Final Project for Advanced FEM (ME46050)

Xusen Qin

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

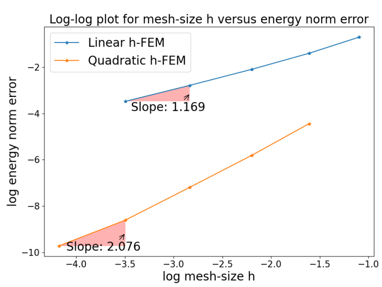
I. Introduction

This 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.

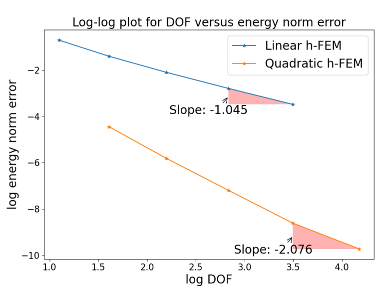
II. PROBLEM 1

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 $h = \frac{1}{\mathrm{DOF}^d}$, 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 6.0×10^{-5}. These computed rates align closely with the theoretical expectations.

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

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.

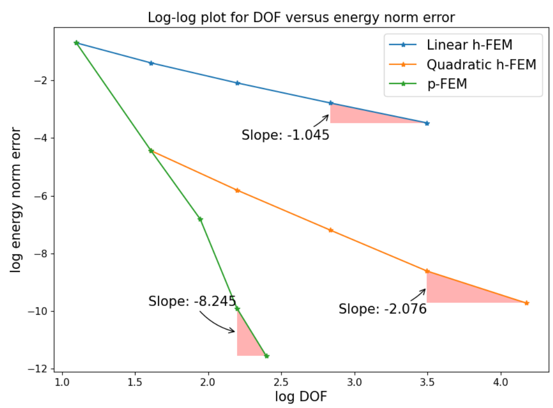


Figure 2: *log-log plot of the error versus DOF in h-version and p-version FEM.*

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 $\epsilon(\Omega)$:

$$\left\| u - u^h \right\|_{\epsilon(\Omega)} \leqslant C_1 h^{\beta_h} \|u\|_{\epsilon(\Omega)} \tag{2}$$

We went:

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

where U is the exact strain energy.

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

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

Now we obtain the error of the strain energy:

$$U_e = U - U^h. \tag{5}$$

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

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

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.
- β_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 obtained by Eq.**??**:

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

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:

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

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

$$h \cong \frac{1}{N^{1/dimensionality}} \tag{9}$$

The expression of $\mathbf{Q}$ is given by:

$$Q = \frac{\log\left(N_1/N_0\right)}{\log\left(N_2/N_1\right)} \tag{10}$$

The term $\mathbf{Q}$ 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.

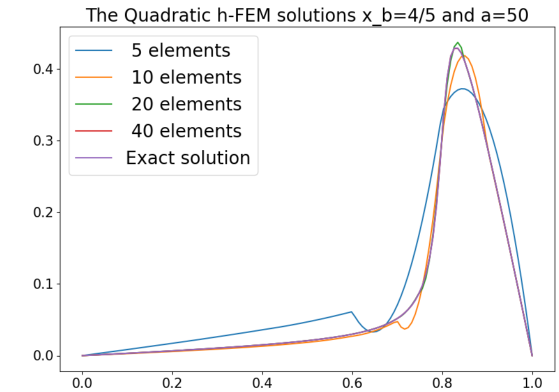


Figure 3: *The Quadratic h-FEM solutions x_b=4/5 and a=50 with different element numbers.*

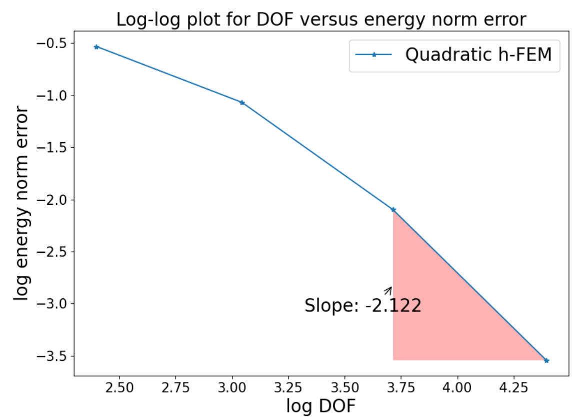


Figure 4: *Log-log plot for DOF versus energy norm error.*

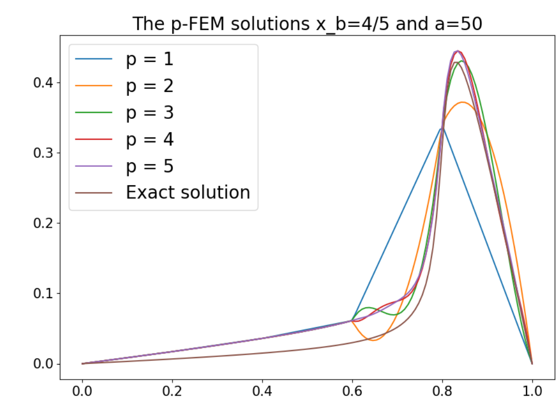


Figure 5: *The p-FEM solutions x_b=4/5 and a=50 with different element numbers.*

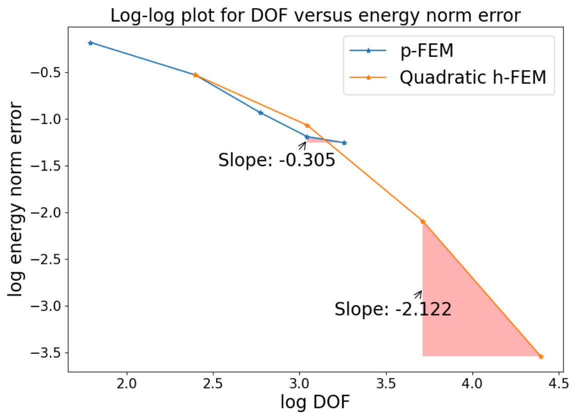


Figure 6: *Log-log plot for DOF versus energy norm error in p-FEM and h-FEM.*

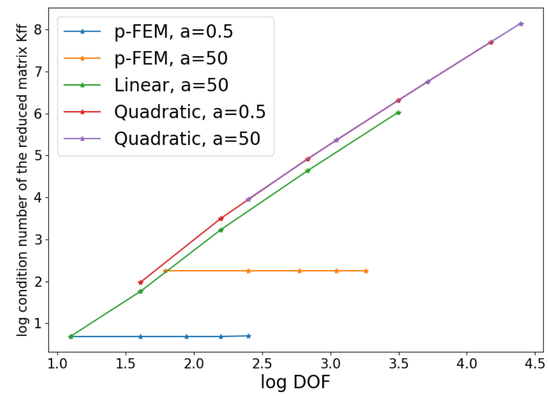


Figure 7: *Log-log plot for the condition number of the reduced matrix K_{ff} versus energy norm error.*

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.

