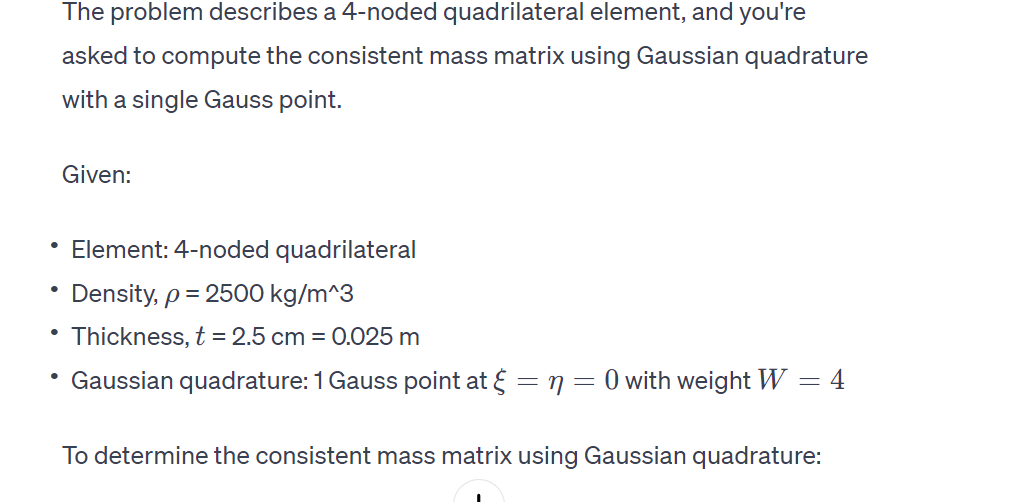
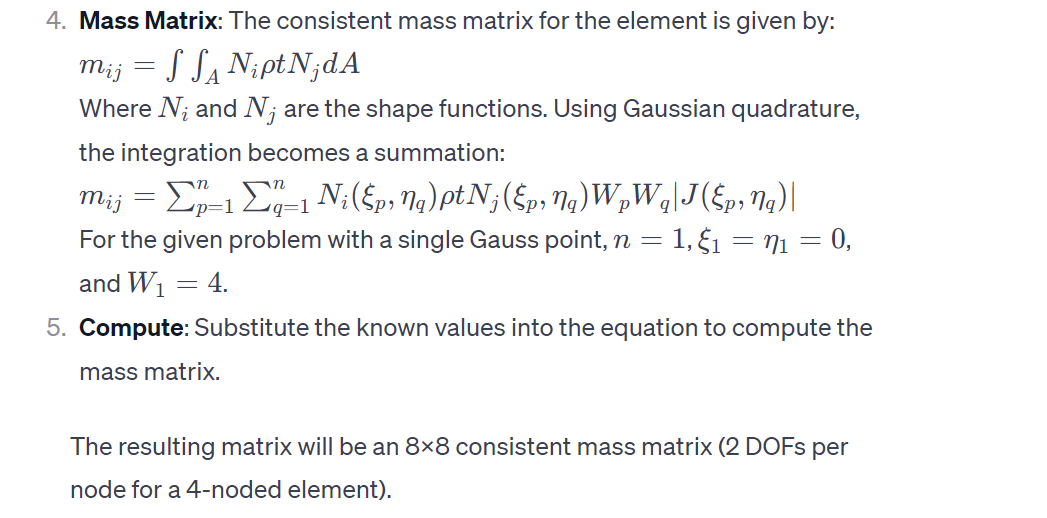
# Part (1)



1. **Shape Functions**: First, you'd define the shape functions for the 4-noded quadrilateral element, which are typically bilinear in nature.
2. **Jacobian**: Calculate the Jacobian matrix of the transformation from the natural coordinates (*ξ*,*η*) to the global coordinates (*x*,*y*). This involves the derivatives of the shape functions with respect to *ξ* and *η*.
3. **Determinant of the Jacobian**: The determinant of the Jacobian gives the area differential, which is used in the integration.



