Homework 4

Problem 1. Point Cloud Fusion. The goal is to fuse the point clouds of three different views of a scene into a single point cloud.

Method 1 problem 1 code:

```
clc
clear all

% Load depth images
depth1 = imread('depth1.png');
depth2 = imread('depth2.png');
depth3 = imread('depth3.png');

% Load RGB images
rgb1 = imread('rgb1.png');
rgb2 = imread('rgb2.png');
rgb3 = imread('rgb3.png');

% Compute image derivatives Ix and Iy
Ix1 = conv2(double(rgb1(:,:,1)), [-1, 0, 1], 'same');
Iy1 = conv2(double(rgb1(:,:,1)), [-1; 0; 1], 'same');
Ix2 = conv2(double(rgb2(:,:,1)), [-1, 0, 1], 'same');
Iy2 = conv2(double(rgb2(:,:,1)), [-1; 0; 1], 'same');
Ix3 = conv2(double(rgb3(:,:,1)), [-1, 0, 1], 'same');
Iy3 = conv2(double(rgb3(:,:,1)), [-1; 0; 1], 'same');

% Apply Gaussian smoothing to the derivatives
sigma = 1; % Standard deviation of Gaussian
G = fspecial('gaussian', [5, 5], sigma);
Ix1_smooth = conv2(Ix1, G, 'same');
Iy1_smooth = conv2(Iy1, G, 'same');
Ix2_smooth = conv2(Ix2, G, 'same');
Iy2_smooth = conv2(Iy2, G, 'same');
Ix3_smooth = conv2(Ix3, G, 'same');
Iy3_smooth = conv2(Iy3, G, 'same');

% Compute the Harris operator response function
alpha = 0.04; % Harris corner constant
M1 = (Ix1_smooth .* Ix1_smooth) .* (Iy1_smooth .* Iy1_smooth) - alpha *
((Ix1_smooth .* Ix1_smooth) + (Iy1_smooth .* Iy1_smooth)).^2;
M2 = (Ix2_smooth .* Ix2_smooth) .* (Iy2_smooth .* Iy2_smooth) - alpha *
((Ix2_smooth .* Ix2_smooth) + (Iy2_smooth .* Iy2_smooth)).^2;
M3 = (Ix3_smooth .* Ix3_smooth) .* (Iy3_smooth .* Iy3_smooth) - alpha *
((Ix3_smooth .* Ix3_smooth) + (Iy3_smooth .* Iy3_smooth)).^2;

% Apply non-maximum suppression
corner_thresh = 0.01 * max(max([M1(:); M2(:); M3(:)])); % Threshold for
corner detection
corners1 = imregionalmax(M1) & (M1 > corner_thresh);
corners2 = imregionalmax(M2) & (M2 > corner_thresh);
corners3 = imregionalmax(M3) & (M3 > corner_thresh);

% Pick the 100 corners with the strongest response
max_corners = 100;
```

```
[y1, x1] = find(corners1);
[y2, x2] = find(corners2);
[y3, x3] = find(corners3);
% Sort corners based on their response values
[sorted_values1, sorted_indices1] = sort(M1(corners1), 'descend');
[sorted_values2, sorted_indices2] = sort(M2(corners2), 'descend');
[sorted_values3, sorted_indices3] = sort(M3(corners3), 'descend');

% Select the top 100 corners with the strongest response
strongest_corners1 = [x1(sorted_indices1(1:max_corners)),
y1(sorted_indices1(1:max_corners))];
strongest_corners2 = [x2(sorted_indices2(1:max_corners)),
y2(sorted_indices2(1:max_corners))];
strongest_corners3 = [x3(sorted_indices3(1:max_corners)),
y3(sorted_indices3(1:max_corners))];

% Display the corners on the images
figure;
subplot(1, 3, 1);
imshow(rgb1);
hold on;
plot(strongest_corners1(:, 1), strongest_corners1(:, 2), 'r*');
title('Corners on rgb1');
subplot(1, 3, 2);
imshow(rgb2);
hold on;
plot(strongest_corners2(:, 1), strongest_corners2(:, 2), 'r*');
title('Corners on rgb2');
subplot(1, 3, 3);
imshow(rgb3);
hold on;
plot(strongest_corners3(:, 1), strongest_corners3(:, 2), 'r*');
title('Corners on rgb3');

% Define camera intrinsic parameters
S = 5000; % Scale factor
K = [525.0, 0, 319.5; 0, 525.0, 239.5; 0, 0, 1]; % Intrinsic matrix

% Initialize arrays to store 3D points
points3D_rgb1 = zeros(size(strongest_corners1, 1), 3);
points3D_rgb2 = zeros(size(strongest_corners2, 1), 3);
points3D_rgb3 = zeros(size(strongest_corners3, 1), 3);

% Iterate over each detected corner and compute its 3D point
for i = 1:size(strongest_corners1, 1)
    x = strongest_corners1(i, 1);
    y = strongest_corners1(i, 2);
    d = double(depth1(y, x)) / S; % Convert depth to real-world scale
    if d ~= 0
        points3D_rgb1(i, :) = [d * (x - K(1, 3)) / K(1, 1), d * (y - K(2, 3))
/ K(2, 2), d];
    end
end

for i = 1:size(strongest_corners2, 1)
```

```
    x = strongest_corners2(i, 1);
    y = strongest_corners2(i, 2);
    d = double(depth2(y, x)) / S;
    if d ~= 0
        points3D_rgb2(i, :) = [d * (x - K(1, 3)) / K(1, 1), d * (y - K(2, 3))
/ K(2, 2), d];
    end
end

for i = 1:size(strongest_corners3, 1)
    x = strongest_corners3(i, 1);
    y = strongest_corners3(i, 2);
    d = double(depth3(y, x)) / S;
    if d ~= 0
        points3D_rgb3(i, :) = [d * (x - K(1, 3)) / K(1, 1), d * (y - K(2, 3))
/ K(2, 2), d];
    end
end

% Remove rows with zeros (where depth is unknown)
points3D_rgb1 = points3D_rgb1(any(points3D_rgb1, 2), :);
points3D_rgb2 = points3D_rgb2(any(points3D_rgb2, 2), :);
points3D_rgb3 = points3D_rgb3(any(points3D_rgb3, 2), :);

% Count the number of 3D points for each RGB image
num_points_rgb1 = size(points3D_rgb1, 1);
num_points_rgb2 = size(points3D_rgb2, 1);
num_points_rgb3 = size(points3D_rgb3, 1);

% Display the number of 3D points for each RGB image
disp(['Number of 3D points for rgb1: ', num2str(num_points_rgb1)]);
disp(['Number of 3D points for rgb2: ', num2str(num_points_rgb2)]);
disp(['Number of 3D points for rgb3: ', num2str(num_points_rgb3)]);

% Compute the rank transform in 5x5 windows for each image
rank_transform1 = zeros(size(rgb1, 1), size(rgb1, 2));
rank_transform2 = zeros(size(rgb2, 1), size(rgb2, 2));
rank_transform3 = zeros(size(rgb3, 1), size(rgb3, 2));

for i = 3:size(rgb1, 1) - 2
    for j = 3:size(rgb1, 2) - 2
        rank_transform1(i, j) = sum(sum(M1(i-2:i+2, j-2:j+2) > M1(i, j)));
        rank_transform2(i, j) = sum(sum(M2(i-2:i+2, j-2:j+2) > M2(i, j)));
        rank_transform3(i, j) = sum(sum(M3(i-2:i+2, j-2:j+2) > M3(i, j)));
    end
end

% Compute SAD distances between corners of image 2 and image 1
matches_img1 = zeros(size(strongest_corners2, 1), 2);
for k = 1:size(strongest_corners2, 1)
    x2 = strongest_corners2(k, 1);
    y2 = strongest_corners2(k, 2);

    % Define window boundaries, ensuring it stays within the image bounds
    window_min_x2 = max(1, x2 - 5);
```

```
    window_max_x2 = min(size(rgb2, 2), x2 + 5);
    window_min_y2 = max(1, y2 - 5);
