

# Final Project for Advanced FEM (ME46050)

XUSEN QIN

Student ID: 5594979, email X.Qin-2@student.tudelft.nl, edition: 2022-2023

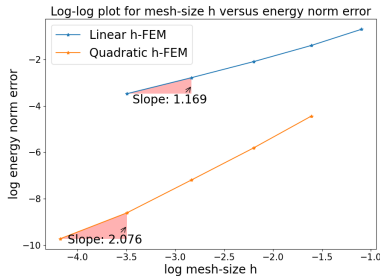
## I. INTRODUCTION

**T**his report addresses two advanced finite element problems. The first problem constructs a solver for a one-dimensional Poisson equation using both h-version and p-version finite elements. The second problem develops a solver for a two-dimensional stress distribution of elliptical inhomogeneity in plane elasticity, employing the h-version FEM with T3 elements and Q4 elements.

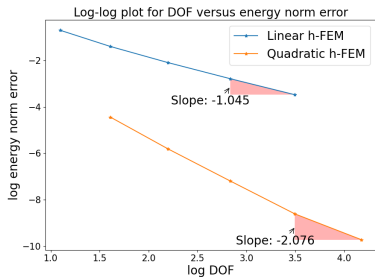
## II. PROBLEM 1

### i. Question 1

The definition of the 1-D Poisson equation is in the Appendix.?? The code for finite element method, shape functions as well as the Gaussian integration method in Appendix.??, ??, and ??.



(a) log-log plot for the error versus meshsize



(b) log-log plot for the error versus DOF

**Figure 1:** The log-log figure for the energy norm error versus mesh size and DOF.

The precise strain energy for this problem is given

as  $U=0.03559183822564316$ . To determine the rates of convergence in the energy norm for both element types, we focus on terminal convergence by considering the last two points in the convergence plots.

The formula for the convergence rate can be found in Eq.??, which can also be defined as the slope of the log-log plot. The mesh size  $h$  is defined as  $h = \frac{1}{\text{DOF}^d}$ , where  $d$  is the dimensionality of the problem. In this case, we take  $d = 1$ . For both elements, the error decreases with the increase of the DOFs and the decrease of the mesh size. It's noteworthy that the convergence rate for the quadratic elements is approximately greater than that for the linear elements. Given the smoothness of the solution, the theoretical rates of convergence are typically -1.045 for linear elements and -2.076 for quadratic elements concerned with DOFs, which is equal to the order of the polynomials. For the computed errors, the linear elements exhibit an error of approximately 0.031, while the quadratic elements have a significantly smaller error of about  $6.0 \times 10^{-5}$ . These computed rates align closely with the theoretical expectations.

$$\text{Rate} = \frac{\log(\text{error}_2) - \log(\text{error}_1)}{\log(\text{DOF}_2) - \log(\text{DOF}_1)} \quad (1)$$

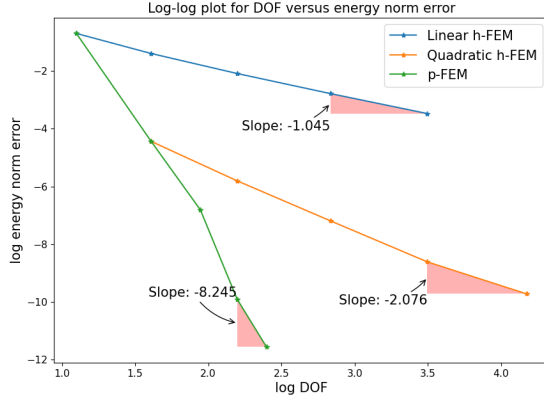
### ii. Question 2

In the log-log plot in Fig.?? of the relative error in the energy norm versus the number of DOFs, the slopes of the plotted lines represent these rates. The following items are observed.

- For Linear h-FEM: The rate of convergence is approximately -1.045.
- For Quadratic h-FEM: The rate of convergence is approximately -2.076.
- For p-FEM: The rate of convergence is approximately -8.230.

The negative values for the convergence rates indicate that the error decreases as the number of DOFs increases, which is expected in a convergence study. Notably, the rate of convergence of the linear element is close to 1, and the quadratic element is close to 2, respectively, which indicates that the convergence rate of h-FEM is equal to the polynomial order. From the

rates, it's evident that the p-FEM has the steepest convergence, indicating a faster reduction in error with increasing DOFs compared to the other methods.



**Figure 2:** log-log plot of the error versus DOF in h-version and p-version FEM.

### iii. Question 3

In order to estimate the error in finite element method solutions, we use a posteriori error analysis based on the energy norms described in the following processes.

Considering the algebraic convergence of the energy norm error for exact solution  $u$  and the finite elements solution  $u_h$  in energy space  $\epsilon(\Omega)$ :

$$\|u - u^h\|_{\epsilon(\Omega)} \leq C_1 h^{\beta_h} \|u\|_{\epsilon(\Omega)} \quad (2)$$

We went:

$$\|u\|_{\epsilon(\Omega)} = \sqrt{U} \quad (3)$$

where  $U$  is the exact strain energy.

Considering the relation between the energy and binary term in the finite element methods.

$$\begin{aligned} u(u) &= \frac{1}{2} B(u, u), \\ \|u\|_e &= \sqrt{\frac{1}{2} B(u, u)}, \\ \|u - u^h\|_e &= \frac{1}{2} B(u - u^h, u - u^h) \\ &= \frac{1}{2} B(u, u) - \frac{1}{2} B(u^h, u^h), \end{aligned} \quad (4)$$

Now we obtain the error of the strain energy:

$$U_e = U - U^h. \quad (5)$$

By using the energy values obtained from three different mesh sizes, a system of equations can be constructed to determine the exact solution  $U$ :

$$\begin{aligned} U - U^{h_0} &= C_1^2 h_0^{2\beta_h} U \quad (\text{I}) \\ U - U^{h_1} &= C_1^2 h_1^{2\beta_h} U \quad (\text{II}) \\ U - U^{h_2} &= C_1^2 h_2^{2\beta_h} U \quad (\text{III}) \end{aligned} \quad (6)$$

In these equations:

- $U^{h_0}$ ,  $U^{h_1}$ , and  $U^{h_2}$  are the FEM approximated solutions for mesh sizes  $h_0$ ,  $h_1$ , and  $h_2$  respectively.
- $C_1$  is a coefficient.
- $\beta_h$  is an exponent that determines the convergence rate of error reduction as mesh size decreases.

The logarithmic relationship between the errors for different mesh sizes can be obtained by Eq.??:

$$\begin{aligned} \text{Take } \frac{\log(\text{I})}{\log(\text{II})} : \log \left( \frac{U - U^{h_0}}{U - U^{h_1}} \right) &= 2\beta_h \log \left( \frac{h_0}{h_1} \right) \\ \text{Take } \frac{\log(\text{II})}{\log(\text{III})} : \log \left( \frac{U - U^{h_1}}{U - U^{h_2}} \right) &= 2\beta_h \log \left( \frac{h_1}{h_2} \right) \end{aligned} \quad (7)$$

These equations provide insight into how the error changes logarithmically as the mesh size changes.

Using the above relationships, the a posteriori error estimate, which is a measure of the relative error, is expressed as:

$$\frac{\log \left( \frac{U - U^{h_0}}{U - U^{h_1}} \right)}{\log \left( \frac{U - U^{h_1}}{U - U^{h_2}} \right)} = \frac{\log \left( \frac{h_0}{h_1} \right)}{\log \left( \frac{h_1}{h_2} \right)} = Q \quad (8)$$

Considering the relation between the mesh size  $h$  and the DOF ( $N$ ):

$$h \cong \frac{1}{N^{1/\text{dimensionality}}} \quad (9)$$

The expression of  $Q$  is given by:

$$Q = \frac{\log(N_1/N_0)}{\log(N_2/N_1)} \quad (10)$$

The term  $Q$  gives a weighted comparison of the errors between different mesh sizes. This relationship becomes pivotal in understanding the error behavior across different mesh sizes.

By repeatedly applying the aforementioned process for multiple mesh sizes and averaging the computed energies, a more accurate representation of the solution's energy is achieved, which provides a reliable posterior error estimate.

	Energy	Relative Error
Linear	0.034626674	2.7117(%)
Quadratic	0.035591726	0.000314(%)
Exact solution	0.035591838	/

**Table 1:** Energy obtained by a posterior estimate and Relative Error values for different FEM methods

As shown in the table.??, for the linear FEM, the energy is computed to be 0.03463, which results in a relative error of 2.7117%. This indicates a slight deviation from the exact solution. On the other hand, the quadratic FEM provides an energy value of 0.03559, which is extremely close to the exact solution with a minuscule relative error of 0.000314%. This suggests that the quadratic FEM is significantly more accurate than the linear FEM for this problem.

In summary, while the linear FEM offers a reasonable approximation, the quadratic FEM provides an almost exact match to the true solution in terms of energy.

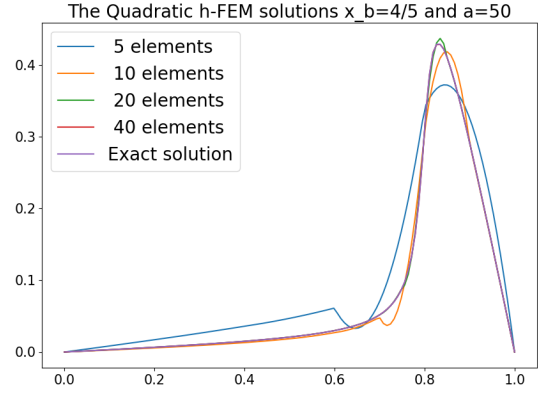
The code for a posterior estimate is provided in the Appendix.??.

