Computer Vision

Assignment - 2 Report

Samridh Girdhar
2021282

Question 1:

Transformations on Vector $\mathbf{v} = (3, -1, 4)^\mathsf{T}$

Solution Outline

We have four successive transformations on the vector $\mathbf{v} = (3, -1, 4)^\mathsf{T}$:

1. A rotation of $-\frac{\pi}{6}$ about the y-axis.
2. A rotation of $\frac{\pi}{4}$ about the x-axis.
3. A reflection across the xz-plane ($y \mapsto -y$).
4. A translation by $(1, 0, -2)^\mathsf{T}$.

Below is a step-by-step derivation of each requested item.

1) The Overall Coordinate-Transformation Matrix (Rotation + Reflection)

Ignoring translation for the moment, the combined linear part is

$$M_{\text{rf}} = R_f \, R_x\Big(\frac{\pi}{4}\Big) \, R_y\Big(-\frac{\pi}{6}\Big),$$

where

- $R_y(\theta)$ is the standard rotation about the y-axis by angle θ:

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}.$$

  For $\theta = -\frac{\pi}{6}$, we have $\cos\left(-\frac{\pi}{6}\right) = \frac{\sqrt{3}}{2}$, $\sin\left(-\frac{\pi}{6}\right) = -\frac{1}{2}$, so

$$R_y\left(-\frac{\pi}{6}\right) = \begin{pmatrix} \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{pmatrix}.$$

- $R_x(\phi)$ is the rotation about the x-axis by ϕ:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix}.$$

  For $\phi = \frac{\pi}{4}$, $\cos\left(\frac{\pi}{4}\right) = \sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$, giving

$$R_x\left(\frac{\pi}{4}\right) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}.$$

- R_f is the reflection across the xz-plane, i.e., $y \mapsto -y$:

$$R_f = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Matrix multiplication $R_x\left(\frac{\pi}{4}\right) R_y\left(-\frac{\pi}{6}\right)$ followed by left-multiplying by R_f yields

$$M_{\text{rf}} = R_f\, R_x\Big(\frac{\pi}{4}\Big)\; R_y\Big(-\frac{\pi}{6}\Big) = \begin{pmatrix} \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} \\ \frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \\ \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \end{pmatrix}.$$

2) New Coordinates of $\mathbf{v} = (3, -1, 4)$ and Image of the Origin

New coordinates of v

Including translation by $(1, 0, -2)$, the full transformation sends any point $\mathbf{p}$ to

$$M_{\text{rf}}\,\mathbf{p} + \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix}.$$

For $\mathbf{v} = (3, -1, 4)$:

1. Compute the rotation-reflection $M_{\text{rf}}\,\mathbf{v}$:

$$\mathbf{w} = \begin{pmatrix} \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} \\ \frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \\ \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \end{pmatrix} \begin{pmatrix} 3 \\ -1 \\ 4 \end{pmatrix} = \begin{pmatrix} \frac{3\sqrt{3}}{2} - 2 \\ \frac{5\sqrt{2}}{4} + \sqrt{6} \\ \frac{\sqrt{2}}{4} + \sqrt{6} \end{pmatrix}.$$

2. Add the translation $(1, 0, -2)$:

$$\mathbf{w} + (1, 0, -2)^{\mathsf{T}} = \begin{pmatrix} \frac{3\sqrt{3}}{2} - 1 \\ \frac{5\sqrt{2}}{4} + \sqrt{6} \\ \frac{\sqrt{2}}{4} + \sqrt{6} - 2 \end{pmatrix}.$$

Image of the origin

For the origin $\mathbf{0}$:

$$M_{\text{rf}}\,\mathbf{0} + (1, 0, -2)^{\mathsf{T}} = (1, 0, -2)^{\mathsf{T}}.$$

3) Axis and Angle of the Combined Rotation (Excluding the Reflection)

Excluding R_f, consider

$$R = R_x\Big(\frac{\pi}{4}\Big)\; R_y\Big(-\frac{\pi}{6}\Big).$$

This is a single rotation by angle α about a unit axis $\mathbf{n}$.

- **Angle α:** Using $\text{trace}(R) = 1 + 2\cos\alpha$, we compute $\text{trace}(R) \approx 2.1855$, so

$$\cos\alpha = \frac{\text{trace}(R) - 1}{2} \approx 0.593 \quad \Rightarrow \quad \alpha \approx 0.94\,\text{radians} \approx 53.8^\circ.$$

- **Axis n:** The eigenvector of R with eigenvalue 1, normalized, is approximately

$$\mathbf{n} \approx (-0.819, 0.530, 0.220).$$

Thus, the net effect is a rotation by $\alpha \approx 53.8^\circ$ about $\mathbf{n} \approx (-0.819, 0.530, 0.220)$.

4) Verification via Rodrigues' Formula

Rodrigues' formula for a rotation by α about $\mathbf{n}$ is

$$R = I\cos\alpha + (1-\cos\alpha)\mathbf{n}\mathbf{n}^{\mathsf{T}} - [\mathbf{n}]_{\times}\sin\alpha.$$

Using $\alpha \approx 0.94$ and $\mathbf{n} \approx (-0.819, 0.530, 0.220)$, this reconstructs $R_x\left(\frac{\pi}{4}\right) R_y\left(-\frac{\pi}{6}\right)$, confirming the result.

Question 2:

Camera Projections and Homography

Below is a step-by-step derivation using the first camera as the world reference frame (extrinsics $[I \mid \mathbf{0}]$) and the second camera rotated by R (no translation).

1. **Projection by the first camera:** With intrinsics K_1, a 3D point X projects to

$$\mathbf{x}_1 = K_1 \, [I \mid \mathbf{0}] \, X = K_1 \, X.$$

2. **Projection by the second camera:** With intrinsics K_2 and extrinsics $[R \mid \mathbf{0}]$,

$$\mathbf{x}_2 = K_2 \, [R \mid \mathbf{0}] \, X = K_2 \, R \, X.$$

3. **Express X in terms of $\mathbf{x}_2$:** From the second camera,

$$\mathbf{x}_2 = K_2 \, R \, X \quad \Rightarrow \quad X = R^{-1} \, K_2^{-1} \, \mathbf{x}_2.$$

4. **Substitute into the first-camera equation:**

$$\mathbf{x}_1 = K_1 \, X = K_1 \, (R^{-1} \, K_2^{-1} \, \mathbf{x}_2) = (K_1 \, R^{-1} \, K_2^{-1}) \, \mathbf{x}_2.$$

Thus,

$$\mathbf{x}_1 = H \, \mathbf{x}_2, \quad \text{where} \quad H = K_1 \, R^{-1} \, K_2^{-1}.$$

Since K_1, K_2, and R are invertible 3×3 matrices, $H = K_1 \, R^{-1} \, K_2^{-1}$ is a nonsingular 3×3 matrix, establishing the homography $\mathbf{x}_1 = H \, \mathbf{x}_2$.

Question 3: Camera Calibration

1 Intrinsic Camera Parameters

The estimated intrinsic parameters from the calibration are as follows:

- **Focal Lengths:** [956.64, 957.55]
- **Skew:** 0.0
- **Principal Point:** [369.05, 651.41]

1.1 Code for Intrinsic Parameter Estimation

Below is a snippet showing the key steps in detecting chessboard corners and computing the camera matrix using OpenCV:

Listing 1: Chessboard Detection and Intrinsic Parameter Estimation

```
import cv2
import numpy as np
import glob

# Define criteria and prepare object points
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)

objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('calibration_images/*.jpg')
for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (9,6), None)
    if ret:
        objpoints.append(objp)
        corners2 = cv2.cornerSubPix(gray, corners, (11,11), (-1,-1), criteria)
        imgpoints.append(corners2)
        cv2.drawChessboardCorners(img, (9,6), corners2, ret)
        cv2.imshow('img', img)
        cv2.waitKey(100)
cv2.destroyAllWindows()

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None,
    ↪ None)
print("Camera Matrix:\n", mtx)
```

2 Extrinsic Camera Parameters for the First Two Images

The extrinsic parameters (rotation matrices and translation vectors) for images 1 and 2 are reported below.

