CHONNAM NATIONAL UNIVERSITY

School of Electronics & Computer Engineering

3D VISION

Homework 4 Report

Programming Computer Vision with Python

3D RECONSTRUCTION

Student:
Tran Nguyen Quynh Tram - 198507

Professor:
이 칠 우

November 30, 2019

1 Problem Description

First, take two images with your smartphone as described in the lecture, then obtain the F-matrix and restore the three-dimensional information of the two images.

2 Problem Solution

2.1 Introduction

Multiple view geometry is the field studying the relationship between cameras and features when there are correspondences between many images that are taken from varying viewpoints. The most important constellation is **two-view geometry**.

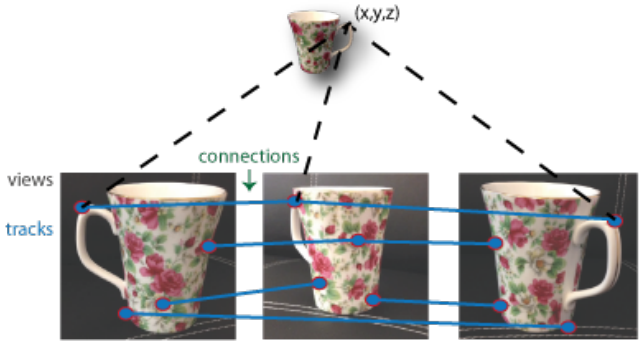

Figure 1: Multiple view geometry recovery based on structure from motion

With two views of a scene and corresponding points in these views there are geometric constraints on the image points as a result of the relative orientation of the cameras, the properties of the cameras, and the position of the 3D points. These geometric relationships are described by what is called **epipolar geometry**.

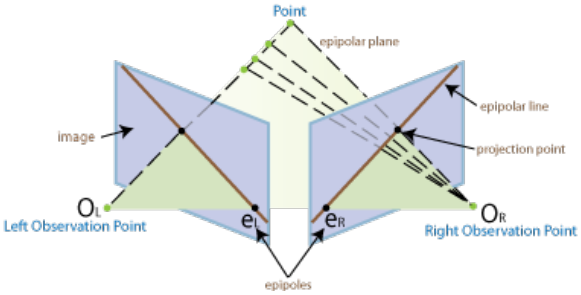

Figure 2: Epipolar Geometry

This problem in the exercise is that given two images from the camera of my mobile-phone, we need to build the 3D information of two images.

2.2 Proposed method [1]

Our 3D reconstruction recovery algorithm consists of following steps:

1. Calibrating our camera used to take two images. From there, we determine camera matrix parameters known as calibration matrix K.
2. Find the local features of two images using SIFT [2], and applying brute-force matching to detect the corresponding feature points between two images.

3. Applying the 8-points algorithm integrated in RANSAC to detect Fundamental matrix with input data of the corresponding feature points between two images. From there, we check again by visualizing the epilines of two images.
4. Calculating the essential matrix and recovering pose between two cameras.
5. Calculating 3D points based on triangulate points.

3 Step 1: Camera Calibration

3.1 Introduction

Pinhole Camera Model:

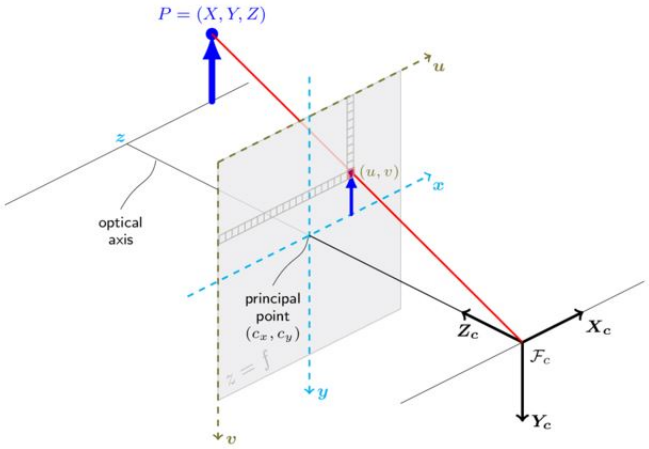

Figure 3: Pinhole Camera Model

Perspective projection: $x = f\left(\frac{X}{Z}\right), \quad y = f\left(\frac{Y}{Z}\right)$

$$x = \begin{bmatrix} x \\ y \end{bmatrix} = \frac{f}{Z}\begin{bmatrix} X \\ Y \end{bmatrix}$$

$$Z\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$Zx = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} X$$

$$\lambda x' = K\Pi_0 X = \begin{bmatrix} fs_x & fs_\theta & o_x \\ 0 & fs_y & o_y \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = K\Pi_0 g X_0$$

$$= \begin{bmatrix} fs_x & fs_\theta & o_x \\ 0 & fs_y & o_y \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

o_x: x-coordinate of the principal point in pixels,
o_y: y-coordinate of the principal point in pixels,
$fs_x = \alpha_x$: size of unit length in horizontal pixels,
$fs_y = \alpha_y$: size of unit length in vertical pixels,
α_x/α_y: aspect ratio σ,
fs_θ: skew of the pixel, often close to zero.

