PURDUE UNIVERSITY

Elmore Family School of Electrical and Computer Engineering

Computer Vision

Homework 8

Adithya Sineesh

Email : asineesh@purdue.edu

Submitted: November 14, 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, in reality we cannot see ω in the camera image. We know that the points x on the conic have to satisfy $x^T \omega x$. However since $K^T K^1$ is positive definite, $x^T \omega x > 0$ for all real values of x. This means that all the pixels on the image of the absolute conic have imaginary coordinates.

From the above equation of ω we can see that it only depends on K i.e. the intrinsic camera parameters and not on its translation or rotation with respect to the camera. Every plane in the world coordinates intersects the absolute conic Ω at exactly two points (the circular points of that plane) which also lie on ω. As ω is a symmetric matrix, with a minimum of 3 camera positions we can find the value of ω and thereby K.

2 Theory

In this homework, Zhang's algorithm will be implemented for camera calibration. Let us first look at the concepts behind them.

2.1 Canny Edge Detection

I initially converted all the images to grayscale and applied a 3x3 Guassian kernel to them. Then I applied the OpenCV implementation of the Canny edge detector to obtain the edges of the image.

2.2 Hough Transform Line Detection

I applied the OpenCV implementation of the Hough transform to obtain all the lines from the edges of the image. I then separate them into horizontal and vertical lines based on the theta value returned by the function . If $\frac{\pi}{4} < \theta < \frac{3\pi}{4}$ it is a horizontal line, else it is a vertical line

However there are a lot of extraneous lines. So, I compute the distances among the lines and if it is less than a threshold, I group them together and take the resultant as their average. I then calculate all the points of intersection of the resultant 8 vertical and 10 horizontal lines (ideally).

2.3 Calculating the image of the absolute conic

We can write the relationship between the homogeneous pixel coordinates $\vec{x} = (x, y, w)^T$ and the homogeneous world coordinates $\vec{X} = (X, Y, Z, W)^T$ as follows:

$$\vec{x} = K[R|T]\vec{X}$$

where, K is the intrinsic camera matrix, R is the rotation matrix and t is the translation vector. For $Z = 0$ in the world coordinates, the above equation becomes:

$$\vec{x} = H\vec{X_M}$$

where,

$H = [\vec{h_1}, \vec{h_2}, \vec{h_3}]$

$\vec{X_M} = (X, Y, W)^T$

The image of the two circular points $H\vec{I}$ and $H\vec{J}$ that lie on the image of the absolute conic ω is given as follows

$$(H\vec{I})^T\omega(H\vec{I}) = ([\vec{h_1}, \vec{h_2}, \vec{h_3}] \cdot (1, i, 0)^T)^T\omega([\vec{h_1}, \vec{h_2}, \vec{h_3}] \cdot (1, i, 0)^T)$$
$$(H\vec{J})^T\omega(H\vec{J}) = ([\vec{h_1}, \vec{h_2}, \vec{h_3}] \cdot (1, -i, 0)^T)^T\omega([\vec{h_1}, \vec{h_2}, \vec{h_3}] \cdot (1, -i, 0)^T)$$

These equations can be simplified as

$$(H\vec{I})^T\omega(H\vec{I}) = (\vec{h_1} + i\vec{h_2})^T\omega(\vec{h_1} + i\vec{h_2})$$
$$(H\vec{J})^T\omega(H\vec{J}) = (\vec{h_1} - i\vec{h_2})^T\omega(\vec{h_1} - i\vec{h_2})$$

Separating the real and imaginary parts, we get the following equations

$$\vec{h_1}^T\omega\vec{h_2} = 0 \tag{1}$$

$$\vec{h_1}^T\omega\vec{h_1} - \vec{h_2}^T\omega\vec{h_2} = 0 \tag{2}$$

ω is a matrix which can be written as

$$\begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{pmatrix}$$

We can now construct a vector of knowns $\vec{V_{ij}}$ as

$$\begin{pmatrix} \mathrm{h}_{i1}h_{j1} \\ \mathrm{h}_{i1}h_{j2} + h_{i2}h_{j1} \\ \mathrm{h}_{i2}h_{j2} \\ \mathrm{h}_{i3}h_{j1} + h_{i1}h_{j3} \\ \mathrm{h}_{i3}h_{j2} + h_{i2}h_{j3} \\ \mathrm{h}_{i3}h_{j3} \end{pmatrix}$$

Since ω is a symmetric matrix, it will have 6 unknowns. A vector of unknowns $\vec{b}$ is therefore created as

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

Using these two vectors, we can rewrite equation 1 as $\vec{V_{12}}^T\vec{b} = 0$ and equation 2 as $(\vec{V_{11}} - \vec{V_{22}})^T\vec{b} = 0$. They can be expressed together as $V\vec{b} = 0$ where V is given by

$$\begin{pmatrix} \vec{V_{12}}^T \\ (\vec{V_{11}} - \vec{V_{22}})^T \end{pmatrix}$$

Each $V\vec{b} = 0$ gives us two equations and since there are 6 unknowns in $\vec{b}$, we need at least 3 camera positions. We can stack them up as follows for n camera positions.

$$\begin{pmatrix} \mathrm{V}_1 \\ \mathrm{V}_2 \\ \mathrm{V}_3 \\ : \\ \mathrm{V}_n \end{pmatrix} \times b = 0$$

We can now solve the above equation for b using linear least square. This will give us the values of the image of the absolute conic ω.

2.4 Calculating the intrinsic parameters of the camera

We know that $\omega = K^T K^1$ and K is

$$\begin{pmatrix} \alpha_x & \mathrm{s} & \mathrm{x}_o \\ 0 & \alpha_y & \mathrm{y}_o \\ 0 & 0 & 1 \end{pmatrix}$$

From Zhang's paper, we can calculate the above values from ω as follows

$$y_o = \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2}$$
$$\lambda = \omega_{33} - \frac{\omega_{13}^2 + y_o(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}}$$
$$\alpha_x = \sqrt{\frac{\lambda}{\omega_{11}}}$$
$$\alpha_y = \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}}$$
$$s = -\frac{\omega_{11}\alpha_x^2\alpha_y}{\lambda}$$
$$x_o = \frac{sy_o}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda}$$

Since ω is homogeneous and K is not homogeneous, λ takes care of the scaling between them.

2.5 Calculating the extrinsic parameters of the camera

We know the following relation

$$K^{-1}[\vec{h_1}, \vec{h_2}, \vec{h_3}] = [\vec{r_1}, \vec{r_2}, \vec{t}]$$

where $R = [\vec{r_1}, \vec{r_2}, \vec{r_3}]$ and $H = [\vec{h_1}, \vec{h_2}, \vec{h_3}]$.

Since all the columns of the R matrix must be of unit magnitude, we calculate the scale factor as follows:

$$\xi = \frac{1}{||K^{-1}\vec{h_1}||}$$

Now we can find all the values of R and t as follows:

$$\vec{r_1} = \xi K^{-1}\vec{h_1}$$
$$\vec{r_2} = \xi K^{-1}\vec{h_2}$$

$$\vec{r_3} = \vec{r_1} \times \vec{r_2}$$
$$\vec{t} = \xi K^{-1} \vec{h_3}$$

However R may still not be orthonormal. So to obtain R_{ortho} we perform SVD as follows:

$$U, D, V^T = SVD(R)$$
$$R_{ortho} = UV^T$$

2.6 Refining the calibration parameters

Even though we have obtained the values for K, R and t, they are not optimal. To improve their accuracy we make use of Levenberg-Marquadt (LM) optimization. The cost function used by this non-linear method is the squares of the Euclidean distance which we have to minimize and is given as follows:

$$d^2 = \Sigma_i \Sigma_j ||\vec{x_{ij}} - K[R_i|t_i]\vec{X_{Mj}}||^2$$

where, $\vec{X_{Mj}}$ is the j^{th} point on the calibration and $\vec{x_{ij}}$ is the actual corresponding point in the i^{th} camera position.

However there is an issue. R is a 3x3 matrix but has only 3 degrees of freedom. But if we try to optimize R in this form, the final result can have more than 3 degrees of freedom. So we use the Rodrigues representation for the 3x3 R matrix during its optimization and the normal representation while calculating the cost.

2.7 Rodrigues Representation

1. As R is of the form

$$\begin{pmatrix} \mathrm{r}_{11} & \mathrm{r}_{12} & \mathrm{r}_{13} \\ \mathrm{r}_{21} & \mathrm{r}_{22} & \mathrm{r}_{23} \\ \mathrm{r}_{31} & \mathrm{r}_{32} & \mathrm{r}_{33} \end{pmatrix}$$

its Rodrigues representation $\vec{w} = (w_x, w_y, w_z)^T$ is

$$\frac{\phi}{\sin 2\phi} \begin{pmatrix} \mathrm{r}_{32} - r_{23} \\ \mathrm{r}_{13} - r_{31} \\ \mathrm{r}_{21} - r_{12} \end{pmatrix}$$

where $\phi = cos^{-1}(\frac{trace(R)-1}{2})$.

2. We can transform $\vec{w}$ back to R by first constructing the W matrix as follows

$$\begin{pmatrix} 0 & \text{-}\mathrm{w}_z & \mathrm{w}_y \\ \mathrm{w}_z & 0 & \text{-}\mathrm{w}_x \\ \text{-}\mathrm{w}_y & \mathrm{w}_x & 0 \end{pmatrix}$$

Then we use the equation

$$R = I_{3\times 3} + \frac{\sin\phi}{\phi} W + \frac{1-\cos\phi}{\phi^2} W^2$$

where $\phi = ||\vec{w}||$

3 Experiment

3.1 Task 1

For Task 1, we were given a dataset of 40 images. I used 30 images among them to compute the intrinsic and extrinsic calibration parameters. I have chosen the 5th image of the dataset to be the fixed view.

3.2 Task 2

For Task 1, I constructed a dataset of 24 images. I used 16 images among them to compute the intrinsic and extrinsic calibration parameters. I have chosen the 1st image of the dataset to be the fixed view.

