ECE661 Computer Vision HW8

Chengjun Guo
guo456@purdue.edu

14 November 2022

1 Theory question

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.

No, we can not see Absolute Conic, it is imaginary and located in the imaginary plane π_∞. In a camera image, we can only see its projection in the image. This means that the plane corresponding to each pose will sample the π_∞ at two circular points. With these known points, we can find the intrinsic parameter.

2 Logic

My code includes 3 main parts. The first part is to find all intersections. For this part, I used houghlines to generate possible lines, my own algorithm to filter the lines and find the intersections. The houghlines function from opencv is using a voting algorithm to find the lines with high votes. Then I used theta to separate the vertical lines and horizontal lines. To filter them, I firstly use K means to generate expected number of centroids(8 for vertical, 10 for horizontal). Then I sort the centroids and calculate the difference between the nearest centroids. Since the lines should be isometric, I drop the image with difference variance larger than my threshold. With the qualified lines, the corners can be generated. The second part is to calculate the intrinsic parameter K and extrinsic parameters R, t lists. The last part is to update the LM refined parameters and apply visible comparison.

2.1 Corner detection

For the canny edge detection, I blur the image with a large kernel to filter out the noise first. Then the canny edge detection will find the intensity gradient

of the image. After we get the gradient magnitude and direction, the image is looped through with a window to remove points that are not local maximum.
For houghlines algorithm, it is using a voting system to vote the polar lines. The reason we are implementing another parameter space is that the original cartesian coordinates will approach infinity when processing the vertical lines. The polar coordinates for the lines will have a evenly distributed bins for voting. For this project, we applied cv2 houghlines function to generate candidate lines. For the lines filter, I used K means from scipy to generate 8 and 10 centroids for vertical and horizontal lines by their rho. Since it might not be precise, I sort the centroids and calculate the difference between the nearest centroids. Since the lines should be isometric, I drop the image with difference variance larger than my threshold. The remained images would be expected images.
With the k means result, the mean of each group is known. I use those lines to find intersection between vertical lines and horizontal lines.

2.2 Zhang's Algorithm

2.2.1 Calculation of intrinsic parameter

We first assume that the calibration pattern lays on the xy plane where z=0. In this case, the projection will becomes:

$$X = K[R|t] \begin{pmatrix} x \\ y \\ 0 \\ z \end{pmatrix}$$

We can define 3 by 3 matrix H:

$$H = K[r_1 r_2 t]$$

As we mentioned in the previous section, the absolute conic is sampled at two circular points $[1, i, 0]^T$ and $[1, -i, 0]^T$ by camera image. Thus, for each image, we can have two equations:

$$h_1^T \omega h_1 - h_2^T \omega h_2 = 0$$

and

$$h_1^T \omega h_2 = 0$$

where h_1 and h_2 are the first two columns of H. Since $\omega = K^{-T} K^{-1}$ is symmetric, we can represent it as:

$$\omega = \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{12} & \omega_{22} & \omega_{23} \\ \omega_{13} & \omega_{23} & \omega_{33} \end{bmatrix}$$

For each h_1 and h_2, we can define a vector:

$$\boldsymbol{v}_{ij} = \begin{pmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{pmatrix}$$

For ω, we can use a 6 by 1 vector to represent it:

$$\boldsymbol{b} = \begin{pmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{22} \\ \omega_{13} \\ \omega_{23} \\ \omega_{33} \end{pmatrix}$$

Thus, for each image, we can form:

$$Vb = \begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} = 0$$

For different images. we can vertically stack the V for each image. With linear least-squares, we can find b that can be reconstruct as ω. With ω, we can find K by the following formulas:

$$x_0 = \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2}$$

$$\lambda = \omega_{33} - \frac{\omega_{13}^2 + x_0\left(\omega_{12}\omega_{13} - \omega_{11}\omega_{23}\right)}{\omega_{11}}$$

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

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

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

$$y_0 = \frac{sx_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda}$$

K will be:

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.2.2 Calculation of extrinsic parameter for each image

As we got the intrinsic matrix K, since the homography is given, we get $K^{-1}H = [r_1 r_2 t]$. Since the rotation matrix have to be orthonormal, we normalize R and thus the formula for extrinsic parameter R will be:

$$\begin{aligned} \vec{r_1} &= \varepsilon K^{-1}\vec{h}_1 \\ \vec{r_2} &= \varepsilon K^{-1}\vec{h}_2 \\ \vec{r_3} &= \vec{r}_1 X \vec{r}_2 \\ \vec{t} &= \varepsilon K^{-1}\vec{h}_3 \\ \varepsilon &= \frac{1}{\|K^{-1}\vec{h}_1\|} \end{aligned}$$

2.2.3 Parameter Refinement

For the parameter refinement, the Levenberg Marquardt algorithm will be implemented. The cost function would be:

$$d^2_{\text{geom}} = \sum_i \sum_j \left\| \overrightarrow{x_{ij}} - K\left[R_i \mid t_i\right] \overrightarrow{x_M, j} \right\|^2$$

For the parameter update, since R only have 3 degree of freedom, we transform it into a 3 by 1 vector w for the parameter refinement:

$$\varphi = \cos^{-1}\left(\frac{\text{trace}(R) - 1}{2}\right)$$

$$\boldsymbol{w} = \frac{\varphi}{2\sin(\varphi)} \begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix}$$

Then we can pass the rodrigues representation $p = (K, w_i, t_i | i = 1..n)$ to the parameter refinement.
After the refinement, R can be reconstructed by:

$$\varphi = \|\boldsymbol{w}\|$$

$$[\boldsymbol{w}]_X = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix}$$

$$\boldsymbol{R} = e^{[\boldsymbol{w}]_X} = \boldsymbol{I}_{3\times 3} + \frac{\sin(\varphi)}{\varphi}[\boldsymbol{w}]_X + \frac{1 - \cos(\varphi)}{\varphi^2}[\boldsymbol{w}]^2_X$$

3 Data and images:

3.1 Dataset1

3.1.1 Data

K before LM:

$$\begin{bmatrix} 7.08429023e+02 & -2.51364620e-01 & 2.45433477e+02 \\ 0.00000000e+00 & 7.06353536e+02 & 3.18122856e+02 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

K after LM:

$$\begin{bmatrix} 712.25068496 & 1.59665469 & 320.27568448 \\ 0. & 710.55187632 & 242.43116991 \\ 0. & 0. & 1. \end{bmatrix}$$

Picture1 R before LM:

$$\begin{bmatrix} 0.76915501 & -0.20775349 & 0.60435012 \\ 0.23025114 & 0.97225924 & 0.0411871 \\ -0.59614175 & 0.10747304 & 0.79565354 \end{bmatrix}$$

Picture1 t before LM: [40.84655428 -780.13114812 2240.34933209]
Picture1 R after LM:

$$\begin{bmatrix} 0.78624859 & -0.18098151 & 0.59081203 \\ 0.20339487 & 0.97866382 & 0.02911438 \\ -0.58347552 & 0.09727699 & 0.80628376 \end{bmatrix}$$

