```
% AMSC 714 - Homework 4 - Problem 1 Solver
% Solves Poisson equation on L-shaped domain for smooth and nonsmooth solutions.

clear; clc; close all;

% --- Problem Parameters ---
k_values = 2:7;
h_values = 2.^(-k_values);
n_levels = length(k_values);

% Storage for results
L2_errors_smooth = zeros(1, n_levels);
H1_errors_smooth = zeros(1, n_levels);
L2_errors_nonsmooth = zeros(1, n_levels);
H1_errors_nonsmooth = zeros(1, n_levels);

% --- Define Exact Solutions and Forcing Functions ---

% Smooth Case
u_s = @(x) cos(pi*x(1)).*sin(pi*x(2));
f_s = @(x) 2*pi^2*cos(pi*x(1)).*sin(pi*x(2));
grad_u_s = @(x) [-pi*sin(pi*x(1)).*sin(pi*x(2)); pi*cos(pi*x(1)).*cos(pi*x(2))]; %↙
Returns 2x1

% Nonsmooth Case
u_ns = @(x) nonsmooth_solution(x);
f_ns = @(x) 0 * x(1); % Zero RHS (ensure output is scalar)
grad_u_ns = @(x) grad_nonsmooth_solution(x); % Returns 2x1

% --- Main Loop for Refinement Levels ---
for idx = 1:n_levels
    k = k_values(idx);
    N = 2^k;
    h = h_values(idx);
    fprintf('Running k=%d, N=%d, h=%f\n', k, N, h);

    % 1. Generate Mesh
    [vertex_coordinates, elem_vertices, dirichlet_nodes] = generate_mesh_L_shape↙
(N);

    % 2. Solve FEM
    % Smooth Case
    fprintf('  Solving smooth case...\n');
    uh_smooth = solve_fem(vertex_coordinates, elem_vertices, dirichlet_nodes, f_s,↙
u_s);

    % Nonsmooth Case
    fprintf('  Solving nonsmooth case...\n');
    uh_nonsmooth = solve_fem(vertex_coordinates, elem_vertices, dirichlet_nodes,↙
f_ns, u_ns);

    % 3. Compute Errors
    fprintf('  Computing errors...\n');
```

```
    % Smooth Errors
    L2_errors_smooth(idx) = compute_L2_error(elem_vertices, vertex_coordinates, ↙
uh_smooth, u_s);
    H1_errors_smooth(idx) = compute_H1_error(elem_vertices, vertex_coordinates, ↙
uh_smooth, grad_u_s);

    % Nonsmooth Errors
    L2_errors_nonsmooth(idx) = compute_L2_error(elem_vertices, vertex_coordinates, ↙
uh_nonsmooth, u_ns);
    H1_errors_nonsmooth(idx) = compute_H1_error(elem_vertices, vertex_coordinates, ↙
uh_nonsmooth, grad_u_ns);

    fprintf('  Errors (Smooth):   L2=%.3e, H1=%.3e\n', L2_errors_smooth(idx), ↙
H1_errors_smooth(idx));
    fprintf('  Errors (Nonsmooth): L2=%.3e, H1=%.3e\n', L2_errors_nonsmooth(idx), ↙
H1_errors_nonsmooth(idx));
end

% --- Plotting Results ---
fprintf('Plotting results...\n');
figure;
loglog(h_values, L2_errors_smooth, 'o-', 'LineWidth', 1.5, 'MarkerSize', 8, ↙
'DisplayName', 'L2 Error (Smooth)');
hold on;
loglog(h_values, H1_errors_smooth, 's-', 'LineWidth', 1.5, 'MarkerSize', 8, ↙
'DisplayName', 'H1 Error (Smooth)');
loglog(h_values, L2_errors_nonsmooth, 'o--', 'LineWidth', 1.5, 'MarkerSize', 8, ↙
'DisplayName', 'L2 Error (Nonsmooth)');
loglog(h_values, H1_errors_nonsmooth, 's--', 'LineWidth', 1.5, 'MarkerSize', 8, ↙
'DisplayName', 'H1 Error (Nonsmooth)');

% Add reference slopes
loglog(h_values, (H1_errors_smooth(1)/h_values(1)) * h_values, 'k:', 'DisplayName', ↙
'O(h)');
loglog(h_values, (L2_errors_smooth(1)/h_values(1)^2) * h_values.^2, 'k-.', ↙
'DisplayName', 'O(h^2)');
loglog(h_values, (H1_errors_nonsmooth(end)/h_values(end)^(2/3)) * h_values.^(2/3), ↙
'r:', 'DisplayName', 'O(h^{2/3})');
loglog(h_values, (L2_errors_nonsmooth(end)/h_values(end)^(4/3)) * h_values.^(4/3), ↙
'r-.', 'DisplayName', 'O(h^{4/3})');


set(gca, 'XDir','reverse'); % h decreases to the right
xlabel('Mesh size h');
ylabel('Error');