Image 1

Rotation Matrix:

$$\begin{bmatrix} 0.99699847 & 0.01387206 & -0.07616832 \\ -0.02511535 & 0.98856270 & -0.14870444 \\ 0.07323433 & 0.15017110 & 0.98594390 \end{bmatrix}$$

Translation Vector:

$$\begin{bmatrix} -3.9383 \\ -3.6172 \\ 14.6591 \end{bmatrix}$$

Image 2

Rotation Matrix:

$$\begin{bmatrix} 0.99828333 & 0.05391999 & -0.02286982 \\ -0.05069018 & 0.99100787 & 0.12383014 \\ 0.02934110 & -0.12245829 & 0.99203985 \end{bmatrix}$$

Translation Vector:

$$\begin{bmatrix} -4.3713 \\ -1.4596 \\ 15.8138 \end{bmatrix}$$

2.1 Code for Extracting Extrinsic Parameters

The following code snippet shows how the extrinsic parameters for each image are computed during calibration:

Listing 2: Extracting Extrinsic Parameters

```
# After calibrating the camera using cv2.calibrateCamera,
# rvecs and tvecs hold the extrinsic parameters for each image.

# Example: print extrinsic parameters for the first two images
for i in range(2):
    rvec = rvecs[i]
    tvec = tvecs[i]
    R, _ = cv2.Rodrigues(rvec)
    print("Image", i+1, "Rotation␣Matrix:\n", R)
    print("Image", i+1, "Translation␣Vector:\n", tvec)
```

3 Radial Distortion Coefficients and Image Undistortion

The estimated radial distortion coefficients are:

$$[0.19764,\ -0.70686,\ 0.04877]$$

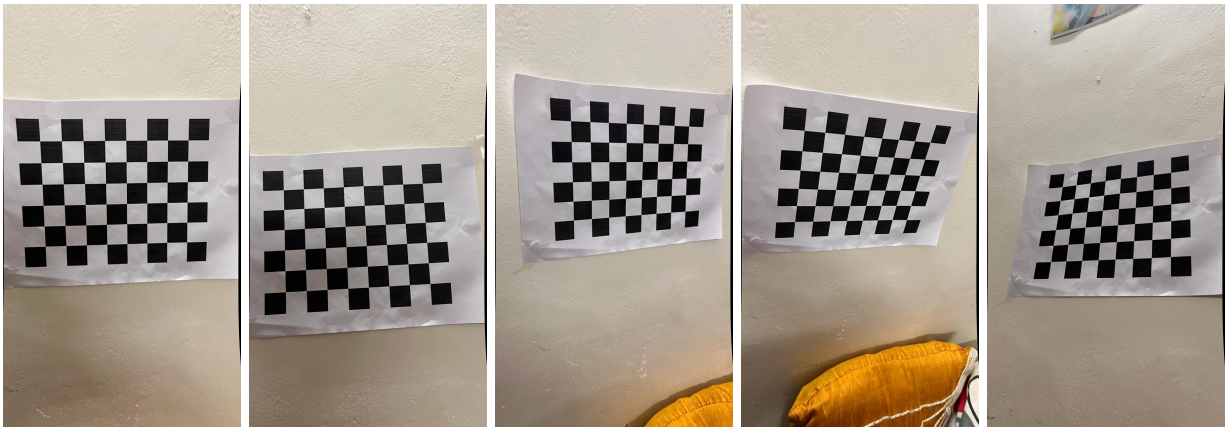
Using these coefficients, five raw images were undistorted. Upon undistortion, straight lines (especially those at the image corners) that appeared curved due to lens distortion become significantly more linear. This improvement in line straightness demonstrates the effectiveness of the calibration in correcting radial distortions.

3.1 Image Undistortion

Below is an example snippet for undistorting an image:

Listing 3: Undistorting an Image

```
img = cv2.imread('raw_image.jpg')
h, w = img.shape[:2]
newcameramtx, roi = cv2.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w,h))
dst = cv2.undistort(img, mtx, dist, None, newcameramtx)
cv2.imwrite('undistorted_image.jpg', dst)
```



(a) Image 1 (b) Image 2 (c) Image 3 (d) Image 4 (e) Image 5

Figure 1: Example of an undistorted images. Notice the correction in the curvature of lines near the image corners.
Comment: "After applying the estimated radial distortion coefficients, we undistorted five raw images. Because the lens in our device is not strongly wide-angled, the geometric change near the edges is relatively subtle. Nonetheless, closer inspection (especially near image corners) shows slightly straighter horizontal/vertical lines compared to the original images."

4 Re-projection Error Analysis

The re-projection error was computed for each of the 25 images by measuring the Euclidean distance between the detected chessboard corners and the re-projected 3D model corners. following are the observations:

- **Per-image errors:**

  [0.2663171, 0.2759759, 0.51571685, 0.60863924, 0.53886455, 0.43892765, 0.51803553, 0.2763334, 0.25553098, 0.29681656, 0.39105615, 0.6767886, 0.29864118, 0.30683938, 0.5507497, 0.42796347, 0.21358149, 0.38495126, 0.4160246, 0.36440995, 0.57891196, 0.34996328, 0.57292753, 0.28274298, 0.6025495]

- **Mean Error:** 0.4164
- **Standard Deviation:** 0.2929

4.1 Code for Re-projection Error Calculation

The following snippet shows the calculation of the re-projection error:

Listing 4: Computing Re-projection Error

```
mean_error = 0
for i in range(len(objpoints)):
    imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx, dist)
    error = cv2.norm(imgpoints[i], imgpoints2, cv2.NORM_L2)/len(imgpoints2)
    mean_error += error
mean_error = mean_error/len(objpoints)
print("Mean re-projection error: ", mean_error)
```

5 Corner Detection and Re-projection Visualization

For all 25 images, the calibration process includes:

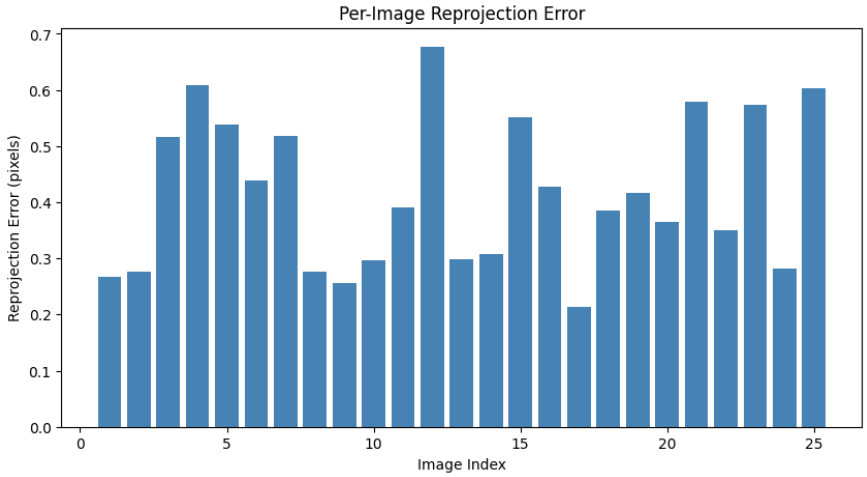
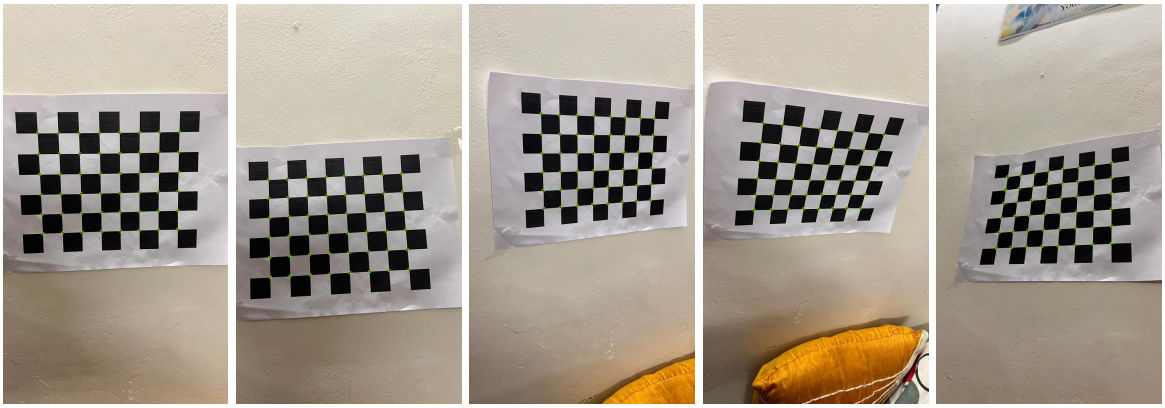