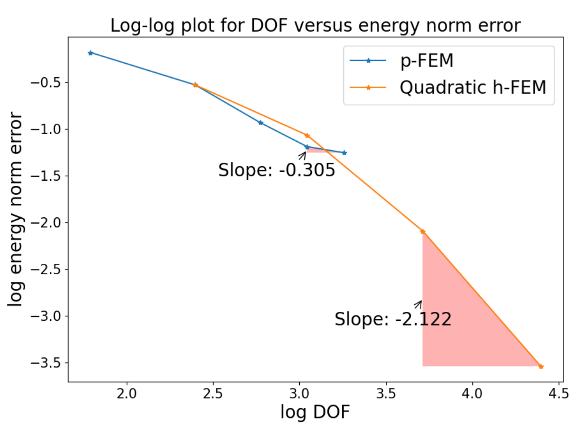
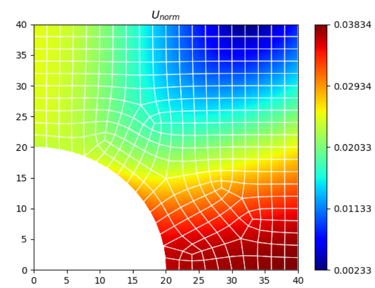


Figure 8: *Log-log plot for DOF versus energy norm error in p-FEM and h-FEM when Gaussian points are selected as 6.*

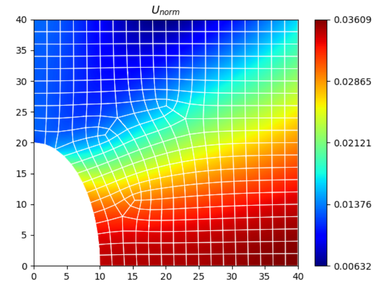
III. PROBLEM 2

i. Questions 1

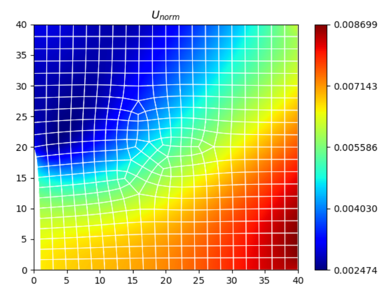
The definition of the traction-free hole in an infinite plate with different a/b ratios is shown in Appendix.**??** of the report. The code for finite element method, shape functions as well as the Gaussian integration method in Appendix.**??**, **??**, and **??**. For the mesh size $h/L = 0.05$ and 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:

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

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	$U_{x,\text{cal}}$	$U_{x,\text{FEM}}$	$U_{y,\text{cal}}$	$U_{y,\text{FEM}}$
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.**??**

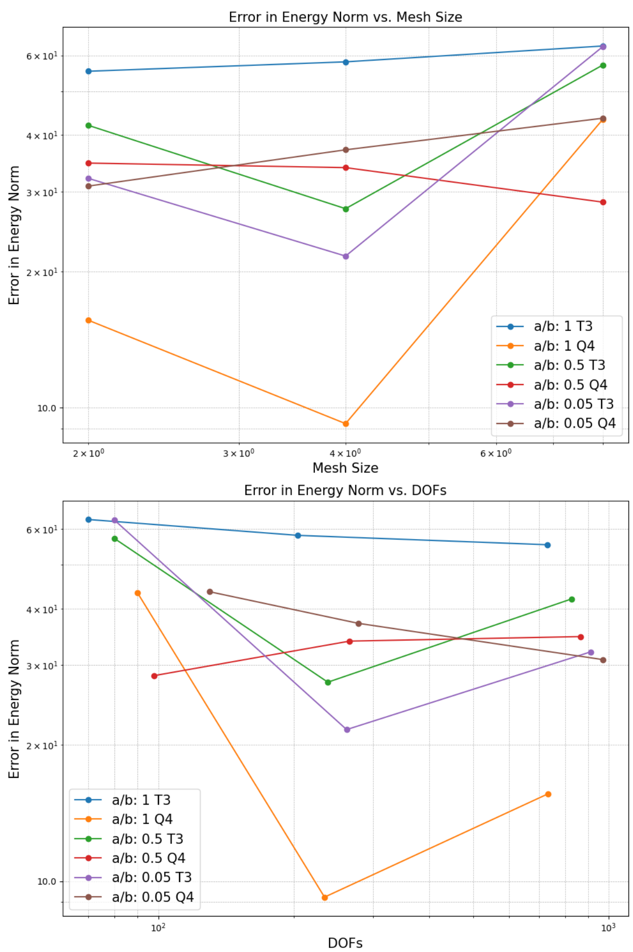


Figure 10: *Log-log plot of energy norm error versus DOFs.*

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 ratie 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*

