ECE 661 Homework 7:
Instructor: Prof. Avi Kak
TA: Dave Kim

Tommy Chang

November 20, 2012

1 Introduction

The goal of this homework is to apply Zhang's[2] camera calibration.

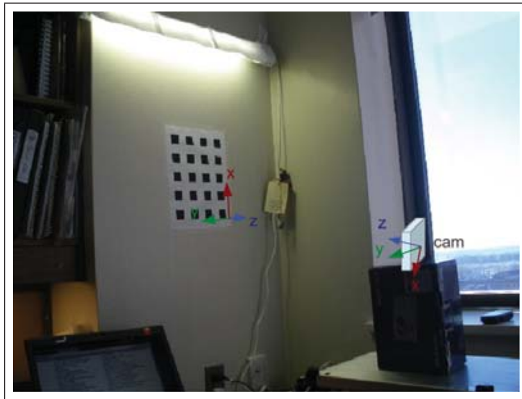
2 A Brief Outline of the Calibration Procedure

1. Place the camera at a marked location and record an image of the calibration pattern. The extrinsic pose parameters calculated will be for this location of the camera.
2. Move the camera and turn it so that it assumes different positions and orientation. Record an image of the calibration pattern for each position/orientation. Note, the calibration pattern is fixed at the same location on the wall. Record at least 20 images.
3. Extract the corners on the calibration pattern for all images.
4. Use the extracted corners and the known physical dimensions of the calibration pattern to calculate the camera calibration parameters.
5. Verify the camera calibration parameters by re-project the physical corners from the calibration pattern onto the images.

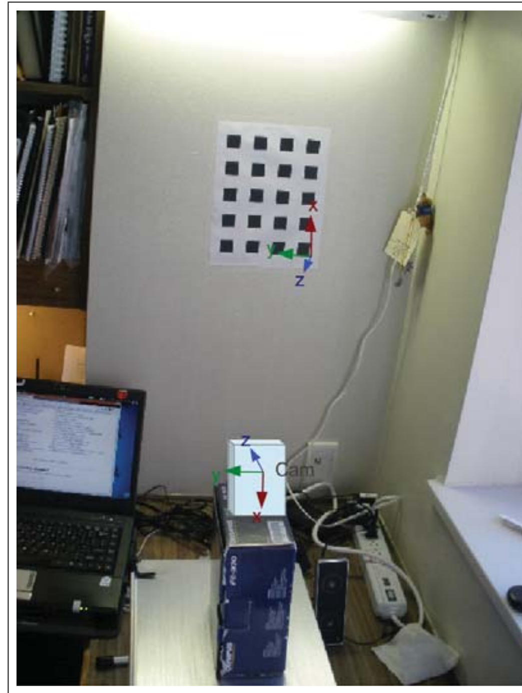
3 Setup

Figure 1 shows the marked location of the camera and its relative pose to the calibration pattern on the wall. The placement of the camera is drawn as a cyan box. The *camera coordinate frame* is marked by the X,Y,Z axes. The image taken at this setup is shown in Figure 2 (Note that the camera is place on its side.) The *world coordinate frame* is marked on this image as well.

Section 7 shows the result of camera external parameters. The external parameters describes how the *world frame* is located and oriented *relative* to the *camera frame*. For example, the external parameter having a translation vector $t = [t_x t_y t_z]^T$ means that the *world frame* is located at (t_x, t_y, t_y) in the camera coordinate. (i.e., with respect to the origin of the *camera frame*.) In this setup, we would expect the external parameter to have a positive t_z value because the *world frame* is located in the +z direction of the *camera frame*.



(a) Camera Setup



(b) Camera Setup in another view.

Figure 1: The blue color band and its segmented result.

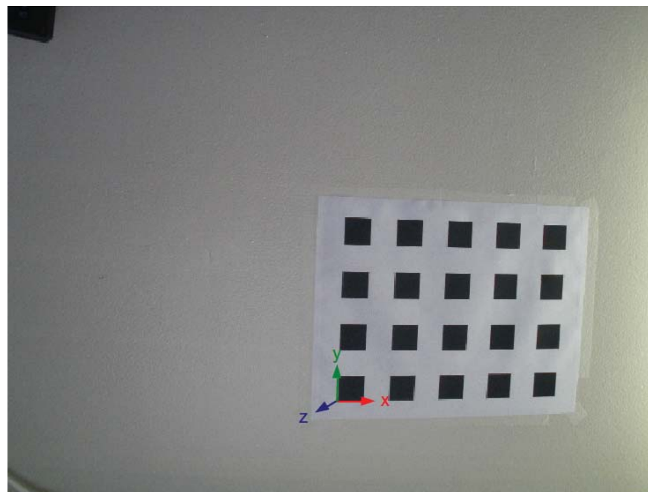


Figure 2: Image taken by the camera placed at the setup position/orientation. The *world frame* is shown by the 3 labeled axes at a corner – X is up, Y is to the right and Z is out of the picture. All corners are separated by 1.0 inch in both X and Y directions.

4 Corner Extraction

1. Apply the Canny edge detector to the image.
2. Use the Hough transform to fit straight lines to the Canny edges.
3. Initialize the corners as the intersections of the fitted straight lines.
4. Refine the initial corners to the actual corner locations if the radial distortion is severe.

4.1 Canny edge detector

I used the OpenCV Canny detector[5]. It has two required parameters:

1. first threshold: This parameters is used to find the initial strong edges. The default range of this parameter is from 0 to (9 * 255), ie. 255 times the default 3x3 kernel size. The higher the value, the less edges is found. The value I used is (1.5 * 255).
2. second threshold: This is used to find the weaker edges that are continuous from the strong edges. The value I used is 255.

4.2 Hough Transform to detect lines and initial corners

I used the OpenCV HoughLines2[3] function. It has three required parameters:

1. rho: Distance resolution in pixel-related units. I used 1 pixel for this parameter.
2. theta: Angle resolution measured in radians. I used $\pi/180$ radians.
3. threshold: Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than threshold. I used 50 pixels.

One common issue with Hough Transform is that the same line may be detected multiple times. A Non-maxim suppression scheme is needed to achieve a clean result. My approach is the following:

1. Sort lines detected by Hough Transform by their angles and group them into either vertical or horizontal lines.
2. For each group, if two lines intersect inside the image, then just keep the line with more pixels (i.e., one with a larger accumulator value).
3. For each group, if two lines are too near each other, then just keep the line with more pixels. Since we know exactly how many lines to expect in each group, this nearness threshold can be dynamically set at the run-time to produce the number of expected lines. Typical values found are < 5 pixels.

   More specifically, the line distance in the vertical group is determined by the formulas:

   $$d_x = \rho \ \cos(\theta),$$

   and similarly for the horizontal group:

   $$d_y = \rho \ \sin(\theta),$$

where ρ and θ are the line output from Hough Transform. Although OpenCV's Hough Transform implementation does not return the accumulator values (i.e. number of pixels on the line), this information can be indirectly obtained from the output ordering. Specifically, the output are sorted by the accumulator value so the first line in the output has the most number of pixels on it.

4.3 Refine corners to sub-pixel accuracy

Once the lines are extracted correctly from the Hough Transform, the initial location of the corners can be computed as the intersections between the lines. However, these initial corners may not be exactly on the corners of the calibration pattern due to lens distortion – resulting physically straight-lines becoming curved when projected onto the image.

OpenCV has an implementation of a sub-pixel corner detector[4]. This detector requires an initial guess of the corner location. It iteratively solves a set of simultaneous equations to pinpoint the exact corner location with sub-pixel accuracy. The other relevant parameter is the sampling window size, which specifies a region around the initial guess. The true corner is assumed to be inside this sampling region.

At the run-time this window parameter can be dynamically set to the averaged distance among adjacent lines in the horizontal and vertical directions.

4.4 Results

Figure 4 shows four results of Canny edge detector, Hough Transform, initial corners, and refined corners.

5 Intrinsic, Extrinsic, and Radial Distortion

5.1 Intrinsic parameters

The intrinsic parameters can be solved from the IAC (Image of the Absolute Conic) method[2][8]. The relevant equation are given below:

ω The Image of the Absolute Conic

K The camera calibration matrix

H The planar homograph: image = H * points on Z=0

P The Projection matrix (transform onto the image)

$\vec{I}, \vec{J}$ circular points

Ω_∞ the absolute conic

$\vec{I}^T \Omega_\infty \vec{I}$ circular points are on the absolute conic

$\omega = (K\ R)^{-T} \Omega_\infty (K\ R)^{-1}$ the transform of the absolute conic under P (i.e., to the image plane)

The last equation simplifies to:

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

The internal parameters (i.e., the camera calibration matrix), K, can be solved from the transformation of the absolute conic under H (i.e., to the image plane):

$$(H\vec{I})^T \omega (H\vec{I}) = 0$$

To solve the above system of equations, at least 3 H's are needed.