#### iv. Question 4

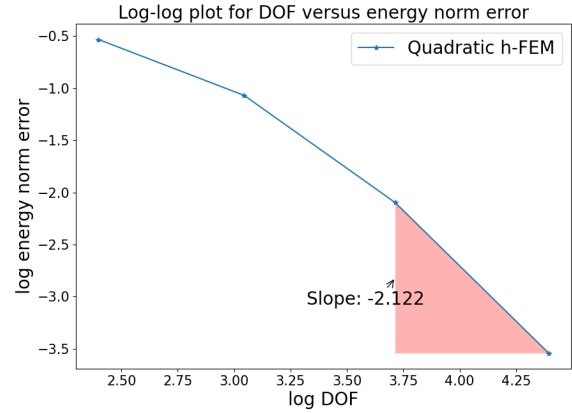
In the h-version study using the quadratic finite element method, we analyzed the model with varying mesh sizes, namely 5, 10, 20, and 40 evenly spaced elements. Fig.?? represents the h-FEM solutions with four mesh sizes. A comparison of the numerical solutions against the exact solution provided insights into the accuracy of the employed method. From Fig.?? it was discernible that the graph wasn't strictly linear. However, by focusing on the terminal two data points, we derived an asymptotic rate of convergence of  $-2.122$ . This suggests a quadratic rate of reduction in error relative to the refinement in element size. For this specific problem, the exact strain energy is given by  $U = 1.585854059271320$ , and the computed results  $U_{FEM} = 1.5845186616720888$  is close to this value.

#### v. Question 5

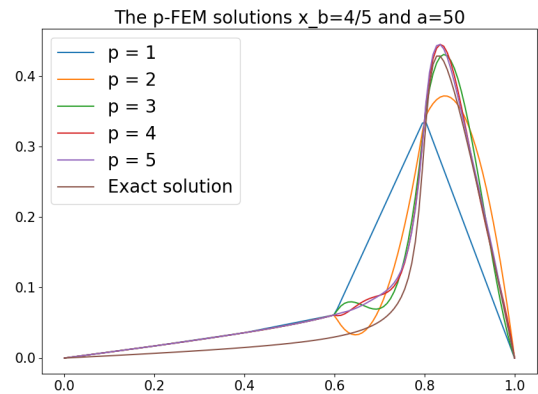
From the log-log plot in Fig.??, the computed rate of convergence for the p-version was approximately  $-4.882$ , whereas for the h-version, it was  $-2.2122$ . The convergence rate of quadratic p-FEM is faster than h-FEM in this problem. As a result, the p-FEM can achieve higher accuracy with fewer degrees of freedom.



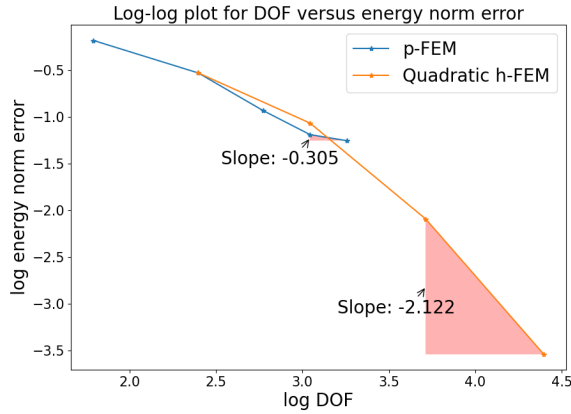
**Figure 3:** The Quadratic h-FEM solutions  $x_b=4/5$  and  $a=50$  with different element numbers.



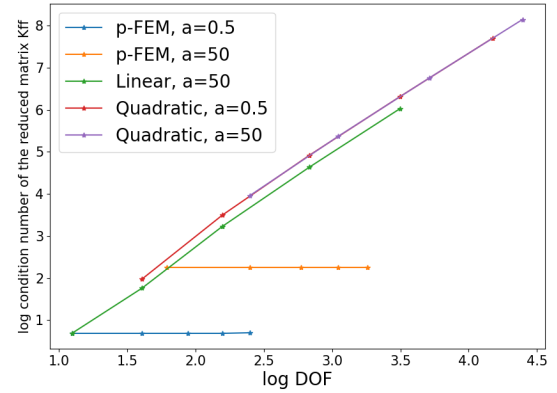
**Figure 4:** Log-log plot for DOF versus energy norm error.



**Figure 5:** The p-FEM solutions  $x_b=4/5$  and  $a=50$  with different element numbers.



**Figure 6:** Log-log plot for DOF versus energy norm error in p-FEM and h-FEM.



**Figure 7:** Log-log plot for the condition number of the reduced matrix  $K_{ff}$  versus energy norm error.

## vi. Question 6

The stability of numerical methods in finite element analysis can be assessed using the condition number of the stiffness matrix.

Observing the log-log plot in Fig.??:

- **p-FEM:** The condition number remains constant regardless of the DOFs increase, indicating its robustness.
- **Quadratic h-FEM:** Condition number growth is consistent with increasing DOFs and is unaffected by equation parameter changes (both for  $a = 0.5$  and  $a = 50$ ).
- **Linear vs Quadratic h-FEM:** Both show similar growth trends, but the linear version has a slightly lower condition number for similar DOFs.

In summary, p-FEM stands out in stability, while h-FEM versions show predictable growth trends, with the linear version in slightly better condition.

## vii. Question 7

### vii.1 Comparison of Results and Conclusions on Strong Gradients

From the results obtained, several conclusions can be drawn regarding the behavior of the finite element methods under study, especially in problems with strong gradients or sharp features.

- **Convergence Rate and Accuracy:** The convergence rate, represented as the slope of the log-log plot, provides insights into the efficacy of the different finite element methods. The error decreased

with the increase of the DOFs and the decrease of the mesh size. Notably, the quadratic elements exhibited a more significant convergence rate than the linear ones, reflecting the theoretical expectations. Meanwhile, the rate of convergence for h-FEM is equal to the polynomial order.

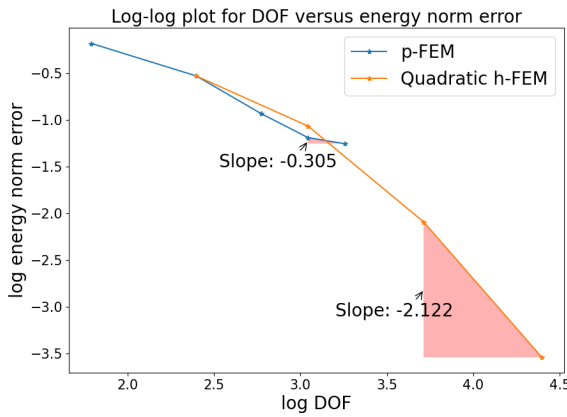
- **Linear h-version:** For the linear FEM, the energy was computed to be somewhat deviated from the exact solution, especially in problems with sharp features. This indicates that while the linear h-FEM offers a reasonable approximation, there's a clear margin for improvement in accuracy for such problems.
- **Quadratic h-version:** In sharp gradient problems, the quadratic h-FEM showed its strength by providing an energy value that was extremely close to the exact solution, emphasizing its higher accuracy.
- **p-version vs. h-version in Sharp Problems:** In problems with sharp gradients, the p-version exhibited remarkable resilience and adaptability. Despite its slower convergence rate, it outperformed both the linear and quadratic h-versions in terms of accuracy for comparable DOFs. This suggests that the p-version, with its adaptability, can better capture local variations and sharp features without requiring extensive mesh refinements that h-version methods might demand.
- **Effect of Strong Gradients:** The p-FEM is particularly effective for problems with strong gradients or sharp features. Its higher-order polynomial approximations and local refinement capabilities allow it to capture complex variations in the solution more accurately than methods like h-FEM.

This adaptability often results in higher accuracy with fewer computational resources.

- **Stability and Robustness:** The stability of finite element methods, assessed by the condition number of the stiffness matrix, highlighted the robustness of the p-FEM. Its condition number remains invariant with increasing DOFs, ensuring consistent performance. On the other hand, the h-FEM versions, both linear and quadratic, exhibit predictable growth in condition numbers, with the linear version showing a slight edge in conditioning. The quadratic h-FEM's stability remains consistent even with changes in equation parameters.

### vii.2 Quadrature Points and Computation Efficiency

In p-FEM, higher-order shape functions demand precise integral evaluations, achieved effectively with Gauss quadrature. Given the complexity of these functions, 9 Gauss points were selected to ensure accurate integration of non-linear shape functions. While more Gauss points increase precision, they also require more computational effort. As is shown in Fig.??, when setting the Gaussian point as 6, the result of p-FEM cannot represent the convergence well in a strong gradient problem.



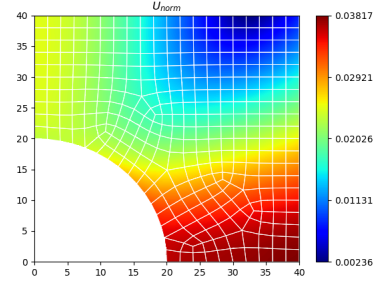
**Figure 8:** Log-log plot for DOF versus energy norm error in p-FEM and h-FEM when Gaussian points are selected as 6.

## III. PROBLEM 2

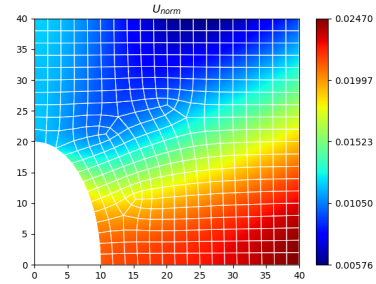
### i. Questions 1

The definition of the traction-free hole in an infinite plate with different  $a/b$  ratios is shown in Appendix.??

