ECE661: Homework 7

Rahul Deshmukh
deshmuk5@purdue.edu
PUID: 0030004932

November 13, 2018

The goal of this homework is to implement Zhang's Algorithm for camera calibration. We are given a set of photos taken from different viewpoints and orientations for a given calibration pattern. The World coordinates of the pattern are known a priori, and for simplification we will be assuming that the calibration pattern is on the world Z=0 plane.

We are required to find out the intrinsic and extrinsic parameters of the camera and the poses for all of the images. This task involves the following sub-tasks:

i) Corner Detection

ii) Finding Intrinsic parameters using Zhang's Algorithm

iii) Finding Extrinsic parameters

iv) Parameter refinement using Non-Linear Least Squares

1 Corner detection

This is the first and most critical task of the camera calibration. To detect the corners, I have used the following steps:

1) Detect lines using Canny edge detector. The canny operator gives a set disconnected lines and will also have multiple lines at the same location because of image noise.

2) After obtaining the lines in the form of a binary mask using canny, I am then using Hough Transformation to get the polar parameters(ρ,θ) of the lines using a suitable threshold. The set of lines obtained using Hough transformation will again have multiple lines at the same location and thus we need to have a mechanism to detect/reject false lines.

3) After Identifying the lines, the next step is to classify lines into the vertical or horizontal lines with the help of the parameter θ.

4) Now we need to deal with the issue of multiple lines for a single class at a time ie either vertical or horizontal. These lines can be either intersecting lines and/or parallel and very close lines and/or non-parallel but close lines. I am using the following criteria for these three types of lines:

   i) For the intersecting lines, I am creating a count array which stores the information whether the line will be removed or not, for every **i-j** pair of lines I am marking **j**th lines which are intersecting in the image region by assigning a value of 1 to the count array. At the end of the loop, the lines which have a count value of 1 will be removed.

ii) For the parallel lines, I have a similar approach except now the decision to assign the value of 1 to the count array depends on the distance between the parallel lines. I am using a relative threshold of 10**-2 as my decision threshold.

iii) Now that we have removed the intersecting and parallel lines, we are left with only the lines which are not parallel and intersecting and yet are very close. For this task, I am using K-means clustering to identify the cluster of lines based on ρ values of the lines with the prior information of how many classes I need to identify ie the number of lines in the pattern.

5) After Cleaning of the lines, The next step is to sort the lines based on their intercept values (X intercepts for vertical lines and Y intercepts for horizontal lines).

6) Now I can number the corners with the help of sorted lines. My numbering scheme is such that the upper left corner gets the value of 1 and then I assign values in increasing order first by rows and then by columns.

7) And finally I am refining the detected corners, using an inbuilt function in OpenCV called 'cornerSubPix'.

After estimating the Corners, in our case 80 in numbers. I am then estimating the Homography between the image and real world using Linear Least squares as we had done in homework 2.

2 Zhang's Algorithm

After Identifying the corners, The next step is to estimate the intrinsic and extrinsic parameters using Zhang's Algorithm which can be broken into two parts:

I) For the first part, we build the V matrix for all the N images to obtain the image of the absolute conic ω. The V matrix is built using the Homography of the N images. For any homography $H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}$, Zhang's Formula for V_{ij} is given as :

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

Note: Here h_i is the i^{th} column of the Homography Matrix and the **notation** h_{ij} is such that the value should be the j^{th} element in the column vector h_i

We then form the equations for finding the image of the absolute conic by constructing the following equation for one image:

$$Vb = 0$$

Where $b = \begin{bmatrix} w_{11} & w_{12} & w_{22} & w_{13} & w_{23} & w_{33} \end{bmatrix}$ and V is :

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

In such a manner we then build the V matrix for all N images and stack them together to solve for $\vec{b}$ in linear least squares sense by finding the Null vector of V corresponding to the smallest singular value.

II) After estimating the image of the absolute conic ω, we can then estimate the intrinsic parameters of the camera using the Zhang's formulas:

$$y_0 = \frac{w_{12}w_{13} - w_{11}w_{23}}{w_{11}w_{22} - w_{12}^2}$$
$$\lambda = w_{33} - \frac{w_{13} + y_0(w_{12}w_{13} - w_{11}w_{23})}{w_{11}}$$
$$\alpha_x = \sqrt{\frac{\lambda}{w_{11}}}$$
$$\alpha_y = \sqrt{\frac{\lambda w_{11}}{w_{11}w_{22} - w_{12}^2}}$$
$$s = -\frac{w_{12}\alpha_x^2\alpha_y}{\lambda}$$
$$x_0 = \frac{sy_0}{\alpha_y} - \frac{w_{13}\alpha_x^2}{\lambda}$$

We then form our Intrinsic camera Matrix as:

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

3 Extrinsic Parameters

After obtaining the estimate of the Intrinsic parameters (K) using Zhang's Algorith, we are now in a position to find the extrinsic parameters ie Rotation and translation for all the images. We estimate them using the following formulas:

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = K \begin{bmatrix} r_1 & r_2 & t \end{bmatrix}$$
$$\Rightarrow K^{-1} \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = \begin{bmatrix} r_1 & r_2 & t \end{bmatrix}$$

$$r_1 = \xi K^{-1} h_1$$
$$r_2 = \xi K^{-1} h_1$$
$$r_3 = \xi r_1 r_2$$
$$t = \xi K^{-1} h_1$$

where $\xi = \frac{1}{||K^{-1}h_1||} = \frac{1}{||K^{-1}h_2||} = \frac{1}{||K^{-1}h_3||}$

We then Obtain our Rotation matrix as $R = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix}$. This matrix is still not orthogonal and needs to be conditioned. We do this by finding the singular value decomposition of R.

$$UDV^T = svd(R)$$

$$R_{conditioned} = UV^T$$

We need to carry out this procedure for all N images and thus we obtain the Extrinsic parameters for all images.

4 Parameter Refinement

After obtaining the estimates of the parameters we now need to refine them using non-linear least squares. For this task we set up the total geometric distance as our cost function. Also, as rotation has only three degrees of freedom we make use of Rodriguez's representation of rotation to convert the R matrix into a three parameter vector $\vec{w} = \begin{bmatrix} w_x & w_y & w_z \end{bmatrix}$.
This three parameter vector is written out into a 'cross-product' representation $[w]_X$:

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

This representation comes from the third order alternating tensor $\xi_{ijk} = \vec{e_i}.(\vec{e_j} x \vec{e_k})$ as any cross product w x u can be written as $\vec{w}$x$\vec{u}$= $\xi_{ijk} w_j u_k = [w]_X \vec{u}$. Thus $[w]_X$ is a second order tensor given by $[w]_X = \xi_{ijk} w_j e_i \otimes e_j$

The Rodriguez's Rotation formula is then given as:

$$R = e^{[w]_X} = I + \frac{sin\phi}{\phi}[w]_X + \frac{1 - cos\phi}{\phi^2}[w]_X^2$$

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

And the Back Transformation is given by:

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

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

Now coming back to parameter refinement, After converting the R rotation matrix obtained from section 3 into three vector parameter $\vec{w}$ using Rodriguez's formula for each image, we can then proceed to setting up a cost function for our Non-Linear Least Squares solver. The cost function is defined as the sum of squares of the geometric distance (d^2_{geom}) between the actual corner and the estimated corner. The cost function and parameters passed will be slightly different when we account for radial distortion or not. The cost function and parameters for the two cases will be as follows:

I) Without Radial Distortion:
The cost function is given as:

$$d^2_{geom} = \Sigma_i \Sigma_j ||x_{ij} - \hat{x}_{ij}||^2 = \Sigma_i \Sigma_j ||x_{ij} - K[R_i t_i] X_{M,j}||^2$$

$$= \Sigma_i \Sigma_j ||x_{ij} - K[R_i t_i] X_{M,j}||^2 = \Sigma_i \Sigma_j ||x_{ij} - K \begin{bmatrix} r_{i,1} & r_{i,2} & t_i \end{bmatrix}||^2$$

and the parameters would be the 5 intrinsic parameters and (3+3)=6 Extrinsic parameters for every image. Therefore our parameter vector for LM becomes of size 5+6*N.

II) With Radial distortion:
The only change here is that instead of using $\hat{x}_{ij}$ for the geometric distance. We account for radial distortion using:

$$r^2 = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2$$
$$\hat{x}_{rad} = \hat{x} + (\hat{x} - x_0)[k_1 r^2 + k_2 r^4]$$
$$\hat{y}_{rad} = \hat{y} + (\hat{y} - y_0)[k_1 r^2 + k_2 r^4]$$

and then use $\hat{x}_{rad}^{\rightarrow}$ as our new estimate to. Thus, we have two more parameters to refine for radial distortion.

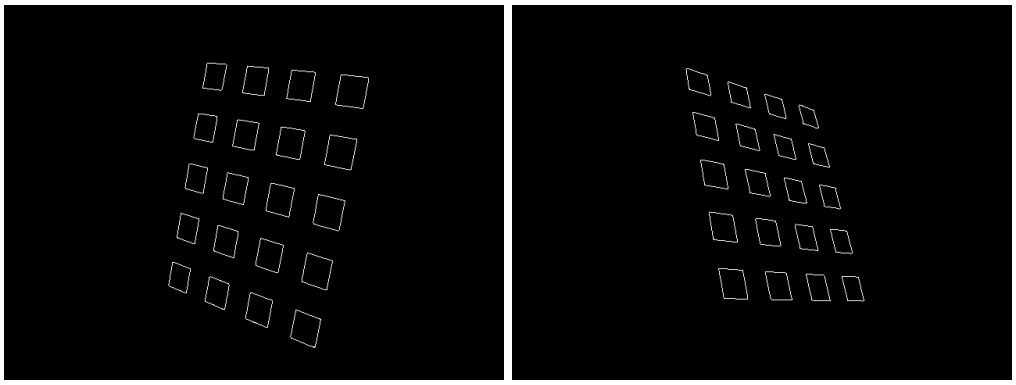
Also, as we will be optimizing our parameters which has the terms $\vec{w}$ in it and for the cost function we need the Rotation Matrix R. Therefore in our cost function we will need to do this conversion using Rodriguez's formula.

Further, as LM may require large number of function calls before giving the optimized values. Therefore, it is best to vectorize the inner loop on j for the cost function to reduce the computational time.

Interestingly, It was observed that LM was not taking a significantly long time.

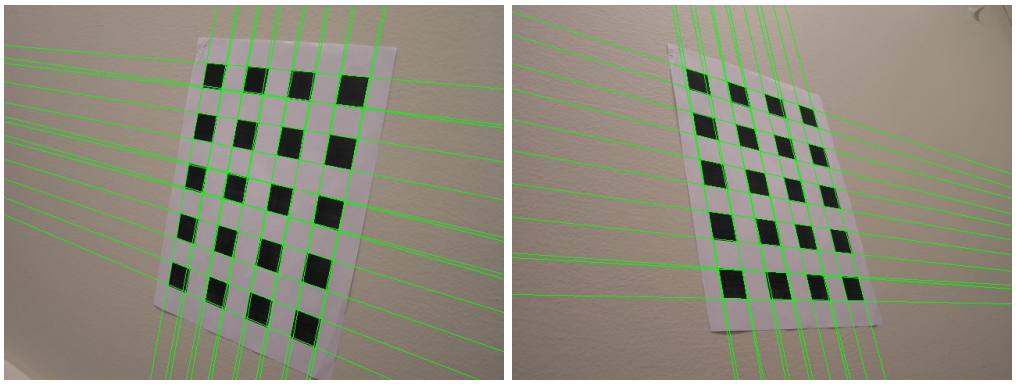
5 Results

5.1 Provided Dataset (Total of 40 Images)



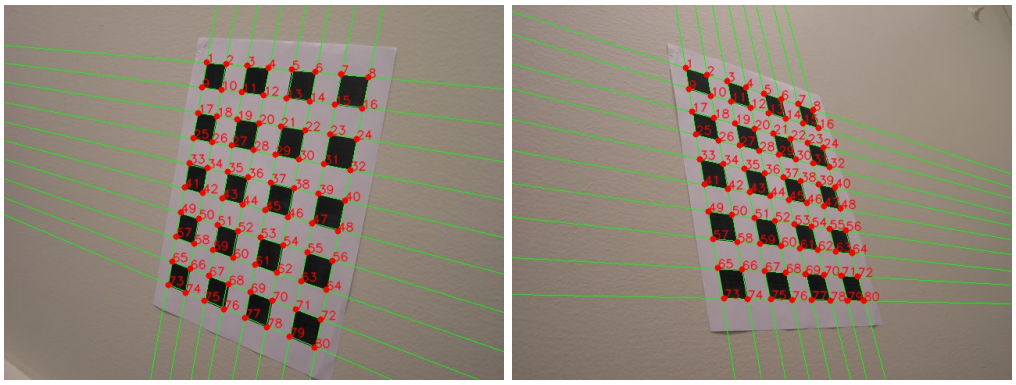
(a) Pic 1 (b) Pic 10

Figure 1: Canny Edge Detection



(a) Pic 1 (b) Pic 10

Figure 2: Raw Hough Lines (has multiple lines)



(a) Pic 1 (b) Pic 10

Figure 3: Refined Hough Lines and labelled corners

$$K = \begin{bmatrix} 722.39863975 & 2.08302863 & 323.24790932 \\ 0. & 722.96090524 & 239.27037781 \\ 0. & 0. & 1. \end{bmatrix}$$

$$R_1 = \begin{bmatrix} 0.78495029 & -0.18442465 & 0.59147323 \\ 0.20072094 & 0.97887818 & 0.03884082 \\ -0.58614345 & 0.08823295 & 0.80538861 \end{bmatrix}$$

$$t_1 = \begin{bmatrix} -47.94965239 & -126.21006969 & 549.26112011 \end{bmatrix}$$

$$R_5 = \begin{bmatrix} 0.98732844 & -0.15816623 & 0.01288364 \\ 0.1586787 & 0.98497094 & -0.0682152 \\ -0.00190067 & 0.06939516 & 0.99758744 \end{bmatrix}$$

$$t_5 = \begin{bmatrix} -54.77512944 & -120.27045784 & 522.97867824 \end{bmatrix}$$

$$R_{10} = \begin{bmatrix} 0.74886637 & 0.1866424 & -0.63589604 \\ 0.1364149 & 0.89556277 & 0.42350714 \\ 0.64852921 & -0.40389595 & 0.64519604 \end{bmatrix}$$

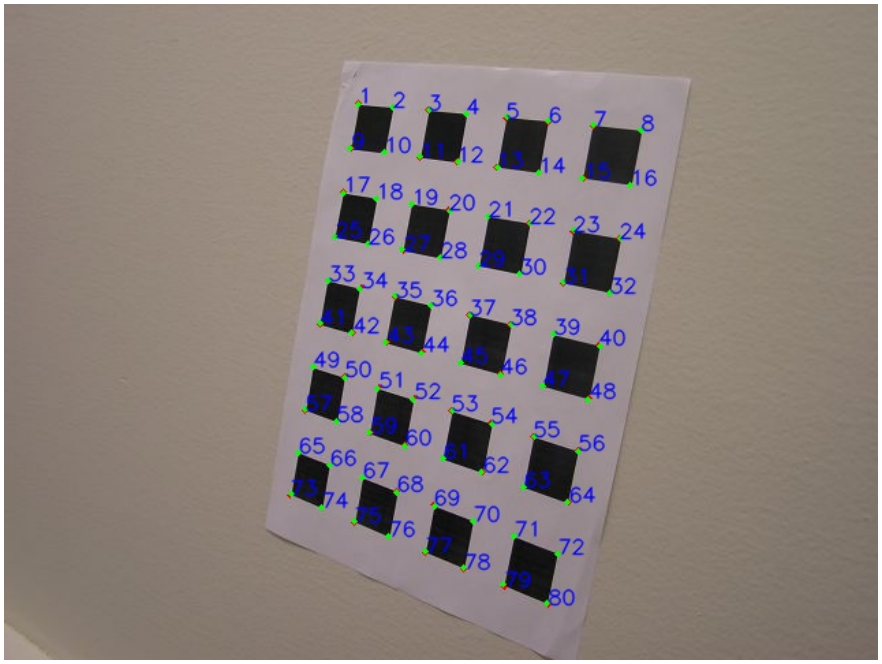
$$t_{10} = \begin{bmatrix} -74.67281702 & -117.91753275 & 533.8682772 \end{bmatrix}$$

$$R_{34} = \begin{bmatrix} 0.93283546 & -0.12599479 & -0.33755491 \\ 0.08699755 & 0.98790887 & -0.12832568 \\ 0.34964186 & 0.09034029 & 0.93251767 \end{bmatrix}$$

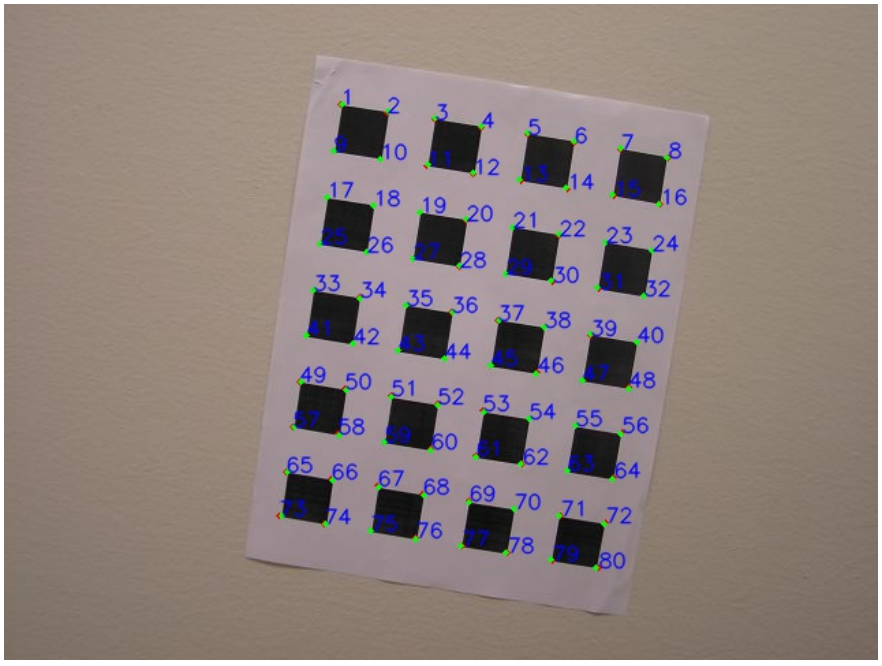
$$t_{34} = \begin{bmatrix} -74.29512163 & -121.84136838 & 390.00455392 \end{bmatrix}$$

Radial Distortion Parameters Estimated:

$$k1, k2 = \begin{bmatrix} -2.90994256331e-07 & 2.73228363513e-13 \end{bmatrix}$$

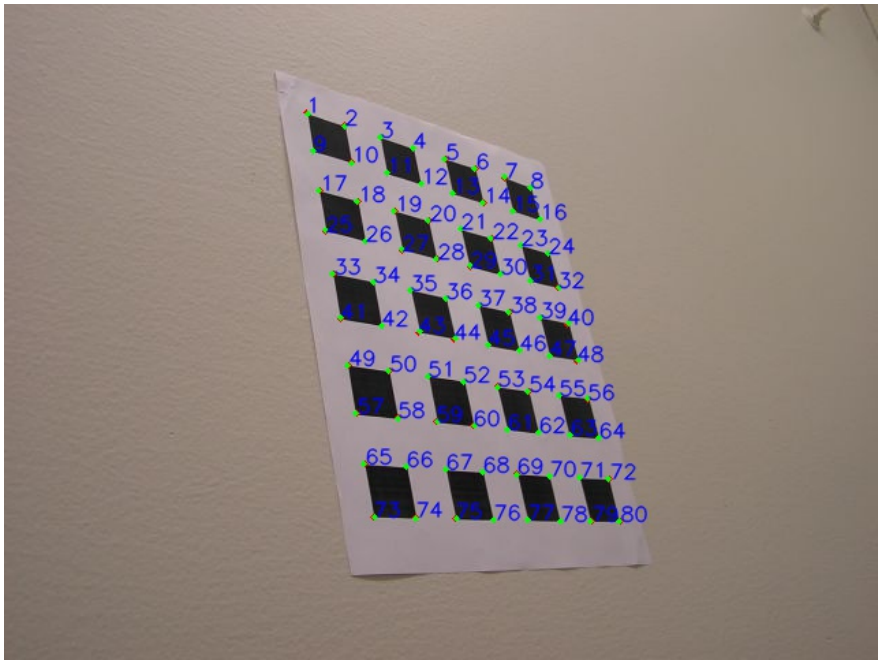


(a) Pic 1

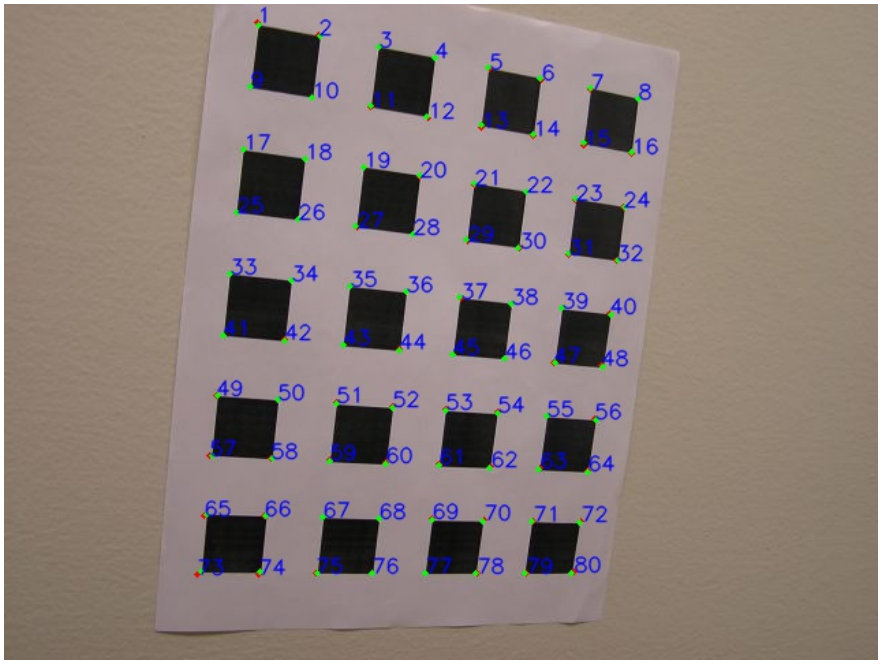


(b) Pic 5

Figure 4: Re-projected points (red:reprojected , green: original)

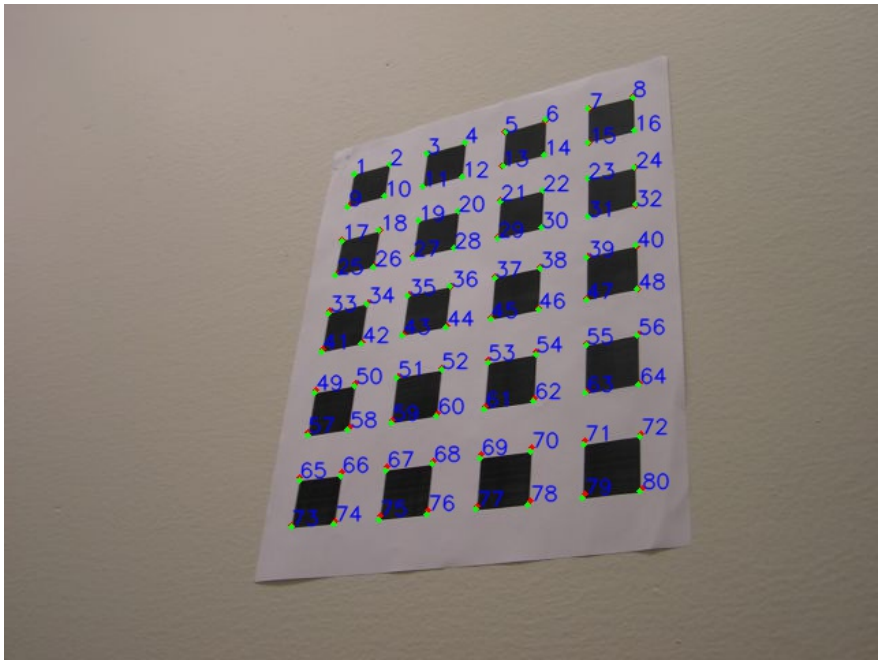


(a) Pic 10

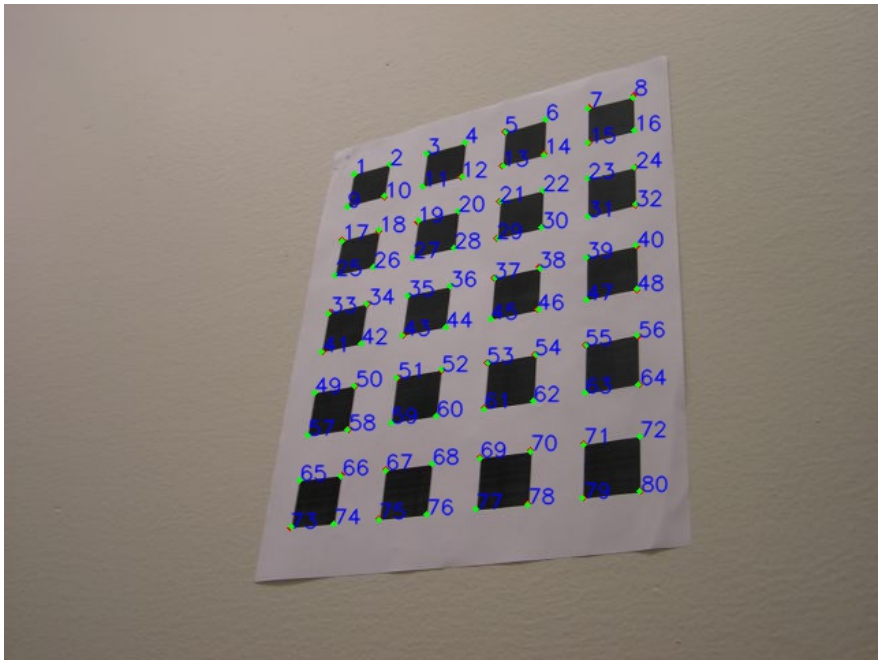


(b) Pic 34

Figure 5: Re-projected points (red:reprojected , green: original)

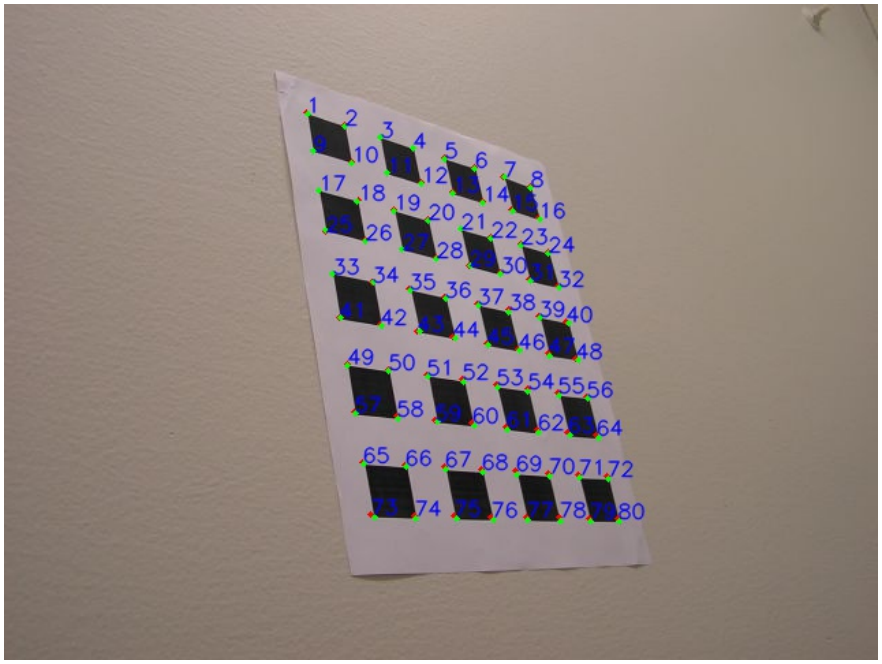


(a) Pic 9 b4 LM

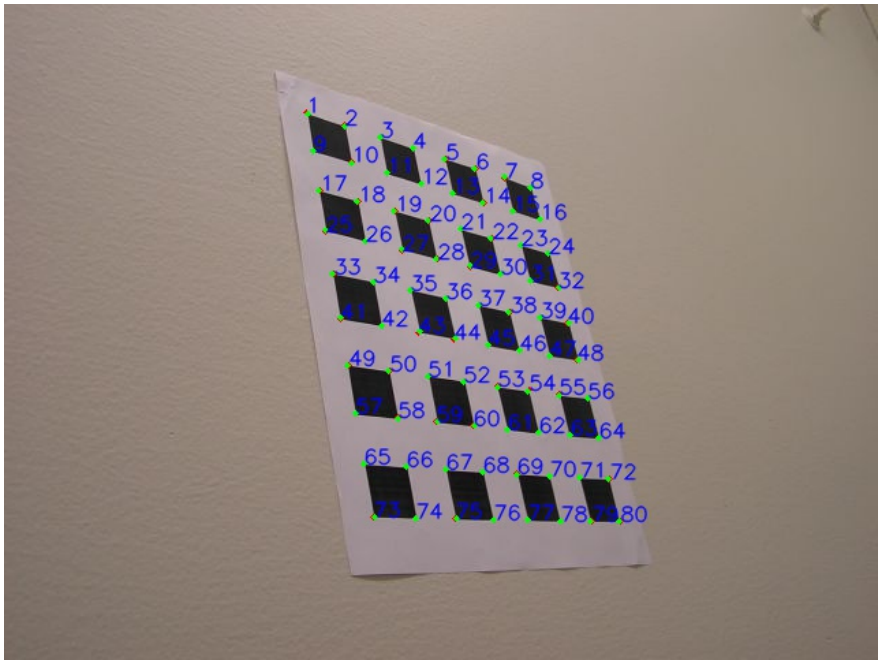


(b) Pic 9 LM

Figure 6: gains of LM (Comparison of linear least squares solution and LM)

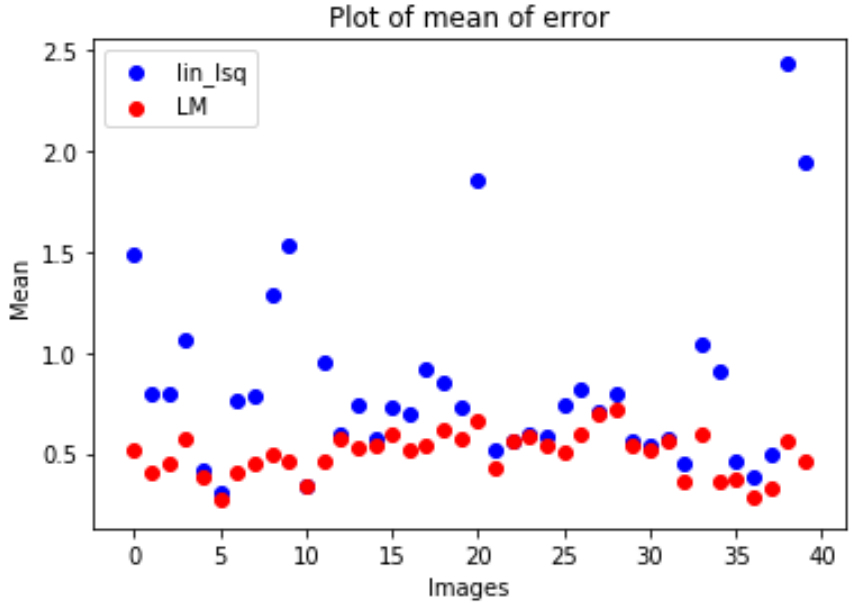


(a) Pic 10 b4 LM

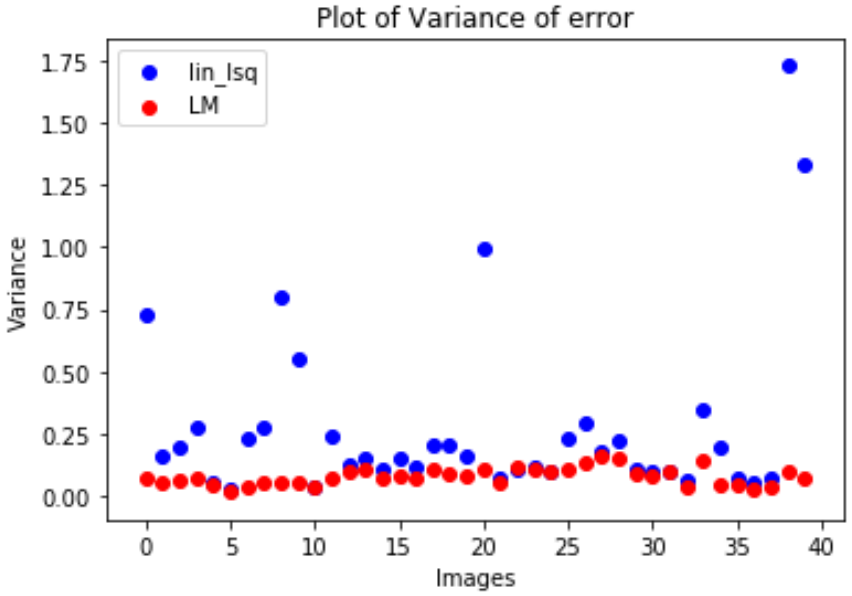


(b) Pic 10 LM

Figure 7: gains of LM (Comparison of linear least squares solution and LM)



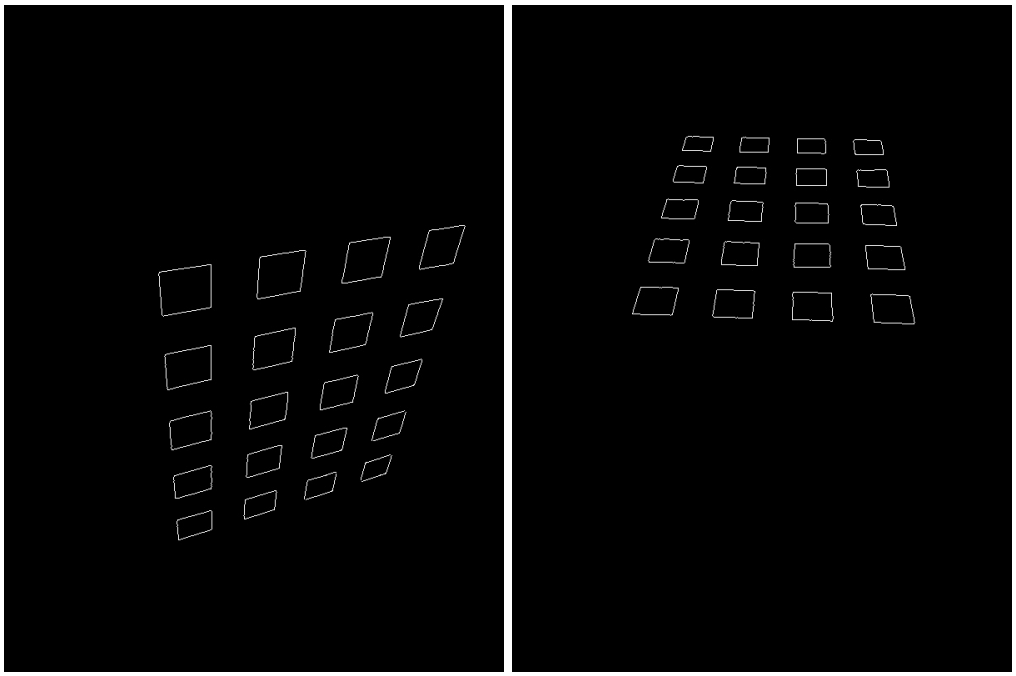
(a) mean



(b) variance

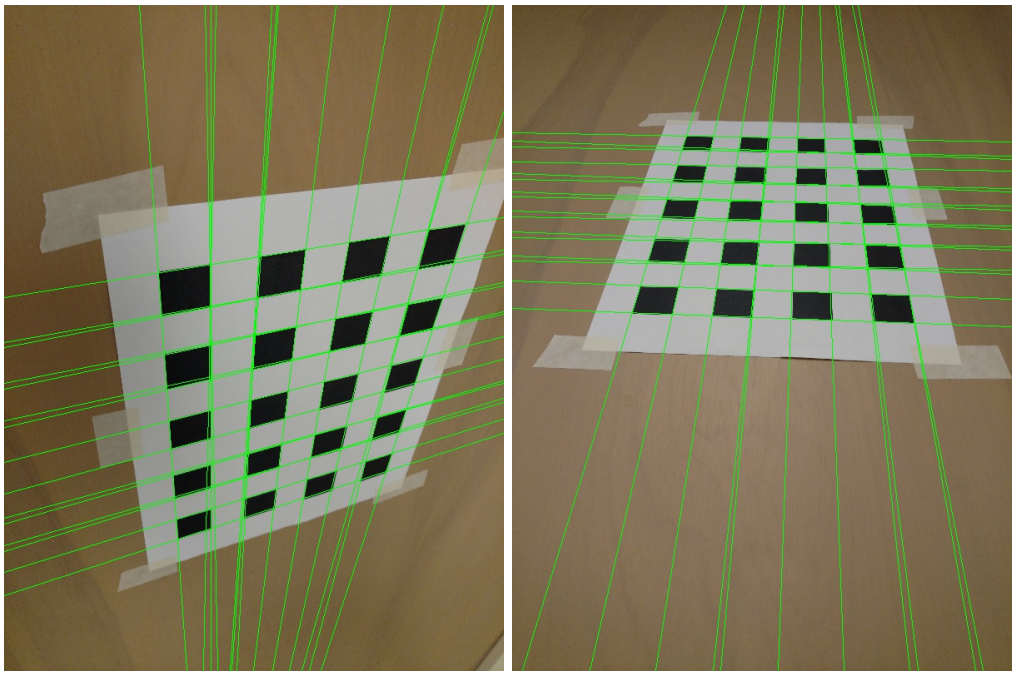
Figure 8: mean and variance of error

5.2 My Dataset (Total of 20 Images)



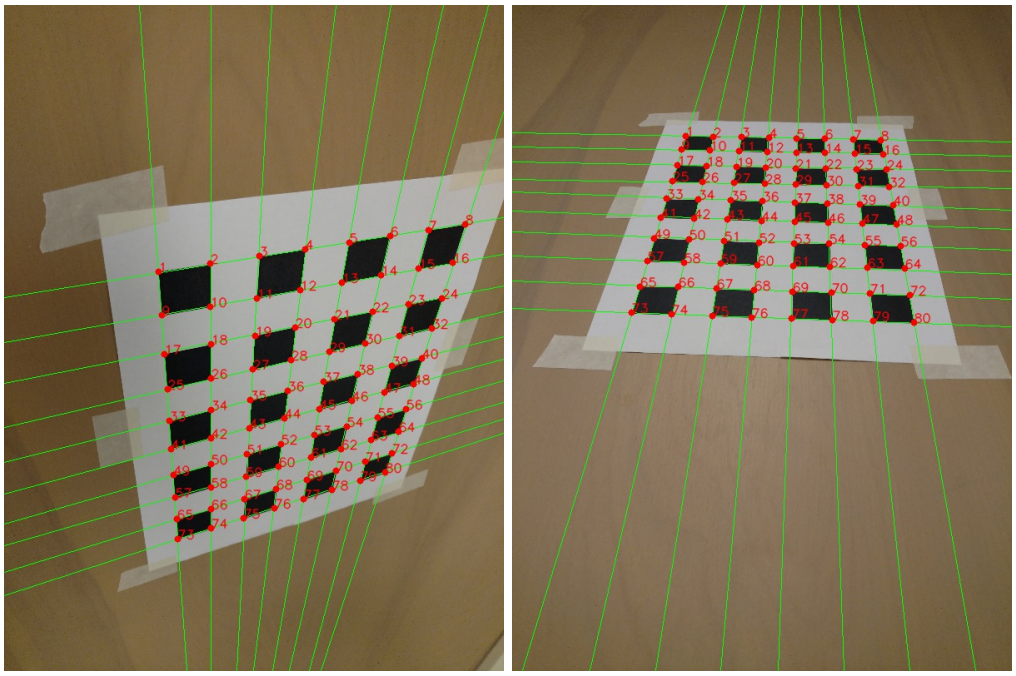
(a) Pic 1 (b) Pic 10

Figure 9: Canny Edge Detection



(a) Pic 1 (b) Pic 10

Figure 10: Raw Hough Lines (has multiple lines)



(a) Pic 1 (b) Pic 10

Figure 11: Refined Hough Lines and labelled corners

$$K = \begin{bmatrix} 648.59156045 & -0.96936208 & 305.08785562 \\ 0. & 646.20029932 & 397.33638047 \\ 0. & 0. & 1. \end{bmatrix}$$

Note: The first image was taken in such a way that the Principal axis was approximately perpendicular to the pattern and also the x-axis of image was roughly along the horizontal axis of the pattern. This means that the **ground-truth** value of R_1 should be I_{3x3} (ie no rotation) and that seems to be the case.

$$R_1 = \begin{bmatrix} 0.99989253 & 0.01123401 & -0.00941987 \\ -0.01150349 & 0.9995115 & -0.02905917 \\ 0.00908882 & 0.02916441 & 0.99953331 \end{bmatrix}$$

$$t_1 = \begin{bmatrix} -85.36780025 & -111.83310434 & 386.34466365 \end{bmatrix}$$

$$R_2 = \begin{bmatrix} 0.98585291 & 0.01200621 & -0.16718221 \\ -0.01754915 & 0.99934281 & -0.03171723 \\ 0.16669153 & 0.03420243 & 0.98541571 \end{bmatrix}$$

$$t_2 = \begin{bmatrix} -65.51639203 & -102.2268285 & 441.39259791 \end{bmatrix}$$

$$R_3 = \begin{bmatrix} 9.57386031e-01 & 3.95463517e-04 & -2.88811066e-01 \\ -1.04035166e-02 & 9.99397288e-01 & -3.31183857e-02 \\ 2.88623899e-01 & 3.47117305e-02 & 9.56813117e-01 \end{bmatrix}$$

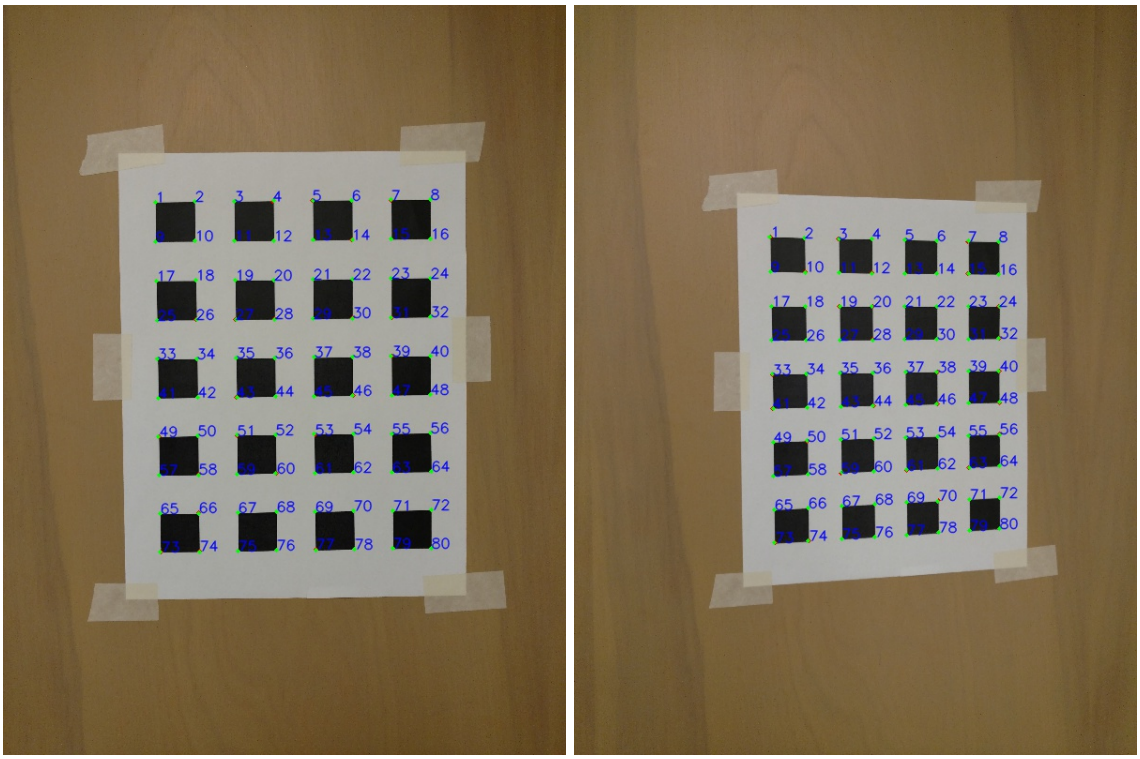
$$t_3 = \begin{bmatrix} -52.91659396 & -121.77692599 & 454.36672719 \end{bmatrix}$$

$$R_4 = \begin{bmatrix} 0.91629823 & -0.00606833 & -0.40045066 \\ -0.00109178 & 0.99984364 & -0.01764956 \\ 0.40049515 & 0.01660946 & 0.91614833 \end{bmatrix}$$

$$t_4 = \begin{bmatrix} -6.85437414 & -77.6459047 & 492.02542694 \end{bmatrix}$$

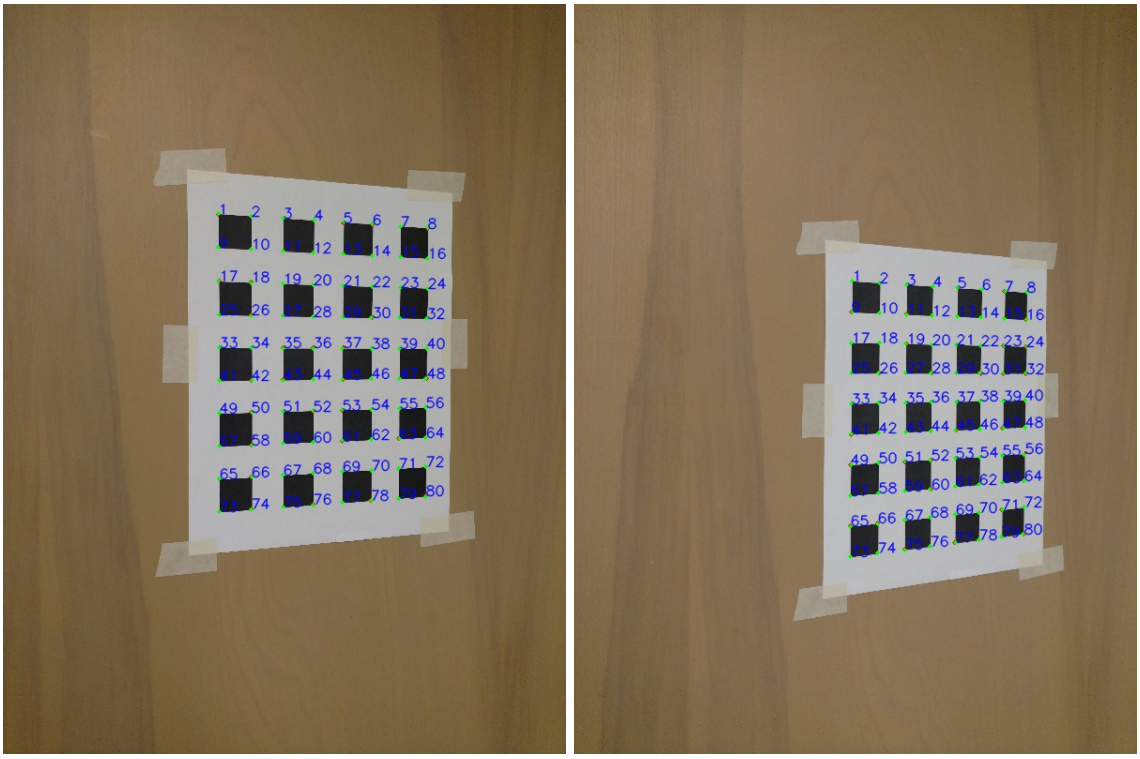
Radial Distortion Parameters Estimated:

$$k1, k2 = \begin{bmatrix} 2.613778217 - 07 & -2.67921218401e - 12 \end{bmatrix}$$



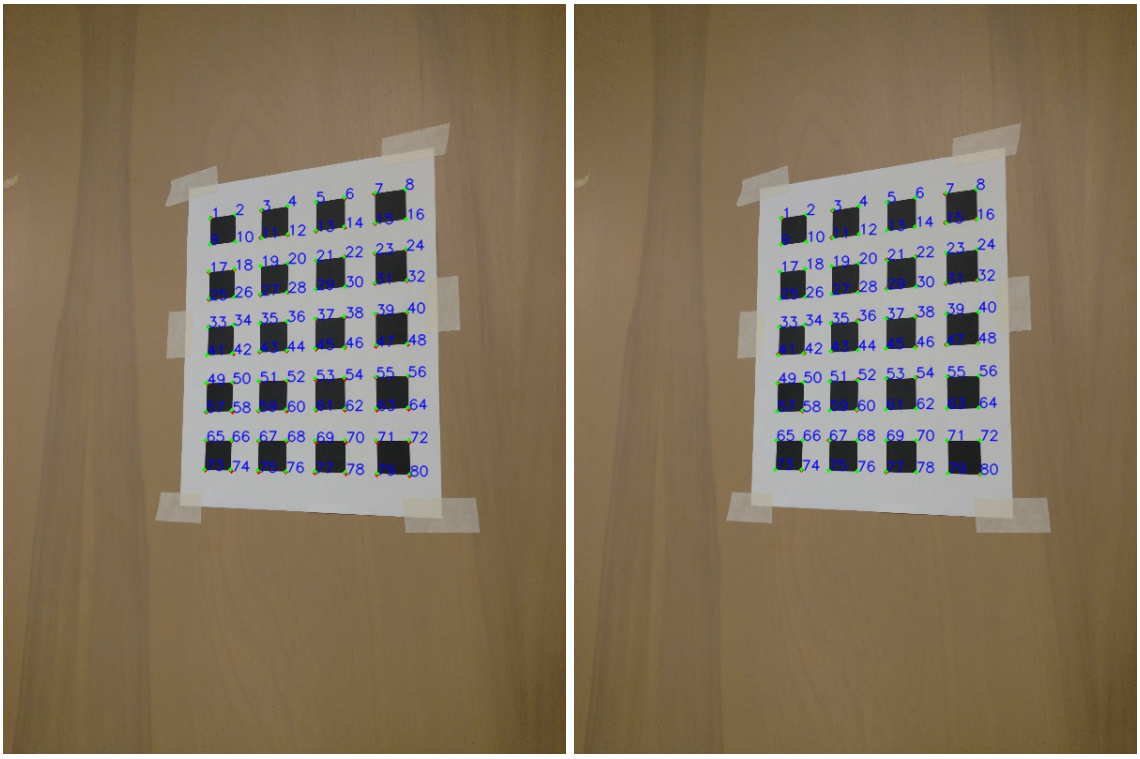
(a) Pic 1 (b) Pic 2

Figure 12: Re-projected points (red:reprojected , green: original)



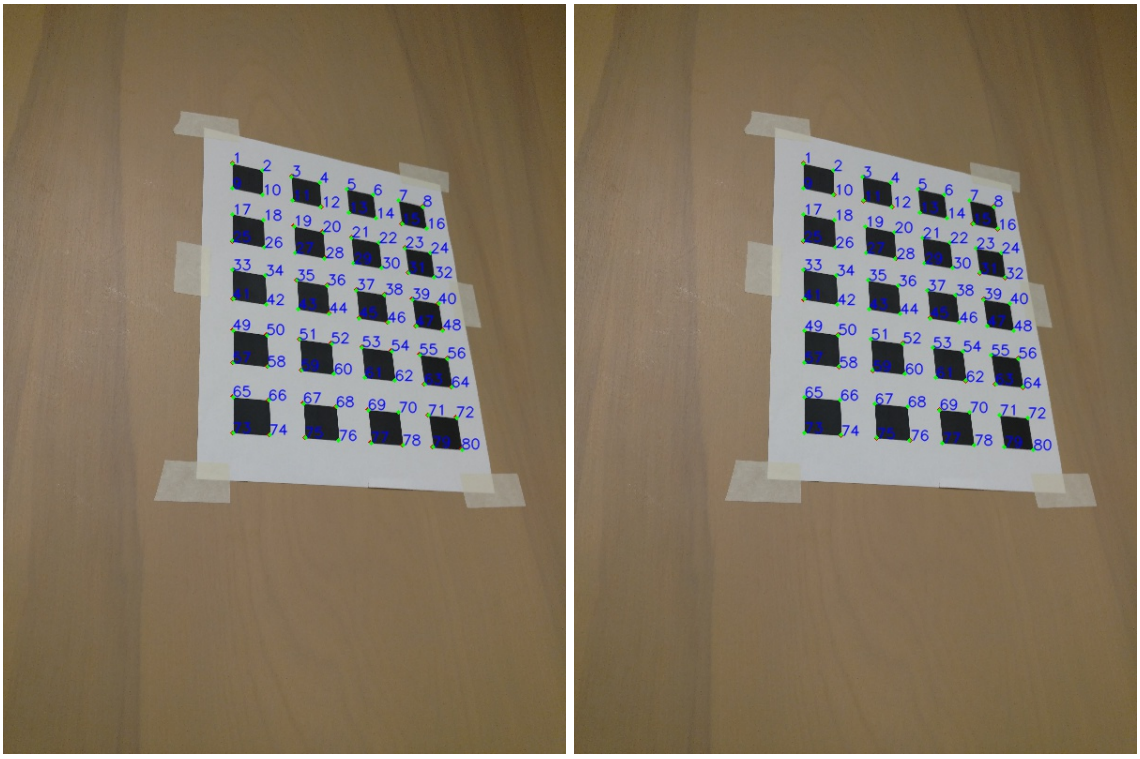
(a) Pic 3 (b) Pic 4

Figure 13: Re-projected points (red:reprojected , green: original)



(a) Pic 5 b4 LM (b) Pic 5 LM

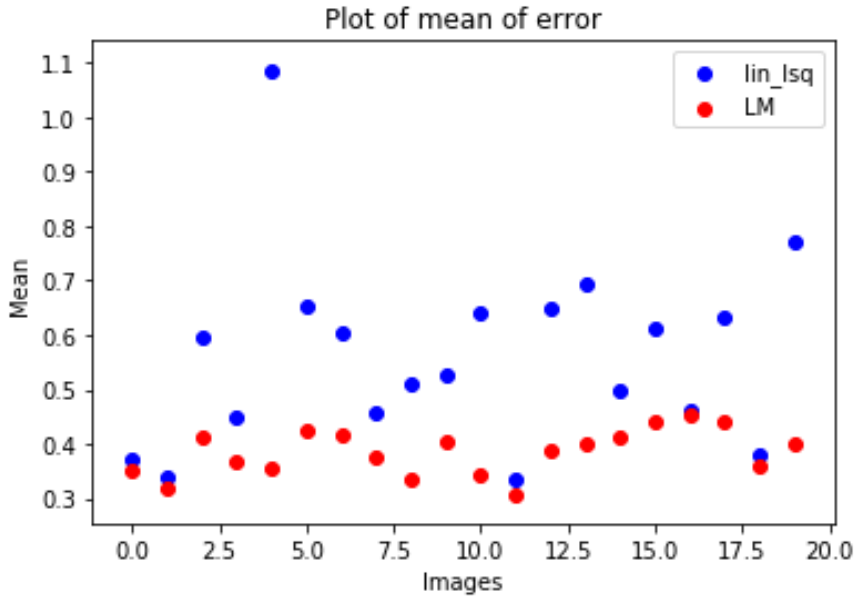
Figure 14: gains of LM (Comparison of linear least squares solution and LM)



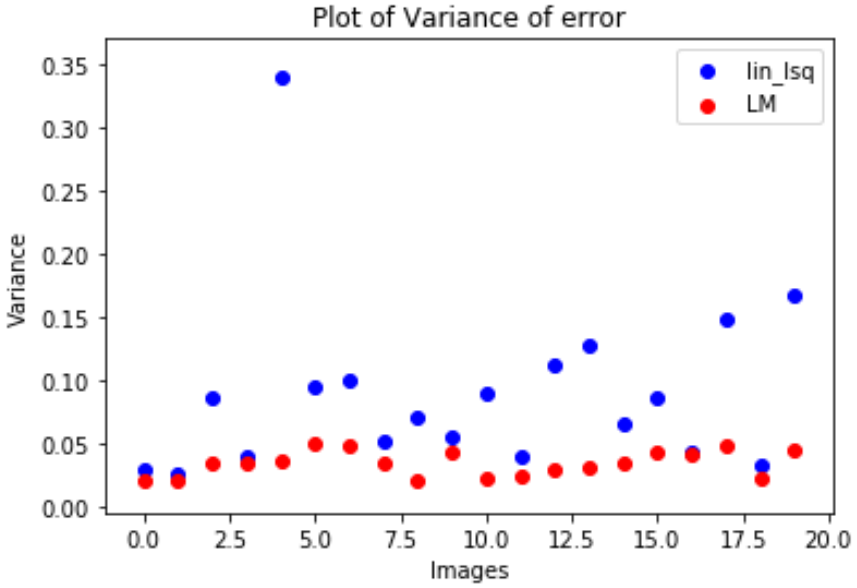
(a) Pic 11 b4 LM

(b) Pic 11 LM

Figure 15: gains of LM (Comparison of linear least squares solution and LM)



(a) mean



(b) variance

Figure 16: mean and variance of error

6 Source Code:

6.1 Main File

```
"""
ECE661: hw8 main file
@author: Rahul Deshmukh
email: deshmuk5@purdue.edu
PUID: 0030004932
"""
#%%
# import libraries
import cv2
import os
import numpy as np
import sys
sys.path.append('../../')
import MyCVModule as MyCV

#%%
#----------------------Corner detection-----------------#
# define readpath
readpath='../Files/Dataset1/raw'
savepath='../Files/Dataset1/'
img_list=os.listdir(readpath)
# calibration pattern grid def
pattern_dim=(5,4)
# world coordinates of the grid
# measured sizes of grid
sq=25.0 # length of side of square in the pattern
world_coord=[]
for i in range(2*pattern_dim[0]):
    for j in range(2*pattern_dim[1]):
        world_coord.append([j*sq,i*sq])

# define thresholds for Canny
minVal=200 #200 for provided 500 for my
maxVal=500  #500 for provided  600 for my
# Hough transform related constants
HoughThresh=52 # 52 for provided dataset 50 for my
font = cv2.FONT_HERSHEY_SIMPLEX

# loop over dataset images
# initialize
H=[] # list to store iH for all images
V=[] # list to store zhangs V for all images
AllCorners=[] # list to store all corners in the images
for i in range(len(img_list)):
    # read image in color
    color_img=cv2.imread(readpath+'/Pic_'+str(i+1)+'.jpg',1)
    gray_img=cv2.imread(readpath+'/Pic_'+str(i+1)+'.jpg',0)
    # find image size
    m,n=gray_img.shape
    # find lines using canny and hough
    edges,lines=MyCV.findLines(gray_img,minVal,maxVal,
                               HoughThresh)
    cv2.imwrite(savepath+'edges/Pic_'+str(i+1)+'.jpg',edges)
    # draw lines onto the color image
    temp_img=MyCV.drawLine_old(color_img,lines)
    cv2.imwrite(savepath+'labelled/Pic_'+str(i+1)+'temp.jpg',temp_img)
    # find corners
    corners,hor_lines,ver_lines=MyCV.findCorners(lines,(m,n),pattern_dim)
    #draw hor lines and ver lines
    draw_img=MyCV.drawLine(color_img,hor_lines)
```

```
        draw_img=MyCV.drawLine(draw_img,ver_lines)
        corners=MyCV.refine_corners(gray_img,corners,10)
        AllCorners.append(corners)
        for j in range(len(corners)):
            draw_img=cv2.circle(draw_img,(int(corners[j][0]),int(corners[j][1])),4,(0,0,255),-1)
            draw_img=cv2.putText(draw_img,str(j+1),(int(corners[j][0]),int(corners[j][1])), font
                ,0.5,(0,0,255),1,cv2.LINE_AA)
        cv2.imwrite(savepath+'labelled/Pic_'+str(i+1)+'.jpg',draw_img)
        # find homography using world coordinates
        iH=MyCV.HMat_Pts(np.array(world_coord),np.array(corners))
        H.append(iH)
        iV=MyCV.Zhang_V(iH)
        V.append(iV)

np.save('./saved_data/V.npy',V)
np.save('./saved_data/H.npy',H)
np.save('./saved_data/AllCorners.npy',AllCorners)
#-------------------------------------------------------------#
#%%
#---------------------Zhang's Algo II-----------------------#
# find w from the equation Vb=0 using svd and null vector
# convert V list to a matrix
Vmat=np.zeros((2*len(img_list),6),dtype=np.float32)
for i in range(len(img_list)): Vmat[2*i:2*i+2,:]=V[i]
# do SVD of Vmat
u,d,vt=np.linalg.svd(Vmat)
b=vt[-1,:]# last row of vt
w=np.array([[b[0],b[1],b[3]],
            [b[1],b[2],b[4]],
            [b[3],b[4],b[5]]])
# find intrinsic parameters of K from w
y0=(w[0,1]*w[0,2]-w[0,0]*w[1,2])/(w[0,0]*w[1,1]-w[0,1]**2)
lam=w[2,2]-(w[0,2]**2+y0*(w[0,1]*w[0,2]-w[0,0]*w[1,2]))/(w[0,0])
a_x=np.sqrt(lam/w[0,0])
a_y=np.sqrt((lam*w[0,0])/(w[0,0]*w[1,1]-w[0,1]**2))
s=-1*(w[0,1]*(a_x**2)*(a_y))/(lam)
x0= s*y0/a_y - w[0,2]*a_x**2/lam
K=np.array([[a_x,s,x0],
            [0,a_y,y0],
            [0,0,1]])
np.save('./saved_data/K.npy',K)
#-------------------------------------------------------------#
#%%
#--------Extrinsic Parameters: R and t--------------------#
R = [] # list to store R
t = [] # list to store t
w = [] # list to store w: rodriguez rep of Rot
K_inv=np.linalg.inv(K)
for i in range(len(img_list)):
    y=1/np.linalg.norm(K_inv@H[i][:,0]) # constant for scaling
    r1= y*(K_inv@(H[i][:,0]))
    r2= y*(K_inv@(H[i][:,1]))
    r3= np.cross(r1,r2)
    it= y*(K_inv@(H[i][:,-1]))
    iR=np.array([r1.T,r2.T,r3.T])
    iR=iR.T
    # conditioning of R
    u,d,vt=np.linalg.svd(iR)
    iR=u@vt
    R.append(iR)
    t.append(it)
    # find w using rodriguez
    iw=MyCV.Rot_mat2vec(iR)
    w.append(iw)
```

```
np.save('./saved_data/R.npy',R)
np.save('./saved_data/t.npy',t)
np.save('./saved_data/w.npy',w)
#-------------------------------------------------------------#
#%%
#-------------Reprojection with linear least squares estimate--------#
linlsq_savepath=savepath+'reproject_linlsq/'
mean_linlsq=[]
var_linlsq=[]
for i in range(len(img_list)):
    img=cv2.imread(readpath+'/Pic_'+str(i+1)+'.jpg',1)
    rep_img,imean,ivar=MyCV.ReprojectPoints(img,world_coord,
                                            AllCorners[i],K,R[i],t[i])
    mean_linlsq.append(imean)
    var_linlsq.append(ivar)
    cv2.imwrite(linlsq_savepath+'Pic_'+str(i+1)+'.jpg',rep_img)

np.save('./saved_data/mean_linlsq.npy',mean_linlsq)
np.save('./saved_data/var_linlsq.npy',var_linlsq)
#-------------------------------------------------------------------#
#%%
#-------------Refinement of Calibration Parameters-------------#
rad_dist = 0 # user parameter defining to include radial distortion
k1,k2= np.zeros(2) # initialization fo radial distortion parameters
#k1=1;k2=1;
# prepare initial solution p for LM
if rad_dist==0:
    # p= [K,w1,t1,w2,t2,...wn,tn]
    p0=np.zeros(5+6*len(img_list))
    p0[:5]=np.array([a_x,a_y,s,x0,y0])
    for i in range(len(img_list)):
        p0[6*i+5:6*i+8]=w[i]
        p0[6*i+8:6*i+11]=t[i]
else:
    # p = [K,w1,t1,w2,t2,...wn,tn,k1,k2]
    p0=np.zeros(7+6*len(img_list))
    p0[:5]=np.array([a_x,a_y,s,x0,y0])
    for i in range(len(img_list)):
        p0[6*i+5:6*i+8]=w[i]
        p0[6*i+8:6*i+11]=t[i]
    p0[-2]=k1;   p0[-1]=k2
# call to LM for refining the parameters
from scipy.optimize import least_squares
import time

start_time=time.time()

if rad_dist==0:
    optim=least_squares(MyCV.CostFun_cam_cal_linear,p0,
                        method='lm',args=(AllCorners,world_coord))
    p_star=optim['x']
    np.save('./saved_data/p_star_linear.npy',p_star)
else:
    optim=least_squares(MyCV.CostFun_cam_cal_radial,p0,
                        method='lm',args=(AllCorners,world_coord))
    p_star=optim['x']
    np.save('./saved_data/p_star_radial.npy',p_star)
np.save('./saved_data/optim.npy',optim)
#-------------------------------------------------------------#
end_time=time.time()
total_time=end_time-start_time
np.save('./saved_data/total_time.npy',total_time)
#%%
# make final refined K_ref and R_ref from p_star
a_x=p_star[0]; a_y=p_star[1]; s=p_star[2]
```

```
x0=p_star[3]; y0=p_star[4];
if rad_dist==1:
    k1=p_star[-2]; k2=p_star[-1]
    print('Radial Distortion parameters: k1='+str(k1)+' k2='+str(k2))
K_ref=np.array([[a_x,s,x0],
                [0,a_y,y0],
                [0,0,1]])
R_ref=[]
t_ref=[]
for i in range(len(img_list)):
    iw=p_star[6*i+5:6*i+8]
    it=p_star[6*i+8:6*i+11]
    iR=MyCV.Rot_vec2mat(iw)
    R_ref.append(iR)
    t_ref.append(it)

np.save('./saved_data/K_ref.npy',K_ref)
np.save('./saved_data/R_ref.npy',R_ref)
np.save('./saved_data/t_ref.npy',t_ref)
#%%
#-------------------------Reprojection-----------------------------------------#
linlsq_savepath=savepath+'reproject_LM/'
mean_LM=[]
var_LM=[]
for i in range(len(img_list)):
    img=cv2.imread(readpath+'/Pic_'+str(i+1)+'.jpg',1)
    rep_img,imean,ivar=MyCV.ReprojectPoints(img,world_coord,AllCorners[i],
                                            K_ref,R_ref[i],t_ref[i])
    mean_LM.append(imean)
    var_LM.append(ivar)
    cv2.imwrite(linlsq_savepath+'Pic_'+str(i+1)+'.jpg',rep_img)
np.save('./saved_data/mean_LM.npy',mean_LM)
np.save('./saved_data/var_LM.npy',var_LM)
#------------------------------------------------------------------------------#
# make plots of mean and var for both cases linlsq and LM
import matplotlib.pyplot as plt
plt.scatter(np.arange(len(mean_linlsq)),mean_linlsq,c='b',label='lin_lsq')
plt.scatter(np.arange(len(mean_LM)),mean_LM,c='r',label='LM')
plt.xlabel('Images')
plt.ylabel('Mean')
plt.title('Plot_of_mean_of_error')
plt.legend()
plt.show()
plt.scatter(np.arange(len(var_linlsq)),var_linlsq,c='b',label='lin_lsq')
plt.scatter(np.arange(len(var_LM)),var_LM,c='r',label='LM')
plt.xlabel('Images')
plt.ylabel('Variance')
plt.title('Plot_of_Variance_of_error')
plt.legend()
plt.show()
```