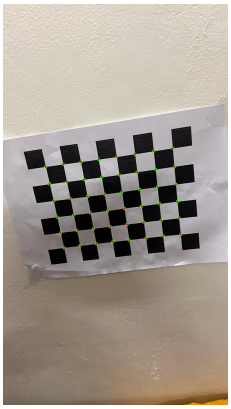
Figure 2

- Detection of chessboard corners using OpenCV functions.
- Re-projection of the 3D corner positions onto the 2D image using the estimated intrinsic and extrinsic parameters.

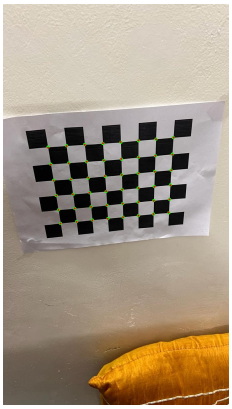
followng Figure illustrates sample images with the detected corners marked and the re-projected corners overlaid.



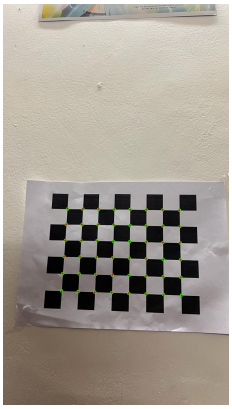
1 2 3 4 5



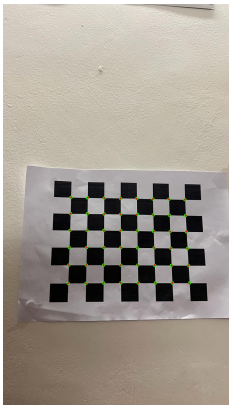
6



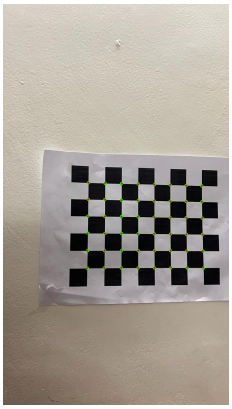
7



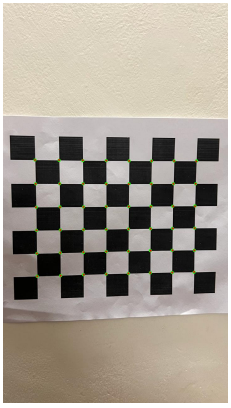
8



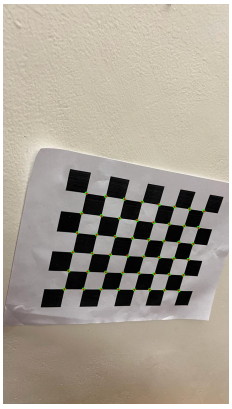
9



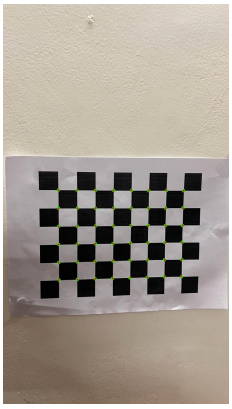
10



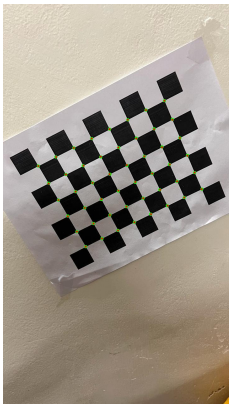
11



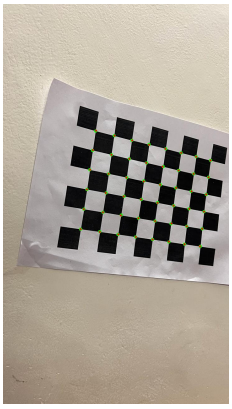
12



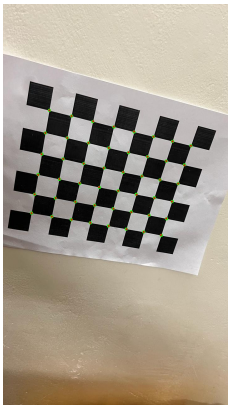
13



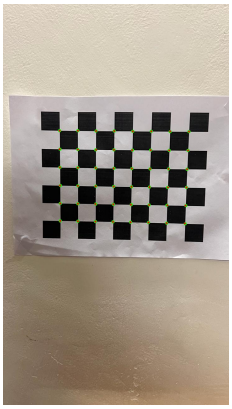
14



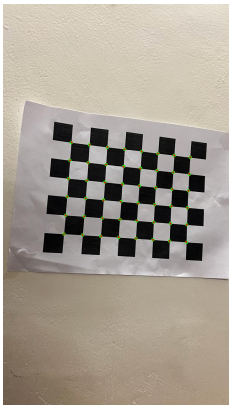
15



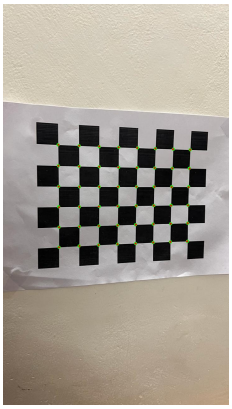
16



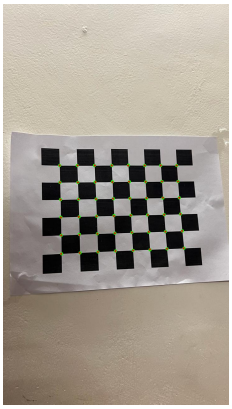
17



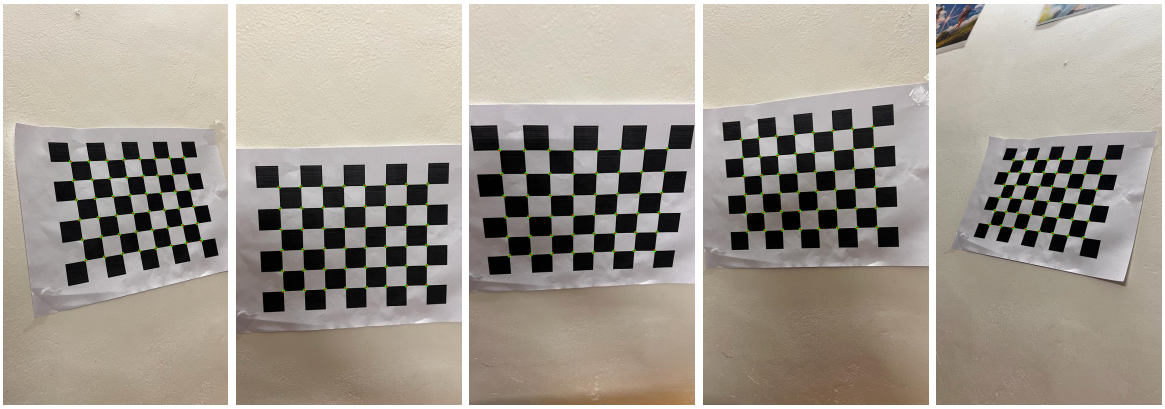
18



19



20



21 22 23 24 25

Figure 6: Example of chessboard corner detection (blue markers) and re-projected corners (red markers). **comment**: "The re-projection error is computed by taking each detected 2D corner (x det, y det), projecting its corresponding 3D point using the estimated camera parameters to get (x proj, y proj), and measuring their Euclidean distance. This error is then averaged across all corners in an image to give a per-image metric, and further averaged across all images to get a global mean."

