```
function [normalized_points, T] = normalize_points_2d(points)
    % Normalize 2D image points.
    % Inputs:
    %   points: Nx2 matrix of INHOMOGENOUS 2D coordinates
    % Outputs:
    %   normalized_points: Nx3 matrix of NORMALIZED, HOMOGENOUS points
    %   T: 3x3 similarity transformation matrix

    % Convert to homogeneous coordinates
    points_h = [points, ones(size(points, 1), 1)];

    % Compute centroid of the points
    centroid = mean(points, 1);

    % Translate points to have centroid at the origin
    translated_points = points - centroid;

    % Compute scaling factor to make the average distance from the origin sqrt(2)
    avg_dist = mean(sqrt(sum(translated_points.^2, 2)));
    scale = sqrt(2) / avg_dist;

    % Construct the similarity transformation matrix
    T = [scale, 0, -scale * centroid(1);
         0, scale, -scale * centroid(2);
         0, 0, 1];

    % Normalize the points
    normalized_points = (T * points_h')';
end


function [normalized_points, T] = normalize_points_3d(points)
    % Normalize 3D space points.
    % Inputs:
    %   points: Nx3 matrix of INHOMOGENOUS 3D coordinates
    % Outputs:
    %   normalized_points: Nx4 matrix of NORMALIZED, HOMOGENOUS points
    %   T: 4x4 similarity transformation matrix

    % Convert to homogeneous coordinates
    points_h = [points, ones(size(points, 1), 1)];

    % Compute centroid of the points
    centroid = mean(points, 1);

    % Translate points to have centroid at the origin
    translated_points = points - centroid;

    % Compute scaling factor to make the average distance from the origin sqrt(3)
    avg_dist = mean(sqrt(sum(translated_points.^2, 2)));
    scale = sqrt(3) / avg_dist;

    % Construct the similarity transformation matrix
    T = [scale, 0, 0, -scale * centroid(1);
         0, scale, 0, -scale * centroid(2);
         0, 0, scale, -scale * centroid(3);
         0, 0, 0, 1];

    % Normalize the points
```

```
    normalized_points = (T * points_h')';
end

function P_tilde = dlt(image_points, world_points)
    % Compute the direct linear transformation.
    % Inputs:
    %   image_points: Nx3 matrix of (normalized) HOMOGENOUS 2D image points
    %   world_points: Nx4 matrix of (normalized) HOMOGENOUS 3D world points
    % Output:
    %   P: 3x4 camera matrix

    num_points = size(world_points, 1);
    A = zeros(2 * num_points, 12);

    for i = 1:num_points
        X = world_points(i, :);
        x = image_points(i, 1);
        y = image_points(i, 2);

        A(2*i-1, :) = [zeros(1, 4), -X, y * X];
        A(2*i, :) = [X, zeros(1, 4), -x * X];
    end

    % Solve for P using SVD
    [~, ~, V] = svd(A);
    P_tilde = reshape(V(:, end), [4, 3])';
end

function P = compute_camera_matrix(image_points, world_points)
    % Compute the camera matrix using the Gold Standard algorithm.
    % Inputs:
    %   image_points: Nx2 matrix of 2D INHOMOGENOUS image coordinates
    %   world_points: Nx3 matrix of 3D INHOMOGENOUS world coordinates
    % Output:
    %   P: 3x4 camera matrix

    n_image_points = size(image_points, 1);
    n_world_points = size(world_points, 1);

    if n_image_points ~= n_world_points
        error("Mismatched number of image and world coordinates.");
    end
    if n_image_points < 6
        error("At least six pairs of correspondence required.");
    end

    % Step 1: Normalize points
    [image_points_normalized, U] = normalize_points_2d(image_points);
    [world_points_normalized, T] = normalize_points_3d(world_points);

    % Step 2: Apply DLT with normalized coordinates
    P_tilde = dlt(image_points_normalized, world_points_normalized);

    % Step 3: Denormalize the camera matrix
    P = U \ P_tilde * T;
end

% Demo
image_points = [
```

```
    100, 200;
    200, 200;
    200, 300;
    100, 300;
    150, 250;
    120, 280
];
world_points = [
    0, 0, 0;
    1, 0, 0;
    1, 1, 0;
    0, 1, 0;
    0.5, 0.5, 1;
    0.2, 0.8, 0.5
];

compute_camera_matrix(image_points, world_points)
```