as y increases. This is reasonable since the total flux is constant regardless of the area size. However we also observe a lower output flow compared to the input which is not expected. The reason could be computational errors however with a relative error of 2.7% a software bug is most likely the culprit.

3.3 Variation 2

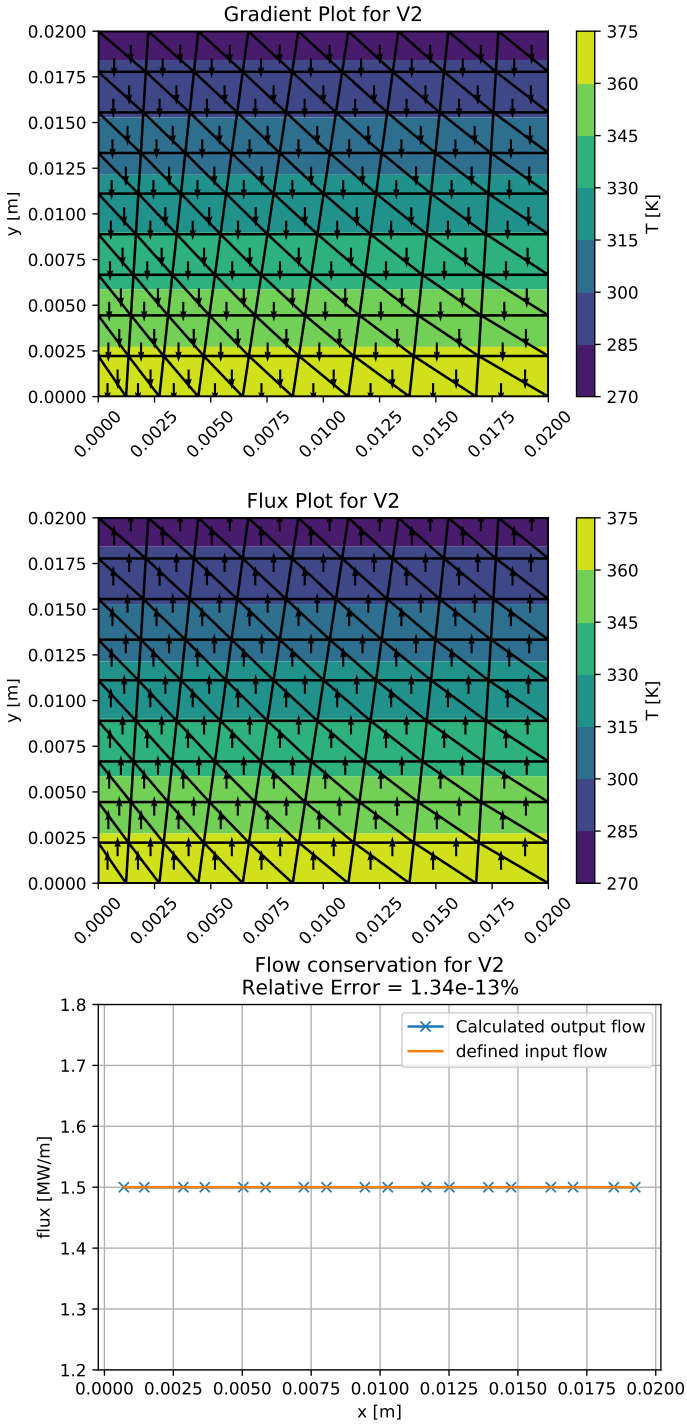

Fig. 4: For the biased mesh, the results are similar to Variation 0. This seems reasonable, since the flux and temperature gradient should be independent of the chosen mesh and should be determined by the geometry and the thermal conductivity only.

