Introduction to the Finite Element Method

Shmuel Osovski

2026-03-15

Table of contents

1. Preface

These lecture notes accompany the course *Introduction to the Finite Element Method*. They cover the mathematical foundations, 1D and multi-dimensional finite element formulations, implementation details, and error analysis.

The notes are designed so that students can follow along during lectures without needing to copy derivations — instead, focus on understanding the concepts and annotating where needed.

1.1. How to Use These Notes

The chapters follow the lecture sequence. Each chapter contains:

- Mathematical derivations presented step by step
- Python code demonstrating the concepts computationally (click to expand code blocks)
- Key definitions and results highlighted in callout boxes

1.2. Prerequisites

Students should be familiar with:

- Linear algebra (matrix operations, eigenvalues)
- Calculus (integration by parts, partial derivatives)
- Basic Python programming
- Introductory continuum mechanics

Part I.

Part I: Foundations

2. Introduction and 1D Setup

2.1. Setup

2.2. The Strong Form

```{python}
#| code-fold: true
fem.display_strong_form()
```

Strong form of the equation:

$$A(x)\frac{d^2}{dx^2}y(x) + B(x)\frac{d}{dx}y(x) + C(x)y(x) = F(x)$$

With boundary conditions:

$y(0) = 0$

$y(L) = 0$

where x ∈ [0, L]

2.3. Deriving the Weak Form

```{python}
#| code-fold: true
weak_eq = fem.derive_weak_form()
```

Step 1: Multiply the strong form by a test function v(x):

EXPLANATION: We start with the strong form of our differential equation and multiply both sides by a test function v(x)

This is the first step in converting to the weak form.

The test function v(x) will eventually vanish at the boundaries, helping us incorporate boundary conditions naturally.

$$\left(A(x)\frac{d^2}{dx^2}y(x) + B(x)\frac{d}{dx}y(x) + C(x)y(x)\right)v(x) = F(x)v(x)$$

Step 2: Integrate over the domain [0, L]:

EXPLANATION: We integrate both sides of the equation over the entire domain [0, L].

This transforms our pointwise equation into an integral equation that will hold over the whole domain.

This step moves us from a local formulation to a global one, which is a key aspect of the finite element method.

$$\int_0^L \left(A(x)\frac{d^2}{dx^2}y(x) + B(x)\frac{d}{dx}y(x) + C(x)y(x) \right) v(x)\,dx = \int_0^L F(x)v(x)\,dx$$

Step 3: Expand the left-hand side:

EXPLANATION:

We expand the left-hand side to separate the terms with different derivatives of y. This makes it easier to apply integration by parts to the second-derivative term.

Breaking down the differential operator allows us to handle each term appropriately based on the order of derivatives.

$$\int_0^L A(x)v(x)\frac{d^2}{dx^2}y(x)\,dx + \int_0^L B(x)v(x)\frac{d}{dx}y(x)\,dx + \int_0^L C(x)v(x)y(x)\,dx = \int_0^L F(x)v(x)\,dx$$

Step 4: Integration by parts for the term with second derivative:

EXPLANATION:

This is the critical step that reduces the order of derivatives in our equation. We apply integration by parts to the term containing the second derivative of y.

The formula for integration by parts is:

$\int u \cdot dv = [u \cdot v] - \int v \cdot du$

For our case:

$u = v(x) \cdot A(x)$ and $dv = d^2y/dx^2\ dx$

$du = d(v(x) \cdot A(x))/dx\ dx$ and $v = dy/dx$

Using the product rule:

$d/dx(v \cdot A \cdot dy/dx) = v \cdot A \cdot d^2y/dx^2 + (v \cdot A)' \cdot dy/dx$

Rearranging:

$v \cdot A \cdot d^2y/dx^2 = d/dx(v \cdot A \cdot dy/dx) - (v \cdot A)' \cdot dy/dx$

This substitution allows us to replace the second derivative with first derivatives, reducing the continuity requirements on our approximation functions.

After integration by parts:

$$\int_0^L A(x)v(x)\frac{d^2}{dx^2}y(x)\,dx = -A(0)v(0)\ \frac{d}{dx}y(x)\Big|_{x=0} + A(L)v(L)\frac{d}{dL}y(L) - \int_0^L \left(A(x)\frac{d}{dx}v(x) + v(x)\frac{d}{dx}A(x) \right) \frac{d}{dx}y(x)$$

Step 5: Apply boundary conditions v(0) = v(L) = 0:

EXPLANATION:

We choose our test functions v(x) to vanish at the domain boundaries (v(0) = v(L) = 0).

This is a crucial step that eliminates the boundary terms from integration by parts. By carefully selecting test functions that satisfy these conditions, we naturally incorporate the essential (Dirichlet) boundary conditions into our formulation.

This is one of the elegant aspects of the finite element method.

The boundary term $[v \cdot A \cdot dy/dx]_0^L$ vanishes because $v(0) = v(L) = 0$.

Step 6: Assemble the final weak form:

EXPLANATION: Now we collect all terms to form the complete weak formulation.

This form requires lower continuity requirements on our solution (only first derivatives appear).

The weak form is equivalent to the strong form for sufficiently smooth solutions, but it allows us to find approximate solutions with less smoothness.

This is the foundation for the finite element approximation where we'll represent the solution as a combination of piecewise polynomial functions.

$$-\int_0^L \left(A(x)\frac{d}{dx}v(x) + v(x)\frac{d}{dx}A(x) \right) \frac{d}{dx}y(x)\,dx + \int_0^L B(x)v(x)\frac{d}{dx}y(x)\,dx + \int_0^L C(x)v(x)y(x)\,dx = \int_0^L F(x)v(x)\,dx$$

2.4. Shape Functions for Discretization

```{python}
#| code-fold: true
N = 5
phi, a, b, y_approx, v_approx = fem.define_shape_functions(N)
```

Step 7: Define shape functions for N elements:

EXPLANATION:

Now we discretize the problem by representing both the solution y(x) and test function v(x) as linear combinations of shape functions.

This transforms our continuous problem into a discrete one with a finite number of unknowns (the coefficients).

The shape functions are typically chosen to be simple piecewise polynomials (often linear) that are non-zero only over a small portion of the domain .

This 'local support' property often leads to sparse matrices in the final system.

$[\phi_1(x),\ \phi_2(x),\ \phi_3(x),\ \phi_4(x),\ \phi_5(x)]$

Step 8: Express solution and test functions using shape functions:

EXPLANATION:

We approximate both the solution y(x) and test function v(x) using the same set of shape functions.

The solution is written as a sum of shape functions with unknown coefficients a_j. Similarly, the test function is written as a sum with coefficients b_i.

This is the Galerkin approach, where we use the same functions for both approximation and testing.

$y(x) = a_1\phi_1(x) + a_2\phi_2(x) + a_3\phi_3(x) + a_4\phi_4(x) + a_5\phi_5(x)$

$v(x) = b_1\phi_1(x) + b_2\phi_2(x) + b_3\phi_3(x) + b_4\phi_4(x) + b_5\phi_5(x)$

Where:

- a_j are unknown constants we need to solve for
- b_i are arbitrary constants for the test function
- $\varphi_i(x)$, $\varphi_j(x)$ are known shape functions (typically piecewise linear functions)

NOTE: In the finite element method, we often choose shape functions that are equal to 1 at their corresponding node and 0 at all other nodes.

This makes it easy to enforce boundary conditions and interpret the coefficients a_j as the solution values at the nodes.

2.5. Assembling the FEM System

```
```{python}
#| code-fold: true
K, F_vec = fem.assemble_fem_system(weak_eq, phi, a, b, y_approx, v_approx, N)
```
```

** Step 9: Substitute discretized functions into the weak form:**

EXPLANATION: We now substitute our approximations for y(x) and v(x) into the weak form.

This transforms the continuous weak form into a discrete algebraic system of equations.

Since the test function coefficients b_i are arbitrary, we can collect terms and set up a system of equations for the unknown coefficients a_j.

Step 10: Assemble the system matrix and right-hand side vector:

EXPLANATION:

When we substitute the discretized functions and collect terms, we obtain a linear system of equations K·a = F, where:

- K is the stiffness matrix (or system matrix)
- a is the vector of unknown coefficients
- F is the load vector (right-hand side)

Each entry K[i,j] is computed by evaluating an integral that involves the shape functions phi_i and phi_j and their derivatives.

Similarly, each entry F[i] comes from an integral involving phi_i and the force function F(x).

For a simplified case where A(x) = B(x) = C(x) = 1, the entries would be:

$$K_{ij} = \int_0^L \frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx + \int_0^L \phi_i(x)\frac{d\phi_j}{dx}dx + \int_0^L \phi_i(x)\phi_j(x)dx$$

$$F_i = \int_0^L \phi_i(x)F(x)dx$$

NOTE: In practice, the assembly process is typically done by looping over elements,computing the element contributions, and adding them to the global system.

This element-by-element approach is more efficient and aligns with the local support property of shape functions.

2.6. Putting It All Together

```{python}
#| code-fold: true
fem.finite_element_method_1d_demo()
```

==========

INTRODUCTION TO THE FINITE ELEMENT METHOD

Lecture #1: Introduction + 1D Setup

==========

Strong form of the equation:

$$A(x)\frac{d^2}{dx^2}y(x) + B(x)\frac{d}{dx}y(x) + C(x)y(x) = F(x)$$

With boundary conditions:

$y(0) = 0$

$y(L) = 0$

where x ∈ [0, L]

==========

DERIVING THE WEAK FORM

==========

Step 1: Multiply the strong form by a test function v(x):

EXPLANATION: We start with the strong form of our differential equation and multiply both sides by a test function v(x)

This is the first step in converting to the weak form.

The test function v(x) will eventually vanish at the boundaries, helping us incorporate boundary conditions naturally.

$$\left(A(x)\frac{d^2}{dx^2}y(x) + B(x)\frac{d}{dx}y(x) + C(x)y(x)\right)v(x) = F(x)v(x)$$

Step 2: Integrate over the domain [0, L]:

EXPLANATION: We integrate both sides of the equation over the entire domain [0, L].

This transforms our pointwise equation into an integral equation that will hold over the whole domain.

This step moves us from a local formulation to a global one, which is a key aspect of the finite element method.

$$\int_0^L \left(A(x)\frac{d^2}{dx^2}y(x) + B(x)\frac{d}{dx}y(x) + C(x)y(x)\right)v(x)\,dx = \int_0^L F(x)v(x)\,dx$$

Step 3: Expand the left-hand side:

EXPLANATION:

We expand the left-hand side to separate the terms with different derivatives of y. This makes it easier to apply integration by parts to the second-derivative term.

Breaking down the differential operator allows us to handle each term appropriately based on the order of derivatives.

$$\int_0^L A(x)v(x)\frac{d^2}{dx^2}y(x)\,dx + \int_0^L B(x)v(x)\frac{d}{dx}y(x)\,dx + \int_0^L C(x)v(x)y(x)\,dx = \int_0^L F(x)v(x)\,dx$$

Step 4: Integration by parts for the term with second derivative:

EXPLANATION:

This is the critical step that reduces the order of derivatives in our equation. We apply integration by parts to the term containing the second derivative of y.

The formula for integration by parts is:

$\int u \cdot dv = [u \cdot v] - \int v \cdot du$

For our case:

$u = v(x) \cdot A(x)$ and $dv = d^2y/dx^2\ dx$

$du = d(v(x) \cdot A(x))/dx\ dx$ and $v = dy/dx$

Using the product rule:

$d/dx(v \cdot A \cdot dy/dx) = v \cdot A \cdot d^2y/dx^2 + (v \cdot A)' \cdot dy/dx$

Rearranging:

$v \cdot A \cdot d^2y/dx^2 = d/dx(v \cdot A \cdot dy/dx) - (v \cdot A)' \cdot dy/dx$

This substitution allows us to replace the second derivative with first derivatives, reducing the continuity requirements on our approximation functions.

After integration by parts:

$$\int_0^L A(x)v(x)\frac{d^2}{dx^2}y(x)\,dx = -A(0)v(0)\,\frac{d}{dx}y(x)\Big|_{x=0} + A(L)v(L)\frac{d}{dL}y(L) - \int_0^L \left(A(x)\frac{d}{dx}v(x) + v(x)\frac{d}{dx}A(x)\right)\frac{d}{dx}y(x)$$

Step 5: Apply boundary conditions v(0) = v(L) = 0:

EXPLANATION:

We choose our test functions v(x) to vanish at the domain boundaries (v(0) = v(L) = 0).

This is a crucial step that eliminates the boundary terms from integration by parts. By carefully selecting test functions that satisfy these conditions, we naturally incorporate the essential (Dirichlet) boundary conditions into our formulation.

This is one of the elegant aspects of the finite element method.

The boundary term [v·A·dy/dx]$_0^L$ vanishes because v(0) = v(L) = 0.

Step 6: Assemble the final weak form:

EXPLANATION: Now we collect all terms to form the complete weak formulation.

This form requires lower continuity requirements on our solution (only first derivatives appear).

The weak form is equivalent to the strong form for sufficiently smooth solutions, but it allows us to find approximate solutions with less smoothness.

This is the foundation for the finite element approximation where we'll represent the solution as a combination of piecewise polynomial functions.

$$-\int_0^L \left(A(x)\frac{d}{dx}v(x) + v(x)\frac{d}{dx}A(x)\right)\frac{d}{dx}y(x)\,dx + \int_0^L B(x)v(x)\frac{d}{dx}y(x)\,dx + \int_0^L C(x)v(x)y(x)\,dx = \int_0^L F(x)v(x)\,dx$$

==========

DISCRETIZATION USING SHAPE FUNCTIONS

==========

Step 7: Define shape functions for N elements:

EXPLANATION:

Now we discretize the problem by representing both the solution y(x) and test function v(x) as linear combinations of shape functions.

This transforms our continuous problem into a discrete one with a finite number of unknowns (the coefficients).

The shape functions are typically chosen to be simple piecewise polynomials (often linear) that are non-zero only over a small portion of the domain .

This 'local support' property often leads to sparse matrices in the final system.

$[\phi_1(x),\ \phi_2(x),\ \phi_3(x),\ \phi_4(x),\ \phi_5(x)]$

Step 8: Express solution and test functions using shape functions:

EXPLANATION:

We approximate both the solution y(x) and test function v(x) using the same set of shape functions.

The solution is written as a sum of shape functions with unknown coefficients a_j. Similarly, the test function is written as a sum with coefficients b_i.

This is the Galerkin approach, where we use the same functions for both approximation and testing.

$y(x) = a_1\phi_1(x) + a_2\phi_2(x) + a_3\phi_3(x) + a_4\phi_4(x) + a_5\phi_5(x)$

$v(x) = b_1\phi_1(x) + b_2\phi_2(x) + b_3\phi_3(x) + b_4\phi_4(x) + b_5\phi_5(x)$

Where:

- a_j are unknown constants we need to solve for
- b_i are arbitrary constants for the test function
- $\varphi_i(x)$, $\varphi_j(x)$ are known shape functions (typically piecewise linear functions)

NOTE: In the finite element method, we often choose shape functions that are equal to 1 at their corresponding node and 0 at all other nodes.

This makes it easy to enforce boundary conditions and interpret the coefficients a_j as the solution values at the nodes.

==========

ASSEMBLING THE FEM SYSTEM

==========

** Step 9: Substitute discretized functions into the weak form:**

EXPLANATION: We now substitute our approximations for y(x) and v(x) into the weak form.

This transforms the continuous weak form into a discrete algebraic system of equations.

Since the test function coefficients b_i are arbitrary, we can collect terms and set up a system of equations for the unknown coefficients a_j.

Step 10: Assemble the system matrix and right-hand side vector:

EXPLANATION:

When we substitute the discretized functions and collect terms, we obtain a linear system of equations K·a = F, where:

- K is the stiffness matrix (or system matrix)
- a is the vector of unknown coefficients
- F is the load vector (right-hand side)

Each entry K[i,j] is computed by evaluating an integral that involves the shape functions phi_i and phi_j and their derivatives.

Similarly, each entry F[i] comes from an integral involving phi_i and the force function F(x).

For a simplified case where A(x) = B(x) = C(x) = 1, the entries would be:

$$K_{ij} = \int_0^L \frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx + \int_0^L \phi_i(x)\frac{d\phi_j}{dx}dx + \int_0^L \phi_i(x)\phi_j(x)dx$$

$$F_i = \int_0^L \phi_i(x)F(x)dx$$

NOTE: In practice, the assembly process is typically done by looping over elements,computing the element contributions, and adding them to the global system.

This element-by-element approach is more efficient and aligns with the local support property of shape functions.

==========

SOLVING THE FEM SYSTEM

==========

The final step would be solving the linear system:

True

After solving for the unknown coefficients a_j, we can reconstruct the solution y(x)

2.7. Example: Convergence with Mesh Refinement

The following plots show the FEM solution for increasing numbers of elements, demonstrating how the numerical solution converges to the exact solution as the mesh is refined.

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 2, figsize=(10, 8))
x_exact = np.linspace(0, 1, 100)
y_exact = 0.5 * x_exact * (1 - x_exact)

for idx, n_elem in enumerate([2, 5, 10, 20]):
    ax = axes[idx // 2, idx % 2]

    # --- Compute FEM solution ---
    n_nodes = n_elem + 1
    h = 1.0 / n_elem
    K = np.zeros((n_nodes, n_nodes))
    F_vec = np.zeros(n_nodes)
    for e in range(n_elem):
        k_e = np.array([[1, -1], [-1, 1]]) / h
        f_e = np.array([h / 2, h / 2])
        K[e:e+2, e:e+2] += k_e
        F_vec[e:e+2] += f_e
    a = np.linalg.solve(K[1:-1, 1:-1], F_vec[1:-1])
```

```
    y_fem = np.zeros(n_nodes)
    y_fem[1:-1] = a
    x_fem = np.linspace(0, 1, n_nodes)

    # --- Plot on the subplot ---
    ax.plot(x_exact, y_exact, 'b-', label='Exact: y = 0.5x(1-x)')
    ax.plot(x_fem, y_fem, 'ro-', label=f'FEM ({n_elem} elements)')
    ax.set_title(f'{n_elem} elements')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.grid(True)
    ax.legend(fontsize=8)

plt.tight_layout()
plt.show()
```

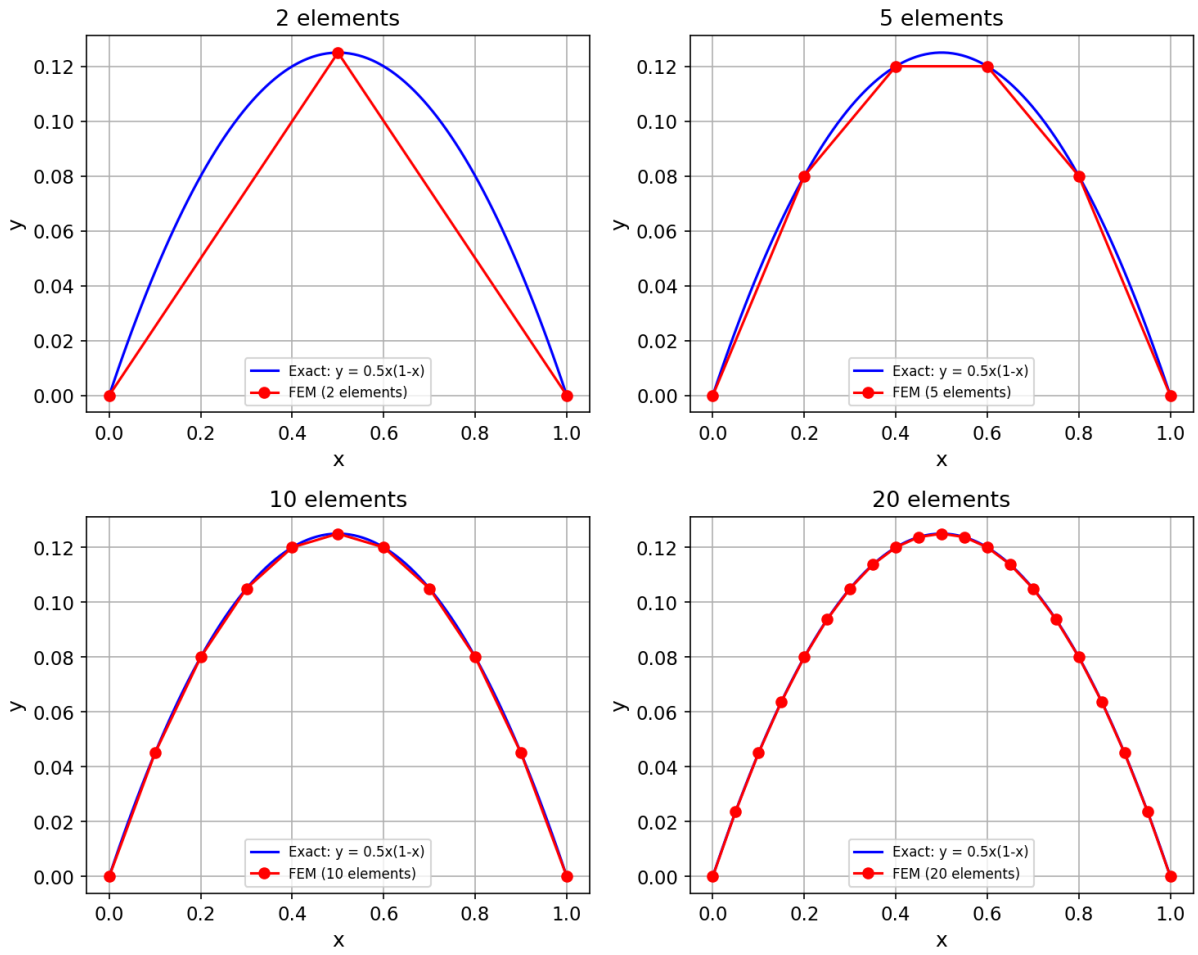


Figure 2.1.: FEM solution with varying mesh refinement

3. Deriving the Algebraic System and Galerkin's Method

3.1. Setup

3.2. Review: The Weak Form

```
```{python}
#| code-fold: true
print_styled("Assuming constant A, B, C and Dirichlet BCs v(0)=v(L)=0:")
weak_term1 = -Integral(A * diff(v, x) * diff(y, x), (x, 0, L))
weak_term2 = Integral(v * B * diff(y, x), (x, 0, L))
weak_term3 = Integral(v * C * y, (x, 0, L))
rhs_term = Integral(v * F, (x, 0, L))
weak_form_eq = Eq(weak_term1 + weak_term2 + weak_term3, rhs_term)
print_styled("The weak form is:")
Lprint(weak_form_eq)
```
```

Assuming constant A, B, C and Dirichlet BCs v(0)=v(L)=0:

The weak form is:

$$-\int_0^L A\frac{d}{dx}v(x)\frac{d}{dx}y(x)\,dx+\int_0^L Bv(x)\frac{d}{dx}y(x)\,dx+\int_0^L Cv(x)y(x)\,dx=\int_0^L F(x)v(x)\,dx$$

3.3. Discretization Using Shape Functions

```
```{python}
#| code-fold: true
print_styled("The weak form is:")
Lprint(weak_form_eq)
print_styled("Substitute Discretized Approximations")
print_styled("Approximate solution $y(x)$ and test function $v(x)$ using shape
↪  functions $\phi_k(x)$:")

y_approx = Sum(a[j] * phi(j,x), (j, 1, N))
v_approx = Sum(b[i] * phi(i,x), (i, 1, N))

print_styled("Approximate solution $y(x)$:")
```
```

```
Lprint(Eq(y, y_approx, evaluate=False))

print_styled("Test function $v(x)$:")
Lprint(Eq(v, v_approx, evaluate=False))

print_styled("Derivatives:")
dy_dx_approx = Derivative(y_approx, x)
dv_dx_approx = Derivative(v_approx, x)
Lprint(Eq(Derivative(y,x), dy_dx_approx, evaluate=False))
Lprint(Eq(Derivative(v,x), dv_dx_approx, evaluate=False))

print_styled("Substitute sums into the weak form terms:")
term1_subst_sym = weak_term1.subs({v: v_approx, y: y_approx})
term2_subst_sym = weak_term2.subs({v: v_approx, y: y_approx})
term3_subst_sym = weak_term3.subs({v: v_approx, y: y_approx})
rhs_subst_sym = rhs_term.subs({v: v_approx})

print_styled("Term 1 ($-\int A v' y' dx$):")
Lprint(term1_subst_sym)
print_styled("Term 2 ($\int B v y' dx$):")
Lprint(term2_subst_sym)
print_styled("Term 3 ($\int C v y dx$):")
Lprint(term3_subst_sym)
print_styled("RHS Term ($\int v F dx$):")
Lprint(rhs_subst_sym)
```

The weak form is:

$$-\int_0^L A\frac{d}{dx}v(x)\frac{d}{dx}y(x)\,dx+\int_0^L Bv(x)\frac{d}{dx}y(x)\,dx+\int_0^L Cv(x)y(x)\,dx=\int_0^L F(x)v(x)\,dx$$

Substitute Discretized Approximations

Approximate solution $y(x)$ and test function $v(x)$ using shape functions $\phi_k(x)$:

Approximate solution $y(x)$:

$$y(x)=\sum_{j=1}^{N}a_j\phi_j(x)$$

Test function $v(x)$:

$$v(x)=\sum_{i=1}^{N}b_i\phi_i(x)$$

Derivatives:

$$\frac{d}{dx}y(x)=\frac{\partial}{\partial x}\sum_{j=1}^{N}a_j\phi_j(x)$$

$$\frac{d}{dx}v(x)=\frac{\partial}{\partial x}\sum_{i=1}^{N}b_i\phi_i(x)$$

Substitute sums into the weak form terms:

Term 1 ($-\int Av'y'dx$):

$$-\int_0^L A\left(\frac{\partial}{\partial x}\sum_{j=1}^{N} a_j\phi_j(x)\right)\frac{\partial}{\partial x}\sum_{i=1}^{N} b_i\phi_i(x)\,dx$$

Term 2 ($\int Bvy'dx$):

$$\int_0^L B\left(\frac{\partial}{\partial x}\sum_{j=1}^{N} a_j\phi_j(x)\right)\sum_{i=1}^{N} b_i\phi_i(x)\,dx$$

Term 3 ($\int Cvydx$):

$$\int_0^L C\left(\sum_{j=1}^{N} a_j\phi_j(x)\right)\sum_{i=1}^{N} b_i\phi_i(x)\,dx$$

RHS Term ($\int vFdx$):

$$\int_0^L F(x)\sum_{i=1}^{N} b_i\phi_i(x)\,dx$$

3.4. Rearranging the Summation and Integrals

```{python}
#| code-fold: true
print_styled("Assuming continuity, swap integration and summation, then factor out $b_i$.")
print_styled(r"The weak form $\sum_{\text{term}} (\text{Term}) = \text{RHS}$ becomes:")
Lprint(r"\sum_{i=1}^N b_i \left( [\text{Inner part for Term 1}]_i + [\text{Inner part for Term 2}]_i + [\text{Inner part for Term 3}]_i \right) = \sum_{i=1}^N b_i [\text{Inner part for RHS}]_i")

print_styled(r"Where the inner parts (coefficients of $b_i$) are:")

term1_inner = Sum( -A * Integral(diff(phi(i,x), x) * diff(phi(j,x), x), (x, 0, L)) * a[j], (j, 1, N))
term2_inner = Sum(  B * Integral(phi(i,x) * diff(phi(j,x), x), (x, 0, L)) * a[j], (j, 1, N))
term3_inner = Sum(  C * Integral(phi(i,x) * phi(j,x), (x, 0, L)) * a[j], (j, 1, N))
rhs_inner = Integral(phi(i,x) * F, (x, 0, L))

print_styled("Inner part for Term 1 (coefficient of $b_i$):")
Lprint(term1_inner)
print_styled("Inner part for Term 2 (coefficient of $b_i$):")
Lprint(term2_inner)
print_styled("Inner part for Term 3 (coefficient of $b_i$):")
Lprint(term3_inner)
```

```
print_styled("Inner part for RHS (coefficient of $b_i$):")
Lprint(rhs_inner)
```

Assuming continuity, swap integration and summation, then factor out b_i.

The weak form $\sum_{\text{term}}(\text{Term}) = \text{RHS}$ becomes:

$$\sum_{i=1}^{N} b_i \left([\text{Inner part for Term 1}]_i + [\text{Inner part for Term 2}]_i + [\text{Inner part for Term 3}]_i\right) = \sum_{i=1}^{N} b_i [\text{Inner part for RHS}]_i$$

Where the inner parts (coefficients of b_i) are:

Inner part for Term 1 (coefficient of b_i):

$$\sum_{j=1}^{N} -Aa_j \int_0^L \frac{d}{dx}\phi_i(x)\frac{d}{dx}\phi_j(x)\,dx$$

Inner part for Term 2 (coefficient of b_i):

$$\sum_{j=1}^{N} Ba_j \int_0^L \phi_i(x)\frac{d}{dx}\phi_j(x)\,dx$$

Inner part for Term 3 (coefficient of b_i):

$$\sum_{j=1}^{N} Ca_j \int_0^L \phi_i(x)\phi_j(x)\,dx$$

Inner part for RHS (coefficient of b_i):

$$\int_0^L F(x)\phi_i(x)\,dx$$

3.5. Obtaining N Equations

```{python}
#| code-fold: true
print_styled("The rearranged form is:")
print_styled("$\\sum_{i=1}^N b_i ( \\text{LHS}_i - \\text{RHS}_i ) = 0$.")
print_styled("Since this must hold for arbitrary $b_i$, the term in parentheses must be zero for each $i$:")
print_styled("  $\\text{LHS}_i - \\text{RHS}_i = 0$.")

equation_for_i = Eq(term1_inner + term2_inner + term3_inner, rhs_inner)
print_styled("This gives N equations (for $i = 1, ..., N$):")
Lprint(equation_for_i)
```

The rearranged form is:

$\sum_{i=1}^{N} b_i(\text{LHS}_i - \text{RHS}_i) = 0$.

Since this must hold for arbitrary b_i, the term in parentheses must be zero for each i:

$\text{LHS}_i - \text{RHS}_i = 0$.

This gives N equations (for $i = 1, ..., N$):

$$\sum_{j=1}^{N} -Aa_j \int_0^L \frac{d}{dx}\phi_i(x)\frac{d}{dx}\phi_j(x)\,dx + \sum_{j=1}^{N} Ba_j \int_0^L \phi_i(x)\frac{d}{dx}\phi_j(x)\,dx + \sum_{j=1}^{N} Ca_j \int_0^L \phi_i(x)\phi_j(x)\,dx = \int_0^L F(x)\phi_i(x)\,dx$$

3.6. Assembling the Matrix System

