ECE 661 Computer Vision (2018 Fall)
Homework 8

Runzhe Zhang

November 8, 2018

1. Introduction

The goal in this homework is to implement the popular Zhang's algorithm for camera calibration. A format description of the algorithm can be found in the Zhang's technical report.

For this assignment, we can assume the camera to be a pin-hole camera. This implies that a complete calibration procedure will involve estimating all the 5 intrinsic parameters and the 6 extrinsic parameters that determine the position and orientation of the camera with respect to a reference world coordinate system. This requires us to establish correspondences between image points and their world coordinates. To this end, we can download and use the checker-board pattern from. The checker-board pattern consists of alternating black and white squares. We will be using the corners of these squares in our calibration procedure.

The procedure of the camera calibration(reference: Homework 8):

1. **Creating the dataset**
   (a) Print out the calibration pattern that is provided at [I] and mount it on a wall or a large piece of cardboard. Now choose one of the corners on the pattern as your world origin and measure the world coordinates of all the other corner points on the pattern with a ruler. Number the corners appropriately. A particular corner should get the same number label in all the images. (This does not mean that you need to write any numbers on the actual calibration pattern).
   (b) For the very first image of the calibration pattern on the wall, position the camera such that its Principal Axis is approxi- mately perpendicular to the plane of the wall on which you mounted the calibration pattern. Also make sure that the x-axis of the image is very roughly along the horizontal axis of the calibration pattern and the y-axis of the image very roughly along the vertical axis of the pattern. These conditions are meant to be satisfied only very, very approximately. Despite the approximations involved, the distance between the camera and calibration pattern that you can measure manually will serve as a check on your calculations of the calibration parameters. In the following discussion, this image will be referred to as 'Fixed Image'.
   (c) Now move your camera in different directions and capture images of the calibration pattern. Obviously you will need to rotate/tilt the camera in order to capture the calibration pattern from different positions. A minimum of 20 different poses of the camera is required for good camera calibration.
2. **Zhang's Algorithm**
   (a) Detecting corners
      i. Extract edges using the Canny edge detector. You can use any open-source implementation of Canny edge detector like cvCanny function from OpenCV.

ii. Fit straight lines to the edges using the Hough transform. You can again use any open-source implementation of Hough lines transform like the cvHoughLines or the more efficient HoughLinesP functions from OpenCV.

iii. The corners will be the intersection points of these lines.

iv. Depending on the accuracy of your corner detection, you might wish to improve your results. You can refer to the previous year solutions for improving this accuracy. However do note that it is not necessary to detect 100% of the true corners in every image. Since you will be using the Levenberg Marquadt non-linear optimization to refine your calibration, you should have robust calibration as long as you detect sufficiently high number of corners with good accuracy in each image.

v. Assign labels to the corners using the same numbering scheme that you used in the previous section. These labels should be indicated on every output image in your report.

(b) Calibration

i. You need to establish correspondences between the extracted corners in each image and their world coordinates.

ii. Implement Zhang's calibration algorithm.

iii. Use the Levenberg Marquadt algorithm for non-linear optimization. The Levmar package [II] is a very good resource for this. It can be used with C++, Matlab and Python.

iv. To measure the accuracy of your camera-calibration, reproject the corner points from 4 or more views back into the 'Fixed Image'. You can do a visual comparison of the locations of the original corners vis-a-vis the reprojected corner points. In each of the (4 or more) images, measure the error for each point using the Euclidean Distance measure. Calculate an estimate of the mean and variance of this error.

v. Show a minimum of 2 images where one can see the improvement of your calibration estimate by using the LM optimization.

vi. Compare your estimated camera pose for the 'Fixed Image' with the measured ground-truth.

2. Corner Detection

1. **Canny Edge Detection**
   From the given image, we firstly transfer it into grayscale image. Then blur it with a Gaussian kernel. Aftwewards we apply the Canny edge detector to the blurred image. The parameters are chosen empirically. The function call is like: cv2.Canny(blur, 2500, 5000, apertureSize=5).

2. **Hough Transform**
   From the edges detected by Canny, we use the Hough Transform to fit straight lines to the detected edges. The parameters are chosen empirically. Notice that the last parameter (50) indicates how perfect the line needs to be to be detected. We set a relatively small value to make sure that we don't miss any lines. The function call is like: cv2.HoughLines(edge,1,np.pi/180,50)

3. **False Line Filter**
   One problem with Hough transform is that one actual line might be detected multiple times. From the result we can clearly find out that they are very close to each other. This problem is partially due to the radial distortion. To filter these false lines out and make sure that we only have one detected line for each true line, we apply some engineering tricks here. Firstly we need to sort the detected lines. For all the detected lines, we firstly group them into horizontal groups and vertical groups. For all the lines in each group, we sort them using a specified order. For horizontal lines we sort them from upside down. For vertical lines we sort them from left side to the right. Then, for successive lines, we calculate the distance between them. If the distance is less than a given threshold (chosen empirically), we only select the line near the upper left. After this technique, we will only have one line detection for each true line.

4. **Localize the Corners**
   After we have found all the lines in the horizontal group and the vertical group, we find the intersection between each pair of lines from two groups. The intersections are the initial position of the corners that we are trying to localize. Due to radial distortion, the positions of these corners are not perfectly accurate, thus we apply a SubPixel refinement algorithm to the detected initial corner position. After the refinement the positions of all the corners are very accurate, and will be used later for our camera calibration algorithm. For better visualization, we put index of each corner on the image.

3. Camera Calibrationr

To calculate the camera parameters, we need to establish the correspondence between the image pixel or the corner and the true location in the 3D model.

3.1 Intrinsic Parameter K Estimation

After we have built the correspondence between the corners on the model in 3D space with the 2D pixel location, extracted using the procedure described in the last section, we estimate the Homography between them. The intrinsic parameters, K, can be soloved using the image of the absolute conic method. We can write the pixel coordinate in homogeneous:

$$\vec{x} = [R|\vec{t}] \begin{bmatrix} x \\ y \\ 0 \\ w \end{bmatrix} = H\vec{x_w}$$

The image of the Absolute Conic is given by

$$\omega = K^{-T}K^{-1}$$

We denote the H, and both cicular points on ω, so we get the two equations:

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

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

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

We can convert equations to the form

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \end{bmatrix} \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{bmatrix} \begin{bmatrix} h_{21} \\ h_{22} \\ h_{23} \end{bmatrix} = 0, \vec{v_{12}}^T \vec{b} = 0$$

To estimate homography, we basically use the SVD of the established correspondence. After we have achieved H, we construct the V_{ij} using the formula provided by Zhang. Use at least 3 camera positions and stack them in a matrix (V is a 2*6 matrix) to solve using linear least square minimization for $\vec{b}$.

$$v = \begin{bmatrix} \vec{v_{12}}^T \\ \vec{v_{11}}^T - \vec{v_{22}}^T \end{bmatrix}$$

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

3.2 extrinsic Parameter R,T Estimation

Now we need to estimate the extrinsic parameters, they are basically the rotation and translation matrix between the camera center and the world origin. Now, to calculate the extrinsic parameters R and $\vec{t}$ of the camera for one of its positions vis-a-vis the $Z = 0$ plane that holds the calibration pattern. The homography H is given

$$H = \begin{bmatrix} \vec{h}_1 & \vec{h}_2 & \vec{h}_3 \end{bmatrix}$$

The parameters can be got by

$$k^{-1} \begin{bmatrix} \vec{h}_1 & \vec{h}_2 & \vec{h}_3 \end{bmatrix} = \begin{bmatrix} \vec{r}_1 & \vec{r}_2 & \vec{r}_3 \end{bmatrix}$$

$$\vec{r}_1 = k^{-1}\vec{h}_1$$
$$\vec{r}_2 = k^{-1}\vec{h}_2$$
$$\vec{r}_3 = \vec{r}_1 X \vec{r}_2$$
$$\vec{t} = k^{-1}\vec{h}_3$$

To make rows and columns of R orthonormal we can obtain R by doing SVD and $SVD(R) = UDV^T$ and making $R = UV^T$.

Following these formulas we construct a rotation and translation matrix for each image, these are the extrinsic parameters of a camera.

4. REFINING THE CALIBRATION PARAMETERS

Notice that the parameters that we estimate are based on linear optimization. However, this problem has a lot of non-linear nature, thus we need to use some non-linear optimization algorithm to refine the intrinsic and extrinsic parameters that we have estimated. To that end, we need to construct a residual function. Here we define the residual function as the sum of Euclidean distance between the ground truth for each corner and the projected coordinate for all the corners in all the images, using the parameters we estimated before.
Notation:
$\vec{x}_{m,j}$: The jth salient point on the calibration pattern.
$\vec{x}_{i,j}$: The actual image point for $\vec{x}_{m,j}$ in the ith position of the camera.
$\hat{\vec{x}}_{i,j}$: The projected image point for $\vec{x}_{m,j}$ using P for ith camera position. R_i: The rotation matrix for the ith position of the camera.
$\vec{t}_i$: The translation vector for ith position of the camera.
K: The camera calibration matrix for the intrinsic parameters.

If we could assume that the calculated camera calibration parameters have zero error, we'll have $\vec{x}_{i,j} = \hat{\vec{x}}_{i,j}$. In general, the Euclidean distance $||\vec{x}_{i,j} - \hat{\vec{x}}_{i,j}||$ tells us something about how far the calibration is with respect to the jth salient point in the ith position of the camera. Aggregating the error distances for all salient points and for all the camera positions, we have $d^2_{geom} = ||\vec{X} - f(\vec{p})||^2$. Use the Rodrigues Representation to represent 3-parameter representation of a rotation matrix R.

$$\omega = \frac{\varphi}{2sin\varphi}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix}, \varphi = cos^{-1}\frac{trace(R) - 1}{2}.$$

To transform back from Rodriguez, we use

$$R = I + \frac{sin\varphi}{\varphi}[W]_x + \frac{1 - cos\varphi}{\varphi}[W]^2_x, W_x = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & -\omega_x & 0 \end{bmatrix}.$$

We throw these residual functions into the non- linear optimization solver for the parameter refinement. Here we use the optimize package from SciPy.

5. Rodrigues Representation

Each rotation matrix contains 9 elements. However, it only has 3 DOF. If we directly use the rotation matrix elements as the parameter in the non-linear optimization method, it's very likely that we will get weird results, thus we use the Rodrigues representation. It is a succinct representation of a 3x3 rotation matrix in a 3x1 vector form. We use those parameters to feed the non-linear optimization algorithm.

6. Condition the Rotation Matrix

Before converting each R_i into a Rodriguz vector $\vec{\omega}_i$, we must condition the R_i to make it orthonormal. To show how a rotation matrix can be conditioned, let Q represent the computed rotation matrix. Our goal is to find the best orthonormal approximation R to Q. We have to minimize $||R-Q||_F^2$ subject to $R^TR = I$. The solution to the problem is found by carrying out SVD. $[U,D,V] = SVD(Q)$ and we set R equal to UV^T.

7. Incorparate the Radial Distoration

The linear model works well for long focal length cameras. The pinhole model breaks down for short focal-length cameras. Let $(\hat{x},\hat{y})$ be the predicted position of a pixel using the pinwhole model and the tentative set of calibration parameters. And let $(\hat{x}_{rad},\hat{y}_{rad})$ be the pixel coordinates that would be predicted with the radial distortion.

$$\hat{x}_{rad} = \hat{x} + (\hat{x} - x_0)[k_1r^2 + k_2r^4]$$

$$\hat{y}_{rad} = \hat{y} + (\hat{y} - y_0)[k_1r^2 + k_2r^4]$$

$$r^2 = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2$$

We can invoke the nonlinear least-squares part of the calibration procedure to refine $K, R, \vec{t}$. Where (x_0, y_0) are the image center and r is the distance between the point to the image center. According to Tommy's report, we need to firstly back-project the image points onto the physical plane, apply this radial distortion, then bring it back to the image plane.

