Homework 9

Fangda Li
li1208@purdue.edu
ECE 661 - Computer Vision

November 22, 2016

1 Methodology

This section describes the relevant methods and steps in order to calibrate a camera by posing a checkerboard pattern in front of it using Zhang's algorithm.

1.1 Corner Extraction

The following steps are taken to extract corners on the checkerboard pattern in a raster order.

1. Use Canny edge detector on the input image to extract edges.
2. Apply Hough transform on the edge image to extract lines.
3. Write the lines in their homogeneous representations and divide the lines into two groups based on their gradients: vertical and horizontal.
4. Rank the lines in each group based on their intersection with x-axis or y-axis.
5. For each pair between the sorted horizontal lines and vertical lines, find the intersection using the cross product of their homogeneous representations: $\overrightarrow{p} = \overrightarrow{l_1} \times \overrightarrow{l_2}$.

Note that any new intersection that is within a certain threshold to a previous intersection is rejected to prevent duplicate corners. Additionally, the width of the square and the distance between adjacent squares are both $25mm$.

1.2 Zhang's Algorithm

This subsection describes the method to find the camera intrinsic matrix $\mathbf{K}$ and the extrinsic parameter matrix $(\mathbf{R}|\overrightarrow{t})$ for a given pose. Subsequently, the camera matrix for the camera in a given pose is represented as $\mathbf{P} = \mathbf{K}(\mathbf{R}|\overrightarrow{t})$.

1.2.1 Obtaining Intrinsic Parameters

First of all, the camera intrinsic matrix $\mathbf{K}$ is defined as

$$\mathbf{K} = \begin{pmatrix} \alpha_x & s & x_0 \\ 0. & \alpha_y & y_0 \\ 0. & 0. & 1. \end{pmatrix}, \tag{1}$$

where α_x and α_y are the scale factors along x and y axes, (x_0, y_0) is the coordinate of the principle point, and s is the skew factor. In Zhang's algorithm, the camera matrix K is obtained by exploiting the idea that the image of absolute conic remains the same regardless the camera pose. The image of absolute conic $\mathbf{w}$ can be obtained using homography from corner correspondences between images. Moreover, the image of the absolute conic Ω_∞ on the camera sensor plane is given by

$$\mathbf{w} = (\mathbf{K}\ \mathbf{R})^{-T}\Omega_\infty(\mathbf{K}\ \mathbf{R})^{-1}. \tag{2}$$

Since Ω_∞ is a 3 by 3 identity matrix and $\mathbf{R}$ is a orthonormal rotation matrix, $\mathbf{w}$ further simplifies to

$$\mathbf{w} = \mathbf{K}^{-T}\mathbf{K}^{-1}. \tag{3}$$

As a result, after we have obtained $\mathbf{w}$, Cholesky decomposition can be used to obtain $\mathbf{K}^{-1}$ and then we can further obtain $\mathbf{K}$. More specifically, we can establish one-to-one relationship between $\mathbf{w}$ and the parameters in $\mathbf{K}$ as shown below,

$$y_0 = \frac{w_{12}w_{13} - w_{11}w_{23}}{w_{11}w_{22} - w_{12}^2}, \tag{4}$$

$$\lambda = w_{33} - \frac{w_{13}^2 + y_0(w_{12}w_{13} - w_{11}w_{23})}{w_{11}}, \tag{5}$$

$$\alpha_x = \sqrt{\frac{\lambda}{w_{11}}}, \tag{6}$$

$$\alpha_y = \sqrt{\frac{\lambda w_{11}}{w_{11}w_{22} - w_{12}^2}}, \tag{7}$$

$$s = -\frac{w_{12}\alpha_x^2\alpha_y}{\lambda}, \tag{8}$$

$$x_0 = \frac{sy_0}{\alpha_y} - \frac{w_{13}\alpha_x^2}{\lambda}. \tag{9}$$

Now we have obtained the intrinsic camera matrix $\mathbf{K}$.

1.2.2 Obtaining Extrinsic Parameters

The extrinsic parameters (for both rotation and translation) of a given camera pose is defined as

$$(\mathbf{R}|\overrightarrow{t}) = (\overrightarrow{r_1}\ \overrightarrow{r_2}\ \overrightarrow{r_3}|\overrightarrow{t}). \tag{10}$$

Once we have the intrinsic matrix $\mathbf{K}$, the extrinsic parameters can be obtained using

$$\overrightarrow{r_1} = \epsilon \mathbf{K}^{-1}\overrightarrow{h_1}, \tag{11}$$

$$\overrightarrow{r_2} = \epsilon \mathbf{K}^{-1}\overrightarrow{h_2}, \tag{12}$$

$$\overrightarrow{r_3} = \overrightarrow{r_1} \times \overrightarrow{r_2}, \tag{13}$$

$$\epsilon = \frac{1}{\|\mathbf{K}^{-1}\overrightarrow{h_1}\|}. \tag{14}$$

Note that since the scale factor does not make $\overrightarrow{r_2}$ and $\overrightarrow{r_3}$ unit magnitude, we further condition the column vectors in the rotation matrix to be orthonormal. For that purpose, we decompose $\mathbf{R}$ using SVD and reset all the singular values to be 1.

1.2.3 Refining Both Intrinsic and Extrinsic Parameters

In this experiment, non-linear optimization algorithm Levenberg-Marquardt is used to further refine the initial estimates of $\mathbf{K}$ and $(\mathbf{R}|\overrightarrow{t})$. The sum of geometric distances between all the extracted corners and their corresponding reprojected corners is used as the cost function as shown below

$$d^2_{geom} = \sum_i \sum_j \|\overrightarrow{x_{i,j}} - \mathbf{K}(\mathbf{R}_i|\overrightarrow{t_i})\overrightarrow{x}_{M,j}\|^2, \tag{15}$$

where $\overrightarrow{x_{i,j}}$ is the jth extracted corner from the ith pose using method described in Subsection 1.1 and $\overrightarrow{x}_{M,j}$ is the jth corner on the checkerboard pattern in world frame.