title('FEM Convergence on L-shaped Domain (Problem 1)');
legend('show', 'Location', 'best');
grid on;
hold off;

% --- Calculate and Display EOC ---
fprintf('\nExperimental Orders of Convergence (EOC):\n');
eoc_L2_s = log2(L2_errors_smooth(1:end-1) ./ L2_errors_smooth(2:end));
```

```
eoc_H1_s = log2(H1_errors_smooth(1:end-1) ./ H1_errors_smooth(2:end));
eoc_L2_ns = log2(L2_errors_nonsmooth(1:end-1) ./ L2_errors_nonsmooth(2:end));
eoc_H1_ns = log2(H1_errors_nonsmooth(1:end-1) ./ H1_errors_nonsmooth(2:end));

fprintf('  Smooth L2 EOC:   %s\n', sprintf('%.2f ', eoc_L2_s));
fprintf('  Smooth H1 EOC:   %s\n', sprintf('%.2f ', eoc_H1_s));
fprintf('  Nonsmooth L2 EOC: %s\n', sprintf('%.2f ', eoc_L2_ns));
fprintf('  Nonsmooth H1 EOC: %s\n', sprintf('%.2f ', eoc_H1_ns));

fprintf('\nDone.\n');


% ================================================================
% ================================================================

function u = nonsmooth_solution(x)
    % Calculates u(x,y) = r^(2/3) * sin(2*theta/3)
    r_sq = x(1,:).^2 + x(2,:).^2;
    u = zeros(size(x(1,:))); % Ensure row vector output if input is 2xn
    valid = r_sq > eps; % Avoid calculation at origin
    if any(valid)
        r = sqrt(r_sq(valid));
        theta = atan2(x(2,valid), x(1,valid));
        u(valid) = r.^(2/3) .* sin(2/3 * theta);
    end
    % Ensure output dimensions match input expectations (e.g., scalar if input is ↙
2x1)
    if size(x, 2) == 1
        u = u(1);
    end
end

% ----------------------------------------------------------------

function grad = grad_nonsmooth_solution(x)
    % Calculates gradient of u(r,theta) = r^(2/3) * sin(2*theta/3)
    r_sq = x(1)^2 + x(2)^2;
    if r_sq <= 1e-12 % Increased tolerance for gradient singularity
        grad = [0; 0]; % Gradient is singular at origin
    else
        r = sqrt(r_sq);
        theta = atan2(x(2), x(1));

        % Derivatives w.r.t r and theta
        dudr = (2/3) * r^(-1/3) * sin(2/3 * theta);
        dudt = r^(2/3) * (2/3) * cos(2/3 * theta);

        % Derivatives of r, theta w.r.t x, y
        drdx = x(1)/r; drdy = x(2)/r;
        dtdx = -x(2)/r_sq; dtdy = x(1)/r_sq;

        % Chain rule
        gradx = dudr * drdx + dudt * dtdx;
```

```
        grady = dudr * drdy + dudt * dtdy;
        grad = [gradx; grady]; % Return as 2x1 column vector
    end
end

% ----------------------------------------------------------------

function [vertex_coordinates, elem_vertices, dirichlet_nodes] =↙
generate_mesh_L_shape(N)
    % Based on gen_mesh_L_shape.m
    vertex_coordinates = [];
    elem_vertices = [];
    dirichlet_nodes = [];

    deltax = 1/N;
    deltay = 1/N;

    % Vertex coordinates
    vertex_list = [];
    xmin = -1; xmax = 0;
    ymin = -1; ymax = 0;
    for i = 0 : N-1
      for j = 0 : N
        vertex_list = [vertex_list; xmin + j*deltax, ymin + i*deltay];
      end
    end
    xmin = -1; xmax = 1;
    ymin =  0; ymax = 1;
    for i = 0 : N
      for j = 0 : 2*N
        vertex_list = [vertex_list; xmin + j*deltax, ymin + i*deltay];
      end
    end
    vertex_coordinates = vertex_list;

    % Element vertices
    elem_list = [];
    for i = 0 : N-1
      for j = 1 : N
        v1 = i*(N+1)+j;
        v2 = (i+1)*(N+1)+j+1;
        v3 = (i+1)*(N+1)+j;
        v4 = i*(N+1)+j+1;
        elem_list = [elem_list; v1, v2, v3];
        elem_list = [elem_list; v2, v1, v4];
      end
    end