Figure 4: Camera Model Equation

Camera Calibration

- Unknown: Intrinsic + m x extrinsic parameters (5* + m x 6 DoF)
- Given: 3D points $\mathbf{X}_1$, $\mathbf{X}_2$, ..., $\mathbf{X}_n$ and their projected points from jth camera $\mathbf{x}_i^j$
- Constraints: n x m x projection $\mathbf{x}_i^j = \mathrm{K}\left[\,\mathrm{R}_j|\,\mathbf{t}_j\,\right]\mathbf{X}_i$
- Solutions
  - OpenCV `cv::calibrateCamera()` and `cv::initCameraMatrix2D()`
  - Camera Calibration Toolbox for MATLAB, http://www.vision.caltech.edu/bouguetj/calib_doc/
  - GML C++ Camera Calibration Toolbox, http://graphics.cs.msu.ru/en/node/909
  - DarkCamCalibrator, http://darkpgmr.tistory.com/139

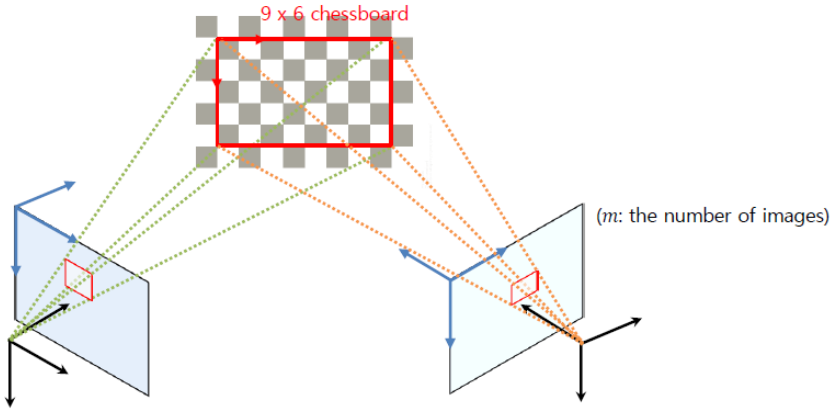

Figure 5: Camera Calibration

3.2 Results

We print 8x8 chessboard from OpenCV library, and stick it on the carton paper. After that, we take 20 photos using Note8, and apply calibration function of OpenCV library.

```
        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(gray, pattern_size, None)

        # If found, add object points, image points (after refining them)
        if ret == True:
            corners = np.squeeze(corners)
            corners2 = cv2.cornerSubPix(gray, corners, (pattern_size[0]//2, pattern_size[1]// 2), (-1,-1),criteria)
            scale_corners = cv2.cornerSubPix(org_gray, corners2 * scale, (pattern_size[0]//2, pattern_size[1]// 2), (-1,-1),crit
eria)

            # Draw and display the corners
            ret_image = drawChessboardCorners(org_image, pattern_size, scale_corners, ret, is_show = False)

            images.append(org_image)
            obj_points.append(objp)
            img_points.append(scale_corners)

            plt.subplot(1, cols, c + 1), plt.imshow(ret_image[...,::-1])
            plt.title(f"Image {idx_image}")
            plt.axis("off")
```

Figure 6: Detect chessboard

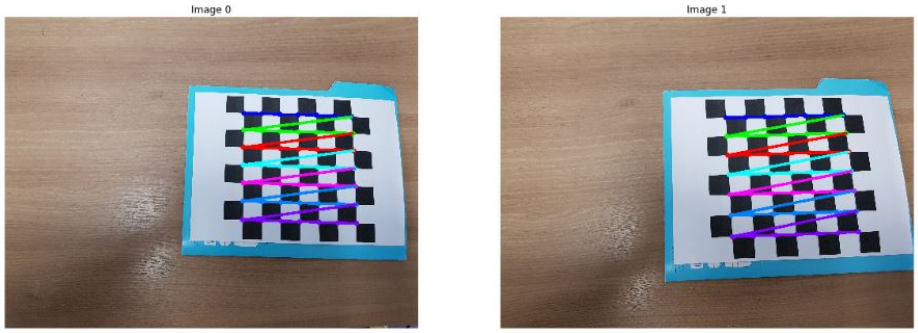

Figure 7: Detect chessboard results

We make two times camera calibration. First time, we calibrate on all images. After that, we remove outlier images to focus on inlier images for processing the second calibration.

```
In [9]: # calculate camera distortion
        rms, camera_matrix, dist_coefs, _rvecs, _tvecs = cv2.calibrateCamera(obj_points, img_points,
                                                                             (org_width, org_height), None, None)

In [10]: print("\nRMS:", rms)
         print("camera matrix:\n", camera_matrix)
         print("distortion coefficients: ", dist_coefs.ravel())

RMS: 1.1175393693050937
camera matrix:
 [[3.55790612e+03 0.00000000e+00 2.05991926e+03]
 [0.00000000e+00 3.45462241e+03 1.49734905e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
distortion coefficients:  [ 0.08791635 -0.26319236  0.01067034 -0.00325333 -1.33233556]

In [11]: obj_points_test = obj_points # obj_points_filter
         img_points_test = img_points # img_points_filter

         mean_error = 0
         for i in range(len(obj_points_test)):
             img_points2, _ = cv2.projectPoints(obj_points_test[i], _rvecs[i], _tvecs[i], camera_matrix, dist_coefs)
             img_points2    = np.squeeze(img_points2)
             error = cv2.norm(img_points_test[i],img_points2, cv2.NORM_L2)/len(img_points2)
             mean_error += error
         # for
         print("total error: ", mean_error/len(obj_points_test))

total error:  0.15705153472766437
```

