Finite Element Analysis Report

Hamidreza Owji

May 28, 2024

Abstract

This report presents the implementation and results of a finite element analysis (FEA) for various types of elements including Constant Strain Triangle (CST), Linear Strain Triangle (LST), Constant Strain Rectangle (CSR), Linear Strain Rectangle, and Constant Strain Hexahedra (CSH). The primary focus is on the CST elements, detailing the methodology, implementation, and results. Similar structures are followed for other elements.

Contents

1 Introduction **5**

2 Methodology **6**
2.1 Finite Element Method (FEM) Principles 6
2.2 Types of Elements 6
2.3 Mathematical Formulations 7
2.3.1 B Matrix 7
2.3.2 D Matrix 7

3 Constant Strain Triangle (CST) Elements **8**
3.1 Implementation 8
3.1.1 Code Structure 8
3.1.2 Key Functions 8
3.2 Full modules 10
3.2.1 Mesh_Tri3_extractor.py 10
3.2.2 FEM_CST_general.py 11
3.2.3 Problem Description 17
3.2.4 Mesh and Boundary Conditions 17
3.3 Discussion 20

4 Linear Strain Triangle (LST) Elements **21**
4.1 Implementation 21
4.2 Results 21
4.3 Discussion 21

5 Constant Strain Rectangle (CSR) Elements **22**
5.1 Implementation 22
5.2 Results 22

5.3 Discussion 22

6 Linear Strain Rectangle (LSR) Elements 23

6.1 Implementation 23

6.2 Results 23

6.3 Discussion 23

7 Constant Strain Hexahedra (CSH) Elements 24

7.1 Implementation 24

7.2 Results 24

7.3 Discussion 24

8 Conclusion 25

9 References 26

A Code 27

A.1 Full Code 27

List of Figures

3.1 Finite Element Analysis of a Rectangular Steel Plate 16
3.2 Generated mesh by Salome 18
3.3 Generated mesh by Salome 20

List of Tables

Chapter 1

Introduction

Finite Element Analysis (FEA) is a powerful numerical technique used to solve complex structural, fluid, and thermal problems by discretizing the domain into smaller elements. This report focuses on the implementation of FEA for different types of elements, with a detailed examination of Constant Strain Triangle (CST) elements. The objective is to understand the behavior of materials under various conditions and compare the results for different element types.

Chapter 2

Methodology

2.1 Finite Element Method (FEM) Principles

The Finite Element Method (FEM) involves breaking down a complex domain into smaller, simpler parts called elements. Each element is defined by nodes, and the relationships between nodal displacements and element strains are captured by matrices such as the B matrix and the D matrix.

2.2 Types of Elements

This report considers the following types of elements:

- Constant Strain Triangle (CST)
- Linear Strain Triangle (LST)
- Constant Strain Rectangle (CSR)
- Linear Strain Rectangle (LSR)
- Constant Strain Hexahedra (CSH)

2.3 Mathematical Formulations

2.3.1 B Matrix

The B matrix bridges the gap between the physical displacements of the nodes and the strains in the material. It is derived from the shape functions of the element.

2.3.2 D Matrix

The D matrix, or material matrix, relates the stress and strain in the material, defined by the material properties such as Young's modulus and Poisson's ratio.

Chapter 3

Constant Strain Triangle (CST) Elements

3.1 Implementation

3.1.1 Code Structure

The code for CST elements is structured into several modules, each responsible for different aspects of the FEA process. The primary modules are:

- FEM_CST_general
- Mesh_Tri3_extractor
- FEM_CST_plotting

3.1.2 Key Functions

Computing the B Matrix

```python
def compute_area_and_B_matrix(coords):
        """Compute the area and B matrix for a CST element.
            """
        A = 0.5 * abs(coords[0][0]*(coords[1][1]-coords
            [2][1]) +
        coords[1][0]*(coords[2][1]-coords[0][1]) +
        coords[2][0]*(coords[0][1]-coords[1][1]))

        b = np.array([coords[1][1]-coords[2][1], coords
            [2][1]-coords[0][1], coords[0][1]-coords[1][1]])
        c = np.array([coords[2][0]-coords[1][0], coords
            [0][0]-coords[2][0], coords[1][0]-coords[0][0]])

        B = np.zeros((3, 6))
        B[0, ::2] = b
        B[1, 1::2] = c
        B[2, ::2] = c
        B[2, 1::2] = b
        B /= (2*A)

        return A, B
```