```
```{python}
#| code-fold: true
print_styled("The $i$-th equation:")
print_styled("$\\sum_{j=1}^N [ ... ]_{ij} a_j = F_i$")
print_styled("leads to the matrix form:")
print_styled("$K a = F$")

K_ij = -A * Integral(diff(phi(i,x), x) * diff(phi(j,x), x), (x, 0, L)) + \
        B * Integral(phi(i,x) * diff(phi(j,x), x), (x, 0, L)) + \
        C * Integral(phi(i,x) * phi(j,x), (x, 0, L))
F_i = Integral(phi(i,x) * F, (x, 0, L))

print_styled("Where the matrix/vector components are:")
Lprint(Eq(Symbol('K_{ij}'), K_ij))
Lprint(Eq(Symbol('F_i'), F_i))

K_sym = MatrixSymbol('K', N, N)
a_sym = MatrixSymbol('a', N, 1)
F_sym = MatrixSymbol('F', N, 1)
matrix_eq = Eq(K_sym * a_sym, F_sym, evaluate=False)
print_styled("The matrix system:")
Lprint(matrix_eq)

print_styled("Writing the full system for $i=1, ..., N$ gives the matrix form
↪  $Ka=F$:")

latex_matrix_form = r"""
\begin{bmatrix}
K_{11} & K_{12} & \dots & K_{1N} \\
K_{21} & K_{22} & \dots & K_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
K_{N1} & K_{N2} & \dots & K_{NN}
\end{bmatrix}
```
```

```
\begin{bmatrix}
a_{1} \\
a_{2} \\
\vdots \\
a_{N}
\end{bmatrix}
=
\begin{bmatrix}
F_{1} \\
F_{2} \\
\vdots \\
F_{N}
\end{bmatrix}
"""
Lprint(latex_matrix_form)

print_styled("This linear system $Ka = F$ can be solved for the unknown
↪  coefficients vector $a = [a_1, a_2, ..., a_N]^T$.")
```

The i-th equation:

$\sum_{j=1}^{N}[...]_{ij}a_j = F_i$

leads to the matrix form:

$Ka = F$

Where the matrix/vector components are:

$$K_{ij} = -A\int_0^L \frac{d}{dx}\phi_i(x)\frac{d}{dx}\phi_j(x)\,dx + B\int_0^L \phi_i(x)\frac{d}{dx}\phi_j(x)\,dx + C\int_0^L \phi_i(x)\phi_j(x)\,dx$$

$$F_i = \int_0^L F(x)\phi_i(x)\,dx$$

The matrix system:

$Ka = F$

Writing the full system for $i = 1, ..., N$ gives the matrix form $Ka = F$:

$$\begin{bmatrix} K_{11} & K_{12} & \dots & K_{1N} \\ K_{21} & K_{22} & \dots & K_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ K_{N1} & K_{N2} & \dots & K_{NN} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_N \end{bmatrix}$$

This linear system $Ka = F$ can be solved for the unknown coefficients vector $a = [a_1, a_2, ..., a_N]^T$.

3.7. Note on Boundary Conditions

```
```{python}
#| code-fold: true
print_styled("We assumed $y(0)=0$ and $y(L)=0$. How are these applied to the
↪  matrix system?")
print_styled("If using nodal basis functions where:")
print_styled("$\phi_k(x_{node}) = \delta_{knode}$")
print_styled("then:")
print_styled("$y(x_1) = y(0) = \sum a_j \phi_j(x_1) = a_1$.")
print_styled("Similarly")
print_styled("$y(x_N) = y(L) = a_N$.")
print_styled("So, for $y(0)=y(L)=0$, we know $a_1 = 0$ and $a_N = 0$.")
print_styled("These known values must be incorporated when solving $Ka=F$.
↪  Common methods include:")
print_styled("  - Modifying the K matrix and F vector (e.g., setting
↪  rows/columns to enforce the condition).")
print_styled("  - Partitioning the system into known and unknown degrees of
↪  freedom.")
print_styled("Handling boundary conditions precisely is a key part of FEM
↪  implementation and will be detailed later.")
```
```

We assumed $y(0) = 0$ and $y(L) = 0$. How are these applied to the matrix system?

If using nodal basis functions where:

$\phi_k(x_{node}) = \delta_{knode}$

then:

$y(x_1) = y(0) = \sum a_j \phi_j(x_1) = a_1$.

Similarly

$y(x_N) = y(L) = a_N$.

So, for $y(0) = y(L) = 0$, we know $a_1 = 0$ and $a_N = 0$.

These known values must be incorporated when solving $Ka = F$. Common methods include:

- Modifying the K matrix and F vector (e.g., setting rows/columns to enforce the condition).
- Partitioning the system into known and unknown degrees of freedom.

Handling boundary conditions precisely is a key part of FEM implementation and will be detailed later.

3.8. Weighted Residuals and Galerkin's Method

```{python}
#| code-fold: true
print_styled("Let's revisit the core idea from a different perspective.")
print_styled("Our original differential equation is $A(y) = F$, where $A$ is
↪  the differential operator.")
def differential_operator(y_var):
    return A * diff(y_var, x, 2) + B * diff(y_var, x) + C * y_var
eq = sp.Eq(differential_operator(y), F)
Lprint(eq)
print_styled("Next, we represent $y(x)$ as a linear combination of basis
↪  functions $\phi_j(x)$, resulting in an approximated function $y^N(x)
↪  \\approx y(x)$:")

yn = Function('y^N')(x)
Lprint(Eq(yn, y_approx, evaluate=False))

print_styled("Now, we can define the residual $r^N(x)$ for the approximate
↪  solution $y^N(x)$ given by $A(y^N)-F=r^N$:")
A_op = lambda func: A * diff(func, x, 2) + B * diff(func, x) + C * func
r_N = Symbol('r^N')
residual_def = Eq(r_N, A_op(y_approx) - F, evaluate=False)
Lprint(residual_def)
print_styled("The residual $r^N(x)$ is the error in the approximation. The goal
↪  of FEM is to make this residual 'small' in some sense.")
n = sp.Symbol('n', integer=True, positive=True)
PI = sp.Symbol('PI', positive=True)
print_styled("Since $r^N$ is a function on $[0,L]$, we can measure its 'size'
↪  using:")
Lprint(Eq(PI, Integral(r_N**2, (x, 0, L)), evaluate=False))
print_styled("A possible approach is to minimize $\\Pi$ by taking the
↪  derivative with respect to the coefficients $a_j$ and setting it to zero.")
```

Let's revisit the core idea from a different perspective.

Our original differential equation is $A(y) = F$, where A is the differential operator.

$$A\frac{d^2}{dx^2}y(x) + B\frac{d}{dx}y(x) + Cy(x) = F(x)$$

Next, we represent $y(x)$ as a linear combination of basis functions $\phi_j(x)$, resulting in an approximated function $y^N(x) \approx y(x)$:

$$y^N(x) = \sum_{j=1}^{N} a_j \phi_j(x)$$

Now, we can define the residual $r^N(x)$ for the approximate solution $y^N(x)$ given by $A(y^N) - F = r^N$:

$$r^N = A\sum_{j=1}^{N} a_j \frac{d^2}{dx^2}\phi_j(x) + B\sum_{j=1}^{N} a_j \frac{d}{dx}\phi_j(x) + C\sum_{j=1}^{N} a_j \phi_j(x) - F(x)$$

The residual $r^N(x)$ is the error in the approximation. The goal of FEM is to make this residual 'small' in some sense.

Since r^N is a function on $[0, L]$, we can measure its 'size' using:

$$\pi = \int_0^L \left(r^N\right)^2 \, dx$$

A possible approach is to minimize Π by taking the derivative with respect to the coefficients a_j and setting it to zero.

3.8.1. The Galerkin Choice

```
```{python}
#| code-fold: true
print_styled("In the Galerkin approach, we define weighting functions
↪  $w_i(x)$.")
w = Function('w')
weighted_residual_eq = Eq(Integral(r_N * w(i, x), (x, 0, L)), 0)
Lprint(weighted_residual_eq)
print_styled("The **Method of Weighted Residuals** requires the residual to be
↪  orthogonal to the weighting functions $w_i(x)$.")
print_styled("The **Galerkin Method** makes a specific choice: use the *same*
↪  functions for weighting as for the basis/shape functions.")
print_styled("This means we set $w_i(x) = \\phi_i(x)$:")
galerkin_eq_from_WR = weighted_residual_eq.subs({w(i,x): phi(i,x)})
Lprint(galerkin_eq_from_WR)
print_styled(r"This must hold for $i = 1, 2, ..., N$.")

print_styled("Substituting the definition of $r^N$ into the Galerkin equation
↪  gives:")
galerkin_expanded = galerkin_eq_from_WR.subs({r_N: A_op(y_approx) - F})
Lprint(galerkin_expanded)

print_styled("Applying integration by parts to the second derivative term and
↪  requiring that $\\phi_i$ satisfy the boundary conditions, we obtain:")
print_styled(r"$\int_0^L \left( -A \frac{d\phi_i}{dx}
↪  \sum_{j=1}^N\frac{d\phi_j}{dx} + B \phi_i \sum_{j=1}^N\frac{d\phi_j}{dx} +
↪  C \phi_i \sum_{j=1}^N\phi_j - \phi_i F \right) dx = 0$")

print_styled("This is the *exact same* $i$-th equation we derived earlier from
↪  the weak form.")
print_styled("Therefore, the Galerkin method provides a powerful theoretical
↪  justification: it systematically minimizes the residual by making it
↪  orthogonal to the basis functions used for the approximation.")
```
```

In the Galerkin approach, we define weighting functions $w_i(x)$.

$$\int_0^L r^N w(i,x)\,dx = 0$$

The **Method of Weighted Residuals** requires the residual to be orthogonal to the weighting functions $w_i(x)$.

The **Galerkin Method** makes a specific choice: use the *same* functions for weighting as for the basis/shape functions.

This means we set $w_i(x) = \phi_i(x)$:

$$\int_0^L r^N \phi_i(x)\,dx = 0$$

This must hold for $i = 1, 2, ..., N$.

Substituting the definition of r^N into the Galerkin equation gives:

$$\int_0^L \left(A\sum_{j=1}^{N} a_j \frac{d^2}{dx^2}\phi_j(x) + B\sum_{j=1}^{N} a_j \frac{d}{dx}\phi_j(x) + C\sum_{j=1}^{N} a_j\phi_j(x) - F(x) \right) \phi_i(x)\,dx = 0$$

Applying integration by parts to the second derivative term and requiring that ϕ_i satisfy the boundary conditions, we obtain:

$\int_0^L \left(-A\frac{d\phi_i}{dx}\sum_{j=1}^{N}\frac{d\phi_j}{dx} + B\phi_i\sum_{j=1}^{N}\frac{d\phi_j}{dx} + C\phi_i\sum_{j=1}^{N}\phi_j - \phi_i F \right) dx = 0$

This is the *exact same* i-th equation we derived earlier from the weak form.

Therefore, the Galerkin method provides a powerful theoretical justification: it systematically minimizes the residual by making it orthogonal to the basis functions used for the approximation.

4. Function Spaces

Hilbert, L^2, and Sobolev Spaces

4.1. Setup

4.2. Motivation: Why Special Function Spaces?

In FEM, we transform strong forms of PDEs (like $\nabla \cdot (k\nabla u) = f$) into **weak forms** using integration by parts. Weak forms look like: Find u such that

$$\int_\Omega (k\nabla u \cdot \nabla v \, d\Omega) = \int_\Omega f v \, d\Omega$$

for all suitable "test functions" v.

This involves integrals of functions and their derivatives. **What kind of functions u and v can we "legally" plug into these integrals?** Do they need to be smooth? Can they have kinks?

4.3. From Vectors to Functions

You know **vector spaces** like $\mathbb{R}^3$: vectors, addition, scalar multiplication. We measure vector "size" (norm) using the dot product:

$$\|x\|^2 = x_1^2 + x_2^2 + x_3^2 = x \cdot x, \qquad \|x\| = \sqrt{x \cdot x}$$

Functions can also form vector spaces! We can add functions $(f + g)$ and scale them (cf). The challenge is defining a "dot product" and "size" (norm) for functions, and ensuring our space is "complete" (no missing limits).

4.4. Hilbert Spaces: The General Framework

A **Hilbert Space** is a vector space (often of functions) with an **inner product** $\langle f, g \rangle$. The inner product lets us define:

- **Norm (Size):** $\|f\| = \sqrt{\langle f, f \rangle}$
- **Orthogonality:** f is orthogonal to g if $\langle f, g \rangle = 0$

Crucially, Hilbert spaces are **complete**: every Cauchy sequence converges to a point *within* the space. This is vital for proving solution existence and convergence of approximations like FEM.

4.5. $L^2(\Omega)$: Square-Integrable Functions

The most common Hilbert space in introductory FEM.

$L^2(\Omega)$ is the space of functions f defined on a domain Ω whose square has a finite integral:

$$\int_\Omega |f(x)|^2 \, dx < \infty$$

Think: functions with finite "energy" or finite mean-square value.

Inner Product:

$$\langle f, g \rangle_{L^2} = \int_\Omega f(x) g(x) \, dx$$

Norm (L^2 Norm):

$$\|f\|_{L^2} = \sqrt{\int_\Omega |f(x)|^2 \, dx}$$

$L^2(\Omega)$ is a Hilbert space, also sometimes called $H^0(\Omega)$.

4.5.1. Examples

```
```{python}
#| code-fold: true
#| fig-cap: "Left: sin(x) is in L²[0, π]. Right: 1/x is NOT in L²[0, π] as its
↪  square diverges near x=0."
#| output: true
x1 = np.linspace(0.01, np.pi, 400)
x2 = np.linspace(0, np.pi, 400)

f1 = np.sin(x2)
f2 = 1 / x1

fig, axs = plt.subplots(1, 2, figsize=(10, 4))

axs[0].plot(x2, f1, label='f(x) = sin(x)')
axs[0].fill_between(x2, f1**2, alpha=0.3, label='f(x)²')
axs[0].set_title('Function in L²[0, π]')
```
```

```
axs[0].set_xlabel('x')
axs[0].set_ylabel('f(x)')
axs[0].legend()
axs[0].grid(True)
axs[0].set_ylim(bottom=0)

axs[1].plot(x1, f2, label='g(x) = 1/x')
axs[1].fill_between(x1, f2**2, alpha=0.3, label='g(x)$^2$')
axs[1].set_title('Function NOT in L$^2$[0, $\pi$]')
axs[1].set_xlabel('x')
axs[1].set_ylabel('g(x)')
axs[1].legend()
axs[1].grid(True)
axs[1].set_ylim(0, 50)

plt.tight_layout()
plt.show()
```

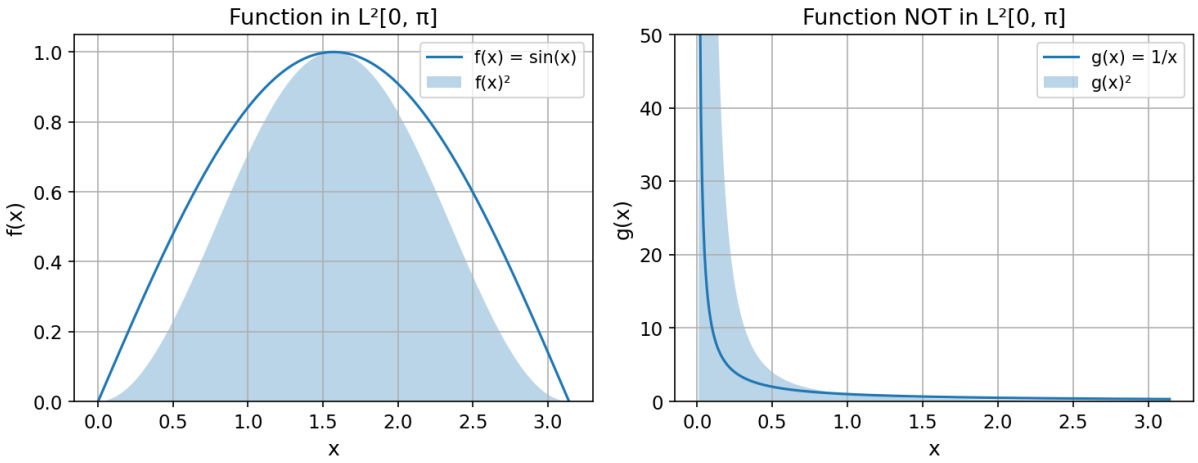


Figure 4.1.: Left: sin(x) is in $L^2[0, \pi]$. Right: 1/x is NOT in $L^2[0, \pi]$ as its square diverges near x=0.

> **Note**
>
> Functions can be unbounded (like $1/\sqrt{x}$ near 0 on $[0, 1]$) and still be in L^2. Functions can also be discontinuous and be in L^2.

4.6. Need for Derivatives: Sobolev Spaces

Our weak form involves derivatives $\nabla u \cdot \nabla v$. Requiring functions u, v to be only in $L^2(\Omega)$ is not enough — we need their derivatives to also be square-integrable.

Problem

What if u is continuous but has a kink (like $|x|$)? Its classical derivative isn't defined everywhere or might not be in L^2.

4.7. Sobolev Space $H^1(\Omega)$

$H^1(\Omega)$ is the simplest Sobolev space relevant to many FEM problems.

Definition

$H^1(\Omega)$ is the space of functions f such that both f and its first weak derivatives are in $L^2(\Omega)$:

$$H^1(\Omega) = \{f \in L^2(\Omega) \mid \nabla f \in L^2(\Omega)\}$$

A **weak derivative** is a generalization of the derivative using integration by parts. It allows functions with "corners" (like $|x|$) to still have a derivative within this framework, provided it's an L^2 function.
$H^1(\Omega)$ is also a Hilbert space.

4.7.1. H^1 Inner Product and Norm

Inner Product: Includes terms for both the function and its derivative:

$$\langle f, g \rangle_{H^1} = \int_\Omega f(x)g(x)\,dx + \int_\Omega \nabla f(x) \cdot \nabla g(x)\,dx$$

Norm (H^1 Norm):

$$\|f\|_{H^1}^2 = \|f\|_{L^2}^2 + \|\nabla f\|_{L^2}^2$$

If $\|f\|_{H^1}$ is finite, the function is "well-behaved enough" for typical second-order PDE weak forms.

4.7.2. H^1 vs L^2: Examples

```
```{python}
#| code-fold: true
#| fig-cap: "Left: x^2 is smooth, both function and derivative (2x) are in L^2,
↪  so x^2 ∈ H^1. Right: Step function is in L^2, but its derivative (a Dirac
↪  delta) is not in L^2, so Step ∉ H^1."
#| output: true
x = np.linspace(-1, 1, 400)

f1 = x**2
f1_deriv = 2*x
```
```

```
f2 = np.sign(x)

fig, axs = plt.subplots(1, 2, figsize=(10, 4))

axs[0].plot(x, f1, label='f(x) = x²')
axs[0].plot(x, f1_deriv, label="f'(x) = 2x", linestyle='--')
axs[0].set_title('Function in H¹[-1, 1]')
axs[0].set_xlabel('x')
axs[0].set_ylabel('Value')
axs[0].legend()
axs[0].grid(True)

axs[1].plot(x, f2, label='g(x) = sign(x) (Step)')
axs[1].set_title('Function in L²[-1, 1], but NOT H¹[-1, 1]')
axs[1].set_xlabel('x')
axs[1].set_ylabel('Value')
axs[1].legend()
axs[1].grid(True)

plt.tight_layout()
plt.show()
```

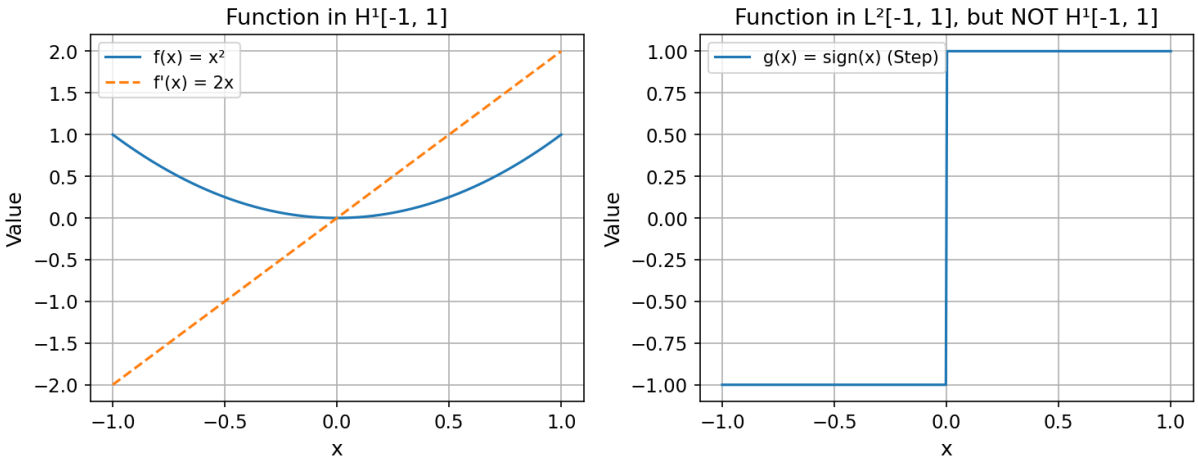


Figure 4.2.: Left: x^2 is smooth, both function and derivative (2x) are in L^2, so $x^2 \in H^1$. Right: Step function is in L^2, but its derivative (a Dirac delta) is not in L^2, so Step $\notin H^1$.

Key observations:

- Smooth functions (like polynomials) are typically in H^1
- Functions with "jumps" (like the step function) are often in L^2 but not H^1
- Functions with "kinks" (like $|x|$) can be in H^1 because their weak derivative sign(x) exists and is in L^2

4.8. Summary

- **Weak Formulations**: FEM relies on integral (weak) forms of PDEs

- L^2 (H^0): Guarantees that integrals of functions themselves ($\int f v \, dx$) make sense. Useful for source terms, mass matrices
- H^1: Guarantees that integrals involving first derivatives ($\int \nabla f \cdot \nabla v \, dx$) also make sense. The natural solution space for many second-order problems (heat, elasticity, potential flow). Ensures stiffness matrices are well-defined
- Using these spaces provides a rigorous mathematical foundation for FEM, ensuring solutions exist and approximations converge

4.9. Beyond H^1

Higher-order Sobolev spaces exist, like $H^k(\Omega)$, which include functions with weak derivatives up to order k. Specific spaces such as $H(\text{div})$ and $H(\text{curl})$ are useful for vector fields in fluid dynamics and electromagnetics.

Part II.

Part II: 1D Finite Elements

5. Galerkin's Method and 1D FEM

5.1. Setup

5.2. From Last Week

Key insight: The weak form from Lecture 1 and Galerkin's method from Lecture 2 are fundamentally the same.

Galerkin's approach (review):

1. Compute the residual $Ay_N - f = r_N(x)$
2. Force the residual to be orthogonal to each approximation function: $\int_0^L r_N(x)\phi_i(x)\,dx = 0$, for $i = 1, 2, \ldots, N$
3. Solve the resulting set of coupled equations

However, the primary challenge remains: **no systematic way for choosing the approximation functions (yet)**. The Finite Element Method (FEM) is designed specifically to address this problem — it is based on Galerkin's method, provides a computationally systematic and efficient approach, removes restrictions on differentiability, and creates a framework for handling complex geometries.

5.3. A 1D Model Problem

Today we start solving our first PDE using the finite element method.

As is often the case in real life problems, materials are not necessarily homogeneous. Moreover, the physical solution we are looking for is not necessarily smooth.

Consider the 1D bar shown below, composed of two different elastic solids:

```
```{python}
#| code-fold: true
#| fig-cap: "1D bar composed of two different materials"
create_fem_diagram(
    width=8,
    height=0.8,
    num_elements=12,
    use_gradient=False,
    element_color='#c0c0c0',
    show_element_labels=True,
)
```
```

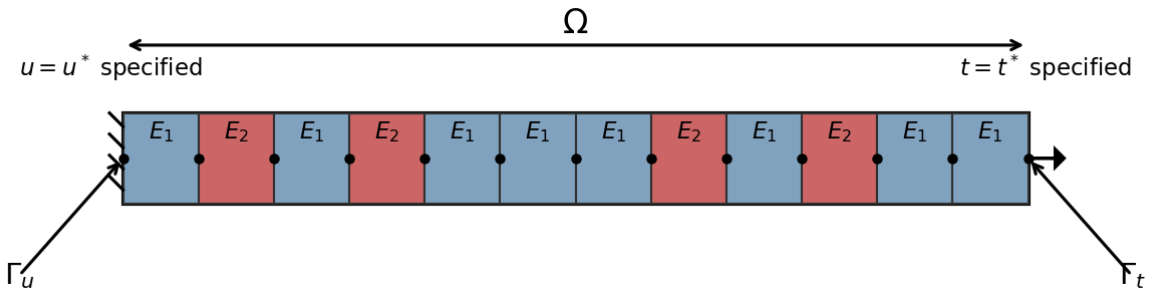


Figure 5.1.: 1D bar composed of two different materials

The bar is subjected to tractions t^* on its left side (Γ_t), while the right side is fixed (Γ_u). The bar is composed of two different materials with Young's moduli E_1 and E_2.

5.4. Strong Form

```
```{python}
#| code-fold: true
print_styled("The strong form of the PDE:")
Strong_form = diff(sigma,x)+f
Lprint(Eq(Strong_form,0))
print_styled('This PDE is a balance of forces, with $f$ being the body force
↪  and $\\sigma$ the stress.')
print_styled("The boundary conditions:")
print_styled("$u|_{\\Gamma_u} = u^*$")
print_styled("$t|_{\\Gamma_t} = t^*$")
```
```

The strong form of the PDE:

$$f + \frac{d}{dx}\sigma(x) = 0$$

This PDE is a balance of forces, with f being the body force and σ the stress.

The boundary conditions:

$u|_{\Gamma_u} = u^*$

$t|_{\Gamma_t} = t^*$

Since we assume the body to be linearly elastic:

```
```{python}
#| code-fold: true
print_styled("The stress $\\sigma$ is given by:")
Elastic_1d = E*epsilon
Lprint(Eq(sigma,Elastic_1d))
print_styled('where $E$ is the Young\'s modulus and $\\epsilon$ is the
↪  strain.')
```
```

The stress σ is given by:

$\sigma(x) = E\epsilon(x)$

where E is the Young's modulus and ϵ is the strain.

For 1D small displacements:

```{python}
#| code-fold: true
print_styled("The strain is:")
Strain_1d = diff(u,x)
Lprint(Eq(epsilon,Strain_1d))
print_styled("We can write the stress as:")
Lprint(Eq(sigma,E*diff(u,x)))
```

The strain is:

$\epsilon(x) = \frac{d}{dx}u(x)$

We can write the stress as:

$\sigma(x) = E\frac{d}{dx}u(x)$

5.5. Getting the Weak Form

To get the weak form we multiply by a test function v and integrate over the domain Ω:

```{python}
#| code-fold: true
residual = Integral(r*v, (x,0,L))
weak_term = Integral((diff(sigma,x)+f)*v, (x,0,L))
Lprint(Eq(weak_term,0))
```

$$\int_0^L \left(f + \frac{d}{dx}\sigma(x)\right) v(x)\,dx = 0$$

Using integration by parts:

```{python}
#| code-fold: true
print_styled("Using integration by parts:")
chain_rule = Derivative(sigma*v,x)
Lprint(Eq(chain_rule,chain_rule.doit(),evaluate=False))
print_styled("And thus:")
Lprint(Eq(v*Derivative(sigma,x), Derivative(sigma*v,x)-sigma*Derivative(v,x)))
```

Using integration by parts:

$$\frac{d}{dx}\sigma(x)v(x) = \sigma(x)\frac{d}{dx}v(x) + v(x)\frac{d}{dx}\sigma(x)$$

And thus:

$$v(x)\frac{d}{dx}\sigma(x) = -\sigma(x)\frac{d}{dx}v(x) + \frac{d}{dx}\sigma(x)v(x)$$

```{python}
#| code-fold: true
print_styled("We can write the weak form as:")
weak_form_before_bc =
↪  Integral(chain_rule,(x,0,L))-Integral(sigma*Derivative(v,x),(x,0,L))+Integral(f*v,(x,0,L
Lprint(Eq(weak_form_before_bc,0))
```

We can write the weak form as:

$$\int_0^L fv(x)\,dx - \int_0^L \sigma(x)\frac{d}{dx}v(x)\,dx + \int_0^L \frac{d}{dx}\sigma(x)v(x)\,dx = 0$$

Requiring $v|_{\Gamma_u} = 0$ and $\sigma|_{\Gamma_t} = t^*$:

```{python}
#| code-fold: true
print_styled("Starting from the boundary term:")
weak_form_3 = Integral(chain_rule,(x,0,L))
sv = sigma*v
Lprint(Eq(weak_form_3,sv.subs({x:L})-sv.subs({x:0})))
print_styled("Recalling that $v$ is 0 on $\\Gamma_u$:")
Lprint(Eq(weak_form_3,t_star*v.subs({x:L})))
print_styled("The weak form becomes:")
weak_LHS = Integral(sigma*Derivative(v,x),(x,0,L))
weak_RHS = Integral(f*v,(x,0,L))+t_star*v.subs({x:L})
Lprint(Eq(weak_LHS,weak_RHS))
```

Starting from the boundary term:

$$\int_0^L \frac{d}{dx}\sigma(x)v(x)\,dx = -\sigma(0)v(0) + \sigma(L)v(L)$$

Recalling that v is 0 on Γ_u:

$$\int_0^L \frac{d}{dx}\sigma(x)v(x)\,dx = t^*v(L)$$

The weak form becomes:

$$\int_0^L \sigma(x)\frac{d}{dx}v(x)\,dx = t^*v(L) + \int_0^L fv(x)\,dx$$

5.6. The Weak Form

To get the final weak form in terms of the displacement u, we introduce the constitutive and kinematic relations:

```{python}
#| code-fold: true
print_styled("Substituting the constitutive relation:")
weak_LHS_sub = weak_LHS.subs({sigma:E*diff(u,x)})
Lprint(Eq(weak_LHS_sub,weak_RHS))
```

Substituting the constitutive relation:

$$\int_0^L E\frac{d}{dx}u(x)\frac{d}{dx}v(x)\,dx = t^*v(L) + \int_0^L fv(x)\,dx$$

The final equation is the **Weak Form** or **Variational Formulation**. We can write it as:

$$\mathbb{B}(u,v) = \mathbb{F}(v)$$

where:

- $\mathbb{B}(u,v) = \int_0^L \frac{dv}{dx}E\frac{du}{dx}\,dx$ represents internal virtual work
- $\mathbb{F}(v) = \int_0^L fv\,dx + t^*v|_{\Gamma_t}$ represents external virtual work

This form requires less smoothness on the solution u compared to the strong form and is the starting point for the Finite Element Method.

5.7. Function Spaces for the Weak Form

For well-defined integrals, we need:

$$\forall v, \quad \int_0^L \frac{dv}{dx}\sigma\,dx < \infty, \quad \int_0^L fv\,dx < \infty$$

This leads to using Sobolev spaces:

$$u \in H^1(\Omega) \iff \int_\Omega \frac{du}{dx}\frac{du}{dx}\,dx + \int_\Omega u\,u\,dx < \infty$$

Our problem becomes: Find $u \in H^1(\Omega)$, $u|_{\Gamma_u} = u^*$, such that $\forall v \in H^1(\Omega)$, $v|_{\Gamma_u} = 0$:

$$\int_0^L \frac{dv}{dx}E\frac{du}{dx}\,dx = \int_0^L fv\,dx + t^*v|_{\Gamma_t}$$

5.8. Approximating the Solution

We discretize the problem using:

$$u_N \approx u(x) = \sum_{j=1}^{N} a_j \phi_j(x), \quad v = \phi_i(x)$$

This gives us a linear system $[K]\{a\} = \{F\}$ where:

$$K_{ij} = \int_0^L \frac{d\phi_i}{dx} E \frac{d\phi_j}{dx}\, dx, \qquad F_i = \int_0^L f\phi_i\, dx + t^* \phi_i|_{\Gamma_t}$$

5.9. Basis Functions in FEM

The key insight of FEM is a systematic way to construct basis functions:

- Functions are built piecewise over subdomains ("elements")
- Usually low-degree polynomials
- Smooth enough to be in $H^1(\Omega)$
- Form a nodal basis where $\phi_i(x_j) = \delta_{ij}$ (Kronecker delta)
- The sum over the basis functions satisfies: $\sum_{j \in \text{element}} \phi_j(x) = 1$ (partition of unity)
- Each basis function is non-zero only on elements sharing node i

5.9.1. Linear Elements

For linear elements, we define:

$$\phi_i(x) = \begin{cases} \frac{x - x_{i-1}}{h_i}, & x_{i-1} \le x \le x_i \\ 1 - \frac{x - x_i}{h_{i+1}}, & x_i \le x \le x_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

where $h_i = x_i - x_{i-1}$. The derivatives are:

$$\frac{d\phi_i}{dx} = \begin{cases} \frac{1}{h_i}, & x_{i-1} \le x \le x_i \\ -\frac{1}{h_{i+1}}, & x_i \le x \le x_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

5.9.2. Visualization of Basis Functions

