ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

Theory

Corner detection

The canny edge detector from OpenCV is used to find edges, which are used to find hough lines in the image. To find hough lines, the angle resolution used is pi/180, and number of votes required for a line to be established is 50. Next, if two lines are very similar (difference in r being less than a threshold of 15, and difference in theta being less than a threshold of 1 radian), one of the duplicates is removed.

Then I classify the lines as horizontal and vertical by looking at their angle. The intersection of any horizontal line with any vertical line gives us a corner. Since the elimination of duplicate lines may not have been efficient, I also attempt to eliminate duplicate corners by removing one of two corners if the Manhattan distance between them is too less.

Camera calibration: Zhang's algorithm

Finding intrinsic parameters:

Images of the calibration pattern from at least 3 different orientations from the camera are required. To find world points, an image of the calibration pattern is taken straight from the front in a way such that the x and y axes roughly correspond to the horizontal and vertical axes of the world, and the principal axis is roughly perpendicular to the wall on which the pattern is mounted. Homographies are found between the world points and the image points of this pattern.

The image of the absolute conic is found as follows:

Vb = 0,

Where b = $\begin{bmatrix} w11 \\ w12 \\ w22 \\ w13 \\ w23 \\ w33 \end{bmatrix}$, a rearranged version of the w matrix (image of absolute conic) that we are solving for.

V = $\begin{bmatrix} V\_1 \\ V\_2 \\ . \\ . \\ . \\ V\_n \end{bmatrix}$, where n is number of images, and V_i, stands for the V-matrix of the ith image.

The V-matrix of an image is given by V_{image} = $\begin{bmatrix} transpose(V_{12)} \\ transpose(V_{11} - V_{22}) \end{bmatrix}$

Where V_{ij} = $\begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix}$, where h_{ij} is the element in the ith column and jth row of h, the homography between the image and the world.

The intrinsic parameter matrix is given by K = $\begin{bmatrix} alpha_x & s & x_0 \\ 0 & alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$

Where $x_0 = \frac{w_{12}w_{13} - w_{11}\, w_{23}}{w_{11}w_{22} - w_{12}w_{12}}$

Lambda = $w_{33} - \frac{w_{13}w_{13} + x_0(w_{12}\, w_{13} - w_{11}w_{23})}{w_{11}}$

Alpha$_x$ = $\sqrt{lambda/w_{11}}$

Alpha$_y$ = $\sqrt{lambda * w_{11}/(w_{11}w_{22} - w_{12} * w_{12})}$

S = $\frac{-w_{12}alpha_x^2 alpha_y}{lambda}$

Y$_0$ = $\frac{sx_0}{alpha_y} - \frac{w_{13}alpha_x^2}{lambda}$

Finding extrinsic parameters:

Epsilon = $1/||K^{-1}h_1||$

r_1 = epsilon*$K^{-1}h_1$, first rotation vector

r_2 = epsilon*$K^{-1}h_2$, second rotation vector

t = epsilon*$K^{-1}h_3$, translation vector

$r_3 = r_1 \times r_2$, third rotation vector

R = [r_1 r_2 r_3]. P = [R|t] is the camera projection matrix.

Refining the parameters:

To find a more accurate camera calibration matrix, the Levenberg Marquardt algorithm is used, by exploiting the fact that a small Euclidean distance between the actual image corner and the image corner found using the projection matrix is desired.

Formally, the function to minimize is $d_{geom}^2 = \sum_i \sum_j ||\, x_{ij} - K[R_i|t_i]x_{Mij}||$, where x_{ij} is the j^th corner of image I, and x_{Mij} is the corresponding corner in the calibration pattern.

This can also be expressed as

$d_{geom}^2 = \sum_i \sum_j ||\, x_{ij} - f(K, r_i, t_i, x_{Mij})||$

where r_i is the Rodrigues representation of the rotation matrix R_i of image i.