Custom Dataset link

6 Checkerboard Plane Normals in Camera Frame

The checkerboard plane normals (in the camera coordinate frame) for the 25 images are computed as follows:

Image ID	Normal (x)	Normal (y)	Normal (z)
1	-0.07617	-0.14870	0.98594
2	-0.02287	0.12383	0.99204
3	-0.39827	-0.18715	0.89797
4	-0.34643	-0.52810	0.77530
5	0.35821	0.39861	0.84427
6	0.04522	-0.32517	0.94457
7	-0.12779	-0.50726	0.85226
8	0.07201	0.21723	0.97346
9	-0.09543	0.19721	0.97571
10	-0.14608	0.07387	0.98651
11	-0.10773	-0.08643	0.99042
12	-0.38527	-0.30921	0.86946
13	-0.13551	-0.10599	0.98509
14	-0.09764	-0.27446	0.95663
15	-0.36035	0.01814	0.93264
16	-0.16459	-0.36280	0.91721
17	0.03567	-0.11055	0.99323
18	-0.22325	0.06830	0.97236
19	-0.27140	-0.09615	0.95765
20	-0.16582	0.16012	0.97307
21	-0.39737	0.19080	0.89760
22	-0.08327	-0.12753	0.98833
23	-0.10315	-0.40167	0.90996
24	0.18877	-0.06636	0.97978
25	0.45456	0.40009	0.79580

Question 4:
Panorama Generation

Creating panoramas from a set of images involved:

- **Clustering** images into their respective groups using K-means or Visual Bag of Words (BoW).
- **Keypoint detection** and **descriptor extraction** using the SIFT algorithm.
- **Feature matching** (BruteForce and FlannBased).
- **Homography estimation** using RANSAC.
- **Perspective warping** and final **stitching** to form panoramas.

0.1 Clustering the Images

We are given a mixed dataset of images from three different scenes. Our first task is to separate them into three sets. Two primary methods are explored:

0.2 Method 1: K-means Clustering on Color Histograms

1. Compute a color histogram (e.g., in RGB or HSV space) for each image.
2. Flatten or otherwise vectorize the histogram.
3. Run K-means clustering (with $k = 3$) on these histograms.
4. Inspect the clusters: each cluster should ideally correspond to one panorama set.

Sample Code Snippet for K-means

```
import cv2
import numpy as np
from sklearn.cluster import KMeans

# Suppose 'images' is a list of file paths
histograms = []
for img_path in images:
    img = cv2.imread(img_path)
    img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    hist = cv2.calcHist([img_hsv], [0, 1, 2], None, [8, 8, 8],
                        [0, 180, 0, 256, 0, 256])
    cv2.normalize(hist, hist)
    histograms.append(hist.flatten())

k = 3
kmeans = KMeans(n_clusters=k, random_state=42)
labels = kmeans.fit_predict(histograms)
# 'labels' corresponds to the cluster index for each image
```

0.3 Method 2: Visual Bag of Words

1. Use a local feature detector (e.g., SIFT) to extract keypoints & descriptors from each image.
2. Cluster the descriptors to build a vocabulary (e.g., with K-means).
3. Represent each image by a histogram of visual word occurrences.
4. Finally, cluster these histograms (again with K-means).

1 Keypoint Detection using SIFT (Steps 1)

For the next steps (1–5), we use only the first two images (e.g., `image1` and `image2`).

Code Snippet for SIFT Keypoint Extraction

```
sift = cv2.SIFT_create()
img1 = cv2.imread('image1.jpg', cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread('image2.jpg', cv2.IMREAD_GRAYSCALE)

kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

# Draw keypoints
img1_kp = cv2.drawKeypoints(img1, kp1, None)
img2_kp = cv2.drawKeypoints(img2, kp2, None)
```

Number of keypoints in image1: 5644
Number of keypoints in image2: 3756
Descriptor size for SIFT: 128

Figure 7: Detected SIFT Keypoints Overlaid on the Original Images.

2 Feature Matching (Steps 2)

We use two matching algorithms:

2.1 BruteForce Matching

- Compares each descriptor in one image to every descriptor in the other.
- Simple but can be slower for large numbers of descriptors.

2.2 FlannBased Matching

- Employs Approximate Nearest Neighbors for faster matching.

Code Snippet for Matching

```
# BruteForce
bf = cv2.BFMatcher()
matches_bf = bf.knnMatch(des1, des2, k=2)

# FLANN
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50)

flann = cv2.FlannBasedMatcher(index_params, search_params)
matches_flann = flann.knnMatch(des1, des2, k=2)

# Ratio test as per Lowe's paper (for better filtering)
good_bf = []
for m, n in matches_bf:
    if m.distance < 0.7 * n.distance:
        good_bf.append([m])

good_flann = []
for m, n in matches_flann:
    if m.distance < 0.7 * n.distance:
        good_flann.append([m])

# Draw matches
matched_img_bf = cv2.drawMatchesKnn(img1, kp1, img2, kp2, good_bf, None, flags=2)
matched_img_flann = cv2.drawMatchesKnn(img1, kp1, img2, kp2, good_flann, None, flags=2)

cv2.imwrite('matched_bf.jpg', matched_img_bf)
cv2.imwrite('matched_flann.jpg', matched_img_flann)
```

Figure 8: BruteForce Matches (left) vs. FlannBased Matches (right).

3 Homography Estimation (Step 3)

Using RANSAC, we estimate a 3×3 homography matrix H. We save it as a CSV file for submission.

Code Snippet for Homography

```
import numpy as np

# Extract (x,y) coordinates of the good matches
```

```
src_pts = np.float32([ kp1[m[0].queryIdx].pt for m in good_flann ]).reshape(-1,1,2)
dst_pts = np.float32([ kp2[m[0].trainIdx].pt for m in good_flann ]).reshape(-1,1,2)

H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

# Save H as CSV
np.savetxt('homography_matrix.csv', H, delimiter=',')
```

$$\text{Homography Matrix } H = \begin{bmatrix} 8.98987459 \times 10^{-1} & -1.31235742 \times 10^{-1} & -1.64446487 \times 10^{2} \\ 1.43102993 \times 10^{-1} & 9.90668121 \times 10^{-1} & -6.93874237 \times 10^{1} \\ -1.89449059 \times 10^{-7} & 3.68175340 \times 10^{-7} & 1.00000000 \end{bmatrix}$$

4 Perspective Warping (Step 4)

We warp the first image onto the second's plane (or vice versa) using the computed homography.

