Report 2023Autumn

bcynuaa

January 31, 2024

Contents

1 Problem 1

1.1 Question 1

Suppose we have a v distributed in domain, we have:

$$\int_{\Omega}(-\Delta u - f)(u - v)\mathrm{d}\Omega = 0 \tag{1}$$

considering an arbitrary w, we use integrate by part:

$$\int_{\Omega} w\Delta u\mathrm{d}\Omega = \int_{\partial\Omega} w\frac{\partial u}{\partial n}\mathrm{d}\Gamma - \int_{\Omega}\nabla w\cdot\nabla u\mathrm{d}\Omega \tag{2}$$

and by Cauchy-Schwarz inequality:

$$\int_{\Omega}\nabla w\cdot\nabla u\mathrm{d}\Omega \leq \frac{1}{2}\left(\int_{\Omega}\nabla w\cdot\nabla w\mathrm{d}\Omega + \int_{\Omega}\nabla u\cdot\nabla u\mathrm{d}\Omega\right) \tag{3}$$

apply (2) to (1), we have:

$$\begin{aligned} -\int_{\Omega} u\nabla^2 u\mathrm{d}\Omega + \int_{\Omega} v\nabla^2 u\mathrm{d}\Omega - \int_{\Omega} fu\mathrm{d}\Omega + \int_{\Omega} fv\mathrm{d}\Omega = 0 \\ \int_{\Omega}\nabla u\cdot\nabla u\mathrm{d}\Omega - \int_{\partial\Omega} u\frac{\partial u}{\partial n}\mathrm{d}\Gamma - \int_{\Omega} fu\mathrm{d}\Omega = \\ \int_{\Omega}\nabla v\cdot\nabla u\mathrm{d}\Omega - \int_{\partial\Omega} v\frac{\partial u}{\partial n}\mathrm{d}\Gamma - \int_{\Omega} fv\mathrm{d}\Omega \\ \int_{\Omega}\nabla u\cdot\nabla u\mathrm{d}\Omega - \int_{\partial\Omega} u\bar{t}\mathrm{d}\Gamma - \int_{\Omega} fu\mathrm{d}\Omega = \\ \int_{\Omega}\nabla v\cdot\nabla u\mathrm{d}\Omega - \int_{\partial\Omega} v\bar{t}\mathrm{d}\Gamma - \int_{\Omega} fv\mathrm{d}\Omega \end{aligned} \tag{4}$$

apply (3) above, we will have:

$$\begin{aligned} \int_{\Omega}\nabla u\cdot\nabla u\mathrm{d}\Omega - \int_{\partial\Omega} u\bar{t}\mathrm{d}\Gamma - \int_{\Omega} fu\mathrm{d}\Omega \leq \\ \frac{1}{2}\int_{\Omega}(\nabla u\cdot\nabla u + \nabla v\cdot\nabla v)\Omega - \int_{\partial\Omega} v\bar{t}\mathrm{d}\Gamma - \int_{\Omega} fv\mathrm{d}\Omega \end{aligned} \tag{5}$$

denote $U[w]$ as below:

$$U[w] = \int_{\Omega}\frac{1}{2}\nabla w\cdot\nabla w\mathrm{d}\Omega - \int_{\partial\Omega} w\bar{t}\mathrm{d}\Gamma - \int_{\Omega} fw\mathrm{d}\Omega \tag{6}$$

we will have:

$$U[u] \leq U[v] \tag{7}$$

finally we reduce the Poisson equation to a minimization problem, and the solution of the Poisson equation is the minimizer of $U[u]$.

1.2 Question 2

From integral by part(2), we have:

$$\int_{\Omega} \nabla w \cdot \nabla \tilde{u} \mathrm{d}\Omega = \int_{\partial\Omega} w \frac{\partial \tilde{u}}{\partial n} \mathrm{d}\Gamma - \int_{\Omega} w \Delta \tilde{u} \mathrm{d}\Omega \tag{8}$$

thus we have:

$$\int_{\Omega} w(-\Delta \tilde{u} - f) \mathrm{d}\Omega + \int_{\partial\Omega} w \left(\frac{\partial \tilde{u}}{\partial n} - \bar{t} \right) \mathrm{d}\Gamma = 0 \tag{9}$$

when $w = 0$ at $\partial\Omega$, we have:

$$\int_{\Omega} w(-\Delta \tilde{u} - f) \mathrm{d}\Omega = 0 \tag{10}$$

for the arbitrary w, we have:

$$-\Delta \tilde{u} - f = 0 \tag{11}$$

Consider $\psi = \tilde{u} - u$, which satisfies:

$$\nabla^2 \psi = 0 \quad \text{in} \quad \Omega, \quad \psi = 0 \quad \text{on} \quad \partial\Omega \tag{12}$$

thus:

$$0 = \int_{\Omega} \psi \nabla^2 \psi \mathrm{d}\Omega = \int_{\partial\Omega} \psi \frac{\partial \psi}{\partial n} \mathrm{d}\Gamma - \int_{\Omega} \nabla \psi \cdot \nabla \psi \mathrm{d}\Omega \tag{13}$$

we will have:

$$\nabla \psi = \vec{0} \quad \text{in} \quad \Omega \tag{14}$$

for $\psi = 0$ at $\partial\Omega$, we have:

$$\psi = 0 \quad \text{in} \quad \Omega \tag{15}$$

which indicates that $\tilde{u} = u$.

1.3 Question 3

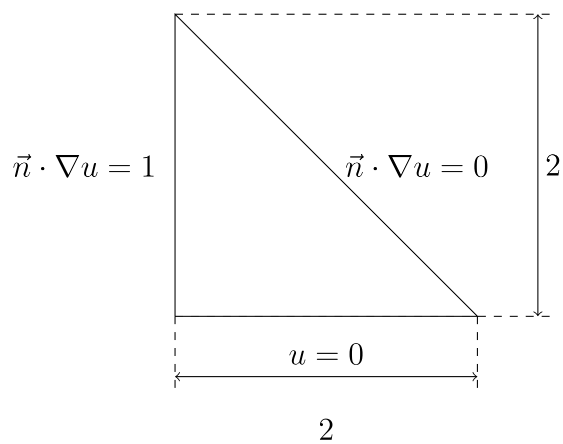


Figure 1: fig: Problem Setting

Considering a shape function ϕ_i in single triangular element, we have:

$$\sum_j a_j \int_{\Omega_e} \nabla\phi_i \cdot \nabla\phi_j \mathrm{d}\Omega = \int_{\partial\Omega_e} \phi_i \frac{\partial u}{\partial n} \mathrm{d}\Gamma \tag{16}$$

Let's consider an element as below:

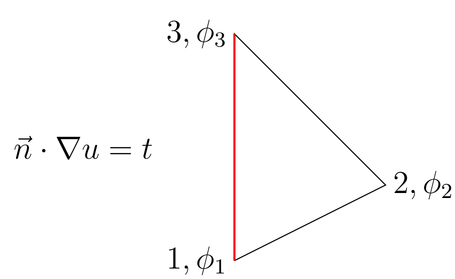


Figure 2: fig: Single Element with Neumann Boundary Condition

we will have the integral as below:

$$
\begin{aligned}
\int_3^1 \phi_2 t dl &= 0 \\
\int_3^1 \phi_1 t dl &= \frac{t}{2} l_{13} \\
\int_3^1 \phi_3 t dl &= \frac{t}{2} l_{13}
\end{aligned}
\tag{17}
$$

this integral seems to devide part tl_{13} into two parts, which are added separately to node 1 and node 3.

