ECE 66100 Computer Vision **Name:** *Nikita Ravi (ravi30)*
Homework 8
11/14/2022

1 Theoretical Questions

1. In Lecture 20, we showed that the image of the Absolute Conic Ω_∞ is given by $\omega = K^{-T}K^{-1}$. As you know, the Absolute Conic resides in the plane π_∞ at infinity. Does the derivation we went through in Lecture 20 mean that you can actually see ω in a camera image? Give reasons for both 'yes' and 'no' answers. Also, explain in your own words the role played by this result in camera calibration.

   Answer: **NO** because we cannot see ω since all the pixels of the absolute conic are imaginary, in other words $K^{-T}K^{-1}$ is positive definite. In the derivation of ω, the rotation and translation matrix remains independent for the absolute conic, making the intrinsic matrix K the only parameter of the camera, making it easier to find.

2 Programming Section

2.1 Theoretical Background

2.1.1 Canny Edge Detection

The Canny edge detector is derived by optimizing a convolutional function $h(x, y)$ against a signal $f(x)$ to identify peaks which corresponds to the location of the jump discontinuities in $f(x)$, with respect to three criteria:

1) The Signal-to-Noise (SNR) ratio at the true location of the edge

2) The Localization of the detected edge vis-a-vis that of the true edge

3) Maximize the distance between where the true edge is detected and the nearest spurious edge

Canny Edge Detection was achieved by using Open-CV's builtin function cv2.Canny(). Before this, the image that would be passed into the function is first converted to grayscale and then filtered with 3x3 Gaussian kernel to minimize the noise in the image. The threshold for the Canny Edge detector was between 255 and 383.

2.1.2 Hough Lines

The approach for detecting straight line features in an image even if the pixels on the lines are not connected involve:

1. Applying an edge detector to the image and binarized its output

2. Divide the parameter space (slope (m), y-intercept (c)) of the line containing disconnected pixels into bins

3. In a raster can of the binarized image in step 1, when we encounter a non-zero pixel, increment the bin counts in all of the bins that correspond to that pixel

4. Threshold the 2D parameter-space histogram constructed

However, the parameter space (m, c) is not efficient in the actual implementation because the slope can get as big as ∞ so instead the Hough transform uses polar coordinates (ρ, θ) where ρ is the distance from the origin to the line corresponding to the pixel at an angle θ from the x-axis. To implement hough transform, we use Open-CV's builtin function cv2.houghlines() with a ρ of one, θ of $\frac{\text{ratio}\times\pi}{180}$, and a threshold of 60 for the provided dataset and 50 for the custom dataset.

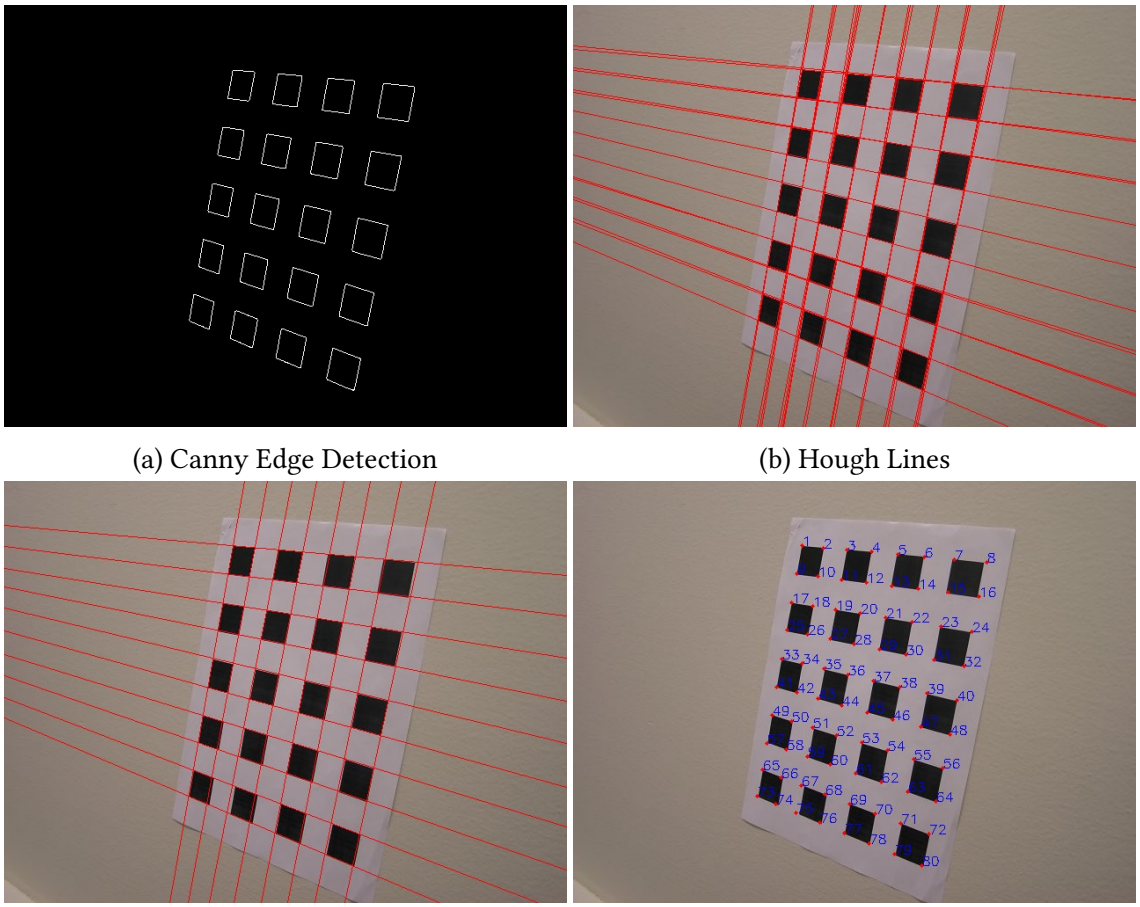
All the vertical lines are found within the ranges $\frac{-\pi}{4} \leq \theta \leq \frac{\pi}{4}$ and $\frac{3\pi}{4} \leq \theta \leq \frac{5\pi}{4}$. As a result, any line from the hough transform that satisfies $cos^2(\theta) > \frac{1}{2}$ are vertical lines, otherwise they are horizontal.

Since the hough transform gives out many lines, we need to now refine them. This is achieved by grouping lines that have a distance less than a specific threshold, called the non-maximum threshold. The threshold for each horizontal and vertical group is calculated by taking the difference between the farthest and closest lines and measuring the average distance between the two lines.

2.1.3 Corner Detection

The corners are detected at the intersection of all horizontal and vertical lines. This is accomplished by taking the cross product of the lines in its homogeneous space. By using this methodology, we successfully extracted 80 corners from 39 images out of 40 in the provided dataset and 16 out of 20 images in the custom dataset.

2.1.3.1 Results



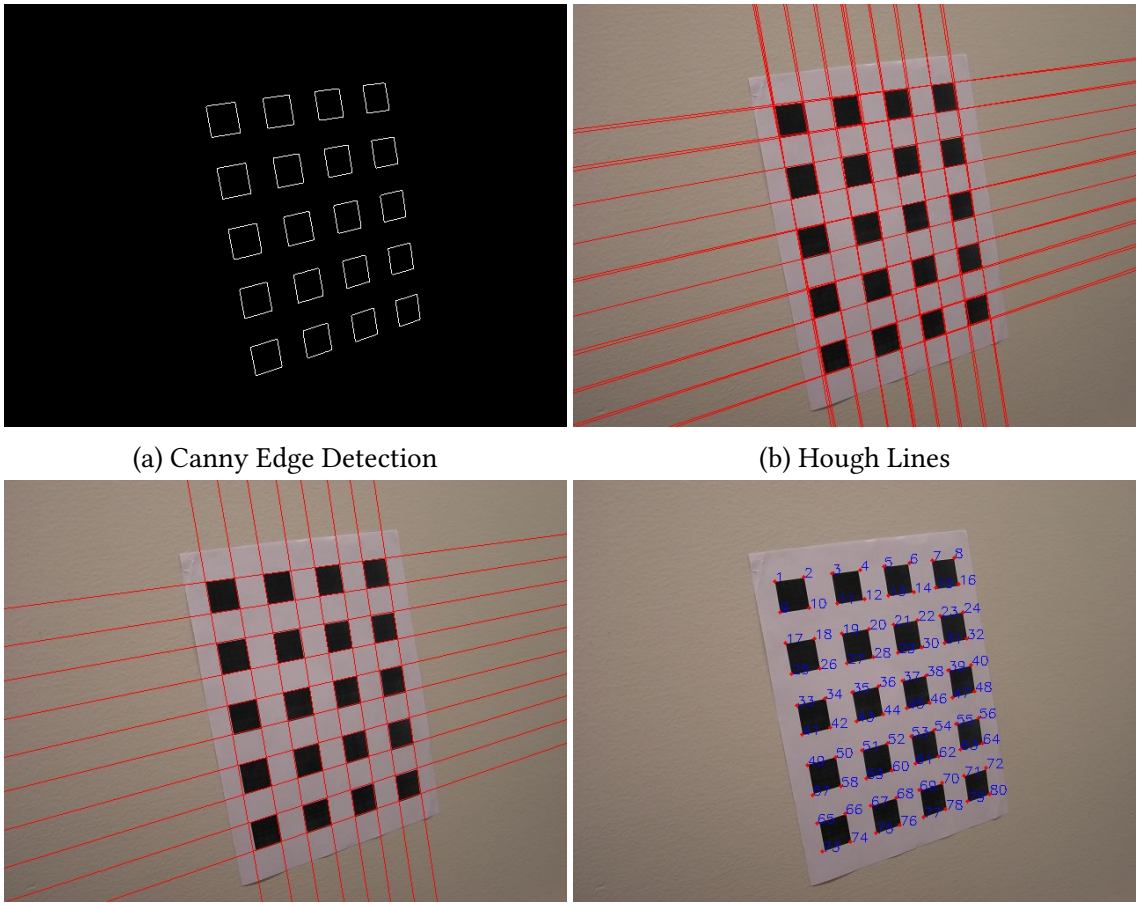
(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 1: Provided Picture 1 Corners



