ME574 Project Ancillary File

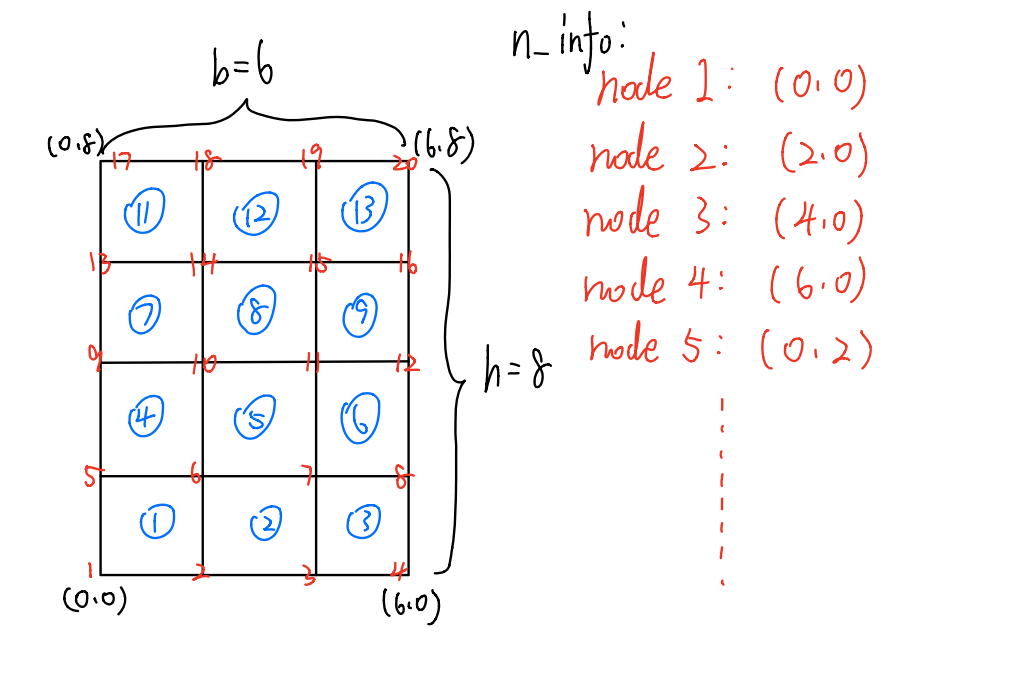
Juntao Zhang & Zhaoyi Jiang

In this project, students used python to do a Finite Element Analysis of a rectangular shape plate to calculate its Second Moment of Inertia, which is J. The code will be explained in the following paragraphs.

# Code explained:

The length and the width of the rectangle was defined as in the picture. b is the length and h is the width. n\_b and n\_h are the element numbers. G and theta can be any number. The picture shows the rectangle and the parameters. The elements were number in blue and nodes were numbered in red. They all followed the same order: from left to right and bottom to top.

In the first part below, we need to find the coordinate of each node and list them in a single column called n\_info. So we have the code and the n\_info to be:



**import** numpy **as** np

b **=** 6

h **=** 8

n\_b **=** 3

n\_h **=** 4

G **=** 200

theta **=** 10

x\_coord **=** np**.**transpose(np**.**kron(np**.**ones((1,n\_h**+**1)), np**.**linspace(0,b,n\_b**+**1)))

y\_coord **=** np**.**transpose(np**.**asmatrix(np**.**repeat(np**.**linspace(0,h,n\_h**+**1), n\_b**+**1)))

n\_info **=** np**.**hstack((x\_coord, y\_coord))

print('n\_info')

print(n\_info)

n\_info

[[0. 0.]

[2. 0.]

[4. 0.]

[6. 0.]

[0. 2.]

[2. 2.]

[4. 2.]

[6. 2.]

[0. 4.]

[2. 4.]

[4. 4.]

[6. 4.]

[0. 6.]

[2. 6.]

[4. 6.]

[6. 6.]

[0. 8.]