You may read the report last year in folder 'FEM/Autumn2022' for more details of the derivation of the integral. Another option is to watch video bilibili: https://www.bilibili.com/video/BV1qD4y1J7ZU/?spm_id_from=333.999.0.0, a video by me on last year's homework.

Generally, we have:

$$
\left[\sum_e \int_{\Omega_e} \nabla \phi_i^e \cdot \nabla \phi_j^e \mathrm{d}\Omega\right] (\tilde{u} + u_\Gamma) = \left[\sum_e \int_{\partial\Omega_e} \phi_i^e \frac{\partial u}{\partial n} \mathrm{d}\Gamma\right]
\tag{18}
$$

1.3.1 MMA Solution

To verify the solution by hand-writing code, we use Mathematica to solve the problem first. The notebook is printed as below:

In[ ]:=
```
Clear["Global`*"];
清除
nodelist = {{0, 0}, {2, 0}, {0, 2}};
tri = Polygon[nodelist];
      多边形
tri
```

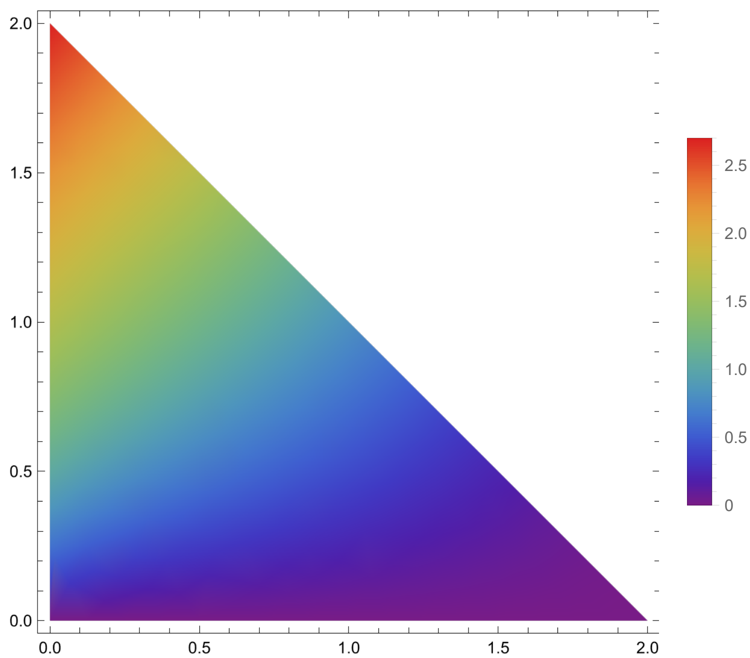
Out[ ]=
```
Polygon[ Number of points: 3
         Embedding dimension: 2 ]
```

In[ ]:=
```
res = NDSolveValue[{-∇_{x,y}^2 u[x, y] ==
      数值解的值
  NeumannValue[1., x == 0],
  诺伊曼边值
 DirichletCondition[u[x, y] == 0., y == 0]}, u, {x, y} ∈
 狄里克雷条件
 tri]
```

Out[ ]=
```
InterpolatingFunction[ Domain: {{0., 2.}, {0., 2.}}
                       Output: scalar ]
```

In[ ]:=
```
figure = DensityPlot[res[x, y], {x, y} ∈ tri,
         密度图
  PlotLegends → Automatic, ColorFunction → "Rainbow"]
  绘图的图例     自动         颜色函数
```

Out[ ]=



In[ ]:=
```
Export["..//Desktop//problem1_mma_solution.pdf", figure];
导出
Export["..//Desktop//problem1_mma_solution.png", figure]
导出
```

Out[ ]=
```
..//Desktop//problem1_mma_solution.png
```

1.3.2 Julia Code Solution

First we need include the package we use:

```
using DelaunayTriangulation, CairoMakie;
using SparseArrays, LinearAlgebra;
```

here, `DelaunayTriangulation` is used to generate the mesh, `CairoMakie` is used to plot the mesh and contour. `SparseArrays` and `LinearAlgebra` are used to solve the linear system.

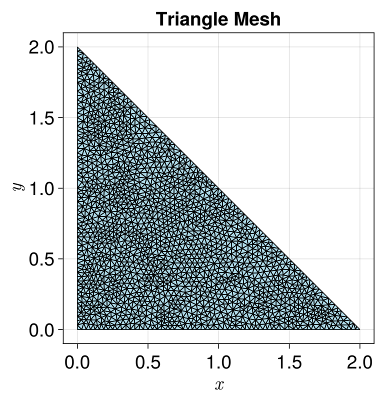


Figure 3: Triangle Mesh

Then we define the domain and generate / draw the triangular mesh, shown in Fig. 3.

```
const triangle_edge_length = 2.;
const max_area = 1e-3;
const min_angle = 31.5;
points = [
    (0., 0.),
    (triangle_edge_length, 0.),
    (0., triangle_edge_length),
    (0., 0.)
];
boundary_nodes, points = convert_boundary_points_to_indices(
    points);
triangle = triangulate(points; boundary_nodes);
refine!(triangle; max_area=max_area, min_angle=min_angle);

```

```
figure = Figure(fontsize=24)
axes = Axis(figure[1, 1], title="Triangle Mesh", titlealign=:
    center, width=400, height=400, xlabel=L"$x$", ylabel=L"$y$")
triplot!(axes, triangle, triangle_color=:lightblue)
save("triangle_mesh.pdf", figure)
save("triangle_mesh.png", figure)
figure
```

