ME5204 - Finite Element Analysis

Assignment 3 - L^2 Projection

Name - Jay Prajapati
Roll number - ME21B143

Table of Contents

1 Question 1

- The main aim of this question is to first get the Interpolated function i.e. L^2 Projection of the following function given below.

$$f(x) = e^{\sin\left(\frac{\pi x^2}{4}\right)} \tag{1}$$

```
def func(x):
    return np.exp(np.sin(0.25 * np.pi * x * x))
```

- The other important thing is to get the interpolated function in a way that error is reduced since interpolating function does not pass through points which is given below.

$$min\ ||e||^2 = (f(x) - I_h f(x))^2 \tag{2}$$

- Now, we know that error is orthogonal to space of polynomials g(x). This is called Galerkin orthogonality.

$$\int_\Omega R(x)g(x)dx = 0 \tag{3}$$

where R(x) is given as

$$R(x) = f(x) - I_h(x)$$

- The interpolating function is taken as piecewise polynomials which will lead to a sparse matrix A and possibly lead to a unique solution.

$$I_h(x) = \sum_{i=1}^{n} C_i \phi_i(x) \tag{4}$$

- The $\phi(x)$ function for x_i and its neighbourhood looks like below.

$$\phi_1(x) = \begin{cases} \frac{x_i - x}{x_i - x_{i-1}} & \text{if } x \in (x_{i-1}, x_i) \\ 0 & else \end{cases}$$

$$\phi_2(x) = \begin{cases} \frac{x-x_{i-1}}{x_i-x_{i-1}} & \text{if } x \in (x_{i-1}, x_i) \\ \frac{x_i-x}{x_{i+1}-x_i} & \text{if } x \in (x_i, x_{i+1}) \end{cases}$$

$$\phi_3(x) = \begin{cases} 0 & \text{if } x \in (x_{i-1}, x_i) \\ \frac{x-x_i}{x_{i+1}-x_i} & \text{if } x \in (x_i, x_{i+1}) \end{cases}$$

- Substituting (4) in (3) and solving, we get

$$(\int_\Omega \phi_I \phi_J dx)\ C_I = \int_\Omega f(x)\phi_I dx \tag{5}$$

$$AC = F \tag{6}$$

- We will be using Element-based thinking approach to get A and F matrix i.e. compute local a and local f for every element and do assembly it bigger matrix. Below code explains the assembly.

```
def Assembly(a, f, A, F, n):
    for i in range(2):
        for j in range(2):
            A[n[i]][n[j]] += a[i][j]

    for i in range(2):
        F[n[i]] += f[i]
```

- For evaluating the integrals, I have used 3-point numerical integration method for which gauss points and weights are given by

$$\int\int f(x,y)dA = \int\int f(\varepsilon,\eta)\ |J|\ d\varepsilon\ d\eta = \sum_I f(\varepsilon,\eta) * |J| * w_I$$

$$Gauss\ points = (-\sqrt{0.6}, 0, \sqrt{0.6})$$

$$Weights = (5/9, 8/9, 5/9)$$

1.1 Interpolated function

- The following procedure shows how to evaluate the interpolated function given number of nodes:-

1. **Step 1**:- Initialize the A and F matrix as shown below. Also, divide the interval $I = [0, 3]$ into num_nodes using linspace functionality of python. And similarly assign nodes to each element.
2. **Step 2**:- Start the loop over all elements. Access indices of nodes in the elements and extract its coord.
3. **Step 3**:- Start loop over gauss integration points for each element. Initialize the Shape function denoted by N and its derivative by dN. Evaluate the Jacobian with following equation.

$$Jac = dN^T coord$$

4. **Step 4**:- Evaluate the points value and then calculate the local matrix a and f using below given formula.

$$a = N^T N \times Jac \times wt[j]$$

$$f = N \times func(X) \times Jac \times wt[j]$$

5. **Step 5**:- Assembly the evaluated local a and local f matrix using the Assembly function show above.
6. **Step 6**:- Final step is to get the C matrix after completing both the loops.