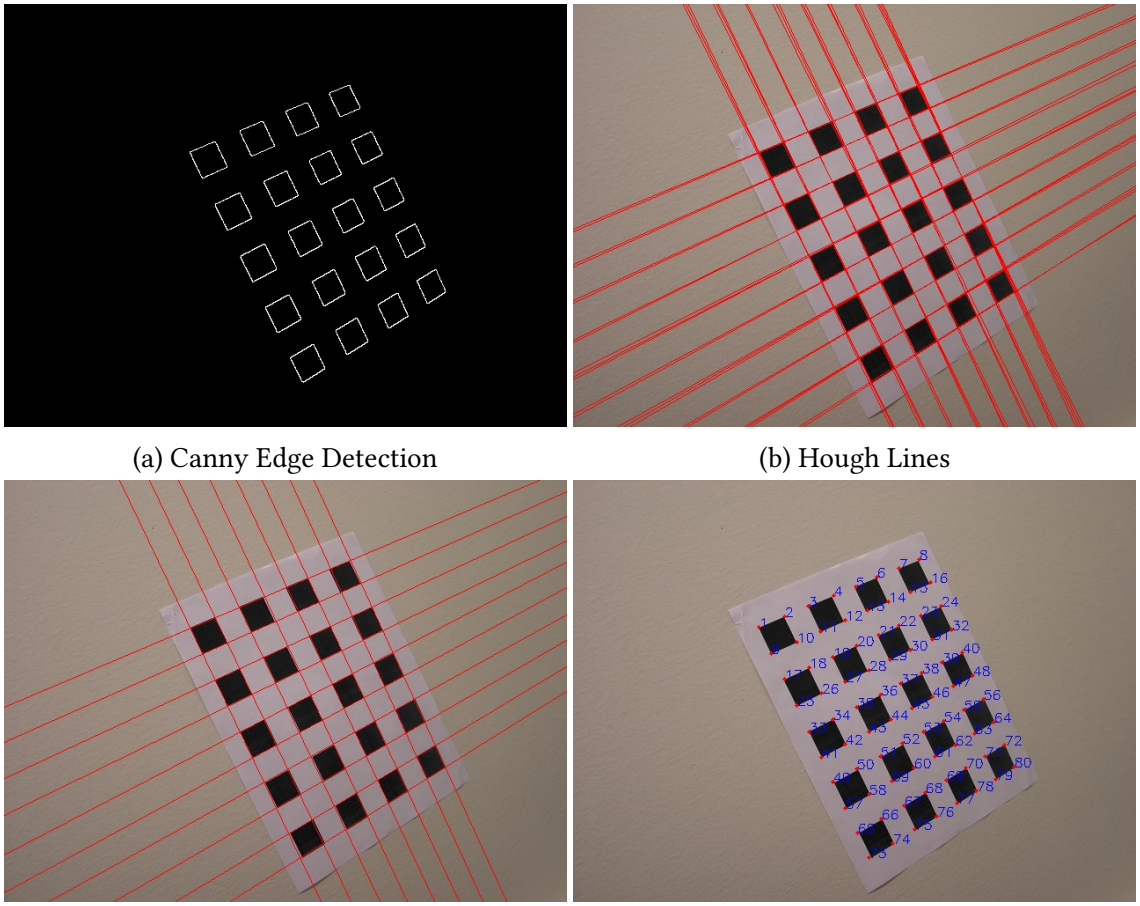
(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 2: Provided Picture 3 Corners



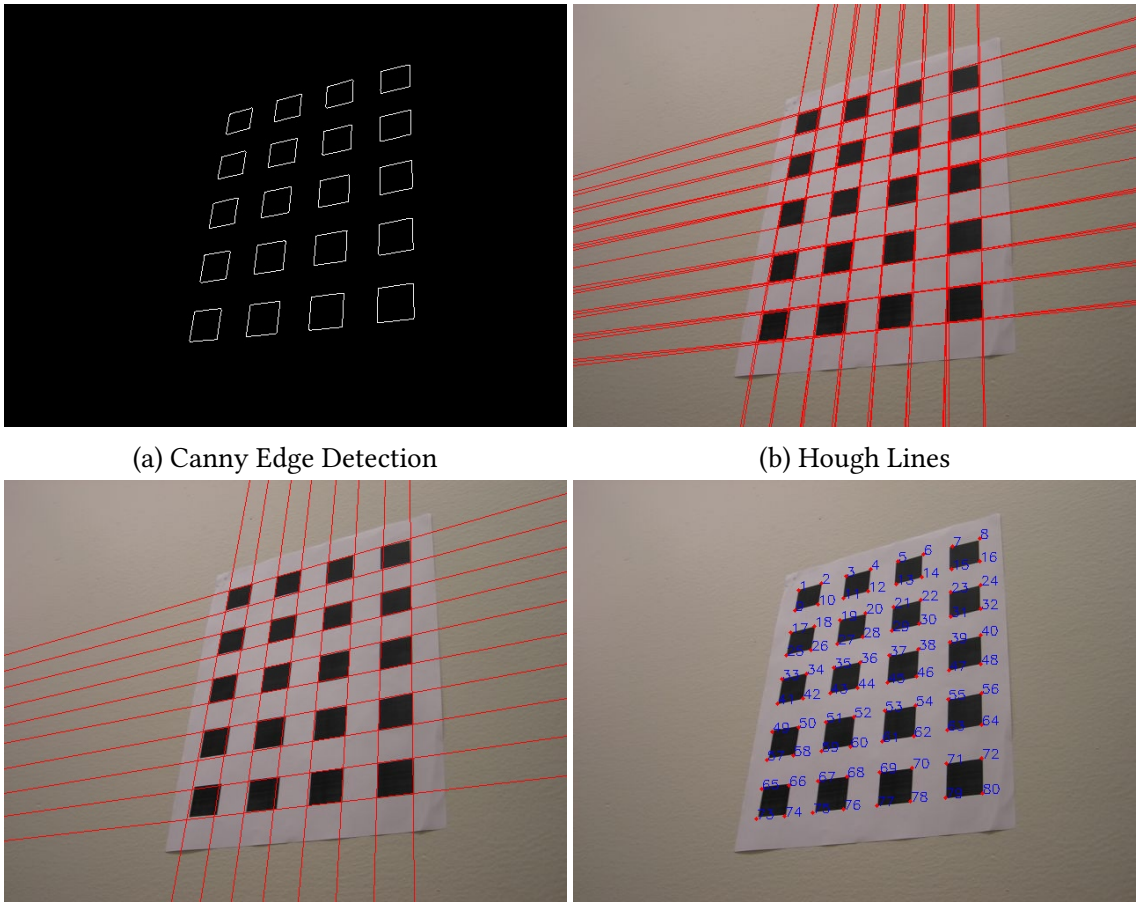
(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 3: Provided Picture 4 Corners



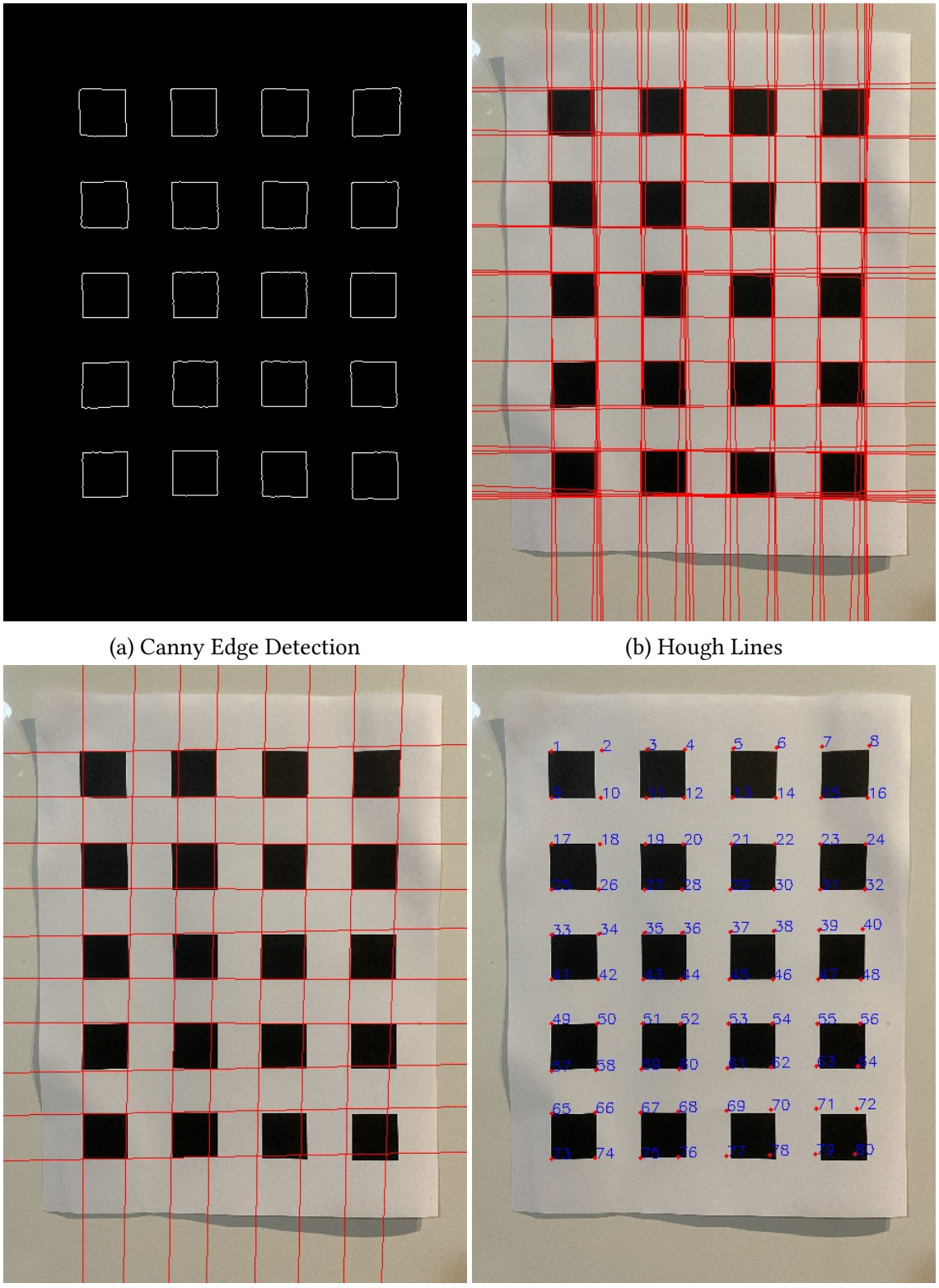
(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 4: Provided Picture 9 Corners