8. The Extra Credit

1. The radial distortion was incorporated in my LM optimization algorithm;
2. I accumulate the Euclidean distance on all the corners of all the images, and then calculate the mean, standard deviation and maximum value of them. Clearly after the LM algorithm, all the three metrics dropped significantly, thus proving the effectiveness of LM algorithm.
3. I experimented with where to condition the rotation matrix, not a lot of significant difference came to me. If I also add the rotation condition in the non- linear optimization algorithm, the average re-projection error only decreases from 0.372586046767 to 0.372585946477, which is totally ignorable.

9. Result

9.1 Dataset 1

9.1.1 Two Dataset1 Samples for Corner Detection — Sample 1

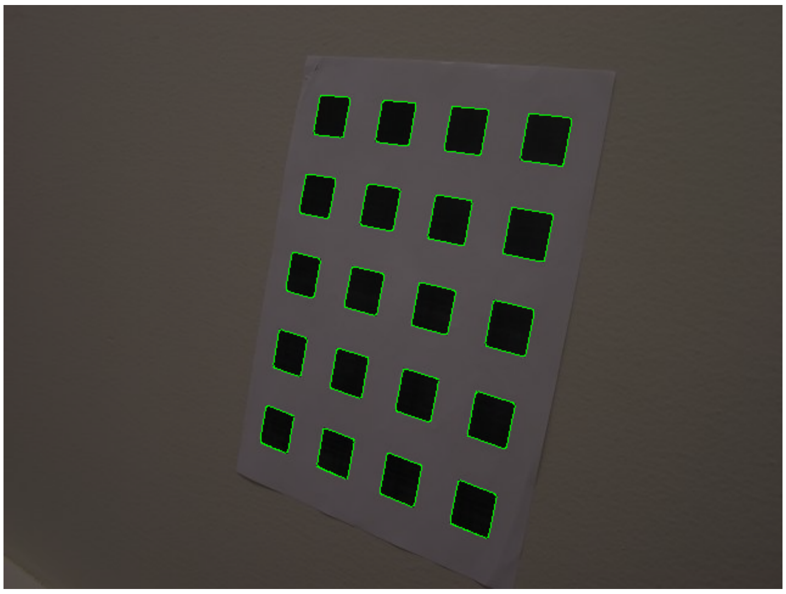

Figure 1: Dataset1 Sample 1 Canny edge detector result

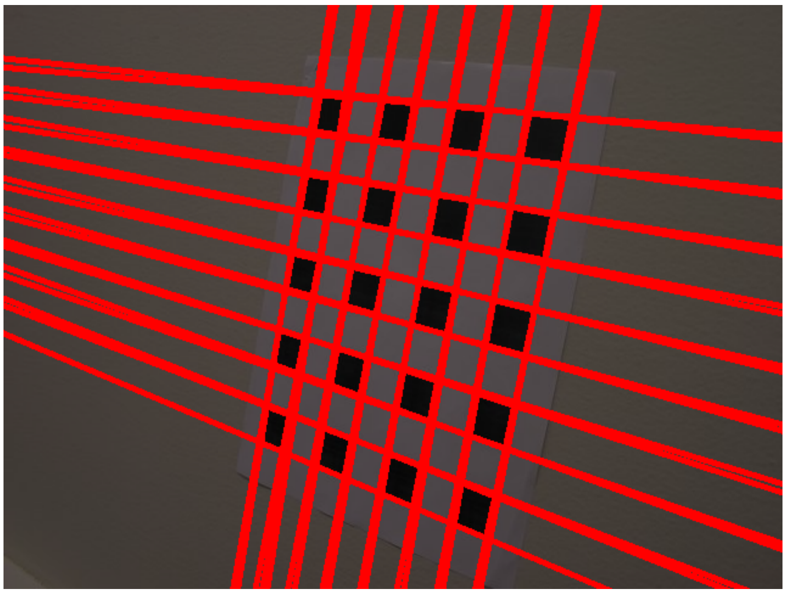

Figure 2: Dataset1 Sample 1 Hough Line transform result

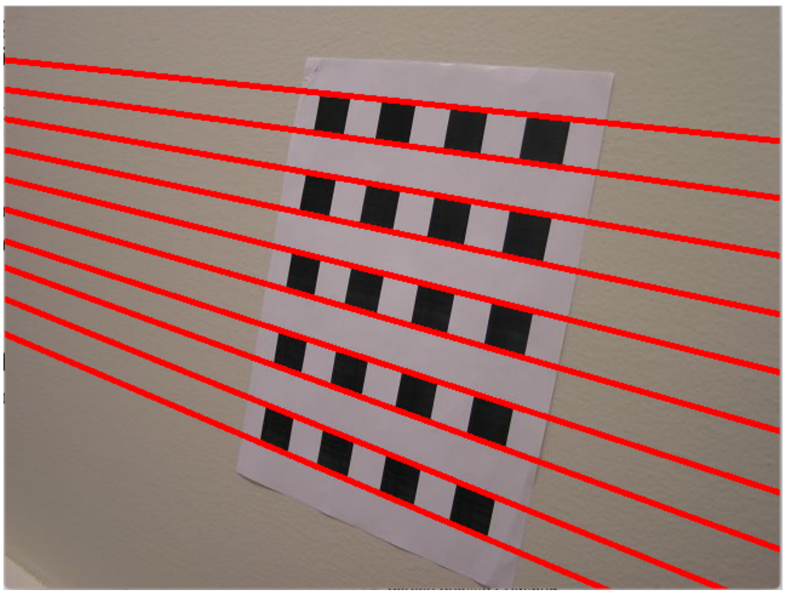
Figure 3: Selected horizontal line group

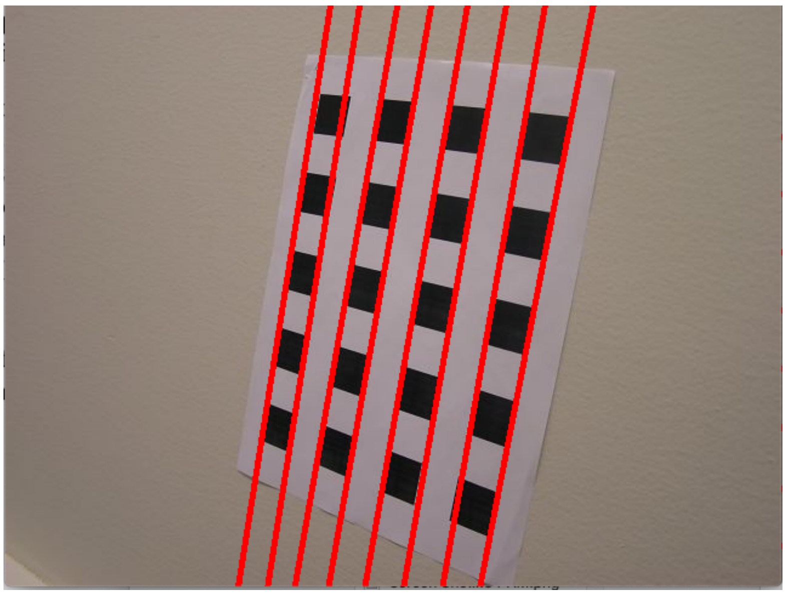
Figure 4: Selected vertical line group

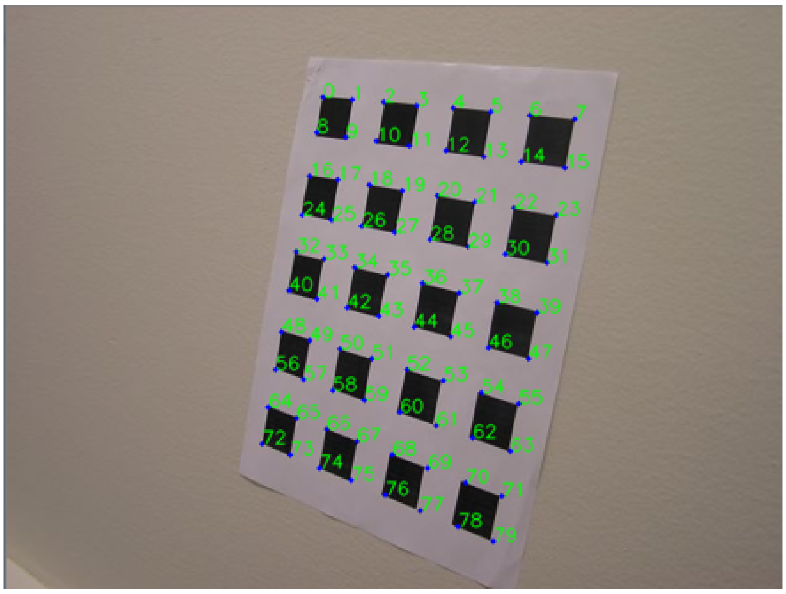

Figure 5: Dataset1 Sample 1: Initial corner position as the intersection between horizontal and vertical line groups, with labels indicating corner index

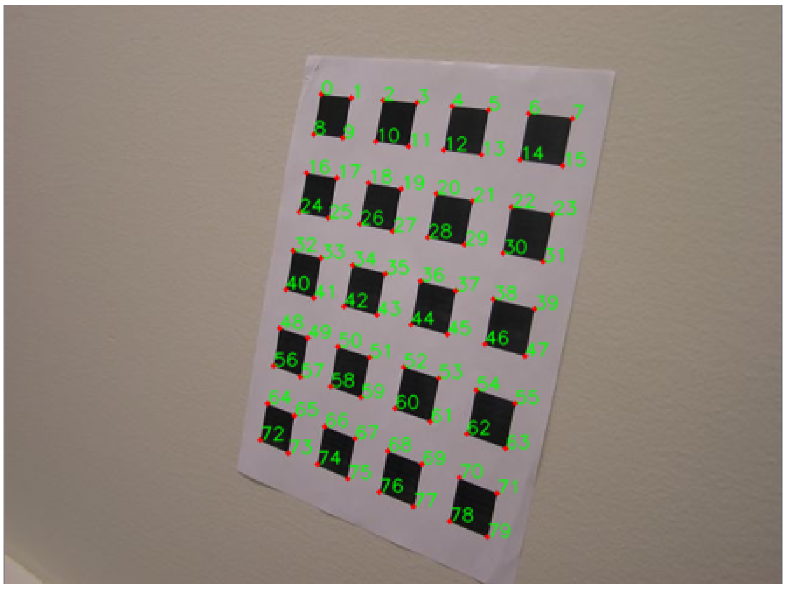

Figure 6: Dataset1 Sample 1: Refined corner position after subpixel optimization, we can clearly see an improvement on the localization of the corners

