1 习题 3.14

考虑位移场 $u(x) = x^3, 0 \leq x \leq 1$。此为位移场的精确解。

1.1 一维两节点线性单元位移场计算方法

将求解域 $[0,1]$ 划分为 n 个单元。令节点 I 的为位移值给定，$U_I = (x_I)^3$。采用两节点线性单元计算每个单元的位移场 $u^e(x) = \boldsymbol{N^e(x)L^e d}$, 并绘制精确位移 $u(x)$ 和有限元位移 $u^e(x)$ 的曲线。这里 $\boldsymbol{L^e d = d^e}$，其中 $\boldsymbol{L^e}$ 是单元的提取矩阵，$\boldsymbol{d^e}$ 是单元位移列阵，$\boldsymbol{d}$ 是总体位移列阵。

我们将求解域平均分为 n 个单元。从左至右依次对节点编号为 1，2，3⋯⋯，这样便于我们直接组装各个节点的位移而不需要写出提取矩阵。为了计算位移场，我们需要计算各个节点的形函数 $\boldsymbol{N^e}$。对与一维线弹性问题，形函数为 $\boldsymbol{N^e} = [N_1^e\ N_2^e]$ 的形式。在每一个单元节点处的位移被给定为 $U_I = (x_I)^3$ 的情况下，所有单元的计算方式是完全一致的。单个单元形函数的计算方法有如下几步：

- 根据完备性要求，写出包含待定的系数的形函数 $u^e(x) = \alpha_0^e + \alpha_1^e x$。
- 带入边界条件，使得形函数满足连续性要求 $u^e(x_I^e) = u_I^e$
- 由边界条件组成的 Vandermonde 矩阵 $\boldsymbol{M^e}$ 和单元节点位移列阵 $\boldsymbol{d^e}$ 写出试探函数系数列阵 $\boldsymbol{\alpha^e}$。
- 将系数列阵 $\boldsymbol{\alpha^e}$ 带入形函数，得到单元形函数 $\boldsymbol{N^e = (M^e)^{-1} d^e}$

将每一个单元的单元形函数线性组合得到的位移 $u^e(x)$ 相加，即可得到总位移场。我们在编写程序时，直接使用了推倒完成的一维二节点线性单元形函数以完成数值计算。

1.2 一维两节点线性单元应变场计算方法

我们有如下方法用于计算每一个单元的应变:

$$\varepsilon^e(x) = \boldsymbol{B}^e \boldsymbol{d}^e \boldsymbol{B}^e = \frac{1}{l^e}\begin{bmatrix} -1 & 1 \end{bmatrix}$$

精确的应变场为:

