ECE 661: Computer Vision

HOMEWORK 8, FALL 2018

Arindam Bhanja Chowdhury

abhanjac@purdue.edu

1 Overview

The goal of this homework is to implement the popular Zhang's algorithm for camera calibration. The camera model assumed for this assignment will be a pin hole camera. This implies that a complete calibration process will involve all the 5 intrinsic and 6 extrinsic parameters that determine the position and orientation of the camera with respect to a reference world coordinate system. This will require to establish correspondences between images and their world coordinates. A checker board pattern is provided for this calibration. Another set of pattern should be created by us and the calibration parameters should be calculated for each of these two datasets of checker board images.

The given dataset consists of 40 images of the checker board in taken from different viewpoints and the dataset we created consists of 20 images.

2 Zhang's Algorithm

The camera calibration algorithm developed by Zhang is used to find the 3x3 intrinsic parameter matrix K of the camera and also the extrinsic parameters R and t with respect to a chosen frame of reference. R is the rotation matrix from the physical world coordinate to the camera frame coordinates and t is the translation vector that should be applied to the coordinate values after the R is applied to a vector in the world coordinate. The overall equation is like

$$X_{cam} = RX_{world} + t \tag{1}$$

And the K matrix is given by

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

α_x and α_y are the focal length of the camera in the x and y direction in terms of number of pixels using the x and y sampling rate respectively. x_0 and y_0 are the integer coordinates of the principle points in the image plane and s is the skew parameter in the xy-layout of the sensor cells of the camera.

2.1 Calculation of Intrinsic Parameters

The algorithm assumes that the calibration pattern is in the z=0 plane in the world frame and the images of the patterns are obtained from different veiwpoints.

Denoting the pixel coordinates by the homogeneous vectors $x = [x, y, z, w]^T]$ we have the following

$$x = K[R|t] \times [X, Y, 0, W]^T = HX_M \tag{3}$$

or

$$x = K[r_1, r_2, r_3|t] \times [X, Y, 0, W]^T = HX_M \tag{4}$$

or since the 3rd element (Z) of the $[X, Y, 0, W]^T$ is 0, so this can also be written as

$$x = K[r_1, r_2|t] \times [X, Y, W]^T = HX_M \tag{5}$$

where the $X_M = [X, Y, W]^T$. The subscript M refers to Model, meaning the calibration pattern. H can be written as

$$H = [h_1, h_2, h_3] = K[r_1, r_2|t] \tag{6}$$

which is a homography matrix with h_1, h_2, h_3 as its three column vectors. And $R = [r_1, r_2, r_3]$ is the rotation matrix whose column vectors are r_1, r_2, r_3.

Zhang's algorithm is based on the fact that the camera image of the Absolute conic is independent of the R and t and that it is given by $\omega = K^{-T}K^{-1}$. Also, any plane in the world frame samples the absolute conic at exactly two points. The images of these two points fall on the conic ω in the camera image plane. Each of these two points must obey the conic constraint $x^T\omega x = 0$. When the coordinates of two image points are plugged into the conic constraint equation, the following equations are obtained

$$h_1^T\omega h_1 = h_2^T\omega h_2 \tag{7}$$

and

$$h_1^T\omega h_2 = 0 \tag{8}$$

From the calibration pattern the pixel locations of the corners of the black boxes can be found out, and the physical separation between these pixels can be used to find the world coordinates of the pixels. So using the pixel locations as x and the world locations as X_M, for all the corners of all the black boxes in the pattern, the Homography H can be estimated using linear least squares from the equation $x = HX_M$. The method for doing this is the same as done in Homework 2 and is mentioned in the Appendix section. This will give the values of all the columns of H matrix.

Given the h's for a number of different positions of the camera, the goal is to estimate ω and from there estimate K.

Now if ω is given by

$$\omega = \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{12} & \omega_{22} & \omega_{23} \\ \omega_{13} & \omega_{23} & \omega_{33} \end{bmatrix} \tag{9}$$

This is a symmetric matrix.

Using (7), (8) and (9) the following equation can be written

$$\begin{bmatrix} V_{12}^T \\ V_{11}^T - V_{22}^T \end{bmatrix}_{2\times 6} \begin{bmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{22} \\ \omega_{13} \\ \omega_{23} \\ \omega_{33} \end{bmatrix}_{6\times 1} = 0_{2\times 1} \tag{10}$$

Where

$$V_{ij} = \begin{bmatrix} h_{1i}h_{1j} \\ h_{1i}h_{2j} + h_{2i}h_{1j} \\ h_{2i}h_{2j} \\ h_{3i}h_{1j} + h_{1i}h_{3j} \\ h_{3i}h_{2j} + h_{2i}h_{3j} \\ h_{3i}h_{3j} \end{bmatrix} \tag{11}$$

The first suffix is the row index and the second suffix is the column index.

Now there will be 2 such equation obtained from each of the viewpoint of the calibration pattern, so if there are 40 viewpoints, then the overall V matrix will be 80x6 matrix.

Since H is already obtained by linear least squares, so V is known. Now using the linear least squares again, the ω elements can be obtained.

Now, the values of the intrinsic parameters of the K matrix can be obtained as per the following equations from Zhang's report

$$\begin{aligned} y_0 &= \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2} \\ \lambda &= \omega_{33} - \frac{\omega_{33}^2 + y_0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}} \\ \alpha_x &= \sqrt{\frac{\lambda}{\omega_{11}}} \\ \alpha_y &= \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}} \\ s &= -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda} \\ x_0 &= \frac{sy_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda} \end{aligned} \tag{12}$$

Thus the K matrix can be formed using (2).

2.2 Calculation of Extrinsic Parameters

For finding extrinsic parameters, the equation (5) and (6) is used. Since (5) can be written as

$$K[r_1, r_2|t] = [h_1, h_2, h_3] = H \tag{13}$$

so, it can be said that,

$$\begin{aligned} r_1 &= K^{-1}h_1 \\ r_2 &= K^{-1}h_2 \\ r_3 &= r_1 \times r_2 \\ t &= K^{-1}h_3 \end{aligned} \tag{14}$$

Now, since the r's are the columns of the rotation matrix, so they all should be of unit magnitude. So, they must all be made unit magnitude by some scale factor. In the ideal case, this scale factor should be

$$\xi_1 = \frac{1}{||K^{-1}h_1||} = \xi_2 = \frac{1}{||K^{-1}h_2||} \tag{15}$$

But they may not be the same due to small errors in calculations. Hence the magnitudes of r's are made unity only using
$\xi_1 = \frac{1}{||K^{-1}h_1||}$.
So, now the equation becomes

$$\begin{aligned} r_1 &= \xi_1 K^{-1} h_1 \\ r_2 &= \xi_1 K^{-1} h_2 \\ r_3 &= \xi_1 r_1 \times r_2 \\ t &= \xi_1 K^{-1} h_3 \end{aligned} \tag{16}$$

2.3 Refining the Calibtation Parameters

To develop a more intuitively accessible measure of the accuracy of the calculated camera paremeters, we proceed as follows: For each position of the camera used for calibration, we form the projection matrix $P = K[R|t]$ using the calculated calibration parameters. Whereas K will be the same for all positions of the camera, the extrinsic parameters R and t will be specific to each. We usi these P matrices to projecta set of salient points from the calibration pattern in the Z=0 plane into the image planes for each of camera positions. The euclidean distance between the projected pixels and where they actually occur in the images gives us a meaningful metric for accessing the quality of the calculated calibration parameters. In what follows, let's develop an analytic exptession for this metric.

NOTATION:

- $X_{m,i}$: The j^{th} salient point of the physical calibration pattern.
- $X_{i,j}$: The actual image point for $X_{m,i}$ in the i^{th} position of the camera.
- $\hat{X_{i,j}}$: The projected image point for $X_{m,i}$ using P for i^{th} position of camera.
- R_i: The Rotation matrix for the i^{th} position of the camera.
- t_i: The translational vector for the i^{th} 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 will have $X_{i,j} = \hat{X_{i,j}}$. In general the euclidean distance $||X_{i,j} - \hat{X_{i,j}}||$ tells us something about how far the calibration is with respect to the j^{th} salient point in the i^{th} position of the camera. We can visualize this euclidean distance by looking at images that show both the projected and the actulally images calibration pattern salient points.

Aggregating the error distances for all salient points and for all the camera postions, we have the

following set of equations to calculate the overall error d^2_{geom}

$$\begin{aligned} X_{m,j} &= [x_{m,j}, y_{m,j}, 0] \\ Xh_{m,j} &= [x_{m,j}, y_{m,j}, 0, 1] \qquad [converting\ to\ homogeneous\ form] \\ [x\hat{h}_{i,j}, y\hat{h}_{i,j}, w\hat{h}_{i,j}]^T &= X\hat{h}_{i,j} = K[R|t]Xh_{m,j} = K[r_{i,1}, r_{i,2}|t]Xh_{m,j} \\ [\hat{x_{i,j}}, \hat{y_{i,j}}]^T &= \hat{X_{i,j}} = [\frac{x\hat{h}_{i,j}}{w\hat{h}_{i,j}}, \frac{y\hat{h}_{i,j}}{w\hat{h}_{i,j}}] \qquad [converting\ to\ planar\ form] \\ d^2_{geom} &= \sum_i \sum_j ||X_{i,j} - \hat{X_{i,j}}||^2 \\ d^2_{geom} &= ||X - f(p)||^2 \qquad [converting\ to\ a\ more\ simple\ form] \end{aligned} \tag{17}$$

Using the simple form for the d^T_{geom} allows us to use the method of nonlinear least squares minimization like Levenberg-Marqrardt (LM) technique for minimizing the error with optimization.

Before applying the LM optimization technique the R should be represented as a vector and not a matrix. This is because R has 9 elements but only 3 degrees of freedom. In any optimization algorithm the number of variables used ot represent an entity must equal (strictly) the degree of freedom of the entity. If we minimize R as a 9 element entity, then probably there will be some absurd result for the elements of R. So to represent R as a 3 element entity, we use the Rodrigues representation in which a rotation matrix is represented as a vector $w = [w_x, w_y, w_z]^T$ such that the unit vector $\frac{w}{||w||}$ and the magnitude $||w||$ represents the direction of the axis of rotation of R and the angle Φ of clickwise rotation around this axis.

In order to go from w to R, we need to first represent w by a 3x3 matrix

$$[w]_x = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} \tag{18}$$

It can easily be seen that the matrix product is equal to a cross product like the following

$$[w]_x g = w \times g \tag{19}$$

To construct the R matrix for a given vector from w, we use the equation

$$R = I_{3\times 3} + \frac{\sin\Phi}{\Phi}[w]_x + \frac{1-\cos\Phi}{\Phi^2}[w]^2_x \tag{20}$$

Where $\Phi = ||w||$

And to construct the vector w for a given R, we first write for the angle Φ as

$$\Phi = 0.5 * \arccos\left(trace(R)\right) - 1 \tag{21}$$

and subsequently,

$$w = \frac{\Phi}{2\sin\Phi}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{22}$$

The r's are the elements of the R matrix.

But before these formulas can be used, we have to make sure that R is orthonormal. Otherwise there may be errors in finding the arccos of the trace of R as that may not be less than 1. This

should be done before converting R into Rodrigues format. This is done by calculating the SVD of the raw R (R_{raw}) and then making all the singular values 1, and then reconverting the SVD to create the orthonormal R ($R_{orthonormal}$) like the following steps:

$$\begin{aligned} R_{raw} &= U_r D_r V_r^T \\ R_{orthonormal} &= U_r V_r^T \end{aligned} \tag{23}$$

Only after this conversion the R can be safely converted into Rodrigues format and used for further processing to find the optimized set of parameters using LM algorithm.

2.4 Incorporating the Radial Distortion

To calibrate a camera with radial distortion, first we carry out all of the steps to find a set of tentative values for the intrinsic and extrinsic parameters for each position of the camera used for calibration. Now, let $(\hat{x}, \hat{y})$ be the predicted position of a pixel using the pinhole model and the tentative set of calibration parameters and let $(\hat{x_r ad}, \hat{y_r ad})$ be the pixel coordinates that would be predicted if you included the radial distortion, then we can write from Zhang's report that

$$\begin{aligned} \hat{x_r ad} &= \hat{x} + (\hat{x} - x_0)[k_1 r^2 + k_2 r^4] \\ \hat{y_r ad} &= \hat{y} + (\hat{y} - y_0)[k_1 r^2 + k_2 r^4] \end{aligned} \tag{24}$$

Where, k_1 and k_2 are the radial distortion parameters and $r^2 = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2$. The x_0 and y_0 are the currently available principle point location on the image plane of the camera.

To incorporate the optimization of k_1 and k_2, they are both initialized to 0 and the prediction function f(p) in (17) will first calculate the pixel coordinates for each salient point on the calibration pattern using just hte pinhole model, and then invoke the radial distortion model to find the final pixel coordinates. This would automatically make estimating k_1 and k_2 a part of the minimization procedure.

3 Procedure

- The calibration pattern given is printed and mounted on a wall and the camera is moved in different directions to capture images of the pattern from different viewpoints. This is how we created our dataset.
- For the very first pattern on the wall, position the camera in shuch a way that its Principle Axis is approximately perpendicular to the plane of the wall. Also it had to be made 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 approximately. The checker board patterns from two different viewpoints are shown in Fig 1 to 4.
- The objective is to find the corners of the checker board squares in the image.
- For this first the image is turned into grayscale and then Otsu's thresholding is applied to it to separate out the black squares from the white background.

- Now, there are always some amount of unwanted black regions because of the shadows in the image. To remove these, this thresholded binary image is dilated and eroded by some amount to remove these noisy regions. So now the only black regions of the images are the black squares of the checker board.
- The edges are then extracted from the images using Canny edge detector. These are shown in the Fig 5 to 8.
- Then straight lines are fitted to the edges using Hough transform (HoughLinesP function in OpenCV) and these lines are then extended to the boundary of the image.
- There will be a number of straight lines which are redundant among these group of straght lines obtained using Hough transform. To filter the unwanted straight lines, first the lines are gouuped into two groups, horizontal lines and vertical lines, depending on their slopes.
- Then the horizontal lines are sorted into a list according to their y intercept values, such that the line along the top margin of the top left square (1st square) is at the beginning of the sorted list. Then the average gap between the y intercepts of the consecutive lines in this list is calculated.
- Now, if the gap between any two consecutive lines in this list seems to be less then some fraction (for our case the fraction of 0.5 worked pretty well) of the average gap (calculated earlier) then this line is rejected. This is because it implies that it must be a redundant line otherwise it would have been separated by proper gap (which will be close to the average gap and not drastically smaller that the average gap value). There can be multiple lines which are rejected like this.
- The same process is followed to eliminate the redundant lines from the vertical lines group. Here the x intercept is used instead. These images with the final filtered groups of hough lines are shown in the Fig 9 to 12. Red denotes horizontal and blue denotes vertical lines.
- The intersection points of these lines are then used to locate the corners of the checker board. There are altogether 20 black boxes in the pattern, and so the corners of these boxes are numbered as 1 to 80 as shown in the Fig 13 to 16.
- Once this corner detection is achieved, the physical world coordinate of these points are calculated. The black boxes in the patterns are all 25mm apart, so the world coordinate of the corner no. 1 will be (0,0), that of corner 2 will be (25mm, 0), that of the 9th corner will be (0, 25mm), 10th corner will be (25mm, 25mm), 11th corner will be (50mm, 25mm) and so on. The z coordinate of all the corners will be 0, as the pattern is considered to be on the z=0 plane in the world coordinate.
- Now, a correspondence is established between the extracted corners in each image and their world coordinates.
- Zhang's algorithm is implemented using the theory and equations mentioned in the earlier sections to find the intrinsic and extrinsic parameters of the camera. The intrinsic parameters will be common for all the different viewpoints but there will be separate set of extrinsic parameters for each of the different veiwpoints. Levenberg-Marqardt (LM) algorithm is used for non-linear optimization to refine the parameters.
- During the optimization the Radial Distortion parameters are also evaluated.