```
```{python}
#| code-fold: true
#| fig-cap: "Lagrange shape functions of degree 1, 2, and 3"
fem_viz = FEMShapeFunctions()
fig1 = fem_viz.plot_multielement_shape_functions(
    num_elements=3,
```
```

```
    element_type='lagrange',
    degrees=[1, 2, 3])
```

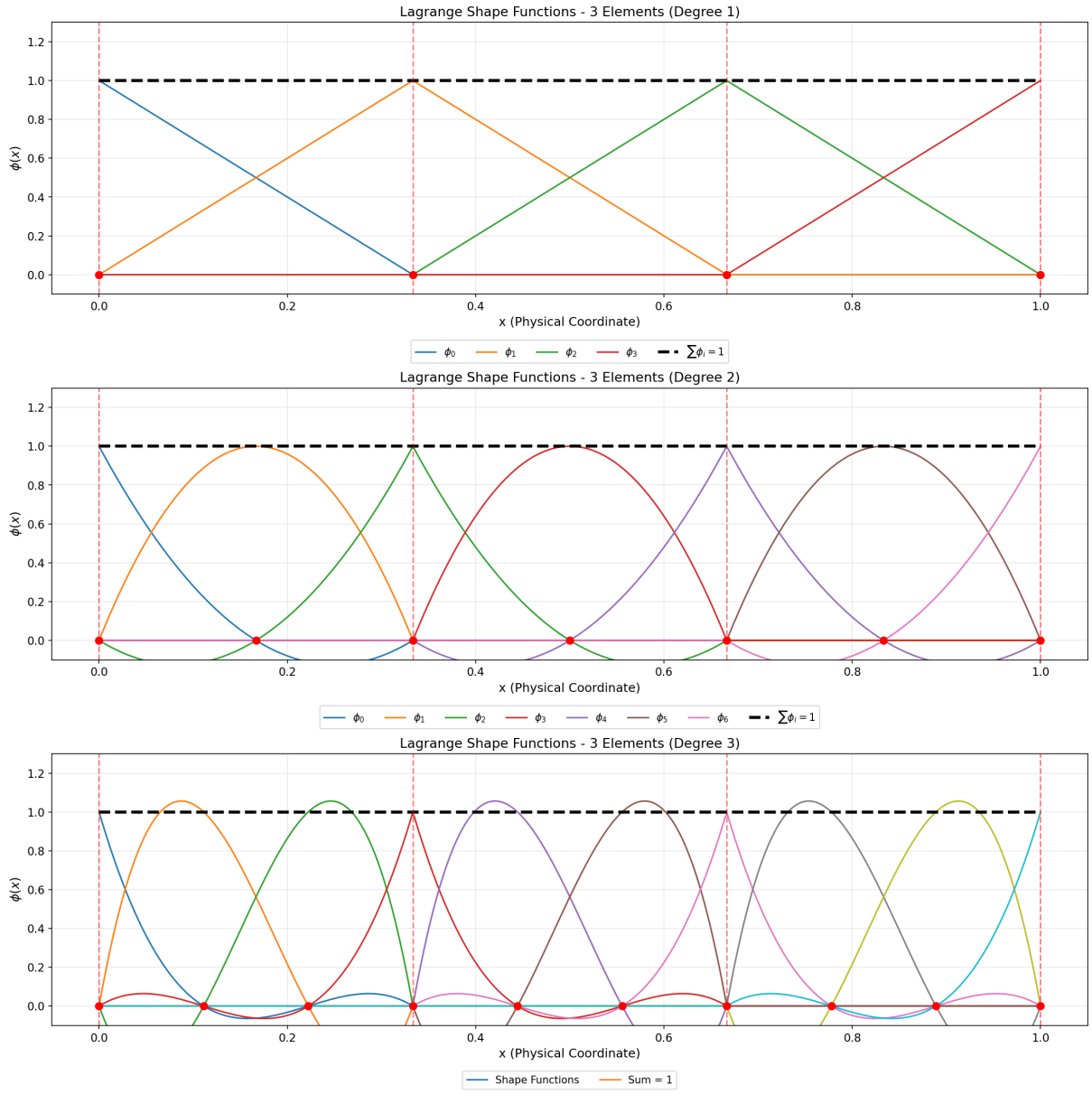


Figure 5.2.: Lagrange shape functions of degree 1, 2, and 3

```{python}
#| code-fold: true
#| fig-cap: "Derivatives of Lagrange shape functions"
fem_viz = FEMShapeFunctions()
fig2 = fem_viz.plot_multielement_shape_function_derivatives(
    num_elements=3,
    element_type='lagrange',
    degrees=[1, 2]
)
```

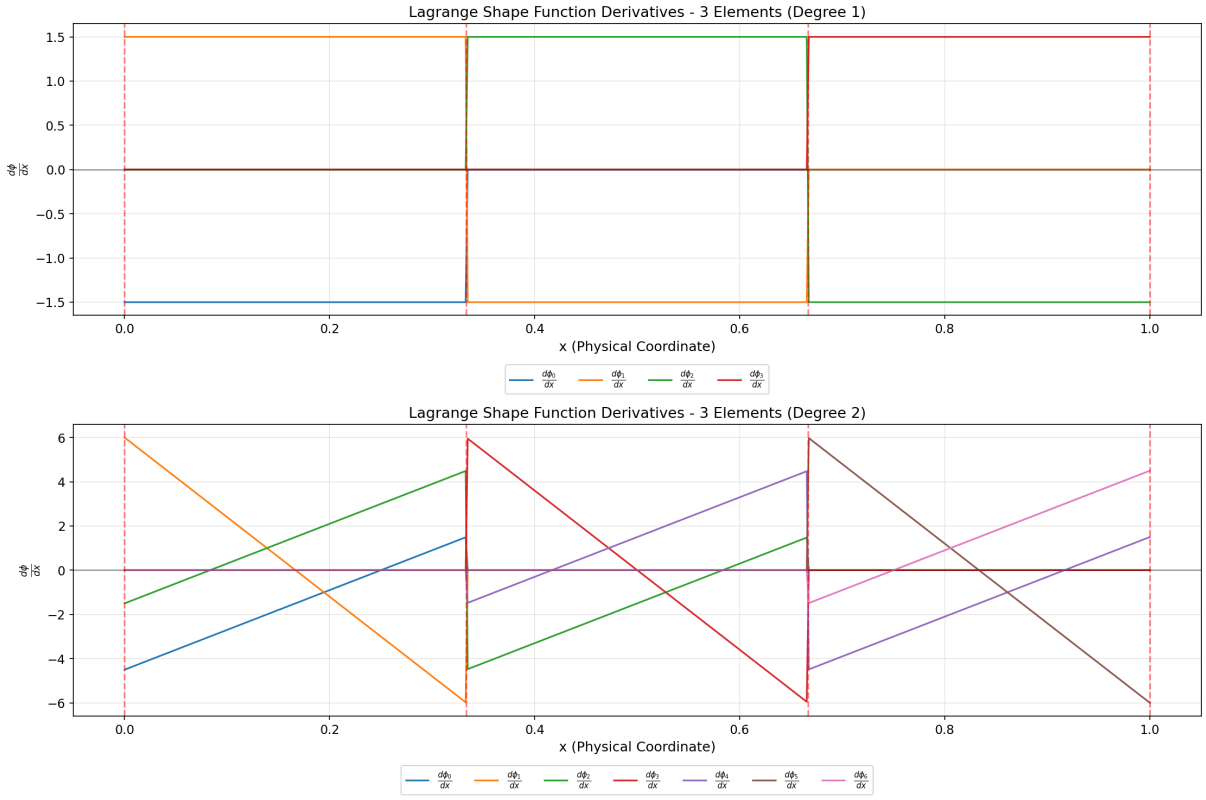


Figure 5.3.: Derivatives of Lagrange shape functions

```{python}
#| code-fold: true
#| fig-cap: "Partition of unity for linear elements"
fem_viz = FEMShapeFunctions()
fig3 = fem_viz.plot_multielement_partition_of_unity_per_element(
    num_elements=3,
    degree=1
)
```

5.10. Building the Stiffness Matrix

For example, to compute K_{11}:

$$K_{11} = \int_0^L \frac{d\phi_1}{dx} E \frac{d\phi_1}{dx}\, dx = \int_0^{x_1} E \frac{1}{h_1^2}\, dx = \frac{1}{h_1^2} \int_0^{x_1} E(x)\, dx$$

Similarly for K_{12}:

$$K_{12} = \int_0^L \frac{d\phi_1}{dx} E \frac{d\phi_2}{dx}\, dx = -\frac{1}{h_1^2} \int_0^{x_1} E(x)\, dx$$

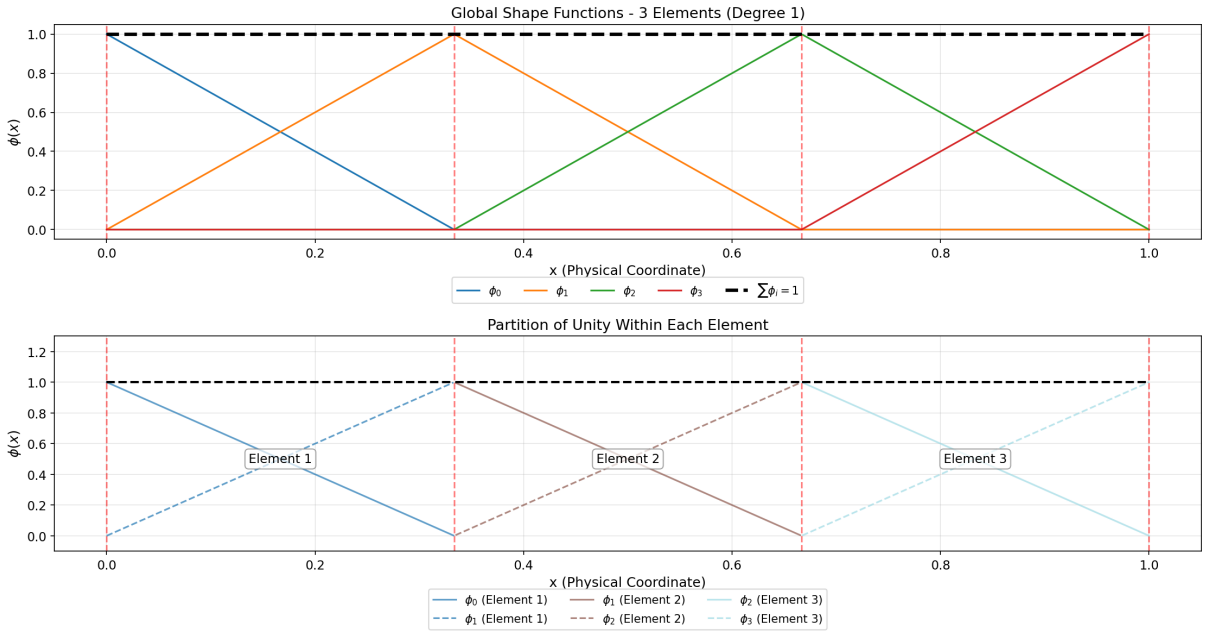


Figure 5.4.: Partition of unity for linear elements

And computing K_{13}:

$$K_{13} = \int_0^L \frac{d\phi_1}{dx} E \frac{d\phi_3}{dx}\, dx = 0$$

since ϕ_1 and ϕ_3 have no overlapping support. This pattern shows why FEM matrices have a **banded structure** — there are only local interactions between adjacent elements.

5.11. Element-wise Assembly

We recognize that:

$$K_{ij}^g = \sum_{\text{elem}} K_{ij}^e, \quad \text{where } K_{ij}^e = \int_{\Omega_e} \frac{d\phi_i}{dx} E \frac{d\phi_j}{dx}\, dx$$

$$F_i^g = \sum_{\text{elem}} F_i^e, \quad F_i^e = \int_{\Omega_e} \phi_i f\, dx + t^* \phi_i|_{\Gamma_t \cap \Omega_e}$$

This element-wise assembly is key to FEM efficiency.

5.12. Reference Element Transformation

To make calculations systematic, we define a master (reference) element in a local coordinate system $\zeta \in [-1, 1]$.

We need a mapping from reference to physical coordinates:

$$x = \sum_{i=1}^{2} \chi_i \hat{\phi}_i(\zeta) = M_x(\zeta)$$

where χ_i are the physical coordinates of the nodes and $\hat{\phi}_i(\zeta)$ are the shape functions in reference coordinates.

Shape Functions in Reference Space: For linear elements in 1D:

$$\hat{\phi}_1 = \frac{1}{2}(1-\zeta) \quad \text{and} \quad \hat{\phi}_2 = \frac{1}{2}(1+\zeta)$$

5.13. Differential Relations

We need the mapping between derivatives:

$$\frac{d}{d\zeta} = \frac{dx}{d\zeta}\frac{d}{dx} = J\frac{d}{dx}$$

And the inverse relation:

$$\frac{d}{dx} = \frac{d\zeta}{dx}\frac{d}{d\zeta} = \frac{1}{J}\frac{d}{d\zeta}$$

where $J = \frac{dx}{d\zeta}$ is the Jacobian of the transformation.

5.14. Numerical Integration: Gaussian Quadrature

For numerical integration, we use Gaussian quadrature:

$$\int_{x_i}^{x_{i+1}} F(x)\,dx = \int_{-1}^{1} F(x(\zeta))J(\zeta)\,d\zeta \approx \sum_{q=1}^{G} w_q F(x(\zeta_q))J(\zeta_q)$$

Gauss quadrature can integrate a polynomial of order $2G-1$ exactly with G points.

Gauss rule	ζ_i	w_i
2	± 0.5773502692	1.0
3	0.0	0.8888888889
	± 0.7745966692	0.5555555556

Two points are sufficient for linear elements since the integrands in K_{ij} have at most cubic terms.

5.15. Computing Element Matrices with Quadrature

The element stiffness matrix is computed as:

$$K_{ij}^e = \sum_{q=1}^{G} w_q \left[\frac{d\hat{\phi}_i}{d\zeta} \frac{d\zeta}{dx} \right] E \left[\frac{d\hat{\phi}_j}{d\zeta} \frac{d\zeta}{dx} \right] J \Big|_{\zeta=\zeta_q}$$

Similarly for the force vector:

$$F_i^e = \sum_{q=1}^{G} w_q \hat{\phi}_i(\zeta_q) f(x(\zeta_q)) J(\zeta_q)$$

plus any boundary terms.

5.16. Post-Processing

After obtaining the nodal values a_i, we can compute:

- **Displacements** at any point: $u(x) = \sum_{i=1}^{N} a_i \phi_i(x)$
- **Strains** at any point: $\varepsilon(x) = \frac{du}{dx} = \sum_{i=1}^{N} a_i \frac{d\phi_i}{dx}$
- **Stresses** at any point: $\sigma(x) = E(x) \sum_{i=1}^{N} a_i \frac{d\phi_i}{dx}$

5.17. Implementing Boundary Conditions

Essential (Dirichlet) boundary conditions (e.g., $u(0) = u^*$):

$$\begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ K_{21} & K_{22} & \dots & \dots & K_{2N} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ K_{N1} & K_{N2} & \dots & \dots & K_{NN} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} u^* \\ F_2 \\ \vdots \\ F_N + \dots \end{bmatrix}$$

6. Time-Dependent 1D Problems

6.1. Problem Introduction

In the following, we use the FEM methods for spatial discretization and the finite difference method for time discretization.

Consider the Balance of Linear momentum equation in 1D:

$$\begin{aligned} \text{grad} \cdot \sigma + f(x,t) &= \rho_0 \frac{\partial^2 u}{\partial t^2} \\ &= \rho_0 \frac{\partial v}{\partial t} \\ &= \rho_0 a(x,t) \end{aligned}$$

where:

- $\sigma(x,t)$ is the stress
- $f(x,t)$ is the body force
- ρ_0 is the density
- u is the displacement
- v is the velocity
- a is the acceleration

6.2. Single Point in Space

For a specific point in space, we can write the equation as:

$$m\dot{v} = F = ma \quad \text{(I)}$$

where all the forces acting on the point are lumped into a single force F and the mass is $m = \rho_0 \Delta x$.

In the following, we will use the Taylor expansion around the time $t + \theta\Delta t$ to expand the quantities of interest at times t and $t + \Delta t$ where θ is a parameter that can be chosen to optimize the time integration scheme.

6.2.1. Velocity Expansions

For the velocities:

$$v(t+\Delta t) = v(t+\theta\Delta t) + \left.\frac{dv}{dt}\right|_{t+\theta\Delta t}(1-\theta)\Delta t + \frac{1}{2}\left.\frac{d^2v}{dt^2}\right|_{t+\theta\Delta t}(1-\theta)^2(\Delta t)^2 + O(\Delta t)^3 \quad (2.1)$$

$$v(t) = v(t+\theta\Delta t) - \left.\frac{dv}{dt}\right|_{t+\theta\Delta t}\theta\Delta t + \frac{1}{2}\left.\frac{d^2v}{dt^2}\right|_{t+\theta\Delta t}\theta^2(\Delta t)^2 + O(\Delta t)^3 \quad (2.2)$$

6.2.2. Position Expansions

For the position:

$$u(t+\Delta t) = u(t+\theta\Delta t) + \left.\frac{du}{dt}\right|_{t+\theta\Delta t}(1-\theta)\Delta t + \frac{1}{2}\left.\frac{d^2u}{dt^2}\right|_{t+\theta\Delta t}(1-\theta)^2(\Delta t)^2 + O(\Delta t)^3 \quad (2.3)$$

$$u(t) = u(t+\theta\Delta t) - \left.\frac{du}{dt}\right|_{t+\theta\Delta t}\theta\Delta t + \frac{1}{2}\left.\frac{d^2u}{dt^2}\right|_{t+\theta\Delta t}\theta^2(\Delta t)^2 + O(\Delta t)^3 \quad (2.4)$$

6.2.3. Force Averaging

For the Force we obtain a weighted average of the forces at the two time steps:

$$F(t+\theta\Delta t) = \theta F(t+\Delta t) + (1-\theta)F(t) \quad (2.5)$$

6.2.4. Velocity Time Derivative

Subtracting Equations (2.1) and (2.2) we obtain:

$$\left.\frac{dv}{dt}\right|_{t+\theta\Delta t} = \frac{v(t+\Delta t) - v(t)}{\Delta t} + \hat{O}(\Delta t) \quad (2.6)$$

where $\hat{O}(\Delta t)$ is the error term that depends on the choice of θ. For $\theta = 1/2$ we obtain $\hat{O} = O(\Delta t)^2$ and otherwise we get $\hat{O} = O(\Delta t)$.

By using equation (2.6) and inserting it into equation (I) we obtain:

$$v(t+\Delta t) = v(t) + \frac{\Delta t}{m}F(t+\theta\Delta t) + \hat{O}(\Delta t)^2 \quad (2.7)$$

6.2.5. Velocity at Intermediate Time

If we take the weighted sum of (2.1) and (2.2) we obtain:

$$v(t+\theta\Delta t) = \theta v(t+\Delta t) + (1-\theta)v(t) + O(\Delta t)^2 \quad (2.8)$$

6.2.6. Position Time Derivative

We can do the same for the positions and obtain:

$$\frac{u(t+\Delta t)-u(t)}{\Delta t} = v(t+\theta\Delta t) + \hat{O}(\Delta t) \quad (2.9)$$

Which combined with equation (2.8) gives us:

$$u(t+\Delta t) = u(t) + (\theta v(t+\Delta t) + (1-\theta)v(t))\Delta t + \hat{O}(\Delta t)^2 \quad (2.10)$$

6.2.7. Final Position Update

Now, we can use equations (2.10) and (2.7) to obtain:

$$u(t+\Delta t) = u(t) + v(t)\Delta t + \frac{\theta(\Delta t)^2}{m}F(t+\theta\Delta t) + \hat{O}(\Delta t)^2 \quad (2.11)$$

6.2.8. Time Integration Schemes

When $\theta = 1$ we obtain an implicit integration scheme (backward Euler) and when $\theta = 0$ we obtain an explicit integration scheme (forward Euler).

The backward Euler scheme is unconditionally stable, while the forward Euler scheme is conditionally stable and requires a small time step to ensure stability.

For both we obtain $\hat{O}(\Delta t)^2 = O(\Delta t)^2$.

For $\theta = 1/2$ we obtain the "midpoint rule" which is unconditionally stable and $\hat{O}(\Delta t)^2 = O(\Delta t)^3$.

6.3. FEM Formulation

The continuum Equation we try to solve is:

$$\rho_0\frac{\partial^2 u}{\partial t^2} = \rho_0\frac{\partial v}{\partial t} = \nabla\cdot\sigma + f(x,t) \quad (3.0)$$

We will make our life easier by defining $F = \nabla\cdot\sigma + f(x,t)$.

Using the equations we derived above we can reformulate the problem as:

$$\rho_0 v(t+\Delta t) = \rho_0 v(t) + \Delta t(\theta F(t+\Delta t) + (1-\theta)F(t)) \quad (3.1)$$

6.3.1. Weak Form Development

Multiplying by the test function and integrating by parts we obtain:

$$\int_\Omega v \cdot \rho_0 v(t+\Delta t)\, d\Omega = \int_\Omega v \cdot \rho_0 v(t)\, d\Omega + \Delta t \int_\Omega v \cdot (\theta \Psi(t+\Delta t) + (1-\theta)\Psi(t))\, d\Omega \quad (3.2)$$

And after applying the divergence theorem and enforcing $v = 0$ on the boundary we obtain:

$$\begin{aligned}\int_\Omega v \cdot \rho_0 v^{t+\Delta t}\, d\Omega = \int_\Omega v \cdot \rho_0 v^t\, d\Omega \\ + \Delta t\theta \left(- \int_\Omega \nabla v : \sigma\, d\Omega + \int_{\Gamma_t} v \cdot (\sigma \cdot n)\, dA + \int_\Omega v \cdot f\, d\Omega \right)^{t+\Delta t} \\ + \Delta t(1-\theta) \left(- \int_\Omega \nabla v : \sigma\, d\Omega + \int_{\Gamma_t} v \cdot t^*\, dA + \int_\Omega v \cdot f\, d\Omega \right)^{t} \end{aligned} \quad (3.3)$$

6.3.2. Discretized System

Plugging in our shape function, and defining the standard matrices with $\{a\}$ being the coefficients vector, we obtain:

$$[M]\{a\}^{t+\Delta t} = [M]\{a\}^t + (\Delta t\theta)\left(-[K]\{a\}^{t+\Delta t} + \{R_f\}^{t+\Delta t} + \{R_t\}^{t+\Delta t}\right) + \Delta t(1-\theta)\left(-[K]\{a\}^t + \{R_f\}^t + \{R_t\}^t\right)$$

where we use:

$$\{a\}^{t+\Delta t} = \{a\}^t + \Delta t(\theta\{\dot{a}\})^{t+\Delta t} + (1-\theta)\{\dot{a}\}^t \quad (3.5)$$

6.4. Implicit Backward Euler Scheme

For $\theta = 1$ we get the implicit backward Euler scheme:

$$\left([M]\{\ddot{a}\}^{t+\Delta t} + \Delta t[K]\{a\}^{t+\Delta t}\right) = [M]\{\ddot{a}\}^t + \Delta t\left(\{R_t\}^{t+\Delta t} + \{R_f\}^{t+\Delta t}\right) \quad (3.6)$$

where we use:

$$\{a\}^{t+\Delta t} = \{a\}^t + \Delta t(\{\dot{a}\})^{t+\Delta t} \quad (3.7)$$

6.5. Explicit Forward Euler Scheme

For $\theta = 0$ we get the explicit forward Euler scheme:

$$\{\dot{a}\}^{t+\Delta t} = \{\dot{a}\}^t + \Delta t[M]^{-1}\left(-[K]\{a\}^t + \{R_f\}^t + \{R_t\}^t\right) \qquad (3.8)$$

where we use:

$$\{a\}^{t+\Delta t} = \{a\}^t + \Delta t\{\dot{a}\}^t \qquad (3.9)$$

Part III.

Part III: Multi-Dimensional FEM

7. From 1D to Multi-Dimensional FEM

7.1. Mathematical Preliminaries

Looking at the following equation:

$$\text{div}(\mathbf{K} \cdot \text{grad}(T))$$

if $\mathbf{K}$ is a constant multiplied by the identity matrix, we can write it as:

$$\text{div}(\mathbf{K} \cdot \text{grad}(T)) = \text{div}(\mathbf{K} \left[\frac{\partial T}{\partial x}\mathbf{e}_1 + \frac{\partial T}{\partial y}\mathbf{e}_2 + \frac{\partial T}{\partial z}\mathbf{e}_3\right]) = \mathbf{K}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) = \mathbf{K}\nabla^2 T$$

7.1.1. Temperature as a Scalar Field

$$T \text{ - Scalar} \Rightarrow \text{grad}(T) = \left(\frac{\partial T}{\partial x}\mathbf{e_1} + \frac{\partial T}{\partial y}\mathbf{e_2} + \frac{\partial T}{\partial z}\mathbf{e_3}\right) \text{ - vector}$$

7.1.2. Conductivity Matrix Operation

$$\mathbf{K} \cdot \text{grad}(T) = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \text{transpose}\left(\left(\frac{\partial T}{\partial x}\mathbf{e_1} + \frac{\partial T}{\partial y}\mathbf{e_2} + \frac{\partial T}{\partial z}\mathbf{e_3}\right)\right)$$

7.1.3. Heat Flux Vector

Let's define $\mathbf{q} = -\mathbf{K} \cdot \text{grad}(T)$ as the heat flux vector.

Divergence of heat flux:

$$\text{div}(\mathbf{q}) = \frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + \frac{\partial q_z}{\partial z}$$

7.2. The Product Rule for Divergence

To answer the question "What happens when we multiply the heat conduction equation by a test function V?", we need to use the **product rule for divergence**:

$$\text{div}(V\mathbf{K}\ \text{grad}(T)) = \text{grad}(V) \cdot \mathbf{K} \cdot \text{grad}(T) + V\ \text{div}(\mathbf{K} \cdot \text{grad}(T))$$

However, if $\mathbf{K} = k$ is constant, we can simplify this to:

$$\text{div}(V\mathbf{K}\ \text{grad}(T)) = k\text{div}(V\ \text{grad}(T)) = k(\text{grad}(V) \cdot \text{grad}(T) + V\nabla^2(T))$$

7.2.1. Detailed Expansion

Let's expand $\text{div}(V\mathbf{K}\ \text{grad}(T))$ step by step:

$$\text{div}(V\mathbf{K}\ \text{grad}(T)) = \frac{\partial}{\partial x}\left(VK\frac{\partial T}{\partial x}\right) + \frac{\partial}{\partial y}\left(VK\frac{\partial T}{\partial y}\right) + \frac{\partial}{\partial z}\left(VK\frac{\partial T}{\partial z}\right)$$

Applying the product rule to each term:

$$= \frac{\partial V}{\partial x}K\frac{\partial T}{\partial x} + \frac{\partial V}{\partial y}K\frac{\partial T}{\partial y} + \frac{\partial V}{\partial z}K\frac{\partial T}{\partial z} + VK\frac{\partial^2 T}{\partial x^2} + VK\frac{\partial^2 T}{\partial y^2} + VK\frac{\partial^2 T}{\partial z^2}$$

We can group these terms into two parts:

Gradient Terms:

$$\text{grad}(V) \cdot \mathbf{K} \cdot \text{grad}(T) = \left(\frac{\partial V}{\partial x}, \frac{\partial V}{\partial y}, \frac{\partial V}{\partial z}\right) \cdot \mathbf{K} \cdot \left(\frac{\partial T}{\partial x}, \frac{\partial T}{\partial y}, \frac{\partial T}{\partial z}\right)$$

For constant K, this becomes:

$$= K\left(\frac{\partial V}{\partial x}\frac{\partial T}{\partial x} + \frac{\partial V}{\partial y}\frac{\partial T}{\partial y} + \frac{\partial V}{\partial z}\frac{\partial T}{\partial z}\right) = A$$

Second Derivative Terms:

$$V\ \text{div}(\mathbf{K} \cdot \text{grad}(T)) = VK\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) = B$$

7.2.2. The Key Relationship

From the product rule, we have:

$$\operatorname{div}(V\mathbf{K}\ \operatorname{grad}(T)) = A + B$$

Therefore:

$$B = \operatorname{div}(V\mathbf{K}\ \operatorname{grad}(T)) - A$$

Which gives us:

$$\operatorname{div}(\mathbf{K}\ \operatorname{grad}(T))V = \operatorname{div}(V\mathbf{K}\ \operatorname{grad}(T)) - \operatorname{grad}(V) \cdot \mathbf{K} \cdot \operatorname{grad}(T)$$

7.2.3. Physical Interpretation

This mathematical identity is the foundation for:

1. **Integration by parts** in the weak formulation
2. **Converting strong form to weak form** in finite element methods
3. **Moving derivatives from the solution to the test function**

The term $\operatorname{div}(V\mathbf{K}\ \operatorname{grad}(T))$ will become a boundary integral when integrated over the domain, while $\operatorname{grad}(V) \cdot \mathbf{K} \cdot \operatorname{grad}(T)$ becomes the bilinear form in the finite element formulation.

7.2.4. Weak Form Foundation

When integrated over a domain Ω, this relationship becomes:

$$\int_\Omega V\ \operatorname{div}(\mathbf{K}\ \operatorname{grad}(T))\, d\Omega = \int_\Omega \operatorname{div}(V\mathbf{K}\ \operatorname{grad}(T))\, d\Omega - \int_\Omega \operatorname{grad}(V) \cdot \mathbf{K} \cdot \operatorname{grad}(T)\, d\Omega$$

Using the divergence theorem, the first integral on the right becomes a boundary integral, leading to the standard weak form used in finite element analysis.

7.3. FEM in 2D/3D: Heat Equation

7.3.1. Problem Setup

Governing equation: $\operatorname{div}(\mathbf{q}) = 0$ where we assumed that there is no source term for simplicity.

Constitutive equation (Fourier's law): $\mathbf{q} = -\mathbf{K} \cdot \operatorname{grad}(T)$

Boundary conditions:

- $\Gamma_{T_0} : T = T_0$ (Essential BC)
- $\Gamma_{T_1} : T = T_1$ (Essential BC)
- $\Gamma_q : \mathbf{q} = \mathbf{q}_n^*$ (Natural BC)

where $\mathbf{q}_n^*$ is the prescribed heat flux on the boundary Γ_q in the direction of the (outwards) normal to the surface.

Combined form: $\text{div}(\mathbf{q}) = -\text{div}(\mathbf{K} \cdot \text{grad}(T)) = 0$

7.4. Weak Form Derivation

Step 1: Multiply by test function V and integrate over domain

$$\int_\Omega \text{div}(\mathbf{K} \cdot \text{grad}(T)) V \, d\Omega = 0$$

Step 2: Apply the product rule result

$$\int_\Omega \text{div}(V\mathbf{K} \cdot \text{grad}(T)) \, d\Omega - \int_\Omega \text{grad}(V) \cdot \mathbf{K} \cdot \text{grad}(T) \, d\Omega = 0$$

Step 3: Apply divergence theorem to first integral

$$\int_\Omega \text{div}(V\mathbf{K} \cdot \text{grad}(T)) \, d\Omega = \int_\Omega \text{div}(V\mathbf{q}) \, d\Omega = \int_{\partial\Omega} V\mathbf{q} \cdot \mathbf{n} \, d\Gamma$$

Final weak form:

$$\int_{\partial\Omega} V\mathbf{q} \cdot \mathbf{n} \, d\Gamma - \int_\Omega \text{grad}(V) \cdot \mathbf{K} \cdot \text{grad}(T) \, d\Omega = 0$$

Finite element approximation:

$$V_h = \sum_{i=1}^{N} b_i \phi_i(x, y), \quad T_h = \sum_{j=1}^{N} a_j \phi_j(x, y) \Rightarrow \{a_j\}$$

7.5. 2D/3D Finite Elements: Geometry and Shape Functions

7.5.1. Transition from 1D to 2D/3D

Key differences:

- **1D:** Geometric description is simpler - element domain is simply a line
- **2D/3D:** Geometry plays a crucial role and complexity increases significantly

Most common finite element geometries:

- **2D:** Triangular elements and Rectangular elements
- **3D:** Tetrahedron elements and Hexahedron elements

7.5.2. 2D Rectangular Elements: Linear Shape Functions

Design requirements for shape functions:

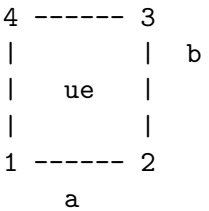
- **Simplicity:** Describe change in the element's domain with minimal complexity
- **Local support:** Each shape function corresponds to one node, value of "1" at one node, "0" at all other nodes

1D analogy: $u^e = a_i\phi_i^g + a_{i+1}\phi_{i+1}^g$ (where $u^e \in [x_i, x_{i+1}]$)

2D extension: For a linear rectangular element, we need **four shape functions** to describe u^e

7.5.3. Rectangular Element with Linear Shape Functions

Consider a rectangular element with nodes numbered as follows:



Shape functions for specific rectangular element:

$$\phi_1^g = \frac{1}{ab}(a-x)(b-y)$$

$$\phi_2^g = \frac{1}{ab}x(b-y)$$

$$\phi_3^g = \frac{1}{ab}xy$$

$$\phi_4^g = \frac{1}{ab}(a-x)y$$

Verification:

- $\phi_1^g(0,0) = 1$, $\phi_1^g(a,0) = 0$, $\phi_1^g(0,b) = 0$, $\phi_1^g(a,b) = 0$ ✓
- $\phi_2^g(0,0) = 0$, $\phi_2^g(a,0) = 1$, $\phi_2^g(0,b) = 0$, $\phi_2^g(a,b) = 0$ ✓

Element solution: $u^e = a_1\phi_1^g + a_2\phi_2^g + a_3\phi_3^g + a_4\phi_4^g = u^e(x,y)$

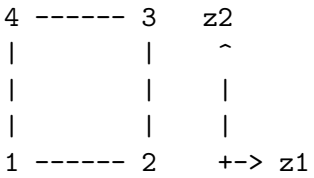
7.6. Master Element Concept in 2D

7.6.1. Mapping to Master Element

Purpose: Map general elements to a standardized **master element** for easier computation

Coordinate transformation: $[x, y] \to [\zeta_1, \zeta_2] \in [-1, 1] \times [-1, 1]$

Master element node numbering convention:



Master element shape functions:

$$\hat{\phi}_1 = \phi_1^e(\zeta_1, \zeta_2) = \frac{1}{4}(1 - \zeta_1)(1 - \zeta_2)$$

$$\hat{\phi}_2 = \phi_2^e(\zeta_1, \zeta_2) = \frac{1}{4}(1 + \zeta_1)(1 - \zeta_2)$$

$$\hat{\phi}_3 = \phi_3^e(\zeta_1, \zeta_2) = \frac{1}{4}(1 + \zeta_1)(1 + \zeta_2)$$

$$\hat{\phi}_4 = \phi_4^e(\zeta_1, \zeta_2) = \frac{1}{4}(1 - \zeta_1)(1 + \zeta_2)$$

Bi-linear nature: Each $\hat{\phi}_i$ equals "1" at the i-th node and "0" at other nodes, creating **bi-linear** shape functions.

Example expansion: $\hat{\phi}_1 = \frac{1}{4}(1 - \zeta_1)(1 - \zeta_2) = \frac{1}{4}(1 - \zeta_1 - \zeta_2 + \zeta_1\zeta_2)$

The term $\zeta_1\zeta_2$ makes it bi-linear (linear in both coordinates).

7.7. Jacobian and Coordinate Transformation

7.7.1. Jacobian Matrix

Definition:

$$\mathbf{F} = \begin{bmatrix} \frac{\partial x}{\partial \zeta_1} & \frac{\partial x}{\partial \zeta_2} \\ \frac{\partial y}{\partial \zeta_1} & \frac{\partial y}{\partial \zeta_2} \end{bmatrix}, \quad J = \mathrm{Det}(\mathbf{F}) = \frac{\partial x}{\partial \zeta_1}\frac{\partial y}{\partial \zeta_2} - \frac{\partial x}{\partial \zeta_2}\frac{\partial y}{\partial \zeta_1}$$

Critical requirement: $J > 0$ throughout the element for a "good" mapping

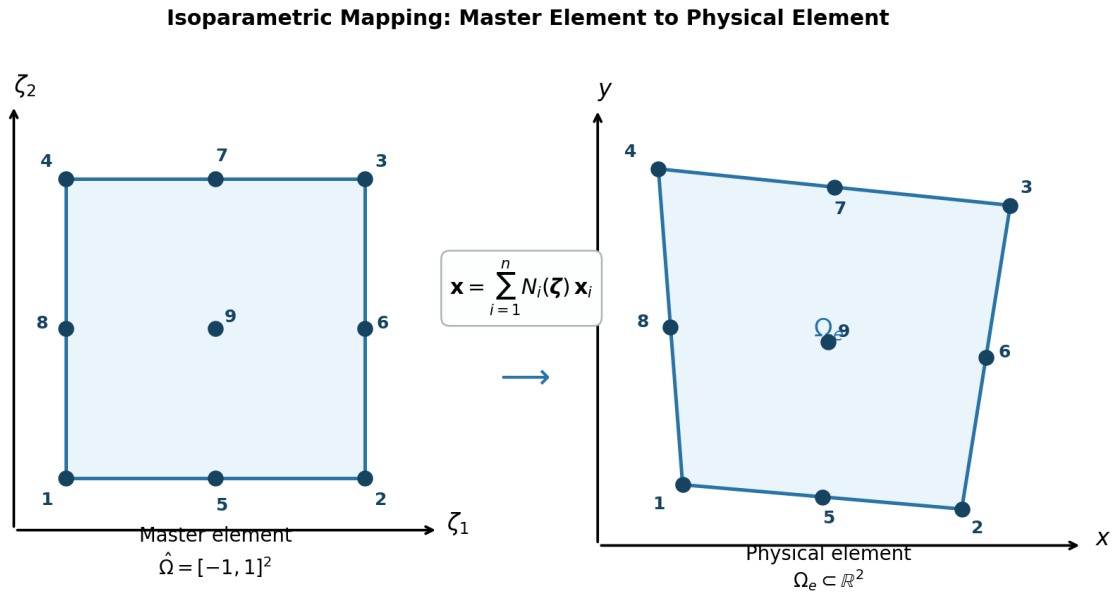


Figure 7.1.: Isoparametric mapping from the master element $\hat{\Omega} = [-1, 1]^2$ in reference coordinates (ζ_1, ζ_2) to a physical element Ω_e in global coordinates (x, y) via the interpolation $\mathbf{x} = \sum_i N_i(\boldsymbol{\zeta})\, \mathbf{x}_i$.

7.7.2. Isoparametric Mapping

We will use **isoparametric** mapping, where the same shape functions are used for both geometry and field variables.

Coordinate transformation:

$$x(\chi_1, \chi_2) = \sum_{i=1}^{4} \chi_{1i} \hat{\phi}_i = \chi_{11}\hat{\phi}_1 + \chi_{12}\hat{\phi}_2 + \chi_{13}\hat{\phi}_3 + \chi_{14}\hat{\phi}_4$$

$$y(\chi_1, \chi_2) = \sum_{i=1}^{4} \chi_{2i} \hat{\phi}_i$$

Where χ_{1i} are global x-coordinates and χ_{2i} are global y-coordinates of element nodes.

7.8. Element Quality: Good vs Bad Elements

7.8.1. Acceptable Elements

Case 1 - Rectangular element: $0 < J(\zeta_1, \zeta_2) < \infty$ throughout element

- Jacobian is constant
- Acceptable

Case 3 - Distorted but convex: $0 < J(\zeta_1, \zeta_2) < \infty$ throughout element

- Jacobian varies but remains positive and bounded
- Acceptable

7.8.2. Unacceptable Elements

Case 2 - Incorrect node numbering: $J(\zeta_1, \zeta_2) < 0$ throughout element

- Nodes numbered incorrectly, turning element "inside out"
- Unacceptable

Case 4 - Partially negative Jacobian: $J(\zeta_1, \zeta_2) < 0$ in some regions

- Can cause singularities in stiffness matrix
- Unacceptable

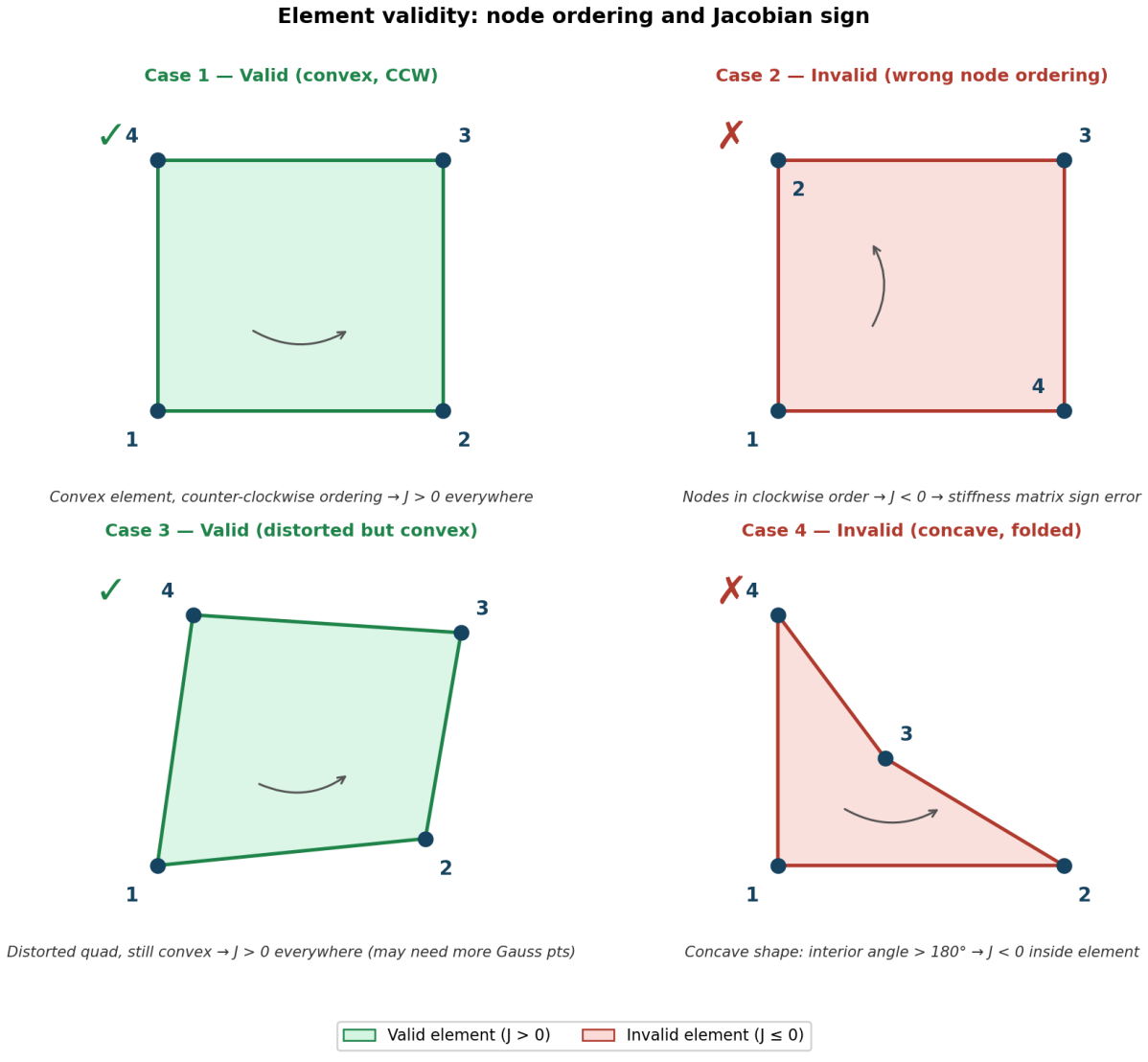


Figure 7.2.: Element validity: Cases 1 and 3 are valid (convex, CCW node ordering, $J > 0$ everywhere). Case 2 fails due to clockwise node ordering ($J < 0$). Case 4 fails because the concave (folded) shape causes $J < 0$ inside the element.

Key insight for linear elements: The primary indicator of problematic elements is **non-convexity**, even with correct numbering.

7.9. Bookkeeping in 2D Finite Elements

7.9.1. Difference from 1D

1D: Node numbering is intuitive and element connectivity is straightforward