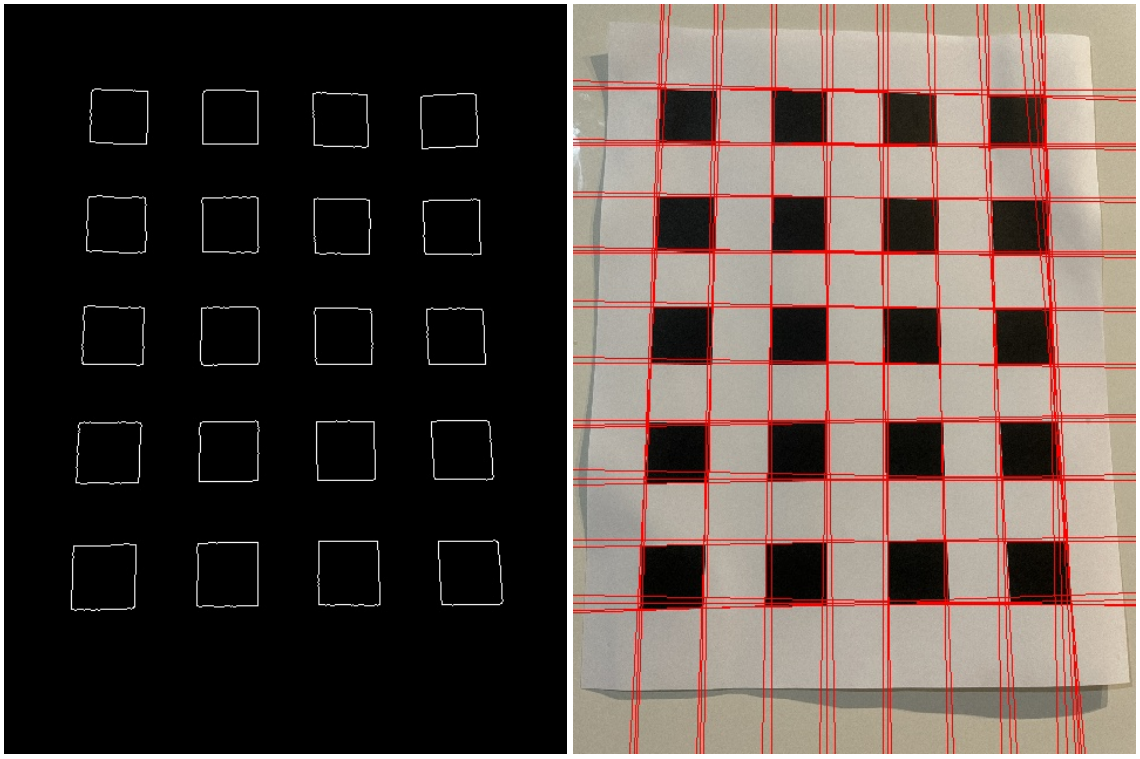
(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 5: Custom Picture 1 Corners



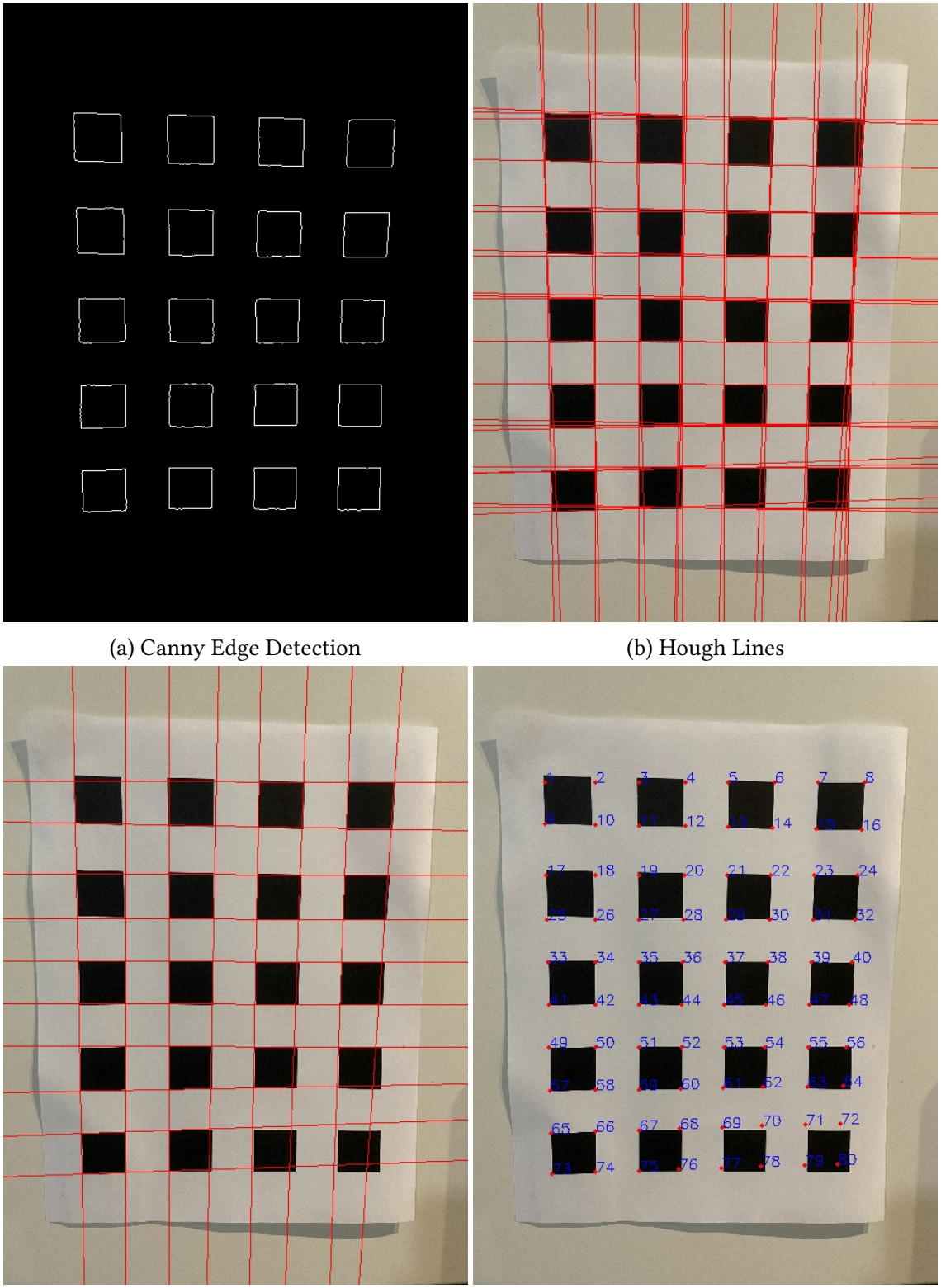
(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 6: Custom Picture 2 Corners



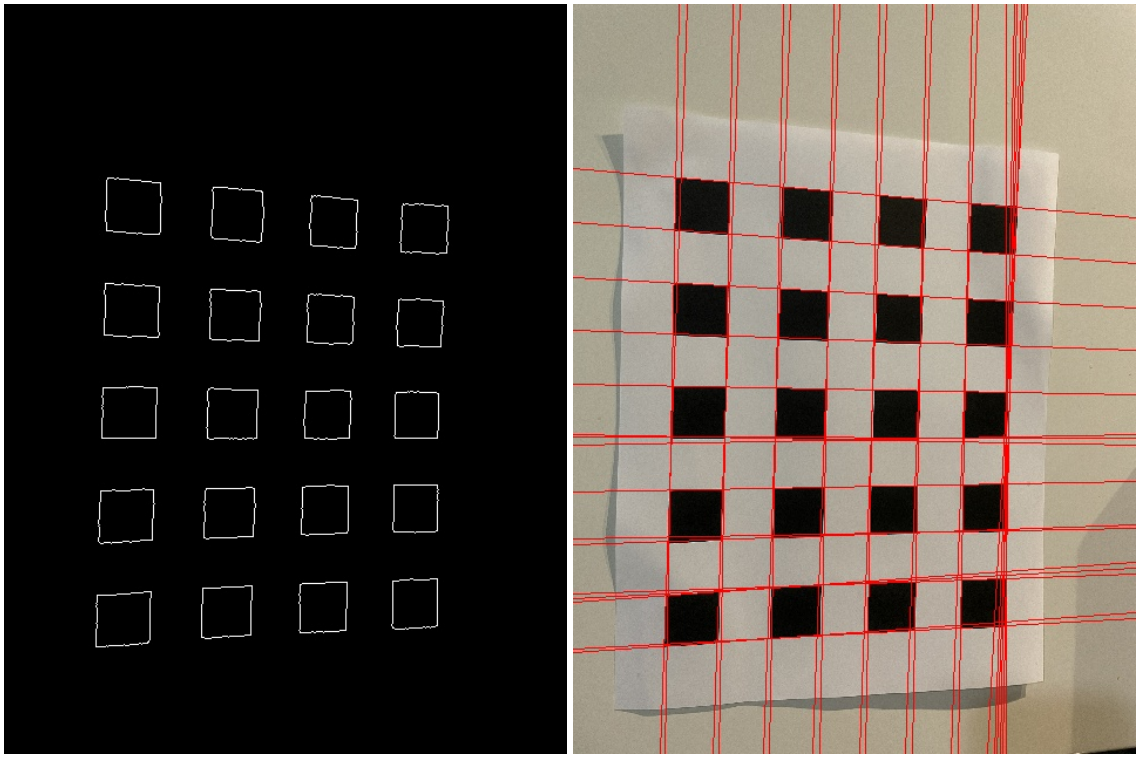
(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 7: Custom Picture 3 Corners



