1. Salient Features of the Code

The code is split in the following way:

```
Assignment2
├── Pre_Processor.py
├── Post_Processor.py
├── Processor1
│   ├── Processor1.py
│   ├── Shape_Function2D.py
│   ├── Jacobian.py
│   ├── Gauss_Points.py
│   └── Shape_Function_derivative.py
├── Processor2
│   ├── Processor2.py
│   ├── Shape_Function.py
│   └── Gauss_Points.py
└── Processor3
    ├── Processor3.py
    ├── Shape_Function.py
    └── Gauss_Points.py
```

The division between Processor1, Processor2 and Processor3 has been made to keep the conduction within the body, specified heat boundary condition and convective boundary condition and their resulting matrices seperate. They are eventually combined in the Super_Processor which gives the final output.
Now, each subroutine is described one by one:

- **Pre_Processor**

  This subroutine reads the data from the input file in order to send them to the Super_Processor. Some symbols:

  k: Thermal conductivity

  r_var: Heat generated per unit volume per unit time

  n_e: Number of elements

  n_e_dof: Number of degrees of freedom per element

  n_g_dof: Total number of degrees of freedom

  gcv: Global coordinates of the nodes. Numpy array with shape ($n\_g\_dof \times 2$)

  C: Connectivity matrix. Numpy array with shape ($n\_e \times n\_e\_dof$)

  q_star: Specified heat at C2 boundary

  n_b: Number of C2 boundary elements

  n_b_dof: Number of degrees of freedom per C2 boundary

  C_d: Connectivity matrix for C2 boundary ($n\_b \times n\_b\_dof$)

  h: Convective heat transfer coefficient

  T_inf: Ambient Temperature

  n_c: Number of C3 boundary elements

n_c_dof: Number of degrees of freedom per C3 boundary element

C_dd: Connectivity matrix for C3 boundary elements ($n\_c \times n\_c\_dof$).

ess_mat: Essential boundary condition matrix of size ($n \times 2$), where each row contains the node index and the corresponding prescribed value of the essential boundary condition.

- **Processor1**

  This subroutine calculates the global coefficient matrix and the right side vector for the conduction within the body by performing global assembly on the corresponding local matrices. The local matrices are calculated by performing double integration on the shape function or its derivative using gauss points for both x and y directions. In order to do so, it utilizes subroutines for calculating gauss points, jacobian, t matrix and the shape functions and their derivatives for 2D FEM. Some symbols:

  K: Global stiffness matrix ($n\_g\_dof \times n\_g\_dof$)

  F: Global force vector ($n\_g\_dof \times 1$)

  n_g: Number of gauss points.

  gauss_e: Gauss points ($n\_g \times 1$)

  gauss_w: Gauss weights ($n\_g \times 1$)

  N_e: Shape function ($n\_e\_dof \times 1$)

  B_e: Derivative of shape function ($2 \times n\_e\_dof$)

  J: Jacobian matrix (2×2)

  k_e: Elemental stiffness matrix ($n\_e\_dof \times n\_e\_dof$)

  f_e: Elemental force vector ($n\_e\_dof \times 1$)

- **Shape_Function2D**

  This function calculates the Lagrangian Shape Function vector for 2D FEM in natural coordinates by taking as input, the number of degrees of freedom per element (**p**)(general value) and the gauss points (**e, n**) at which the vector is calculated. It calls a subroutine which returns the 1D Lagrangian Shape Function for a point. It returns **N** which represents the Shape Function vector. Some symbols:

  n_e_dof: Number of degrees of freedom per element (1D)

  num: Index of the node for which the 1D Lagrangian shape function is being calculated.

  coords: Array containing the coordinates of the nodes

  arr_n: Value of the 1D Lagrangian shape function for the given degree of freedom

  p: Number of degrees of freedom per element (2D)

  e: X-coordinate gauss value

  n: Y-coordinate gauss value

  N: Array containing the values of the shape functions evaluated at the given coordinate

  dof_b: Number of degrees of freedom per boundary

- **Shape_Function_derivative**

  This subroutine returns the derivative of the Shape Function vector by performing numerical differentiation (Central Difference Method). It takes in as input **p, e, n, a_e, b_e** and uses the Jacobian and Shape_Function2D subroutines as methods in its code. It transforms **B** to **B_conv** using the **t_matrix** function. It returns **B_conv** which represents the derivative of the Shape Function vector.

- **Jacobian**

  This subroutine calculates the Jacobian matrix **J** of the given mapping between domain coordinates and natural coordinates by taking in as input the length and breadth of the element **a_e, b_e**. It also calculates the **t_mat** (used for transforming **B**) using the jacobian in a separate function.

- **Gauss_Points**

  This subroutine returns the Gauss points and corresponding Gauss weights by taking the number of Gauss points (up to 20) as an argument.

- **Processor2**

  This subroutine takes input the specified neumann boundary condition on heat flux (secondary variable) (**q**), no. of boundary elements (**n_b**), no. of dof per boundary element (**n_b_dof**), no. of global boundary nodes (**n_g_dof**), global coordinate vector (**gcv**) (taken as input from mesh data) and connectivity matrix of boundary elements present on C2 boundary (**C_d**). Output is line elemental force vector (**Qd**) calculated using 1D numerical integration scheme with certain number of gauss points and corresponding weights.

- **Shape_Function1D for C2 and C3 boundaries**

  This function calculates the Lagrangian Shape Function vector for 1D FEM in natural coordinates by taking as input, the number of degrees of freedom per element (**p**)(general value) and the gauss points (**e**) at which the vector is calculated. It calls a subroutine which returns the 1D Lagrangian Shape Function for a point. It returns **N** which represents the Shape Function vector.

