GENERAL

boundary element method matlab code

Published: September 17, 2026 | Index ID: #-

Introduction

When engineers and scientists tackle complex boundary value problems—whether they involve acoustic waves, electromagnetic fields, fluid flow, or structural mechanics—the Boundary Element Method (BEM) has become a go-to computational tool. Unlike finite element or finite difference techniques that discretize the entire domain, BEM reduces the dimensionality of the problem by transforming partial differential equations into integral equations defined solely on the boundary. This elegant reduction leads to fewer degrees of freedom and often more accurate solutions for problems with infinite or semi infinite domains.

MATLAB, with its powerful matrix manipulation capabilities and extensive scientific libraries, is a natural platform for implementing BEM. The combination of MATLAB's user friendly syntax, built in visualization, and robust numerical solvers makes it an ideal environment to prototype, test, and deploy **boundary element method MATLAB code**. In this article we'll walk through the fundamentals of BEM, outline the steps to build a MATLAB implementation, present a complete example, and discuss performance considerations, extensions, and resources to help you master BEM in MATLAB.

Why Choose BEM for Boundary Only Problems?

Before diving into code, it's useful to understand why BEM is attractive for certain classes of problems:

- Dimensionality Reduction: A 3 D volume problem becomes a 2 D surface problem; a 2 D domain reduces to a 1 D boundary. This often translates into fewer unknowns and faster computations.
- Exact Representation of Infinite Domains: BEM naturally handles radiation or far field conditions without artificial boundaries, making it suitable for acoustics, electromagnetics, and potential flow.
- Higher Accuracy Near Boundaries: Since the discretization is already on the boundary, field values adjacent to the surface are captured with high fidelity.
- Compatibility with Meshless and Hybrid Methods: BEM can be coupled with finite element or finite difference methods to treat heterogeneous media.

Core Concepts of the Boundary Element Method

At its heart, BEM solves boundary integral equations derived from the governing differential equations. Let's briefly recap the main steps:

1. Formulate the Governing Equation: Typically a linear partial differential equation (PDE) such as Laplace's, Helmholtz's, or the linear elasticity equations.
2. Apply Green's Second Identity: Convert the PDE into an integral equation that relates the field and its normal derivative on the boundary.
3. Discretize the Boundary: Divide the boundary into elements (line segments in 2 D, surface patches in 3 D). Choose shape functions to approximate field variables within each element.
4. Assemble System Matrix: Evaluate integrals over each element to populate the coefficient matrix. This step often involves singularity handling for self interaction terms.
5. Apply Boundary Conditions: Insert known values (Dirichlet or Neumann) into the system, reducing it to a solvable linear system.

6. Solve and Post Process: Use a linear solver to obtain unknown boundary values, then compute field quantities at desired points.

Getting Started: Setting Up Your MATLAB Workspace

Before writing any code, make sure you have the following:

- MATLAB R2020b or newer (the example uses functions introduced in R2019a).
- The Symbolic Math Toolbox (optional, for analytical integrals).
- Basic familiarity with matrix operations and plotting in MATLAB.

Create a new folder for your project, e.g., bem_mixed_boundary, and open MATLAB's current directory to that folder. This keeps all scripts, data files, and figures organized.

Step by Step: Building a Simple 2 D Laplace BEM in MATLAB

We'll implement a classic Laplace problem in two dimensions: a unit circle with mixed boundary conditions. The analytical solution is known, so we can validate our code easily.

1. Define the Geometry and Mesh

For a unit circle, the boundary can be parameterized by an angle ;,à We'll discretize it into N equally spaced nodes.

```
function [nodes, elements] = circle_mesh(N)
    theta = linspace(0, 2*pi, N+1);
    theta(end) = []; % remove duplicate point at 2<
    nodes = [cos(theta); sin(theta)]'; % Nx2 matrix
    elements = [(1:N); mod(1:N, N)+1]'; % Nx2 connectivity
end
```

The nodes array holds the (x, y) coordinates, while elements stores indices of the start and end nodes for each line segment.

2. Compute Element Lengths and Normals

We need the length of each boundary element and the outward normal vector for integration.

```
function [lengths, normals] = element_properties(nodes, elements)
    % Vector differences for each element
    delta = nodes(elements(:,2), :) - nodes(elements(:,1), :);
    lengths = sqrt(sum(delta.^2, 2));
    % Midpoint normals (rotate delta by 90° and normalize)
    normals = [ -delta(:,2), delta(:,1) ] ./ lengths;
end
```

3. Assemble the Influence Coefficient Matrix

For Laplace's equation in 2 D, the fundamental solution (Green's function) is:

G(r) = - (1/2< ' Æâ‡"•

where r is the distance between field and source points. The influence matrix A contains integrals of G and its

normal derivative over each element. For linear shape functions, these integrals can be evaluated analytically or numerically. Here we use numerical quadrature with 3 Gauss points per element.

```
function A = assemble_matrix(nodes, elements, lengths, normals)
    N = size(nodes, 1);
    A = zeros(N, N);
    % Gauss points and weights for 3  point integration
    gp = [-sqrt(3/5), 0, sqrt(3/5)]';
    w  = [5/9, 8/9, 5/9]';
    for i = 1:N % field element
        xi1 = nodes(elements(i,1), :);
        xi2 = nodes(elements(i,2), :);
        Li  = lengths(i);
        ni  = normals(i, :);
        for j = 1:N % source element
            xj1 = nodes(elements(j,1), :);
            xj2 = nodes(elements(j,2), :);
            Lj  = lengths(j);
            % If i==j handle singularity (self  interaction)
            if i == j
                % Analytical integral for self  interaction of linear element
                A(i,j) = (Li/(2*pi))*(log(2) - 1);
            else
                % Numerical integration over source element
                integral = 0;
                for k = 1:3
                    % Parametric coordinate
                    s = (gp(k)+1)/2;
                    % Shape functions
                    N1 = 1 - s;
                    N2 = s;
                    % Source point
                    x = N1 * xj1 + N2 * xj2;
                    r = norm(x - xi1); % distance to field point
                    integral = integral + w(k) * N1 * (-1/(2*pi)*log(r));
                end
                A(i,j) = (Li/2) * integral; % Jacobian factor
            end
        end
    end
end
```

Note: This implementation is intentionally simple and may not be optimal for large problems. For production code, vectorize loops and precompute shape function derivatives.

4. Apply Boundary Conditions

We'll impose a Dirichlet condition (prescribed potential) on the left half of the circle (;€ " [< ó"À 3< ó%Ò• and a Neumann condition (prescribed normal derivative) on the right half. First, identify which nodes belong to each condition.

```
function [dirichlet_nodes, neumann_nodes] = classify_boundary(nodes)
    theta = atan2(nodes(:,2), nodes(:,1));
    dirichlet_nodes = find(theta >= pi/2 & theta
```

Next, build the right hand side vector b and modify the matrix to enforce Dirichlet conditions.

```
function [A_mod, b_mod] = apply_bc(A, nodes, dirichlet_nodes, neumann_nodes)
    N = size(A,1);
    b = zeros(N,1);
    % Prescribed values
    u_D = 1; % Dirichlet potential
    q_N = 0; % Neumann flux
    % Apply Neumann conditions (already in A)
    b(neumann_nodes) = q_N;
    % Enforce Dirichlet conditions by row replacement
    for idx = dirichlet_nodes'
        A(idx,:) = 0;
        A(idx,idx) = 1;
        b(idx) = u_D;
    end
    A_mod = A; b_mod = b;
end
```

5. Solve the Linear System

Now we can solve for the unknown boundary potentials.

```
function u = solve_bem(A, b)
    u = A \ b;
end
```

6. Post Processing: Evaluate Field at Arbitrary Points

To compute the potential at a point x0 inside the domain, we integrate the Green's function against the boundary data.

```
function phi = evaluate_field(x0, nodes, elements, lengths, normals, u)
    N = size(nodes,1);
    phi = 0;
    for i = 1:N
        xi1 = nodes(elements(i,1), :);
        xi2 = nodes(elements(i,2), :);
```

```
        Li  = lengths(i);
        ni  = normals(i, :);
        % Gauss integration
        gp = [-sqrt(3/5), 0, sqrt(3/5)]';
        w  = [5/9, 8/9, 5/9]';
        integral = 0;
        for k = 1:3
            s = (gp(k)+1)/2;
            N1 = 1 - s;
            N2 = s;
            x = N1 * xi1 + N2 * xi2;
            r = norm(x0 - x);
            integral = integral + w(k) * N1 * (-1/(2*pi)*log(r));
        end
        phi = phi + (Li/2) * integral * u(i);
    end
end
```

7. Visualize the Result

Finally, plot the potential distribution on a grid and overlay the boundary.

```
function plot_solution(nodes, elements, u)
    % Create grid
    [X,Y] = meshgrid(-1.5:0.05:1.5, -1.5:0.05:1.5);
    Z = zeros(size(X));
    % Evaluate at grid points
    for i = 1:numel(X)
        Z(i) = evaluate_field([X(i), Y(i)], nodes, elements, ...
            element_lengths(nodes, elements), element_normals(nodes, elements), u);
    end
    contourf(X, Y, Z, 50, 'LineColor', 'none');
    colorbar; hold on;
    plot(nodes(:,1), nodes(:,2), 'k-', 'LineWidth', 2);
    title('Laplace Potential via B
```

Tags: #boundary #element #method #matlab #code