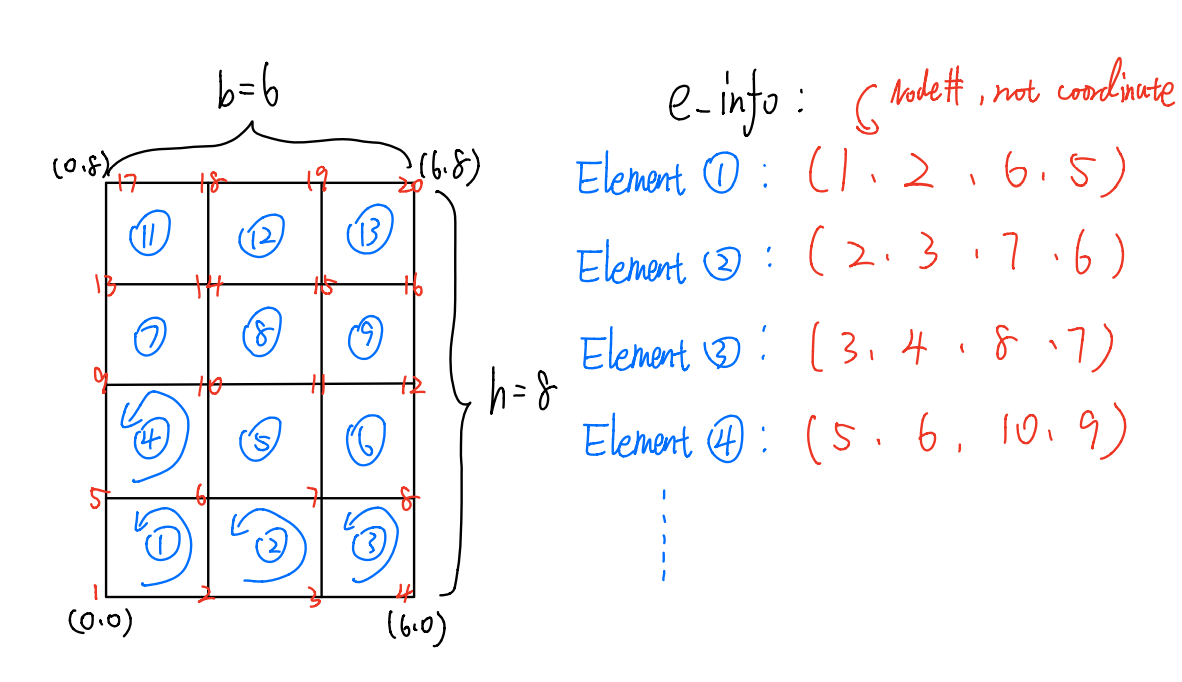
[2. 8.]

[4. 8.]

[6. 8.]]

-------------------------------------------------------------------------------------------------------------------------------

In the second part below, we need to list the number of nodes around each element in the order of counterclockwise in a row, and list all rows in a single column. It will be called e\_info. So we get e\_info to be:



**for** i **in** range(n\_h):

**for** j **in** range (n\_b):

top\_lef **=** (i**+**1**-**1)**\***(n\_b**+**1)**+**j**+**1

top\_rig **=** top\_lef**+**1

bot\_lef **=** top\_lef**+**(n\_b**+**1)

bot\_rig **=** top\_rig**+**(n\_b**+**1)

add **=** np**.**asmatrix(np**.**hstack((bot\_lef, bot\_rig, top\_rig, top\_lef)))

e\_matrix **=** np**.**vstack((e\_matrix, add))

e\_matrix **=** np**.**delete(e\_matrix, 0, 0)

e\_info **=** np**.**fliplr(e\_matrix)

print('e\_info')

print(e\_info)

e\_info

[[ 1. 2. 6. 5.]

[ 2. 3. 7. 6.]

[ 3. 4. 8. 7.]

[ 5. 6. 10. 9.]

[ 6. 7. 11. 10.]

[ 7. 8. 12. 11.]

[ 9. 10. 14. 13.]

[10. 11. 15. 14.]

[11. 12. 16. 15.]