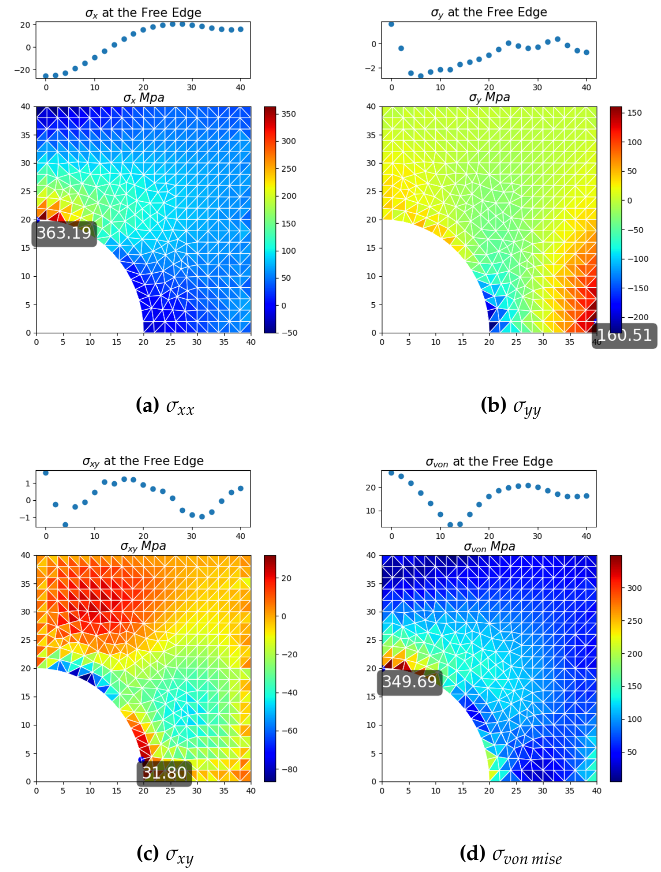
a/b ratio	Error in σ_x (%)	Error in σ_y (%)
1	57.33	805.00
0.5	40.39	1130.77
0.05	31.37	1820.00

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

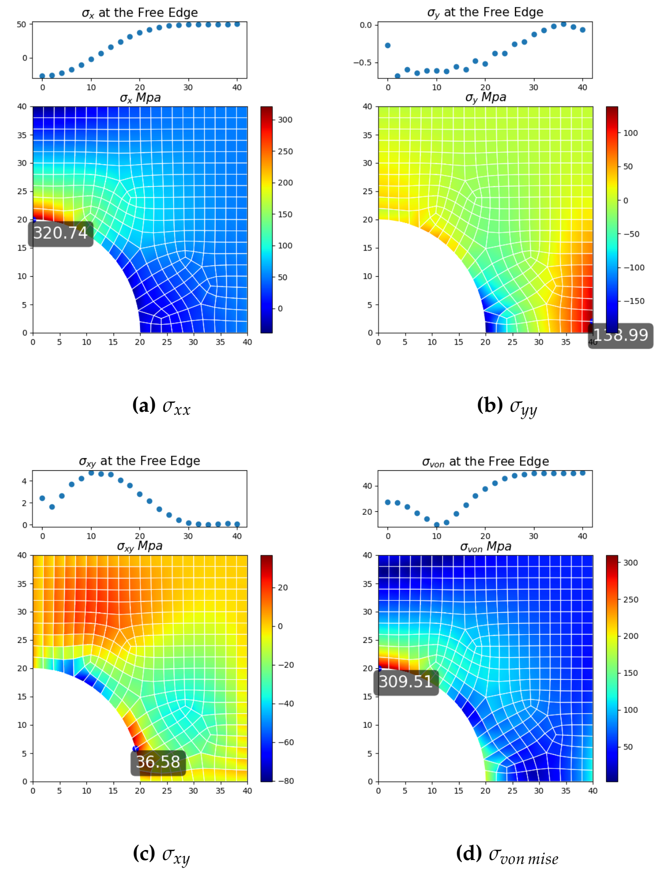
The Table.**??** presents the stress concentration factors (SCF) for T3 and Q4 elements at different a/b ratios. It is evident that the experimentally observed SCFs are significantly different from the theoretical predictions (given by $K_c = 1 + 2\frac{b}{a}$).

- 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.



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy} **(d)** $\sigma_{von\,mise}$

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



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy} **(d)** $\sigma_{von\,mise}$

Figure 12: *Stress fields for a/b=1 with Q4 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

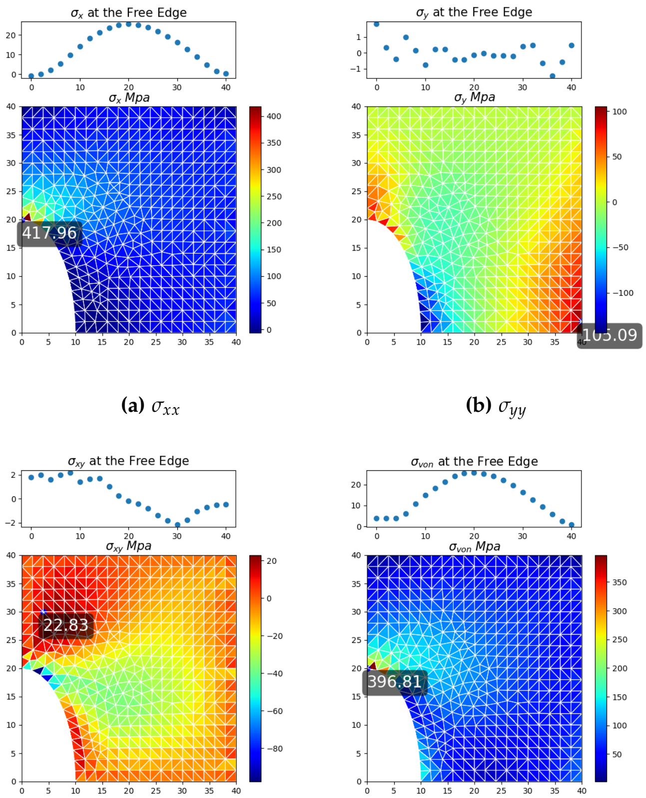
vii. Question 7

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 selected 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