    window_max_y2 = min(size(rgb2, 1), y2 + 5);

    window2 = rank_transform2(window_min_y2:window_max_y2,
window_min_x2:window_max_x2); % Dynamic window around the corner in image 2

    best_match_distance = Inf;
    best_match_index = 0;

    for m = 1:num_points_rgb1
        if isempty(points3D_rgb1(m, :)) % Skip points without 3D
representation
            continue;
        end

        x1 = strongest_corners1(m, 1);
        y1 = strongest_corners1(m, 2);

        % Define window boundaries, ensuring it stays within the image bounds
        window_min_x1 = max(1, x1 - 5);
        window_max_x1 = min(size(rgb1, 2), x1 + 5);
        window_min_y1 = max(1, y1 - 5);
        window_max_y1 = min(size(rgb1, 1), y1 + 5);

        window1 = rank_transform1(window_min_y1:window_max_y1,
window_min_x1:window_max_x1); % Dynamic window around the corner in image 1

        % Ensure the windows have the same size
        window1_resized = imresize(window1, size(window2));

        % Compute SAD distance
        sad_distance = sum(abs(window1_resized(:) - window2(:)));

        if sad_distance < best_match_distance
            best_match_distance = sad_distance;
            best_match_index = m;
        end
    end

    matches_img1(k, :) = [best_match_index, best_match_distance];
end

% Select the top 10 matches for each corner of image 2
[sorted_distances_img1, sorted_indices_img1] = sort(matches_img1(:, 2));
top_matches_img1 = matches_img1(sorted_indices_img1(1:10), :);

% Compute SAD distances between corners of image 2 and image 3
matches_img3 = zeros(size(strongest_corners2, 1), 2);
for k = 1:size(strongest_corners2, 1)
    x2 = strongest_corners2(k, 1);
    y2 = strongest_corners2(k, 2);

    % Define window boundaries, ensuring it stays within the image bounds
    window_min_x2 = max(1, x2 - 5);
```

```
    window_max_x2 = min(size(rgb2, 2), x2 + 5);
    window_min_y2 = max(1, y2 - 5);
    window_max_y2 = min(size(rgb2, 1), y2 + 5);

    window2 = rank_transform2(window_min_y2:window_max_y2,
window_min_x2:window_max_x2);

    best_match_distance = Inf;
    best_match_index = 0;

    for m = 1:num_points_rgb3
        if isempty(points3D_rgb3(m, :))
            continue;
        end

        x3 = strongest_corners3(m, 1);
        y3 = strongest_corners3(m, 2);

        % Define window boundaries, ensuring it stays within the image bounds
        window_min_x3 = max(1, x3 - 5);
        window_max_x3 = min(size(rgb3, 2), x3 + 5);
        window_min_y3 = max(1, y3 - 5);
        window_max_y3 = min(size(rgb3, 1), y3 + 5);

        window3 = rank_transform3(window_min_y3:window_max_y3,
window_min_x3:window_max_x3); % Dynamic window around the corner in image 3

        % Ensure the windows have the same size
        window3_resized = imresize(window3, size(window2));

        % Compute SAD distance
        sad_distance = sum(abs(window3_resized(:) - window2(:)));

        if sad_distance < best_match_distance
            best_match_distance = sad_distance;
            best_match_index = m;
        end
    end

    matches_img3(k, :) = [best_match_index, best_match_distance];
end

% Select the top 10 matches for each corner of image 2 with image 3
[sorted_distances_img3, sorted_indices_img3] = sort(matches_img3(:, 2));
top_matches_img3 = matches_img3(sorted_indices_img3(1:10), :);

% Display the matches
figure;
imshow(rgb2);
hold on;
for i = 1:10
    % Image 1 matches
    x1 = strongest_corners1(top_matches_img1(i, 1), 1);
    y1 = strongest_corners1(top_matches_img1(i, 1), 2);
    plot(x1, y1, 'go');
```

```
    plot([x1, strongest_corners2(i, 1)], [y1, strongest_corners2(i, 2)], 'g-
');

    % Image 3 matches
    x3 = strongest_corners3(top_matches_img3(i, 1), 1);
    y3 = strongest_corners3(top_matches_img3(i, 1), 2);
    plot(x3, y3, 'bo');
    plot([x3, strongest_corners2(i, 1)], [y3, strongest_corners2(i, 2)], 'b-
');
end
title('Top 10 matches between corners of image 2 and image 1 (green) / image
3 (blue)');



% Estimate Relative Pose between Images 2 and 1
% Initialize variables to store the best transformation parameters
best_R1 = [];
best_t1 = [];
best_inliers_count1 = 0;
best_inliers_idx1 = [];

% RANSAC parameters
num_iterations = 1000;
threshold_distance = 0.5; % Threshold for inlier selection

for iter = 1:num_iterations
    % Randomly select 3 matches
    rand_indices = randperm(size(top_matches_img1, 1), 3);
    selected_matches = top_matches_img1(rand_indices, :);

    % Extract corresponding 3D points
    points3D_img2 = points3D_rgb2(selected_matches(:, 1), :);
    points3D_img1 = points3D_rgb1(selected_matches(:, 1), :);

    % Compute vectors for rotation estimation
    v2_1 = points3D_img2(1, :) - points3D_img2(2, :);
    v2_2 = points3D_img2(2, :) - points3D_img2(3, :);
    v1_1 = points3D_img1(1, :) - points3D_img1(2, :);
    v1_2 = points3D_img1(2, :) - points3D_img1(3, :);

    % Estimate rotation matrix
    R1_transpose = [v2_1; v2_2; cross(v2_1, v2_2)] / [v1_1; v1_2; cross(v1_1,
v1_2)];
    [U, ~, V] = svd(R1_transpose);
    R1 = U * V';

    % Estimate translation vector
    t1 = points3D_img2(1, :)' - R1 * points3D_img1(1, :)';

    % Apply transformation to all 3D points of image 2 instead of image 1
    transformed_points_img2 = (R1 * points3D_rgb2' + t1)';

    % Compute distances between transformed points and original points of
image 2
```

```
    distances = sqrt(sum((transformed_points_img2 - points3D_rgb2).^2, 2));


    % Count inliers
    inliers_idx = find(distances < threshold_distance);
    inliers_count = numel(inliers_idx);

    % Check if this model is the best so far
    if inliers_count > best_inliers_count1
        best_inliers_count1 = inliers_count;
        best_inliers_idx1 = inliers_idx;
        best_R1 = R1;
        best_t1 = t1;
    end
end

% Use best transformation parameters
R1 = best_R1;
t1 = best_t1;
inliers_idx1 = best_inliers_idx1;
inliers_count1 = best_inliers_count1;