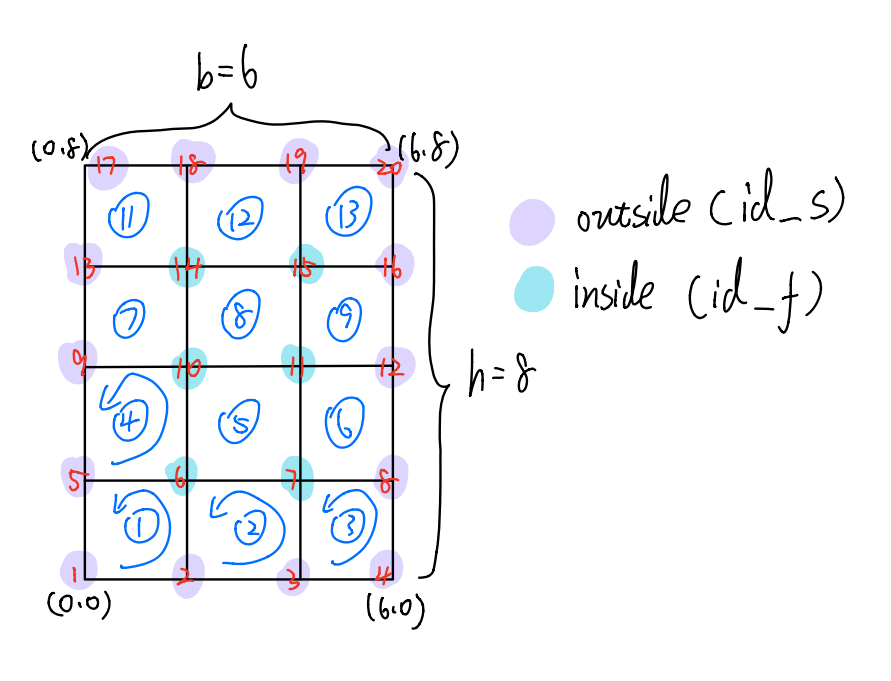
[13. 14. 18. 17.]

[14. 15. 19. 18.]

[15. 16. 20. 19.]]

-------------------------------------------------------------------------------------------------------------------------------

In the third part, we firstly listed all nodes and rearrange them in the actual way. Since we need to separate the node on the outside and inside, we made all nodes at the outside to be zero and got rid of them to get the nodes inside as id\_f, and we subtracted id\_f from all nodes to get id\_s, which is the list of nodes at outside.



all\_node **=** np**.**linspace(1, (n\_b**+**1)**\***(n\_h**+**1),(n\_b**+**1)**\***(n\_h**+**1))

all\_node\_op **=** np**.**transpose(np**.**asmatrix(np**.**linspace(1, (n\_b**+**1)**\***(n\_h**+**1),(n\_b**+**1)**\***(n\_h**+**1))))

all\_matrix **=** np**.**flipud(np**.**reshape(all\_node\_op, (n\_h**+**1, n\_b**+**1)))

all\_matrix[0,] **=** 0

all\_matrix[**-**1,] **=** 0

all\_matrix[:,0] **=** 0

all\_matrix[:,**-**1] **=** 0

id\_f **=** np**.**transpose(np**.**sort(all\_matrix[all\_matrix **!=** 0], axis**=None**))

id\_s **=** np**.**transpose(np**.**asmatrix(np**.**setdiff1d(all\_node, id\_f)))

print('all\_node')

print(all\_node)

print('all\_matrix')

print(all\_matrix)

print()

print('id\_f')

print(id\_f)

print()

print('id\_s')

print(id\_s)

all\_node

[ 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18.

19. 20.]

all\_matrix

[[ 0. 0. 0. 0.]

[ 0. 14. 15. 0.]

[ 0. 10. 11. 0.]

[ 0. 6. 7. 0.]

[ 0. 0. 0. 0.]]

id\_f

[[ 6.]

[ 7.]

[10.]

[11.]

[14.]

[15.]]

id\_s

[[ 1.]

[ 2.]

[ 3.]

[ 4.]

[ 5.]

[ 8.]

[ 9.]

[12.]

[13.]

[16.]

[17.]