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

Results

Dataset 1

K =

```
 [[716.61411032  -4.1341376  332.20459845]
 [  0.         717.1089712  242.02798697]
 [  0.           0.           1.        ]]
```

For image 1:

```
r1 = [ 0.98083761 -0.1652457   0.1032058 ]
r2 = [ 0.28119432  0.74235906 -0.55250727]
r3 = [0.01468369 0.5709408  0.77459984]
t = [-7.96227457  0.46668865 21.86777919]
```

For image 2:

```
r1 = [ 0.9972895  -0.03109675  0.06668313]
r2 = [-0.07499797  0.90431679  0.56938607]
r3 = [-0.07800873 -0.57284385  0.89953345]
t = [-6.56772576  0.63799186 19.84652736]
```

For image 3:

```
r1 = [0.9721737  0.18457743 0.14425485]
r2 = [-0.25399232  0.94084618  0.45558099]
r3 = [-0.05163166 -0.47954348  0.96154717]
t = [-6.07657703 -0.37821762 19.93294045]
```

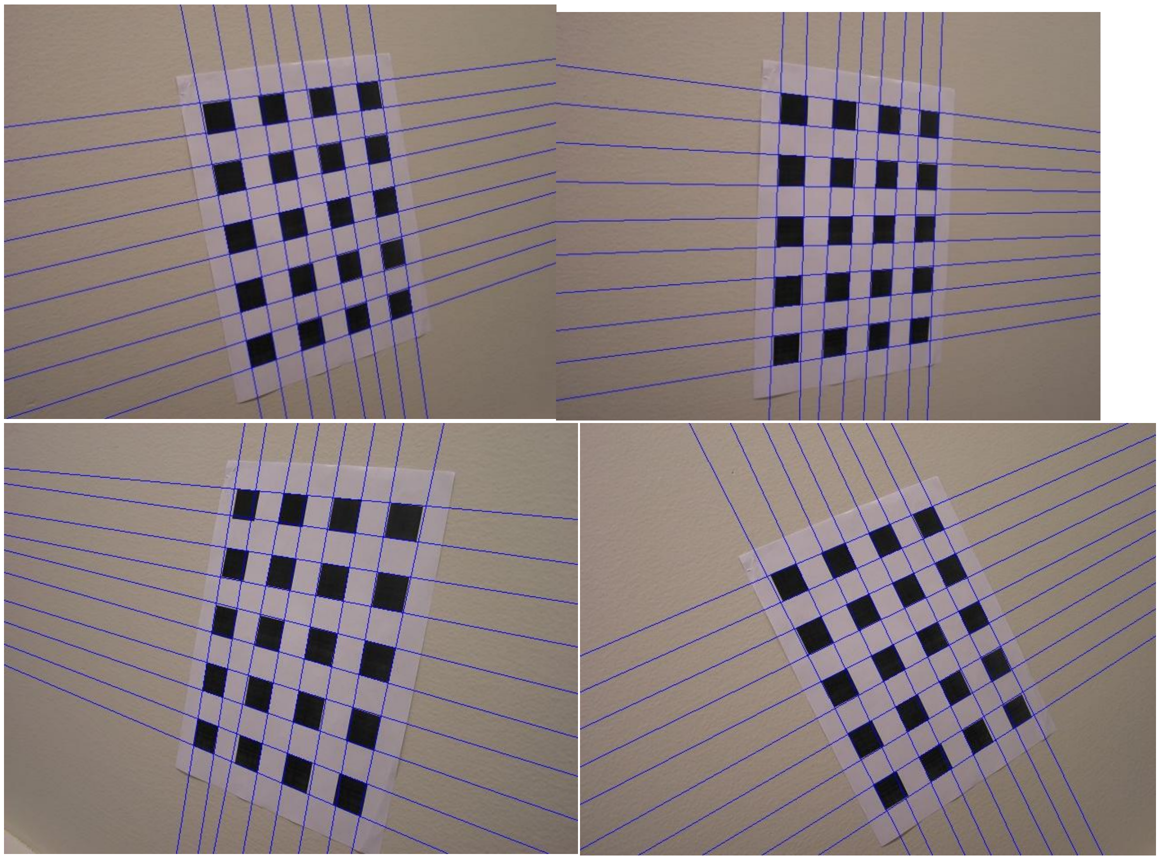
For image 4:

```
r1 = [0.89788243 0.4382681  0.04157184]
r2 = [-0.47349433  0.88345045  0.42845829]
r3 = [ 0.15105294 -0.4043892   1.0007521 ]
t = [-4.97334544 -0.98183795 21.73508992]
```