In the following code, we generate the stiffness matrix:

```
function generateStiffnessMatrix(triangle::Triangulation)
    n_nodes = length(triangle.points);
    stiffness_matrix = spzeros(n_nodes, n_nodes);
    for triangle_element in each_triangle(triangle)
        id1, id2, id3 = triangle_element;
        x1, y1 = triangle.points[id1];
        x2, y2 = triangle.points[id2];
        x3, y3 = triangle.points[id3];
        double_area = det(
            [1 x1 y1;
            1 x2 y2;
            1 x3 y3]
        );
        k11 = (x2 - x3)^2 + (y2 - y3)^2;
        k12 = (x1 - x3) * (-x2 + x3) + (y1 - y3) * (-y2 + y3);
        k13 = (x1 - x2) * (x2 - x3) + (y1 - y2) * (y2 - y3);
        k22 = (x1 - x3)^2 + (y1 - y3)^2;
        k23 = (x1 - x2) * (-x1 + x3) + (y1 - y2) * (-y1 + y3);
        k33 = (x1 - x2)^2 + (y1 - y2)^2;
        # first row
        stiffness_matrix[id1, id1] += k11 / double_area / 2;
        stiffness_matrix[id1, id2] += k12 / double_area / 2;
        stiffness_matrix[id1, id3] += k13 / double_area / 2;
        # second row
        stiffness_matrix[id2, id1] += k12 / double_area / 2;
        stiffness_matrix[id2, id2] += k22 / double_area / 2;
        stiffness_matrix[id2, id3] += k23 / double_area / 2;
        # third row
        stiffness_matrix[id3, id1] += k13 / double_area / 2;
        stiffness_matrix[id3, id2] += k23 / double_area / 2;
        stiffness_matrix[id3, id3] += k33 / double_area / 2;
    end
    return stiffness_matrix;
end

stiffness_matrix = generateStiffnessMatrix(triangle);
```

```
stiffness_matrix

figure = Figure(fontsize=24)
axes = Axis(figure[1, 1], title="Stiffness Matrix", titlealign=:
    center, width=400, height=400)
spy!(axes, rotr90(stiffness_matrix), markersize=4, marker=:
    circle, framecolor=:blue)
hidedecorations!(axes)
save("../images/stiffness_matrix.pdf", figure)
save("../images/stiffness_matrix.png", figure)
figure
```

The stiffness matrix is a sparse matrix, which is shown in Fig. 4.

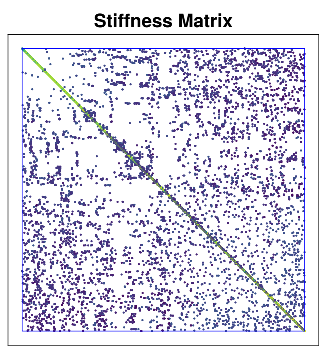


Figure 4: Stiffness Matrix

For Neumann boundary condition, we need to generate the source term vector:

```
function isLeftBoundaryNode(x, y)
    return x == 0.
end

function generateSourceVector(triangle::Triangulation,
    neumann_boundary_value::Float64)
    n_nodes = length(triangle.points);
    source_vector = zeros(n_nodes);
    for triangle_element in each_triangle(triangle)
        boundary_nodes = [];
        id1, id2, id3 = triangle_element;
```

```
        x1, y1 = triangle.points[id1];
        x2, y2 = triangle.points[id2];
        x3, y3 = triangle.points[id3];
        node_dict = Dict(id1 => (x1, y1), id2 => (x2, y2), id3 => (x3, y3));
        for (id, (x, y)) in node_dict
            if isLeftBoundaryNode(x, y)
                push!(boundary_nodes, id);
            end
        end
        if length(boundary_nodes) == 2
            n1_id, n2_id = boundary_nodes;
            n1_x, n1_y = node_dict[n1_id];
            n2_x, n2_y = node_dict[n2_id];
            edge_length = sqrt((n1_x - n2_x)^2 + (n1_y - n2_y)^2);
            source_vector[n1_id] += neumann_boundary_value * edge_length / 2;
            source_vector[n2_id] += neumann_boundary_value * edge_length / 2;
        else
            continue;
        end
    end
    return source_vector;
end
source_vector = generateSourceVector(triangle, 1.);
```

The linear problem has Dirichlet boundary condition, thus we need to modify the stiffness matrix and source vector. Below code is fit for the problem with Dirichlet boundary condition:

```
function solve(
    stiffness_matrix::SparseMatrixCSC,
    source_vector::Vector,
    known_nodes_ids::Vector,
    known_nodes_values::Vector
)::Vector
    @assert length(known_nodes_ids) == length(known_nodes_values);
    n_nodes = length(source_vector);
    solution_vector = zeros(n_nodes);
    solution_vector[known_nodes_ids] .= known_nodes_values;
    unknown_nodes_ids = setdiff(1:n_nodes, known_nodes_ids);
    part_stiffness_matrix = stiffness_matrix[unknown_nodes_ids, unknown_nodes_ids];
```

```
    part_source_vector = (source_vector .- stiffness_matrix * solution_vector)[unknown_nodes_ids];
    solution_vector[unknown_nodes_ids] .= part_stiffness_matrix \ part_source_vector;
    return solution_vector;
end
```

Then we need to get the bottom nodes ID and put them to function above:

```
function isBottomBoundaryNode(x, y)
    return y == 0.;
end

known_nodes_ids = [];
known_nodes_values = [];
for (id, (x, y)) in enumerate(triangle.points)
    if isBottomBoundaryNode(x, y)
        push!(known_nodes_ids, id);
        push!(known_nodes_values, 0.);
    else
        continue;
    end
end
solution_vector = solve(stiffness_matrix, source_vector, known_nodes_ids, known_nodes_values);
```

Finally, we plot the solution:

```
figure = Figure(fontsize=24)
axes = Axis(figure[1, 1], title="Solution", titlealign=:center, width=400, height=400, xlabel=L"$x$", ylabel=L"$y$")
tr = tricontourf!(axes, triangle, levels=51, solution_vector, colormap=:gist_rainbow)
Colorbar(figure[1, 2], tr, label="Colorbar", labelpadding=10, width=20, height=400)
save("../images/solution.pdf", figure)
save("../images/solution.png", figure)
figure
```

The solution is shown in Fig. 5, which is consistent with the solution by Mathematica. What's more, the detailed code can be found in folder `problem1/src/solution.jl` or `problem1/draft/solution.ipynb`. To run the code, you need to install Julia and the packages we use. And the package `CairoMakie` may take a long time to compile while being imported.