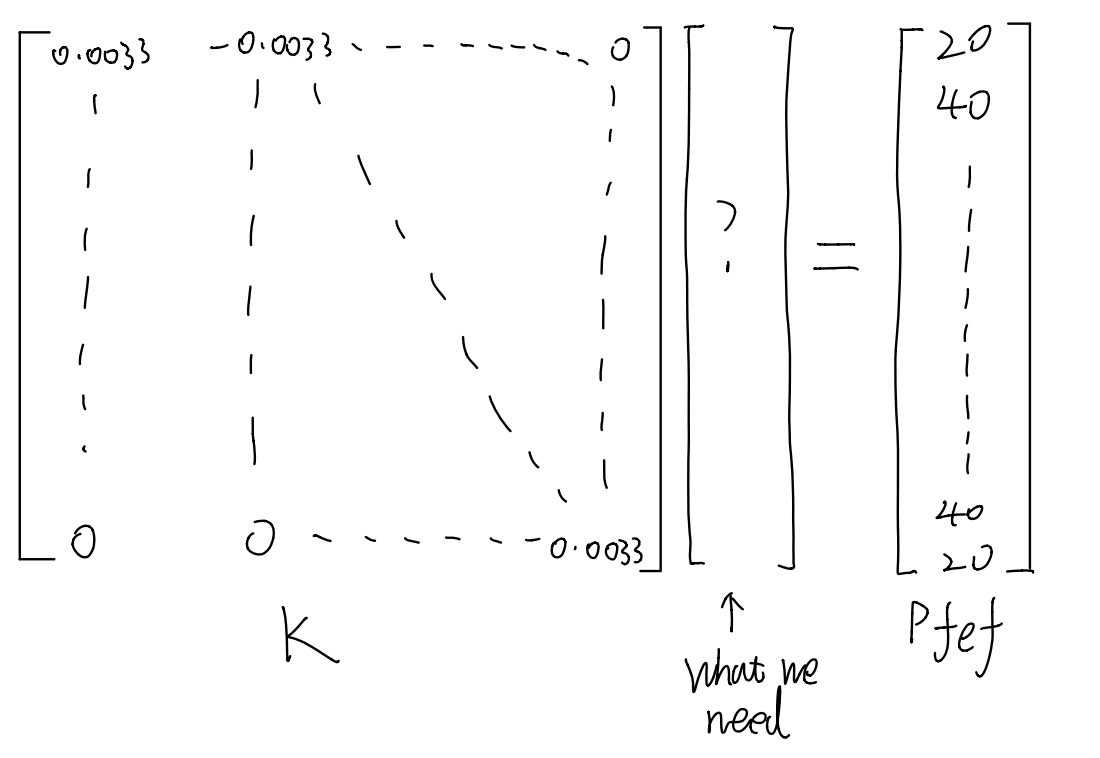
[18.]

[19.]

[20.]]

-------------------------------------------------------------------------------------------------------------------------------

In the Forth Part, we calculated the stiffness matrix, K, which refers to the property of all nodes. It is on the left hand of the equation. We also calculated the Pfef, which is the reaction results of all nodes. It is on the right hand side of the equation. The code and results are here:



K **=** np**.**zeros((num\_node, num\_node))

num\_e **=** np**.**size(e\_info, axis**=**0)

**for** e **in** range(num\_e):

i **=** int(e\_info[e,0])

j **=** int(e\_info[e,1])

k **=** int(e\_info[e,2])

l **=** int(e\_info[e,3])

Lx **=** n\_info[j**-**1,0]**-**n\_info[i**-**1,0]

Ly **=** n\_info[k**-**1,1]**-**n\_info[j**-**1,1]

r **=** Ly**/**Lx

k11 **=** (1**+**r**\*\***2)**/**(3**\***r)

k12 **=** 1**/**(6**\***r)**-**r**/**3

k13 **=** **-**(1**+**r**\*\***2)**/**(6**\***r)

k14 **=** (**-**2**+**r**\*\***2)**/**(6**\***r)

ke **=** (1**/**G)**\***np**.**asmatrix([[k11,k12,k13,k14],

[k12,k11,k14,k13],

[k13,k14,k11,k12],

[k14,k13,k12,k11]])

K[i**-**1,i**-**1] **=** K[i**-**1,i**-**1]**+**ke[0,0]

K[i**-**1,j**-**1] **=** K[i**-**1,j**-**1]**+**ke[0,1]

K[i**-**1,k**-**1] **=** K[i**-**1,k**-**1]**+**ke[0,2]

K[i**-**1,l**-**1] **=** K[i**-**1,l**-**1]**+**ke[0,3]

K[j**-**1,i**-**1] **=** K[j**-**1,i**-**1]**+**ke[1,0]

K[j**-**1,j**-**1] **=** K[j**-**1,j**-**1]**+**ke[1,1]

K[j**-**1,k**-**1] **=** K[j**-**1,k**-**1]**+**ke[1,2]

K[j**-**1,l**-**1] **=** K[j**-**1,l**-**1]**+**ke[1,3]

K[k**-**1,i**-**1] **=** K[k**-**1,i**-**1]**+**ke[2,0]

K[k**-**1,j**-**1] **=** K[k**-**1,j**-**1]**+**ke[2,1]

K[k**-**1,k**-**1] **=** K[k**-**1,k**-**1]**+**ke[2,2]

