Development of a Julia-Based Tool to Aid in Solving a Finite Element Analysis Problem

By Ciara Woellhof

Overview

Introduction to FEA

Setting up the problem

Introduction to FEA

- Definition:
  - Finite Element Analysis is a numerical method for solving problems in engineering and mathematical physics
- Applications:
  - Structural analysis
  - Heat transfer
  - Fluid Dynamics
- How does it work?
  - By dividing a complex problem into smaller pieces called finite elements. A combination of all elements is used to obtain a final result.

Basic Equations in FEA

To set up a problem via FEA, you first need to define **governing equations** that will be used to specify the behavior of the physical system.

In our case, we will be doing **structural analysis** so we can use: **Ku = f**

- K: the stiffness matrix
- u: The displacement vector
- f: The force vector

Stiffness Matrix

$$[k] = \frac{EI}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix}$$

Basic Equations in Beam Bending

To solve a simple beam bending problem, I used the **Euler-Bernoulli Beam Theory**

$$\frac{d^2}{dx^2}\left(EI\frac{d^2w}{dx^2}\right) = q(x)$$

- w(x): Transverse displacement
- E: Young's modulus
- I: Moment of inertia
- q(x): Distributed load

Setting up the Problem

To solve an FEA problem in Julia, I will be using packages: LinearAlgebra, Plots, MLJ, MLJDecisionTreeInterface, Random

LinearAlgebra: Provides tools for working with vectors, matrices, and linear algebra equations. This will be helpful for creating and manipulating K,u, and f

Plots: Will be used to create visualizations of the results that Julia provides for us

MLJ: Loading and training the model

MLJDecisionTreeInterface: Provides decision tree model for use with MLJ (Decision Tree Regressor)

Random: Ability to produce random numbers and shuffle data (Creating synthetic data)

BeamFEA Struct

Struct BeamFEA: Defines a new type, BeamFEA, to store properties of the beam

- **Length:** Total length of beam
- **num_elements:** Number of finite elements the beam is divided into
- **young_modulus:** Material property indicating stiffness
- **moment_inertia:** Geometric property of the beam's cross section
- **element_length:** length of each finite element

```julia
struct BeamFEA
    length::Float64
    num_elements::Int
    young_modulus::Float64
    moment_inertia::Float64
    element_length::Float64

    # Constructor
    function BeamFEA(length::Float64, num_elements::Int,
                     young_modulus::Float64, moment_inertia::Float64)
        new(length, num_elements, young_modulus, moment_inertia,
            length / num_elements)
    end
end
```

Element Stiffness Matrix

compute_element_stiffness: Calculates the stiffness matrix for each beam element

- Parameters
- **beam**: the BeamFEA object containing beam properties
- Local Variables:
- **L**: Length of the element
- **EI**: Product of young's modulus and moment of inertia

Where the stiffness matrix, **k**, is a 4 by 4 matrix which represents the relationship between forces and displacements at the elements nodes, scaled by EI

```
function compute_element_stiffness(beam::BeamFEA)
    L = beam.element_length
    EI = beam.young_modulus * beam.moment_inertia

    # 4x4 element stiffness matrix for Euler-Bernoulli beam
    k = [
        12/L^3     6/L^2    -12/L^3    6/L^2;
        6/L^2      4/L      -6/L^2     2/L;
        -12/L^3   -6/L^2    12/L^3    -6/L^2;
        6/L^2      2/L      -6/L^2     4/L
    ] * EI

    return k
end
```

Global Stiffness Matrix Assembly

assemble_global_stiffness: Constructs the global stiffness matrix by assembling the element stiffness matrix

Parameters:

- beam: The BeamFEA object containing beam properties

Local Variables:

- total_dof: Total degrees of freedom for system (2 per node)
- k: Initialized as zero matrix of size total_dof by total_dof
- k_elements: Stiffness matrix for a single element, computed using compute_element_stiffness