Figure 8: The first calibration

```
In [13]: # calculate camera distortion
         rms, camera_matrix, dist_coefs, _rvecs, _tvecs = cv2.calibrateCamera(obj_points_filter, img_points_filter,
                                                                              (org_width, org_height), None, None)

In [14]: print("\nRMS:", rms)
         print("camera matrix:\n", camera_matrix)
         print("distortion coefficients: ", dist_coefs.ravel())

RMS: 1.1109043547739383
camera matrix:
 [[3.79408157e+03 0.00000000e+00 2.06101782e+03]
 [0.00000000e+00 3.68800639e+03 1.49115819e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
distortion coefficients:  [ 0.10490643 -0.37527144  0.01191867 -0.00377387 -1.86075126]

In [15]: obj_points_test = obj_points_filter
         img_points_test = img_points_filter

         mean_error = 0
         for i in range(len(obj_points_test)):
             img_points2, _ = cv2.projectPoints(obj_points_test[i], _rvecs[i], _tvecs[i], camera_matrix, dist_coefs)
             img_points2    = np.squeeze(img_points2)
             error = cv2.norm(img_points_test[i],img_points2, cv2.NORM_L2)/len(img_points2)
             mean_error += error
         # for
         print("total error: ", mean_error/len(obj_points_test))

total error:  0.15603712482212448
```

Figure 9: The second calibration

4 Step 2: Find corresponding features between two images

4.1 Feature Extraction

We use SIFT [2] for extraction the keypoints from two images captured by our mobile-phone.

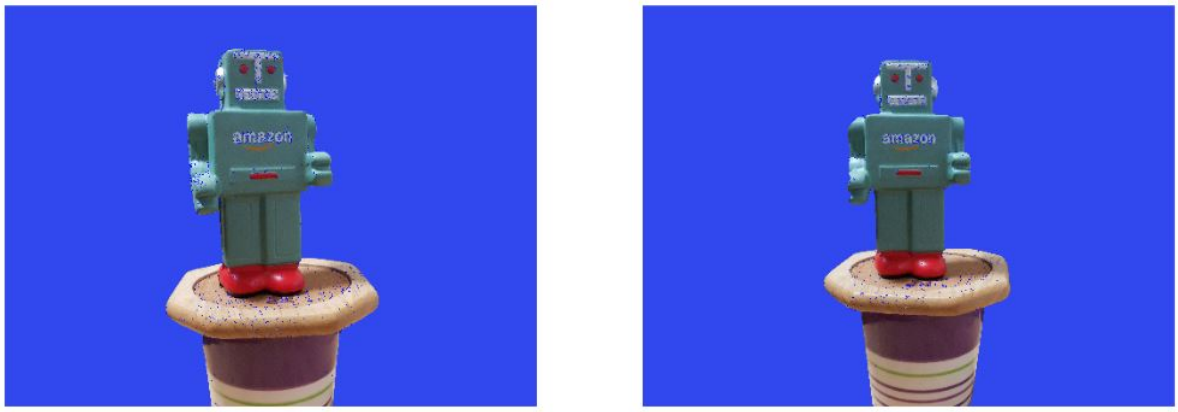

Figure 10: Feature Extraction

4.2 Feature Matching

We use the Brute-force descriptor matcher from OpenCV library. It means that for each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one. This descriptor matcher supports masking permissible matches of descriptor sets. To enhance matching performance, we apply two-side matching.