K[k**-**1,l**-**1] **=** K[k**-**1,l**-**1]**+**ke[2,3]

K[l**-**1,i**-**1] **=** K[l**-**1,i**-**1]**+**ke[3,0]

K[l**-**1,j**-**1] **=** K[l**-**1,j**-**1]**+**ke[3,1]

K[l**-**1,k**-**1] **=** K[l**-**1,k**-**1]**+**ke[3,2]

K[l**-**1,l**-**1] **=** K[l**-**1,l**-**1]**+**ke[3,3]

print('K')

print(K)

K

[[ 0.00333333 -0.00083333 0. 0. -0.00083333 -0.00166667

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[-0.00083333 0.00666667 -0.00083333 0. -0.00166667 -0.00166667

-0.00166667 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[ 0. -0.00083333 0.00666667 -0.00083333 0. -0.00166667

-0.00166667 -0.00166667 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[ 0. 0. -0.00083333 0.00333333 0. 0.

-0.00166667 -0.00083333 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[-0.00083333 -0.00166667 0. 0. 0.00666667 -0.00166667

0. 0. -0.00083333 -0.00166667 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[-0.00166667 -0.00166667 -0.00166667 0. -0.00166667 0.01333333

-0.00166667 0. -0.00166667 -0.00166667 -0.00166667 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[ 0. -0.00166667 -0.00166667 -0.00166667 0. -0.00166667

0.01333333 -0.00166667 0. -0.00166667 -0.00166667 -0.00166667

0. 0. 0. 0. 0. 0.

0. 0. ]

[ 0. 0. -0.00166667 -0.00083333 0. 0.

-0.00166667 0.00666667 0. 0. -0.00166667 -0.00083333

0. 0. 0. 0. 0. 0.

0. 0. ]

[ 0. 0. 0. 0. -0.00083333 -0.00166667

0. 0. 0.00666667 -0.00166667 0. 0.

-0.00083333 -0.00166667 0. 0. 0. 0.

0. 0. ]

[ 0. 0. 0. 0. -0.00166667 -0.00166667

-0.00166667 0. -0.00166667 0.01333333 -0.00166667 0.

-0.00166667 -0.00166667 -0.00166667 0. 0. 0.

0. 0. ]

[ 0. 0. 0. 0. 0. -0.00166667

-0.00166667 -0.00166667 0. -0.00166667 0.01333333 -0.00166667

0. -0.00166667 -0.00166667 -0.00166667 0. 0.

0. 0. ]

[ 0. 0. 0. 0. 0. 0.

-0.00166667 -0.00083333 0. 0. -0.00166667 0.00666667

0. 0. -0.00166667 -0.00083333 0. 0.

0. 0. ]

[ 0. 0. 0. 0. 0. 0.

0. 0. -0.00083333 -0.00166667 0. 0.

0.00666667 -0.00166667 0. 0. -0.00083333 -0.00166667

0. 0. ]

[ 0. 0. 0. 0. 0. 0.

0. 0. -0.00166667 -0.00166667 -0.00166667 0.

-0.00166667 0.01333333 -0.00166667 0. -0.00166667 -0.00166667

-0.00166667 0. ]

[ 0. 0. 0. 0. 0. 0.

0. 0. 0. -0.00166667 -0.00166667 -0.00166667

0. -0.00166667 0.01333333 -0.00166667 0. -0.00166667

-0.00166667 -0.00166667]

[ 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. -0.00166667 -0.00083333

0. 0. -0.00166667 0.00666667 0. 0.

-0.00166667 -0.00083333]

[ 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

-0.00083333 -0.00166667 0. 0. 0.00333333 -0.00083333

0. 0. ]

[ 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

-0.00166667 -0.00166667 -0.00166667 0. -0.00083333 0.00666667

-0.00083333 0. ]

[ 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. -0.00166667 -0.00166667 -0.00166667 0. -0.00083333

0.00666667 -0.00083333]

[ 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. -0.00166667 -0.00083333 0. 0.

