Final Project for Advanced FEM (ME46050)

Xusen Qin

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

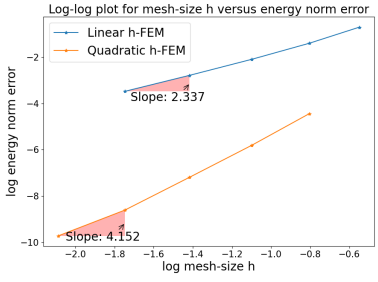
I. Introduction

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 element and Q4 elements.

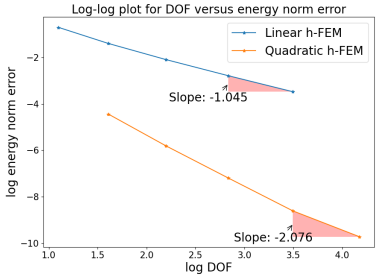
II. PRMBLEM 1

i. Question 1

The code for finite element method, shape functions as well as the gaussion integration method in Appendix.A, B, and C.



(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.1, which can also be defined as the slope of the log-log plot. 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 2 for linear elements and 4 for quadratic elements. 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.2 of the relative error in the energy norm versus the number of DOFs, the slopes of the plotted lines represent these rates. The inclusion of h-version plots in the same figure provides a comparative perspective, showcasing the efficiency and accuracy of each method relative to the other.

Given the computed convergence rates for the different finite element methods, we observe the following rates:

- 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 -8.245.

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. 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.

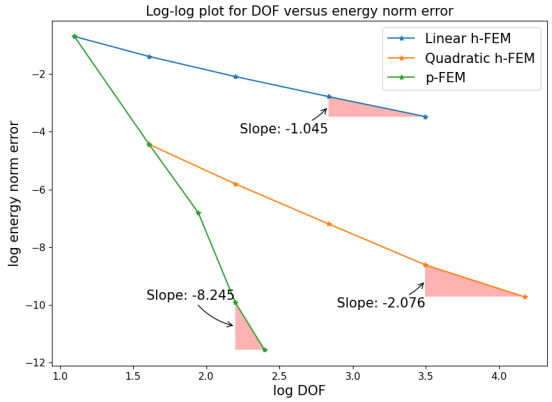


Figure 2: *log-log plot of the error versus DOF in h-version and p-version FEM.*

iii. Question 3

In order to estimate the error in our 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 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.6:

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

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.

Certainly, based on the table provided, here's a suitable answer:

	Energy	Relative Error
Linear	0.034626674	2.7117(%)
Quadratic	0.035591726	0.00132(%)
Exact solution	0.035591838	/

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

The table.1 presents the energy values obtained using different Finite Element Methods (FEM) and their respective relative errors when compared to the exact solution.

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 0.00132%. This suggests that the quadratic FEM is significantly more accurate than the linear FEM for this problem. The exact solution, as expected, has an energy of 0.03559 with no relative error.

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.D.

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.3 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.4 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 our computed results closely mirrored this value.

v. Question 5

In the p-version study, the problem was analyzed using 5 elements with polynomial degrees ranging from $p = 1$ to $p = 5$. The numerical solutions obtained

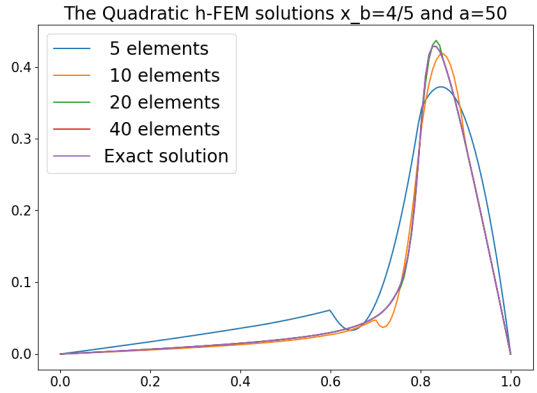


Figure 3: *The Quadratic h-FEM solutions x_b=4/5 and a=50 with different element numbers.*