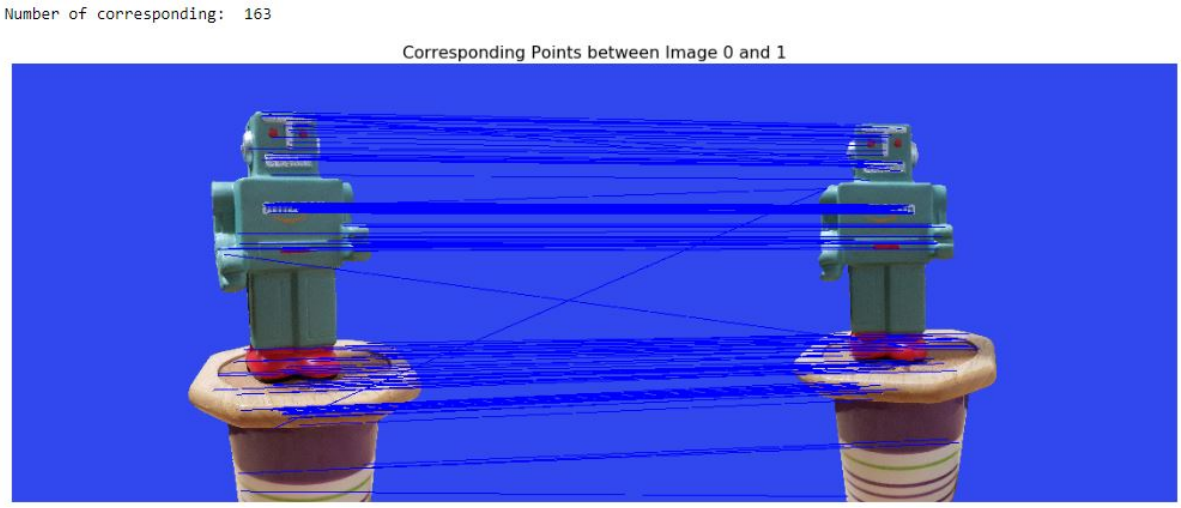

Figure 11: Corresponding feature points between two images

5 Step 3: Find Fundamental Matrix

5.1 Introduction

Assume we have matched points x⇔x' with outliers

8-Point Algorithm for Recovering F

- Correspondence Relation

$$\mathbf{x}'^T\mathbf{F}\mathbf{x} = 0$$

1. Normalize image coordinates

$$\tilde{\mathbf{x}} = \mathbf{T}\mathbf{x} \quad \tilde{\mathbf{x}}' = \mathbf{T}'\mathbf{x}'$$

2. RANSAC with 8 points
   - Randomly sample 8 points
   - Compute F via least squares
   - Enforce $\det(\tilde{\mathbf{F}}) = 0$ by SVD
   - Repeat and choose F with most inliers
3. De-normalize: $\mathbf{F} = \mathbf{T}'^T\tilde{\mathbf{F}}\mathbf{T}$

Figure 12: Eight-point algorithm to recovery fundamental matrix

5.2 Results

In here, we use the function from OpenCV to find fundamental matrix. To enhance the error of algorithm, we apply two times. First time, we run the algorithm to detect fundamental matrix and inliers. After that, we only use the corresponding inliers to run again the algorithm.

```
# https://stackoverflow.com/questions/31737688/epipolar-geometry-pose

# calculate fundamentral matrix
F1, mask = cv2.findFundamentalMat(src_pts, dst_pts, cv2.FM_RANSAC)
print("Fundamental Matrix 1: \n", F1)
```

```
Fundamental Matrix 1:
 [[ 4.50428613e-08  7.73193510e-07 -1.46574465e-03]
 [-6.23802580e-07  2.18077360e-08 -1.71273250e-03]
 [ 1.11896945e-03  9.71023499e-04  1.00000000e+00]]
```

```
mean_error = 0
for i in range(len(src_pts)):
    pt1 = np.array([[src_pts[i][0]], [src_pts[i][1]], [1]])
    pt2 = np.array([[dst_pts[i][0], dst_pts[i][1], 1]])

    mean_error = mean_error + np.dot(np.dot(pt2, F1),pt1)
# for
mean_error = mean_error / len(src_pts)
print("Mean Error Fundamental Matrix: ", float(abs(mean_error)))
```

```
Mean Error Fundamental Matrix:  0.0017200513445076746
```

Figure 13: First time for calculating fundamental matrix

Enhance Fundamental Matrix Again with Inliners

```
# Selecting only the inliers
src_pts1 = src_pts[mask.ravel()==1]
dst_pts1 = dst_pts[mask.ravel()==1]
```

```
F2, mask = cv2.findFundamentalMat(src_pts1, dst_pts1, cv2.FM_8POINT)
print("Fundamental Matrix 2: \n", F2)
```

```
Fundamental Matrix 2:
 [[ 3.28726210e-08  6.28633476e-07 -1.11853747e-03]
 [-4.45601675e-07  1.03360996e-08 -2.55527848e-03]
 [ 8.07348845e-04  1.72912356e-03  1.00000000e+00]]
```

```
mean_error = 0
for i in range(len(src_pts1)):
    pt1 = np.array([[src_pts1[i][0]], [src_pts1[i][1]], [1]])
    pt2 = np.array([[dst_pts1[i][0], dst_pts1[i][1], 1]])

    mean_error = mean_error + np.dot(np.dot(pt2, F2),pt1)
# for
mean_error = mean_error / len(src_pts1)
print("Mean Error Fundamental Matrix: ", float(abs(mean_error)))
```

```
Mean Error Fundamental Matrix:  6.974256743896157e-05
```

```
# Selecting only the inliers
src_pts2 = src_pts1[mask.ravel()==1]
dst_pts2 = dst_pts1[mask.ravel()==1]
```

Figure 14: Second time for calculating fundamental matrix

We draw epipolar lines for checking results.

```
# Find epilines corresponding to points in right image (second image) and
# drawing its lines on left image
lines1 = cv2.computeCorrespondEpilines(dst_pts2.reshape(-1, 1, 2), 2, F2)
lines1 = lines1.reshape(-1, 3)
img5, img6 = drawlines(images[0].copy(), images[1].copy(), lines1, dst_pts2, src_pts2)

# Find epilines corresponding to points in left image (first image) and
# drawing its lines on right image
lines2 = cv2.computeCorrespondEpilines(src_pts2.reshape(-1, 1, 2), 1, F2)
lines2 = lines2.reshape(-1,3)
img3, img4 = drawlines(images[1].copy(), images[0].copy(), lines2, src_pts2, dst_pts2)

plt.figure(figsize=(20,10))
plt.subplot(121),plt.imshow(img5[...,::-1]), plt.axis("off")
plt.subplot(122),plt.imshow(img3[...,::-1]), plt.axis("off")
plt.show()
```