9.1.2 Two Dataset1 Samples for Corner Detection — Sample 2

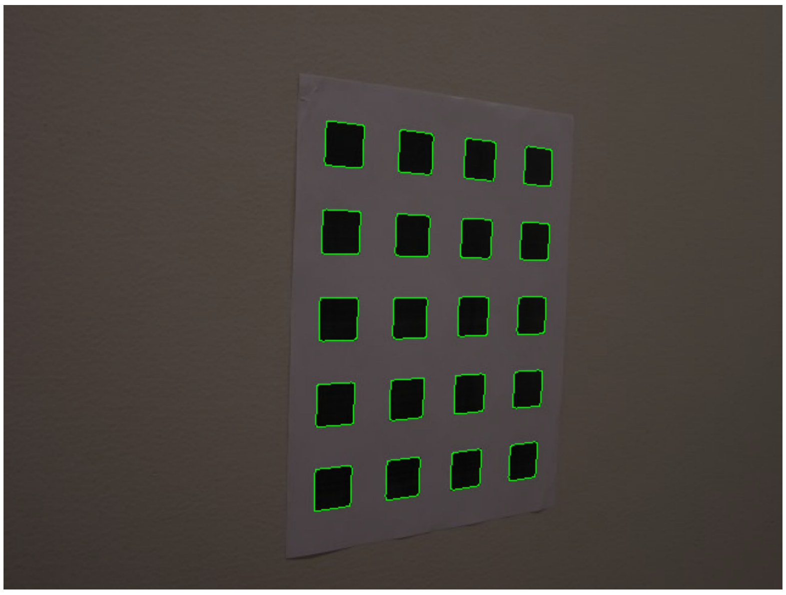

Figure 7: Dataset1 Sample 2 Canny edge detector result

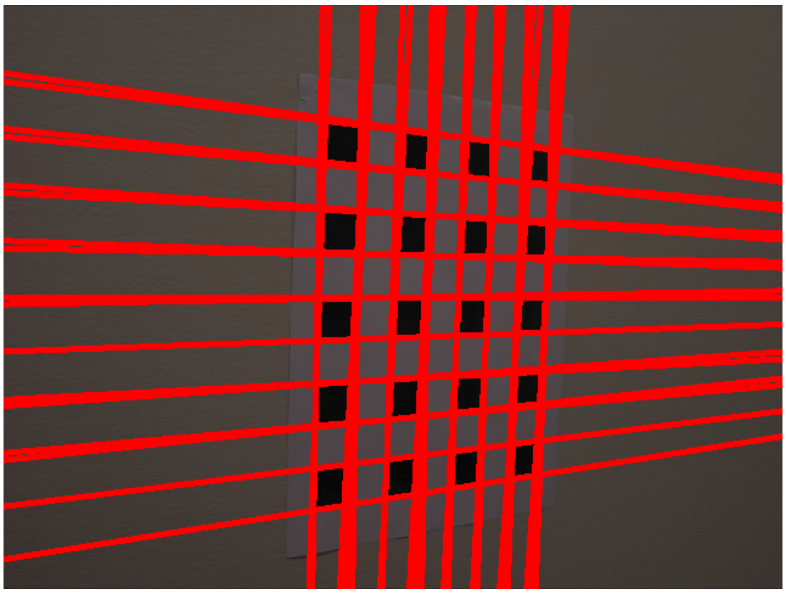

Figure 8: Dataset1 Sample 2 Hough Line transform result

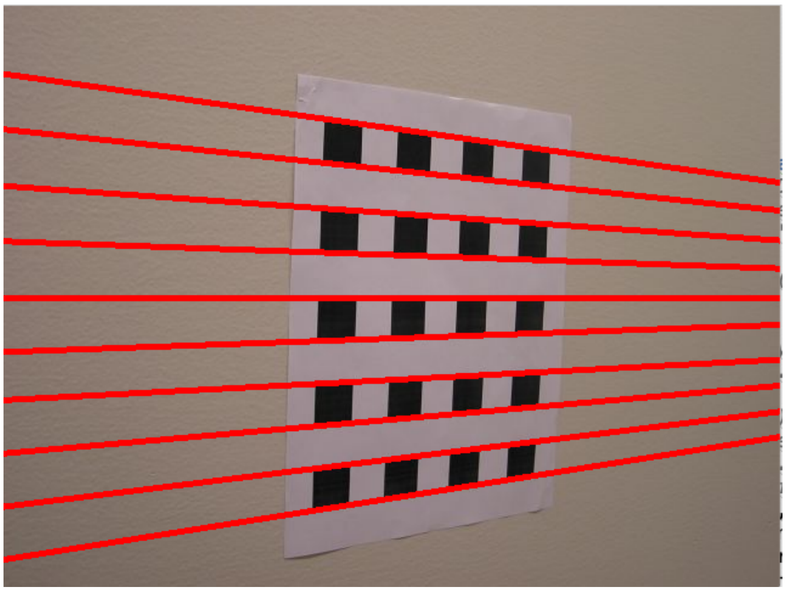

Figure 9: Dataset1 Sample 2 Selected horizontal line group

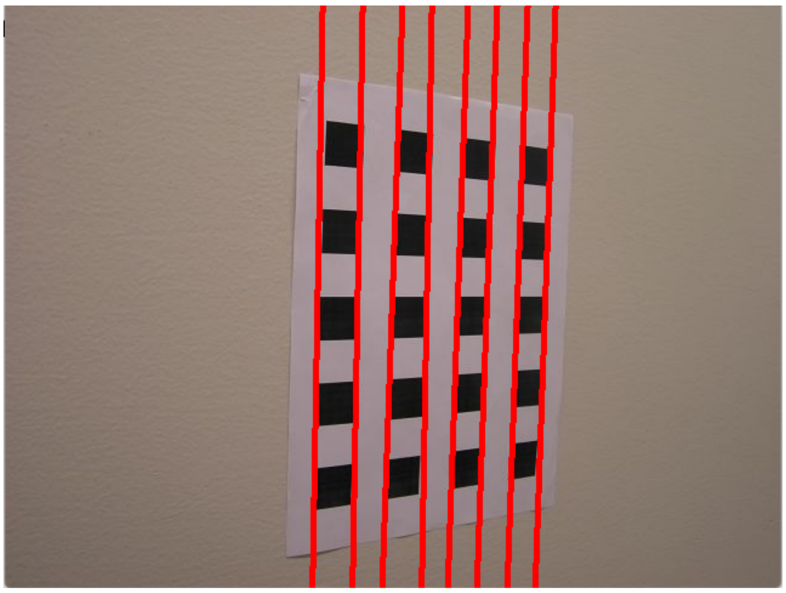

Figure 10: Dataset1 Sample 2 Selected vertical line group

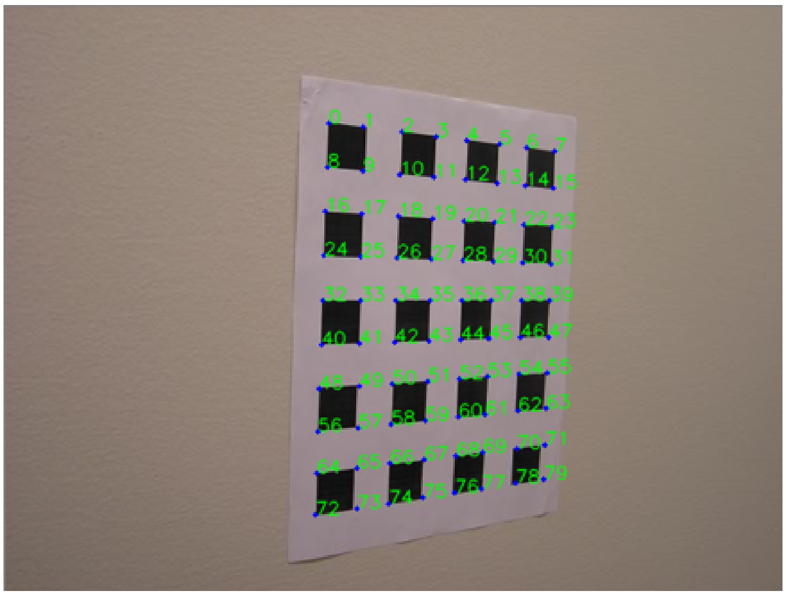
Figure 11: Dataset1 Sample 2: Initial corner position as the intersection between horizontal and vertical line groups, with labels indicating corner index

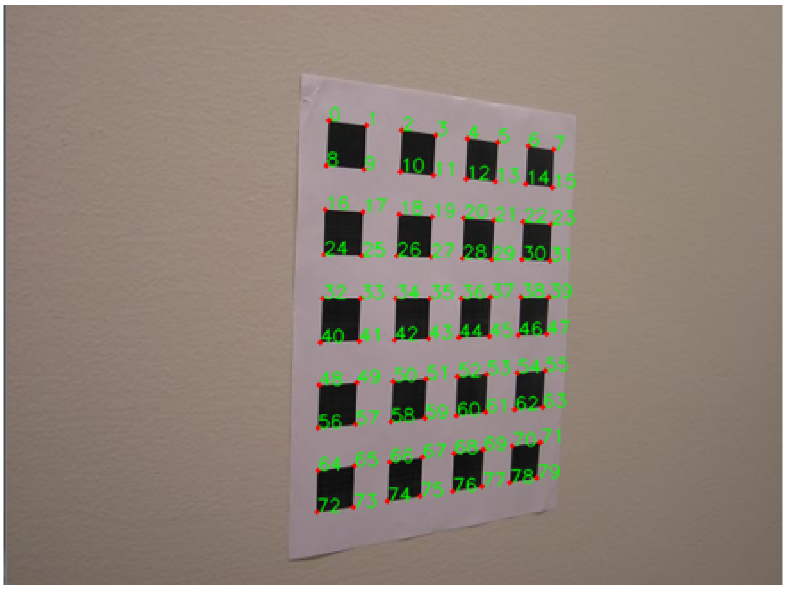
Figure 12: Dataset1 Sample 2: Refined corner position after subpixel optimization, we can clearly see an improvement on the localization of the corners

9.1.3 Two Dataset 1 Samples for Corner Re-projection — Calculation

Below are the results calculated based on the first 20 images in the provided image set, after we get the intrinsic and extrinsic parameters, we project the model points onto the image plane and calculate the Euclidean distance between them as a measure of our calibration algorithm.

Before we apply the LM non-linear optimization for parameter refinement:

1. **The parameter K:**

$$K_{before\ LM} = \begin{bmatrix} 715.153 & -1.095 & 206.188 \\ 0 & 713.739 & 340.444 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Mean re-projection error on all images = 0.773
3. Standard deviation of re-projection error on all images = 0.671
4. Maximum re-projection error on all corners = 5.31

After we apply the LM non-linear optimization for parameter refinement:

1. **The parameter K:**

$$K_{before\ LM} = \begin{bmatrix} 465.03 & -0.349 & 241.47 \\ 0 & 464.01 & 179.01 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Radial distortion coefficient: $k_1 = 0.003755, k_2 = -0.0358$
3. Mean re-projection error on all images = 0.372
4. Standard deviation of re-projection error on all images = 0.230
5. Maximum re-projection error on all corners = 1.523

9.1.4 Two Dataset 1 Samples for Corner Re-projection — Sample 1

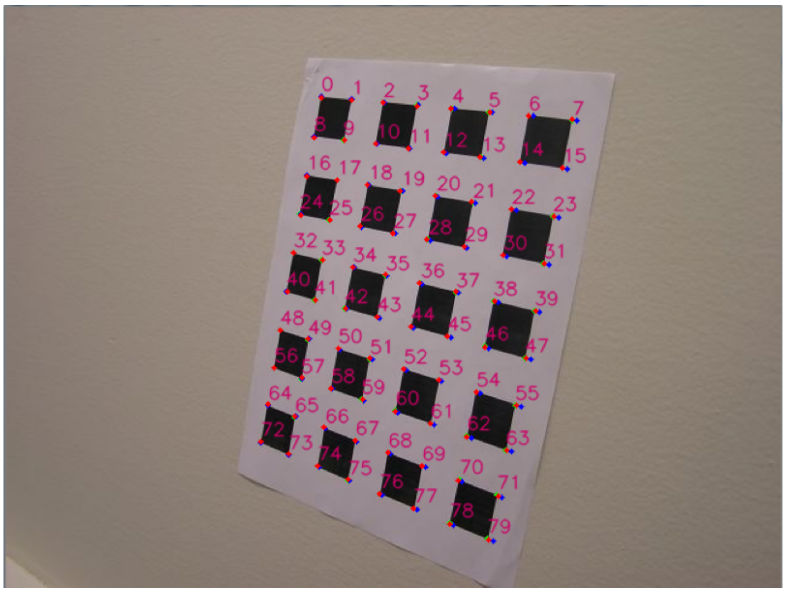

Figure 13: Dataset1 Sample 1: first example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\vec{t}\,] = \begin{bmatrix} 0.984 & 0.162 & 0.074 & -139.616 \\ -0.175 & 0.807 & 0.564 & -88.582 \\ 0.032 & -0.568 & 0.822 & 752.228 \end{bmatrix}$$