ECE 661 HW08

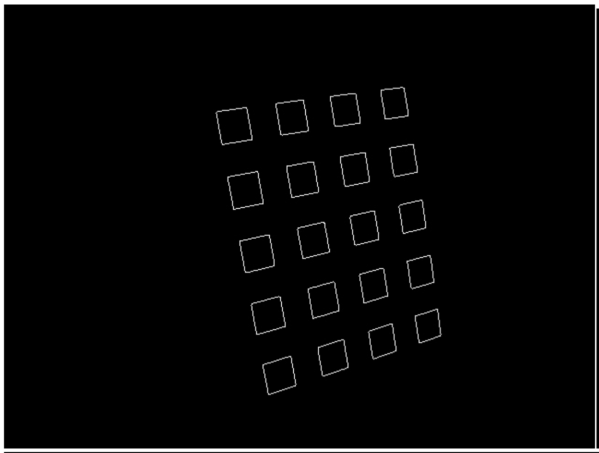
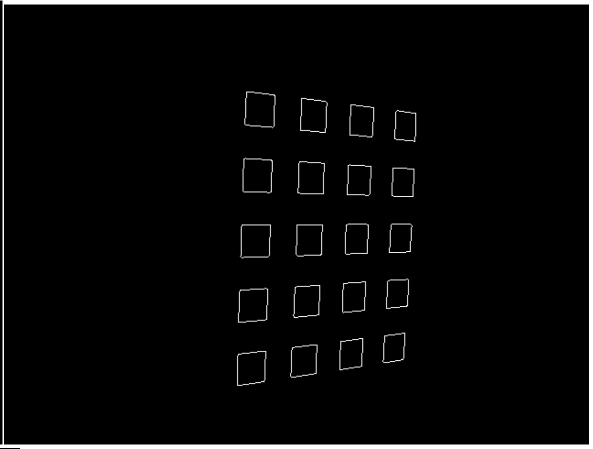
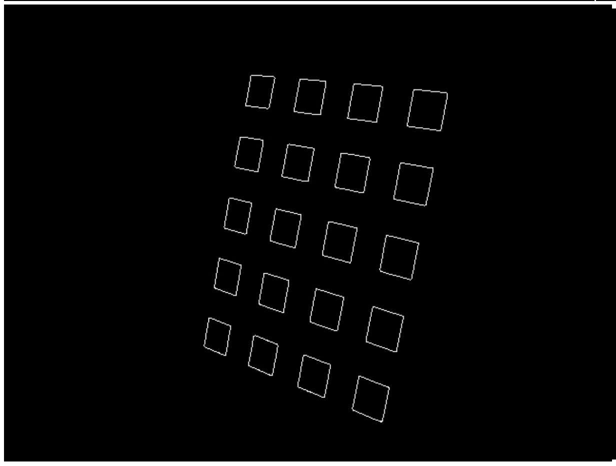
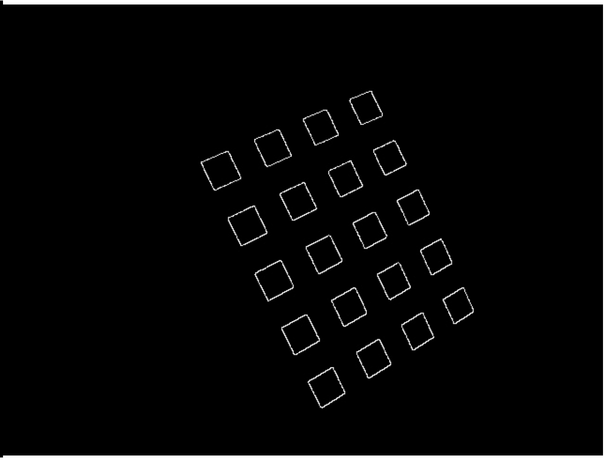
Rehana Mahfuz (rmahfuz@purdue.edu)