Figure 15: Epipolar lines

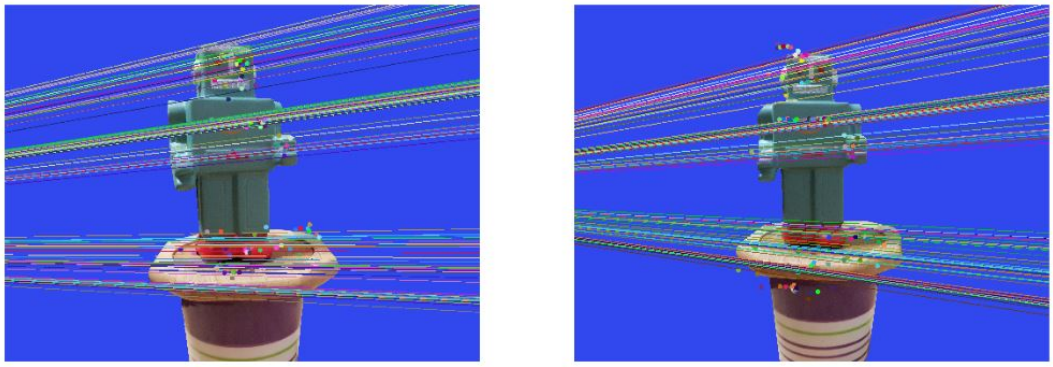

Figure 16: Visualize of epipolar lines

6 Step 4: Find Essential Matrix and recover pose

6.1 Introduction

Relative Camera Pose Estimation (~ Fundamental/Essential Matrix Estimation)

- Unknown: Rotation and translation R, **t** (5 DoF; up-to scale "scale ambiguity")
- Given: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1)$, ..., $(\mathbf{x}_n, \mathbf{x}'_n)$ and camera matrices K, K'
- Constraints: n x epipolar constraint ($\mathbf{x}'^{\mathsf{T}}\mathbf{F}\mathbf{x} = 0$ or $\hat{\mathbf{x}}'^{\mathsf{T}}\mathbf{E}\hat{\mathbf{x}} = 0$)
- Solutions (OpenCV)
  - **Fundamental matrix**: 7/8-point algorithm (7 DoF)
    - **Estimation**: `cv::findFundamentalMat()` → 1 solution
    - **Conversion to E**: $\mathrm{E} = \mathrm{K}'^{\mathsf{T}}\mathrm{F}\mathrm{K}$
    - **Decomposition with positive-depth check**: `cv::recoverPose()`

F
$\mathrm{E} = \mathrm{K}'^{\mathsf{T}}\mathrm{F}\mathrm{K}$
E
$\mathrm{E} = [\mathbf{t}]_{\times}\mathrm{R}$
R, **t**

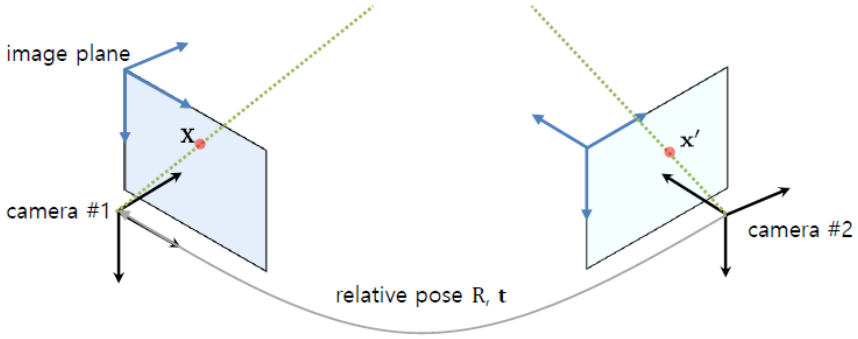

Figure 17: Recovery Pose Estimation

6.2 Results

```
# Load camera information matrix
camera_info = dict(np.load("calibrate1.npz"))
K = camera_info["camera_matrix"] # for camera matrix of camera 1, 2
D = camera_info["dist_coefs"] # for distortion coeffs of cameras
```

```
## Estimate relative pose of two cameras
E2 = K.T.dot(F2).dot(K) # calculate essential matrix from fundamental matrix and calibration matrix
_, R, t, _ = cv2.recoverPose(E2, src_pts2, dst_pts2) # camera pose
```

Figure 18: Recovery Pose Result

7 Step 5: Build 3D Points

7.1 Introduction

Triangulation (Point Localization)