    first = N*(N+1); % Offset for upper part nodes
    for i = 0 : N-1
      for j = 1 : 2*N
        v1 = first + i*(2*N+1)+j;
        v2 = first + (i+1)*(2*N+1)+j+1;
        v3 = first + (i+1)*(2*N+1)+j;
```

```
        v4 = first + i*(2*N+1)+j+1;
        elem_list = [elem_list; v1, v2, v3];
        elem_list = [elem_list; v2, v1, v4];
      end
    end
    elem_vertices = elem_list;

    % Dirichlet nodes
    dirichlet_list = [];
    % Base (y = -1)
    dirichlet_list = [dirichlet_list, 1:(N+1)];
    % Lower vertical sides (-1 < y < 0, x=-1 or x=0)
    for i = 1 : N-1
        dirichlet_list = [dirichlet_list, i*(N+1)+1, i*(N+1)+N+1];
    end
    dirichlet_list = [dirichlet_list, N*(N+1)+1]; % Include corner (-1,0)
    % Lower part of upper block (y=0, 0<=x<=1)
    for j = 1 : N+1 % These indices are relative to the second block
         node_idx = N*(N+1) + (N+1) + j; % Node index in full list
         dirichlet_list = [dirichlet_list, node_idx];
    end
     % Add node (0,0) if not already included - it's N*(N+1)+N+1 from first block
    dirichlet_list = [dirichlet_list, N*(N+1)+N+1];

    % Upper vertical sides (0 < y < 1, x=-1 or x=1)
    first_upper = N*(N+1);
    for i = 1 : N-1
        dirichlet_list = [dirichlet_list, first_upper+i*(2*N+1)+1, first_upper+i*↙
(2*N+1)+2*N+1];
    end
    % Top (y=1)
    last_row_start_idx = N*(N+1) + N*(2*N+1) + 1;
    dirichlet_list = [dirichlet_list, last_row_start_idx : (last_row_start_idx +↙
2*N)];

    dirichlet_nodes = unique(dirichlet_list); % Ensure unique nodes and sort
end

% ----------------------------------------------------------------

function uh = solve_fem(vertex_coordinates, elem_vertices, dirichlet_nodes, fc_f,↙
fc_gD)
    % Based on fem.m
    coef_a = 1.0;
    coef_c = 0.0;

    n_vertices = size(vertex_coordinates, 1);
    n_elem = size(elem_vertices, 1);

    A  = sparse(n_vertices, n_vertices);
    fh = zeros(n_vertices, 1);

    % Gradients of basis functions in reference element
```

```
    grd_bas_fcts = [ -1 -1 ; 1 0 ; 0 1 ]' ; % Shape 2x3

    for el = 1 : n_elem
        v_elem = elem_vertices( el, : ); % Indices (row vector)
        v_coords = vertex_coordinates(v_elem, :)'; % Coordinates (2x3 matrix)

        v1 = v_coords(:,1); v2 = v_coords(:,2); v3 = v_coords(:,3);

        % Midpoints for quadrature
        m12 = (v1 + v2) / 2;
        m23 = (v2 + v3) / 2;
        m31 = (v3 + v1) / 2;

        % Eval RHS f at quadrature points
        f12 = feval(fc_f, m12); f23 = feval(fc_f, m23); f31 = feval(fc_f, m31);

        % Affine map B and area
        B = [ v2-v1  v3-v1 ];
        el_area = abs(det(B)) * 0.5;
        if el_area < 1e-14
            warning('Element %d has zero area.', el);
            continue;
        end

        % Element load vector (midpoint rule)
        f_el = [ (f12+f31)*0.5 ; (f12+f23)*0.5 ; (f23+f31)*0.5 ] * (el_area/3);
        fh( v_elem ) = fh( v_elem ) + f_el;

        % Element stiffness matrix
        Binv = inv(B);
        el_stiff = coef_a * grd_bas_fcts' * (Binv*Binv') * grd_bas_fcts * el_area;

        % Element mass matrix (for reaction term if coef_c ~= 0)
        el_mass = coef_c * el_area * [ 1/6 1/12 1/12 ; 1/12 1/6 1/12; 1/12 1/12 ↙
1/6];
        el_mat = el_stiff + el_mass;

        % Assemble into global matrix
        A( v_elem, v_elem ) = A( v_elem, v_elem ) + el_mat;
    end

    % Apply Dirichlet boundary conditions