(a) Canny Edge Detection

(b) Hough Lines

(c) Refined Lines

(d) Corners

Figure 8: Custom Picture 4 Corners

2.1.4 Levenberg-Marquadt Non-Linear Optimization

As mentioned in homework 5, the Levenberg-Marquardt algorithm combines the best of Gradient Descent and Gauss-Newton by adding a heuristic damping coefficient μ which when manipulated can steer between the Gradient Descent approach or the Gauss Newton approach. At the beginning, μ is set to an extremely large value so that the steps taken along the convex function mimics Gradient Descent but when the estimation gets closer to the local minimum, μ decreases thus allowing for the steps to mimic Gauss-Newton.

For this homework assignment, the Levenberg-Marquardt algorithm is implemented by using the built-in SciPy function. The cost function is given by the error between the actual coordinates in the range space and the computed coordinates using the homography matrix H.

2.1.5 Zhang's Algorithm

Zhang's algorithm involves computing two camera parameters: the intrinsic matrix K and the extrinsic matrix which is a combination of the rotation matrix R and translation matrix t. We then make use of Levenberg-Marquadt non-linear optimization to refine the intrinsic and extrinsic matrices from its initial guess.

2.1.5.1 Calculating the Intrinsic Matrix

Initially, we make the assumption that the plane in which the calibration checkerboard is laid on is on the plane $z = 0$. So the corner on the image in the real world has coordinates of $\vec{x} = [x, y, 0, w]^T$. Therefore the same point is projected onto the 2D image world as shown below

$$\vec{y} = H\vec{x} = K[R|\vec{t}][x, y, 0, w]^T \tag{1}$$

where $H = [h_1, h_2, h_3]$ is the homography matrix.

For each position of the camera, the Ω_∞ will be sampled at $I = [1, i, 0]^T$ and $J = [1, -i, 0]^T$. Therefore these circular points reside on the conic ω in the image plane where $\omega = K^{-T}K^{-1}$ as mentioned in the theoretical problem by a homography matrix H as shown below:

$$H\vec{I} = \vec{h_1} + i\vec{h_2} \tag{2}$$

$$H\vec{J} = \vec{h_1} - i\vec{h_2} \tag{3}$$

These circular points must obey the conic constraint

$$(H\vec{I})^T\omega(H\vec{I}) = (\vec{h_1} + i\vec{h_2})^T\omega(\vec{h_1} + i\vec{h_2}) = 0 \tag{4}$$

$$(H\vec{J})^T\omega(H\vec{J}) = (\vec{h_1} - i\vec{h_2})^T\omega(\vec{h_1} - i\vec{h_2}) = 0 \tag{5}$$

Equations (4) and (5) are further simplified down to give

$$\vec{h_1}\omega\vec{h_1} = \vec{h_2}\omega\vec{h_2} \tag{6}$$

$$\vec{h_1}\omega\vec{h_2} = 0 \tag{7}$$

Since ω is symmetric, we only have six unknowns which is expressed as a null vector

$$\vec{b} = [\omega_{11}, \omega_{12}, \omega_{22}, \omega_{13}, \omega_{23}, \omega_{33}]$$

to the matrix combination of the homography matrices V represented in equations (6) and (7). V is denoted as

$$V = \begin{bmatrix} \vec{V_{12}}^T \\ (\vec{V_{11}} - \vec{V_{22}})^T \end{bmatrix} \tag{8}$$

where

$$V_{ij} = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix} \tag{9}$$

The null vector $\vec{b}$ is calculated by finding the last column of the left eigenvectors that is given from the SVD of matrix V.

From the matrix ω we can now derive the intrinsic matrix K by constructing it as

$$K = \begin{bmatrix} \alpha_x & s & x0 \\ 0 & \alpha_y & y0 \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

According to Zhang's algorithm, these 5 unknowns are calculated by

$$y0 = \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2} \tag{11}$$

$$\lambda = \omega_{33} - \frac{\omega_{13}^2 + y0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}} \tag{12}$$

$$\alpha_x = \sqrt{\frac{\lambda}{\omega_{11}}} \tag{13}$$

$$\alpha_y = \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}} \tag{14}$$

$$s = -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda} \tag{15}$$

$$x0 = \frac{sy0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda} \tag{16}$$

2.1.5.1.1 Intrinsic Matrix Results

Before LM for Provided Dataset

$$K = \begin{bmatrix} 715.78 & -7.94 & 324.38 \\ 0.00 & 712.32 & 239.20 \\ 0 & 0 & 1 \end{bmatrix} \tag{17}$$

After LM for Provided Dataset

$$K = \begin{bmatrix} 718.05 & -9.07 & 327.38 \\ 0.00 & 714.41 & 242.36 \\ 0 & 0 & 1 \end{bmatrix} \tag{18}$$

Before LM for Custom Dataset

$$K = \begin{bmatrix} 529.55 & -10.19 & 254.17 \\ 0.00 & 534.64 & 326.68 \\ 0 & 0 & 1 \end{bmatrix} \tag{19}$$

After LM for Custom Dataset

$$K = \begin{bmatrix} 495.11 & -9.67 & 250.90 \\ 0.00 & 499.95 & 320.53 \\ 0 & 0 & 1 \end{bmatrix} \tag{20}$$

2.1.5.2 Calculating the Extrinsic Matrix

Given that $K^{-1}[\vec{h_1}, \vec{h_2}, \vec{h_3}] = [\vec{r_1}, \vec{r_2}, \vec{t}]$, the rotation matrix R has to be orthonormal so we normalize R and $\vec{t}$ with a scaling factor $\xi = 1/||k^{-1}\vec{h_1}||$. Therefore,

$$\begin{bmatrix} \vec{r_1} & \vec{r_2} & \vec{t} \end{bmatrix} = \xi K^{-1} \begin{bmatrix} \vec{h_1} & \vec{h_2} & \vec{h_3} \end{bmatrix} \tag{21}$$

$$\vec{r_3} = \xi \vec{r_1} \times \vec{r_2} \tag{22}$$

Because the rotation matrix R in the real world has 3 Degrees of Freedom, R is represented as Rodriquez vector

$$\vec{\omega} = \frac{\phi}{2 \sin \phi} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{23}$$

where

$$\phi = \arccos \frac{\text{trace}(R) - 1}{2} \tag{24}$$

and R is reconstructed from the Rodriguez matrix $\vec{\omega}$ as shown below

$$R = I_{3\times3} + \frac{\sin \phi}{\phi}[\omega]_x + \frac{1 - \cos \phi}{\phi^2}[\omega]_x^2 \tag{25}$$

and $\phi = ||\vec{\omega}||$

2.1.5.2.1 Extrinsic Matrix Results

Before LM Rotation Matrix for Provided Dataset

$$R = \begin{bmatrix} 0.792 & -0.179 & 0.584 \\ 0.198 & 0.980 & 0.0313 \\ -0.577 & 0.0910 & 0.811 \end{bmatrix} \tag{26}$$

After LM Rotation Matrix for Provided Dataset

$$R = \begin{bmatrix} 0.79 & -0.18 & 0.58 \\ 0.20 & 0.98 & 0.02 \\ -0.57 & 0.10 & 0.81 \end{bmatrix} \tag{27}$$

Before LM Translation Matrix for Provided Dataset

$$T = \begin{bmatrix} -19.96 & -50.55 & 217.00 \end{bmatrix} \tag{28}$$

After LM Translation Matrix for Provided Dataset

$$T = \begin{bmatrix} -20.91 & -51.38 & 216.81 \end{bmatrix} \tag{29}$$

Before LM Rotation Matrix for Custom Dataset

$$R = \begin{bmatrix} 0.99 & 0.0047 & -0.0060 \\ -0.0050 & 0.999 & -0.038 \\ 0.0058 & 0.038 & 0.99 \end{bmatrix} \tag{30}$$

After LM Rotation Matrix for Custom Dataset

$$R = \begin{bmatrix} 0.99 & 0.0041 & -0.0058 \\ -0.0043 & 0.99 & -0.037 \\ 0.0056 & 0.037 & 0.99 \end{bmatrix} \tag{31}$$