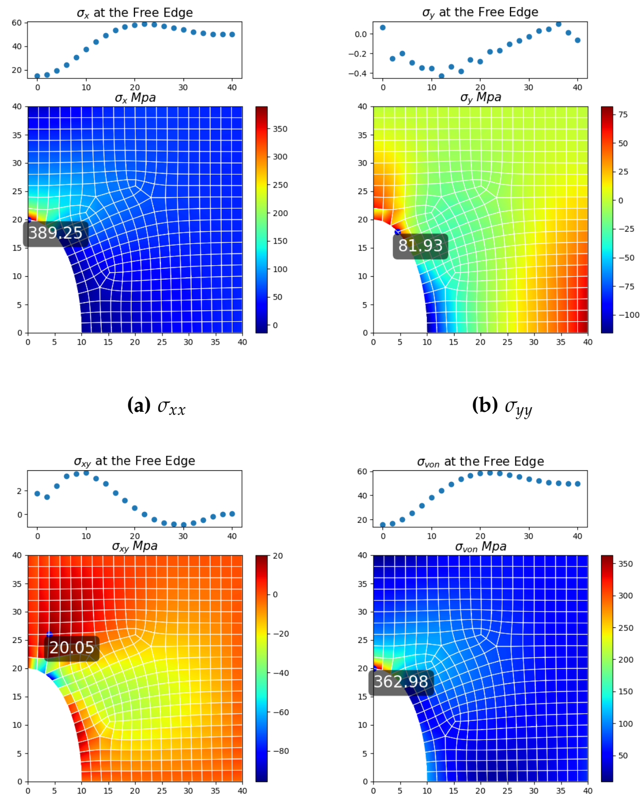
Table 9: *Allowable stress values for different a/b ratios.*

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.



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy} **(d)** $\sigma_{von\ mise}$

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



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy} **(d)** $\sigma_{von\ mise}$

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.

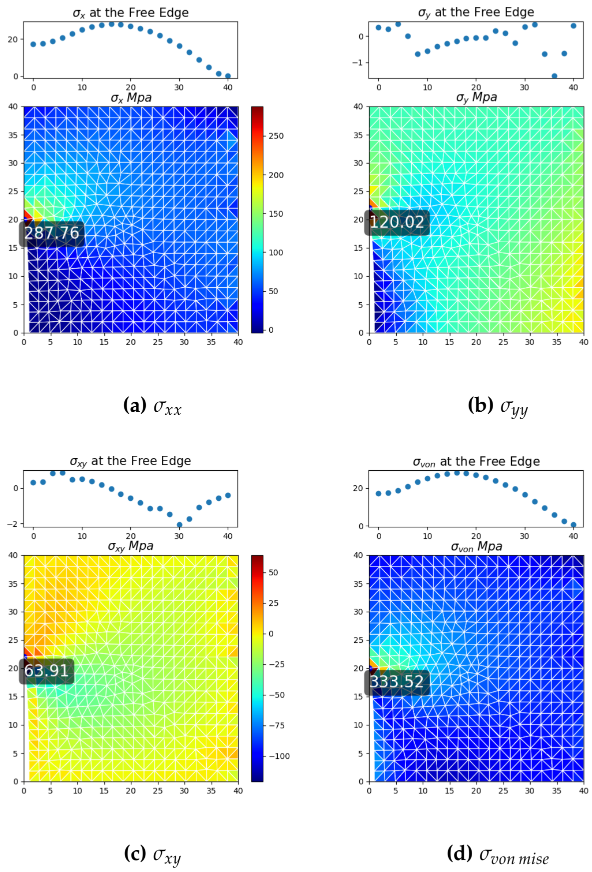
Interestingly, the lowest strain energy was observed for $a/b = 0.05$. This can be attributed to the shape of the hole, which is a very narrow ellipse at this ratio. In essence, the structure behaves almost like a solid plate with a crack, rather than a plate with a hole. The narrowness of the ellipse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

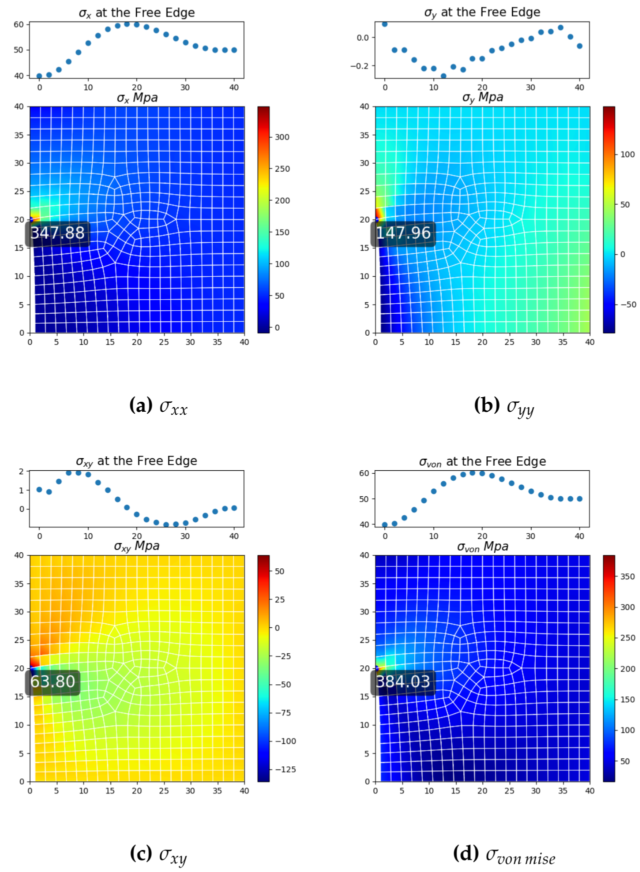
Stress and Displacement Errors

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.