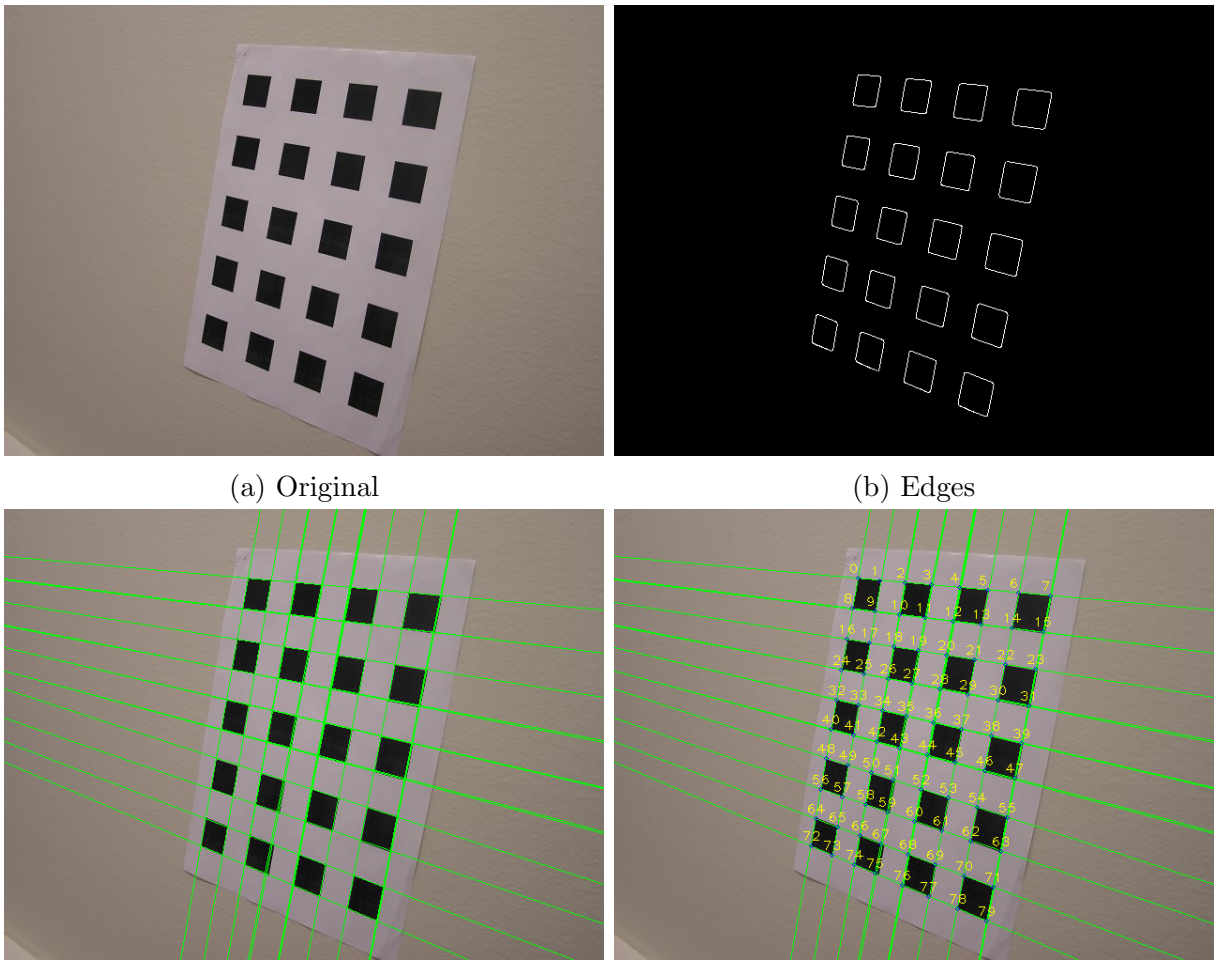
2 Results

In this section, the camera calibration results obtained from both the provided and the self-recorded dataset are shown.

2.1 Provided Dataset

Camera intrinsic matrix K before non-linear optimization is

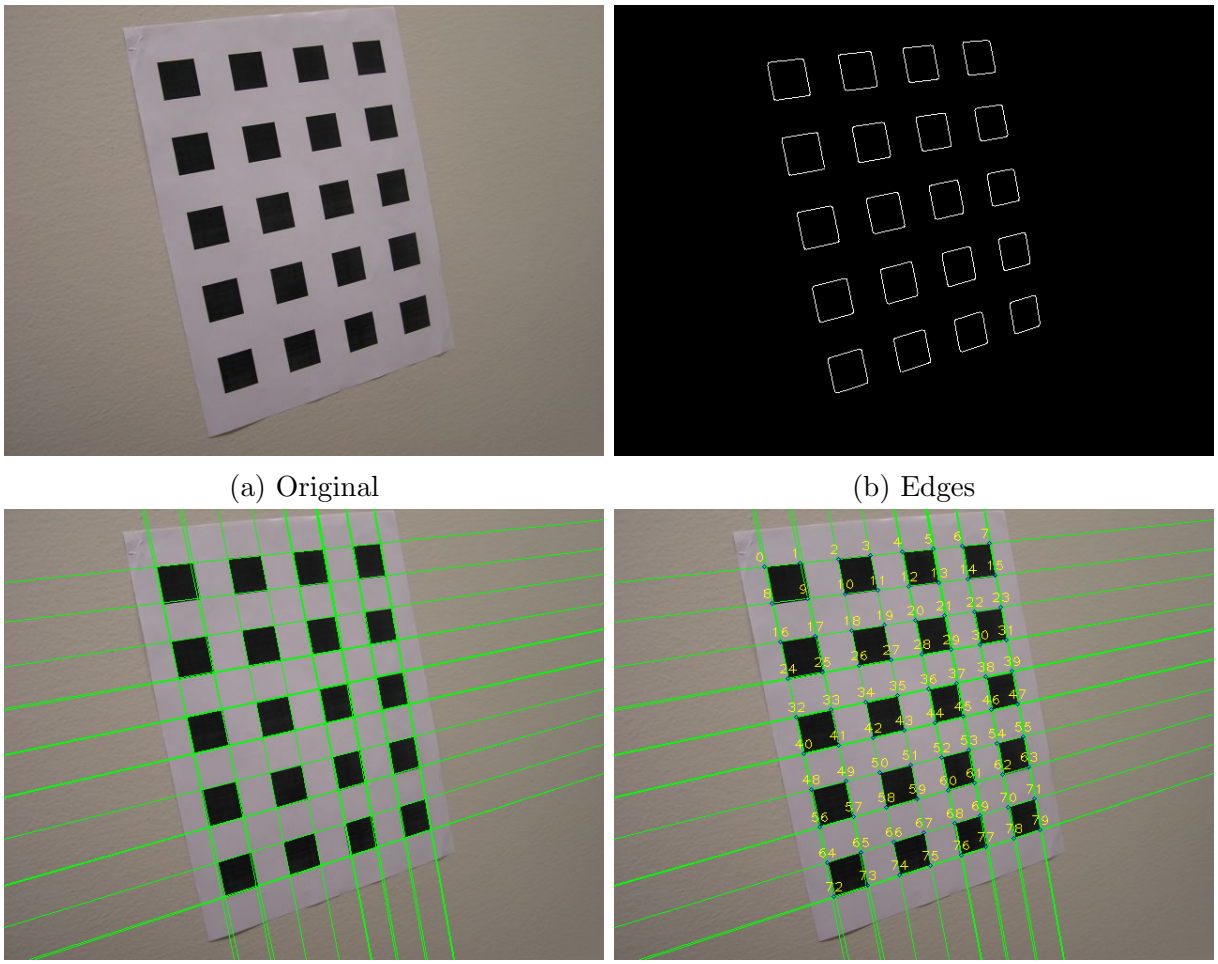
$$K = \begin{pmatrix} 718.69344112 & 3.7734745 & 319.32009093 \\ 0. & 719.99638721 & 237.70640619 \\ 0. & 0. & 1. \end{pmatrix}. \tag{16}$$