$$\varepsilon(x) = \frac{du(x)}{dx} = 3x^2$$

1.3 2 单元位移与应变场

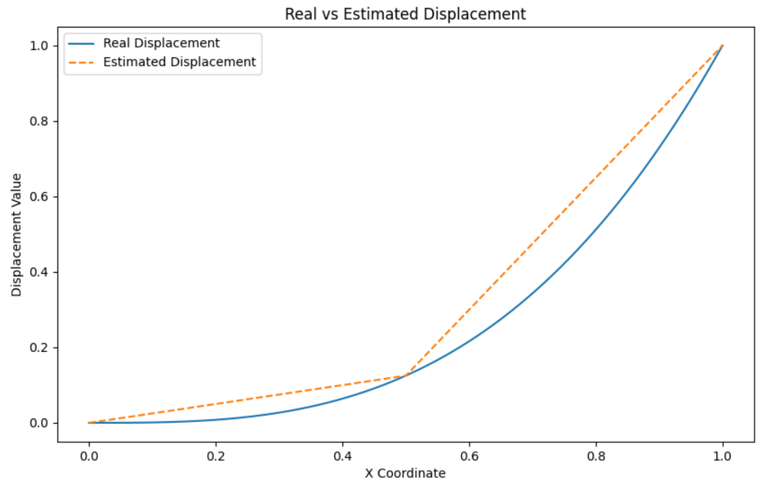

图 1: 2 单元位移场

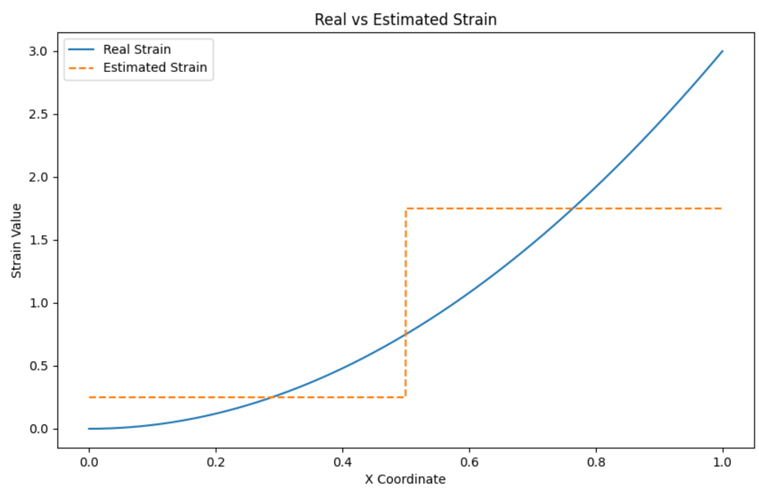

图 2: 2 单元应变场

1.4 4 单元位移与应变场

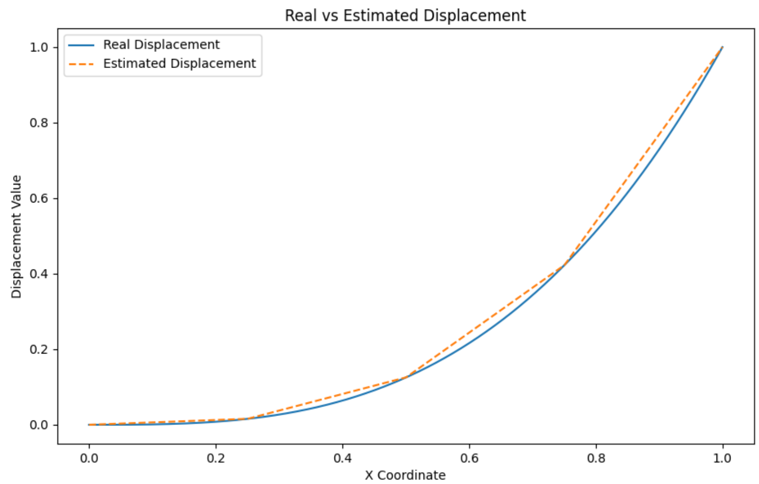

图 3: 4 单元位移场

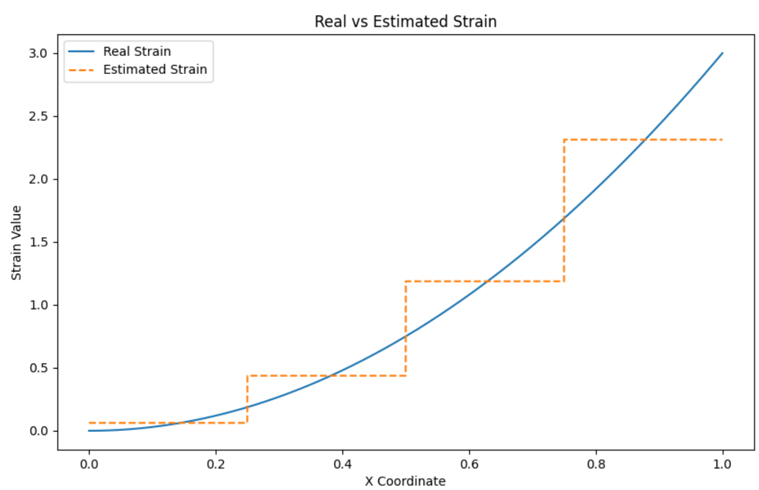

图 4: 4 单元应变场

1.5 8 单元位移与应变场

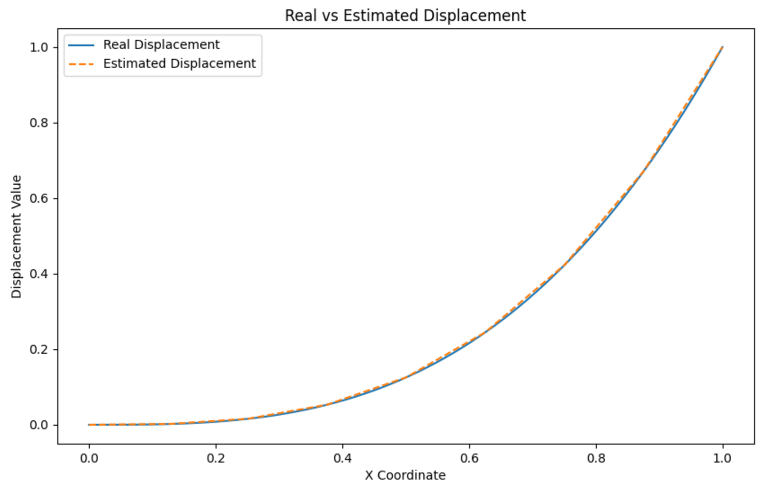