9.1.5 Two Dataset 1 Samples for Corner Re-projection — Sample 2

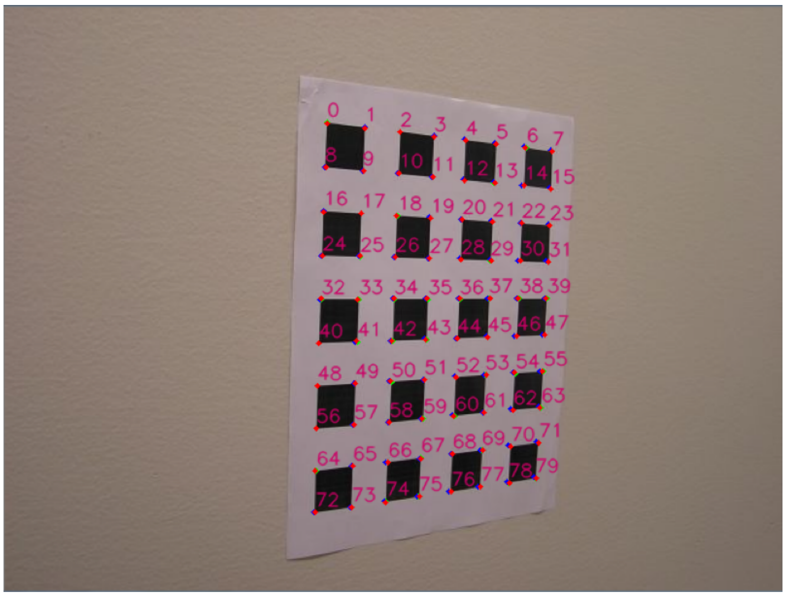

Figure 14: Dataset1 Sample 2: second example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\overrightarrow{t}] = \begin{bmatrix} 0.999 & 0.012 & -0.042 & -105.799 \\ -0.033 & 0.816 & -0.577 & -74.702 \\ 0.027 & 0.578 & 0.516 & 676.098 \end{bmatrix}$$

9.1.6 Two Dataset 1 Samples for Corner Re-projection — Sample 3

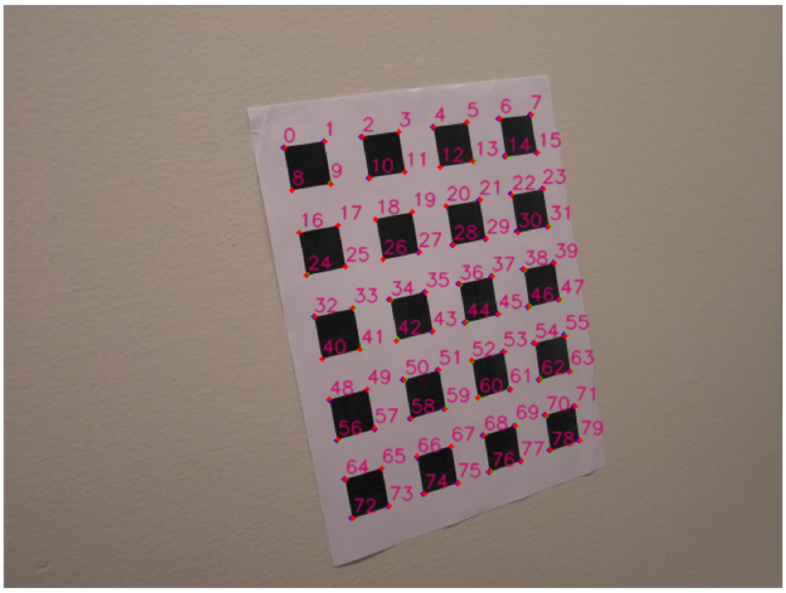

Figure 15: Dataset1 Sample 3: third example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\vec{t}\,] = \begin{bmatrix} 0.982 & -0.191 & 0.012 & -87.358 \\ 0.174 & 0.866 & -0.468 & -110.098 \\ 0.079 & 0.462 & 0.884 & 685.518 \end{bmatrix}$$

9.1.7 Two Dataset 1 Samples for Corner Re-projection — Sample 4

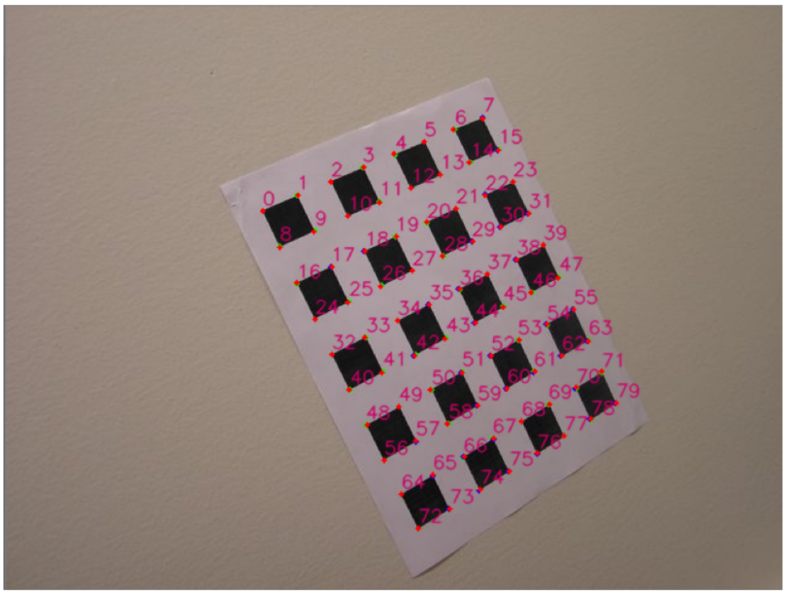

Figure 16: Dataset1 Sample 4: fourth example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\overrightarrow{t}] = \begin{bmatrix} 0.897 & -0.407 & 0.171 & -40.371 \\ 0.441 & 0.811 & -0.386 & -139.407 \\ 0.018 & 0.421 & 0.907 & 746.816 \end{bmatrix}$$

9.2 Dataset 2

9.2.1 Two Dataset1 Samples for Corner Detection — Sample 1

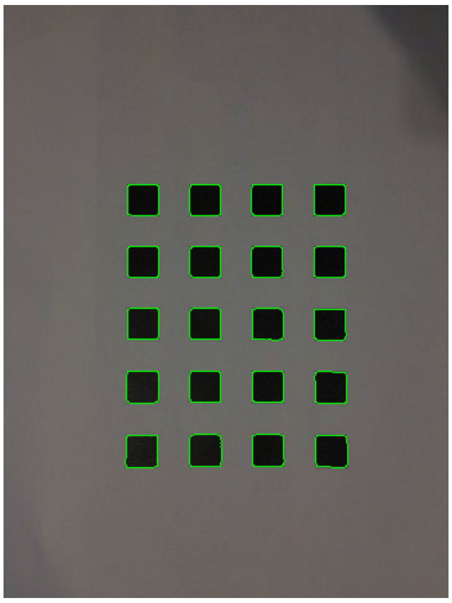

Figure 17: Dataset 2 Sample 1 Canny edge detector result

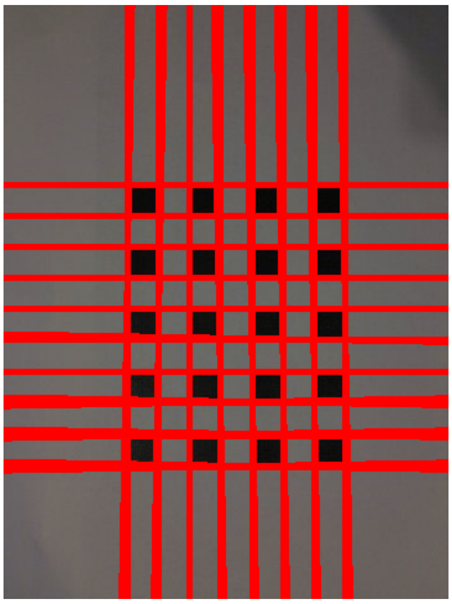

Figure 18: Dataset 2 Sample 1 Hough Line transform result

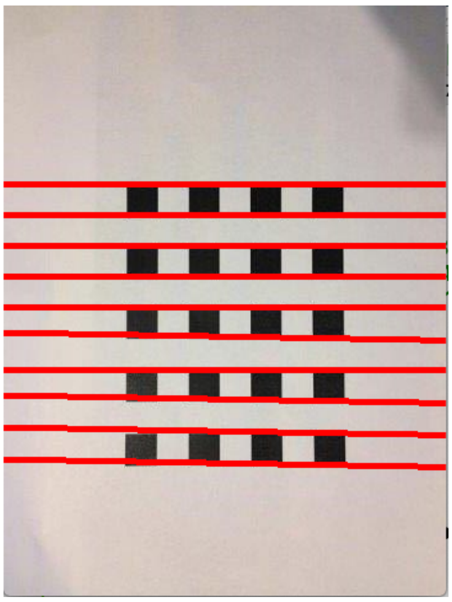
Figure 19: Dataset 2 Sample 1 Selected horizontal line group

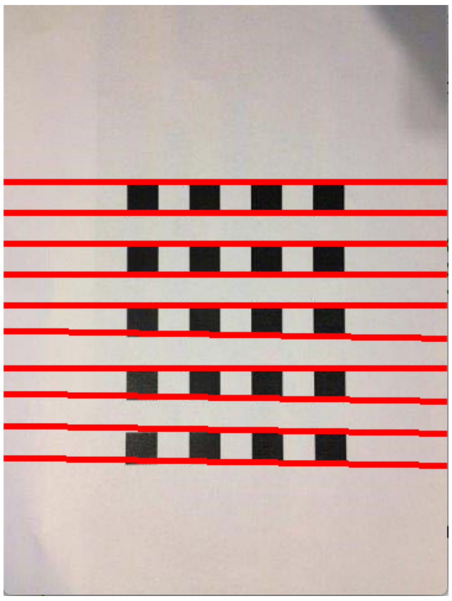
Figure 20: Dataset 2 Sample 1 Selected vertical line group

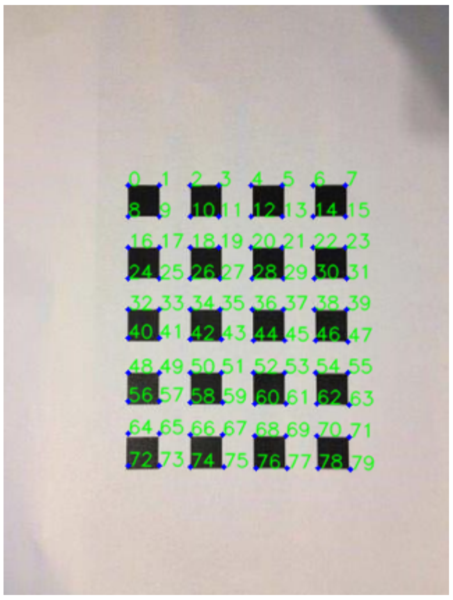

Figure 21: Dataset 2 Sample 1: Initial corner position as the intersection between horizontal and vertical line groups, with labels indicating corner index

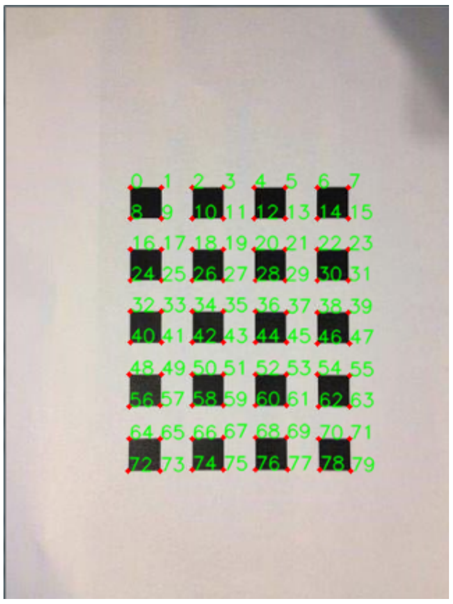

Figure 22: Dataset 2 Sample 1: Refined corner position after subpixel optimization, we can clearly see an improvement on the localization of the corners