- Finally these parameters are used to create the camera matrix which are multiplied with the given world coordinates. This will give an estimate of the location of these points in the image. These points are then reprojected back to the image to check how close they are relative to the original true corners. These reprojected corners are shown in the Fig 17 to 24. The Left image shows the reprojected corners (in Blue) without using LM algorithm. The right image shows the reprojected corners (in Green) after refining the camera paramters using LM optimization. The true postion of the corners found earlier by the intersection of the hough lines are shown in Red.
- The difference between the parameters found before LM optimzation and after LM optimization are not too big, so the overall difference is not very apparent in the image itself. But it is apparent from the parameter matrix element values.

4 Results

Intrinsic parameter matrix (K) before optimization for GIVEN dataset:

$$K = \begin{bmatrix} 728.03017302 & -2.98571451 & 323.02275774 \\ 0.0 & 727.9076025 & 233.62489805 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Radial distortion parameters (k_1 and k_2) before optimization for GIVEN dataset: $k_1 = 0.0$ and $k_2 = 0.0$

Refined Intrinsic parameter matrix (K) after LM optimization for GIVEN dataset:

$$K = \begin{bmatrix} 739.38253289 & -2.82779116 & 329.17239648 \\ 0.0 & 738.55692537 & 236.5889407 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Radial distortion parameters (k_1 and k_2) after LM optimization for GIVEN dataset: $k_1 = -2.818831062302062 \times 10^{-7}$ which is basically 0 and $k_2 = 5.695739404194863e \times 10^{-13}$ which is basically 0.

Intrinsic parameter matrix (K) before optimization for CREATED dataset:

$$K = \begin{bmatrix} 659.22045225 & -3.81899913 & 251.95646213 \\ 0.0 & 496.40063875 & 341.95411803 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Radial distortion parameters (k_1 and k_2) before optimization for CREATED dataset: $k_1 = 0.0$ and $k_2 = 0.0$

Refined Intrinsic parameter matrix (K) after LM optimization for CREATED dataset:

$$K = \begin{bmatrix} 663.40990528 & -1.58409888 & 253.10201321 \\ 0.0 & 496.41010958 & 333.08496853 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Radial distortion parameters (k_1 and k_2) after LM optimization for CREATED dataset: $k_1 = 2.1307304315998365 \times 10^{-7}$ which is basically 0 and $k_2 = -1.1426411602026568 \times 10^{-12}$ which is basically 0.

Overall projection error for all the corners of all the viewpoints for GIVEN dataset is given by:

Mean Error: 1.5258552543229036 (without optimization)

Error Variance: 0.1618650688979799 (without optimization)

Mean Error: 1.4073987164138568 (after LM optimization)

Error Variance: 0.223840298367582 (after LM optimization)

Overall projection error for all the corners of all the viewpoints for CREATED dataset is given by:

Mean Error: 1.7684077947339376 (without optimization)

Error Variance: 0.5305846665572558 (without optimization)

Mean Error: 0.849774311169679 (after LM optimization)

Error Variance: 0.034582343704737895 (after LM optimization)

Reprojection error (without LM) for image in Fig 21 (from CREATED dataset):

Mean: 2.153187575384726, Variance: 0.8715048907232958

Reprojection error (after LM optimization) for image in Fig 21 (from CREATED dataset):

Mean: 1.098939945816788, Variance: 0.36019333822431077

Rotation Matrix (after LM optimization) for image in Fig 21 (from CREATED dataset):

$$R = \begin{bmatrix} 0.999858434 & -0.01682561 & 1.05037132 \times 10^{-4} \\ 0.016622166 & 0.98676194 & -0.161321336 \\ 0.002610683 & 0.16130024 & 0.986901928 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 21 (from CREATED dataset):

$t = -95.43612084, -137.4873768, 474.96113317]^T$

Reprojection error (without LM) for image in Fig 22 (from CREATED dataset):

Mean: 1.3854830839431365, Variance: 0.3520938898125652

Reprojection error (after LM optimization) for image in Fig 22 (from CREATED dataset):

Mean: 0.8093756073906369, Variance: 0.17425913106883872

Rotation Matrix (after LM optimization) for image in Fig 22 (from CREATED dataset):

$$R = \begin{bmatrix} 0.82572996 & -0.5560837 & 0.0945566 \\ 0.56405151 & 0.81521062 & -0.13144401 \\ -0.00398967 & 0.16187205 & 0.98680369 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 22 (from CREATED dataset):

$t = [-24.50195067, -165.84519408, 568.15452403]^T$

Reprojection error (without LM) for image in Fig 23 (from CREATED dataset):

Mean: 2.199327553108472, Variance: 0.7398101129468406

Reprojection error (after LM optimization) for image in Fig 23 (from CREATED dataset):

Mean: 0.6602288768081548, Variance: 0.10179773602385267

Rotation Matrix (after LM optimization) for image in Fig 23 (from CREATED dataset):

$$R = \begin{bmatrix} 0.7292468 & 0.07993202 & -0.67956602 \\ -0.28042048 & 0.94083268 & -0.1902583 \\ 0.62415019 & 0.32930949 & 0.7085138 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 23 (from CREATED dataset):

$t = [-56.54012466, -42.13168915, 491.42285429]^T$

Reprojection error (without LM) for image in Fig 24 (from CREATED dataset):

Mean: 1.586176405441985, Variance: 0.43065837352060665

Reprojection error (after LM optimization) for image in Fig 24 (from CREATED dataset):

Mean: 0.8944430984043749, Variance: 0.27585144490331703

Rotation Matrix (after LM optimization) for image in Fig 24 (from CREATED dataset):

$$R = \begin{bmatrix} 0.99925598 & 0.0126823 & 0.03642305 \\ 0.00757522 & 0.86145483 & -0.5077777 \\ -0.03781661 & 0.50767582 & 0.86071782 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 24 (from CREATED dataset):

$t = [-106.846755, -95.74741577, 376.73323264]^T$

Reprojection error (without LM) for image in Fig 17 (from GIVEN dataset):

Mean: 1.8649651260617879, Variance: 0.5299750682496147

Reprojection error (after LM optimization) for image in Fig 17 (from GIVEN dataset):

Mean: 0.9749041963706405, Variance: 0.2485560396464187

Rotation Matrix (after LM optimization) for image in Fig 17 (from GIVEN dataset):

$$R = \begin{bmatrix} 0.81897657 & 0.43232906 & -0.37731812 \\ -0.41163579 & 0.90074645 & 0.13860665 \\ 0.39979164 & 0.04180205 & 0.91565235 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 17 (from GIVEN dataset):

$t = [-89.90498109, -53.02191074, 560.11315055]^T$

Reprojection error (without LM) for image in Fig 18 (from GIVEN dataset):

Mean: 1.2637059197199796, Variance: 0.5228674021190924

Reprojection error (after LM optimization) for image in Fig 18 (from GIVEN dataset):

Mean: 1.0343858996588424, Variance: 0.34271402861311473

Rotation Matrix (after LM optimization) for image in Fig 18 (from GIVEN dataset):

$$R = \begin{bmatrix} 0.99944331 & 0.02271605 & 0.02443462 \\ -0.0321368 & 0.8522093 & 0.52221311 \\ -0.00896079 & -0.52270765 & 0.85246491 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 18 (from GIVEN dataset):
$t = [-93.03789722, -117.35310859, 575.41384766]^T$
Reprojection error (without LM) for image in Fig 19 (from GIVEN dataset):
Mean: 1.9276495263798694, Variance: 0.6798774060958883
Reprojection error (after LM optimization) for image in Fig 19 (from GIVEN dataset):
Mean: 1.1365762506844987, Variance: 0.6512184115538331
Rotation Matrix (after LM optimization) for image in Fig 19 (from GIVEN dataset):

$$R = \begin{bmatrix} 0.89039137 & -0.01204457 & 0.45503642 \\ 0.10049188 & 0.98018649 & -0.17069222 \\ -0.44396463 & 0.19771035 & 0.87395997 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 19 (from GIVEN dataset):
$t = [-53.85267052, -108.31196677, 526.22151783]^T$
Reprojection error (without LM) for image in Fig 20 (from GIVEN dataset):
Mean: 1.6271276331185713, Variance: 0.6681276069230475
Reprojection error (after LM optimization) for image in Fig 20 (from GIVEN dataset):
Mean: 1.5053168160134305, Variance: 0.652995657423364
Rotation Matrix (after LM optimization) for image in Fig 20 (from GIVEN dataset):

$$R = \begin{bmatrix} 0.79539573 & -0.17863951 & 0.57916626 \\ 0.1674094 & 0.9831561 & 0.0733361 \\ -0.58251156 & 0.03862665 & 0.8119041 \end{bmatrix}$$

Translation Vector (after LM optimization) for image in Fig 20 (from GIVEN dataset):
$t = [-103.62247964, -136.90351851, 500.0751343]^T$

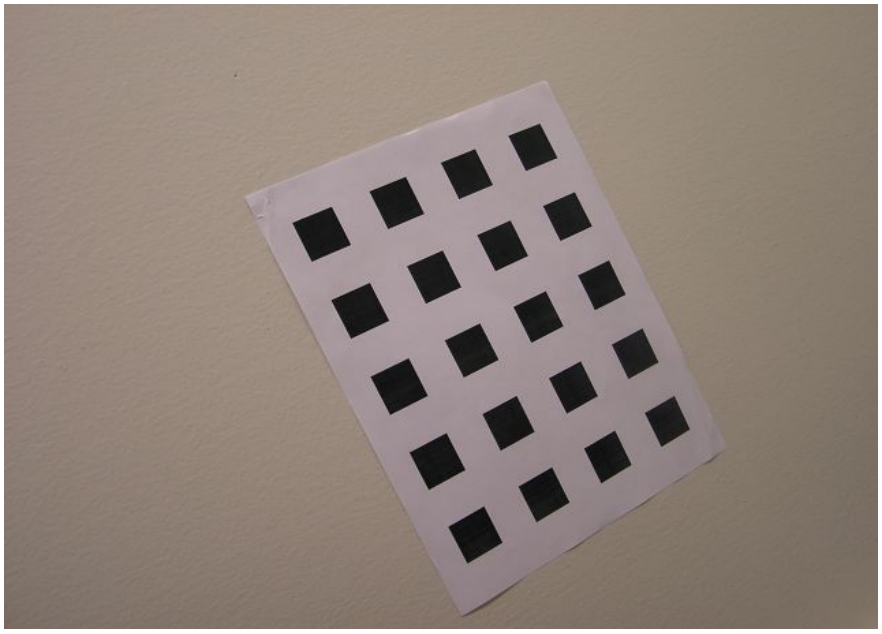

Figure 1: Sample image 1 for original checker board pattern from given dataset.

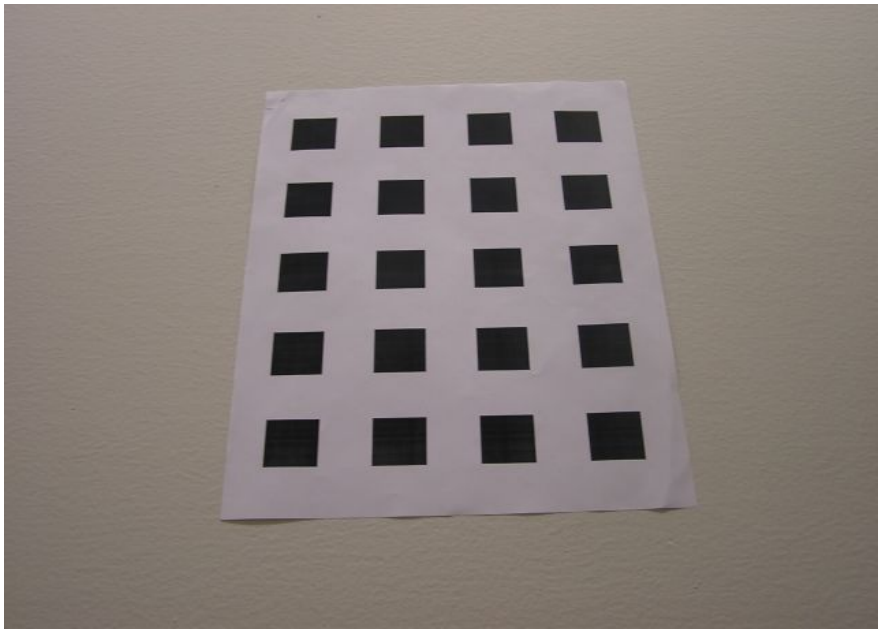

Figure 2: Sample image 2 for original checker board pattern from given dataset.

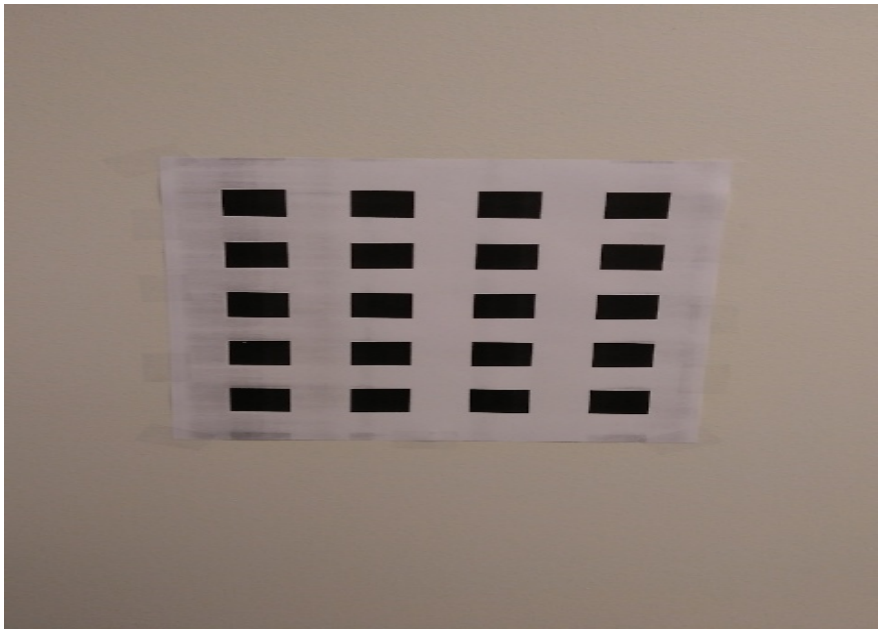

Figure 3: Sample image 1 for original checker board pattern from created dataset.