-0.00083333 0.00333333]]

num\_e **=** np**.**size(e\_info, axis**=**0)

**for** e **in** range(num\_e):

i\_node **=** int(e\_info[e,0])

j\_node **=** int(e\_info[e,1])

k\_node **=** int(e\_info[e,2])

l\_node **=** int(e\_info[e,3])

Lx **=** n\_info[j\_node**-**1,0]**-**n\_info[i\_node**-**1,0]

Ly **=** n\_info[k\_node**-**1,1]**-**n\_info[j\_node**-**1,1]

Pfefe **=** Lx**\***Ly**\***2**\***theta**\***np**.**asmatrix([[1**/**4],[1**/**4],[1**/**4],[1**/**4]])

Pfef[i\_node**-**1] **=** Pfef[i\_node**-**1]**+**Pfefe[0]

Pfef[j\_node**-**1] **=** Pfef[j\_node**-**1]**+**Pfefe[1]

Pfef[k\_node**-**1] **=** Pfef[k\_node**-**1]**+**Pfefe[2]

Pfef[l\_node**-**1] **=** Pfef[l\_node**-**1]**+**Pfefe[3]

print('Pfef')

print(Pfef)

Pfef

[[20.]

[40.]

[40.]

[20.]

[40.]

[80.]

[80.]

[40.]

[40.]

[80.]

[80.]

[40.]

[40.]

[80.]

[80.]

[40.]

[20.]

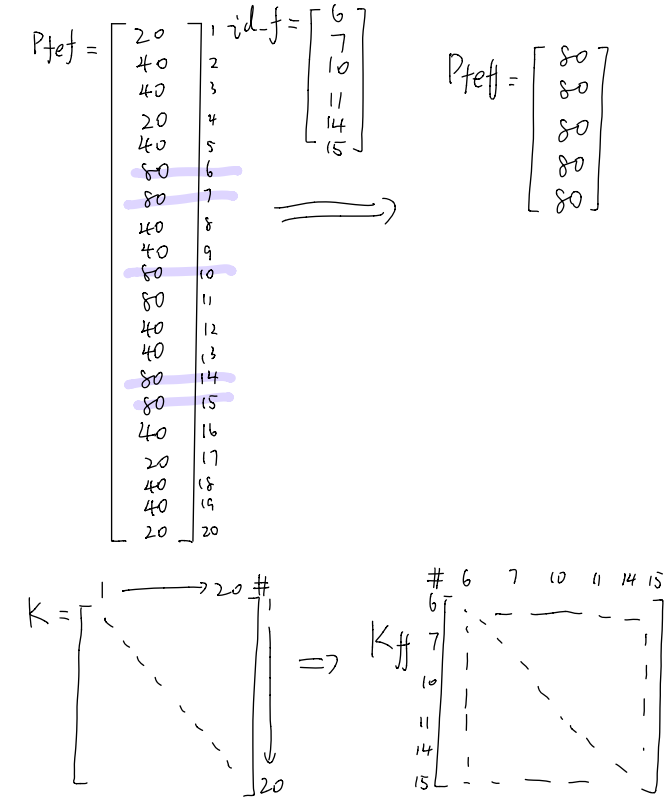
[40.]

[40.]

[20.]]

-------------------------------------------------------------------------------------------------------------------------------

In the fifth part, we used id\_f, which is the number of inside nodes, to reduce the size of Pfef and K to Pfeff and Kff, which are what really matter regarding to calculation of J. Here are the code and results:



Pfef\_list **=** np**.**asarray(Pfef)**.**reshape(**-**1)**.**tolist()

id\_f\_index **=** np**.**asarray(id\_f)**.**reshape(**-**1)**.**tolist()

id\_f\_index[:] **=** [id\_f\_index **-** 1 **for** id\_f\_index **in** id\_f\_index]

Pfeff **=** np**.**asmatrix(np**.**take(Pfef\_list, id\_f\_index))**.**transpose()

print('Pfeff')

print(Pfeff)

Pfeff

[[80.]

[80.]

[80.]

[80.]

[80.]

[80.]]

In [17]:

Kff **=** np**.**zeros((np**.**size(id\_f), np**.**size(id\_f)))

**for** i **in** range(np**.**size(id\_f)):

**for** j **in** range(np**.**size(id\_f)):

Kff[i,j] **=** K[int(id\_f[i])**-**1,int(id\_f[j])**-**1]

print('Kff')

print(Kff)

Kff

[[ 0.01333333 -0.00166667 -0.00166667 -0.00166667 0. 0. ]

[-0.00166667 0.01333333 -0.00166667 -0.00166667 0. 0. ]

[-0.00166667 -0.00166667 0.01333333 -0.00166667 -0.00166667 -0.00166667]