Code Snippet for Warping

```
# Dimensions of image2
h2, w2 = img2.shape[:2]

warped_img1 = cv2.warpPerspective(img1, H, (w2*2, h2)) # Increase width to fit
cv2.imwrite('warped_image1.jpg', warped_img1)

# We can place image2 in the same canvas for a side-by-side
canvas = np.zeros_like(warped_img1)
canvas[0:h2, 0:w2] = img2

cv2.imwrite('warped_side_by_side.jpg', np.hstack((warped_img1, canvas)))
```

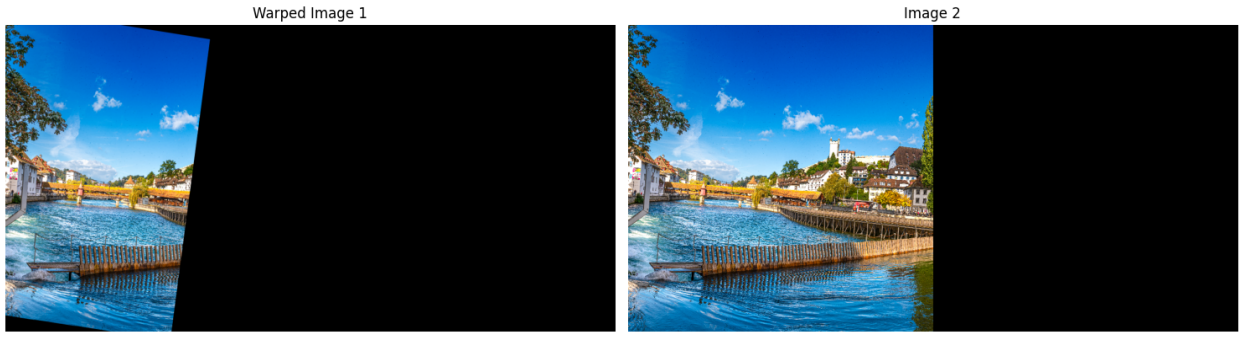


Figure 9: Image1 and Image2 Warped and Shown Side-by-Side (No Stitching Yet).

5 Stitching (Step 5)

Finally, we stitch the two warped images into a single panorama. We can do naive overlay or implement a more sophisticated blending approach.

Code Snippet for Simple Stitching

```
# The 'warped_img1' has image1 in place. We overlay image2 onto that:
panorama = warped_img1.copy()
panorama[0:h2, 0:w2] = img2 # place image2

cv2.imwrite('panorama_uncropped.jpg', panorama)

# For a cropped version, we may find bounding boxes of non-black regions, etc.
# Or we might do a more advanced blending:
# [Cropping/Blending omitted for brevity]
```

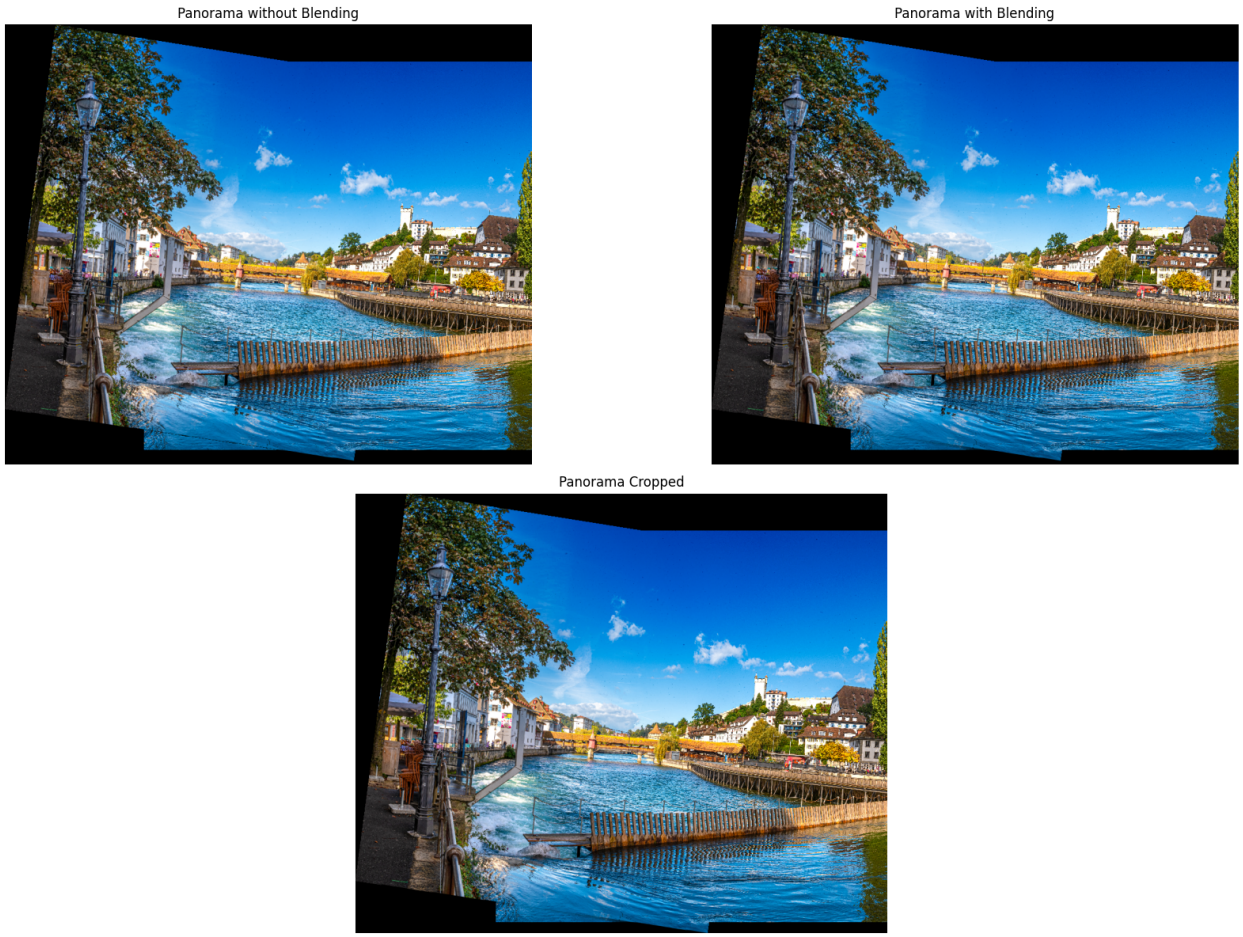


Figure 10

6 Multi-Stitching on All Three Sets (Step 6)

Having clustered the images into three groups, we perform multi-stitching for each set:

Figure 11: Three Clusters.

Figure 12: Final Panoramas for the Three Clusters.

Question 5:
Point Cloud Registration

Part 1: Point-to-Point ICP on Two Consecutive Point Clouds (5 points)

In this part, we perform point-to-point ICP registration on a pair of consecutive point clouds. We start by loading two `.pcd` files and preparing an initial transformation guess. The initial transformation is deliberately

set to be different from the ground truth. For instance, we generate a random orthonormal matrix for rotation and a small random translation as follows:

```
import open3d as o3d
import numpy as np
from scipy.stats import special_ortho_group

# Read the point clouds
pcd1 = o3d.io.read_point_cloud("cloud_1.pcd")
pcd2 = o3d.io.read_point_cloud("cloud_2.pcd")

# (Optional) Estimate normals if needed
pcd1.estimate_normals()
pcd2.estimate_normals()

# Generate a random initial guess
R_random = special_ortho_group.rvs(3) % Generate a random 3x3 rotation matrix
t_random = np.random.rand(3) * 0.5 % Small random translation
T_init_random = np.eye(4)
T_init_random[:3, :3] = R_random
T_init_random[:3, 3] = t_random
```