```
1 -- 2 -- 3 -- 4 -- 5 -- 6 -- 7 -- 8 -- 9
```

2D: More complex connectivity patterns require systematic bookkeeping

7.9.2. Element Connectivity Tables

Element-to-node connectivity:

Element	Node 1	Node 2	Node 3	Node 4
1	20	21	2	1
2	21	22	3	2

Node coordinate table:

Node	X-coord	Y-coord
1	X_1	Y_1
2	X_2	Y_2
...	...	...

Importance: Essential for assembling the global stiffness matrix $[K_{ij}^g]$ and load vector $\{F_i^g\}$, and for enforcing boundary conditions.

7.10. Shape Functions: Differential Properties

7.10.1. Coordinate Transformation Relations

Forward transformation ($\zeta \to \mathbf{x}$):

$$\frac{\partial}{\partial \zeta_1} = \frac{\partial}{\partial x}\frac{\partial x}{\partial \zeta_1} + \frac{\partial}{\partial y}\frac{\partial y}{\partial \zeta_1}$$

$$\frac{\partial}{\partial \zeta_2} = \frac{\partial}{\partial x}\frac{\partial x}{\partial \zeta_2} + \frac{\partial}{\partial y}\frac{\partial y}{\partial \zeta_2}$$

Inverse transformation ($\mathbf{x} \to \zeta$):

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial \zeta_1}\frac{\partial \zeta_1}{\partial x} + \frac{\partial}{\partial \zeta_2}\frac{\partial \zeta_2}{\partial x}$$

$$\frac{\partial}{\partial y} = \frac{\partial}{\partial \zeta_1}\frac{\partial \zeta_1}{\partial y} + \frac{\partial}{\partial \zeta_2}\frac{\partial \zeta_2}{\partial y}$$

Matrix form:

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \mathbf{F} \begin{bmatrix} d\zeta_1 \\ d\zeta_2 \end{bmatrix}, \quad \begin{bmatrix} d\zeta_1 \\ d\zeta_2 \end{bmatrix} = \mathbf{F}^{-1} \begin{bmatrix} dx \\ dy \end{bmatrix}$$

7.10.2. Gradient Calculation in Master Element

For test function V:

$$\text{grad}(V) = \begin{bmatrix} \frac{\partial V}{\partial x} \\ \frac{\partial V}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial \zeta_1}{\partial x} & \frac{\partial \zeta_2}{\partial x} \\ \frac{\partial \zeta_1}{\partial y} & \frac{\partial \zeta_2}{\partial y} \end{bmatrix}^T \begin{bmatrix} \frac{\partial V}{\partial \zeta_1} \\ \frac{\partial V}{\partial \zeta_2} \end{bmatrix}$$

For temperature field T:

$$\text{grad}(T) = \begin{bmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial \zeta_1}{\partial x} & \frac{\partial \zeta_2}{\partial x} \\ \frac{\partial \zeta_1}{\partial y} & \frac{\partial \zeta_2}{\partial y} \end{bmatrix}^T \begin{bmatrix} \frac{\partial T}{\partial \zeta_1} \\ \frac{\partial T}{\partial \zeta_2} \end{bmatrix}$$

8. The B Matrix

8.1. Extension to 3D Vector Problems

8.1.1. Linear Elasticity Example

Governing equation: $\operatorname{div}(\boldsymbol{\sigma}) + \mathbf{f} = 0$

Constitutive relation: $\boldsymbol{\sigma} = \mathbf{E} \cdot \boldsymbol{\varepsilon} = \mathbf{E} \cdot \operatorname{grad}^s(\mathbf{u})$

Divergence theorem in 3D:

$$\int_\Omega \operatorname{div}(\boldsymbol{\sigma})\, d\Omega = \int_{\partial\Omega} \boldsymbol{\sigma} \cdot \mathbf{n}\, da$$

Key extensions for 3D:

- **Vector unknowns:** Displacement field $\mathbf{u} = [u_x, u_y, u_z]^T$
- **Matrix operations:** Stress and strain tensors
- **Global/local transformations:** More complex connectivity
- **Vector problem connectivity:** Multiple DOFs per node

8.2. 3D Element Example

8.3. The B Matrix: Relating Nodal Values to Field Gradients

In the Finite Element Method, we approximate a continuous field (like temperature T or displacement u) within an element using shape functions N_j and nodal values. For example, the temperature T(x,y) at any point within an element can be approximated as:

$$T(x, y) \approx \sum_{j=1}^{n_{\text{nodes}}} N_j(x, y) a_j = \mathbf{N}\mathbf{a}^e$$

where $N_j(x, y)$ are the shape functions, a_j are the nodal temperatures for the element, $\mathbf{N}$ is the row vector of shape functions, and $\mathbf{a}^e$ is the column vector of nodal temperatures for the element.

The physical behavior of the system (e.g., heat flux, stress/strain) often depends on the gradient of this field. The B matrix is a cornerstone of FEM as it directly relates the nodal values of an element to the gradient of the field (or its derivatives) within that element.

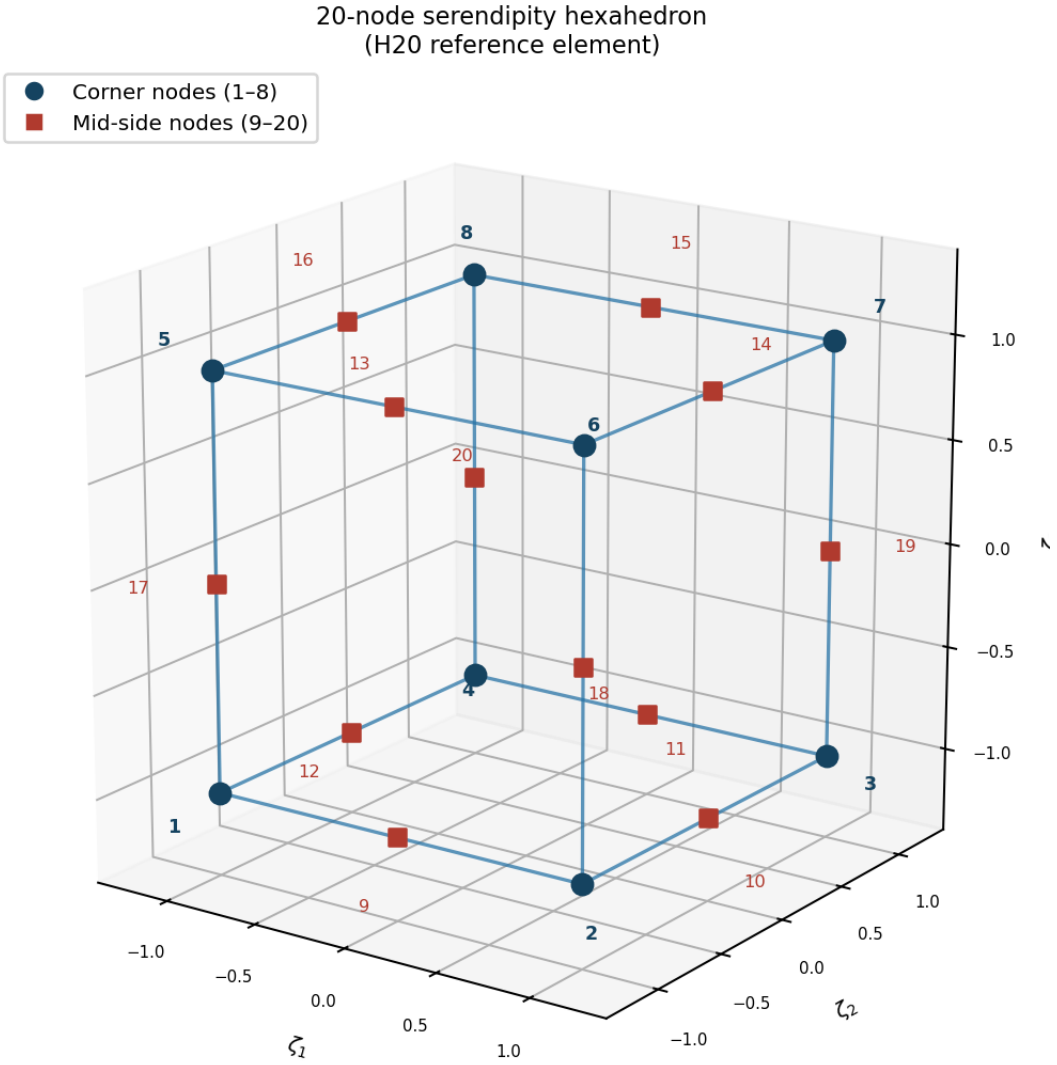


Figure 8.1.: 20-node hexahedron element

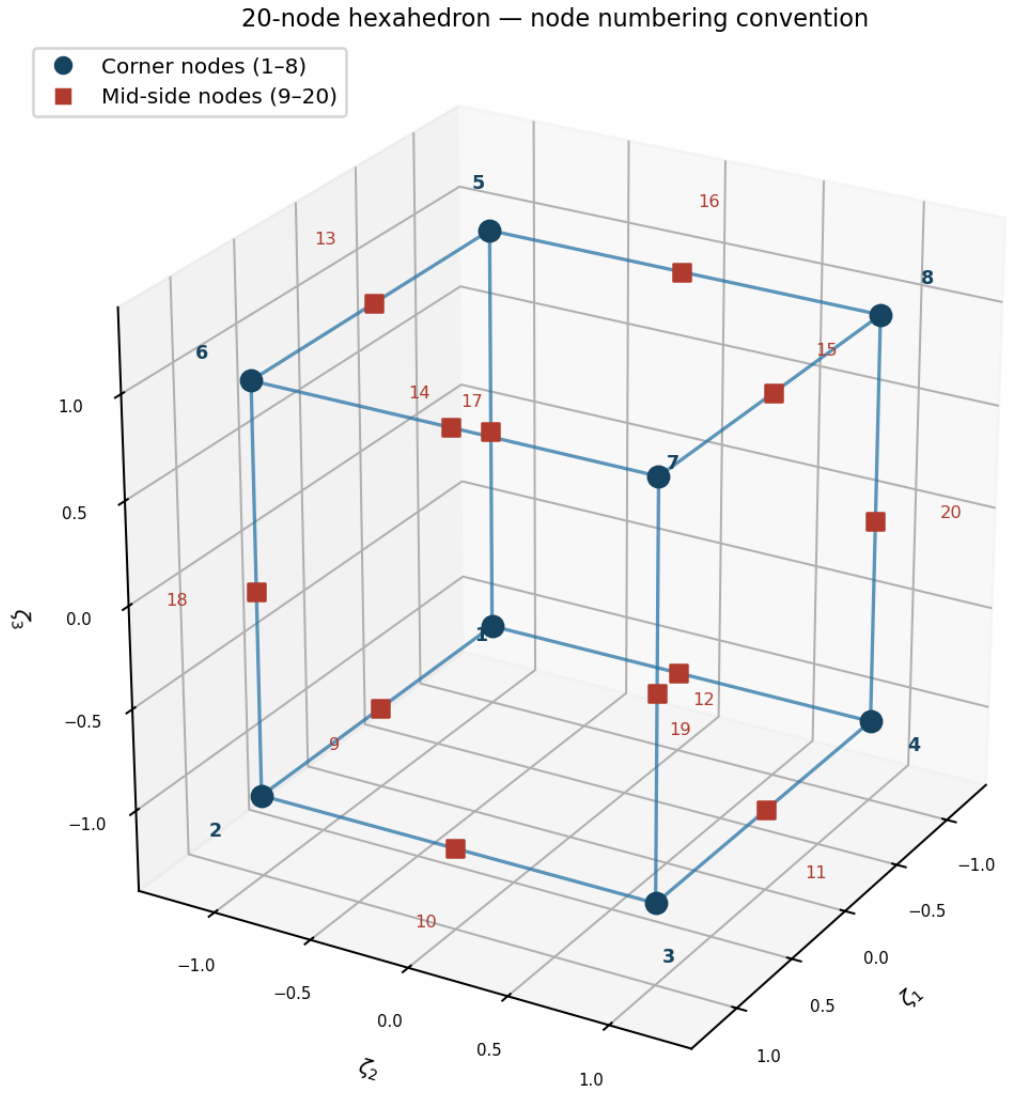


Figure 8.2.: 20-node hexahedron element (refined)

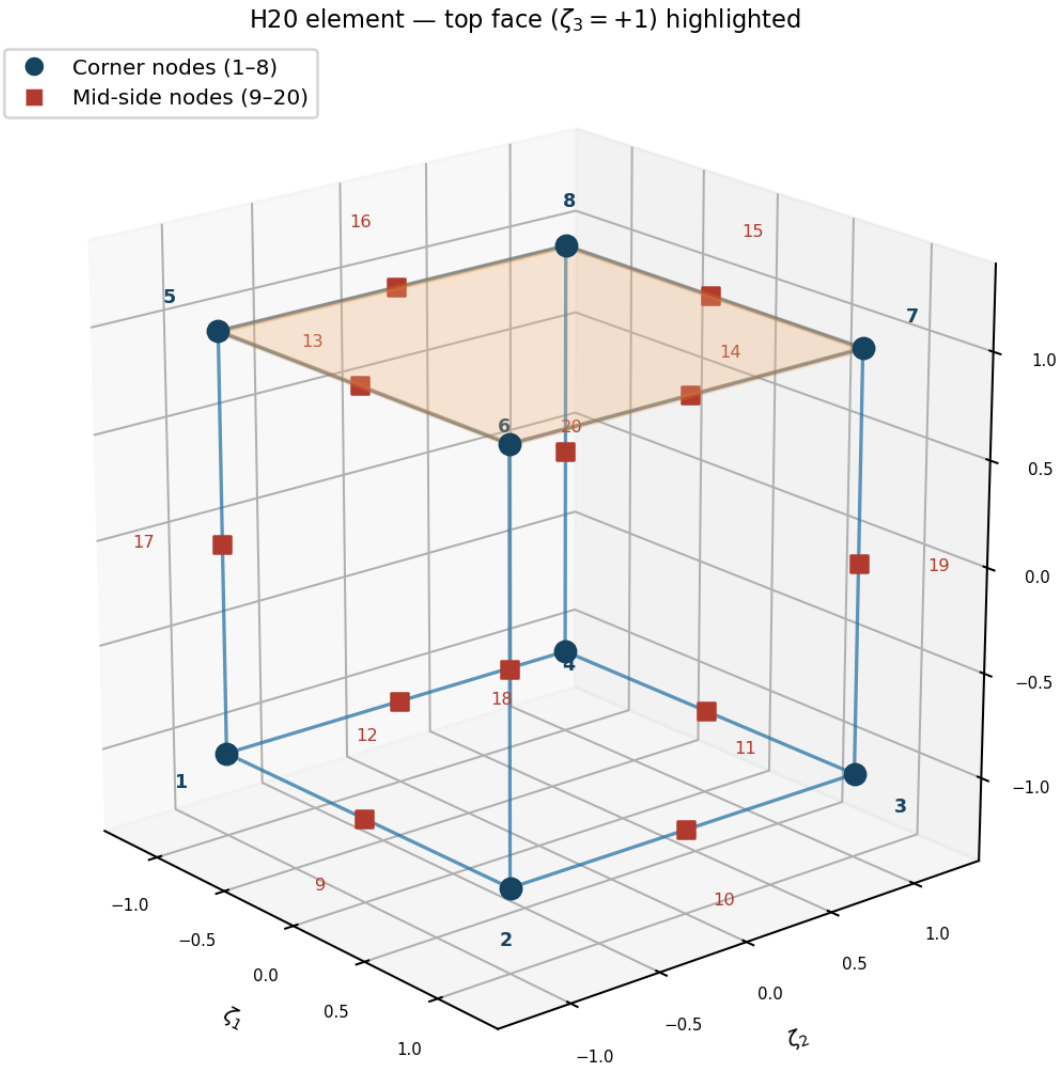


Figure 8.3.: 20-node hexahedron element (field view)

8.4. For Scalar Field Problems (Heat Conduction)

In heat conduction, the heat flux $\mathbf{q}$ is related to the temperature gradient $\text{grad}(T)$ by Fourier's Law: $\mathbf{q} = -\mathbf{K} \cdot \text{grad}(T)$.

The weak form integral that leads to the stiffness matrix, $\int_{\Omega^e} \text{grad}(V) \cdot \mathbf{K} \cdot \text{grad}(T)\, d\Omega$, requires evaluating these gradients.

Let $T = \mathbf{N}\mathbf{a}^e$. The temperature gradient in 2D is:

$$\text{grad}(T) = \begin{Bmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \end{Bmatrix} = \begin{Bmatrix} \frac{\partial (\sum N_j a_j)}{\partial x} \\ \frac{\partial (\sum N_j a_j)}{\partial y} \end{Bmatrix} = \sum_{j=1}^{n_{\text{nodes}}} \begin{bmatrix} \frac{\partial N_j}{\partial x} \\ \frac{\partial N_j}{\partial y} \end{bmatrix} a_j$$

This can be written in matrix form as:

$$\text{grad}(T) = \mathbf{B}\mathbf{a}^e$$

Where the B matrix for a 2D scalar problem (like heat conduction) for an element with n_{nodes} is:

$$\mathbf{B} = \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_{n_{\text{nodes}}}}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \cdots & \frac{\partial N_{n_{\text{nodes}}}}{\partial y} \end{bmatrix}$$

Each column j of the B matrix corresponds to node j and contains the partial derivatives of its shape function N_j with respect to the global coordinates. For a 3D scalar problem, another row for $\frac{\partial N_j}{\partial z}$ would be added.

8.5. Role in Element Stiffness Matrix

The element stiffness matrix for heat conduction, derived from the weak form, is given by:

$$[\mathbf{k}^e] = \int_{\Omega^e} \mathbf{B}^T \mathbf{K} \mathbf{B}\, d\Omega$$

Here, $\mathbf{K}$ is the material property matrix (thermal conductivity tensor). For an isotropic material in 2D with scalar conductivity k, $\mathbf{K} = \begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix}$. The integration is performed over the volume (or area in 2D) of the element Ω^e.

8.6. Calculation of B Matrix using Master Element Coordinates

Shape functions N_j are typically defined in local (master element) coordinates (ζ_1, ζ_2) (often denoted ξ, η). Their derivatives with respect to global coordinates (x,y) are found using the chain rule and the Jacobian matrix $\mathbf{F}$ of the coordinate transformation:

$$\begin{Bmatrix} \frac{\partial N_j}{\partial x} \\ \frac{\partial N_j}{\partial y} \end{Bmatrix} = \mathbf{F}^{-T} \begin{Bmatrix} \frac{\partial N_j}{\partial \zeta_1} \\ \frac{\partial N_j}{\partial \zeta_2} \end{Bmatrix}$$

where $\mathbf{F}^{-T} = (\mathbf{F}^{-1})^T$.

The Jacobian matrix $\mathbf{F}$ is defined as $\mathbf{F} = \begin{bmatrix} \frac{\partial x}{\partial \zeta_1} & \frac{\partial x}{\partial \zeta_2} \\ \frac{\partial y}{\partial \zeta_1} & \frac{\partial y}{\partial \zeta_2} \end{bmatrix}$.

The derivatives $\frac{\partial N_j}{\partial \zeta_1}$ and $\frac{\partial N_j}{\partial \zeta_2}$ are easily computed from the master element shape function definitions. The components of $\mathbf{F}^{-T}$ depend on the specific element geometry.

8.7. For Vector Field Problems (Elasticity)

The concept of the B matrix is fundamental in structural mechanics (elasticity), where it relates nodal displacements to strains within an element.

The displacement field $\mathbf{u}$ is approximated as $\mathbf{u} = \mathbf{N}\mathbf{d}^e$, where $\mathbf{d}^e$ is the vector of all nodal displacements for the element, and $\mathbf{N}$ is the matrix of shape functions arranged appropriately for a vector field.

The strain $\boldsymbol{\varepsilon}$ is obtained by differentiating the displacement field: $\boldsymbol{\varepsilon} = \mathbf{L}\mathbf{u}$, where $\mathbf{L}$ is a differential operator matrix. Combining these gives:

$$\boldsymbol{\varepsilon} = \mathbf{L}(\mathbf{N}\mathbf{d}^e) = (\mathbf{L}\mathbf{N})\mathbf{d}^e = \mathbf{B}\mathbf{d}^e$$

The B matrix in elasticity thus contains derivatives of shape functions. The element stiffness matrix is then computed as:

$$[\mathbf{k}^e] = \int_{\Omega^e} \mathbf{B}^T \mathbf{D} \mathbf{B} \, d\Omega$$

where $\mathbf{D}$ is the material elasticity matrix (constitutive matrix relating stress to strain).

8.8. B Matrix for 2D Elasticity (Plane Stress / Plane Strain)

For a 2D elasticity problem, the displacement at any point within an element is $\mathbf{u}(x, y) = \begin{Bmatrix} u(x, y) \\ v(x, y) \end{Bmatrix}$.

The nodal displacement vector for a node i is $\mathbf{d}_i = \begin{Bmatrix} u_i \\ v_i \end{Bmatrix}$.

The displacement field is interpolated as:

$$\begin{Bmatrix} u \\ v \end{Bmatrix} = \sum_{i=1}^{n_{\text{nodes}}} \begin{bmatrix} N_i(x,y) & 0 \\ 0 & N_i(x,y) \end{bmatrix} \begin{Bmatrix} u_i \\ v_i \end{Bmatrix}$$

The strain vector in 2D is:

$$\boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} = \begin{Bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{Bmatrix}$$

The B matrix contribution for node i is:

$$\mathbf{B}_i = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \end{bmatrix}$$

$$\mathbf{B} = [\mathbf{B}_1 \quad \mathbf{B}_2 \quad \dots \quad \mathbf{B}_{n_{\text{nodes}}}]$$

So, if an element has n_{nodes} nodes, the B matrix will have 3 rows (for $\varepsilon_x, \varepsilon_y, \gamma_{xy}$) and $2 \times n_{\text{nodes}}$ columns.

8.9. B Matrix for 3D Elasticity

For a 3D elasticity problem, the displacement at any point is $\mathbf{u}(x,y,z) = \begin{Bmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{Bmatrix}$.

The nodal displacement vector for node i is $\mathbf{d}_i = \begin{Bmatrix} u_i \\ v_i \\ w_i \end{Bmatrix}$.

The displacement field is interpolated as:

$$\begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = \sum_{i=1}^{n_{\text{nodes}}} \begin{bmatrix} N_i(x,y,z) & 0 & 0 \\ 0 & N_i(x,y,z) & 0 \\ 0 & 0 & N_i(x,y,z) \end{bmatrix} \begin{Bmatrix} u_i \\ v_i \\ w_i \end{Bmatrix}$$

The strain vector (engineering strains) in 3D is:

$$\boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{Bmatrix} = \begin{Bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial w}{\partial z} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \\ \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \end{Bmatrix}$$

For each node, its contribution to the strain is related to the nodal displacements of that node:

$$\mathbf{B}_i = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_i}{\partial z} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial z} & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} & 0 & \frac{\partial N_i}{\partial x} \end{bmatrix}$$

The full element B matrix is assembled as $\mathbf{B} = [\mathbf{B}_1 \quad \mathbf{B}_2 \quad \dots \quad \mathbf{B}_{n_{\text{nodes}}}]$.

If an element has n_{nodes} nodes, the B matrix will have 6 rows and $3 \times n_{\text{nodes}}$ columns. The derivatives $\frac{\partial N_i}{\partial x}$, $\frac{\partial N_i}{\partial y}$, and $\frac{\partial N_i}{\partial z}$ are again found using the Jacobian transformation from the master element coordinates.

8.10. Key Takeaway (B Matrix)

The B matrix is a crucial component in finite element analysis.

It contains the spatial derivatives of the element shape functions, effectively translating the discrete nodal degrees of freedom into continuous gradient fields (like temperature gradients) or deformation measures (like mechanical strains) within an element.

This matrix is essential for forming the element stiffness matrix and, consequently, for solving the overall system of equations that describes the physical problem.

The components of B are generally functions of position within the element (unless using simple elements like constant strain triangles), and are evaluated numerically during the integration process to form the element stiffness matrix.

8.11. Shape Functions: Differential Properties

To evaluate terms like grad(V) and grad(T) in the weak form, we need the derivatives of shape functions (and consequently, field variables) with respect to global coordinates (x,y).

However, shape functions $N(\zeta_1, \zeta_2)$ are defined in the master element coordinates (ζ_1, ζ_2).

We use the chain rule and the Jacobian of the coordinate mapping to transform these derivatives.

8.11.1. Coordinate Transformation and the Jacobian Matrix

Recall the isoparametric mapping from master coordinates (ζ_1, ζ_2) to global coordinates (x,y):

$$x = \sum_{i=1}^{n_{\text{nodes}}} N_i(\zeta_1, \zeta_2) x_i$$

$$y = \sum_{i=1}^{n_{\text{nodes}}} N_i(\zeta_1, \zeta_2) y_i$$

The Jacobian matrix of this transformation is:

$$\mathbf{F} = \begin{bmatrix} \frac{\partial x}{\partial \zeta_1} & \frac{\partial x}{\partial \zeta_2} \\ \frac{\partial y}{\partial \zeta_1} & \frac{\partial y}{\partial \zeta_2} \end{bmatrix}$$

The determinant of this matrix, $J = \det(\mathbf{F})$, is used in changing variables for integration: $dx\,dy = J\,d\zeta_1\,d\zeta_2$.

Differentials are related by:

$$\begin{Bmatrix} dx \\ dy \end{Bmatrix} = \mathbf{F} \begin{Bmatrix} d\zeta_1 \\ d\zeta_2 \end{Bmatrix}$$

$$\begin{Bmatrix} d\zeta_1 \\ d\zeta_2 \end{Bmatrix} = \mathbf{F}^{-1} \begin{Bmatrix} dx \\ dy \end{Bmatrix}$$

$$\mathbf{F}^{-1} = \begin{bmatrix} \frac{\partial \zeta_1}{\partial x} & \frac{\partial \zeta_1}{\partial y} \\ \frac{\partial \zeta_2}{\partial x} & \frac{\partial \zeta_2}{\partial y} \end{bmatrix}$$

8.11.2. Transformation of Derivatives

We need to express derivatives with respect to global coordinates (x,y) in terms of derivatives with respect to master coordinates (ζ_1, ζ_2), since our shape functions N_i are given as $N_i(\zeta_1, \zeta_2)$.

Consider a function $V(\zeta_1, \zeta_2)$. Its derivatives with respect to ζ_1 and ζ_2 can be related to its derivatives with respect to x and y using the chain rule:

$$\frac{\partial V}{\partial \zeta_1} = \frac{\partial V}{\partial x}\frac{\partial x}{\partial \zeta_1} + \frac{\partial V}{\partial y}\frac{\partial y}{\partial \zeta_1}$$

$$\frac{\partial V}{\partial \zeta_2} = \frac{\partial V}{\partial x}\frac{\partial x}{\partial \zeta_2} + \frac{\partial V}{\partial y}\frac{\partial y}{\partial \zeta_2}$$

$$\begin{Bmatrix} \frac{\partial V}{\partial \zeta_1} \\ \frac{\partial V}{\partial \zeta_2} \end{Bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \zeta_1} & \frac{\partial y}{\partial \zeta_1} \\ \frac{\partial x}{\partial \zeta_2} & \frac{\partial y}{\partial \zeta_2} \end{bmatrix} \begin{Bmatrix} \frac{\partial V}{\partial x} \\ \frac{\partial V}{\partial y} \end{Bmatrix} = \mathbf{F}^T \begin{Bmatrix} \frac{\partial V}{\partial x} \\ \frac{\partial V}{\partial y} \end{Bmatrix}$$

$$\begin{Bmatrix} \frac{\partial V}{\partial x} \\ \frac{\partial V}{\partial y} \end{Bmatrix} = (\mathbf{F}^T)^{-1} \begin{Bmatrix} \frac{\partial V}{\partial \zeta_1} \\ \frac{\partial V}{\partial \zeta_2} \end{Bmatrix} = \mathbf{F}^{-T} \begin{Bmatrix} \frac{\partial V}{\partial \zeta_1} \\ \frac{\partial V}{\partial \zeta_2} \end{Bmatrix}$$

$$\mathbf{F}^{-T} = \left(\begin{bmatrix} \frac{\partial x}{\partial \zeta_1} & \frac{\partial x}{\partial \zeta_2} \\ \frac{\partial y}{\partial \zeta_1} & \frac{\partial y}{\partial \zeta_2} \end{bmatrix} \right)^T = \begin{bmatrix} \frac{\partial x}{\partial \zeta_1} & \frac{\partial y}{\partial \zeta_1} \\ \frac{\partial x}{\partial \zeta_2} & \frac{\partial y}{\partial \zeta_2} \end{bmatrix}$$

8.11.3. Gradient Calculation in Master Element Coordinates

Using the relationship derived above, the gradient of a test function V (or temperature field T) in global coordinates is:

$$\text{grad}(V) = \begin{Bmatrix} \frac{\partial V}{\partial x} \\ \frac{\partial V}{\partial y} \end{Bmatrix} = \mathbf{F}^{-T} \begin{Bmatrix} \frac{\partial V}{\partial \zeta_1} \\ \frac{\partial V}{\partial \zeta_2} \end{Bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \zeta_1} & \frac{\partial y}{\partial \zeta_1} \\ \frac{\partial x}{\partial \zeta_2} & \frac{\partial y}{\partial \zeta_2} \end{bmatrix} \begin{Bmatrix} \frac{\partial V}{\partial \zeta_1} \\ \frac{\partial V}{\partial \zeta_2} \end{Bmatrix}$$

$$\frac{\partial V}{\partial x} = \frac{\partial x}{\partial \zeta_1}\frac{\partial V}{\partial \zeta_1} + \frac{\partial x}{\partial \zeta_2}\frac{\partial V}{\partial \zeta_2}$$

$$\frac{\partial V}{\partial y} = \frac{\partial y}{\partial \zeta_1}\frac{\partial V}{\partial \zeta_1} + \frac{\partial y}{\partial \zeta_2}\frac{\partial V}{\partial \zeta_2}$$

8.11.4. Implementation Steps

To implement this:

1. For each shape function $N_i(\zeta_1, \zeta_2)$, calculate its derivatives $\frac{\partial N_i}{\partial \zeta_1}$ and $\frac{\partial N_i}{\partial \zeta_2}$ analytically in the master element.

2. At each integration point (ζ_1, ζ_2) within the master element:

   - Calculate the Jacobian matrix $\mathbf{F}$ using the derivatives of the mapping functions $x(\zeta_1, \zeta_2)$ and $y(\zeta_1, \zeta_2)$.
   - Compute $\mathbf{F}^{-1}$ and then $\mathbf{F}^{-T}$.
   - Use $\mathbf{F}^{-T}$ to transform the local derivatives $\begin{Bmatrix} \frac{\partial N_i}{\partial \zeta_1} \\ \frac{\partial N_i}{\partial \zeta_2} \end{Bmatrix}$ into global derivatives $\begin{Bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{Bmatrix}$.
   - These global derivatives form the columns of the B matrix for scalar problems, or parts of the columns for vector problems.

8.12. Summary: Key Concepts

8.12.1. Mathematical Foundation

- Product rule for divergence enables weak form derivation
- Divergence theorem converts domain integrals to boundary integrals
- Integration by parts transfers derivatives to test functions

8.12.2. Geometric Considerations

- Master element concept simplifies integration
- Jacobian must be positive for valid elements
- Element quality depends on convexity and node ordering

8.12.3. Implementation Details

- Shape functions provide local approximation
- Connectivity tables manage global assembly
- Coordinate transformations enable numerical integration

8.12.4. Extension to Higher Dimensions

- Same mathematical principles apply
- Increased complexity in bookkeeping and assembly
- Vector problems require multiple DOFs per node

9. 2D Heat Transfer

9.1. The Strong Form

The governing equation for 2D heat transfer is:

$$\text{div}(\mathbf{K} \cdot \text{grad}(T)) + Q = 0$$

with boundary conditions:

$$T = T_A \text{ on } \partial\Omega_A, \quad q = q^* \text{ on } \partial\Omega_B$$

where: - $\mathbf{K}$ is the thermal conductivity tensor - Q is the heat source term - q^* is a heat flux boundary condition

9.2. The Weak Form

To develop the weak form, we multiply the strong form by a test function V and integrate over the domain:

$$\int_\Omega \text{grad}(V) \cdot \mathbf{K} \cdot \text{grad}(T)\, d\Omega = \int_\Omega QV\, d\Omega - \int_{\partial\Omega_B} Vq^*\, da$$

9.2.1. Test and Trial Functions

We choose to take both test and trial functions from the same function space, which is spanned by the shape functions ϕ_i defined via the master element coordinates (ζ_1, ζ_2) and master element shape functions $\hat{\phi}$.

$$T = \sum_{j=1}^{N} a_j \phi_j$$
$$V = \sum_{i=1}^{N} b_i \phi_i$$

Replacing the integral over sums with sum over integrals:

$$\sum_{i=1}^{N} b_i \left[\sum_{j=1}^{N} \left(\int_\Omega \text{grad}\phi_i \cdot \mathbf{K} \cdot \text{grad}\phi_j\, d\Omega \right) a_j \right] - \sum_{i=1}^{N} b_i \left(\int_\Omega \phi_i Q d\Omega - \int_{\partial\Omega_B} \phi_i q^* da \right) = 0$$

9.3. Element Stiffness Matrix

The above equation represents a summation of **element** integrals over **all nodes**.

For element e whose nodes are I, J:

$$K_{IJ}^e = \int_{\Omega^e} \text{grad}\phi_I \cdot \mathbf{K} \cdot \text{grad}\phi_J \, d\Omega$$

Assuming isotropic material with constant thermal conductivity $\mathbf{K} = k\mathbf{I}$, we can simplify:

$$K_{IJ}^e = k \int_{\Omega^e} \text{grad}\phi_I \cdot \text{grad}\phi_J \, d\Omega$$

This leads to the element stiffness matrix:

$$[\mathbf{k}^e] = k \int_{\Omega^e} \mathbf{B}^T \mathbf{B} \, d\Omega$$

where $\mathbf{B}$ is the B matrix containing the derivatives of the shape functions ϕ_i with respect to global coordinates (x,y):

$$[\mathbf{B}\phi^e] = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \frac{\partial \phi_1}{\partial x_2} \\ \frac{\partial \phi_2}{\partial x_1} & \frac{\partial \phi_2}{\partial x_2} \\ \frac{\partial \phi_3}{\partial x_1} & \frac{\partial \phi_3}{\partial x_2} \\ \frac{\partial \phi_4}{\partial x_1} & \frac{\partial \phi_4}{\partial x_2} \end{bmatrix}$$

Such that:

$$K^e = \int_{\Omega^e} k[B\phi^e][B\phi^e]^T \, d\Omega$$

9.4. Isoparametric Mapping and Jacobian

Using isoparametric mapping, we can express the shape functions in terms of master element coordinates (ζ_1, ζ_2) which are used for interpolating the space as well:

$$\mathbf{x}(\zeta_1, \zeta_2) = \sum_{I=1}^{n_{\text{nodes}}} x_I \hat{\phi}_I(\zeta_1, \zeta_2)$$

The derivatives transform according to:

$$\frac{\partial \phi_I}{\partial x_j} = \sum_{k=1}^{dim} \frac{\partial \hat{\phi}_I}{\partial \zeta_k} \frac{\partial \zeta_k}{\partial x_j}$$

The term $[\mathbf{B}\hat{\phi}^e]_{Ij}$ can be expressed as:

$$\frac{\partial \hat{\phi}_I}{\partial x_j} = \sum_{k=1}^{dim} \frac{\partial \phi_I}{\partial x_j} \frac{\partial x_j}{\partial \zeta_k}$$

Or equivalently:

$$\mathbf{B}\hat{\phi}^e = \mathbf{B}\phi^e\mathbf{F}\mathbf{B}\phi^e = \mathbf{B}\hat{\phi}^e\mathbf{F}^{-1}$$

With the Jacobian matrix defined as:

$$F_{ij} = \frac{\partial x_i}{\partial \zeta_j} = \sum_{I=1}^{nodes} x_I^j \frac{\partial \hat{\phi}_I(\zeta)}{\partial \zeta_j}$$

The determinant of $\mathbf{F}$ (denoted J) is used to transform between the area in the master element and the area in the global coordinates:

$$d\Omega = Jd\hat{\Omega}$$

where $J = \det(\mathbf{F})$ and $d\hat{\Omega}$ is the area in the master element coordinates.

9.5. Weak Form Terms in Master Element Coordinates

We can now write all the terms in the weak form at the element level:

$$\mathbf{K}^e = \int_{\Omega^e} \mathbf{K}(\mathbf{x}(\zeta))(\mathbf{B}\hat{\phi}^e)(\mathbf{F}^T\mathbf{F})^{-1}(\mathbf{B}\hat{\phi}^e)^T \, Jd\hat{\Omega}$$

$$\mathbf{f}^e = \int_{\Omega^e} \hat{\phi}^e(\zeta) f(\mathbf{x}(\zeta)) \, Jd\hat{\Omega}$$

9.6. Flux Boundary Condition in Master Element Coordinates

The only remaining task is to define the flux boundary condition $\int_{\partial\Omega_B} \phi_i q^* da$ in terms of the master element coordinates.

We start with relating da (global) and dA (local):

$$d\zeta = \frac{d\zeta}{dA}\, dA = \mathbf{M}\, dA d\mathbf{x} = \frac{d\mathbf{x}}{da}\, da = \mathbf{m}\, da$$

The differential area relationship is:

$$dx_i = \sum_{j=1}^{dim} \frac{\partial x_i}{\partial \zeta_j} d\zeta_j = \sum_{j=1}^{dim} F_{ij} d\zeta_j \Rightarrow d\mathbf{x} = \mathbf{F}d\zeta$$

Using the norm properties:

$$d\zeta \cdot d\zeta = dA^2 \quad ; \quad d\mathbf{x} \cdot d\mathbf{x} = da^2$$

$$da^2 = \mathbf{F}d\zeta \cdot \mathbf{F}d\zeta = dA^2(\mathbf{FM}) \cdot (\mathbf{FM})$$

Therefore:

$$\frac{da}{dA} = \sqrt{(\mathbf{FM})^T(\mathbf{FM})}$$

The boundary integral in terms of master element coordinates becomes:

$$Q_i^s = -\int_{\partial\Omega_B} \phi_i q^* da = -\int_{\partial\hat{\Omega}_B} \hat{\phi}_i(\zeta) q^*(\zeta(\mathbf{x}))\sqrt{(\mathbf{FM})^T(\mathbf{FM})} dA$$

The flux boundary condition is applied on some line of the 2D element, which means either ζ_1 or ζ_2 will be constant and equal to ± 1.

9.7. 2D Finite Element Assembly Algorithm

Algorithm: 2D Finite Element Assembly

Input: Number of elements N_e, Gauss points n_g
Output: Global stiffness matrix $\mathbf{K}$ and force vector $\mathbf{F}$
Steps:
for elem $= 1$ to N_e do
  Set $\mathbf{K}^e = \mathbf{0}$, $\mathbf{f}^e = \mathbf{0}$; Form $\mathbf{X}^e$
  for $m = 1$ to n_g do (Gauss points)
    for $n = 1$ to n_g do
      $\boldsymbol{\zeta} = [\zeta_1, \zeta_2] = [\text{Points}[m], \text{Points}[n]]$