5.2 Extrinsic parameters

From the definition of the camera projection matrix, P, and a given homography H:

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

$H\vec{X} = \vec{x}$, where

$\vec{X}$ a point in the world

$\vec{x}$ the image of $\vec{X}$

R the 3x3 rotation matrix

$\vec{t}$ the translation vector

the external parameters, R and $\vec{t}$ can be solved.

However, the rotation matrix obtained this way may not have the property: $R^T R = I$. An extra step to condition the rotation matrix is subsequently performed [2].

5.3 Radial distortion

Radial distortion can be modeled by the following process [8]:

1. Assume no skew and no external parameters in the projection matrix:

$$P = K = \begin{bmatrix} \alpha_x & 0 & x_o \\ 0 & \alpha_x & y_o \\ 0 & 0 & 1 \end{bmatrix}$$

2. Backproject undistorted pixel (x,y) to the corresponding point $(X_{\text{cam}}, Y_{\text{cam}}, Z_{\text{cam}})$ in the *camera frame* by taking the inverse transform:

$$\begin{bmatrix} X_{\text{cam}} \\ Y_{\text{cam}} \\ 1 \end{bmatrix} = K^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Note $Z_{\text{cam}} = 1$

3. Apply radial distortion parameters (k_1 and k_2) to get the distorted coordinate (X^d_{cam}, Y^d_{cam}):

$$r^2 = X^2_{\text{cam}} + Y^2_{\text{cam}}$$
$$X^d_{\text{cam}} = X_{\text{cam}}\ (1 + k_1\ r^2 + k_2\ r^4)$$
$$Y^d_{\text{cam}} = Y_{\text{cam}}\ (1 + k_1\ r^2 + k_2\ r^4)$$

4. Finally re-project the distorted coordinate onto the pixel (x_d, y_d):

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = K \begin{bmatrix} X^d_{\text{cam}} \\ Y^d_{\text{cam}} \\ 1 \end{bmatrix}$$

6 Refining the Calibration

The idea is to have a global optimal calibration by minimizing the re-projection error of all the data taken.

Let $\vec{X}_{M,j}$ be the jth corner of the calibration pattern lying on the world Z=0 plane (i.e, the model plane). And let $\vec{x}_{M,i,j}$ be the projected pixel of $\vec{X}_{M,j}$. Let $\vec{x}_{i,j}$ be the jth corner on the ith image.

Then, the goal is to minimize the geometric distance:

$$d^2 = \sum_i \sum_j \|\vec{x}_{i,j} - \vec{X}_{M,i,j}\|^2$$

This can be done by using a none-linear least squares optimization method such as the Levenberg-Marquartdt method.[7]

$\vec{x}_{M,i,j}$ is a function of all the calibration parameters to be optimized as well as the corner on the model plane:

$$d^2 = \sum_i \sum_j \|\vec{x}_{i,j} - f(K, \vec{r}_i, t_i, k_1, k_2, \vec{X}_{M,j})\|^2,$$

where $\vec{r}_i$ is the Rodrigues representation of the rotation matrix R_i. The Rodrigues representation (OpenCV has built-in Rodrigues implementation[6].) is a compact rotation representation and has only 4 parameters consisting of an axis of rotation and angle of rotation. It is important not to use the 3x3 rotation matrix directly as they would be treated as independent variables during the optimization.

6.1 Results

Below are the result of global optimization.

- On provided 40-images data set:

$$\text{Interal parameters:} K = \begin{bmatrix} 727.825389 & 1.0363175 & 321.665895 \\ 0. & 728.94595 & 240.388295 \\ 0. & 0. & 1. \end{bmatrix}$$

Radial Distortion : $k_1 = -0.15726582388, \quad k_2 = 0.10701098002$

Max re-projection error = 1.59 pixels

- On my 20-images data set:

$$\text{Interal parameters:} K = \begin{bmatrix} 705.640229 & 0.463950353 & 312.397783 \\ 0. & 703.551374 & 235.200342 \\ 0. & 0. & 1. \end{bmatrix}$$

Radial Distortion : $k_1 = -0.3390339977, \quad k_2 = 0.50399862554$

Max re-projection error = 1.63 pixels

7 Assessment of Calibration Accuracy

To visually assess the calibration accuracy, the physical calibration pattern (ie., corners) in the *world frame* are projected back into the images. The projection uses the globally optimized camera parameters as well as the distortion parameters.

7.1 Results

Figures 5 to 8 show 4 reprojected result and the corresponding external parameters.

8 Figures

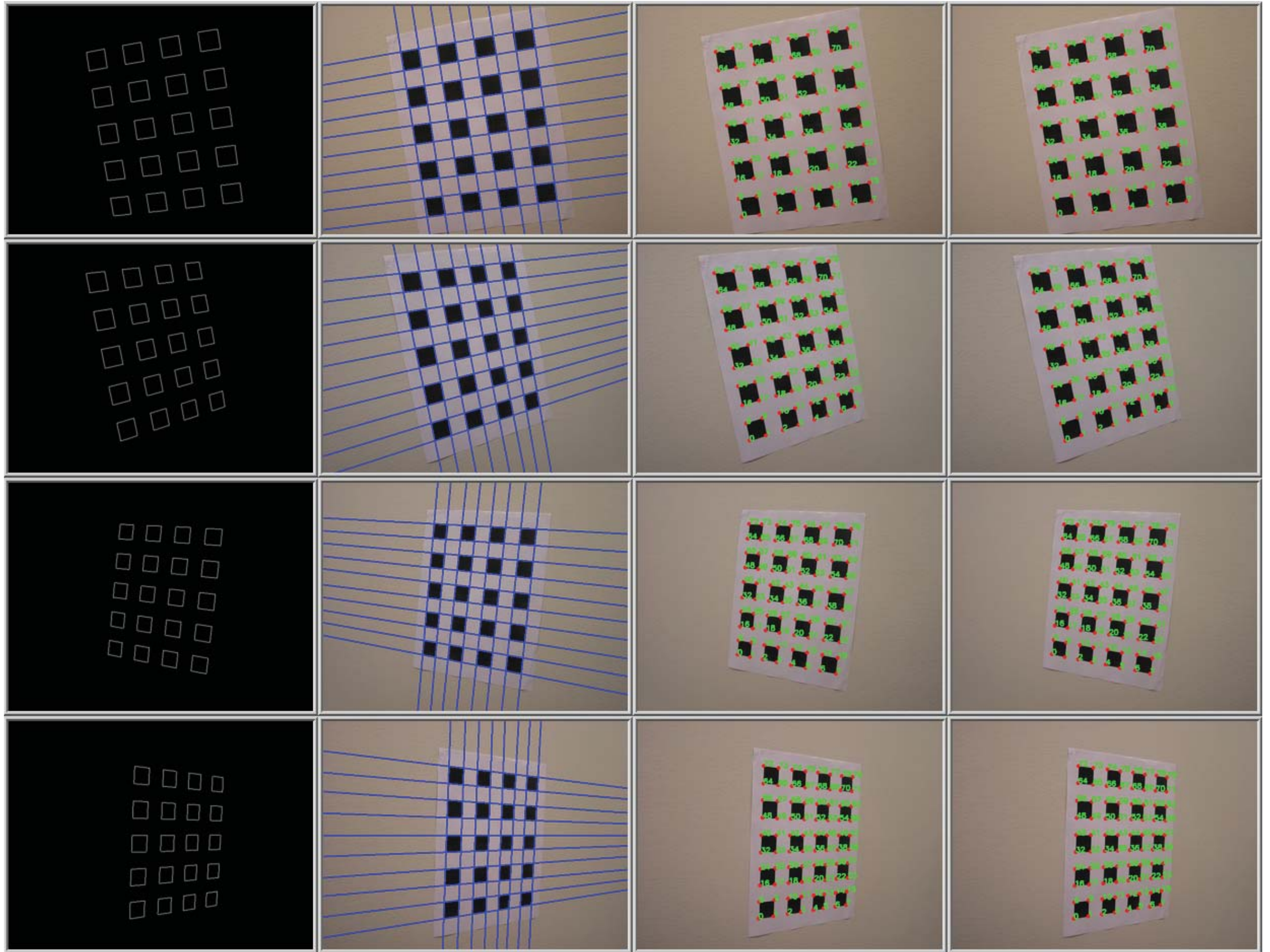


Figure 3: Four images from the provided data set. First column = Canny edge detector. Second column = Hough Transform. Third column = initial corns as intersections of lines. Fourth column = refined corner with sub-pixel accuracy