Picture1 t after LM: [-186.76039921 -516.0272053 2156.29574598]

Picture2 R before LM:

$$\begin{bmatrix} 0.85029841 & -0.01094941 & -0.52618697 \\ -0.03912859 & 0.99570145 & -0.08394984 \\ 0.52484433 & 0.09197137 & 0.84621492 \end{bmatrix}$$

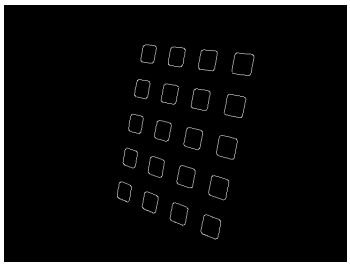
Picture2 t before LM: [47.77915899 -586.99502616 1847.14204786]
Picture2 R after LM:

$$\begin{bmatrix} 0.83311586 & -0.04139052 & -0.55154763 \\ -0.0047936 & 0.99661823 & -0.0820313 \\ 0.55307774 & 0.07098548 & 0.83010004 \end{bmatrix}$$

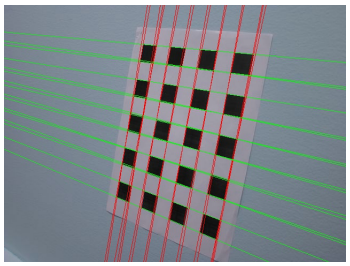
Picture2 t after LM: [-152.92350815 -407.41128196 1948.4596025]

3.1.2 Images

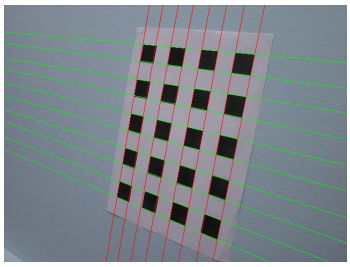
Corner detection:



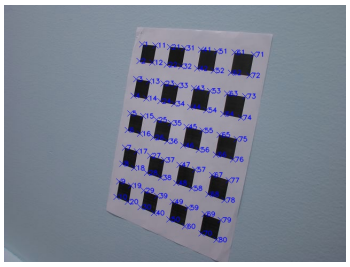
(a) canny edge



(b) hough lines

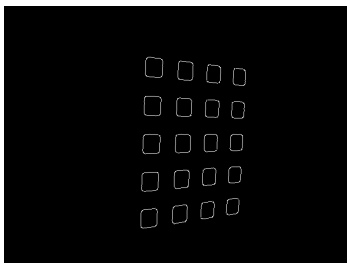


(c) filtered lines

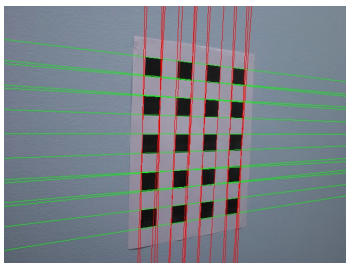


(d) intersections

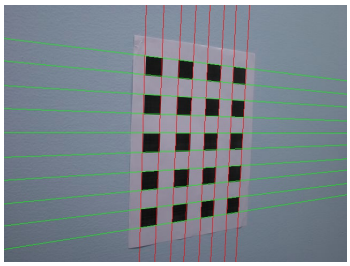
Figure 1: Picture 1



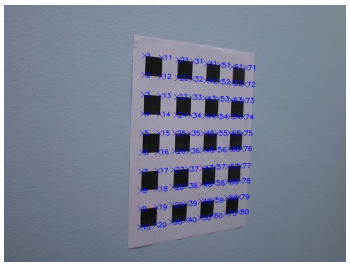
(a) canny edge



(b) hough lines



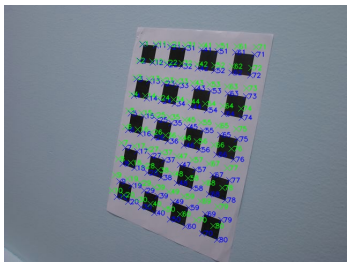
(c) filtered lines



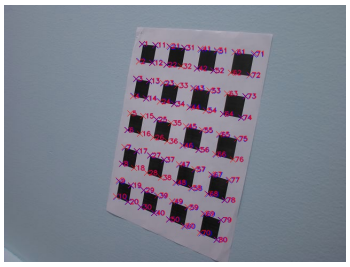
(d) intersections

Figure 2: Picture 2

Points reprojection:

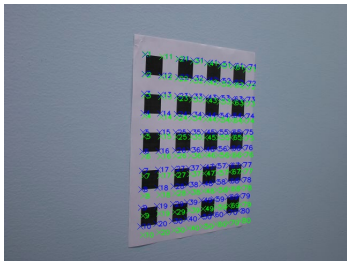


(a) without LM

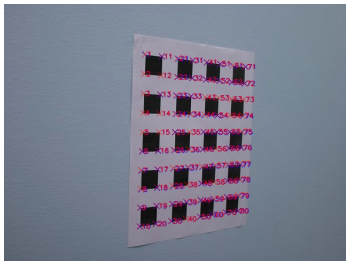


(b) with LM

Figure 3: Picture 1



(a) without LM



(b) with LM

Figure 4: Picture 2

3.2 Dataset2

3.2.1 Data

K before LM:

$$\begin{bmatrix} 1.28651078e+03 & 4.62926902e-01 & 8.46505222e+02 \\ 0.00000000e+00 & 1.29244914e+03 & 6.33409639e+02 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

K after LM:

$$\begin{bmatrix} 1.22692230e+03 & 6.53100705e-01 & 6.32554671e+02 \\ 0.00000000e+00 & 1.23384313e+03 & 8.31109873e+02 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

Picture3 R before LM:

$$\begin{bmatrix} -0.99490696 & 0.048005 & -0.08863218 \\ 0.04346193 & 0.99767493 & 0.05249568 \\ 0.09094616 & 0.04837619 & -0.99468012 \end{bmatrix}$$

Picture3 t before LM: [-17.37713415 -272.78518053 1237.3122035]
Picture3 R after LM:

$$\begin{bmatrix} -0.99514894 & 0.04535463 & -0.08730142 \\ 0.04103887 & 0.99787488 & 0.05061151 \\ 0.08941136 & 0.04678324 & -0.99489544 \end{bmatrix}$$

Picture3 t after LM: [191.49307336 -468.80312542 1198.99621678]

Picture5 R before LM:

$$\begin{bmatrix} -0.99556857 & 0.06145523 & -0.07117924 \\ 0.0600047 & 0.99794802 & 0.02234267 \\ 0.07240625 & 0.01797257 & -0.99721328 \end{bmatrix}$$