(a) Original (b) Edges

(c) Hough transform (d) Corner detection

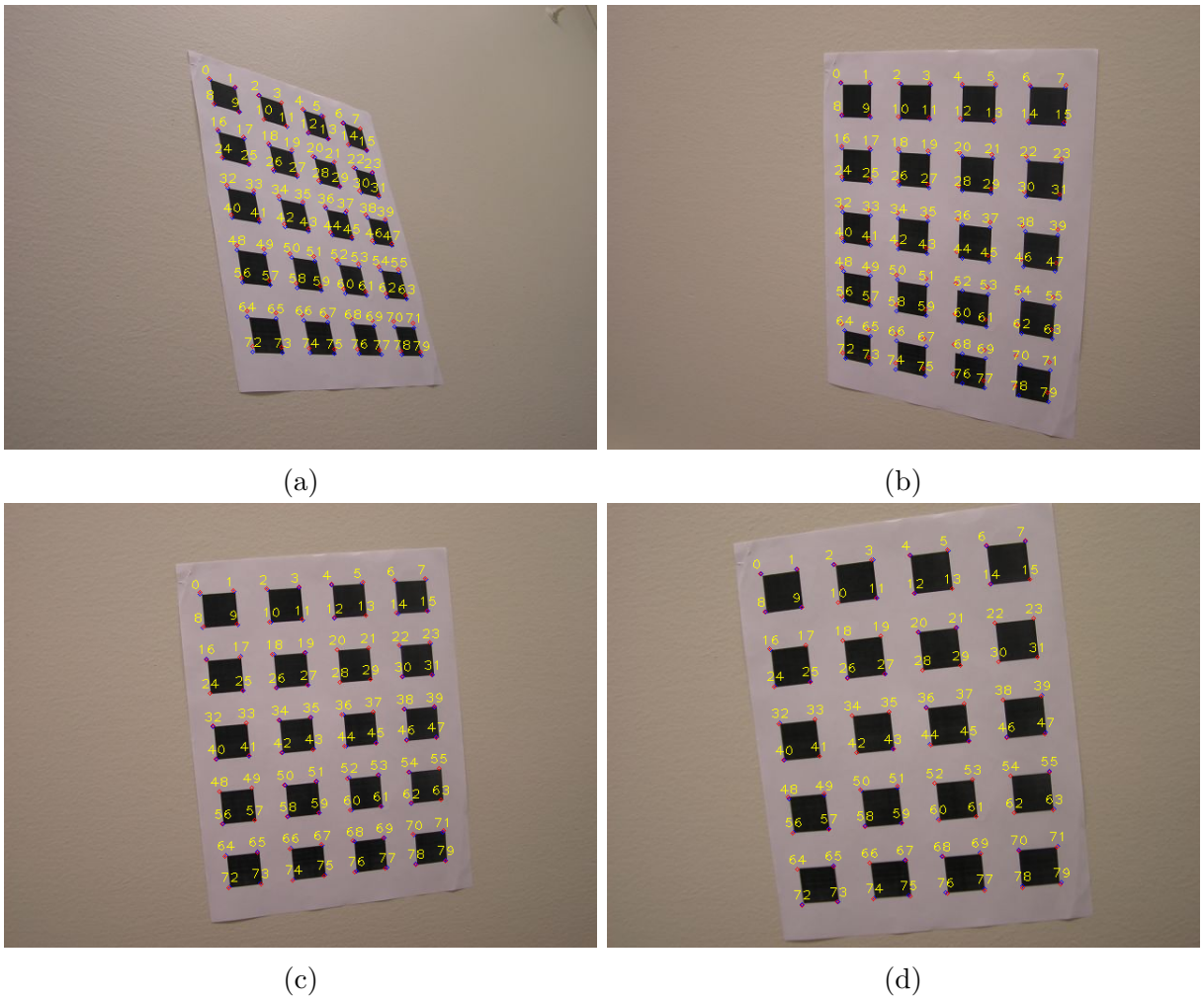
Figure 1: Corner extraction on *Pic_1.jpg*



(a) Original (b) Edges

(c) Hough transform (d) Corner detection

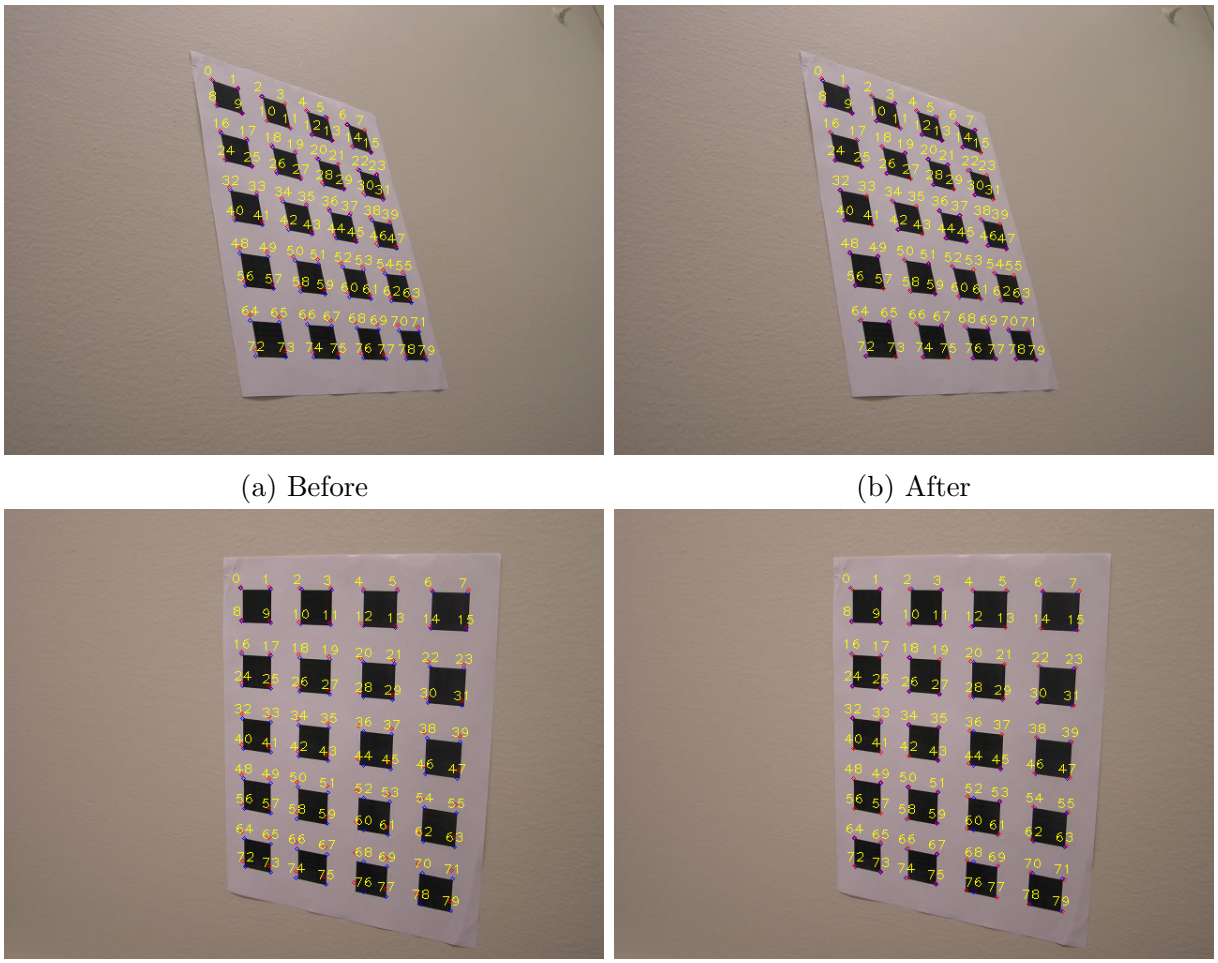
Figure 2: Corner extraction on *Pic_17.jpg*



(a) (b)

(c) (d)

Figure 3: Corner reprojections; Extracted corners are in blue and reprojected corners are in red.



(a) Before (b) After

(c) Before (d) After

Figure 4: Corner reprojections before and after non-linear optimization; Extracted corners are in blue and reprojected corners are in red.

Before optimization, the mean and variance of the Euclidean distance between the extracted corners and reprojected corners are 1.6244 and 2.1793, respectively. Camera intrinsic matrix K after non-linear optimization is

$$K_{refined} = \begin{pmatrix} 724.92435081 & 4.97612844 & 319.44336444 \\ 0. & 725.68450003 & 239.90575963 \\ 0. & 0. & 1. \end{pmatrix}. \tag{17}$$

After optimization, the mean and variance of the Euclidean distance between the extracted corners and reprojected corners are 1.0111 and 0.5943, respectively. In terms of mean, the quantitative improvement is about 37.8%. Furthermore, the improvement can be illustrated visually in Figure 4.

The extrinsic rotation and translation matrices $(R|\overrightarrow{t})$ after optimization from the 4 different views in Figure 3 are shown respectively below.

The extrinsic matrix for Figure 3a is

$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.75230586 & 0.1803458 & -0.63364918 & -72.05307666 \\ 0.13570207 & 0.89875918 & 0.41691351 & -119.27655034 \\ 0.64468662 & -0.39963398 & 0.6516685 & 537.46085348 \end{pmatrix}. \tag{18}$$

The extrinsic matrix for Figure 3b is

$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.82807658 & -0.01098266 & 0.56050741 & -50.8333923 \\ 0.05282902 & 0.99688771 & -0.05851483 & -110.6590152 \\ -0.5581203 & 0.07806581 & 0.82607957 & 564.09841786 \end{pmatrix}. \tag{19}$$

The extrinsic matrix for Figure 3c is

$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.98609229 & 0.08033216 & -0.14549479 & -73.74974304 \\ -0.08974268 & 0.99419778 & -0.0593046 & -99.01416102 \\ 0.13988653 & 0.0715369 & 0.98757999 & 501.82830922 \end{pmatrix}. \tag{20}$$

The extrinsic matrix for Figure 3d is

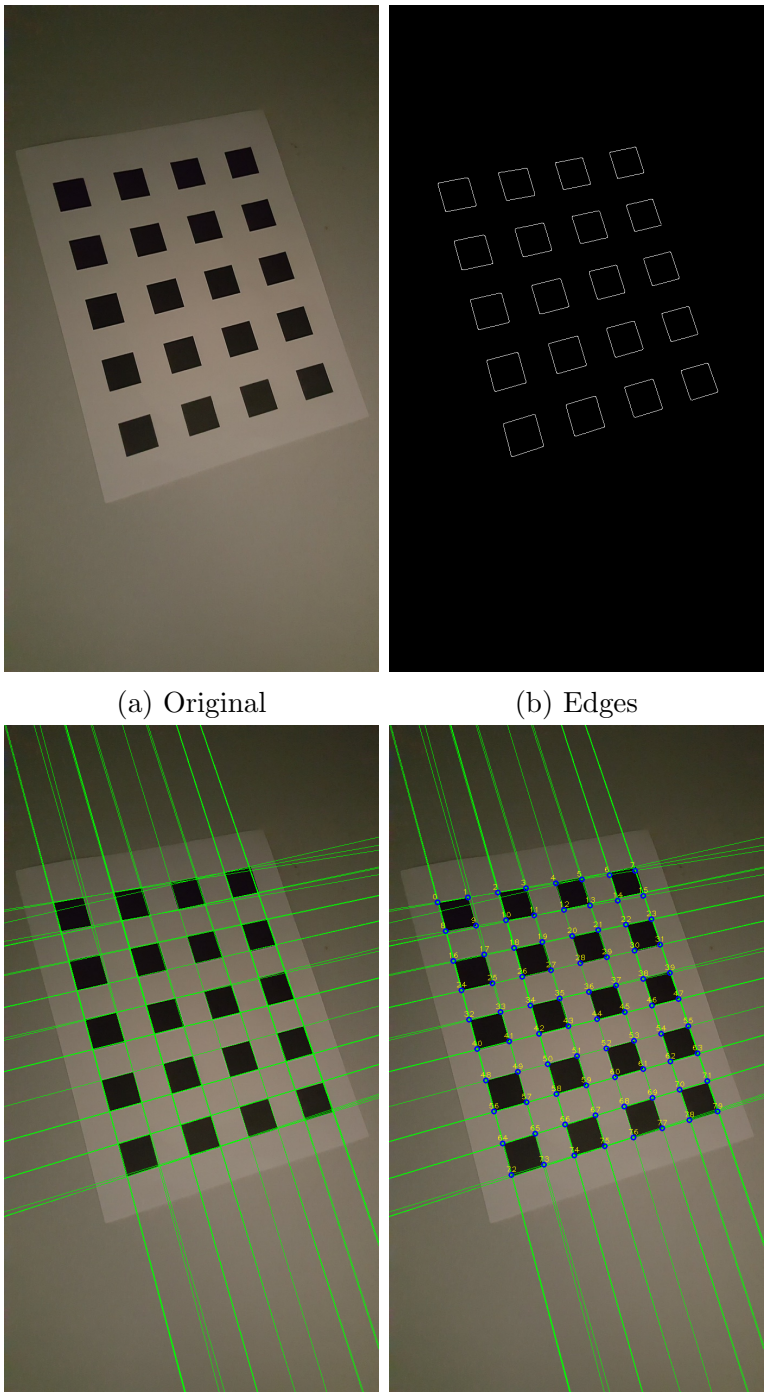
$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.99119653 & 0.10672754 & 0.0783497 & -94.31559486 \\ -0.10153932 & 0.99254114 & -0.06746739 & -100.98138359 \\ -0.08496593 & 0.05891786 & 0.99464037 & 446.36545472 \end{pmatrix}. \tag{21}$$

2.2 Self-recorded Dataset

Camera intrinsic matrix K before non-linear optimization is

$$K = \begin{pmatrix} 963.36024844 & 5.49904103 & 374.35955352 \\ 0. & 972.39541799 & 602.42975897 \\ 0. & 0. & 1. \end{pmatrix}. \tag{22}$$

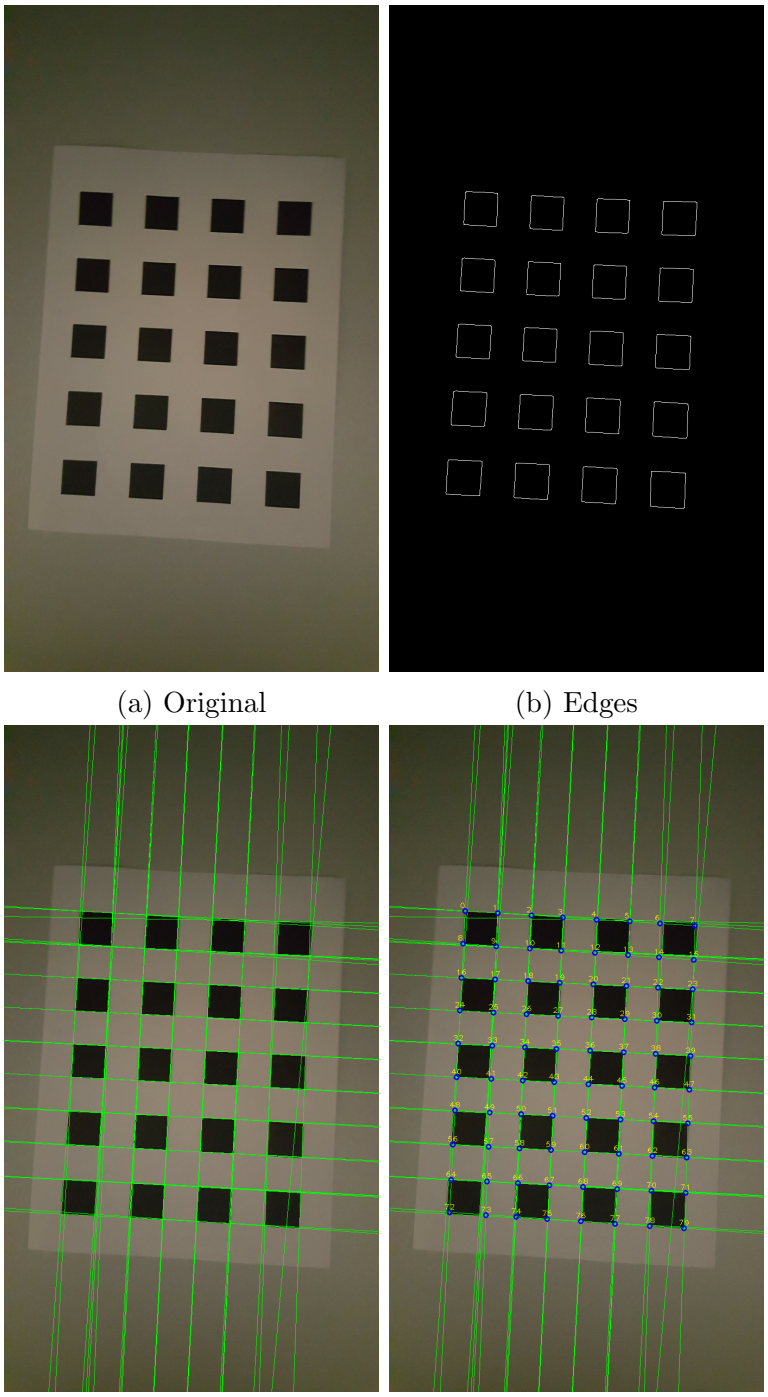
Before optimization, the mean and variance of the Euclidean distance between the extracted corners and reprojected corners are 2.4010 and 3.2782, respectively. Camera intrinsic matrix



(a) Original (b) Edges

(c) Hough transform (d) Corner detection

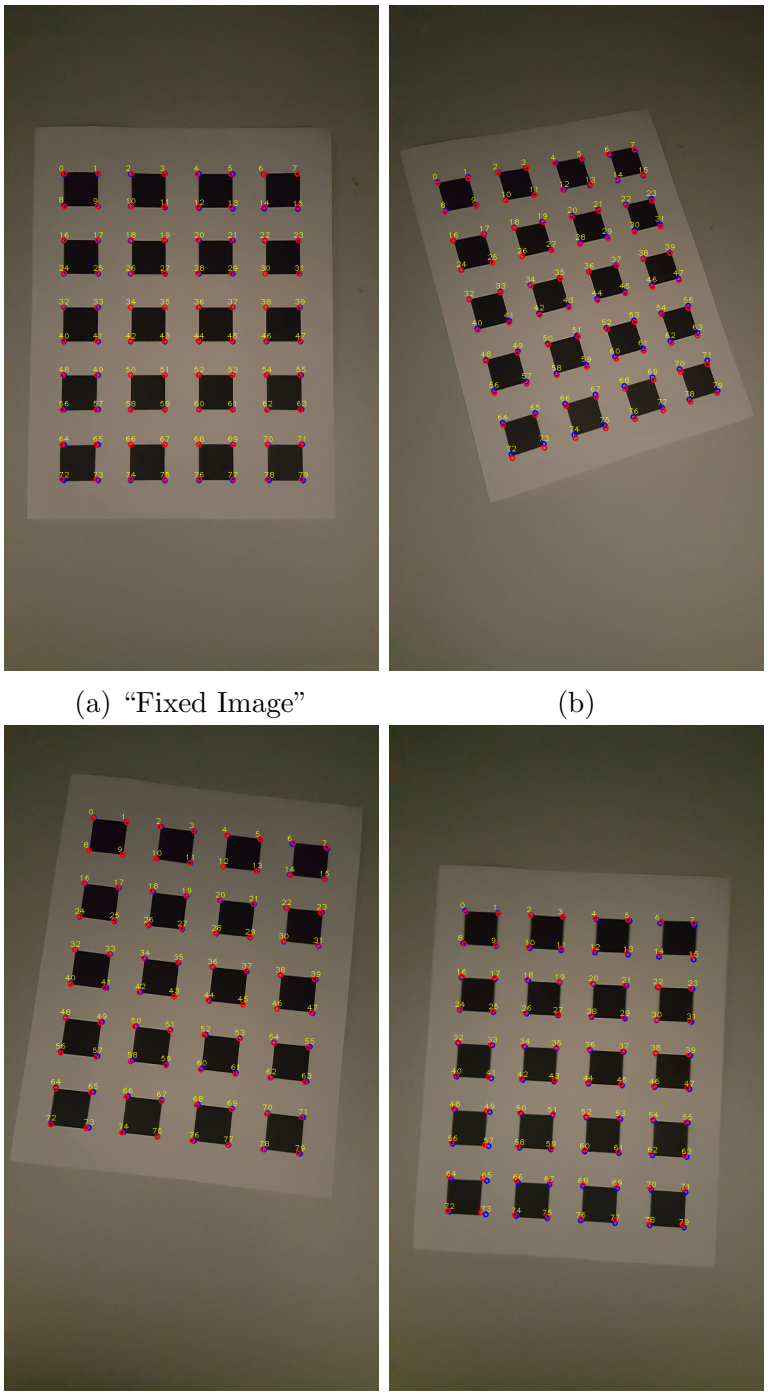
Figure 5: Corner extraction on self-recorded dataset