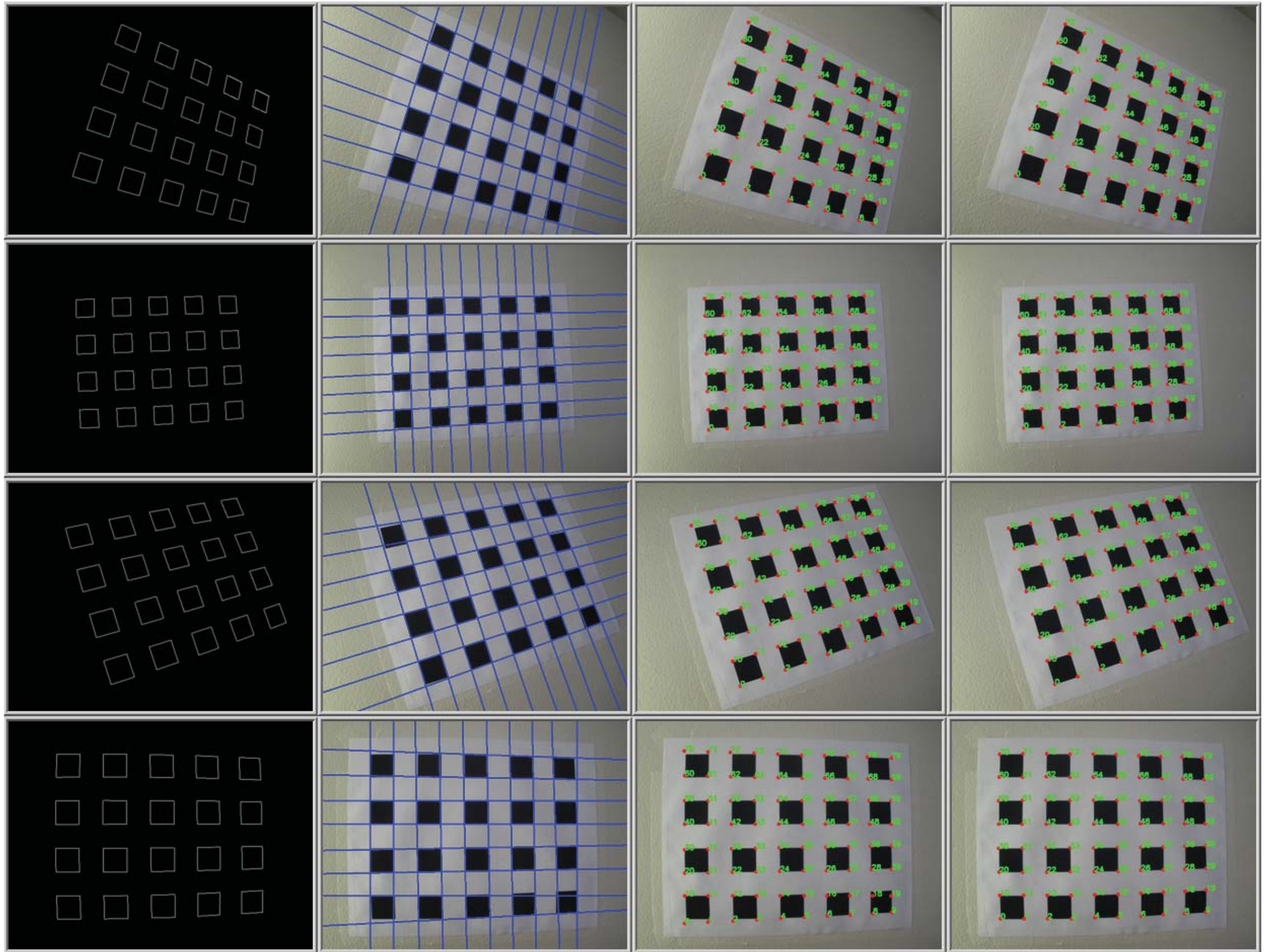
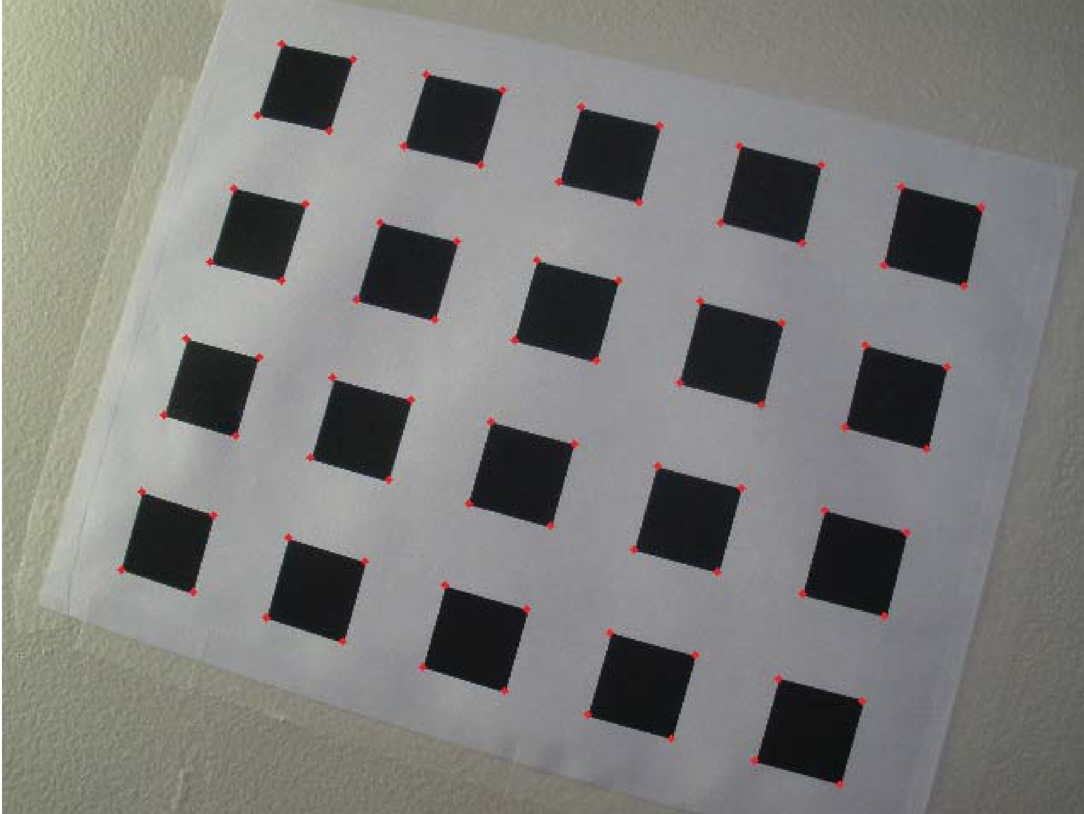
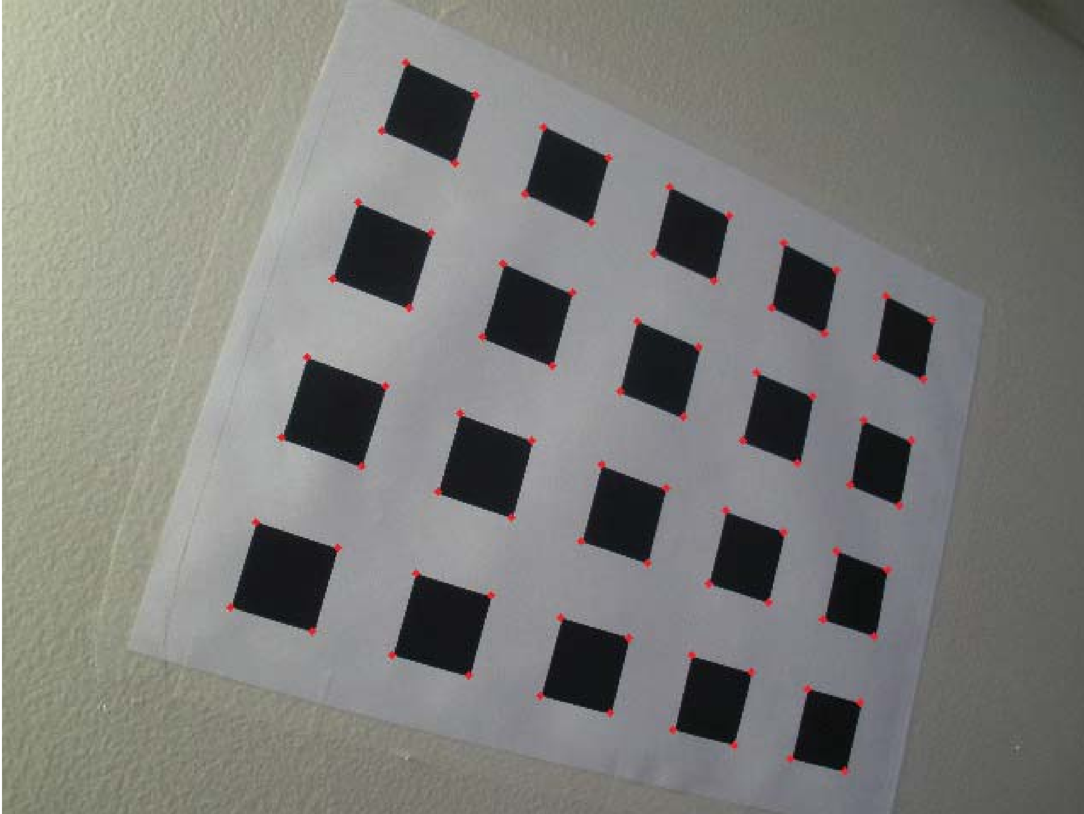


Figure 4: Four images from data set taken with my cheap camera. The columns are ordered the same way as in Figure 4. A noticeable radial distortion can be seen in the last image.