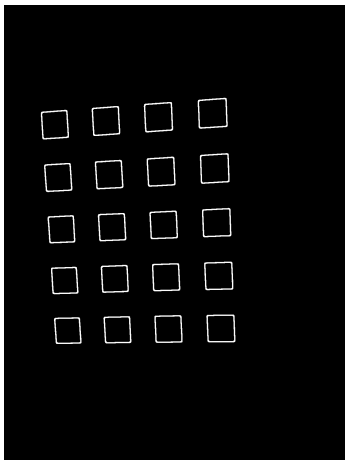
Picture5 t before LM: [134.99999502 -343.13230896 1285.15295894]
Picture5 R after LM:

$$\begin{bmatrix} -0.99596698 & 0.05759189 & -0.06879641 \\ 0.05627057 & 0.99819481 & 0.02099368 \\ 0.06988129 & 0.0170378 & -0.9974098 \end{bmatrix}$$

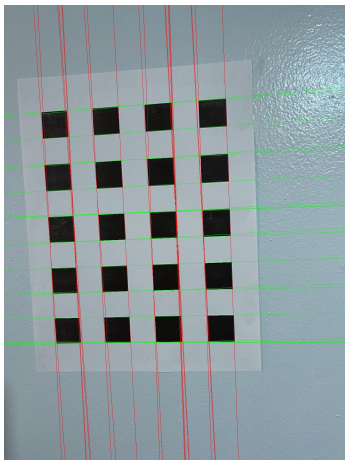
Picture5 t after LM: [353.35709587 -547.28809258 1243.88423983]

3.2.2 Images

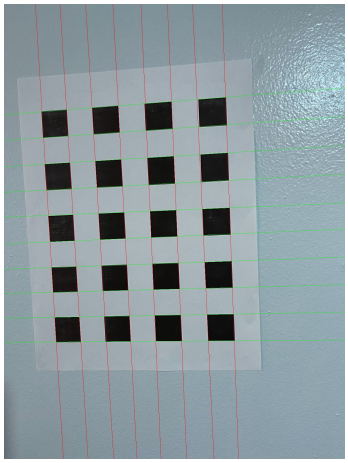
Corner detection:



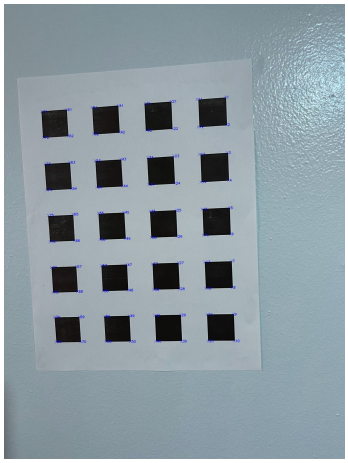
(a) canny edge



(b) hough lines

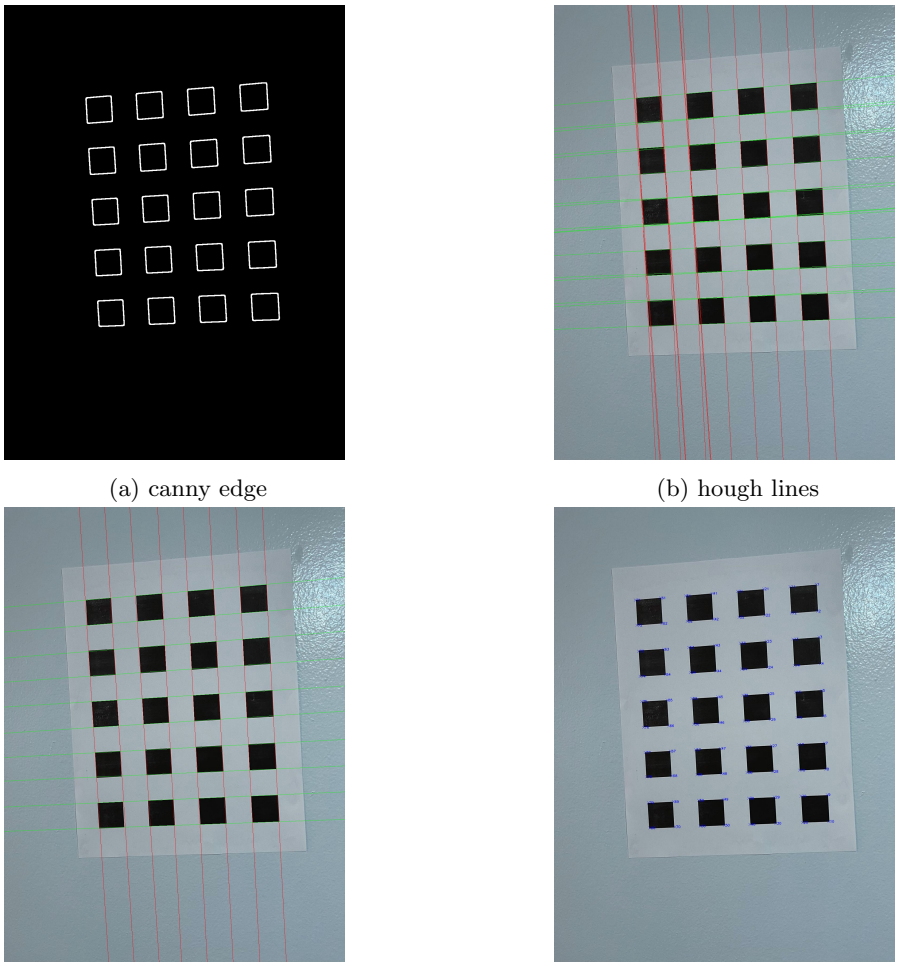


(c) filtered lines



(d) intersections

Figure 5: Picture 3



(a) canny edge

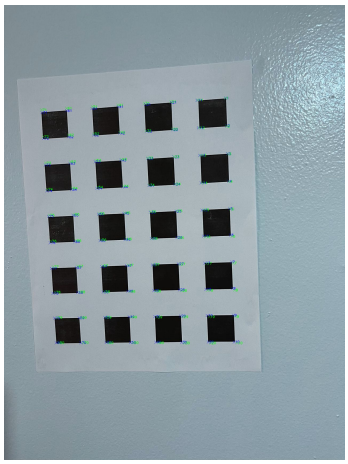
(b) hough lines

(c) filtered lines

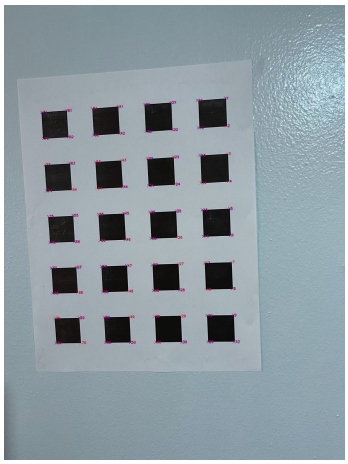
(d) intersections

Figure 6: Picture 5

Points reprojection:

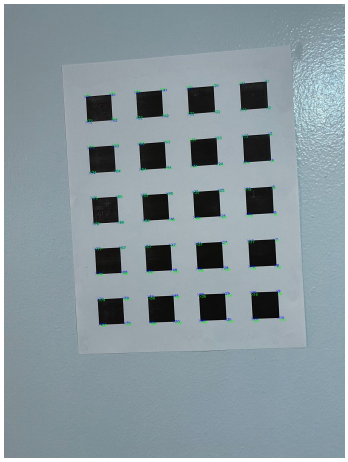


(a) without LM

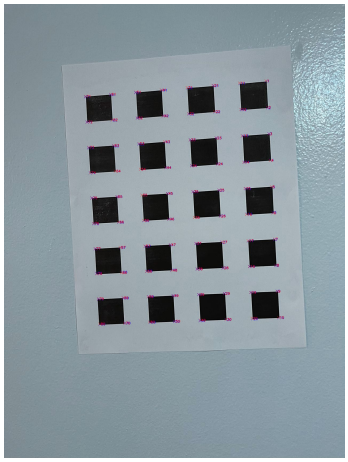


(b) with LM

Figure 7: Picture 3



(a) without LM



(b) with LM

Figure 8: Picture 5

4 Code

dataset1:

```
import cv2
import numpy as np
import os
os.environ["OMP_NUM_THREADS"] = '1'
import sklearn
import skimage
from sklearn import metrics
from sklearn.cluster import KMeans,DBSCAN
from skimage import data, io, filters, transform,color
from matplotlib import pyplot as plt
from scipy.optimize import least_squares


def corner_detection(img_name, image,save_img = 0):
    #generate lines
    image2 = image.copy()
    img2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    img2 = cv2.GaussianBlur(img2, (7, 7), 3)
    # img2 = cv2.Canny(img2, lower, upper)
    edges = cv2.Canny(img2, 50, 150, 3)
    if save_img == 1:
        cv2.imwrite(img_name + '_edge.jpg', edges)
    # lines = cv2.HoughLinesP(edges, rho = 1,theta = 1*np
        .pi/180,threshold = 25,minLineLength=150,
        maxLineGap=1000)
    lines = cv2.HoughLines(edges, 1, np.pi / 180, 40)
    # for line in lines:
    #     for x1, y1, x2, y2 in line:
    #         cv2.line(image, (x1, y1), (x2, y2), (0, 0,
        255), 1)
    vlines = []
    hlines = []
    for line in lines:
        rho, theta = line[0]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 1000 * (-b))
        y1 = int(y0 + 1000 * (a))
        x2 = int(x0 - 1000 * (-b))
        y2 = int(y0 - 1000 * (a))
        if np.cos(theta) ** 2 > 0.5:
            cv2.line(image2, (x1, y1), (x2, y2), (0, 0,
                255), 1)
            vlines.append(line)
```

```
        else:
            cv2.line(image2, (x1, y1), (x2, y2), (0, 255,
                0), 1)
            hlines.append(line)
    if save_img == 1:
        cv2.imwrite(img_name + '_lines.jpg', image2)

    if len(vlines) < 8 or len(hlines) < 10:
        print(img_name + 'doesnt have enough samples')
        return None

    #filter lines
    image3 = image.copy()
    filtered_vline, filtered_hline = filter_lines(vlines,
        hlines)
    if save_img == 1:
        for line in filtered_vline:
            rho, theta = line[0]
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho
            x1 = int(x0 + 1000 * (-b))
            y1 = int(y0 + 1000 * (a))
            x2 = int(x0 - 1000 * (-b))
            y2 = int(y0 - 1000 * (a))
            cv2.line(image3, (x1, y1), (x2, y2), (0, 0,
                255), 1)
        for line in filtered_hline:
            rho, theta = line[0]
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho
            x1 = int(x0 + 1000 * (-b))
            y1 = int(y0 + 1000 * (a))
            x2 = int(x0 - 1000 * (-b))
            y2 = int(y0 - 1000 * (a))
            cv2.line(image3, (x1, y1), (x2, y2), (0, 255,
                0), 1)
        cv2.imwrite(img_name + '_filtered_lines.jpg',
            image3)


    #filter the pattern where variance of the pairwise
        difference is too large
```

```
    filtered_vline = sorted(filtered_vline, key=lambda i:
        i[0,0])
    filtered_hline = sorted(filtered_hline, key=lambda i:
        i[0,0])
    v_diff = np.diff(np.array(filtered_vline)[:,0,0])
    var_v_diff = np.var(v_diff, ddof=0)
    h_diff = np.diff(np.array(filtered_hline)[:,0,0])
    var_h_diff = np.var(h_diff, ddof=0)
    # print(img_name,var_v_diff,var_h_diff)
    if var_v_diff + var_h_diff > 100:
        print(img_name+' dont have good filtered lines')
        return None

    #get intersections
    image4 = image.copy()
    corners = []
    filtered_vline = sorted(filtered_vline, key=lambda i:
        i[0,0])
    filtered_hline = sorted(filtered_hline, key=lambda i:
        i[0,0])
    for line1 in filtered_vline:
        for line2 in filtered_hline:
            corners.append(get_intersection(line1,line2))
    if save_img==1:
        for count,corner in enumerate(corners):
            cv2.drawMarker(image4,(int(corner[0]),int(
                corner[1])),(255,0,0), markerType=1,
                markerSize=10, thickness=1)
            cv2.putText(image4,str(count+1),(int(corner
                [0]+4),int(corner[1]+4)),cv2.
                FONT_HERSHEY_SIMPLEX,0.4,(255,0,0),1,cv2.
                LINE_AA)
        cv2.imwrite(img_name + '_intersections.jpg',
            image4)
    return corners


def filter_lines(vlines,hlines):
    #filter lines through k mean cluster
    filtered_vline = []
    filtered_hline = []
    vlines = np.array(vlines)
    hlines = np.array(hlines)
    # print(vlines.shape)
    kmeans = KMeans(n_clusters=8).fit(vlines[:,0,0].
        reshape(-1,1))
```

```
    vline_group = kmeans.labels_
    for i in range(8):
        # print(vlines[np.where(vline_group == i)])