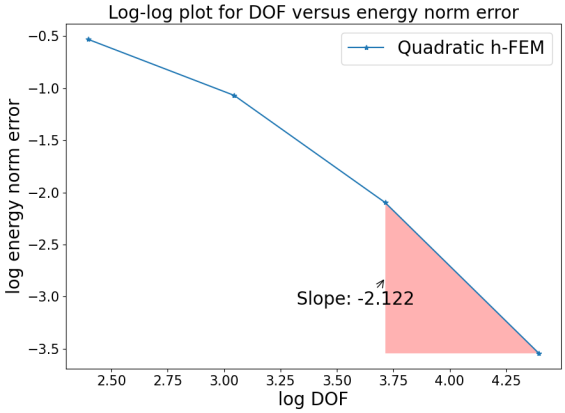


Figure 4: *Log-log plot for DOF versus energy norm error.*

were juxtaposed against the exact solution, providing a comprehensive understanding of the precision of our approach, as depicted in Fig.5. Further insights were gleaned from the log-log plot showcasing the relative error in the energy norm against the number of DOFs, illustrated in Fig.6.

From the log-log plot, the computed rate of convergence for the p-version was approximately -0.305, whereas for the h-version, it was -2.2122. The convergence rate of quadratic h-FEM is faster than p-FEM in this problem. However, for a comparable range of DOFs (specifically from 11 to 21 DOFs), the p-version displayed a lower error than the h-version. This underscores the efficacy of the p-version in yielding more accurate results with fewer degrees of freedom.

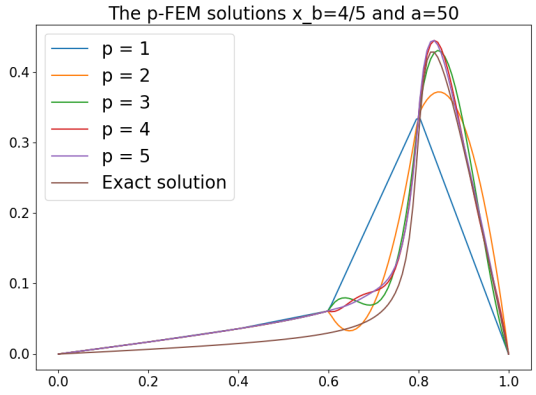


Figure 5: *The p-FEM solutions x_b=4/5 and a=50 with different element numbers.*

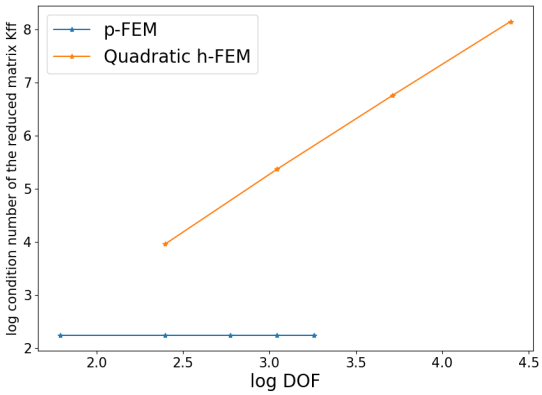


Figure 7: *Log-log plot for the condition number of the reduced matrix K_{ff} versus energy norm error.*

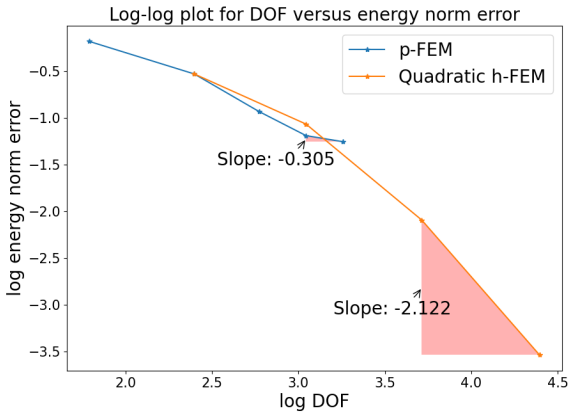


Figure 6: *Log-log plot for DOF versus energy norm error in p-FEM and h-FEM.*

vi. Question 6

Log-log plot for the condition number of the reduced matrix K_{ff} versus energy norm error for the h-FEM and p-FEM results is represented in Fig.7.