disp(['Estimated relative pose between images 2 and 1:']);
disp(['Rotation matrix (R1):']);
disp(R1);
disp(['Translation vector (t1):']);
disp(t1);
disp(['Number of inliers: ', num2str(inliers_count1)]);


% Repeat the above process for images 2 and 3

best_R3 = [];
best_t3 = [];
best_inliers_count3 = 0;
best_inliers_idx3 = [];

for iter = 1:num_iterations
    % Randomly select 3 matches
    rand_indices = randperm(size(top_matches_img3, 1), 3);
    selected_matches = top_matches_img3(rand_indices, :);

    % Extract corresponding 3D points
    points3D_img2 = points3D_rgb2(selected_matches(:, 1), :);
    points3D_img3 = points3D_rgb3(selected_matches(:, 1), :);

    % Compute vectors for rotation estimation
    v2_1 = points3D_img2(1, :) - points3D_img2(2, :);
    v2_2 = points3D_img2(2, :) - points3D_img2(3, :);
    v3_1 = points3D_img3(1, :) - points3D_img3(2, :);
    v3_2 = points3D_img3(2, :) - points3D_img3(3, :);

    % Estimate rotation matrix
    R3_transpose = [v2_1; v2_2; cross(v2_1, v2_2)] / [v3_1; v3_2; cross(v3_1,
v3_2)];
```

```
    [U, ~, V] = svd(R3_transpose);
    R3 = U * V';

    % Estimate translation vector
    t3 = points3D_img2(1, :)' - R3 * points3D_img3(1, :)';

    % Apply transformation to all 3D points of image 2 instead of image 1
    transformed_points_img3 = (R3 * points3D_rgb2' + t3)';

    % Compute distances between transformed points and original points of
image 2
    distances = sqrt(sum((transformed_points_img3 - points3D_rgb2).^2, 2));

    % Count inliers
    inliers_idx = find(distances < threshold_distance);
    inliers_count = numel(inliers_idx);

    % Check if this model is the best so far
    if inliers_count > best_inliers_count3
        best_inliers_count3 = inliers_count;
        best_inliers_idx3 = inliers_idx;
        best_R3 = R3;
        best_t3 = t3;
    end
end

% Use best transformation parameters
R3 = best_R3;
t3 = best_t3;
inliers_idx3 = best_inliers_idx3;
inliers_count3 = best_inliers_count3;

disp(['Estimated relative pose between images 2 and 3:']);
disp(['Rotation matrix (R3):']);
disp(R3);
disp(['Translation vector (t3):']);
disp(t3);
disp(['Number of inliers: ', num2str(inliers_count3)]);


% Initialize arrays to store 3D points and colors
points3D_all_rgb1 = zeros(size(rgb1, 1), size(rgb1, 2), 3);
colors_all_rgb1 = zeros(size(rgb1, 1), size(rgb1, 2), 3);
points3D_all_rgb2 = zeros(size(rgb2, 1), size(rgb2, 2), 3);
colors_all_rgb2 = zeros(size(rgb2, 1), size(rgb2, 2), 3);
points3D_all_rgb3 = zeros(size(rgb3, 1), size(rgb3, 2), 3);
colors_all_rgb3 = zeros(size(rgb3, 1), size(rgb3, 2), 3);

% Iterate over each pixel to compute 3D points and colors
for y = 1:size(rgb1, 1)
    for x = 1:size(rgb1, 2)
        % Compute 3D points for image 1
        Z1 = double(depth1(y, x)) / S;
        if Z1 ~= 0
            points3D_all_rgb1(y, x, :) = (K \ [x; y; 1]) * Z1;
```

```
            colors_all_rgb1(y, x, :) = double(rgb1(y, x, :));
        end

        % Compute 3D points for image 2
        Z2 = double(depth2(y, x)) / S;
        if Z2 ~= 0
            points3D_all_rgb2(y, x, :) = (K \ [x; y; 1]) * Z2;
            colors_all_rgb2(y, x, :) = double(rgb2(y, x, :));
        end

        % Compute 3D points for image 3
        Z3 = double(depth3(y, x)) / S;
        if Z3 ~= 0
            points3D_all_rgb3(y, x, :) = (K \ [x; y; 1]) * Z3;
            colors_all_rgb3(y, x, :) = double(rgb3(y, x, :));
        end
    end
end

% Transform 3D points of image 1 using R1 and t1
transformed_points_all_rgb1 = zeros(size(points3D_all_rgb1));

for y = 1:size(rgb1, 1)
    for x = 1:size(rgb1, 2)
        if depth1(y, x) ~= 0  % Only process points with non-zero depth
            % Apply transformation to each 3D point

            % Matrix multiplication
            temp_result = R1 * squeeze(points3D_all_rgb1(y, x, :));
            % Addition
            temp_result = temp_result' + t1';
            % Scalar division
            temp_result = temp_result / S;
            % Transpose and assignment
            transformed_points_all_rgb1(y, x, :) = temp_result';
        end
    end
end

% Transform 3D points of image 3 using R3 and t3
transformed_points_all_rgb3 = zeros(size(rgb3, 1), size(rgb3, 2), 3);
for y = 1:size(rgb3, 1)
    for x = 1:size(rgb3, 2)
        if depth3(y, x) ~= 0
            % Apply transformation to each 3D point
            % Matrix multiplication
            temp_result = R3 * squeeze(points3D_all_rgb3(y, x, :));
            % Addition
            temp_result = temp_result' + t3';
            % Scalar division
            temp_result = temp_result / S;
            % Transpose and assignment
            transformed_points_all_rgb3(y, x, :) = temp_result';
        end
    end
```

```
end

% Merge transformed 3D points and colors
merged_points = [reshape(transformed_points_all_rgb1, [], 3);
reshape(transformed_points_all_rgb3, [], 3); reshape(points3D_all_rgb2, [],
3)];
merged_colors = [reshape(colors_all_rgb1, [], 3); reshape(colors_all_rgb3,
[], 3); reshape(colors_all_rgb2, [], 3)];

% Write merged point cloud to a PLY file
writePLY('merged_point_cloud.ply', merged_points, merged_colors);


function writePLY(filename, points, colors)
    % Open file for writing
    fid = fopen(filename, 'w');

    % Write header
    fprintf(fid, 'ply\n');
    fprintf(fid, 'format ascii 1.0\n');
    fprintf(fid, 'element vertex %d\n', size(points, 1));
    fprintf(fid, 'property float x\n');
    fprintf(fid, 'property float y\n');
    fprintf(fid, 'property float z\n');
    fprintf(fid, 'property uchar red\n');
    fprintf(fid, 'property uchar green\n');
    fprintf(fid, 'property uchar blue\n');
    fprintf(fid, 'end_header\n');

    % Write points and colors
    for i = 1:size(points, 1)
        fprintf(fid, '%f %f %f %d %d %d\n', points(i, 1), points(i, 2),
points(i, 3), colors(i, 1), colors(i, 2), colors(i, 3));
    end

    % Close file
    fclose(fid);
end
```