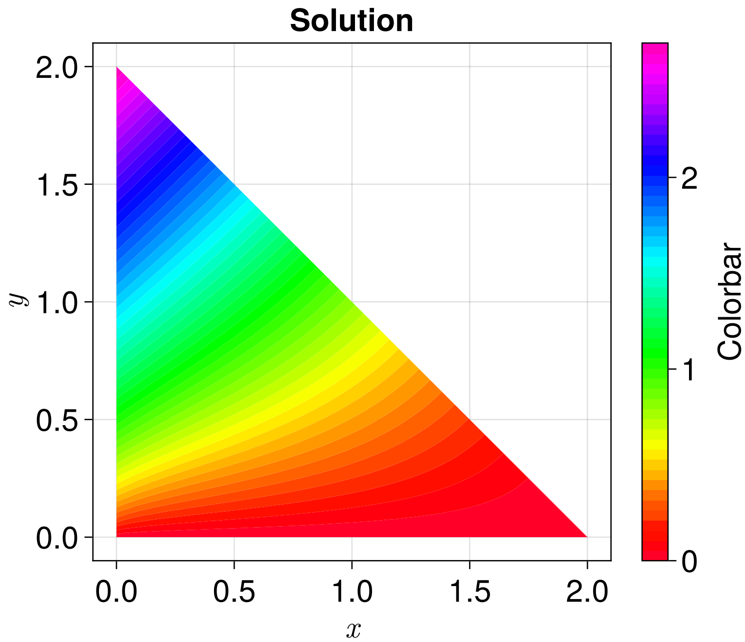


Figure 5: Solution of Problem 1

2 Problem 2

2.1 Problem Statement

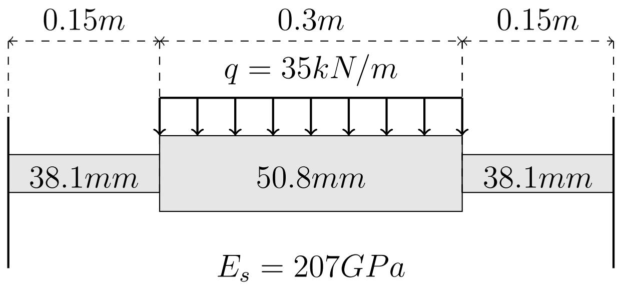


Figure 6: Problem 2

As seen in Figure 6, the beam is made of three parts, with the left and right parts having the same length of $0.15m$, and the middle part having the length of $0.3m$. The diameter of the left and right parts is $38.1mm$, and the diameter of the middle part is $50.8mm$. The beam is made of steel with the Young's modulus of $207GPa$. The beam is subjected to a distributed force of $35kN/m$ in the middle part.

We will first propose an analytical solution to this problem, and then use finite element method to solve this problem. The results will be compared between FEM code and analytical solution.

2.2 Analytical Solution

2.2.1 Symmetrical Analysis

For the symmetrical problem, we can divide the beam into two parts at the middle.

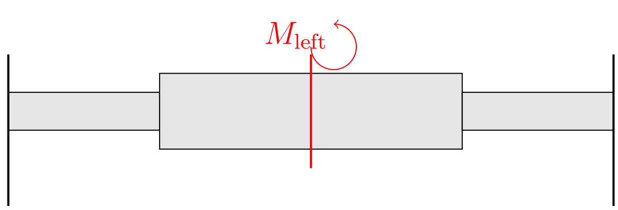


Figure 7: Symmetrical Problem

In Material Mechanics, for a symmetrical problem, internal force and moment in the middle part excludes torsion F_s but includes bending moment M_s and axial force F_N. For the left beam system part, the effect of right beam system part can be reduced to a simple bending moment M_{left}.

What's more, the rotation angle θ in the middle is 0 for such symmetrical problem.

With superposition principle, the rotation angle θ at the middle is contributed by q and $M_L = M_{\text{left}}$.Denote:

- $L = 0.15m$;
- $E = E_s = 207GPa$;
- $q = 35kN/m$;
- $J_1 = \frac{\pi}{64}d_1^4$, $J_2 = \frac{\pi}{64}d_2^4$;
- $M_L = M_{\text{left}}$;

θ in the middle part can be devided into two parts:

$$\theta = \theta_M + \theta_q \tag{19}$$

where θ_M is the rotation angle caused by M_L, and θ_q is the rotation angle caused by q. θ_M can be calculated by:

$$\theta_M = \frac{LM_L}{EJ_2} + \frac{LM_L}{EJ_1} \tag{20}$$

θ_q can be calculated by 2 parts. One part θ_q^1 is contributed by left beam part, and the other part θ_q^2 is contributed by middle beam part.

$$\theta_q = \theta_q^1 + \theta_q^2 \tag{21}$$

θ_q^1 can be calculated by the equivalent force system of q applied on the left beam part (seen in Figure 8):

$$\theta_q^1 = -\frac{qLL^2}{2EJ_1} - \frac{\frac{1}{2}qL^2L}{EJ_1} = -\frac{L^3q}{EJ_1} \tag{22}$$

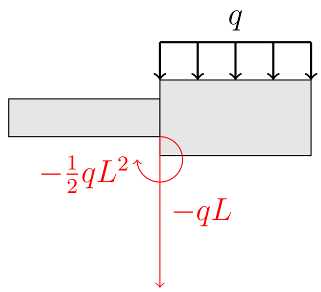


Figure 8: Equivalent Force System of q on Left Beam Part

θ_q^2 is contributed by q on the middle beam part:

$$\theta_q^2 = -\frac{qL^3}{6EJ_2} \tag{23}$$

thus θ_q is:

$$\theta_q = -\frac{L^3 q}{6EJ_2} - \frac{L^3 q}{EJ_1} \tag{24}$$

for $\theta_q + \theta_M = 0$, we have:

$$-\frac{L^3 q}{6EJ_2} + \frac{LM_L}{EJ_2} - \frac{L^3 q}{EJ_1} + \frac{LM_L}{EJ_1} = 0 \tag{25}$$

finally, we have M_L:

$$M_L = \frac{L^2 q\left(J_1 + 6J_2\right)}{6\left(J_1 + J_2\right)} \tag{26}$$

2.2.2 Distribution of θ

$\theta(x)$ can also calculated by 2 parts:

$$\theta(x) = \theta_M(x) + \theta_q(x) \tag{27}$$

For $0 \leq x \leq L$, $\theta_M(x)$ is:

$$\theta_M(x) = \frac{M_L x}{EJ_1} = \frac{L^2 qx\left(J_1 + 6J_2\right)}{6EJ_1\left(J_1 + J_2\right)} \tag{28}$$

and $\theta_q(x)$ is:

$$\theta_q(x) = -\frac{qLx^2}{EJ_1} - \frac{qL\left(L - x + \frac{L}{2}\right)}{EJ_1} = \frac{Lqx\left(-3L + x\right)}{2EJ_1} \tag{29}$$

thus $\theta(L)$ is:

$$\theta(L) = -\frac{L^3 q}{EJ_1} + \frac{L^3 q\left(J_1 + 6J_2\right)}{6EJ_1\left(J_1 + J_2\right)} \tag{30}$$