      Evaluate $\mathbf{x}(\boldsymbol{\zeta}) = (\mathbf{X}^e)^T\hat{\boldsymbol{\phi}}(\boldsymbol{\zeta})$, $\mathbf{F}$, $\det(\mathbf{F})$
      $\mathbf{K}^e \leftarrow \mathbf{K}^e + w_m w_n K(\mathbf{x}(\boldsymbol{\zeta}))\mathbf{D}\hat{\boldsymbol{\phi}}^e(\mathbf{F}^T\mathbf{F})^{-1}(\mathbf{D}\hat{\boldsymbol{\phi}}^e)^T \det(\mathbf{F})$
      $\mathbf{f}^e \leftarrow \mathbf{f}^e + w_m w_n f(\mathbf{x}(\boldsymbol{\zeta}))\hat{\boldsymbol{\phi}}(\boldsymbol{\zeta}) \det(\mathbf{F})$
    end for
  end for
  Assemble into global matrix $[\mathbf{K}]N \times N$ and vector $[\mathbf{F}]N \times 1$
end for

Part IV.

Part IV: Elasticity

10. Continuum Mechanics Fundamentals

10.1. Displacement and Configuration

In continuum mechanics, we deal with continuous media and their deformation under external forces.

Displacement u: The change in position of a point in the body, defined as $\mathbf{u} = \mathbf{x} - \mathbf{x}_0$, where $\mathbf{x}_0$ is the original position, often written using capital $\mathbf{X}$.

Configuration:

- $\mathbf{X}$: The original position of a point in the body before deformation, with E_i being the coordinate system in the reference configuration: $\mathbf{X} = X_i E_i$
- $\mathbf{x}$: The current position of the same point after deformation, with e_i being the coordinate system in the current configuration: $\mathbf{x} = x_i e_i$

10.2. Deformation Gradient

The **Deformation Gradient F** is a tensor that describes the local deformation of the material. It relates the current configuration to the reference configuration:

$$\mathbf{F} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial x_1}{\partial X_1} & \frac{\partial x_1}{\partial X_2} & \cdots \\ \frac{\partial x_2}{\partial X_1} & \frac{\partial x_2}{\partial X_2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

10.3. Strain

Strain $\boldsymbol{\varepsilon}$ is a measure of deformation defined as the symmetric part of the deformation gradient:

$$\boldsymbol{\varepsilon} = \frac{1}{2}(\mathbf{F}^T\mathbf{F} - \mathbf{I})$$

where $\mathbf{I}$ is the identity tensor.

Under the assumption of small deformations, the strain can be approximated as:

$$\boldsymbol{\varepsilon} \approx \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T) = \nabla^s\mathbf{u}$$

where $\nabla\mathbf{u}$ is the gradient of the displacement field and $\nabla^s\mathbf{u}$ is the symmetric gradient.

10.4. Stress and Constitutive Relation

Stress $\boldsymbol{\sigma}$ is a measure of internal forces within the material, defined using a constitutive relation that relates stress to strain. For linear elasticity:

$$\boldsymbol{\sigma} = \mathbf{C} : \boldsymbol{\varepsilon}$$

where $\mathbf{C}$ is the elasticity tensor (material property matrix) and $\boldsymbol{\sigma}$ is the Cauchy stress tensor.

10.5. Tractions

Tractions $\mathbf{t}$ represent the force per unit area acting on a surface, related to stress by:

$$\mathbf{t} = \boldsymbol{\sigma} \cdot \mathbf{n}$$

where $\mathbf{n}$ is the outward normal to the surface.

10.6. Mass Conservation

The principle that mass is conserved in a closed system. In continuum mechanics, this is expressed as:

$$\int_\Omega \rho \, dv = \text{constant} = \int_\Omega \rho_0 \, dV \Rightarrow \rho J = \rho_0$$

where ρ is the current density, ρ_0 is the reference density, and $J = \det(\mathbf{F})$ is the Jacobian determinant of the deformation gradient.

10.7. Balance of Linear Momentum

The balance of linear momentum states:

$$\int_\Omega \rho \frac{\partial^2 \mathbf{x}}{\partial t^2} \, dv = \int_\Omega \mathbf{f} \, dv + \int_{\partial\Omega} \mathbf{t} \, da$$

This leads to the equation of motion:

$$\operatorname{div}(\boldsymbol{\sigma}) + \mathbf{f} = \rho \ddot{\mathbf{u}}$$

10.8. Balance of Angular Momentum

The balance of angular momentum is expressed as:

$$\frac{d}{dt} \int_\Omega \mathbf{x} \times (\rho v) \, dv = \int_\Omega \mathbf{x} \times \mathbf{f} \, dv + \int_{\partial\Omega} \mathbf{x} \times \mathbf{t} \, da$$

10.9. The Weak Form Derivation

For quasi-static problems, we assume the acceleration terms are zero, so the balance of linear momentum becomes:

$$\nabla \cdot \boldsymbol{\sigma} + \mathbf{f} = 0$$

with

$$\nabla \cdot \boldsymbol{\sigma} = \left(\frac{\partial \sigma_{ij}}{\partial x_j} \right) \mathbf{e}_i$$

where σ_{ij} are the components of the stress tensor.

10.9.1. Multiplication by Test Function and Integration

We multiply by a test function $\mathbf{v}$ and integrate over the domain:

$$\int_\Omega (\nabla \cdot \boldsymbol{\sigma} + \mathbf{f}) \cdot \mathbf{v} \, d\Omega = 0$$

10.9.2. Product Rule Application

We use the product rule for the divergence of a tensor contracted with a vector:

$$\nabla \cdot (\boldsymbol{\sigma} \cdot \mathbf{v}) = (\nabla \cdot \boldsymbol{\sigma}) \cdot \mathbf{v} + \boldsymbol{\sigma} : \nabla(\mathbf{v})$$

where:

$$\boldsymbol{\sigma} : \nabla(\mathbf{v}) = \sigma_{ij} \frac{\partial v_j}{\partial x_i}$$

Rearranging:

$$(\nabla \cdot \boldsymbol{\sigma}) \cdot \mathbf{v} = \nabla \cdot (\boldsymbol{\sigma} \mathbf{v}) - \boldsymbol{\sigma} : \nabla \mathbf{v}$$

10.9.3. Divergence Theorem Application

Applying the divergence theorem to the integrated equation:

$$\int_\Omega \nabla \cdot (\boldsymbol{\sigma} \mathbf{v}) d\Omega - \int_\Omega \boldsymbol{\sigma} : \nabla \mathbf{v} d\Omega + \int_\Omega \mathbf{f} \cdot \mathbf{v} d\Omega = 0$$

$$\implies \int_{\partial\Omega} (\boldsymbol{\sigma} \mathbf{v}) \cdot \mathbf{n} d\Gamma - \int_\Omega \boldsymbol{\sigma} : \nabla \mathbf{v} d\Omega + \int_\Omega \mathbf{f} \cdot \mathbf{v} d\Omega = 0$$

The term $(\boldsymbol{\sigma n})$ is the traction vector $\boldsymbol{t}$. The term $\nabla \boldsymbol{v}$ is the gradient of the test function, and its symmetric part is the strain tensor for the test function, $\boldsymbol{\varepsilon}(\boldsymbol{v})$.

10.9.4. Final Weak Form

This derivation yields the weak form of the momentum equation, which forms the foundation for finite element elasticity analysis.

11. Quasi-Static Elasticity in 3D

11.1. Strong Form and Method of Weighted Residuals

Starting from the equation of motion in the quasi-static case:

$$\nabla \cdot \boldsymbol{\sigma} + \mathbf{f} = 0$$

We multiply by a test function $\mathbf{v}$ and integrate over the domain Ω. This is the method of weighted residuals, where $\boldsymbol{r}$ is the residual:

$$\int_\Omega (\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{f}) \cdot \mathbf{v}\, d\Omega = 0 = \int_\Omega \boldsymbol{r} \cdot \boldsymbol{v} d\Omega$$

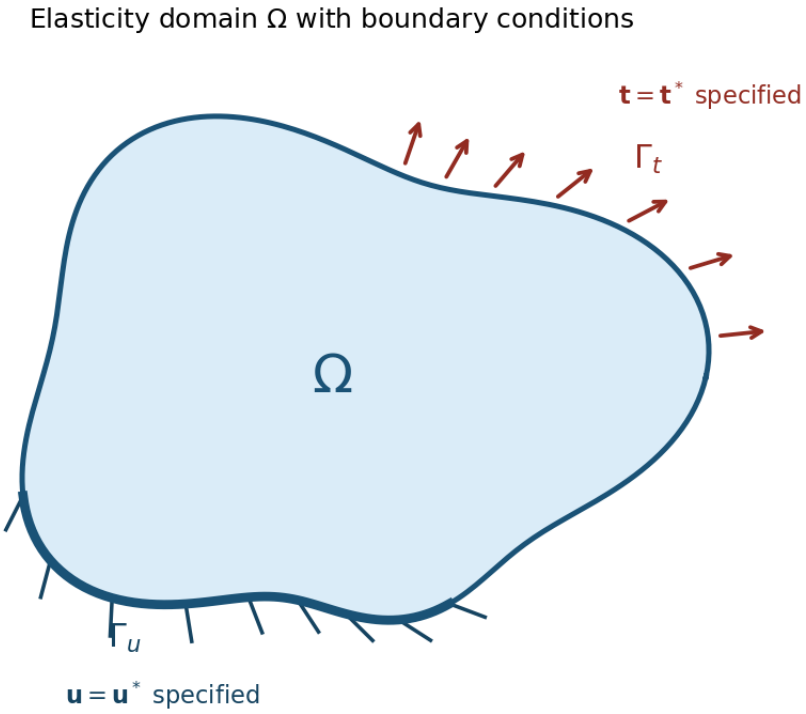


Figure 11.1.: Elasticity domain Ω with Neumann boundary Γ_t (prescribed traction $\mathbf{t} = \mathbf{t}^*$, outward arrows) and Dirichlet boundary Γ_u (prescribed displacement $\mathbf{u} = \mathbf{u}^*$, hatched).

11.2. Product Rule and Integration by Parts

Recall the product rule for the divergence of a tensor contracted with a vector:

$$\nabla \cdot (\boldsymbol{\sigma}\mathbf{v}) = (\nabla \cdot \boldsymbol{\sigma}) \cdot \mathbf{v} + \boldsymbol{\sigma} : \nabla\mathbf{v}$$

We can write:

$$(\nabla \cdot \boldsymbol{\sigma}) \cdot \mathbf{v} = \nabla \cdot (\boldsymbol{\sigma}\mathbf{v}) - \boldsymbol{\sigma} : \nabla\mathbf{v}$$

Substituting this into our integrated equation and applying the divergence theorem:

$$\int_\Omega \nabla \cdot (\boldsymbol{\sigma}\mathbf{v}) d\Omega - \int_\Omega \boldsymbol{\sigma} : \nabla\mathbf{v} d\Omega + \int_\Omega \mathbf{f} \cdot \mathbf{v} d\Omega = 0$$

$$\implies \int_{\partial\Omega} (\boldsymbol{\sigma}\mathbf{v}) \cdot \mathbf{n} d\Gamma - \int_\Omega \boldsymbol{\sigma} : \nabla\mathbf{v} d\Omega + \int_\Omega \mathbf{f} \cdot \mathbf{v} d\Omega = 0$$

The term $(\boldsymbol{\sigma}\boldsymbol{n})$ is the traction vector $\boldsymbol{t}$. The term $\nabla\boldsymbol{v}$ is the gradient of the test function, and its symmetric part is the strain tensor for the test function, $\boldsymbol{\varepsilon}(\boldsymbol{v})$.

11.3. Boundary Conditions and Natural Boundary Terms

The boundary $\partial\Omega$ is composed of two parts:

- Γ_u (where displacements are prescribed)
- Γ_t (where tractions are prescribed)

On Γ_u we enforce Dirichlet boundary conditions, meaning the displacement field $\mathbf{u}$ is prescribed as $\mathbf{u}^*$.

On Γ_t we enforce Neumann boundary conditions, meaning the traction vector $\boldsymbol{\sigma}\boldsymbol{n} = \boldsymbol{t}$ is prescribed as $\boldsymbol{t}^*$.

This is the natural boundary condition that will be incorporated into the weak form. The boundary integral becomes:

$$\int_{\partial\Omega} \mathbf{t} \cdot \mathbf{v} d\Gamma = \int_{\Gamma_t} \mathbf{t}^* \cdot \mathbf{v} d\Gamma + \int_{\Gamma_u} \mathbf{t} \cdot \underbrace{\mathbf{v}}_{=0} d\Gamma = \int_{\Gamma_t} \mathbf{t}^* \cdot \mathbf{v} d\Gamma$$

11.4. Final Weak Form

We can now state our problem in its final weak form:

Find the displacement field $\boldsymbol{u} \in H^1(\Omega)$ such that $\boldsymbol{u} = \boldsymbol{u}^*$ on the Dirichlet boundary Γ_u for all test functions $\boldsymbol{v} \in H^1$ for which $\boldsymbol{v}|_{\Gamma_u} = 0$ satisfying the weak form:

$$\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) d\Omega = \int_\Omega \mathbf{f} \cdot \mathbf{v} d\Omega + \int_{\Gamma_t} \mathbf{t}^* \cdot \mathbf{v} d\Gamma$$

11.5. Constitutive Law and Elasticity Tensor

In the weak form, we have the stress tensor $\boldsymbol{\sigma}$ expressed in terms of the strain tensor $\boldsymbol{\varepsilon}$ and the displacement field $\mathbf{u}$.

We use the constitutive law for linear elasticity $\boldsymbol{\sigma} = \boldsymbol{L} : \boldsymbol{\varepsilon}$, where $\boldsymbol{L}$ is the fourth-order elasticity tensor. For an isotropic material, the matrix form of $\boldsymbol{L}$ (in Voigt notation) is given by:

$$\mathbf{L} = \frac{E}{(1+\nu)(1-2\nu)} \begin{pmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{pmatrix}$$

where E is Young's modulus and ν is Poisson's ratio.

The term $\boldsymbol{\sigma} = \boldsymbol{L} : \boldsymbol{\varepsilon}$ refers to the double contraction of the elasticity tensor with the strain tensor, which can be expressed in index notation as:

$$\sigma_{ij} = L_{ijkl}\varepsilon_{kl}$$

where σ_{ij} are the components of the stress tensor, L_{ijkl} are the components of the elasticity tensor, and ε_{kl} are the components of the strain tensor.

11.6. Voigt Notation for Strain and Stress

In Voigt notation, the strain tensor is represented as a vector:

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{31} \\ 2\varepsilon_{12} \end{bmatrix}$$

and the stress tensor is represented as:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{31} \\ \sigma_{12} \end{bmatrix}$$

11.7. Bilinear and Linear Forms

We continue from our weak form:

$$\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v})\, d\Omega = \int_\Omega \mathbf{f} \cdot \mathbf{v}\, d\Omega + \int_{\Gamma_t} \mathbf{t}^* \cdot \mathbf{v}\, d\Gamma$$

Now, we substitute the constitutive law into the left-hand side:

$$\int_\Omega (\boldsymbol{L} \cdot \boldsymbol{\varepsilon}(\mathbf{u})) \cdot \boldsymbol{\varepsilon}(\mathbf{v})\, d\Omega = \int_\Omega \mathbf{f} \cdot \mathbf{v}\, d\Omega + \int_{\Gamma_t} \mathbf{t}^* \cdot \mathbf{v}\, d\Gamma$$

Where we dropped the double contraction (:) and replaced it with $\cdot$ since we use Voigt notation for the strain and stress tensors.

This equation has a standard structure. The left side is linear with respect to both the solution u and the test function v. The right side is linear only with respect to the test function v.

We can define a **bilinear form** $B(u, v)$ from the left-hand side, which represents the internal virtual work:

$$B(\mathbf{u}, \mathbf{v}) := \int_\Omega \boldsymbol{\varepsilon}(\mathbf{v}) \cdot \boldsymbol{L} \cdot \boldsymbol{\varepsilon}(\mathbf{u})\, d\Omega$$

And we define a **linear form** $F(v)$ from the right-hand side, which represents the external virtual work done by body forces and applied tractions:

$$F(\mathbf{v}) := \int_\Omega \mathbf{f} \cdot \mathbf{v}\, d\Omega + \int_{\Gamma_t} \mathbf{t}^* \cdot \mathbf{v}\, d\Gamma$$

The problem is now stated in its abstract form: Find the displacement u (satisfying Dirichlet BCs) such that for all valid test functions v:

$$B(\mathbf{u}, \mathbf{v}) = F(\mathbf{v})$$

This abstract form is the direct starting point for the Finite Element discretization, where we will replace the infinite-dimensional function spaces with finite-dimensional approximations.

Recalling the definition of the strain tensor:

$$\boldsymbol{\varepsilon}(\mathbf{v}) = \frac{1}{2}(\nabla \mathbf{v} + \nabla \mathbf{v}^T)$$

We can express the bilinear form in terms of the gradient of the test function:

$$B(\mathbf{u}, \mathbf{v}) = \int_\Omega \boldsymbol{\varepsilon}(\mathbf{v}) \cdot \boldsymbol{L} \cdot \boldsymbol{\varepsilon}(\mathbf{u})\, d\Omega = \int_\Omega \left(\frac{1}{2}(\nabla \mathbf{v} + \nabla \mathbf{v}^T)\right) : \boldsymbol{L} : \left(\frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T)\right) d\Omega$$

11.8. Function Spaces for Test and Trial Functions

To ensure our weak form is mathematically well-posed, the solution (trial) function $\boldsymbol{u}$ and the test function $\boldsymbol{v}$ can't be just any function. They must belong to specific function spaces.

Why do we need restrictions? The bilinear form $\mathbf{B(u,v)}$ involves integrals of derivatives of the functions. For these integrals to be well-defined and finite, the functions must have a certain degree of smoothness.

11.8.1. Hilbert-Sobolev Spaces

A function u belongs to H^1 if both the function and its first derivatives are square-integrable over the domain Ω:

$$H^1(\Omega) = \left(\mathbf{u} \in L^2(\Omega) \;\middle|\; \nabla\mathbf{u} \in L^2(\Omega)\right)$$

In other words:

$$\int_\Omega \nabla\boldsymbol{u} \cdot \nabla\boldsymbol{u}\, d\Omega + \int_\Omega \boldsymbol{u} \cdot \boldsymbol{u}\, d\Omega < \infty$$

This ensures that the energy norm is finite.

11.8.2. Handling Dirichlet Conditions

The test functions $\boldsymbol{v}$ must also respect the homogeneous Dirichlet boundary conditions. This leads us to the space $H^1_0(\Omega)$, which is a subspace of $H^1(\Omega)$.

It contains all functions that are zero on the Dirichlet boundary Γ_u:

$$H^1_0(\Omega) = \left[\mathbf{v} \in H^1(\Omega) \;\middle|\; \mathbf{v} = \mathbf{0} \text{ on } \Gamma_u\right]$$

11.9. The Lifting Technique for Non-Zero Displacements

If our prescribed displacement $\boldsymbol{u}^*$ on the boundary Γ_u is not zero, we must handle this carefully using the **lifting** technique.

Mathematically, we handle this by splitting the solution u into two parts:

$$\mathbf{u} = \mathbf{u}_0 + \mathbf{u}_D$$

where $\boldsymbol{u}_D$ is a known function (aka "lifting function") that satisfies the Dirichlet boundary condition on Γ_u, and $\boldsymbol{u}_0$ is the unknown part of the solution that we seek.

11.9.1. Modified Weak Form

We substitute this decomposition into our weak form:

$$B(\mathbf{u}_0 + \mathbf{u}_D, \mathbf{v}) = F(\mathbf{v})$$

Using the linearity of B(·,v):

$$B(\mathbf{u}_0, \mathbf{v}) + B(\mathbf{u}_D, \mathbf{v}) = F(\mathbf{v})$$

We then move all the known quantities to the right-hand side:

$$B(\mathbf{u}_0, \mathbf{v}) = F(\mathbf{v}) - B(\mathbf{u}_D, \mathbf{v})$$

Our new problem is to find the homogeneous component $\boldsymbol{u}_0 \in H_0^1(\Omega)$ that satisfies this modified equation for all test functions $\boldsymbol{v} \in H_0^1(\Omega)$:

$$B(\mathbf{u}_0, \mathbf{v}) = F(\mathbf{v}) - B(\mathbf{u}_D, \mathbf{v})$$

This means we are looking for a solution $\boldsymbol{u}_0$ that satisfies the weak form with respect to the homogeneous space $H_0^1(\Omega)$, while $\boldsymbol{u}_D$ is already known and satisfies the Dirichlet boundary condition.

Once $\boldsymbol{u}_0$ is found, the final solution is simply $\boldsymbol{u} = \boldsymbol{u}_0 + \boldsymbol{u}_D$. In practice, this is handled during the assembly of the finite element system.

11.10. Principle of Minimum Potential Energy

An alternative but equivalent way to derive the weak form for elasticity problems is by minimizing a potential energy functional. This principle states that the exact solution to an elastic problem is the one that minimizes the total potential energy.

11.10.1. Total Potential Energy

The total potential energy of the system is the sum of the internal strain energy and the potential energy of the external loads:

$$\Pi(\mathbf{u}) = \underbrace{\frac{1}{2}\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{u})\, d\Omega}_{\text{Strain Energy}} - \underbrace{\left(\int \Omega \mathbf{f} \cdot \mathbf{u}\, d\Omega + \int_{\Gamma_t} \bar{\mathbf{t}} \cdot \mathbf{u}\, d\Gamma\right)}_{\text{Potential of External Loads}}$$

11.10.2. Finding the Minimum

The solution u that minimizes the functional Π is found by setting its directional derivative to zero for any arbitrary perturbation (test function) v. This is analogous to setting the first derivative to zero in standard calculus:

$$\lim_{\epsilon \to 0} \frac{\Pi(\mathbf{u} + \epsilon \mathbf{v}) - \Pi(\mathbf{u})}{\epsilon} = 0$$

Calculating this derivative (which is the first variation of the functional) yields:

$$\delta\Pi = \int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v})\, d\Omega - \int_\Omega \mathbf{f} \cdot \mathbf{v}\, d\Omega - \int_{\Gamma_t} \bar{\mathbf{t}} \cdot \mathbf{v}\, d\Gamma = 0$$

Rearranging this gives us our familiar weak form:

$$\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v})\, d\Omega = \int_\Omega \mathbf{f} \cdot \mathbf{v}\, d\Omega + \int_{\Gamma_t} \bar{\mathbf{t}} \cdot \mathbf{v}\, d\Gamma$$

This shows that the solution satisfying the weak form is also the one that minimizes the total potential energy of the system.

11.11. Proof that Weak Form Solution Minimizes Energy

We previously showed that minimizing the potential energy functional leads to the weak form. Now we will prove the converse: the unique solution u to the weak form is the unique minimizer of the potential energy functional $L(\mathbf{w})$.

Define the Energy Functional:

$$L(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{2} B(\mathbf{w}, \mathbf{w}) - F(\mathbf{w})$$

The Goal: We want to show that $J(u) \leq J(w)$ for any other admissible function $\boldsymbol{w}$ (a function that satisfies the same Dirichlet boundary conditions as $\boldsymbol{u}$).

Let's evaluate the energy of any admissible function w: We can write any such w as the sum of the true solution u and an error function e, so w=u+e.

Since u and w both satisfy the same Dirichlet boundary conditions, the error function e must be zero on the Dirichlet boundary Γ_u and thus belongs to the space $H_0^1(\Omega)$, which is the space of functions that are square-integrable and have square-integrable first derivatives, and are zero on the Dirichlet boundary.

Let's substitute w=u+e into the energy functional:

$$L(\mathbf{w}) = L(\mathbf{u} + \mathbf{e}) = \frac{1}{2} B(\mathbf{u} + \mathbf{e}, \mathbf{u} + \mathbf{e}) - F(\mathbf{u} + \mathbf{e})$$

Using the linearity of the forms, this expands to:

$$L(\mathbf{w}) = \frac{1}{2} \left(B(\mathbf{u}, \mathbf{u}) + 2B(\mathbf{u}, \mathbf{e}) + B(\mathbf{e}, \mathbf{e}) \right) - \left(F(\mathbf{u}) + F(\mathbf{e}) \right)$$

Now, we rearrange the expression:

$$L(\mathbf{w}) = \underbrace{\left(\frac{1}{2}B(\mathbf{u},\mathbf{u}) - F(\mathbf{u})\right)}_{L(\mathbf{u})} + \underbrace{(B(\mathbf{u},\mathbf{e}) - F(\mathbf{e}))}_{=0} + \underbrace{\frac{1}{2}B(\mathbf{e},\mathbf{e})}_{\frac{1}{2}|\mathbf{e}|_E^2}$$

The middle term is zero because u is the true solution, so it satisfies the weak form $B(u,v) = F(v)$ for any test function v, including our error function e.

The Final Result: This simplifies to a fundamental relationship:

$$L(\mathbf{w}) = L(\mathbf{u}) + \frac{1}{2}|\mathbf{w} - \mathbf{u}|_E^2$$

Since the energy norm $||w-u||_E^2$ is always greater than or equal to zero, this proves that $L(w) \geq L(u)$.

The energy of any admissible function is greater than or equal to the energy of the true solution. The minimum is achieved only when $||w-u||_E^2 = 0$, which means $w = u$.

11.12. FEM Approximation and Error Estimates

The Finite Element Method does not find the exact solution, but rather an approximate solution u_h. **The accuracy of this approximation is a central concern.**

Discretization: We divide our domain Ω into smaller elements of characteristic size h. We then approximate the solution using simple polynomial functions (e.g., linear, quadratic) on each element.

Error Measurement: The error is the difference between the exact solution u and the finite element solution u_h. We often measure this error using the energy norm:

$$||\mathbf{e}||_E^2 = ||\mathbf{u} - \mathbf{u}_h||_E^2 = B(\mathbf{u} - \mathbf{u}_h, \mathbf{u} - \mathbf{u}_h)$$

This norm is physically meaningful as it represents the strain energy of the error.

11.12.1. A Priori Error Estimate

For well-posed problems, we can estimate the error before solving. The error in the energy norm is bounded by a constant C (independent of the mesh) times terms involving the element size h and the polynomial order p of the shape functions:

$$||\mathbf{u} - \mathbf{u}_h||_E^2 \leq C^2(u,p)h^{2\gamma}$$

Where $\gamma = \min(r-1, p)$, with r being the regularity of the solution and p being the polynomial order of the finite element shape functions.

Key Takeaway: The error decreases as the mesh is refined (as h gets smaller) or as the polynomial order of the elements (p) is increased. We can compare the solutions for two different meshes or polynomial orders to estimate the convergence rate of the method.

Part V.

Part V: Implementation & Error Analysis

12. FEM Implementation

12.1. Discretize the Domain

The first step in any Finite Element analysis is to move from the continuous domain Ω to a discrete one. We replace the single integral over the whole domain with a sum of integrals over small, simple subdomains called elements.

Our weak form is:

$$\underbrace{\int_\Omega \boldsymbol{\varepsilon}(\mathbf{v})^T \mathbf{L} \boldsymbol{\varepsilon}(\mathbf{u})\, d\Omega}_{\text{Bilinear Form } B(\mathbf{u},\mathbf{v})} = \underbrace{\int \Omega \mathbf{v}^T \mathbf{f}\, d\Omega + \int_{\Gamma_t} \mathbf{v}^T \bar{\mathbf{t}}\, d\Gamma}_{\text{Linear Form } F(\mathbf{v})}$$

Note: We've switched to transpose notation for Voigt vectors, common in implementation.

We approximate the domain Ω as a mesh of N_{el} finite elements, each with its own subdomain Ω_e.

The integral over Ω becomes a sum of integrals over each element Ω_e:

The bilinear and linear forms are now expressed as sums:

$$B(\mathbf{u}, \mathbf{v}) = \sum_{e=1}^{N_{el}} \int_{\Omega_e} \boldsymbol{\varepsilon}(\mathbf{v})^T \mathbf{L} \boldsymbol{\varepsilon}(\mathbf{u})\, d\Omega_e$$

$$F(\mathbf{v}) = \sum_{e=1}^{N_{el}} \left(\int_{\Omega_e} \mathbf{v}^T \mathbf{f}\, d\Omega_e + \int_{\Gamma_{t,e}} \mathbf{v}^T \bar{\mathbf{t}}\, d\Gamma_e \right)$$

where $\Gamma_{t,e}$ is the part of the element's boundary that lies on the traction boundary Γ_t.

12.2. Approximate Fields with Shape Functions

Within each element, we approximate the continuous displacement field using simple polynomial functions (shape functions) and the displacement values at the element's nodes.

The displacement u(x) at any point inside an element is interpolated from the element's nodal displacements d_e using shape functions:

$$\mathbf{u}(\mathbf{x}) \approx \mathbf{u}_h(\mathbf{x}) = \mathbf{N}(\mathbf{x}) \mathbf{d}_e$$

N(x) is the shape function matrix. For a 3D problem, it relates the 3 displacement components to the nodal values. For an element with n nodes, it's a 3×3n matrix:

$$\mathbf{N} = \begin{bmatrix} \phi_1 & 0 & 0 & \phi_2 & 0 & 0 & \dots & \phi_n & 0 & 0 \\ 0 & \phi_1 & 0 & 0 & \phi_2 & 0 & \dots & 0 & \phi_n & 0 \\ 0 & 0 & \phi_1 & 0 & 0 & \phi_2 & \dots & 0 & 0 & \phi_n \end{bmatrix}$$

where $\phi_i(x)$ is the scalar shape function for node i.