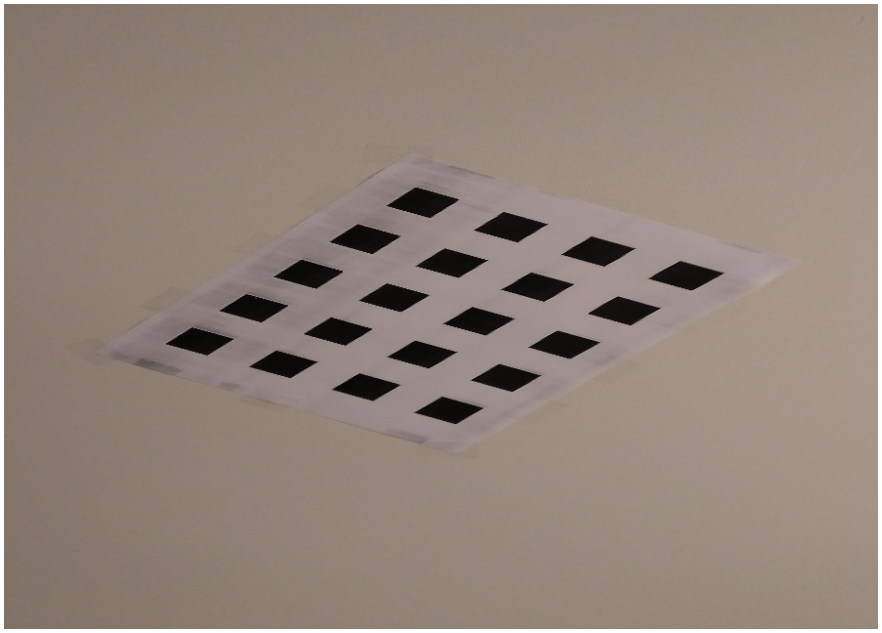

Figure 4: Sample image 2 for original checker board pattern from created dataset.

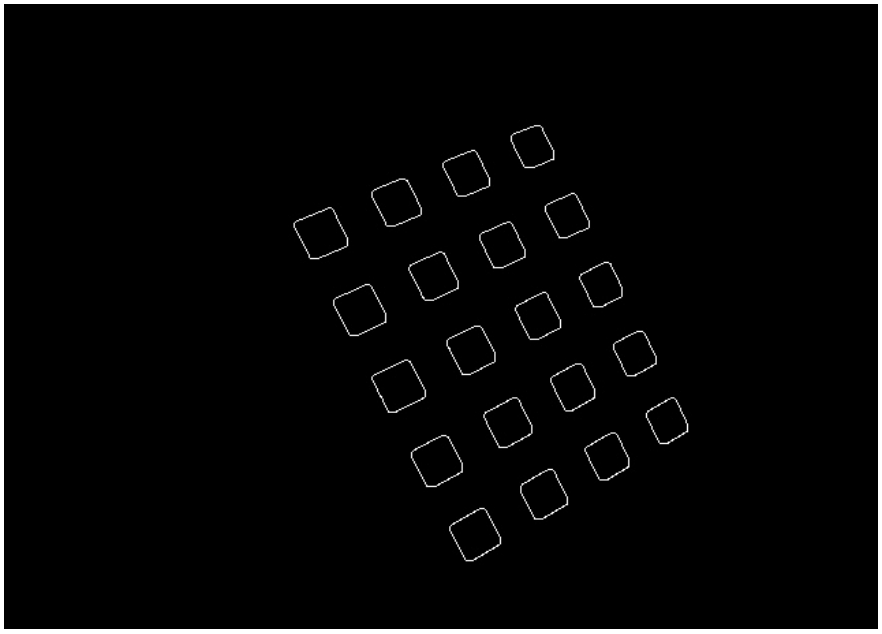

Figure 5: Edge detected image for pattern (from given dataset) in Fig 1.

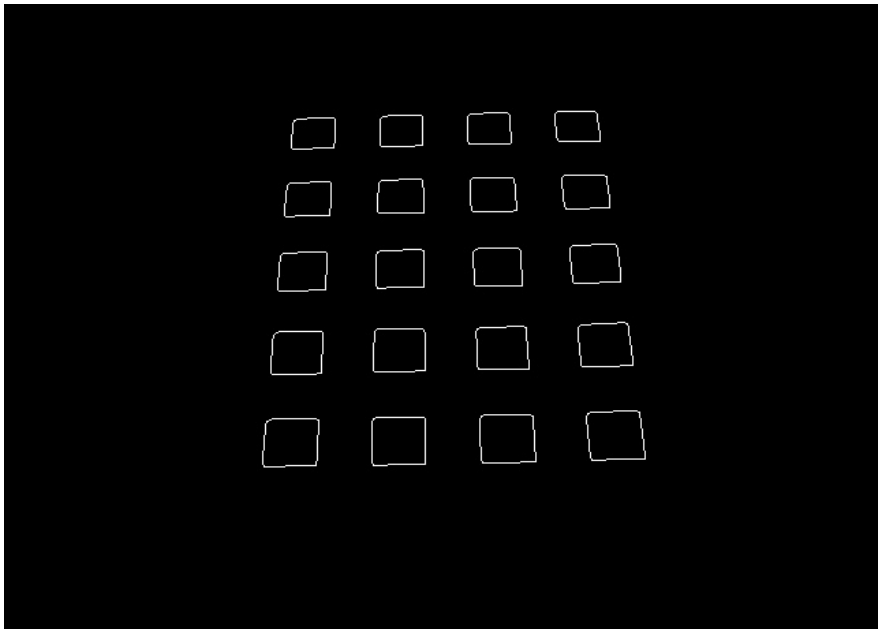

Figure 6: Edge detected image for pattern (from given dataset) in Fig 2.

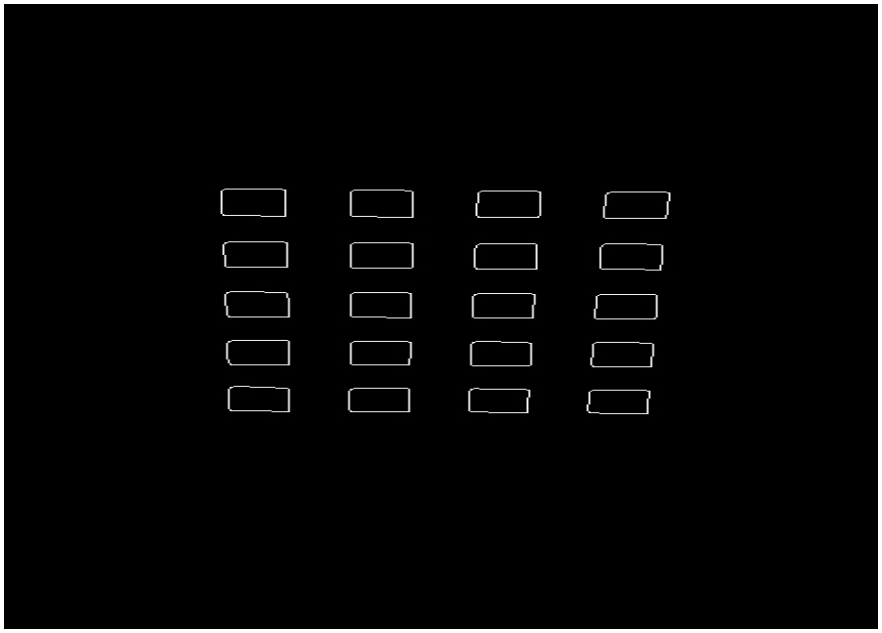

Figure 7: Edge detected image for pattern (from created dataset) in Fig 3.

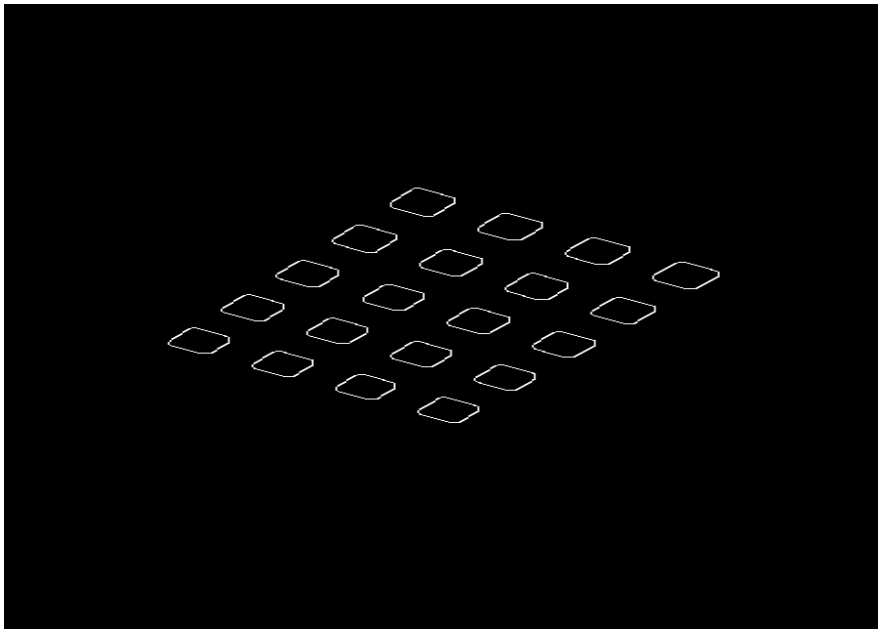

Figure 8: Edge detected image for pattern (from created dataset) in Fig 4.

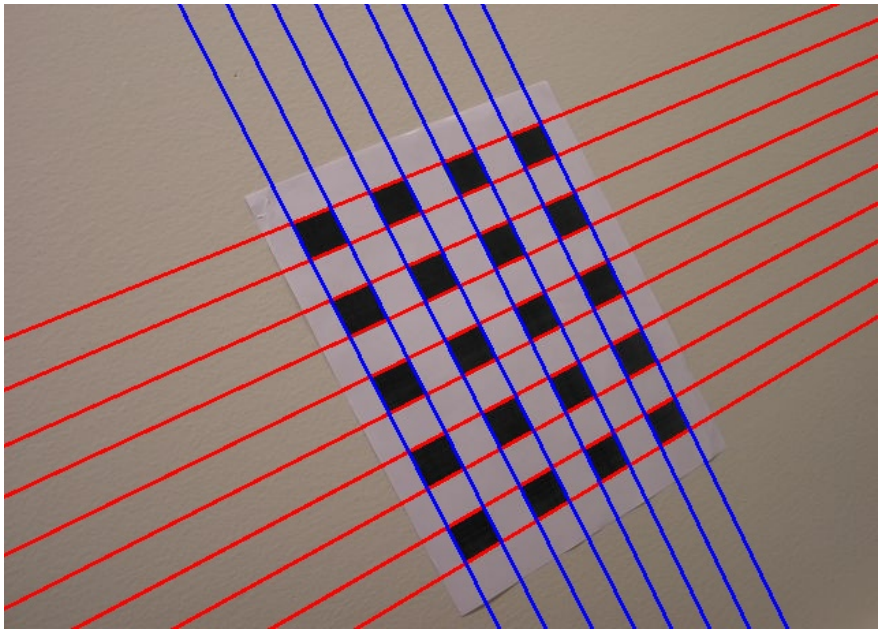

Figure 9: Line detected image for pattern (from given dataset) in Fig 1.

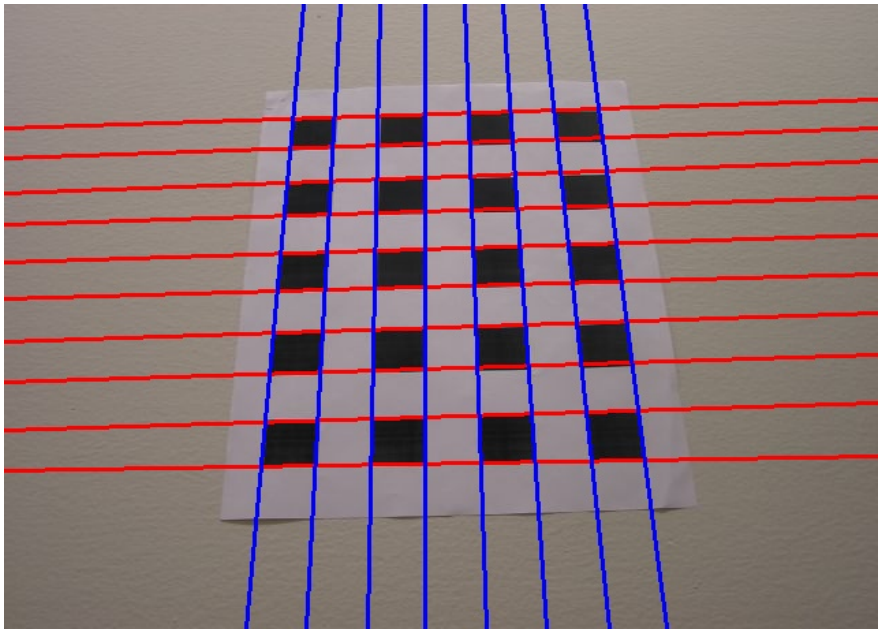

Figure 10: Line detected image for pattern (from given dataset) in Fig 2.

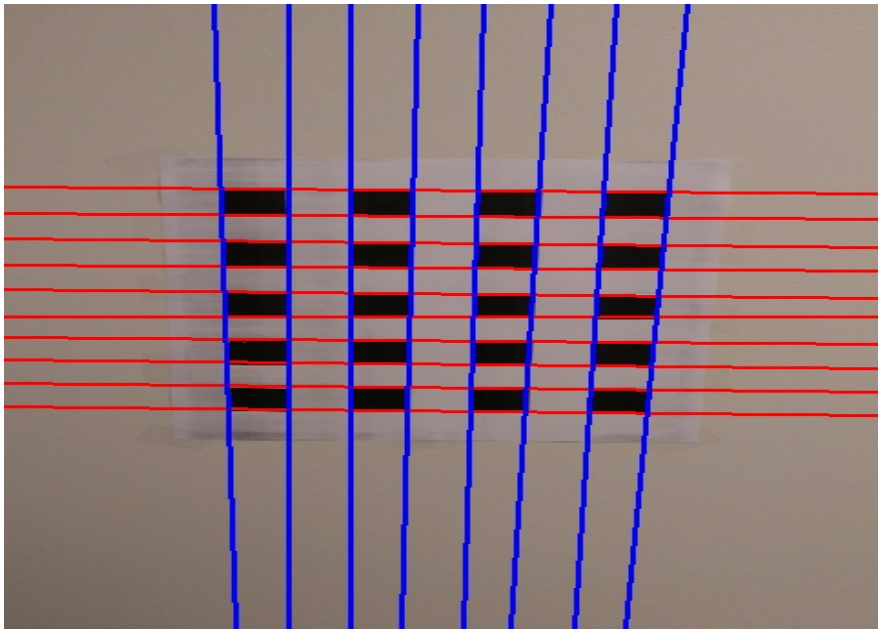

Figure 11: Line detected image for pattern (from created dataset) in Fig 3.