And for $L \leq x \leq 2L$, $\theta(x) = \theta_M(x) + \theta_q(x) + \theta(L)$, where $\theta_M(x)$ is:

$$\theta_M(x) = \frac{M_L(x-L)}{EJ_2} = \frac{L^2 q(x-L)\left(J_1 + 6J_2\right)}{6EJ_2\left(J_1 + J_2\right)} \tag{31}$$

and for $\theta_q(x)$, we have:

$$\begin{aligned} \theta_q(x) &= -\frac{q(x-L)^3}{6EJ_2} - \frac{q(2L-x)(x-L)^2}{2EJ_2} - \frac{\frac{1}{2}q(2L-x)^2(x-L)}{EJ_2} \\ &= \frac{q\left(7L^3 - 12L^2x + 6Lx^2 - x^3\right)}{6EJ_2} \end{aligned} \tag{32}$$

finally we have $\theta(x)$ at $L \leq x \leq 2L$:

$$\begin{aligned} \theta(x) &= \theta_M(x) + \theta_q(x) + \theta(L) \\ &= \frac{q\left(6J_1L^3 - 11J_1L^2x + 6J_1Lx^2 - J_1x^3 - 4J_2L^3 - 6J_2L^2x + 6J_2Lx^2 - J_2x^3\right)}{6EJ_2\left(J_1 + J_2\right)} \end{aligned} \tag{33}$$

thus we have the distribution of $\theta(x)$:

$$\theta(x) = \begin{cases} \frac{L^2qx(J_1+6J_2)}{6EJ_1(J_1+J_2)} + \frac{Lqx(-3L+x)}{2EJ_1} & 0 \leq x \leq L \\ \frac{q\left(6J_1L^3-11J_1L^2x+6J_1Lx^2-J_1x^3-4J_2L^3-6J_2L^2x+6J_2Lx^2-J_2x^3\right)}{6EJ_2(J_1+J_2)} & L \leq x \leq 2L \end{cases} \tag{34}$$

the other half of the beam can be calculated by symmetry.

2.2.3 Distribution of $w(x)$

Similar to the derivation of $\theta(x)$, $w(x)$ can be calculated by 2 parts:

$$w(x) = w_M(x) + w_q(x) \tag{35}$$

For $0 \leq x \leq L$, $w_M(x)$ is:

$$w_M(x) = \frac{M_L x^2}{2EJ_1} = \frac{L^2 q x^2\left(J_1 + 6J_2\right)}{12EJ_1\left(J_1 + J_2\right)} \tag{36}$$

and $w_q(x)$ is:

$$w_q(x) = -\frac{qLx^3}{3EJ_1} - \frac{qL\left(L - x + \frac{L}{2}\right)x^2}{2EJ_1} = \frac{Lqx^2\left(-9L + 2x\right)}{12EJ_1} \tag{37}$$

thus for $0 \leq x \leq L$, $w(x)$ is:

$$w(x) = \frac{Lqx^2\left(L\left(J_1 + 6J_2\right) - \left(J_1 + J_2\right)\left(9L - 2x\right)\right)}{12EJ_1\left(J_1 + J_2\right)} \tag{38}$$

which is exactly the integral of $\theta(x)$. And for $x = L$:

$$w(L) = -\frac{L^4q\left(6J_1 + J_2\right)}{12EJ_1\left(J_1 + J_2\right)} \tag{39}$$

For $L \leq x \leq 2L$, $w_M(x)$ is:

$$w_M(x) = \frac{M_L(x-L)^2}{2EJ_2} = \frac{L^2q\left(J_1 + 6J_2\right)\left(L - x\right)^2}{12EJ_2\left(J_1 + J_2\right)} \tag{40}$$

and $w_q(x)$ is:

$$\begin{aligned} w_q(x) = &-\frac{q(x-L)^4}{8EJ_2} - \frac{q(2L-x)(x-L)^3}{3EJ_2} - \frac{\frac{1}{2}q(2L-x)^2(x-L)^2}{2EJ_2} \\ = &\frac{J_1L^4q}{12EJ_1J_2 + 12EJ_2^2} - \frac{6J_1L^4q}{12EJ_1^2 + 12EJ_1J_2} \\ &- \frac{J_1L^4q}{6EJ_1^2 + 6EJ_1J_2} - \frac{2J_1L^3qx}{12EJ_1J_2 + 12EJ_2^2} \\ &+ \frac{J_1L^3qx}{6EJ_1^2 + 6EJ_1J_2} + \frac{J_1L^2qx^2}{12EJ_1J_2 + 12EJ_2^2} \\ &+ \frac{6J_2L^4q}{12EJ_1J_2 + 12EJ_2^2} - \frac{J_2L^4q}{12EJ_1^2 + 12EJ_1J_2} \\ &- \frac{6J_2L^4q}{6EJ_1^2 + 6EJ_1J_2} - \frac{12J_2L^3qx}{12EJ_1J_2 + 12EJ_2^2} \\ &+ \frac{6J_2L^3qx}{6EJ_1^2 + 6EJ_1J_2} + \frac{6J_2L^2qx^2}{12EJ_1J_2 + 12EJ_2^2} \\ &- \frac{11L^4q}{24EJ_2} + \frac{7L^3qx}{6EJ_2} - \frac{L^2qx^2}{EJ_2} + \frac{Lqx^3}{3EJ_2} - \frac{qx^4}{24EJ_2} + \frac{L^4q}{EJ_1} - \frac{L^3qx}{EJ_1} \end{aligned} \tag{41}$$

thus for $L \le x \le 2L$, $w(x)$ is:

$$w(x) = w_q(x) + w_M(x) + w(L) + \theta(L)(x - L) \tag{42}$$

which is exactly the integral of $\theta(x)$. Finally we have the distribution of $w(x)$ (too long to write here).

2.2.4 Figure of $\theta(x)$ and $w(x)$'s Distribution

Substitute the numerical values into the equations above, we have the distribution of $\theta(x)$ and $w(x)$.

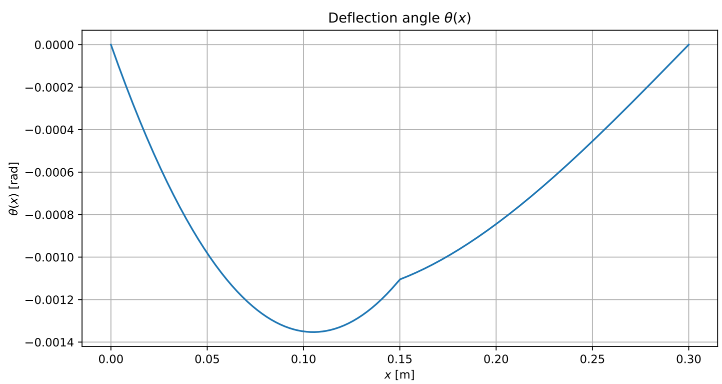


Figure 9: $\theta(x)$'s Distribution

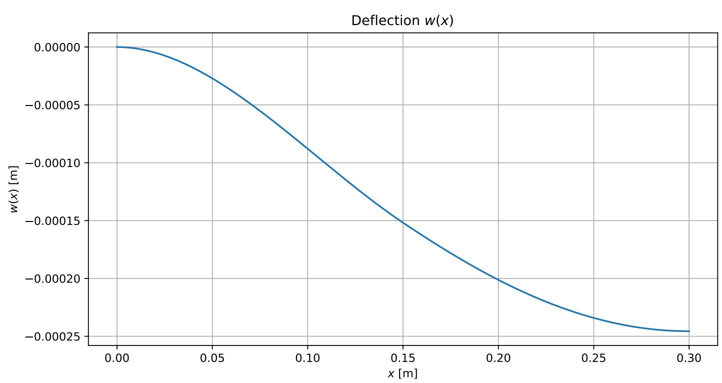


Figure 10: $w(x)$'s Distribution

The detailed derivation can be found in `problem2/src/analytical.py` or `problem2/draft/analytical.ipynb`, which uses `sympy` to do the symbolic calculation. I shall not present the derivation code here.