图 5: 8 单元位移场

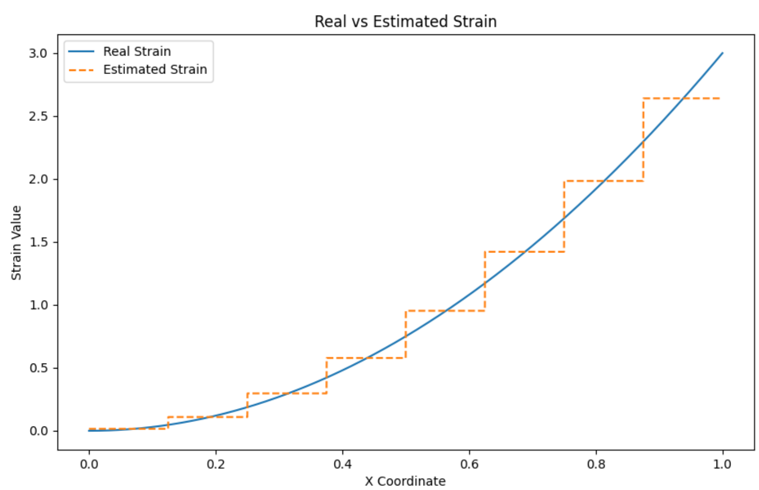

图 6: 8 单元应变场

1.6 差值函数的 L_2 范数误差计算方法

差值的误差可以由如下的 L_2 范数进行计算：

$$\|e\|_{L2} = (\int_0^L [u^e(x) - u(x)]^2 dx)^{\frac{1}{2}}$$

其中 $u(x) = x^3$ 为精确位移，$u^e(x)$ 为使用形函数计算的位移，L 为每一个单元的长度，在本题中和单元的数目相关：$L = \frac{1}{n}$，其中 n 为单元数。在程序中，我们仍然按照计算位移和应变场的方式：分单元计算再组装。这里计算积分时，我们使用高斯求积的方法计算。由于被积函数为最高次为 6，则我们采用 4 点高斯求积即可精确计算各项积分。