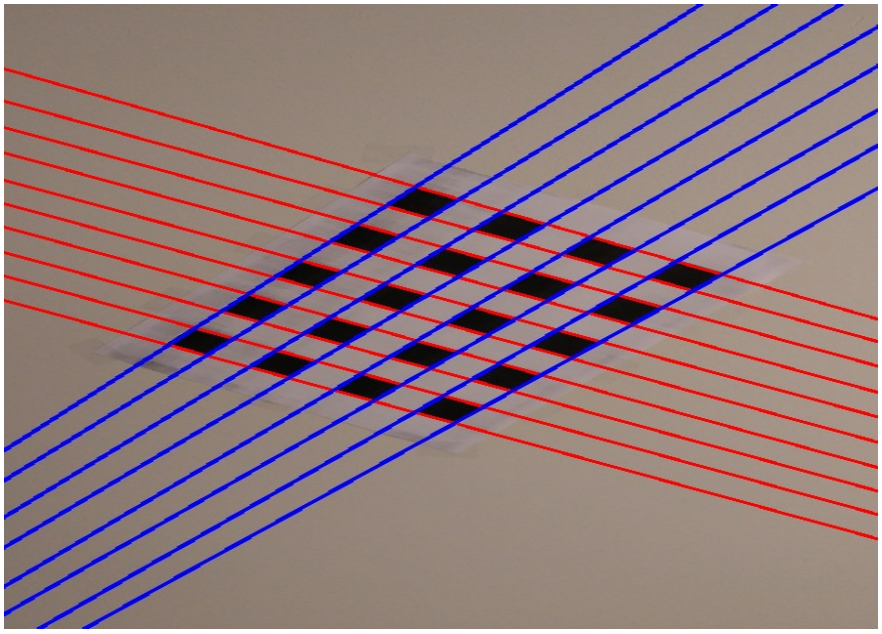

Figure 12: Line detected image for pattern (from created dataset) in Fig 4.

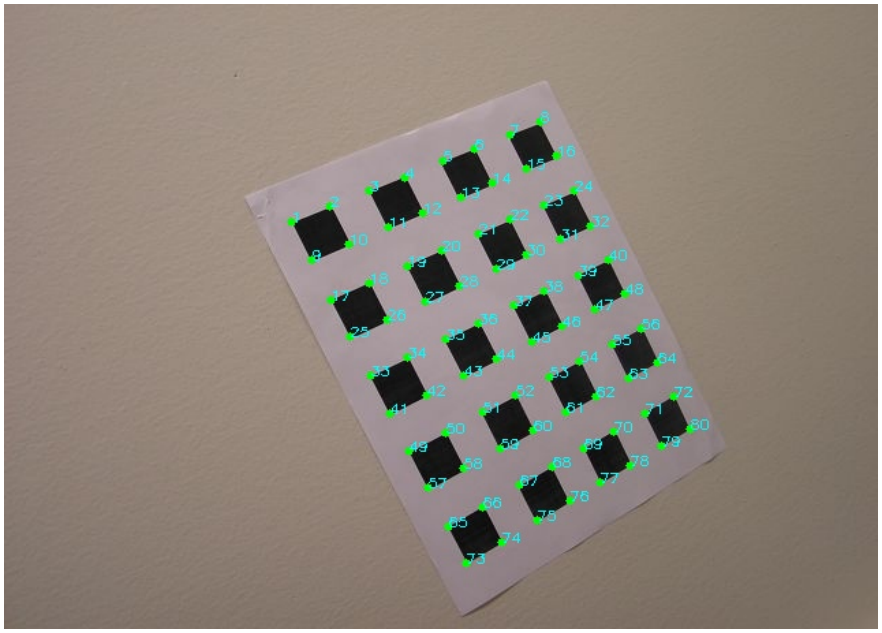

Figure 13: Corner detected (and numbered) image for pattern (from given dataset) in Fig 1.

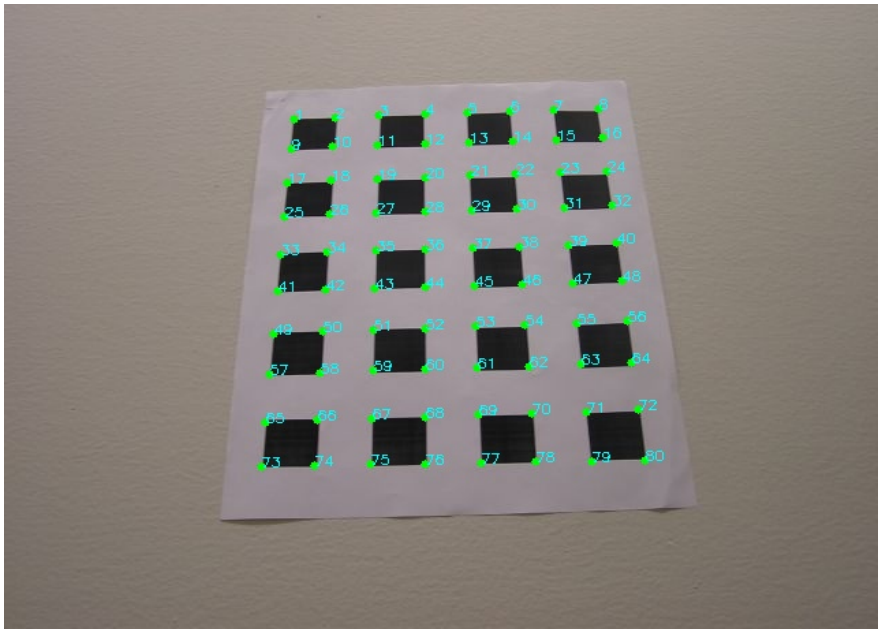

Figure 14: Corner detected (and numbered) image for pattern (from given dataset) in Fig 2.

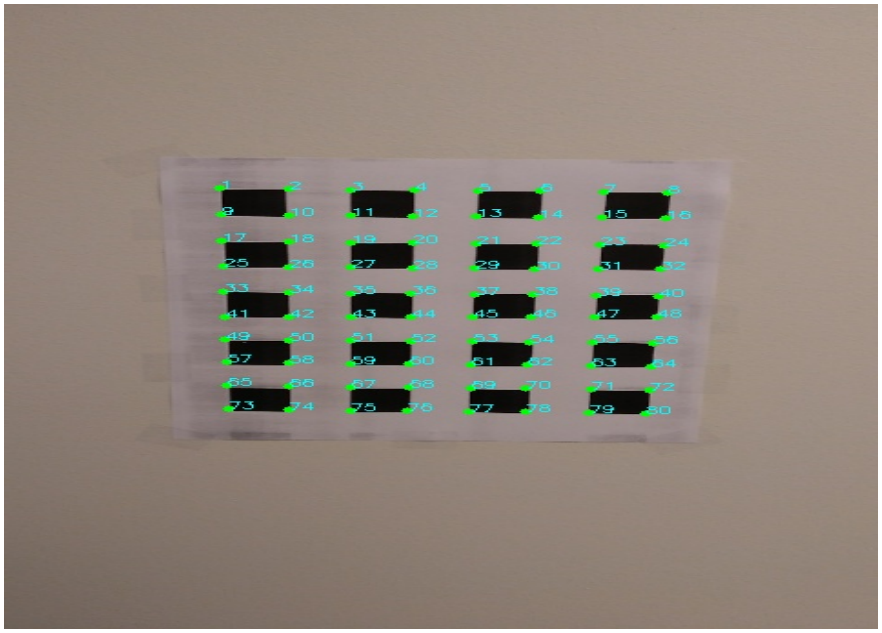

Figure 15: Corner detected (and numbered) image for pattern (from created dataset) in Fig 3.

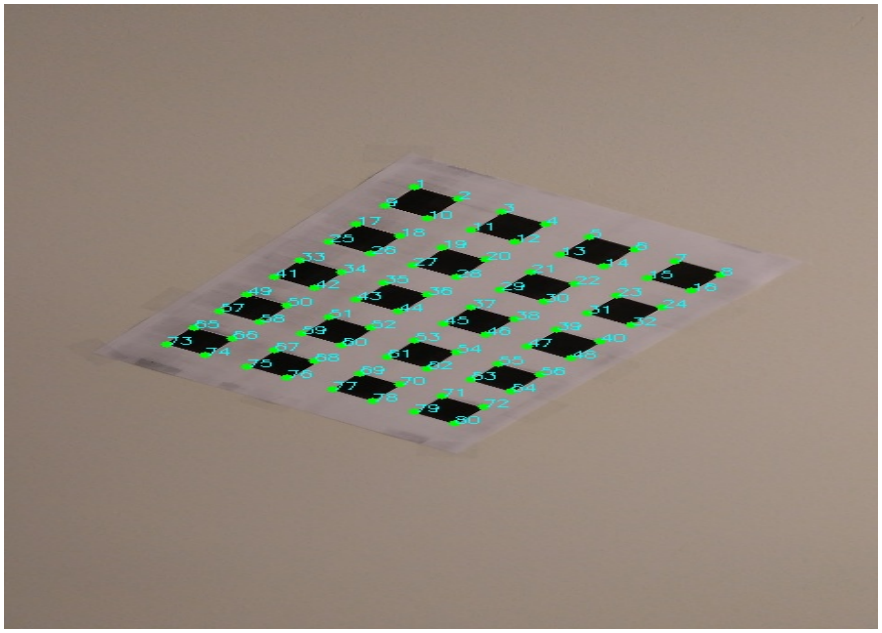

Figure 16: Corner detected (and numbered) image for pattern (from created dataset) in Fig 4.

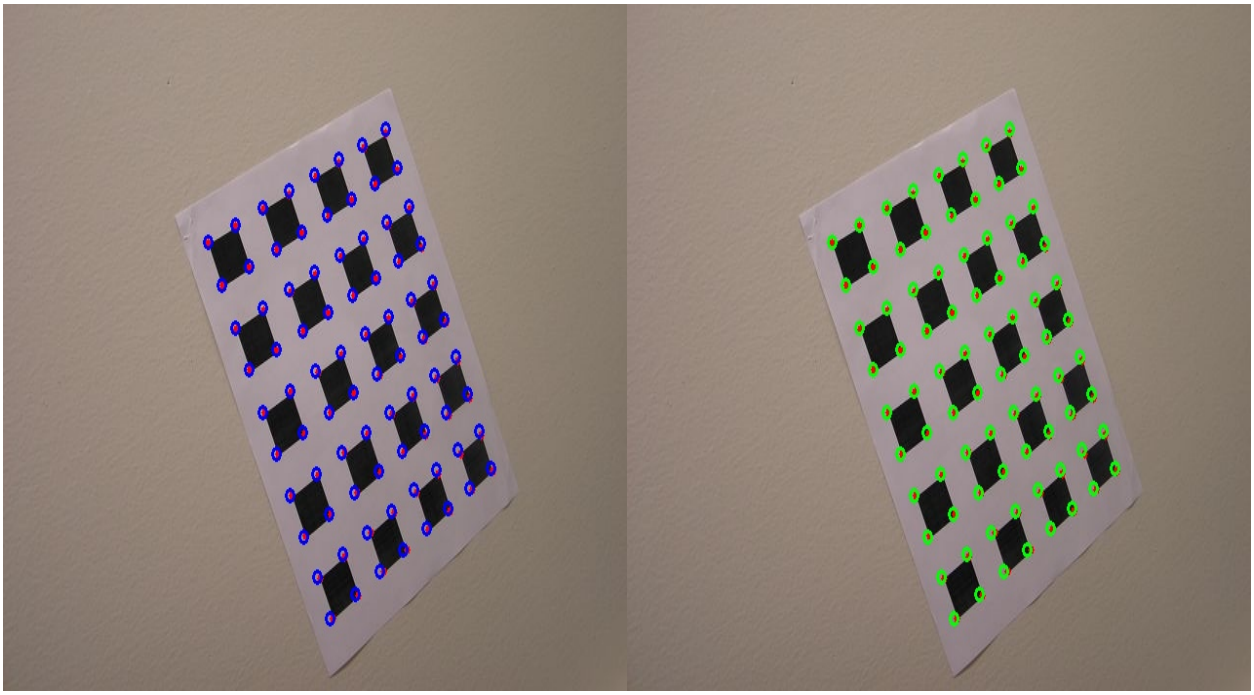

Figure 17: Reprojected corners for a viewpoint pattern (from given dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

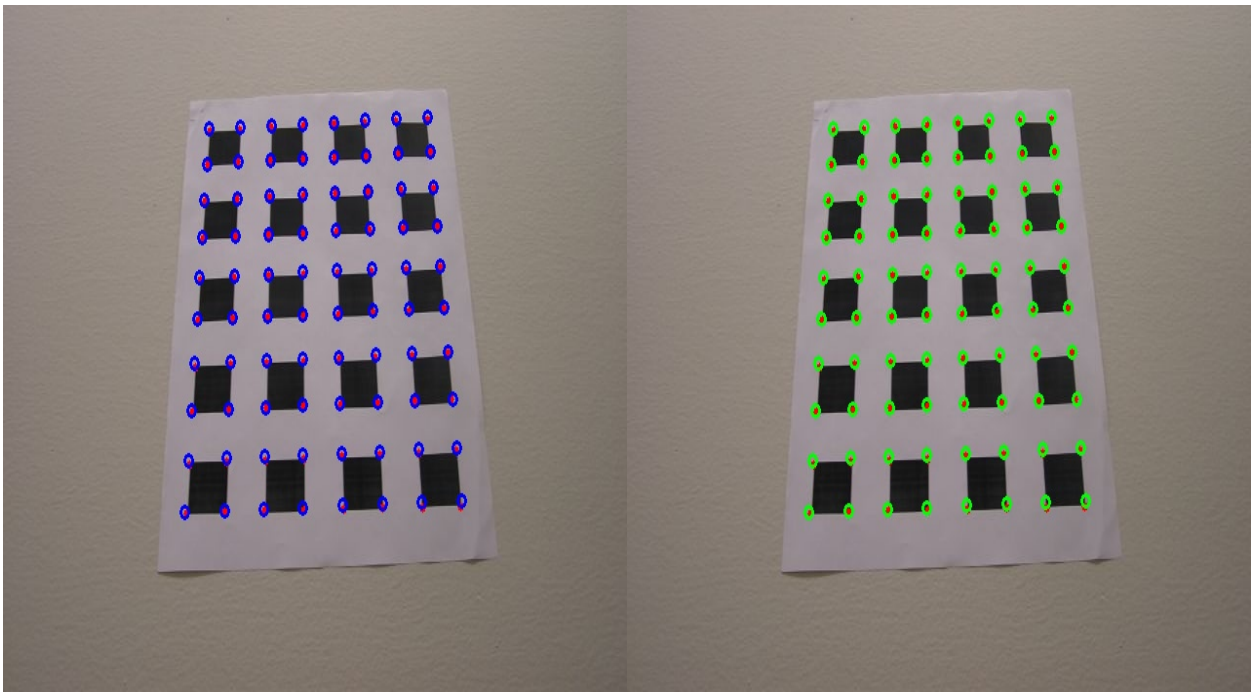

Figure 18: Reprojected corners for a second viewpoint pattern (from given dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