- Unknown: Position of a 3D point **X** (3 DoF)
- Given: Point correspondence $(\mathbf{x}, \mathbf{x}')$, camera matrices $(\mathrm{K}, \mathrm{K}')$, and relative pose $(\mathrm{R}, \mathbf{t})$
- Constraints: 2 x projection $\mathbf{x} = \mathrm{K}[\,\mathrm{I} \mid \mathbf{0}\,]\,\mathbf{X}$, $\mathbf{x}' = \mathrm{K}'[\,\mathrm{R} \mid \mathbf{t}\,]\,\mathbf{X}$
- Solution (OpenCV): `cv::triangulatePoints()`

Special case) Stereo cameras

$$\mathrm{R} = \mathrm{I},\ \mathbf{t} = \begin{bmatrix} -b \\ 0 \\ 0 \end{bmatrix},\ \mathrm{K} = \mathrm{K}' = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\therefore Z = \frac{f}{x - x'} b$$

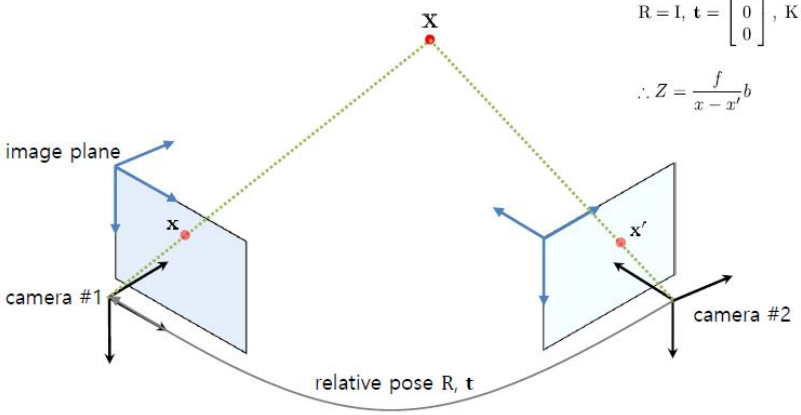

Figure 19: Triangulation

7.2 Results

```
## Calculate camera projection
camera_project_1 = np.matmul(K, np.eye(3, 4, dtype=np.float))

Rt = np.hstack([R, t])
camera_project_2 = np.matmul(K, Rt)

print("Camera Projection 1: \n", camera_project_1)
print("Camera Projection 2: \n", camera_project_2)
```

```
Camera Projection 1:
 [[3.79408157e+03 0.00000000e+00 2.06101782e+03 0.00000000e+00]
 [0.00000000e+00 3.68800639e+03 1.49115819e+03 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00 0.00000000e+00]]
Camera Projection 2:
 [[ 3.27842945e+03 -1.14381693e+02  2.80743060e+03 -1.71336377e+03]
 [-1.31670436e+02  3.72120896e+03  1.40007522e+03  9.91453162e+02]
 [-2.07807689e-01  2.54618706e-02  9.77838257e-01  6.16977345e-01]]
```

```
X = cv2.triangulatePoints(camera_project_1, camera_project_2, src_pts2.reshape(-1, 1, 2), dst_pts2.reshape(-1, 1, 2)).T
```

```
X[:, 0] = X[:, 0] / X[:, 3]
X[:, 1] = X[:, 1] / X[:, 3]
X[:, 2] = X[:, 2] / X[:, 3]
X[:, 3] = X[:, 3] / X[:, 3]
```

```
with open(params["pointcloud_files"], "wt") as f:
    for xx in X:
        f.writelines(f"{xx[0]} {xx[1]} {xx[2]}\n")
    # for
# with
```