1.7 2 节点线性单元的 L_2 范数误差的计算

使用上述方法计算杆件被分为 2，4，8 个单元时，使用有限元方法计算的位移的 L_2 范数误差，绘制误差的对数和单元的长度的对数这一双对数曲线如下：

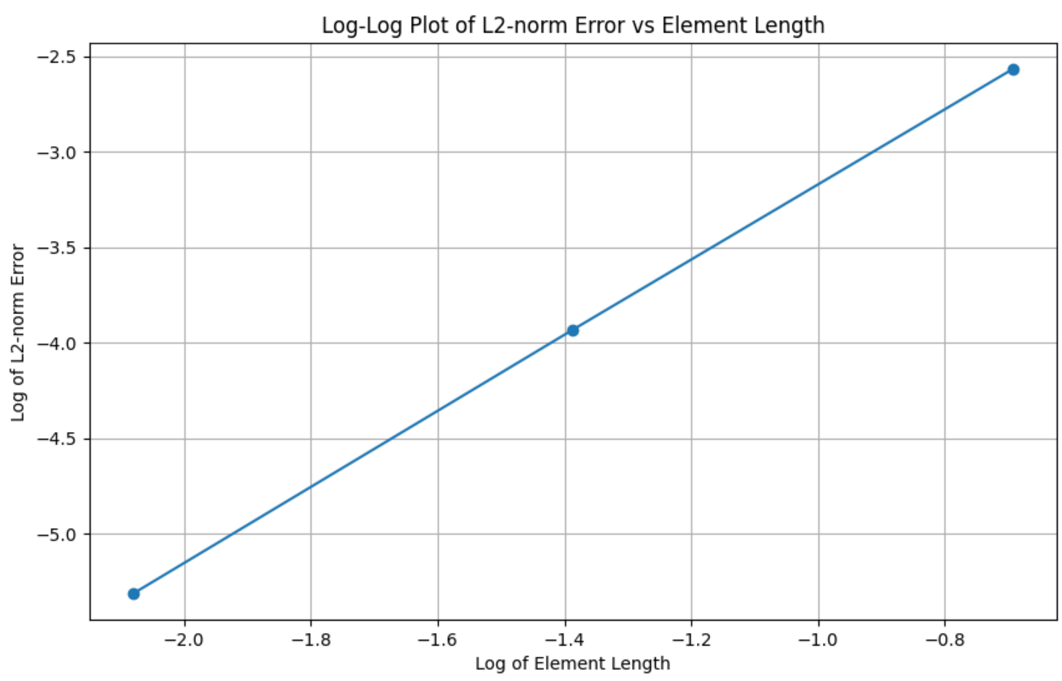

图 7: 误差-单元长度双对数曲线

L_2 范数误差的对数关于单元长度的对数的图线为一直线。这是因为误差与单元长度之间存在幂律关系。这种关系可以用以下公式表达：

$$\|e\|_{L2} = C \cdot L^p$$

其中，C 是一个常数，L 是单元长度，p 是表征误差随单元长度变化的幂指数。取两边的对数，我们得到：

$$\log(\|e\|_{L2}) = \log(C) + p \cdot \log(L)$$

这是一个线性关系，其中 $\log(C)$ 是 y 轴截距，p 是直线的斜率。直线的斜率代表着单元长度变化与误差的关系，斜率越大，误差减小的速度关于单元长度的变化速度更快。在程序中我们直接计算并输出了误差、直线的斜率和截距。

```
Error of 2 elements is:  0.07666796065160589
Error of 4 elements is:  0.019616628863701076
Error of 8 elements is:  0.004931859322266601
Slope:  1.966546670684943 Intercept:  3.799859989473584
```