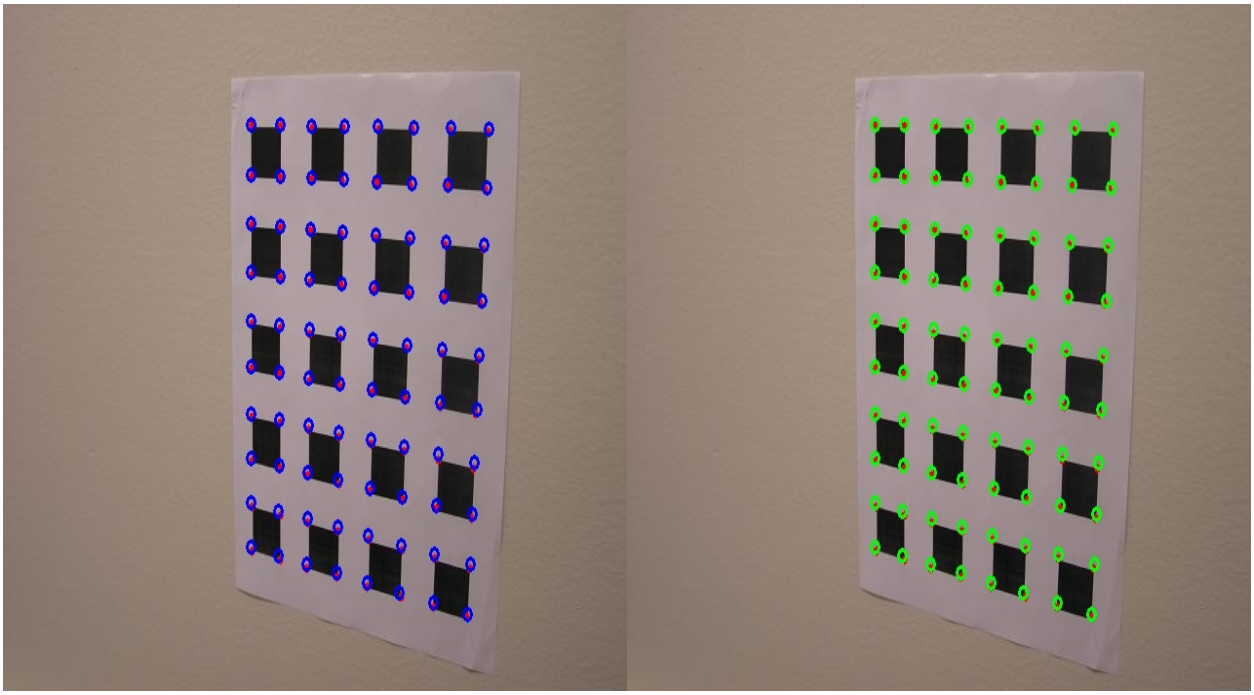

Figure 19: Reprojected corners for a third veiwpoint pattern (from given dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

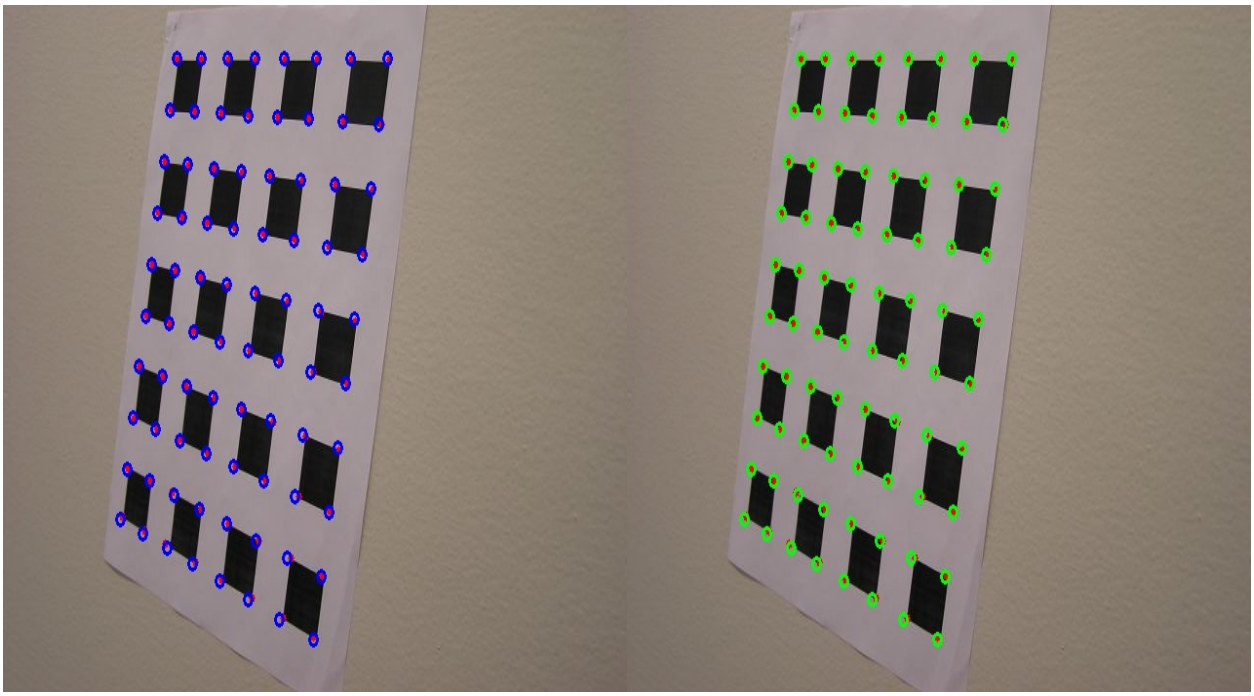

Figure 20: Reprojected corners for a third veiwpoint pattern (from given dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

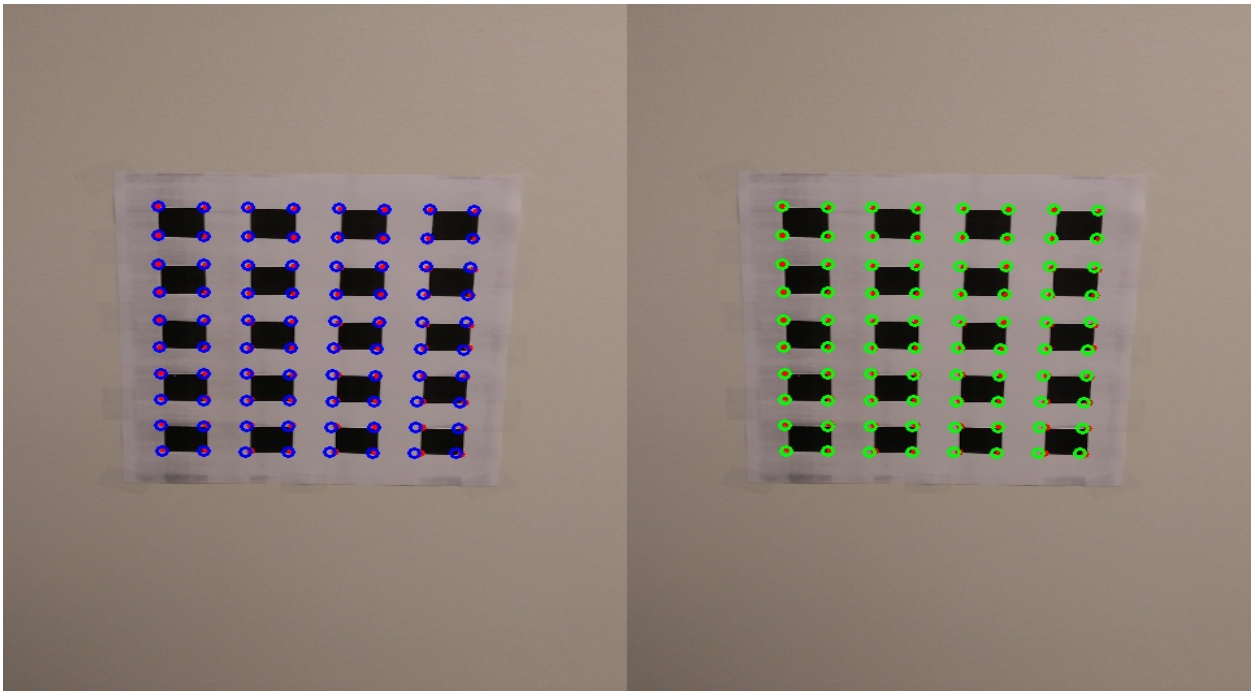

Figure 21: Reprojected corners for a viewpoint pattern (from created dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

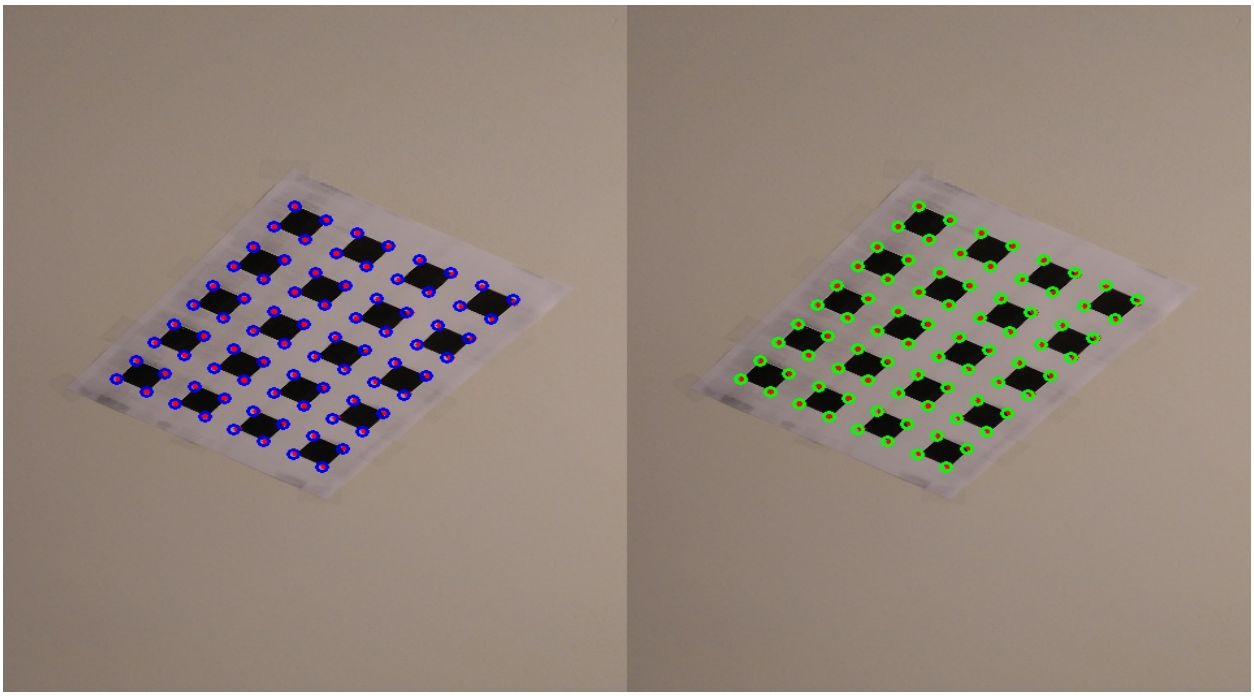

Figure 22: Reprojected corners for a second veiwpoint pattern (from created dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

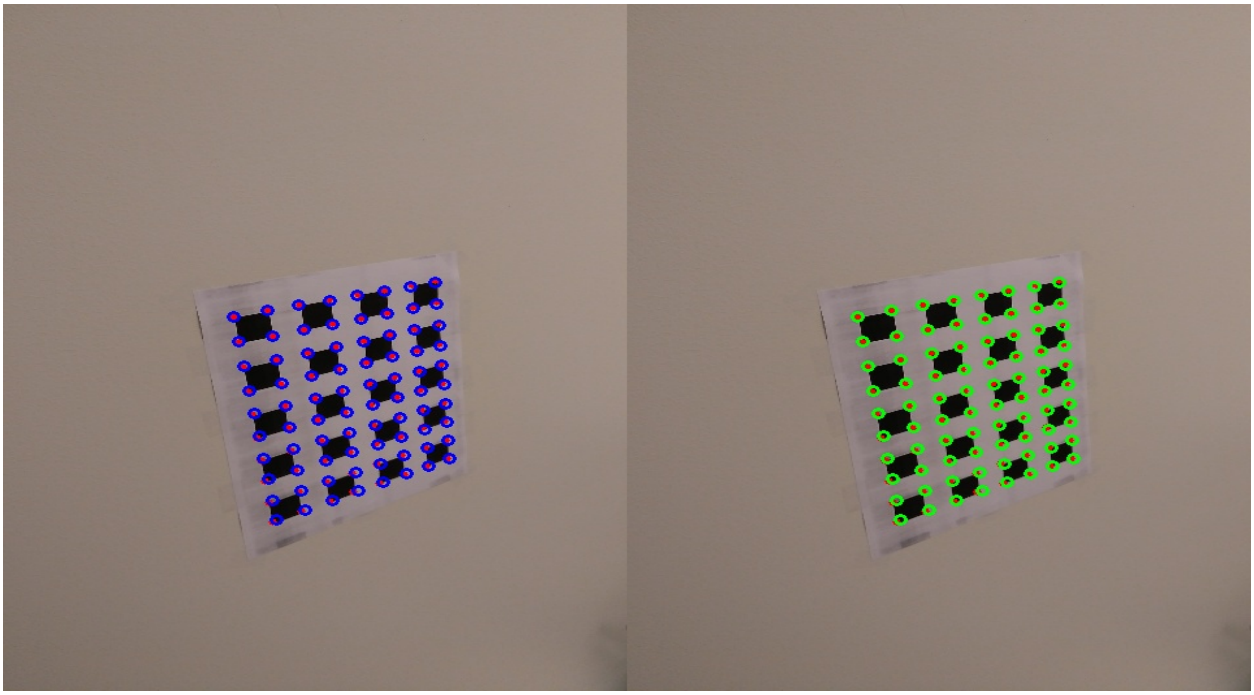

Figure 23: Reprojected corners for a third veiwpoint pattern (from created dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

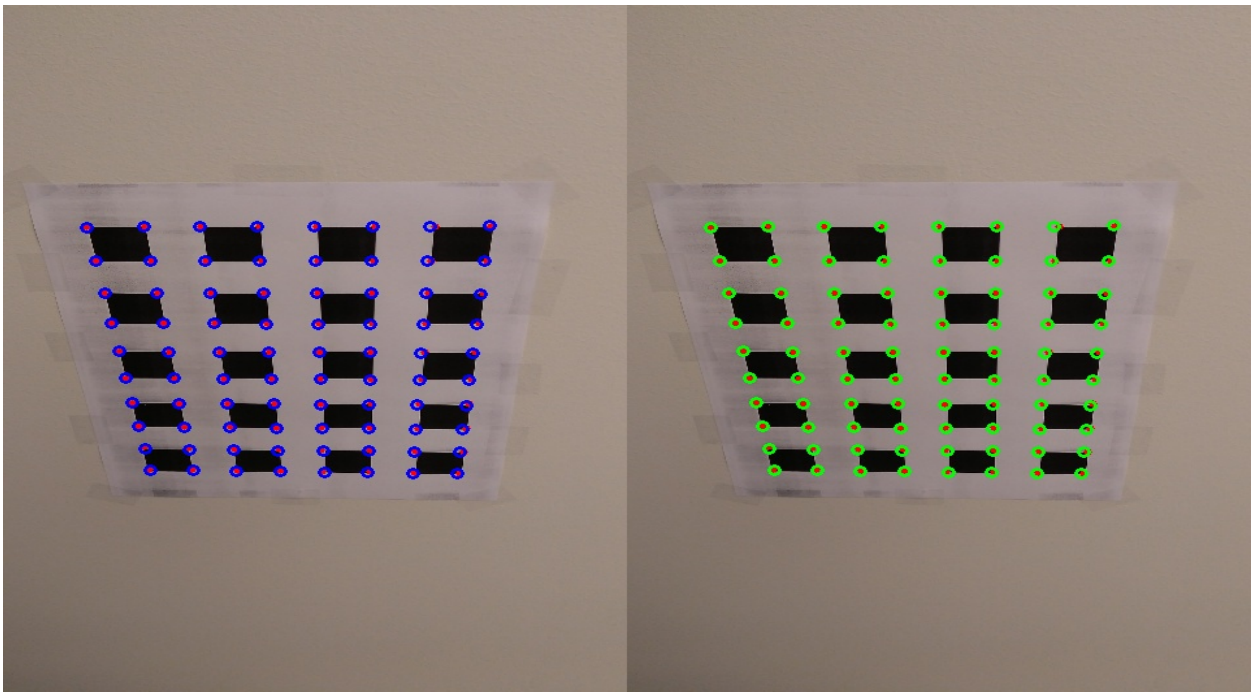

Figure 24: Reprojected corners for a third veiwpoint pattern (from created dataset). Left image: reprojected corners (Blue) without LM algorithm. Right image: reprojected corners (Green) after using LM. True postion of the corners are in Red.

