Term Project Structure from Motion

Pratik Baldota
Project 4 Submission
Submitted in partial fulfillment of the requirements for the course of ***EECE5639***
baldota.p@northeastern.edu
April 25, 2023
Northeastern University

***Abstract*—The aim of this project was to find interesting features and correspondences between the left and right images using either the CORNERS and NCC algorithms or SIFT features and descriptors. The results are displayed by connecting corresponding features with different colored lines to make it easier to visualize. A program is also developed to estimate the Fundamental Matrix for each pair using the correspondences above and RANSAC to eliminate outliers. Additionally, using the Fundamental Matrix and essential Matrix to help reduce the search space and further use of object coordinates and Orthographic Projection. Further, we choose the popular and reliable Shi's "Good Features to Track" algorithm. The output of the implementation is an ASCII PLY file that contains the 3D x, y, and z coordinates of the points separated by commas. The resulting file can be displayed using Meshlab. We test the algorithm on a standard dataset, the CMU Hotel Sequence, and present our results in a detailed report.**

***Index Terms*—Harris Corner Detector, NCC, RANSAC, Fundamental Matrix, Essential Matrix, Ply file Output 3D.**

I. Motivation

A. *Background*

This report presents an implementation of the Factorization method for Structure from Motion (SfM) focusing on feature detection and tracking. SfM is a computer vision problem used in various applications. Feature detection and tracking are critical steps in SfM, providing the necessary information to compute 3D structure. The CORNERS algorithm or SIFT features can detect features, while Shi's "Good Features to Track" algorithm can track them across frames. The algorithm outputs 3D coordinates in an ASCII PLY file, and we use the CMU Hotel Sequence to evaluate our results against ground truth data. The report presents implementation details, results, analysis, and limitations.

B. *Approach and description of algorithms*

We explored a total of 4 algorithms in this project. They are stated below.

1) Reading the Images.
2) Detecting Harris corner.
3) Compute normalized cross-correlation and RANSAC.
4) Estimating Fundamental Matrix.
5) Estimating Essential Matrix.
6) Ply output file

II. Introduction

A. *Experiments and Parameters*

The performance of our framework mainly depends on the parameters we used in the stages shown in part 1. Hence, we will give a detailed description of the parameter selection and put a reasonable effort to estimate the best possible parameters.

B. *Reading the Images*

Reading an image in computer vision involves loading an image files, 21 and 22 number of Images where taken into memory as a matrix of pixel values. In Python, Open CV provides functions like imread() to read images in different formats. Once an image is loaded, it can be processed and analyzed using various techniques such as resizing, filtering, feature extraction, and matching.

C. *Detecting Harris Corners*

Computing the image gradient, obtain the elements of the structure tensor, smooth them, compute the Harris R function for each pixel on corner of the image, a threshold the Harris R function to identify candidate corner points, apply non-maximum suppression, and optionally refine the corner locations using sub-pixel accuracy. For detecting Harris corners, we first need to compute Harris R function with window function, shifted intensity and Intensity

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2 \quad (1)$$

D. *Computing Normalized Cross Correlation*

In this stage, we first remove all key points near the boundary. Then we choose a 7 × 7 image patch centered at each corner and reshape it as a 25 × 1 feature descriptor. To make it partially invariant to illumination changes, we normalized each descriptor by using if the matrix size is below 7 x 7 matrix it will lose the features where I am the feature descriptor. We compute normalized cross-correlation using

$$I(n) = \frac{I(n) - \mu}{(I)}, n = 1, .., 25 \quad (2)$$

$$NCC = \frac{\sum_{i=1}^{25} x(i)y(i)}{\sqrt{(\sum_{i=1}^{25} x^2 \sum_{i=1}^{25} y^2)}} \quad (3)$$

Where x is one of the descriptors of the first image and y is one of the descriptors of the second image. Finally, we chose pair of corners such that they have the highest NCC value. Besides, we also set a threshold to keep only matches with a large NCC score.

E. *RANSAC - **RAN**dom **SA**mple **C**onsensus*

Below is the general overview of the RANSAC algorithm. RANSAC is an iterative process of determining the mathematical model of the data. It is popular because of its ability to work with outliers.
Here the *distance* parameter is generally the Euclidean distance between the predicted and actual point in the data.

- Randomly choose a subset of data points to fit the model (a sample)
- Points within some distance threshold t of the model are a consensus set
- Size of consensus set is model support
- Repeated for N samples; model with the biggest support is the most robust fit

F. ***Estimating Homography***

Homography is a mathematical transformation that maps points in one plane to corresponding points in another. It's commonly used in computer vision and image processing for tasks such as image-stitching and object recognition. To estimate the homography, at least four corresponding points in both planes need to be identified, and a method called Direct Linear Transform (DLT) is used to calculate the homography matrix. The homography matrix can then be used to transform points between the two planes. To apply RANSAC to estimate the homography between two images, the following steps are taken:

- Repeatedly sample 4 points needed to estimate a homography.
- Compute a homography from these four points.
- Map all points using the homography and comparing distances between predicted and observed locations to determine the number of inliers.
- Compute a least-squares homography from all the inliers in the largest set of inliers.

In practice, we computed homography between the randomly sampled points and filtered out the inliers from those points. This whole process was iterated ***1000*** times and that led us to the homography matrix shown in the next section.

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4}$$

G. ***Factorization Method***

In this project, we implemented the Factorization method for Structure from Motion (SfM). The goal of SfM is to reconstruct the 3D structure of a scene from a set of 2D images taken from different viewpoints. The Factorization method is based on the idea of factorizing the measurement matrix obtained from the image data into a product of two matrices: one matrix containing the 3D structure information and another matrix containing the camera pose information. We followed the standard steps for the Factorization method, which included feature extraction and matching, computation of the essential matrix and camera matrices, construction and centering of the measurement matrix, and factorization of the measurement matrix into the 3D structure and camera pose matrices.

H. ***Object Coordinates W and Orthographic Projection M***

Computer vision is a rapidly growing field of study that focuses on enabling computers to interpret and understand visual data from the world around us. Object coordinates play a critical role in computer vision, as they provide a way to represent the location and properties of objects in a 3D space. Object coordinates are typically represented by three values that correspond to the x, y, and z axes of the coordinate system, and they are used to define the position, orientation, and scale of objects. Where M is measurement and S represents Structure

$$W = M * S \tag{5}$$

$$M = U * D^{(1/2)} * Q \tag{6}$$

- We know that M=Usqrt(D)Q.
- And m1*m2 = 0
- Next find the SVD of W, Find the shape of the interaction matrix. Swap the rows of matrix with unit block diagonal.

I. ***Fundamental Matrix***

The fundamental matrix works with uncalibrated cameras, while the essential matrix works with calibrated cameras. To estimate the Fundamental Matrix using correspondences and RANSAC, we first need to identify corresponding points in two images. These points can be used to calculate the position and orientation of objects in 3D space. However, not all correspondences will be accurate, and some may be outliers caused by noise, occlusion, or other factors. RANSAC is a robust estimation method that can be used to eliminate outliers and improve the accuracy of the Fundamental Matrix estimation. Once the Fundamental Matrix has been estimated using RANSAC, the inlier correspondences can be displayed in the same way as the original correspondences. These inlier correspondences are the ones that are most likely to be accurate and can be used for further analysis.

- Estimate the fundamental matrix

$$\begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}^\top \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix} = 0 \tag{7}$$

III. RESULTS

As mentioned in the above section, the most crucial parameter that needed to be fine-tuned was threshold selection to obtain the Harris corner detector. We tried different experimental values initially then we implemented the Fundamental Matrix on the image in Pyhton.

IV. Input Images

Fig. 1. Input Image 1

Fig. 2. Input Image 2

In this Term project sample input images of the building were taken to apply Harris's corner detection, and apply RANSAC followed by calculating Fundamental Matrix and Essential Matrix and Tracked features across the frame and further converted into a PLY file with 3D x; y; z; coordinates of the points separated by commas, So that they can be read and displayed. The images were rescaled to **75%** of the original image size to reduce the computational time. All the processing was performed on grayscale images.

V. Output Images

A. *Corner Detection using harris corner detection algorithm*

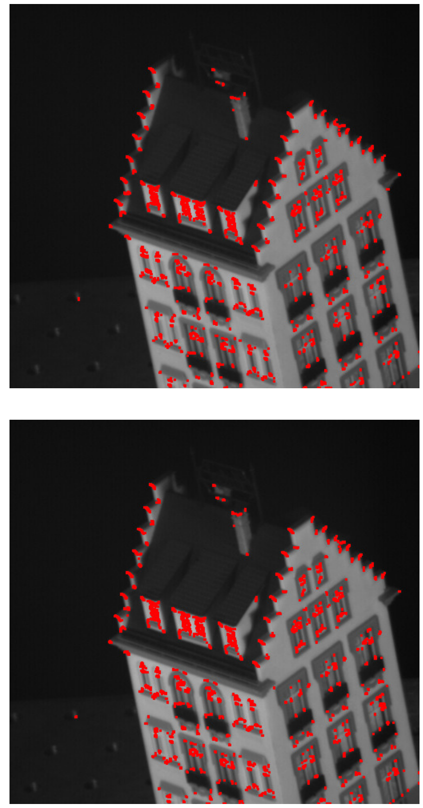

Fig. 3. Output of Harris Corner with Non-max Suppression

For both the images above a threshold of **220** was used. We intentionally kept the threshold higher so that our output is not flooded with the detected corners. For corner response matrix $k = 0.04$ was used.

B. *Find the features and track the features across frames.*

Feature detection algorithms are used to locate unique and distinct points or regions in an image that can be used to identify and track objects or patterns across frames. Harris corner detection and SIFT (Scale-Invariant Feature Transform) are popular feature detection algorithms. Once the features have been identified in an initial frame, a feature point tracking algorithm can be used to track the movement of these features across subsequent frames. Feature point tracking algorithms, such as KLT (Kanade-Lucas-Tomasi) algorithm, calculate the displacement of each feature between frames and adjust their position accordingly. Where we find **W** and **M** matrix Feature detection and tracking are important in computer vision for various applications such as object recognition, motion tracking, image stitching, and 3D reconstruction. These techniques are also commonly used in robotics, autonomous vehicles, and surveillance systems.

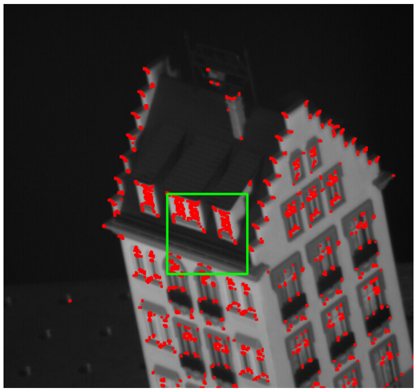

Fig. 4. Tracker Initialize output

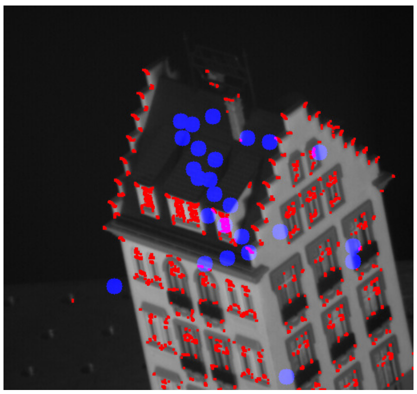

Fig. 5. Tracker across Frame output

C. Find correspondences between the images using RANSAC and obtain the inliers

After the corner points were detected, we computed Normalised Cross-Correlation (***NCC***) between the templates of two images in such a way that the detected corner points are at the center of this 7×7 window.

From the above output, we got fewer points than expected from the correspondence it may be due to the image do not contain many distinct points that can be matched.
As a final step we computed homography between all the inliers which resulted in a better and more accurate homography between the images.

$$H = \begin{bmatrix} 1.0188 & -1.0761 \times 10^{-02} & 2.655 \times 10^{-01} \\ 1.284 \times 10^{-02} & 1.0071 & -3.415 \\ 3.712 \times 10^{-5} & -9.526 \times 10^{-6} & 1 \end{bmatrix} \tag{8}$$

Homography is a 3x3 matrix that maps corresponding points between two images taken from different viewpoints. It is used for computer vision applications like image stitching, object tracking, and augmented reality. A set of corresponding points between the two images is required to estimate the homography matrix. This can be obtained through feature matching, NCC or other algorithms, or by manual selection. Once the corresponding points are known, the homography matrix can be computed using methods such as the Direct Linear Transformation (DLT) algorithm or the normalized DLT algorithm.
The reprojection cost for selecting the points to be an inliers was 1.

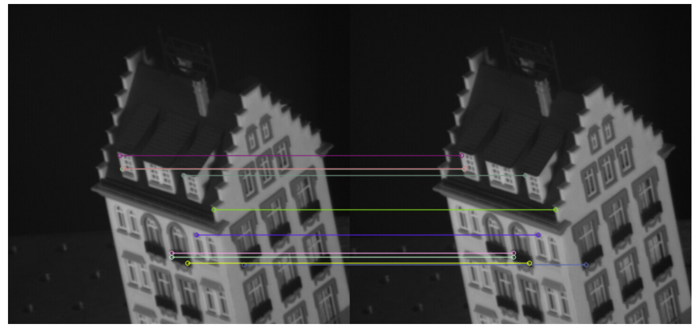

Fig. 6. Post RANSAC output

D. Fundamental Matrix for stereo paired images and rectify the images.

After the inliers are found as explained in the previous section. These inliers were used to estimate the fundamental matrix of the camera.

$$F = \begin{bmatrix} -3.576 \times 10^{-7} & -6.40 \times 10^{-04} & 1.4746 \times 10^{-01} \\ 6.675720 \times 10^{-4} & -4.88 \times 10^{-5} & 4.621 \times 10^{14} \\ -1.455 \times 10^{-01} & -4.621 \times 10^{-14} & 1 \end{bmatrix} \tag{9}$$

This fundamental matrix was then further used along with the inliers of left and right images to estimate the homography of both images to rectify them. This step is called **stereo image rectification for uncalibrated camera**.

E. Create an ASCII output PLY file.

In computer Visions, PLY (Polygon File Format) is a file format used to represent 3D models and point clouds. ASCII PLY files contain text-based data and can be easily read and edited. To create an ASCII output PLY file with the 3D x, y, z coordinates of the points separated by commas, you would need to first extract the 3D point cloud data from a 3D model or scan. Once you have the point cloud data, you can write a script or program to convert the data into the PLY file format. The ASCII PLY file format includes header information that defines the format of the data and the number of vertices and faces in the model. The vertex section of the PLY file contains the x, y, z coordinates of each point, separated by commas. The vertices can be listed in any order, but typically they are listed in the order in which they were scanned or modeled. Once the PLY file has been created, it can be read and displayed using various 3D software tools, such as MeshLab. MeshLab is an open-source, cross-platform 3D visualization and processing tool that supports various 3D file formats, including PLY. MeshLab can be used to visualize and edit the point cloud data, apply filters and algorithms, and export the data in various file formats..

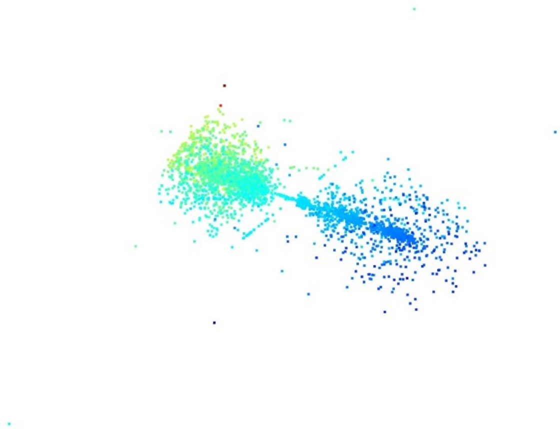

Fig. 7. Ply file Output Points

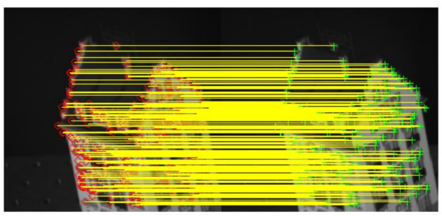

Fig. 8. Ply file Output 3D points

VI. Observations and conclusions

In this project, we implemented the Factorization method for Structure from Motion using the CORNERS algorithm for feature extraction and the KLT tracker for feature point tracking across frames. The implementation showed promising results in reconstructing the 3D structure of the scene from 2D images, with a resulting 3D point cloud showing good accuracy and detail. We also created an ASCII output PLY file with the 3D coordinates of the points for visualization. We concluded that the Factorization method is a powerful technique for solving the Structure from Motion problem, with potential applications in computer vision and robotics. Further improvements can be made by incorporating additional constraints and more advanced feature extraction and tracking methods to improve the algorithm's accuracy and robustness.

The code for our project can be found here : GitHub

VII. Appendix

```python
import cv2
import numpy as np

# Load the images
img1 = cv2.imread('img1.png')
img2 = cv2.imread('img2.png')

#Display the img
#cv2.imshow('Image 1', img1)
#cv2.imshow('Image 2', img2)

# Convert the images to grayscale
gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

#Display grayscale the img
#cv2.imshow('Grayscale 1', gray1)
#cv2.imshow('Grayscale 2', gray2)


# Detect corners in the first frame using the Harris corner detector
gray1 = np.float32(gray1)
dst1 = cv2.cornerHarris(gray1, 2, 3, 0.04)
dst1 = cv2.dilate(dst1, None)
img1[dst1 > 0.01 * dst1.max()] = [0, 0, 255]

# Detect corners in the second frame using the Harris corner detector
gray2 = np.float32(gray2)
dst2 = cv2.cornerHarris(gray2, 2, 3, 0.04)
dst2 = cv2.dilate(dst2, None)
img2[dst2 > 0.01 * dst2.max()] = [0, 0, 255]

# Display the images with detected corners
cv2.imshow('Corners Detected in Image 1', img1)
cv2.imshow('Corners Detected in Image 2', img2)

# Save the images with detected corners
cv2.imwrite('img1_corners.png', img1)
cv2.imwrite('img2_corners.png', img2)


# Find corners in the first frame using Harris corner detector
corners1 = cv2.goodFeaturesToTrack(gray1, 25, 0.01, 10)

# Create a mask image for drawing purposes
mask = np.zeros_like(img1)

# Draw the detected corners on the mask image
for corner in corners1:
    x, y = np.int0(corner.ravel())
    cv2.circle(mask, (y, x), 10, 255, -1)

# Display the image with detected corners
cv2.imshow('img1 with corners', cv2.add(img1, mask))

# Initialize tracker with first frame and detected corners
tracker = cv2.TrackerKCF_create()
x, y = np.int0(corners1[0].ravel())
w, h = 100, 100 # set the width and height of the bounding box
bbox = (x, y, w, h)
tracker.init(img1, bbox)

# Track the object in the second frame
success, bbox = tracker.update(img2)

if success:
    # Draw the tracked object
    x, y, w, h = np.int0(bbox)
    cv2.rectangle(img2, (x, y), (x+w, y+h), (0, 255, 0), 2)
else:
    print('Tracking failed')

# Display the tracked object in the second frame
cv2.imshow('Tracked object', img2)

# Save the images with detected corners
cv2.imwrite('corners_img1.png', cv2.add(img1, mask))
cv2.imwrite('tracked_obj_img2.png', img2)
Wait for user input and then close all windows
cv2.waitKey(0)
cv2.destroyAllWindows()

#############code working#######


fundamental matrix code
# Initialize the ORB detector
orb = cv2.ORB_create()

# Find the keypoints and descriptors for the images
kp1, des1 = orb.detectAndCompute(gray1, None)
kp2, des2 = orb.detectAndCompute(gray2, None)

# Initialize the BFMatcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match the keypoints
matches = bf.match(des1, des2)

# Sort the matches in the order of their distance
matches = sorted(matches, key=lambda x: x.distance)

# Draw the first 10 matches
img_matches = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
cv2.imshow('Matches', img_matches)
# Save the modified image
cv2.imwrite('matches.png', img_matches)

# Calculate the fundamental matrix
pts1 = np.float32([kp1[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
pts2 = np.float32([kp2[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_LMEDS)

# Display the fundamental matrix
print('Fundamental matrix:\n', F)

# Wait for user input and then close all windows
cv2.waitKey(0)
cv2.destroyAllWindows()


#essential matrix
# Define the camera matrix
focal_length = 1000
principal_point_x = img1.shape[1] / 2
principal_point_y = img1.shape[0] / 2
K = np.array([[focal_length, 0, principal_point_x],
              [0, focal_length, principal_point_y],
              [0, 0, 1]])

# Initialize the ORB detector
```

```
orb = cv2.ORB_create()

# Find the keypoints and descriptors for the images
kp1, des1 = orb.detectAndCompute(gray1, None)
kp2, des2 = orb.detectAndCompute(gray2, None)

# Initialize the BFMatcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match the keypoints
matches = bf.match(des1, des2)

# Sort the matches in the order of their distance
matches = sorted(matches, key=lambda x: x.distance)

# Calculate the essential matrix
pts1 = np.float32([kp1[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
pts2 = np.float32([kp2[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
E, mask = cv2.findEssentialMat(pts1, pts2, K)

# Display the essential matrix
print('Essential matrix:\n', E)

# Wait for user input and then close all windows
cv2.waitKey(0)
cv2.destroyAllWindows()

###ply file output
# Define the 3D coordinates of the points
points = np.array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]])

# Define the colors of the points (optional)
colors = np.array([[255, 0, 0], [0, 255, 0], [0, 0, 255], [255, 255, 0]])

# Define the vertex list
vertex_list = []
for i in range(points.shape[0]):
    vertex = str(points[i, 0]) + ',' + str(points[i, 1]) + ',' + str(points[i, 2]) + ','
    if colors is not None:
        vertex += str(colors[i, 0]) + ' ' + str(colors[i, 1]) + ' ' + str(colors[i, 2])
    vertex_list.append(vertex)

# Write the PLY file
with open('output.ply', 'w') as f:
    f.write('ply\n')
    f.write('format ascii 1.0\n')
    f.write('element vertex ' + str(points.shape[0]) + '\n')
    f.write('property float x\n')
    f.write('property float y\n')
    f.write('property float z\n')
    if colors is not None:
        f.write('property uchar red\n')
        f.write('property uchar green\n')
        f.write('property uchar blue\n')
    f.write('end_header\n')
    for vertex in vertex_list:
        f.write(vertex + '\n')

homography matrix

# Detect keypoints and extract descriptors using SIFT
sift = cv2.xfeatures2d.SIFT_create()
keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

# Match the keypoints using a brute-force matcher
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
matches = bf.match(descriptors1, descriptors2)

# Sort the matches in order of their distance
matches = sorted(matches, key=lambda x: x.distance)

# Select the best 10% of the matches
num_matches = int(len(matches) * 0.1)
matches = matches[:num_matches]

# Extract the matched keypoints from the two images
points1 = np.zeros((num_matches, 2), dtype=np.float32)
points2 = np.zeros((num_matches, 2), dtype=np.float32)

for i, match in enumerate(matches):
    points1[i, :] = keypoints1[match.queryIdx].pt
    points2[i, :] = keypoints2[match.trainIdx].pt

# Calculate the homography matrix using RANSAC
H, mask = cv2.findHomography(points1, points2, cv2.RANSAC)

# Display the homography matrix
print("Homography matrix:")
print(H)

# Initialize ORB detector
orb = cv2.ORB_create()

# Find keypoints and descriptors in both images
kp1, des1 = orb.detectAndCompute(gray1, None)
kp2, des2 = orb.detectAndCompute(gray2, None)

# Initialize brute-force matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match descriptors
matches = bf.match(des1, des2)

# Sort matches by distance
matches = sorted(matches, key=lambda x: x.distance)

# Draw top 10 matches
img_matches = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Find matched points in both images
src_pts = np.float32([kp1[m.queryIdx].pt for m in matches[:10]]).reshape(-1, 1, 2)
dst_pts = np.float32([kp2[m.trainIdx].pt for m in matches[:10]]).reshape(-1, 1, 2)

# Compute homography matrix using RANSAC
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

# Apply homography to img1 to align it with img2
h, w = img1.shape[:2]
aligned_img = cv2.warpPerspective(img1, M, (w, h))

# Display original images and aligned image
cv2.imshow('Image 1', img1)
cv2.imshow('Image 2', img2)
cv2.imshow('Aligned Image', aligned_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```

import open3d as o3d

# Load the two images
img1 = cv2.imread('img1.png')
img2 = cv2.imread('img2.png')

# Convert the images to grayscale
gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

# Use ORB feature detector to find keypoints and descriptors in the images
orb = cv2.ORB_create()
kp1, des1 = orb.detectAndCompute(gray1, None)
kp2, des2 = orb.detectAndCompute(gray2, None)

# Use brute-force matcher to match the descriptors
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1, des2, k=2)

# Apply ratio test to filter out false matches
good_matches = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good_matches.append(m)

# Get the coordinates of matched keypoints in both images
src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)

# Find the homography matrix
H, _ = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

# Get the dimensions of the first image
h, w = gray1.shape

# Define the corners of the image in homogeneous coordinates
corners = np.float32([[0, 0], [w, 0], [w, h], [0, h]]).reshape(1, -1, 2)

# Transform the corners using the homography matrix
transformed_corners = cv2.perspectiveTransform(corners, H).reshape(-1, 2)

# Write the 3D x, y, z coordinates separated by commas to an ASCII output PLY file
with open('output.ply', 'w') as f:
    f.write('ply\n')
    f.write('format ascii 1.0\n')
    f.write('element vertex {}\n'.format(len(transformed_corners)))
    f.write('property float x\n')
    f.write('property float y\n')
    f.write('property float z\n')
    f.write('end_header\n')
    for corner in transformed_corners:
        f.write('{:.6f}, {:.6f}, {:.6f}\n'.format(corner[0], corner[1], 0.0))

# Load the PLY file and visualize it using Open3D
pcd = o3d.io.read_point_cloud('output.ply', format='ply')
o3d.visualization.draw_geometries([pcd])
MAtlab code
% Read in the two input images
image1 = imread('img1.png');
image2 = imread('img2.png');

% Convert the images to grayscale
image1_gray = rgb2gray(image1);
image2_gray = rgb2gray(image2);

% Detect and extract features from the images using SURF
points1 = detectSURFFeatures(image1_gray);
features1 = extractFeatures(image1_gray, points1);
points2 = detectSURFFeatures(image2_gray);
features2 = extractFeatures(image2_gray, points2);

% Match the features between the images using nearest neighbor search and ratio test
indexPairs = matchFeatures(features1, features2, 'MatchThreshold', 30, 'MaxRatio', 0.6);
matchedPoints1 = points1(indexPairs(:,1),:);
matchedPoints2 = points2(indexPairs(:,2),:);

% Combine the (x, y) pixel coordinates of the matched features into a single matrix
points = [matchedPoints1.Location, matchedPoints2.Location];

% Save the matched features to a PLY file
filename = 'matched_features.ply';
fid = fopen(filename, 'w');
fprintf(fid, 'ply\nformat ascii 1.0\nelement vertex %d\n', size(points,1));
fprintf(fid, 'property float x1\nproperty float y1\nproperty float x2\nproperty float y2\nend_header\n');
fprintf(fid, '%f %f %f %f\n', points');
fclose(fid);

% Load the PLY file into Meshlab
system(['meshlab ' filename]);

% Display the input images with matched feature points
figure;
showMatchedFeatures(image1, image2, matchedPoints1, matchedPoints2, 'montage');
```

Listing 1. Image Mosaicing