Loop iterates over each element, mapping local DOF to global indices and adding the element stiffness matrix to the global matrix, k

```julia
function assemble_global_stiffness(beam::BeamFEA)
    # Total degrees of freedom
    total_dof = 2 * (beam.num_elements + 1)
    K = zeros(total_dof, total_dof)

    # Get element stiffness matrix
    k_element = compute_element_stiffness(beam)

    # Assembly process
    for i in 1:beam.num_elements
        # Map local DOFs to global
        idx = [2*i-1, 2*i, 2*i+1, 2*i+2]

        # Add element stiffness to global matrix
        K[idx, idx] += k_element
    end

    return K
end
```

Solving Beam Deflection

solve_beam_deflection: Computes the deflection oof the beam under a specific load

Parameters

- **beam:** The BeamFEA object containing beam properties
- **force_magnitude:** Magnitude of the applied force at the free end

Process:

1. Assemble **k**
2. Initialze force vector, **F**, with zeros and apply force at the last degree of freedom
3. Apply boundary conditions to fix the first two degrees of freedom (fixed end)
4. Solve the linear system **k/F** to find u, the displacement vector

```julia
function solve_beam_deflection(beam::BeamFEA, force_magnitude::Float64)
    # Assemble global stiffness matrix
    K = assemble_global_stiffness(beam)

    # Create force vector
    total_dof = 2 * (beam.num_elements + 1)
    F = zeros(total_dof)
    F[end] = force_magnitude

    # Apply fixed-free boundary conditions
    # Constrain first two DOFs (fixed end)
    K[1, :] .= 0
    K[:, 1] .= 0
    K[2, :] .= 0
    K[:, 2] .= 0

    K[1, 1] = 1
    K[2, 2] = 1

    # Solve for displacements
    displacement = K \ F

    return displacement
end
```

Visualization of Beam Deformation

Visualize_beam_deformation: Creates plots to visualize the undeformed and deformed shapes of the beam.

Parameters

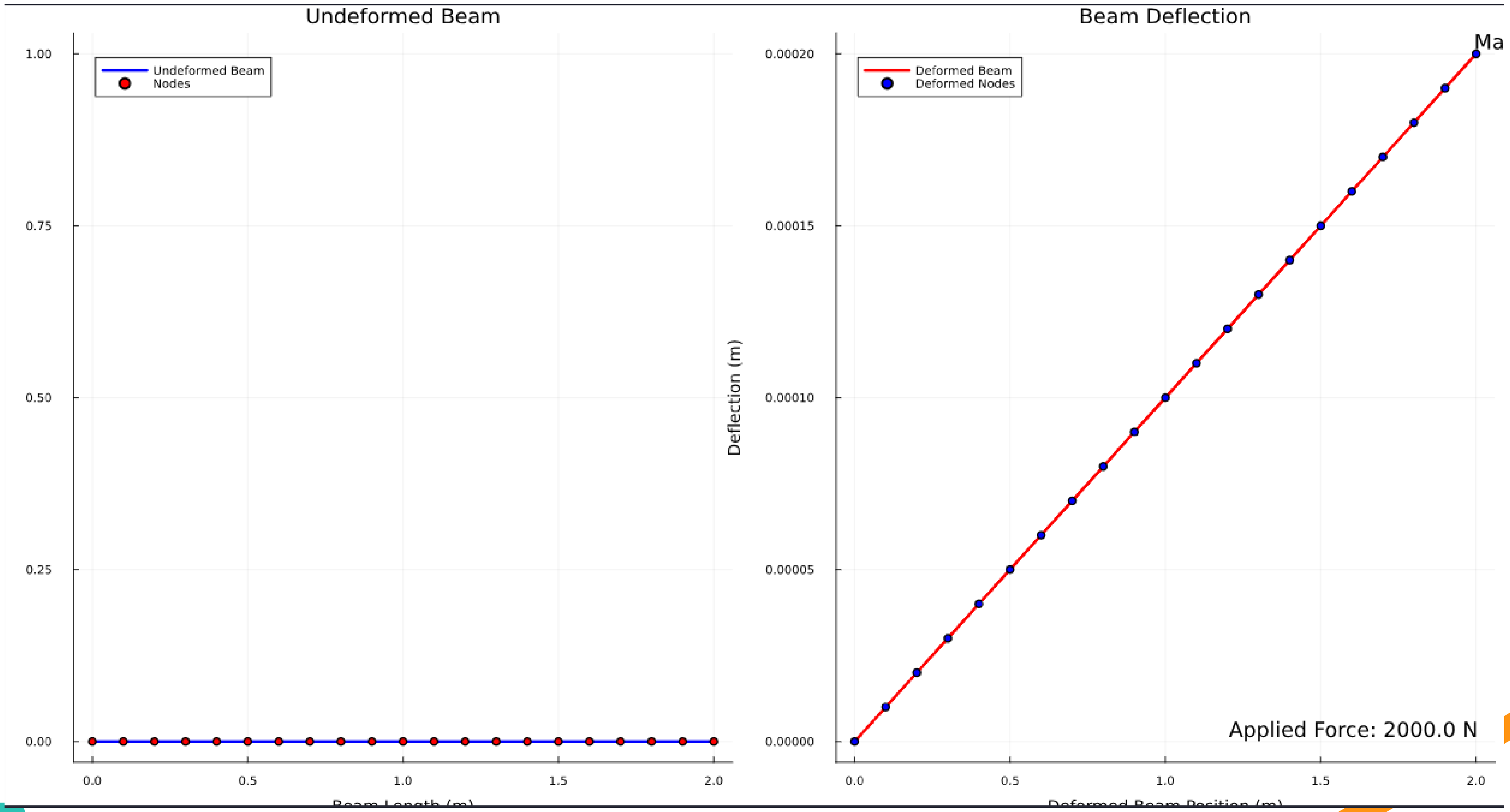
- **beam:** The BeamFEA object containing beam properties
- **displacement:** Displacement vector obtained from solution
- **force_magnitude:** Magnitude of the applied force for annotation