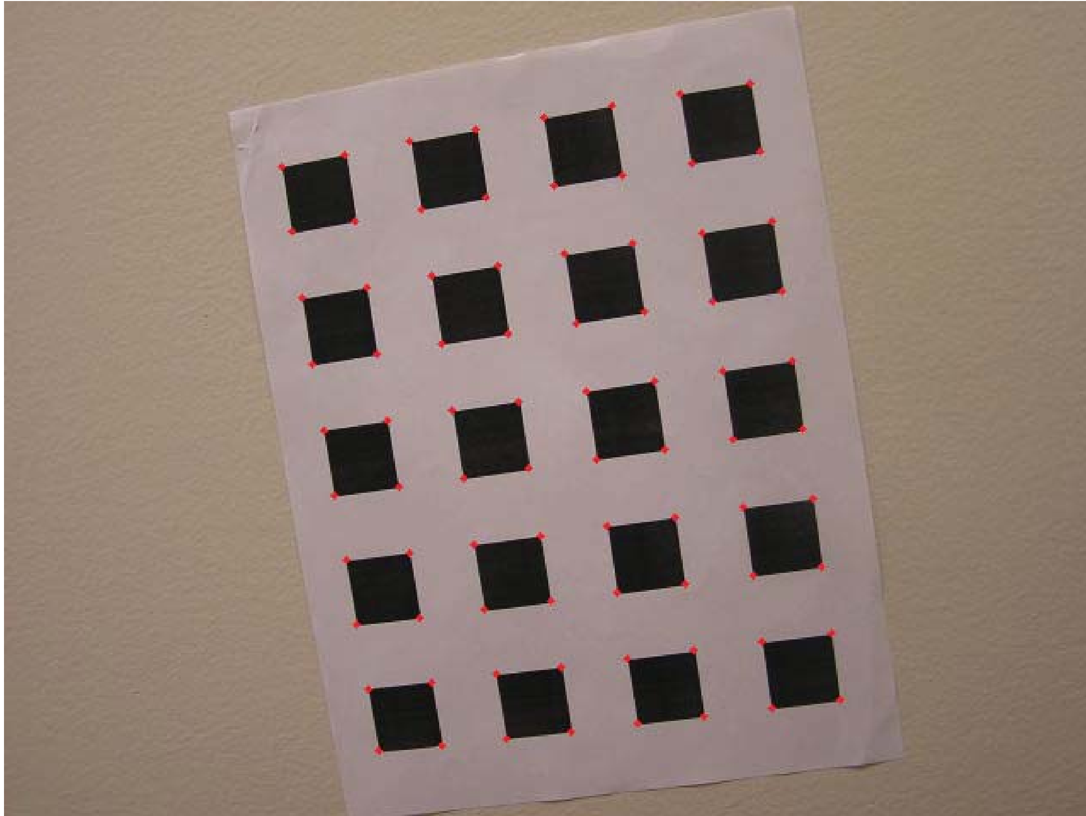
$$\text{External parameters:}[R|\vec{t}] = \begin{bmatrix} 0.95737211 & 0.26821168 & -0.10724339 & | & -5.12534494 \\ 0.25537448 & -0.95940575 & -0.11968494 & | & 2.11675885 \\ -0.13499082 & 0.0871958 & -0.98700272 & | & 14.33029119 \end{bmatrix}$$

Figure 5: Reprojected result and the external parameters for frame1 of my data set.



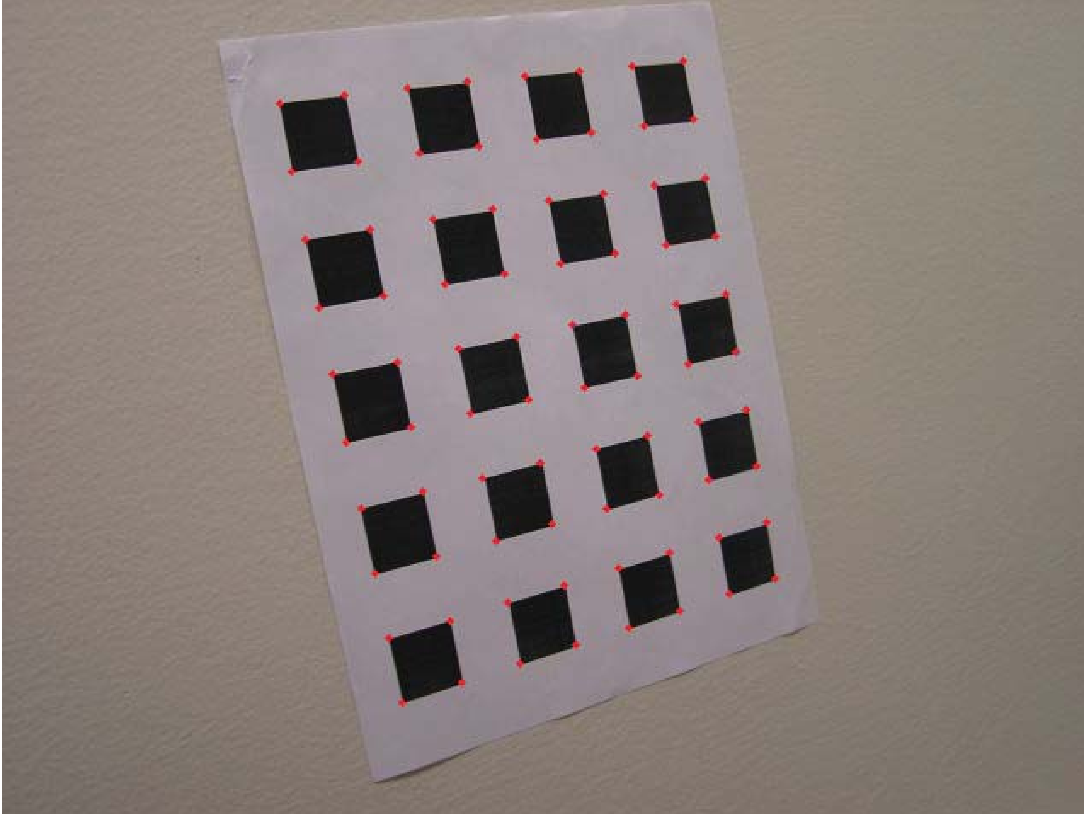
$$\text{External parameters:}[R|\vec{t}] = \begin{bmatrix} 0.86638487 & 0.25469514 & 0.42954353 & | & -3.37431283 \\ 0.33312942 & -0.93557163 & -0.11717729 & | & 2.31362012 \\ 0.37202425 & 0.24461422 & -0.89541155 & | & 12.97031149 \end{bmatrix}$$

Figure 6: Reprojected result and the external parameters for frame2 of my data set.



$$\text{External parameters:}[R|\vec{t}] = \begin{bmatrix} 0.98067968 & -0.15187249 & -0.12329685 & | & -2.65356255 \\ -0.14402305 & -0.98707348 & 0.07030859 & | & 5.23780151 \\ -0.132381 & -0.05119262 & -0.98987605 & | & 18.76736432 \end{bmatrix}$$

Figure 7: Reprojected result and the external parameters from the provided data set.



$$\text{External parameters:}[R|\vec{t}] = \begin{bmatrix} 0.87925714 & -0.18547251 & 0.43875601 & | & -2.21539713 \\ -0.20586596 & -0.97857956 & -0.00111792 & | & 4.40755262 \\ 0.42956501 & -0.08934199 & -0.89860554 & | & 18.40556658 \end{bmatrix}$$

Figure 8: Reprojected result and the external parameters from the provided data set.

