```
% file fem_modified.m
%
% Solves the problem
% - div ( A grad u ) + b * grad u + c u = f  in Omega
%    u = g_D          on Gamma_D
%    du/dn = g_N      on Gamma_N (Note: Neumann conditions are not fully handled in
convection part yet)
%
% Requires mesh files:
%   elem_vertices.txt
%   vertex_coordinates.txt
%   dirichlet.txt
%   neumann.txt (optional)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  problem data
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

coef_a = 0.01*eye(2); coef_b = [0; 0]; coef_c = 1;  %(i)
%coef_a = [3 -11; -11 45]; coef_b = [0; 0]; coef_c = 1; %(ii)
%coef_a = eye(2); coef_b = [30; 60]; coef_c = 0;%(iii)
%coef_a = eye(2); coef_b = [0; 0]; coef_c = -180; %(iv)

% Right-hand side function f
fc_f = @(x) 1.0; % f = 1

% Dirichlet data, function g_D
fc_gD = @(x) 0.0; %  g_D = 0 )

% Neumann data, function g_N
fc_gN = @(x) 0.0; %  g_N = 0


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  start of resolution
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% --- Load Mesh Data ---
if (exist('elem_vertices.txt','file')==2)
    elem_vertices = load('elem_vertices.txt');
else
    error('PANIC! no elem_vertices.txt file');
end
if (exist('vertex_coordinates.txt','file')==2)
    vertex_coordinates = load('vertex_coordinates.txt');
else
    error('PANIC! no vertex_coordinates.txt file');
end

if (exist('dirichlet.txt','file')==2)
    dirichlet = load('dirichlet.txt');
else
    dirichlet = [];
```

```
end

if (exist('neumann.txt','file')==2)
    neumann = load('neumann.txt');
else
    neumann = [];
end

% --- Initialization ---
n_vertices = size(vertex_coordinates, 1);
n_elem = size(elem_vertices, 1);

A_mat = sparse(n_vertices, n_vertices); % Renamed from A to avoid conflict with ↙
matrix A coefficient
fh = zeros(n_vertices, 1);
grad_uh_mag = zeros(n_elem, 1); % To store gradient magnitude per element
grad_uh_comp = zeros(n_elem, 2); % To store gradient components per element


% Gradients of the basis functions in the reference element T_ref = {(0,0),(1,0),↙
(0,1)}
% grad(phi_1) = [-1; -1], grad(phi_2) = [1; 0], grad(phi_3) = [0; 1]
grd_bas_fcts = [ -1 -1 ; 1 0 ; 0 1 ]'; % Store as 2x3 matrix: [grad(phi1) grad↙
(phi2) grad(phi3)]

% Quadrature points (midpoints of sides) and weights for reference element
% Using the midpoint rule mentioned in original fem.m: \int_T f approx |T|/3 * sum↙
(f(midpoints))
% For element matrix integrals like \int_T c*phi_i*phi_j, the original code used↙
analytical formula.