4 Results

4.1 Task1

4.1.1 Fixed Image

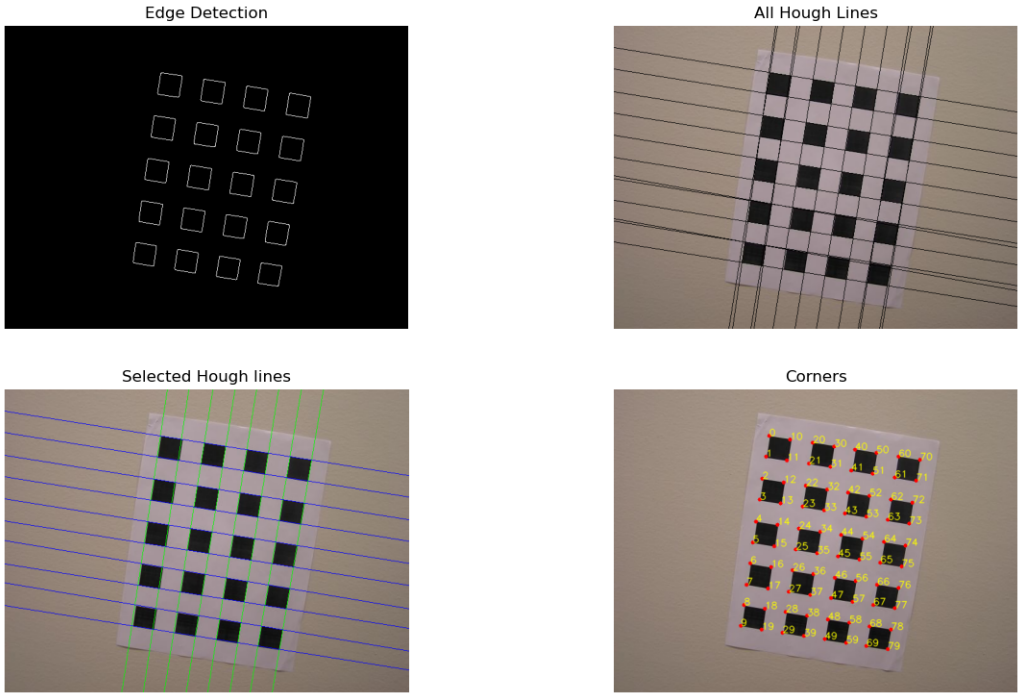


Figure 1: For the fixed image

4.1.2 Intrinsic Camera Matrix

```
The intrinsic camera caliberation parameter before LM refinement is
 [[704.55793689   1.3782471  245.57112836]
 [  0.         709.09265189 317.11587573]
 [  0.           0.           1.        ]]

The intrinsic camera caliberation parameter after LM refinement is
 [[735.43935418   0.69385808 246.12174319]
 [  0.         744.12941793 311.69053472]
 [  0.           0.           1.        ]]
```

Figure 2:

4.1.3 Image 1

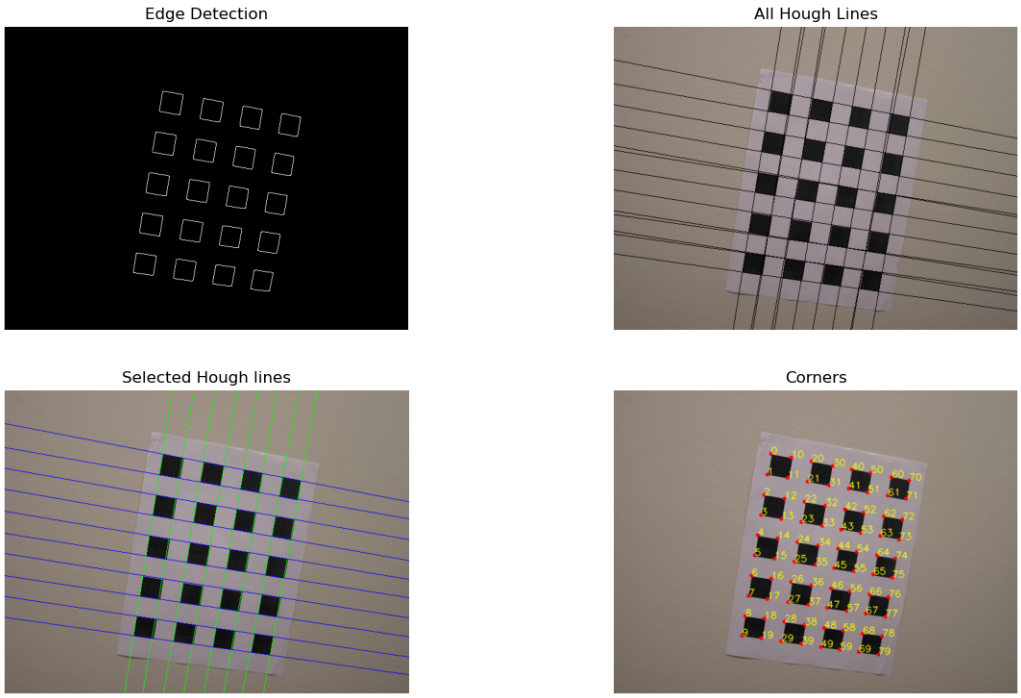


Figure 3: For the first image to be re-projected

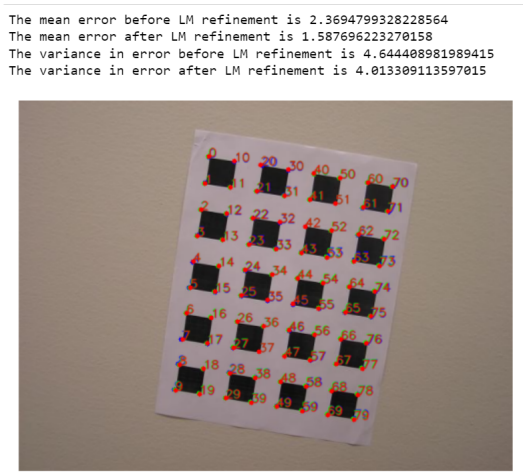


Figure 4: The first image re-projected to the fixed image. Blue are the actual points, Green are the re-projected points without LM and Red are the re-projected points with LM

```
The R matrix before LM refinement is
 [[ 0.984763    0.16258309 -0.06171361]
 [-0.16839072  0.98012319 -0.10489567]
 [ 0.04343268  0.11368937  0.99256654]]

The R matrix after LM refinement is
 [[ 0.98489652  0.16184145 -0.06153198]
 [-0.16829534  0.97833681 -0.12055608]
 [ 0.04068803  0.12909081  0.99079768]]

The t vector before LM refinement is
 [-44.1697478  -20.37440218 214.79102805]

The t vector after LM refinement is
 [-44.40192841 -18.84045809 224.79242679]
```

Figure 5: Extrinsic Camera Parameters

4.1.4 Image 2

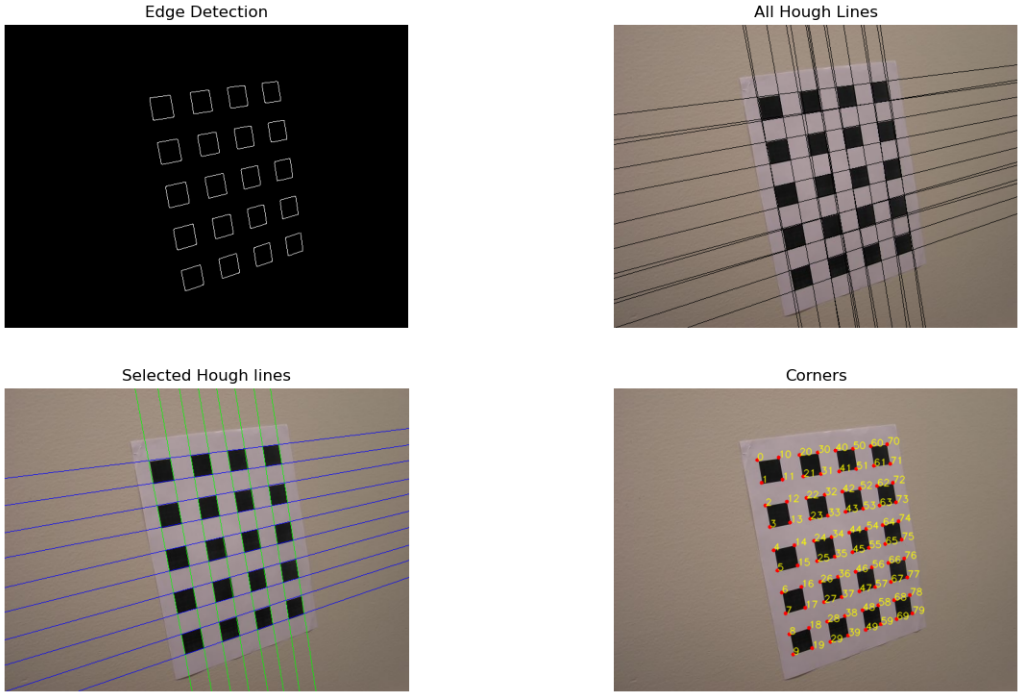


Figure 6: For the second image to be re-projected

```
The mean error before LM refinement is 5.025794169755225
The mean error after LM refinement is 1.6451253036778
The variance in error before LM refinement is 35.58838597737733
The variance in error after LM refinement is 2.854361976738941
```

Figure 7: The second image re-projected to the fixed image. Blue are the actual points, Green are the re-projected points without LM and Red are the re-projected points with LM

```
The R matrix before LM refinement is
 [[ 0.9789797  -0.20232252 -0.0257749 ]
 [ 0.16777061  0.87069164 -0.46233008]
 [ 0.11598177  0.44828749  0.88633321]]

The R matrix after LM refinement is
 [[ 0.97896591 -0.20231065 -0.02638443]
 [ 0.16810885  0.87313963 -0.45756595]
 [ 0.11560776  0.44350601  0.88878415]]

The t vector before LM refinement is
 [-36.44966214 -24.53481072 195.36188406]

The t vector after LM refinement is
 [-36.69976249 -22.87745425 204.2785367 ]
```

Figure 8: Extrinsic Camera Parameters

4.2 Task2

4.2.1 Fixed Image

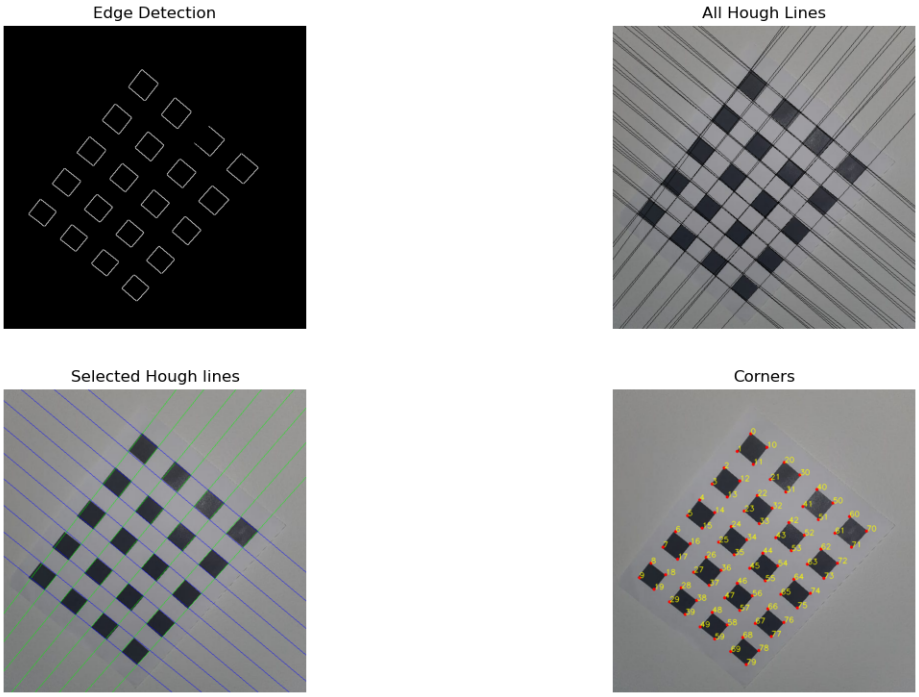


Figure 9: For the fixed image

4.2.2 Intrinsic Camera Matrix

```
The intrinsic camera caliberation parameter before LM refinement is
 [[653.37800437  -2.70180296 284.32783632]
 [  0.         651.75085841 340.83652843]
 [  0.           0.           1.        ]]

The intrinsic camera caliberation parameter after LM refinement is
 [[652.24746242  -1.46653519 284.76141222]
 [  0.         651.3547318  334.51810043]
 [  0.           0.           1.        ]]
```

Figure 10:

4.2.3 Image 1

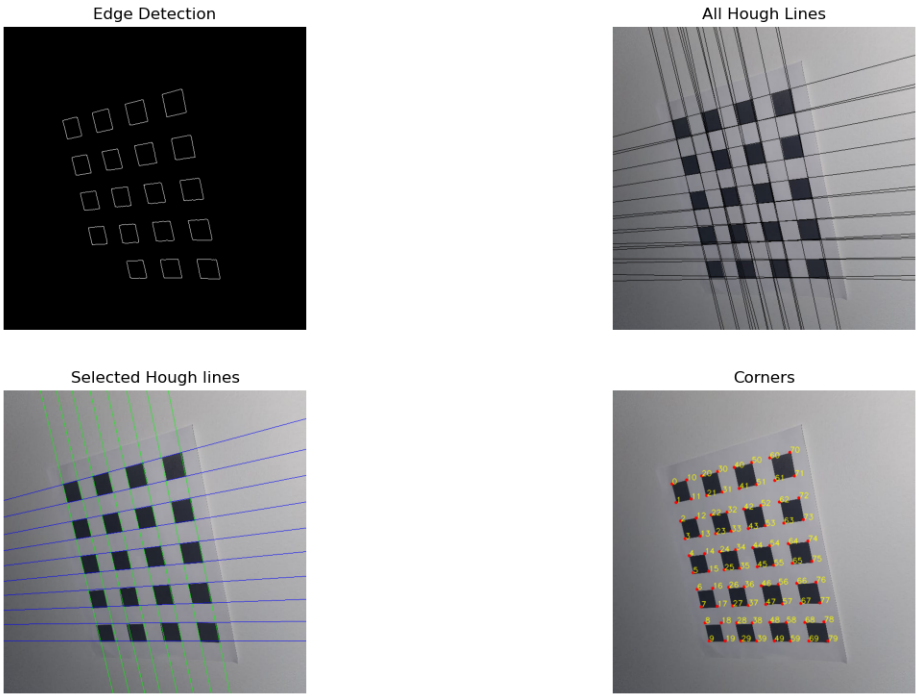


Figure 11: For the first image to be re-projected

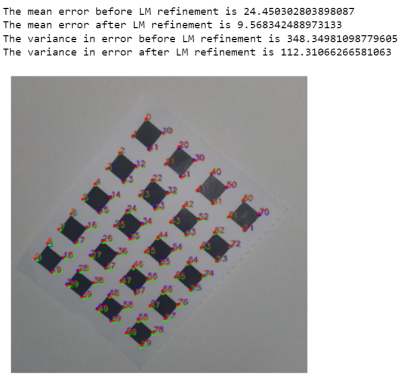


Figure 12: The first image re-projected to the fixed image. Blue are the actual points, Green are the re-projected points without LM and Red are the re-projected points with LM

```
The R matrix before LM refinement is
 [[ 0.97373011 -0.13328292 -0.18462214]
 [ 0.19563905  0.90455792  0.37881436]
 [ 0.11651193 -0.40498225  0.90687063]]

The R matrix after LM refinement is
 [[ 0.97071982 -0.12991232 -0.202054  ]
 [ 0.2023932   0.89537581  0.39666001]
 [ 0.12938324 -0.42594009  0.89545241]]

The t vector before LM refinement is
 [-26.7306836  -60.23068243 176.11273189]

The t vector after LM refinement is
 [-26.49844296 -58.10712862 175.0127035 ]
```

Figure 13: Extrinsic Camera Parameters

4.2.4 Image 2

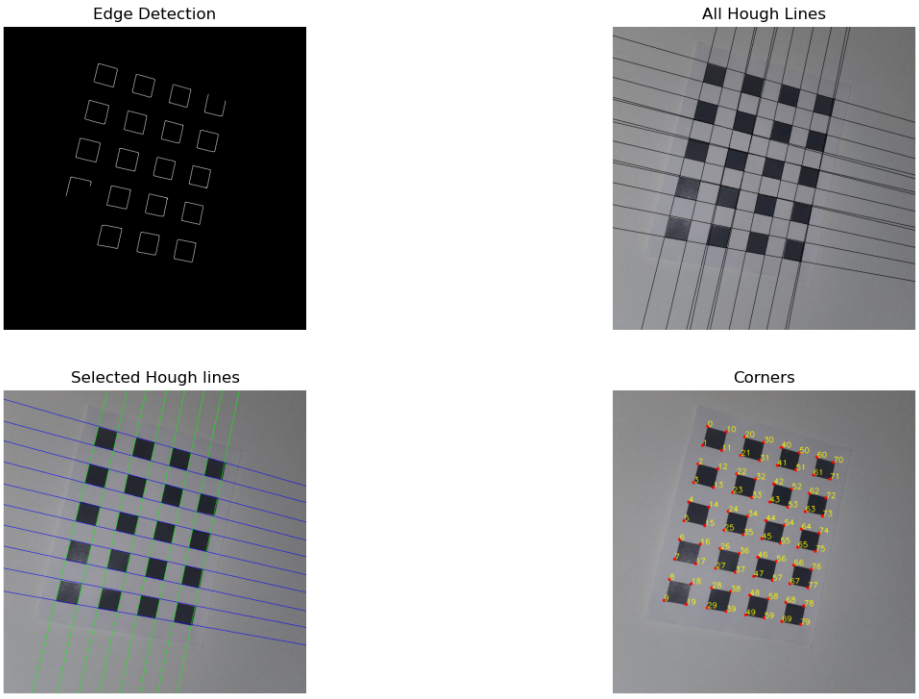


Figure 14: For the second image to be re-projected

```
The mean error before refinement is 13.196126315996906
The mean error after refinement is 7.171267303131373
The variance in error before refinement is 81.74357653973614
The variance in error after refinement is 31.423523507411186
```

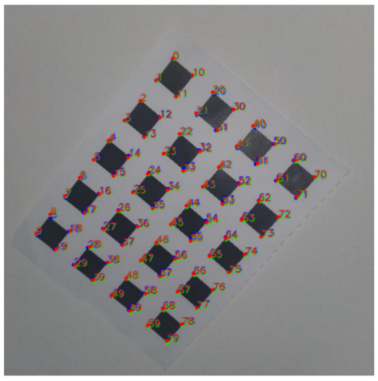


Figure 15: The second image re-projected to the fixed image. Blue are the actual points, Green are the re-projected points without LM and Red are the re-projected points with LM

```
The R matrix before LM refinement is
 [[ 0.97331205  0.22550916  0.04253545]
 [-0.21100974  0.95230737 -0.22042133]
 [-0.09021385  0.20556334  0.97447687]]

The R matrix after LM refinement is
 [[ 0.97315902  0.22530084  0.04691535]
 [-0.21029876  0.95340456 -0.21631962]
 [-0.0934663   0.20064715  0.97519473]]

The t vector before LM refinement is
 [-55.06045698 -39.29773882 168.07661424]

The t vector after LM refinement is
 [-54.95456986 -37.32032362 167.70233081]
```

Figure 16: Extrinsic Camera Parameters

In [1]:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import math
from scipy.optimize import least_squares
```

In [2]:

```
#function to calculate the projective homography between two images
def proj_homography(dcord,rcord):
    A = np.zeros((8,8))
    B = np.zeros((8,1))
    H = np.ones((3,3))

    #Generating the matrix equation AB = C using the coordinates of the ROIs
    for i in range(4):
        A[2*i,0] = dcord[i,0]
        A[2*i,1] = dcord[i,1]
        A[2*i,2] = 1
        A[2*i,6] = -1*dcord[i,0]*rcord[i,0]
        A[2*i,7] = -1*dcord[i,1]*rcord[i,0]

        A[2*i+1,3] = dcord[i,0]
        A[2*i+1,4] = dcord[i,1]
        A[2*i+1,5] = 1
        A[2*i+1,6] = -1*dcord[i,0]*rcord[i,1]
        A[2*i+1,7] = -1*dcord[i,1]*rcord[i,1]

        B[2*i,0] = rcord[i,0]
        B[2*i+1,0] = rcord[i,1]


    C = np.dot(np.linalg.pinv(A),B);

    #Obtaining the Homography H
    for i in range(3):
        for j in range(3):
            if i==2 & j ==2:
                break
            H[i,j]=C[3*i +j,0]

    return H
```

In [3]:

```
#function to calculate the homography between two images using inhomogeneous linear least s
def LLS_homography(dcord,rcord):
    n = dcord.shape[0]
    A = np.zeros((2*n,8))
    B = np.zeros((2*n,1))
    H = np.ones((3,3))

    #Generating the matrix equation AB = C using the coordinates of the ROIs
    for i in range(n):
        A[2*i,0] = dcord[i,0]
        A[2*i,1] = dcord[i,1]
        A[2*i,2] = 1
        A[2*i,6] = -1*dcord[i,0]*rcord[i,0]
        A[2*i,7] = -1*dcord[i,1]*rcord[i,0]

        A[2*i+1,3] = dcord[i,0]
        A[2*i+1,4] = dcord[i,1]
        A[2*i+1,5] = 1
        A[2*i+1,6] = -1*dcord[i,0]*rcord[i,1]
        A[2*i+1,7] = -1*dcord[i,1]*rcord[i,1]

        B[2*i,0] = rcord[i,0]
        B[2*i+1,0] = rcord[i,1]


    C = np.dot(np.matmul(np.linalg.pinv(A.T@A),A.T),B);

    #Obtaining the Homography H
    for i in range(3):
        for j in range(3):
            if i==2 & j ==2:
                break
            H[i,j]=C[3*i +j,0]

    return H
```

In [4]:

```
#Returns the inliers for a given homography H
def theinliers(H,im1,im2,delta):
    him1 = np.insert(im1,2,1,axis=1)
    him2 = np.insert(im2,2,1,axis=1)

    him2_calc = np.matmul(H,him1.T)
    im2_calc = him2_calc/him2_calc[2,:]
    im2_calc = im2_calc[0:2,:].T
    error  = np.abs(im2-im2_calc)
    t_error = np.sum(error**2,axis=1)

    idx = np.where(t_error<delta)[0]

    return idx
```

In [5]:

```
#Implementation of RANSAC and resultant homography is refined using inbulit LM method
def RANSAC(im1,im2):
    delta = 3.0
    p = 0.99
    epsilon = 0.75
    n = 4
    ntotal = im1.shape[0]
    N = np.log(1-p)/np.log(1-(1-epsilon)**n)
    N = int(N)
    M = (1-epsilon)*ntotal

    most_inliers = []
    max_inliers=0

    for i in range(N):
        idx = np.random.choice(list(range(ntotal)),n)

        Dcord=[]
        Rcord=[]
        for j in idx:
            Dcord.append(im1[j])
            Rcord.append(im2[j])
        dcord = np.asarray(Dcord)
        rcord = np.asarray(Rcord)
        H = proj_homography(dcord,rcord)
        inliers = theinliers(H,im1,im2,delta**2)
        if len(inliers)>M and len(inliers)>max_inliers:
            max_inliers = len(inliers)
            most_inliers = inliers

    H_final = LLS_homography(im1[most_inliers],im2[most_inliers])
    H_finals = LM_inbuilt(im1[most_inliers],im2[most_inliers],H_final)

    return H_finals
```

In [6]:

```
#Cost function of LM method
def cost_LM(h,dcoord,rcoord):
    H = h.reshape((3,3))
    hdcoord = np.insert(dcoord,2,1,axis=1)
    hrcoord = np.insert(rcoord,2,1,axis=1)

    hrcoord_calc = np.matmul(H,hdcoord.T)
    hrcoord_calc = hrcoord_calc/hrcoord_calc[2,:]
    hrcoord_calc = hrcoord_calc.T
    cost  = np.abs(hrcoord-hrcoord_calc)

    return cost.sum(axis=1)**2
```

In [7]:

```
#Inbuilt LM function
def LM_inbuilt(dcoord,rcoord,H):
    h  = H.flatten()
    LM = least_squares(cost_LM,h,args=(dcoord,rcoord),method="lm")
    h_LM = LM.x
    H_LM = h_LM.reshape((3,3))
    return H_LM
```

In [8]:

```
#To select the relevant lines from all the Hough lines
def selected_lines(lines, x11,x12,x21,x22, ver):
    if ver:
        dist = lines[:,0]*np.cos(lines[:,1])
        thresh = 0.05*(np.max(np.abs(dist))-np.min(np.abs(dist)))
    else:
        dist = lines[:,0]*np.sin(lines[:,1])
        thresh = 0.05*(np.max(np.abs(dist))-np.min(np.abs(dist)))

    sort = np.argsort(dist,axis =0)
    dist_sort = dist[sort]
    prim = []
    sec = []
    for i in range(dist_sort.shape[0]) :
        if i == 0:
            sec.append(sort[i])
        elif dist_sort[i]-dist_sort[i-1] < thresh :
            sec.append(sort[i])
        else :
            prim.append(sec)
            sec = [sort[i]]

    prim.append(sec)

    x11_list = []
    x12_list = []
    x21_list = []
    x22_list = []
    for i in prim:
        x11_list.append(np.mean(x11[i],axis=0).astype('int32'))
        x12_list.append(np.mean(x12[i],axis=0).astype('int32'))
        x21_list.append(np.mean(x21[i],axis=0).astype('int32'))
        x22_list.append(np.mean(x22[i],axis=0).astype('int32'))

    return np.array(x11_list), np.array(x12_list), np.array(x21_list),np.array(x22_list)
```

In [9]:

```
#Input of the image
def inter_pts(img,plot=False):
    if(plot):
        f = plt.figure(figsize=(15,9))

        ax1 = f.add_subplot(221)
        ax2 = f.add_subplot(222)
        ax3 = f.add_subplot(223)
        ax4 = f.add_subplot(224)
        ax1.axis('off')
        ax2.axis('off')
        ax3.axis('off')
        ax4.axis('off')
        ax1.title.set_text('Edge Detection')
        ax2.title.set_text('All Hough Lines')
        ax3.title.set_text('Selected Hough lines')
        ax4.title.set_text('Corners')
    cimg = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
    gimg = cv2.GaussianBlur(cv2.cvtColor(img,cv2.COLOR_BGR2GRAY),(3,3),1.4)

    fimg = cimg.copy()
    fimg1 = cimg.copy()
    can = cv2.Canny(gimg,255*1.5 ,255) #Canny output
    if(plot):
        ax1.imshow(can,cmap='gray')


    #Plotting the horizontal and vertical Hough lines
    hlines = cv2.HoughLines(can, 1, np.pi / 180, 50, None, 0, 0)
    lines = hlines.reshape(hlines.shape[0],hlines.shape[2])
    r = lines[:,0]
    theta = lines[:,1]
    a = np.cos(theta)
    b = np.sin(theta)
    x11 = (a*r-1000*b).astype('int32')
    x12 = (b*r+1000*a).astype('int32')
    x21 = (a*r+1000*b).astype('int32')
    x22 = (b*r-1000*a).astype('int32')

    for i in range(len(x11)):
        cv2.line(fimg1,(x11[i],x12[i]),(x21[i],x22[i]),(0,0,0),1)

    if(plot):
        ax2.imshow(fimg1)


    ovx11 = x11[np.where(b**2<0.5)]
    ovx12 = x12[np.where(b**2<0.5)]
    ovx21 = x21[np.where(b**2<0.5)]
    ovx22 = x22[np.where(b**2<0.5)]

    vx11,vx12,vx21,vx22 = selected_lines(lines[np.where(b**2<0.5)],ovx11,ovx12,ovx21,ovx22,

    ohx11 = x11[np.where(b**2>0.5)]
    ohx12 = x12[np.where(b**2>0.5)]
    ohx21 = x21[np.where(b**2>0.5)]
    ohx22 = x22[np.where(b**2>0.5)]

    hx11,hx12,hx21,hx22 = selected_lines(lines[np.where(b**2>0.5)],ohx11,ohx12,ohx21,ohx22,
```

```
    for i in range(len(vx11)):
        cv2.line(fimg,(vx11[i],vx12[i]),(vx21[i],vx22[i]),(0,255,0),1)

    for i in range(len(hx11)):
        cv2.line(fimg,(hx11[i],hx12[i]),(hx21[i],hx22[i]),(0,0,255),1)

    if(plot):
        ax3.imshow(fimg) #All Hough lines

    #Finding the intersections
    hp1 = np.append([hx11],[hx12],axis=0)
    hp1 = np.append(hp1,np.ones((1,hp1.shape[1])),axis=0).T

    hp2 = np.append([hx21],[hx22],axis=0)
    hp2 = np.append(hp2,np.ones((1,hp2.shape[1])),axis=0).T

    vp1 = np.append([vx11],[vx12],axis=0)
    vp1 = np.append(vp1,np.ones((1,vp1.shape[1])),axis=0).T

    vp2 = np.append([vx21],[vx22],axis=0)
    vp2 = np.append(vp2,np.ones((1,vp2.shape[1])),axis=0).T

    vl = np.cross(vp1,vp2)
    hl = np.cross(hp1,hp2)

    points=[]

    for i in range(vl.shape[0]):
        point = np.cross(hl,vl[i])
        points.append((point[:,:2]/point[:,2].reshape(-1,1)))

    pts = np.concatenate(points,axis=0)

    pimg = cimg.copy()
    for i in range (pts.shape [0]):
        if np.abs(pts[i,0].item())==float('inf'):
            pts = np.ones((10,2))
            break
        point = (int(pts[i,0].item()),int(pts[i,1].item()))
        pimg = cv2.circle(pimg,point,radius =3,color=(255,0,0),thickness = -1)
        pimg = cv2.putText(pimg,str(i),point,cv2.FONT_HERSHEY_SIMPLEX,0.5,(255,255,0),1,cv2

    if(plot):
        ax4.imshow(pimg)
    pts[:,[0,1]] = pts[:,[1,0]]

    return pts
```

In [10]:

```
#Wrapper function for the homography
def Homography_wrapper(img,plot):
    pts = inter_pts(img,plot)
    if(pts.shape[0]!=80):
        return None,None
    else:
        xx,yy = np.meshgrid(np.arange(0,10),np.arange(0,8))
        rcoords = np.vstack((xx.reshape(-1),yy.reshape(-1))).T
        H = RANSAC(rcoords*10,pts)
        return H,pts
```

In [11]:

```
#Calculates the image of the absolute conic
def Calc_omega(H_all):
    n = len(H_all)
    V = np.zeros((2*n,6))


    #Generating the matrix equation Vb = 0
    for i in range(n):
        V[2*i,0] = H_all[i][0][0]*H_all[i][0][1]
        V[2*i,1] = H_all[i][0][0]*H_all[i][1][1] + H_all[i][1][0]*H_all[i][0][1]
        V[2*i,2] = H_all[i][1][0]*H_all[i][1][1]
        V[2*i,3] = H_all[i][2][0]*H_all[i][0][1] + H_all[i][0][0]*H_all[i][2][1]
        V[2*i,4] = H_all[i][2][0]*H_all[i][1][1] + H_all[i][1][0]*H_all[i][2][1]
        V[2*i,5] = H_all[i][2][0]*H_all[i][2][1]

        V[2*i+1,0] = H_all[i][0][0]*H_all[i][0][0] - H_all[i][0][1]*H_all[i][0][1]
        V[2*i+1,1] = 2*H_all[i][0][0]*H_all[i][1][0]  -2*(H_all[i][0][1]*H_all[i][1][1])
        V[2*i+1,2] = H_all[i][1][0]*H_all[i][1][0] - H_all[i][1][1]*H_all[i][1][1]
        V[2*i+1,3] = 2*H_all[i][2][0]*H_all[i][0][0] - 2*(H_all[i][2][1]*H_all[i][0][1])
        V[2*i+1,4] = 2*H_all[i][2][0]*H_all[i][1][0] -2*(H_all[i][2][1]*H_all[i][1][1])
        V[2*i+1,5] = H_all[i][2][0]*H_all[i][2][0] - H_all[i][2][1]*H_all[i][2][1]


    u,s,vh = np.linalg.svd(V)
    b = vh[-1]

    omega = np.array([[ b[0],b[1],b[3]],[b[1],b[2],b[4]],[b[3],b[4],b[5]]])

    return omega
```

In [12]:

```
#Calculates the intrinsic Camera parameters
def Calc_K(H_list):
    o = Calc_omega(H_list)
    K = np.zeros((3,3))
    K[1,2] = (o[0,1]*o[0,2]-o[0,0]*o[1,2])/(o[0,0]*o[1,1]-o[0,1]*o[0,1])
    lamda = o[2,2] -((o[0,2]*o[0,2]+K[1,2]*(o[0,1]*o[0,2]-o[0,0]*o[1,2]))/(o[0,0]))
    K[0,0] = np.sqrt(lamda/o[0,0])
    K[1,1] = np.sqrt((lamda*o[0,0])/(o[0,0]*o[1,1]-o[0,1]*o[0,1]))
    K[0,1] = -(o[0,1]*K[0,0]*K[0,0]*K[1,1])/lamda
    K[0,2] = (K[0,1]*K[1,2]/K[1,1]) - (o[0,2]*K[0,0]*K[0,0]/lamda)
    K[2,2] = 1.0

    return K
```

In [13]:

```
#Calculates the extrinsic Camera parameters
def Calc_Rt(H_nonull,K):
    H_list = np.array(H_nonull)
    h1 = H_list[:,:,0].T
    h2 = H_list[:,:,1].T
    h3 = H_list[:,:,2].T
    Kinv = np.linalg.pinv(K)
    r1t = Kinv@h1
    r2t = Kinv@h2
    tt = Kinv@h3
    eps = 1/(np.sqrt((r1t*r1t).sum(axis=0)))
    r1 = eps*r1t
    r2 = eps*r2t
    r3 = np.cross(r1.T,r2.T).T
    t = eps*tt
    u,d,v = np.linalg.svd(np.array([r1,r2,r3]).T)
    return t.T, np.matmul(u,v)
```

In [14]:

```
#Calculating reprojection error
def mean_var_error(ptso,pts1):
    cost = np.abs(ptso-pts1)
    error = cost.sum(axis=1)**2
    return error.mean(),np.var(error)
```

In [15]:

```
#Plots the original and the reprojected points
def Plot_points(H_fixed,i1,K,R,t,K_new,R_new,t_new,l):
    Rt = np.concatenate([[R[:,:,0]],[R[:,:,1]],[t.reshape((R.shape[0],-1))]],axis=0)
    RT = np.einsum('ijk->jki', Rt)
    H_recon = np.matmul(K,RT)
    H_final_recon = (H_recon.T/H_recon.max(axis=1).max(axis=1)).T

    Rt_new = np.concatenate([[R_new[:,:,0]],[R_new[:,:,1]],[t_new.reshape((R_new.shape[0],-
    RT_new = np.einsum('ijk->jki', Rt_new)
    H_recon_new = np.matmul(K_new,RT_new)
    H_final_recon_new = (H_recon_new.T/H_recon_new.max(axis=1).max(axis=1)).T

    imgo = cv2.imread("HW8-Files/Dataset1/Pic_"+str(5)+".jpg")
    ptso = inter_pts(imgo,False)
    ptso[:,[0,1]] = ptso[:,[1,0]]
    pimg = cv2.cvtColor(imgo,cv2.COLOR_BGR2RGB)


    H = H_final_recon[i1]@np.linalg.pinv(H_fixed)
    H_new = H_final_recon_new[i1]@np.linalg.pinv(H_fixed)

    img1 = cv2.imread("HW8-Files/Dataset1/Pic_"+str(l[i1]+1)+".jpg")
    pt1 = inter_pts(img1,False)
    ones = np.ones((pt1.shape[0],1))
    hdcord = np.append(pt1,ones,1).T


    temp = np.linalg.pinv(H/H.max())@hdcord
    temp[0,:] = temp[0,:]/temp[2,:]
    temp[1,:] = temp[1,:]/temp[2,:]
    pts1 = np.array([temp[0,:],temp[1,:]]).T
    pts1[:,[0,1]] = pts1[:,[1,0]]

    temp2 = np.linalg.pinv(H_new/H_new.max())@hdcord
    temp2[0,:] = temp2[0,:]/temp2[2,:]
    temp2[1,:] = temp2[1,:]/temp2[2,:]
    pts2 = np.array([temp2[0,:],temp2[1,:]]).T
    pts2[:,[0,1]] = pts2[:,[1,0]]

    M,v = mean_var_error(ptso,pts1)
    M_LM,v_LM = mean_var_error(ptso,pts2)


    for i in range (ptso.shape [0]):
        pointo = (int(ptso[i,0].item()),int(ptso[i,1].item())) #Actual Point
        point1 = (int(pts1[i,0].item()),int(pts1[i,1].item())) #Points without any refineme
        point2 = (int(pts2[i,0].item()),int(pts2[i,1].item())) #Points with refinement

        pimg = cv2.circle(pimg,pointo,radius =3,color=(0,0,255),thickness = -1)
        pimg = cv2.putText(pimg,str(i),pointo,cv2.FONT_HERSHEY_SIMPLEX,0.5,(0,0,255),1,cv2.

        pimg = cv2.circle(pimg,point1,radius =3,color=(0,255,0),thickness = -1)
        pimg = cv2.putText(pimg,str(i),point1,cv2.FONT_HERSHEY_SIMPLEX,0.5,(0,255,0),1,cv2.

        pimg = cv2.circle(pimg,point2,radius =3,color=(255,0,0),thickness = -1)
        pimg = cv2.putText(pimg,str(i),point2,cv2.FONT_HERSHEY_SIMPLEX,0.5,(255,0,0),1,cv2.

    plt.imshow(pimg)
    plt.axis('off')
```

```
    return M,v,M_LM,v_LM
```

In [16]:

```
#Converts R to Rodrigues representation
def parameterized_comp(k,R_all,t_all):
    K = np.asarray([k[0,0],k[0,1],k[0,2],k[1,1],k[1,2]])
    phi = np.arccos((np.trace(R_all.T)-1)/2)
    wee = (phi/(2*np.sin(phi))*np.array([R_all[:,2,1]-R_all[:,1,2],R_all[:,0,2]-R_all[:,2,0
    W = wee.flatten()
    Wt = np.append(W,t_all.flatten())
    ans = np.append(K,Wt)

    return ans
```

In [17]:

```
#Converts from Rodrigues representation to R
def deparameterized_comp(ans,n_img=30):
    k = ans[0:5]
    K = np.array([[k[0],k[1],k[2]],[0,k[3],k[4]],[0,0,1]])

    w = ans[5:3*n_img+5]
    w = w.reshape(n_img,-1)
    I = np.eye(3,3)
    phi = np.linalg.norm(w,axis=1)
    zero = np.zeros((n_img))

    W33X = np.array([[zero,-w[:,2],w[:,1]],[w[:,2],zero ,-w[:,0]],[-w[:,1],w[:,0],zero ]])
    W3X = np.einsum('ijk->kij', W33X)
    Re = (np.sin(phi)/phi)[:, None, None]*W3X + ((1-np.cos(phi))/(phi**2))[:, None, None]*n
    R = Re+I
    t = ans[3*n_img+5:]
    T = t.reshape(n_img,-1)

    return K,R,T
```

In [18]:

```
#Cost function for Zhang LM
def cost_Func_Zhang(ans,dcoord,rcoord):
    K,R,t = deparameterized_comp(ans)
    cost_all=[]
    for i in range(R.shape[0]):
        rg = R[i]
        tg = t[i]
        RTG = np.concatenate([rg[:,0:1],rg[:,1:2],tg.reshape((-1,1))],axis=1)
        H_g = np.dot(K,RTG)
        H = H_g/H_g.max()

        r = rcoord[i]
        hdcoord = np.insert(dcoord,2,1,axis=1)
        hrcoord = np.insert(r,2,1,axis=1)

        hrcoord_calc = np.matmul(H,hdcoord.T)
        hrcoord_calc = hrcoord_calc/hrcoord_calc[2,:]
        hrcoord_calc = hrcoord_calc.T
        cost  = np.abs(hrcoord-hrcoord_calc)
        cost= cost.sum(axis=1)**2
        cost_all.append(cost)

    final_cost = np.array(cost_all)
    return final_cost.flatten()
```

In [19]:

```
#Inbuilt LM function
def LM_inbuilt_Zhang(pts,K1,R1,T1):
    xx,yy = np.meshgrid(np.arange(0,10),np.arange(0,8))
    rcoords = np.vstack((xx.reshape(-1),yy.reshape(-1))).T
    ansy = parameterized_comp(K1,R1,T1)
    LM = least_squares(cost_Func_Zhang,ansy,args=(rcoords*10,pts),method="lm")
    p_LM = LM.x
    K_new,R_new,t_new = deparameterized_comp(p_LM)
    return K_new,R_new,t_new
```

In [20]:

```
#Generating all the homographies
H_all = []
listi = []
inter_pts_nonull = []
all_pts = []
for i in range(40):
    img = cv2.imread("HW8-Files/Dataset1/Pic_"+str(i+1)+".jpg")
    H,pts = Homography_wrapper(img,plot=False)
    if H is not None:
        listi.append(i+1)
        inter_pts_nonull.append(pts)
    H_all.append(H)
    all_pts.append(pts)

all_pts_nonull = np.array(inter_pts_nonull)
H_all_nonull = list(filter(lambda item: item is not None, H_all))
```

```
C:\Users\adisi\AppData\Local\Temp\ipykernel_14508\4074767644.py:91: RuntimeW
arning: divide by zero encountered in divide
  points.append((point[:,:2]/point[:,2].reshape(-1,1)))
C:\Users\adisi\AppData\Local\Temp\ipykernel_14508\4074767644.py:91: RuntimeW
arning: invalid value encountered in divide
  points.append((point[:,:2]/point[:,2].reshape(-1,1)))
C:\Users\adisi\AppData\Local\Temp\ipykernel_14508\1101415931.py:7: RuntimeWa
rning: divide by zero encountered in divide
  im2_calc = him2_calc/him2_calc[2,:]
C:\Users\adisi\AppData\Local\Temp\ipykernel_14508\1101415931.py:7: RuntimeWa
rning: invalid value encountered in divide
  im2_calc = him2_calc/him2_calc[2,:]
```

In [21]:

```
index_nums = [0,1,2,4,5,7,8,10,11,13,14,15,16,18,20,22,23,24,26,28,29,30,31,32,33,34,35,37,
H_trim = [H_all[val] for val in index_nums]
pts_trim = [all_pts[val] for val in index_nums]
p_trim = np.array(pts_trim)
```

In [22]:

```
#Calculating the calibration parameters
K= Calc_K(H_trim)
t,R = Calc_Rt(H_trim,K)
```

In [23]:

```
#Refining the parameters using LM
K_new,R_new,t_new = LM_inbuilt_Zhang(p_trim,K,R,t)
```

In [29]:

```
a,b,c,d = Plot_points(H_trim[3],23,K,R,t,K_new,R_new,t_new,index_nums)
print("The mean error before LM refinement is",a)
print("The mean error after LM refinement is",c)
print("The variance in error before LM refinement is",b)
print("The variance in error after LM refinement is",d)
```

```
The mean error before LM refinement is 2.3694799328228564
The mean error after LM refinement is 1.587696223270158
The variance in error before LM refinement is 4.644408981989415
The variance in error after LM refinement is 4.013309113597015
```

In [28]:

```
print("The R matrix before LM refinement is\n",R[23])
print("\nThe R matrix after LM refinement is\n",R_new[23])
print("\nThe t vector before LM refinement is\n",t[23])
print("\nThe t vector after LM refinement is\n",t_new[23])
```

```
The R matrix before LM refinement is
 [[ 0.984763    0.16258309 -0.06171361]
 [-0.16839072  0.98012319 -0.10489567]
 [ 0.04343268  0.11368937  0.99256654]]

The R matrix after LM refinement is
 [[ 0.98489652  0.16184145 -0.06153198]
 [-0.16829534  0.97833681 -0.12055608]
 [ 0.04068803  0.12909081  0.99079768]]

The t vector before LM refinement is
 [-44.1697478  -20.37440218 214.79102805]

The t vector after LM refinement is
 [-44.40192841 -18.84045809 224.79242679]
```

In [30]:

```
a,b,c,d = Plot_points(H_trim[3],2,K,R,t,K_new,R_new,t_new,index_nums)
print("The mean error before LM refinement is",a)
print("The mean error after LM refinement is",c)
print("The variance in error before LM refinement is",b)
print("The variance in error after LM refinement is",d)
```

```
The mean error before LM refinement is 5.025794169755225
The mean error after LM refinement is 1.6451253036778
The variance in error before LM refinement is 35.58838597737733
The variance in error after LM refinement is 2.854361976738941
```

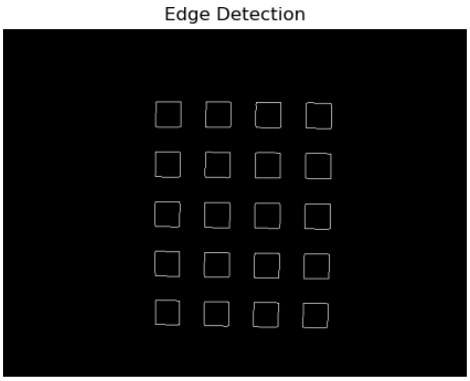
In [31]:

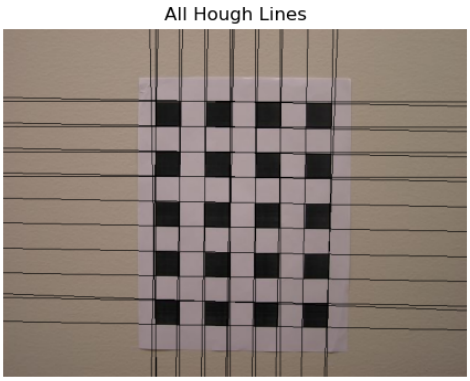
```
print("The R matrix before LM refinement is\n",R[2])
print("\nThe R matrix after LM refinement is\n",R_new[2])
print("\nThe t vector before LM refinement is\n",t[2])
print("\nThe t vector after LM refinement is\n",t_new[2])
```

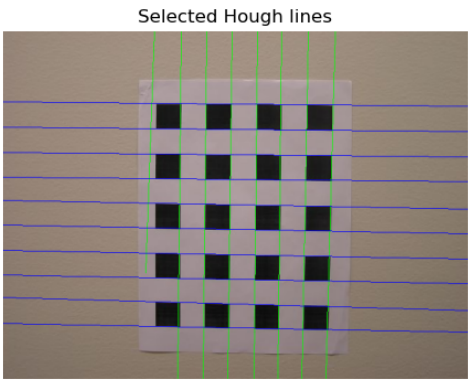
```
The R matrix before LM refinement is
 [[ 0.9789797  -0.20232252 -0.0257749 ]
 [ 0.16777061  0.87069164 -0.46233008]
 [ 0.11598177  0.44828749  0.88633321]]

The R matrix after LM refinement is
 [[ 0.97896591 -0.20231065 -0.02638443]
 [ 0.16810885  0.87313963 -0.45756595]
 [ 0.11560776  0.44350601  0.88878415]]

The t vector before LM refinement is
 [-36.44966214 -24.53481072 195.36188406]

The t vector after LM refinement is
 [-36.69976249 -22.87745425 204.2785367 ]
```

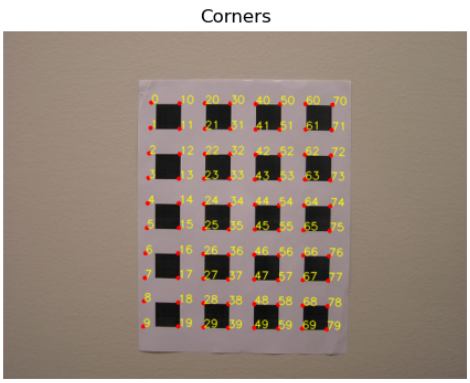
In [55]:

```
i=10
dimg = cv2.imread("HW8-Files/Dataset1/Pic_"+str(i+1)+".jpg")
H = inter_pts(dimg,True)
```









In [35]:

```
np.set_printoptions(suppress=True)
print("The intrinsic camera caliberation parameter before LM refinement is \n", K)
print("\nThe intrinsic camera caliberation parameter after LM refinement is \n", K_new)
```

```
The intrinsic camera caliberation parameter before LM refinement is
 [[704.55793689   1.3782471  245.57112836]
 [  0.         709.09265189 317.11587573]
 [  0.           0.           1.        ]]

The intrinsic camera caliberation parameter after LM refinement is
 [[735.43935418   0.69385808 246.12174319]
 [  0.         744.12941793 311.69053472]
 [  0.           0.           1.        ]]
```

In [ ]:

In [1]:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import math
from scipy.optimize import least_squares
```

In [2]:

```
#function to calculate the projective homography between two images
def proj_homography(dcord,rcord):
    A = np.zeros((8,8))
    B = np.zeros((8,1))
    H = np.ones((3,3))

    #Generating the matrix equation AB = C using the coordinates of the ROIs
    for i in range(4):
        A[2*i,0] = dcord[i,0]
        A[2*i,1] = dcord[i,1]
        A[2*i,2] = 1
        A[2*i,6] = -1*dcord[i,0]*rcord[i,0]
        A[2*i,7] = -1*dcord[i,1]*rcord[i,0]

        A[2*i+1,3] = dcord[i,0]
        A[2*i+1,4] = dcord[i,1]
        A[2*i+1,5] = 1
        A[2*i+1,6] = -1*dcord[i,0]*rcord[i,1]
        A[2*i+1,7] = -1*dcord[i,1]*rcord[i,1]

        B[2*i,0] = rcord[i,0]
        B[2*i+1,0] = rcord[i,1]


    C = np.dot(np.linalg.pinv(A),B);

    #Obtaining the Homography H
    for i in range(3):
        for j in range(3):
            if i==2 & j ==2:
                break
            H[i,j]=C[3*i +j,0]

    return H
```

In [3]:

```
#function to calculate the homography between two images using inhomogeneous linear least s
def LLS_homography(dcord,rcord):
    n = dcord.shape[0]
    A = np.zeros((2*n,8))
    B = np.zeros((2*n,1))
    H = np.ones((3,3))

    #Generating the matrix equation AB = C using the coordinates of the ROIs
    for i in range(n):
        A[2*i,0] = dcord[i,0]
        A[2*i,1] = dcord[i,1]
        A[2*i,2] = 1
        A[2*i,6] = -1*dcord[i,0]*rcord[i,0]
        A[2*i,7] = -1*dcord[i,1]*rcord[i,0]

        A[2*i+1,3] = dcord[i,0]
        A[2*i+1,4] = dcord[i,1]
        A[2*i+1,5] = 1
        A[2*i+1,6] = -1*dcord[i,0]*rcord[i,1]
        A[2*i+1,7] = -1*dcord[i,1]*rcord[i,1]

        B[2*i,0] = rcord[i,0]
        B[2*i+1,0] = rcord[i,1]


    C = np.dot(np.matmul(np.linalg.pinv(A.T@A),A.T),B);

    #Obtaining the Homography H
    for i in range(3):
        for j in range(3):
            if i==2 & j ==2:
                break
            H[i,j]=C[3*i +j,0]

    return H
```

In [4]:

```
#Returns the inliers for a given homography H
def theinliers(H,im1,im2,delta):
    him1 = np.insert(im1,2,1,axis=1)
    him2 = np.insert(im2,2,1,axis=1)

    him2_calc = np.matmul(H,him1.T)
    im2_calc = him2_calc/him2_calc[2,:]
    im2_calc = im2_calc[0:2,:].T
    error  = np.abs(im2-im2_calc)
    t_error = np.sum(error**2,axis=1)

    idx = np.where(t_error<delta)[0]

    return idx
```

In [5]:

```
#Implementation of RANSAC and resultant homography is refined using inbulit LM method
def RANSAC(im1,im2):
    delta = 3.0
    p = 0.99
    epsilon = 0.75
    n = 4
    ntotal = im1.shape[0]
    N = np.log(1-p)/np.log(1-(1-epsilon)**n)
    N = int(N)
    M = (1-epsilon)*ntotal

    most_inliers = []
    max_inliers=0

    for i in range(N):
        idx = np.random.choice(list(range(ntotal)),n)

        Dcord=[]
        Rcord=[]
        for j in idx:
            Dcord.append(im1[j])
            Rcord.append(im2[j])
        dcord = np.asarray(Dcord)
        rcord = np.asarray(Rcord)
        H = proj_homography(dcord,rcord)
        inliers = theinliers(H,im1,im2,delta**2)
        if len(inliers)>M and len(inliers)>max_inliers:
            max_inliers = len(inliers)
            most_inliers = inliers

    H_final = LLS_homography(im1[most_inliers],im2[most_inliers])
    H_finals = LM_inbuilt(im1[most_inliers],im2[most_inliers],H_final)

    return H_finals
```

In [6]:

```
#Cost function of LM method
def cost_LM(h,dcoord,rcoord):
    H = h.reshape((3,3))
    hdcoord = np.insert(dcoord,2,1,axis=1)
    hrcoord = np.insert(rcoord,2,1,axis=1)

    hrcoord_calc = np.matmul(H,hdcoord.T)
    hrcoord_calc = hrcoord_calc/hrcoord_calc[2,:]
    hrcoord_calc = hrcoord_calc.T
    cost  = np.abs(hrcoord-hrcoord_calc)

    return cost.sum(axis=1)**2
```

In [7]:

```
#Inbuilt LM function
def LM_inbuilt(dcoord,rcoord,H):
    h  = H.flatten()
    LM = least_squares(cost_LM,h,args=(dcoord,rcoord),method="lm")
    h_LM = LM.x
    H_LM = h_LM.reshape((3,3))
    return H_LM
```

In [8]:

```
#To select the relevant lines from all the Hough lines
def selected_lines(lines, x11,x12,x21,x22, ver):
    if ver:
        dist = lines[:,0]*np.cos(lines[:,1])
        thresh = 0.05*(np.max(np.abs(dist))-np.min(np.abs(dist)))
    else:
        dist = lines[:,0]*np.sin(lines[:,1])
        thresh = 0.05*(np.max(np.abs(dist))-np.min(np.abs(dist)))

    sort = np.argsort(dist,axis =0)
    dist_sort = dist[sort]
    prim = []
    sec = []
    for i in range(dist_sort.shape[0]) :
        if i == 0:
            sec.append(sort[i])
        elif dist_sort[i]-dist_sort[i-1]<thresh :
            sec.append(sort[i])
        else :
            prim.append(sec)
            sec = [sort[i]]

    prim.append(sec)

    x11_list = []
    x12_list = []
    x21_list = []
    x22_list = []
    for i in prim:
        x11_list.append(np.mean(x11[i],axis=0).astype('int32'))
        x12_list.append(np.mean(x12[i],axis=0).astype('int32'))
        x21_list.append(np.mean(x21[i],axis=0).astype('int32'))
        x22_list.append(np.mean(x22[i],axis=0).astype('int32'))

    return np.array(x11_list), np.array(x12_list), np.array(x21_list),np.array(x22_list)
```

In [44]:

```
#Input of the image
def inter_pts(img,plot=False):
    if(plot):
        f = plt.figure(figsize=(15,9))
        ax1 = f.add_subplot(221)
        ax2 = f.add_subplot(222)
        ax3 = f.add_subplot(223)
        ax4 = f.add_subplot(224)
        ax1.axis('off')
        ax2.axis('off')
        ax3.axis('off')
        ax4.axis('off')
        ax1.title.set_text('Edge Detection')
        ax2.title.set_text('All Hough Lines')
        ax3.title.set_text('Selected Hough lines')
        ax4.title.set_text('Corners')
    cimg = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
    gimg = cv2.GaussianBlur(cv2.cvtColor(img,cv2.COLOR_BGR2GRAY),(3,3),0.9)

    fimg = cimg.copy()
    fimg1 = cimg.copy()
    can = cv2.Canny(gimg,350 ,255) #Canny output
    if(plot):
        ax1.imshow(can,cmap='gray')

    #Plotting the horizontal and vertical Hough lines
    hlines = cv2.HoughLines(can, 1, np.pi / 180, 50, None, 0, 0)
    lines = hlines.reshape(hlines.shape[0],hlines.shape[2])
    r = lines[:,0]
    theta = lines[:,1]
    a = np.cos(theta)
    b = np.sin(theta)
    x11 = (a*r-1000*b).astype('int32')
    x12 = (b*r+1000*a).astype('int32')
    x21 = (a*r+1000*b).astype('int32')
    x22 = (b*r-1000*a).astype('int32')

    for i in range(len(x11)):
        cv2.line(fimg1,(x11[i],x12[i]),(x21[i],x22[i]),(0,0,0),1)

    if(plot):
        ax2.imshow(fimg1)


    ovx11 = x11[np.where(b**2<0.5)]
    ovx12 = x12[np.where(b**2<0.5)]
    ovx21 = x21[np.where(b**2<0.5)]
    ovx22 = x22[np.where(b**2<0.5)]

    vx11,vx12,vx21,vx22 = selected_lines(lines[np.where(b**2<0.5)],ovx11,ovx12,ovx21,ovx22,

    ohx11 = x11[np.where(b**2>0.5)]
    ohx12 = x12[np.where(b**2>0.5)]
    ohx21 = x21[np.where(b**2>0.5)]
    ohx22 = x22[np.where(b**2>0.5)]

    hx11,hx12,hx21,hx22 = selected_lines(lines[np.where(b**2>0.5)],ohx11,ohx12,ohx21,ohx22,
```

```
    for i in range(len(vx11)):
        cv2.line(fimg,(vx11[i],vx12[i]),(vx21[i],vx22[i]),(0,255,0),1)

    for i in range(len(hx11)):
        cv2.line(fimg,(hx11[i],hx12[i]),(hx21[i],hx22[i]),(0,0,255),1)

    if(plot):
        ax3.imshow(fimg) #All Hough lines

    #Finding the intersections
    hp1 = np.append([hx11],[hx12],axis=0)
    hp1 = np.append(hp1,np.ones((1,hp1.shape[1])),axis=0).T

    hp2 = np.append([hx21],[hx22],axis=0)
    hp2 = np.append(hp2,np.ones((1,hp2.shape[1])),axis=0).T

    vp1 = np.append([vx11],[vx12],axis=0)
    vp1 = np.append(vp1,np.ones((1,vp1.shape[1])),axis=0).T

    vp2 = np.append([vx21],[vx22],axis=0)
    vp2 = np.append(vp2,np.ones((1,vp2.shape[1])),axis=0).T

    vl = np.cross(vp1,vp2)
    hl = np.cross(hp1,hp2)

    points=[]

    for i in range(vl.shape[0]):
        point = np.cross(hl,vl[i])
        points.append((point[:,:2]/point[:,2].reshape(-1,1)))

    pts = np.concatenate(points,axis=0)

    pimg = cimg.copy()
    for i in range (pts.shape [0]):
        if np.abs(pts[i,0].item())==float('inf'):
            pts = np.ones((10,2))
            break
        point = (int(pts[i,0].item()),int(pts[i,1].item()))
        pimg = cv2.circle(pimg,point,radius =3,color=(255,0,0),thickness = -1)
        pimg = cv2.putText(pimg,str(i),point,cv2.FONT_HERSHEY_SIMPLEX,0.5,(255,255,0),1,cv2

    if(plot):
        ax4.imshow(pimg)
    pts[:,[0,1]] = pts[:,[1,0]]

    return pts
```

In [10]:

```
#Wrapper function for the homography
def Homography_wrapper(img,plot):
    pts = inter_pts(img,plot)
    if(pts.shape[0]!=80):
        return None,None
    else:
        xx,yy = np.meshgrid(np.arange(0,10),np.arange(0,8))
        rcoords = np.vstack((xx.reshape(-1),yy.reshape(-1))).T
        H = RANSAC(rcoords*10,pts)
        return H,pts
```

In [11]:

```
#Calculates the image of the absolute conic
def Calc_omega(H_all):
    n = len(H_all)
    V = np.zeros((2*n,6))


    #Generating the matrix equation Vb = 0
    for i in range(n):
        V[2*i,0] = H_all[i][0][0]*H_all[i][0][1]
        V[2*i,1] = H_all[i][0][0]*H_all[i][1][1] + H_all[i][1][0]*H_all[i][0][1]
        V[2*i,2] = H_all[i][1][0]*H_all[i][1][1]
        V[2*i,3] = H_all[i][2][0]*H_all[i][0][1] + H_all[i][0][0]*H_all[i][2][1]
        V[2*i,4] = H_all[i][2][0]*H_all[i][1][1] + H_all[i][1][0]*H_all[i][2][1]
        V[2*i,5] = H_all[i][2][0]*H_all[i][2][1]

        V[2*i+1,0] = H_all[i][0][0]*H_all[i][0][0] - H_all[i][0][1]*H_all[i][0][1]
        V[2*i+1,1] = 2*H_all[i][0][0]*H_all[i][1][0]  -2*(H_all[i][0][1]*H_all[i][1][1])
        V[2*i+1,2] = H_all[i][1][0]*H_all[i][1][0] - H_all[i][1][1]*H_all[i][1][1]
        V[2*i+1,3] = 2*H_all[i][2][0]*H_all[i][0][0] - 2*(H_all[i][2][1]*H_all[i][0][1])
        V[2*i+1,4] = 2*H_all[i][2][0]*H_all[i][1][0] -2*(H_all[i][2][1]*H_all[i][1][1])
        V[2*i+1,5] = H_all[i][2][0]*H_all[i][2][0] - H_all[i][2][1]*H_all[i][2][1]


    u,s,vh = np.linalg.svd(V)
    b = vh[-1]

    omega = np.array([[ b[0],b[1],b[3]],[b[1],b[2],b[4]],[b[3],b[4],b[5]]])

    return omega
```

In [12]:

```
#Calculates the intrinsic Camera parameters
def Calc_K(H_list):
    o = Calc_omega(H_list)
    K = np.zeros((3,3))
    K[1,2] = (o[0,1]*o[0,2]-o[0,0]*o[1,2])/(o[0,0]*o[1,1]-o[0,1]*o[0,1])
    lamda = o[2,2] -((o[0,2]*o[0,2]+K[1,2]*(o[0,1]*o[0,2]-o[0,0]*o[1,2]))/(o[0,0]))
    K[0,0] = np.sqrt(lamda/o[0,0])
    K[1,1] = np.sqrt((lamda*o[0,0])/(o[0,0]*o[1,1]-o[0,1]*o[0,1]))
    K[0,1] = -(o[0,1]*K[0,0]*K[0,0]*K[1,1])/lamda
    K[0,2] = (K[0,1]*K[1,2]/K[1,1]) - (o[0,2]*K[0,0]*K[0,0]/lamda)
    K[2,2] = 1.0

    return K
```

In [13]:

```
#Calculates the extrinsic Camera parameters
def Calc_Rt(H_nonull,K):
    H_list = np.array(H_nonull)
    h1 = H_list[:,:,0].T
    h2 = H_list[:,:,1].T
    h3 = H_list[:,:,2].T
    Kinv = np.linalg.pinv(K)
    r1t = Kinv@h1
    r2t = Kinv@h2
    tt = Kinv@h3
    eps = 1/(np.sqrt((r1t*r1t).sum(axis=0)))
    r1 = eps*r1t
    r2 = eps*r2t
    r3 = np.cross(r1.T,r2.T).T
    t = eps*tt
    u,d,v = np.linalg.svd(np.array([r1,r2,r3]).T)
    return t.T, np.matmul(u,v)
```

In [14]:

```
#Calculating reprojection error
def mean_var_error(ptso,pts1):
    cost = np.abs(ptso-pts1)
    error = cost.sum(axis=1)**2
    return error.mean(),np.var(error)
```

In [49]:

```
#Plots the original and the reprojected points
def Plot_points(H_fixed,i1,K,R,t,K_new,R_new,t_new,l):
    Rt = np.concatenate([[R[:,:,0]],[R[:,:,1]],[t.reshape((R.shape[0],-1))]],axis=0)
    RT = np.einsum('ijk->jki', Rt)
    H_recon = np.matmul(K,RT)
    H_final_recon = (H_recon.T/H_recon.max(axis=1).max(axis=1)).T

    Rt_new = np.concatenate([[R_new[:,:,0]],[R_new[:,:,1]],[t_new.reshape((R_new.shape[0],-
    RT_new = np.einsum('ijk->jki', Rt_new)
    H_recon_new = np.matmul(K_new,RT_new)
    H_final_recon_new = (H_recon_new.T/H_recon_new.max(axis=1).max(axis=1)).T

    imgo = cv2.imread("HW8-Files/Dataset2/Pic_"+str(1)+".jpeg")
    ptso = inter_pts(imgo,False)
    ptso[:,[0,1]] = ptso[:,[1,0]]
    pimg = cv2.cvtColor(imgo,cv2.COLOR_BGR2RGB)


    H = H_final_recon[i1]@np.linalg.pinv(H_fixed)
    H_new = H_final_recon_new[i1]@np.linalg.pinv(H_fixed)

    img1 = cv2.imread("HW8-Files/Dataset2/Pic_"+str(l[i1]+1)+".jpeg")
    pt1 = inter_pts(img1,False)
    ones = np.ones((pt1.shape[0],1))
    hdcord = np.append(pt1,ones,1).T


    temp = np.linalg.pinv(H/H.max())@hdcord
    temp[0,:] = temp[0,:]/temp[2,:]
    temp[1,:] = temp[1,:]/temp[2,:]
    pts1 = np.array([temp[0,:],temp[1,:]]).T
    pts1[:,[0,1]] = pts1[:,[1,0]]

    temp2 = np.linalg.pinv(H_new/H_new.max())@hdcord
    temp2[0,:] = temp2[0,:]/temp2[2,:]
    temp2[1,:] = temp2[1,:]/temp2[2,:]
    pts2 = np.array([temp2[0,:],temp2[1,:]]).T
    pts2[:,[0,1]] = pts2[:,[1,0]]

    M,v = mean_var_error(ptso,pts1)
    M_LM,v_LM = mean_var_error(ptso,pts2)


    for i in range (ptso.shape [0]):
        pointo = (int(ptso[i,0].item()),int(ptso[i,1].item())) #Actual Point
        point1 = (int(pts1[i,0].item()),int(pts1[i,1].item())) #Points without any refineme
        point2 = (int(pts2[i,0].item()),int(pts2[i,1].item())) #Points with refinement

        pimg = cv2.circle(pimg,pointo,radius =3,color=(0,0,255),thickness = -1)
        pimg = cv2.putText(pimg,str(i),pointo,cv2.FONT_HERSHEY_SIMPLEX,0.5,(0,0,255),1,cv2.

        pimg = cv2.circle(pimg,point1,radius =3,color=(0,255,0),thickness = -1)
        pimg = cv2.putText(pimg,str(i),point1,cv2.FONT_HERSHEY_SIMPLEX,0.5,(0,255,0),1,cv2.

        pimg = cv2.circle(pimg,point2,radius =3,color=(255,0,0),thickness = -1)
        pimg = cv2.putText(pimg,str(i),point2,cv2.FONT_HERSHEY_SIMPLEX,0.5,(255,0,0),1,cv2.

    plt.imshow(pimg)
    plt.axis('off')
```

```
    return M,v,M_LM,v_LM
```

In [16]:

```
#Converts R to Rodrigues representation
def parameterized_comp(k,R_all,t_all):
    K = np.asarray([k[0,0],k[0,1],k[0,2],k[1,1],k[1,2]])
    phi = np.arccos((np.trace(R_all.T)-1)/2)
    wee = (phi/(2*np.sin(phi)))*np.array([R_all[:,2,1]-R_all[:,1,2],R_all[:,0,2]-R_all[:,2,0
    W = wee.flatten()
    Wt = np.append(W,t_all.flatten())
    ans = np.append(K,Wt)

    return ans
```

In [17]:

```
#Converts from Rodrigues representation to R
def deparameterized_comp(ans,n_img=16):
    k = ans[0:5]
    K = np.array([[k[0],k[1],k[2]],[0,k[3],k[4]],[0,0,1]])

    w = ans[5:3*n_img+5]
    w = w.reshape(n_img,-1)
    I = np.eye(3,3)
    phi = np.linalg.norm(w,axis=1)
    zero = np.zeros((n_img))

    W33X = np.array([[zero,-w[:,2],w[:,1]],[w[:,2],zero ,-w[:,0]],[-w[:,1],w[:,0],zero ]])
    W3X = np.einsum('ijk->kij', W33X)
    Re = (np.sin(phi)/phi)[:, None, None]*W3X + ((1-np.cos(phi))/(phi**2))[:, None, None]*n
    R = Re+I
    t = ans[3*n_img+5:]
    T = t.reshape(n_img,-1)

    return K,R,T
```

In [18]:

```
#Cost function
def cost_Func_Zhang(ans,dcoord,rcoord):
    K,R,t = deparameterized_comp(ans)
    cost_all=[]
    for i in range(R.shape[0]):
        rg = R[i]
        tg = t[i]
        RTG = np.concatenate([rg[:,0:1],rg[:,1:2],tg.reshape((-1,1))],axis=1)
        H_g = np.dot(K,RTG)
        H = H_g/H_g.max()

        r = rcoord[i]
        hdcoord = np.insert(dcoord,2,1,axis=1)
        hrcoord = np.insert(r,2,1,axis=1)

        hrcoord_calc = np.matmul(H,hdcoord.T)
        hrcoord_calc = hrcoord_calc/hrcoord_calc[2,:]
        hrcoord_calc = hrcoord_calc.T
        cost  = np.abs(hrcoord-hrcoord_calc)
        cost= cost.sum(axis=1)**2
        cost_all.append(cost)

    final_cost = np.array(cost_all)
    return final_cost.flatten()
```

In [19]:

```
#Inbuilt LM function
def LM_inbuilt_Zhang(pts,K1,R1,T1):
    xx,yy = np.meshgrid(np.arange(0,10),np.arange(0,8))
    rcoords = np.vstack((xx.reshape(-1),yy.reshape(-1))).T
    ansy = parameterized_comp(K1,R1,T1)
    LM = least_squares(cost_Func_Zhang,ansy,args=(rcoords*10,pts),method="lm")
    p_LM = LM.x
    K_new,R_new,t_new = deparameterized_comp(p_LM)
    return K_new,R_new,t_new
```

In [20]:

```
#Generating all the homographies
H_all = []
listi = []
inter_pts_nonull = []
all_pts = []
for i in range(24):
    img = cv2.imread("HW8-Files/Dataset2/Pic_"+str(i+1)+".jpeg")
    H,pts = Homography_wrapper(img,plot=False)
    if H is not None:
        listi.append(i+1)
        inter_pts_nonull.append(pts)
    H_all.append(H)
    all_pts.append(pts)

all_pts_nonull = np.array(inter_pts_nonull)
H_all_nonull = list(filter(lambda item: item is not None, H_all))
```

```
C:\Users\adisi\AppData\Local\Temp\ipykernel_10764\1101415931.py:7: RuntimeWa
rning: divide by zero encountered in divide
  im2_calc = him2_calc/him2_calc[2,:]
C:\Users\adisi\AppData\Local\Temp\ipykernel_10764\1101415931.py:7: RuntimeWa
rning: invalid value encountered in divide
  im2_calc = him2_calc/him2_calc[2,:]
```

In [21]:

```
index_nums = [0,1,2,3,5,6,7,9,12,14,15,17,18,20,21,23]
H_trim = [H_all[val] for val in index_nums]
pts_trim = [all_pts[val] for val in index_nums]
p_trim = np.array(pts_trim)
```

In [22]:

```
#Calculating the calibration parameters
K= Calc_K(H_trim)
t,R = Calc_Rt(H_trim,K)
```

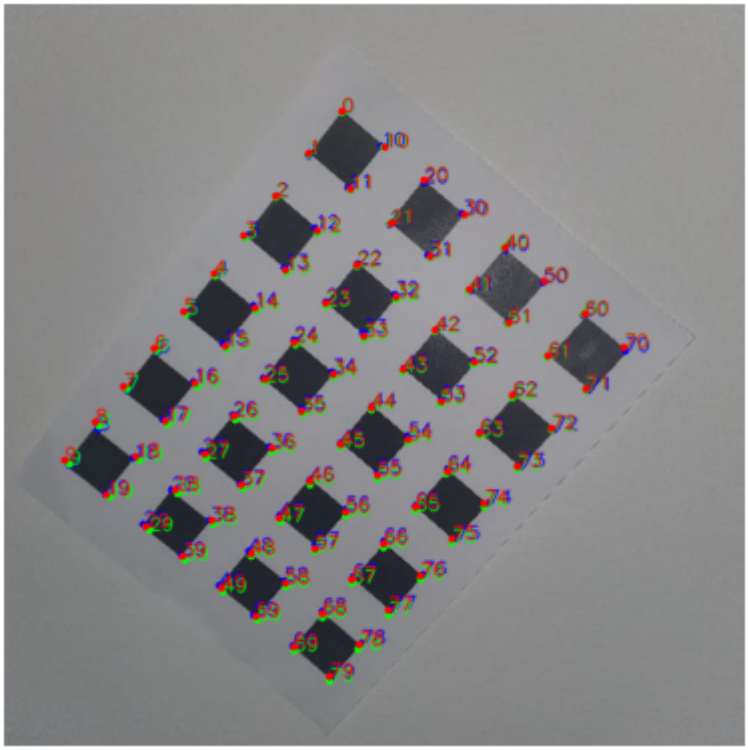
In [23]:

```
#Refining the parameters using LM
K_new,R_new,t_new = LM_inbuilt_Zhang(p_trim,K,R,t)
```

In [50]:

```
a,b,c,d = Plot_points(H_trim[0],6,K,R,t,K_new,R_new,t_new,index_nums)
print("The mean error before LM refinement is",a)
print("The mean error after LM refinement is",c)
print("The variance in error before LM refinement is",b)
print("The variance in error after LM refinement is",d)
```

```
The mean error before LM refinement is 24.450302803898087
The mean error after LM refinement is 9.568342488973133
The variance in error before LM refinement is 348.34981098779605
The variance in error after LM refinement is 112.31066266581063
```



In [30]:

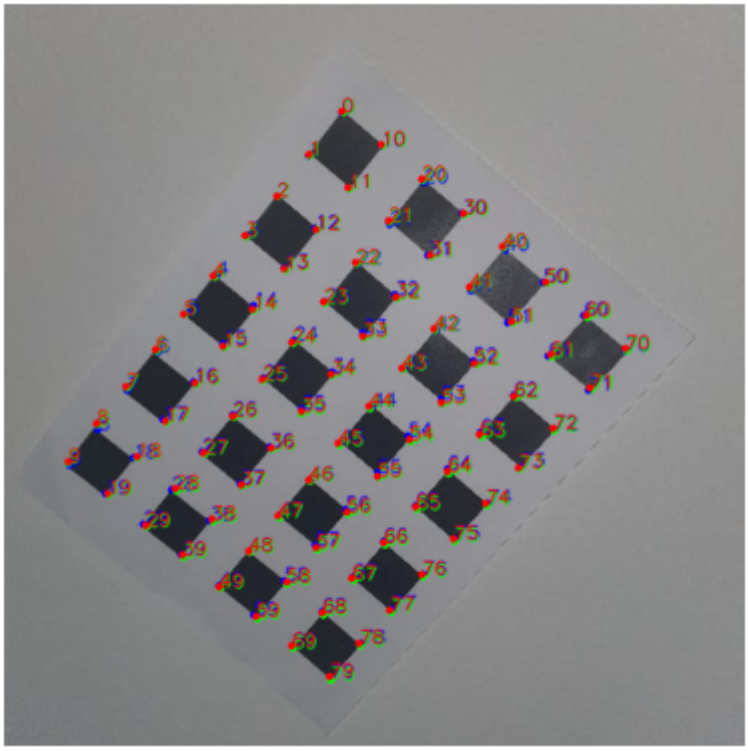
```
print("The R matrix before LM refinement is\n",R[6])
print("\nThe R matrix after LM refinement is\n",R_new[6])
print("\nThe t vector before LM refinement is\n",t[6])
print("\nThe t vector after LM refinement is\n",t_new[6])
```

```
The R matrix before LM refinement is
 [[ 0.97373011 -0.13328292 -0.18462214]
 [ 0.19563905  0.90455792  0.37881436]
 [ 0.11651193 -0.40498225  0.90687063]]

The R matrix after LM refinement is
 [[ 0.97071982 -0.12991232 -0.202054  ]
 [ 0.2023932   0.89537581  0.39666001]
 [ 0.12938324 -0.42594009  0.89545241]]

The t vector before LM refinement is
 [-26.7306836  -60.23068243 176.11273189]

The t vector after LM refinement is
 [-26.49844296 -58.10712862 175.0127035 ]
```

In [51]:

```
a,b,c,d = Plot_points(H_trim[0],5,K,R,t,K_new,R_new,t_new,index_nums)
print("The mean error before refinement is",a)
print("The mean error after refinement is",c)
print("The variance in error before refinement is",b)
print("The variance in error after refinement is",d)
```

```
The mean error before refinement is 13.196126315996906
The mean error after refinement is 7.171267303131373
The variance in error before refinement is 81.74357653973614
The variance in error after refinement is 31.423523507411186
```



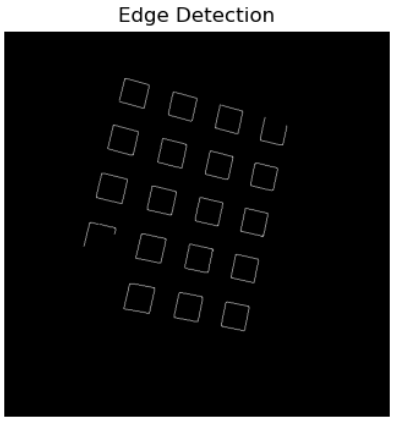
In [32]:

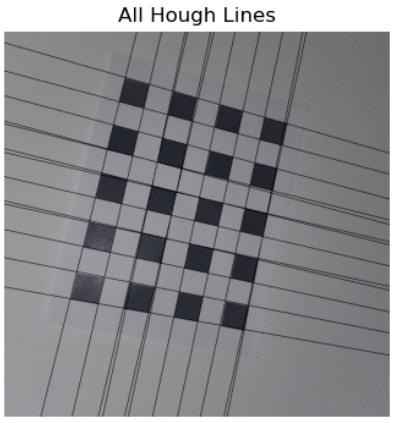
```
print("The R matrix before LM refinement is\n",R[5])
print("\nThe R matrix after LM refinement is\n",R_new[5])
print("\nThe t vector before LM refinement is\n",t[5])
print("\nThe t vector after LM refinement is\n",t_new[5])
```

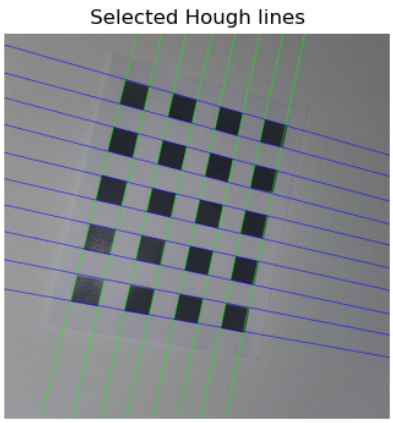
```
The R matrix before LM refinement is
 [[ 0.97331205  0.22550916  0.04253545]
 [-0.21100974  0.95230737 -0.22042133]
 [-0.09021385  0.20556334  0.97447687]]

The R matrix after LM refinement is
 [[ 0.97315902  0.22530084  0.04691535]
 [-0.21029876  0.95340456 -0.21631962]
 [-0.0934663   0.20064715  0.97519473]]

The t vector before LM refinement is
 [-55.06045698 -39.29773882 168.07661424]

The t vector after LM refinement is
 [-54.95456986 -37.32032362 167.70233081]
```

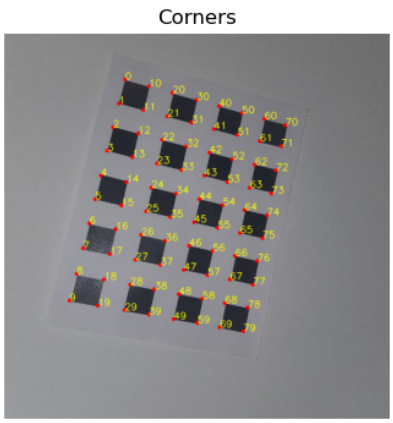
In [48]:

```
i=6
dimg = cv2.imread("HW8-Files/Dataset2/Pic_"+str(i+1)+".jpeg")
H = inter_pts(dimg,True)
```









In [35]:

```
np.set_printoptions(suppress=True)
print("The intrinsic camera caliberation parameter before LM refinement is \n", K)
print("\nThe intrinsic camera caliberation parameter after LM refinement is \n", K_new)
```

```
The intrinsic camera caliberation parameter before LM refinement is
 [[653.37800437  -2.70180296 284.32783632]
 [  0.         651.75085841 340.83652843]
 [  0.           0.           1.        ]]

The intrinsic camera caliberation parameter after LM refinement is
 [[652.24746242  -1.46653519 284.76141222]
 [  0.         651.3547318  334.51810043]
 [  0.           0.           1.        ]]
```