References

[1] Homework 7 problem set: http://web.ics.purdue.edu/~kim497/ece661/homework/hw7.pdf

[2] Zhengyou Zhang, "A Flexible New Technique for Camera Calibration", Technical Report MSR-TR-98-71, Microsoft Corporation

[3] cv.HoughLines2 OpenCV documentation and tutorial, http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html#hough-lines

[4] cv.FindCornerSubPix OpenCV documentation, http://opencv.willowgarage.com/documentation/python/imgproc_feature_detection.html?highlight=findcornersubpix#FindCornerSubPix

[5] cv.Canny OpenCV documentation, http://opencv.willowgarage.com/documentation/python/imgproc_feature_detection.html?highlight=canny#Canny

[6] cv.Rodrigues2 OpenCV documentation, http://opencv.willowgarage.com/documentation/python/calib3d_camera_calibration_and_3d_reconstruction.html?highlight=rodrigues#Rodrigues2

[7] Numpy and Scipy Documentation: Non-Linear Least Squares http://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html#least-square-fitting-leastsq

[8] R. Hartley, and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd, Cambridge University Press., 2004.

A Source Code Listing

```
1  #!/usr/bin/python
2  #
3  #  ECE 661 Homework 7
4  #  Instructor: Prof. Avi Kak
5  #  TA: Dave Kim
6  #
7  #  Tommy Chang
8  #
9  # 2012-11-06 (Tue)
10 # http://web.ics.purdue.edu/~kim497/ece661/homework/hw7.pdf
11
12 from pylab import *
13 import cv, cv2, time, os, re
14 from scipy.optimize import leastsq
15
16 global font
17 font = cv.InitFont (cv.CV_FONT_HERSHEY_SIMPLEX, 0.5, 0.5, thickness=2)
18
19 windowNames = set ([])
20
21 def CloseWindowNow (name="window"):
22     cv.DestroyWindow (name)
23     for idx in range(0,4): # hack to close the window properly for opencv 2.1
24         cv.WaitKey (1000)
25
26 def CloseAllWindowNow ():
```

```
    global windowNames
    nameList = list (windowNames)
    for idx in range (len (nameList)):
        cv.DestroyWindow (nameList[idx])
    for idx in range(0,4): # hack to close the window properly for opencv 2.1
        cv.WaitKey (1000)

def DisplayImage (imgOrMat, name=None):
    global windowNames
    if (name == None):
        # functionCallStr = inspect.stack()[1][4][0]
        # name = re.sub (r'.*\((.*?)\).*', r'\1', functionCallStr)
        # name = name.rstrip("\n")
        name = "window"
    windowNames.add (name)
    cv.ShowImage (name, imgOrMat)
    cv.WaitKey (1000)

def LoadImage (imgFileName):
    image = cv.LoadImageM (imgFileName, cv.CV_LOAD_IMAGE_COLOR)
    return image

def GetImageFiles (dirName, extName):
    imageFiles = os.listdir (dirName)
    imageFiles = filter (lambda x: re.match(r'.*\.'+extName, x), imageFiles)
    imageFiles = [dirName + "/" + imageFiles[idx] for
                  idx in range(len(imageFiles))]
    return imageFiles


def OverlayHoughLines (lines, image):
    newImage = cv.CloneMat (image)
    for idx in xrange (len (lines)):
        # get line in polar coordinate:
        rho   = lines [idx][0]
        theta = lines [idx][1]

        # the center point on the line:
        cosa = cos(theta); sina = sin(theta)
        x0 = cosa*rho
        y0 = sina*rho;

        # Need the two extreme end points:
        # since dot product gives us: x cosa + y sina = r
        # the slope intersect-form is : y = (-cosa/sina) x + (r/sina)
        # solving the slope as : deltay/deltax = (-cosa/sina)
        # Note: line can draw outside the image border.
        pt1x = int (x0 + 1000 * sina)
        pt1y = int (y0 - 1000 * cosa)
        pt2x = int (x0 - 1000 * sina)
        pt2y = int (y0 + 1000 * cosa)
        cv.Line (newImage, (pt1x, pt1y), (pt2x, pt2y), cv.RGB(0, 0, 255), 2)
    return newImage

def OverlayLabelCorners (corners, image, color=cv.RGB(0, 0, 255)):
    image = cv.CloneMat (image)
    nRows = len (corners)
    nCols = len (corners[0])
    cornerId = 0
    for rIdx in xrange (nRows):
        for cIdx in xrange (nCols):
            center = tuple (map (int, corners[rIdx][cIdx]))
```

```
            cv.Circle (image, center, 4, (0, 0, 255), -1)
            cv.PutText (image, str(cornerId), center, font, cv.RGB(0, 255, 0))
            cornerId += 1
    return image

def OverlayCorners (corners, image, size=4, color=cv.RGB(0, 0, 255)):
    # here corners is simply a list of corners in (x,y)
    image = cv.CloneMat (image)
    for idx in xrange (len (corners)):
        center = tuple (map (int, corners[idx]))
        cv.Circle (image, center, size, (0, 0, 255), -1)
    return image

def GetHoughLineinHC (houghLine):
    rho = houghLine[0]
    theta = houghLine[1]
    pt0 = array ([rho * cos(theta), rho * sin(theta), 1.0])
    pt1 = array ([pt0[0] + 100 * sin(theta),
                  pt0[1] - 100 * cos(theta),
                  1.0])
    lineHC = cross (pt0, pt1)
    return lineHC

def RealCoord (v):
    # convert a image coordinate to Homogeneous coordinate:
    if type (v) == ndarray:
        if v.ndim == 1:
            v[0] /= v[2]
            v[1] /= v[2]
            v[2] = 1
        else:
            v[0][0] /= v[2][0]
            v[1][0] /= v[2][0]
            v[2][0] = 1
    elif type (v) == cv2.cv.cvmat:
        v[0,0] /= v[2,0]
        v[1,0] /= v[2,0]
        v[2,0] = 1
    else:
        assert (0)
    return v

def FindCheckerLines (lines, checkerDim, imageDim):
    # checkerDim is (m, n) where m is the number rows (eg, 4) of black
    # squares.  and n is number of columns (eg 5) of black squares.
    #
    # imageDim is dimension of the image (eg, 480 rows, 640 cols)
    #
    # This function returns a list of houghlines.  Non-maxima
    # suppression is done by making sure no lines of the same group
    # (vertical or horizontal) intersect on the image.  The line's
    # quality is determined by the order in the lines.  The longest
    # line appears first. (ie. low "badness" value)
    #
    # Group lines into either horizontal or vertical:
    thetas = array ([lines[idx][1] for idx in range(len (lines))])
    thetas -= pi/2             # horizontal lines is [-45..45]
    lineBadness = range (len (thetas))
    hIdxs = map (int, where (abs(thetas) < pi/4)[0])
    horzLines = [lines[idx] for idx in hIdxs]
    hLineBadness = [lineBadness[idx] for idx in hIdxs]
    vIdxs = map (int, where (abs(thetas) >= pi/4)[0])
```

```
    vertLines = [lines[idx] for idx in vIdxs]
    vLineBadness = [lineBadness[idx] for idx in vIdxs]
    assert (len (hIdxs) + len (vIdxs) == len (thetas))

    # Sort each group by distance, aslo include the badness measure:
    hLineAndBadness = zip (hLineBadness, horzLines)
    horzLines = sorted (horzLines, key=lambda elm: elm[0] * sin (elm[1]))
    hLineAndBadness = sorted (hLineAndBadness,
                              key=lambda elm: elm[1][0] * sin (elm[1][1]))
    vLineAndBadness = zip (vLineBadness, vertLines)
    vertLines = sorted (vertLines, key=lambda elm: elm[0] * cos (elm[1]))
    vLineAndBadness = sorted (vLineAndBadness,
                              key=lambda elm: elm[1][0] * cos (elm[1][1]))
    #
    # Supress bad lines, group based on the line nearness and intersection:
    def GroupLines (hvLinesWithBadness, sinCos, minSep):
        uniqueLines = []
        uniqueLinesBadness = []
        # print "lines in a group", len (hvLinesWithBadness)
        hvLine = hvLinesWithBadness[0][1]
        uniqueLines.append (hvLine)
        badness = hvLinesWithBadness[0][0]
        uniqueLinesBadness.append (badness)
        for idx in xrange (1, len (hvLinesWithBadness)):
            hvLine = hvLinesWithBadness[idx][1]
            badness = hvLinesWithBadness[idx][0]
            lineHC = GetHoughLineinHC (hvLine)

            # check intersection
            curLineHC = GetHoughLineinHC (hvLine)
            uniqueLineHC = GetHoughLineinHC (uniqueLines[-1])
            corner = cross (curLineHC, uniqueLineHC)
            RealCoord (corner)
            # print "Corner", corner
            if (((corner[0] >=0) and (corner[0] < imageDim[1])) and
                ((corner[1] >=0) and (corner[1] < imageDim[0]))):
                # print "skip corner cross"
                if (uniqueLinesBadness[-1] > badness):
                    uniqueLines[-1] = hvLine
                    uniqueLinesBadness[-1] = badness;
                continue

            # check distance
            uniqueLineDist = uniqueLines[-1][0] * sinCos (uniqueLines[-1][1])
            curDist = hvLine[0] * sinCos (hvLine[1])
            deltaDist = abs (uniqueLineDist - curDist)
            # print "deltaDist=", deltaDist
            if (deltaDist <= minSep):
                # print "skip near"
                if (uniqueLinesBadness[-1] > badness):
                    uniqueLines[-1] = hvLine
                    uniqueLinesBadness[-1] = badness;
                continue

            # new group starts:
            uniqueLines.append (hvLine)
            uniqueLinesBadness.append (badness)
            sepDist = deltaDist
            # print "newline ", len (uniqueLines)
            # print hvLine
        return (uniqueLines, sepDist)

```

```
213
214     # find the best separation distance:
215     minSep = 1.0
216     isDone = False
217     while isDone == False:
218         (uniqueHlines, hSepDist) = GroupLines (hLineAndBadness, sin, minSep)
219         (uniqueVlines, vSepDist) = GroupLines (vLineAndBadness, cos, minSep)
220         if ((len (uniqueHlines) == 2 * checkerDim[0]) and
221             (len (uniqueVlines) == 2 * checkerDim[1])):
222             isDone = True
223         elif ((len (uniqueHlines) < 2 * checkerDim[0]) or
224               (len (uniqueVlines) < 2 * checkerDim[1])):
225             print "filter too much Can't separate???"
226             assert (0)
227             isDone = True
228         else:
229             minSep += 1
230             print "minSep", minSep
231
232     assert (len (uniqueHlines) >= 2 * checkerDim[0])
233     assert (len (uniqueVlines) >= 2 * checkerDim[1])
234     bestLines = uniqueHlines + uniqueVlines
235     sepDist = int ((hSepDist + vSepDist) / 2.0)
236     print "sepDist=", sepDist
237     return (bestLines, sepDist)
238
239
240 def ReshapeCorners (cornersSet, dim):
241     # dim = (number or rows, number of columns)
242     #
243     # convert 1D to 2D list:
244     nRows = dim[0]
245     nCols = dim[1]
246     corners = [[cornersSet[i*nCols+j] for j in range(nCols)]
247                for i in range(nRows)]
248     return corners
249
250
251 def FindCheckerCorners (bestLines, sepDist, checkerDim):
252     # checkerDim is (m, n) where m is the number rows (eg, 4) of black
253     # squares.  and n is number of columns (eg 5) of black squares.
254     #
255     # This function returns a 2D array.  Each elment is a corner in a
256     # tuple (x,y). Note: x is column and y is row.  Use
257     # ReshapeCorners() to convert to 1D array if needed.
258     #
259     # Group lines into either horizontal or vertical:
260     thetas = array ([bestLines[idx][1] for idx in range(len (bestLines))])
261     thetas -= pi/2
262     hIdxs = where (abs(thetas) < pi/4)[0]
263     horzLines = [bestLines[idx] for idx in hIdxs]
264     vIdxs = where (abs(thetas) >= pi/4)[0]
265     vertLines = [bestLines[idx] for idx in vIdxs]
266     assert (len (hIdxs) + len (vIdxs) == len (thetas))
267
268     nHlines = len (horzLines)
269     nVlines = len (vertLines)
270     print nHlines, nVlines
271     assert (nHlines == 2 * checkerDim[0])
272     assert (nVlines == 2 * checkerDim[1])
273
274     # Find the intersection of lines:
```

```
275     corners = [[() for j in range(nVlines)] for i in range(nHlines)]
276     horzLines = sorted (horzLines, key=lambda elm: elm[0] * sin (elm[1]),
277                         reverse=True)
278     vertLines = sorted (vertLines, key=lambda elm: elm[0] * cos (elm[1]),
279                         reverse=False)
280     for hIdx in xrange (nHlines):
281         hLineHC = GetHoughLineinHC (horzLines [hIdx])
282         for vIdx in xrange (nVlines):
283             vLineHC = GetHoughLineinHC (vertLines [vIdx])
284             # do line intersection:
285             corner = cross (hLineHC, vLineHC)
286             RealCoord (corner)
287             corners [hIdx][vIdx] = (corner[0], corner[1])
288
289     return corners
290
291
292 def RefineCorners (gray, corners, wSize):
293     corners_1D = [corner for row in corners for corner in row]
294     corners_1D = cv.FindCornerSubPix (gray, corners_1D,
295                                       (wSize/2, wSize/2), (2, 2),
296                                       (cv.CV_TERMCRIT_ITER |
297                                        cv.CV_TERMCRIT_EPS,
298                                        10, 0.01))
299     corners_2D = ReshapeCorners (corners_1D, (len(corners), len(corners[0])))
300     return corners_2D
301
302 def SolveHomographyByDLT_V2 (xyDataSet):
303     # Solve the Homography by using Least Square Method for
304     # homogeneous equations.
305     #
306     # xyDataSet is a set of correspondence pairs. It is a list of
307     #  sublists of the form: [(X1, Y1), (X2, Y2)] Here, (X1, Y1), (X2,
308     #  Y2) is a correspondance pair.
309     #
310     # We want to construct a H such that Data2 = H Data1
311     nData = len (xyDataSet)
312     assert (nData >= 5)
313
314     # precondition the data first to avoid numerical precision cutoff:
315     def PreconditionData (xyDataSet):
316         # xyDataSet is a set of correspondence pairs. It is a list of
317         #  sublists of the form: [(X1, Y1), (X2, Y2)] Here, (X1, Y1), (X2,
318         #  Y2) is a correspondance pair.
319         #
320         # 1.) find the center:
321         nData = len (xyDataSet)
322         data1X = [xyDataSet[idx][0][0] for idx in range(nData)]
323         data1Y = [xyDataSet[idx][0][1] for idx in range(nData)]
324         data2X = [xyDataSet[idx][1][0] for idx in range(nData)]
325         data2Y = [xyDataSet[idx][1][1] for idx in range(nData)]
326         all1X = array (data1X)
327         all1Y = array (data1Y)
328         mean1X = int (all1X.mean())
329         mean1Y = int (all1Y.mean())
330         all2X = array (data2X)
331         all2Y = array (data2Y)
332         mean2X = int (all2X.mean())
333         mean2Y = int (all2Y.mean())
334         # 2.) make data relative to the center:
335         def offsetCoord (datum):
336             X1 = datum[0][0]
```

```
337             Y1 = datum[0][1]
338             X2 = datum[1][0]
339             Y2 = datum[1][1]
340             return [(X1 - mean1X, Y1 - mean1Y),
341                     (X2 - mean2X, Y2 - mean2Y)]
342         xyDataSet = map (offsetCoord, xyDataSet)
343         # 3.) The pre-conditional transformation:
344         H1_cond = array ([[1, 0, -mean1X],
345                           [0, 1, -mean1Y],
346                           [0, 0, 1]], dtype=float)
347         H2_cond = array ([[1, 0, -mean2X],
348                           [0, 1, -mean2Y],
349                           [0, 0, 1]], dtype=float)
350         return (xyDataSet, H1_cond, H2_cond)
351     (xyDataSet_cond, H1_cond, H2_cond) = PreconditionData (xyDataSet);
352 
353     # Generate the design matrix A:
354     def GenerateArow (pair):
355         X1 = pair[0][0]
356         Y1 = pair[0][1]
357         X2 = pair[1][0]
358         Y2 = pair[1][1]
359         return [0,   0,   0, -X1, -Y1, -1,  Y2*X1,  Y2*Y1,  Y2,
360                 X1, Y1,   1,   0,   0,  0, -X2*X1, -X2*Y1, -X2]
361     A = array (map (GenerateArow, xyDataSet_cond), dtype=float)
362     A = A.reshape (nData*2, 9)
363 
364     # SVD decomposition: (becareful, if useing eigen decomposition,
365     # the eigen vectors are in columns, not row)
366     (U, D, V) = svd (A)
367     minSvalIdx = D.argmin()
368     # (D_, U_) = eig (dot(transpose(A), A))
369     # U_ = transpose (U_)
370     # minEvalIdx = D_.argmin()
371 
372     # get the minimum eigen vector:
373     assert (D[minSvalIdx] > 0)
374     minEigVec = V[minSvalIdx]
375     H_ = minEigVec.reshape (3, 3)
376 
377     # undo the pre-data conditioning
378     H = dot (inv (H2_cond), dot (H_, H1_cond))
379     return H
380 
381 def GetCorrespondencePairs (corners_2D, cornerSep):
382     corners_1D = [corner for row in corners_2D for corner in row]
383     nRows = len (corners_2D)
384     nCols = len (corners_2D[0])
385     cornerPairs = [()] * (nRows * nCols)
386     idx = 0
387     for rIdx in range (nRows):
388         for cIdx in range (nCols):
389             x_model = float (cIdx) * cornerSep
390             y_model = float (rIdx) * cornerSep