Hough lines in 4 images:



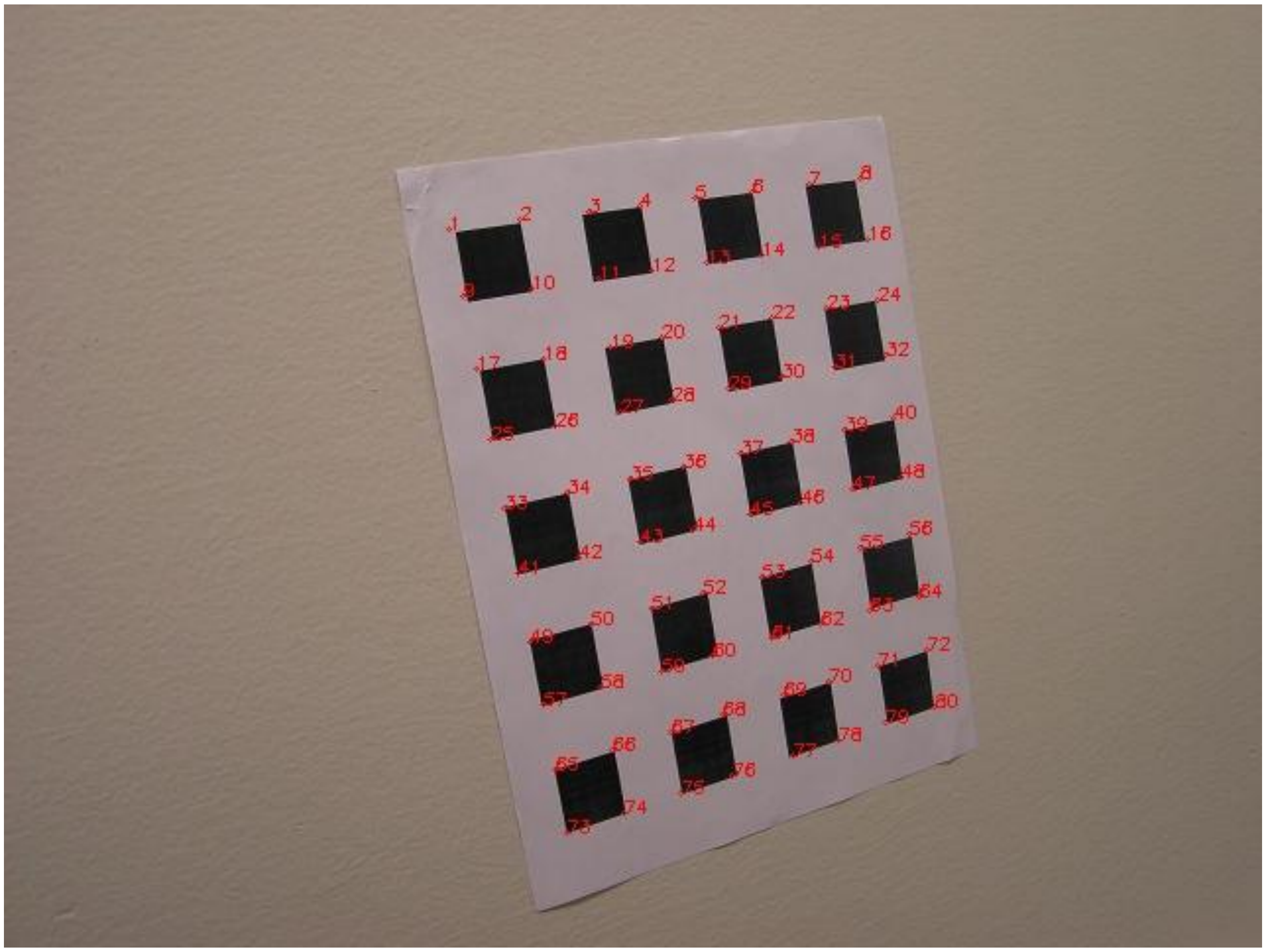
ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