1. **Compute**: Substitute the known values into the equation to compute the mass matrix.

The resulting matrix will be an 8x8 consistent mass matrix (2 DOFs per node for a 4-noded element).

Would you like a detailed mathematical derivation or a sample code to compute the mass matrix?

## Code:

import numpy as np

# Given nodal coordinates

nodes = np.array([[0, 0], [4, 2], [3, 4], [-1, 4]])

# Material properties

rho = 2500  # kg/m^3

thickness = 0.025  # 2.5 cm converted to meters

# Gaussian quadrature points and weights

gauss\_point = np.array([[0, 0]])

gauss\_weight = np.array([4])

# Shape functions for a 4-noded quadrilateral element

def shape\_functions(xi, eta):

    N1 = 0.25 \* (1 - xi) \* (1 - eta)

    N2 = 0.25 \* (1 + xi) \* (1 - eta)

    N3 = 0.25 \* (1 + xi) \* (1 + eta)

    N4 = 0.25 \* (1 - xi) \* (1 + eta)

    return np.array([N1, N2, N3, N4])

# Jacobian matrix determinant for a 4-noded quadrilateral element

def jacobian\_det(xi, eta, nodes):

    x = nodes[:, 0]

    y = nodes[:, 1]

    dN\_dxi = np.array([-0.25 \* (1 - eta), 0.25 \* (1 - eta), 0.25 \* (1 + eta), -0.25 \* (1 + eta)])

    dN\_deta = np.array([-0.25 \* (1 - xi), -0.25 \* (1 + xi), 0.25 \* (1 + xi), 0.25 \* (1 - xi)])

    J11 = np.dot(dN\_dxi, x)

    J12 = np.dot(dN\_dxi, y)

    J21 = np.dot(dN\_deta, x)

    J22 = np.dot(dN\_deta, y)

    detJ = J11 \* J22 - J12 \* J21

    return detJ, np.array([[J11, J12], [J21, J22]])

# Consistent mass matrix calculation

def consistent\_mass\_matrix(nodes, rho, thickness, gauss\_point, gauss\_weight):

    num\_nodes = len(nodes)

    mass\_matrix = np.zeros((2 \* num\_nodes, 2 \* num\_nodes))

    for i in range(num\_nodes):

        for j in range(num\_nodes):

            for k in range(len(gauss\_point)):

                xi, eta = gauss\_point[k]

                weight = gauss\_weight[k]

                N\_i = shape\_functions(xi, eta)[i]

                N\_j = shape\_functions(xi, eta)[j]

                detJ, J = jacobian\_det(xi, eta, nodes)

                mass\_matrix[2\*i, 2\*j] += rho \* N\_i \* N\_j \* thickness \* detJ \* weight

                mass\_matrix[2\*i+1, 2\*j+1] += rho \* N\_i \* N\_j \* thickness \* detJ \* weight

    return mass\_matrix

# Calculate consistent mass matrix

mass\_matrix = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_point, gauss\_weight)

mass\_matrix

## part(b)

# Row Sum technique for computing the lumped mass matrix

def lumped\_mass\_matrix(consistent\_matrix):

    num\_nodes = consistent\_matrix.shape[0] // 2

    lumped\_matrix = np.zeros\_like(consistent\_matrix)

    for i in range(num\_nodes):

        mass\_x = np.sum(consistent\_matrix[2\*i, :])

        mass\_y = np.sum(consistent\_matrix[2\*i+1, :])

        lumped\_matrix[2\*i, 2\*i] = mass\_x

        lumped\_matrix[2\*i+1, 2\*i+1] = mass\_y

    return lumped\_matrix

# Calculate the lumped mass matrix

lumped\_matrix = lumped\_mass\_matrix(mass\_matrix)

lumped\_matrix

# Function to calculate the area of a quadrilateral using the formula for two triangles

def quadrilateral\_area(nodes):