We do exactly the same for the test function v, relating it to a vector of arbitrary nodal values c_e:

$$\mathbf{v}(\mathbf{x}) \approx \mathbf{v}_h(\mathbf{x}) = \mathbf{N}(\mathbf{x})\mathbf{c}_e$$

12.3. Compute Strains with the B-Matrix

With the displacement field approximated, we can find the strain by taking its derivatives. This process creates the crucial strain-displacement matrix, B.

The strain vector $\boldsymbol{\varepsilon}$ is found by applying a differential operator matrix, ∂, to the displacement vector $\boldsymbol{u}$:

$$\boldsymbol{\varepsilon} = \partial\mathbf{u} \quad \text{where for 3D engineering strain,} \quad \partial = \begin{pmatrix} \partial_x & 0 & 0 \\ 0 & \partial_y & 0 \\ 0 & 0 & \partial_z \\ 0 & \partial_z & \partial_y \\ \partial_z & 0 & \partial_x \\ \partial_y & \partial_x & 0 \end{pmatrix}$$

For the approximated displacement field:

$$\boldsymbol{\varepsilon}_h = \partial\left(\mathbf{N}\mathbf{d}_e\right) = \left(\partial\mathbf{N}\right)\mathbf{d}_e$$

We define the strain-displacement matrix B as the derivatives of the shape function matrix:

$$\mathbf{B} := \partial\mathbf{N}$$

This gives the direct algebraic relationship between strain and nodal displacements for an element:

$$\boldsymbol{\varepsilon}_h = \mathbf{B}\mathbf{d}_e$$

The B matrix contains the spatial derivatives of the shape functions and is a function of the position within the element.

12.4. The B-Matrix from the Symmetric Gradient

The crucial B-matrix is not arbitrary; it arises directly from the fundamental definition of the small strain tensor, ε, as the symmetric part of the displacement gradient.

12.4.1. Start with the Strain Definition

$$\boldsymbol{\varepsilon} = \frac{1}{2}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right)$$

12.4.2. Form the Voigt Strain Vector

For implementation, we use Voigt notation with engineering shear strains γ_{ij}, where $\gamma_{ij} = \varepsilon_{ij} + \varepsilon_{ji}$, so $\gamma_{ij} = 2\varepsilon_{ij}$ for symmetric strains:

$$\boldsymbol{\varepsilon}_{\text{Voigt}} = \begin{pmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ \gamma_{yz} \\ \gamma_{xz} \\ \gamma_{xy} \end{pmatrix} = \begin{pmatrix} \partial u/\partial x \\ \partial v/\partial y \\ \partial w/\partial z \\ (\partial v/\partial z) + (\partial w/\partial y) \\ (\partial u/\partial z) + (\partial w/\partial x) \\ (\partial u/\partial y) + (\partial v/\partial x) \end{pmatrix}$$

12.4.3. Define the Differential Operator

We can express this relationship using a differential operator matrix, ∂, that acts on the displacement vector:

$$\boldsymbol{\varepsilon}_{\text{Voigt}} = \underbrace{\begin{pmatrix} \partial_x & 0 & 0 \\ 0 & \partial_y & 0 \\ 0 & 0 & \partial_z \\ 0 & \partial_z & \partial_y \\ \partial_z & 0 & \partial_x \\ \partial_y & \partial_x & 0 \end{pmatrix}}_{\partial} \underbrace{\begin{pmatrix} u \\ v \\ w \end{pmatrix}}_{\mathbf{u}}$$

This explicitly shows how the structure of ∂ is determined by the physics of strain.

12.4.4. Introduce the FE Approximation

Finally, substitute the approximation for the displacement field, $u_h = Nd_e$, into the strain definition:

$$\boldsymbol{\varepsilon}_h = \partial(\mathbf{N}\mathbf{d}_e) = (\partial \mathbf{N})\mathbf{d}_e$$

This gives us our definition of the B-matrix, $B := \partial N$, and the final element-level relationship:

$$\boldsymbol{\varepsilon}_h = \mathbf{B}\mathbf{d}_e$$

12.5. The Algebraic System: $k_e d_e = f_e$

By substituting the shape function approximations for u and v into the element integral, we transform the differential equation into a system of linear algebraic equations.

Consider the bilinear form for one element, and substitute $\boldsymbol{\varepsilon}(u) = \boldsymbol{B}\boldsymbol{d}_e$ and $\boldsymbol{\varepsilon}(v) = \boldsymbol{B}\boldsymbol{c}_e$:

$$\int_{\Omega_e} (\mathbf{B}\mathbf{c}_e)^T \mathbf{L}(\mathbf{B}\mathbf{d}_e)\, d\Omega_e = \mathbf{c}_e^T \left(\int \Omega_e \mathbf{B}^T \mathbf{L}\mathbf{B}\, d\Omega_e \right) \mathbf{d}_e$$

We define the term in the parentheses as the **element stiffness matrix** k_e:

$$\mathbf{k}_e := \int \Omega_e \mathbf{B}^T \mathbf{L}\mathbf{B}\, d\Omega_e$$

Similarly, for the linear form, we get the **element force vector** f_e:

$$\mathbf{f}_e := \int \Omega_e \mathbf{N}^T \mathbf{f}\, d\Omega_e + \int_{\Gamma_{t,e}} \mathbf{N}^T \bar{\mathbf{t}}\, d\Gamma_e$$

The weak form for the element must hold for any arbitrary c_e, which can only be true if:

$$\mathbf{k}_e \mathbf{d}_e = \mathbf{f}_e$$

12.6. Numerical Integration

The integrals for k_e and f_e involve the B and N matrices, which can be complex polynomials. These are almost always computed numerically using **Gaussian Quadrature**.

The idea is to approximate an integral by a weighted sum of the integrand's value at specific sample points, known as Gauss points.

First, we map the element Ω_e to a simple reference element. The integral transforms as:

$$\int_{\Omega_e} g(\mathbf{x})\, d\Omega_e = \int_{\hat{\Omega}} g(\mathbf{x}(\boldsymbol{\xi}))\, |\mathbf{J}(\boldsymbol{\xi})|\, d\hat{\Omega}$$

where |J| is the determinant of the Jacobian of the coordinate mapping.

The integral over the reference element is then approximated by the sum:

$$\int_{\hat{\Omega}} \tilde{g}(\boldsymbol{\xi})\, d\hat{\Omega} \approx \sum_{i=1}^{N_{gp}} w_i \tilde{g}(\boldsymbol{\xi}_i)$$

where w_i are the Gauss weights and $\boldsymbol{\xi}_i$ are the Gauss point coordinates.

12.6.1. Application to Element Stiffness

Applying this to our stiffness matrix:

$$\mathbf{k}_e = \int \Omega_e \mathbf{B}^T \mathbf{L} \mathbf{B} \, d\Omega \approx \sum_{i=1}^{N_{gp}} \left(\mathbf{B}(\boldsymbol{\xi}_i)^T \mathbf{L} \mathbf{B}(\boldsymbol{\xi}_i) \right) |\mathbf{J}(\boldsymbol{\xi}_i)| w_i$$

The force vector f_e is computed in the same way:

$$\mathbf{f}_e = \int \Omega_e \mathbf{N}^T \mathbf{f} \, d\Omega_e + \int_{\Gamma_{t,e}} \mathbf{N}^T \mathbf{t}^* \, d\Gamma_e \approx \sum_{i=1}^{N_{gp}} \left(\mathbf{N}(\boldsymbol{\xi}_i)^T \mathbf{f}(\boldsymbol{\xi}_i) + \mathbf{N}(\boldsymbol{\xi}_i)^T \mathbf{t}^*(\boldsymbol{\xi}_i) \right) |\mathbf{J}(\boldsymbol{\xi}_i)| w_i$$

12.7. Algorithmic Overview

12.7.1. Step-by-step Procedure

1. Initialize: Create empty global stiffness matrix K and force vector F.
2. Loop over Elements: For each element e in the mesh...
3. Initialize Element Matrices: Create empty local stiffness matrix k_e and force vector f_e.
4. Loop over Gauss Points: For each Gauss point i in the element...
5. Calculate at Point: Evaluate the B matrix and the Jacobian determinant |J| at this point's coordinates.
6. Accumulate Integral: Compute the integrand for k_e and f_e. Multiply by the Gauss weight and |J| and add to the element matrices.
7. Assemble: Once the loops over Gauss points are done, add the completed element matrices k_e and f_e into the global K and F based on the element's node numbering.
8. Apply Boundary Conditions: Modify the global system $Kd = F$ to enforce the prescribed Dirichlet displacements.
9. Solve: Solve the final system of linear equations for the vector of all nodal displacements d.
10. Post-Process: Use the computed displacements d to find element strains and stresses using the same matrices B and L as before at the Gauss points.

12.8. Full FEM Algorithm

Algorithm: Finite Element Procedure for 3D Elasticity

Input: Mesh (Ω_e), Material Properties ($\mathbf{L}$), Body Forces ($\mathbf{f}$), Boundary Conditions ($\bar{\mathbf{u}}, \bar{\mathbf{t}}$)
Output: Nodal Displacements ($\mathbf{d}$), Element Stresses ($\boldsymbol{\sigma}$), Element Strains ($\boldsymbol{\varepsilon}$)
Steps:
Initialize Global Stiffness $\mathbf{K} \leftarrow \mathbf{0}$ and Force $\mathbf{F} \leftarrow \mathbf{0}$.
For each element e in the mesh:

a. Initialize Element Stiffness $\mathbf{k}_e \leftarrow \mathbf{0}$ and Force $\mathbf{f}_e \leftarrow \mathbf{0}$.

b. For each Gauss point i in element e:

- Compute $\mathbf{B}(\boldsymbol{\xi}_i)$, Jacobian determinant $|\mathbf{J}(\boldsymbol{\xi}_i)|$, and weight w_i.
- Accumulate stiffness: $\mathbf{k}_e \mathrel{+}= \mathbf{B}^T\mathbf{L}\mathbf{B}\,|\mathbf{J}|w_i$
- Accumulate body forces: $\mathbf{f}_e \mathrel{+}= \mathbf{N}^T\mathbf{f}\,|\mathbf{J}|w_i$

c. Assemble traction boundary forces into $\mathbf{f}_e$ if applicable.

d. Assemble local into global: Assemble$(\mathbf{K}, \mathbf{F}, \mathbf{k}_e, \mathbf{f}_e)$.

Apply Dirichlet boundary conditions to the global system $(\mathbf{K}, \mathbf{F})$.
Solve the linear system $\mathbf{K}\mathbf{d} = \mathbf{F}$ for the displacement vector $\mathbf{d}$.
For each element e in the mesh (Post-processing):

a. Extract element displacements d_e from global d.

b. Compute strains $\boldsymbol{\varepsilon}_e = \mathbf{B}d_e$ and stresses $\boldsymbol{\sigma}_e = \mathbf{L}\boldsymbol{\varepsilon}_e$ at desired points.

12.9. Surface Integrals and Nanson's Formula

To compute integrals over an element's surface in the real domain, such as for traction boundary conditions, we must map them back to the simple reference domain. This requires a transformation for the surface area element itself.

Consider the surface integral over the traction boundary Γ_t of an element:

$$\int_{\Gamma_{t,e}} \bar{\mathbf{t}}\, d\Gamma_e$$

The integration domain $\Gamma_{t,e}$ is a 2D surface of the 3D element Γ_e. The surface domain $\Gamma_{t,e}$ is potentially distorted in real space. Mapping it to the reference element will allow us to perform the surface integral over a planar well-defined surface (e.g. a triangle or quadrilateral).

The mapping between the reference ($\boldsymbol{\xi}$) and real space ($\boldsymbol{x}$) is done using the coordinate mapping and its gradient (Jacobian):

$$|\boldsymbol{F}| = |\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}| = \boldsymbol{J}$$

The traction $\bar{\mathbf{t}}$ is defined by the stress and the vector normal to the surface $\mathbf{n}$:

$$\bar{\mathbf{t}} = \boldsymbol{\sigma} \cdot \mathbf{n}$$

12.9.1. Nanson's Formula

Nanson's Formula provides the transformation for surface integrals:

$$\boldsymbol{n} d\Gamma_e = |\boldsymbol{J}|\boldsymbol{J}^{-T} \cdot \boldsymbol{N} d\hat{\Gamma}$$

where $\boldsymbol{N}$ is the normal vector in the reference element, and $d\hat{\Gamma} = d\xi_i d\xi_j$ is the differential area in the reference element:

$$\mathbf{n}d\Gamma = \det(\mathbf{J})(\mathbf{J}^{-T})\hat{\mathbf{n}}d\hat{\Gamma}$$

The surface Jacobian is defined as:

$$J_s = \det(\mathbf{J})(\mathbf{J}^{-T})\hat{\mathbf{n}}$$

The surface integral becomes:

$$\int_{\Gamma_e} g(\mathbf{x})d\Gamma = \int_{\hat{\Gamma}} g(\mathbf{x}(\boldsymbol{\xi}))J_s(\boldsymbol{\xi})d\hat{\Gamma}$$

13. Error Estimation and Adaptivity

13.1. Simple Error Estimates for Vector Problems

After computing a solution $\mathbf{u}_h$, we need to assess its quality.

A posteriori error estimators use the solution itself to estimate the error in each element, guiding us on where the mesh needs improvement.

The core idea is to check for violations of physical principles. For elasticity, the exact stress tensor is continuous across element boundaries. Our FE stress field, $\boldsymbol{\sigma}_h$, is usually discontinuous. The size of this **stress jump** can be used as an indicator of local error.

For a face γ between elements K^+ and K^-, the jump is:

$$\mathbf{j}_\gamma = \boldsymbol{\sigma}_h^+ \mathbf{n} - \boldsymbol{\sigma}_h^- \mathbf{n}$$

The local error indicator η_K for element K sums the jumps over its faces:

$$\eta_K^2 = \frac{1}{2} \sum_{\gamma \subset \partial K} \int_\gamma ||\mathbf{j}_\gamma||^2 \, d\Gamma$$

The total estimated error is then $\eta^2 = \sum_K \eta_K^2$. This gives us a computable error value for each element.

13.2. Local Mesh Refinement (h-Adaptivity)

The power of a posteriori error estimates lies in driving **adaptive mesh refinement**.

Instead of uniformly refining the entire mesh (which is inefficient), we add elements only where the error is large. This is a simple and powerful feedback loop.

13.2.1. The Adaptive Algorithm

1. Solve: Compute the FE solution $\mathbf{u}_h$ on the current mesh.

2. Estimate: For every element K, compute the local error indicator η_K.

3. Mark: Identify elements for refinement. A common strategy is to mark all elements K where the error is a significant fraction of the maximum:

$$\text{Mark element } K \text{ if } \eta_K > \theta \cdot \eta_{\max} \quad (\text{e.g., } \theta = 0.5)$$

4. Refine: Subdivide each marked element into smaller ones (**h-refinement**).

5. Repeat: Loop back to Step 1 and solve on the new, locally denser mesh.

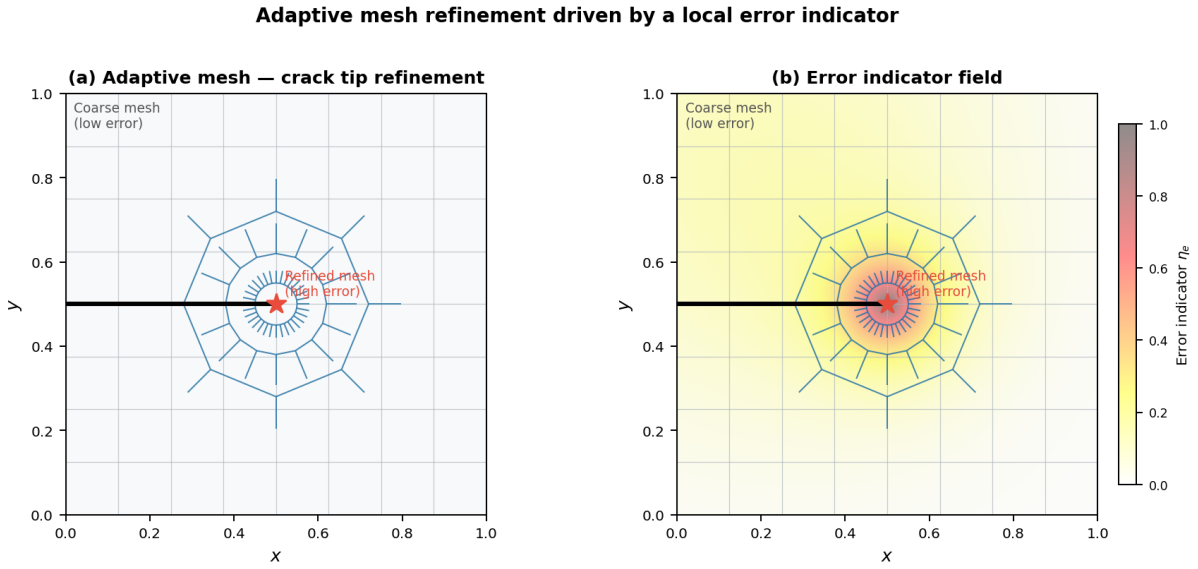


Figure 13.1.: Adaptive mesh refinement driven by a local error indicator. (a) The mesh is refined automatically near the crack tip where the error indicator η_e is large, while remaining coarse elsewhere. (b) The corresponding error indicator field, which drives the refinement loop.

13.3. A Posteriori Recovery Methods: The Core Idea

Recovery methods work on a simple principle:

The raw FE stress field ($\boldsymbol{\sigma}_h$) is "noisy" and inaccurate, but we can recover a better, smoother stress field ($\boldsymbol{\sigma}^*$) from it. The difference between the two is an excellent measure of the error.

Why recover? FE stresses are calculated from derivatives, which amplifies error and makes the field discontinuous. The recovered field $\boldsymbol{\sigma}^*$ is continuous and much closer to the true stress $\boldsymbol{\sigma}$.

13.3.1. Error Estimation Formula

We estimate the error by assuming the recovered solution is the "truth." The error in energy for an element K is the energy of the difference:

$$\eta_K^2 = \int_K (\boldsymbol{\sigma}^* - \boldsymbol{\sigma}_h)^T \mathbf{L}^{-1} (\boldsymbol{\sigma}^* - \boldsymbol{\sigma}_h)\, dK$$

The main challenge is finding a good way to "recover" $\boldsymbol{\sigma}^*$.

13.4. The Zienkiewicz-Zhu (ZZ) Estimator

The **Zienkiewicz-Zhu (ZZ) estimator** is a classic recovery technique that builds a smooth stress field using special, highly accurate points within elements.

13.4.1. Superconvergent Points

For many elements, there are specific locations—called **superconvergent points**—where the calculated stresses are much more accurate than anywhere else. For common elements, the **Gauss points** used for integration (or some of them) are superconvergent.

13.4.2. The ZZ Method

The method constructs a global, continuous stress field $\boldsymbol{\sigma}^*$ that best fits the high-quality stress values found at these superconvergent points.

1. The recovered field is approximated using the *same shape functions* as the displacement field:

$$\boldsymbol{\sigma}^*(\mathbf{x}) = \mathbf{N}(\mathbf{x})\boldsymbol{\sigma}^*_{\text{nodal}}$$

Here, $\boldsymbol{\sigma}^*_{\text{nodal}}$ is a vector of unknown, continuous stress values at the mesh nodes.

2. These nodal stresses are found by a global **least-squares fit** to the superconvergent point data. We find $\boldsymbol{\sigma}^*_{\text{nodal}}$ that minimizes:

$$\sum_{\text{elements}} \sum_{\text{Gauss pts}} ||\mathbf{N}(\boldsymbol{\xi}_i)\boldsymbol{\sigma}^*_{\text{nodal}} - \boldsymbol{\sigma}_h(\boldsymbol{\xi}_i)||^2$$

13.5. Superconvergent Patch Recovery (SPR)

Solving a global least-squares problem can be expensive. A more practical variant is **Superconvergent Patch Recovery (SPR)**, which works locally on patches of elements.

The SPR algorithm finds the recovered stress at a **single node** p:

13.5.1. 5-Step SPR Algorithm

1. **Define Patch:** Identify all elements connected to node p.
2. **Assume Polynomial:** Assume a simple polynomial for the recovered stress over the patch:

$$\boldsymbol{\sigma}^*(\mathbf{x}) = \mathbf{P}(\mathbf{x})\mathbf{a}$$

3. **Sample Points:** Collect the raw FE stress values $\boldsymbol{\sigma}_h$ from all superconvergent (Gauss) points within the patch.
4. **Local Fit:** Solve a small least-squares problem to find the polynomial coefficients $\mathbf{a}$ that best fit the sampled data.
5. **Evaluate at Node:** Use the fitted polynomial to calculate the recovered stress value directly at the node's location: $\boldsymbol{\sigma}^*_p = \mathbf{P}(\mathbf{x}_p)\mathbf{a}$.

This is repeated for all nodes to define the continuous field $\boldsymbol{\sigma}^*$, which is then used to compute the error indicators η_K.

13.6. Comparing Error Estimators

A direct comparison highlights the different strategies for evaluating FE error. One method checks for violations of physical laws, while the other tries to construct a better solution.

13.6.1. Stress Jump (Residual) Estimator

Core Philosophy: "How much does my solution violate the physical laws?"

What it Measures: Violation of equilibrium at element interfaces.

Physical Intuition: Unbalanced forces at element boundaries.

Mathematical Basis: Rigorous. Provides a proven upper bound on the error.

Pros: - Mathematically sound - Conceptually linked to the PDE

Cons: - Can sometimes underestimate the error - Doesn't provide a "better" solution to look at

13.6.2. Zienkiewicz-Zhu (Recovery) Estimator

Core Philosophy: "Can I construct a better solution from the one I have?"

What it Measures: The difference between the raw FE stress and a "cleaner" field.

Physical Intuition: Filtering a noisy signal to get a smooth one.

Mathematical Basis: Heuristic. Relies on the phenomenon of superconvergence.

Pros: - Often very accurate - Provides a useful, smooth stress field σ^* for visualization

Cons: - More complex to implement

13.6.3. Summary

The Jump Method is like a "referee" checking if the rules are followed. The ZZ Method is like an "artist" creating a better picture from a blurry photo.

13.7. A Posteriori Residual Methods: The Concept

Residual-based methods offer a more mathematically rigorous approach. They don't need a "recovered" solution. Instead, they directly measure how poorly the FE solution satisfies the original governing differential equation.

13.7.1. The Residual

The residual is what's left over when you plug the approximate solution $\mathbf{u}_h$ back into the governing PDE. If $\mathbf{u}_h$ were the exact solution, the residual would be zero everywhere:

$$\mathbf{r} = \nabla \cdot \boldsymbol{\sigma}_h + \mathbf{f} \neq \mathbf{0}$$

13.7.2. Key Idea

These methods produce a proven **upper bound** on the solution error. This means we can guarantee the true error is no larger than our estimate (times a constant):

$$\|\mathbf{u} - \mathbf{u}_h\|_E \leq \eta_{\text{residual}}$$

The final estimator is a sum of terms measuring the residual inside elements, the jumps between elements, and the error in satisfying boundary conditions.

13.8. Breaking Down the Residual Error Estimate

The total residual error is a sum of local contributions that capture all the ways our solution fails to be exact. The squared error is bounded by a sum of these terms:

$$\|\mathbf{u} - \mathbf{u}_h\|_E^2 \leq C_1 \sum_K h_K^2 \|\mathbf{r}_K\|_{L^2(K)}^2 + C_2 \sum_\gamma h_\gamma \|\mathbf{j}_\gamma\|_{L^2(\gamma)}^2 + C_3 \sum_{\gamma_t} h_\gamma \|\mathbf{t}_{\gamma_t}\|_{L^2(\gamma_t)}^2$$

13.8.1. Three Components of the Residual

1. Interior Residual ($\mathbf{r}_K$): Measures how well the equilibrium equation is satisfied *inside* each element K:

$$\mathbf{r}_K = \nabla \cdot \boldsymbol{\sigma}_h + \mathbf{f}$$

2. Interface Residual ($\mathbf{j}_\gamma$): This is the familiar **stress jump** across interior element faces, γ. It measures the violation of force balance between elements:

$$\mathbf{j}_\gamma = \boldsymbol{\sigma}_h^+ \mathbf{n} - \boldsymbol{\sigma}_h^- \mathbf{n}$$

3. Boundary Residual ($\mathbf{t}_{\gamma_t}$): Measures how well the solution satisfies the prescribed traction $\bar{\mathbf{t}}$ on Neumann boundaries γ_t:

$$\mathbf{t}_{\gamma_t} = \bar{\mathbf{t}} - \boldsymbol{\sigma}_h \mathbf{n}$$

13.8.2. Local Error Bound

The local error at an element is given by:

$$C_1 h_K^2 \|\mathbf{r}_K\|_{L^2(K)}^2 + C_2 h_\gamma \|\mathbf{j}_\gamma\|_{L^2(\gamma)}^2 + C_3 h_\gamma \|\mathbf{t}_{\gamma_t}\|_{L^2(\gamma_t)}^2$$

This decomposition allows us to identify which elements have the largest contributions to the total error, guiding adaptive refinement strategies.

14. FEniCSx Tutorial

14.1. FEniCSx

- Modern computational platform for solving partial differential equations using the finite element method
- Fully open-source and free to use
- Developed by a large community of researchers and engineers
- Written in C++ with Python bindings
- Designed for complex engineering problems you'll encounter in practice
- Can be used for both academic research and industrial applications
- Runs on laptops, workstations, and High-Performance Computers ("supercomputers")

14.2. FEniCSx for Mechanical Engineers

What you've learned:

Derivvation of the weak form Simple linear problems Simple geometries Theoretical foundations Matrix formulations Direct solvers

Real engineering:

Complex 3D geometries Nonlinear materials Coupled physics Large-scale systems Complex boundary conditions Sophisticated solvers

FEniCSx bridges this gap!

14.3. From Theory to Practice

- Once you developed the weak form in its algebric form:

$$K\mathbf{u} = \mathbf{f}$$

- FEniCSx handles:
  - Automatic mesh handling and partitioning
  - Assembly of K and $\mathbf{f}$
  - Efficient solvers (through external libraries like PETSc)
  - Post-processing

You focus on: Physics, boundary conditions, and engineering insight

14.4. Key Advantages

- High-level interface - Express problems in mathematical notation
- Automatic (symbolic) differentiation
- Access to advanced solvers and preconditioners
- Parallel computing - Scales from laptop to supercomputer
- Extensible - Custom materials, elements, and physics

14.5. Workflow Overview

1. Define the problem in mathematical terms (weak form)
2. Create a mesh of the geometry
    - simple geometries can be created directly in FEniCSx
    - complex geometries can be imported from external mesh generators (e.g. GMSH,Tetgen, ...)
3. Define function spaces (element types, polynomial orders)
4. Apply boundary conditions
5. Write the weak form using FEniCSx's UFL (Unified Form Language)
6. Solve the system of equations
7. Post-process results (visualization, data extraction)

In the following slides we will go through this workflow step by step.

The full code is available as a python file at the end of this presentation.

14.6. Simple Example - Problem Definition

The problem on which we will demonstrate the workflow is a simple 2D beam bending problem.

The strong form of the problem is given as:

$$-\nabla \cdot \sigma(u) = f \quad \text{in } \Omega$$

where σ is the stress tensor, u are the displacements Ω the domain and f the body forces.

The constitutive relation, relating the displacements to the stress tensor is given as:

$$\sigma(u) = \lambda \text{tr}(\epsilon(u))I + 2\mu\epsilon(u)$$
$$\epsilon(u) = \nabla^s u$$

with:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \text{Lame's first parameter}$$
$$\mu = \frac{E}{2(1+\nu)} \text{Lame's second parameter}$$
$$E = \text{Young's modulus}$$
$$\nu = \text{Poisson's ratio}$$
$$\nabla^s u = \frac{1}{2}(\nabla u + \nabla^T u) \text{the symmetric gradient}$$

The weak form of the problem is given as:

$$\underbrace{\int_\Omega \sigma(u) : \epsilon(v)\, d\Omega}_{a(u,v)} = \underbrace{\int_\Omega f \cdot v\, d\Omega + \int_{\partial\Omega_T} T \cdot v\, d\partial\Omega_T}_{L(v)}$$

where $v \in V$ is the test function (with V being the vector test function space), $\partial\Omega_T$ is the part of the boundary on which tractions ($T = \sigma \cdot n$) are applied.

Rewriting the weak forms in terms of a linear and bilinear forms we can phrase the problem as:

Problem Statement

Find $u \in V$ such that:

$$a(u, v) = L(v) \quad \forall v \in V$$

14.7. Simple Example: Setup

To run this example you will need to have FEniCSx installed on your system.
You can find the installation instructions on the FEniCSx website.
You can also use google's Colab to run FEniCSx notebooks without having to install anything on your local machine.
Simply open a new notebook and run the following code to install FEniCSx:

```
try:
    import dolfinx
except ImportError:
    !wget
    ↪  "https://fem-on-colab.github.io/releases/fenicsx-install-release-real.sh"
    ↪  \
    -O "/tmp/fenicsx-install.sh" && bash "/tmp/fenicsx-install.sh"
    import dolfinx
```

Note

We are using the python interface to FEniCSx - "dolfinx"

However, the same concepts apply to the C++ interface as well.

14.8. Simple Example: Mesh generation

We will make use of the built-in meshe generator in FEniCSx to create a simple rectangular mesh for our beam.

We also import numpy and mpi4py for numerical operations and parallel computing, respectively.

```
from dolfinx import mesh
import ufl
import numpy as np
from mpi4py import MPI
# Define the geometry of the beam
Length = 20.0  # Length of the beam
Height = 1.0  # Height of the beam

Nx = 20  # Number of elements in the x-direction
Ny = 4  # Number of elements in the y-direction
#create the rectangular mesh
domain = mesh.create_rectangle(MPI.COMM_WORLD,
                               [np.array([0, 0]), np.array([Length,
↪  Height])],
                               [Nx, Ny],

↪  cell_type=mesh.CellType.quadrilateral)
gdim = domain.geometry.dim # dimension
```

14.9. Simple Example: Define function space

Next, we define the function space for our problem.

We will use a vector function space with Lagrange elements of degree 1 (P1) for the displacements with gdim=2 since this is a 2D problem and we have u_x, u_y.

```
from dolfinx import fem

# Define function space
Vdegree = 1  # Degree of the polynomial
V = fem.functionspace(domain, ("P", Vdegree, (gdim,)))
# V is a vector function space with gdim=2
# Vdegree=1 means linear elements (P1)

#Define the test and trial functions
u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)
```

14.10. Simple Example: Define boundary conditions

We need to define the boundary conditions for our problem.

1. We will fix the left edge of the beam (x=0).

   We do so by defining a function that identifies the left boundary and applying a Dirichlet boundary condition with zero displacement.

```
# Define Dirichlet boundary condition (fixed left edge)
def left_boundary(x):
    return np.isclose(x[0], 0)
#locate the dofs on the left boundary
fixed_side = fem.locate_dofs_geometrical(V, left_boundary)
# Define Dirichlet boundary condition
bc = [fem.dirichletbc(np.zeros(gdim), fixed_side, V)]
```

2. We apply body force of $f = (0, -\rho g)$ on the beam, where ρ is the density and g is the gravitational acceleration.

```
from dolfinx import default_scalar_type
# Define body force (gravity)
rho = fem.Constant(domain, 2700.) # density of the material
g = fem.Constant(domain, 9.81) # gravitational acceleration
# Define body force vector
f = fem.Constant(domain, default_scalar_type((0, -rho * g)))
```

3. In our problem definition we did not define any traction boundary conditions.

   However, as an example, we show how to apply a spatially varying traction on the bottom surface of the beam.

```
def bottom_boundary(x):
    return np.isclose(x[1], 0.0)
# Mark the boundaries with unique tags
fdim = gdim - 1  # The boundary dimension for 2D problems
bottom_facet = mesh.locate_entities_boundary(domain, fdim,
↪  bottom_boundary)
# Create a MeshTag: we tag the bottom boundary with a unique tag
bottom_tag = np.full_like(bottom_facet, 1)
facet_tag = mesh.meshtags(domain, fdim, bottom_facet, bottom_tag)
# Define a function for the tracitons
Traction_Function = fem.Function(V,name="Traction_Function")
# Define the traction function
tractions = lambda x: np.vstack((np.zeros_like(x[0],
                        dtype=default_scalar_type), -1.5 * x[0]))
# Assign the traction function to the Traction_Function
Traction_Function.interpolate(tractions)
# Define facet normal vector
n_vector = ufl.FacetNormal(domain)
```

14.11. Simple Example: Construct the weak form

Now we can construct the weak form of the problem using the UFL (Unified Form Language) in FEniCSx which we prevously imported.

First, we need to define the integration measure for the boundary

```
# Define the integration measure for the boundary
ds = ufl.Measure("ds", domain=domain, subdomain_data=facet_tag)
```

Now we are almost ready to define the bilinear form $a(u, v)$ and the linear form $L(v)$.

In the bilinear form, we have the integral of the stress tensor $\sigma(u)$ multiplied by the symmetric gradient $\epsilon(v)$

So we will use the `ufl` module to define the stress tensor and the symmetric gradient.

Since we did not define the elastic properties of the material yet, we will do so now.

```
# Define material properties
E = fem.Constant(domain, 70e9)  # Young's modulus in Pa
nu = fem.Constant(domain, 0.3)  # Poisson's ratio
# Define Lame's parameters
lame = E * nu / ((1 + nu) * (1 - 2 * nu))  # Lame's first
↪  parameter
mu = E / (2 * (1 + nu))  # Lame's second parameter - shear
↪  modulus

def strain(v):
    """Define the strain tensor"""
    return ufl.sym(ufl.grad(v)) # the symmetric gradient

def stress(u):
     return lame * ufl.tr(strain(u))*ufl.Identity(gdim) + 2 * mu
     ↪  * strain(u)
     # the stress tensor
```

We can now define the bilinear form $a(u, v)$ and the linear form $L(v)$.

```
# Define the bilinear form a(u,v)
a = ufl.inner(stress(u), strain(v)) * ufl.dx
# ufl.dx is the integration measure over the domain

# Define the linear form L(v)
# L = ufl.dot(f, v) * ufl.dx

# if you wish to apply the traction boundary condition you can
# add it to the linear form as follows:
L = ufl.dot(f, v) * ufl.dx  + ufl.dot(Traction_Function,v) *
↪  ds(1)
# ds(1) is the integration measure over the bottom boundary
```

Note that we did not make use of the normal vector `n_vector` in the traction definition as we defined the traction as a vector operating in y direction and our boundary is straight in the x direction. for a curved boundary you would need to use the normal vector to define the traction in the correct direction.

14.12. Simple Example: Solve the linear problem

FEniCSx contain has a "LinearProblem" class to handle the linear problem formulation.

```
from dolfinx.fem.petsc import LinearProblem
```

We can now create a `LinearProblem` object and solve the problem.

```
# Create the linear problem
problem = LinearProblem(a, L, bcs=bc,
                petsc_options={"ksp_type": "preonly", "pc_type":
↪  "lu"})
```

In the above code, we pass the bilinear form `a`, the linear form `L`, and the boundary conditions `bc` to the `LinearProblem` constructor.

We also specify some PETSc options for the solver, such as using a direct solver (`ksp_type: preonly`) and a LU factorization (`pc_type: lu`).

PETSc is a powerful library for solving linear systems and provides various solvers and preconditioners.

We can now solve the problem by calling the `solve` method of the `LinearProblem` object.

```
# Solve the linear problem
u_solution = problem.solve()
```

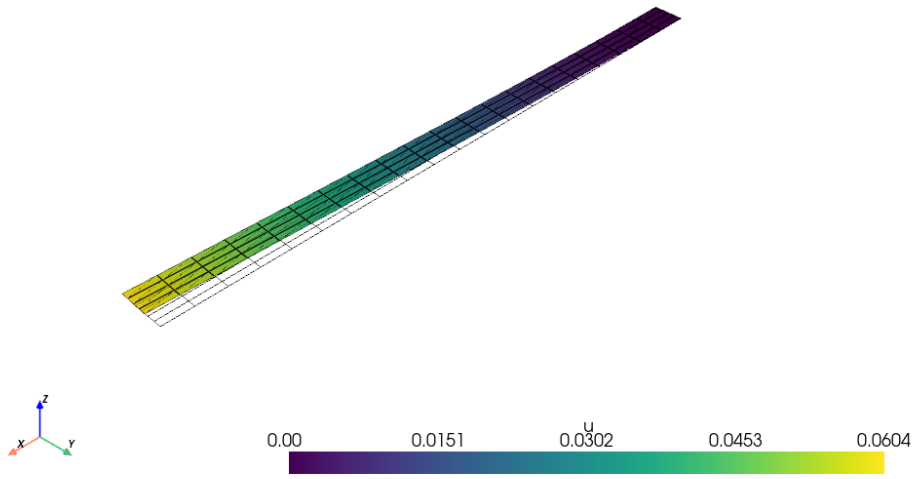
The `solve` method returns the solution vector `u_solution`, which contains the displacements at each node of the mesh.

14.13. Simple Example: Postprocessing

After solving the problem, we can visualize the results and extract useful information.

First, let's visualize the displacements using the `pyvista` library, which is a powerful visualization library for Python.

```
import pyvista as pv
pv.OFF_SCREEN = True
from dolfinx import plot
pv.set_jupyter_backend('trame')
# Create a PyVista plotter
plotter = pv.Plotter()
# Create a PyVista mesh from the FEniCSx mesh
topology, cell_types, geometry = plot.vtk_mesh(V)
grid = pv.UnstructuredGrid(topology, cell_types, geometry)
#Since Pyvista expects a 3D vector field, we need to reshape the
↪  solution
u_2d = u_solution.x.array.reshape(geometry.shape[0], gdim)
u_3d = np.zeros((geometry.shape[0], 3))
u_3d[:, :gdim] = u_2d  # Copy the x and y components
# Add the displacements ato the grid
grid["u"] = u_3d
# plot the undeformed mesh
undeformed = plotter.add_mesh(grid, style="wireframe", color="k")
# plot the deformed mesh
#create a warped mesh, and scale the displacements by a factor of
↪  your choice
factor = 10.
warped = grid.warp_by_vector("u",factor=factor)
#add the deformed mesh to the plotter
deformed = plotter.add_mesh(warped, show_edges=True)
plotter.show_axes()
#plot the two meshes.
if not pv.OFF_SCREEN:
    plotter.show()
else:
    disp_figure = plotter.screenshot("displacements.png")
```



As you recall, the displacements are defined on the nodes of the mesh. To visualize the stress we need to project it onto the appropriate function space.

Let's plot the Von Mises (or equivalent) stress.

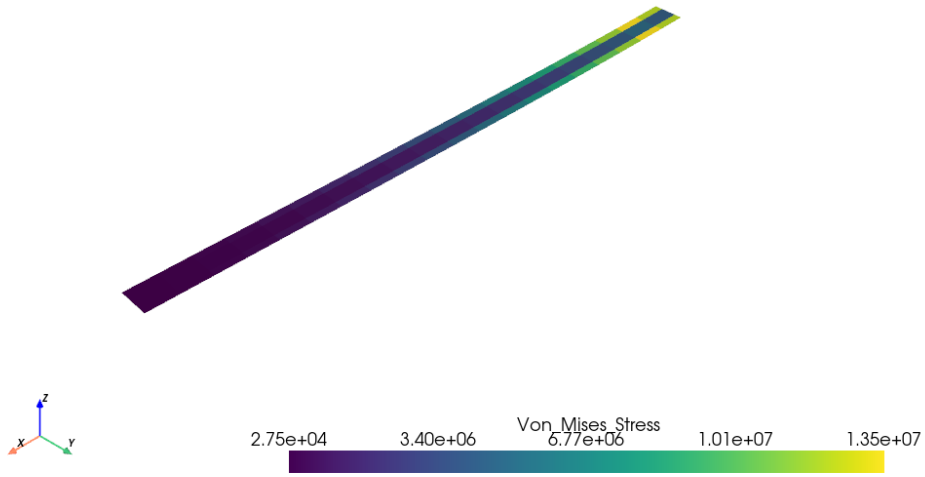
```
# Compute the stress from the solution
stress_solution = stress(u_solution)
#transfer to deviatoric stress
deviatoric_stress = (stress_solution
                     - ufl.tr(stress_solution) / gdim *
                     ↪  ufl.Identity(gdim))
# Compute the Von Mises stress
von_mises_stress = ufl.sqrt(3.0 / 2.0 *
                            ufl.inner(deviatoric_stress,
↪  deviatoric_stress))
# Create a FunctionSpace for the stress
stress_space = fem.functionspace(domain, ("DG", 0))
# Create an expression for the Von Mises stress
Mises_expression = fem.Expression(von_mises_stress,

↪  stress_space.element.interpolation_points())
VM_stress = fem.Function(stress_space)
VM_stress.interpolate(Mises_expression)
```

Now we can visualize the Von Mises stress using the `pyvista` library.

Note that now we assign values to the cells of the mesh (elements) instead of the nodes.