6.2 Functions

```
#%%
#-------------------------------------HW8--------------------------------------#
#%%
# function for reprojecting the world points onto the image
def ReprojectPoints(img,world_coord,Corners,K,R,t):
    """
    Input: img: colored image
           world_coord: list of list of coordinates [[x1,y1],[x2,y2],...]
           corners: list of list of original coordinates of corners [[x1,y1],[x2,y2],...]
           K: Intrinsic parameter matrix 3x3
           R: Rotation matrix for this image 3x3
           t: translation vector for this image 3x1
```

```
    Output: rep_img: img with reprojected points color image
            mean_e mean of error using Euclidean distance
            var_e: variance of error using Euclidean distance
    """
    # convert world_coord to HC
    X_hc= np.ones((len(world_coord),3))
    X_hc[:,:-1]=np.array(world_coord)
    X_hc=X_hc.T # hc coordinates as col vectors
    # make camera projection matrix P
    P= np.array([R[:,0].T,R[:,1].T,t.T])
    P=K@P.T
    #find reprojected points
    rep_pt_hc= P@X_hc
    # convert to physical coordinates for plotting
    rep_pt= np.linalg.inv(np.diag(rep_pt_hc[-1,:]))@rep_pt_hc.T
    rep_pt=rep_pt.T # physical coordinates as col vectors
    rep_pt=rep_pt[:-1,:]
    # find Euclidean distance error, mean and var
    e=np.array(Corners).T-rep_pt
    e=np.linalg.norm(e,axis=0)
    mean_e=np.mean(e)
    var_e=np.var(e)
    # plot corners on image
    rep_img=np.copy(img)
    font = cv2.FONT_HERSHEY_SIMPLEX
    for i in range(len(world_coord)):
        rep_img=cv2.circle(img,(int(rep_pt[0,i]),int(rep_pt[1,i])),1,(0,0,255),-1)
        rep_img=cv2.circle(img,(int(Corners[i][0]),int(Corners[i][1])),1,(0,255,0),-1)
        rep_img=cv2.putText(img,str(i+1),(int(rep_pt[0,i]),int(rep_pt[1,i])), font
            ,0.5,(255,0,0),1,cv2.LINE_AA)
    return(rep_img,mean_e,var_e)
#%%
# function for getting the cost function for camera calibration
# with radial distortion
def CostFun_cam_cal_radial(p,x,x_m):
    """
    Input: p: parameters of the cost function, is of size 5+N*6
           format: [K,w1,t1,w2,t2,...wn,tn,k1,k2] where
           K=a_x,a_y,s,x0,y0
           wi=wx,wy,wz
           ti=tx,ty,tz
           all contatenated into a single vector
           x: real coordinates of the corners in i images
           x_m: model coordinates in real world
    Output: cost function value: a scalar value of sum of squares of error
    """
    # make K: intrinsic matrix
    a_x=p[0]; a_y=p[1]; s=p[2]
    x0=p[3]; y0=p[4]; k1=p[-2]; k2=p[-1]
    K=np.array([[a_x,s,x0],
                [0,a_y,y0],
                [0,0,1]])
    # make Rotation matrices R
    num_img=int((len(p)-7)/6)
    # double loop for finding dgeom**2
    N=len(x_m)
    cost=np.zeros(2*num_img*N)
    for i in range(num_img):
        iw=p[6*i+5:6*i+8]
        it=p[6*i+8:6*i+11]
        iR=Rot_vec2mat(iw)
        est_map=np.array([iR[:,0].T,iR[:,1].T,it.T])
        est_map=K@(est_map.T) # mapping function for finding estimate
        xij=np.array(x[i]); xij=xij.T # coordinates as col vectors
        x_m_hc=np.ones((len(x_m),3)); x_m_hc[:,:-1]=np.array(x_m)
```

```
        x_m_hc=x_m_hc.T # coordinates as col vectors
        # find estimate using pinhole model
        x_hat_hc=est_map@x_m_hc
        x_hat=np.linalg.inv(np.diag(x_hat_hc[-1,:]))@x_hat_hc.T
        x_hat=x_hat.T; x_hat=x_hat[:-1,:] # physical coordinate
        # find coordinates with radial distortion
        diff=x_hat-(np.kron(np.array([x0,y0]),np.ones((N,1)))).T  # differnces as col
            vectors
        r_2=np.sum(np.square(diff),axis=0) #sqauring and summing
        m=k1*r_2+k2*np.square(r_2)
        m=np.vstack((m,m))
        x_hat_rad = x_hat + np.multiply(m, diff)
        temp= xij-x_hat_rad
        cost[i*2*N:(i+1)*2*N]=np.hstack((temp[0,:],temp[1,:]))
    return(cost)
# function for getting the cost function for camera calibration
# without radial distortion
def CostFun_cam_cal_linear(p,x,x_m):
    """
    Input: p: parameters of the cost function, is of size 5+N*6 : pinhole model
           format: [K,w1,t1,w2,t2,...wn,tn] where
           K=a_x,a_y,s,x0,y0
           wi=wx,wy,wz
           ti=tx,ty,tz
           all contatenated into a single vector
           x: real coordinates of the corners in i images in the format:
              [corners1,corners2,.....cornersN] , where corners1 is another list of the form
              [[x1,y1],[x2,y2],.....,[xn,yn]]: n= number of corners in our calibration
                  pattern
           x_m: model coordinates in real world in the format of list of list of coordinates
                [[x1,y1],[x2,y2],.....,[xn,yn]], note same will be repeated N times for all N
                    images
    Output: cost function value: a scalar value of sum of squares of error
    """
    # make K: intrinsic matrix
    a_x=p[0]; a_y=p[1]; s=p[2]
    x0=p[3]; y0=p[4]
    K=np.array([[a_x,s,x0],
                [0,a_y,y0],
                [0,0,1]])
    # make Rotation matrices R
    num_img=int((len(p)-5)/6)
    # double loop for finding dgeom**2
    N=len(x_m)
    cost=np.zeros(2*num_img*N)
    for i in range(num_img):
        iw=p[6*i+5:6*i+8]
        it=p[6*i+8:6*i+11]
        iR=Rot_vec2mat(iw)
        est_map=np.array([iR[:,0].T,iR[:,1].T,it.T])
        est_map=K@(est_map.T) # mapping function for finding estimate
        xij=np.array(x[i]); xij=xij.T # coordinates as col vectors
        x_m_hc=np.ones((len(x_m),3)); x_m_hc[:,:-1]=np.array(x_m)
        x_m_hc=x_m_hc.T # coordinates as col vectors
        # find estimate using pinhole model
        x_hat_hc=est_map@x_m_hc #estimate
        x_hat=np.linalg.inv(np.diag(x_hat_hc[-1,:]))@x_hat_hc.T
        x_hat=x_hat.T; x_hat=x_hat[:-1,:] # physical coordinate
        temp= xij-x_hat
        cost[i*2*N:(i+1)*2*N]=np.hstack((temp[0,:],temp[1,:]))
    return(cost)
#%%
# function for obtaining Rotation matrix from w vector
def Rot_vec2mat(w):
    """
```

```
    Input: w: col vector of size 3: three parameters of rotation [wx,wy,wz]
    Output: R: np.array of size 3x3: The rotation matrix
    Using Rodriguez Rotation Formula
    """
    # make Wx from w
    Wx=np.array([[0,-1*w[2],w[1]],
                 [w[2],0,-1*w[0]],
                 [-1*w[1],w[0],0]])
    phi=np.linalg.norm(w)
    R=np.eye(3) + (np.sin(phi)/phi)*(Wx) + ((1-np.cos(phi))/phi**2)*(Wx@Wx)
    return(R)
# function for obtaining w vector from Rotation matrix
def Rot_mat2vec(R):
    """
    Input: R: Rotation matrix, format 3x3 np.array
    Output: w: size 3 vector, format np.array [wx,wy,wz]
    """
    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(w)
#%%
# ----------Zhang's Algo I----------#
# function for finding V from H for one image
def Zhang_V(iH):
    """
    Input: iH: Homography for ith image: size 3x3
    Output: V: matrix for solving for abs conic size 2x6
    """
    # build iV matrix for w: image of abs conic
    iv_12=Zhang_Vij(iH,0,1)
    iv_11=Zhang_Vij(iH,0,0)
    iv_22=Zhang_Vij(iH,1,1)
    iV=np.array([iv_12.T,(iv_11-iv_22).T])
    return(iV)
#----------#
# function for making V from H for Zhang's
def Zhang_Vij(H,i,j):
    """
    Input: H: 3x3 homography st H*world=img
           i,j: indices
    Output: v: vector for zhang's algorithm
    """
    # b=[w11 w12 w22 w13 w23 w33]
    v=np.zeros(6)
    v[0]=H[0,i]*H[0,j]
    v[1]=H[0,i]*H[1,j]+H[1,i]*H[0,j]
    v[2]=H[1,i]*H[1,j]
    v[3]=H[2,i]*H[0,j]+H[0,i]*H[2,j]
    v[4]=H[2,i]*H[1,j]+H[1,i]*H[2,j]
    v[5]=H[2,i]*H[2,j]
    return(v)
#%%
# function for refining corners
def refine_corners(img,corners,window):
    """
    Input:img: gray scale image
          corners: list of list of corners coordinates
          window: size of neigborhood
    Output: ref_corner: list of refined corners
    """
    # convert corner list to numpy array for cornersubpix
    corner_array=np.zeros((len(corners),1,2),dtype=np.float32)
    for i in range(len(corners)): corner_array[i,0,:]=corners[i]
```

```
    #define criteria for subpix
    criteria=(cv2.TERM_CRITERIA_EPS+cv2.TERM_CRITERIA_MAX_ITER,10,0.01)
    ref_corners=cv2.cornerSubPix(img,corner_array,(window,window),(-1,-1),criteria)
#     ref_corners=corner_array
    ref_corner_list=[]