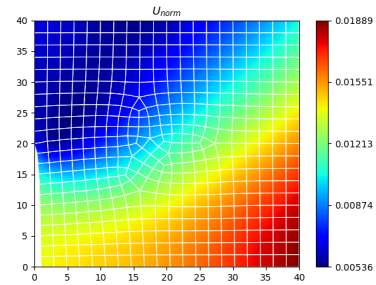
of the report. The code for finite element method, shape functions as well as the Gaussian integration method in Appendix.??, ??, and ??. For the mesh size  $h/L = 0.05$  and Q4 mesh, the displacement fields for different  $a/b$  are represented in Fig.??.



(a) Displacement field for  $a/b=1$



(b) Displacement field for  $a/b=0.5$



(c) Displacement field for  $a/b=0.05$

**Figure 9:** displacement fields for different  $a/b$  : (a)  $a/b=1$ ; (b)  $a/b=0.5$ ; and (c)  $a/b=0.05$ .

### ii. Question 2

The accurate stress solution at each point is computed using the given formula [?]. This is then multiplied by the inverse of the stiffness matrix  $C$  in plain stress assumption, which is given by:

$$C = \frac{E}{1-\nu^2} \begin{pmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{pmatrix} \quad (11)$$

to obtain the strain at each point. Due to the presence of an ellipse in the middle of the plate, the value of the strain at the edge point cannot represent the total strain of the part. Hence, the total strain at the edge points is simplified as the exact strain of the edge point plus the strain at the ellipse vertices:

$$\epsilon_x(L, 0) = \epsilon_x(L, 0) + \epsilon_x(a, L),$$

$$\epsilon_y(L, 0) = \epsilon_y(L, 0) + \epsilon_y(L, b).$$

The displacements are then approximated as displacement = strain  $\times$  40mm.

The results and relative error are tabulated in Table.?? and Table.??.

a/b ratio	$U_{x,cal}$	$U_{x,FEM}$	$U_{y,cal}$	$U_{y,FEM}$
1	0.01588	0.03816	0.0004068	-0.02473
0.5	0.01688	0.0247	-0.00484	-0.01205
0.05	0.0176	0.01889	-0.00536	-0.00658

**Table 2:** Computed displacement by simplified assumptions for different a/b ratios

a/b ratio	Error in $U_x$ (%)	Error in $U_y$ (%)
1	58.39	101.64
0.5	31.66	59.83
0.05	6.83	18.54

**Table 3:** Relative Errors Between Calculated and FEM-obtained Displacements for Different a/b Ratios

Table ?? shows that the relative errors in displacements  $U_x$  and  $U_y$  decrease as the a/b ratio diminishes. The highest errors occur at  $a/b = 1$ , reaching up to 101.64% for  $U_y$ , suggesting the model's reduced reliability for circular holes. In contrast, the model appears more accurate for narrower ellipses, with errors below 20% at  $a/b = 0.05$ .

### iii. Question 3

The computed strain energy values for different a/b ratios using different mesh types (T3 and Q4) are presented in Table.?. Due to the limitations in calculating the exact strain energy, the computed values from the numerical simulations are considered representative

for each mesh type (T3 and Q4). The table shows that the strain energy values vary with the a/b ratio. Specifically, the energy values are highest for a ratio of 1 and decrease as the ratio decreases. This suggests that the structure is more energetically stable when a/b is closer to 1.

**Table 4:** Computed strain energy values for different a/b ratios (unit kJ).

a/b ratio	T3	Q4	Average
1	25.898	22.337	24.1175
0.5	17.145	18.377	17.761
0.05	15.449	14.443	14.946

While we cannot compare these values to the exact strain energy due to computational constraints, the exact strain energy in the following questions is the energy calculated by a posterior error estimate.

### iv. Question 4

The log-log plot for energy norm error versus mesh size and DOF are represented in Fig.??

**Table 5:** Convergence Rates based on Mesh Size

a/b Ratio	Element Type	Convergence Rate
1	T3	0.092621
1	Q4	0.736803
0.5	T3	0.221261
0.5	Q4	-0.143155
0.05	T3	0.483199
0.05	Q4	0.249004

**Table 6:** Convergence Rates based on DOFs

a/b Ratio	Element Type	Convergence Rate
1	T3	-0.054701
1	Q4	-0.486695
0.5	T3	-0.131251
0.5	Q4	0.091080
0.05	T3	-0.275252
0.05	Q4	-0.171758

Considering the fluctuation of the error is sensitive to the mesh size (the error reaches the lowest when the mesh size is 4), the convergence rates are calculated by the first point and the last point of the energy list.

The convergence rates for different elements with different a/b ratios concerning the mesh size and DOF are represented in Table.?? and Table.??



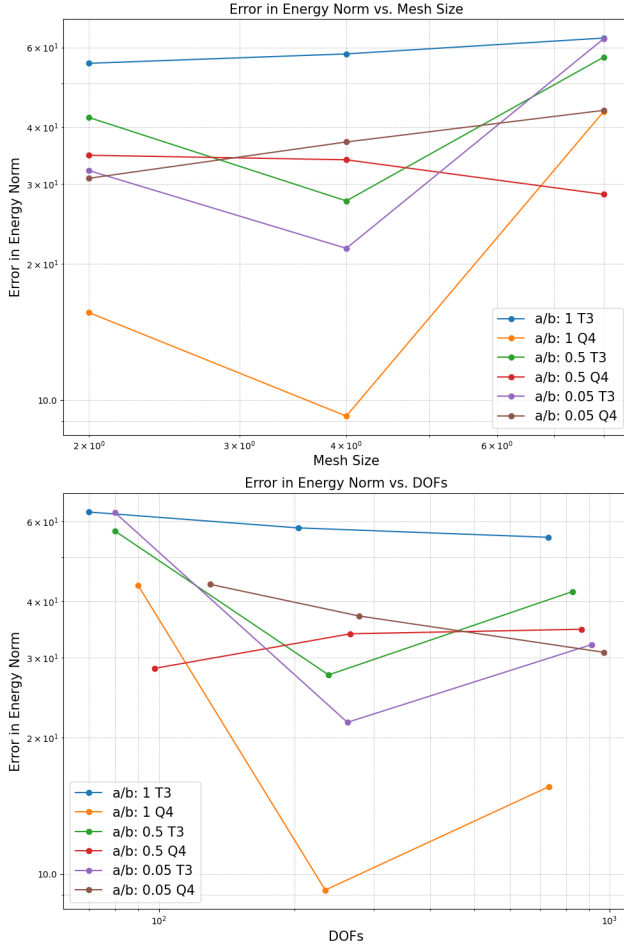


Figure 10: Log-log plot of energy norm error versus DOFs.

### Convergence Rates based on Mesh Size

- For T3 elements with  $a/b$  ratios of 1, 0.5, and 0.05, the observed convergence rates are 0.092621, 0.221261, and 0.483199, respectively.
- For Q4 elements with  $a/b$  ratios of 1, 0.5, and 0.05, the observed convergence rates are 0.736803, -0.143155, and 0.249004, respectively.

### Convergence Rates based on DOFs

- For T3 elements with  $a/b$  ratios of 1, 0.5, and 0.05, the observed convergence rates are -0.054701, -0.131251, and -0.275252, respectively.
- For Q4 elements with  $a/b$  ratios of 1, 0.5, and 0.05, the observed convergence rates are -0.486695, 0.091080, and -0.171758, respectively.

It is observed that only for  $a/b = 1$  in T3 mesh and  $a/b = 0.5$  in Q4 mesh, the log-log plots appear to be linear. However, the convergence rates for all

curves are lower than the expected theoretical values. The convergence rates of each mesh type in h-FEM in 2D are equal to the order of the interpolation polynomials over 2. Therefore, the experimentally observed convergence rates do not align with the theoretical predictions. However, from a trend perspective, it is observed that the error decreases as the mesh size gets smaller and also diminishes as the degrees of freedom (DOF) increase.

### v. Question 5

Fig.??, Fig.??, Fig.??, Fig.?? and Fig.?? represents the stress field of  $a/b$  ratio equal to 1, 0.5, 0.05. For the Q4 element, the max stress value is selected by superconvergent patch recovery (SPR). The definition of the SPR method is in Appendix.?? The relative errors between the exact analytical solution and the FEM-obtained stress values are presented in Table ??.

Table 7: Relative Errors Between Exact and FEM-obtained Stresses for Different  $a/b$  Ratios