```
warped.cell_data["Von_Mises_Stress"] = VM_stress.x.array
warped.set_active_scalars("Von_Mises_Stress")
plotter = pv.Plotter()
plotter.add_mesh(warped)
plotter.show_axes()
if not pv.OFF_SCREEN:
    plotter.show()
else:
    VM_Stress_figure =
↪   plotter.screenshot(f"Von_Mises_Stress.png")
```



14.14. All together

```
#%%
from dolfinx import mesh
import ufl
import numpy as np
from mpi4py import MPI
# Define the geometry of the beam
Length = 20.0  # Length of the beam
Height = 1.0  # Height of the beam
Nx = 20  # Number of elements in the x-direction
Ny = 4  # Number of elements in the y-direction
#create the rectangular mesh
domain = mesh.create_rectangle(MPI.COMM_WORLD,
                               [np.array([0, 0]), np.array([Length,
↪  Height])],
                               [Nx, Ny],

↪  cell_type=mesh.CellType.quadrilateral)
gdim = domain.geometry.dim # dimension
#%%
from dolfinx import fem

# Define function space
Vdegree = 1  # Degree of the polynomial
V = fem.functionspace(domain, ("P", Vdegree, (gdim,)))
# V is a vector function space with gdim=2
# Vdegree=1 means linear elements (P1)

#Define the test and trial functions
u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)
#%%
# Define Dirichlet boundary condition (fixed left edge)
def left_boundary(x):
    return np.isclose(x[0], 0)
#locate the dofs on the left boundary
fixed_side = fem.locate_dofs_geometrical(V, left_boundary)
# Define Dirichlet boundary condition
bc = [fem.dirichletbc(np.zeros(gdim), fixed_side, V)]

#%%
from dolfinx import default_scalar_type
# Define body force (gravity)
rho = fem.Constant(domain, 2700.) # density of the material
g = fem.Constant(domain, 9.81) # gravitational acceleration
# Define body force vector
f = fem.Constant(domain, default_scalar_type((0, -rho * g)))
#%%
def bottom_boundary(x):
    return np.isclose(x[1], 0.0)
# Mark the boundaries with unique tags
fdim = gdim - 1  # The boundary dimension for 2D problems
bottom_facet = mesh.locate_entities_boundary(domain, fdim,
↪  bottom_boundary)
# Create a MeshTag: we tag the bottom boundary with a unique tag
bottom_tag = np.full_like(bottom_facet, 1)
facet_tag = mesh.meshtags(domain, fdim, bottom_facet, bottom_tag)
```

Part VI.

Part VI: Structural Elements

15. Structural Elements: Beam Theory

15.1. Structural Elements

When analyzing large structures like bridges, aircraft wings, or building frames, creating a full 3D solid model can be computationally prohibitive.

Advantage:

- **Computational Efficiency**: Reduce 3D problems to 1D or 2D
- **Physical Insight**: Directly incorporate structural behavior
- **Engineering Relevance**: Match real-world design practices
- **Reduced DOF**: Significantly fewer degrees of freedom

Common Structural Elements:

- **Truss Elements**: Axial loading only
- **Beam Elements**: Bending, shear, and axial loads
- **Frame Elements**: Combined beam behavior with rigid connections
- **Shell Elements**: Thin-walled structures
- **Plate Elements**: Flat structural components

15.2. What Kind of Beam? It Depends on the Physics

We will focus on 1D beam elements, but even here, we have choices based on the underlying physical assumptions.

Element Type	Resists	Key Assumption	Best For
Strut/Bar	Axial Loads	Joints are perfect pins.	Trusses
Euler-Bernoulli Beam	Bending & Axial	Sections stay plane and **perpendicular** to the axis.	Long, thin beams
Timoshenko Beam	Bending, Axial & **Shear**	Sections stay plane but **not necessarily perpendicular**.	Short, thick beams

15.3. Review of Beam Theory

Consider a straight rod undergoing pure bending in 2D

Let the rod be aligned with the x-axis, and let the z-axis be the down pointing vertical axis.

Assuming that:

- The Cross-sections remain straight
- Horizontal displacement are described by the rotation $\psi(x)$
- Small strain approximation applies (so the vertical displacement $w(x)$ is much larger that the contribution of the rotation)

For a given deflection $w(x)$ and rotation $\psi(x)$ the displacemnt field is given by:

$$\mathbf{u} = \begin{bmatrix} -\psi(x)z \\ 0 \\ w(x) \end{bmatrix}$$

The strain field is given by:

$$\boldsymbol{\varepsilon} = \begin{pmatrix} -\psi'(x)z & 0 & \frac{1}{2}(w'(x) - \psi(x)) \\ 0 & 0 & 0 \\ \frac{1}{2}(w'(x) - \psi(x)) & 0 & 0 \end{pmatrix}.$$

Using Hooke's law, the stress field is given by:

$$\sigma_{11}(x, z) = E\varepsilon_{11}(x, z) = -E\psi'(x)z$$

$$\sigma_{13}(x, z) = 2G\varepsilon_{13}(x, z) = G(w'(x) - \psi(x))$$

where: - E: Young's modulus - G: Shear modulus

15.4. Resultant Forces and Moments

Normal Force:

$$N(x) = \int_A \sigma_{11}(x, z)\, dA = -\int_A E\psi'(x)z\, dA = -E\psi'(x)\int_A z\, dA$$

Assumin that the x-axis is going through the nuetral axis we obtain:

$$N(x) = -E\psi'(x)\int_A z\, dA = 0$$

Shear Force:

$$Q(x) = \int_A \sigma_{13}(x, z)\, dA = \mu A(w'(x) - \psi(x))$$

Bending Moment:

$$M(x) = \int_A z\sigma_{11}(x,z)\,dA = -EI_y\psi'(x)$$

Area Moment of Inertia

$$I_y = \int_A z^2\,dA$$

15.5. Shear Correction Factor

15.5.1. Physical Justification

- Shear stress varies across cross-section
- Must be zero at free edges
- Correction factor κ accounts for non-uniform distribution

$$Q(x) = k\mu A(w'(x) - \psi(x))$$

Typical Values

- Rectangular cross-section:

$$\kappa = \frac{10(1+\nu)}{12+11\nu} \approx \frac{5}{6}$$

- Circular cross-section:

$$\kappa = \frac{6(1+\nu)}{7+6\nu} \approx \frac{6}{7}$$

15.6. Equilibrium Relations

Assume a distributed load $q(x)$ acting along the beam.

Force Equilibrium

From equilibrium of infinitesimal beam segment:

$$Q(x+dx) - Q(x) = -q(x)dx$$

rearranging, and taking the limit as $dx \to 0$ gives:

$$q(x) = -\frac{Q(x+dx) - Q(x)}{dx} \to -\frac{dQ}{dx}(x)$$

Moment Equilibrium

Angular momentum balance yields:

$$M(x+dx) - M(x) - (Q(x+dx) + Q(x))\frac{dx}{2} = 0$$

Approximating $(Q(x+dx) + Q(x))dx$ as $(Q(x) + dQ + Q(x))dx$ and assuming that dQ is small enough to be neglected, we can write:

$$Q(x) \to \frac{dM}{dx}(x)$$

Combined Relations

$$\begin{aligned} q(x) &= -Q'(x) \\ M'(x) &= Q(x) \\ M''(x) &= -q(x) \end{aligned}$$

15.7. Beam Element Types

15.7.1. Three Main Categories

1. **Strut Elements**
   - Axial deformation only
   - No bending resistance
   - 1 DOF per node (axial displacement)
2. **Euler-Bernoulli Beams**
   - Plane sections remain plane and perpendicular
   - The beam is sufficiently slender such that the transverse force is negligible: $Q(x) \approx 0$
3. **Timoshenko Beams**
   - Plane sections remain plane but not necessarily perpendicular
   - we solve for both the transverse displacement $w(x)$ and the rotation $\psi(x)$

15.8. Euler-Bernoulli vs Timoshenko

15.8.0.1. Euler-Bernoulli Theory

Key Assumption:

$$Q(x) \approx 0 \Rightarrow \psi(x) \approx w'(x)$$

Governing Equation:

$$M(x) = -EI_y w''(x)$$

$$M''(x) = -\frac{d^2}{dx^2}(EI_y w''(x)) = -q(x)$$

For constant EI_y:

$$\begin{aligned} EI_y w''''(x) &= q(x) \\ EI_y w'''(x) &= -Q(x) \\ EI_y w''(x) &= -M(x) \end{aligned}$$

15.8.1. Timoshenko Theory

Independent Variables: $w(x)$ and $\psi(x)$

Governing Equations:

$$\begin{aligned} (EI_y \psi'(x))'' &= q(x) \\ w'(x) &= \psi(x) - \frac{1}{\kappa \mu A}(EI_y \psi'(x))' \end{aligned}$$

15.9. When to Use Each Theory?

15.9.1. Euler-Bernoulli Applications

- **Slender beams**: $L/h > 10$
- Static analysis of thin beams
- Simple bending problems

15.9.2. Timoshenko Applications

- **Short/thick beams**: $L/h < 10$
- High-frequency dynamic analysis
- Composite beams
- When shear deformation is significant

15.10. Finite element Analysis with beam elements

For Euler-Bernoulli beam we can derive the energy density as:

$$W = \frac{1}{2}\varepsilon_{11}\sigma_{11} = \frac{1}{2}E(\psi'(x))^2 z^2$$

And since $\psi(x) \approx w'(x)$ we can write:

$$W = \frac{1}{2} EI_y (w''(x))^2$$

Assuming a homogeneous beam (constant E), the total potential energy of the beam is given by:

$$\Pi = \int_0^L W \, dA = EI_y w''(x)^2$$

15.11. Problem Description

We will look at a linear homogeneous beam with constant EI_y subjected to:

1. A distributed load $q(x)$ acting in the z-direction
2. Bending moments M_1 and M_2 about the y-axis (which is the axis into the plane)
3. Two forces F_1 and F_2 at the ends of the beam acting in opposite directions in the z-direction

The Strong Form

$$EI_y \frac{d^4 w(x)}{dx^4} = q(x)$$

With boundary conditions:

$$F_1 = EI_y \frac{d^3 w(0)}{dx^3} \quad , \quad F_2 = -EI_y \frac{d^3 w(L)}{dx^3}$$
$$M_1 = EI_y \frac{d^2 w(0)}{dx^2} \quad , \quad M_2 = -EI_y \frac{d^2 w(L)}{dx^2}$$

For a deflection $w(x)$ we can write the potential energy functional as:

$$\int_0^L \left[\frac{1}{2} EI_y (w'')^2 - q(x) w(x) \right] dx$$
$$- F_1 w(0) - F_2 w(L) + M_1 \frac{dw}{dx}|_{x=0} + M_2 \frac{dw}{dx}|_{x=L}$$

where we used the fact that for small deflections the beam rotation is approximated by the first derivative of the defkection and thus the work done by the torque is simplified to $M\theta = M \frac{dw}{dx}$.

15.12. Weak Formulation

Continuing from the potential energy functional

$$\int_0^L \left[\frac{1}{2}EI_y(w'')^2 - q(x)w(x)\right] dx \\ - F_1 w(0) - F_2 w(L) + M_1 \frac{dw}{dx}|_{x=0} + M_2 \frac{dw}{dx}|_{x=L}$$

We note that the deflection $w(x)$ must poses second derivatives which are square integrable.

Dirichlet Boundary Conditions

defining the Dirichlet boundary conditions as:

- **Deflection**: $w = \hat{w}$
- **Rotation**: $w' = \hat{\theta}$

We require the space of admissible functions to be:

$$w \in \mathcal{V} = [w \in H^2(0, L) \, : \, w = \hat{w}, \, \frac{dw}{dx} = \hat{\theta} \quad \text{at the boundaries}]$$

Following the standard procedure, we can derive the weak form from the potential energy functional, arriving at :

$$\mathcal{I} = \frac{1}{2}\mathcal{B}(w, v) - \mathcal{L}(w)$$

with

$$\mathcal{B}(w, v) = \int_0^L EI_y \frac{d^2w}{dx^2}\frac{d^2v}{dx^2}dx$$

$$\mathcal{L}(w) = \int_0^L qwdx + F_1 w(0) + F_2 w(L) - M_1 \frac{dw}{dx}|_{x=0} - M_2 \frac{dw}{dx}|_{x=L}$$

15.13. Shape Functions

To solve the weak form, we need to approximate the deflection $w(x)$ as $w^h(x)$ using shape functions.

$$w^h(x) = \sum_{i=1}^{n} N_i(x) w_i$$

and we can write the algebric problem as:

$$\mathbf{Kw} = \mathbf{f}$$

with $\mathbf{w}$ veing the vector of unknown coefficients w_i and $\mathbf{f}$ being the vector of load contributions.

The stiffness matrix $\mathbf{K}$ is given by:

$$\mathbf{K}_{ij} = \int_0^L EI_y \frac{d^2N_i}{dx^2} \frac{d^2N_j}{dx^2} dx$$

Let's consider an Euler-Bernoulli beam element with two nodes.

From the construction of $\mathbf{K}$ it is evident that the if we use polynomial shape functions, they must be quadratic at least, so that they will have two continuous derivatives.

As the beam equation is a fourth order differential equation, we will choose the shape functions to be spanned by the basis $\{1, x, x^2, x^3\}$, such that:

$$w_e^h(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \quad \text{for } x \in [0, L_e]$$

Now, we have four unknown coefficients a_i for each two noded element.

We will assign them such that each node will have 2 degrees of freedom: the deflection w_i and the deflection angle $\theta = \frac{dw}{dx}$.

Our Nodal conditions are thus easily shown to be:

$$\begin{aligned} w_e^h(0) &= w_1^e, \quad & w_e^h(L_e) &= w_2^e \\ w_{e,x}^h(0) &= \theta_1^e, \quad & w_{e,x}^h(L_e) &= \theta_2^e \end{aligned}$$

And we can rewrite $w^h(x)$ as:

$$w_e^h(x) = \sum_{i=1}^{2} [N_i^e(x) w_i^e + M_i^e(x) \theta_i^e]$$

where $N_i^e(x)$ are the shape functions for the deflection and $M_i^e(x)$ are the shape functions for the rotation.

This finally leads us to the following shape functions:

$$\begin{aligned} N_1^e(x) &= 1 - 3\left(\frac{x}{L_e}\right)^2 + 2\left(\frac{x}{L_e}\right)^3 \\ N_2^e(x) &= 3\left(\frac{x}{L_e}\right)^2 - 2\left(\frac{x}{L_e}\right)^3 \\ M_1^e(x) &= x - 2\frac{x^2}{L_e} + 2\frac{x^3}{L_e^2} \\ M_2^e(x) &= -\frac{x^2}{L_e} + \frac{x^3}{L_e^2} \end{aligned}$$

These polynomials are known as "Hermite polynomials"

15.14. Element Stiffness Matrix

It is common to write the nodal degrees of freedom as:

$$\mathbf{w}^e = \begin{bmatrix} w_1^e \\ \theta_1^e \\ w_2^e \\ \theta_2^e \end{bmatrix}$$

and thus the shape functions follow the same order:

$$\mathbf{N}^e = \begin{bmatrix} N_1^e(x) \\ M_1^e(x) \\ N_2^e(x) \\ M_2^e(x) \end{bmatrix}$$

Second Derivatives of Shape Functions

$$N_{1,xx} = \frac{1}{L_e^2}(-6 + 12\frac{x}{L_e})$$

$$N_{2,xx} = \frac{1}{L_e^2}(6 - 12\frac{x}{L_e})$$

$$M_{1,xx} = \frac{1}{L_e}(-4 + 6\frac{x}{L_e})$$

$$M_{2,xx} = \frac{1}{L_e}(-2 + 6\frac{x}{L_e})$$

Element Stiffness Matrix

$$\mathbf{K}^e = \frac{2EI_y}{L_e^3}\begin{bmatrix} 6 & 3L_e & -6 & 3L_e \\ 3L_e & 2L_e^2 & -3L_e & L_e^2 \\ -6 & -3L_e & 6 & -3L_e \\ 3L_e & L_e^2 & -3L_e & 2L_e^2 \end{bmatrix}$$

and the force vector is given by:

$$\tilde{\mathbf{F}}^e = \begin{bmatrix} F_1 \\ M_1 \\ F_2 \\ M_2 \end{bmatrix}$$

So that the algebric problem for the element is given by:

$$\frac{2EI_y}{L_e^3}\begin{bmatrix} 6 & 3L_e & -6 & 3L_e \\ 3L_e & 2L_e^2 & -3L_e & L_e^2 \\ -6 & -3L_e & 6 & -3L_e \\ 3L_e & L_e^2 & -3L_e & 2L_e^2 \end{bmatrix}\begin{bmatrix} w_1^e \\ \theta_1^e \\ w_2^e \\ \theta_2^e \end{bmatrix} = \begin{bmatrix} F_1 \\ M_1 \\ F_2 \\ M_2 \end{bmatrix}$$

15.15. Timoshenko Beam Elements

For Timoshenko beam elements, The element stiffness matrix is given by:

$$\tilde{\mathbf{K}}^e = \frac{EI_y}{L_e^3(1+\Phi)} \begin{bmatrix} 12 & 6L_e & -12 & 6L_e \\ 6L_e & (4+\Phi)L_e^2 & -6L_e & (2-\Phi)L_e^2 \\ -12 & -6L_e & 12 & -6L_e \\ 6L_e & (2-\Phi)L_e^2 & -6L_e & (4+\Phi)L_e^2 \end{bmatrix}$$

where

$$\Phi = \frac{12EI_y}{\kappa\mu AL_e^2}$$

Φ represents the ratio between bending and shearing of the beam. for a beam whose shear deformation is negligible, $\Phi \to 0$ and we recover the Euler-Bernoulli beam element.

15.16. Shear Locking Phenomenon

15.16.1. When Does Shear Locking Occur?

For very slender beams ($\Phi \to 0$): - Timoshenko elements become overly stiff - Shear strain $\gamma = w' - \theta \to 0$ - Artificial constraint leads to poor results

15.16.2. Remedies for Shear Locking

1. **Reduced Integration**: Under-integrate shear terms
2. **Enhanced Elements**: Add internal DOFs
3. **Mixed Formulations**: Treat shear strain as independent variable
4. **Use Euler-Bernoulli**: For slender beams

15.17. Comparison: Euler-Bernoulli vs Timoshenko

Aspect	Euler-Bernoulli	Timoshenko
Continuity	C^1	C^0
DOFs/node	2 (w, w')	2 (w, ψ)
Shape Functions	Hermite cubic	Linear
Coupling	$\psi = w'$	Independent w, ψ
Shear	Neglected	Included
Applications	Slender beams	All beam types
PDE Order	4th order	2nd order system

15.18. Industrial Applications

15.18.1. Structural Engineering

- **Building frames**: Steel and concrete structures
- **Bridges**: Continuous and simply supported spans
- **Towers**: Communication and transmission towers

15.18.2. Mechanical Engineering

- **Machine frames**: Manufacturing equipment
- **Automotive**: Chassis and suspension components
- **Aerospace**: Wing and fuselage structures

15.18.3. Civil Infrastructure

- **Pipelines**: Long-span pipe supports
- **Cranes**: Boom and jib analysis
- **Offshore**: Platform and jacket structures

15.19. Advanced Topics

15.19.1. Geometric Nonlinearity

- **Large deformations**: $P - \Delta$ effects
- **Buckling analysis**: Eigenvalue problems
- **Co-rotational formulations**: Update element orientation

15.19.2. Material Nonlinearity

- **Plastic hinges**: Concentrated plasticity
- **Distributed plasticity**: Fiber models
- **Composite materials**: Layered beam theory

15.19.3. Dynamic Analysis

- **Mass matrices**: Consistent vs lumped
- **Modal analysis**: Natural frequencies and modes
- **Time integration**: Newmark, HHT-α methods

15.19.4. Advanced Beam Theories

- **Refined beam theories**: Reddy, Levinson theories
- **Warping effects**: Non-uniform torsion
- **Composite beams**: Multi-layer analysis

15.19.5. Computational Advances

- **Isogeometric analysis**: NURBS-based elements
- **Meshfree methods**: Moving least squares
- **Machine learning**: Data-driven constitutive models

15.19.6. Multi-objective Optimization

- **Minimize weight**
- **Maximize stiffness**
- **Control natural frequencies**
- **Satisfy stress constraints**

15.19.7. Multiphysics Applications

- **Thermal effects**: Thermal expansion and buckling
- **Fluid-structure interaction**: Wind and seismic loading
- **Electromagnetic**: Induction heating analysis

15.20. Summary and Key Takeaways

15.20.1. Element Selection Guidelines

1. **Use Euler-Bernoulli** for slender beams (L/h $>$ 10)
2. **Use Timoshenko** for thick beams or dynamics
3. **Consider shear locking** for very slender beams with Timoshenko elements
4. **Choose appropriate mesh density** based on length scales

15.20.2. Implementation Best Practices

1. **Always verify** solutions with analytical benchmarks
2. **Perform convergence studies** for new problems
3. **Check boundary conditions** carefully
4. **Consider element aspect ratios**

15.20.3. Physical Understanding

1. **C^1 continuity** required for Euler-Bernoulli beams
2. **Independent rotation field** in Timoshenko theory
3. **Shear deformation** becomes important for thick beams
4. **Coordinate transformations** essential for general orientations

16. Beam Elements in FEniCSx

16.1. Example: Cantilever Beam in FEniCSx

Let's solve a cantilever beam problem: fixed at $x = 0$ and subjected to a uniform distributed load.

Since FEniCSx doesnt support C1 continous elements, required for the Euler-Bernoulli beam, we will use the Timoshenko formulation.

16.2. Timoshenko Beam in FEniCSx: Setup

To run this example you will need to have FEniCSx installed on your system.

You can find the installation instructions on the FEniCSx website.

You can also use google's Colab to run FEniCSx notebooks without having to install anything on your local machine.

Simply open a new notebook and run the following code to install FEniCSx:

```
try:
    import dolfinx
except ImportError:
    !wget
    ↪  "https://fem-on-colab.github.io/releases/fenicsx-install-release-real.sh"
    ↪  \
    -O "/tmp/fenicsx-install.sh" && bash
    ↪  "/tmp/fenicsx-install.sh"
    import dolfinx
```

Note

We are using the python interface to FEniCSx - "dolfinx"

However, the same concepts apply to the C++ interface as well.

We begin by importing the necessary libraries from FEniCSx and other dependencies:

```
iimport dolfinx
from dolfinx import fem, mesh, io
from mpi4py import MPI
import ufl
import numpy as np
import pyvista
import basix
```

16.3. Timoshenko Beam in FEniCSx: Problem Parameters

Next, we define the problem parameters:

```
# --- 1. Problem Parameters ---
# Geometry
L = 10.0  # Length of the beam
n_elem = 10 # Number of beam elements
h = 0.5   # Height of the beam cross-section
b = 0.2   # Width of the beam cross-section
↪  (out-of-plane)

# Material Properties (Steel)
E = 70e9 # Young's modulus (Pa)
nu = 0.3   # Poisson's ratio
G = E / (2 * (1 + nu)) # Shear modulus (μ)

# Load
q_val = -4000.0 # Uniformly distributed load in
↪  y-direction (N/m)

# Cross-section properties
S = b * h         # Area
I = b * h**3 / 12 # Second moment of area (for bending
↪  in x-y plane)
kappa = 5.0 / 6.0   # Shear correction factor for
↪  rectangular cross-section
```

16.4. Timoshenko Beam in FEniCSx: Mesh and Function space

Mesh Generation

While our problem is two-dimensional, our beam elements are one-dimensional.

We will create a 1D mesh and embed it in a 2D space.

```
# --- 2. Mesh Generation ---
domain = mesh.create_interval(MPI.COMM_WORLD, n_elem,
↪  [0, L])
```

Function Space Definition

We use a mixed formulation with two independent fields: - Displacement u: A 2D vector field. - Rotation θ: A scalar field.

```
# --- 3. Function Space Definition ---
Ue = basix.ufl.element("P", domain.basix_cell(), 1,
↪  shape=(2,))
Te = basix.ufl.element("P", domain.basix_cell(), 1)
# Mixed element for displacement and rotation
W = fem.functionspace(domain,
↪  basix.ufl.mixed_element([Ue, Te]))
# create subspaces for displacement and rotation
V_u, _ = W.sub(0).collapse()
V_theta, _ = W.sub(1).collapse()
```

Note

We first defined Ue and Te as the elements for the displacement and rotation fields, respectively.

Then we created a mixed function space W that combines these two elements into one.

We then collapsed the mixed function space W into two separate function spaces: V_u for displacement and V_theta for rotation.

This allows us to work with the displacement and rotation fields independently (for imposing boundary conditions) while still maintaining the mixed formulation.

16.5. Timoshenko Beam in FEniCSx: The Weak Form

Now that we have the mesh and function space, we can define the weak form of the Timoshenko beam problem.

We start by defining the test and trial functions:

```
# --- 4. Variational Formulation ---
# Define trial and test functions for the mixed system
(du, dtheta) = ufl.TrialFunctions(W)
(v_u, v_theta) = ufl.TestFunctions(W)
```

Next, we define the generalized strains: 1. Axial strain (stretching in the beam direction) 2. Bending curvature (related to the rotation) 3. Shear strain (related to the transverse displacement)

And the associated stress tensors: 1. Normal Force 2. Bending Moment 3. Shear Force

```
def get_strains(u, theta):
    # Axial strain (stretching)
    delta = u[0].dx(0)
    # Bending curvature
    chi = theta.dx(0)
    # Shear strain
    gamma = u[1].dx(0) - theta
    return ufl.as_vector([delta, chi, gamma])

# Get strains for trial and test functions
strains_trial = get_strains(du, dtheta)
strains_test = get_strains(v_u, v_theta)

# Define constitutive law
# [Normal Force, Bending Moment, Shear Force]
C = ufl.diag(ufl.as_vector([E * S, E * I, kappa * G *
↪  S]))
```

Next, we define the weak form of the problem:

```
# The Bilinear form
# a = integral( (C * strain_trial) . strain_test ) * dx
dx_shear = ufl.dx(metadata={"quadrature_degree": 0}) #
↪  reduced integration
a = ( E * S * strains_trial[0] * strains_test[0] *
↪  ufl.dx + # Axial part
      E * I * strains_trial[1] * strains_test[1] *
↪  ufl.dx + # Bending part
      kappa * G * S * strains_trial[2] * strains_test[2]
↪  * dx_shear ) # Shear part (reduced integration)
# The Linear form
q = fem.Constant(domain,
↪  dolfinx.default_scalar_type(q_val))
l_form = q * v_u[1] * ufl.dx # Distributed load on the
↪  y-component of displacement
```

Note

In the bilinear form `a`, we use `ufl.dx` for the standard integration over the domain, and `dx_shear` for the shear part, which uses reduced integration (quadrature degree 0).

This will help avoid locking issues that can occur with shear terms in Timoshenko beam elements.

16.6. Timoshenko Beam in FEniCSx: Boundary Conditions

As we stated earlier, the boundary conditions consist of: 1. Fixed boundary at $x = 0$ (both displacement and rotation are zero). 2. Distributed load applied uniformly along the beam.

We have already defined the load in the linear form `l_form`.

```
# --- 5. Boundary Conditions ---
def clamped_boundary(x):
    return np.isclose(x[0], 0)

# Locate DOFs for displacement and rotation at the
↪  clamped end (x=0)
clamped_dofs_u = fem.locate_dofs_geometrical((W.sub(0),
↪  V_u), clamped_boundary)
clamped_dofs_theta =
↪  fem.locate_dofs_geometrical((W.sub(1), V_theta),
↪  clamped_boundary)

# Create zero-value functions for the Dirichlet BCs
# displacement (u_x, u_y) and rotation (theta) are all
↪  zero.
u0 = fem.Function(V_u, name="u_clamped")
u0.x.array[:] = 0.0
theta0 = fem.Function(V_theta, name="theta_clamped")
theta0.x.array[:] = 0.0

# Apply the boundary conditions
bc_u = fem.dirichletbc(u0, clamped_dofs_u, W.sub(0))
bc_theta = fem.dirichletbc(theta0, clamped_dofs_theta,
↪  W.sub(1))
bcs = [bc_u, bc_theta]
```

16.7. Timoshenko Beam in FEniCSx: Solving the linear system

As in the previous example, we make use of the `LinearProblem` class to define the linear probelm and it's `solve` method to solve the linear system.

```
# --- 6. Solving the linear system ---
# w_sol will hold the solution for both displacement and
↪  rotation
w_sol = fem.Function(W, name="solution")
# Set up and solve the linear problem
from dolfinx.fem.petsc import LinearProblem
problem = LinearProblem(a, l_form, bcs=bcs, u=w_sol,

↪  petsc_options={"ksp_type": "preonly", "pc_type":
↪  "lu"})
problem.solve()
```

16.8. Timoshenko Beam in FEniCSx: Post-processing

We can extract the displacement and rotation fields from the solution function `w_sol`:

```
# --- 7. Post-processing ---
# Extract displacement and rotation from the solution
u_sol = w_sol.sub(0).collapse()
theta_sol = w_sol.sub(1).collapse()
```

Before visualizing the results, we will compare the numerical beam deflection with the analytical solution at the beam tip ($x = L$).

```
# Analytical solution for Timoshenko beam tip deflection
↪  under uniform load `q`
w_analytical_T = (q_val * L**4) / (8 * E * I) + (q_val *
↪  L**2) / (2 * kappa * G * S)
print(f"Timoshenko analytical tip deflection:
↪  {w_analytical_T:.6e}")

# Get the computed deflection at the tip (x=L)
print("\n--- Solution Results ---")
print(f"Maximum vertical displacement:
↪  {np.max(np.abs(u_sol.x.array[1::2])):.6f} m")
print(f"Tip displacement (x-direction):
↪  {u_sol.x.array[-2]:.6e} m")
print(f"Tip displacement (y-direction):
↪  {u_sol.x.array[-1]:.6e} m")
```

Finally, we can visualize the results using PyVista:

First we create a grid for visualization:

```
# --- 8. Visualization with PyVista ---
from dolfinx import plot
# The 1D mesh needs to be converted to a 3D object for
↪  pyvista plotting
gdim = domain.geometry.dim
topology, cell_types, geometry = plot.vtk_mesh(domain,
↪  gdim)
grid = pyvista.UnstructuredGrid(topology, cell_types,
↪  geometry)
```

Next, we add the displacements field to the grid:

```
# Add the computed displacement vector to the grid

# We need to create a 3D vector from our 2D displacement
↪  solution for pyvista
u_plot = np.zeros((geometry.shape[0], 3))
value_dim = V_u.dofmap.bs # Block size of the vector
↪  function space, which is 2 since u_sol has 2
↪  components
u_plot[:, :value_dim] = u_sol.x.array.reshape(-1,
↪  value_dim)
grid["Displacement"] = u_plot

# Warp the mesh by the displacement vector to show the
↪  deformed shape
factor = 20.  # Factor to exaggerate the deformation for
↪  visualization
warped = grid.warp_by_vector("Displacement",
↪  factor=factor)
# Create plotter
plotter = pyvista.Plotter()
plotter.add_text("Deformed Beam Structure ",
↪  font_size=15)
plotter.add_mesh(grid, style='wireframe', color='gray',
↪  label='Undeformed')
plotter.add_mesh(warped, show_edges=True, line_width=5,
↪  color='red', label='Deformed')
plotter.add_legend()
plotter.view_xy() # View in the x-y plane
plotter.show()
```

16.9. All together

Here is the complete code for the Timoshenko beam example in FEniCSx: