The Report of B31MV-Assignment3

Wenbin Lin

Section 1: Sparse Reconstruction

Part 1: Implement the Eight-Point Algorithm

Report on the estimated fundamental matrix F

For this issue, the eight-point algorithm is applied to calculate the fundamental matrix F. The algorithm can be divided into five steps.

Step1: Normalize the coordinates. To make the algorithm numerically stable, we normalize the point coordinates by scaling them down using a transformation matrix T, typically with `1/M`, where `M` is the max image dimension. The maximum image dimension of the two images is chosen as the M parameter. The normalization process is really simple, just rescaling points to a small numerical range `[0,1]`. Choosing the algorithm, just because of the requirements of the issue, we should use the scale parameter M. If not needed for this, Hartley Normalization is more commonly used, for it is more stable.

Step 2: Construct matrix A. We form a matrix `A` based on the epipolar constraint: $x_2^T \cdot F \cdot x_1 = 0$. Each row of `A` corresponds to one pair of point correspondences.

Step 3: Solve Af = 0 using SVD. We solve the homogeneous system using Singular Value Decomposition. The fundamental matrix F is found from the last column of `V` (or last row of `V.T`), reshaped into a 3x3 matrix.

Step 4: Enforce rank-2 constraint. The fundamental matrix must be rank-2 (not full rank). We apply SVD on F, set the smallest singular value to zero, and reconstruct F.

Step 5: Unnormalize the fundamental matrix. We reverse the normalization transformation to get the final matrix F. For the final F, F_final / F_final[2,2] is performed to fit F[2,2] to 1 and make the entries of F in a more reasonable numeric range.

By default, the eight-point algorithm is not robust to outliers. It assumes that all point correspondences are perfect matches.Even one or two bad matches can significantly distort the result of F. The fundamental matrix F and the code is displayed as follows:

```
Scale parameter M = 640
Estimated Fundamental Matrix F:
 [[-8.55629053e-07  3.14052683e-05 -7.31606892e-03]
 [ 1.42140129e-05  3.14691357e-07  2.59284111e-01]
 [ 3.96151277e-03 -2.69985755e-01  1.00000000e+00]]
```

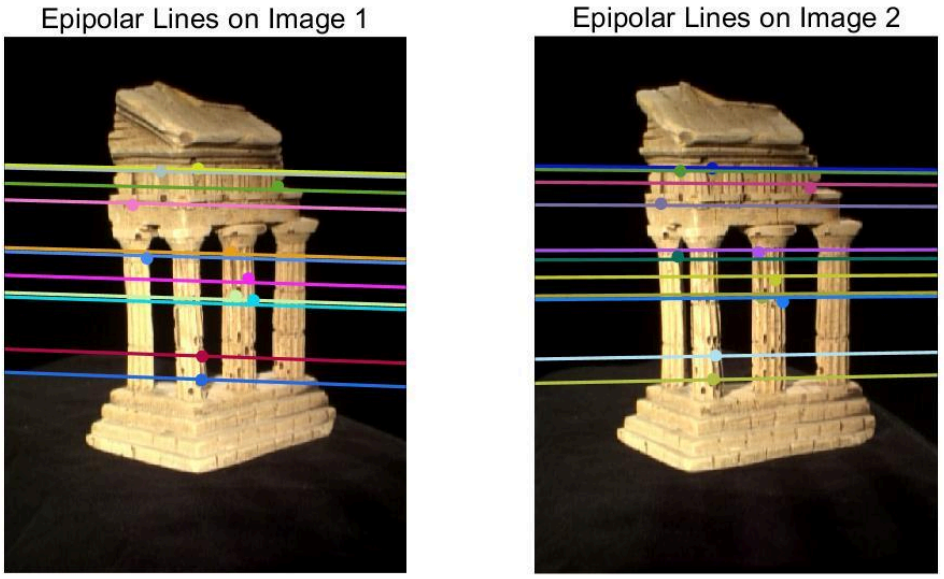
```
% Load the correspondences
data = load('data/some_corresp.mat');
pts1 = double(data.pts1);
pts2 = double(data.pts2);
% Read images to compute M
im1 = imread('data/im1.png');
im2 = imread('data/im2.png');
M = max([size(im1,1), size(im1,2), size(im2,1), size(im2,2)]); % M = max image dimension
```

```
% Compute Fundamental Matrix
F = opencv_demo1(pts1, pts2, M);
format shortE;
disp('Estimated Fundamental Matrix F:');
disp(F);
function F = eight_point(pts1, pts2, M)
% Compute the Fundamental Matrix using the normalized eight-point algorithm
% Inputs:
%   pts1: N x 2 matrix of points in image 1
%   pts2: N x 2 matrix of points in image 2
%   M: scale parameter for normalization
% Output:
%   F: Fundamental Matrix (3x3)
   T = [1/M, 0, 0;
        0, 1/M, 0;
        0, 0,   1];
   pts1_h = [pts1, ones(size(pts1,1),1)]';
   pts2_h = [pts2, ones(size(pts2,1),1)]';
   pts1_norm = T * pts1_h;
   pts2_norm = T * pts2_h;
   N = size(pts1,1);
   A = zeros(N,9);
   for i=1:N
      x1 = pts1_norm(1,i);
      y1 = pts1_norm(2,i);
      x2 = pts2_norm(1,i);
      y2 = pts2_norm(2,i);
      A(i,:) = [x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1];
   end
   [~, ~, V] = svd(A);
   F_norm = reshape(V(:,end), 3, 3)';
   [U, S, V] = svd(F_norm);
   S(3,3) = 0;
   F_norm = U * S * V';
   F = T' * F_norm * T;
   F = F / F(3,3);
end
```

Include a visualization of some epipolar lines on the images using the computed F

Given a pair of corresponding points $\mathbf{x}_1 \in \text{Image } 1$ and $\mathbf{x}_2 \in \text{Image } 2$, the epipolar constraint is $\mathbf{x}_2^\top \cdot \mathbf{F} \cdot \mathbf{x}_1 = 0$. From this, the epipolar line in image 2 is $\mathbf{l}_2 = F \cdot \mathbf{x}_1$, and the epipolar line in image 1 is: $\mathbf{l}_1 = F^\top \cdot \mathbf{x}_2$.Each line $\mathbf{l} = [a, b, c]^T$ describes the line equation

$ax + by + c = 0$. After calculating the epipolar line, we can draw the correspondence line in the images. The following is the picture of ten epipolar lines of the two images and the code.




```
%% --- Epipolar Lines Visualization ---
figure;
N = size(pts1, 1);
% --- Image 1: Epipolar lines from pts2 to im1 ---
subplot(1,2,1);
imshow(im1); hold on;
title('Epipolar Lines on Image 1');
for i = 1:10:N
    % Compute epipolar line in image 1 from pts2
    l = F' * [pts2(i,:), 1]';
    x = 1:size(im1,2);
    y = -(l(1)*x + l(3)) / l(2);
    % Generate random RGB color
    c = rand(1,3);
    % Plot line and corresponding point in same color
    plot(x, y, 'Color', c, 'LineWidth', 1.2);
    plot(pts1(i,1), pts1(i,2), '.', 'Color', c, 'MarkerSize', 15);
end
% --- Image 2: Epipolar lines from pts1 to im2 ---
subplot(1,2,2);
```

```
imshow(im2); hold on;
title('Epipolar Lines on Image 2');
for i = 1:10:N
    % Compute epipolar line in image 2 from pts1
    l = F * [pts1(i,:), 1]';
    x = 1:size(im2,2);
    y = -(l(1)*x + l(3)) / l(2);
    % Generate random RGB color
    c = rand(1,3);
    % Plot line and corresponding point in same color
    plot(x, y, 'Color', c, 'LineWidth', 1.2);
    plot(pts2(i,1), pts2(i,2), '.', 'Color', c, 'MarkerSize', 15);
end
```

Part 2: Find Epipolar Correspondences

Describe the similarity metric used for matching points

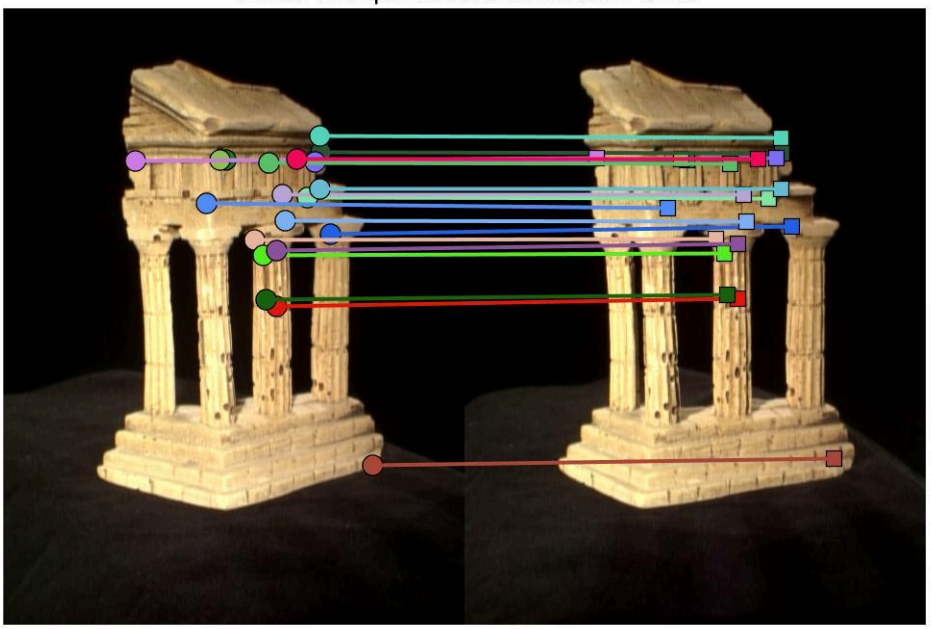
The similarity metric is known as the sum of squared differences (SSD), which is a classical patch-based similarity measure used to compare two image regions (patches) centered around two points. It is computed as the sum of squared intensity differences between corresponding pixels in two patches.

$$\mathrm{SSD}=\sum_{i=-w}^{w}\sum_{j=-w}^{w}\left[I_1(x_1+i,y_1+j)-I_2(x_2+i,y_2+j)\right]^2$$

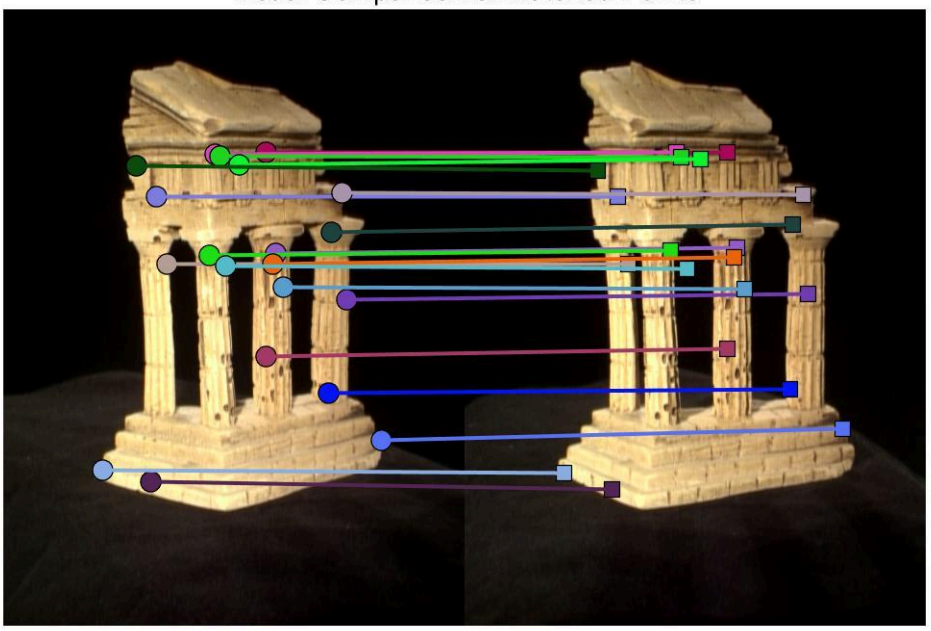
You take a small patch centered at (x_1,y_1) in image 1. Then you slide this patch along the epipolar line in image 2 and extract patches centered at candidate points (x_2,y_2).For each candidate patch, you compute the SSD. The point with the smallest SSD is considered the best correspondence.

We apply the SSD algorithm and the two images and the pts1 and the fundamental matrix to calculate the correspondence point of pts2 through the epipolar line. After that we draw the pts1 and pts2 points in the image1 and image2 and match them together. The picture (each picture only shows 20 points for its clarity)and the code is displayed in the following.

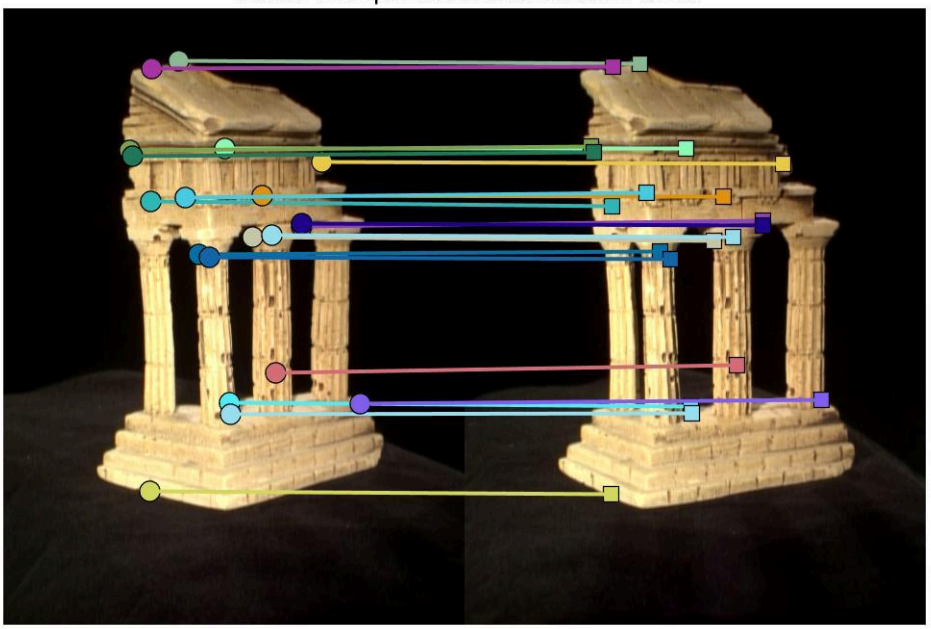
Visual Comparison of Matched Points



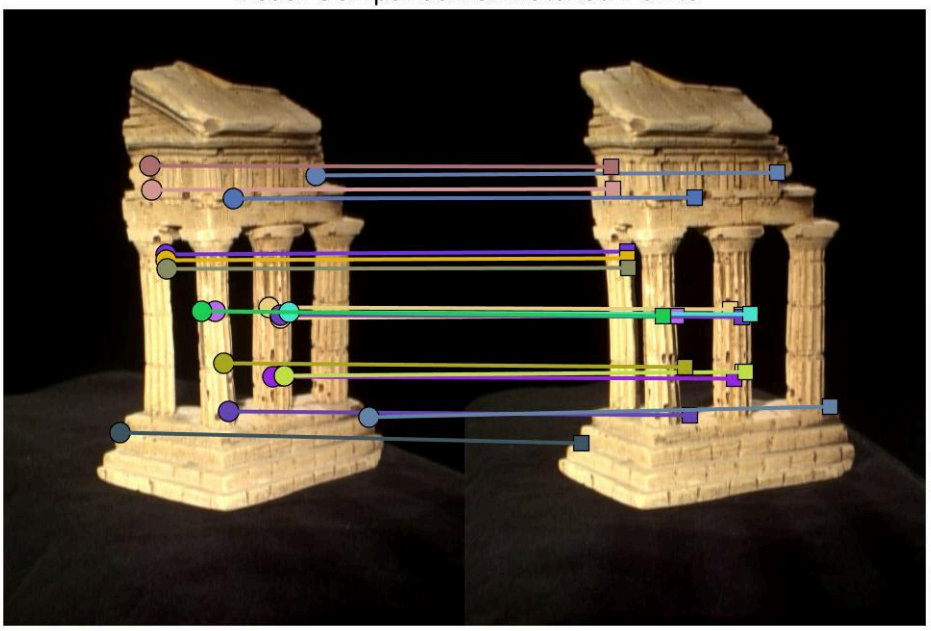
Visual Comparison of Matched Points



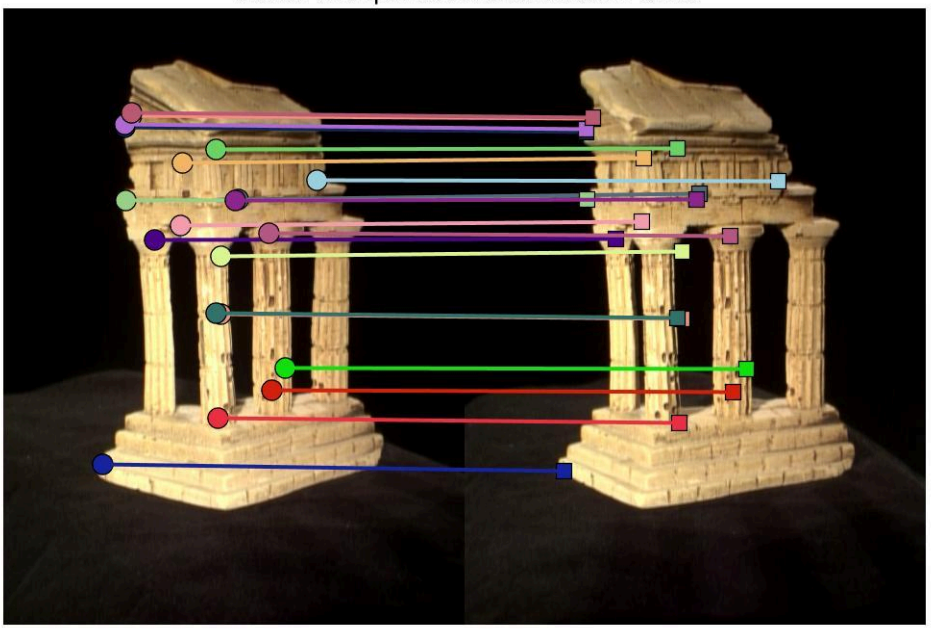
Visual Comparison of Matched Points



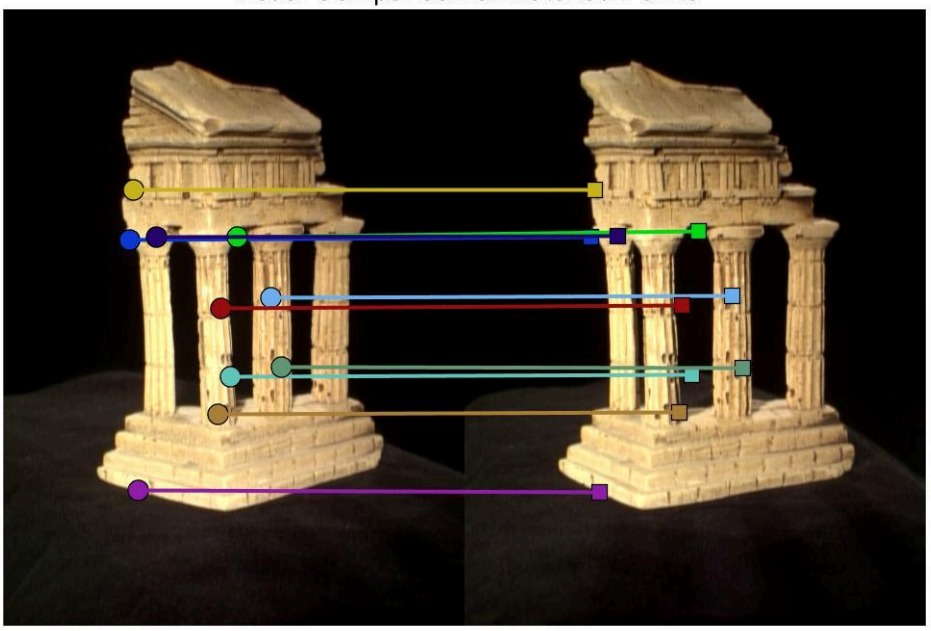
Visual Comparison of Matched Points



Visual Comparison of Matched Points



Visual Comparison of Matched Points



```
epi_pts2 = epipolar_correspondence(im1, im2, F, pts1);
visualize_point_matches('data/im1.png', 'data/im2.png', pts1, epi_pts2);
function visualize_point_matches(im1_path, im2_path, pts1, epi_pts2)
```

```
    % Load images
    im1 = imread(im1_path);
    im2 = imread(im2_path);
    % Resize to same height if needed (optional)
    if size(im1,1) ~= size(im2,1)
        H = max(size(im1,1), size(im2,1));
        im1 = imresize(im1, [H NaN]);
        im2 = imresize(im2, [H NaN]);
    end
    % Concatenate images horizontally
    combined = [im1, im2];
    % Offset for image 2 (x direction)
    offset = size(im1,2);
    N = size(pts1, 1);
    % Pick only 20 points to display
    idx = round(linspace(1, size(pts1,1), 20));
    batch_size = 20;
    for start_idx = 1:batch_size:N
        end_idx = min(start_idx + batch_size - 1, N);
        batch_idx = start_idx:end_idx;
          % Display
        figure; imshow(combined); hold on;
        title(sprintf('Visual Comparison of Matched Points %d', idx));
        for i = 1:length(batch_idx)
            p1 = pts1(batch_idx(i), :);
            %p1 = pts1(idx(i), :);
            %p2 = epi_pts2(idx(i), :);
            p2 = epi_pts2(batch_idx(i), :);
            % Random color
            c = rand(1,3);
            % Plot point in image 1
            plot(p1(1), p1(2), 'o', 'MarkerSize', 8, 'MarkerFaceColor', c, 'MarkerEdgeColor', 'k');
            % Plot point in image 2 (shifted x)
            plot(p2(1)+offset, p2(2), 's', 'MarkerSize', 8, 'MarkerFaceColor', c,
'MarkerEdgeColor', 'k');
            % Draw line between points
            line([p1(1), p2(1)+offset], [p1(2), p2(2)], 'Color', c, 'LineWidth', 1.5);
        end
    end
end
epi_pts2 = epipolar_correspondence(im1, im2, F, pts1);
visualize_point_matches('data/im1.png', 'data/im2.png', pts1, epi_pts2);
function visualize_point_matches(im1_path, im2_path, pts1, epi_pts2)
    % Load images
    im1 = imread(im1_path);
    im2 = imread(im2_path);
    % Resize to same height if needed (optional)
```

```
    if size(im1,1) ~= size(im2,1)
        H = max(size(im1,1), size(im2,1));
        im1 = imresize(im1, [H NaN]);
        im2 = imresize(im2, [H NaN]);
    end
    % Concatenate images horizontally
    combined = [im1, im2];
    % Offset for image 2 (x direction)
    offset = size(im1,2);
    N = size(pts1, 1);
    % Pick only 20 points to display
    idx = round(linspace(1, size(pts1,1), 20));
    batch_size = 20;
    for start_idx = 1:batch_size:N
        end_idx = min(start_idx + batch_size - 1, N);
        batch_idx = start_idx:end_idx;
          % Display
        figure; imshow(combined); hold on;
        title(sprintf('Visual Comparison of Matched Points %d', idx));
        for i = 1:length(batch_idx)
            p1 = pts1(batch_idx(i), :);
            %p1 = pts1(idx(i), :);
            %p2 = epi_pts2(idx(i), :);
            p2 = epi_pts2(batch_idx(i), :);
            % Random color
            c = rand(1,3);
            % Plot point in image 1
            plot(p1(1), p1(2), 'o', 'MarkerSize', 8, 'MarkerFaceColor', c, 'MarkerEdgeColor', 'k');
            % Plot point in image 2 (shifted x)
            plot(p2(1)+offset, p2(2), 's', 'MarkerSize', 8, 'MarkerFaceColor', c,
'MarkerEdgeColor', 'k');
            % Draw line between points
            line([p1(1), p2(1)+offset], [p1(2), p2(2)], 'Color', c, 'LineWidth', 1.5);
        end
    end
end
```

Discuss any cases where the matching algorithm consistently fails, including possible reasons

For this issue, in my case, most of the pts2 calculated the correspondence point of pts2 through the epipolar line match the img2 well, but several fail. The reasons are as follows.

- Repetitive patterns

I think this is the most important reason for the mismatching. The building has many repetitive patterns. The SSD metric cannot distinguish between repeated patterns.

- Inaccurate fundamental matrix (F)

Due to outliers or noisy correspondences, F maybe is not accurate, the epipolar lines may deviate from the real ones. And it can not find the correspondence point of pts without a big searching window. For a big searching window, we should use big window size and search range, and this may cause other point mismatches.

- Illumination Changes Between Images

Lighting or exposure changes between the two images can significantly affect pixel intensities, which breaks the assumption behind SSD. SSD compares raw intensity values. If the same patch appears darker/brighter due to lighting, the algorithm will wrongly consider it a bad match.

- Perspective or Scale Distortions

When the camera views the scene from significantly different angles, even corresponding points may appear distorted. SSD assumes that patches are **identically shaped**, which isn't true under affine transformations.

Part 3: Compute the Essential Matrix

Present the computed essential matrix E for the temple image pair

For this issue, the aim is to Compute the Essential Matrix from F, K1, and K2, and the Essential Matrix E can be computed by the following equation.

$$E = K_2^\top F K_1$$

After calculation, the essential matrix E and the code is displayed in the following.

```
Computed Essential Matrix (E):
 [[-1.97788595e+00  7.28595429e+01  2.71050678e-01]
 [ 3.29762023e+01  7.32718170e-01  4.02317237e+02]
 [ 1.09649003e+01 -3.97365153e+02  6.89649744e-01]]
```

```
%% --- Compute Essential Matrix ---
E = compute_essential_matrix(F, K1, K2)
disp('Estimated Essential Matrix E:');
disp(E);
function E = compute_essential_matrix(F, K1, K2)
% Compute Essential Matrix from Fundamental Matrix and Intrinsics
   E = K2' * F * K1;
end
```

Part 4: Implement Triangulation

Develop a function named triangulate that computes the 3D points from pairs of 2D points in the two images. The function should accept four arguments P1, pts1, P2 and pts2. Triangulation is the process of recovering the 3D position of a point from its 2D projections in two images, which calculates 3D points from P1,P2,pts1,pts2.

In homogeneous coordinates:

$$x_1 = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}, \quad x_2 = \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

The projection equations are:

$$x_1 = P_1 X_h \quad \text{and} \quad x_2 = P_2 X_h$$

where $X_h = [X, Y, Z, 1]^T$ is the homogeneous representation of the 3D point.
Each projection gives 2 linear equations (for x and y):
From P_1:

$$x_1 P_{1,3}^T - P_{1,1}^T = 0$$

$$y_1 P_{1,3}^T - P_{1,2}^T = 0$$

From P_2:

$$x_2 P_{2,3}^T - P_{2,1}^T = 0$$

$$y_2 P_{2,3}^T - P_{2,2}^T = 0$$

Then we can build matrix A:

$$A = \begin{bmatrix} x_1 P_{1,3}^T - P_{1,1}^T \\ y_1 P_{1,3}^T - P_{1,2}^T \\ x_2 P_{2,3}^T - P_{2,1}^T \\ y_2 P_{2,3}^T - P_{2,2}^T \end{bmatrix}$$

This is a homogeneous linear system: $AX_h = 0$
Since A is rank-deficient, its solution is the right singular vector corresponding to the smallest singular value of A.Thus, the 3D point is $X = [X_h[0], X_h[1], X_h[2]]$. The code for triangulation through P1,P2,pts1,pts2 is in the following.

```
function [pts3d, reprojection_error] = triangulate_check(P1, pts1, P2, pts2)
% Triangulate 3D points and compute mean reprojection error
% Inputs:
%   P1, P2: 3x4 projection matrices
%   pts1, pts2: Nx2 matching points
% Outputs:
%   pts3d: Nx3 triangulated 3D points
%   reprojection_error: mean reprojection error across both views
    N = size(pts1, 1);
    pts3d = zeros(N, 3);
    err1 = zeros(N, 1);
    err2 = zeros(N, 1);
    for i = 1:N
        A = [pts1(i,1)*P1(3,:) - P1(1,:);
             pts1(i,2)*P1(3,:) - P1(2,:);
             pts2(i,1)*P2(3,:) - P2(1,:);
             pts2(i,2)*P2(3,:) - P2(2,:)];
        [~, ~, V] = svd(A);
        X = V(:, end);
        X = X ./ X(4);
```

```
        pts3d(i, :) = X(1:3)';
        % Reproject
        x1_proj = P1 * X;
        x1_proj = x1_proj(1:2) / x1_proj(3);
        x2_proj = P2 * X;
        x2_proj = x2_proj(1:2) / x2_proj(3);
        % Compute per-point reprojection errors
        err1(i) = norm(x1_proj' - pts1(i,:));
        err2(i) = norm(x2_proj' - pts2(i,:));
    end
    % Average reprojection error over all points
    reprojection_error = mean(err1 + err2);
end
```

Explain the method used to determine the correct extrinsic matrix

For this issue, given us the essential matrix E and intrinsic matrix of two cameras K1, K2, calculate the extrinsic matrix R and t. If we know the extrinsic matrix R and t, we can calculate the projection matrix P.

The first camera is always assumed to be the reference camera placed at the origin of the world coordinate system. Thus, its projection matrix is:

$$P_1 = K_1 \cdot [I|0]$$

- I is the 3×3 identity matrix (no rotation)
- 0 is a 3×1 zero vector (no translation)
- K_1 is the intrinsic matrix of the first camera

P_1 maps 3D points in world coordinates directly into the first image. And the essential matrix E encodes the relative rotation R and translation t between the two cameras. Since $E = [t]_\times R$,we can decompose E by performing Singular Value Decomposition (SVD): $E = USV^T$.

Then define the matrix W for rotation recovery,This special matrix is used to extract R.

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then we can generate the four candidate extrinsics. The decomposition gives 4 possible solutions for [R|t]:

Solution	Rotation	Translation
case1	$R_1 = UWV^T$	$t = +U[:, 2]$
case2	$R_1 = UWV^T$	$t = -U[:, 2]$
case3	$R_2 = UW^TV^T$	$t = +U[:, 2]$
case4	$R_2 = UW^TV^T$	$t = -U[:, 2]$

from the R, t, we can calculate the P_2 matrix, and select the best one by the above triangulate_check() function by the lowest err.

Report the re-projection error, which should be less than 2 pixels if implemented correctly

From the previous section, we can calculate the P_1,P_2 by the essential matrix E and the intrinsic matrix of two camera K_1 and K_2, then use the P_1,P_2 ,pts1,pts2 to calculate the correspondence points pts3d of pts1 and pts2, and then calculate the re-projection points of pts3d, using the following equations:

$$proj1 = P1 \cdot pts3d$$
$$proj2 = P2 \cdot pts3d$$

Then we use the sum of squared differences for the total re-projection error, for the mean error, the total error is divided by the 2*(number of pts1). After calculation, the mean re-projection error is 3.79 pixels. The reprojection error is larger than 2 pixels, the reason may be the F is inaccurate, is an approximation, for there are some outliers in the pts1 and pts2 pair. The code for calculating the P_1, P_2 and the mean re-projection error is in the following.

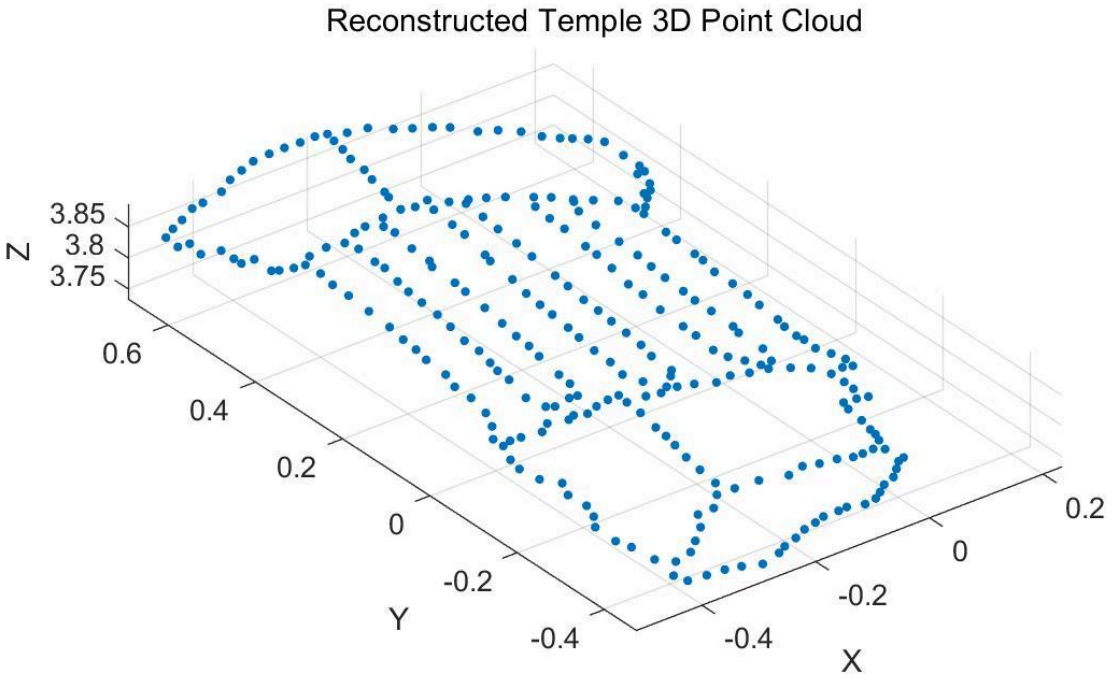
```
% --------------------------
% Implement Triangulation and Implement Triangulation
% --------------------------
clear; clc;
%% --- Load Data ---
data = load('data/some_corresp.mat');
pts1 = double(data.pts1);
pts2 = double(data.pts2);
intrinsics = load('data/intrinsics.mat');
K1 = intrinsics.K1;
K2 = intrinsics.K2;
im1 = imread('data/im1.png');
im2 = imread('data/im2.png');
M = max([size(im1,1), size(im1,2), size(im2,1), size(im2,2)]);
%% --- Compute Fundamental Matrix ---
F = eight_point(pts1, pts2, M);
%% --- Compute Essential Matrix ---
E = compute_essential_matrix(F, K1, K2)
%% --- compute R1,t1,R2,t2,P1,P2 ---
[P1, P2, R1, t1, R2, t2] = compute_camera_matrices_from_E(E, K1, K2, pts1, pts2)
[pts3d, reprojection_error] = triangulate_check(P1, pts1, P2, pts2)
disp(['Mean Reprojection Error: ', num2str(reprojection_error)]);

function [P1, P2, R, t, R2, t2] = compute_camera_matrices_from_E(E, K1, K2, pts1, pts2)
% Computes P1, P2, R, t from Essential matrix and intrinsics
    % Step 1: Decompose E into R, t
    [U, ~, V] = svd(E);
    if det(U) < 0, U = -U; end
```

```
    if det(V) < 0, V = -V; end
    W = [0 -1 0; 1 0 0; 0 0 1];
    R1 = U * W * V';
    R2 = U * W' * V';
    t1 =  U(:,3);
    t2 = -U(:,3);
    % Step 2: Construct P1 and candidates for P2
    P1 = K1 * [eye(3), zeros(3,1)];
    % Four possible configurations of P2
    P2_options = cat(3, ...
        K2 * [R1,  t1], ...
        K2 * [R1,  t2], ...
        K2 * [R2,  t1], ...
        K2 * [R2,  t2]);
    % Step 3: Pick correct P2 by cheirality check
    best_count = 0;
    for i = 1:4
        P2_test = P2_options(:,:,i);
        pts3d = triangulate_points(P1, pts1, P2_test, pts2);
        % Check how many points are in front of both cameras
        X_h = [pts3d, ones(size(pts3d,1),1)]';
        in_front1 = (P1(3,:) * X_h) > 0;
        in_front2 = (P2_test(3,:) * X_h) > 0;
        count = sum(in_front1 & in_front2);
        if count > best_count
            best_count = count;
            P2 = P2_test;
            if i == 1, R = R1; t = t1;
            elseif i == 2, R = R1; t = t2;
            elseif i == 3, R = R2; t = t1;
            elseif i == 4, R = R2; t = t2;
            end
        end
    end
end
% Triangulation helper function
function pts3d = triangulate_points(P1, pts1, P2, pts2)
    N = size(pts1,1);
    pts3d = zeros(N,3);
    for i = 1:N
        A = [pts1(i,1)*P1(3,:) - P1(1,:);
             pts1(i,2)*P1(3,:) - P1(2,:);
             pts2(i,1)*P2(3,:) - P2(1,:);
             pts2(i,2)*P2(3,:) - P2(2,:)];
        [~, ~, V] = svd(A);
        X = V(:,end);
        X = X ./ X(4);
```

```
        pts3d(i,:) = X(1:3)';
    end
end
```

Part 5: Full 3D Reconstruction Script

From the above code, we can calculate the 3D points pts3d from the points provided in ”data/temple coords.mat”. Then we can use the matlab to reconstruct 3D points of the image. And the P1,P2,R1,t1,R2,t2 is saved in the file data/extrinsics.mat. The picture of the reconstruction in 3D and the code is in the following.




```
% -------------------------
% Full Pipeline for Sparse 3D Reconstruction (MATLAB Version) + Epipolar Visualization + Extrinsics Saving
% -------------------------
clear; clc;
%% --- Load Data ---
data = load('data/some_corresp.mat');
pts1 = double(data.pts1);
pts2 = double(data.pts2);
intrinsics = load('data/intrinsics.mat');
```

```
K1 = intrinsics.K1;
K2 = intrinsics.K2;
im1 = imread('data/im1.png');
im2 = imread('data/im2.png');
M = max([size(im1,1), size(im1,2), size(im2,1), size(im2,2)]);
%% --- Compute Fundamental Matrix ---
F = eight_point(pts1, pts2, M);
%% --- Compute Essential Matrix ---
E = K2' * F * K1;
%% --- calculate P1, P2, R1, t1, R2, t2 from E, K1, K2, pts1, pts2---
[P1, P2, R1, t1, R2, t2] = compute_camera_matrices_from_E(E, K1, K2, pts1, pts2);
%% --- Load temple_coords for 3D reconstruction ---
temple = load('data/temple_coords.mat');
temple_pts1 = double(temple.pts1);
temple_pts2 = epipolar_correspondence(im1, im2, F, temple_pts1);
%% --- Load temple_pts3d from P1, temple_pts1, P2, temple_pts2 ---
[temple_pts3d, reprojection_error] = triangulate_check(P1, temple_pts1, P2, temple_pts2);
%% --- Save Extrinsics ---
save('data/extrinsics.mat','P1','P2','R1','t1','R2','t2');
%% --- Plot 3D points ---
figure; scatter3(temple_pts3d(:,1), temple_pts3d(:,2), temple_pts3d(:,3), 10, 'filled');
xlabel('X'); ylabel('Y'); zlabel('Z');
title('Reconstructed Temple 3D Point Cloud');
grid on; axis equal;
data = load('data/extrinsics.mat');
disp(data.t1)
```

Section 2: Dense Reconstruction

Part 1: Image Rectification

Create a program that calculates the rectification matrices for a pair of images. The function should take as arguments K1 , K2 , R1 , R2 , t1 , t2 and return values img1 rect , img2 rect , M1 ,M2 , K1p , K2p , R1p , R2p , t1p and t2p.

A dense stereo rectification algorithm is a process in stereo vision used to geometrically transform two images so that corresponding points lie along the same horizontal lines (epipolar lines become aligned and parallel). This significantly simplifies dense stereo matching (i.e., computing correspondences for every pixel), making it a one-dimensional search problem. It can be divided into 7 steps:
1. Compute Original Projection Matrices
The projection matrix for each camera is:

$$P_i = K_i \cdot [R_i \mid t_i], \quad i = 1, 2$$

2. Compute Optical Centers

Get camera centers (world coordinates) using $C_i = -R_i^\top t_i$. The baseline vector is $\mathbf{b} = C_2 - C_1$.

3. Define New Rectified Camera Coordinate System

Create a new camera frame aligned with the baseline:

z-axis: direction of baseline

$$\mathbf{z} = \frac{\mathbf{b}}{||\mathbf{b}||}$$

x-axis: cross product between old z-axis (e.g., optical axis) and new z

y-axis: $\mathbf{y} = \mathbf{z} \times \mathbf{x}$

Form new rotation matrix:

$$R_{rect} = \begin{bmatrix} \mathbf{x}^\top \\ \mathbf{y}^\top \\ \mathbf{z}^\top \end{bmatrix}$$

Both cameras are transformed to share this orientation (canonical configuration).

4. Update Camera Parameters

Assume intrinsics remain the same (or take average for symmetric rectification):

New intrinsics:

$$K_1' = K_1, \quad K_2' = K_2$$

New rotation matrices:

$$R_1' = R_{rect}, \quad R_2' = R_{rect}$$

New translations:

$$t_i' = -R' \cdot C_i, \quad i = 1, 2$$

5. Compute New Projection Matrices

$$P_i' = K_i'[R_i'|t_i']$$

6. Compute Rectification Homographies

Rectify the original images using a homography:

$$H_1 = P_1'(:, 1:3) \cdot P_1(:, 1:3)^{-1}$$
$$H_2 = P_2'(:, 1:3) \cdot P_2(:, 1:3)^{-1}$$

These homographies map image pixels from the original view to the rectified view.

7. Warp Images

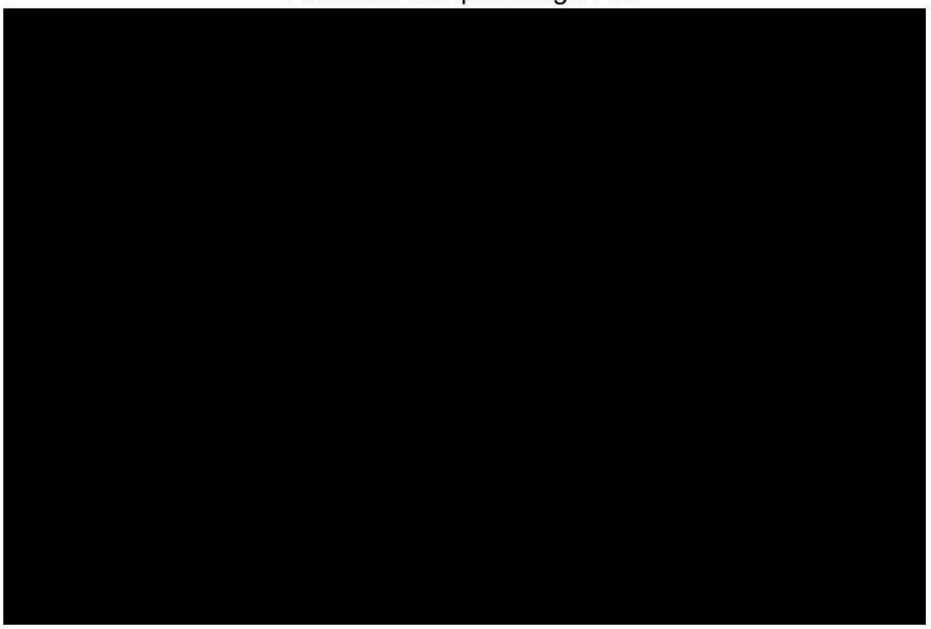
Use H_1 and H_2 to warp images I_1, I_2, creating:

$$I_1^{rect} = H_1(I_1)$$
$$I_2^{rect} = H_2(I_2)$$

This step aligns epipolar lines and enables 1D matching along horizontal lines.

I write and perform the code, but it seems to not work well and I only get a black picture. The reason may be the R1, R2, t1, t2 are inaccurate or there are still some bugs in my code. The picture and the code is in the following.

Rectified Temple Image Pair



```
% Load stereo images
im1 = imread('data/im1.png');
im2 = imread('data/im2.png');
% Load intrinsics
load('data/intrinsics.mat');  % gives K1, K2
% Load your previously computed extrinsics
load('data/extrinsics.mat');  % gives R1, R2, t1, t2
disp(R1);
disp(t1);
disp(R2);
disp(t2);
% Perform rectification
[img1_rect, img2_rect, M1, M2, K1p, K2p, R1p, R2p, t1p, t2p] = ...
    image_rectification(im1, im2, K1, K2, R1, R2, t1, t2)
% Display the result
figure;
imshowpair(img1_rect, img2_rect, 'montage');
title('Rectified Temple Image Pair');
function [img1_rect, img2_rect, M1, M2, K1p, K2p, R1p, R2p, t1p, t2p] = ...
    image_rectification(img1, img2, K1, K2, R1, R2, t1, t2)
%IMAGE_RECTIFICATION Computes stereo rectification for a pair of calibrated cameras
% 1. Compute projection matrices
P1 = K1 * [R1, t1];
P2 = K2 * [R2, t2];
% 2. Compute the new stereo baseline
```

```
% Center of each camera
c1 = -R1' * t1;
c2 = -R2' * t2;
baseline = c2 - c1;
% 3. Compute the new rectified camera coordinate system
z = baseline / norm(baseline);
v1 = R1(3,:)';  % optical axis of camera 1
x = cross(v1, z); x = x / norm(x);
y = cross(z, x);
% 4. Form new rotation matrix
R_rect = [x'; y'; z'];
% 5. New intrinsic matrices (assume unchanged or average)
K1p = K1;
K2p = K2;
% 6. Compute new extrinsic matrices
R1p = R_rect;
R2p = R_rect;
t1p = -R1p * c1;
t2p = -R2p * c2;
% 7. Compute new projection matrices
P1p = K1p * [R1p, t1p];
P2p = K2p * [R2p, t2p];
% 8. Compute rectification homographies (Hartley method)
H1 = P1p(:,1:3) / P1(:,1:3);
H2 = P2p(:,1:3) / P2(:,1:3);
M1 = H1;  % rectification matrix for image 1
M2 = H2;  % rectification matrix for image 2
% 9. Warp images
tform1 = projective2d(M1');
tform2 = projective2d(M2');
img1_rect = imwarp(img1, tform1, 'OutputView', imref2d(size(img1)));
img2_rect = imwarp(img2, tform2, 'OutputView', imref2d(size(img2)));
end
```

Part 2: Dense Window Matching for Disparity

Develop a program that computes a disparity map from a pair of rectified images. The program should accept as arguments im1 , im2 , max disp , win size.

For this issue, it just asks to compute a disparity map from a pair of rectified images, using block matching with Sum of Squared Differences (SSD). For each pixel in the left rectified image (im1), its best matching pixel in the right rectified image (im2) along the same scanline (row) by minimizing the SSD over a square patch. The horizontal distance (in pixels) between matching pixels is the disparity. The function can be divided into seven steps.

1. Preprocessing

   Convert color images to grayscale (if necessary).

   Convert images to `double` precision for accurate computations.

   Pad images to handle border issues while sliding patches.

2. Window Setup

Define the half window size:

$$\text{half\_w} = \left\lfloor \frac{\text{win\_size}}{2} \right\rfloor$$

This allows extracting a window centered at each pixel.

3. For Each Pixel (x, y) in im1

Skip the borders (leave at least `half_w` pixels on each side).

Extract a reference patch centered at `(x, y)` in `im1`.

4. Search Over Disparities

For each disparity $d \in [0, \min(\text{max\_disp}, x - \text{half\_w})]$:

Shift left by d pixels to define matching pixel `(x - d, y)` in `im2`.

Extract the comparison patch centered at `(x - d, y)` in `im2`.

5. Compute SSD

$$\text{SSD} = \sum_{i,j} [\text{patch1}(i,j) - \text{patch2}(i,j)]^2$$

6. Choose the Best Match

Keep track of the disparity d with the lowest SSD.

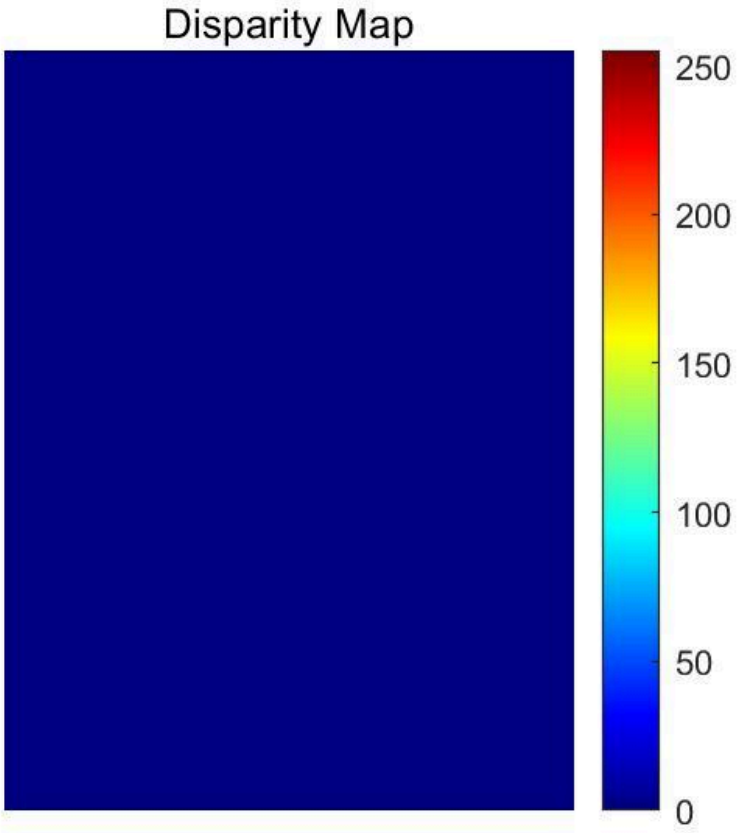
Set this d as the value for `disparity_map(y, x)`.

7. Postprocessing

Normalize disparity map to [0, 255] for visualization.

Cast to uint8 or leave as double depending on use case.

For the last step, I do not get the right rectified images, in this step, I do not get the correct disparity image. The disparity image and the code is in the following.




```
im1 = imread('data/img1_rectified.png');
im2 = imread('data/img2_rectified.png');
% Convert to grayscale if needed
if size(im1,3) == 3, im1 = rgb2gray(im1); end
if size(im2,3) == 3, im2 = rgb2gray(im2); end
% Parameters
max_disp = 50;
win_size = 9;
% Compute disparity
disp_map = compute_disparity_map(im1, im2, max_disp, win_size);
% Display result
```

```
figure; imshow(disp_map, []);
title('Disparity Map');
colormap(gca, jet); colorbar;
function disparity_map = compute_disparity_map(im1, im2, max_disp, win_size)
% Compute disparity map from rectified grayscale image pair
% Inputs:
%   im1, im2   - rectified grayscale images (same size)
%   max_disp   - maximum disparity to search (pixels)
%   win_size   - window size (odd integer, e.g., 5, 9)
% Output:
%   disparity_map - disparity value for each pixel
    % Convert to double if needed
    if ~isa(im1, 'double')
        im1 = double(im1);
    end
    if ~isa(im2, 'double')
        im2 = double(im2);
    end
    % Get image size
    [H, W] = size(im1);
    half_win = floor(win_size / 2);
    disparity_map = zeros(H, W);
    % Pad images to handle borders
    im1_pad = padarray(im1, [half_win, half_win], 'replicate');
    im2_pad = padarray(im2, [half_win, half_win + max_disp], 'replicate');
    % Compute disparity using Sum of Squared Differences (SSD)
    for y = 1 + half_win : H - half_win
        for x = 1 + half_win + max_disp : W - half_win
            best_offset = 0;
            min_ssd = inf;
            template = im1_pad(y:y+2*half_win, x:x+2*half_win);
            for d = 0 : max_disp
                block = im2_pad(y:y+2*half_win, x - d:x - d + 2*half_win);
                ssd = sum((template - block).^2, 'all');
                if ssd < min_ssd
                    min_ssd = ssd;
                    best_offset = d;
                end
            end
            disparity_map(y, x) = best_offset;
        end
    end
    % Normalize for display (optional)
    disparity_map = uint8(255 * disparity_map / max_disp);
end
```

Part 3: Depth Map

Construct a function that generates a depth map from a disparity map. The function should accept as arguments dispM, K1 , K2 , R1 , R2 , t1 , t2.

Given a disparity map and camera parameters (intrinsics and extrinsics), compute the depth of each pixel in the scene from a stereo image pair.

$$\text{depth}(x, y) = \frac{b \cdot f}{\text{disparity}(x, y)}$$

b: baseline (distance between the two camera centers)
f: focal length (typically from the intrinsic matrix)
disparity(x,y): horizontal shift between corresponding pixels

The process of generating a depth map from a disparity map can be divided into five steps.
1.Extract Intrinsic and Extrinsic Parameters:

The intrinsic camera matrix K_1 provides the focal length f.

The extrinsic parameters (R_1, t_1 and R_2, t_2) are used to compute the relative position of the two cameras, specifically the baseline b.

2.Calculate Baseline:

First, compute the camera centers C1C_1C1 and C2C_2C2 in world coordinates:

$$C_1 = -R_1^\top t_1$$
$$C_2 = -R_2^\top t_2$$

The baseline b=bb= is then the distance between these two camera centers:

$$b = \text{norm}(C_1 - C_2)$$

3. Initialize Depth Map:

Create a zero-filled depth map of the same size as the disparity map, initialized to zeros.

4.Compute Depth for Each Pixel:

Loop through each pixel (y, x) in the disparity map. For each pixel, if the disparity disp(y,x) is non-zero, compute the depth using the formula:

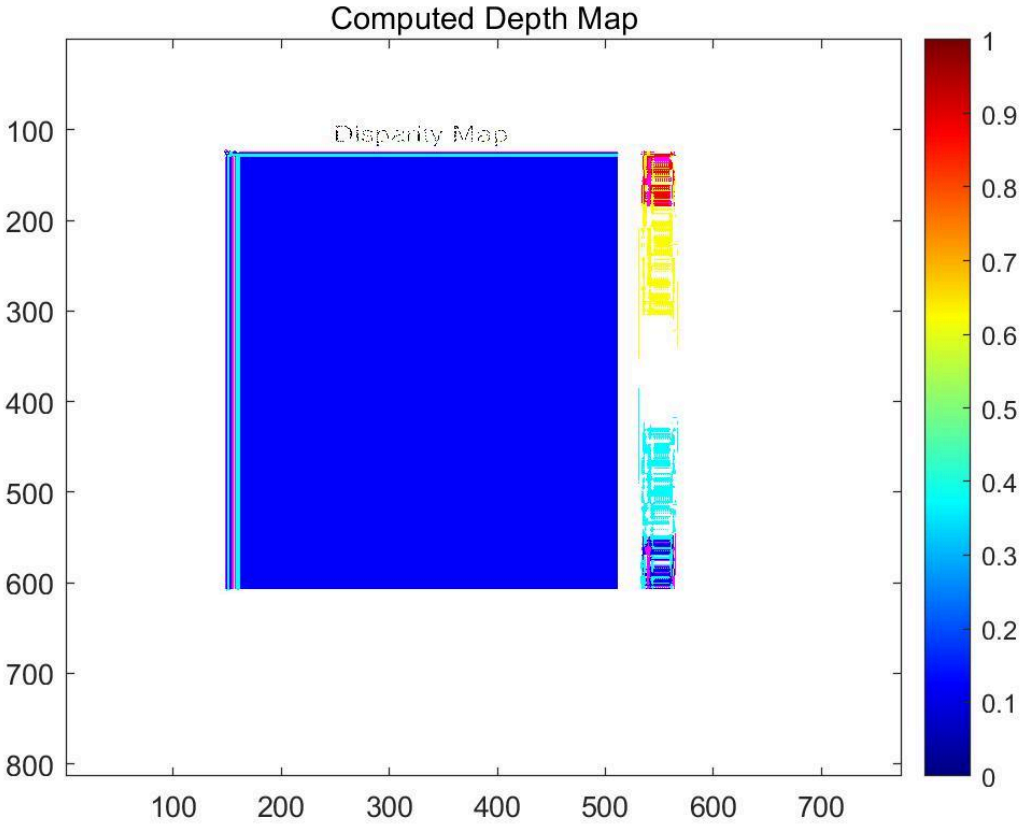
$$\text{depth}(y, x) = \frac{b \cdot f}{\text{disp}(y, x)}$$

If disp(y,x)=0, set the depth at that pixel to 0 to avoid division by zero.

5. Return Depth Map

After processing all the pixels, return the depth map.

In this issue, the results of my last two parts are not correct, so the depth map is not correct either. The depth map and the code are in the following.




```
dispM = imread('results/disparity_image.jpg');
depthM = compute_depth_map(dispM, K1, K2, R1, R2, t1, t2);
figure;
imagesc(depthM); colormap('jet'); colorbar;
title('Computed Depth Map');
function depthM = compute_depth_map(dispM, K1, K2, R1, R2, t1, t2)
    f = K1(1, 1);
    % Compute camera centers in world coordinates
    C1 = -R1' * t1;
    C2 = -R2' * t2;
    % Compute baseline (distance between optical centers)
    b = norm(C1 - C2);
    % Initialize depth map
    depthM = zeros(size(dispM));
    % Compute depth for each pixel
    valid = dispM > 0;  % mask where disparity is non-zero
    depthM(valid) = (b * f) ./ dispM(valid);
end
```