Method 2:

Part 1. Harris Corner Detection.

```
image1_top_100_coords = detectCorners('rgb1.png');
image2_top_100_coords = detectCorners('rgb2.png');
image3_top_100_coords = detectCorners('rgb3.png');

function top_100_coords = detectCorners(image_path)
    im = imread(image_path);
    im_gray = rgb2gray(im);
    dx = [-1 0 1; -1 0 1; -1 0 1];
    dy = dx';
    Ix = conv2(double(im_gray), dx, 'same');
    Iy = conv2(double(im_gray), dy, 'same');
    sigma = 1.4;
    kernel_size = 5;
    gaussian_kernel = fspecial('gaussian', [kernel_size, kernel_size],
sigma);
    smoothed_Ix = conv2(Ix, gaussian_kernel, 'same');
    smoothed_Iy = conv2(Iy, gaussian_kernel, 'same');
    Ix2 = smoothed_Ix .^ 2;
    Iy2 = smoothed_Iy .^ 2;
    Ixy = smoothed_Ix .* smoothed_Iy;

    kernel_size = 3;
    sum_all = ones([kernel_size, kernel_size]);
    Sx2 = conv2(Ix2, sum_all, 'same');
    Sy2 = conv2(Iy2, sum_all, 'same');
    Sxy = conv2(Ixy, sum_all, 'same');

    k = 0.04;

    [numOfRows, numOfColumns] = size(im_gray);
    corners = zeros(numOfRows, numOfColumns);
    for x = 2:numOfRows-1
        for y = 2:numOfColumns-1
            M = [Sx2(x, y), Sxy(x, y); Sxy(x, y), Sy2(x, y)];
            R = det(M) - k * (trace(M) ^ 2);
            corners(x, y) = R;
        end
    end
    suppressed_corners = zeros(size(corners));
    for x = 2:size(corners, 1)-1
        for y = 2:size(corners, 2)-1
            neighborhood = corners(x-1:x+1, y-1:y+1);
            if corners(x, y) == max(neighborhood(:))
                suppressed_corners(x, y) = corners(x, y);
            end
        end
    end
```

```
    corners = suppressed_corners;
    corner_responses = corners(:);
    [~, sorted_indices] = sort(corner_responses, 'descend');
    top_100_indices = sorted_indices(1:100);
    [top_100_rows, top_100_cols] = ind2sub(size(corners), top_100_indices);
    top_100_coords = [top_100_rows, top_100_cols];
end
```

Part 2. Corners to 3D points.

```
depth_map1 = imread("depth1.png");
depth_map2 = imread("depth2.png");
depth_map3 = imread("depth3.png");
fx = 525.0;
fy = 525.0;
cx = 319.5;
cy = 239.5;
S = 5000;
image1_top_100_rows = image1_top_100_coords(:, 1);
image1_top_100_cols = image1_top_100_coords(:, 2);
image2_top_100_rows = image2_top_100_coords(:, 1);
image2_top_100_cols = image2_top_100_coords(:, 2);
image3_top_100_rows = image3_top_100_coords(:, 1);
image3_top_100_cols = image3_top_100_coords(:, 2);
valid_corners_3d_1 = compute3DCoordinates(depth_map1, [image1_top_100_rows,
image1_top_100_cols], fx, fy, cx, cy, S);
valid_corners_3d_2 = compute3DCoordinates(depth_map2, [image2_top_100_rows,
image2_top_100_cols], fx, fy, cx, cy, S);
valid_corners_3d_3 = compute3DCoordinates(depth_map3, [image3_top_100_rows,
image3_top_100_cols], fx, fy, cx, cy, S);
function valid_corners_3d = compute3DCoordinates(depth_map, top_100_coords,
fx, fy, cx, cy, S)
    num_corners = size(top_100_coords, 1);
    valid_corners_3d = zeros(num_corners, 3);
    valid_count = 0;
    for i = 1:num_corners
        x = top_100_coords(i, 2);
        y = top_100_coords(i, 1);
        depth_value = double(depth_map(y, x)) / S;
        if depth_value > 0
            X = (x - cx) * depth_value / fx;
            Y = (y - cy) * depth_value / fy;
            Z = depth_value;
            valid_count = valid_count + 1;
            valid_corners_3d(valid_count, :) = [X, Y, Z];
        end
    end
    valid_corners_3d = valid_corners_3d(1:valid_count, :);
    disp(['Total valid 3D points: ', num2str(valid_count)]);
```

```
end
```

output:

```
Total valid 3D points: 63
Total valid 3D points: 83
Total valid 3D points: 67
```