    for i = 1:length(dirichlet_nodes)
      diri_node = dirichlet_nodes(i);
      A(diri_node,:) = 0; % Clear row
      A(diri_node, diri_node) = 1;
      fh(diri_node) = feval(fc_gD, vertex_coordinates(diri_node, :)'); % Apply g_D
    end

    % Solve system
    uh = A \ fh;
end
```

```
% ----------------------------------------------------------------

function error = compute_L2_error(elem_vertices, vertex_coordinates, uef, u_exact)
    % Based on L2_err.m
    n_elem = size(elem_vertices,1);
    integral_sq_diff = 0;

    for i = 1 : n_elem
        v_elem = elem_vertices(i, :); % Node indices
        v_coords = vertex_coordinates( v_elem, : )'; % Node coordinates 2x3

        v1 = v_coords(:,1); v2 = v_coords(:,2); v3 = v_coords(:,3);

        % Midpoints
        m12 = (v1 + v2) / 2; m23 = (v2 + v3) / 2; m31 = (v3 + v1) / 2;

        % Exact solution at midpoints
        u12 = feval(u_exact, m12); u23 = feval(u_exact, m23); u31 = feval(u_exact,↙
m31);

        % FE solution at midpoints (linear interpolation)
        uef12 = ( uef(v_elem(1)) + uef(v_elem(2)) ) / 2;
        uef23 = ( uef(v_elem(2)) + uef(v_elem(3)) ) / 2;
        uef31 = ( uef(v_elem(3)) + uef(v_elem(1)) ) / 2;

        % Element area
        B = [ v2-v1 , v3-v1 ];
        el_area = abs(det(B))/2;
        if el_area < 1e-14; continue; end

        % Midpoint quadrature for squared error integral
        integral_el = ((u12 - uef12)^2 + (u23 - uef23)^2 + (u31 - uef31)^2) / 3 *↙
el_area;
        integral_sq_diff = integral_sq_diff + integral_el;
    end
    error = sqrt(integral_sq_diff);
end

% ----------------------------------------------------------------

function error = compute_H1_error(elem_vertices, vertex_coordinates, uef,↙
grad_u_exact)
    % Based on H1_err.m
    n_elem = size(elem_vertices,1);
    integral_sq_diff_grad = 0;

    % Gradients of basis functions in reference element (2x3)
    grd_bas_fcts = [ -1 -1 ; 1 0 ; 0 1 ]' ;

    for i = 1 : n_elem
        v_elem = elem_vertices(i, :); % Node indices
        v_coords = vertex_coordinates( v_elem, : )'; % Node coordinates 2x3
```

```
        v1 = v_coords(:,1); v2 = v_coords(:,2); v3 = v_coords(:,3);

        % Affine map and area
        B = [ v2-v1 , v3-v1 ];
        el_area = abs(det(B)) / 2;
         if el_area < 1e-14; continue; end
        BinvT = (B') \ eye(2); % Equivalent to inv(B')

        % Gradient of FE function (constant on element)
        grad_uef_elem = BinvT * (grd_bas_fcts * uef(v_elem)); % Should be 2x1

        % Midpoints
        m12 = (v1 + v2) / 2; m23 = (v2 + v3) / 2; m31 = (v3 + v1) / 2;

        % Exact gradient at midpoints
        gradu12 = feval(grad_u_exact, m12); % Expect 2x1
        gradu23 = feval(grad_u_exact, m23); % Expect 2x1
        gradu31 = feval(grad_u_exact, m31); % Expect 2x1

        % Differences (squared norm) at midpoints
        diff12_sq = sum((grad_uef_elem - gradu12).^2);
        diff23_sq = sum((grad_uef_elem - gradu23).^2);
        diff31_sq = sum((grad_uef_elem - gradu31).^2);

        % Midpoint quadrature for squared gradient error integral
        integral_el = (diff12_sq + diff23_sq + diff31_sq) / 3 * el_area;
        integral_sq_diff_grad = integral_sq_diff_grad + integral_el;
    end
    error = sqrt(integral_sq_diff_grad);
end
% ================================================================
```