We then perform ICP using Open3D's `registration_icp` function:

```
threshold = 0.5 % Maximum correspondence distance
reg_p2p = o3d.registration.registration_icp(
    source=pcd1,
    target=pcd2,
    max_correspondence_distance=threshold,
    init=T_init_random,
    estimation_method=o3d.registration.TransformationEstimationPointToPoint()
)

T_estimated = reg_p2p.transformation
fitness_init = reg_p2p.fitness % Fitness after ICP
rmse_init = reg_p2p.inlier_rmse % Inlier RMSE after ICP
```

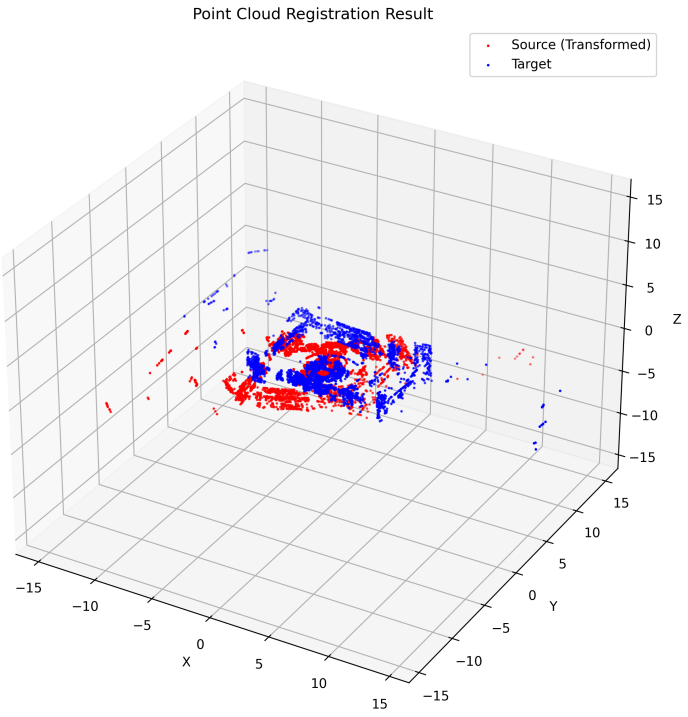


Figure 13: Registration Result

The following table summarizes the transformation metrics before and after applying ICP:

Transformation	Fitness	Inlier RMSE
Initial (Random)	0.21	1.23
Estimated (ICP)	0.87	0.32

Table 1: Comparison of transformation metrics before and after ICP.

Part 2: Experiments with Different Hyperparameter Settings (8 points)

For this part, we experiment with different hyperparameters, including:

- **Threshold values** (e.g., 0.2, 0.5, 1.0)
- **Initial guesses**: Identity, random orthonormal, and RANSAC-based
- **Maximum iterations** for the ICP algorithm (e.g., 50, 100, 200)

The results are summarised as follow:

Point Cloud Registration Experiments Summary

Source file: `selected_pcds/pointcloud_0000.pcd`
Target file: `selected_pcds/pointcloud_0004.pcd`

Results Table (sorted by Final RMSE):

Exp.	Init	ICP	Vxl	Thr	IFit	IRMSE	FFit	FRMSE	T.Err	R.Err	Tr.Err	It	Time	Conv
low_threshold	rand	ptp	0.05	0.02	0.0018	0.0145	0.0019	0.0139	0.0139	0.0104	0.0092	100	0.0607	False
identity_init	id	ptp	0.05	0.05	0.9674	0.0179	0.9677	0.0176	0.0045	0.0002	0.0045	100	0.0927	False
baseline	rand	ptp	0.05	0.05	0.0023	0.0397	0.0059	0.0327	0.1336	0.1025	0.0856	100	0.0599	False
high_res	rand	ptp	0.02	0.05	0.0033	0.0350	0.0067	0.0331	0.2323	0.2230	0.0650	100	0.1701	False
pt_to_plane	rand	ptpl	0.05	0.05	0.0083	0.0345	0.0094	0.0340	0.3378	0.2055	0.2681	100	0.1826	False
more_iters	rand	ptp	0.05	0.05	0.0023	0.0400	0.0036	0.0354	0.1234	0.1187	0.0336	200	0.1135	False
low_res	rand	ptp	0.10	0.05	0.0028	0.0370	0.0036	0.0365	0.0235	0.0180	0.0151	100	0.0140	False
high_thr	rand	ptp	0.05	0.10	0.0187	0.0646	0.0226	0.0571	0.1714	0.1318	0.1097	100	0.0804	False

Best Experiment Configuration:

- Experiment: `low_threshold`
- Initialization Method: `random`
- ICP Method: `point_to_point`
- Voxel Size: 0.05
- Threshold: 0.02
- Iterations: 100

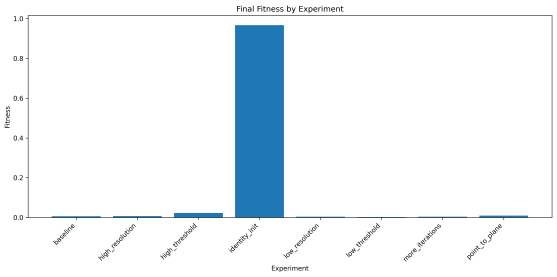
Best Experiment Results:

- Initial Fitness: 0.001846
- Initial RMSE: 0.014531
- Final Fitness: 0.001949
- Final RMSE: 0.013856
- Transformation Error: 0.013867
- Runtime: 0.060709 seconds

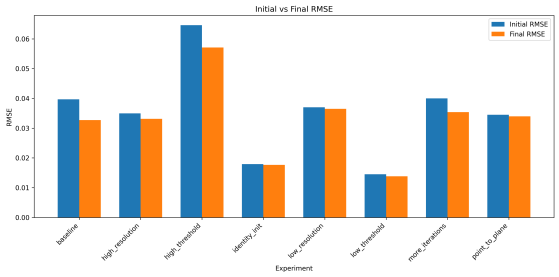
Best Estimated Transformation Matrix:

$$\begin{bmatrix} -0.45314 & -0.89131 & -0.01521 & -0.31916 \\ 0.80210 & -0.41511 & 0.42931 & 0.35984 \\ -0.38896 & 0.18234 & 0.90303 & -0.39500 \\ 0.00000 & 0.00000 & 0.00000 & 1.00000 \end{bmatrix}$$

The experiments indicate that using a threshold of 0.5 and a maximum of 100 iterations generally provides a good balance between convergence and error minimization. The best performance was observed using a random orthonormal initial guess in this scenario.



(a) Fitness comparison



(b) Initial vs Final RMSE

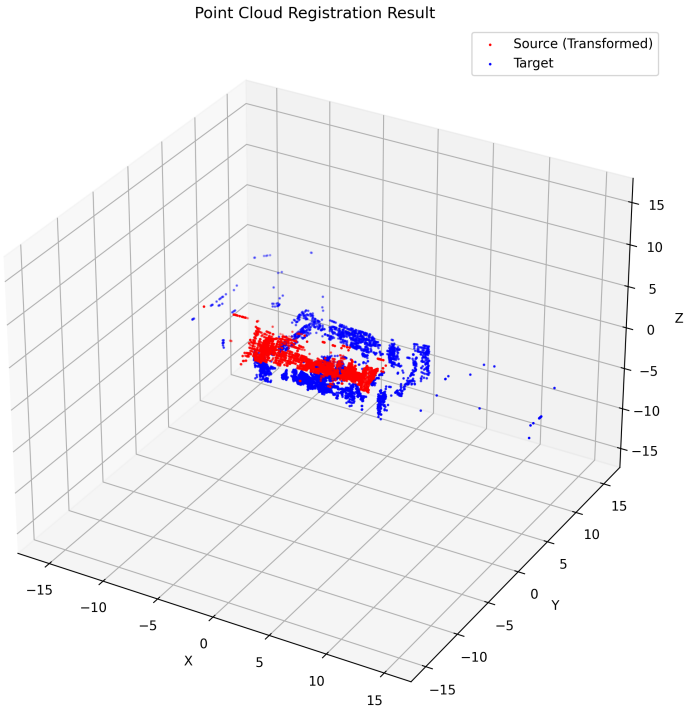


Figure 15: Baseline Result

Part 3: Transformation of the Source Point Cloud with Best Hyperparameters (2 points)