9.2.2 Two Dataset 2 Samples for Corner Detection — Sample 2

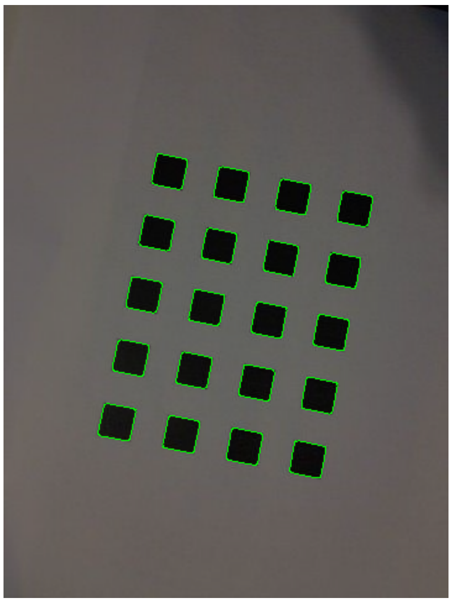
Figure 23: Dataset 2 Sample 2 Canny edge detector result

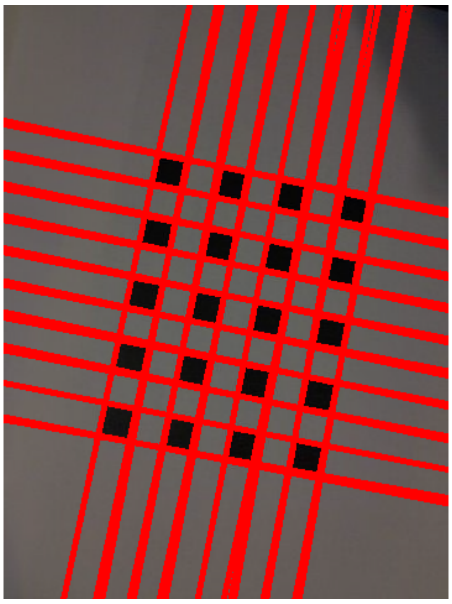
Figure 24: Dataset 2 Sample 2 Hough Line transform result

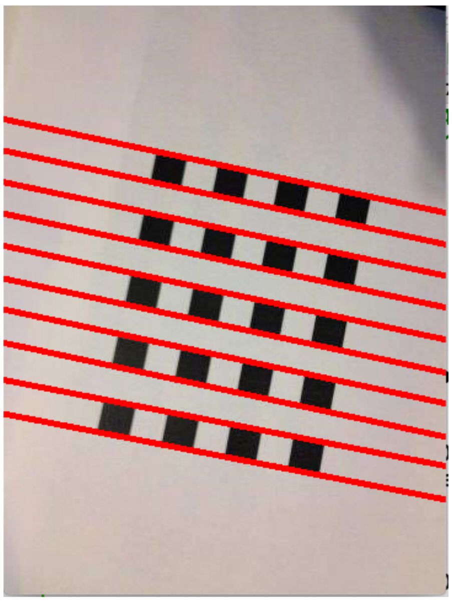
Figure 25: Dataset 2 Sample 2 Selected horizontal line group

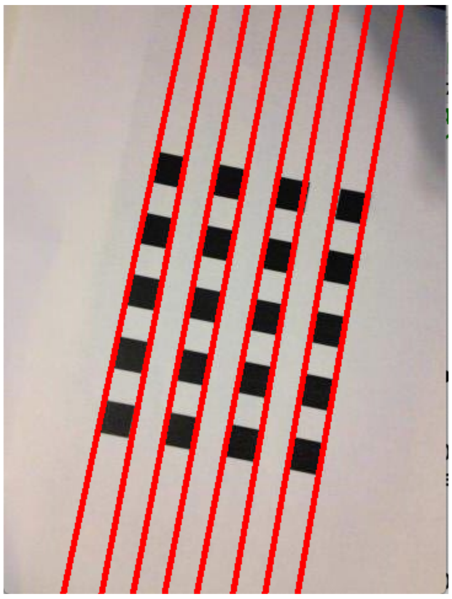
Figure 26: Dataset 2 Sample 2 Selected vertical line group

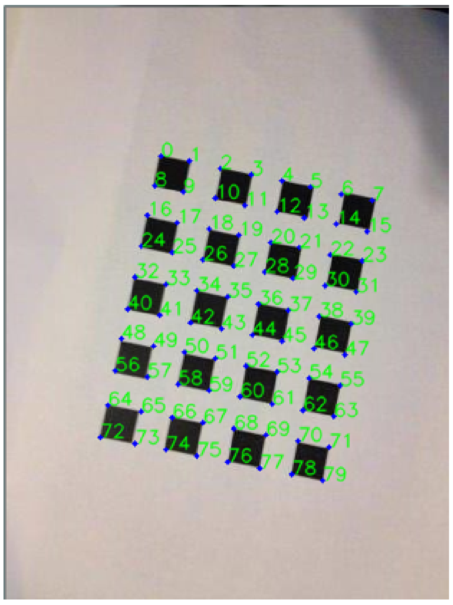

Figure 27: Dataset 2 Sample 2: Initial corner position as the intersection between horizontal and vertical line groups, with labels indicating corner index

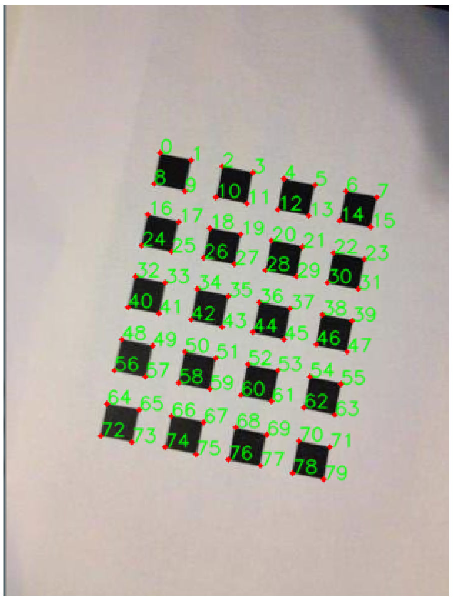

Figure 28: Dataset 2 Sample 2: Refined corner position after subpixel optimization, we can clearly see an improvement on the localization of the corners

9.2.3 Two Dataset 2 Samples for Corner Re-projection — Calculation

Below are the results calculated based on the first 20 images in the provided image set, after we get the intrinsic and extrinsic parameters, we project the model points onto the image plane and calculate the Euclidean distance between them as a measure of our calibration algorithm.

Before we apply the LM non-linear optimization for parameter refinement:

1. **The parameter K:**

$$K_{before\ LM} = \begin{bmatrix} 466.76 & 1.13 & 238.23 \\ 0 & 466.15 & 178.86 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Mean re-projection error on all images = 0.6046
3. Standard deviation of re-projection error on all images = 0.3816
4. Maximum re-projection error on all corners = 2.5918

After we apply the LM non-linear optimization for parameter refinement:

1. **The parameter K:**

$$K_{before\ LM} = \begin{bmatrix} 465.03 & -0.349 & 241.47 \\ 0 & 464.01 & 179.01 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Radial distortion coefficient: $k_1 = 0.006818, k_2 = 0.069678$
3. Mean re-projection error on all images = 0.372
4. Standard deviation of re-projection error on all images = 0.230
5. Maximum re-projection error on all corners = 1.523

9.2.4 Two Dataset 2 Samples for Corner Re-projection — Sample 1

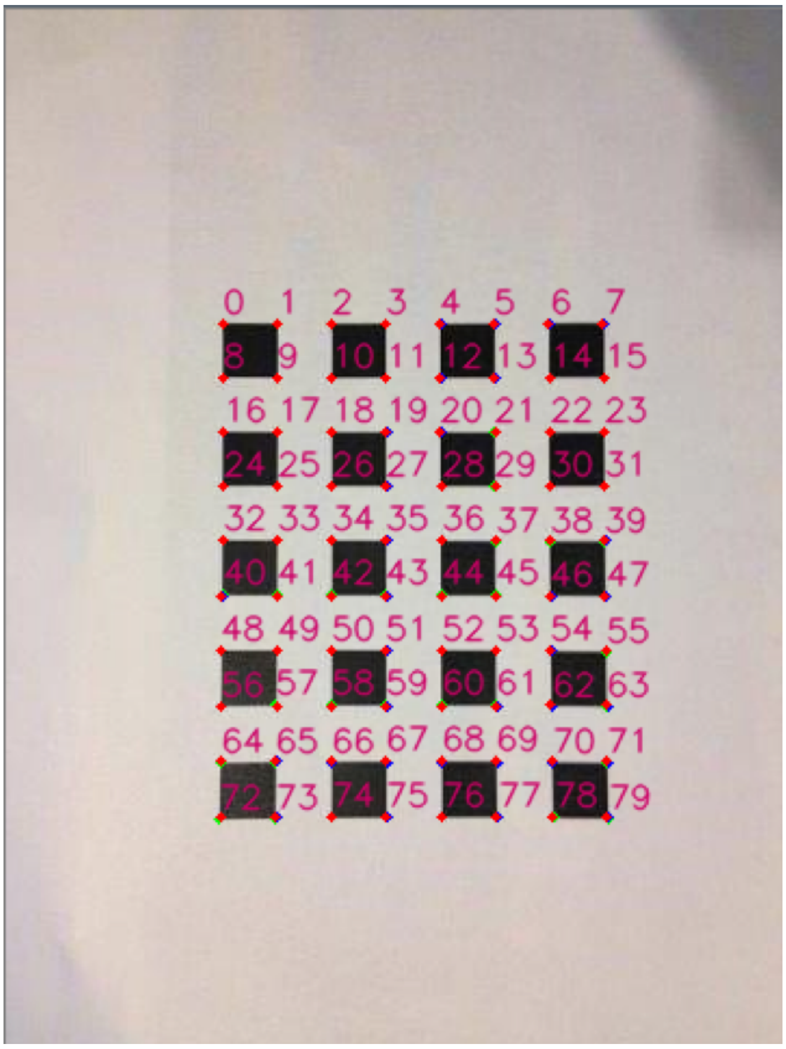

Figure 29: Dataset 2 Sample 1: first example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\vec{t}\,] = \begin{bmatrix} 0.999 & -0.0001 & 0.036 & -50.031 \\ 0.001 & 0.999 & -0.005 & -40.926 \\ -0.036 & 0.005 & 0.999 & 240.586 \end{bmatrix}$$