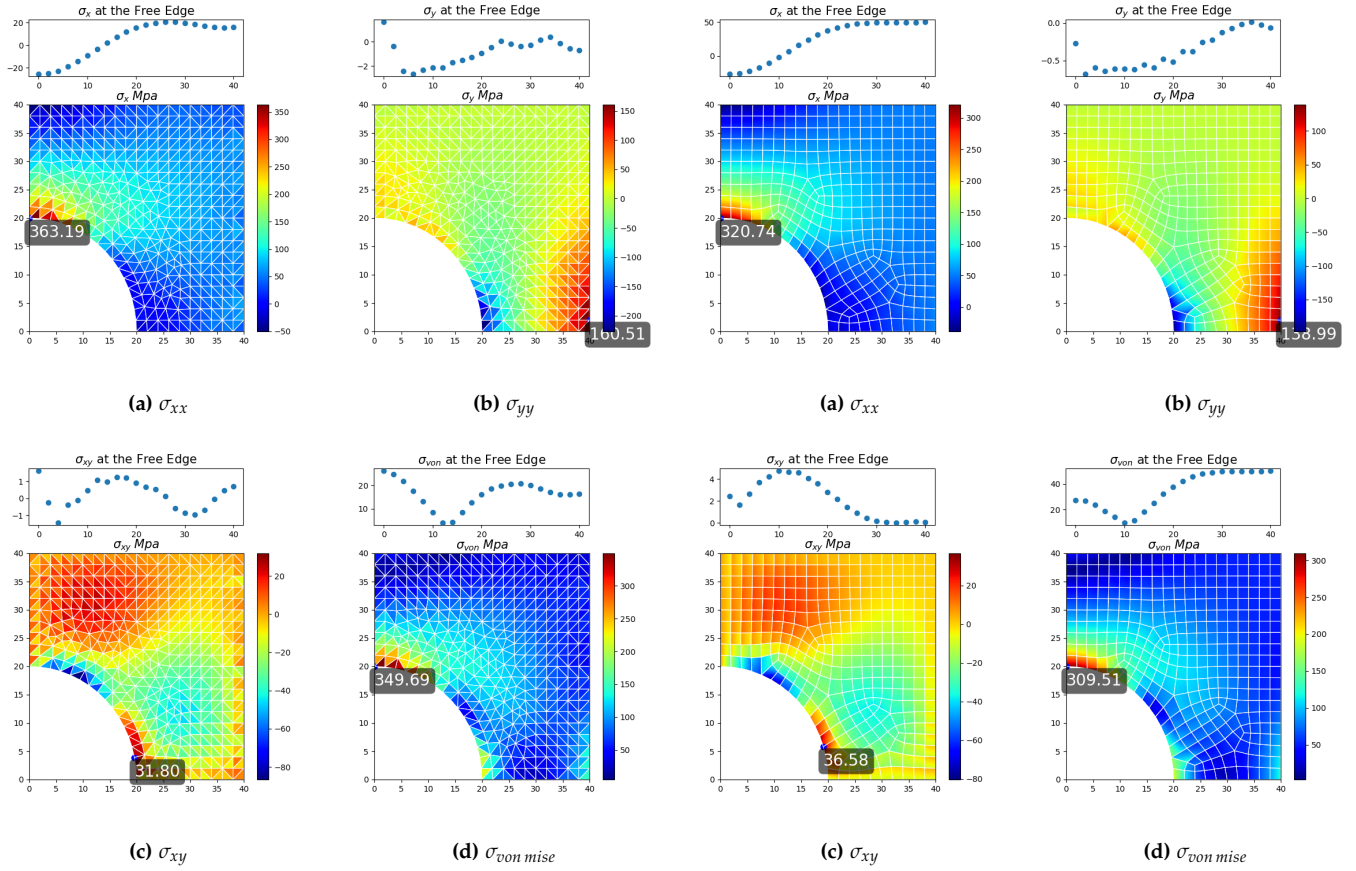
$a/b$ ratio	Error in $\sigma_x$ (%)	Error in $\sigma_y$ (%)
1	57.33	805.00
0.5	40.39	1130.77
0.05	31.37	1820.00

Based on the relative error data, it is evident that the stress values have a higher level of relative error compared to the displacements. The relative errors in stress range from 31.37% to 1820.00%, while for displacements, they range from 6.83% to 101.64%. Therefore, in this particular case, the displacements appear to be more accurately predicted than the stresses.

### vi. Question 6

The Table.?? presents the stress concentration factors (SCF) for T3 and Q4 elements at different  $a/b$  ratios. It is evident that the experimentally observed SCFs are significantly different from the theoretical predictions (given by  $K_c = 1 + 2\frac{b}{a}$ ).

- For an  $a/b$  ratio of 1, both T3 and Q4 elements show SCFs (7.3 and 6.4, respectively) that are much higher than the theoretical value of 3. The average SCF is 6.85, which is more than twice the theoretical prediction.
- At an  $a/b$  ratio of 0.5, the SCFs for T3 and Q4 are 8.4 and 7.8, respectively, with an average of 8.1.


 Figure 11: Stress fields for  $a/b=1$  with T3 mesh.

This is also significantly higher than the theoretical value of 5.

- Interestingly, for an  $a/b$  ratio of 0.05, the SCFs are lower than the theoretical value. The SCFs for T3 and Q4 are 5.8 and 7, respectively, with an average of 6.4, which is far below the theoretical value of 41.

Table 8: Stress Concentration Factors for T3 and Q4 Elements

$a/b$ Ratio	T3	Q4	Average	Theory
1	7.3	6.4	6.85	3
0.5	8.4	7.8	8.1	5
0.05	5.8	7	6.4	41

## vii. Question 7

The allowable stress values for the material under non-failing conditions at different  $a/b$  ratios are presented in the following Table.?. Von Mises criterion is se-

 Figure 12: Stress fields for  $a/b=1$  with Q4 mesh.

lected as the criterion for the maximum allowable stress (shown in Appendix.?).

$a/b$ ratio	T3	Q4	Average
1	30	33.9	31.95
0.5	26.4	28.9	27.65
0.05	31.4	27.3	29.35

 Table 9: Allowable stress values for different  $a/b$  ratios.

The table shows that the material has relatively consistent allowable stress values across different  $a/b$  ratios. Specifically, the average allowable stress is approximately 30, varying slightly from 27.65 to 31.95. This suggests that the material's allowable stress is not significantly influenced by the  $a/b$  ratio, indicating good material robustness under varying conditions.



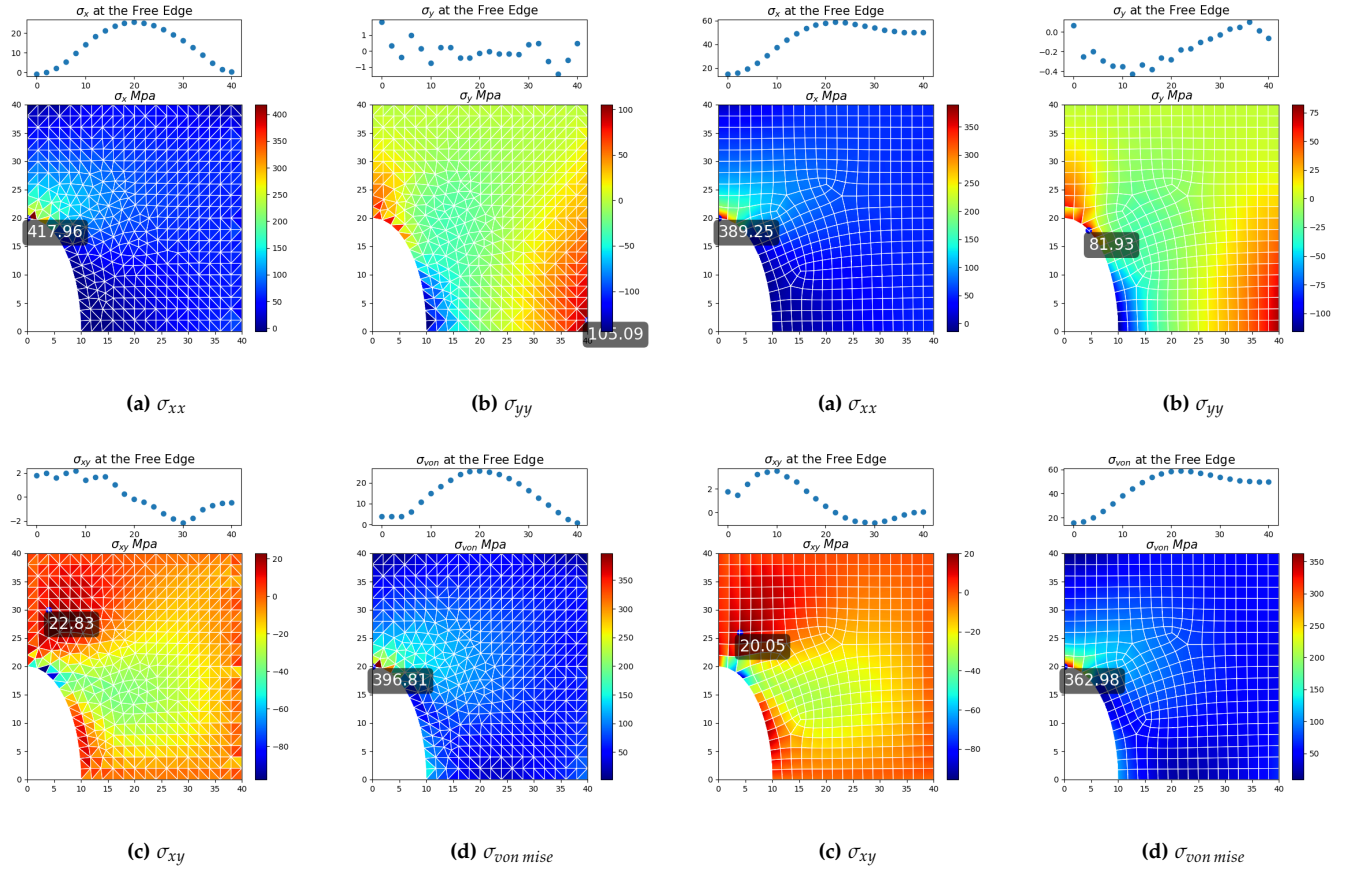

 Figure 13: Stress fields for  $a/b=0.5$  with T3 mesh.

 Figure 14: Stress fields for  $a/b=0.5$  with Q4 mesh.

### viii. Question 8

#### Strain Energy and Structural Integrity

The calculated strain energy values for the three different  $a/b$  ratios indicate varying levels of structural integrity. For  $a/b = 1$ , the strain energy was highest, which suggests that the structure is energetically less stable when the hole is a circle. This could be attributed to higher localized stresses around the hole, making the structure more susceptible to energy accumulation.

For  $a/b = 0.5$ , the strain energy reduced compared to  $a/b = 1$ . This indicates that the structure becomes more stable as the hole becomes less elliptical. This could be due to a more uniform stress distribution around the hole, thus lowering the overall strain energy.