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy} **(d)** $\sigma_{von\,mise}$

Figure 15: *Stress fields for a/b=0.05 with T3 mesh.*



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy} **(d)** $\sigma_{von\,mise}$

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. Defeinition of the 1-D Possion equation

The 1-D Poisson equation is given as:

$$-\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = \frac{2\left(a + a^3 b(b - x + 1)\right)}{\left(a^2 b^2 + 1\right)^2} \tag{12}$$

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

$$u(x) = (1 - x)\left(\arctan(ab) + \arctan\left(ax_b\right)\right) \tag{13}$$

B. Finite elements methods in 1-D main codes

```
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):
    N=6
    mesh = np.linspace(domain[0],
        domain[1], num_elems+1)
    ori_phi_phip = {'phis': [], '
        phips': []}
    for elem in range(num_elems):
        scale = [mesh[elem], mesh[
            elem+1]]
        phis, phips = shape_class(
            scale, p)
        ori_phi_phip['phis'].append(
            phis)
        ori_phi_phip['phips'].append(
            phips)


    linear_phi_phip = {'phis': [], '
        phips': []}  # Linear
    for elem in range(num_elems):
        linear_phis = []
        linear_phips = []
        for idx in range(len(
            ori_phi_phip['phis'][elem
            ])):
            if ori_phi_phip['phis'][
                elem][idx].p < 2:
                phi = ori_phi_phip['
                    phis'][elem][idx]
                phip = ori_phi_phip['
                    phips'][elem][idx
                    ]
                linear_phi_phip['phis
                    '].append(phi)
                linear_phi_phip['
                    phips'].append(
                    phip)
                linear_phis.append(
                    phi)
                linear_phips.append(
                    phip)
        linear_K_sub = np.zeros((len(
            linear_phips), len(
            linear_phips)))
        for indx, x in np.ndenumerate
            (linear_K_sub):
            linear_K_sub[indx] =
                G_integrate(
                mul(linear_phips[indx
                    [0]],
                    linear_phips[indx
                    [-1]]), N=6,
                    scale=
                    linear_phips[indx
                    [0]].scale)
            if abs(linear_K_sub[indx
                ]) < 1e-10:
                linear_K_sub[indx] =
                    0
        linear_F_sub = np.zeros(len(
            linear_K_sub))
        for indx in range(len(
            linear_F_sub)):
            linear_F_sub[indx] =
                G_integrate(
                mul(rhs_func,
                    linear_phis[indx
                    ]), N=N, scale=
                    linear_phis[indx
                    ].scale)
        if elem == 0:
            K = linear_K_sub
            F = linear_F_sub
        else:
            K = assemble(K,
                linear_K_sub)
            F = assemble(F,
                linear_F_sub)

    linear_num = len(F)

    nonlinear_phi_phip = {'phis': [],
        'phips': []}
    for order in range(2, p+1):  #
        Non Linear
        for elem in range(num_elems):
            for idx in range(len(
                ori_phi_phip['phis'][
```

```
elem])):
                if (ori_phi_phip['phis'][elem][idx].p == order) or (ori_phi_phip['phips'][elem][idx].p == order):
                    nonlinear_phi = ori_phi_phip['phis'][elem][idx]
                    nonlinear_phip = ori_phi_phip['phips'][elem][idx]
                    nonlinear_phi_phip['phis'].append(nonlinear_phi)
                    nonlinear_phi_phip['phips'].append(nonlinear_phip)
                    nonlinear_K_sub = np.zeros((2, 2))

                    nonlinear_K_sub[-1, -1] = G_integrate(mul(nonlinear_phip, nonlinear_phip),N=N, scale=nonlinear_phip.scale)
                    nonlinear_F_sub = np.zeros(2)
                    nonlinear_F_sub[-1] = G_integrate(mul(rhs_func, nonlinear_phi), N=N, scale=nonlinear_phi.scale)

                    K = assemble(K, nonlinear_K_sub)
                    F = assemble(F, nonlinear_F_sub)
                else:
                    pass

    # Applying boundary condition

    K[0, 1:] = 0.0
    K[linear_num-1, :linear_num-1] = 0.0
    F[0] = BCs[0]* K[0, 0]
    F[linear_num-1] = BCs[-1] * K[linear_num-1, linear_num-1]

    U = -la.solve(K, F)
    phi_phip = {'phis': [], 'phips': []}
    phi_phip['phis'] = joint_funcs(linear_phi_phip['phis']) + nonlinear_phi_phip['phis']
    phi_phip['phips'] = joint_funcs(linear_phi_phip['phips']) + nonlinear_phi_phip['phips']
    u_list = []
    for i in range(len(phi_phip['phis'])):
        u_list.append(mul(U[i], phi_phip['phis'][i]))
    uh = plus(u_list)
    if verbose == True:
        print(f"Shape class: {shape_class.__name__}, Number of elements: {num_elems}, Polynomial order:{p},  Domain: {domain}, Boundary conditions: {BCs}")
        x_data = np.linspace(domain[0], domain[1], 101)
        plt.plot(x_data, exact_func(x_data), label='Analytical solution')
        plt.plot(x_data, uh(x_data), label='FEM solution {} elements'.format(num_elems))
        for i in range(len(phi_phip['phis'])):
            func = phi_phip['phis'][i]
            plt.plot(x_data, U[i]*func(x_data))
        plt.legend()
        plt.show()
```