$$C = A^{-1}\ F \tag{7}$$

- These values of C substituted in (4) gives you the Interpolated function. Below code shows all the steps.

```
def Compute_Interpolated_Function(num_nodes):
    nodes = np.linspace(0, 3, num_nodes)
    elements = np.column_stack((np.arange(num_nodes - 1),
                                np.arange(1, num_nodes)))

    A = np.zeros((num_nodes, num_nodes))
    F = np.zeros((num_nodes, 1))

    for e in elements:
        n = np.array(e[:2], dtype=int)
        coord = np.array([nodes[n[0]], nodes[n[1]]])
```

```python
            for j in range(3):
                pt = gp[j]
                N = np.array([(1 - pt)/2, (1 + pt)/2])
                dN = np.array([-0.5, 0.5])
                Jac = dN.T @ coord
                X = N @ coord
                a = np.outer(N, N) * Jac * wt[j]
                f = func(X) * wt[j] * Jac * N
                Assembly(a, f, A, F, n)

        A_inv = np.linalg.inv(A)
        C = A_inv @ F
        return C

    C = Compute_Interpolated_Function(num_nodes)
```

1.2 Number of points for convergence

- The following shows the procedure to compute error in L^2 Projection.
- A slight modification in code for computing gives us the error. We will evaluate the Actual functions' value using X and interpolated functions' value using given formula.

$$I_h(x) = N \times local\_c^T$$

- The error can be computed using given formula and then at the end will take square root of the error.

$$error^2 = \sum_{i=1}^{n} \sum_{j=1}^{3} (f(x_i) - I_h(x_i))^2 \times Jac \times wt[j]$$

```python
    def Compute_Error(C, num_nodes):
        nodes = np.linspace(0, 3, num_nodes)
        elements = np.column_stack((np.arange(num_nodes - 1),
                                    np.arange(1, num_nodes)))
        error = 0.0

        for e in elements:
            n = np.array(e[:2], dtype=int)
            local_c = C[n, 0]
            coord = np.array([nodes[n[0]], nodes[n[1]]])
```

```
            for j in range(3):
                pt = gp[j]
                N = np.array([(1 - pt)/2, (1 + pt)/2])
                dN = np.array([-0.5, 0.5])
                Jac = dN.T @ coord
                X = N @ coord
                f_x = func(X)
                I_h_x = N @ local_c.T
                error += (f_x - I_h_x)**2 * Jac * wt[j]

        error = np.sqrt(error)
        return error

    error = Compute_Error(C, num_nodes)
```

- The following function was run for given num_nodes and computed the error until it reached less than 1×10^{-5} .

```
    num_nodes_list = np.array([10, 25, 50, 100, 200, 300, 400, 500
                                                         , 600, 700])
    mesh = 3 / (num_nodes_list - 1)
    error = []

    for num_node in num_nodes_list:
        C = Compute_Interpolated_Function(num_node)
        error.append(Compute_Error(C, num_node))

    error = np.array([error])
```

- The solution converged at 700 number of nodes as error is $9.98350303369096 \times 10^{-6}$ and for 699, error is $1.0012132352988598 \times 10^{-5}$.

Nodes	**Mesh size**	**Error**
10	0.33333333	7.62830609e-02
25	0.125	8.72076571e-03
50	0.06122449	2.05932399e-03
100	0.03030303	4.99749830e-04
200	0.01507538	1.23306490e-04
300	0.01003344	5.45857195e-05
400	0.0075188	3.06462914e-05
500	0.00601202	1.95919062e-05
600	0.00500835	1.35956178e-05
699	0.00429799	1.00121324e-05
700	0.00429185	9.98350303e-06

- Hence, the number of points required such that error in L^2 norm is less than 1×10^{-5} is **700**.