Interestingly, the lowest strain energy was observed for  $a/b = 0.05$ . This can be attributed to the shape of the hole, which is a very narrow ellipse at this ratio. In essence, the structure behaves almost like a solid plate with a crack, rather than a plate with a hole. The narrowness of the ellipse minimizes the global

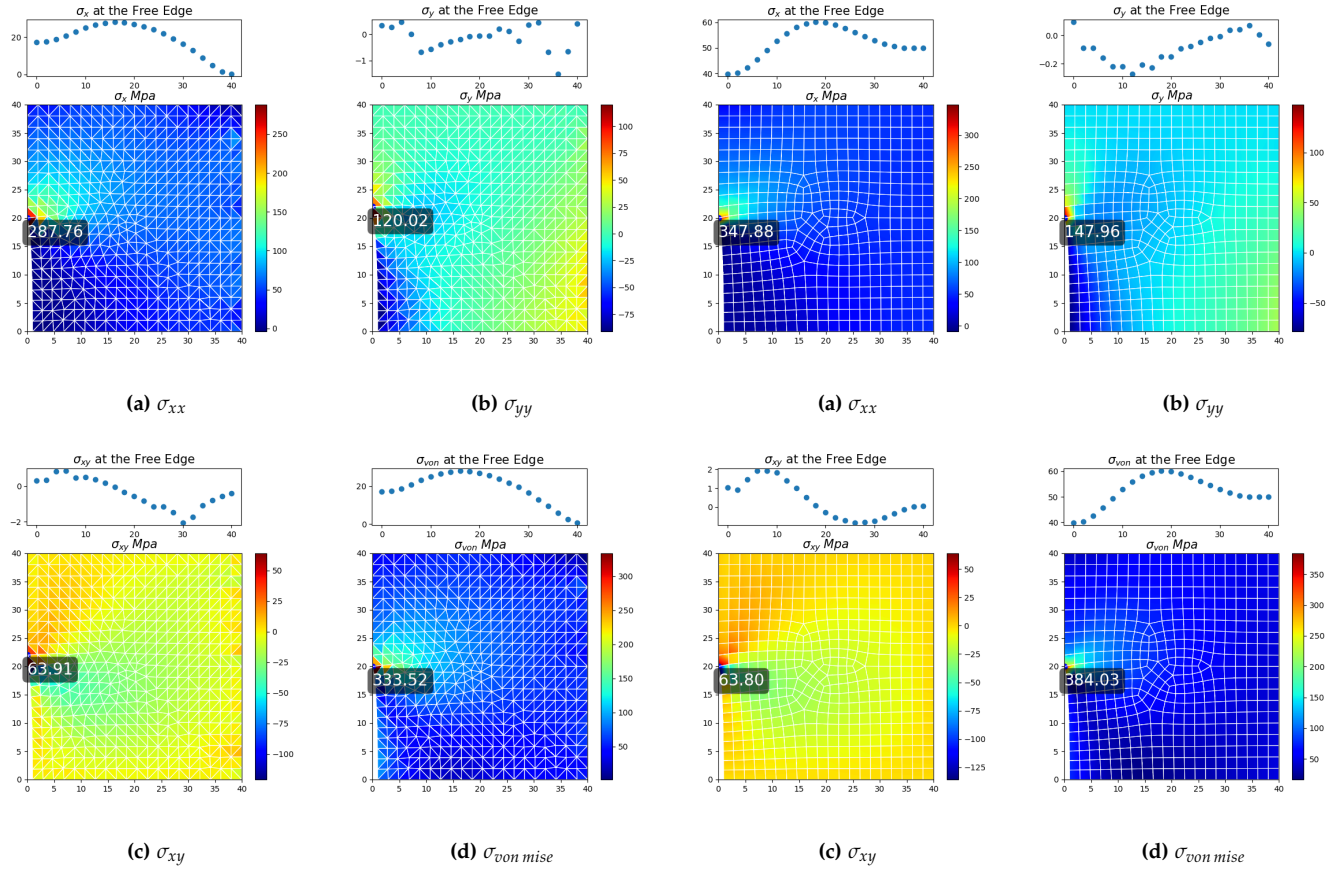
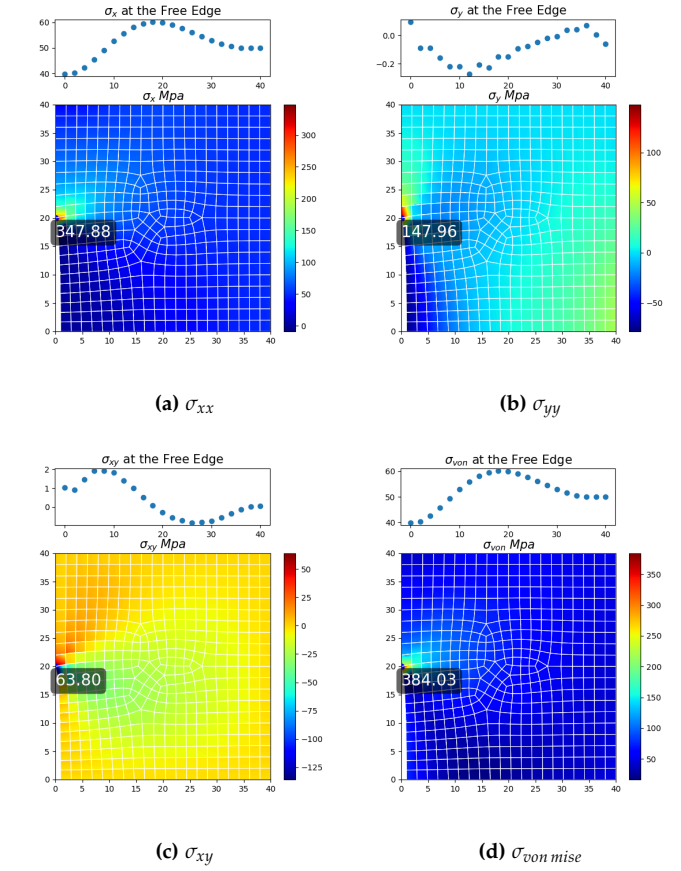
deformation, making the structure resemble a nearly intact plate.

#### Convergence Rates

It was observed that the convergence rates deviated from the theoretical predictions, especially when  $a/b = 1$ . This could be attributed to the narrow ellipse at the center when  $a/b = 1$ , making the structure highly sensitive to mesh size. A fine mesh is needed for more accurate results, particularly in regions with strong gradients or stress concentrations.

#### Stress and Displacement Errors

Both stress and displacement errors were considered. The stress errors were notably higher than those for displacements. For instance, stress errors ranged from 31.37% to 1820.00%, whereas displacement errors were between 6.83% and 101.64%. This suggests that the Finite Element model is more reliable for predicting displacements than for stresses.

Figure 15: Stress fields for  $a/b=0.05$  with T3 mesh.Figure 16: Stress fields for  $a/b=0.05$  with Q4 mesh.

## Stress Concentration Factors (SCF)

The SCFs deviated significantly from theoretical values, particularly at  $a/b = 1$  and  $a/b = 0.5$ . This could be due to the coarse mesh used in the simulations. Also, the stress values at these concentrations were observed to increase as the mesh size decreased, possibly indicating the presence of a singularity. A singularity in this context means that the stress becomes infinite at a point, which is a known issue in the Finite Element Analysis of problems with sharp corners or re-entrant angles.

## Mesh Sensitivity

The observed sensitivity to mesh size, especially at  $a/b = 1$ , indicates that the simulations are less reliable in capturing the true behavior of the structure at this particular geometry. The abrupt stress values at points of stress concentration further support this observation. Therefore, the mesh size and element type play a crucial role in the simulation's reliability.

## Validity of the Simulations

While the study offers valuable insights into the performance of T3 and Q4 elements under various conditions, the findings should be interpreted with caution due to several limitations, such as mesh sensitivity and the potential for singularities. Furthermore, the observed convergence rate of the energy norm error deviates from the theoretical rate, particularly when the ratio  $a/b=0.3$ . For T3 elements in this case, the convergence rate exhibits a counterintuitive trend compared to other results. As such, a more refined definition of the mesh is warranted.

## REFERENCES

- [Jin et al., 2014] Jin, Xiaoqing and Wang, Zhanjiang and Zhou, Qinghua and Keer, Leon M and Wang, Qian (2014). On the solution of an elliptical inhomogeneity in plane elasticity by the equivalent inclusion method. *Journal of Elasticity*, 114:1–18.