- **Processor3**

  This subroutine takes input heat transfer coefficient (**h**), ambient temperature (**T_inf**), no. of boundary elements (**n_c**), no. of dof per boundary element (**n_c_dof**), no. of global boundary nodes (**n_g_dof**), global coordinate vector (**gcv**) (taken as input from mesh data) and connectivity matrix of boundary elements present on C3 boundary (**C_dd**). Output is line elemental stiffness coefficient matrix (**Kdd**) and line element force vector (**Qdd**) calculated using 1D numerical integration scheme with certain number of gauss points and corresponding weights.

- **Super_Processor**

  This subroutine takes in the output of the Pre_Processor and sends it to the Processor1, Processor2 and Processor3 in order to obtain **K, Kdd, Q, Qd, Qdd**. It then adds these to obtain **global_K and global_Q**. Next, it applies the essential boundary conditions using **ess_mat** and solves the system of equations to obtain **prim_var**. Some symbols:

  K: Global stiffness matrix computed by Processor1 ($n\_g\_dof \times n\_g\_dof$)

  Q: Global force vector computed by Processor1 ($n\_g\_dof \times 1$)

Qd: Global force vector computed by Processor2 ($n\_g\_dof \times 1$)

Kdd: Global stiffness matrix computed by Processor3 ($n\_g\_dof \times n\_g\_dof$)

Qdd: Global force vector computed by Processor3 ($n\_g\_dof \times 1$)

global_K: Combined global stiffness matrix after adding contributions from Processor1 and Processor3 ($n\_g\_dof \times n\_g\_dof$)

global_Q: Combined global force vector after adding contributions from Processor1, Processor2, and Processor3 ($n\_g\_dof \times 1$)

prim_var: Solution vector containing the primary variables ($n\_g\_dof \times 1$)

- **Post_Processor**

  This subroutine takes in **gcv, prim_var** from the Super_Processor and can also be used for further calculations. It writes the values of the obtained Primary variables at their corresponding coordinate values in a csv file.

The local element in our code follows different numbering than usual, the same change has been made in the connectivity matrices for proper functioning. For the values of the constants we

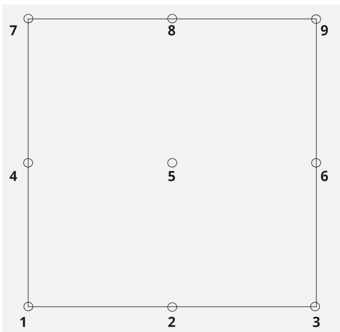

Figure 1: Local node numbering

have used $k = 45\ W/mK(steel)$, $h = 5\ W/m^2K$ (air - natural convection), $T\_inf = 300\ K$, $r\_var = 2\ W/m^2$, $q\_star = 0\ W/m^2$ (mentioned in the question), $T\_star = 320K$.

2. Code

2.A. The Pre-Processor

Listing 1: Pre_Processor.py

```
import csv
import numpy as np

def pre_processor():
    rows=[]

    with open('values.csv','r') as f:
        csvreader = csv.reader(f)
        for row in csvreader:
            rows.append(row)

    f.close()

    #NODE NUMBERING STARTS FROM 0
```

```python

    #n_e, n_e_dof, n_g_dof, gcv, C, q_star, n_b, n_b_dof, C_d, n_c,
        n_c_dof, C_dd, ess_mat

    # Constants
    k = float(rows[0][0])
    r_var = float(rows[1][0])
    q_star = float(rows[2][0])
    h = float(rows[3][0])
    T_inf =  float(rows[4][0])


    # Element properties
    n_e_x = int(rows[5][0]) #Total number of elements in x direction in
        domain
    n_e_y = int(rows[6][0])
    n_e = n_e_x*n_e_y
    n_b = int(rows[7][0])
    n_c = int(rows[8][0])
    n_e_dof = int(rows[9][0]) #Number of dof per element
    n_b_dof = n_c_dof = int((n_e_dof)**(1/2)) #Number of dof per
        boundary element


    # Creation of C
    root = int(n_e_dof**(1/2))
    r_len = root*n_e_x - (n_e_x - 1)
    C = np.zeros((n_e_x*n_e_y, n_e_dof))
    el = 0
    for i in range(n_e_y):
        for j in range(n_e_x):
            start = j*(root - 1) + (root - 1)*(i*r_len)
            k = 0
            while k<n_e_dof:
                for l in range(root):
                    C[el][k] = int(start + l)
                    k += 1
                start += r_len
            el += 1

    n_g_dof = int(C[el-1][k-1] + 1)

    # Global Coordinate Vector
    gcv = [] #Global coordinate vector - X coord, Y coord at node number
    for i in range(0, len(rows[10]), 2):
        gcv.append([float(rows[10][i]), float(rows[10][i+1])])

    # C'
    C_d = np.zeros((n_b, n_b_dof))
    for i in range(0, n_b):
        for j in range(0, n_b_dof):
            C_d[i][j] = int(rows[11 + i][j])

    # C''
    C_dd = np.zeros((n_c,n_c_dof))
    for i in range(0, n_c):
        for j in range(0, n_c_dof):
            C_dd[i][j] = int(rows[11 + n_b + i][j])
```

```

    # Essential Boundary Condition Matrix
    ess_mat = []
    for i in range(0, len(rows[11 + n_b + n_c]), 2):
        ess_mat.append([int(rows[11+ n_b + n_c][i]),int(rows[11 + n_b +
            n_c][i+1])]) #Node number, value

    return k, r_var, n_e, n_e_dof, n_g_dof, gcv, C, q_star, n_b,
        n_b_dof, C_d, h, T_inf, n_c, n_c_dof, C_dd, ess_mat
```