Part 3. Corner Matching.

```
[image1_top_100_rows, image1_top_100_cols] = detectCorners('rgb1.png');
[image2_top_100_rows, image2_top_100_cols] = detectCorners('rgb2.png');
[image3_top_100_rows, image3_top_100_cols] = detectCorners('rgb3.png');
image1 = imread('rgb1.png');
image2 = imread('rgb2.png');
image3 = imread('rgb3.png');
gray_image1 = rgb2gray(image1);
gray_image2 = rgb2gray(image2);
gray_image3 = rgb2gray(image3);
rank_transformed_image1 = rank_transform(gray_image1);
rank_transformed_image2 = rank_transform(gray_image2);
rank_transformed_image3 = rank_transform(gray_image3);
distances_1_2 = zeros(100, 100);
distances_2_3 = zeros(100, 100);
window_size = 11;
padded_image1 = padarray(rank_transformed_image1, [floor(window_size/2),
floor(window_size/2)], 0);
padded_image2 = padarray(rank_transformed_image2, [floor(window_size/2),
floor(window_size/2)], 0);
padded_image3 = padarray(rank_transformed_image3, [floor(window_size/2),
floor(window_size/2)], 0);
for i = 1:100
    corner1_row = image1_top_100_rows(i);
    corner1_col = image1_top_100_cols(i);
    window1 = padded_image1(corner1_row:corner1_row+window_size-1,
corner1_col:corner1_col+window_size-1);

    for j = 1:100
        corner2_row = image2_top_100_rows(j);
        corner2_col = image2_top_100_cols(j);
        window2 = padded_image2(corner2_row:corner2_row+window_size-1,
corner2_col:corner2_col+window_size-1);

        sad = sum(abs(window1(:) - window2(:)));
        distances_1_2(i, j) = sad;
    end
end
for i = 1:100
    corner2_row = image2_top_100_rows(i);
    corner2_col = image2_top_100_cols(i);
    window2 = padded_image2(corner2_row:corner2_row+window_size-1,
corner2_col:corner2_col+window_size-1);
```

```
    for j = 1:100
        corner3_row = image3_top_100_rows(j);
        corner3_col = image3_top_100_cols(j);
        window3 = padded_image3(corner3_row:corner3_row+window_size-1,
corner3_col:corner3_col+window_size-1);

        sad = sum(abs(window2(:) - window3(:)));
        distances_2_3(i, j) = sad;
    end
end


matches_1_2 = zeros(10, 3);
matches_2_3 = zeros(10, 3);
for i = 1:10
    index_1 = i;
    index_2 = i;
    matches_1_2(i, 1:2) = [image1_top_100_rows(index_1),
image1_top_100_cols(index_1)];
    matches_1_2(i, 3) = distances_1_2(index_1, index_2);
end
for i = 1:10
    index_2 = i;
    index_3 = i;
    matches_2_3(i, 1:2) = [image2_top_100_rows(index_2),
image2_top_100_cols(index_2)];
    matches_2_3(i, 3) = distances_2_3(index_2, index_3);
end
disp("Top 10 matches for images 1 and 2:");
disp(matches_1_2);

disp("Top 10 matches for images 2 and 3:");
disp(matches_2_3);



function [top_100_rows, top_100_cols] = detectCorners(image_path)
    im = imread(image_path);
    im_gray = rgb2gray(im);
    dx = [-1 0 1; -1 0 1; -1 0 1];
    dy = dx';
    Ix = conv2(double(im_gray), dx, 'same');
    Iy = conv2(double(im_gray), dy, 'same');
    sigma = 1.4;
    kernel_size = 5;
    gaussian_kernel = fspecial('gaussian', [kernel_size, kernel_size],
sigma);
    smoothed_Ix = conv2(Ix, gaussian_kernel, 'same');
    smoothed_Iy = conv2(Iy, gaussian_kernel, 'same');
    Ix2 = smoothed_Ix .^ 2;
    Iy2 = smoothed_Iy .^ 2;
    Ixy = smoothed_Ix .* smoothed_Iy;

    kernel_size = 3;
```

```
    sum_all = ones([kernel_size, kernel_size]);
    Sx2 = conv2(Ix2, sum_all, 'same');
    Sy2 = conv2(Iy2, sum_all, 'same');
    Sxy = conv2(Ixy, sum_all, 'same');

    k = 0.04;

    [numOfRows, numOfColumns] = size(im_gray);
    corners = zeros(numOfRows, numOfColumns);
    for x = 2:numOfRows-1
        for y = 2:numOfColumns-1
            M = [Sx2(x, y), Sxy(x, y); Sxy(x, y), Sy2(x, y)];
            R = det(M) - k * (trace(M) ^ 2);
            corners(x, y) = R;
        end
    end
    suppressed_corners = zeros(size(corners));
    for x = 2:size(corners, 1)-1
        for y = 2:size(corners, 2)-1
            neighborhood = corners(x-1:x+1, y-1:y+1);
            if corners(x, y) == max(neighborhood(:))
                suppressed_corners(x, y) = corners(x, y);
            end
        end
    end
    corners = suppressed_corners;
    corner_responses = corners(:);
    [~, sorted_indices] = sort(corner_responses, 'descend');
    top_100_indices = sorted_indices(1:100);
    [top_100_rows, top_100_cols] = ind2sub(size(corners), top_100_indices);
end
function rank_transformed_image = rank_transform(image)
    window_size = 5;
    padded_image = padarray(image, [floor(window_size/2),
floor(window_size/2)], 0);
    rank_transformed_image = zeros(size(image));
    for i = 1:size(image, 1)
        for j = 1:size(image, 2)
            window = padded_image(i:i+window_size-1, j:j+window_size-1);
            center_pixel_rank = sum(sum(window <=
window(floor(window_size/2)+1, floor(window_size/2)+1)));
            rank_transformed_image(i, j) = center_pixel_rank;
        end
    end
end
```

output: Top 10 matches for images 1 and 2:

```
 122  128  557

 234   29  790

  89  582  841
```

```
 40  587  789

125  122  883

231  106  791

229   32  775

135  577  819

179  134  626

141  368  700
```

Top 10 matches for images 2 and 3:

```
107  245  450

174  262  895

165  244  818

207  186  710

131  253  824

186  245  971

157  248  882

161  246  748

204  167  761

154  269  851
```

Part 4. Pose estimation.

```
num_points_to_select = 10;
P2 = valid_corners_3d_1(1:num_points_to_select, :);
P1 = matches_1_2(1:num_points_to_select, :);
if size(P2, 2) ~= size(P1, 2)
    error('Number of columns in P2 and P1 do not match.');
end
P2_centered = P2 - mean(P2, 1);
P1_centered = P1 - mean(P1, 1);
H = P2_centered' * P1_centered;
```