2.3 Julia Code Solution

You may find the detailed derivation in last year's report or my video. To conclude, we have the relation on this cell part as the format $[K]u = b$:

$$EJ \begin{bmatrix} \frac{12}{h^3} & \frac{6}{h^2} & -\frac{12}{h^3} & \frac{6}{h^2} \\ \frac{6}{h^2} & \frac{4}{h} & -\frac{6}{h^2} & \frac{2}{h} \\ -\frac{12}{h^3} & -\frac{6}{h^2} & \frac{12}{h^3} & -\frac{6}{h^2} \\ \frac{6}{h^2} & \frac{2}{h} & -\frac{6}{h^2} & \frac{4}{h} \end{bmatrix} \begin{bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{bmatrix} = q_{ex} \begin{bmatrix} \frac{h}{2} \\ \frac{h^2}{12} \\ \frac{h}{2} \\ -\frac{h^2}{12} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ F_{ex} \\ -M_{ex} \end{bmatrix} \tag{43}$$

And the application of q_{ex}, F_{ex}, M_{ex} is shown in fig.11.

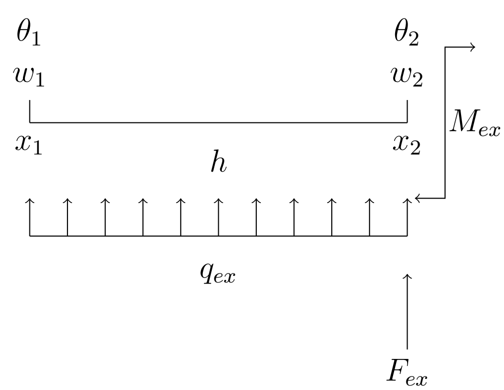


Figure 11: Bernoulli-Euler beam element

What's more, after calculation, minor-interpolation is used to get the value distributed inside an element (cell). This part can also be seen in last year's report.

For this year's task requires to use the minimum element number to achieve the accuracy, I make a comparison between 2 cases:

- 3 elements, with one elements in two sides and one element in the middle;
- 4 elements, an extra element is added in the middle.

At the beginning, I think 3 elements is enough to achieve the accuracy. However, I find that the error is still large. Thus I add an extra element in the middle, whose results quite fits for the analytical solution. The julia code is shown below.

First, import the necessary packages, where `LinearAlgebra` is used to do matrix calculation, `CairoMakie` is used to plot the figure, and `SparseArrays` is used to store the sparse matrix.

```
using LinearAlgebra, CairoMakie, SparseArrays;
```

Then, we define a struct of `HermiteBeam` and its constructor:

```
struct HermiteBeam
    n_nodes_::Int
    n_elements_::Int
    x_mesh_::Vector{Float64}
    h_::Float64
    e_s_::Float64
    inertia_::Float64
    q_::Float64
end

function HermiteBeam(n_nodes::Int, x_0::Float64, beam_length::
    Float64, e_s::Float64, inertia::Float64, q::Float64)
    h = beam_length / (n_nodes - 1);
    x_mesh = Vector(LinRange(x_0, x_0 + beam_length, n_nodes));
    return HermiteBeam(n_nodes, n_nodes-1, x_mesh, h, e_s,
        inertia, q);
end
```

By the derivation above and in last year's report, the stiffness matrix $[K]$ and force vector b can be calculated as below:

```
function generateBeamFEM(hermite_beams::Vector{HermiteBeam})
    n_nodes = sum([hermite_beam.n_nodes_ for hermite_beam in
        hermite_beams]);
    n_nodes -= length(hermite_beams) - 1;
    stiffness_matrix = spzeros(2*n_nodes, 2*n_nodes);
    source_vector = zeros(2*n_nodes);
    for (i_beam, beam) in enumerate(hermite_beams)
        start_node_id = sum([b.n_nodes_ for b in hermite_beams
            [1:i_beam-1]]) - (i_beam - 1);
        h = beam.h_;
        part_stiffness_matrix = [
            12/h^3 6/h^2 -12/h^3 6/h^2;
            6/h^2 4/h -6/h^2 2/h;
```

```
            -12/h^3 -6/h^2 12/h^3 -6/h^2;
            6/h^2 2/h -6/h^2 4/h
        ] * beam.e_s_ * beam.inertia_;
        part_source_vector = [h/2, h^2/12, h/2, -h^2/12] * beam.
            q_;
        # insert part stiffness matrix to stiffness matrix
        # and part source vector to source vector
        for i_element = 1: beam.n_elements_
            stiffness_matrix[
                2*start_node_id + 2*i_element - 1: 2*
                    start_node_id + 2*i_element + 2,
                2*start_node_id + 2*i_element - 1: 2*
                    start_node_id + 2*i_element + 2
            ] .+= part_stiffness_matrix;
            source_vector[
                2*start_node_id + 2*i_element - 1: 2*
                    start_node_id + 2*i_element + 2
            ] .+= part_source_vector;
        end
    end
    return stiffness_matrix, source_vector;
end
```

For we need compare 2 cases, we define a struct of `Problem` as below, as well as a function to generate the `HermiteBeams`:

```
struct Problem
    e_s_::Vector{Float64}
    j_::Vector{Float64}
    l_::Vector{Float64}
    n_::Vector{Int}
    q_::Vector{Float64}
end

function generateHermiteBeams(problem::Problem)
    hermite_beams = HermiteBeam[];
    for i = 1: length(problem.e_s_)
        n = problem.n_[i];
        beam_length = problem.l_[i];
        start_x = sum(problem.l_[1:i]) - beam_length;
        e_s = problem.e_s_[i];
        inertia = problem.j_[i];
        q = problem.q_[i];
        hermite_beam = HermiteBeam(n, start_x, beam_length, e_s,
            inertia, q);
        push!(hermite_beams, hermite_beam);
```

```
    end
    return hermite_beams;
end
```