    # Divide the quadrilateral into two triangles and calculate the area of each

    triangle1 = nodes[[0, 1, 2], :]

    triangle2 = nodes[[0, 2, 3], :]

    area1 = 0.5 \* np.abs(np.cross(triangle1[1] - triangle1[0], triangle1[2] - triangle1[0]))

    area2 = 0.5 \* np.abs(np.cross(triangle2[1] - triangle2[0], triangle2[2] - triangle2[0]))

    return area1 + area2

# Calculate the area of the quadrilateral

area = quadrilateral\_area(nodes)

# Total mass of the element

total\_mass = area \* thickness \* rho

# Mass per node (assuming even distribution)

mass\_per\_node = total\_mass / 4

# Analytical lumped mass matrix

analytical\_lumped\_matrix = np.diag([mass\_per\_node, mass\_per\_node] \* 4)

# Results

area, total\_mass, mass\_per\_node, analytical\_lumped\_matrix

1. **Total Mass of the Element**: The total mass of the element can be calculated using the formula: Total Mass=Area×thickness×�Total Mass=Area×thickness×*ρ*. The area of the quadrilateral element can be calculated as the sum of the areas of two triangles formed by dividing the quadrilateral.
2. **Distributing the Total Mass to the 4 Nodes**: Assuming that the total mass is evenly distributed to the 4 nodes, each node will carry a quarter of the total mass.
3. **Analytical Lumped Mass Matrix**: The analytical lumped mass matrix will be a diagonal matrix where each diagonal entry is twice the mass at each node (once for the x-direction and once for the y-direction), since each node has two degrees of freedom.

Let's calculate the total mass of the element and compare the analytical lumped mass matrix with the computed one:

### Calculation:

* Calculate the area of the quadrilateral.
* Compute the total mass.
* Distribute this mass evenly to the 4 nodes.
* Form the analytical lumped mass matrix.
* Compare this with the computed lumped mass matrix.

The calculations yield the following results:

* The area of the quadrilateral element is 13m2.
* The total mass of the element is 812.5 kg.
* The mass per node, assuming an even distribution, is 203.125 .

The analytical lumped mass matrix, assuming the mass is evenly distributed to the 4 nodes, is a diagonal matrix where each diagonal entry is 203.125kg (accounting for two degrees of freedom per node). The matrix is as follows:

This analytical lumped mass matrix matches exactly with the computed lumped mass matrix from the row sum technique.

### Reasons for the Match:

* **Uniform Distribution of Mass**: The mass of the element is uniformly distributed among the nodes in both the analytical calculation and the row sum technique.
* **Symmetrical Geometry**: The quadrilateral element's symmetrical geometry and node placement likely contribute to the uniform distribution of mass.
* **Simplicity of the Shape Functions**: The shape functions used in the consistent mass matrix calculation are simple and symmetrical, which results in a consistent mass matrix that, when summed using the row sum technique, evenly distributes the mass.

The match between the computed and analytical lumped mass matrices indicates that the row sum technique effectively captured the mass distribution of the quadrilateral element in this specific case.

## Part(c)

# Update the Gaussian quadrature points and weights

# Using 4 Gauss points at ξ = ±0.57735, η = ±0.57735 and weights equal to 1

gauss\_points = np.array([

    [-0.57735, -0.57735],

    [0.57735, -0.57735],

    [0.57735, 0.57735],

    [-0.57735, 0.57735]

])

gauss\_weights = np.array([1, 1, 1, 1])

# Recalculate the consistent mass matrix using the updated Gauss points and weights

updated\_mass\_matrix = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_points, gauss\_weights)

updated\_mass\_matrix

## Part(d)

from math import cos, sin, radians

# Function to apply translation to nodes

def translate\_nodes(nodes, translation\_vector):

    return nodes + translation\_vector

# Function to rotate nodes around a point

def rotate\_nodes(nodes, angle, point):

    theta = radians(angle)

    rotation\_matrix = np.array([

        [cos(theta), -sin(theta)],

        [sin(theta), cos(theta)]