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.
- **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**: For problems with strong gradients or sharp features, the quadratic FEM is better suited due to its adaptability and ability to capture local variations more accurately.

vii.2 Quadrature Points and Computation Efficiency

Regarding the Gauss quadrature points, six points were chosen for the finite element computations. The rationale behind this choice was twofold. Firstly, the use of a higher number of Gauss points typically results in increased precision during integration, thus ensuring more accurate results. Secondly, given the efficiency of the computational setup, the addition of these extra points did not noticeably affect the overall computational time. As such, opting for six Gauss points struck a balance between computational efficiency and the desire for enhanced accuracy in the results.

III. On LaTeX

The following is a compilation of frequently used LaTeX constructs that you can use to build your report. You can find further information in the LaTeX cheat sheet provided in class.

Itemized lists are given in this format:

- Item 1;
- Item 2;
- ...
- Item N.

Enumerated lists are given in this format:

1. First item;
2. Second item;
3. ...
4. Last item.

Citations can be included by using the `cite` command: [**?**]. You also have to add the bibliographic entry at the end of the file.

Text requiring further explanation[1].

Figures 9 and 8 show, respectively, figures that take the entire width or only one columen.

You can also add subfigures using the `subcaption` package. You can refer to the total figure with 10 or to individual components as 10a-c.

Here there is a complex table prepared in LaTeX that showcases several improvements over standard tables, including footnotes, handling multiple hierarchical levels using `multirow` and `multicolumn`, different alignment options using custom column commands, partial rules, and coloring particular cells or even entire columns.

[1] Example footnote

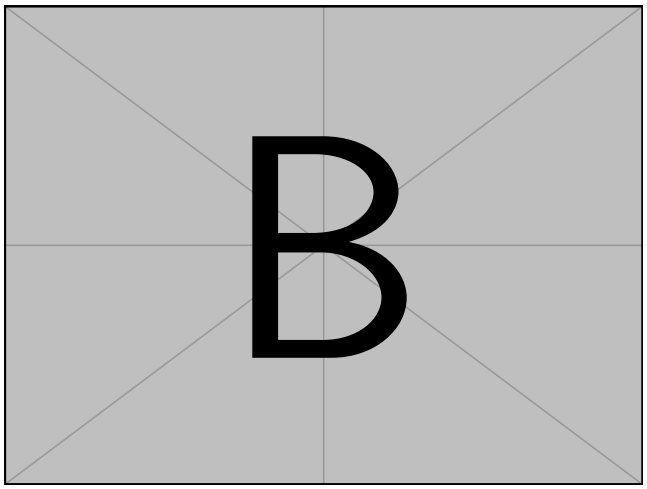


Figure 8: *A figure that takes a single column.*

	categories		
	A	B	C†
A	1	2	3
	4	5	6
	7	8	9

† Center-aligned

Equations can be written using hte `equation` environment:

$$s\boldsymbol{v} = \boldsymbol{T}\boldsymbol{v} \tag{11}$$

Notice the notation for scalars, vectors, and tensors. In addition, you can use $\boldsymbol{\nabla}, \boldsymbol{\nabla}\cdot$ to denote gradient and divergence. If you need to align equations, you can use the `align` environment:

$$\mathrm{d}x \qquad \frac{\mathrm{d}}{\mathrm{d}x} \qquad \frac{\mathrm{d}f}{\mathrm{d}x} \qquad \frac{\mathrm{d}^n f}{\mathrm{d}x^n}, \tag{12}$$

$$\frac{\partial}{\partial x} \qquad \frac{\partial f}{\partial x} \qquad \frac{\partial^n f}{\partial x^n}. \tag{13}$$

Here you also have the syntax for braces and (partial) derivatives. Equations can also be referred with (12) and (13). If a single equation number is needed for an equation spanning multiple lines, use the `aligned` environment inside the `equation` environment:

$$\begin{aligned} &|a| \quad \|b\|, \\ &(a) \quad [b] \quad \{c\} \quad |d|, \end{aligned} \tag{14}$$

where you see the syntax used for braces.