9.2.5 Two Dataset 2 Samples for Corner Re-projection — Sample 2

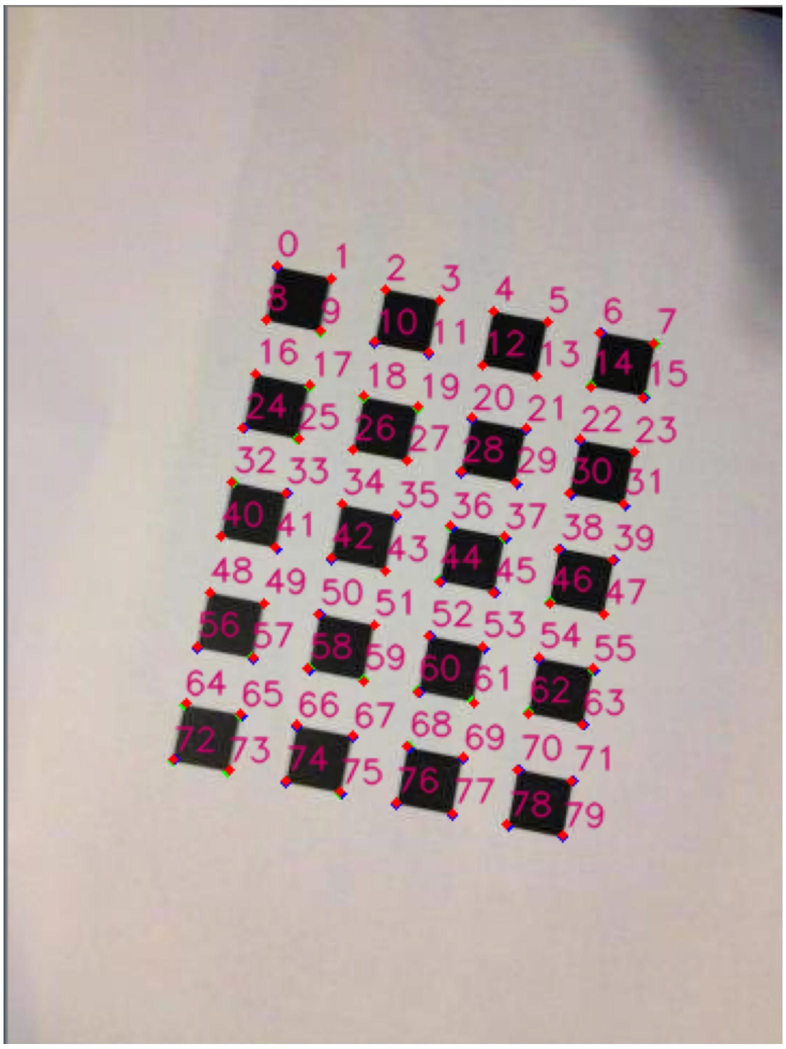

Figure 30: Dataset 2 Sample 2: second example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\overrightarrow{t}] = \begin{bmatrix} 0.979 & 0.197 & 0.053 & -62.718 \\ -0.196 & 0.980 & -0.024 & -28.076 \\ -0.056 & 0.014 & 0.998 & 237.129 \end{bmatrix}$$

9.2.6 Two Dataset 2 Samples for Corner Re-projection — Sample 3

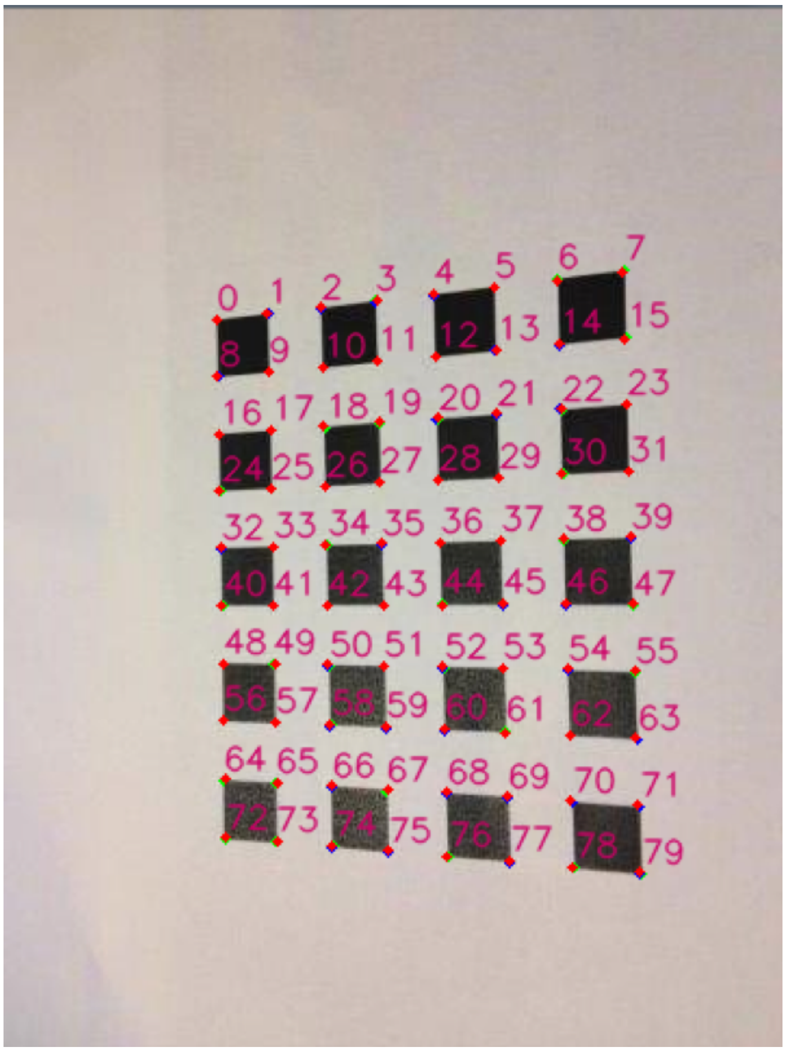

Figure 31: Dataset2 Sample 3: third example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\overrightarrow{t}] = \begin{bmatrix} 0.999 & -0.029 & 0.015 & -48.535 \\ 0.022 & 0.939 & 0.343 & -40.060 \\ -0.024 & -0.343 & 0.939 & 229.151 \end{bmatrix}$$

9.2.7 Two Dataset 2 Samples for Corner Re-projection — Sample 4

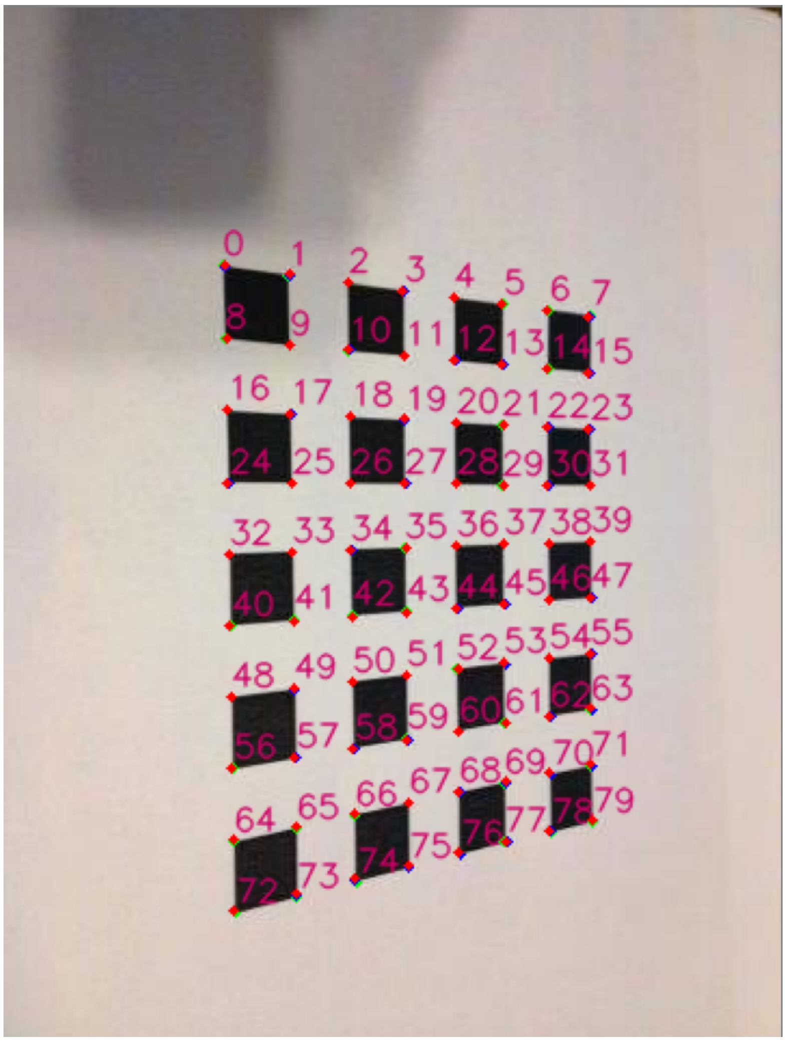

Figure 32: Dataset 2 Sample 4: fourth example of re-projection results with labels. Green dots: ground truth extracted from Canny and Hough lines. Blue dots: re-projection on the image plane using the initial camera parameter. Red dots: re-projection on the image plane using the LM-optimized camera parameter.

The external parameters is :

$$[R|\overrightarrow{t}] = \begin{bmatrix} 0.999 & -0.020 & -0.012 & -47.482 \\ 0.011 & 0.829 & -0.558 & -30.003 \\ 0.021 & 0.557 & 0.819 & 180.967 \end{bmatrix}$$

10. Code

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt
from pylab import *
from scipy.optimize import leastsq
import glob

"""
read in the image, and display them
"""
def read_raw_image(num, imgset = '1'):
    global debug
    if imgset == '1':
        print 'Read in raw image: Dataset' + imgset + '/Pic_' + num + '.JPG'
        imgBGR = cv2.imread('Dataset' + imgset + '/Pic_' + num + '.JPG')
    else:
        print 'Read in raw image: Dataset' + imgset + '/pic (' + num + ').JPG'
        imgBGR = cv2.imread('Dataset' + imgset + '/pic (' + num + ').JPG')


    if imgBGR is None:
        print 'Error in read in raw image...quit...'
        return -1
    imgGRAY = cv2.cvtColor(imgBGR, cv2.COLOR_BGR2GRAY)
    if debug == 1:
        cv2.imshow('imgBGR', imgBGR)
        cv2.imshow('imgGRAY', imgGRAY)
        cv2.waitKey(0)
        cv2.destroyWindow('imgBGR')
        cv2.destroyWindow('imgGRAY')
    return imgBGR, imgGRAY

"""
Perform the Canny edge detector on the image to find the border of the checker mark
Then use the HoughLine to find the lines connecting the corners of checker mark
"""
def CannyEdge_HoughLine(imgBGR, imgGRAY, HoughThres = 50):
    global debug
    print 'Canny edge detecting...'
    blur = cv2.GaussianBlur(imgGRAY, (5,5), 0)
    edge = cv2.Canny(blur, 2500, 4000, apertureSize=5)
    vis = imgBGR.copy()
    vis /= 2 # decrease the brightness of image so that edges stand out!
    vis[edge != 0] = (0, 255, 0)
    if debug == 1:
        cv2.imshow('Canny edges', vis)
        cv2.waitKey(0)
        cv2.imwrite('CannyEdge.png',vis)

    print 'Hough line detecting...'

```

```
##    The HoughLinesP didn't work very well. The arguments are very hard to tune
##    minLineLength = 100
##    maxLineGap = 10
##    lines = cv2.HoughLinesP(edge,1,np.pi/180,50,minLineLength,maxLineGap)
##    for x1,y1,x2,y2 in lines[0]:
##        cv2.line(vis,(x1,y1),(x2,y2),(0,0,255),2)

    lines = cv2.HoughLines(edge,1,np.pi/180,HoughThres)
    if debug == 1:
        print 'lines: \n', lines
    # get line in polar coordinate
    for rho,theta in lines[0]:
        a = np.cos(theta)
        b = np.sin(theta)
        # (x0,y0) the center point on the line
        x0 = rho*a
        y0 = rho*b
        # need two extreme points to draw the line
        x1 = int(x0 + 1000*(-b))
        y1 = int(y0 + 1000*(a))
        x2 = int(x0 - 1000*(-b))
        y2 = int(y0 - 1000*(a))

        cv2.line(vis,(x1,y1),(x2,y2),(0,0,255),3)

    if debug == 1:
        cv2.imshow('Hough Lines', vis)
        cv2.waitKey(0)
        cv2.destroyWindow('Canny edges')
        cv2.destroyWindow('Hough Lines')
        cv2.imwrite('HoughLine.png', vis)
    return lines

"""
return the line in HC from the line in 2D physical space
"""
def GetLineHC(line):
    # get parameters of the line
    rho = line[0]
    theta = line[1]

    # get two points from the line, represent them in HC
    pt0 = np.array([rho*np.cos(theta), rho*np.sin(theta), 1.0])
    pt1 = np.array([pt0[0] + 100*np.sin(theta), pt0[1] - 100*np.cos(theta), 1.0])

    # get the line in HC
    lineHC = np.cross(pt0, pt1)
    lineHC[0] /= lineHC[2]
    lineHC[1] /= lineHC[2]
    lineHC[2] = 1.0
    return lineHC