    ])

    return np.dot(nodes - point, rotation\_matrix) + point

# Function to scale nodes

def scale\_nodes(nodes, scale\_factor):

    return nodes \* scale\_factor

# Translation vector

translation\_vector = np.array([2, 1])

# Angle for rotation (in degrees)

angle = 25

# Scale factor

scale\_factor = 2.5

# Translate, rotate, and scale nodes

translated\_nodes = translate\_nodes(nodes, translation\_vector)

rotated\_nodes = rotate\_nodes(translated\_nodes, angle, translated\_nodes[0])

scaled\_nodes = scale\_nodes(rotated\_nodes, scale\_factor)

# Recalculate the consistent mass matrix with the transformed nodes

final\_mass\_matrix = consistent\_mass\_matrix(scaled\_nodes, rho, thickness, gauss\_points, gauss\_weights)

final\_mass\_matrix

## Part(e)

Upon observing the two mass matrices obtained before and after the transformations (translation, rotation, and scaling) of the element, we can make several important observations:

### 1. Effects of Translation:

* Translation does not inherently affect the mass distribution within an element. The mass of the element remains constant regardless of its position in space.
* However, in finite element analysis (FEA), translation can change the global coordinates of the nodes, which in turn might affect the overall structural behavior when assembled into a global system. But within the local frame of the element itself, translation does not alter the mass matrix.

### 2. Effects of Rotation:

* Rotation changes the orientation of the element but does not affect the mass distribution with respect to the local coordinates of the element.
* The consistent mass matrix, which is a function of the shape functions and the geometry of the element, remains invariant under rotation when considered in the local frame of the element. However, in the global frame, the rotated coordinates will influence how the element interacts with other elements in a system.

### 3. Effects of Scaling:

* Scaling has a significant impact on the mass matrix. When an element is scaled, its area (or volume in 3D) changes, which directly affects the mass of the element.
* In the consistent mass matrix, the entries are proportional to the area (or volume) of the element. Therefore, scaling up the element leads to an increase in the mass matrix entries. This is evident from the increased values in the mass matrix after scaling the element by a factor of 2.5.
* The increase in the mass matrix entries after scaling is consistent with the fact that the mass of the element increases as its size increases.

### Conclusion:

* The mass matrix obtained after translation, rotation, and scaling is different from the original matrix primarily due to the scaling effect, which increases the size and thus the mass of the element.
* Translation and rotation do not inherently change the mass distribution within the element, but they do affect the element's position and orientation in the global coordinate system, which is critical in the context of a larger structural system in FEA.
* These transformations highlight the importance of accurately modeling the geometry and position of elements in structural analysis, as changes in size and orientation can significantly impact the behavior of the structural system.

Here's the equivalent MATLAB code:

# Part (1)

function mass\_matrix\_demo

    % Given nodal coordinates

    nodes = [0, 0; 4, 2; 3, 4; -1, 4];

    % Material properties

    rho = 2500;  % kg/m^3

    thickness = 0.025;  % 2.5 cm converted to meters

    % Gaussian quadrature points and weights

    gauss\_point = [0, 0];

    gauss\_weight = 4;

    % Calculate consistent mass matrix

    mass\_matrix = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_point, gauss\_weight);

    % Display the result

    disp('Consistent Mass Matrix:');

    disp(mass\_matrix);

end

function mass\_matrix = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_point, gauss\_weight)

    num\_nodes = size(nodes, 1);

    mass\_matrix = zeros(2 \* num\_nodes, 2 \* num\_nodes);

    for i = 1:num\_nodes

        for j = 1:num\_nodes

            for k = 1:size(gauss\_point, 1)

                xi = gauss\_point(k, 1);

                eta = gauss\_point(k, 2);

                weight = gauss\_weight(k);