1.3 Plot for error vs mesh size

- The graph for error vs mesh size is plotted using matplotlib as shown below. The linear regression method is used to evaluate the slope i.e. rate of convergence of the L^2 Projection.

```
def Plot_Error_Mesh():
    num_nodes_list = np.array([10, 25, 50, 100, 200, 300, 400,
                                         500, 600, 700])
    mesh = 3 / (num_nodes_list - 1)
    error = []

    for num_node in num_nodes_list:
        C = Compute_Interpolated_Function(num_node)
```

```
        error.append(Compute_Error(C, num_node))

    error = np.array([error])
    log_mesh = np.log(mesh.T).flatten()
    log_error = np.log(error.T).flatten()

    # Perform linear regression to get the slope and intercept
    slope, intercept, r_value, p_value, std_err
                          = stats.linregress(log_mesh, log_error)

    # Plotting the log-log plot
    plt.figure(figsize=(8, 6))
    plt.plot(log_mesh, log_error, marker='o'
                               , linestyle='-', label='Data')
    plt.plot(log_mesh, slope * log_mesh + intercept
        , linestyle='--', color='red', label=f'Fit: slope={slope:.15f}')
    plt.xlabel('log(mesh size)')
    plt.ylabel('log(error)')
    plt.title('Log-Log Plot of Error vs. Mesh size')
    plt.grid(True)
    plt.legend()
    plt.show()

    # Print slope value
    print(f"Rate of Convergence is : {slope}")

Plot_Error_Mesh()
```

- Below given is the log-log plot of error as function of mesh size. For which rate of convergence comes out to be **2.036702130217686**.

$$Rate\ of\ Convergence = \mathbf{2.036702130217686}$$

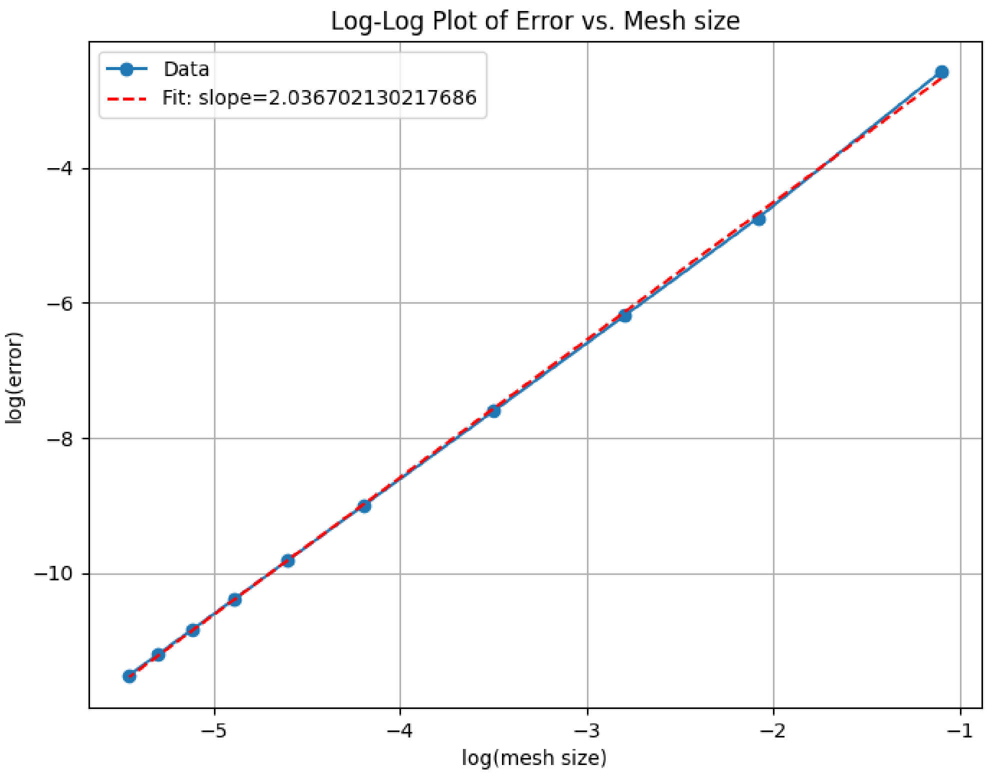

Figure 1: Error vs Mesh size

1.4 Overlay of Actual and Interpolated function

- The Actual function is evaluated at 10,000 points and interpolated function at 1000 points. And both are shown on the same plot to be represented as an overlay.

```
def Plot_Comparison():
    num_nodes = 1000
    nodes = np.linspace(0, 3, num_nodes)
    C = Compute_Interpolated_Function(num_nodes)
    fine_points = np.linspace(0, 3, 10000)
    actual_func_values = func(fine_points)

    # Plot the actual function and the interpolated function
    plt.figure(figsize=(10, 6))

    # Plot actual function with a solid black line and no markers
```

```python
    plt.plot(fine_points, actual_func_values, label='Actual Function',
                                color='black', linestyle='-', linewidth=2)

    # Plot interpolated function with dashed line
    plt.plot(nodes, C, label='Interpolated Function', color='red',
                                                  linestyle='--')
    plt.xlabel('x')
    plt.ylabel('Function Value')
    plt.title('Actual Function vs Interpolated Function')
    plt.grid(True)
    plt.legend()
    plt.show()
```