Computing the D Matrix

```
def compute_D_matrix(E, nu):
        """Compute the D matrix (material matrix)."""
        return E / (1-nu**2) * np.array([[1, nu, 0], [nu, 1,
            0], [0, 0, (1-nu)/2]])
```

Computing the Stiffness Matrix

```
def compute_stiffness_matrix(coords, D):
        """Compute the stiffness matrix for a CST element."""
        A, B = compute_area_and_B_matrix(coords)
        return A * np.dot(B.T, np.dot(D, B))
```

Assembling the Global Stiffness Matrix

```
def assemble_global_stiffness(elements, D):
        """Assemble the global stiffness matrix."""
        num_nodes = max([node for elem in elements for node
            in elem['nodes']])
        K_global = np.zeros((2*num_nodes, 2*num_nodes))

        for elem in elements:
        k = compute_stiffness_matrix(elem['coords'], D)
        for i in range(3):
        for j in range(3):
        m, n = elem['nodes'][i], elem['nodes'][j]
        K_global[2*m-2:2*m, 2*n-2:2*n] += k[2*i:2*i+2, 2*j:2*j+2]

        return K_global
```

Computing Global Forces

```
def compute_global_forces(K_global, U):
        """Compute the global forces from the global stiffness matrix and nodal displacements."""
        return np.dot(K_global, U)
```

3.2 Full modules

3.2.1 Mesh_Tri3_extractor.py

This module is responsible getting mesh information from MED file

```
import h5py
import numpy as np

def divide_list_into_sublists(lst, n):
        for i in range(0, len(lst), n):
                yield lst[i:i + n]

def extract_coordinates(lst, number_of_coordinates):
        for i in range(0, len(lst), number_of_coordinates):
                yield lst[i:i + number_of_coordinates]
def generate_elements(node_coordinates, element_node_connectivity):
        elements = []
        for element in element_node_connectivity:
                element_dict = {
                        'nodes': element,
                        'coords': np.array([node_coordinates[node - 1] for node in element])
                }
                elements.append(element_dict)
        return elements

def read_mesh_data(file_name):
        with h5py.File(file_name, 'r') as file:
                # Read coordinate data
                coo_dataset = file['ENS_MAA/Mesh_1/-0000000000000000001-0000000000000000001/NOE/COO']
                coo_data = coo_dataset[:]
                num_nodes = len(coo_data) // 2
                subcoord = list(extract_coordinates(coo_data, num_nodes))
                node_coordinates = [group for group in zip(*subcoord)]

                # Read TRIA3/NOD dataset for TRIA3 elements
                tr3_dataset = file['ENS_MAA/Mesh_1/-0000000000000000001-0000000000000000001/MAI/TR3/NOD']
                tr3_data = tr3_dataset[:]
                num_triangles = len(tr3_data) // 3
                sublists = list(divide_list_into_sublists(tr3_data, num_triangles))
                element_node_connectivity = [group for group in zip(*sublists)]

        return node_coordinates, element_node_connectivity
```