Process:

1. Extract y-direction displacements from displacement vector
2. Generate x-coordinates for the undeformed and deformed beam

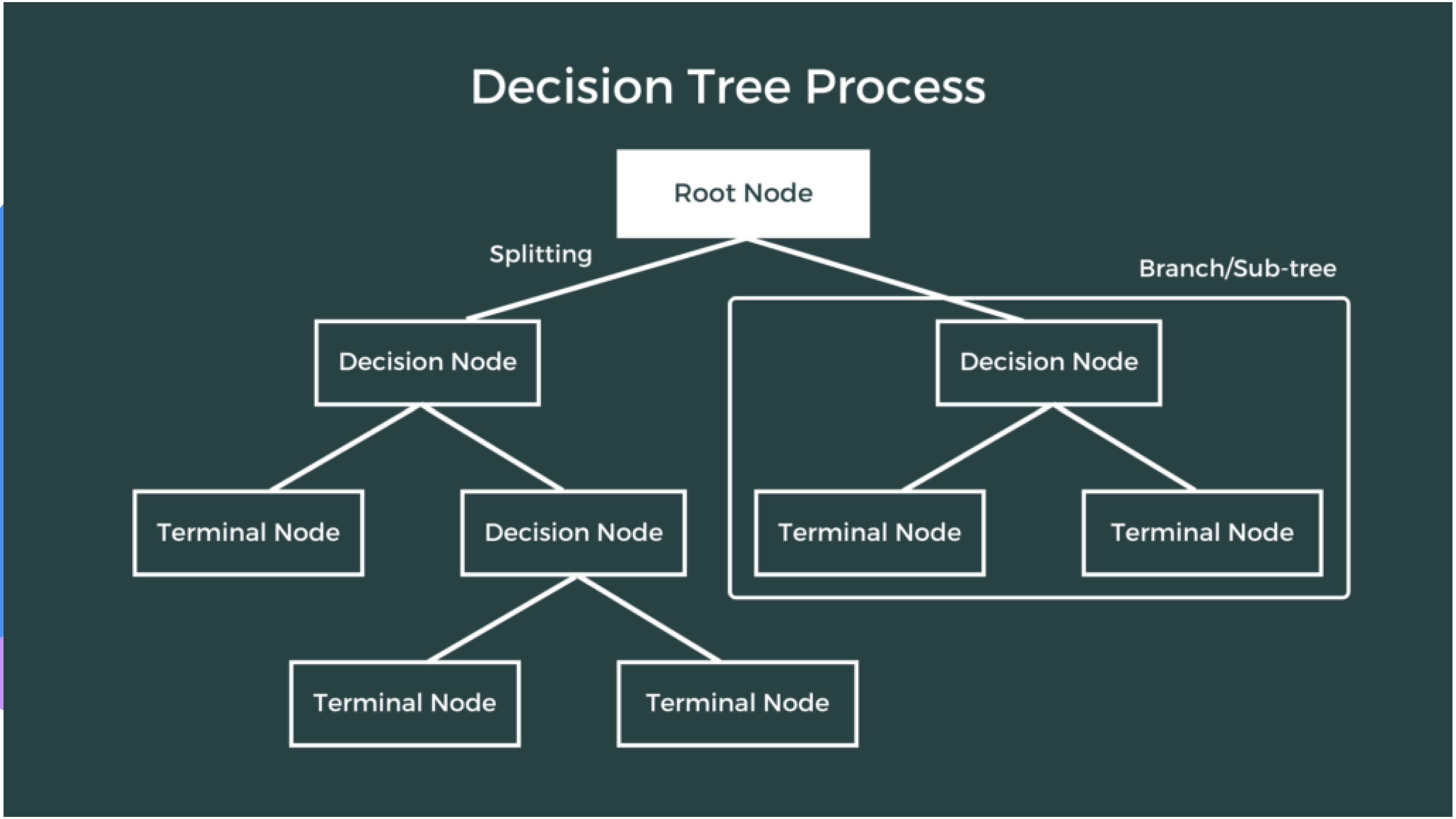
```julia
function visualize_beam_deformation(beam::BeamFEA, displacement::Vector{Float64}, force_magnitude::Float64)
    # Extract y-direction displacements
    y_displacements = displacement[2:2:end]

    # Create x-coordinates for undeformed and deformed beam
    x_undeformed = range(0, beam.length, length=beam.num_elements+1)
    x_deformed = [x_undeformed[i] + y_displacements[i] for i in 1:length(x_undeformed)]

    # Create plot with undeformed and deformed beam
    p = plot(layout = @layout[a b], size=(1200, 500),
             legend=:topleft, title="Beam Deformation Visualization")

    # Undeformed beam plot
    plot!(p[1], x_undeformed, zeros(length(x_undeformed)),
          linewidth=3, color=:blue, label="Undeformed Beam",
          title="Undeformed Beam",
          xlabel="Beam Length (m)",
          ylabel="Vertical Position (m)")
    scatter!(p[1], x_undeformed, zeros(length(x_undeformed)),
             color=:red, label="Nodes")

    # Deformed beam plot
    plot!(p[2], x_deformed, y_displacements,
          linewidth=3, color=:red, label="Deformed Beam",
          title="Beam Deflection",
          xlabel="Deformed Beam Position (m)",
          ylabel="Deflection (m)")
    scatter!(p[2], x_deformed, y_displacements,
             color=:blue, label="Deformed Nodes")

    # Annotations
    max_deflection = maximum(abs.(y_displacements))
    max_deflection_location = x_deformed[argmax(abs.(y_displacements))]

    annotate!(p[2], max_deflection_location, max_deflection,
              text("Max Deflection: $(round(max_deflection, digits=4)) m",
                   :left, :bottom))

    # Add force annotation to the deformed beam plot
    annotate!(p[2], beam.length, 0,
              text("Applied Force: $(round(force_magnitude, digits=1)) N",
                   :right, :bottom))

    display(p)
    return p
end
```

Visualizations



Machine Learning

Using Machine Learning to Predict Maximum Deflection


Decision Tree Process
Root Node
Splitting
Branch/Sub-tree
Decision Node
Decision Node
Terminal Node
Decision Node
Terminal Node
Terminal Node
Terminal Node
Terminal Node

Machine Learning Overview

Why use machine learning?