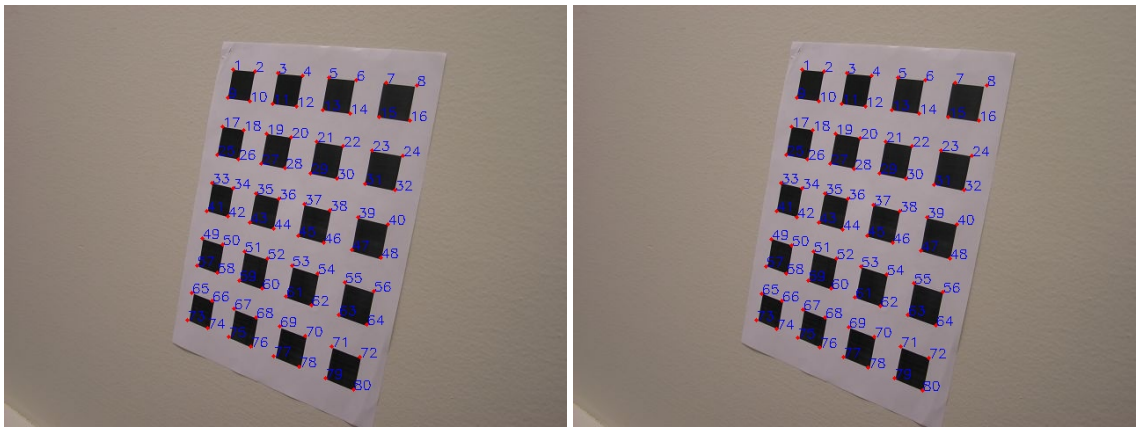
Before LM Translation Matrix for Custom Dataset

$$T = \begin{bmatrix} -37.44 & -50.52 & 113.77 \end{bmatrix} \tag{32}$$

After LM Translation Matrix for Custom Dataset

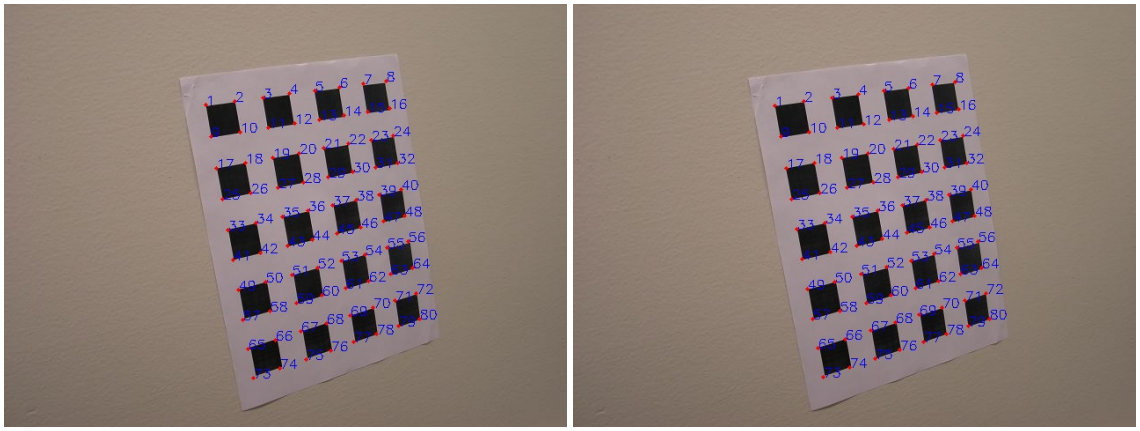
$$T = \begin{bmatrix} -36.83 & -48.91 & 105.69 \end{bmatrix} \tag{33}$$

2.1.5.2.2 Before and After LM Projections



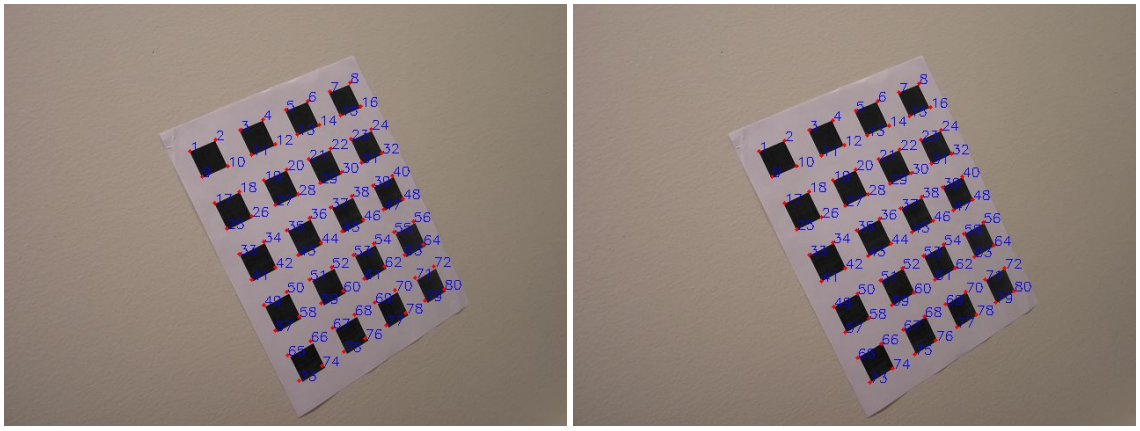
(a) Before LM: Mean=0.807 (b) After LM: Mean=0.718

Figure 9: Provided Before and After LM of Picture 1



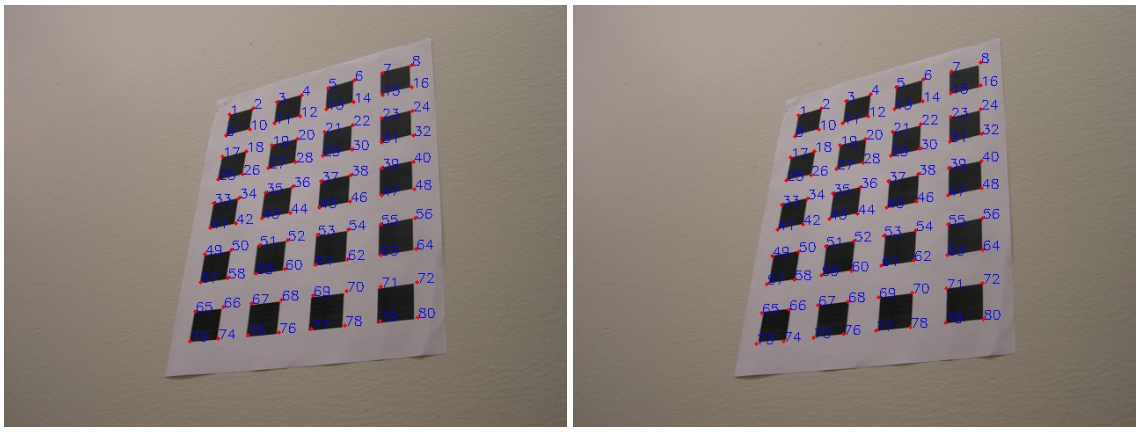
(a) Before LM: Mean=0.812 (b) After LM: Mean=0.506

Figure 10: Provided Before and After LM of Picture 3



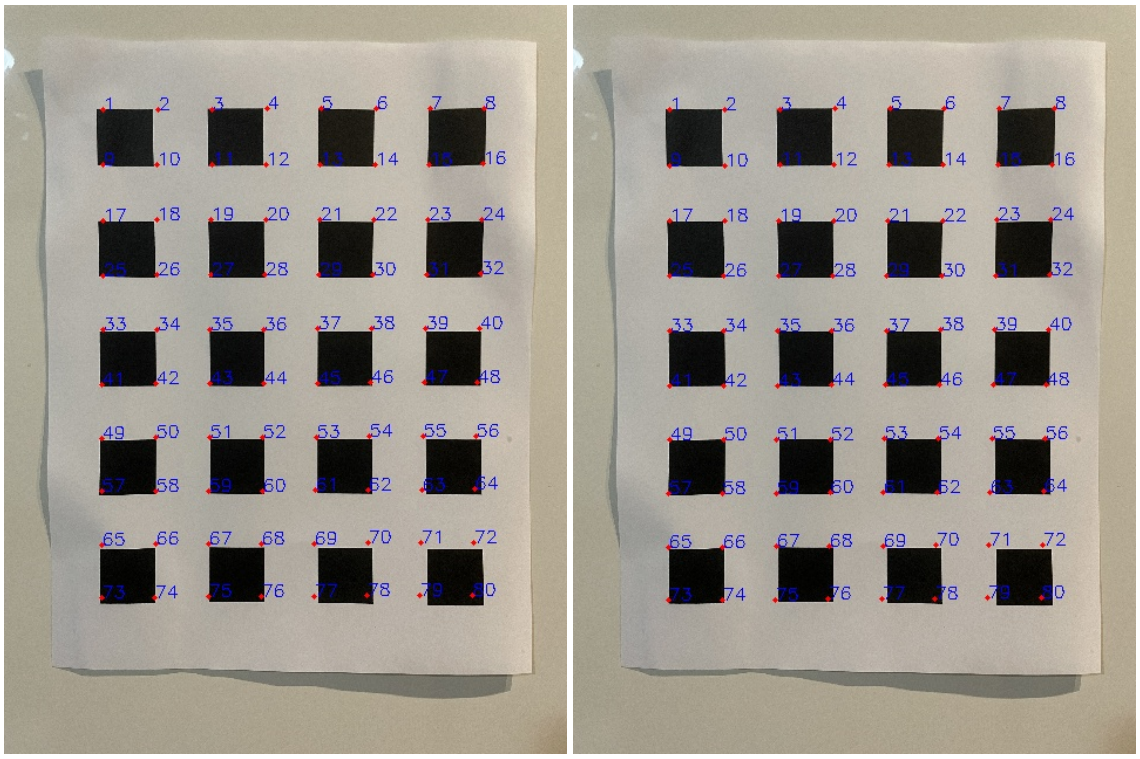
(a) Before LM: Mean=1.704 (b) After LM: Mean=0.581