## A. DEFINITION OF THE 1-D POISSON EQUATION

The 1-D Poisson equation is given as:

$$-\frac{d^2u}{dx^2} = \frac{2(a + a^3b(b-x+1))}{(a^2b^2 + 1)^2} \quad (12)$$

with boundary conditions (BCs)  $u(0) = u(1) = 0$ , where  $a$  is constant and  $b = x - x_b$ . The solution to this boundary value problem (BVP) is given by

$$u(x) = (1-x) (\arctan(ab) + \arctan(ax_b)) \quad (13)$$

## B. FINITE ELEMENTS METHODS IN 1-D MAIN CODES

```

1  def FEM_1D(shape_class = Hierarchical
    , p = 3, num_elems = 3, domain =
      (0, 1), rhs_func = rhs_fn(a=50, xb
        =0.8), exact_func=exact_fn
          (0.5,0.8), BCs = (0, 0), verbose
            = False):
2      N=6
3      mesh = np.linspace(domain[0],
        domain[1], num_elems+1)
4      ori_phi_phip = {'phis': [], '
        phips': []}
5      for elem in range(num_elems):
6          scale = [mesh[elem], mesh[
            elem+1]]
7          phis, phips = shape_class(
            scale, p)
8          ori_phi_phip['phis'].append(
            phis)
9          ori_phi_phip['phips'].append(
            phips)
10
11
12     linear_phi_phip = {'phis': [], '
        phips': []} # Linear
13     for elem in range(num_elems):
14         linear_phis = []
15         linear_phips = []
16         for idx in range(len(
            ori_phi_phip['phis'][elem
            ])):
17             if ori_phi_phip['phis'][
                elem][idx].p < 2:
18                 phi = ori_phi_phip['
                    phis'][elem][idx]
19                 phip = ori_phi_phip['
                    phips'][elem][idx]

```

```

20         linear_phi_phip['phis'
        ].append(phi)
21         linear_phi_phip['
            phips'].append(
                phip)
22         linear_phis.append(
            phi)
23         linear_phips.append(
            phip)
24     linear_K_sub = np.zeros((len(
        linear_phips), len(
            linear_phips)))
25     for indx, x in np.ndenumerate
        (linear_K_sub):
26         linear_K_sub[indx] =
            G_integrate(
27             mul(linear_phips[indx
                [0]],
                    linear_phips[indx
                        [-1]]), N=6,
                    scale=
                        linear_phips[indx
                            [0]].scale)
28         if abs(linear_K_sub[indx
            ]) < 1e-10:
29             linear_K_sub[indx] =
                0
30     linear_F_sub = np.zeros(len(
        linear_K_sub))
31     for indx in range(len(
        linear_F_sub)):
32         linear_F_sub[indx] =
            G_integrate(
33             mul(rhs_func,
                linear_phis[indx
                    ]), N=N, scale=
                        linear_phis[indx
                            ].scale)
34     if elem == 0:
35         K = linear_K_sub
36         F = linear_F_sub
37     else:
38         K = assemble(K,
            linear_K_sub)
39         F = assemble(F,
            linear_F_sub)
40
41     linear_num = len(F)
42
43     nonlinear_phi_phip = {'phis': [],
        'phips': []}
44     for order in range(2, p+1): #
        Non Linear
45         for elem in range(num_elems):
46             for indx in range(len(
                ori_phi_phip['phis'][

```

<pre> 47         elem])):             if (ori_phi_phip['                 phis'][elem][idx                     ].p == order) or                 (ori_phi_phip['                     phips'][elem][idx                         ].p == order): 48                 nonlinear_phi =                     ori_phi_phip[                         'phis'][elem                             ][idx] 49                 nonlinear_phip =                     ori_phi_phip[                         'phips'][elem                             ][idx] 50                 nonlinear_phi_phip                     ['phis'].                     append(                         nonlinear_phi                     ) 51                 nonlinear_phi_phip                     ['phips'].                     append(                         nonlinear_phip                     ) 52                 nonlinear_K_sub =                     np.zeros((2,                         2)) 53 54                 nonlinear_K_sub                     [-1, -1] =                     G_integrate(                         mul(                             nonlinear_phip                             ,                             nonlinear_phip                         ), N=N, scale=                             nonlinear_phip                                 .scale) 55                 nonlinear_F_sub =                     np.zeros(2) 56                 nonlinear_F_sub                     [-1] =                     G_integrate(                         mul(rhs_func,                             nonlinear_phi                                 ), N=N, scale                                     =                             nonlinear_phi                                 .scale) 57 58                 K = assemble(K,                     nonlinear_K_sub                         ) </pre>	<pre> 59                 F = assemble(F,                     nonlinear_F_sub                         ) 60                 else: 61                     pass 62 63                 # Applying boundary condition 64 65                 K[0, 1:] = 0.0 66                 K[linear_num-1, :linear_num-1] =                     0.0 67                 F[0] = BCs[0]* K[0, 0] 68                 F[linear_num-1] = BCs[-1] * K[                     linear_num-1, linear_num-1] 69 70                 U = -la.solve(K, F) 71                 phi_phip = {'phis': [], 'phips':                     []} 72                 phi_phip['phis'] = joint_funcs(                     linear_phi_phip['phis']) +                     nonlinear_phi_phip['phis'] 73                 phi_phip['phips'] = joint_funcs(                     linear_phi_phip['phips']) +                     nonlinear_phi_phip['phips'] 74                 u_list = [] 75                 for i in range(len(phi_phip['phis                     '])): 76                     u_list.append(mul(U[i],                         phi_phip['phis'][i])) 77                 uh = plus(u_list) 78                 if verbose == True: 79                     print(f"Shape class: {                         shape_class.__name__},                             Number of elements: {                                 num_elems},                                     Polynomial   order: {p},   Domain: {   domain},   Boundary   conditions: {BCs}") 80                     x_data = np.linspace(domain                         [0], domain[1], 101) 81                     plt.plot(x_data, exact_func(                         x_data), label='                             Analytical solution') 82                     plt.plot(x_data, uh(x_data),                         label='FEM solution {                             elements'.format(                                 num_elems)) 83                     for i in range(len(phi_phip['                         phis'])): 84                         func = phi_phip['phis'][i                             ] 85                         plt.plot(x_data, U[i]*                             func(x_data)) 86                     plt.legend() 87                     plt.show() </pre>
---	--

```

88     eigenvalues = np.linalg.eigvals(K
89     )
90     cont_K = max(eigenvalues)/min(
        eigenvalues)
    return U, phi_phip, uh, cont_K

```

Listing 1: Finite elements methods in 1-D main code

### C. DEFINITION OF THE SHAPE FUNCTIONS IN 1D

```

1  def Legendre(x=np.linspace(-1, 1, 100),
2      p=5):
3
4      if p == 0:
5          return 1
6      elif p == 1:
7          return x
8
9      else:
10         return ((2*p-1)*x*Legendre(x, p
11             -1)+(1-p)*Legendre(x, p-2))/p
12
13 class shape_function:
14     def __init__(self, scale=[-1, 1]):
15         :
16         self.scale = scale
17         self.x_l = scale[0]
18         self.x_r = scale[1]
19         self.range = [-1, 1]
20
21     def expression(self, x):
22         return 1 - (x - self.x_l) / (
23             self.x_r - self.x_l)
24
25     def mapping(self, x):
26         scale = self.scale
27         range = self.range
28         x_normalized = (x - scale[0])
29             / (scale[1] - scale[0])
30         return range[0] +
31             x_normalized * (range[1]
32                 - range[0])
33
34     def __call__(self, x):
35         x = np.asarray(x) # convert
36             x to a numpy array if it'
37             s not already
38         expression_vectorized = np.
39             vectorize(self.expression
40                 , otypes=['d'])
41         return np.where((self.scale
42             [0] <= x) & (x <= self.
43                 scale[-1])),

```

```

        expression_vectorized(x),
        0)
31
32 class phi_func_l(shape_function):
33     def __init__(self, scale, p):
34         super().__init__(scale)
35         self.p = p
36         self.range = [0, 1]
37     def expression(self, x):
38         if self.p == 0:
39             phi = 1-self.mapping(x)
40         elif self.p == 1:
41             phi = self.mapping(x)
42         else:
43             raise AssertionError("p
44                 should be 0 or 1 in
45                 linear shape function,
46                 not {}".format(self.p))
47         return phi
48
49 class phip_func_l(shape_function):
50     def __init__(self, scale, p):
51         super().__init__(scale)
52         self.range = [0, 1]
53         self.p = p
54     def expression(self, x):
55         scale_up = 1/(self.scale[1]-self.
56             scale[0])
57
58         if self.p == 0:
59             phip = np.zeros_like(self.
60                 mapping(x))-1
61         elif self.p == 1:
62             phip = np.zeros_like(self.
63                 mapping(x))+1
64         else:
65             raise AssertionError("p
66                 should be 0 or 1 in
67                 linear shape function,
68                 not {}".format(self.p))
69         return phip*scale_up
70
71 class phi_func_q(shape_function):
72     def __init__(self, scale, p):
73         super().__init__(scale)
74         self.range = [0, 1]
75         self.p = p
76     def expression(self, x):
77         xx = self.mapping(x)
78         if self.p == -1:
79             phi = (xx-1)*(xx-0.5)*2
80         elif self.p == 0:
81             phi = -xx*(xx-1)*4
82         elif self.p == 1:
83             phi = xx*(xx-0.5)*2
84         else:

```

```

76         raise AssertionError("p
           should be -1, 0 or 1 in
           quadratic shape function,
           not {}".format(self.p))
77     return phi
78
79 class phip_func_q(shape_function):
80     def __init__(self, scale, p):
81         super().__init__(scale)
82         self.range = [0, 1]
83         self.p = p
84     def expression(self, x):
85         scale_up = 1/(self.scale[1]-self.
           scale[0])
86         xx = self.mapping(x)
87         if self.p == -1:
88             phip = 4*xx - 3.0
89         elif self.p == 0:
90             phip = 4-8*xx
91         elif self.p == 1:
92             phip = 4*xx - 1.0
93         else:
94             raise AssertionError("p
           should be -1, 0 or 1 in
           quadratic shape function,
           not {}".format(self.p))
95         return phip*scale_up
96
97 class phi_func_h(shape_function):
98     def __init__(self, scale, p):
99         super().__init__(scale)
100        self.p = p
101    def expression(self, x):
102        scale = self.scale
103        i = self.p
104        if i == 0:
105            phi = (1-self.mapping(x))/2
106        elif i == 1:
107            phi = (1+self.mapping(x))/2
108        else:
109            phi = 1/np.sqrt(4*i-2)*(
                Legendre(self.mapping(x),
                    i)-Legendre(self.mapping
                        (x), i-2))
110        return phi
111
112 class phip_func_h(shape_function):
113     def __init__(self, scale, p):
114         super().__init__(scale)
115         self.p = p
116     def expression(self, x):
117         scale_up = 2/(self.scale[1]-self.
           scale[0])
118         i = self.p
119
120         if i == 0:

```

```

121         phip = np.zeros_like(self.
           mapping(x))-0.5
122     elif i == 1:
123         phip = np.zeros_like(self.
           mapping(x))+0.5
124     else:
125         phip = np.sqrt(i-1/2)*(
           Legendre(self.mapping(x),
               i-1))
126         return phip*scale_up
127
128 def Hierarchical(scale, p):
129     phis = []
130     phips = []
131     start=0
132
133     for i in range(start, p+1):
134         new_phi = phi_func_h(scale, i)
135         new_phip = phip_func_h(scale,i)
136         phis.append(new_phi)
137         phips.append(new_phip)
138     return phis, phips
139
140 def linear(scale, p):
141     phis = []
142     phips = []
143     p = 1
144     for i in range(p+1):
145         new_phi = phi_func_l(scale, i)
146         new_phip = phip_func_l(scale,i)
147         phis.append(new_phi)
148         phips.append(new_phip)
149     return phis, phips
150
151 def quadratic(scale, p):
152     phis = []
153     phips = []
154     p = 1
155     for i in range(-1, p+1):
156         new_phi = phi_func_q(scale, i)
157         new_phip = phip_func_q(scale,i)
158         phis.append(new_phi)
159         phips.append(new_phip)
160     return phis, phips

```

Listing 2: Definition of the shape functions in 1-D

## D. DEFINITION OF GAUSSIAN INTEGRATE IN 1D

```

1 def G_integrate(u, N=3, scale=(0, 1)):
2     N = N
3     a = scale[0]
4     b = scale[1]
5     x, w = roots_legendre(N)

```



```

6
7 xp = x*(b-a)/2+(b+a)/2
8 wp = w*(b-a)/2
9
10 s = 0
11 for i in range(N):
12     s += wp[i]*u(xp[i])
13 return s

```

Listing 3: Definition of Gaussian integrate in 1D

## E. ENERGY CALCULATOR IN 1D

```

1 def cal_energy(U_array, phi_phip_array)
2 :
3     U_energy = 0
4     u_prime_list = []
5     scales = []
6     for i in range(len(phi_phip_array['phis'])):
7         u_prime = mul(U_array[i],
8             phi_phip_array['phips'][i])
9         u_prime_list.append(u_prime)
10        scales.append(u_prime.scale)
11    flat_scales = [item for sublist in
12        scales for item in sublist]
13    rounded_scales = [round(num, 5) for
14        num in flat_scales]
15    nodes = list(set(rounded_scales))
16    mesh = np.linspace(min(nodes), max(
17        nodes), len(nodes))
18    for i in range(len(mesh)-1):
19        scale = [mesh[i], mesh[i+1]]
20        U_energy+=G_integrate(mul(plus(
21            u_prime_list), plus(
22            u_prime_list)),N=9, scale=
23            scale)
24    return U_energy/2

```

Listing 4: Energy calculator in 1D

## F. A POSTERIORI ERROR ESTIMATE

```

1 def posterior_energy(energy_list_array,
2     DOFs_array):
3     if len(energy_list_array)<3:
4         raise AssertionError("The value
5             of energy should be greater
6             than three!")
7     elif len(energy_list_array)!= len(
8         DOFs_array):
9         raise AssertionError("The number
10             of energy values should be

```

```

equal to the number of DOFs!"
    )
6 def equation(U, U0, U1, U2, Q):
7     return ((U-U0)/(U-U1) / ((U-U1)/(
8         U-U2))**Q - 1)**2
9
10 i = 0
11 U_list = []
12 while i+3 <= len(energy_list_array):
13     U0, U1, U2 = energy_list_array[i:
14         i+3]
15     h0, h1, h2 = 1/np.sqrt(DOFs_array
16         [i:i+3])
17     # print(h0, h1, h2)
18     N0, N1, N2 = DOFs_array[i:i+3]
19     # Q = np.log((h0/h1))/np.log((h1/
20         h2))
21     Q = np.log((N1/N0))/np.log((N2/N1
22         ))
23     initial_guess = np.mean(
24         energy_list_array)
25     # Use minimize
26     U_solution = minimize(equation,
27         initial_guess, args=(U0, U1,
28         U2, Q)).x
29     U_list.append(U_solution)
30     i+=1
31 return np.mean(U_list)

```

Listing 5: A posteriori error estimate

## G. THE DEFINITION OF THE TRACTION-FREE HOLE IN AN INFINITE PLATE WITH DIFFERENT A/B RATIOS

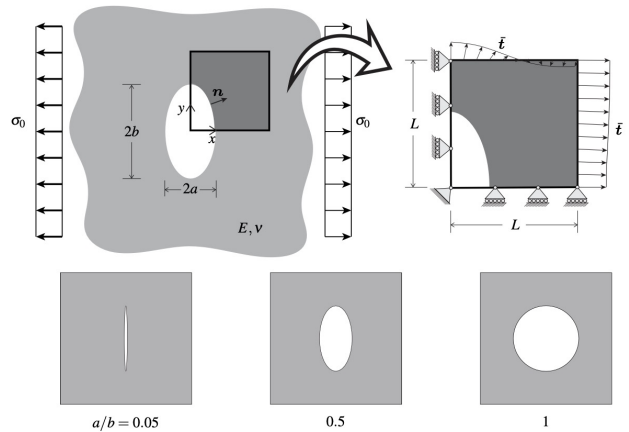


Figure 17: The definition of the traction-free hole in an infinite plate with different a/b ratios.

## H. FINITE ELEMENTS METHODS IN 2D

```

1  def FEM(a_b, mesh_size, mesh_shape,
2      GPN=2, show=False):
3      Load_x = 50 # N/mm
4      Load_y = 0 # N/mm
5      nodes_coord, element_nodes =
6          create_mesh(a_b, mesh_shape,
7              mesh_size)
8      nodes_list = Boundary(nodes_coord,
9          a_b)
10     element_list = []
11     if mesh_shape == 0:
12         element_nodes = element_nodes.
13             reshape(-1, 3)
14     elif mesh_shape == 1:
15         element_nodes = element_nodes.
16             reshape(-1, 4)
17
18     for ele_lst in element_nodes:
19         this_nodes = [
20             node for id in ele_lst for
21                 node in nodes_list if
22                     node.id == id]
23
24         try:
25             elem = Q4(this_nodes, GPN=GPN
26                 )
27         except:
28             elem = T3(this_nodes, GPN=GPN
29                 )
30
31         elem.a_b = a_b
32         element_list.append(elem)
33     DOFs = 2*len(nodes_list)
34     glo_K = np.zeros((DOFs, DOFs))
35     glo_F = np.zeros(DOFs)
36
37     for elem in element_list: # Assemble
38         Force vector
39         loc_F = elem.F
40         for i, node_i in enumerate(elem.
41             nodes):
42             global_dof = 2 * node_i.id
43             # print(loc_F[2*i])
44             if abs(node_i.xy[0]-40) < 1e
45                 -3:
46                 glo_F[global_dof] +=
47                     Load_x * loc_F[2*i]
48                 # glo_F[global_dof] +=
49                     Load_x * 1
50             glo_F[global_dof + 1] +=
51                 Load_y * loc_F[2*i+1]
52
53     for elem in element_list: # Assemble
54         Stiffness matrix
55         loc_K = elem.K

```

```

37     for i, node_i in enumerate(elem.
38         nodes):
39         for j, node_j in enumerate(
40             elem.nodes):
41             for dof_i in range(2):
42                 for dof_j in range(2)
43                     :
44                     global_dof_i = 2
45                         * node_i.id +
46                             dof_i
47                     global_dof_j = 2
48                         * node_j.id +
49                             dof_j
50
51                     glo_K[
52                         global_dof_i
53                     ][
54                         global_dof_j]
55                     += loc_K[2 *
56                         i + dof_i
57                     ][2*j + dof_j
58                     ]
59
60     for elem in element_list: # Boundary
61         condition
62
63     for i, node_i in enumerate(elem.
64         nodes):
65         for dof_i in range(2):
66             global_dof_i = 2 * node_i
67                 .id + dof_i
68
69             if node_i.BC[dof_i] == 1:
70
71                 glo_K[global_dof_i,
72                     :] = 0
73                 # glo_K[:,
74                     global_dof_i] = 0
75                 glo_K[global_dof_i,
76                     global_dof_i] = 1
77                 e15
78                 glo_F[global_dof_i] =
79                     0
80
81     U = np.linalg.solve(glo_K, glo_F)
82     for id in range(len(nodes_list)):
83         displacement = np.array([U[id*2],
84             U[id*2+1]])
85         nodes_list[id].value =
86             displacement
87
88     if show == True:
89         x_coords = [node.xy[0] for node
90             in nodes_list]
91         y_coords = [node.xy[1] for node
92             in nodes_list]