3.2.2 FEM_CST_general.py

This module is responsible getting mesh information from MED file

```
import numpy as np
from Mesh_Tri3_extractor import generate_elements, read_mesh_data
from FEM_CST_plotting import (plot_mesh, plot_displacements, plot_mesh_with_boundary_conditions, plot_loads, plot_mesh_with_loads)

def compute_area_and_B_matrix(coords):
        """Compute the area and B matrix for a CST element."""
        A = 0.5 * abs(coords[0][0]*(coords[1][1]-coords[2][1]) +
        coords[1][0]*(coords[2][1]-coords[0][1]) +
        coords[2][0]*(coords[0][1]-coords[1][1]))

        b = np.array([coords[1][1]-coords[2][1], coords[2][1]-coords[0][1], coords[0][1]-coords[1][1]])
        c = np.array([coords[2][0]-coords[1][0], coords[0][0]-coords[2][0], coords[1][0]-coords[0][0]])

        B = np.zeros((3, 6))
        B[0, ::2] = b
        B[1, 1::2] = c
        B[2, ::2] = c
        B[2, 1::2] = b
        B /= (2*A)

        return A, B

def compute_D_matrix(E, nu):
        """Compute the D matrix (material matrix)."""
        return E / (1-nu**2) * np.array([[1, nu, 0], [nu, 1, 0], [0, 0, (1-nu)/2]])

def compute_stiffness_matrix(coords, D):
        """Compute the stiffness matrix for a CST element."""
        A, B = compute_area_and_B_matrix(coords)
        return A * np.dot(B.T, np.dot(D, B))

def assemble_global_stiffness(elements, D):
        """Assemble the global stiffness matrix."""
        num_nodes = max([node for elem in elements for node in elem['nodes']])
        K_global = np.zeros((2*num_nodes, 2*num_nodes))

        for elem in elements:
        k = compute_stiffness_matrix(elem['coords'], D)
        for i in range(3):
        for j in range(3):
        m, n = elem['nodes'][i], elem['nodes'][j]
        K_global[2*m-2:2*m, 2*n-2:2*n] += k[2*i:2*i+2, 2*j:2*j+2]

        return K_global

def compute_global_forces(K_global, U):
        """Compute the global forces from the global stiffness matrix and nodal displacements."""
        return np.dot(K_global, U)

def compute_element_forces(element, U, D):
        """Compute the element forces from the element
```

```
            stiffness matrix and nodal displacements."""
        k = compute_stiffness_matrix(element['coords'], D)
        u_element = np.array([U[2*node-2:2*node] for node in
            element['nodes']]).flatten()
        return np.dot(k, u_element)

def compute_element_stress(element, U, D):
        """Compute the stress within an element."""
        u_element = np.array([U[2*node-2:2*node] for node in
            element['nodes']]).flatten()
        A, B = compute_area_and_B_matrix(element['coords'])
        epsilon = np.dot(B, u_element)
        sigma = np.dot(D, epsilon)
        return sigma

def compute_principal_stresses_and_angles(sigma):
        """Compute the principal stresses from the stress
            vector."""
        sigma_x, sigma_y, tau_xy = sigma
        sigma_avg = 0.5 * (sigma_x + sigma_y)
        R = np.sqrt(((sigma_x - sigma_y) * 0.5)**2 + tau_xy
            **2)
        sigma_1 = sigma_avg + R
        sigma_2 = sigma_avg - R
        theta_rad = 0.5 * np.arctan2(2 * tau_xy, sigma_x -
            sigma_y)
        theta_deg = np.degrees(theta_rad)
        return sigma_1, sigma_2, theta_deg

# Example usage
E = 2.1e6  # Modulus of elasticity in Pa (for steel)
nu = 0.3  # Poisson's ratio (for steel)
D = compute_D_matrix(E, nu)  # Compute D matrix once and