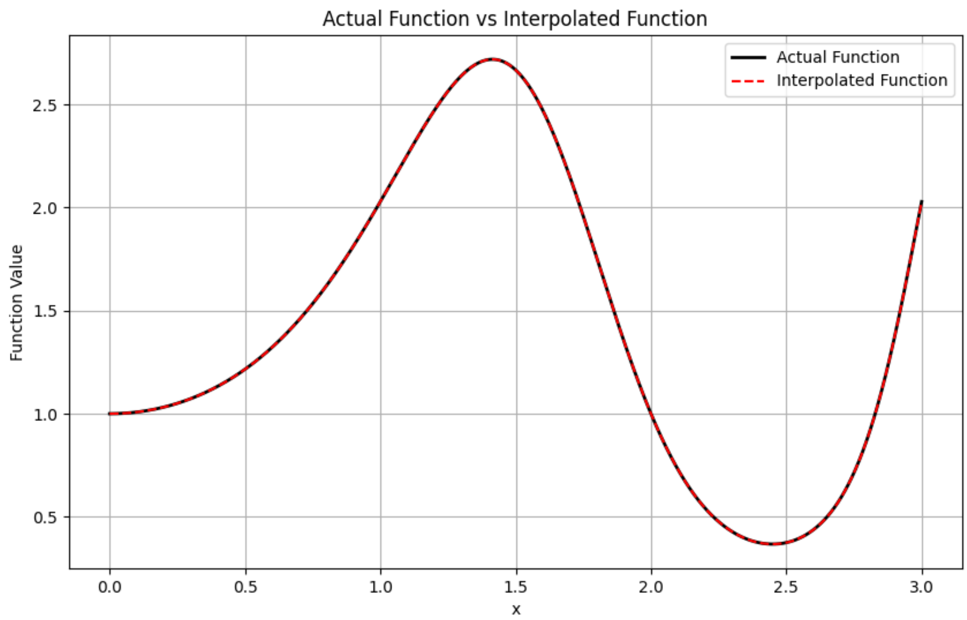

Figure 2: Actual vs Interpolated Function

- From the above plot, we can infer that our Interpolated function is good approximation of the given function (1).

2 Question 2

- The main objective of this question is to first get L^2 Projection for IITM_Map. The Assumption is that the heat source follows a Multivariate Gaussian distribution with centre of Gajendra circle as its mean. Equation of which is represented below.

$$f(x) = \frac{exp(\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu))}{2\pi \ (det(\Sigma))^{-1/2}} \tag{8}$$

where μ is [286.9, 260.6] extracted from geo file (centre of black circle). Also, base radius is equal to diameter of inner circle which is equal to 3 units.

- Another Assumption is that we assume that 99.73% is distributed or covered by this Gaussian distribution hence sigma will be r/3, Also, due to symmetry covariance of x and y is assumed to be zero. So, covariance matrix looks like this

$$\Sigma = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{9}$$

- Below code represents how the function is calculated.

```
cov = np.array([[sig, 0], [0, sig]])
cov_det = np.linalg.det(cov)
cov_inv = np.linalg.inv(cov)

def func(x):
    numerator = np.exp(-0.5 * (x - mean).T @ cov_inv @ (x - mean))
    denominator = 2 * np.pi * np.sqrt(cov_det)
    return numerator / denominator
```

2.1 L2 Projection

- The L^2 Projection of the given function is taken as a piecewise polynomials which is given below.

$$I_h(x) = \sum_{i=1}^{n} C_i \phi_i(x) \tag{10}$$

- Now, we know that error is orthogonal to space of polynomials g(x). This is called Galerkin orthogonality.