5 Appendix

Let $P = [p_x, p_y]^T$ be a point in planar coordinate of the source image and $P_1 = [p_{x1}, p_{y1}]^T$ be the corresponding planar point in the transformed image. Let H be the Homography matrix that does this transformation. And let P_h and P_{h1} are the homogeneous coordinates of the points P and P_1, then the following equation can be written.

$$P_{h1} = HP_h \tag{25}$$

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{26}$$

Now, in the homogeneous coordinate system,

$$P_h = [p_{hx}, p_{hy}, p_{hw}]^T \tag{27}$$

$$P_{h1} = [p_{hx1}, p_{hy1}, p_{hw1}]^T \tag{28}$$

So the P_{h1} can be converted to its planar form P_1 in the following manner.

$$P_1 = [p_{x1}, p_{y1}]^T = \left[\frac{p_{hx1}}{p_{hw1}}, \frac{p_{hy1}}{p_{hw1}}\right]^T \tag{29}$$

Expanding the equation (26) with the assumption of $h_{33} = 1$:

$$\begin{aligned} p_{hx1} &= h_{11}p_{hx} + h_{12}p_{hy} + h_{13}p_{hw} \\ p_{hy1} &= h_{21}p_{hx} + h_{22}p_{hy} + h_{23}p_{hw} \\ p_{hw1} &= h_{31}p_{hx} + h_{32}p_{hy} + h_{33}p_{hw} \end{aligned} \tag{30}$$

Now, the equations for p_{x1} and p_{y1} can be written with the help of (30) as follows.

$$\begin{aligned} p_{x1} = \frac{p_{hx1}}{p_{hw1}} &= \frac{h_{11}p_{hx} + h_{12}p_{hy} + h_{13}p_{hw}}{h_{31}p_{hx} + h_{32}p_{hy} + h_{33}p_{hw}} \\ p_{y1} = \frac{p_{hy1}}{p_{hw1}} &= \frac{h_{21}p_{hx} + h_{22}p_{hy} + h_{23}p_{hw}}{h_{31}p_{hx} + h_{32}p_{hy} + h_{33}p_{hw}} \end{aligned}$$

Dividing the numerator and dinominator of the right hand side of the equation by p_{hw} we have,

$$\begin{aligned} p_{x1} = \frac{h_{11}\frac{p_{hx}}{p_{hw}} + h_{12}\frac{p_{hy}}{p_{hw}} + h_{13}}{h_{31}\frac{p_{hx}}{p_{hw}} + h_{32}\frac{p_{hy}}{p_{hw}} + h_{33}} &= \frac{h_{11}p_x + h_{12}p_y + h_{13}}{h_{31}p_x + h_{32}p_y + h_{33}} \\ p_{y1} = \frac{h_{21}\frac{p_{hx}}{p_{hw}} + h_{22}\frac{p_{hy}}{p_{hw}} + h_{23}}{h_{31}\frac{p_{hx}}{p_{hw}} + h_{32}\frac{p_{hy}}{p_{hw}} + h_{33}} &= \frac{h_{21}p_x + h_{22}p_y + h_{23}}{h_{31}p_x + h_{32}p_y + h_{33}} \end{aligned}$$

which can be simplified as,

$$\begin{aligned} h_{11}p_x + h_{12}p_y + h_{13} - h_{31}p_xp_{x1} - h_{32}p_yp_{x1} - h_{33}p_{x1} = 0 \\ h_{21}p_x + h_{22}p_y + h_{23} - h_{31}p_xp_{y1} - h_{32}p_yp_{y1} - h_{33}p_{y1} = 0 \end{aligned} \tag{31}$$

Now, in this task, the points $P = [p_x, p_y]^T$ and $P_1 = [p_{x1}, p_{y1}]^T$ are already available from the given images (those have to be picked up manually). So the h elements of the Homography matrix are

the actual unknowns. And there are altogether 8 of them. So, at least 8 equations like those in (31) are needed. For this at least 4 points are needed from the images. So 3 more points Q, R and S are also selected from each of the images. These points will also form the same derivation as above and the 8 equations obtained are

$$\begin{aligned}
h_{11}p_x + h_{12}p_y + h_{13} - h_{31}p_xp_{x1} - h_{32}p_yp_{x1} - h_{33}p_{x1} = 0 \\
h_{21}p_x + h_{22}p_y + h_{23} - h_{31}p_xp_{y1} - h_{32}p_yp_{y1} - h_{33}p_{y1} = 0 \\
h_{11}q_x + h_{12}q_y + h_{13} - h_{31}q_xq_{x1} - h_{32}q_yq_{x1} - h_{33}q_{x1} = 0 \\
h_{21}q_x + h_{22}q_y + h_{23} - h_{31}q_xq_{y1} - h_{32}q_yq_{y1} - h_{33}q_{y1} = 0 \\
h_{11}r_x + h_{12}r_y + h_{13} - h_{31}r_xr_{x1} - h_{32}r_yr_{x1} - h_{33}r_{x1} = 0 \\
h_{21}r_x + h_{22}r_y + h_{23} - h_{31}r_xr_{y1} - h_{32}r_yr_{y1} - h_{33}r_{y1} = 0 \\
h_{11}s_x + h_{12}s_y + h_{13} - h_{31}s_xs_{x1} - h_{32}s_ys_{x1} - h_{33}s_{x1} = 0 \\
h_{21}s_x + h_{22}s_y + h_{23} - h_{31}s_xs_{y1} - h_{32}s_ys_{y1} - h_{33}s_{y1} = 0
\end{aligned} \tag{32}$$

(32) in matrix form is the following

$$\begin{bmatrix}
p_x & p_y & 1 & 0 & 0 & 0 & -p_xp_{x1} & -p_yp_{x1} & -p_{x1} \\
0 & 0 & 0 & p_x & p_y & 1 & -p_xp_{y1} & -p_yp_{y1} & -p_{y1} \\
q_x & q_y & 1 & 0 & 0 & 0 & -q_xq_{x1} & -q_yq_{x1} & -q_{x1} \\
0 & 0 & 0 & q_x & q_y & 1 & -q_xq_{y1} & -q_yq_{y1} & -q_{y1} \\
r_x & r_y & 1 & 0 & 0 & 0 & -r_xr_{x1} & -r_yr_{x1} & -r_{x1} \\
0 & 0 & 0 & r_x & r_y & 1 & -r_xr_{y1} & -r_yr_{y1} & -r_{y1} \\
s_x & s_y & 1 & 0 & 0 & 0 & -s_xs_{x1} & -s_ys_{x1} & -s_{x1} \\
0 & 0 & 0 & s_x & s_y & 1 & -s_xs_{y1} & -s_ys_{y1} & -s_{y1}
\end{bmatrix}
\begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix}
=
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{33}$$

Which can be also represented as

$$A_{8\times 9}h_{9\times 1} = [0]_{8\times 1}$$

Now, if there were more correspondences, then there would have been more equations. So if there were altogether N correspondences, there equations would be like

$$A_{2N\times 9}h_{9\times 1} = [0]_{2N\times 1} \tag{34}$$

So, to find the solution of h, the linear least squares (LLS) method should be used. For this the condition will be to minimize the term Ah, subjected to some constraints on the magnitude of h, otherwise the LLS will return a h = 0 trivial solution which is not useful at all. So we apply the constraint $||h|| = 1$ to prevent having the trivial solution.

Now, to solve this Singular Value Decomposition (SVD) is used.

By SVD, $A_{m\times n} = U_{m\times m}D_{m\times n}V^T_{n\times n}$ where U and V are orthonormal matrices, (so $UU^T = U^TU = I$ and $VV^T = V^TV = I$) and D is a diagonal matrix having the singular values along the diagonal in decreasing order of their magnitudes.

Now, since U and V are orthonormal, so it will also preserve norms, i.e. $||Ux|| = ||x||$ and same is true for V as well.

Also, let $V^Th = y$ or $h = V^{-T}y = Vy$, since $V^{-1} = V^T$ (for orthonormality).

So, $||h||^2 = h^Th = y^TV^TVy = y^T(V^TV)y = y^TIy = y^Ty = ||y||^2$.

So, it can be written that $||h|| = ||y||$.

These results can be used to derive the following,
$||Ah|| = ||UDV^T h|| = ||DV^T h|| = ||DV^T h|| = ||Dy||$.
So, mimimizing Ah with the constraint of $||h|| = 1$ implies mimimizing Dy with the constraint of $||y|| = 1$.
Finding this is trivial, as this corresponds to the smallest singular value in D. So the solution to the above equation is $y_{n\times 1} = [0, 0, 0, 0, 0, ...0, 0, 1]^T_{n\times 1}$. So, since $h = Vy$ (derived earlier), h is equal to the last column vector of V matrix.