        line = np.mean(vlines[np.where(vline_group == i)
            ],axis=0)
        # print(lines)
        filtered_vline.append(line)

    hlines = np.array(hlines)
    kmeans = KMeans(n_clusters=10).fit(hlines[:,0,0].
        reshape(-1,1))
    hline_group = kmeans.labels_
    for i in range(10):
        line = np.mean(hlines[np.where(hline_group == i)
            ], axis=0)
        filtered_hline.append(line)
    return filtered_vline,filtered_hline


def get_intersection(line1,line2):
    rho1, theta1 = line1[0]
    rho2, theta2 = line2[0]
    l1_pt0 = np.array([rho1 * np.cos(theta1), rho1 * np.
        sin(theta1), 1.0])
    l1_pt1 = np.array([l1_pt0[0] + 100 * np.sin(theta1),
        l1_pt0[1]-100 * np.cos(theta1), 1.0])
    l2_pt0 = np.array([rho2 * np.cos(theta2), rho2 * np.
        sin(theta2), 1.0])
    l2_pt1 = np.array([l2_pt0[0] + 100 * np.sin(theta2),
        l2_pt0[1] - 100 * np.cos(theta2), 1.0])
    l1 = np.cross(l1_pt0,l1_pt1)
    l2 = np.cross(l2_pt0, l2_pt1)
    l1 = l1/l1[2]
    l2 = l2/l2[2]
    intersection = np.cross(l1,l2)
    return intersection/intersection[2]


def homography_calculate(X, X_prime):
    A = np.zeros((2 * len(X), 8))
    B = np.zeros((2 * len(X), 1))
    for i in range(len(X)):
        A[2 * i, 0] = X[i][0]
        A[2 * i, 1] = X[i][1]
        A[2 * i, 2] = 1
        A[2 * i, 6] = -X[i][0] * X_prime[i][0]
```

```
        A[2 * i, 7] = -X[i][1] * X_prime[i][0]

        A[2 * i + 1, 3] = X[i][0]
        A[2 * i + 1, 4] = X[i][1]
        A[2 * i + 1, 5] = 1
        A[2 * i + 1, 6] = -X[i][0] * X_prime[i][1]
        A[2 * i + 1, 7] = -X[i][1] * X_prime[i][1]
        B[2 * i] = X_prime[i][0]
        B[2 * i + 1] = X_prime[i][1]

    A_plus = np.matmul(np.linalg.pinv(np.matmul(A.T,A)),A
        .T)
    H_vec = np.ndarray.flatten(np.dot(A_plus, B))
    H = np.ones((3, 3))
    H[0, :] = H_vec[0:3]
    H[1, :] = H_vec[3:6]
    H[2, :2] = H_vec[6:8]
    return H


def ground_truth(grid_size = 100, hline_num = 10,
    vline_num = 8):
    x = np.linspace(0, grid_size * (vline_num - 1),
        vline_num)
    y = np.linspace(0, grid_size * (hline_num - 1),
        hline_num)
    yv, xv = np.meshgrid(y, x)
    A,B = xv.reshape((-1, 1)), yv.reshape((-1, 1))
    ans = np.concatenate([A,B], axis=1)
    return np.concatenate([xv.reshape((-1, 1)), yv.
        reshape((-1, 1))], axis=1)


def construct_vij(hi,hj):
    V_ij = np.zeros((6,1))
    V_ij[0] = hi[0]*hj[0]
    V_ij[1] = hi[0]*hj[1] + hi[1]*hj[0]
    V_ij[2] = hi[1]*hj[1]
    V_ij[3] = hi[2]*hj[0] + hi[0]*hj[2]
    V_ij[4] = hi[2]*hj[1] + hi[1]*hj[2]
    V_ij[5] = hi[2]*hj[2]
    return V_ij

def compute_omega(h_list):
    V = np.zeros((2*len(h_list),6))
    for count,h in enumerate(h_list):
```

```
            V[2 * count] = construct_vij(h[:,0],h[:,1]).T
            V[2 * count + 1] = construct_vij(h[:,0],h[:,0]).T
                - construct_vij(h[:,1],h[:,1]).T
    _,_,v = np.linalg.svd(V)
    b = v[-1]
    omega = np.array([[b[0],b[1],b[3]],[b[1],b[2],b[4]],[
        b[3],b[4],b[5]]])
    return omega

def compute_K(omega1):
    omega = np.array(np.copy(omega1))
    x_0 = (omega[0,1]*omega[0,2] - omega[0,0]*omega[1,2])
        /(omega[0,0]*omega[1,1] - omega[0,1]**2)
    lambd = omega[2,2] - (omega[0,2] ** 2 + x_0 * (omega
        [0,1] * omega[0,2] - omega[0,0] * omega[1,2]))/
        omega[0,0]
    alpha_x = np.sqrt(lambd/omega[0,0])
    alpha_y = np.sqrt(lambd * omega[0,0]/(omega[0,0] *
        omega[1,1] - omega[0,1] ** 2))
    s = -(omega[0,1] * alpha_y * (alpha_x ** 2))/lambd
    y_0 = (s * x_0 / alpha_y) - omega[0,2] * (alpha_x **
        2) / lambd
    K = np.array([[alpha_x, s, x_0],[0,alpha_y,y_0
        ],[0,0,1]])
    return K

def compute_RT(H, K):
    r12_t = np.dot(np.linalg.inv(K), H)
    scale_factor = 1 / np.linalg.norm(r12_t[:,0])
    R = np.zeros((3,3))
    R[:,0] = scale_factor * r12_t[:,0]
    R[:,1] = scale_factor * r12_t[:,1]
    R[:,2] = np.cross(R[:,0],R[:,1])
    t = scale_factor * r12_t[:,2]
    u,_,v = np.linalg.svd(R)
    return np.dot(u,v), t

def construct_param(R,t):
    phi = np.arccos((np.trace(R)-1)/2)
    w = phi / (2 * np.sin(phi)) * np.array([R[2,1] - R
        [1,2], R[0,2] - R[2,0], R[1,0] - R[0,1]])
    return np.array([w[0], w[1], w[2],t[0],t[1],t[2]])

def reconstruct_R(w):
    phi = np.linalg.norm(w)
```

```
    w_x = 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 * w_x + (1-np.cos(phi
        ))/(phi ** 2) * np.linalg.matrix_power(w_x,2)
    return R


def cost(p_list,corner_list,gt):
    projected_list = []
    K = np.array([[p_list[0], p_list[1], p_list[2]], [0,
        p_list[3], p_list[4]], [0, 0, 1]])
    p_list = p_list[5:].reshape(-1, 6)
    for p in p_list:
        w = np.array([p[0],p[1],p[2]])