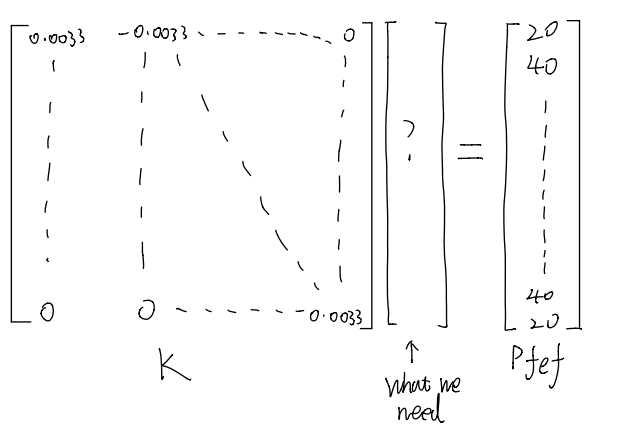
[-0.00166667 -0.00166667 -0.00166667 0.01333333 -0.00166667 -0.00166667]

[ 0. 0. -0.00166667 -0.00166667 0.01333333 -0.00166667]

[ 0. 0. -0.00166667 -0.00166667 -0.00166667 0.01333333]]

-------------------------------------------------------------------------------------------------------------------------------

In the sixth part, we calculated the df matrix using Kff and Pfeff. Then we used the df matrix and area of each element to calculated the J.



area\_elm **=** (b**/**n\_b)**\***(h**/**n\_h)

vol **=** sum(df**\***area\_elm)

J **=** 2 **\*** vol**/**(G**\***theta)

print('df')

print(df)

print()

print('area\_elm')

print(area\_elm)

print()

print('J')

print(J)

df

[[10536.58536585]

[10536.58536585]

[12878.04878049]

[12878.04878049]

[10536.58536585]

[10536.58536585]]

area\_elm

4.0

J

[[271.6097561]]

# Parallel computing explained

When running the code we found the most time consuming part is the reduction of K, which is how we get Kff in the fifth part. We used parallel method to reduced the running time of that part. The total time was reduced by a factor 40. Here is the code and results:

**from** numba **import** cuda

*# @cuda.jit(device=True)*

*# def arrsub(ele):*

*# return ele-1*

@cuda**.**jit()

**def** update\_kernel(d\_out,d\_id\_f,d\_k):

nx,ny **=** d\_id\_f**.**shape[0], d\_id\_f**.**shape[0]

i,j **=** cuda**.**grid(2)

**if** i **<** nx:

**if** j **<** ny:

*# cx = arrsub(id\_f[i])*

*# cy = arrsub(id\_f[j])*

d\_out[i,j] **=** d\_k[d\_id\_f[i]**-**1,d\_id\_f[j]**-**1]

*# for i in range(len(id\_f)):*

*# tmp = np.asscalar(id\_f.A[i])*

*# # print(id\_f.astype(int).reshape(-1,1).A[i])*

*# b.append(tmp)*

**def** caller(id\_f,K):

tmp **=**[]

**for** i **in** range(len(id\_f)):

tmp**.**append(id\_f**.**A[i]**.**item())

tmp **=** np**.**array(tmp)**.**astype(int)

d\_k **=** cuda**.**to\_device(K)

nx,ny **=** np**.**size(id\_f),np**.**size(id\_f)

*# print(tmp)*

d\_id\_f **=** cuda**.**to\_device(tmp)

d\_out **=** cuda**.**device\_array([nx,ny])

*# Kff = np.zeros((np.size(id\_f), np.size(id\_f)))*

TPB **=** 16

threads **=** TPB, TPB

blocks **=** (nx**+**TPB**-**1)**//**TPB, (ny**+**TPB**-**1)**//**TPB

e\_start **=** cuda**.**event()

e\_end **=** cuda**.**event()

e\_start**.**record()

update\_kernel[blocks, threads](d\_out,d\_id\_f, d\_k)

e\_end**.**record()

e\_end**.**synchronize()

kernel\_duration **=** cuda**.**event\_elapsed\_time(e\_start, e\_end)

**return** d\_out**.**copy\_to\_host()

p\_Kff**=**caller(id\_f,K)

df **=** np**.**dot(np**.**linalg**.**inv(p\_Kff), Pfeff)

area\_elm **=** (b**/**n\_b)**\***(h**/**n\_h)

vol **=** sum(df**\***area\_elm)

J **=** 2 **\*** vol**/**(G**\***theta)

Time of Parallel: 0.7673 seconds

Time of Serial: 29.5563 seconds