```

```

67     temperatures = [np.linalg.norm(
68         node.value) for node in
69         nodes_list]
70
71     tri = []
72     for c in element_nodes:
73         tri.append([c[0], c[1], c
74             [2]])
75     try:
76         tri.append([c[0], c[2], c
77             [3]])
78     except:
79         pass
80
81     plt.tricontourf(x_coords,
82         y_coords, temperatures,
83         triangles=tri, levels=15,
84         cmap=plt.cm.jet)
85     plt.colorbar(label='Displacement
86         in magnitude')
87     plt.title('Displacements
88         Distribution')
89
90     plt.show()
91     return U, nodes_coord, copy.deepcopy(
92         element_list)
93
94 def draw(elements_list, dir='xy', type=
95     'disp', show = True):
96     global_min = min([np.min([output(
97         test_element(xy[0], xy[1], type),
98         dir, type)
99         for xy in
100             test_element.
101             sample_points
102             (refine)])
103         for test_element
104             in
105             elements_list
106         ])
107     global_max = max([np.max([output(
108         test_element(xy[0], xy[1], type),
109         dir, type)
110         for xy in
111             test_element.
112             sample_points
113             (refine)])
114         for test_element
115             in
116             elements_list
117         ])
118
119     for test_element in elements_list:
120         test_inputs = test_element.
121             sample_points(refine)
122         test_mapping = test_element.
123             mapping(test_inputs)
124         test_output = [output(
125             test_element(xy[0], xy[1],
126                 type), dir, type)
127             for xy in
128                 test_inputs]
129         test_x, test_y, test_z =
130             grid_to_mat(test_mapping,
131                 test_output)
132         # plt.scatter(test_mapping[:, 0],
133             test_mapping[:, 1], s=1, c=
134             test_output)
135         plt.imshow(test_z, extent=(
136             test_mapping[:, 0].min(),
137             test_mapping[:, 0].max(),
138             test_mapping[:, 1].min(),
139             test_mapping[:, 1].max()),
140             origin='lower', aspect='auto',
141             interpolation='none', cmap='
142             jet', vmin=global_min, vmax=
143             global_max)
144         vertices = test_element.vertices
145         vertices = np.vstack([vertices,
146             vertices[0]])
147         vertices_x, vertices_y = zip(*
148             vertices)
149         plt.plot(vertices_x, vertices_y,
150             color='white',
151             linewidth=0.7)
152
153     plt.xlim(0, 40)
154     plt.ylim(0, 40)
155     # Display the color bar
156     cbar = plt.colorbar()
157     ticks = np.linspace(global_min,
158         global_max, num=5)
159     cbar.set_ticks(ticks)
160     if type == 'disp':
161         type_str = 'U'
162     elif type == 'strain':
163         type_str = '\\epsilon'
164     elif type == 'stress':
165         type_str = '\\sigma'
166     dir_str = "{s}" % dir
167     plt.title(rf"${type_str}_{dir_str}$")
168     if show:
169         plt.show()

```

Listing 6: Finite elements methods in 2D

# I. DEFINITIONS OF SHAPE FUNCTIONS FOR T3 AND Q4 ELEMENTS

```

1 class shape_fns:
2     def __init__(self, scale_x = [0,
3         1], scale_y = [0, 1], p=0):
4         self.scale_x = scale_x
5         self.scale_y = scale_y
6         self.p = p
7
8     def expression(self, xi, eta):
9         return 1-xi-eta
10
11     def __call__(self, x=0, y=0):
12         return self.expression(x, y)
13
14 class T3_phi(shape_fns):
15     def expression(self, xi, eta):
16         if self.p == 0:
17             return xi
18         elif self.p == 1:
19             return eta
20         elif self.p == 2:
21             return 1-xi-eta
22         else:
23             raise ValueError("p should be 0, 1 or 2 in T3 element shape functions, not {}".format(self.p))
24
25
26 class T3_phipx(shape_fns):
27     def expression(self, xi=0, eta=0):
28         if self.p == 0:
29             return 1
30         elif self.p == 1:
31             return 0
32         elif self.p == 2:
33             return -1
34         else:
35             raise ValueError("p should be 0, 1 or 2 in T3 element shape functions, not {}".format(self.p))
36
37
38 class T3_phipy(shape_fns):
39     def expression(self, xi=0, eta=0):
40         if self.p == 0:
41             return 0
42         elif self.p == 1:
43             return 1

```

```

44     elif self.p == 2:
45         return -1
46     else:
47         raise ValueError("p should be 0, 1 or 2 in T3 element shape functions, not {}".format(self.p))
48
49
50 class Q4_phi(shape_fns):
51     def expression(self, xi=0, eta=0):
52         if self.p == 0:
53             return (xi-1)*(eta-1)/4
54         elif self.p == 1:
55             return (1 + xi) * (1 - eta) / 4
56         elif self.p == 2:
57             return (1 + xi) * (1 + eta) / 4
58         elif self.p == 3:
59             return (1 - xi) * (1 + eta) / 4
60         else:
61             raise ValueError("p should be 0, 1, 2 or 3 in Q4 element shape functions, not {}".format(self.p))
62
63 class Q4_phipx(shape_fns):
64     def expression(self, xi=0, eta=0):
65         if self.p == 0:
66             return (eta - 1)/4
67         elif self.p == 1:
68             return (1 - eta)/4
69         elif self.p == 2:
70             return (1 + eta)/4
71         elif self.p == 3:
72             return -(1 + eta)/4
73         else:
74             raise ValueError("p should be 0, 1, 2 or 3 in Q4 element shape functions, not {}".format(self.p))
75
76
77 class Q4_phipy(shape_fns):
78     def expression(self, xi=0, eta=0):
79         if self.p == 0:
80             return (xi - 1)/4
81         elif self.p == 1:
82             return -(xi + 1)/4
83         elif self.p == 2:
84             return (1 + xi)/4

```

```

85     elif self.p == 3:
86         return (1 - xi)/4
87     else:
88         raise ValueError("p should be 0, 1, 2 or 3 in Q4 element shape functions, not {}".format(self.p))

```

Listing 7: Definitions of shape functions for T3 and Q4 elements

## J. GAUSSIAN POINTS IN 2D

```

1  def Gauss_points(element, order):
2      if element.shape == 'quad':
3          xi, wi = np.polynomial.legendre.leggauss(order)
4          points = [(x, y) for x in xi for y in xi]
5          weights = [wx * wy for wx in wi for wy in wi]
6
7      elif element.shape == 'triangle':
8          NGP_data = {
9              1: {
10                 'points': np.array([(1/3, 1/3)]),
11                 'weights': np.array([1/2])
12             },
13             3: {
14                 'points': np.array([(1/6, 1/6), (2/3, 1/6), (1/6, 2/3)]),
15                 'weights': np.array([1/6, 1/6, 1/6])
16             },
17             4: {
18                 'points': np.array([(1/3, 1/3), (0.6, 0.2), (0.2, 0.6), (0.2, 0.2)]),
19                 'weights': np.array([-27/96, 25/96, 25/96, 25/96])
20             }
21         }
22         if order == 2:
23             order = 3
24         points, weights = NGP_data[order]['points'], NGP_data[order]['weights']
25     else:

```

```

26         raise ValueError("Shape not supported")
27
28     return points, weights

```

Listing 8: Gaussian points in 2D

## K. VON MISES STRESS

The von Mises criterion, also known as the von Mises yield criterion or von Mises flow criterion, is a widely-used model for predicting the yielding behavior of ductile materials. Mathematically, it can be expressed as:

$$\sigma_{vM} = \sqrt{\sigma_x^2 - \sigma_x \sigma_y + \sigma_y^2 + 3\tau_{xy}^2}$$

Here,  $\sigma_x$  and  $\sigma_y$  are the normal stresses in the  $x$  and  $y$  directions, respectively, and  $\tau_{xy}$  is the shear stress in the  $xy$  plane.

The code for Von Mises stress is as follows:

```

1  def Von_Mise(sigma_x, sigma_y, tau_xy):
2      result = np.sqrt(sigma_x**2 - sigma_x*sigma_y + sigma_y**2 + 3*tau_xy**2)
3      return result

```

Listing 9: Von Mises stress

## L. STRAIN ENERGY IN 2D

```

1  def cal_energy(elements_list, GPN = 2):
2      E = 200e3
3      nu = 0.3
4      D = E / (1 - nu**2) * np.array([
5          [1, nu, 0],
6          [nu, 1, 0],
7          [0, 0, (1-nu)/2]
8      ])
9      energy = 0
10     for elem in elements_list:
11         elem_energy = 0
12         points, Ws = Gauss_points(elem, GPN)
13         loop = 0

```

```

14     scale = 4 if elem.shape=="
        triangle" else 1
15     for g in range(len(Ws)):
16         xy = points[g]
17         W = Ws[g]
18         strain_list = elem(xy[0], xy
            [1], 'strain')
19         dN = elem.gradshape(xy[0], xy
            [1])
20         # J = jacobian(self.vertices,
            dN)
21         J = np.dot(dN , elem.vertices
            )
22         J_det = np.linalg.det(J)
23         B = elem.B_matrix(J, dN)
24         this_energy = 0.5 * W *
            strain_list.T @ D @
            strain_list * J_det **
            scale
25         elem_energy += this_energy
26         loop+=1
27     energy+=elem_energy
28     return energy[0][0]

```

Listing 10: Strain energy in 2D

where  $m$  is the number of nodes shared by adjacent elements.

## M. IMPLEMENTATION OF SUPERCONVERGENT PATCH RECOVERY

The Superconvergent Patch Recovery (SPR) method refines the stress distribution by extrapolating and averaging the stresses. The key steps can be mathematically represented as follows:

1. **Stress at Gauss Points:** The stress  $\sigma_{\text{Gauss}}$  is initially calculated at the Gauss points of each finite element.

$$\sigma_{\text{Gauss}} = C : \varepsilon_{\text{Gauss}}$$

where  $C$  is the material stiffness matrix, and  $\varepsilon_{\text{Gauss}}$  is the strain at the Gauss points.

2. **Extrapolation to Nodes:** The stress is then extrapolated to the nodes  $N$  of each element using shape functions  $N_i$ .

$$\sigma_{\text{Node}} = \sum_{i=1}^n N_i \sigma_{\text{Gauss},i}$$

where  $n$  is the number of Gauss points.

3. **Averaging at Nodes:** Finally, the nodal stresses are averaged across adjacent elements to get a smoother stress distribution  $\sigma_{\text{avg}}$ .

$$\sigma_{\text{avg}} = \frac{1}{m} \sum_{j=1}^m \sigma_{\text{Node},j}$$