Predict maximum deflection quickly without solving entire FEA problem each time

Workflow:

1. Data Generation
    - Simulate data using FEA for various beam configurations and loads
2. Model Training
    - Train machine learning model to predict maximum deflection
3. Prediction
    - Use trained model to predict deflection for new beam configurations

Simulating Data for Training

Data Simulation Function:

- Generates synthetic data by varying beam properties and applied forces
- Computes the resulting maximum deflection using FEA
- Stores the data for training the machine learning model

```julia
# Simulate data for training the machine learning model
function simulate_data(num_samples::Int)
    data = []
    for _ in 1:num_samples
        beam_length = rand(1.0:0.1:5.0)      # meters
        num_elements = rand(5:1:50)          # elements
        young_modulus = rand(50e9:1e9:300e9) # Pa
        moment_inertia = rand(1e-6:1e-6:1e-3) # m^4
        force = rand(100.0:100.0:5000.0)     # Newtons

        beam = BeamFEA(beam_length, num_elements, young_modulus, moment_inertia)
        displacements = solve_beam_deflection(beam, force)
        max_deflection = maximum(abs.(displacements[2:2:end]))

        push!(data, (beam_length, num_elements, young_modulus, moment_inertia, force, max_deflection))
    end
    return data
end
```

Preparing Data for Machine Learning

- Data Preparation:
- Convert simulated data into a DataFrame

```
# Create DataFrame from the simulated data
df = DataFrame(data, [:beam_length, :num_elements, :young_modulus, :moment_inertia, :force, :max_deflection])
```

- Separate features (input variables) and target (output variables)

```
# Define features and target
features = select(df, Not(:max_deflection))
target = df.max_deflection
```

- Split data into training and tests sets for model evaluation

```
# Split data into training and test sets
X_train, y_train, X_test, y_test = partition_data(features, target, 0.8)
```

Training Machine Learning Model

Model Selection:

- Decision Tree Regressor is chosen for simplicity and interpretability

Training the Model:

- Train the model using training data
- Evaluate the model's performance using Root Mean Squared Error (RMSE) on the test set

```
# Train a machine learning model
mach = machine(model, X_train, y_train)
fit!(mach)
```

Evaluating the Model

Model Evaluation:

- Predict the maximum deflection on the test set
- Compute RMSE to assess models' accuracy
  - .0015014413006339351

```
# Predict on the test set
y_pred = predict(mach, X_test)
rms = sqrt(mean((y_test .- y_pred).^2))
println("Root Mean Squared Error: $rms")
```

Using Model for Prediction

Prediction Function:

- Use the trained model to predict max deflection for new beam configuration and loads

```julia
# Function to predict maximum deflection using the trained model
function predict_max_deflection(beam_length::Float64, num_elements::Int,
    young_modulus::Float64, moment_inertia::Float64,
    force::Float64, model)
new_data = DataFrame(beam_length=beam_length, num_elements=num_elements,
young_modulus=young_modulus, moment_inertia=moment_inertia,
force=force)
max_deflection_pred = predict(model, new_data)
return max_deflection_pred
end
```

Demonstration of Beam FEA with ML Prediction:

- Demonstrates entire process
- Visualizes beam deformation and prints predicted max deflection from FEA and ML

```julia
# Main function to demonstrate beam FEA with ML prediction
function main()
    beam_length = 2.0       # meters
    num_elements = 20       # elements
    young_modulus = 200e9   # Pa (steel)
    moment_inertia = 1e-4   # m^4
    force = 1000.0          # Newtons

    beam = BeamFEA(beam_length, num_elements, young_modulus, moment_inertia)
    displacements = solve_beam_deflection(beam, force)
    visualize_beam_deformation(beam, displacements, force)

    max_deflection = maximum(abs.(displacements[2:2:end]))
    println("Maximum deflection (FEA): ", max_deflection, " meters")

    max_deflection_pred = predict_max_deflection(beam_length, num_elements, young_modulus, moment_inertia, force, mach)
    println("Predicted maximum deflection (ML): ", max_deflection_pred, " meters")

    return displacements
end
```

Conclusion

Key Takeaways:

- FEA provides detailed insight into structural behavior
- Machine learning can be integrated to predict outcomes quickly and efficiently
- The combination of both can offer a useful tool for design optimization

```
Maximum deflection (FEA): 0.00040000000000014242 meters
Predicted maximum deflection (ML): [0.00047020798134305558] meters
```

Thank you