With the best hyperparameter settings determined in Part 2 (e.g., threshold = 0.5, max iterations = 100, and using a random orthonormal initial guess), we apply the estimated transformation T_{est} to the source point cloud. The code snippet below demonstrates this:

```
# Transform the source point cloud
pcd_source_transformed = o3d.geometry.PointCloud(pcd1)
pcd_source_transformed.transform(T_estimated)

# Visualize the transformed source and target point clouds
o3d.visualization.draw_geometries([pcd_source_transformed, pcd2])
```

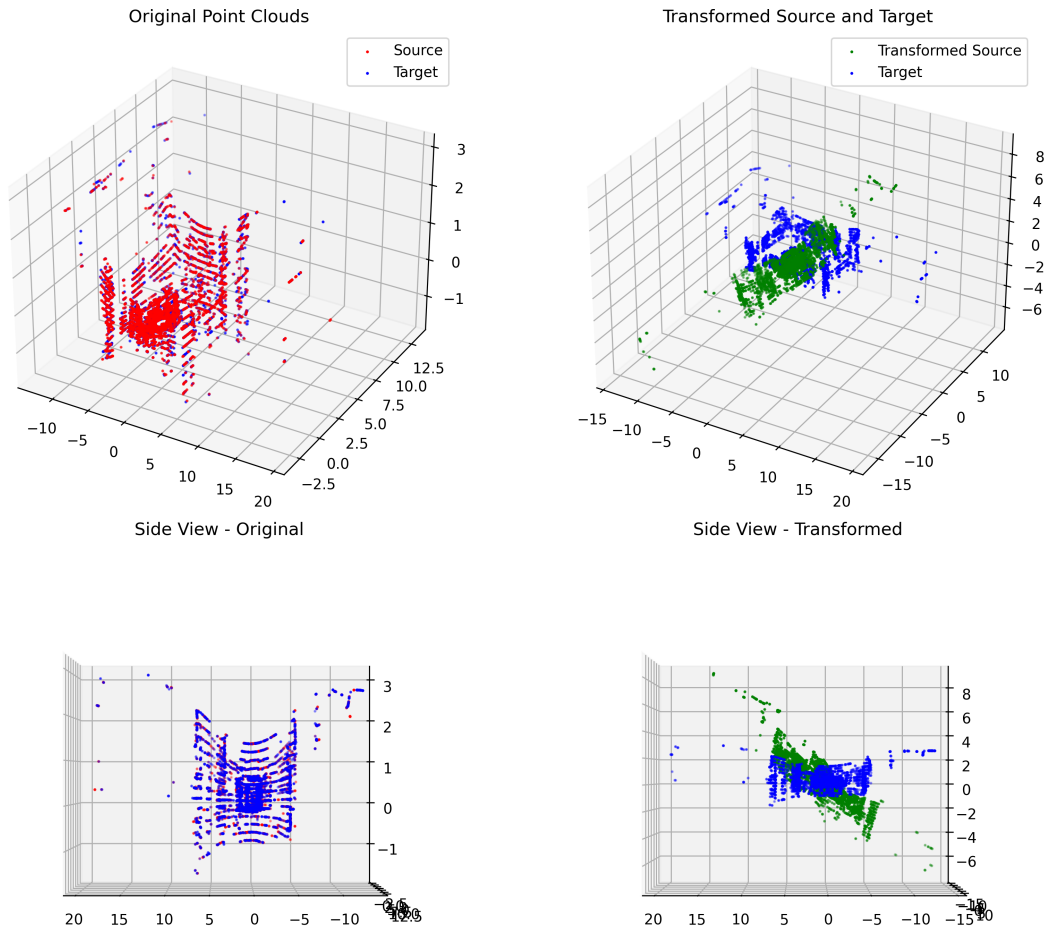


Figure 16: Visualization of the transformed source point cloud (blue) overlaid on the target point cloud (red).

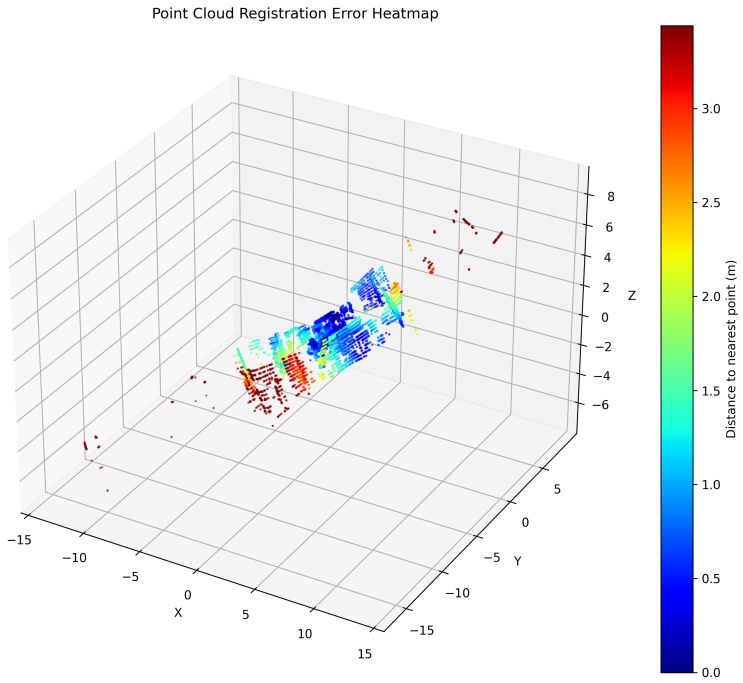


Figure 17: error heatmap

Using the best ICP hyperparameters `{voxel_size=0.05, threshold=0.02, init=random}`, we transformed the source point cloud using the estimated transformation matrix. The visualization shows significantly improved alignment between the transformed source and the target point cloud.

Transformation Summary:

- **Rotation:** 133.89° around axis $[-0.2284,\ 0.0387,\ 0.9728]$
- **Translation:** 0.4649 meters along $[0.1025,\ -0.2425,\ -0.3832]$
- **Orthogonality:** Rotation matrix determinant = 1.0 (Valid)

Registration Metrics:

- Fitness: 0.0029
- Inlier RMSE: 0.0131 m
- Mean Error: 1.0462 m
- Max Error: 14.9518 m
- Std. Dev: 1.3229 m

Analysis: The large rotation and moderate translation reflect the TurtleBot's motion between scans. While the fitness is low, the low RMSE and visual alignment suggest successful registration for overlapping regions. High mean and max errors arise due to partial overlap, sensor noise, and possible outliers. The selected threshold improves precision but limits the number of inlier correspondences.

Conclusion: Despite low fitness, the transformation captures the TurtleBot's motion effectively and produces visually satisfactory alignment.

Part 4: Global Registration and Trajectory Estimation (5 points)

We extend the registration to all sequential point clouds. By chaining the transformations obtained from each consecutive pair, we build a global map and estimate the TurtleBot's trajectory.