        t = np.array([p[3],p[4],p[5]])
        R = reconstruct_R(w)
        projected=apply_proj(K,R,t,gt)
        projected_list.append(projected)
    gt_corners = np.concatenate(corner_list)[:,:2]
    proj = np.concatenate(projected_list)
    return (proj - gt_corners).flatten()


def apply_proj(K,R,t,source):
    Rt = np.concatenate([R[:,0].reshape((-1,1)),R[:,1].
        reshape((-1,1)),t.reshape((-1,1))], axis=1)
    H = np.dot(K,Rt)
    if source.shape[1] == 2:
        X_vec = np.insert(source, 2, 1, axis=1)
    else:
        X_vec = np.array(source)
    X_prime_cal = np.matmul(H, X_vec.T).T
    X_prime_cal = X_prime_cal / X_prime_cal[:, 2].reshape
        (-1, 1)
    return X_prime_cal[:,:2]



if __name__ == '__main__':
    path = 'Dataset1/'
    # two sample images
    img_name = 'Pic_1'
    img = io.imread(path+img_name+'.jpg')
    corners1 = corner_detection(img_name,img,save_img=1)
    img_name = 'Pic_2'
    img = io.imread(path+img_name+'.jpg')
```

```
corners2 = corner_detection(img_name,img,save_img=1)

# path1 = 'test/'
# for i in range(40):
#     img_name = 'Pic_'+str(i+1)
#     img = io.imread(path+img_name+'.jpg')
#     corners = corner_detection(path1+img_name,img,
    save_img=1)

gt = ground_truth()
H_list = []
invalid_index = []
corner_list = []
for i in range(40):
    img_name = 'Pic_'+str(i+1)
    img = io.imread(path+img_name+'.jpg')
    corners = corner_detection(img_name,img)
    if corners == None:
        invalid_index.append(i)
        continue
    corners = np.array(corners).astype(int)
    corner_list.append(corners)
    H_list.append(homography_calculate(gt,corners))
omega = compute_omega(H_list)
# print(omega)
K = compute_K(omega)
print(K)
R_list = []
t_list = []
for h in H_list:
    R, t = compute_RT(h, K)
    R_list.append(R)
    t_list.append(t)
print(R_list[0])
print(R_list[1])
print(t_list[0])
print(t_list[1])
param_list = [K[0,0], K[0,1], K[0,2],K[1,1], K[1,2]]
refined_param_list = []
for R,t,corner in zip(R_list,t_list,corner_list):
    param = construct_param(R, t)
    for i in param:
        param_list.append(i)
param_list = np.array(param_list)
# loss_before_LM = cost(param_list,corner_list,gt)
```

```
# print('loss before:',loss_before_LM,'sum:',np.sum(
    loss_before_LM))
sol = least_squares(cost,param_list,args=[corner_list
    ,gt],method='lm')
refined_p_list = sol.x
# loss_after_LM = cost(refined_p_list, corner_list,
    gt)
# print('loss after:', loss_after_LM,'sum:',np.sum(
    loss_after_LM))
refined_K = np.array([[refined_p_list[0],
    refined_p_list[1], refined_p_list[2]], [0,
    refined_p_list[3], refined_p_list[4]], [0, 0, 1]])
refined_Rt_list = refined_p_list[5:].reshape(-1, 6)
refined_R_list = []
refined_t_list = []
for p in refined_Rt_list:
    w = np.array([p[0],p[1],p[2]])
    t = np.array([p[3],p[4],p[5]])
    R = reconstruct_R(w)
    refined_R_list.append(R)
    refined_t_list.append(t)
print(refined_K)
print(refined_R_list[0])
print(refined_R_list[1])
print(refined_t_list[0])
print(refined_t_list[1])
i = 0
for cnt in range(40):
    if cnt in invalid_index:
        continue
    img_name = 'Pic_' + str(cnt + 1)
    img = io.imread(path + img_name + '.jpg')
    refined_corner = apply_proj(refined_K,
        refined_R_list[i],refined_t_list[i],gt)
    unrefined_corner = apply_proj(K,R_list[i],t_list[
        i],gt)
    for count,corner in enumerate(corner_list[i]):
        cv2.drawMarker(img,(int(corner[0]),int(corner
            [1])),(255,0,0), markerType=1, markerSize
            =10, thickness=1)
        cv2.putText(img,str(count+1),(int(corner
            [0]+4),int(corner[1]+4)),cv2.
            FONT_HERSHEY_SIMPLEX,0.4,(255,0,0),1,cv2.
            LINE_AA)
    img2 = img.copy()
    for count, corner in enumerate(refined_corner):
```

```
            cv2.drawMarker(img, (int(corner[0]), int(
                corner[1]))), (0, 0, 255), markerType=1,
                markerSize=10, thickness=1)
            cv2.putText(img, str(count + 1), (int(corner
                [0] + 4), int(corner[1] + 4)), cv2.
                FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1,
                 cv2.LINE_AA)
        cv2.imwrite('test2/' + img_name + '_with_LM.jpg',
            img)

        for count, corner in enumerate(unrefined_corner):
            cv2.drawMarker(img2, (int(corner[0]), int(
                corner[1]))), (0, 255, 0), markerType=1,
                markerSize=10, thickness=1)
            cv2.putText(img2, str(count + 1), (int(corner
                [0] + 4), int(corner[1] + 4)), cv2.
                FONT_HERSHEY_SIMPLEX, 0.4, (0, 255, 0), 1,
                 cv2.LINE_AA)
        cv2.imwrite('test2/' + img_name + '_without_LM.
            jpg', img2)

        i += 1
```