Some results will require log-log plots, which can be created in LaTeX using the `pgfplots` package:

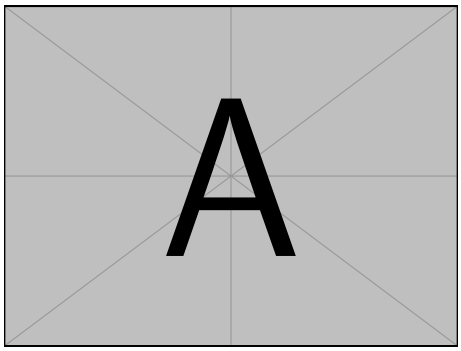
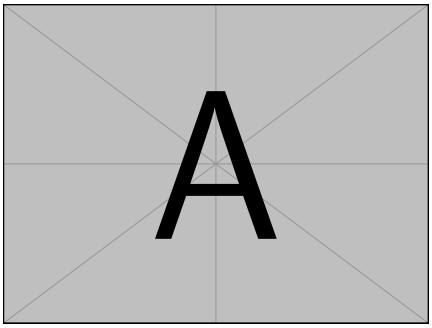
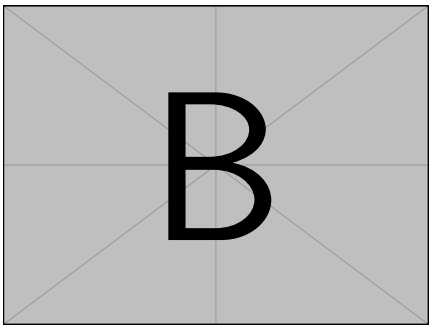


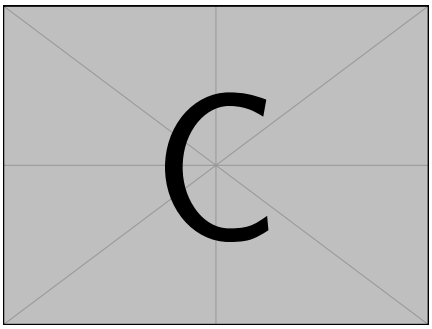
Figure 9: *A figure that spans both columns.*



(a)



(b)



(c)

Figure 10: *This is the caption for the entire figure: (a) Left; (b) Middle; and (c) Right.*

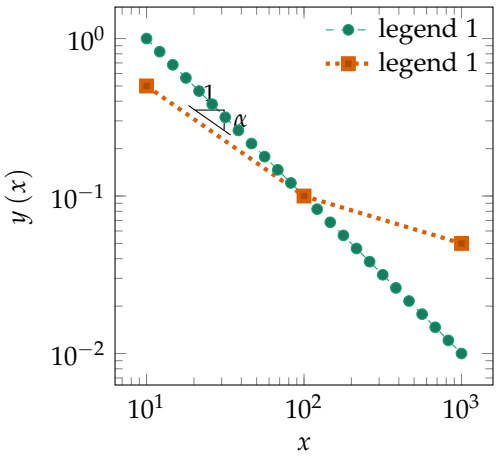


Figure 11: *A log-log plot.*

IV. Discussion

The discussion is a very important part of the report, so make sure you write it properly.

Remember there is a 10-page limit!

References

A. 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)
```

```
    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_php = {'phis': [], 'phips':
        []}
    phi_php['phis'] = joint_funcs(
        linear_phi_php['phis']) +
        nonlinear_phi_php['phis']
    phi_php['phips'] = joint_funcs(
        linear_phi_php['phips']) +
        nonlinear_phi_php['phips']
    u_list = []
    for i in range(len(phi_php['phis
        '])):
        u_list.append(mul(U[i],
            phi_php['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_php['
            phis'])):
            func = phi_php['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_php, uh, cont_K
```

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

B. Defeinition of the shape functions in 1-D

```
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*

C. 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*

D. 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 4: *A posteriori error estimate*