For the beam is hanged on both sides towards the wall, the Dirichlet boundary condition is 0 for w and θ at the beginning and the end of the beam:

```
function knownNodes(problem::Problem)
    known_nodes_ids = [1, 2, 2*(sum(problem.n_) - length(problem.n_) + 1)-1, 2*(sum(problem.n_) - length(problem.n_) + 1)];
    known_nodes_values = [0., 0., 0., 0.];
    return known_nodes_ids, known_nodes_values;
end
```

Similar to problem 1, the **solve** function is defined as below:

```
function solve(
    stiffness_matrix::SparseMatrixCSC,
    source_vector::Vector,
    known_nodes_ids::Vector,
    known_nodes_values::Vector
)::Vector
    @assert length(known_nodes_ids) == length(known_nodes_values);
    n_nodes = length(source_vector);
    solution_vector = zeros(n_nodes);
    solution_vector[known_nodes_ids] .= known_nodes_values;
    unknown_nodes_ids = setdiff(1:n_nodes, known_nodes_ids);
    part_stiffness_matrix = stiffness_matrix[unknown_nodes_ids, unknown_nodes_ids];
    part_source_vector = (source_vector .- stiffness_matrix * solution_vector)[unknown_nodes_ids];
    solution_vector[unknown_nodes_ids] .= part_stiffness_matrix \ part_source_vector;
    return solution_vector;
end
```

Next, let's use the **solve** function to solve the problem, and return the solution w and θ at given nodes:

```
function solveProblem(problem::Problem)
    hermite_beams = generateHermiteBeams(problem);
    stiffness_matrix, source_vector = generateBeamFEM(hermite_beams);
    known_nodes_ids, known_nodes_values = knownNodes(problem);
```

```
    solution_vector = solve(stiffness_matrix, source_vector,
        known_nodes_ids, known_nodes_values);
    w = solution_vector[1:2:end];
    theta = solution_vector[2:2:end];
    return hermite_beams, w, theta;
end
```

As soon as the solution is obtained, a minor-interpolation is used to get the value distributed inside each element (cell), this derivation can be found in last year's report:

```
function hermiteInterpolation(
    hermite_beams::Vector{HermiteBeam},
    ws::Vector,
    thetas::Vector,
    x::Float64
)
    for (i_beam, beam) in enumerate(hermite_beams)
        if x < beam.x_mesh_[1] || x > beam.x_mesh_[end]
            continue;
        else
            start_node_id = sum([b.n_nodes_ for b in
                hermite_beams[1:i_beam-1]]) - (i_beam - 1);
            w_beam = ws[start_node_id+1: start_node_id+beam.
                n_nodes_];
            theta_beam = thetas[start_node_id+1: start_node_id+
                beam.n_nodes_];
            if x in beam.x_mesh_
                index = findfirst(isequal(x), beam.x_mesh_);
                return w_beam[index], theta_beam[index];
            else
                index = findfirst(item->item > x, beam.x_mesh_);
                # if index == 1
                #     index = 2;
                # end
                index -= 1;
                x1 = beam.x_mesh_[index];
                t = x - x1;
                h = beam.h_;
                w_theta = [w_beam[index], theta_beam[index],
                    w_beam[index+1], theta_beam[index+1]];
                coeff1 = [
                    (h-t)^2 * (h + 2*t), t * (h-t)^2 * h, t^2 *
                        (3*h - 2*t), t^2 * (t-h) * h
                ] ./ h^3;
                coeff2 = [
```

```
                    6 * t * (t-h), (h - 3*t) * (h - t) * h, 6 * t * (h-t), t * (3*t - 2*h) * h
                ] ./ h^3;
                return dot(coeff1, w_theta), dot(coeff2, w_theta);
            end
        end
    end
    return 0., 0.;
end

function hermiteInterpolation(
    hermite_beams::Vector{HermiteBeam},
    ws::Vector,
    thetas::Vector,
    x_mesh_minor::Vector
)
    theta_minor = similar(x_mesh_minor);
    w_minor = similar(x_mesh_minor);
    for (i, x) in enumerate(x_mesh_minor)
        w_minor[i], theta_minor[i] = hermiteInterpolation(hermite_beams, ws, thetas, x);
    end
    return w_minor, theta_minor;
end
```

Let's define a function to solve the problem and minor-interpolate the solution:

```
function solveAndMinor(problem::Problem, x_minor::Vector)
    hermite_beams, w, theta = solveProblem(problem);
    w_minor, theta_minor = hermiteInterpolation(hermite_beams, w, theta, x_minor);
    return w_minor, theta_minor;
end
```

Now, let's define 2 cases of problem and solve them as follows:

```
e_s = zeros(3) .+ 207e9;
j = [38.1e-3, 50.8e-3, 38.1e-3] .^4 .* pi ./ 64;
l = [0.15, 0.3, 0.15];
n1 = [2, 2, 2];
n2 = [2, 3, 2];
q = [0., -35e3, 0.];

problem_1 = Problem(e_s, j, l, n1, q);
problem_2 = Problem(e_s, j, l, n2, q);
```

```julia

x_minor = Vector(LinRange(0., sum(problem_1.l_), 101));
w_minor_1, theta_minor_1 = solveAndMinor(problem_1, x_minor);
w_minor_2, theta_minor_2 = solveAndMinor(problem_2, x_minor);
```

To compare the results with analytical solution, I print the analytical solution using `sympy`, and transform the `sympy` expression to `julia` expression using `codegen` module in `sympy`:

```julia
function analyticalAngle(x)
    out1 = ((x <= 0.15) ? (x .* (0.122599630386343 * x -
        0.0257568362948163)) : (-0.0862028651153974 * x .^ 3 +
        0.0775825786038577 * x .^ 2 - 0.013968317379196 * x -
        0.000464459502472649))
    return out1
end

function analyticalDeflection(x)
    out1 = ((x <= 0.15) ? (x .^ 2 .* (0.0408665434621143 * x -
        0.0128784181474081)) : (-0.0215507162788494 * x .^ 4 +
        0.0258608595346192 * x .^ 3 - 0.00698415868959802 * x .^
        2 - 0.000464459502472649 * x - 1.39767905836677e-6))
    return out1
end

x_analytical = Vector(LinRange(0., sum(problem_1.l_)/2, 21));
w_analytical = analyticalDeflection.(x_analytical);
theta_analytical = analyticalAngle.(x_analytical);
```