"""
```

```
return the intersection between two lines, can return either HC pt or physical pt
"""
def GetIntersection(line1, line2, HC):
    pt = np.cross(line1, line2)
    if HC == 1:
        return pt
    elif HC == 0:
        pt[0] /= pt[2]
        pt[1] /= pt[2]
        return (pt[0], pt[1])

"""
We group the lines into horizontal ones and vertical ones, then find the
    intersection between them.
Since multiple lines will be detected from the edges, we want to apply a non-
    maximum
suppresion to get cleaner lines. We could use gradient map or just HarrisCorner.
"""
def FindCorner(imgBGR, lines, size, minDist = 15):
    global debug
    imgBGR_copy = imgBGR.copy()
    # group all the horizon lines and all the vertical lines
    N = len(lines[0])
    thetas = lines[0,:,1].copy()
    thetas -= 3.1415926/2
    hori_Idx = map(int, np.where(abs(thetas)<3.1415926/4)[0])
    hori_lines = lines[0,hori_Idx,:]
    vert_Idx = map(int, np.where(abs(thetas)>=3.1415926/4)[0])
    vert_lines = lines[0,vert_Idx,:]
    assert(len(hori_lines) + len(vert_lines) == N)

    # sort the lines (horizontal from upside down, vertical from leftside right)
    hori_lines = sorted(hori_lines, key = lambda line:line[0]*np.sin(line[1]),
    reverse = False)
    vert_lines = sorted(vert_lines, key = lambda line:line[0]*np.cos(line[1]),
    reverse = False)

    # select the lines from the Houghlines, there will be multiple lines detected
    # for each edge, here I just simply choose the lefter one or the upper one
    # not guaranteed to work perfectly

    # select horizontal lines
    selected_hori_lines = []
    for i in range(len(hori_lines)):
        rho = hori_lines[i][0]
        theta = hori_lines[i][1]
        if i == 0:
            selected_hori_lines.append(hori_lines[i])
        elif abs(rho*np.sin(theta) - selected_hori_lines[-1:][0][0] * np.sin(
    selected_hori_lines[-1:][0][1])) > minDist:
            selected_hori_lines.append(hori_lines[i])

    # select vertical lines
```

```python
    selected_vert_lines = []
    for i in range(len(vert_lines)):
        rho = vert_lines[i][0]
        theta = vert_lines[i][1]
        if i == 0:
            selected_vert_lines.append(vert_lines[i])
        elif abs(rho * np.cos(theta) - selected_vert_lines[-1:][0][0] * np.cos(selected_vert_lines[-1:][0][1])) > minDist:
            selected_vert_lines.append(vert_lines[i])

    # display the lines, one by one
    if debug == 1:
        imgBGR_copy = imgBGR.copy()
        for i in range(len(selected_hori_lines)):
            rho = selected_hori_lines[i][0]
            theta = selected_hori_lines[i][1]
            a = np.cos(theta)
            b = np.sin(theta)
            # (x0,y0) the center point on the line
            x0 = rho*a
            y0 = rho*b
            # need two extreme points to draw the line
            x1 = int(x0 + 1000*(-b))
            y1 = int(y0 + 1000*(a))
            x2 = int(x0 - 1000*(-b))
            y2 = int(y0 - 1000*(a))

            cv2.line(imgBGR_copy,(x1,y1),(x2,y2),(0,0,255),3)
            cv2.imshow('Horizontal group', imgBGR_copy)
            cv2.waitKey(0)

        imgBGR_copy = imgBGR.copy()
        for i in range(len(selected_vert_lines)):
            rho = selected_vert_lines[i][0]
            theta = selected_vert_lines[i][1]
            a = np.cos(theta)
            b = np.sin(theta)
            # (x0,y0) the center point on the line
            x0 = rho*a
            y0 = rho*b
            # need two extreme points to draw the line
            x1 = int(x0 + 1000*(-b))
            y1 = int(y0 + 1000*(a))
            x2 = int(x0 - 1000*(-b))
            y2 = int(y0 - 1000*(a))

            cv2.line(imgBGR_copy,(x1,y1),(x2,y2),(0,0,255),3)
            cv2.imshow('Vertical group', imgBGR_copy)
            cv2.waitKey(0)

        cv2.destroyWindow('Vertical group')
        cv2.destroyWindow('Horizontal group')

    # detect the number of lines detected,

    N_hori = len(selected_hori_lines)
    N_vert = len(selected_vert_lines)
    # ok, we don't assert here, because it's not guaranteed that all corners can be
     detected correctly
##    assert ((N_hori == size[0] and N_vert == size[1]) or (N_hori == size[1] and
    N_vert == size[0]))

    # find the intersection between each pair of lines from two groups
    intersections = np.zeros((N_hori, N_vert, 2))
    for i in range(N_hori):
        for j in range(N_vert):
            line_h = GetLineHC(selected_hori_lines[i])
            line_v = GetLineHC(selected_vert_lines[j])
            intersections[i,j,:] = GetIntersection(line_h, line_v, HC = 0)
    return intersections

"""
Refine the corners found on the chessboard. Use the non-maximum suppresion on  the
Sobel mask. The window size is about 10-15, depending on the size of the block.
"""
def RefineCorner(imgBGR, corners):
    global debug
    imgBGR_copy = imgBGR.copy()
    gray = cv2.cvtColor(imgBGR, cv2.COLOR_BGR2GRAY)
    N_hori, N_vert, _ = corners.shape
    corners_pixel = np.round(corners).astype(int)
    corners_1D = np.array([corner for row in corners for corner in row], np.float32
    )
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
    if debug:
        print 'corners_1D before subpix: \n', corners_1D
    cv2.cornerSubPix(gray,corners_1D,(11,11),(-1,-1),criteria)
    if debug:
        print 'corners_1D after subpix: \n', corners_1D
    refined_corners = np.reshape(corners_1D, (N_hori, N_vert,2))
    return refined_corners

"""
Display the corners
"""

def DisplayCorner(imgBGR, corners, winname = 'corners', color = (0,255,0), r = 2,
    thick = -1):
    imgBGR_copy = imgBGR.copy()
    if len(corners.shape) == 2:
        corners = np.reshape(corners, (8,10,2))
    N_hori, N_vert,_ = corners.shape
    font = cv2.FONT_HERSHEY_SIMPLEX
    for i in range(N_hori):
        for j in range(N_vert):
            pos = tuple(np.round(corners[i,j]).astype(int))
            cv2.circle(imgBGR_copy, pos, r, color, thick )
            cv2.putText(imgBGR_copy,str(i*N_vert + j),pos, font, 0.5,(0,255,0),1,
    cv2.CV_AA)
    cv2.imshow(winname, imgBGR_copy)
    cv2.waitKey(0)
    return imgBGR_copy

"""
construt Vij from H
"""
def construtVij (H,i,j):
    hi = H[:,i-1]
    hj = H[:,j-1]
    Vij = np.zeros((6,1), np.float)
    Vij[0] = hi[0]*hj[0]
    Vij[1] = hi[0]*hj[1]+hi[1]*hj[0]
    Vij[2] = hi[1]*hj[1]
    Vij[3] = hi[2]*hj[0] + hi[0]*hj[2]
    Vij[4] = hi[2]*hj[1] + hi[1]*hj[2]
    Vij[5] = hi[2]*hj[2]
    return Vij

"""
from the estimated homography H (3x3), construct the vector V (2x6)
"""
def constructV(H):
    V = np.zeros((2,6), np.float)
    # construct V12
    V12 = construtVij(H,1,2)
    V11 = construtVij(H,1,1)
    V22 = construtVij(H,2,2)
    V[0,:] = np.transpose(V12)
    V[1,:] = np.transpose(V11-V22)
    return V

"""
from the calculated b (6x1), construct the symmetric W(3x3)
"""
def constructW(b):
    W = np.zeros((3,3), np.float)
    W[0,0] = b[0]
    W[0,1] = b[1]
    W[0,2] = b[3]
    W[1,0] = W[0,1]
    W[1,1] = b[2]
    W[1,2] = b[4]
    W[2,0] = W[0,2]
    W[2,1] = W[1,2]
    W[2,2] = b[5]
    return W

"""
from the W(3x3), construct K(3x3)
```

```
"""
def constructK(W):
    K = np.zeros((3,3), np.float)
    # rename the variable so that the formula is easier to follow
    w11 = W[0,0]
    w12 = W[0,1]
    w13 = W[0,2]
    w22 = W[1,1]
    w23 = W[1,2]
    w33 = W[2,2]

    # construct the componenet now!
    x0 = (w12*w13 - w11*w23)/(w11*w22 - w12*w12)
    lamda = w33 - (w13*w13 + x0*(w12*w13-w11*w23))/w11
    alphax = np.sqrt(lamda/w11)
    alphay = np.sqrt((lamda*w11)/(w11*w22-w12*w12))
    s = -(w12*alphax*alphax*alphay)/lamda
    y0 = s*x0/alphay - w13*alphax*alphax/lamda

    #fill in K
    K[0,0] = alphax
    K[0,1] = s
    K[0,2] = x0
    K[1,1] = alphay
    K[1,2] = y0
    K[2,2] = 1
    print 'Camera Intrinsic Parameter K: \n', K
    return K

"""
swap the first and second column of a matrix
"""
def swap2Col(my_array):
    new_array = np.zeros((my_array.shape))
    new_array[:, 1] = my_array[:, 0].copy()
    new_array[:, 0] = my_array[:, 1].copy()
    return new_array

"""
Condition the rotation matrix, just get the SVD and set all Eval to 1.
"""
def ConditionR(R):
    U,s,V = np.linalg.svd(R)
    new_R = np.dot(U,V)
    return new_R

"""
Calculate the Extrinsic camera pamameter from H and K, usign Zhang's formula
"""
def CalcExtrin(H,K):
    h1 = H[:,0]
    h2 = H[:,1]
    h3 = H[:,2]
    K_inv = np.linalg.inv(K)
    t = np.dot(K_inv, h3)
    lamda = 1/np.linalg.norm(np.dot(K_inv, h1))
    if t[2] < 0:
        lamda = -lamda
    t *= lamda
    r1 = lamda * np.dot(K_inv, h1)
    r2 = lamda * np.dot(K_inv, h2)
    r3 = np.cross(r1,r2)

    Q = np.transpose(np.array([r1, r2, r3]))
    R = ConditionR(Q)

    return R,t


"""
Estimate the Homography between correspondence points, using SVD
"""
def estimateHomo(source_pts, des_pts):
    print 'my FindHomograph function starts: '
    row, col = source_pts.shape
    if (col == 2 and row >=4 ):
        source_pts = np.transpose(source_pts)
    elif (col >= 4 and row == 2):
        pass
    _, N1 = source_pts.shape

    row, col = des_pts.shape
    if (col == 2 and row >=4 ):
        des_pts = np.transpose(des_pts)
    elif (col >= 4 and row == 2):
        pass
    _, N2 = des_pts.shape

    if (N1 != N2):
        print 'Error! Correspondence pts don''t have the same length!'