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

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

C. Defeinition of the shape functions in 1D

```
def Legendre(x=np.linspace(-1, 1, 100), p=5):

    if p == 0:
        return 1
    elif p == 1:
        return x

    else:
        return ((2*p-1)*x*Legendre(x, p-1)+(1-p)*Legendre(x, p-2))/p

class shape_function:
        def __init__(self, scale=[-1, 1]):
            self.scale = scale
            self.x_l = scale[0]
            self.x_r = scale[1]
            self.range = [-1, 1]

        def expression(self, x):
            return 1 - (x - self.x_l) / (self.x_r - self.x_l)

        def mapping(self, x):
            scale = self.scale
            range = self.range
            x_normalized = (x - scale[0]) / (scale[1] - scale[0])
            return range[0] + x_normalized * (range[1] - range[0])

        def __call__(self, x):
            x = np.asarray(x)  # convert x to a numpy array if it's not already
            expression_vectorized = np.vectorize(self.expression, otypes=['d'])
            return np.where((self.scale[0] <= x) & (x <= self.scale[-1]), expression_vectorized(x), 0)

class phi_func_l(shape_function):
    def __init__(self, scale, p):
        super().__init__(scale)
        self.p = p
        self.range = [0, 1]
    def expression(self, x):
        if self.p == 0:
            phi = 1-self.mapping(x)
        elif self.p == 1:
            phi = self.mapping(x)
        else:
            raise AssertionError("p should be 0 or 1 in linear shape function, not{}".format(self.p))
        return phi

class phip_func_l(shape_function):
    def __init__(self, scale, p):
        super().__init__(scale)
        self.range = [0, 1]
        self.p = p
    def expression(self, x):
        scale_up = 1/(self.scale[1]-self.scale[0])

        if self.p == 0:
            phip =  np.zeros_like(self.mapping(x))-1
        elif self.p == 1:
            phip = np.zeros_like(self.mapping(x))+1
        else:
            raise AssertionError("p should be 0 or 1 in linear shape function, not{}".format(self.p))
        return phip*scale_up

class phi_func_q(shape_function):
    def __init__(self, scale, p):
        super().__init__(scale)
        self.range = [0, 1]
        self.p = p
    def expression(self, x):
        xx = self.mapping(x)
        if self.p == -1:
            phi = (xx-1)*(xx-0.5)*2
        elif self.p == 0:
            phi = -xx*(xx-1)*4
        elif self.p ==1:
            phi = xx*(xx-0.5)*2
        else:
```

```
            raise AssertionError("p should be -1, 0 or 1 in quadratic shape function, not{}".format(self.p))
        return phi

class phip_func_q(shape_function):
    def __init__(self, scale, p):
        super().__init__(scale)
        self.range = [0, 1]
        self.p = p
    def expression(self, x):
        scale_up = 1/(self.scale[1]-self.scale[0])
        xx = self.mapping(x)
        if self.p == -1:
            phip = 4*xx - 3.0
        elif self.p == 0:
            phip = 4-8*xx
        elif self.p ==1:
            phip = 4*xx - 1.0
        else:
            raise AssertionError("p should be -1, 0 or 1 in quadratic shape function, not{}".format(self.p))
        return phip*scale_up

class phi_func_h(shape_function):
    def __init__(self, scale, p):
        super().__init__(scale)
        self.p = p
    def expression(self, x):
        scale = self.scale
        i =self.p
        if i == 0:
            phi = (1-self.mapping(x))/2
        elif i == 1:
            phi = (1+self.mapping(x))/2
        else:
            phi = 1/np.sqrt(4*i-2)*(Legendre(self.mapping(x), i)-Legendre(self.mapping(x), i-2))
        return phi

class phip_func_h(shape_function):
    def __init__(self, scale, p):
        super().__init__(scale)
        self.p = p
    def expression(self, x):
        scale_up = 2/(self.scale[1]-self.scale[0])
        i =self.p

        if i == 0:
            phip =  np.zeros_like(self.mapping(x))-0.5
        elif i == 1:
            phip = np.zeros_like(self.mapping(x))+0.5
        else:
            phip = np.sqrt(i-1/2)*(Legendre(self.mapping(x), i-1))
        return phip*scale_up

def Hierarchical(scale, p):
    phis = []
    phips = []
    start=0

    for i in range(start, p+1):
        new_phi = phi_func_h(scale, i)
        new_phip = phip_func_h(scale,i)
        phis.append(new_phi)
        phips.append(new_phip)
    return phis, phips

def linear(scale, p):
    phis = []
    phips = []
    p = 1
    for i in range(p+1):
        new_phi = phi_func_l(scale, i)
        new_phip = phip_func_l(scale,i)
        phis.append(new_phi)
        phips.append(new_phip)
    return phis, phips

def quadratic(scale, p):
    phis = []
    phips = []
    p = 1
    for i in range(-1, p+1):
        new_phi = phi_func_q(scale, i)
        new_phip = phip_func_q(scale,i)
        phis.append(new_phi)
        phips.append(new_phip)
    return phis, phips
```

Listing 2: *Defeinition of the shape functions in 1-D*

D. Defeinition of Gaussian integrate in 1D

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

```
    xp = x*(b-a)/2+(b+a)/2
    wp = w*(b-a)/2

    s = 0
    for i in range(N):
        s += wp[i]*u(xp[i])
    return s
```

Listing 3: *Defeinition of Gaussian integrate in 1D*

E. Energy calculator in 1D

```
def cal_energy(U_array, phi_phip_array):
    U_energy = 0
    u_prime_list = []
    scales = []
    for i in range(len(phi_phip_array['phis'])):
        u_prime = mul(U_array[i], phi_phip_array['phips'][i])
        u_prime_list.append(u_prime)
        scales.append(u_prime.scale)
    flat_scales = [item for sublist in scales for item in sublist]
    rounded_scales = [round(num, 5) for num in flat_scales]
    nodes = list(set(rounded_scales))
    mesh = np.linspace(min(nodes), max(nodes), len(nodes))
    for i in range(len(mesh)-1):
        scale = [mesh[i], mesh[i+1]]
        U_energy+=G_integrate(mul(plus(u_prime_list), plus(u_prime_list)),N=9, scale=scale)
    return U_energy/2
```