2.B. Processor 1

Listing 2: Processor1.py

```
# Import necessary modules
from . import Gauss_Points as gauss
from . import Shape_Function2D as sf
from . import Shape_Function_derivative as sfd
from . import Jacobian as jac
import numpy as np

def processor_func(k, r_var, n_e, n_e_dof, n_g_dof, gcv, C):

    # Calculation of number of gauss points #CHANGE
    n_g_k = int(((n_e_dof-2)*2 + 1)/2)
    n_g_f = int(n_e_dof/2)
    n_g = max(n_g_k, n_g_f)

    #Initialize global matrices
    K = np.zeros((n_g_dof,n_g_dof)) #coefficient matrix
    F = np.zeros((n_g_dof,1)) #right side vector

    #Loop over all elements
    for i in range(0, n_e):

        #Calculation of gauss weights and points
        gauss_e, gauss_w = gauss.weights(n_g)

        #Initialize element matrices
        k_e = np.zeros((n_e_dof,n_e_dof)) #coefficient matrix
        f_e = np.zeros((n_e_dof,1)) #right side vector

        #Calculating length and breadth of element
        n_m = int(C[i][n_e_dof - 1]); n_1 = int(C[i][0])
        x_m = gcv[n_m][0]; x_1 = gcv[n_1][0]; ae = abs(x_m - x_1)
        y_m = gcv[n_m][1]; y_1 = gcv[n_1][1]; be = abs(y_m - y_1)


        #Loop over gauss points
        for iter in range(n_g):
            wk1 = gauss_w[iter]
            ek1 = gauss_e[iter]

            for j in range(n_g):
                wk2 = gauss_w[j]
                ek2 = gauss_e[j]

                #Calculate shape functions and derivatives
```

```
                N_e = sf.shape_vector(n_e_dof, ek1, ek2)
                B_e = sfd.shape_vector_der(n_e_dof, ek1, ek2, ae, be)

                #Calculate the Jacobian
                J = jac.jacobian(ae, be)

                #Calculate k_e
                arr = (wk1*wk2)*k*np.linalg.det(J)*np.matmul(np.transpose(B_e), B_e)
                k_e = k_e + arr

                #Calculate f_e
                f_e = f_e + (wk1*wk2)*r_var*np.linalg.det(J)*N_e

        # Global assembly of K_e #CHECK
        # K_e(r,s) = k_e(p,q) if C(e,p) = r && C(e,q) = s
        K_e = np.zeros((n_g_dof, n_g_dof))
        for p in range(0, n_e_dof):
            for q in range(0, n_e_dof):
                r = int(C[i][p])
                s = int(C[i][q])
                if r<=n_g_dof and s<=n_g_dof:
                    K_e[r][s] = k_e[p][q]

        K = K + K_e

        # Global assembly of F_e
        #F_e(r) = f_e(p) if C(e,p) = r
        F_e = np.zeros((n_g_dof,1))
        for p in range(0, n_e_dof):
            r = int(C[i][p])
            if r<=n_g_dof:
                F_e[r] = f_e[p]

        F = F + F_e

    #Return necessary data
    return K, F
```

2.C. Processor 2

Listing 3: Processor2.py

```
#This function deals with C2 boundary with q = q* condition (specified heat flux)

from . import Gauss_Points as gauss
from . import Shape_Function as sf
import numpy as np

def processor_func(q_star, n_b, n_b_dof ,n_g_dof , gcv, C_d):

    # Calculation of number of gauss points
    n_g = int(n_b_dof/2)
    #print(n_g)

    #Initialize global matrices
```

```
    Q = np.zeros((n_g_dof ,1)) #right side vector

    #Loop over all elements
    for i in range(0, n_b):

        #Calculation of gauss weights and points
        gauss_e_f , gauss_w_f = gauss.weights(n_g)
        #print("Gauss points: ",gauss_w_k,gauss_w_f)

        #Initialize element matrices
        f_e = np.zeros((n_b_dof ,1)) #right side vector

        #Calculating length of element
        n_m = int(C_d[i][n_b_dof - 1]); n_1 = int(C_d[i][0])
        #will require both x,y #CHANGE
        x_m = gcv[n_m][0]; x_1 = gcv[n_1][0]
        y_m = gcv[n_m][1]; y_1 = gcv[n_1][1]
        lb = (pow((x_m - x_1),2) + pow((y_m - y_1),2))**0.5

        #Loop over gauss points
        for j in range(n_g):
            wk = gauss_w_f[j]
            ek = gauss_e_f[j]

            #Calculate shape functions and derivatives
            N_e = sf.shape_vector(n_b_dof , ek)
            N_e_reshape = np.reshape(N_e, (n_b_dof))

            # Calculate f_e
            f_e = f_e - (wk*q_star*lb*N_e)/2

        # Global assembly of F_e
        #F_e(r) = f_e(p) if C(e,p) = r
        Q_e = np.zeros((n_g_dof ,1))
        for p in range(0, n_b_dof):
            r = int(C_d[i][p])
            if r<=n_g_dof:
                Q_e[r] = f_e[p]

        Q = Q + Q_e

    #Return necessary data
    return Q
```