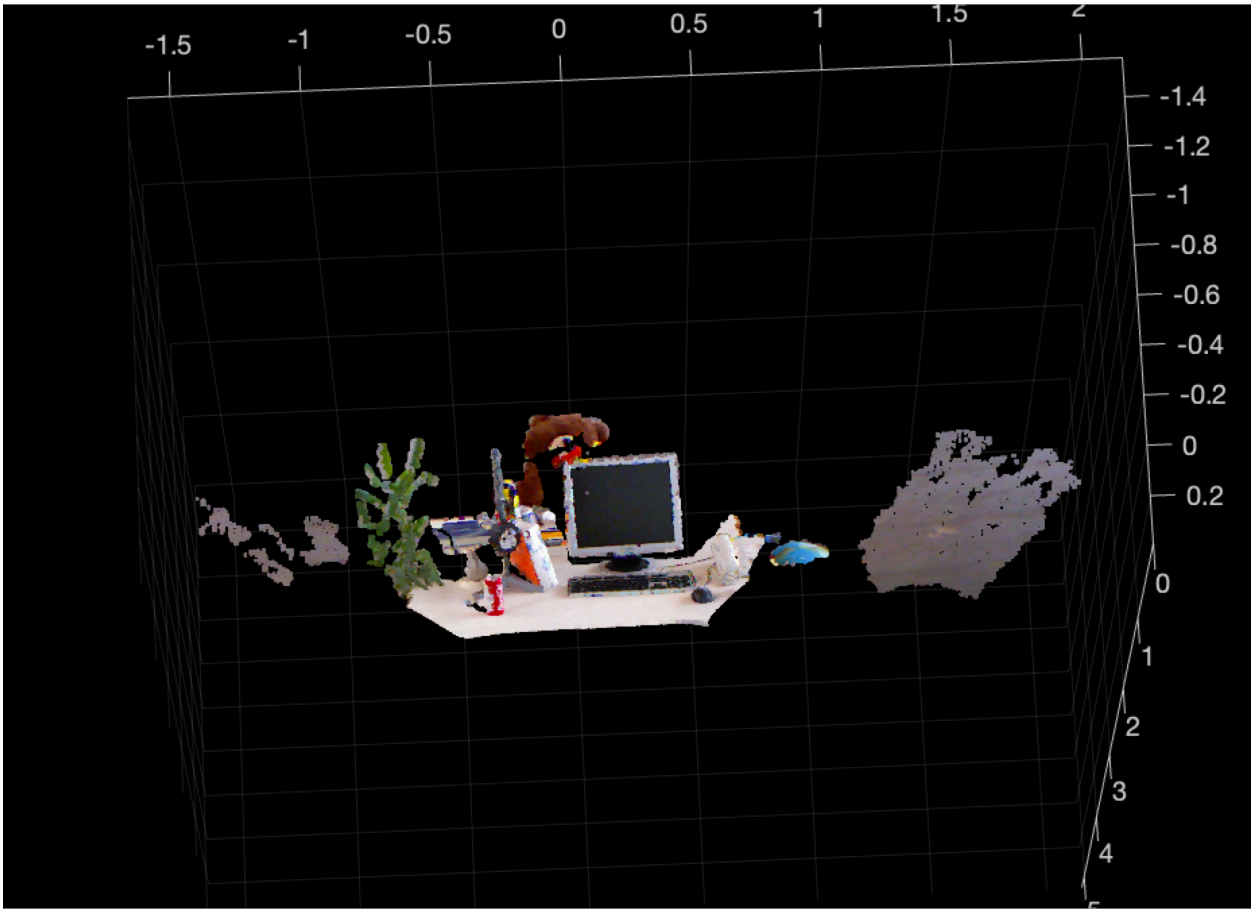
```
[U, ~, Vt] = svd(H);
R = Vt' * U';
if det(R) < 0
    Vt(end, :) = -Vt(end, :);
    R = Vt' * U';
end
t = mean(P2, 1)' - R * mean(P1, 1)';
t = reshape(t, [3, 1]);


Q3 = valid_corners_3d_3(1:num_points_to_select, :);
Q2 = matches_2_3(1:num_points_to_select, :);

if size(Q3, 2) ~= size(Q2, 2)
    error('Number of columns in Q3 and Q2 do not match.');
end
Q3_centered = Q3 - mean(Q3, 1);
Q2_centered = Q2 - mean(Q2, 1);
H_star = Q2_centered' * Q3_centered;
[U_star, ~, Vt_star] = svd(H_star);

R_star = Vt_star' * U_star';
if det(R_star) < 0
    Vt_star(end, :) = -Vt_star(end, :);
    R_star = Vt_star' * U_star';
end
t_star = mean(Q2, 1)' - R_star * mean(Q3, 1)';
t_star = reshape(t_star, [3, 1]);

disp('Relative Pose between Images 2 and 1:');
disp('Rotation matrix:');
disp(R);
disp('Translation vector:');
disp(t);

disp('Relative Pose between Images 2 and 3:');
disp('Rotation matrix:');
disp(R_star);
disp('Translation vector:');
disp(t_star);

output: Relative Pose between Images 2 and 1:
Rotation matrix:
    0.5426   -0.8359   -0.0827
   -0.3815   -0.3329    0.8623
   -0.7484   -0.4363   -0.4996

Translation vector:
  202.6141
 -506.2818
  610.4173

Relative Pose between Images 2 and 3:
Rotation matrix:
    0.2663    0.9437   -0.1962
```

```
   -0.4443   -0.0605   -0.8938
   -0.8554    0.3252    0.4031

Translation vector:
  165.1569
  237.9162
  790.6116
```

Part 5. Finis Coronat Opus.

```
depth1 = imread('depth1.png');
depth2 = imread('depth2.png');
depth3 = imread('depth3.png');
rgb1 = imread('rgb1.png');
rgb2 = imread('rgb2.png');
rgb3 = imread('rgb3.png');
S = 5000;
K = [525.0, 0, 319.5; 0, 525.0, 239.5; 0, 0, 1];
points3D_all_rgb1 = zeros(size(rgb1, 1), size(rgb1, 2), 3);
colors_all_rgb1 = zeros(size(rgb1, 1), size(rgb1, 2), 3);
points3D_all_rgb2 = zeros(size(rgb2, 1), size(rgb2, 2), 3);
colors_all_rgb2 = zeros(size(rgb2, 1), size(rgb2, 2), 3);
points3D_all_rgb3 = zeros(size(rgb3, 1), size(rgb3, 2), 3);
colors_all_rgb3 = zeros(size(rgb3, 1), size(rgb3, 2), 3);

for y = 1:size(rgb1, 1)
    for x = 1:size(rgb1, 2)
        Z1 = double(depth1(y, x)) / S;
        if Z1 ~= 0
            points3D_all_rgb1(y, x, :) = (K \ [x; y; 1]) * Z1;
            colors_all_rgb1(y, x, :) = double(rgb1(y, x, :));
        end
        Z2 = double(depth2(y, x)) / S;
        if Z2 ~= 0
            points3D_all_rgb2(y, x, :) = (K \ [x; y; 1]) * Z2;
            colors_all_rgb2(y, x, :) = double(rgb2(y, x, :));
        end
        Z3 = double(depth3(y, x)) / S;
        if Z3 ~= 0
            points3D_all_rgb3(y, x, :) = (K \ [x; y; 1]) * Z3;
            colors_all_rgb3(y, x, :) = double(rgb3(y, x, :));
        end
    end
end
transformed_points_all_rgb1 = zeros(size(points3D_all_rgb1));

for y = 1:size(rgb1, 1)
    for x = 1:size(rgb1, 2)
        if depth1(y, x) ~= 0
```

```
            temp_result = R * squeeze(points3D_all_rgb1(y, x, :));
            temp_result = temp_result' + t';
            temp_result = temp_result / S;
            transformed_points_all_rgb1(y, x, :) = temp_result';
        end
    end
end
transformed_points_all_rgb3 = zeros(size(rgb3, 1), size(rgb3, 2), 3);
for y = 1:size(rgb3, 1)
    for x = 1:size(rgb3, 2)
        if depth3(y, x) ~= 0
            temp_result = R_star * squeeze(points3D_all_rgb3(y, x, :));
            temp_result = temp_result' + t_star';
            temp_result = temp_result / S;
            transformed_points_all_rgb3(y, x, :) = temp_result';
        end
    end
end


merged_points = [reshape(transformed_points_all_rgb1, [], 3);
reshape(transformed_points_all_rgb3, [], 3); reshape(points3D_all_rgb2, [],
3)];
merged_colors = [reshape(colors_all_rgb1, [], 3); reshape(colors_all_rgb3,
[], 3); reshape(colors_all_rgb2, [], 3)];
writePLY('output.ply', merged_points, merged_colors);
surf_mod = pcread('output.ply');
pcshow(surf_mod);
pcwrite(surf_mod,"output.ply");

function writePLY(filename, points, colors)
    fid = fopen(filename, 'w');
    fprintf(fid, 'ply\n');
    fprintf(fid, 'format ascii 1.0\n');
    fprintf(fid, 'element vertex %d\n', size(points, 1));
    fprintf(fid, 'property float x\n');
    fprintf(fid, 'property float y\n');
    fprintf(fid, 'property float z\n');
    fprintf(fid, 'property uchar red\n');
    fprintf(fid, 'property uchar green\n');
    fprintf(fid, 'property uchar blue\n');
    fprintf(fid, 'end_header\n');


    for i = 1:size(points, 1)
        fprintf(fid, '%f %f %f %d %d %d\n', points(i, 1), points(i, 2),
points(i, 3), ...
                                            uint8(colors(i, 1)),
uint8(colors(i, 2)), uint8(colors(i, 3)));
    end
    fclose(fid);
end
```