                N = shape\_functions(xi, eta);

                N\_i = N(i);

                N\_j = N(j);

                [detJ, ~] = jacobian\_det(xi, eta, nodes);

                mass\_matrix(2\*i-1, 2\*j-1) = mass\_matrix(2\*i-1, 2\*j-1) + rho \* N\_i \* N\_j \* thickness \* detJ \* weight;

                mass\_matrix(2\*i, 2\*j) = mass\_matrix(2\*i, 2\*j) + rho \* N\_i \* N\_j \* thickness \* detJ \* weight;

            end

        end

    end

end

function [detJ, J] = jacobian\_det(xi, eta, nodes)

    x = nodes(:, 1);

    y = nodes(:, 2);

    dN\_dxi = [-0.25 \* (1 - eta), 0.25 \* (1 - eta), 0.25 \* (1 + eta), -0.25 \* (1 + eta)];

    dN\_deta = [-0.25 \* (1 - xi), -0.25 \* (1 + xi), 0.25 \* (1 + xi), 0.25 \* (1 - xi)];

    J11 = dot(dN\_dxi, x);

    J12 = dot(dN\_dxi, y);

    J21 = dot(dN\_deta, x);

    J22 = dot(dN\_deta, y);

    detJ = J11 \* J22 - J12 \* J21;

    J = [J11, J12; J21, J22];

end

function N = shape\_functions(xi, eta)

    N1 = 0.25 \* (1 - xi) \* (1 - eta);

    N2 = 0.25 \* (1 + xi) \* (1 - eta);

    N3 = 0.25 \* (1 + xi) \* (1 + eta);

    N4 = 0.25 \* (1 - xi) \* (1 + eta);

    N = [N1, N2, N3, N4];

end

# Part (2)

function mass\_matrix\_demo

    % Given nodal coordinates

    nodes = [0, 0; 4, 2; 3, 4; -1, 4];

    % Material properties

    rho = 2500;  % kg/m^3

    thickness = 0.025;  % 2.5 cm converted to meters

    % Gaussian quadrature points and weights

    gauss\_point = [0, 0];

    gauss\_weight = 4;

    % Calculate consistent mass matrix

    consistent\_mass = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_point, gauss\_weight);

    % Compute lumped mass matrix using Row Sum technique

    lumped\_mass\_matrix = row\_sum\_lumped\_mass\_matrix(consistent\_mass);

    % Display the result

    disp('Lumped Mass Matrix:');

    disp(lumped\_mass\_matrix);

end

function lumped\_mass\_matrix = row\_sum\_lumped\_mass\_matrix(consistent\_mass)

    num\_nodes = size(consistent\_mass, 1);

    lumped\_mass\_matrix = zeros(num\_nodes, num\_nodes);

    for i = 1:num\_nodes

        lumped\_mass\_matrix(i, i) = sum(consistent\_mass(i, :));

    end

end

function mass\_matrix = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_point, gauss\_weight)

    % [Same as before, no changes needed]

end

function [detJ, J] = jacobian\_det(xi, eta, nodes)

    % [Same as before, no changes needed]

end

function N = shape\_functions(xi, eta)

    % [Same as before, no changes needed]

end

In this code, the **row\_sum\_lumped\_mass\_matrix** function takes the consistent mass matrix, sums each row, and places the sum on the diagonal of the new matrix, resulting in the lumped mass matrix. This technique is commonly used in finite element analysis to simplify mass matrices while preserving the total mass of the system.