2.D. Processor 3

Listing 4: Processor3.py

```
from . import Gauss_Points as gauss
from . import Shape_Function as sf
import numpy as np

def processor_func(h, T_inf , n_c , n_c_dof , n_g_dof , gcv , C_dd):

    # Calculation of number of gauss points
    n_g_k = int((2*(n_c_dof -1) + 1)/2)
    n_g_f = int(n_c_dof/2)
    #print(n_g_k , n_g_f)
```

```python

    #Initialize global matrices
    K = np.zeros((n_g_dof,n_g_dof)) #coefficient matrix
    F = np.zeros((n_g_dof,1)) #right side vector

    #Loop over all elements
    for i in range(0, n_c):

        #Calculation of gauss weights and points
        gauss_e_k, gauss_w_k = gauss.weights(n_g_k)
        gauss_e_f, gauss_w_f = gauss.weights(n_g_f)

        #Initialize element matrices
        k_e = np.zeros((n_c_dof,n_c_dof)) #coefficient matrix
        f_e = np.zeros((n_c_dof,1)) #right side vector

        #Calculating length of element
        n_m = int(C_dd[i][n_c_dof - 1])
        n_1 = int(C_dd[i][0])

        x_m = gcv[n_m][0]; x_1 = gcv[n_1][0];
        y_m = gcv[n_m][1]; y_1 = gcv[n_1][1];
        lc = (pow((x_m - x_1),2) + pow((y_m - y_1),2))**0.5

        #Loop over gauss points
        for j in range(n_g_k):
            wk = gauss_w_k[j]
            ek = gauss_e_k[j]

            #Calculate shape functions and derivatives
            N_e = sf.shape_vector(n_c_dof, ek)
            N_e_reshape = np.reshape(N_e, (n_c_dof))

            #Calculate k_e
            arr = wk*h*(lc/2)*np.matmul(N_e, np.transpose(N_e))
            k_e = k_e + arr

        for j in range(n_g_f):
            wk = gauss_w_f[j]
            ek = gauss_e_f[j]

            #Calculate shape functions and derivatives
            N_e = sf.shape_vector(n_c_dof, ek)
            N_e_reshape = np.reshape(N_e, (n_c_dof))

            # Calculate f_e
            f_e = f_e + (wk*h*T_inf*lc*N_e)/2

        # Global assembly of K_e
        #K_e(r,s) = k_e(p,q) if C(e,p) = r && C(e,q) = s
        K_e = np.zeros((n_g_dof, n_g_dof))
        for p in range(0, n_c_dof):
            for q in range(0, n_c_dof):
                r = int(C_dd[i][p])
                s = int(C_dd[i][q])
                if r<=n_g_dof and s<=n_g_dof:
                    K_e[r][s] = k_e[p][q]

```

```
            K = K + K_e

            # Global assembly of F_e
            #F_e(r) = f_e(p) if C(e,p) = r
            F_e = np.zeros((n_g_dof,1))
            for p in range(0, n_c_dof):
                r = int(C_dd[i][p])
                if r<=n_g_dof:
                    F_e[r] = f_e[p]

            F = F + F_e

        #Return necessary data
        return K, F
```

2.E. Super Processor

Listing 5: Super*processor.py*

```
import Pre_Processor as pre
from Processor1 import Processor1
from Processor2 import Processor2
from Processor3 import Processor3
import numpy as np


def super_processor():

    k, r_var, n_e, n_e_dof, n_g_dof, gcv, C, q_star, n_b, n_b_dof, C_d,
        h, T_inf, n_c, n_c_dof, C_dd, ess_mat = pre.pre_processor()

    K, Q = Processor1.processor_func(k, r_var, n_e, n_e_dof, n_g_dof,
        gcv, C)
    Qd = Processor2.processor_func(q_star, n_b, n_b_dof, n_g_dof, gcv,
        C_d)
    Kdd, Qdd = Processor3.processor_func(h, T_inf, n_c, n_c_dof,
        n_g_dof, gcv, C_dd)

    global_K = K + Kdd
    global_Q = Q + Qd + Qdd

    #Putting essential boundary condition
    for i in ess_mat: #Here i is of the form [node, val = T_star]
        subtract = global_K[i[0]][:]*i[1]
        subtract.shape = (len(global_K),1)
        global_Q = global_Q - subtract
        global_K[:,i[0]] = 0
        global_K[i[0],:] = 0
        global_K[i[0]][i[0]] = 1
        global_Q[i[0]] = i[1]

    #Solving the system of equations
    # [K]{u} = {F}
    prim_var = np.linalg.solve(global_K, global_Q)
    prim_var.shape = (n_g_dof,1)
    #print(prim_var)

    return prim_var, gcv
```

The codes above use the following subroutines for calculating the Gauss points, the Shape functions and their derivatives (1d and 2d), Jacobian and t matrix.

2.F. Gauss points and weights subroutine

Listing 6: Gauss_weights.py

```
def weights(n):

    if n == 1:
        ep = [0.0]
        w = [2.0]
    elif n == 2:
        ep = [-0.5773502691896257, 0.5773502691896257]
        w = [1.0, 1.0]
    elif n == 3:
        ep = [-0.7745966692414834, 0.0, 0.7745966692414834]
        w = [0.5555555555555556, 0.8888888888888888, 0.5555555555555556]
    elif n == 4:
        ep = [-0.8611363115940526, -0.3399810435848563,
            0.3399810435848563, 0.8611363115940526]
        w = [0.3478548451374538, 0.6521451548625461,
            0.6521451548625461, 0.3478548451374538]

    #SIMILAR CODE AFTER THIS FOR N UPTO 20

    else:
        raise ValueError("Gaussian quadrature points and weights not
            implemented for n > 20.")

    return ep, w
```