    reuse

node_coordinates, element_node_connectivity = read_mesh_data(
    'Mesh_1.med')
elements = generate_elements(node_coordinates,
    element_node_connectivity)

K_global = assemble_global_stiffness(elements, D)
# print(K_global)

# Identify boundary nodes and apply conditions. This gives us
     index not node number
```

```python
left_boundary_nodes = [node + 1 for node, coord in enumerate(
    node_coordinates) if coord[0] == 0]
# print('left boundary: ', left_boundary_nodes)

# Initialize the displacement vector with zeros (as a
    starting assumption)
U = np.zeros(2 * len(node_coordinates))

# Identify nodes with x=140 and apply a load of 1000 in the x
     direction
F_external = np.zeros_like(U)

# Node index where the load will be applied (Python uses 0-
    based indexing, so node 3 is indexed as 2)
node_index = 3 - 1 # Adjust for 0-based indexing by
    subtracting 1

# Apply a load of 1000 in the x direction to node 3
F_external[2*node_index + 1] = -1000  # Apply load in x
    direction. For y direction, use 2*node_index + 1

# print('f external: ', F_external)

fixed_dof = []
for node in left_boundary_nodes:  # assuming
    left_boundary_nodes contain fixed nodes
        fixed_dof.extend([2*(node -1), 2*(node -1) +1])

K_reduced = np.delete(K_global, fixed_dof, axis=0)  # Remove
    rows
K_reduced = np.delete(K_reduced, fixed_dof, axis=1)  # Remove
     columns
F_reduced = np.delete(F_external, fixed_dof)
# print('f reduced: ', F_reduced)
U_reduced = np.linalg.solve(K_reduced, F_reduced)
U_full = np.zeros_like(U)
free_dof = set(range(len(U))) - set(fixed_dof)
free_dof = list(free_dof)
U_full[free_dof] = U_reduced


F_global_calculated = compute_global_forces(K_global, U_full)

# Uncomment the following lines for debugging:
# print("Global Forces:")
# print(F_global_calculated)

# Uncomment the following blocks for debugging:
# for elem in elements:
#     F_element = compute_element_forces(elem, U_full, D)
#     print(f"Element Forces for nodes {elem['nodes']}:")
#     print(F_element)

# for elem in elements:
#     sigma_element = compute_element_stress(elem, U_full, D)
#     print(f"Element Stress for nodes {elem['nodes']}:")
#     print(sigma_element)

# for elem in elements:
#     sigma_element = compute_element_stress(elem, U_full, D)
#     sigma_1, sigma_2, theta_deg =
    compute_principal_stresses_and_angles(sigma_element)
#     print(f"Principal Stresses for element with nodes {elem
    ['nodes']}:")
#     print(f"Maximum (sigma_1): {sigma_1}")
#     print(f"Minimum (sigma_2): {sigma_2}")
#     print(f"Angle of Maximum Stress (degrees): {theta_deg
    }")

# for i, coord in enumerate(node_coordinates):
#     x, y = coord  # Unpack the x and y coordinates of the
    node
#     dx = U_full[2*i]   # x displacement of node i
#     dy = U_full[2*i+1] # y displacement of node i
#     print(f"Node {i+1}: x = {x:.6f}, y = {y:.6f}, dx = {dx
    :.6f}, dy = {dy:.6f}")

# print('max dis= ', max(U_full))
plot_displacements(node_coordinates, U_full, 'Nodal
    Displacements', scale_factor=1000)
# print(U_full)
# plot_mesh(elements, node_coordinates)
# plot_displacements(node_coordinates, U, 'stress')

# plot_mesh_with_boundary_conditions(elements,
    node_coordinates, left_boundary_nodes)
# plot_loads(node_coordinates, F_external, 1)
# plot_mesh_with_loads(elements, node_coordinates,
    left_boundary_nodes, F_external)