$$\int_{\Omega} R(x)g(x)dx = 0 \tag{11}$$

where R(x) is given as

$$R(x) = f(x) - I_h(x)$$

- Substituting (10) in (11), we get

$$(\int_{\Omega} \phi_I \phi_J dx)\ C_I = \int_{\Omega} f(x)\phi_I dx \tag{12}$$

$$AC = F \tag{13}$$

- For evaluating the integrals, I have used 3-point numerical integration method by Gauss-Legrendre Quadrature. The Gauss points and weights are also given below.

$$\int\int f(x,y)dA = \int\int f(\varepsilon,\eta)\ |J|\ d\varepsilon\ d\eta = \sum_I f(\varepsilon,\eta) * |J| * w_I$$

$$Gauss\ points = (2/3, 1/6), (1/6, 2/3), (1/6, 1/6)$$

$$Weights = (1/6, 1/6, 1/6)$$

- The following procedure explains on how is C calculated and then error is calculated for a particular mesh size.

  1. **Step 1**:- Initialize the A and F matrix as shown below. For reading the mesh file, I have used meshio library, the code for which is shown below.

```
def Read_Data():
    file = meshio.read(file_path)
    nodes = file.points
    elements = file.cells_dict['triangle']
    nodes = nodes[:, : 2]
    return nodes, elements

nodes, elements = Read_Data()
num_nodes, d = nodes.shape
```

2. **Step 2**:- Start the loop over all elements. Access indices of nodes in the elements and extract its coord.

3. **Step 3**:- Start loop over gauss integration points for each element. Initialize the Shape function denoted by N and its derivative by dN. Evaluate the Jacobian with following equation.

$$Jac = dN^T coord$$

4. **Step 4**:- Evaluate the points value and then calculate the local matrix a and f using below given formula.

$$a = N^T N \times det(Jac) \times wt[j]$$

$$f = N \times func(X) \times det(Jac) \times wt[j]$$

5. **Step 5**:- Assembly the evaluated local a and local f matrix using the Assembly function show below.

```
def Assembly(a, f, A, F, n):
    for i in range(3):
        for j in range(3):
            A[n[i]][n[j]] += a[i][j]

    for i in range(3):
        F[n[i]] += f[i]
```

6. **Step 6**:- The next step is to get the C matrix after completing both the loops. Here, I have made an **Engineering Approximation** which is that A matrix has 2-3 rows, columns as a zero which leads to a Singular matrix and not getting a solution which is to due to some irregularity in mesh itself. So I have added a tolerance value to A matrix which is very small so that inverse is also evaluated and does not affect the solution much.

$$C = A^{-1}\ F \tag{14}$$

```
def Compute_Interpolated_Function(num_nodes):
    A = np.zeros((num_nodes, num_nodes))
    F = np.zeros((num_nodes, 1))
```

```python
    for e in elements:
        n = np.array(e[:3], dtype=int)
        coord = np.array([nodes[n[0]], nodes[n[1]],
                                         nodes[n[2]]])
        for j in range(3):
            pt = gp[j]
            N = np.array([1 - pt[0] - pt[1], pt[0], pt[1]])
            dN = np.array([[-1, -1], [1, 0], [0, 1]])
            Jac = dN.T @ coord
            X = N @ coord
            a = np.outer(N, N) * np.linalg.det(Jac) * wt[j]
            f = func(X) * wt[j] * np.linalg.det(Jac) * N
            Assembly(a, f, A, F, n)

    # Engineering Approximation
    A += np.eye(A.shape[0]) * 1e-10
    A_inv = np.linalg.inv(A)
    C = A_inv @ F
    return C
```

7. **Step 7**:- For computing error it is similar to question 1, just shape function N and its derivative dN changes.

```python
def Compute_Error(C):
    error = 0.0

    for e in elements:
        n = np.array(e[:3], dtype=int)
        local_c = C[n, 0]
        coord = np.array([nodes[n[0]], nodes[n[1]],
                                         nodes[n[2]]])

        for j in range(3):
            pt = gp[j]
            N = np.array([1 - pt[0] - pt[1], pt[0], pt[1]])
            dN = np.array([[-1, -1], [1, 0], [0, 1]])
            Jac = dN.T @ coord
            X = N @ coord
            f_x = func(X)
            I_h_x = N @ local_c.T
            error += (f_x - I_h_x)**2 * np.linalg.det(Jac)
                                                * wt[j]

    error = np.sqrt(error)
    return error
```

- These values of C substituted in (10) gives you the Interpolated function or the L^2 projection of the given FE mesh.