2.G. 1D Shape Function for C2, C3 boundary subroutine

Listing 7: Shape_Function.py

```
#Lagrangian Shape function construction
import numpy as np

def shape_vector(n_e_dof, e):

    #Generate natural coordinate node vector
    ep = np.linspace(-1, 1, num = n_e_dof)
    arr_n = np.zeros((n_e_dof,1))

    #Iterate over each element in node vector
    for i in range(0, n_e_dof):
        numerator = 1

        # Calculate numerator of Lagrangian shape function
        for j in range(0,n_e_dof):
            if(i!=j): numerator*=(e - ep[j])

        denominator = 1

        # Calculate denominator of Lagrangian shape function
```

```
        for j in range(0, n_e_dof):
            if(i!=j): denominator*=(ep[i] - ep[j])

        # Calculate Lagrangian shape function value for each node
        arr_n[i] = numerator/denominator

    arr_n.shape = (n_e_dof,1)
    return arr_n
```

2.H. Derivative of 1D Shape Function for C2, C3 boundary subroutine

Listing 8: Shape_function_derivative.py

```
import numpy as np
import Shape_Function as sf

def shape_derivative(n_e_dof, e):
    h = 1e-6 #step value

    N_h = sf.shape_vector(n_e_dof, e+h) #Shape function for ep = e+h
    N = sf.shape_vector(n_e_dof, e-h) #Shape function for ep = e-h

    B = (N_h - N)/(2*h) #Central Difference Method
    return B
```

2.I. Jacobian and [t] matrix subroutine

Listing 9: Jacobian.py

```
import numpy as np

#x = (x1e + x2e)/2 + ae*e

def jacobian(ae, be):
    J = np.array([[ae/2, 0],[0, be/2]])
    return J

def t_matrix(ae, be):
    J = jacobian(ae, be)
    t_mat = np.linalg.inv(np.transpose(J))
    return t_mat
```

2.J. 2D Shape Function subroutine

Listing 10: Shape_function2D.py

```
import numpy as np

def L(n_e_dof, num, e, coords):
    numerator = 1

    # Calculate numerator of Lagrangian shape function
    for j in range(0,n_e_dof):
        if(num!=j): numerator*=(e - coords[j])

    denominator = 1
```

```python

    # Calculate denominator of Lagrangian shape function
    for j in range(0, n_e_dof):
        if(num!=j): denominator*=(coords[num] - coords[j])

    # Calculate Lagrangian shape function value
    arr_n = numerator/denominator

    return arr_n

def shape_vector(p, e, n):

    N = np.zeros((p, 1))
    dof_b = int(p**(1/2))
    coords = np.linspace(-1, 1, num = dof_b)

    count = 0
    for i in range(dof_b): #n loop
        for j in range(dof_b): #e loop
            N[count] = L(dof_b, i, n, coords)*L(dof_b, j, e, coords) #remember i,j start at 0
            count+=1

    return N
```

2.K. Derivative of 2D Shape Function subroutine

Listing 11: Shape_Function_derivative.py

```python
import numpy as np
from . import Shape_Function2D as sf
from . import Jacobian as jac

def der_L(n_e_dof, num, e, coords):

    #Numerical differentiation
    h = 1e-6
    N_h = sf.L(n_e_dof, num, e+h, coords) #(n_e_dof, e+h)
    N = sf.L(n_e_dof, num, e-h, coords) #(n_e_dof, e-h)

    B = (N_h - N)/(2*h)
    return B


def shape_vector_der(p, e, n, ae, be): #0 based indexing

    B = np.zeros((2, p))
    dof_b = int(p**(1/2))
    coords = np.linspace(-1, 1, num = dof_b)

    #derivative wrt to e
    count = 0
    for i in range(dof_b): #n loop
        for j in range(dof_b): #e loop
            B[0][count] = sf.L(dof_b, i, n, coords)*der_L(dof_b, j, e, coords)
            count += 1

```

```
    #derivative wrt to n
    count = 0
    for i in range(dof_b): #n loop
        for j in range(dof_b): #e loop
            B[1][count] = der_L(dof_b, i, n, coords)*sf.L(dof_b, j, e,
                coords)
            count += 1

    #conversion from master system to local system
    B_conv = np.matmul(jac.t_matrix(ae, be), B)

    return B_conv
```

2.L. The Post-Processor

Listing 12: Post_Processor.py

```
import numpy as np
import Super_Processor as pro
import csv

prim_var, gcv = pro.super_processor()

with open('answer.csv', "w", newline="") as f:
    writer = csv.writer(f)
    list = ["X coordinate"]
    list2 = ["Y coordinate"]
    list3 = ["Primary Variable"]

    for i in range(len(gcv)):
        list.append(gcv[i][0])
        list2.append(gcv[i][1])
        list3.append(prim_var[i][0])
    writer.writerow(list)
    writer.writerow(list2)
    writer.writerow(list3)
```