Figure 11: Provided Before and After LM of Picture 4



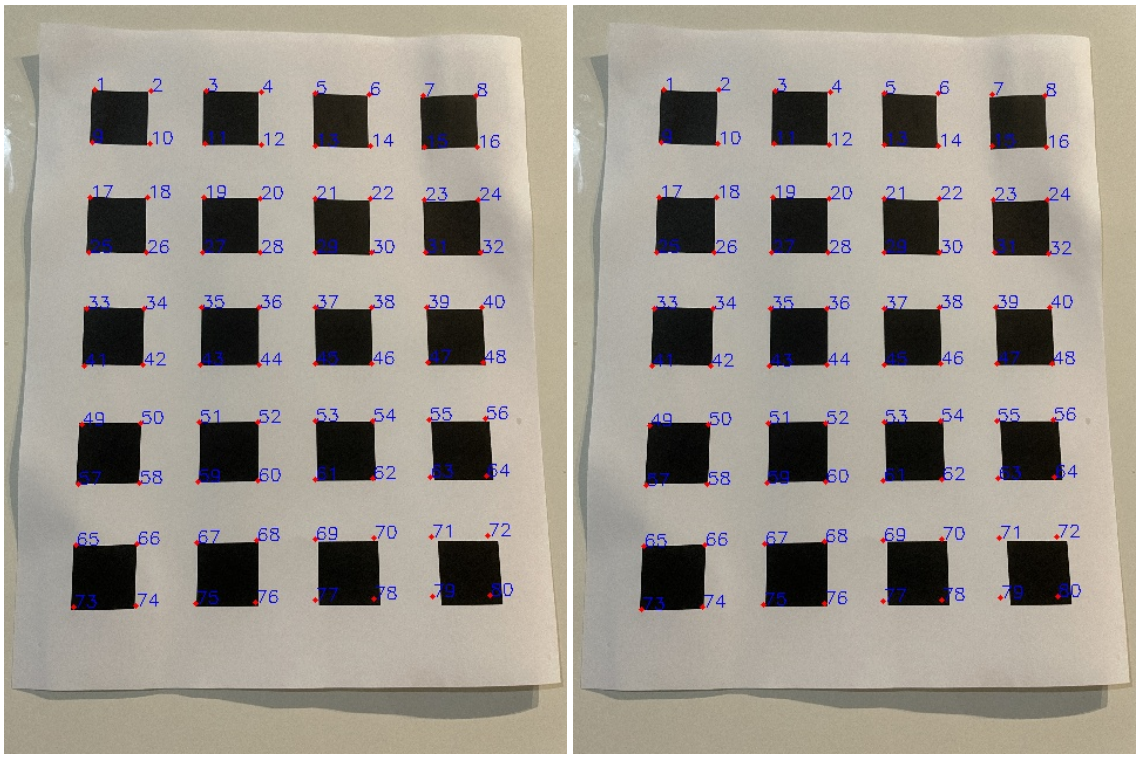
(a) Before LM: Mean=2.492 (b) After LM: Mean=2.154

Figure 12: Provided Before and After LM of Picture 9



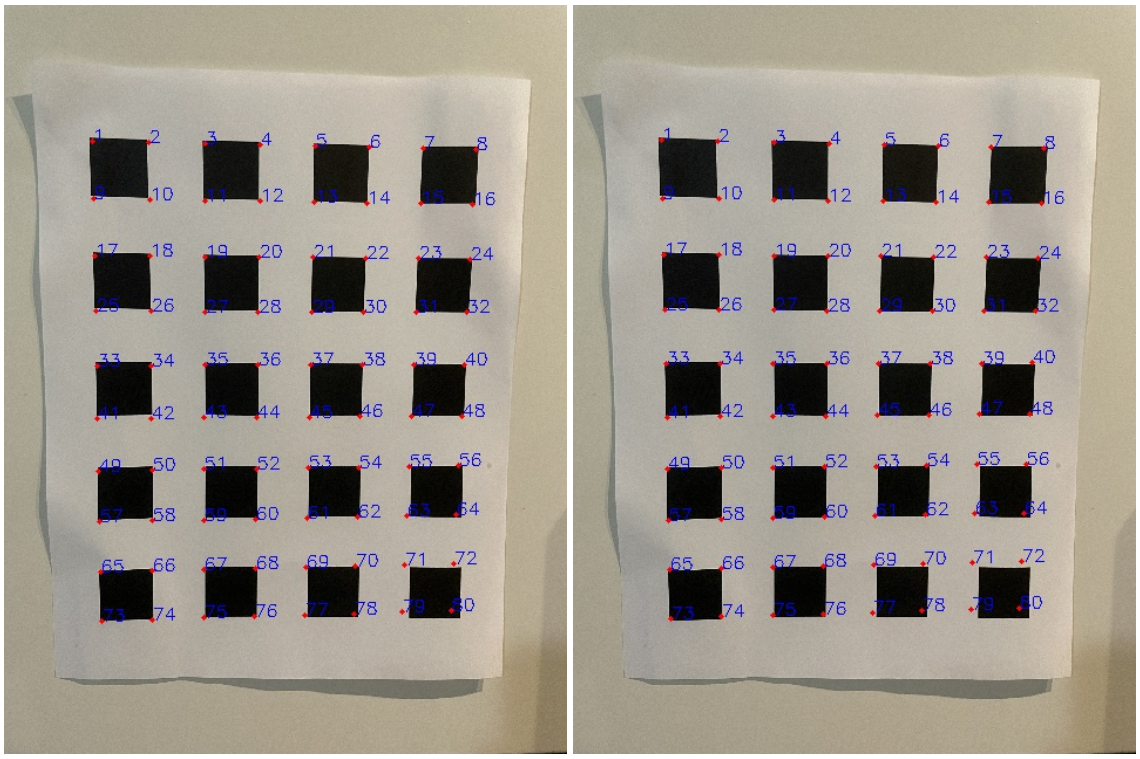
(a) Before LM: Mean=2.878 (b) After LM: Mean=2.211

Figure 13: Custom Before and After LM of Picture 1



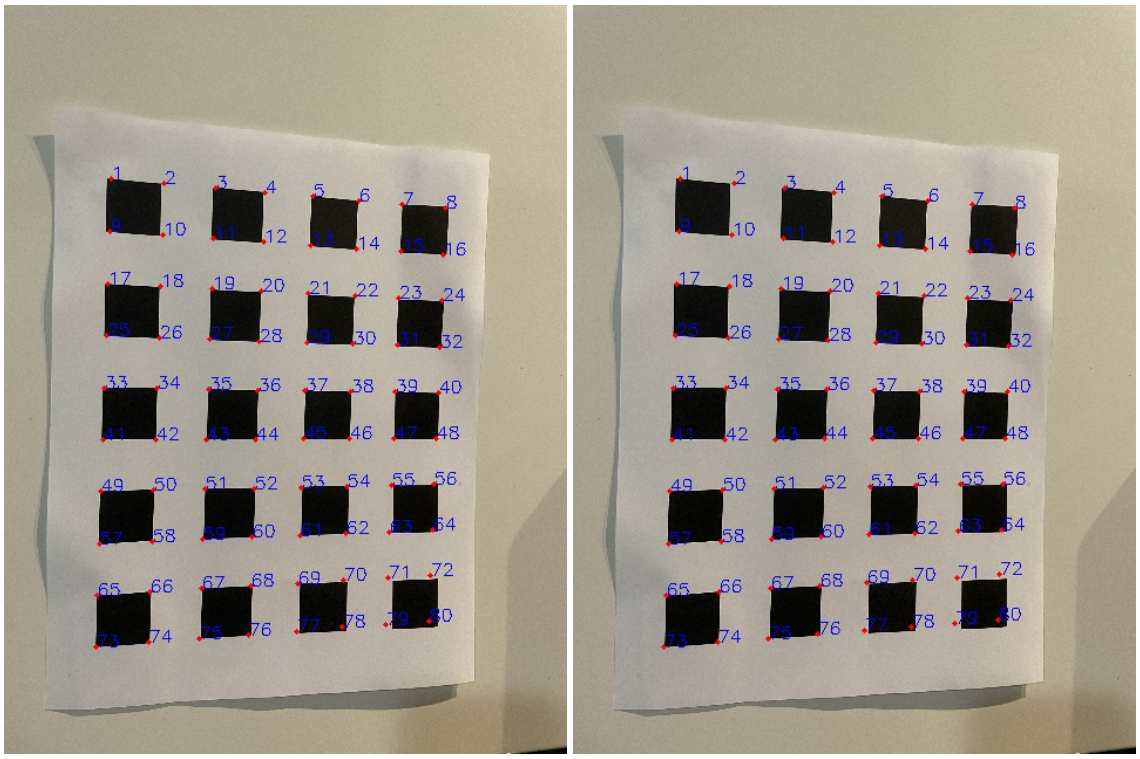
(a) Before LM: Mean=1.992 (b) After LM: Mean=1.762