3.4 Variation 3

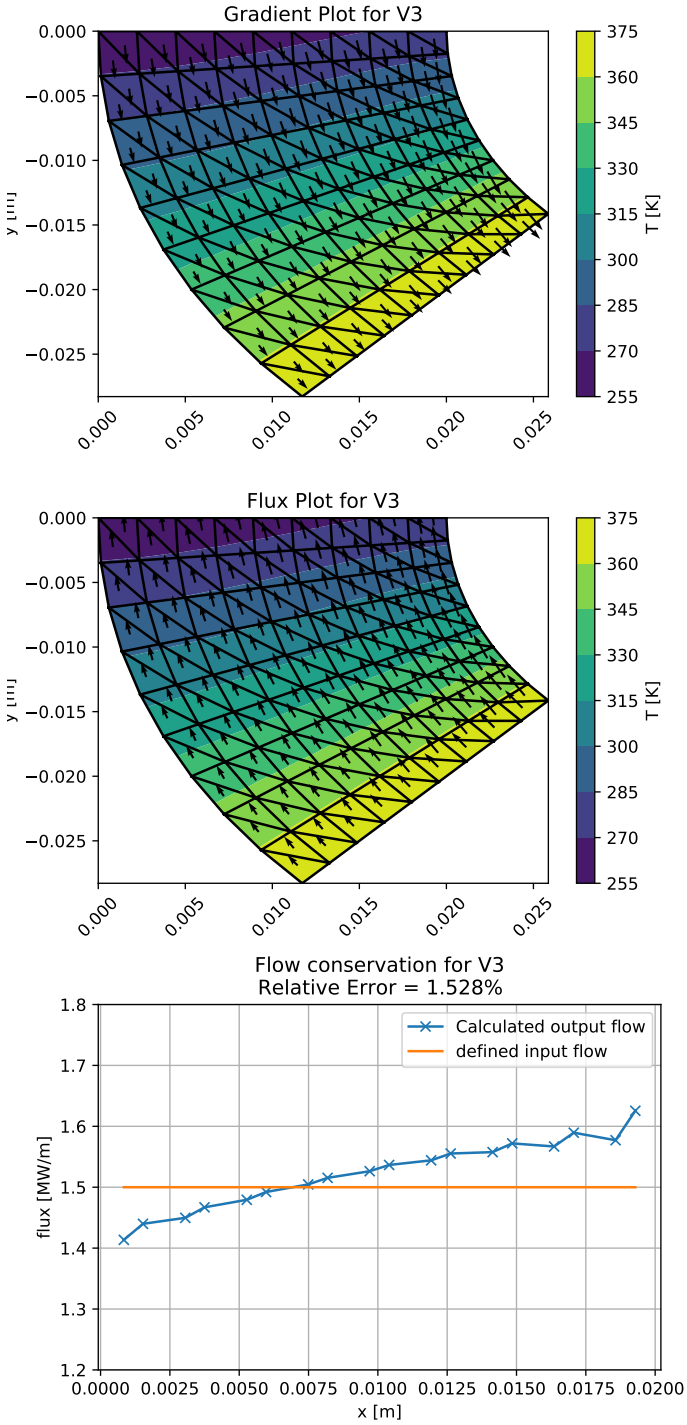

Fig. 5: Since the distance between two opposing boundary points is smaller near the center point of the annulus the gradient has to be steeper as the radial temperature distribution is almost constant. This also results in a high flux on the inside.