This gives the required solution for h.

```
#!/usr/bin/env python

import numpy as np, cv2, os, time, math, copy, matplotlib.pyplot as plt
from scipy import signal, optimize


#================================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 8
#================================================================================

#================================================================================
# FUNCTIONS CREATED IN HW8.
#================================================================================

def rodriguesRtoW( R ):
    '''
    Converting R to vector form w.
    '''
    # Before taking the arccos of a value, care should be taken to check whether
    # that value is less than 1 or not.
    phi = math.acos( ( np.trace(R) - 1 ) * 0.5  )

    w = ( 0.5 * phi / math.sin( phi ) ) * np.array( [ [ R[2,1]-R[1,2] ], \
                                                      [ R[0,2]-R[2,0] ], \
                                                      [ R[1,0]-R[0,1] ] ] )
    return w

#================================================================================

def rodriguesWtoR( w ):
    '''
    Converting w to matrix form R.
    '''
    # Before taking the arccos of a value, care should be taken to check whether
    # that value is less than 1 or not.
    phi = np.linalg.norm( w )

    # Converting w to skew symmetrix form.
    wx = np.array( [ [ 0, -w[2], w[1] ], [ w[2], 0, -w[0] ], [ -w[1], w[0], 0 ] ] )

    R = np.eye(3) + ( math.sin( phi ) / phi ) * wx + \
                ( ( 1 - math.cos( phi ) ) / ( phi * phi ) ) * np.matmul( wx, wx )

    return R

#================================================================================

def refineParamByLM( setOfXij, XWij, K, setOfR, setOfT, k1=None, k2=None ):
    '''
    This function takes in the Xij location of the corners in all the patterns,
    and also the XWij world locations of those corners,
    (The Xij, XW are in planar coordinates. So the XW vector will be
    equal to [xw, yw, 0] transpose (as zw is considered 0 for all the patterns.
    But before calculating the error, those have to be converted to homogeneous
    form) along with R, K, t and then calculates the error between the true Xij and
    the XijEst esimated using the calibration parameters. Then the XijEst (is
    converted back to planar form and) is compared to the Xij and the error is
    calculated. This error is minimized using Levenberg-Marqdth algorithm.
    k1 and k2 are the radial distortion parameters. So if they are not None, then
    that means some valid initialized values are given for these. Hence they are
    also optimized.
    '''
    XWij = [ [ pt[0], pt[1], 0, 1 ] for pt in XWij ]     # Convert to homogeneous form.
```

```
    # zw is 0 for all the corners as the pattern is assumed to be in z=0 plane.

    alphax, s, x0, alphay, y0 = K[0,0], K[0,1], K[0,2], K[1,1], K[1,2]

    # The params list will hold the 5 parameters of K and all the 6 parameters for all
    # the R's and t's (together) for all the patterns.
    # So, length of params is 5 + 3*n + 3*n (n = number of patterns).
    params = [ alphax, s, x0, alphay, y0 ]

    # rectifyRadialDistortion is a flag that indicates if the radial distortion
    # has to be rectified or not. This is true if some valid values of k1 and k2
    # are provided.

    if k1 is not None and k2 is not None:   rectifyRadialDistortion = True
    else:       rectifyRadialDistortion = False

    if rectifyRadialDistortion:     params += [ k1, k2 ]    # Include k1 and k2.

    for i in range( len( setOfXij ) ):
        # Appending the elements of R and t in params
        R, t = setOfR[i], setOfT[i]
        w = rodriguesRtoW( R )
        params += [ w[0], w[1], w[2], t[0], t[1], t[2] ]

    params = np.array( params )

#-------------------------------------------------------------------------------

    def errorFunction( params ):
        '''
        This function takes in the parameters and using XWij, finds the
        estimated values of Xij (XijEst). It then compares them with the actual Xij
        values (which are the location of the corners of the pattern) and outputs
        an error.
        '''
        alphax_1, s_1, x0_1, alphay_1, y0_1 = \
                        params[0], params[1], params[2], params[3], params[4]
        K_1 = np.array( [ [ alphax_1, s_1, x0_1 ], [ 0, alphay_1, y0_1 ], [ 0, 0, 1 ] ] )

        if rectifyRadialDistortion:
            iStart = 7
            k1_1, k2_1 = params[5], params[6]
        else:
            iStart = 5

        for i in range( iStart, len( params ), 6 ):
            R_1 = rodriguesWtoR( [ params[i], params[i+1], params[i+2] ] )
            t_1 = np.array( [ [ params[i+3] ], [ params[i+4] ], [ params[i+5] ] ] )

            P_1 = np.matmul( K_1, np.hstack( ( R_1, t_1 ) ) )

            # Suppose number of corners is 80.
            XWij_1 = np.array( XWij ).T     # Shape 80x4 transposed to 4x80.
            XijEstH = np.matmul( P_1, XWij_1 ).T    # Shape 3x80 transposed to 80x3.

            # Convert to planar.
            for j in range( XijEstH.shape[0] ):     XijEstH[j] /= XijEstH[j][2]

            XijEst = XijEstH[ :, :-1 ]    # Shape now 80x2. Last column removed. Planar form.

            # Index is reconfigured for list setOfXij.
            Xij_1 = setOfXij[ int( ( i - iStart ) / 6 ) ]

            Xij_1 = np.array( Xij_1 )    # Shape 80x2.
```

```
            #print(Xij_1.shape, XijEst.shape)

            # If rectifyRadialDistortion flag is True, then optimize the radial
            # distortion parameters as well. So incorporate them in the error function.
            if rectifyRadialDistortion:
                x, y = XijEst[:, 0], XijEst[:, 1]
                rSq = (x - x0_1)**2 + (y - y0_1)**2
                xRad = x + (x - x0_1) * ( k1_1*rSq + k2_1*rSq**2 )
                yRad = y + (y - y0_1) * ( k1_1*rSq + k2_1*rSq**2 )

                # Remember that the xRad and yRad are all 80x1 vectors.
                XijEst[:, 0] = xRad
                XijEst[:, 1] = yRad

            error = Xij_1 - XijEst
            error = np.linalg.norm( error, axis=1 )     # Taking norm of error. Shape 80x1.

            # This will hold all the errors for all the 80 corners of 40 images.
            # Total 3200 elements.
            errorVector = error if i == iStart else np.hstack( ( errorVector, error ) )

            # LM needs the number of residuals (i.e. the size of error vector) to be
            # more or equal to the size of the variables (params) which are optimized.
            # Hence the error is reshaped the size of the params list (as in this case
            # for the 80 corners, we had error as 80x1 but total number of parameters
            # for 40 images was 5 (intrinsic) + 40 * 6 (extrinsic) = 245, or
            # 7 (intrinsic) + 40 * 6 (extrinsic) = 247 (if k1 and k2 of radial
            # distortion is considered)).

        #print(errorVector.shape)

        return errorVector

#--------------------------------------------------------------------------------

    # Optimized output.
    startTime = time.time()
    paramsNew = optimize.least_squares( errorFunction, params, method='lm' )
    print( f'Time taken by LM optimizer: {time.time() - startTime} sec.' )

    # The optimized vector is obtained as paramsNew.x.
    paramsNew = paramsNew.x
    #print( paramsNew )

    alphaxNew, sNew, x0new, alphayNew, y0new = \
                    paramsNew[0], paramsNew[1], paramsNew[2], paramsNew[3], paramsNew[4]
    Knew = np.array( [ [ alphaxNew, sNew, x0new ], [ 0, alphayNew, y0new ], [ 0, 0, 1 ] ] )

    if rectifyRadialDistortion:
        iStart = 7
        k1new, k2new = paramsNew[5], paramsNew[6]
    else:
        iStart = 5
        k1new, k2new = k1, k2

    setOfRnew, setOfTnew = [], []
    for i in range( iStart, len( paramsNew ), 6 ):
        Rnew = rodriguesWtoR( [ paramsNew[i], paramsNew[i+1], paramsNew[i+2] ] )
        tNew = np.array( [ [ paramsNew[i+3] ], [ paramsNew[i+4] ], [ paramsNew[i+5] ] ] )

        setOfRnew.append( Rnew )
        setOfTnew.append( tNew )

    return Knew, setOfRnew, setOfTnew, k1new, k2new
```

```
#================================================================================

if __name__ == '__main__':

    # TASK: 2.2.1. Finding the corners.

    # Loading the images of given dataset.

    filepath = './Files/Dataset1'
    outputFilepath = './output_images'

    #filepath = './own_dataset'
    #outputFilepath = './output_image_own_dataset'

    listOfFiles = os.listdir( filepath )
    listOfFiles.sort()
    nPatterns = len( listOfFiles )

    # This is the V matrix for calculating the omega matrix.
    # And the homography (H) for every image is also stored in a separate list
    # as those will be used to calculate the R and t for every positon of the
    # calibration pattern.
    # The intersection point lists for all the different pattern images are also
    # stored in a separate list.
    V, setOfH, setOfIntSctPtList = [], [], []

    for idx, i in enumerate( listOfFiles ):

        print( i )

        img = cv2.imread( os.path.join( filepath, i ) )    # Read images
        imgH, imgW, _ = img.shape

        gray = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )      # Convert to grayscale.

        blur = cv2.GaussianBlur( gray, (5,5), 0 )     # Blurring the image.

        # Find otsu's threshold value with.
        threshVal, otsu = cv2.threshold( blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU )

        #print( f'Threshold for color filter: {threshVal}' )

        # Making the white regions dilated and eroded, so that the unwanted
        # black zones in the otsu image disappear.
        structElem = cv2.getStructuringElement( cv2.MORPH_RECT, ksize=( 5, 5 ) )
        dilated = cv2.dilate( otsu, kernel=structElem, anchor=(-1,-1) )
        eroded = cv2.erode( dilated, kernel=structElem, anchor=(-1,-1) )

        # Finding the edges.
        edgeThresh = 10
        edges = cv2.Canny( eroded, threshold1=edgeThresh, threshold2=3*edgeThresh, \
                                                            apertureSize=3 )

        ## Saving edge detected image.
        #cv2.imwrite( os.path.join( outputFilepath, i[:-4] + '_edges' + i[-4:] ), edges )

        # Finding the hough lines.
        # maxLineGap parameter seems to be the most important to merge redundant lines.
        lines = cv2.HoughLinesP( edges, rho=1, theta=math.pi/180, threshold=30, \
                                                minLineLength=15, maxLineGap=250 )

        output = copy.deepcopy( img )

        nLines = lines.shape[0]     # No. of detected lines.
        #print( lines.shape )
```

```
#--------------------------------------------------------------------------

        # Lists to save the vertical lines and horizontal lines and points.
        horiLinesPts, vertLinesPts, horiLinesH, vertLinesH = [], [], [], []

        # Draw the lines on the image.
        for l in lines:
            # Extending the x, y values of the end-points of the lines.
            x1, y1, x2, y2 = l[0,0], l[0,1], l[0,2], l[0,3]

            # Points in homogeneous coordinates.
            X1H, X2H = np.array( [x1,y1,1] ), np.array( [x2,y2,1] )
            LH = np.cross( X1H, X2H )   # Line in homogeneous coordinates.

            # Four corners of the image in homogeneous coordinates.
            tlH, trH = np.array( [0,0,1] ), np.array( [imgW-1,0,1] )
            blH, brH = np.array( [0,imgH-1,1] ), np.array( [imgW-1,imgH-1,1] )

            # Four boundaries of the image in homogeneous coordinates.
            topBoundaryH = np.cross( tlH, trH )
            botBoundaryH = np.cross( blH, brH )
            leftBoundaryH = np.cross( tlH, blH )
            rightBoundaryH = np.cross( trH, brH )
            #print( topBoundaryH, botBoundaryH, leftBoundaryH, rightBoundaryH )

            slopeAngle = math.atan2( y2-y1, x2-x1 )
            slopeAngleDeg = slopeAngle * 180 / math.pi

#--------------------------------------------------------------------------

            # We have to extend the line segments to the boundary of the image.
            # So to find the points that they will intersect on the boundary when
            # extended we do the following.
            if abs( slopeAngleDeg ) < 45:
                # If the slope is less than 45 deg, the line will intersect better
                # with the left and right boundary.

                # Finding the intersection point of the line with the left and right
                # boundary in homogeneous coordinates.

                intsctPtOfLwithLeftBound = np.cross( LH, leftBoundaryH )
                # Now converting the intersection point to the planar coordinates.
                x1new = intsctPtOfLwithLeftBound[0] / intsctPtOfLwithLeftBound[2]
                y1new = intsctPtOfLwithLeftBound[1] / intsctPtOfLwithLeftBound[2]

                #print(intsctPtOfLwithLeftBound, x1new, y1new, slopeAngle, x1, y1, x2, y2)

                intsctPtOfLwithRightBound = np.cross( LH, rightBoundaryH )
                # Now converting the intersection point to the planar coordinates.
                x2new = intsctPtOfLwithRightBound[0] / intsctPtOfLwithRightBound[2]
                y2new = intsctPtOfLwithRightBound[1] / intsctPtOfLwithRightBound[2]

                x1new, y1new, x2new, y2new = int(x1new), int(y1new), int(x2new), int(y2new)
                #cv2.line( output, (x1new, y1new), (x2new, y2new), (0,255,255), 1 )

                horiLinesPts.append( [x1new, y1new, x2new, y2new] )
                horiLinesH.append( LH.tolist() )

#--------------------------------------------------------------------------

            else:
                # And if the slope is more than 45 deg, the line may intersect
                # better with the top and bottom boundary.
```

```
                # Finding the intersection point of the line with the top and bot
                # boundary in homogeneous coordinates.

                intsctPtOfLwithTopBound = np.cross( LH, topBoundaryH )
                # Now converting the intersection point to the planar coordinates.
                x1new = intsctPtOfLwithTopBound[0] / intsctPtOfLwithTopBound[2]
                y1new = intsctPtOfLwithTopBound[1] / intsctPtOfLwithTopBound[2]

                intsctPtOfLwithBotBound = np.cross( LH, botBoundaryH )
                # Now converting the intersection point to the planar coordinates.
                x2new = intsctPtOfLwithBotBound[0] / intsctPtOfLwithBotBound[2]
                y2new = intsctPtOfLwithBotBound[1] / intsctPtOfLwithBotBound[2]

                x1new, y1new, x2new, y2new = int(x1new), int(y1new), int(x2new), int(y2new)
                #cv2.line( output, (x1new, y1new), (x2new, y2new), (255,255,0), 1 )

                vertLinesPts.append( [x1new, y1new, x2new, y2new] )
                vertLinesH.append( LH.tolist() )

#-------------------------------------------------------------------------------

        # Sort the horizontal lines according to the y intercepts.
        horiLinesPtsSorted = sorted( horiLinesPts, key=lambda x: x[1] )
        # Also sorting the list of lines in homogeneous coordinates in the same manner.
        horiLinesHsorted = sorted( horiLinesH, key=lambda x: horiLinesPts[ horiLinesH.index(x) ][1] )

        horiLinesPts = horiLinesPtsSorted
        horiLinesH = horiLinesHsorted

        nHoriLines = len( horiLinesPts )
        #print( horiLinesPts )
        horiLinesPtsArr = np.array( horiLinesPts )

        # This array will have the values of the gap between the horizontal lines.
        horiLinesGaps = horiLinesPtsArr[ 1 : ] - horiLinesPtsArr[ 0 : nHoriLines-1 ]
        avgHoriLinesGap = np.mean( horiLinesGaps, axis=0 )

        # This array contains a record of which pair of consecutive lines has a
        # gap less than half of the average gap.
        # If line 2 and 3 has low gap, then the record 2 will be all false.
        # If lines 2, 3 and 4 has low gap between them, then record 2 and 3 will be
        # all false.
        validHoriGaps = horiLinesGaps > avgHoriLinesGap * 0.5

        #print( validHoriGaps )

        # Will hold horiLinesPts after removing redundant lines.
        # The loop runs till nHoriLines-1 as there are 1 less gaps than the number
        # of lines.
        pureHoriLinesPts = [ horiLinesPts[j] for j in range( nHoriLines-1 ) \
                                             if validHoriGaps[j][1] and validHoriGaps[j][3] ]
        pureHoriLinesH = [ horiLinesH[j] for j in range( nHoriLines-1 ) \
                                         if validHoriGaps[j][1] and validHoriGaps[j][3] ]
        # So long as the records corresponding to both the y intercepts are not True,
        # (both y intercepts meaning element 1 and 3 in the list)
        # (which implies that the gap of the corresponding line with the next was
        # smaller than the average gap), igonre it and don't include it to the
        # list of pureHoriLinesPts.

        # Include the last line which was not considered in the loop.
        pureHoriLinesPts.append( horiLinesPts[ nHoriLines - 1 ] )
        pureHoriLinesH.append( horiLinesH[ nHoriLines - 1 ] )

        print( f'Number of horizontal lines: {len(pureHoriLinesPts)}' )
```

```
        for j in pureHoriLinesPts:
            x1, y1, x2, y2 = j[0], j[1], j[2], j[3]
            cv2.line( output, (x1, y1), (x2, y2), (0,0,255), 2 )

#-------------------------------------------------------------------------------

        # Sort the vertzontal lines according to the y intercepts.
        vertLinesPtsSorted = sorted( vertLinesPts, key=lambda x: x[0] )
        # Also sorting the list of lines in homogeneous coordinates in the same manner.
        vertLinesHsorted = sorted( vertLinesH, key=lambda x: vertLinesPts[ vertLinesH.index(x) ][0] )

        vertLinesPts = vertLinesPtsSorted
        vertLinesH = vertLinesHsorted

        nVertLines = len( vertLinesPts )
        #print( vertLinesPts )
        vertLinesPtsArr = np.array( vertLinesPts )