Figure 20: Triangulation Code

8 View 3D Point Demo

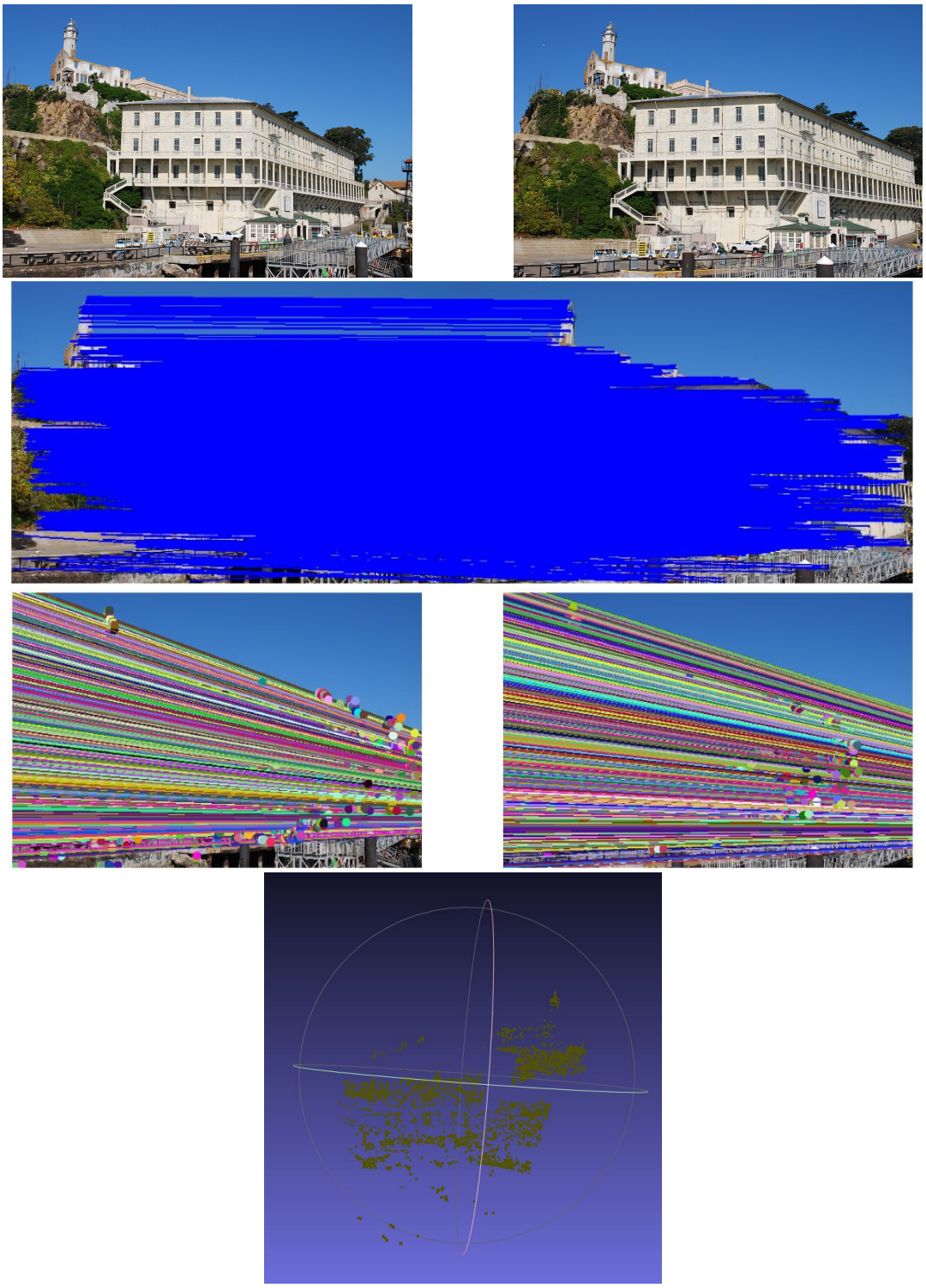

Figure 21: Alcatraz image

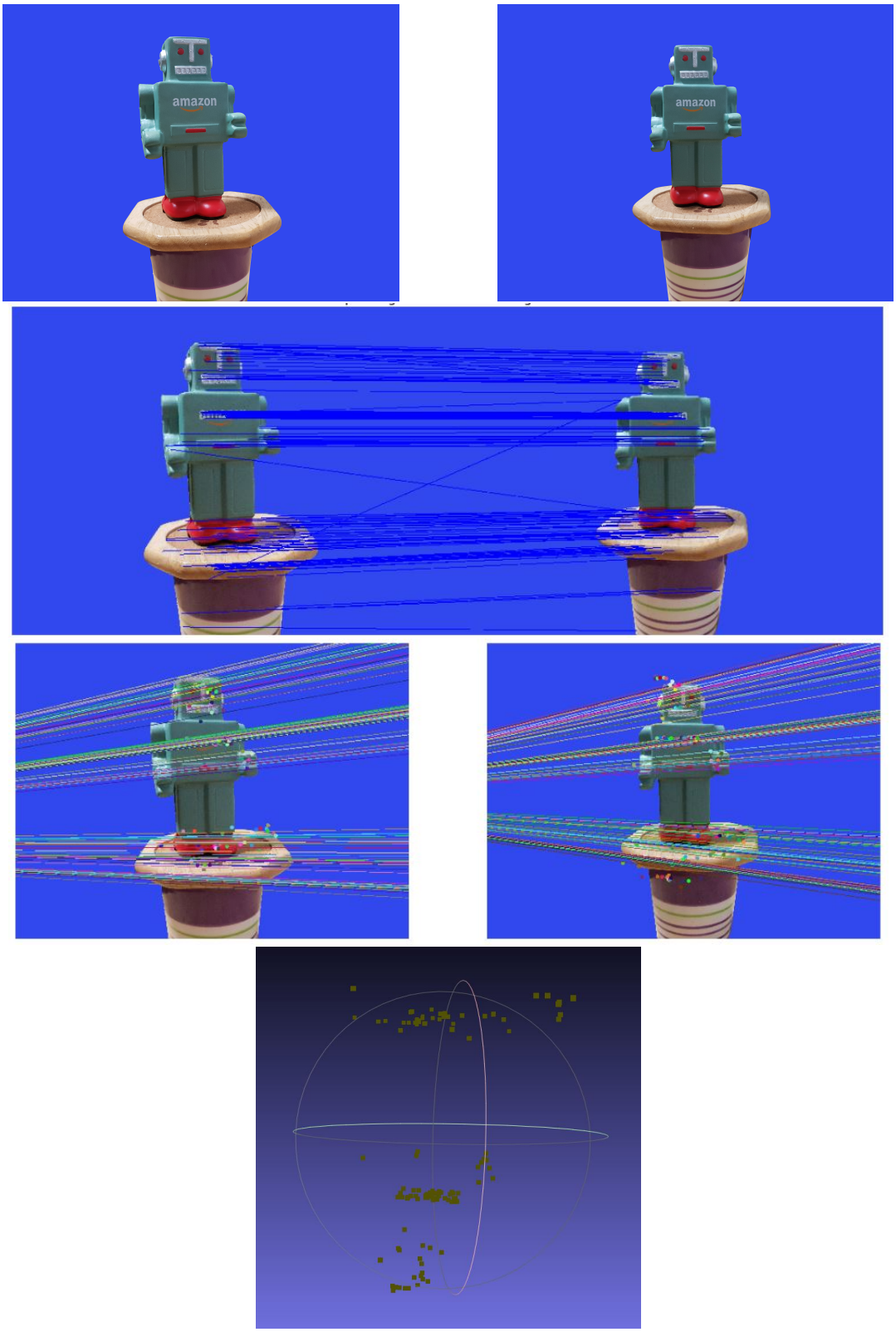

Figure 22: Test1 image

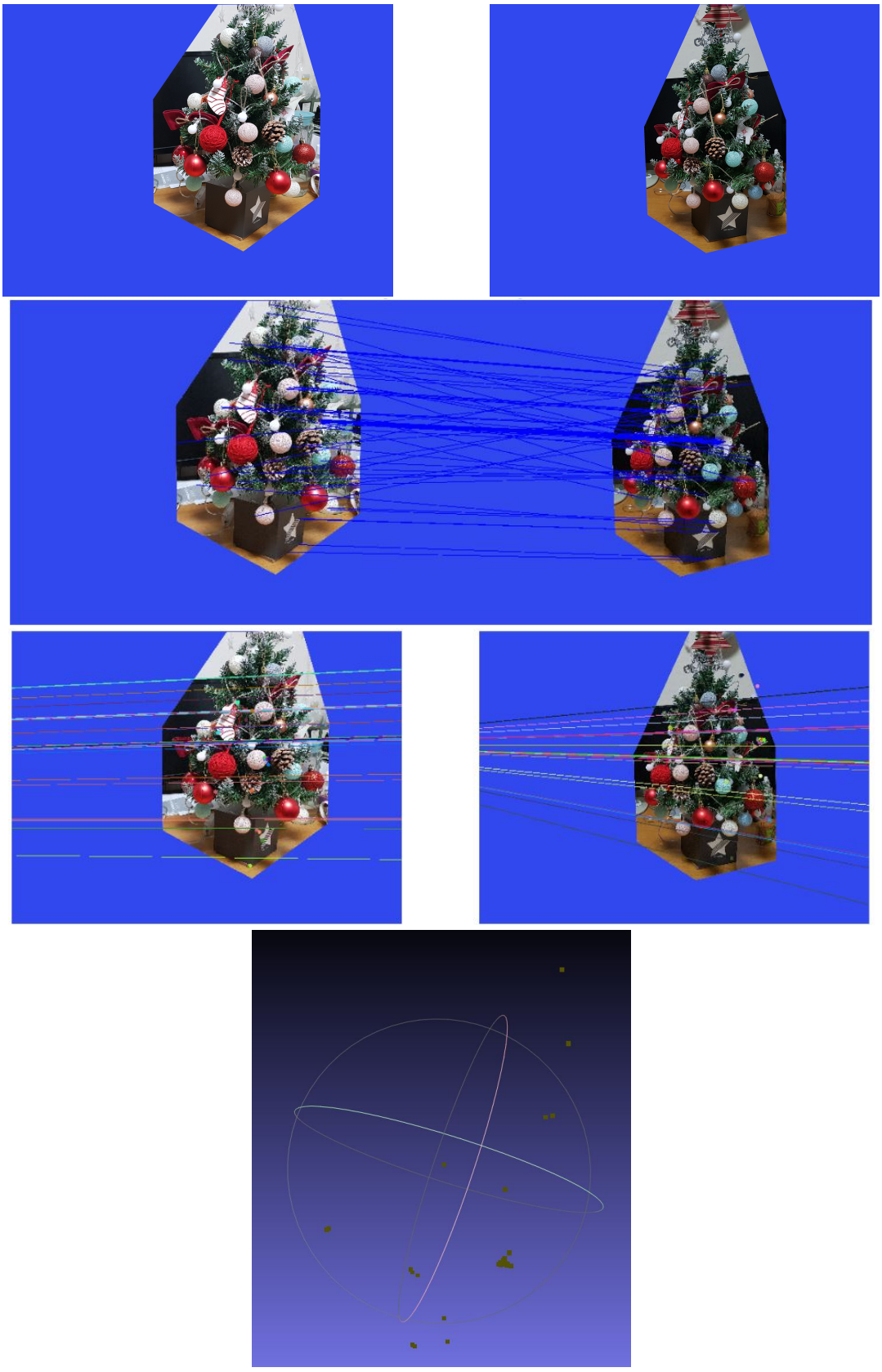

Figure 23: Test1 image

References

[1] S. Choi, "An invitation to 3d vision: A tutorial for every one," 2018.

[2] T. Lindeberg, "Scale Invariant Feature Transform," *Scholarpedia*, vol. 7, no. 5, p. 10491, 2012.