3.5 Variation 4a

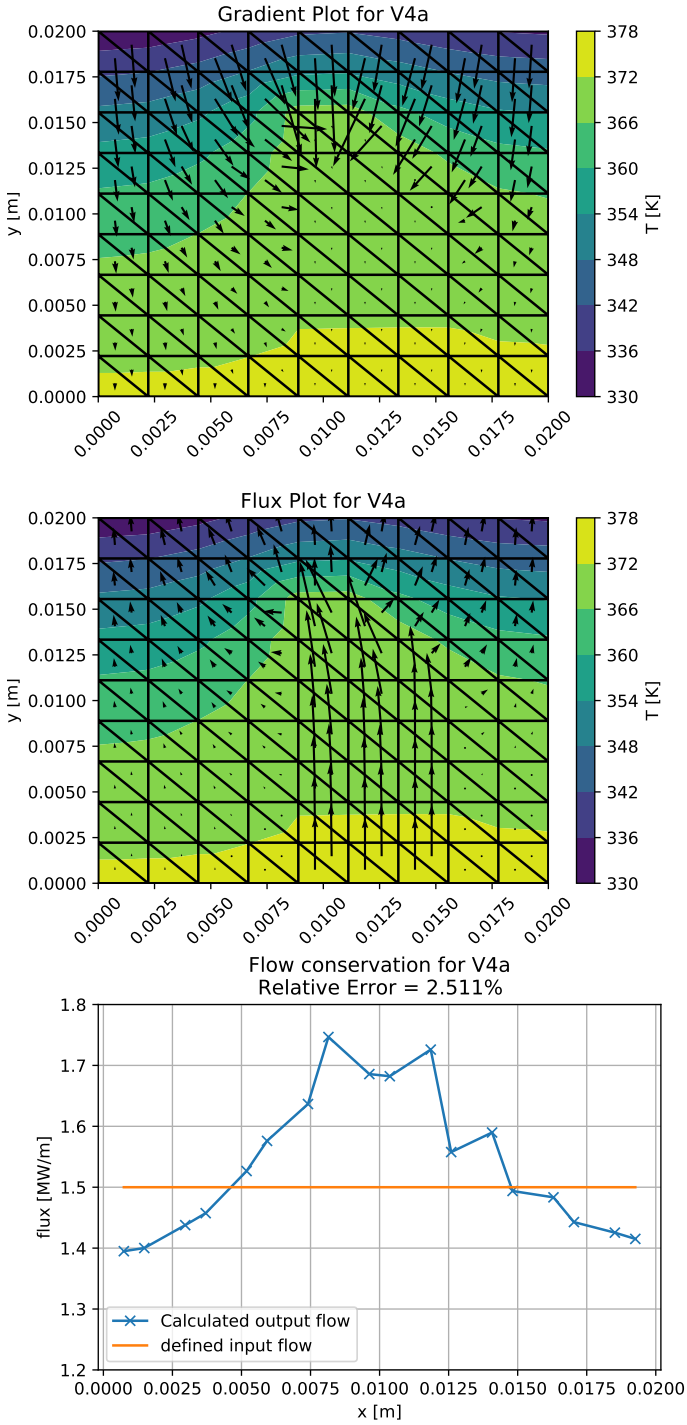

Fig. 6: The mesh has a local increase in heat conductivity in the center. The flux in the area is increased as expected. Since most of the heat energy is "rushing" through the channel of increased conductivity, the gradient is very small within this region as the material is kept warm by the stream of energy. Since the heat sink at y=L is very strong (order of 10^6) the temperature is pulled down in a short distance which results in a steep gradient

3.6 Variation 4b

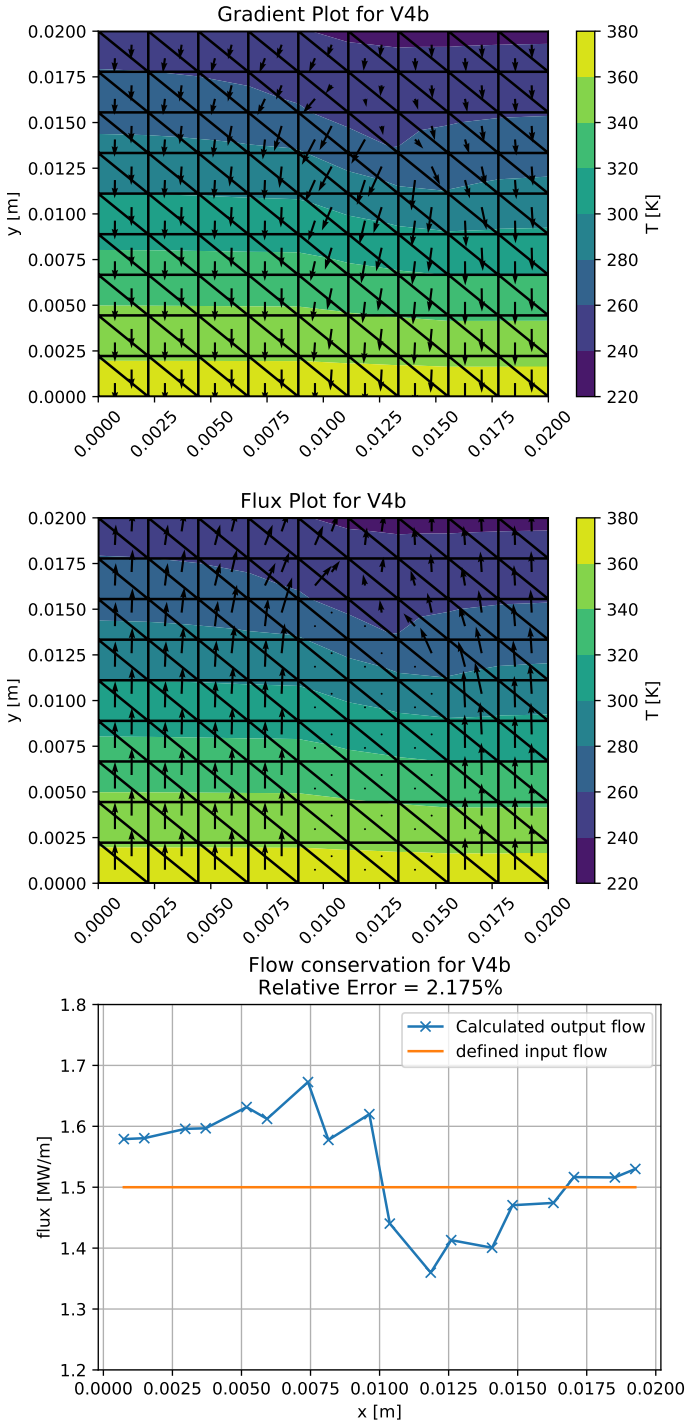

Fig. 7: The mesh has a local decrease in heat conductivity in the center. The flux in the area is decreased as expected. The temperature gradient is very uniform but the heat flux is very small in the area of reduced thermal conductivity. This is because the flux scales with the thermal conductivity.

References

[1] J. Zhu O. Zienkiewicz R. Taylor. *The Finite Element Method: its Basis and Fundamentals (Seventh Edition)*. Oxford, 2013.