Edges in 4 images:

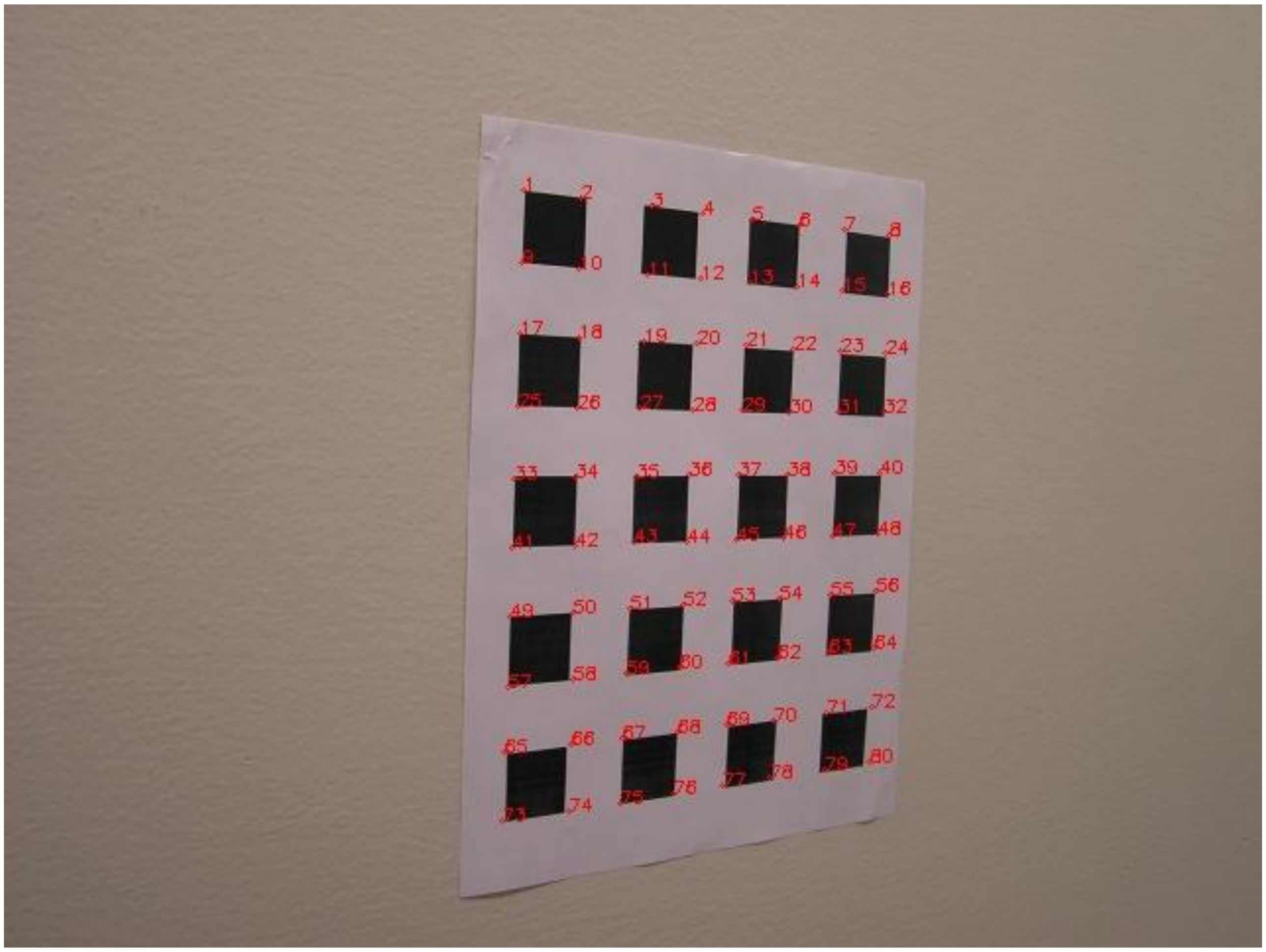





Corners in 4 images