Listing 4: *Energy calculator in 1D*

F. A posteriori error estimate

```
def posterior_energy(energy_list_array, DOFs_array):
    if len(energy_list_array)<3:
        raise AssertionError("The␣value␣of␣energy␣should␣be␣greater␣than␣three!")
    elif len(energy_list_array)!= len(DOFs_array):
        raise AssertionError("The␣number␣of␣energy␣values␣should␣be␣equal␣to␣the␣number␣of␣DOFs!")
    def equation(U, U0, U1, U2, Q):
        return ((U-U0)/(U-U1) / ((U-U1)/(U-U2))**Q - 1)**2

    i = 0
    U_list = []
    while i+3 <= len(energy_list_array):
        U0, U1, U2 = energy_list_array[i:i+3]
        h0, h1, h2 = 1/np.sqrt(DOFs_array[i:i+3])
        # print(h0, h1, h2)
        N0, N1, N2 = DOFs_array[i:i+3]
        # Q = np.log((h0/h1))/np.log((h1/h2))
        Q = np.log((N1/N0))/np.log((N2/N1))
        initial_guess = np.mean(energy_list_array)
        # Use minimize
        U_solution = minimize(equation, initial_guess, args=(U0, U1, U2, Q)).x
        U_list.append(U_solution )
        i+=1
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

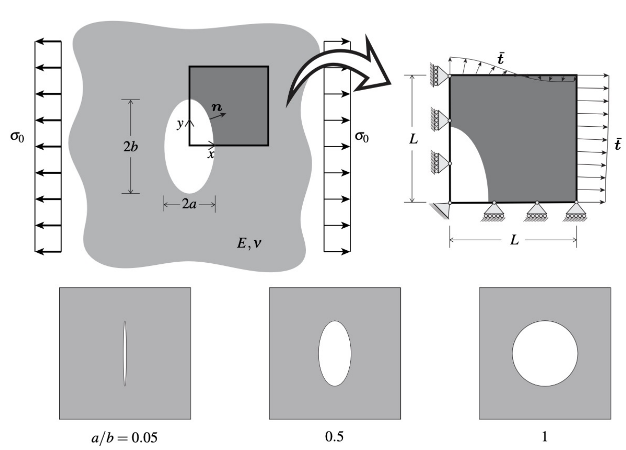


Figure 17: *The definition of the traction-free hole in an infinite plate with different a/b ratios.*

H. Finite elements methods in 2D

```python
def FEM(a_b, mesh_size, mesh_shape, GPN=2, show=False):
    Load_x = 50  # N/mm
    Load_y = 0  # N/mm
    nodes_coord, element_nodes = create_mesh(a_b, mesh_shape, mesh_size)
    nodes_list = Boundary(nodes_coord, a_b)
    element_list = []
    if mesh_shape == 0:
        element_nodes = element_nodes.reshape(-1, 3)
    elif mesh_shape == 1:
        element_nodes = element_nodes.reshape(-1, 4)

    for ele_lst in element_nodes:
        this_nodes = [
            node for id in ele_lst for node in nodes_list if node.id == id]
        try:
            elem = Q4(this_nodes, GPN=GPN)
        except:
            elem = T3(this_nodes, GPN=GPN)
        elem.a_b = a_b
        element_list.append(elem)
    DOFs = 2*len(nodes_list)
    glo_K = np.zeros((DOFs, DOFs))
    glo_F = np.zeros(DOFs)

    for elem in element_list:  # Assemble Force vector
        loc_F = elem.F
        for i, node_i in enumerate(elem.nodes):
            global_dof = 2 * node_i.id
            # print(loc_F[2*i])
            if abs(node_i.xy[0]-40) < 1e-3:
                glo_F[global_dof] += Load_x * loc_F[2*i]
                # glo_F[global_dof] += Load_x * 1
                glo_F[global_dof + 1] += Load_y * loc_F[2*i+1]

    for elem in element_list:  # Assemble Stiffness matrix
        loc_K = elem.K
        for i, node_i in enumerate(elem.nodes):
            for j, node_j in enumerate(elem.nodes):
                for dof_i in range(2):
                    for dof_j in range(2):
                        global_dof_i = 2 * node_i.id + dof_i
                        global_dof_j = 2 * node_j.id + dof_j

                        glo_K[global_dof_i][global_dof_j] += loc_K[2 * i + dof_i][2*j + dof_j]
    for elem in element_list:  # Boundary condition

        for i, node_i in enumerate(elem.nodes):
            for dof_i in range(2):
                global_dof_i = 2 * node_i.id + dof_i

                if node_i.BC[dof_i] == 1:

                    glo_K[global_dof_i, :] = 0
                    # glo_K[:, global_dof_i] = 0
                    glo_K[global_dof_i, global_dof_i] = 1e15
                    glo_F[global_dof_i] = 0

    U = np.linalg.solve(glo_K, glo_F)
    for id in range(len(nodes_list)):
        displacement = np.array([U[id*2], U[id*2+1]])
        nodes_list[id].value = displacement

    if show == True:
        x_coords = [node.xy[0] for node in nodes_list]
        y_coords = [node.xy[1] for node in nodes_list]
```

```
        temperatures = [np.linalg.norm(node.value) for node in nodes_list]

        tri = []
        for c in element_nodes:
            tri.append([c[0], c[1], c[2]])
            try:
                tri.append([c[0], c[2], c[3]])
            except:
                pass

        plt.tricontourf(x_coords, y_coords, temperatures, triangles=tri, levels=15, cmap=plt.cm.jet)
        plt.colorbar(label='Displacement in magnitude')
        plt.title('Displacements Distribution')

        plt.show()
    return U, nodes_coord, copy.deepcopy(element_list)


def draw(elements_list, dir='xy', type='disp', show = True):
    global_min = min([np.min([output(test_element(xy[0], xy[1], type), dir, type)
                              for xy in test_element.sample_points(refine)]) for test_element in elements_list])

    global_max = max([np.max([output(test_element(xy[0], xy[1], type), dir, type)
                              for xy in test_element.sample_points(refine)]) for test_element in elements_list])

    for test_element in elements_list:
        test_inputs = test_element.sample_points(refine)
        test_mapping = test_element.mapping(test_inputs)
        test_output = [output(test_element(xy[0], xy[1], type), dir, type)
                                    for xy in test_inputs]
        test_x, test_y, test_z = grid_to_mat(test_mapping, test_output)
        # plt.scatter(test_mapping[:, 0], test_mapping[:, 1], s=1, c=test_output)
        plt.imshow(test_z, extent=(test_mapping[:, 0].min(), test_mapping[:, 0].max(), test_mapping[:, 1].min(), test_mapping[:, 1].max()), origin='lower', aspect='auto', interpolation='none',cmap='jet', vmin=global_min, vmax=global_max)
        vertices = test_element.vertices
        vertices = np.vstack([vertices, vertices[0]])
        vertices_x, vertices_y = zip(*vertices)
        plt.plot(vertices_x, vertices_y, color='white',
                 linewidth=0.7)

    plt.xlim(0, 40)
    plt.ylim(0, 40)
    # Display the color bar
    cbar = plt.colorbar()
    ticks = np.linspace(global_min, global_max, num=5)
    cbar.set_ticks(ticks)
    if type == 'disp':
        type_str = 'U'
    elif type == 'strain':
        type_str = '\\epsilon'
    elif type == 'stress':
        type_str = '\\sigma'
    dir_str = "{ %s }" % dir
    plt.title(rf"${type_str}_{dir_str}$")
    if show:
    plt.show()
```

Listing 6: *Finite elements methods in 2D*

I. Defeinitions of shape functions for T3 and Q4 elements