dataset2:

```
import cv2
import numpy as np
import os
os.environ["OMP_NUM_THREADS"] = '1'
import sklearn
import skimage
from sklearn import metrics
from sklearn.cluster import KMeans,DBSCAN
from skimage import data, io, filters, transform,color
from matplotlib import pyplot as plt
from scipy.optimize import least_squares


def corner_detection(img_name, image,save_img = 0):
    #generate lines
    image2 = image.copy()
    img2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
```

```
img2 = cv2.GaussianBlur(img2, (9, 9), 3)
# img2 = cv2.Canny(img2, lower, upper)
edges = cv2.Canny(img2, 50, 150, 3)
kernel = np.ones((5, 5), np.uint8)
edges = cv2.dilate(edges, kernel, iterations=1)
# edges = cv2.erode(edges, kernel, iterations=1)
if save_img == 1:
    cv2.imwrite(img_name + '_edge.jpg', edges)
# lines = cv2.HoughLinesP(edges, rho = 1,theta = 1*np
    .pi/180,threshold = 25,minLineLength=150,
    maxLineGap=1000)
lines = cv2.HoughLines(edges, 1, np.pi / 180, 250)
# for line in lines:
#     for x1, y1, x2, y2 in line:
#         cv2.line(image, (x1, y1), (x2, y2), (0, 0,
    255), 1)
vlines = []
hlines = []
for line in lines:
    rho, theta = line[0]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a * rho
    y0 = b * rho
    x1 = int(x0 + 2000 * (-b))
    y1 = int(y0 + 2000 * (a))
    x2 = int(x0 - 2000 * (-b))
    y2 = int(y0 - 2000 * (a))
    if np.cos(theta) ** 2 > 0.5:
        cv2.line(image2, (x1, y1), (x2, y2), (0, 0,
            255), 1)
        vlines.append(line)
    else:
        cv2.line(image2, (x1, y1), (x2, y2), (0, 255,
            0), 1)
        hlines.append(line)
if save_img == 1:
    cv2.imwrite(img_name + '_lines.jpg', image2)

if len(vlines) < 8 or len(hlines) < 10:
    print(img_name + 'doesnt have enough samples')
    return None

#filter lines
image3 = image.copy()
filtered_vline, filtered_hline = filter_lines(vlines,
```

```
    hlines)
if save_img == 1:
    for line in filtered_vline:
        rho, theta = line[0]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 2000 * (-b))
        y1 = int(y0 + 2000 * (a))
        x2 = int(x0 - 2000 * (-b))
        y2 = int(y0 - 2000 * (a))
        cv2.line(image3, (x1, y1), (x2, y2), (0, 0,
            255), 1)
    for line in filtered_hline:
        rho, theta = line[0]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 2000 * (-b))
        y1 = int(y0 + 2000 * (a))
        x2 = int(x0 - 2000 * (-b))
        y2 = int(y0 - 2000 * (a))
        cv2.line(image3, (x1, y1), (x2, y2), (0, 255,
            0), 1)
    cv2.imwrite(img_name + '_filtered_lines.jpg',
        image3)


#filter the pattern where variance of the pairwise
    difference is too large
filtered_vline = sorted(filtered_vline, key=lambda i:
    i[0,0])
filtered_hline = sorted(filtered_hline, key=lambda i:
    i[0,0])
v_diff = np.diff(np.array(filtered_vline)[:,0,0])
var_v_diff = np.var(v_diff, ddof=0)
h_diff = np.diff(np.array(filtered_hline)[:,0,0])
var_h_diff = np.var(h_diff, ddof=0)
# print(img_name,var_v_diff,var_h_diff)
if var_v_diff + var_h_diff > 100:
    print(img_name+' dont have good filtered lines')
    return None

#get intersections
```

```
    image4 = image.copy()
    corners = []
    filtered_vline = sorted(filtered_vline, key=lambda i:
        i[0,0])
    filtered_hline = sorted(filtered_hline, key=lambda i:
        i[0,0])
    for line1 in filtered_vline:
        for line2 in filtered_hline:
            corners.append(get_intersection(line1,line2))
    if save_img==1:
        for count,corner in enumerate(corners):
            cv2.drawMarker(image4,(int(corner[0]),int(
                corner[1])),(255,0,0), markerType=1,
                markerSize=10, thickness=1)
            cv2.putText(image4,str(count+1),(int(corner
                [0]+4),int(corner[1]+4)),cv2.
                FONT_HERSHEY_SIMPLEX,0.4,(255,0,0),1,cv2.
                LINE_AA)
        cv2.imwrite(img_name + '_intersections.jpg',
            image4)
    return corners


def filter_lines(vlines,hlines):
    #filter lines through k mean cluster
    filtered_vline = []
    filtered_hline = []
    vlines = np.array(vlines)
    hlines = np.array(hlines)
    # print(vlines.shape)
    kmeans = KMeans(n_clusters=8).fit(vlines[:,0,0].
        reshape(-1,1))
    vline_group = kmeans.labels_
    for i in range(8):
        # print(vlines[np.where(vline_group == i)])
        line = np.mean(vlines[np.where(vline_group == i)
            ],axis=0)
        # print(lines)
        filtered_vline.append(line)

    hlines = np.array(hlines)
    kmeans = KMeans(n_clusters=10).fit(hlines[:,0,0].
        reshape(-1,1))
    hline_group = kmeans.labels_
    for i in range(10):
        line = np.mean(hlines[np.where(hline_group == i)
```

```
            ], axis=0)
        filtered_hline.append(line)
    return filtered_vline,filtered_hline


def get_intersection(line1,line2):
    rho1, theta1 = line1[0]
    rho2, theta2 = line2[0]
    l1_pt0 = np.array([rho1 * np.cos(theta1), rho1 * np.
        sin(theta1), 1.0])
    l1_pt1 = np.array([l1_pt0[0] + 100 * np.sin(theta1),
        l1_pt0[1]-100 * np.cos(theta1), 1.0])
    l2_pt0 = np.array([rho2 * np.cos(theta2), rho2 * np.
        sin(theta2), 1.0])
    l2_pt1 = np.array([l2_pt0[0] + 100 * np.sin(theta2),
        l2_pt0[1] - 100 * np.cos(theta2), 1.0])
    l1 = np.cross(l1_pt0,l1_pt1)
    l2 = np.cross(l2_pt0, l2_pt1)
    l1 = l1/l1[2]
    l2 = l2/l2[2]
    intersection = np.cross(l1,l2)
    return intersection/intersection[2]


def homography_calculate(X, X_prime):
    A = np.zeros((2 * len(X), 8))
    B = np.zeros((2 * len(X), 1))
    for i in range(len(X)):
        A[2 * i, 0] = X[i][0]
        A[2 * i, 1] = X[i][1]
        A[2 * i, 2] = 1
        A[2 * i, 6] = -X[i][0] * X_prime[i][0]
        A[2 * i, 7] = -X[i][1] * X_prime[i][0]

        A[2 * i + 1, 3] = X[i][0]
        A[2 * i + 1, 4] = X[i][1]
        A[2 * i + 1, 5] = 1
        A[2 * i + 1, 6] = -X[i][0] * X_prime[i][1]
        A[2 * i + 1, 7] = -X[i][1] * X_prime[i][1]
        B[2 * i] = X_prime[i][0]
        B[2 * i + 1] = X_prime[i][1]

    A_plus = np.matmul(np.linalg.pinv(np.matmul(A.T,A)),A
        .T)
    H_vec = np.ndarray.flatten(np.dot(A_plus, B))
    H = np.ones((3, 3))
```

```
    H[0, :] = H_vec[0:3]
    H[1, :] = H_vec[3:6]
    H[2, :2] = H_vec[6:8]
    return H


def ground_truth(grid_size = 100, hline_num = 10,
    vline_num = 8):
    x = np.linspace(0, grid_size * (vline_num - 1),
        vline_num)
    y = np.linspace(0, grid_size * (hline_num - 1),
        hline_num)
    yv, xv = np.meshgrid(y, x)
    A,B = xv.reshape((-1, 1)), yv.reshape((-1, 1))
    ans = np.concatenate([A,B], axis=1)
    return np.concatenate([xv.reshape((-1, 1)), yv.
        reshape((-1, 1))], axis=1)


def construct_vij(hi,hj):
    V_ij = np.zeros((6,1))
    V_ij[0] = hi[0]*hj[0]
    V_ij[1] = hi[0]*hj[1] + hi[1]*hj[0]
    V_ij[2] = hi[1]*hj[1]
    V_ij[3] = hi[2]*hj[0] + hi[0]*hj[2]
    V_ij[4] = hi[2]*hj[1] + hi[1]*hj[2]
    V_ij[5] = hi[2]*hj[2]
    return V_ij