% For convection \int_T (b.grad(phi_j)) * phi_i dx
% We can approximate phi_i by its average 1/3 and grad(phi_j) is constant
% Integral approx = (b . grad(phi_j)) * (Area / 3)
quad_weights_mass = [1/6 1/12 1/12; 1/12 1/6 1/12; 1/12 1/12 1/6]; % Used for↙
reaction term C*phi_i*phi_j


% --- Assemble Stiffness Matrix and Load Vector ---
fprintf('Assembling matrix and vector...\n');
for el = 1 : n_elem
    v_elem = elem_vertices( el, : ); % Vertex indices for this element

    % Get vertex coordinates
    v1 = vertex_coordinates( v_elem(1), :)' ; % Column vector v1 = [x1; y1]
    v2 = vertex_coordinates( v_elem(2), :)' ; % Column vector v2 = [x2; y2]
    v3 = vertex_coordinates( v_elem(3), :)' ; % Column vector v3 = [x3; y3]

    % Affine map: F(x_ref) = B * x_ref + v1, where x_ref is in reference element
    B = [ v2-v1  v3-v1 ]; % Transformation matrix B (2x2)
    Binv = inv(B);          % Inverse transformation
    BinvT = Binv';          % Transpose of inverse

    % Element area
```

```
    el_area = abs(det(B)) * 0.5;

    % --- Assemble Load Vector Contribution ---
    % Quadrature points (midpoints of sides in physical element)
    m12 = (v1 + v2) / 2;
    m23 = (v2 + v3) / 2;
    m31 = (v3 + v1) / 2;

    % Evaluate f at quadrature points
    f12 = fc_f(m12);
    f23 = fc_f(m23);
    f31 = fc_f(m31);

    % Element load vector using midpoint rule for \int_T f * phi_i dx
    % phi_1(m12)=0.5, phi_1(m23)=0, phi_1(m31)=0.5 -> integral f*phi1 approx area/3
* (f12*0.5 + f31*0.5) = area/6 * (f12+f31)
    % phi_2(m12)=0.5, phi_2(m23)=0.5, phi_2(m31)=0 -> integral f*phi2 approx area/3
* (f12*0.5 + f23*0.5) = area/6 * (f12+f23)
    % phi_3(m12)=0, phi_3(m23)=0.5, phi_3(m31)=0.5 -> integral f*phi3 approx area/3
* (f23*0.5 + f31*0.5) = area/6 * (f23+f31)
    % The original code used a slightly different formula, let's stick to that one:
    f_el = [ (f12+f31)*0.5 ; (f12+f23)*0.5 ; (f23+f31)*0.5 ] * (el_area/3);

    % Add to global load vector
    fh( v_elem ) = fh( v_elem ) + f_el;

    % --- Assemble Element Matrix Contribution ---
    % Gradient of basis functions in the physical element (constant vectors)
    % grad(phi_i) = BinvT * grad_ref(phi_i)
    grad_phi_phys = BinvT * grd_bas_fcts; % 2x3 matrix [grad(phi1) grad(phi2) grad
(phi3)]

    % 1. Diffusion Term: \int_T (A grad(phi_j)) . grad(phi_i) dx
    % = (A grad(phi_j)) . grad(phi_i) * area
    % Note: grad_phi_phys(:, j) is grad(phi_j)
    el_mat_diff = zeros(3,3);
    for i = 1:3
        for j = 1:3
            el_mat_diff(i,j) = (coef_a * grad_phi_phys(:,j))' * grad_phi_phys(:,i)
* el_area;
            % Equivalent: trace(grad_phi_phys(:,i)' * coef_a * grad_phi_phys(:,j))
* el_area
        end
    end
    % Faster way using matrix operations:
    % el_mat_diff = grad_phi_phys' * coef_a * grad_phi_phys * el_area;


    % 2. Convection Term: \int_T (b . grad(phi_j)) * phi_i dx
    % Approximation: (b . grad(phi_j)) * \int_T phi_i dx = (b . grad(phi_j)) *
(Area / 3)
    el_mat_conv = zeros(3,3);
     for i = 1:3
```

```
        for j = 1:3
            b_dot_grad_phi_j = coef_b' * grad_phi_phys(:,j);
            el_mat_conv(i,j) = b_dot_grad_phi_j * (el_area / 3);
        end
     end

    % 3. Reaction Term: \int_T c * phi_i * phi_j dx
    el_mat_react = coef_c * el_area * quad_weights_mass;

    % Combine terms for element matrix
    el_mat = el_mat_diff + el_mat_conv + el_mat_react;

    % Add to global matrix
    A_mat( v_elem, v_elem ) = A_mat( v_elem, v_elem ) + el_mat;

end
fprintf('Assembly finished.\n');

% --- Neumann Boundary Conditions ---
% (Note: Convection term b*u might also contribute to boundary integral if using ↙
integration by parts,
% depending on formulation. This implementation adds only g_N for the diffusion ↙
part's natural BC.)
if (~isempty(neumann))
  fprintf('Applying Neumann conditions...\n');
  n_neumann_segments = size(neumann, 1);
  for i = 1:n_neumann_segments % Corrected loop limit
    v_seg = neumann(i, :);
    v1_coords = vertex_coordinates( v_seg(1) , : ); % Row vector
    v2_coords = vertex_coordinates( v_seg(2) , : ); % Row vector
    m_coords = (v1_coords + v2_coords) / 2; % Row vector

    segment_length = norm(v2_coords-v1_coords);

    g1 = fc_gN(v1_coords'); % Pass as column vector
    g2 = fc_gN(v2_coords'); % Pass as column vector
    gm = fc_gN(m_coords');  % Pass as column vector

    % Simpson's rule for \int_edge g_N * phi_i ds
    % phi_1 on edge is linear from 1 to 0. phi_2 is linear from 0 to 1.