    A = np.zeros((2*N1,9), np.float32)
    for i in range(N1):
        X1 = source_pts[0,i]
        Y1 = source_pts[1,i]
        X2 = des_pts[0,i]
        Y2 = des_pts[1,i]
        A[2*i:2*i+2, :] = np.array([[0,0,0,-X1,-Y1, -1, Y2*X1, Y2*Y1, Y2],[X1,Y1
,1,0,0,0,-X2*X1,-X2*Y1,-X2]])
    U,s,V = np.linalg.svd(A)
    minIdx = s.argmin()
    assert(s[minIdx]>0)
    minEigVec = V[minIdx,:]
    H = np.reshape(minEigVec, (3,3))
    return H
```

```python

"""
Add the radial distortion to the pixels. The method is based on Tommy's
    implementation.
"""
def AddRadialDist (Projected_Pixels, K, k1, k2):
    Projected_Pixels = np.vstack((Projected_Pixels, np.ones((1,80))))
    K_inv = np.linalg.inv(K)
    xyz = np.dot(K_inv, Projected_Pixels)
    r2 = xyz[0]*xyz[0] + xyz[1]*xyz[1]
    r4 = r2*r2

    xyz_dist = xyz * (1+k1*r2 + k2 *r4)
    Projected_Pixels_dist = np.dot(K, xyz_dist)
    return [Projected_Pixels_dist[0], Projected_Pixels_dist[1]]

"""
Project the model points onto the image plane using the camera parameters (K,R,t,k1
    ,k2)
"""
def reprojection(corners, K, R, t, H, k1=0, k2=0):
    model = np.load('model.npy')
    # reprojection
    model_HC = np.hstack((model, np.ones((80,1))))

    # direct projection using H
    pix = np.dot(H, np.transpose(model_HC))
    pix[0,:] /= pix[2,:]
    pix[1,:] /= pix[2,:]
    re_proj_err_H = corners - np.transpose(pix[0:2,:])
    repro_err_H = [np.linalg.norm(row) for row in re_proj_err_H]
##    print 'Average Re-projection error |x-HXm| = ', np.mean(repro_err_H)
##    print 'Max Re-projection error |x-HXm| = ', np.amax(repro_err_H)

    # indirect projection using K,R,T
    Rt = np.hstack((R, np.reshape(t,(3,1))))
    model_HC = np.hstack((model, np.zeros((80,1)), np.ones((80,1))))
    pix_indir = np.dot(np.dot(K, Rt), np.transpose(model_HC))
    pix_indir[0,:] /= pix_indir[2,:]
    pix_indir[1,:] /= pix_indir[2,:]
    pix_indir_radial = AddRadialDist (pix_indir[0:2,:], K, k1, k2)
    re_proj_err_KRt = corners - np.transpose(pix_indir_radial)
    repro_err_KRt = [np.linalg.norm(row) for row in re_proj_err_KRt]
##    print 'Average Re-projection error |x-K[R|t]Xm| = ', np.mean(repro_err_KRt)
##    print 'Max Re-projection error |x-K[R|t]Xm| = ', np.amax(repro_err_KRt)
##    print '\n'
    projected_pixels = np.transpose(pix_indir[0:2,:])
    return repro_err_KRt, projected_pixels


"""
Combine all the parameters together into a structure for LM algorihtm
"""
```

```
def toParameter(K, ws, ts, k1, k2, N = 20):
    params = np.zeros((7+N*6,1))
    params[0:3] = np.reshape(K[0,:], (3,1))
    params[3:5] = np.reshape(K[1,1:3], (2,1))
    params[5] = k1
    params[6] = k2
    for i in range(N):
        params[7+6*i: 10+6*i] = np.reshape(ws[:,i], (3,1))
        params[10+6*i: 13+6*i] = np.reshape(ts[:,i], (3,1))
    return params

"""
Split all the parameters to separate values after LM algorihtm
"""
def fromParameter(params, N = 20):
    K = np.zeros((3,3))
    K[0,:] = np.reshape(params[0:3], 3)
    K[1,1:3] = np.reshape(params[3:5], 2)
    K[2,2] = 1
    k1 = params[5]
    k2 = params[6]
    ws = np.zeros((3,N))
    ts = np.zeros((3,N))
    for i in range(N):
        ws[:,i] = np.reshape(params[7+6*i: 10+6*i],3)
        ts[:,i] = np.reshape(params[10+6*i: 13+6*i],3)
    return K, ws, ts, k1, k2

"""
Sum the Eclidean distance between the projection and model points, as a residual
    function
for the LM algorithm to use. This is the cost function for the LM algorithm.
"""
def residual(params, N = 20):
    K, ws, ts, k1, k2 = fromParameter(params, N)
    model = np.load('model.npy')
    model_HC = np.hstack((model, np.zeros((80,1)), np.ones((80,1))))
    all_res = []
    for i in range(N):
        corners = np.load('Data2/corners'+str(i+1)+'.npy')
        t = ts[:,i]
        R = cv2.Rodrigues(ws[:,i])[0]
##        R = ConditionR(Q)
        Rt = np.hstack((R, np.reshape(t,(3,1))))
        pix_indir = np.dot(np.dot(K, Rt), np.transpose(model_HC))
        pix_indir[0,:] /= pix_indir[2,:]
        pix_indir[1,:] /= pix_indir[2,:]
        pix_indir_radial = AddRadialDist (pix_indir[0:2,:], K, k1, k2)
        re_proj_err_KRt = corners - np.transpose(pix_indir_radial)
##        repro_err_KRt = [row[0]**2+row[1]**2 for row in re_proj_err_KRt]
        all_res.append(re_proj_err_KRt)
    res = np.ravel(all_res)
    return res
```

```


"""
main function
step 1: extract corners and estimate H
step 2: estimate the camera parameters and reproject to evaluate the parameters

imgset 1: provided image set
imgset 2: my own image set

set debug = 1 to display intermediate results.
"""
def main():
    global debug
    debug = 1
    imgset = '2'
    step = 2
    if step == 1:
        print "Estimate Homography for each image and generate Vs"
##        images = glob.glob('Dataset2\pic*.JPG')
##        print '\n', len(images), 'images loaded for left camera calib\n'
        ##cv2.namedWindow('img', cv2.WINDOW_NORMAL)

##        img_num = raw_input ('Please enter image number: ')
    ##    img_num = '1'

##        # to get model for my image set
##        refined_corners = np.zeros((10,8,2))
##        for i in range(10):
##            for j in range(8):
##                refined_corners[i,j,:] = [13*i, 13*j] # 13mm for each block
##        refined_corners = np.reshape(refined_corners, (80,2))
##        np.save('model', refined_corners)
        for img_num in range(0,20):
            img_num = str(img_num+1)
            imgBGR, imgGRAY = read_raw_image(img_num, imgset)
            lines = CannyEdge_HoughLine(imgBGR, imgGRAY, 40) # change 50 to a
    smaller number if any line is missing
            corners = FindCorner(imgBGR, lines, (8,10), 15) # change 15 to a bigger
     number if multiple lines are detected
            DisplayCorner(imgBGR, corners, 'corner', (255,0,0))
            refined_corners = RefineCorner(imgBGR, corners)
            DisplayCorner(imgBGR, refined_corners, 'refined corner', (0,0,255))

            N_hori, N_vert, _ = refined_corners.shape
            assert(N_hori == 10 and N_vert == 8)
##            refined_corners -= refined_corners[0,0,:]
            refined_corners = np.reshape(refined_corners, (80,2))
            refined_corners = swap2Col(refined_corners)
##            np.save('model', refined_corners)

            model = np.load('model.npy')
##            haha_model = np.zeros((80,2))
##              haha_refine = np.zeros((80,2))
##              haha_model[:,0] = model[:,1]
##              haha_model[:,1] = model[:,0]
##              haha_refine[:,0] = refined_corners[:,1]
##              haha_refine[:,1] = refined_corners[:,0]

##              H = myfindHomography(haha_model, haha_refine)
            H = estimateHomo(model, refined_corners)
            model_HC = np.hstack((model, np.ones((80,1))))
            refined_corners_HC = np.dot(H, np.transpose(model_HC))
            refined_corners_HC[0,:] /= refined_corners_HC[2,:]
            refined_corners_HC[1,:] /= refined_corners_HC[2,:]
            re_proj_err_H = refined_corners - np.transpose(refined_corners_HC
    [0:2,:])
##              if debug:
##                  print re_proj_err_H
            print 'Re-projection error |x-HX| = ', np.linalg.norm(re_proj_err_H)
            np.save('Data2/H'+img_num, H)
            np.save('Data2/corners'+img_num, refined_corners)
            cv2.destroyAllWindows()

    elif step == 2:
        print 'Intrinsic parameter estimation...'
        N = 20
        Vs = np.zeros((2*N,6), np.float)
        Hs = []
        for i in range(20):
            H = np.load('Data2/H'+str(i+1)+'.npy')
            Hs.append(H)
            V = constructV(H)
            Vs[2*i:2*i+2,:] = V
        U,s,V = np.linalg.svd(Vs)
        b = V[5,:]
        ##print np.linalg.norm(np.dot(Vs, b))
        W = constructW(b)
        K = constructK(W)

        print 'Extrinsic parameter estimation...'
        # calculate the extrinsic parameters R and t
        ws = np.zeros((3,N))
        ts = np.zeros((3,N))

        repro_err = []
        for i in range(N):
         ##   print str(i+1) + 'th image: '
            H = Hs[i]
            R,t = CalcExtrin(H, K)
            w = cv2.Rodrigues(R)[0]
            ws[:,i] = np.reshape(w,3)
            ts[:,i] = np.reshape(t,3)
            refined_corners = np.load('Data2/corners'+str(i+1)+'.npy')
            error, projected_coor = reprojection(refined_corners, K, R, t, H)
            repro_err.append(error)
        repro_err = [num for elem in repro_err for num in elem]
        print 'Mean Re-projection error before LM  |x-K[R|t]Xm| = ', np.mean(repro_err)
        print 'Std Re-projection error before LM  |x-K[R|t]Xm| = ', np.std(repro_err)
        print 'Max Re-projection error before LM  |x-K[R|t]Xm| = ', np.amax(repro_err)

        repro_err = []
        print 'LM optimization starts'
        k1 = 0
        k2 = 0
        params = toParameter(K, ws, ts, k1, k2, N)
        refined_params = leastsq (residual, params)[0]
        newK, newws, newts, newk1, newk2 = fromParameter(refined_params)
        print 'new K: \n', newK
        print 'new k1,k2: (%f,%f)' % (newk1,newk2)
        # reproject, to see if the result is indeed better
        for i in range(N):
        ##    print str(i+1) + 'th image: '
            img_num = str(i+1)
            imgBGR, imgGRAY = read_raw_image(img_num, imgset)
            imgBGR_copy = imgBGR.copy()
            H = Hs[i]

            # calculate the new projection
            newt = newts[:,i]
        ##    print newt
            newQ = cv2.Rodrigues(newws[:,i])[0]
            newR = ConditionR(newQ)
        ##    print newR
            refined_corners = np.load('Data2/corners'+str(i+1)+'.npy')
            error, projected_coor_LM = reprojection(refined_corners, newK, newR, newt, H, newk1, newk2)
            repro_err.append(error)

            # calculate the old projection
            R,t = CalcExtrin(H, K)
            _, projected_coor = reprojection(refined_corners, K, R, t, H, k1, k2)

            # swap x and y
            refined_corners = swap2Col(refined_corners)
            projected_coor_LM = swap2Col(projected_coor_LM)
            projected_coor = swap2Col(projected_coor)
            font = cv2.FONT_HERSHEY_SIMPLEX
            for i in range(80):
                pos = tuple(np.round(refined_corners[i,:]).astype(int))
                cv2.putText(imgBGR_copy,str(i), tuple((pos[0], pos[1]-5)), font, 0.5,(102,0,204),1,cv2.CV_AA)
                # ground truth
                cv2.circle(imgBGR_copy, pos, 2, (0,255,0), -1)

                # init parameter projection
```

```
                pos = tuple(np.round(projected_coor[i,:]).astype(int))
                cv2.circle(imgBGR_copy, pos, 2, (255,0,0), -1)

                # parameter projection after LM
                pos = tuple(np.round(projected_coor_LM[i,:]).astype(int))
                cv2.circle(imgBGR_copy, pos, 2, (0,0,255), -1)

            cv2.imshow('projection', imgBGR_copy)
            cv2.waitKey(0)

        repro_err = [num for elem in repro_err for num in elem]
        print 'Mean Re-projection error after LM  |x-K[R|t]Xm| = ', np.mean(repro_err)
        print 'Std Re-projection error before LM  |x-K[R|t]Xm| = ', np.std(repro_err)
        print 'Max Re-projection error after LM  |x-K[R|t]Xm| = ', np.amax(repro_err)
        cv2.destroyAllWindows()
if __name__ == "__main__":
    main()
```