3. Input File

Input file for the first discretization is:

```
45
2
0
5
300
4
4
4
8
9
-1.0,-1.0,-0.75,-1.0,-0.5,-1.0,-0.25,-1.0,0.0,-1.0,0.25,-1.0,0.5,-1.0,0.75,-1.0,1.0,-1.0,-1.0,-
0.75,-0.75,-0.75,-0.5,-0.75,-0.25,-0.75,0.0,-0.75,0.25,-0.75,0.5,-0.75,0.75,-0.75,1.0,-0.75,-
1.0,-0.5,-0.75,-0.5,-0.5,-0.5,-0.25,-0.5,0.0,-0.5,0.25,-0.5,0.5,-0.5,0.75,-0.5,1.0,-0.5,-1.0,-
0.25,-0.75,-0.25,-0.5,-0.25,-0.25,-0.25,0.0,-0.25,0.25,-0.25,0.5,-0.25,0.75,-0.25,1.0,-0.25,-
1.0,0.0,-0.75,0.0,-0.5,0.0,-0.25,0.0,0.0,0.0,0.25,0.0,0.5,0.0,0.75,0.0,1.0,0.0,-1.0,0.25,-
0.75,0.25,-0.5,0.25,-0.25,0.25,0.0,0.25,0.25,0.25,0.5,0.25,0.75,0.25,1.0,0.25,-1.0,0.5,-
0.75,0.5,-0.5,0.5,-0.25,0.5,0.0,0.5,0.25,0.5,0.5,0.5,0.75,0.5,1.0,0.5,-1.0,0.75,-0.75,0.75,-
0.5,0.75,-0.25,0.75,0.0,0.75,0.25,0.75,0.5,0.75,0.75,0.75,1.0,0.75,-1.0,1.0,-0.75,1.0,-0.5,1.0,-
0.25,1.0,0.0,1.0,0.25,1.0,0.5,1.0,0.75,1.0,1.0,1.0
44,53,62
62,71,80
80,79,78
76,77,76
36,27,18
18,9,0
0,1,2
2,3,4
4,5,6
6,7,8
8,17,26
26,35,44
45,50,54,50,63,50,72,50,73,50,74,50,75,50
```

Input file for the second discretization is:

```
45
2
0
5
300
8
8
8
16
9
-1.0,-1.0,-0.875,-1.0,-0.75,-1.0,-0.625,-1.0,-0.5,-1.0,-0.375,-1.0,-0.25,-1.0,-0.125,-1.0,0.0,-1.0,
0.125,-1.0,0.25,-1.0,0.375,-1.0,0.5,-1.0,0.625,-1.0,0.75,-1.0,0.875,-1.0,1.0,-1.0,-1.0,-0.875,-0.875
,-0.875,-0.75,-0.875,-0.625,...,-0.625,0.875,-0.5,0.875,-0.375,0.875,-0.25,0.875,-0.125,0.875,0.0,
0.875,0.125,0.875,0.25,0.875,0.375,0.875,0.5,0.875,0.625,0.875,0.75,0.875,0.875,0.875,1.0,0.875,
-1.0,1.0,-0.875,1.0,-0.75,1.0,-0.625,1.0,-0.5,1.0,-0.375,1.0,-0.25,1.0,-0.125,1.0,0.0,1.0,0.125,1.0,
0.25,1.0,0.375,1.0,0.5,1.0,0.625,1.0,0.75,1.0,0.875,1.0,1.0,1.0
152,169,186
186,203,220
220,237,254
254,271,288
288,287,286
286,285,284
284,283,282
282,281,280
136,119,102
102,85,68
68,51,34
34,17,0
0,1,2
2,3,4
4,5,6
6,7,8
8,9,10
10,11,12
12,13,14
14,15,16
16,33,50
50,67,84
84,101,118
118,135,152
153,50,170,50,187,50,204,50,221,50,238,50,255,50,272,50,273,50,274,50,275,50,276,50,277,50,
278,50,279,50
```

The entire GCV input hasn't been shown because of its length.
The rows represent:

1. Thermal conductivity
2. Heat generated per unit volume per unit time
3. Specified heat at C2 boundary
4. Convective heat transfer coefficient

5. Ambient Temperature
6. Number of elements in x direction (C1)
7. Number of elements in y direction (C1)
8. Number of elements in C2 boundary
9. Number of elements in C3 boundary
10. Number of DOF per element
11. Global coordinate vector (value at index which is node index)
12. C' matrix
13. C" matrix
14. Essential BC matrix

4. Output File

For the second discretization the number of elements have been doubled by dividing the element length by two. In the first discretization the number of elements is 16 while for the second its 64. Every element has 9 nodes. The output files are:

Table 1: Discretization 1

X coordinate	Y coordinate	Primary Variable
-1	-1	268.5987119
-0.75	-1	319.9540531
-0.5	-1	286.2007723
-0.25	-1	318.7201884
0	-1	287.2895337
0.25	-1	318.2462763
0.5	-1	285.3455288
0.75	-1	318.8888482
1	-1	267.5531131
-1	-0.75	320.3962948
-0.75	-0.75	309.0222842
-0.5	-0.75	311.4114089
-0.25	-0.75	309.1269121
0	-0.75	310.2123188
0.25	-0.75	308.5393709
0.5	-0.75	310.3431849
0.75	-0.75	307.6847155
1	-0.75	319.0841281
-1	-0.5	288.4855979
-0.75	-0.5	312.6386491
-0.5	-0.5	307.8008027
-0.25	-0.5	311.7328084
0	-0.5	308.4398678
0.25	-0.5	310.9241931
0.5	-0.5	306.3076398
0.75	-0.5	310.7242414