2.2 Plot for error vs mesh size

- The below given table summarises all nodes and elements and its corresponding error evaluated according to previous section.

Nodes	**Elements**	**Error**
6450	12534	0.132410239599194
4475	8629	0.169716994230606
4347	8383	0.202336699820492
4189	8074	0.149912547070967
3926	7555	0.178616928667426
2751	5254	0.358493104489654
2148	4088	0.418517213027262
1965	3734	0.378775974974415
1926	3659	0.386533158534293

- The mesh size is evaluated by assuming average area of the mesh's sqaure root which is given as below.

$$h = \sqrt{\frac{Area\ of\ IITM}{Elements}}$$

  where Area of IITM as calculated in previous assignment is 130155.73500000025 sq units

- The below given code is used to generate the following log-log plot of error vs mesh size.

```
def Plot_Error_Mesh_Size():
    error = np.array([0.132410239599194,.., 0.386533158534293])
    Elements = np.array([12534,..., 3659])
    h = np.sqrt(130155.73500000025 / Elements)
```

```python
# Convert to log scale
log_mesh = np.log(h.T).flatten()
log_error = np.log(error.T).flatten()

# Perform linear regression to get the slope and intercept
slope, intercept, r_value, p_value, std_err
               = stats.linregress(log_mesh, log_error)

# Plotting the log-log plot
plt.figure(figsize=(8, 6))
plt.scatter(log_mesh, log_error, marker='o',
                          linestyle='-', label='Data')
plt.plot(log_mesh, slope * log_mesh + intercept,
linestyle='--', color='red', label=f'Fit: slope={slope:.15f}')
plt.xlabel('log(mesh size)')
plt.ylabel('log(error)')
plt.title('Log-Log Plot of Error vs. Mesh size')
plt.grid(True)
plt.legend()
plt.show()

# Print slope value
print(f"Rate of Convergence is : {slope}")
```

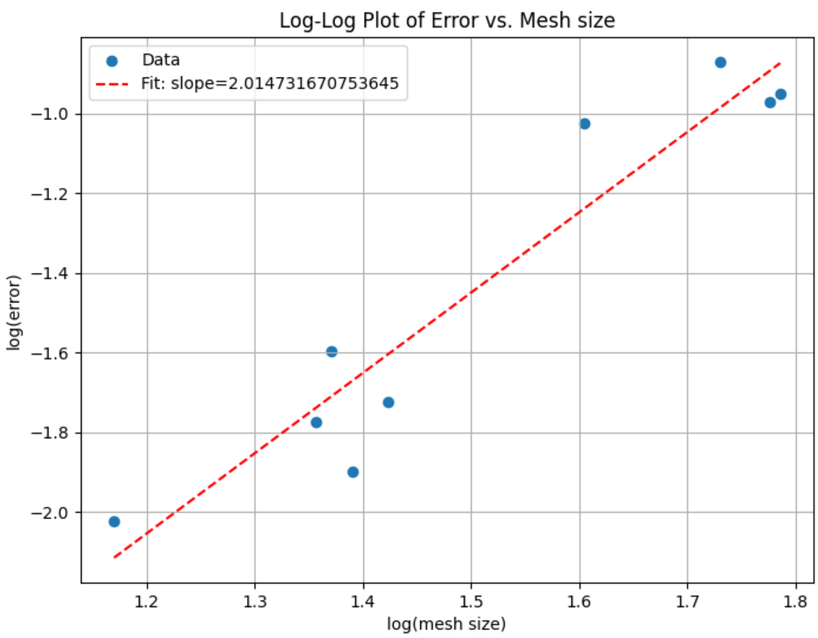

Figure 3: log-log plot of Error vs Mesh size

- The rate of convergence is **2.014731670753645**. Henc it is close to 2, so our assumption and approximation are reasonably good.