    for i in range(ref_corners.shape[0]): ref_corner_list.append(ref_corners[i,0,:])
    return(ref_corner_list)

#%%
# function for finding the corners from lines
def findCorners(lines,img_size,pattern_dim):
    """
    Input: lines: hough lines (N,1,2) has rho=lines[i,0,1] and theta=lines[i,0,1]
           img_size: (m,n) tuple gives size of image st
           m=num row n= num col
           pattern_dim: (a,b) tuple: defining the number of boxes pattern
           a=num box along y b= numbox along x
    Output:Corners: list of corner coordinates [[x1,y1],[x2,y2],....]
    """
    m,n=img_size
    # store all angles in a list
    thetas=[lines[i,0,1] for i in range(lines.shape[0])]
    thetas=np.array(thetas) # thetas obtained from hough tr will always be positive
    thetas+=-np.pi/2
    # categorize lines into hor and vert based on thetas
    i_hor=np.where(np.abs(thetas)<np.pi/4)[0]  # hor line if less than 45 deg
    hor_lines=[lines[i,0,:] for i in i_hor]
    i_ver=np.where(np.abs(thetas)>=np.pi/4)[0] # ver line if more than 45 deg
    ver_lines=[lines[i,0,:] for i in i_ver]

    #refine lines
    hor_lines=RefineLines(hor_lines,img_size,2*pattern_dim[0])
    ver_lines=RefineLines(ver_lines,img_size,2*pattern_dim[1])

    # sort hor and ver lines based on the intercept values
    # y_int:hor ~ rho*sin(t), x_int:ver ~ rho*cos(t)
    hor_lines=sorted(hor_lines,key=lambda x:x[0]*np.sin(x[1]))
    ver_lines=sorted(ver_lines,key=lambda x:x[0]*np.cos(x[1]))

    n_hor=len(hor_lines)
    n_ver=len(ver_lines)
    Corners=[]
    # eq of lines is xcos(t)+ysin(t)-rho: using this to make HC
    for i in range(n_hor):
        i_hor_hc=np.array([np.cos(hor_lines[i][1]),np.sin(hor_lines[i][1]),-1*hor_lines[i
            ][0]])
        for j in range(n_ver):
            j_ver_hc=np.array([np.cos(ver_lines[j][1]),np.sin(ver_lines[j][1]),-1*ver_lines[
                j][0]])
            corner=np.cross(i_hor_hc,j_ver_hc)
            corner=corner[:-1]/corner[-1]
            Corners.append(corner)
    return(Corners,hor_lines,ver_lines)
#%%
#function for removing closely resembling lines
def RefineLines(lines,img_size,final_size):
    """
    Refines lines based on intersection of the lines
    Assuming we have only one category of lines ie hor or ver in the set
    Input: lines: list of list of rho and theta values
           img_size: (m,n) tuple serves as a window in this fn
           final_size:
    Output: ref_lines: list of list of rho and theta of refined lines
    """
    rho=np.array([lines[i][0] for i in range(len(lines))])
```

```
    theta=np.array([lines[i][1] for i in range(len(lines))])
    HC_lines= np.ones((len(lines),3))
    # eq of lines is xcos(t)+ysin(t)-rho: using this to make HC
    HC_lines[:,0]=np.cos(theta)
    HC_lines[:,1]=np.sin(theta)
    HC_lines[:,-1]=-1*rho
    # loop to find intersections within set
    count=np.zeros(len(lines),dtype=int)
    for i in range(len(lines)):
        for j in range(i+1,len(lines)):
            pt=np.cross(HC_lines[i,:],HC_lines[j,:])
            if pt[-1]!=0:
                # when not parallel
                pt=pt[:-1]/pt[-1]
                if pt[0]>=0 and pt[0]<img_size[0] and pt[1]>=0 and pt[1]<img_size[1]:
                    count[j]=1
            else:
                # when lines are parallel
                rho1=lines[i][0];  rho2=lines[j][0]
                if abs((rho1-rho2)/rho1)<10**-2:
                    count[j]=1
    ind=np.where(count==0)[0]
    if len(ind)>final_size:
        #---------------kmeans--------------------#
        """
         now we need to remove the false positive lines
         which would have interesected in the image region
         but were just a little bit shy: we can do this using kmeans
         kmeans on rho values of lines
         """
        temp_lines=[]
        for i in range(len(ind)): temp_lines.append(lines[ind[i]])
        N=len(temp_lines)
        observations=np.array(temp_lines)
        observations=observations[:,0]# only rho
        centroids=kmeans(observations,final_size) # returns a tuple
        # find rho values in lines closest to centroids
        store=np.zeros(final_size,dtype=int)
        for i in range(final_size):
            temp=np.abs(observations-centroids[0][i]*np.ones(N))
            store[i]=np.argmin(temp)
        ref_lines=[]
        for i in range(final_size): ref_lines.append(temp_lines[store[i]]) # only single
            values
        #---------------kmeans--------------------#
    else:
        ref_lines=[]
        for i in range(final_size): ref_lines.append(lines[ind[i]])
    return(ref_lines)
#%%
# function for drawing Line on the calibration pattern
def drawLine(color_img,lines):
    """
    Input: color_img: colored image mxnx3
           lines: list of arrays with [rho,theta] values
           slices 1,2 have values of rho and theta
    Output: draw_img: m,n,3 image with line drawn
    using lines
    """
    # find size of image
    m,n=color_img.shape[:-1]
    #find maximum edge length that can fit in image
    L=np.sqrt(m**2+n**2)
    # make copy of color_img
    draw_img=np.copy(color_img)
```

```
    # for every line
    for i in range(len(lines)):
        # find rho and theta
        rho,theta=lines[i]
        a=np.cos(theta);  b=np.sin(theta);
        x0=rho*a; y0= rho*b; # perpendicular point
        # find pts on line which will lie outside
        # image bounds or on borders
        x1=int(x0+L*(-b))
        y1=int(y0+L*(a))
        x2=int(x0-L*(-b))
        y2=int(y0-L*(a))
        # draw line joining the two points
        cv2.line(draw_img,(x1,y1),(x2,y2),(0,255,0),1)
    return(draw_img)
"""
V0: was used to draw raw hough lines
"""
# function for drawing Line on the calibration pattern
def drawLine_old(color_img,lines):
    """
    Input: color_img: colored image mxnx3
           lines: size N,1,2 N is the number of lines
           slices 1,2 have values of rho and theta
    Output: draw_img: m,n,3 image with line drawn
    using lines
    """
    # find size of image
    m,n=color_img.shape[:-1]
    #find maximum edge length that can fit in image
    L=np.sqrt(m**2+n**2)
    # make copy of color_img
    draw_img=np.copy(color_img)
    # for every line
    for i in range(lines.shape[0]):
        # find rho and theta
        rho,theta=lines[i,0,:]
        a=np.cos(theta);  b=np.sin(theta);
        x0=rho*a; y0= rho*b; # perpendicular point
        # find pts on line which will lie outside
        # image bounds or on borders
        x1=int(x0+L*(-b))
        y1=int(y0+L*(a))
        x2=int(x0-L*(-b))
        y2=int(y0-L*(a))
        # draw line joining the two points
        cv2.line(draw_img,(x1,y1),(x2,y2),(0,255,0),1)
    return(draw_img)
#%%
#function for finding corners
def findLines(img,minVal,maxVal,HoughThresh):
    """
    Input: img: mxn gray scale image
           minVal,maxVal: hysterisis thresholds for canny operator
           HoughThresh: threshoold for hough transform for finding the peak
    Output: edges: binary image with only canny edges
            lines: (rho,theta) format
    """
    # find canny edges for the image
    edges=cv2.Canny(img,minVal,maxVal)
    # find lines using hough transform
    lines = cv2.HoughLines(edges,1,np.pi/180,HoughThresh)
    return(edges,lines)
#=============================HW8=============================#
#%%
```

```
#=============================HW2=============================#
# Homograhpy using points
def HMat_Pts(world,img):
    """
    Returns a homography mat such that H*world=img and H[2,2]=1 ie push forward
    img is on destination Image
    world is on source Image
    img,world=np.array([[x1,y1],[x2,y2],[...],[...]])
    """
    n=np.shape(world)[0]
    m=np.shape(world)[1]
    A=np.zeros((2*n,9))

    block1=np.zeros((n,m+1))
    block1[:,-1]=np.ones((1,n))
    block1[:,:2]=world[:,:]
    A[:n,:3]=block1
    A[n:2*n,3:6]=block1

    block2=np.multiply(world.T,-1*img[:,0])
    block3=np.multiply(world.T,-1*img[:,1])

    A[:n,6:8]=block2.T
    A[n:2*n,6:8]=block3.T
    A[:,-1]=-1*np.reshape(img.T,n*m)
    # solve using null space
    u,d,vt=np.linalg.svd(A)
    h=vt[-1,:]
    H=np.reshape(h,(3,3))
    H=H/H[-1,-1]
    return(H)
#=============================HW2=============================#
```