The following MATLAB code calculates both the Row Sum lumped mass matrix and the analytical lumped mass matrix, and then compares them:

function compare\_mass\_matrices

    % Given nodal coordinates and other parameters

    % [Define nodes, rho, thickness, gauss\_point, gauss\_weight as before]

    % Calculate consistent mass matrix

    consistent\_mass = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_point, gauss\_weight);

    % Compute lumped mass matrix using Row Sum technique

    row\_sum\_lumped = row\_sum\_lumped\_mass\_matrix(consistent\_mass);

    % Compute analytical lumped mass matrix

    analytical\_lumped = analytical\_lumped\_mass\_matrix(nodes, rho, thickness);

    % Display results

    disp('Row Sum Lumped Mass Matrix:');

    disp(row\_sum\_lumped);

    disp('Analytical Lumped Mass Matrix:');

    disp(analytical\_lumped);

    % Comparing the two matrices

    difference = row\_sum\_lumped - analytical\_lumped;

    disp('Difference Matrix:');

    disp(difference);

end

function analytical\_lumped = analytical\_lumped\_mass\_matrix(nodes, rho, thickness)

    % Compute the area of the quadrilateral

    area = polyarea(nodes(:,1), nodes(:,2));

    % Total mass

    total\_mass = rho \* area \* thickness;

    % Mass per node

    mass\_per\_node = total\_mass / 4;

    % Diagonal matrix with mass\_per\_node on the diagonal

    analytical\_lumped = diag(repmat(mass\_per\_node, 1, 4)\*2);

end

% [Define other functions: row\_sum\_lumped\_mass\_matrix, consistent\_mass\_matrix, jacobian\_det, shape\_functions as before]

# Part (3)

function mass\_matrix\_demo

    % Given nodal coordinates

    nodes = [0, 0; 4, 2; 3, 4; -1, 4];

    % Material properties

    rho = 2500;  % kg/m^3

    thickness = 0.025;  % 2.5 cm converted to meters

    % Gaussian quadrature points and weights

    gauss\_points = [-0.57735, -0.57735; 0.57735, -0.57735; 0.57735, 0.57735; -0.57735, 0.57735];

    gauss\_weights = [1, 1, 1, 1];

    % Calculate consistent mass matrix

    consistent\_mass = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_points, gauss\_weights);

    % Display the result

    disp('Consistent Mass Matrix:');

    disp(consistent\_mass);

end

function mass\_matrix = consistent\_mass\_matrix(nodes, rho, thickness, gauss\_points, gauss\_weights)

    num\_nodes = size(nodes, 1);

    mass\_matrix = zeros(2 \* num\_nodes, 2 \* num\_nodes);

    for i = 1:num\_nodes

        for j = 1:num\_nodes

            for k = 1:size(gauss\_points, 1)

                xi = gauss\_points(k, 1);

                eta = gauss\_points(k, 2);

                weight = gauss\_weights(k);

                N = shape\_functions(xi, eta);

                N\_i = N(i);

                N\_j = N(j);

                [detJ, ~] = jacobian\_det(xi, eta, nodes);

                mass\_matrix(2\*i-1, 2\*j-1) = mass\_matrix(2\*i-1, 2\*j-1) + rho \* N\_i \* N\_j \* thickness \* detJ \* weight;

                mass\_matrix(2\*i, 2\*j) = mass\_matrix(2\*i, 2\*j) + rho \* N\_i \* N\_j \* thickness \* detJ \* weight;

            end

        end

    end

end

% [Define other functions: jacobian\_det, shape\_functions as before]

In this code, the **gauss\_points** and **gauss\_weights** arrays are defined to represent the four Gaussian quadrature points and their corresponding weights. The **consistent\_mass\_matrix** function uses these points and weights to compute the mass matrix. The rest of the functions (**jacobian\_det**, **shape\_functions**) remain the same as in the previous implementation.