    % \int phi_1 ds = length/2, \int phi_2 ds = length/2
    % Using Simpson: \int f = L/6 * (f(a)+4f(m)+f(b))
    % \int g_N*phi_1 ds approx L/6 * ( (g1*1 + 4*gm*0.5 + g2*0) ) = L/6 * (g1 + ↙
2*gm)
    % \int g_N*phi_2 ds approx L/6 * ( (g1*0 + 4*gm*0.5 + g2*1) ) = L/6 * (2*gm + ↙
g2)
    f_seg = [ g1+2*gm ;  2*gm+g2 ] * segment_length / 6;

    fh( v_seg ) = fh( v_seg ) + f_seg;
  end
  fprintf('Neumann conditions applied.\n');
end
```

```
% --- Dirichlet Boundary Conditions ---
fprintf('Applying Dirichlet conditions...\n');
for i = 1:length(dirichlet)
  diri_node = dirichlet(i); % Node index
  % Get node coordinates
  diri_coords = vertex_coordinates(diri_node, :); % Row vector

  % Set row in matrix to identity
  A_mat(diri_node,:) = 0; % Zero out the row
  A_mat(diri_node,diri_node) = 1; % Set diagonal to 1

  % Set right-hand side to Dirichlet value g_D
  fh(diri_node) = fc_gD( diri_coords' ); % Pass as column vector
end
fprintf('Dirichlet conditions applied.\n');

% --- Solve the Linear System ---
fprintf('Solving linear system...\n');
uh = A_mat \ fh;
fprintf('System solved.\n');


% --- Post-processing: Calculate Gradient Magnitude ---
fprintf('Calculating gradient magnitude...\n');
for el = 1 : n_elem
    v_elem = elem_vertices( el, : ); % Vertex indices for this element
    v1 = vertex_coordinates( v_elem(1), :)' ;
    v2 = vertex_coordinates( v_elem(2), :)' ;
    v3 = vertex_coordinates( v_elem(3), :)' ;

    B = [ v2-v1  v3-v1 ];
    BinvT = inv(B)';

    % grad(uh)|_T = sum( uh_j * grad(phi_j) )
    % grad(phi_j) = BinvT * grad_ref(phi_j)
    grad_phi_phys = BinvT * grd_bas_fcts; % 2x3 matrix [grad(phi1) grad(phi2) grad↙
(phi3)]

    % Gradient of uh on this element (constant)
    grad_uh_elem = grad_phi_phys * uh(v_elem); % (2x3)*(3x1) = (2x1) vector
    grad_uh_comp(el, :) = grad_uh_elem'; % Store components
    grad_uh_mag(el) = norm(grad_uh_elem); % Calculate L2 norm (magnitude)
end
fprintf('Gradient magnitude calculated.\n');

% --- Plotting ---
fprintf('Plotting results...\n');

% Plot Solution u_h
fig1= figure;
trisurf(elem_vertices, vertex_coordinates(:,1), vertex_coordinates(:,2), uh,↙
'EdgeColor', 'none', 'FaceColor', 'interp');
```

```
view(2); % Top-down view
axis equal;
colorbar;
title('Finite Element Solution u_h');
xlabel('x');
ylabel('y');


% Plot Gradient Magnitude ||grad u_h||
% Need to create values at vertices for smooth plotting or plot piecewise constant
% For piecewise constant plot:
figure;
patch('Faces', elem_vertices, 'Vertices', vertex_coordinates, ...
      'FaceVertexCData', grad_uh_mag, 'FaceColor', 'flat', 'EdgeColor', 'none');
view(2);
axis equal;
colorbar;
title('Gradient Magnitude ||\nabla u_h||_2');
xlabel('x');
ylabel('y');

fprintf('Plotting finished.\n');
```