def compute_omega(h_list):
    V = np.zeros((2*len(h_list),6))
    for count,h in enumerate(h_list):
        V[2 * count] = construct_vij(h[:,0],h[:,1]).T
        V[2 * count + 1] = construct_vij(h[:,0],h[:,0]).T
            - construct_vij(h[:,1],h[:,1]).T
    _,_,v = np.linalg.svd(V)
    b = v[-1]
    omega = np.array([[b[0],b[1],b[3]],[b[1],b[2],b[4]],[
        b[3],b[4],b[5]]])
    return omega

def compute_K(omega1):
    omega = np.array(np.copy(omega1))
    x_0 = (omega[0,1]*omega[0,2] - omega[0,0]*omega[1,2])
        /(omega[0,0]*omega[1,1] - omega[0,1]**2)
    lambd = omega[2,2] - (omega[0,2] ** 2 + x_0 * (omega
```

```
        [0,1] * omega[0,2] - omega[0,0] * omega[1,2]))/
        omega[0,0]
    alpha_x = np.sqrt(lambd/omega[0,0])
    alpha_y = np.sqrt(lambd * omega[0,0]/(omega[0,0] *
        omega[1,1] - omega[0,1] ** 2))
    s = -(omega[0,1] * alpha_y * (alpha_x ** 2))/lambd
    y_0 = (s * x_0 / alpha_y) - omega[0,2] * (alpha_x **
        2) / lambd
    K = np.array([[alpha_x, s, x_0],[0, alpha_y, y_0
        ],[0,0,1]])
    return K

def compute_RT(H, K):
    r12_t = np.dot(np.linalg.inv(K), H)
    scale_factor = 1 / np.linalg.norm(r12_t[:,0])
    R = np.zeros((3,3))
    R[:,0] = scale_factor * r12_t[:,0]
    R[:,1] = scale_factor * r12_t[:,1]
    R[:,2] = np.cross(R[:,0],R[:,1])
    t = scale_factor * r12_t[:,2]
    u,_,v = np.linalg.svd(R)
    return np.dot(u,v), t

def construct_param(R,t):
    phi = np.arccos((np.trace(R)-1)/2)
    w = phi / (2 * np.sin(phi)) * np.array([R[2,1] - R
        [1,2], R[0,2] - R[2,0], R[1,0] - R[0,1]])
    return np.array([w[0], w[1], w[2],t[0],t[1],t[2]])

def reconstruct_R(w):
    phi = np.linalg.norm(w)
    w_x = 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 * w_x + (1-np.cos(phi
        ))/(phi ** 2) * np.linalg.matrix_power(w_x,2)
    return R


def cost(p_list,corner_list,gt):
    projected_list = []
    K = np.array([[p_list[0], p_list[1], p_list[2]], [0,
        p_list[3], p_list[4]], [0, 0, 1]])
    p_list = p_list[5:].reshape(-1, 6)
    for p in p_list:
        w = np.array([p[0],p[1],p[2]])
        t = np.array([p[3],p[4],p[5]])
```

```
            R = reconstruct_R(w)
            projected=apply_proj(K,R,t,gt)
            projected_list.append(projected)
        gt_corners = np.concatenate(corner_list)[:,:2]
        proj = np.concatenate(projected_list)
        return (proj - gt_corners).flatten()


def apply_proj(K,R,t,source):
        Rt = np.concatenate([R[:,0].reshape((-1,1)),R[:,1].
            reshape((-1,1)),t.reshape((-1,1))], axis=1)
        H = np.dot(K,Rt)
        if source.shape[1] == 2:
            X_vec = np.insert(source, 2, 1, axis=1)
        else:
            X_vec = np.array(source)
        X_prime_cal = np.matmul(H, X_vec.T).T
        X_prime_cal = X_prime_cal / X_prime_cal[:, 2].reshape
            (-1, 1)
        return X_prime_cal[:,:2]



if __name__ == '__main__':
        path = 'Dataset2/'
        # two sample images
        img_name = 'Pic_3'
        img = io.imread(path+img_name+'.jpg')
        corners1 = corner_detection(img_name+'_task2',img,
            save_img=1)
        img_name = 'Pic_5'
        img = io.imread(path+img_name+'.jpg')
        corners2 = corner_detection(img_name+'_task2',img,
            save_img=1)

        path1 = 'test3/'
        for i in range(29):
            img_name = 'Pic_'+str(i+1)
            img = io.imread(path+img_name+'.jpg')
            corners = corner_detection(path1+img_name,img,
                save_img=1)

        gt = ground_truth()
        H_list = []
        invalid_index = []
        corner_list = []
```

```
for i in range(29):
    img_name = 'Pic_'+str(i+1)
    img = io.imread(path+img_name+'.jpg')
    corners = corner_detection(img_name,img)
    if corners == None:
        invalid_index.append(i)
        continue
    corners = np.array(corners).astype(int)
    corner_list.append(corners)
    H_list.append(homography_calculate(gt,corners))
omega = compute_omega(H_list)
# print(omega)
K = compute_K(omega)
print(K)
R_list = []
t_list = []
for h in H_list:
    R, t = compute_RT(h, K)
    R_list.append(R)
    t_list.append(t)
print(R_list[1])
print(R_list[2])
print(t_list[1])
print(t_list[2])
param_list = [K[0,0], K[0,1], K[0,2],K[1,1], K[1,2]]
refined_param_list = []
for R,t,corner in zip(R_list,t_list,corner_list):
    param = construct_param(R, t)
    for i in param:
        param_list.append(i)
param_list = np.array(param_list)
# loss_before_LM = cost(param_list,corner_list,gt)
# print('loss before:',loss_before_LM,'sum:',np.sum(
    loss_before_LM))
sol = least_squares(cost,param_list,args=[corner_list
    ,gt],method='lm')
refined_p_list = sol.x
# loss_after_LM = cost(refined_p_list, corner_list,
    gt)
# print('loss after:', loss_after_LM,'sum:',np.sum(
    loss_after_LM))
refined_K = np.array([[refined_p_list[0],
    refined_p_list[1], refined_p_list[2]], [0,
    refined_p_list[3], refined_p_list[4]], [0, 0, 1]])
refined_Rt_list = refined_p_list[5:].reshape(-1, 6)
refined_R_list = []
```

```
refined_t_list = []
for p in refined_Rt_list:
    w = np.array([p[0],p[1],p[2]])
    t = np.array([p[3],p[4],p[5]])
    R = reconstruct_R(w)
    refined_R_list.append(R)
    refined_t_list.append(t)
print(refined_K)
print(refined_R_list[1])
print(refined_R_list[2])
print(refined_t_list[1])
print(refined_t_list[2])
i = 0
for cnt in range(29):
    if cnt in invalid_index:
        continue
    img_name = 'Pic_' + str(cnt + 1)
    img = io.imread(path + img_name + '.jpg')
    refined_corner = apply_proj(refined_K,
        refined_R_list[i],refined_t_list[i],gt)
    unrefined_corner = apply_proj(K,R_list[i],t_list[
        i],gt)
    for count,corner in enumerate(corner_list[i]):
        cv2.drawMarker(img,(int(corner[0]),int(corner
            [1])),(255,0,0), markerType=1, markerSize
            =10, thickness=1)
        cv2.putText(img,str(count+1),(int(corner
            [0]+4),int(corner[1]+4)),cv2.
            FONT_HERSHEY_SIMPLEX,0.4,(255,0,0),1,cv2.
            LINE_AA)
    img2 = img.copy()
    for count, corner in enumerate(refined_corner):
        cv2.drawMarker(img, (int(corner[0]), int(
            corner[1])), (0, 0, 255), markerType=1,
            markerSize=10, thickness=1)
        cv2.putText(img, str(count + 1), (int(corner
            [0] + 4), int(corner[1] + 4)), cv2.
            FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1,
             cv2.LINE_AA)
    cv2.imwrite('test3/' + img_name + '_with_LM.jpg',
        img)

    for count, corner in enumerate(unrefined_corner):
        cv2.drawMarker(img2, (int(corner[0]), int(
            corner[1])), (0, 255, 0), markerType=1,
            markerSize=10, thickness=1)
```

```
        cv2.putText(img2, str(count + 1), (int(corner
            [0] + 4), int(corner[1] + 4)), cv2.
            FONT_HERSHEY_SIMPLEX, 0.4, (0, 255, 0), 1,
             cv2.LINE_AA)
    cv2.imwrite('test3/' + img_name + '_without_LM.
        jpg', img2)

    i += 1
```