# Part (4)

function transformed\_mass\_matrix\_demo

    % Original nodal coordinates

    nodes = [0, 0; 4, 2; 3, 4; -1, 4];

    % Material properties

    rho = 2500;  % kg/m^3

    thickness = 0.025;  % 2.5 cm converted to meters

    % Gaussian quadrature points and weights

    gauss\_points = [-0.57735, -0.57735; 0.57735, -0.57735; 0.57735, 0.57735; -0.57735, 0.57735];

    gauss\_weights = [1, 1, 1, 1];

    % Apply transformations

    transformed\_nodes = transform\_nodes(nodes);

    % Calculate consistent mass matrix for transformed nodes

    consistent\_mass = consistent\_mass\_matrix(transformed\_nodes, rho, thickness, gauss\_points, gauss\_weights);

    % Display the result

    disp('Transformed Consistent Mass Matrix:');

    disp(consistent\_mass);

end

function transformed\_nodes = transform\_nodes(nodes)

    % Translation

    translation = [2; 1];

    nodes\_translated = nodes + translation';

    % Rotation

    theta = 25 \* pi / 180;  % Convert degrees to radians

    rotation\_matrix = [cos(theta), -sin(theta); sin(theta), cos(theta)];

    pivot = nodes\_translated(1, :);  % Pivot point (node 1)

    nodes\_rotated = (nodes\_translated - pivot) \* rotation\_matrix' + pivot;

    % Scaling

    scaling\_factor = 2.5;

    nodes\_scaled = nodes\_rotated \* scaling\_factor;

    transformed\_nodes = nodes\_scaled;

end

% [Define other functions: consistent\_mass\_matrix, jacobian\_det, shape\_functions as before]

In this code, the **transform\_nodes** function performs the translation, rotation, and scaling on the nodal coordinates. The **consistent\_mass\_matrix** function then uses these transformed coordinates to calculate the mass matrix.

# Part (5)

To understand how the mass matrix of a finite element changes with translation, rotation, and scaling, it's essential to consider the nature of each transformation and its impact on the element's geometry. Let's break down each transformation and its likely effect on the mass matrix:

1. **Translation**:
   * Effect: Translation involves moving the entire element by a constant vector. It shifts the element's position but does not change its shape or size.
   * Impact on Mass Matrix: In theory, translation should not affect the mass matrix of an element. The mass matrix depends on the shape, size, and density distribution within the element, none of which are altered by a simple translation.
2. **Rotation**:
   * Effect: Rotation changes the orientation of the element without altering its shape or size.
   * Impact on Mass Matrix: For isotropic materials and uniform thickness, rotation should not change the mass distribution within the element. Hence, the consistent mass matrix, which represents this distribution, should ideally remain unchanged. However, in numerical computations, especially with non-regular shapes, minor numerical differences can arise due to the discretization of the element in the computational model.
3. **Scaling**:
   * Effect: Scaling alters the size of the element, effectively changing its area (in 2D) or volume (in 3D).
   * Impact on Mass Matrix: Scaling has a direct impact on the mass matrix. As the size of the element changes, its mass also changes proportionally. If the scaling is uniform, the mass increases by the square of the scaling factor in 2D (or the cube in 3D). This change should be reflected in the mass matrix, as it represents the distribution of mass across the element.

### Observations in the Report:

In your report, you can discuss these points:

1. **Translation**: Explain that the translation of the element did not alter the mass matrix's values, as expected, because translation does not change the element's internal mass distribution.
2. **Rotation**: Discuss how the rotation of the element might have resulted in minimal or no change in the mass matrix. Emphasize that this holds true for isotropic materials and might vary for anisotropic materials or in cases of numerical approximation.
3. **Scaling**: Highlight the significant changes in the mass matrix due to scaling. Explain that the mass of the element increases with scaling, and this increase is reflected in the mass matrix. The increase is quadratic with respect to the scaling factor in 2D elements.
4. **Numerical vs. Theoretical Consistency**: Discuss any discrepancies as potential results of numerical methods used in the calculation, especially for rotation and scaling. Emphasize the difference between ideal theoretical transformations and practical numerical computations.
5. **Practical Implications**: Finally, reflect on the importance of considering these transformations in finite element analysis, especially when dealing with dynamic problems or problems involving large deformations where the mass distribution significantly influences the results.