391             cornerPairs [idx] = [(x_model, y_model),
392                                  corners_2D[rIdx][cIdx]]
393             idx += 1
394     return cornerPairs
395 
396 def SolveIAC (Hs):
397     # Solve for the image of abolute conic (IAC), aka w.
398     # Here Hs is a list of homographies, each from a image.  image = H * world
```

```
399     # See Zhengyou Zhang's paper about Camera Calibration.
400     def GenerateTwoRows (H):
401         def GetV (hi_vec, hj_vec):
402             return array ([hi_vec[0] * hj_vec[0],
403                            hi_vec[0] * hj_vec[1] + hi_vec[1] * hj_vec[0],
404                            hi_vec[0] * hj_vec[2] + hi_vec[2] * hj_vec[0],
405                            hi_vec[1] * hj_vec[1],
406                            hi_vec[1] * hj_vec[2] + hi_vec[2] * hj_vec[1],
407                            hi_vec[2] * hj_vec[2]])
408         h1_vec = H[:,0]
409         h2_vec = H[:,1]
410         v12_vec = GetV (h1_vec, h2_vec)
411         v11_vec = GetV (h1_vec, h1_vec)
412         v22_vec = GetV (h2_vec, h2_vec)
413         return array ([v12_vec,
414                        v11_vec - v22_vec])
415     assert (len (Hs) >= 3)      # we need at least 3 images to solve IAC
416     V = array (map (GenerateTwoRows, Hs), dtype=float)
417     V = [row for twoRows in V for row in twoRows] # flatten
418 
419     # SVD decomposition: (becareful, if useing eigen decomposition,
420     # the eigen vectors are in columns, not row)
421     (U, D, Vt) = svd (V)
422     minSvalIdx = D.argmin()
423 
424     # get the minimum eigen vector:
425     assert (D [minSvalIdx] > 0)
426     minEigVec = Vt [minSvalIdx]
427 
428     # reconstruct the IAC (symmetric)
429     w = zeros ((3,3))
430     w[0,0] = minEigVec [0]
431     w[0,1] = minEigVec [1]
432     w[0,2] = minEigVec [2]
433     w[1,0] = w[0,1]
434     w[1,1] = minEigVec [3]
435     w[1,2] = minEigVec [4]
436     w[2,0] = w[0,2]
437     w[2,1] = w[1,2]
438     w[2,2] = minEigVec [5]
439 
440     return w
441 
442 def SolveCameraCalibrationMatrix (w):
443     # w is the Image of the Absolute Conic (IAC)
444     # See Zhengyou Zhang's paper about Camera Calibration.
445     yo = (w[0,1]*w[0,2] - w[0,0]*w[1,2]) / (w[0,0]*w[1,1] - w[0,1]*w[0,1])
446     lam = w[2,2] - \
447         (w[0,2]*w[0,2] + yo * (w[0,1]*w[0,2] - w[0,0]*w[1,2])) / w[0,0]
448     alpha_x = sqrt (lam / w[0,0])
449     alpha_y = sqrt (lam*w[0,0] / (w[0,0]*w[1,1] - w[0,1]*w[0,1]))
450     s = - (w[0,1] * alpha_x*alpha_x * alpha_y) / lam
451     xo = (s*yo) / alpha_y - (w[0,2] * alpha_x*alpha_x) / lam
452     K = zeros ((3,3))
453     K[0,0] = alpha_x
454     K[0,1] = s
455     K[0,2] = xo
456     K[1,0] = 0
457     K[1,1] = alpha_y
458     K[1,2] = yo
459     K[2,0] = 0
460     K[2,1] = 0
```

```
461     K[2,2] = 1
462     return K
463
464 def SolveExternalParameters (K, Hs):
465     # K is the intrisic parameter, Hs is a list of planar
466     # homographies.
467     #
468     # Return a list of 3x4 matrix homogeous matrix T that is the world
469     # pose in camera's point of view.  ie., T takes world coordinate
470     # and transform to camera coordinate centered at the camera center
471     # (i.e., center of projection)
472     #
473     K_inv = inv (K)
474     def SolveT (H):
475         h1_vec = H[:,0]
476         h2_vec = H[:,1]
477         h3_vec = H[:,2]
478
479         # Since tz is involved in normalizing the homogeous coordinate
480         # later, we make the normalization divide by a positive value.
481         # This will make the camera frame and and the external
482         # parameters consistant.
483         t_vec = dot (K_inv, h3_vec)
484         mag = norm (dot (K_inv, h1_vec))
485         if (t_vec[2] < 0):
486             mag = -mag
487         t_vec /= mag
488         r1_vec = dot (K_inv, h1_vec) / mag
489         r2_vec = dot (K_inv, h2_vec) / mag
490         r3_vec = cross (r1_vec, r2_vec)
491
492         # condition the rotation:
493         R = zeros ((3,3))
494         R[:,0] = r1_vec
495         R[:,1] = r2_vec
496         R[:,2] = r3_vec
497         (U, D, Vt) = svd (R)
498         R = dot (U, Vt)
499
500         # put together a 4x4 transofmration matrix:
501         Rt = zeros ((3,4))
502         Rt[:,0] = R[:,0]
503         Rt[:,1] = R[:,1]
504         Rt[:,2] = R[:,2]
505         Rt[:,3] = t_vec
506         # T = append (T, [[0, 0, 0, 1]], 0)
507
508         return Rt
509     Rts = map (SolveT, Hs)
510     return Rts
511
512 def ApplyRadialDist_V2 (projPtVec, K, k1, k2):
513     # projPtVec is a 2-D vector
514     xo = K[0,2]
515     yo = K[1,2]
516
517     # compute the radius:
518     dx = (projPtVec[0] - xo)
519     dy = (projPtVec[1] - yo)
520     rSqr = dx*dx + dy*dy
521     rSqrSqr = rSqr * rSqr
522
```

```
523     # apply distortion model:
524     distModel = (1 + k1*rSqr + k2*rSqrSqr)
525     return [xo + dx * distModel, yo + dy * distModel]
526
527 def ApplyRadialDist (projPtVec, K, k1, k2):
528     # projPtVec is a 2-D vector, make it 3D
529     projPtVec = [projPtVec[0], projPtVec[1], 1.0]
530
531     # back-project pixel to xyz:
532     K_inv = array ([[1/K[0,0], 0, -K[0,2]/K[0,0]],
533                     [0, 1/K[1,1], -K[1,2]/K[1,1]],
534                     [0, 0, 1]])
535     xyz = dot (K_inv, projPtVec)
536
537     # compute the radius:  (no center)
538     rSqr = xyz[0]*xyz[0] + xyz[1]*xyz[1]
539     rSqrSqr = rSqr * rSqr
540
541     # apply distortion model:  (set z to 1)
542     xyz_dist = xyz * (1 + k1*rSqr + k2*rSqrSqr)
543     xyz_dist [2] = 1.0;
544
545     # reproject:
546     reprojPtVec = dot (K, xyz_dist)
547
548     return [reprojPtVec[0], reprojPtVec[1]]
549
550 def RefineCalParameters (K, Rts, xyDataSets, modelRadial=True, k1=0, k2=0):
551     # This function uses the non-linear Least squares mehtod to do a
552     # global optimization on all parameters of the calibration by
553     # minimizing the re-projection error.  The parameters includes the
554     # radial distortion (k1 and k2).
555     #
556     # K is the camera calibration matrix.  Ts is a list of external
557     # parameters as a 3x4 homogeous transform, one per image.
558     #
559     # xyDataSets is a list of sublists.  The sublist contains the
560     # points on the model plane and the points on the image.
561     #
562     # The cost function is the residuals since leastsq will do the
563     # square and sum.
564     numIntrinsicParams = 7
565     def ConvertToParams (K, Rts, k1, k2):
566         r_vec = zeros ((1,3))   # need to be 2D for Rodrigues2
567         rtsIdxStart = numIntrinsicParams
568         params = zeros (rtsIdxStart + len (Rts) * 6)
569         params[0:3] = K[0,:]
570         params[3] = K[1,1]
571         params[4] = K[1,2]
572         params[5] = k1
573         params[6] = k2
574         for rtIdx in xrange (len (Rts)):
575             R = copy (Rts [rtIdx][0:3,0:3])
576             cv.Rodrigues2 (cv.fromarray (R), cv.fromarray (r_vec))
577             params [rtsIdxStart : rtsIdxStart+3] = r_vec[0] # turn r_vec to 1-D
578             rtsIdxStart += 3
579             t_vec = Rts [rtIdx][:,3]
580             params [rtsIdxStart : rtsIdxStart+3] = t_vec
581             rtsIdxStart += 3
582         return params
583     def ConvertFromParams (params):
584         k_vec = params[0:5]
```

```
585         K = array ([[k_vec[0], k_vec[1], k_vec[2]],
586                      [0       , k_vec[3], k_vec[4]],
587                      [0       , 0       , 1]])