Figure 14: Custom Before and After LM of Picture 2



(a) Before LM: Mean=2.160 (b) After LM: Mean=1.529

Figure 15: Custom Before and After LM of Picture 3



(a) Before LM: Mean=0.835 (b) After LM: Mean=0.825

Figure 16: Custom Before and After LM of Picture 4

2.2 Source Code

```
# Name: Nikita Ravi
# Class: ECE 66100
# Homework #8
# Deadline: 11/14/2022

# Import Modules
import cv2
import numpy as np
import math
import os
import re
from scipy.optimize import leastsquares

def getimages(path):
  files = [file for file in os.listdir(path)[::-1] if file != ".DSStore"]
  images = [cv2.imread(path + "/" + img) for img in sorted(files, key=lambda x
    : int(re.sub('"D', '', x)))]
  # print(os.listdir(path)[::-1])
  # print(sorted(os.listdir(path)[::-1], key=lambda x: int(re.sub('"D', '', x)
    ))); quit()
  return images

def displayimage(image):
  cv2.imshow("window", image)
  cv2.waitKey(0)
  cv2.destroyAllWindows()
  quit()

def drawlines(image, pts1, pts2, color=(0,0,255), thickness=1):
  copy = image.copy()
  for idx in range(pts1.shape[0]):
    pt1 = (int(pts1[idx, 0]), int(pts1[idx, 1]))
    pt2 = (int(pts2[idx, 0]), int(pts2[idx, 1]))

    cv2.line(copy, pt1, pt2, color=color, thickness=thickness)
  return copy

def drawpoints(image, ptslist, radius, color=(0,0,255), thickness=-1, text=
    True):
  copy = image.copy()
  for idx in range(ptslist.shape[0]):
    pt1, pt2 = int(ptslist[idx, 0]), int(ptslist[idx, 1])
    cv2.circle(copy, (pt1, pt2), radius=radius, color=color, thickness=
    thickness)

    if(text):
      cv2.putText(copy, str(idx + 1), (pt1, pt2), fontFace=cv2.
    FONTHERSHEYSIMPLEX, fontScale=0.5, color=(255,0,0), thickness=1)

  return copy

def createmeshgrid(gridsize, numhorizontallines, numverticallines):
```

```
  x = np.linspace(0, gridsize*(numverticallines-1), numverticallines)
  y = np.linspace(0, gridsize*(numhorizontallines-1), numhorizontallines)

  xmesh, ymesh = np.meshgrid(x,y)
  mesh = np.concatenate([xmesh.reshape((-1,1)), ymesh.reshape((-1,1))], axis
   =1)
  return mesh

def createblurredimage(image, data):
  gray = cv2.cvtColor(image, cv2.COLORBGR2GRAY)
  if(data == "data2"):
    gray = cv2.GaussianBlur(gray, ksize=(3,3), sigmaX=1.4) # dataset 2

  return gray

def cannyedgedetection(idx, image, data="data1"):
  # Canny edge detection finds all the edges on the checkerboard pattern
  blur = createblurredimage(image, data)
  edges = cv2.Canny(blur, 255*1.5, 255)
  cv2.imwrite(os.path.join("hw08/" + data + "/cannyEdges", "image" + str(idx
   + 1) + ".jpg"), edges)

  return edges

def getpointsfromhoughlines(lines):
  rho, theta = lines[:, 0], lines[:, 1]
  a, b = np.cos(theta), np.sin(theta)
  x0, y0 = a * rho, b * rho
  pts1 = np.array([x0 + 1000*(-b), y0 + 1000*(a)])
  pts2 = np.array([x0 - 1000*(-b), y0 - 1000*(a)])
  return pts1.T, pts2.T

def houghlines(edge, image, imageid, ratio=0.5, houghthreshold=60, data="
   data1"):
  # Draw a straight line along the edges of the checkerboard
  lines = cv2.HoughLines(edge, rho=1, theta=ratio*np.pi / 180, threshold=
   houghthreshold)
  lines = np.squeeze(lines)
  if(lines is not None):
    pt1, pt2 = getpointsfromhoughlines(lines)
    houghimage = drawlines(image, pt1, pt2)
    cv2.imwrite(os.path.join("hw08/" + data + "/houghLines", "image" + str(
   imageid + 1) + ".jpg"), houghimage)

  return lines

def cornerdetection(image, imageid, lines, nmsratio=0.24, data="data1"):
  copy = image.copy()
  rho, theta = lines[:, 0], lines[:, 1]

  verticallines = lines[np.where(np.cos(theta)**2 ¿ 0.5)]
  verticaldist = verticallines[:, 0] * np.cos(verticallines[:, 1])
  verticallines = verticallines[np.argsort(verticaldist)]
```

```
  verticallines = nonmaximumsuppression(verticallines, nmsratio, vertical
    = True)
  verticalpts1, verticalpts2 = getpointsfromhoughlines(verticallines)

  horizontallines = lines[np.where(np.cos(theta)**2 ¡= 0.5)]
  horizontaldist = horizontallines[:, 0] * np.sin(horizontallines[:, 1])
  horizontallines = horizontallines[np.argsort(horizontaldist)]
  horizontallines = nonmaximumsuppression(horizontallines, nmsratio,
   vertical = False)
  horizontalpts1, horizontalpts2 = getpointsfromhoughlines(
   horizontallines)

  refinedlinesimage = drawlines(copy, verticalpts1, verticalpts2)
  refinedlinesimage = drawlines(refinedlinesimage, horizontalpts1,
   horizontalpts2)

  cv2.imwrite(os.path.join("hw08/" + data + "/refined", "image" + str(
   imageid + 1) + ".jpg"), refinedlinesimage)

  horizontallineshc, verticallineshc = gethomogeneouslines(
   horizontalpts1, horizontalpts2, verticalpts1, verticalpts2)
  corners = getintersections(horizontallineshc, verticallineshc)
  cornerimage = drawpoints(copy, corners, radius=2)

  cv2.imwrite(os.path.join("hw08/" + data + "/corners", "image" + str(
   imageid + 1) + ".jpg"), cornerimage)
  return corners

def gethomogeneouslines(hpt1, hpt2, vpt1, vpt2):
  vpt1 = np.append(vpt1, np.ones((vpt1.shape[0],1)), axis=1)
  vpt2 = np.append(vpt2, np.ones((vpt2.shape[0],1)), axis=1)

  hpt1 = np.append(hpt1, np.ones((hpt1.shape[0],1)), axis=1)
  hpt2 = np.append(hpt2, np.ones((hpt2.shape[0],1)), axis=1)

  verticalHC = np.cross(vpt1, vpt2)
  horizontalHC = np.cross(hpt1, hpt2)

  return horizontalHC, verticalHC

def getintersections(horizontallines, verticallines):
  corners = []
  for idx in range(horizontallines.shape[0]):
    corner = np.cross(horizontallines[idx], verticallines)
    corner = corner[:,:2]/corner[:,2].reshape((-1,1))
    corners.append(corner)

  corners = np.array(corners)
  corners = corners.reshape((-1,2))
  return corners

def nonmaximumsuppression(lines, nmsratio=0.24, vertical=True):
  validlines = []
  rho, theta = lines[:, 0], lines[:, 1]
```

```
143   totalnumlinejumps = 7 if vertical else 9
144
145   dist = rho * np.cos(theta) if vertical else rho * np.sin(theta) # Distance
       to the vertical/horizontal line
146   nmsthreshold = nmsratio * (np.max(dist) - np.min(dist)) /
       totalnumlinejumps # Taking the average over 7 vertical line jumps
147
148   for idx in range(dist.shape[0] - 1):
149     if(dist[idx + 1] - dist[idx] ¿ nmsthreshold):
150       rhoidx, thetaidx = lines[idx, 0], lines[idx, 1]
151       validlines.append([rhoidx, thetaidx])
152
153     if(idx == dist.shape[0] - 2):
154       rhoidx, thetaidx = lines[idx + 1, 0], lines[idx + 1, 1]
155       validlines.append([rhoidx, thetaidx])
156
157   return np.array(validlines)
158
159 def computehomography(domaincoord, rangecoord):
160   # rangecoord = domaincoord * H.T
161   # From homework #3
162   domaincoord, rangecoord = domaincoord.T, rangecoord.T
163   n = domaincoord.shape[1]
164   A = np.zeros((2*n, 8))
165   b = np.zeros((2*n, 1))
166   H = np.zeros((3,3))
167
168   for idx in range(n):
169     A[2*idx] = [domaincoord[0][idx], domaincoord[1][idx], 1, 0, 0, 0, (-
       domaincoord[0][idx] * rangecoord[0][idx]), (-domaincoord[1][idx] *
       rangecoord[0][idx])]
170     A[2*idx + 1] = [0, 0, 0, domaincoord[0][idx], domaincoord[1][idx], 1, (-
       domaincoord[0][idx] * rangecoord[1][idx]), (-domaincoord[1][idx] *
       rangecoord[1][idx])]
171     b[2*idx] = rangecoord[0][idx]
172     b[2*idx + 1] = rangecoord[1][idx]
173
174   h = np.matmul(np.linalg.pinv(A), b)
175   row = 0
176   for idx in range(0, len(h), 3):
177     spliced = h[idx:idx+3]
178     if(len(spliced) == 3):
179       H[row] = spliced.T
180     else:
181       H[row] = np.append(spliced, [1])
182     row += 1
183
184   return H
185
186 def computeVijcomponent(hi, hj):
187   Vij = np.array([hi[0] * hj[0],
188     hi[0] * hj[1] + hi[1] * hj[0],
189     hi[1] * hj[1],
190     hi[2] * hj[0] + hi[0] * hj[2],
```

```
    hi[2] * hj[1] + hi[1] * hj[2],
    hi[2] * hj[2]])

  return Vij.T

def computeomegamatrix(Hlist):
  V = []
  for H in Hlist:
    h1 = H[:,0]
    h2 = H[:,1]
    h3 = H[:,2]

    v11 = computeVijcomponent(h1,h1)
    v12 = computeVijcomponent(h1,h2)
    v22 = computeVijcomponent(h2,h2)

    V.append(v12.T)
    V.append((v11-v22).T)

  # Compute SVD of v to find b
  u, s, vh = np.linalg.svd(V)
  b = vh[-1] # w11, w12, w22, w13, w23, w33
  w11, w12, w22, w13, w23, w33 = b
  w21 = w12
  w31 = w13
  w32 = w23

  omega = np.array([[w11, w12, w13],
        [w21, w22, w23],
        [w31, w32, w33]])

  return omega

def computeintrinsicmatrix(omega):
  b = omega.flatten()
  w11, w12, w13, w21, w22, w23, w31, w32, w33 = b

  y0 = (w12 * w13 - w11 * w23) / (w11 * w22 - w12**2)
  Lambda = w33 - (w13**2 + y0*(w12*w13-w11*w23)) / w11
  alphax = math.sqrt(Lambda / w11)
  alphay = math.sqrt((Lambda * w11) / (w11 * w22 - w12**2))
  s = -(w12 * alphay * alphax**2) / Lambda
  x0 = (s*y0)/alphay - (w13 * alphax**2)/Lambda

  K = np.array([[alphax, s, x0], [0, alphay, y0], [0, 0, 1]])
  return K

def computeextrinsicmatrix(Hlist, K):
  Rlist = []
  tlist = []
  for H in Hlist:
    r12 = np.dot(np.linalg.inv(K), H)
    scalingfactor = 1 / (np.linalg.norm(r12[:,0]))