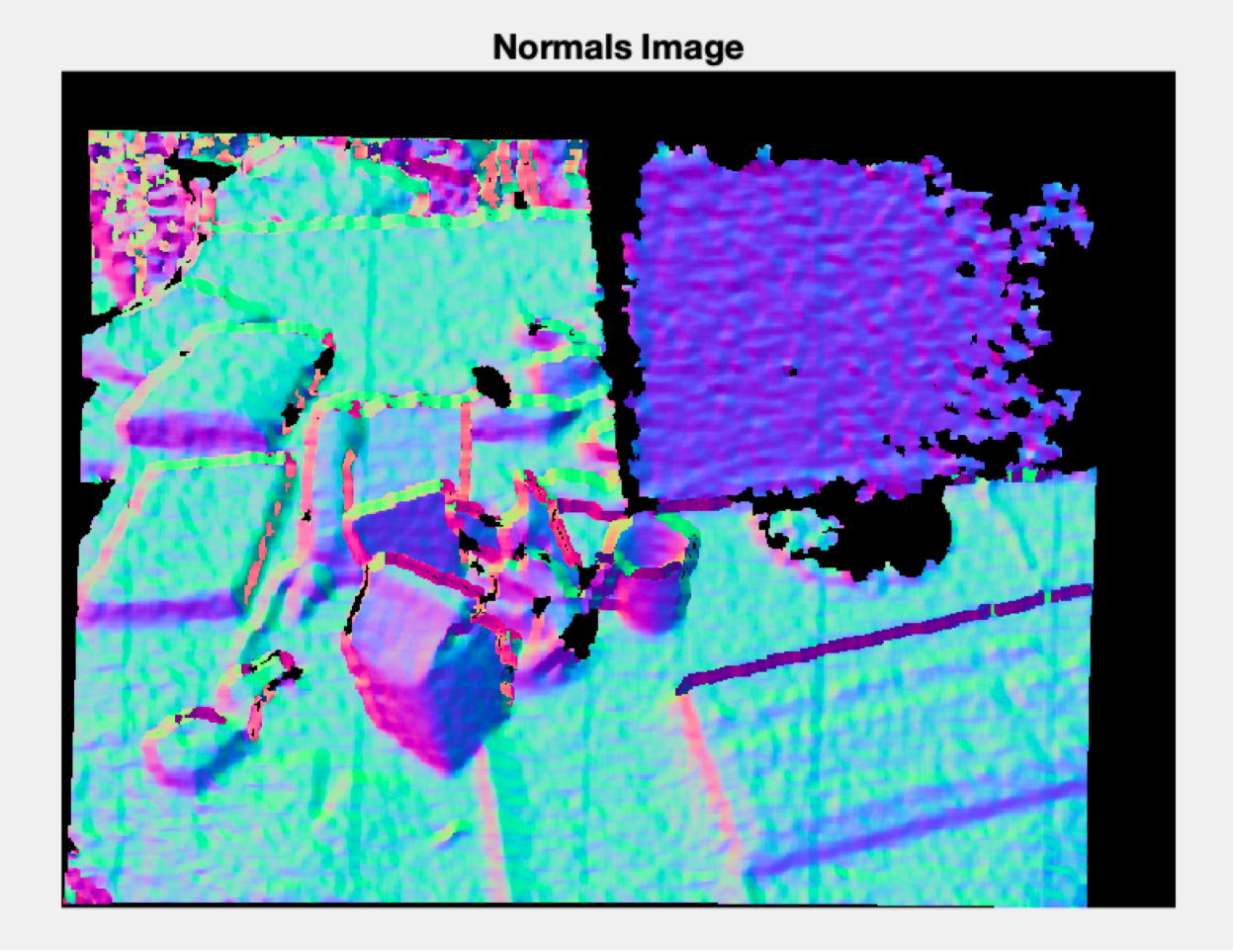
Problem 2. Normal Mapping.

Method 1:

```
% Load the depth image and intrinsic camera parameters
depth = imread('depthn.png');  % Load depth image
rgb = imread('rgbn.png');      % Load RGB image
S = 5000;                      % Scale factor
K = [525.0, 0, 319.5; 0, 525.0, 239.5; 0, 0, 1];  % Intrinsic matrix

% Compute 3D points from depth image
[rows, cols] = size(depth);
points3D = zeros(rows, cols, 3);

for y = 1:rows
    for x = 1:cols
        Z = double(depth(y, x)) / S;
        if Z ~= 0
            points3D(y, x, :) = (K \ [x; y; 1]) * Z;
```

```
        else
            points3D(y, x, :) = [NaN, NaN, NaN];
        end
    end
end

% Compute normals for each pixel using a 7x7 neighborhood
neighborhood_size = 7;
half_size = (neighborhood_size - 1) / 2;
normals = zeros(rows, cols, 3);

for y = 1:rows
    for x = 1:cols
        if ~isnan(points3D(y, x, 1))
            % Collect points in the neighborhood
            x_start = max(1, x - half_size);
            x_end = min(cols, x + half_size);
            y_start = max(1, y - half_size);
            y_end = min(rows, y + half_size);

            neighborhood = reshape(points3D(y_start:y_end, x_start:x_end, :),
[], 3);
            neighborhood = neighborhood(all(~isnan(neighborhood), 2), :);

            if size(neighborhood, 1) >= 3
                % Fit a plane to the neighborhood
                centroid = mean(neighborhood, 1);
                covariance_matrix = cov(neighborhood);
                [V, D] = eig(covariance_matrix);

                % The smallest eigenvalue corresponds to the normal
                [~, min_index] = min(diag(D));
                normal = V(:, min_index);

                % Ensure the normal points toward the camera
                if normal(3) < 0
                    normal = -normal;
                end

                normals(y, x, :) = normal;
            else
                normals(y, x, :) = [NaN, NaN, NaN];
            end
        else
            normals(y, x, :) = [NaN, NaN, NaN];
        end
    end
end


% Create an RGB image to represent the plane normals
normal_image = zeros(rows, cols, 3, 'uint8');

for y = 1:rows
    for x = 1:cols
```

```
        if ~isnan(normals(y, x, 1))
            % Normalize the normal
            n = squeeze(normals(y, x, :));
            n = n / norm(n);

            % Map to RGB values
            R = uint8(255 * (0.5 * (n(1) + 1)));
            G = uint8(255 * (0.5 * (n(2) + 1)));
            B = uint8(255 * (0.5 * (n(3) + 1)));

            normal_image(y, x, :) = [R, G, B];
        else
            normal_image(y, x, :) = [0, 0, 0];
        end
    end
end

% Display the RGB image and the normals
figure;
subplot(1, 2, 1);
imshow(rgb);  % Display the RGB image
title('RGB Image');

subplot(1, 2, 2);
imshow(normal_image);  % Display the normal image
title('Normals Image');
```