        # This array will have the values of the gap between the vertzontal lines.
        vertLinesGaps = vertLinesPtsArr[ 1 : ] - vertLinesPtsArr[ 0 : nVertLines-1 ]
        avgVertLinesGap = np.mean( vertLinesGaps, axis=0 )

        # This array contains a record of which pair of consecutive lines has a
        # gap less than half of the average gap.
        # If line 2 and 3 has low gap, then the record 2 will be all false.
        # If lines 2, 3 and 4 has low gap between them, then record 2 and 3 will be
        # all false.
        validVertGaps = vertLinesGaps > avgVertLinesGap * 0.5

        #print( validVertGaps )

        # Will hold vertLinesPts after removing redundant lines.
        # The loop runs till nVertLines-1 as there are 1 less gaps than the number
        # of lines.
        pureVertLinesPts = [ vertLinesPts[j] for j in range( nVertLines-1 ) \
                                             if validVertGaps[j][0] and validVertGaps[j][2] ]
        pureVertLinesH = [ vertLinesH[j] for j in range( nVertLines-1 ) \
                                         if validVertGaps[j][0] and validVertGaps[j][2] ]
        # So long as the records corresponding to both the y intercepts are not True,
        # (both y intercepts meaning element 0 and 2 in the list)
        # (which implies that the gap of the corresponding line with the next was
        # smaller than the average gap), igonre it and don't include it to the
        # list of pureVertLinesPts.

        # Include the last line which was not considered in the loop.
        pureVertLinesPts.append( vertLinesPts[ nVertLines - 1 ] )
        pureVertLinesH.append( vertLinesH[ nVertLines - 1 ] )

        print( f'Number of vertical lines: {len(pureVertLinesPts)}' )

        for j in pureVertLinesPts:
            x1, y1, x2, y2 = j[0], j[1], j[2], j[3]
            cv2.line( output, (x1, y1), (x2, y2), (255,0,0), 2 )

        ## Saving line detected image.
        #cv2.imwrite( os.path.join( outputFilepath, i[:-4] + '_lines' + i[-4:] ), output )

#-------------------------------------------------------------------------------

        # Now finding the intersection points.
        corners = copy.deepcopy( img )
        intsctPtIdx, intsctPtList = 0, []       # List and index of the intersection points.
        for hl in pureHoriLinesH:
            for vl in pureVertLinesH:
                intsctPtIdx += 1
```

```
                ptH = np.cross( hl, vl )    # Intersection point in homogeneous coordinates.
                pt = [ int( ptH[0] / ptH[2] ), int( ptH[1] / ptH[2] ) ]     # Planar form.
                intsctPtList.append( pt )
                cv2.circle( corners, tuple(pt), 3, (0,255,0), -1 )      # Draw corners.
                cv2.putText( corners, str(intsctPtIdx), tuple(pt), cv2.FONT_HERSHEY_SIMPLEX, \
                                                                0.35, (255, 255, 0), 1 )

        setOfIntSctPtList.append( intsctPtList )    # Storing the intersection point lists.

        nIntsctPt = len( intsctPtList )
        print( f'Number of intersection points: {nIntsctPt}' )

        ## Saving corner detected image.
        #cv2.imwrite( os.path.join( outputFilepath, i[:-4] + '_corners' + i[-4:] ), corners )

#-------------------------------------------------------------------------------

        #cv2.imshow( 'img', img )
        #cv2.imshow( 'blur', otsu )
        #cv2.imshow( 'eroded', eroded )
        #cv2.imshow( 'edges', edges )
        #cv2.imshow( 'output', output )
        #cv2.imshow( 'corners', corners )
        #key = cv2.waitKey(0)
        #if key & 0xFF == 27:    break   # esc to break.

#-------------------------------------------------------------------------------

        # TASK: 2.2.2. Calibration.

        # World coordinate values for the corner points are also calculated.
        # This will stay the same for all the patterns and hence it is only
        # calculated once (during the processing of the first pattern image)
        # and then skipped in the other iterations.
        if idx == 0:

            worldPtList = []

            x00, y00 = 0, 0       # The physical measurement in mm for the 1st corner.

            # These are the lengths of the sides of the black squares.
            # So the corners are separated by this distance in the world coordinate.
            xStep, yStep = 25, 25

            nStepsAlongX = len( pureVertLinesPts )      # No. of squares along x.
            nStepsAlongY = len( pureHoriLinesPts )      # No. of squares along y.

            for pIdx, pt in enumerate( intsctPtList ):
                xW = x00 + xStep * ( pIdx % nStepsAlongX )
                yW = y00 + yStep * int( pIdx / nStepsAlongX )
                worldPtList.append( [ xW, yW ] )
                #print( xW, yW )

#-------------------------------------------------------------------------------

        ## Convert to homogeneous coordinate format.
        #intsctPtListH = [ [ pt[0], pt[1], 1 ] for pt in intsctPtList ]

        # Xi = H * Xw equation (xi and xw known) is written as Ah = 0, to solve for h.
        # h is the vector [h11, h12, h13, h21, h22, h23, h31, h32, h33].
        A = []
        for i in range( nIntsctPt ):
            xw, yw = worldPtList[ i ][0], worldPtList[ i ][1]
            xi, yi = intsctPtList[ i ][0], intsctPtList[ i ][1]
```

```
            A.append( [ xw, yw, 1, 0, 0, 0, -xi*xw, -xi*yw, -xi ] )
            A.append( [ 0, 0, 0, xw, yw, 1, -yi*xw, -yi*yw, -yi ] )

        # Finding the homography for the current image.
        A = np.array( A )

        # Ah = 0 is a homogeneous equation, so a least square minimization of this
        # overdetermined system, without any constraint will always give the trivial
        # h=0 solution. To prevent it a contraint of ||h|| = 1 is applied.
        # Now, the norm of h could have been something else also, but it will always
        # be a scalar number. Hence without loss of generality we can assume this to
        # be 1. This is because any other norm value will just be a scalar multiple of
        # this and also no matter what multiple it is, it will not affect the equation
        # Ah = 0. As, the equation stays the same even after multiplying a scalar to h.

        # To find the h, svd is done on A and the last column of Vm matrix will give
        # the non-trivial h.
        Um, D, VmT = np.linalg.svd( A )

        h = np.transpose( VmT )[:,-1]
        H = np.reshape( h, (3,3) )
        H = H / H[2,2]

        setOfH.append( H )      # Storing the H in a list.

#--------------------------------------------------------------------------------

        # Finding the rows of the V and b matrix for the current image.

        V12 = np.array( [ H[0,0]*H[0,1], \
                          H[0,0]*H[1,1] + H[1,0]*H[0,1], \
                          H[1,0]*H[1,1], \
                          H[2,0]*H[0,1] + H[0,0]*H[2,1], \
                          H[2,0]*H[1,1] + H[1,0]*H[2,1], \
                          H[2,0]*H[2,1] ] )

        V11 = np.array( [ H[0,0]*H[0,0], \
                          H[0,0]*H[1,0] + H[1,0]*H[0,0], \
                          H[1,0]*H[1,0], \
                          H[2,0]*H[0,0] + H[0,0]*H[2,0], \
                          H[2,0]*H[1,0] + H[1,0]*H[2,0], \
                          H[2,0]*H[2,0] ] )

        V22 = np.array( [ H[0,1]*H[0,1], \
                          H[0,1]*H[1,1] + H[1,1]*H[0,1], \
                          H[1,1]*H[1,1], \
                          H[2,1]*H[0,1] + H[0,1]*H[2,1], \
                          H[2,1]*H[1,1] + H[1,1]*H[2,1], \
                          H[2,1]*H[2,1] ] )

        V.append( V12.tolist() )
        V.append( ( V11 - V22 ).tolist() )

#--------------------------------------------------------------------------------

    V = np.array(V)
    #print( V.shape )

    # Now, finding the omega. V*omega = 0 is the equation, which is again an
    # overdetermined homogeneous system of equation, the non-trivial solution of
    # which can be found by imposing a constraint of ||omega|| = 1, (without loss
    # of generality) using the svd of V and then taking the last column of the Vm
    # matrix.
    Um, D, VmT = np.linalg.svd( V )
```

```
    omega = np.transpose( VmT )[:,-1]
    Omega = np.array( [ [ omega[0], omega[1], omega[3] ], \
                        [ omega[1], omega[2], omega[4] ], \
                        [ omega[3], omega[4], omega[5] ] ] )

    #print( f'Omega:\n{Omega}' )

#-------------------------------------------------------------------------------

    # Finding the value of K intrinsic parameter matrix by Zhang's algorithm.
    y0num = Omega[0,1] * Omega[0,2] - Omega[0,0] * Omega[1,2]
    y0den = Omega[0,0] * Omega[1,1] - Omega[0,1]**2
    y0 = y0num / y0den
    Lambda = Omega[2,2] - ( Omega[0,2]**2 + y0 * y0num ) / Omega[0,0]
    alphax = math.sqrt( Lambda / Omega[0,0] )
    alphay = math.sqrt( Lambda * Omega[0,0] / y0den )
    s = -1 * Omega[0,1] * (alphax**2) * alphay / Lambda
    x0 = s * y0 / alphay - Omega[0,2] * (alphax**2) / Lambda

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

    #print( f'intrinsic parameter matrix K:\n{K}' )

#-------------------------------------------------------------------------------

    # Finding the R matrix and the t vector for every pattern.
    Kinv = np.linalg.inv( K )

    # The rotation matrix (R) and the translation vector (t) for every image is
    # also stored in a separate list.
    setOfR, setOfT = [], []

    for idx, i in enumerate( listOfFiles ):
        H = setOfH[ idx ]     # Homography matrix for ith pattern.

        h1, h2, h3 = H[:, 0], H[:, 1], H[:, 2]
        r1 = np.matmul( Kinv, h1 )

        r2 = np.matmul( Kinv, h2 )
        r3 = np.cross( r1, r2 )
        t = np.matmul( Kinv, h3 )

        eta1 = 1 / np.linalg.norm( r1 )      # Reciprocal of the norm of r1.
        eta2 = 1 / np.linalg.norm( r2 )      # Reciprocal of the norm of r1.

        eta = ( eta1 + eta2 ) * 0.5

        # Ideally the eta1 should be equal to eta2, but they will not be the same
        # due to noise in the measurement. So, either eta1 or eta2 can be taken
        # and the final LM optimization will take care of the error and fix it.
        # But we found that when eta = eta1 + eta2, the overall error was better
        # compared to when eta = eta1 or eta = eta2. Although it took more time
        # for the LM optimization to converge with this.

        # This will be used to scale the R and t elements.
        r1, r2, r3, t = eta * r1, eta * r2, eta * r3, eta * t

        # Reshaping r's and t into 3x1 vectors from arrays of size (3,).
        r1 = np.reshape( r1, (3,1) )
        r2 = np.reshape( r2, (3,1) )
        r3 = np.reshape( r3, (3,1) )
        t = np.reshape( t, (3,1) )

        R = np.hstack( (r1, r2, r3) )
```

```
        # Now conditioning R so that it is orthonormal by using svd.
        Um, D, VmT = np.linalg.svd( R )
        R = np.matmul( Um, VmT )    # This is the orthonormal version of R.

        setOfT.append( t )      # Storing the t in a list.
        setOfR.append( R )      # Storing the R in a list.

        #print( f'{i}, R:\n{R}' )

#-------------------------------------------------------------------------------

    # Radial distortion parameters are initilized to 0.
    k1, k2 = 0, 0

#-------------------------------------------------------------------------------

    # Using Levenberg-Marqdth algorithm for finding the refined values of
    # R, K, t. For this we have to send their elements as a vector into the
    # error function.
    setOfXij = setOfIntSctPtList
    XWij = worldPtList

    Knew, setOfRnew, setOfTnew, k1new, k2new = \
                        refineParamByLM( setOfXij, XWij, K, setOfR, setOfT, k1, k2 )

    print( f'K:\n{K}' )
    print( f'Refined K:\n{Knew}' )
    print( f'k1: {k1}, k2: {k2}' )
    print( f'Refined k1: {k1new}, Refined k2: {k2new}' )

    #print( len(setOfR), len(setOfRnew), len(setOfT), len(setOfTnew) )

#-------------------------------------------------------------------------------

    # Reprojction of the points to the image to check the amount of refinement.
    # The mean and variance of the error in position is also calculated (both
    # with and without optimization).

    # Convert worldPtList to homogeneous format.
    XWij = [ [ pt[0], pt[1], 0, 1 ] for pt in worldPtList ]
    # zw is 0 for all the corners as the pattern is assumed to be in z=0 plane.

    XWij = np.array( XWij ).T     # Shape 80x4 transposed to 4x80.

    totalError, totalSqError, totalErrorLm, totalSqErrorLm = 0, 0, 0, 0

    for idx, i in enumerate( listOfFiles ):
        # First estimating the position based on the unoptimized parameters.
        img = cv2.imread( os.path.join( filepath, i ) )    # Read images

#-------------------------------------------------------------------------------

        R, t = setOfR[idx], setOfT[idx]
        P = np.matmul( K, np.hstack( ( R, t ) ) )

        XijEstH = np.matmul( P, XWij ).T    # Shape 3x80 transposed to 80x3.

        # Convert to planar.
        for j in range( XijEstH.shape[0] ):     XijEstH[j] /= XijEstH[j][2]

        # Suppose number of corners is 80.
        XijEst = XijEstH[ :, :-1 ]    # Shape now 80x2. Last column removed. Planar form.
        Xij = setOfIntSctPtList[ idx ]
        Xij = np.array( Xij )    # Shape 80x2.
```

```
        # Incorporate radial distortion parameters as well.
        x, y, x0, y0 = XijEst[0], XijEst[1], K[0,2], K[1,2]
        rSq = (x - x0)**2 + (y - y0)**2
        xRad = x + (x - x0) * ( k1 * rSq + k2 * rSq**2 )
        yRad = y + (y - y0) * ( k1 * rSq + k2 * rSq**2 )

        # Remember that the xRad and yRad are all 80x1 vectors.
        XijEst[0] = xRad
        XijEst[1] = yRad

        error = Xij - XijEst        # This is also an 80x1 vector.
        error = np.linalg.norm( error, axis=1 )     # Taking norm of error. Shape 80x1.
        errorSq = error**2

        error = np.mean( error )    # This error is now the euclidean distance between
        # the actual and estimated Xij value.
        # The mean is calculated and not the sum, as this is an 80x1 vector.
        variance = np.mean( errorSq ) - error**2

        totalSqError += error**2
        totalError += error

#-------------------------------------------------------------------------------

        Rlm, tLm = setOfRnew[idx], setOfTnew[idx]
        Plm = np.matmul( Knew, np.hstack( ( Rlm, tLm ) ) )

        XijEstHlm = np.matmul( Plm, XWij ).T    # Shape 3x80 transposed to 80x3.

        # Convert to planar.
        for j in range( XijEstHlm.shape[0] ):     XijEstHlm[j] /= XijEstHlm[j][2]

        # Suppose number of corners is 80.
        XijEstLm = XijEstHlm[ :, :-1 ]    # Shape now 80x2. Last column removed. Planar form.

        # Incorporate radial distortion parameters as well.
        x, y, x0, y0 = XijEstLm[0], XijEstLm[1], Knew[0,2], Knew[1,2]
        rSq = (x - x0)**2 + (y - y0)**2
        xRad = x + (x - x0) * ( k1new * rSq + k2new * rSq**2 )
        yRad = y + (y - y0) * ( k1new * rSq + k2new * rSq**2 )

        # Remember that the xRad and yRad are all 80x1 vectors.
        XijEstLm[0] = xRad
        XijEstLm[1] = yRad

        errorLm = Xij - XijEstLm        # This is also an 80x1 vector.
        errorLm = np.linalg.norm( errorLm, axis=1 )     # Taking norm of error. Shape 80x1.
        errorSqLm = errorLm**2

        errorLm = np.mean( errorLm )    # This error is now the euclidean distance between
        # the actual and estimated Xij value after optimization.
        # The mean is calculated and not the sum, as this is an 80x1 vector.
        varianceLm = np.mean( errorSqLm ) - errorLm**2

        totalSqErrorLm += errorLm**2
        totalErrorLm += errorLm

#-------------------------------------------------------------------------------

        # Plot the points.
        mapped1 = copy.deepcopy( img )   # Copy of the image on which points will be drawn.
        mapped2 = copy.deepcopy( img )   # Copy of the image on which points will be drawn.

        for pIdx, pt in enumerate( setOfIntSctPtList[ idx ] ):
            cv2.circle( mapped1, tuple(pt), 3, (0,0,255), -1 )
```

```
            #print( int(XijEst[pIdx][0]), int(XijEst[pIdx][1]) )
            cv2.circle( mapped1, ( int(XijEst[pIdx][0]), int(XijEst[pIdx][1]) ), 4, (255,0,0), 2 )
            cv2.circle( mapped2, tuple(pt), 3, (0,0,255), -1 )
            #print( int(XijEstLm[pIdx][0]), int(XijEstLm[pIdx][1]) )
            cv2.circle( mapped2, ( int(XijEstLm[pIdx][0]), int(XijEstLm[pIdx][1]) ), 4, (0,255,0), 2 )

        mapped = np.hstack( ( mapped1, mapped2 ) )
        cv2.imshow( 'mapped (left: without LM; right: after LM)', mapped )
        key = cv2.waitKey(50)
        cv2.destroyAllWindows()

        # Saving edge detected image.
        cv2.imwrite( os.path.join( outputFilepath, i[:-4] + '_mapped' + i[-4:] ), mapped )

#-------------------------------------------------------------------------------

        print( f'Image: {i}' )
        print( f'Rotation Matrix (without LM):\n{R}' )
        print( f'Translation Vector (without LM):\n{t}' )
        print( f'Reprojection error (without LM) Mean: {error}, Variance: {variance}' )
        print( f'Rotation Matrix (after LM optimization):\n{Rlm}' )
        print( f'Translation Vector (after LM optimization):\n{tLm}' )
        print( f'Reprojection error (after LM optimization) Mean: {errorLm}, Variance: {varianceLm}' )

#-------------------------------------------------------------------------------

    meanError = totalError / nPatterns
    errorVariance = totalSqError / nPatterns - meanError**2

    print( f'Mean Error: {meanError} (without optimization)' )
    print( f'Error Variance: {errorVariance} (without optimization)' )

    meanErrorLm = totalErrorLm / nPatterns
    errorVarianceLm = totalSqErrorLm / nPatterns - meanErrorLm**2

    print( f'Mean Error: {meanErrorLm} (after LM optimization)' )
    print( f'Error Variance: {errorVarianceLm} (after LM optimization)' )
```