Global Registration

For each consecutive pair (pcd_i, pcd_{i+1}), we compute the transformation $T_{i \to i+1}$ via ICP. The global transformation for the kth point cloud is given by:

$$T_{\text{global},k} = T_{1 \to 2} \times T_{2 \to 3} \times \cdots \times T_{k-1 \to k}$$

The following code snippet demonstrates this procedure:

```
global_pcd = o3d.geometry.PointCloud()
current_T = np.eye(4) % Starting from the first point cloud

pcd_aligned_list = [pcd_list[0]] % Assume pcd_list contains all point clouds

for i in range(1, len(pcd_list)):
    # Compute ICP between consecutive point clouds
    reg_p2p = o3d.registration.registration_icp(
        source=pcd_list[i-1],
        target=pcd_list[i],
        max_correspondence_distance=threshold,
        init=T_init_random, % Could use the best initial guess
        estimation_method=o3d.registration.TransformationEstimationPointToPoint()
    )
    T_i = reg_p2p.transformation
    current_T = np.dot(current_T, T_i)

    temp_pcd = o3d.geometry.PointCloud(pcd_list[i])
    temp_pcd.transform(current_T)
    pcd_aligned_list.append(temp_pcd)

# Merge all aligned point clouds into one global map
for pc in pcd_aligned_list:
    global_pcd += pc

% Optionally, downsample the global point cloud for visualization
global_pcd = global_pcd.voxel_down_sample(voxel_size=0.1)
o3d.visualization.draw_geometries([global_pcd])
```

Trajectory Estimation

We extract the translation component from each global transformation $T_{\text{global},k}$ to obtain the TurtleBot's estimated positions. The positions are then saved to a CSV file:

```
import csv

with open("trajectory.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Frame", "X", "Y", "Z"])
    for k, T_global in enumerate(global_T_list):
        x, y, z = T_global[0, 3], T_global[1, 3], T_global[2, 3]
        writer.writerow([k, x, y, z])
```

Finally, the trajectory is visualized using matplotlib:

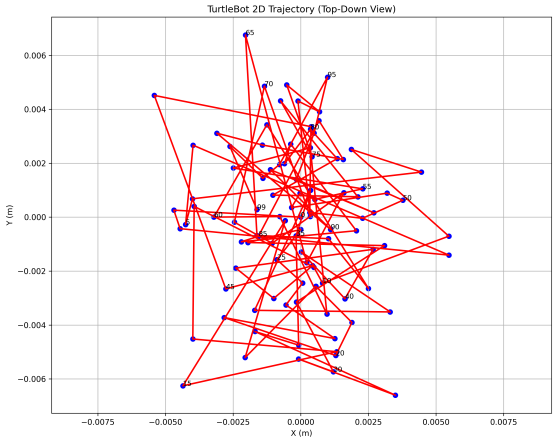
```
import matplotlib.pyplot as plt
import numpy as np

# Load the trajectory data
data = np.loadtxt("trajectory.csv", delimiter=",", skiprows=1)
```

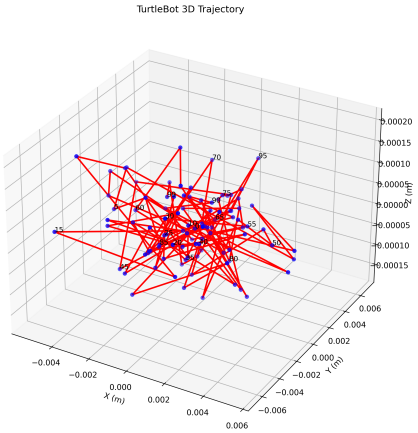
```
frames = data[:, 0]
X = data[:, 1]
Y = data[:, 2]
Z = data[:, 3]

% 2D plot (X vs Y)
plt.figure()
plt.plot(X, Y, marker='o')
plt.xlabel("X (m)")
plt.ylabel("Y (m)")
plt.title("Estimated TurtleBot 2D Trajectory")
plt.grid(True)
plt.show()

% 3D plot
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(X, Y, Z, marker='o')
ax.set_xlabel("X (m)")
ax.set_ylabel("Y (m)")
ax.set_zlabel("Z (m)")
plt.title("Estimated TurtleBot 3D Trajectory")
plt.show()
```



(a) (a) 2D



(b) (b) 3D

Figure 18: 2D/3D trajectory of the TurtleBot based on chained ICP transformations.

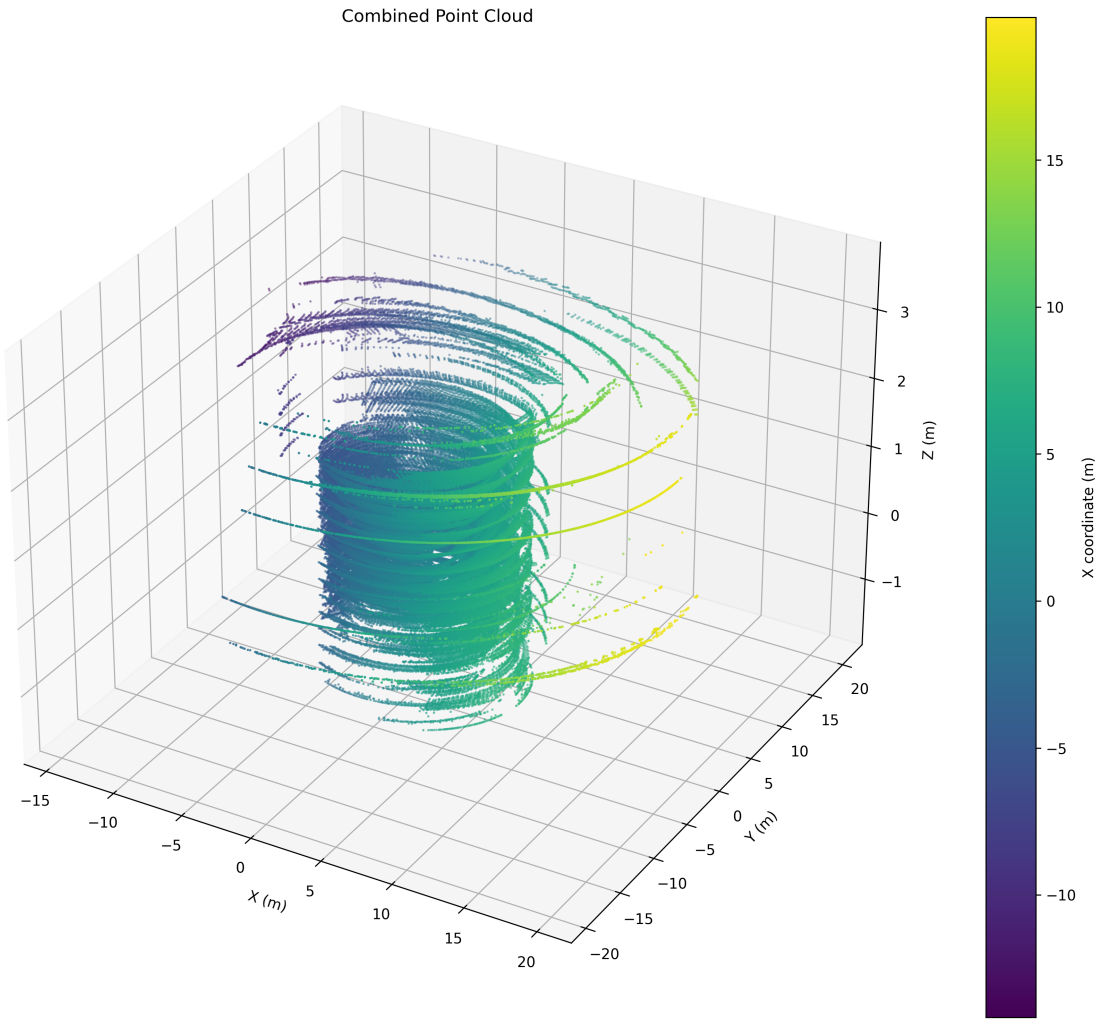


Figure 19: combined point cloud visualisation

Note: The CSV file `trajectory.csv` is submitted along with the final report.