1	-0.5	286.5585137
-1	-0.25	323.1488689
-0.75	-0.25	313.0078968
-0.5	-0.25	313.4604448
-0.25	-0.25	312.0827228
0	-0.25	311.735486
0.25	-0.25	310.9448415
0.5	-0.25	311.2721411
0.75	-0.25	310.0556098
1	-0.25	320.1512782
-1	0	304.3365844
-0.75	0	315.5134005
-0.5	0	313.1498292
-0.25	0	313.5664889
0	0	312.3779365
0.25	0	312.0063712
0.5	0	310.1080006
0.75	0	310.814796
1	0	298.9587885
-1	0.25	320
-0.75	0.25	316.5765496
-0.5	0.25	315.981303
-0.25	0.25	314.7468051
0	0.25	313.8211472
0.25	0.25	312.7502656
0.5	0.25	311.9228716
0.75	0.25	310.2638077
1	0.25	309.4321168
-1	0.5	320
-0.75	0.5	318.7266881
-0.5	0.5	317.0047601
-0.25	0.5	316.1521928
0	0.5	314.7835159
0.25	0.5	313.6768976
0.5	0.5	312.3834892
0.75	0.5	311.8044812
1	0.5	310.7784047
-1	0.75	320
-0.75	0.75	319.3200161
-0.5	0.75	318.6219962
-0.25	0.75	317.5409973
0	0.75	315.9823525
0.25	0.75	314.3526884
0.5	0.75	313.1875706
0.75	0.75	312.3927834
1	0.75	312.1127811
-1	1	320
-0.75	1	320
-0.5	1	320
-0.25	1	320
0	1	316.5286117

0.25	1	314.6511782
0.5	1	313.3839566
0.75	1	312.6763115
1	1	312.3258066

Table 2: Discretization 2

X	Y	Primary Variable	X	Y	Primary Variable
-1	-1	286.9336	-0.125	0	313.0462
-0.875	-1	313.0268	0	0	312.6708
-0.75	-1	296.4421	0.125	0	312.3023
-0.625	-1	312.9076	0.25	0	311.9327
-0.5	-1	297.4705	0.375	0	311.5775
-0.375	-1	312.9824	0.5	0	311.1844
-0.25	-1	297.5396	0.625	0	310.8885
-0.125	-1	312.8819	0.75	0	310.0906
0	-1	297.3648	0.875	0	310.3654
0.125	-1	312.6325	1	0	304.6247
0.25	-1	297.0523	-1	0.125	320
0.375	-1	312.2796	-0.875	0.125	317.3226
0.5	-1	296.5864	-0.75	0.125	316.5785
0.625	-1	311.8864	-0.625	0.125	315.7248
0.75	-1	295.3387	-0.5	0.125	315.1649
0.875	-1	311.9018	-0.375	0.125	314.6526
1	-1	285.8517	-0.25	0.125	314.1985
-1	-0.875	313.1214	-0.125	0.125	313.7644
-0.875	-0.875	307.5281	0	0.125	313.3445
-0.75	-0.875	308.9723	0.125	0.125	312.9334
-0.625	-0.875	308.0844	0.25	0.125	312.5335
-0.5	-0.875	308.8238	0.375	0.125	312.1423
-0.375	-0.875	308.2134	0.5	0.125	311.7726
-0.25	-0.875	308.7822	0.625	0.125	311.3768
-0.125	-0.875	308.1024	0.75	0.125	311.0604
0	-0.875	308.5798	0.875	0.125	310.3116
0.125	-0.875	307.8301	1	0.125	309.9195
0.25	-0.875	308.2494	-1	0.25	320
0.375	-0.875	307.4445	-0.875	0.25	318.6916
0.5	-0.875	307.8551	-0.75	0.25	317.3317
0.625	-0.875	306.9644	-0.625	0.25	316.6353
0.75	-0.875	307.7603	-0.5	0.25	315.9764
0.875	-0.875	306.2918	-0.375	0.25	315.4435
1	-0.875	311.9322	-0.25	0.25	314.9426
-1	-0.75	296.8331	-0.125	0.25	314.4683
-0.875	-0.75	309.2784	0	0.25	314.0042
-0.75	-0.75	307.0262	0.125	0.25	313.5519
-0.625	-0.75	309.3982	0.25	0.25	313.1127
-0.5	-0.75	308.0882	0.375	0.25	312.6964
-0.375	-0.75	309.4708	0.5	0.25	312.2892
-0.25	-0.75	308.1215	0.625	0.25	311.932
-0.125	-0.75	309.3239	0.75	0.25	311.4866
0	-0.75	307.8925	0.875	0.25	311.2977