588         k1 = params [5]
589         k2 = params [6]
590         rtParams = params [numIntrinsicParams:]
591         assert (mod (len (rtParams), 6) == 0)
592         rtParams = rtParams.reshape (len (rtParams)/6, 6)
593         Rts = []
594         r_vec = zeros ((1,3))   # need to be 2D for Rodrigues2
595         for rtIdx in xrange (len (rtParams)):
596             params = rtParams [rtIdx]
597             r_vec[0] = copy (params[0:3]) # r_vec is 2D
598             R = zeros ((3,3))       # the rotation 3x3 matrix
599             cv.Rodrigues2 (cv.fromarray (r_vec), cv.fromarray (R))
600             t_vec = params[3:6]     # the tranlstion vector
601             Rt = zeros ((3,4))
602             Rt[:,0] = R[:,0]
603             Rt[:,1] = R[:,1]
604             Rt[:,2] = R[:,2]
605             Rt[:,3] = t_vec
606             Rts.append (Rt)
607         return (K, Rts, k1, k2)
608     def residuals (params):
609         # unpack the calibration parameters:
610         (K, Rts, k1, k2) = ConvertFromParams (params)
611         # xo = K[0,2];         yo = K[1,2] # the center pixel of image
612         allRes = []
613         for rtIdx in xrange (len (Rts)):
614             # R = Rts [rtIdx][0:3,0:3]
615             # t_vec = Rts [rtIdx][:,3]
616             # Rt = append (R, transpose ([t_vec]), 1)
617             Rt = Rts [rtIdx]
618             P = dot (K, Rt) # the projection matrix
619 
620             # get points on the model plan as 4-vectors and image points as
621             # 3-vector, both in homogeous coordinates
622             xyDataSet = xyDataSets[rtIdx]
623             wrldPts = [row[0] for row in xyDataSet]
624             wrldVecs = array ([[row[0], row[1], 0, 1] for row in wrldPts])
625             imgPts = [row[1] for row in xyDataSet]
626             imgVecs = array ([[row[0], row[1], 1] for row in imgPts])
627             imgVecs = imgVecs [:,0:2] # get rid of last dim
628 
629             # compute the re-projection residual:
630             projdImgVecs = transpose (dot (P, transpose (wrldVecs)))
631             map (RealCoord, projdImgVecs)
632             projdImgVecs = projdImgVecs [:,0:2] # get rid of last dim
633             if modelRadial:
634                 projdImgVecs = map (lambda projPtVec :
635                                     ApplyRadialDist (projPtVec, K, k1, k2),
636                                     projdImgVecs)
637             res = projdImgVecs - imgVecs
638             allRes.append (res)
639 
640         res = [elm for resSet in allRes for row in resSet for elm in row]
641         print max (res)
642         return res
643 
644     # Calculate the optimized solution:
645     initParams = ConvertToParams (K, Rts, k1, k2)
646     plsq = leastsq (residuals, initParams)
```

```
647     finalParams = plsq[0]
648     return ConvertFromParams (finalParams)
649
650
651 if __name__=='__main__':
652     #
653     # Extract corners:
654     #
655     cornerSep = 1.0  # separation between adjacent corner (eg. 1 inch)
656     Hs = []
657     xyDataSets = []
658     imageFiles = GetImageFiles ('testImages', 'jpg')
659     checkerDim = (5, 4)
660     imageFiles = GetImageFiles ('myImages', 'JPG')
661     checkerDim = (4, 5)   # side-way
662     counter = 1
663     for imageFile in imageFiles:
664         print imageFile
665         image = LoadImage (imageFile)
666         imageDim = (image.rows, image.cols)
667         gray = cv.CreateMat (image.height, image.width, cv.CV_8UC1)
668         cv.CvtColor (image, gray, cv.CV_RGB2GRAY)
669         edges = cv.CreateMat (gray.height, gray.width, cv.CV_8UC1)
670         cv.Canny (gray, edges, 255*1.5, 255)
671
672         # Perform hough transform with 1 degree and 1 pixel
673         # resolution.  Note: the output are sorted by the accumulator
674         # values.  A line must have at least 50 pixels on it:
675         lines = cv.HoughLines2 (edges, cv.CreateMemStorage(),
676                                 cv.CV_HOUGH_STANDARD, 1, pi/180, 50)
677         (bestLines, sepDist) = FindCheckerLines (lines, checkerDim, imageDim)
678         houghResultImg = OverlayHoughLines (bestLines, image)
679         allCorners_2D = FindCheckerCorners (bestLines, sepDist, checkerDim)
680         initCornersImg = OverlayLabelCorners (allCorners_2D, image)
681         allCorners_2D = RefineCorners (gray, allCorners_2D, sepDist)
682         refinedCornersImg = OverlayLabelCorners (allCorners_2D, image)
683
684         #
685         # Solve the homography to the model plane:
686         #
687         xyDataSet = GetCorrespondencePairs (allCorners_2D, cornerSep)
688         H = SolveHomographyByDLT_V2 (xyDataSet)
689         Hs.append (H)
690         xyDataSets.append (xyDataSet)
691
692         #
693         # Visual display and save images:
694         #
695         # if (counter < 5):
696         #     DisplayImage (edges)
697         #     cv.SaveImage ("edges_" + str(counter) + ".png", edges)
698         #     cv.SaveImage ("image_" + str(counter) + ".png", image)
699         #     DisplayImage (houghResultImg)
700         #     cv.SaveImage ("hough_" + str(counter) + ".png", houghResultImg)
701         #     DisplayImage (initCornersImg)
702         #     cv.SaveImage ("initCorners_" + str(counter) + ".png",
703         #                   initCornersImg)
704         #     DisplayImage (refinedCornersImg)
705         #     cv.SaveImage ("refinedCorners_" + str(counter) + ".png",
706         #                   refinedCornersImg)
707         counter += 1
708         # raw_input ("press enter to continue")
```

```
709
710     #
711     # Solve for IAC
712     #
713     w = SolveIAC (Hs)
714     K = SolveCameraCalibrationMatrix (w)
715     Rts = SolveExternalParameters (K, Hs)
716     (newK, newRts, k1, k2) = RefineCalParameters (K, Rts, xyDataSets)
717     # (newK, newRts, k1, k2) = (K, Rts, 0, 0)
718
719     #
720     # Reprojection
721     #
722     for imgIdx in xrange (len (imageFiles)):
723         imageFile = imageFiles [imgIdx]
724         print imageFile
725         image = LoadImage (imageFile)
726
727         P = dot (newK, newRts[imgIdx]) # the projection matrix
728
729         xyDataSet = xyDataSets [imgIdx]
730         wrldPts = [row[0] for row in xyDataSet]
731         wrldVecs = array ([[row[0], row[1], 0, 1] for row in wrldPts])
732         imgPts = [row[1] for row in xyDataSet]
733         imgVecs = array ([[row[0], row[1], 1] for row in imgPts])
734         imgVecs = imgVecs [:,0:2] # get rid of last dim
735
736         # compute the re-projection:
737         projdImgVecs = transpose (dot (P, transpose (wrldVecs)))
738         map (RealCoord, projdImgVecs)
739         projdImgVecs = projdImgVecs [:,0:2] # get rid of last dim
740         projdImgVecs = map (lambda projPtVec : ApplyRadialDist (projPtVec,
741                                                                 newK, k1, k2),
742                             projdImgVecs)
743         reProjResultImg = OverlayCorners (projdImgVecs, image, 2)
744         res = projdImgVecs - imgVecs
745         res = [row[0] for row in res]
746         maxRes = max (res)      # max reprojection error
747         print maxRes
748
749         #
750         # Visual display and save images:
751         #
752         if (imgIdx < 5):
753             DisplayImage (reProjResultImg)
754             cv.SaveImage ("reproj_" + str(imgIdx) + ".png", reProjResultImg)
755         # raw_input ("press enter to continue")
```