(a) Original (b) Edges

(c) Hough transform (d) Corner detection

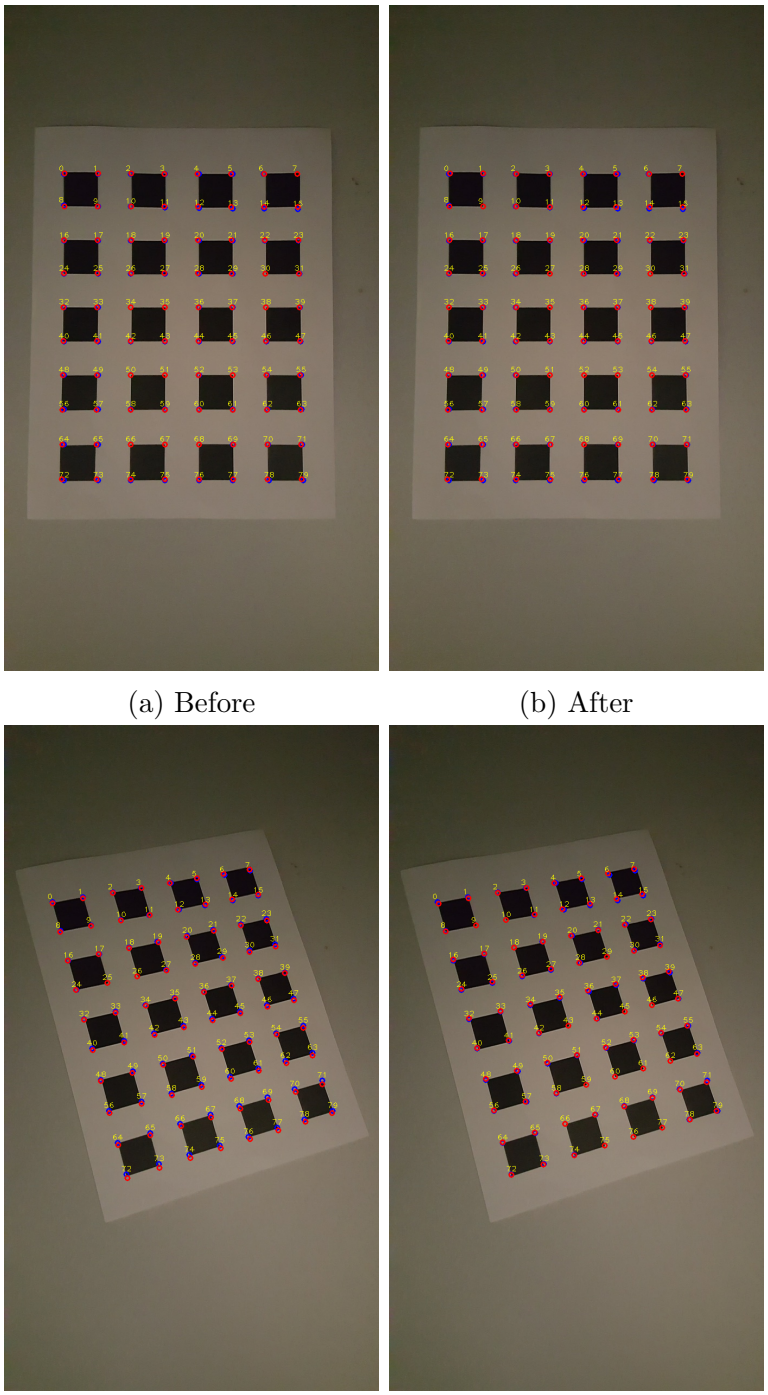
Figure 6: Corner extraction on self-recorded dataset



(a) “Fixed Image” (b)

(c) (d)

Figure 7: Corner reprojections; Extracted corners are in blue and reprojected corners are in red.



(a) Before (b) After

(c) Before (d) After

Figure 8: Corner reprojections before and after non-linear optimization; Extracted corners are in blue and reprojected corners are in red.

K after non-linear optimization is

$$K_{refined} = \begin{pmatrix} 962.92439165 & 7.06962085 & 373.27576831 \\ 0. & 971.63428103 & 594.5974995 \\ 0. & 0. & 1. \end{pmatrix}. \tag{23}$$

After optimization, the mean and variance of the Euclidean distance between the extracted corners and reprojected corners are 1.7548 and 2.1825, respectively. In terms of mean, the quantitative improvement is about 26.9%. However, not much improvement can be illustrated visually, as seen in Figure 8. The might be caused by the size of the images being relatively larger than the provided dataset, making it harder to tell the corner displacements.

The extrinsic rotation and translation matrices $(R|\overrightarrow{t})$ after optimization from the 4 different views in Figure 7 are shown respectively below.

The extrinsic matrix for the "Fixed Image" Figure 7a is

$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.99998724 & -0.00064117 & -0.00501 & -105.79644606 \\ 0.00091283 & 0.99851821 & 0.05441091 & -114.8501585 \\ 0.00496769 & -0.05441479 & 0.99850606 & 393.20508951 \end{pmatrix}. \tag{24}$$

Note that the z component in the translation vector is about $39.3cm$, which is in alignment with the ground-truth measurement of the camera's height $41cm$. The extrinsic matrix for Figure 7b is

$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.94017504 & 0.28589041 & -0.18530397 & -125.22812302 \\ -0.24327467 & 0.94413365 & 0.22232652 & -117.44451396 \\ 0.23851274 & -0.16394608 & 0.95720079 & 430.18572267 \end{pmatrix}. \tag{25}$$

The extrinsic matrix for Figure 7c is

$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.99285695 & -0.11613001 & -0.02736599 & -82.8752485 \\ 0.11832347 & 0.98783726 & 0.10088163 & -169.36194413 \\ 0.01531776 & -0.10339907 & 0.994522 & 389.2511254 \end{pmatrix}. \tag{26}$$

The extrinsic matrix for Figure 7d is

$$(R|\overrightarrow{t})_{refined} = \begin{pmatrix} 0.99850003 & -0.05330842 & 0.01248634 & -95.72341538 \\ 0.05259578 & 0.99727391 & 0.05175358 & -102.85522863 \\ -0.0152112 & -0.05101922 & 0.99858182 & 401.02640623 \end{pmatrix}. \tag{27}$$

3 Source Code

3.1 corner_detection.py

```
#!/usr/bin/python
from pylab import *
import cv2
```

```
def extract_homo_lines_from_image(image):
        '''
                Given a checkerboard input image, return the detected lines.
        '''
        height, width = image.shape[0], image.shape[1]
        # Blur the image to remove background edges
        # 7 by 7 kernel, default sigma
        blurred = cv2.GaussianBlur(image, (7,7), 0)
        # Apply Canny edge
        # minVal and maxVal
        edges = cv2.Canny(blurred, 100, 200)
        figure()
        imshow(edges)
        imsave("./figures/edges_2.png",edges, cmap='gray')
        # Probabilistic Hough transform
        # distance resolution, angle resolution and accumulator threshold
        lines = cv2.HoughLinesP(edges, 1, pi/180, 25, minLineLength=100, maxLineGap=100)
        lines = squeeze(lines)
        # print "Number of lines found before refinement...", lines.shape[0]
        return [cross( array([x1,y1,1]), array([x2,y2,1]) ) for x1,y1,x2,y2 in lines]

def extract_sorted_corners_from_homo_lines(homolines):
        '''
                Given 'vertical' and 'horizontal' lines, return their intersections.
        '''
        homolines = array(homolines)
        # We need to divide the input lines into two groups, horizontal and vertical
        theta = array([arctan(float(-a)/(b+0.01)) for a,b,c in homolines])
        horilines = homolines[logical_and(theta >= -pi/4, theta < pi/4), :]
        vertlines = homolines[logical_or(theta < -pi/4, theta >= pi/4), :]
        # Now we need to sort the lines for future labeling process
        Yinters = -horilines[:,2] / horilines[:,1]
        horilines = horilines[ argsort(Yinters), :]
        Xinters = -vertlines[:,2] / vertlines[:,0]
        vertlines = vertlines[ argsort(Xinters), :]
        # Find the intersections in order
        homocorners = []
        for hline in horilines:
                for vline in vertlines:
                        curr = cross(hline, vline)
                        curr = curr / curr[2]
                        # Skip this corner if it's too close to one of the previous co
                        if min([norm(curr-prev) for prev in homocorners] + [30]) >= 30:
                                homocorners.append(curr)
        return homocorners
```

```
def extract_sorted_corners(image):
        homolines = extract_homo_lines_from_image(image)
        return extract_sorted_corners_from_homo_lines(homolines)

def main():
        # n = 40
        # for i in range(1,n+1):
        #         image = imread('./Dataset1/Pic_{0}.jpg'.format(i))
        #         homolines = extract_homo_lines_from_image(image)
        #         corners = extract_sorted_corners_from_homo_lines(homolines)
        #         print './Dataset1/Pic_{0}.jpg'.format(i), 'Number of corners found',

        image = imread('./Dataset2/15.jpg'.format(17))
        height, width = image.shape[0], image.shape[1]
        homolines = extract_homo_lines_from_image(image)
        corners = extract_sorted_corners_from_homo_lines(homolines)
        for a,b,c in homolines:
                cv2.line(image, (0, -c/b), (width-1,-(c+a*(width-1))/b), color=(0,255,0)
        figure()
        imshow(image)
        imsave("./figures/hough_2.png",image)
        for i,corner in enumerate(corners):
                cv2.circle(image, (corner[0], corner[1]), 4, color=(0,0,255), thickness=
                cv2.putText(image, str(i), (corner[0]-10, corner[1]-5), cv2.FONT_HERSHEY
        figure()
        imshow(image)
        imsave("./figures/corner_2.png",image)
        show()


if __name__ == '__main__':
        main()
```

3.2 llsm.py

```
#!/usr/bin/python
import numpy as np
import cv2

def get_inhomo_system(pts1, pts2):
        '''
                Construct A and b of the inhomogeneous equation system.
                @pts1,pts2: list of matching points
                @return: tuple of A and b
```

```
        '''
        n = len(pts1)
        A = np.zeros((2*n,8))
        b = np.zeros((2*n,1))
        for i in range(n):
                x,y,w = pts1[i][0], pts1[i][1], 1.
                xp,yp,wp = pts2[i][0], pts2[i][1], 1.
                A[2*i] = [0, 0, 0, -wp*x, -wp*y, -wp*w, yp*x, yp*y]
                A[2*i+1] = [wp*x, wp*y, wp*w, 0, 0, 0, -xp*x, -xp*y]
                b[2*i] = -yp*w
                b[2*i+1] = xp*w
        return A, b

def get_llsm_homograhpy_from_points(pts1, pts2):
        '''
                Obtain the homography using all the inliers from ransac.
                @pts1,pts2: list of matching points
                @return: np.ndarray of H
        '''
        A, b = get_inhomo_system(pts1, pts2)
        # h is pseudo-inverse multiplied by b
        h = np.dot( np.dot( np.linalg.inv( np.dot(A.transpose(), A) ), A.transpose() ),
        h = np.append(h, 1.)
        h = h.reshape((3,3))
        return h
```