```

```
    r12 = scalingfactor * r12
    r3 = np.cross(r12[:,0], r12[:,1])

    Q = np.vstack((r12[:,0], r12[:,1], r3)).T
    u, s, vh = np.linalg.svd(Q)

    R = np.matmul(u, vh)
    Rlist.append(R)
    tlist.append(r12[:, 2])
  return Rlist, tlist

def refiningcalibrationparameters(Rlist):
  # Rotation matrix has 9 elements but 3 DoF so we create the Rodriguez
    representation
  W = []
  for R in Rlist:
    r11, r12, r13, r21, r22, r23, r31, r32, r33 = R.flatten()
    phi = np.arccos((np.trace(R) - 1) / 2)
    w = phi / (2 * np.sin(phi)) * np.array([r32-r23, r13-r31, r21-r12])
    W.append(w)

  return W

def reconstructrotation(w):
  phi = np.linalg.norm(w)
  wx = np.array([[0, -w[2], w[1]], [w[2], 0, -w[0]], [-w[1], w[0], 0]])
  R = np.eye(3) + (np.sin(phi) / phi) * wx + ((1 - np.cos(phi)) / (phi ** 2))
    * np.matmul(wx, wx)

  return R

def computeprojectionmatrix(H, domain):
  domainhomogenous = np.hstack((domain, np.ones((domain.shape[0], 1))))
  P = np.dot(H, domainhomogenous.T).T
  P = P[:, :2] / P[:, 2].reshape((domain.shape[0], 1))
  return P

def costfunction(parameters, intersections, mesh):
  projectionpattern = []
  Rlist, tlist, K = separateparameters(parameters)

  for R, t in zip(Rlist, tlist):
    Rt = np.hstack((R[:, :2], np.reshape(t, (3,1))))
    H = np.dot(K, Rt)
    projectionpattern.append(computeprojectionmatrix(H, mesh))

  projectionpattern = np.concatenate(projectionpattern, axis=0)
  intersections = np.concatenate(intersections, axis=0)
  dgeom = projectionpattern - intersections
  return dgeom.flatten()

def combineparameters(Rlist, tlist, K):
  Rt = []
  W = refiningcalibrationparameters(Rlist)
```

```
  for w, t in zip(W, tlist):
    Rt.append(np.append(w, t))

  Rt = np.concatenate(Rt, axis=0)
  Kparameters = np.array([K[0,0], K[0,1], K[0,2], K[1,1], K[1,2]])
  parameters = np.append(Kparameters, Rt)
  return parameters

def separateparameters(parameters):
  numRt = (parameters.shape[0]-5) // 6
  k = parameters[:5]
  K = np.array([[k[0], k[1], k[2]], [0, k[3], k[4]], [0, 0, 1]])

  Rlist = []
  tlist = []
  extrinsic = parameters[5:]

  for idx in range(numRt):
    w = extrinsic[idx*6: idx*6+3]
    R = reconstructrotation(w)
    Rlist.append(R)

    t = extrinsic[idx*6+3: idx*6+6]
    tlist.append(t)

  return Rlist, tlist, K

def projection(mesh, intersections, validimageidx, images, parameters,
   before="before", data="data1"):
  Rlist, tlist, K = separateparameters(parameters)
  Hlist = []
  for R, t in zip(Rlist, tlist):
    Rt = np.hstack((R[:, :2], np.reshape(t, (3,1))))
    H = np.dot(K, Rt)
    Hlist.append(H)

  differences = []
  for idx, H in enumerate(Hlist):
    projection = computeprojectionmatrix(H, mesh)
    diff = intersections[idx] - projection
    differences.append(diff)

    valididx = validimageidx[idx]
    image = images[idx]
    lmimage = drawpoints(image, projection, radius=2)
    cv2.imwrite(os.path.join("hw08/" + data + "/LM", "image" + str(valididx
   + 1) + " " + before + ".jpg"), lmimage)

  accuracy, error  = computeaccuracy(np.array(differences).flatten()),
    computeerror(np.array(differences).flatten())
  return accuracy, error

def computeaccuracy(differences):
  differences = np.array(differences).reshape((-1, 2))
```

```
  norm = np.linalg.norm(differences, axis=1)
  avg = np.average(norm)
  var = np.var(norm)
  maxdistance = np.max(norm)

  return np.array([avg, var, maxdistance])

def computeerror(differences):
  differences = np.array(differences).reshape((-1, 2))
  norm = np.linalg.norm(differences, axis=1)
  N = differences.shape[0] // 80
  errors = []
  for idx in range(N):
    error = -
    currentdistance = norm[idx*80: idx*80+80]
    error["mean"] = np.mean(currentdistance)
    error["variance"] = np.var(currentdistance)
    error["maxdist"] = np.max(currentdistance)
    errors.append(error)

  return errors

def main(images, r=0.5, nms=0.24, hough=60, data="data1"):
  meshgrid = createmeshgrid(gridsize=10, numhorizontallines=10,
   numverticallines=8)
  imageswith80corners = []
  Hlist = []
  cornerslist = []

  for idx, image in enumerate(images):
    edge = cannyedgedetection(idx, image, data)
    lines = houghlines(edge, image, idx, ratio=r, houghthreshold=hough, data
   =data)
    corners = cornerdetection(image, idx, lines, nmsratio=nms, data=data)

    if(corners.shape[0] == 80):
      imageswith80corners.append(idx)
      cornerslist.append(corners)
      Hlist.append(computehomography(meshgrid, corners))

  print("Number of images with 80 corners: ", len(imageswith80corners))
  print("These images are with indices: ")
  print([idx + 1 for idx in imageswith80corners])

  omega = computeomegamatrix(Hlist) # Compute the omega matrix
  K = computeintrinsicmatrix(omega) # Compute Intrinsic K Matrix
  Rlist, tlist = computeextrinsicmatrix(Hlist, K) # Compute the rotation
   matrix
  parameters = combineparameters(Rlist, tlist, K) # Combine extrinsic and
   intrinsic parameters
  lossfunction = costfunction(parameters, cornerslist, meshgrid) # Create
   loss function

```

```
print("=========================Before LM: Intrinsic Matrix
  ============================")
print(K)
print("=========================Before LM: Rotation Matrix
  ============================")
print(Rlist[0])
print("=========================Before LM: Translation Matrix
  ============================")
print(tlist[0])

result = leastsquares(costfunction, parameters, method="lm", args=[
  cornerslist, meshgrid]) # Levenberg-Marquadt
print("
  ##########################################################################################
  ")

Rlist, tlist, K = separateparameters(result.x)
print("=========================After LM: Intrinsic Matrix
  ============================")
print(K)
print("=========================After LM: Rotation Matrix
  ============================")
print(Rlist[0])
print("=========================After LM: Translation Matrix
  ============================")
print(tlist[0])


accbefore, errorbefore = projection(meshgrid, cornerslist,
  imageswith80corners, images, parameters, before="before", data=data)
accafter, errorafter = projection(meshgrid, cornerslist,
  imageswith80corners, images, result.x, before="after", data=data)

print("---------------------------------------Image
  1-----------------------------------------")
print("BEFORE")
score = errorbefore[0]
print("Mean: ", score["mean"])
print("Variance: ", score["variance"])
print("Max Distance: ", score["maxdist"])

print("AFTER")
score = errorafter[0]
print("Mean: ", score["mean"])
print("Variance: ", score["variance"])
print("Max Distance: ", score["maxdist"])

print("---------------------------------------Image
  2-----------------------------------------")
print("BEFORE")
score = errorbefore[1]
print("Mean: ", score["mean"])
print("Variance: ", score["variance"])
print("Max Distance: ", score["maxdist"])
```

```
437
438   print("AFTER")
439   score = errorafter[1]
440   print("Mean: ", score["mean"])
441   print("Variance: ", score["variance"])
442   print("Max Distance: ", score["maxdist"])
443
444   print("----------------------------------------Image
        3----------------------------------------")
445   print("BEFORE")
446   score = errorbefore[2]
447   print("Mean: ", score["mean"])
448   print("Variance: ", score["variance"])
449   print("Max Distance: ", score["maxdist"])
450
451   print("AFTER")
452   score = errorafter[2]
453   print("Mean: ", score["mean"])
454   print("Variance: ", score["variance"])
455   print("Max Distance: ", score["maxdist"])
456
457   print("----------------------------------------Image
        4----------------------------------------")
458   print("BEFORE")
459   score = errorbefore[3]
460   print("Mean: ", score["mean"])
461   print("Variance: ", score["variance"])
462   print("Max Distance: ", score["maxdist"])
463
464   print("AFTER")
465   score = errorafter[3]
466   print("Mean: ", score["mean"])
467   print("Variance: ", score["variance"])
468   print("Max Distance: ", score["maxdist"])
469
470
471 if __name__ == "__main__":
472   imageset1 = getimages(r"hw08/HW8-Files/Dataset1")
473   imageset1copy = [image.copy() for image in imageset1]
474
475   imageset2 = getimages(r"hw08/HW8-Files/Dataset2")
476   imageset2copy = [image.copy() for image in imageset2]
477   imageset2copy = [cv2.resize(image, (480, 640)) for image in
        imageset2copy]
478
479   dataset = 1
480
481   if(dataset == 1):
482     main(imageset1copy, r=0.5, nms=0.24, hough=60, data="data1")
483   else:
484     main(imageset2copy, r=1, nms=0.24, hough=50, data="data2")
```

Listing 1: The Source Code