0.125	-0.75	309.0165	1	0.25	310.8379
0.25	-0.75	307.5192	-1	0.375	320
0.375	-0.75	308.5991	-0.875	0.375	319.0062
0.5	-0.75	306.9864	-0.75	0.375	318.0994
0.625	-0.75	308.1196	-0.625	0.375	317.3503
0.75	-0.75	305.6377	-0.5	0.375	316.736
0.875	-0.75	307.8583	-0.375	0.375	316.1836
1	-0.75	295.4651	-0.25	0.375	315.6665
-1	-0.625	313.8047	-0.125	0.375	315.1576
-0.875	-0.625	308.9229	0	0.375	314.6512
-0.75	-0.625	309.9338	0.125	0.375	314.1519
-0.625	-0.625	309.6211	0.25	0.375	313.6722
-0.5	-0.625	309.7973	0.375	0.375	313.2217
-0.375	-0.625	309.6997	0.5	0.375	312.8122
-0.25	-0.625	309.6801	0.625	0.375	312.4411
-0.125	-0.625	309.5037	0.75	0.375	312.1355
0	-0.625	309.3924	0.875	0.375	311.8978
0.125	-0.625	309.1487	1	0.375	311.809
0.25	-0.625	308.9828	-1	0.5	320
0.375	-0.625	308.6862	-0.875	0.5	319.2691
0.5	-0.625	308.5091	-0.75	0.5	318.5694
0.625	-0.625	308.1164	-0.625	0.5	317.9632
0.75	-0.625	308.2898	-0.5	0.5	317.4075
0.875	-0.625	307.2335	-0.375	0.5	316.8933
1	-0.625	312.1751	-0.25	0.5	316.3767
-1	-0.5	299.1369	-0.125	0.5	315.8453
-0.875	-0.5	310.4339	0	0.5	315.2864
-0.75	-0.5	309.4006	0.125	0.5	314.7299
-0.625	-0.5	310.5492	0.25	0.5	314.1947
-0.5	-0.5	310.3689	0.375	0.5	313.7062
-0.375	-0.5	310.4966	0.5	0.5	313.2726
-0.25	-0.5	310.2645	0.625	0.5	312.9116
-0.125	-0.5	310.2193	0.75	0.5	312.6177
0	-0.5	309.9156	0.875	0.5	312.4417
0.125	-0.5	309.8044	1	0.5	312.3536
0.25	-0.5	309.4464	-1	0.625	320
0.375	-0.5	309.3004	-0.875	0.625	319.4758
0.5	-0.5	308.8361	-0.75	0.625	318.9709
0.625	-0.5	308.7415	-0.625	0.625	318.4977
0.75	-0.5	307.4057	-0.5	0.625	318.0489
0.875	-0.5	308.3681	-0.375	0.625	317.5969
1	-0.5	297.1362	-0.25	0.625	317.1101
-1	-0.375	315.6718	-0.125	0.625	316.5517
-0.875	-0.375	310.8938	0	0.625	315.9254
-0.75	-0.375	311.7831	0.125	0.625	315.2785
-0.625	-0.375	311.4084	0.25	0.625	314.6699
-0.5	-0.375	311.4132	0.375	0.625	314.1277
-0.375	-0.375	311.2512	0.5	0.625	313.668
-0.25	-0.375	311.0878	0.625	0.625	313.2958
-0.125	-0.375	310.8741	0.75	0.625	313.019
0	-0.375	310.6458	0.875	0.625	312.8452

0.125	-0.375	310.3874	1	0.625	312.7859
0.25	-0.375	310.1243	-1	0.75	320
0.375	-0.375	309.8319	-0.875	0.75	319.6601
0.5	-0.375	309.5746	-0.75	0.75	319.3261
0.625	-0.375	309.2102	-0.625	0.75	319.004
0.75	-0.375	309.3206	-0.5	0.75	318.6812
0.875	-0.375	308.3106	-0.375	0.75	318.3377
1	-0.375	313.1591	-0.25	0.75	317.8977
-1	-0.25	301.924	-0.125	0.75	317.3505
-0.875	-0.25	312.7969	0	0.75	316.5558
-0.75	-0.25	311.765	0.125	0.75	315.7868
-0.625	-0.25	312.5031	0.25	0.75	315.0665
-0.5	-0.25	312.2373	0.375	0.75	314.4635
-0.375	-0.25	312.099	0.5	0.75	313.9699
-0.25	-0.25	311.8413	0.625	0.75	313.5848
-0.125	-0.25	311.5939	0.75	0.75	313.3054
0	-0.25	311.3061	0.875	0.75	313.1368
0.125	-0.25	311.0261	1	0.75	313.0782
0.25	-0.25	310.712	-1	0.875	320
0.375	-0.25	310.4212	-0.875	0.875	319.8327
0.5	-0.25	310.0366	-0.75	0.875	319.6665
0.625	-0.25	309.8211	-0.625	0.875	319.5024
0.75	-0.25	308.6937	-0.5	0.875	319.3339
0.875	-0.25	309.5006	-0.375	0.875	319.1439
1	-0.25	298.671	-0.25	0.875	318.8969
-1	-0.125	319.1742	-0.125	0.875	318.2943
-0.875	-0.125	314.1519	0	0.875	317.2789
-0.75	-0.125	314.2695	0.125	0.875	316.1912
-0.625	-0.125	313.5865	0.25	0.875	315.3484
-0.5	-0.125	313.3034	0.375	0.875	314.6839
-0.375	-0.125	312.9571	0.5	0.875	314.1618
-0.25	-0.125	312.6419	0.625	0.875	313.7634
-0.125	-0.125	312.3176	0.75	0.875	313.4808
0	-0.125	311.9928	0.875	0.875	313.3112
0.125	-0.125	311.6627	1	0.875	313.2546
0.25	-0.125	311.3336	-1	1	320
0.375	-0.125	310.9983	-0.875	1	320
0.5	-0.125	310.6974	-0.75	1	320
0.625	-0.125	310.3441	-0.625	1	320
0.75	-0.125	310.4205	-0.5	1	320
0.875	-0.125	309.8461	-0.375	1	320
1	-0.125	314.8604	-0.25	1	320
-1	0	310.8629	-0.125	1	320
-0.875	0	316.1226	0	1	317.6592
-0.75	0	314.7712	0.125	1	316.3545
-0.625	0	314.7219	0.25	1	315.4545
-0.5	0	314.1988	0.375	1	314.7616
-0.375	0	313.8213	0.5	1	314.2279
-0.25	0	313.4224	0.625	1	313.8242
1	1	313.3127	0.75	1	313.5395
			0.875	1	313.3696

5. Team Members' Contribution

Khushi Agrawal - 210514

Contributed to the code in Processor1 folder and Super_Processor.

Ratanlal Sahu - 200774

Contributed to the code in Processor2, Processor3 folders and Post_Processor.

Sanyam Singla - 200879

Contributed to the report and Pre_Processor.