3.3 zhangs.py

```
#!/usr/bin/python
from pylab import *
import cv2
import os
from scipy.optimize import leastsq
from llsm import get_llsm_homograhpy_from_points
from corner_detection import extract_sorted_corners

def get_vij(H, i, j):
        '''
                Construct one row of V.
        '''
        # i=0,1,2; j=0,1,2
        H = H.T
        return array([ H[i,0]*H[j,0],
                                H[i,0]*H[j,1] + H[i,1]*H[j,0],
                                H[i,1]*H[j,1],
```

```
                                H[i,2]*H[j,0] + H[i,0]*H[j,2],
                                H[i,2]*H[j,1] + H[i,1]*H[j,2],
                                H[i,2]*H[j,2] ])

def get_absolute_conic_image(Hs):
        '''
                Given homography matrices,
                obtain W, image of the absolute conic.
        '''
        numH = len(Hs)
        W = zeros((3,3))
        V = zeros((2*numH, 6))
        for k,H in enumerate(Hs):
                V[2*k] = get_vij(H, 0, 1)
                V[2*k+1] = get_vij(H, 0, 0) - get_vij(H, 1, 1)
        # b is the eigenvector of V for the smallest eigenvalue
        u, s, vt = svd(dot( V.T, V ))
        b = vt[-1,:]
        W = array([ [b[0], b[1], b[3]],
                                [b[1], b[2], b[4]],
                                [b[3], b[4], b[5]] ])
        return W

def get_intrinsic_matrix(W):
        '''
                Given the image of absolute conic W,
                return the intrinsic camera matrix K.
        '''
        y0 = (W[0,1]*W[0,2] - W[0,0]*W[1,2]) / (W[0,0]*W[1,1] - W[0,1]**2)
        lamda = W[2,2] - ( W[0,2]**2 + y0*(W[0,1]*W[0,2] - W[0,0]*W[1,2]) ) / W[0,0]
        ax = sqrt(lamda / W[0,0])
        ay = sqrt(lamda*W[0,0] / (W[0,0]*W[1,1] - W[0,1]**2))
        s = - W[0,1]*ax*ax*ay / lamda
        x0 = s*y0 / ay - W[0,2]*ax*ax / lamda
        K = array([ [ax, s, x0],
                                [0., ay, y0],
                                [0., 0., 1.] ])
        return K

def get_world_frame_corners(d):
        '''
                Given the size of squares on the pattern,
                return the coordinates of all the corners in world frame.
        '''
        corners = []
```

```
        for i in range(10):
                for j in range(8):
                        corner = array([j*d, i*d, 1])
                        corners.append(corner)
        return corners

def get_extrinsic_matrix(K, H):
        '''
                Given K and H for a camera pose,
                return the extrinsic matrix [R|t].
        '''
        h1, h2, h3 = H[:,0], H[:,1], H[:,2]
        Kinv = inv(K)
        epsilon = 1. / norm(dot(Kinv, h1))
        r1 = epsilon * dot(Kinv, h1)
        r2 = epsilon * dot(Kinv, h2)
        r3 = cross(r1, r2)
        t = epsilon * dot(Kinv, h3)
        R = vstack((r1,r2,r3)).T
        # Condition the rotation matrix to be orthonormal
        U, _, Vt = svd(R)
        R = dot(U, Vt)
        return hstack((R,t.reshape(3,1)))

def compensate_radial_distortion(arrayImage, x0, y0, k1, k2):
        '''
                Given points on image planes, compensate the radial distortion.
        '''
        r2 = (arrayImage[0,:]-x0)**2 + (arrayImage[1,:]-y0)**2
        r4 = r2**2
        arrayImage[0,:] = arrayImage[0,:] + (arrayImage[0,:]-x0) * (k1*r2 + k2*r4)
        arrayImage[1,:] = arrayImage[1,:] + (arrayImage[1,:]-y0) * (k1*r2 + k2*r4)
        return arrayImage

def reproject_corners(P, wcorners):
        '''
                Re-project pattern corners in world frame to image plane.
        '''
        # z in corners are zeros, i.e. corners are on z plane
        Hhat = P[:, [0,1,3]]
        pcorners = []
        for wcorner in wcorners:
                pcorner = dot(Hhat, wcorner)
                pcorner = pcorner / pcorner[-1]
                pcorners.append(pcorner)
```

```
        return pcorners

def pack_parameters(K, Rts):
        '''
                Given K and Rt matrices for all poses, pack them into a single vector.
        '''
        p = K.flatten()[[0,1,2,4,5]]
        for Rt in Rts:
                R = Rt[:,:3]
                r,_ = cv2.Rodrigues(R)
                r = r.flatten()
                t = Rt[:,3].flatten()
                p = hstack((p,r,t))
        return p

def unpack_parameters(p):
        '''
                Given a single vector, unpack into K and Rt matrices.
        '''
        K = array([[p[0], p[1], p[2]],
                                [0., p[3], p[4]],
                                [0., 0., 1.]])
        Rts = []
        p = p[5:].reshape(-1,6)
        nRows = p.shape[0]
        for i in range(nRows):
                row = p[i,:]
                R,_ = cv2.Rodrigues(row[:3])
                t = row[3:].reshape(3,-1)
                Rt = hstack((R,t))
                Rts.append(Rt)
        return K, Rts

def get_reprojection_accuracy(arrayImage, arrayReproj):
        '''
                Given array of repojected corners and ground truth corners,
                return the mean and variance of the Euclidean distance between them.
        '''
        dist = norm( arrayImage - arrayReproj, axis=0 )
        return mean(dist), var(dist)

def calibrate_camera(dataset, N=40, d=25,levmar=True):
        wcorners = get_world_frame_corners(d)
        # Convert list to array, each column is a point
        arrayWorld = zeros((3,80))
```

```
arrayImage = zeros((3,80*N))
arrayReproj = zeros((3,80*N))
for i,w in enumerate(wcorners): arrayWorld[:,i] = w
Hs = []
images = []
print "Working on dataset...", dataset
# dataset1
if dataset == 1:
        # Compute all homography matrices
        for i in range(1,N+1):
                image = imread('./Dataset1/Pic_{0}.jpg'.format(i))
                images.append(image)
                # Extract corners first
                icorners = extract_sorted_corners(image)
                arrayImage[:,(i-1)*80 : i*80] = array(icorners).T
                print 'Number of corners found...', len(icorners)
                # Compute the homography from world to image plane
                H = get_llsm_homograhpy_from_points(wcorners, icorners)
                Hs.append(H)
# dataset2
elif dataset == 2:
        # Compute all homography matrices
        for i in range(N):
                image = imread('./Dataset2/{0}.jpg'.format(i))
                images.append(image)
                # Extract corners first
                icorners = extract_sorted_corners(image)
                arrayImage[:,i*80 : (i+1)*80] = array(icorners).T
                print 'Number of corners found...', len(icorners)
                # Compute the homography from world to image plane
                H = get_llsm_homograhpy_from_points(wcorners, icorners)
                Hs.append(H)
# Extract the intrinsic matrix
W = get_absolute_conic_image(Hs)
K = get_intrinsic_matrix(W)
# Extract the extrinsic matrix for each image
Rts = []
for H in Hs:
        Rt = get_extrinsic_matrix(K, H)
        Rts.append(Rt)
for i,Rt in enumerate(Rts):
        P = dot(K, Rt)
        arrayReproj[:,i*80 : (i+1)*80] = array(reproject_corners(P, wcorners)).T
print "==========Before LM=========="
print "K =", K
```

```
        print "Euclidean distance: mean, variance =", get_reprojection_accuracy(arrayIma
        if not levmar: return K, Rts
        # Refine the K,R,t estimates with Levenberg-Marquedt
        p_guess = pack_parameters(K, Rts)
        def error_function(p):
                '''
                        Geometric distance as cost function for LevMar.
                '''
                arrayTrans = zeros(arrayImage.shape)
                K, Rts = unpack_parameters(p)
                for i,Rt in enumerate(Rts):
                        P = dot(K, Rt)
                        Hhat = P[:, [0,1,3]]
                        trans = dot(Hhat, arrayWorld)
                        trans = trans / trans[-1,:]
                        arrayTrans[:,i*80 : (i+1)*80] = trans
                error = arrayTrans - arrayImage
                return error[:2,:].flatten()
        print "Optimizing..."
        p_refined, _ = leastsq(error_function, p_guess)
        K_refined, Rts_refined = unpack_parameters(p_refined)
        for i,Rt_refined in enumerate(Rts_refined):
                P_refined = dot(K_refined, Rt_refined)
                arrayReproj[:,i*80 : (i+1)*80] = array(reproject_corners(P_refined, wcor
        print "==========After LM=========="
        print "K_refined =", K_refined
        print "Euclidean distance: mean, variance =", get_reprojection_accuracy(arrayIma
        return K_refined, Rts_refined

def main():
        d = 25
        # Obtain the camera matrix
        K, Rts = calibrate_camera(d=d, levmar=True)

if __name__ == '__main__':
        main()
```

3.4 main.py

```
#!/usr/bin/python
from pylab import *
import cv2
import os
from corner_detection import extract_sorted_corners
from zhangs import calibrate_camera, reproject_corners, get_world_frame_corners
```

```
def test_dataset1():
        d = 25
        N = 40
        dataset = 1
        # Obtain the camera matrix
        K, Rts = calibrate_camera(dataset, N=N, d=d, levmar=True)
        # Backproject to validate
        n = 30
        Rt = Rts[n-1]
        print "Rt =", Rt
        P = dot(K, Rt)
        image = imread('./Dataset1/Pic_{0}.jpg'.format(n))
        height, width = image.shape[0], image.shape[1]
        wcorners = get_world_frame_corners(d)
        pcorners = reproject_corners(P, wcorners)
        corners = extract_sorted_corners(image)
        for i in range(len(corners)):
                pcorner = pcorners[i].astype(int)
                corner = corners[i]
                cv2.circle(image, (corner[0], corner[1]), 2, color=(0,0,255), thickness=
                cv2.circle(image, (pcorner[0], pcorner[1]), 2, color=(255,0,0), thicknes
                cv2.putText(image, str(i), (corner[0]-10, corner[1]-5), cv2.FONT_HERSHEY
        imshow(image)
        # imsave("./figures/reproject_3.png",image)
        show()

def test_dataset2():
        d = 25
        N = 20
        dataset = 2
        # Obtain the camera matrix
        K, Rts = calibrate_camera(dataset, N=N, d=d, levmar=True)
        # Backproject to validate
        n = 15
        Rt = Rts[n]
        print "Rt =", Rt
        P = dot(K, Rt)
        image = imread('./Dataset2/{0}.jpg'.format(n))
        height, width = image.shape[0], image.shape[1]
        wcorners = get_world_frame_corners(d)
        pcorners = reproject_corners(P, wcorners)
        corners = extract_sorted_corners(image)
        for i in range(len(corners)):
                pcorner = pcorners[i].astype(int)
```

```
                corner = corners[i]
                cv2.circle(image, (corner[0], corner[1]), 4, color=(0,0,255), thickness=
                cv2.circle(image, (pcorner[0], pcorner[1]), 4, color=(255,0,0), thicknes
                cv2.putText(image, str(i), (corner[0]-10, corner[1]-5), cv2.FONT_HERSHEY
        imshow(image)
        # imsave("./figures/reproject_4.png",image)
        show()

def main():
        test_dataset1()
        test_dataset2()

if __name__ == '__main__':
        main()
```