1.8 3 节点线性单元 L_2 范数误差的计算

L_2 范数误差的计算方法仍然不变，但是我们使用三节点单元（二次单元）计算杆件的位移场函数。绘制误差的对数和单元长度的对数这一双对数曲线如下：

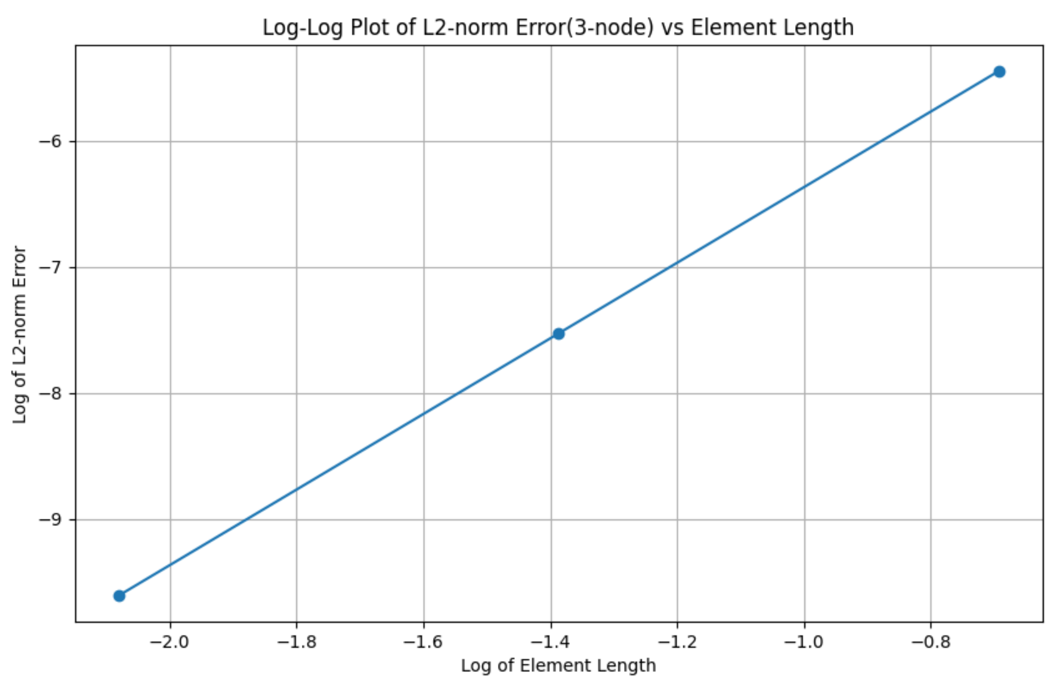

图 8: 误差 (三节点单元)-单元长度双对数曲线

在程序中我们直接计算并输出了误差、直线的斜率和截距。

```
Error of 2 elements is:  0.004312909745889711
Error of 4 elements is:  0.0005391137182362204
Error of 8 elements is:  6.738921477951645e-05
Slope:  2.9999999999999827 Intercept:  15.05116805855655
```

和使用两节点单元的计算结果对比可以发现，三节点单元直线的斜率更大，说明误差的收敛速度关于单元长度缩小的变化率更快。

2 Appendix-Displacement and Strain

All the source code could be founded in my github repository:
https://github.com/kkkjhgjhg4/FEM

```
import numpy as np
import matplotlib.pyplot as plt



def calculate_displacements(number_of_elements, x_element_steps=1000, plot = False):
    # Calculate the element length and coordinate steps
    element_length = 1 / number_of_elements
    x_element_coordinate = np.linspace(0, element_length, x_element_steps)

    # Global coordinates
    x_global_coordinate = np.linspace(0, 1, x_element_steps * number_of_elements)

    # Estimated displacement for one element
    def estimated_displacement_element(x_element_coordinate):
        displacement = np.zeros_like(x_element_coordinate)
        xe2 = x_element_coordinate[-1]
        xe1 = x_element_coordinate[0]
        ue1 = np.power(xe1, 3)
        ue2 = np.power(xe2, 3)

        for i, x in enumerate(x_element_coordinate):
            displacement[i] = ((xe2 - x) * ue1 + (x - xe1) * ue2) / element_length
        return displacement

    # Real displacement for global coordinates
    def real_displacement_global(x_global_coordinate):
        return np.power(x_global_coordinate, 3)

    # Calculate and concatenate estimated displacement for each element
    estimated_disp_global = np.concatenate([
        estimated_displacement_element(x_element_coordinate + i * element_length)
        for i in range(number_of_elements)
```

```
    ])

    # Real displacement for the entire domain
    real_disp_global = real_displacement_global(x_global_coordinate)

    # Plotting
    if plot:
        plot_displacement(estimated_disp_global, real_disp_global,
                                                  x_global_coordinate)

    #return estimated_disp_global, real_disp_global


def calculate_strains(number_of_elements, x_element_steps=1000, plot = False):
    # Calculate the element length and coordinate steps
    element_length = 1 / number_of_elements
    x_element_coordinate = np.linspace(0, element_length, x_element_steps)

    # Global coordinates
    x_global_coordinate = np.linspace(0, 1, x_element_steps *
                                              number_of_elements)

    # Estimated strain for one element
    def estimated_strain_element(x_element_coordinate):
        xe1 = x_element_coordinate[0]
        xe2 = x_element_coordinate[-1]
        ue1 = np.power(xe1, 3)
        ue2 = np.power(xe2, 3)
        strain = (ue2 - ue1) / element_length
        return np.full_like(x_element_coordinate, strain)

    # Real strain for global coordinates
    def real_strain_global(x_global_coordinate):
        return 3 * np.power(x_global_coordinate, 2)

    # Calculate and concatenate estimated strain for each element
    estimated_strain_global = np.concatenate([
        estimated_strain_element(x_element_coordinate + i * element_length)
        for i in range(number_of_elements)
    ])