```

```python
# Assuming U_full is already defined and contains the
    displacements for each node
displacement_magnitudes = np.sqrt(U_full[::2]**2 + U_full
    [1::2]**2)

max_disp_node_index = np.argmax(displacement_magnitudes)

max_disp_magnitude = displacement_magnitudes[
    max_disp_node_index]

max_disp_node_number = max_disp_node_index + 1

max_disp_x = U_full[2*max_disp_node_index]
max_disp_y = U_full[2*max_disp_node_index + 1]

# Print the information
print(f"Node with Maximum Displacement: Node {
    max_disp_node_number}")
print(f"Displacement in X: {max_disp_x:.6f}")
print(f"Displacement in Y: {max_disp_y:.6f}")
print(f"Total Displacement Magnitude: {max_disp_magnitude:.6f
    }")
```

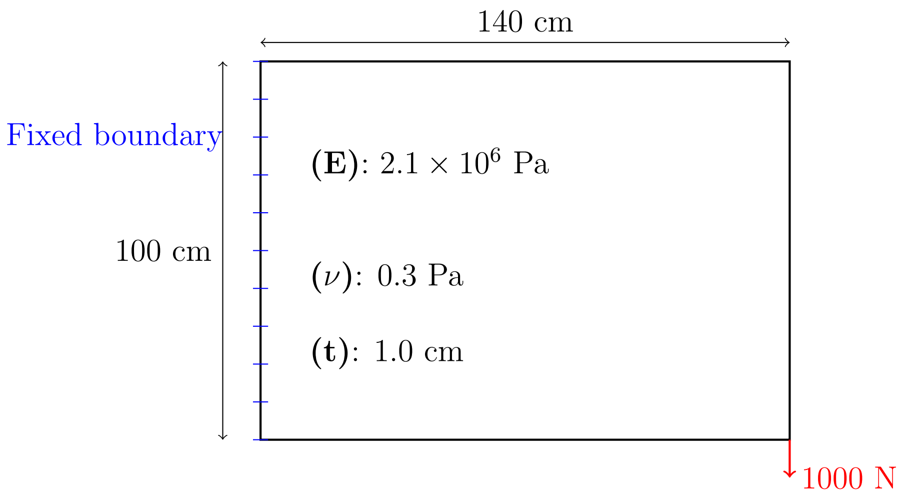

Figure 3.1: Finite Element Analysis of a Rectangular Steel Plate

3.2.3 Problem Description

In this example, we analyze a rectangular steel plate with dimensions 100 cm by 140 cm. The plate has the following properties and loading conditions:

- **Modulus of Elasticity (E)**: 2.1×10^6 Pa (typical for steel)
- **Poisson's Ratio (ν)**: 0.3 (typical for steel)
- **External Load**: A load of 1000 N applied in the x-direction at a node located at $x = 140$ cm and $y = 0$ cm.

The material matrix D is computed using the given modulus of elasticity and Poisson's ratio. The mesh data, including node coordinates and element connectivity, is read from a file named `Mesh_1.med`. Using this mesh data, elements are generated and the global stiffness matrix is assembled.

Boundary conditions are applied to nodes located at $x = 0$ cm (left boundary), and the external load is applied to the specified node.

3.2.4 Mesh and Boundary Conditions

The following Python code demonstrates the steps for setting up and solving the FEA problem:

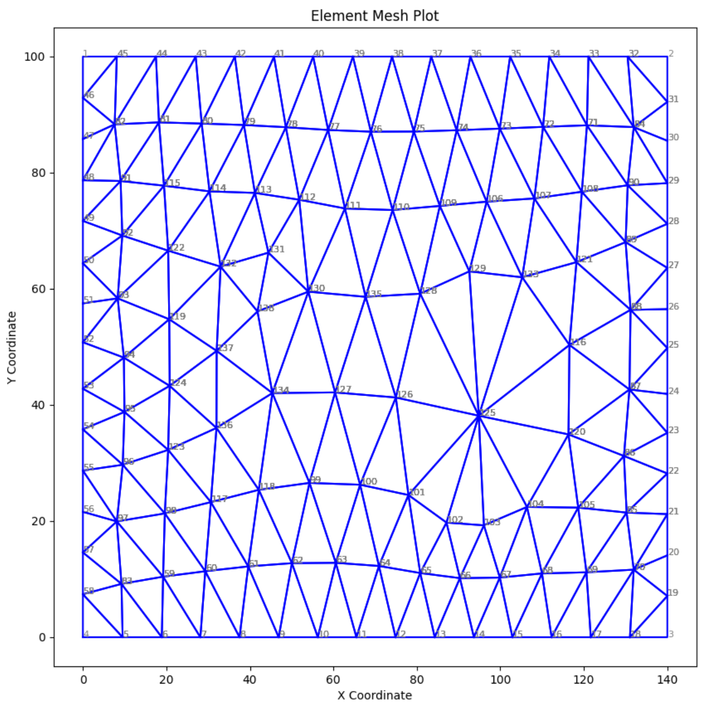

Figure 3.2: Generated mesh by Salome

Listing 3.1: Python code for setting up the FEA problem

```
E = 2.1e6  # Modulus of elasticity in Pa (for steel)
nu = 0.3  # Poisson's ratio (for steel)
D = compute_D_matrix(E, nu)  # Compute D matrix once
    and reuse

node_coordinates, element_node_connectivity =
    read_mesh_data('Mesh_1.med')
elements = generate_elements(node_coordinates,
    element_node_connectivity)

K_global = assemble_global_stiffness(elements, D)

# Identify boundary nodes and apply conditions. This
    gives us index not node number
```

```
        left_boundary_nodes = [node + 1 for node, coord in enumerate(node_coordinates) if coord[0] == 0]

        # Initialize the displacement vector with zeros (as a starting assumption)
        U = np.zeros(2 * len(node_coordinates))

        # Identify nodes with x=140 and apply a load of 1000 in the x direction
        F_external = np.zeros_like(U)

        # Node index where the load will be applied (Python uses 0-based indexing, so node 3 is indexed as 2)
        node_index = 3 - 1 # Adjust for 0-based indexing by subtracting 1

        # Apply a load of 1000 in the x direction to node 3
        F_external[2*node_index + 1] = -1000  # Apply load in x direction. For y direction, use 2*node_index + 1
```

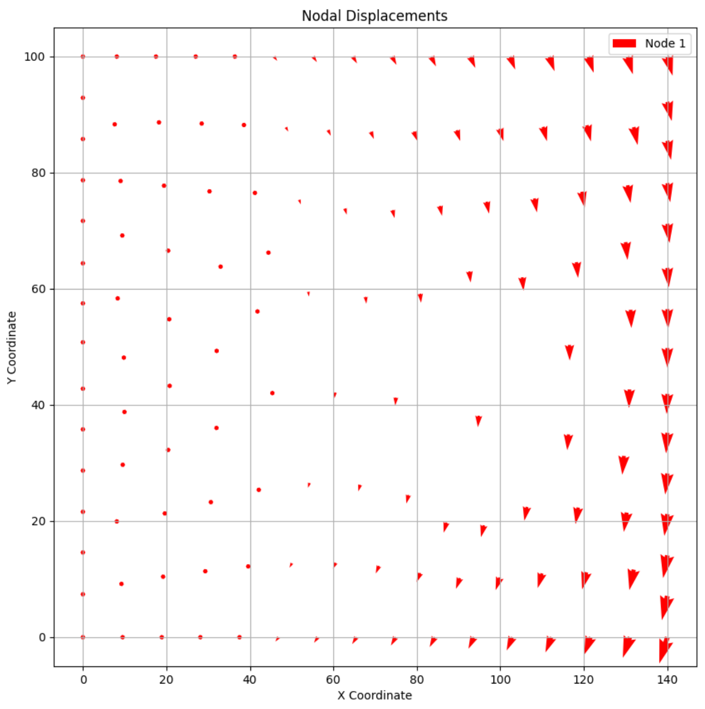

Figure 3.3: Generated mesh by Salome

3.2.5 Results

Node with Maximum Displacement	Node 3
Displacement in X	-0.003867
Displacement in Y	-0.009054
Total Displacement Magnitude	0.009845

3.3 Discussion

The results for CST elements show that...

Chapter 4

Linear Strain Triangle (LST) Elements

4.1 Implementation

4.2 Results

4.3 Discussion

Chapter 5

Constant Strain Rectangle (CSR) Elements

5.1 Implementation

5.2 Results

5.3 Discussion

Chapter 6

Linear Strain Rectangle (LSR) Elements

6.1 Implementation

6.2 Results

6.3 Discussion

Chapter 7

Constant Strain Hexahedra (CSH) Elements

7.1 Implementation

7.2 Results

7.3 Discussion

Chapter 8

Conclusion

Chapter 9

References

Appendix A

Code

A.1 Full Code