Finally, let's plot the results using `CairoMakie`:

```julia
# w figure
fig_w = Figure(resolution = (800, 600), fontsize = 20);
axes_w = Axis(fig_w[1, 1], xlabel = L"$x$", ylabel = L"$w$",
    title = "Deflection");
lines!(axes_w, x_minor, w_minor_1, color = :blue, linewidth = 2,
     label = L"$n_2=2$");
lines!(axes_w, x_minor, w_minor_2, color = :green, linewidth =
    2, label = L"$n_2=3$");
scatter!(axes_w, x_analytical, w_analytical, color = :red,
    linewidth = 2, label = "Analytical");
axislegend(axes_w, framevisible = true, position=:rb);
save("../images/deflection.pdf", fig_w, pt_per_unit = 0.7);
save("../images/deflection.png", fig_w);
fig_w

# theta figure
```

```
fig_theta = Figure(resolution = (800, 600), fontsize = 20);
axes_theta = Axis(fig_theta[1, 1], xlabel = L"$x$", ylabel = L"$
    \theta$", title = "Angle");
lines!(axes_theta, x_minor, theta_minor_1, color = :blue,
    linewidth = 2, label = L"$n_2=2$");
lines!(axes_theta, x_minor, theta_minor_2, color = :green,
    linewidth = 2, label = L"$n_2=3$");
scatter!(axes_theta, x_analytical, theta_analytical, color = :
    red, linewidth = 2, label = "Analytical");
axislegend(axes_theta, framevisible = true, position=:rb);
save("../images/angle.pdf", fig_theta, pt_per_unit = 0.7);
save("../images/angle.png", fig_theta);
fig_theta
```

The result is shown in fig.12 and fig.13.

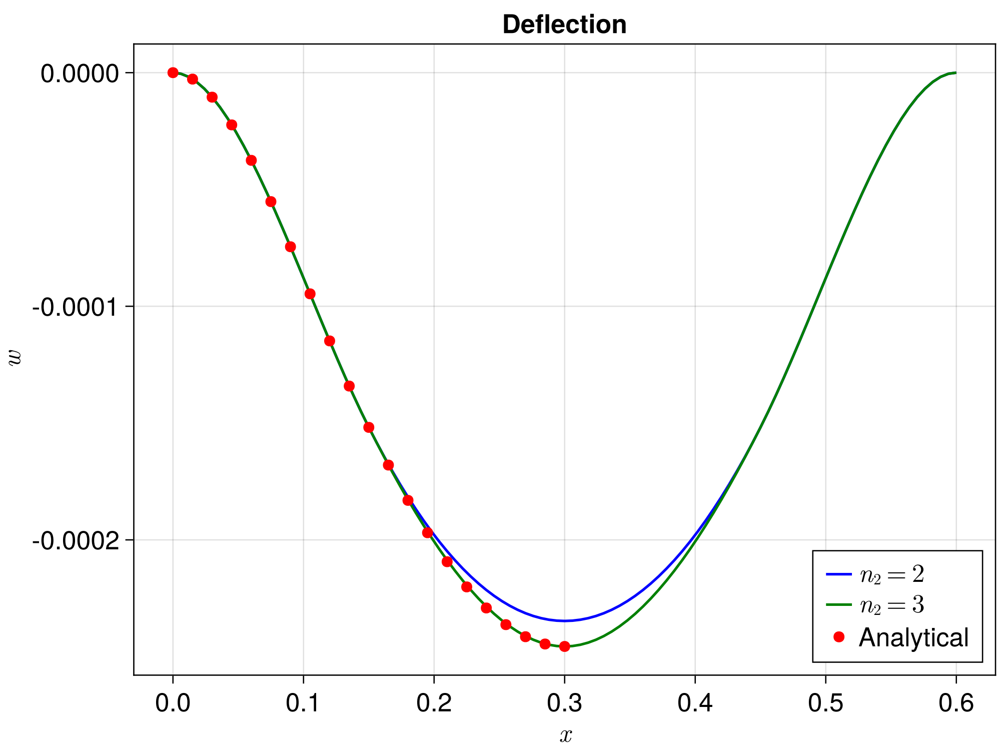


Figure 12: $w(x)$'s Distribution and Comparison

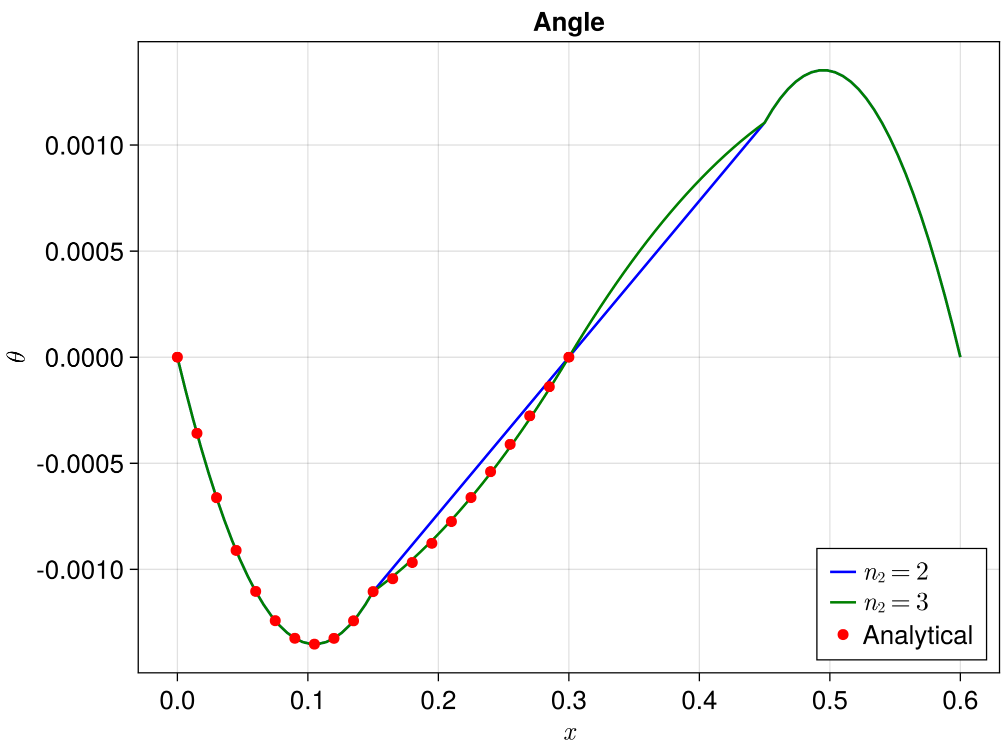


Figure 13: $\theta(x)$'s Distribution and Comparison

Although I check the solution of $n_2 = 2$ and $n_2 = 3$ (which means number of nodes in the middle beam, representing 1 and 2 elements at middle), they share the same result of w and θ at the connecting point of the middle beam and the side beams. However, for interpolation is used to get the value inside the element, the result of $n_2 = 2$ and $n_2 = 3$ is different inside the middle element.

The figure above turns out that $n_2 = 3$ is more accurate than $n_2 = 2$, compared with the analytical solution. Thus, I think the minimum element number to achieve the accuracy in such problem is 4, with 1 element in left and right beam, and 2 elements in the middle beam.