2.3 Contour Plot

- The 3D plot are drawn for both Actual function and interpolated function for comparison. Below given is code for it. IITM_Map.msh file is used for plotting which has 3926 nodes and 7555 elements.

```
def plot_3d_comparison():
    # Compute actual Gaussian values for plotting
    f = np.array([func([x, y]) for x, y in nodes])

    # Create 3D plot
    fig = plt.figure(figsize=(14, 10))

    # Plot Interpolated Function
    ax1 = fig.add_subplot(121, projection='3d')
    tri = Triangulation(nodes[:, 0], nodes[:, 1], elements)
    ax1.plot_trisurf(tri, C.flatten(), cmap='viridis', edgecolor='none')
    ax1.set_title('Interpolated Function')
    ax1.set_xlabel('X Coordinate')
    ax1.set_ylabel('Y Coordinate')
    ax1.set_zlabel('Interpolated Value')

    # Plot Actual Gaussian Function
    ax2 = fig.add_subplot(122, projection='3d')
    tri = Triangulation(nodes[:, 0], nodes[:, 1], elements)
    ax2.plot_trisurf(tri, f, cmap='viridis', edgecolor='none')
    ax2.set_title('Actual Gaussian Function')
    ax2.set_xlabel('X Coordinate')
    ax2.set_ylabel('Y Coordinate')
    ax2.set_zlabel('Gaussian Value')

    plt.show()
```

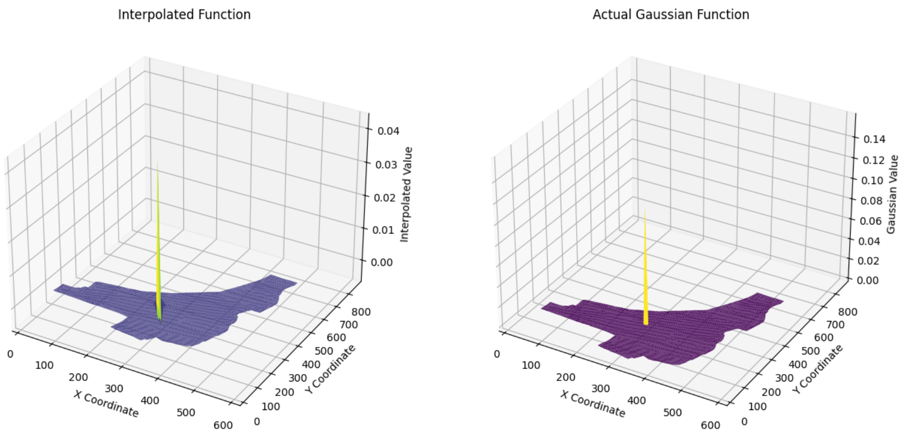

Figure 4: 3D plot for comparison

2.4 Error Plot

- An extension to above part is to plot error for difference between actual and interpolated function (f - C). Below given is code for that and plot also.

```
def Plot_Error():
    f = np.array([func([x, y]) for x, y in nodes])

    # Create 3D plot
    fig = plt.figure(figsize=(14, 10))

    # Plot Interpolated Function
    ax1 = fig.add_subplot(111, projection='3d')
    tri = Triangulation(nodes[:, 0], nodes[:, 1], elements)
    ax1.plot_trisurf(tri, f - C.flatten(), cmap='viridis'
                                          , edgecolor='none')
    ax1.set_title('Error Plot')
    ax1.set_xlabel('X Coordinate')
    ax1.set_ylabel('Y Coordinate')
    ax1.set_zlabel('Difference Value')

    plt.show()
```

- Note that both previous section and this section are plotted for IITM_Map.msh file, which has 3926 nodes and 7555 elements.

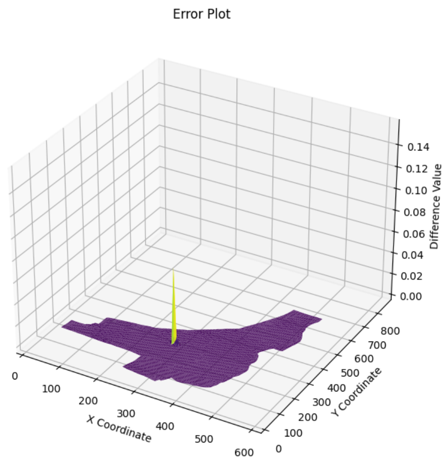

Figure 5: 3D plot for difference between errors