```

```python
    # Real strain for the entire domain
    real_strain_global = real_strain_global(x_global_coordinate)

    # Plotting
    if plot:
        plot_strain(estimated_strain_global, real_strain_global, x_global_coordinate)

    #return estimated_strain_global, real_strain_global


# Plot functions
def plot_displacement(estimated_disp_global, real_disp_global, x_global_coordinate):
    plt.figure(figsize=(10, 6))
    plt.plot(x_global_coordinate, real_disp_global, label='Real Displacement')
    plt.plot(x_global_coordinate, estimated_disp_global, '--', label='Estimated Displacement')
    plt.legend()
    plt.xlabel('X Coordinate')
    plt.ylabel('Displacement Value')
    plt.title('Real vs Estimated Displacement')
    plt.show()

def plot_strain(estimated_strain_global, real_strain_global, x_global_coordinate):
    plt.figure(figsize=(10, 6))
    plt.plot(x_global_coordinate, real_strain_global, label='Real Strain')
    plt.plot(x_global_coordinate, estimated_strain_global, '--', label='Estimated Strain')
    plt.legend()
    plt.xlabel('X Coordinate')
    plt.ylabel('Strain Value')
    plt.title('Real vs Estimated Strain')
    plt.show()





# Calculate displacements for a unit length bar divided into any number of
```

```
                                                  elements
calculate_displacements(number_of_elements=2, plot=True)
calculate_displacements(number_of_elements=4, plot=True)
#calculate_displacements(number_of_elements=8, plot=True)

# Calculate strains for a unit length bar divided into any number of elements
calculate_strains(number_of_elements=2 , plot=True)
calculate_strains(number_of_elements=4 , plot=True)
calculate_strains(number_of_elements=8 , plot=True)
```

3 Appendix-L2-norm error of 2 node element

```
import numpy as np
import matplotlib.pyplot as plt

# Gauss quadrature
def gauss(ngp):
    # This function now ensures that 'gp' and 'w' are NumPy arrays
    if ngp == 1:
        gp = np.array([0])
        w = np.array([2])
    elif ngp == 2:
        gp = np.array([-np.sqrt(1/3), np.sqrt(1/3)])
        w = np.array([1, 1])
    elif ngp == 3:
        gp = np.array([-np.sqrt(3/5), 0, np.sqrt(3/5)])
        w = np.array([5/9, 8/9, 5/9])
    elif ngp == 4:
        gp = np.array([-np.sqrt((3+2*np.sqrt(6/5))/7), -np.sqrt((3-2*np.sqrt(6/5))/7),
                       np.sqrt((3-2*np.sqrt(6/5))/7), np.sqrt((3+2*np.sqrt(6/5))/7)])
        w = np.array([(18-np.sqrt(30))/36, (18+np.sqrt(30))/36,
                      (18+np.sqrt(30))/36, (18-np.sqrt(30))/36])
    else:
        print("Invalid number of Gauss points specified. Only supports 1 to 4 Gauss points.")
        return None, None

    return w, gp



def calculate_error(number_of_elements, x_element_steps=1000):
    element_length = 1 / number_of_elements
    x_global_coordinate = np.linspace(0, 1, x_element_steps * number_of_elements)

    # Get Gauss points and weights
    w, gp = gauss(4)

```

```python
    total_error_square = 0

    for i in range(number_of_elements):
        # Transform Gauss points to the current element's domain
        xe1 = element_length * i
        xe2 = element_length * (i + 1)
        x_gp = 0.5 * (xe1 + xe2) + 0.5 * gp * (xe2 - xe1)

        # Calculate the estimated and real displacements at Transfered Gauss Points
        ue1 = np.power(xe1, 3)
        ue2 = np.power(xe2, 3)
        estimated_disp = ((xe2 - x_gp) * ue1 + (x_gp - xe1) * ue2) / element_length
        real_disp = np.power(x_gp, 3)

        # Calculate error for the current element using Gauss quadrature
        error_square = np.sum(w * np.power(estimated_disp - real_disp, 2)) * element_length / 2
        total_error_square += error_square

    total_error = np.sqrt(total_error_square)

    print('Error of', number_of_elements, "elements is: ", total_error)

    return total_error

# Plot function
def plot_log_error(number_of_elements):
    # Initialize arrays for errors and log lengths
    errors = np.zeros(len(number_of_elements))
    log_lengths = np.zeros(len(number_of_elements))

    # Calculate errors for each number of elements
    for i, n in enumerate(number_of_elements):
        error = calculate_error(number_of_elements = n)
        element_length = 1 / n
        errors[i] = error
        log_lengths[i] = np.log(element_length)

    log_errors = np.log(errors)
```

```

    # Calculate and print slope and intercept
    slope = (log_errors[1] - log_errors[0]) / (log_lengths[1] - log_lengths[0])
    intercept = log_errors[1] - slope * log_errors[1]
    print('Slope: ', slope, "Intercept: ", intercept)

    # Plot
    plt.figure(figsize=(10, 6))
    plt.plot(log_lengths, log_errors, marker='o', linestyle='-')
    plt.xlabel('Log of Element Length')
    plt.ylabel('Log of L2-norm Error')
    plt.title('Log-Log Plot of L2-norm Error vs Element Length')
    plt.grid(True)
    plt.show()


# Calculate and plot error
plot_log_error(np.array([2, 4, 8]))
```