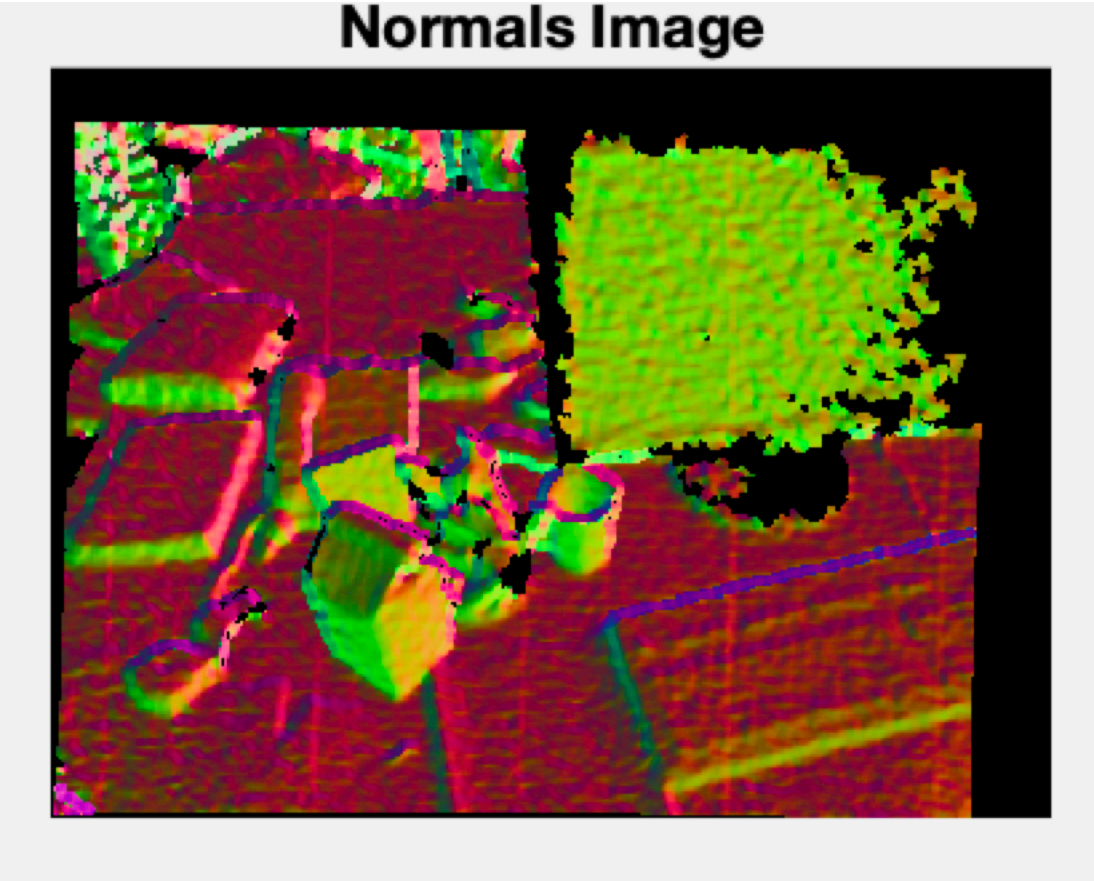
Method 2:

considering $[a, b, c, -d]$, then set the normal to $n = [-a, -b, -c]$. Code is:

```
% Load the depth image and intrinsic camera parameters
depth = imread('depthn.png');  % Load depth image
rgb = imread('rgbn.png');      % Load RGB image
S = 5000;                      % Scale factor
K = [525.0, 0, 319.5; 0, 525.0, 239.5; 0, 0, 1];  % Intrinsic matrix

% Compute 3D points from depth image
[rows, cols] = size(depth);
points3D = zeros(rows, cols, 3);

for y = 1:rows
    for x = 1:cols
        Z = double(depth(y, x)) / S;
        if Z ~= 0
            points3D(y, x, :) = (K \ [x; y; 1]) * Z;
        else
            points3D(y, x, :) = [NaN, NaN, NaN];
        end
    end
```

```
end

% Compute normals for each pixel using a 7x7 neighborhood
neighborhood_size = 7;
half_size = (neighborhood_size - 1) / 2;
normals = zeros(rows, cols, 3);

for y = 1:rows
    for x = 1:cols
        if ~isnan(points3D(y, x, 1)) % Process only valid points
            % Collect points in the neighborhood
            x_start = max(1, x - half_size);
            x_end = min(cols, x + half_size);
            y_start = max(1, y - half_size);
            y_end = min(rows, y + half_size);

            neighborhood = reshape(points3D(y_start:y_end, x_start:x_end, :),
[], 3);
            neighborhood = neighborhood(all(~isnan(neighborhood), 2), :);

            if size(neighborhood, 1) >= 3 % Ensure sufficient non-collinear
points
                % Fit a plane to the neighborhood
                centroid = mean(neighborhood, 1); % Get centroid
                covariance_matrix = cov(neighborhood); % Calculate covariance
                [V, D] = eig(covariance_matrix); % Get eigenvectors and
eigenvalues

                % The eigenvector with the smallest eigenvalue represents the
normal
                [~, min_index] = min(diag(D));
                normal = V(:, min_index);

                % Compute the plane equation to check for orientation
                d = -dot(centroid, normal);

                % If the plane's distance from the origin is negative, flip
the normal
                if d < 0
                    normal = -normal;
                end

                normals(y, x, :) = normal; % Store the corrected normal
            else
                normals(y, x, :) = [NaN, NaN, NaN]; % Insufficient data for
plane fitting
            end
        else
            normals(y, x, :) = [NaN, NaN, NaN]; % Invalid depth, set NaN
        end
    end
end


% Map normals to RGB image
```

```
normal_image = zeros(rows, cols, 3, 'uint8');

for y = 1:rows
    for x = 1:cols
        if ~isnan(normals(y, x, 1))
            % Normalize the normal
            n = squeeze(normals(y, x, :));
            n = n / norm(n);

            % Map to RGB values using the given formula
            R = uint8(255 * (0.5 * (n(1) + 1))); % R = 255 * (0.5 * (a + 1))
            G = uint8(255 * (0.5 * (n(2) + 1))); % G = 255 * (0.5 * (b + 1))
            B = uint8(255 * (0.5 * (n(3) + 1))); % B = 255 * (0.5 * (c + 1))

            normal_image(y, x, :) = [R, G, B]; % Assign RGB representation
        else
            % Assign [0, 0, 0] for invalid or zero-depth points
            normal_image(y, x, :) = [0, 0, 0];
        end
    end
end


% Display the RGB image and the normals
figure;
subplot(1, 2, 1);
imshow(rgb);  % Display the RGB image
title('RGB Image');

subplot(1, 2, 2);
imshow(normal_image);  % Display the normal image
title('Normals Image');
```


Normals Image