```
class shape_fns:
    def __init__(self, scale_x = [0, 1], scale_y = [0, 1], p=0):
        self.scale_x = scale_x
        self.scale_y = scale_y
        self.p = p

    def expression(self, xi, eta):
        return 1-xi-eta

    def __call__(self, x=0, y=0):

        return  self.expression(x, y)


class T3_phi(shape_fns):
    def expression(self, xi, eta):
        if self.p == 0:
             return xi
        elif self.p == 1:
            return eta
        elif self.p == 2:
            return 1-xi-eta
        else:
            raise ValueError("p␣should␣be␣0,␣1␣or␣2␣in␣T3␣element␣shape␣functions,␣not␣{}".format(self.p))


class T3_phipx(shape_fns):
    def expression(self, xi=0, eta=0):
        if self.p == 0:
             return 1
        elif self.p == 1:
            return 0
        elif self.p == 2:
            return -1
        else:
            raise ValueError("p␣should␣be␣0,␣1␣or␣2␣in␣T3␣element␣shape␣functions,␣not␣{}".format(self.p))

class T3_phipy(shape_fns):
    def expression(self, xi=0, eta=0):
        if self.p == 0:
             return 0
        elif self.p == 1:
            return 1
        elif self.p == 2:
            return -1
        else:
            raise ValueError("p␣should␣be␣0,␣1␣or␣2␣in␣T3␣element␣shape␣functions,␣not␣{}".format(self.p))


class Q4_phi(shape_fns):
    def expression(self, xi=0, eta=0):
        if self.p == 0:
            return (xi-1)*(eta-1)/4
        elif self.p == 1:
            return (1 + xi) * (1 - eta)/4
        elif self.p == 2:
            return (1 + xi) * (1 + eta)/4
        elif self.p == 3:
            return (1 - xi) * (1 + eta)/4
        else:
            raise ValueError("p␣should␣be␣0,␣1,␣2␣or␣3␣in␣Q4␣element␣shape␣functions,␣not␣{}".format(self.p))

class Q4_phipx(shape_fns):
    def expression(self, xi=0, eta=0):
        if self.p == 0:
             return (eta - 1)/4
        elif self.p == 1:
            return (1 - eta)/4
        elif self.p == 2:
            return (1 + eta)/4
        elif self.p == 3:
            return -(1 + eta)/4
        else:
            raise ValueError("p␣should␣be␣0,␣1,␣2␣or␣3␣in␣Q4␣element␣shape␣functions,␣not␣{}".format(self.p))


class Q4_phipy(shape_fns):
    def expression(self, xi=0, eta=0):
        if self.p == 0:
            return (xi - 1)/4
        elif self.p == 1:
            return -(xi + 1)/4
        elif self.p == 2:
            return (1 + xi)/4
```

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

Listing 7: *Defeinitions of shape functions for T3 and Q4 elements*

J. Gaussian points in 2D

```
def Gauss_points(element, order):
    if element.shape == 'quad':
        xi, wi = np.polynomial.legendre.leggauss(order)
        points = [(x, y) for x in xi for y in xi]
        weights = [wx * wy for wx in wi for wy in wi]

    elif element.shape == 'triangle':
        NGP_data = {
            1: {
                'points': np.array([(1/3, 1/3)]),
                'weights': np.array([1/2])
            },
            3: {
                'points': np.array([(1/6, 1/6), (2/3, 1/6), (1/6, 2/3)]),
                'weights': np.array([1/6, 1/6, 1/6])
            },
            4: {
                'points': np.array([(1/3, 1/3), (0.6, 0.2), (0.2, 0.6), (0.2, 0.2)]),
                'weights': np.array([-27/96, 25/96, 25/96, 25/96])
            }
        }
        if order == 2:
            order = 3
        points, weights = NGP_data[order]['points'], NGP_data[order]['weights']
    else:
        raise ValueError("Shape not supported")

    return points, weights
```

Listing 8: *Gaussian points in 2D*

K. Von Mise 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:

Von Mises Criterion According to the Implemented Code

The von Mises stress criterion, as implemented in the provided Python code, is computed using the formula:

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

Here, σ_x and σ_y are the normal stresses in the x and y directions, respectively, and τ_{xy} is the shear stress in the xy plane.

The code for Von Mise stress is as follows:

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

Listing 9: *Von Mise stress*

L. Strain Energy in 2D

```
def cal_energy(elements_list, GPN = 2):
  E = 200e3
  nu = 0.3
  D = E / (1 - nu**2)* np.array([
      [1, nu, 0],
      [nu, 1, 0],
      [0, 0, (1-nu)/2]
      ])
  energy = 0
  for elem in elements_list:
      elem_energy = 0
      points, Ws = Gauss_points(elem, GPN)
      loop = 0
```

```
        scale = 4 if elem.shape=="
            triangle" else 1
        for g in range(len(Ws)):
            xy = points[g]
            W = Ws[g]
            strain_list = elem(xy[0], xy
                [1], 'strain')
            dN = elem.gradshape(xy[0], xy
                [1])
            # J = jacobian(self.vertices,
                dN)
            J = np.dot(dN , elem.vertices
                )
            J_det = np.linalg.det(J)
            B = elem.B_matrix(J, dN)
            this_energy = 0.5 * W *
                strain_list.T @ D @
                strain_list * J_det #*
                scale
            elem_energy += this_energy
            loop+=1
        energy+=elem_energy
    return energy[0][0]
```

Listing 10: *Strain energy in 2D*

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 σ_{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 σ_{avg}.
$$\sigma_{\text{avg}} = \frac{1}{m} \sum_{j=1}^{m} \sigma_{\text{Node},j}$$
where m is the number of nodes shared by adjacent elements.