4 Appendix-L2-norm error of 3 node element

```
import numpy as np
import matplotlib.pyplot as plt

# Gauss quadrature
def gauss(ngp):
    # This function now ensures that 'gp' and 'w' are NumPy arrays
    if ngp == 1:
        gp = np.array([0])
        w = np.array([2])
    elif ngp == 2:
        gp = np.array([-np.sqrt(1/3), np.sqrt(1/3)])
        w = np.array([1, 1])
    elif ngp == 3:
        gp = np.array([-np.sqrt(3/5), 0, np.sqrt(3/5)])
        w = np.array([5/9, 8/9, 5/9])
    elif ngp == 4:
        gp = np.array([-np.sqrt((3+2*np.sqrt(6/5))/7), -np.sqrt((3-2*np.sqrt(6/5))/7),
                        np.sqrt((3-2*np.sqrt(6/5))/7), np.sqrt((3+2*np.sqrt(6/5))/7)])
        w = np.array([(18-np.sqrt(30))/36, (18+np.sqrt(30))/36,
                      (18+np.sqrt(30))/36, (18-np.sqrt(30))/36])
    else:
        print("Invalid number of Gauss points specified. Only supports 1 to 4 Gauss points.")
        return None, None

    return w, gp



def calculate_error(number_of_elements, x_element_steps=1000):
    element_length = 1 / number_of_elements
    x_global_coordinate = np.linspace(0, 1, x_element_steps * number_of_elements)

    # Get Gauss points and weights
    w, gp = gauss(4)

```

```
    total_error_square = 0

    for i in range(number_of_elements):
        # Transform Gauss points to the current element's domain
        xe1 = element_length * i
        xe2 = element_length * (i + 0.5)
        xe3 = element_length * (i + 1)
        x_gp = 0.5 * (xe1 + xe3) + 0.5 * gp * (xe3 - xe1)

        # Calculate the estimated and real displacements at Transfered Gauss Points
        ue1 = np.power(xe1, 3)
        ue2 = np.power(xe2, 3)
        ue3 = np.power(xe3, 3)
        estimated_disp = ((x_gp - xe2) * (x_gp - xe3) * ue1 +
                          (x_gp - xe1) * (x_gp - xe3) * (-2) * ue2 +
                          (x_gp - xe1) * (x_gp-xe2) * ue3) * 2 / np.power(element_length, 2)
        real_disp = np.power(x_gp, 3)

        # Calculate error for the current element using Gauss quadrature
        error_square = np.sum(w * np.power(estimated_disp - real_disp, 2)) * element_length / 2
        total_error_square += error_square

    total_error = np.sqrt(total_error_square)

    print('Error of', number_of_elements, "elements is: ", total_error)

    return total_error

# Plot function
def plot_log_error(number_of_elements):
    # Initialize arrays for errors and log lengths
    errors = np.zeros(len(number_of_elements))
    log_lengths = np.zeros(len(number_of_elements))

    # Calculate errors for each number of elements
    for i, n in enumerate(number_of_elements):
        error = calculate_error(number_of_elements = n)
```

```python
        element_length = 1 / n
        errors[i] = error
        log_lengths[i] = np.log(element_length)

    log_errors = np.log(errors)

    # Calculate and print slope and intercept
    slope = (log_errors[1] - log_errors[0]) / (log_lengths[1] - log_lengths[0])
    intercept = log_errors[1] - slope * log_errors[1]
    print('Slope: ', slope, "Intercept: ", intercept)

    # Plot
    plt.figure(figsize=(10, 6))
    plt.plot(log_lengths, log_errors, marker='o', linestyle='-')
    plt.xlabel('Log of Element Length')
    plt.ylabel('Log of L2-norm Error')
    plt.title('Log-Log Plot of L2-norm Error(3-node) vs Element Length')
    plt.grid(True)
    plt.show()


# Calculate and plot error
plot_log_error(np.array([2, 4, 8]))
```