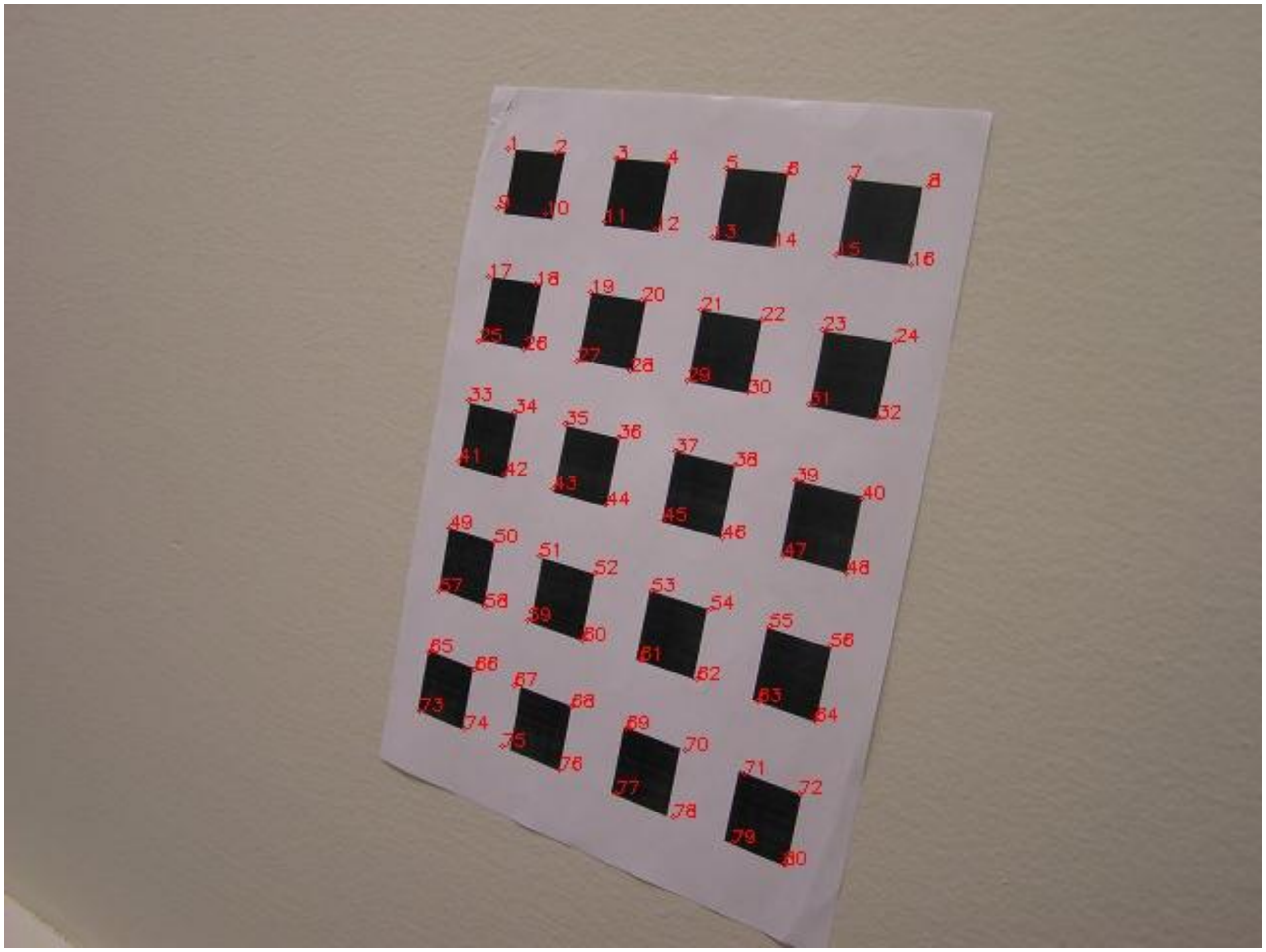
ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



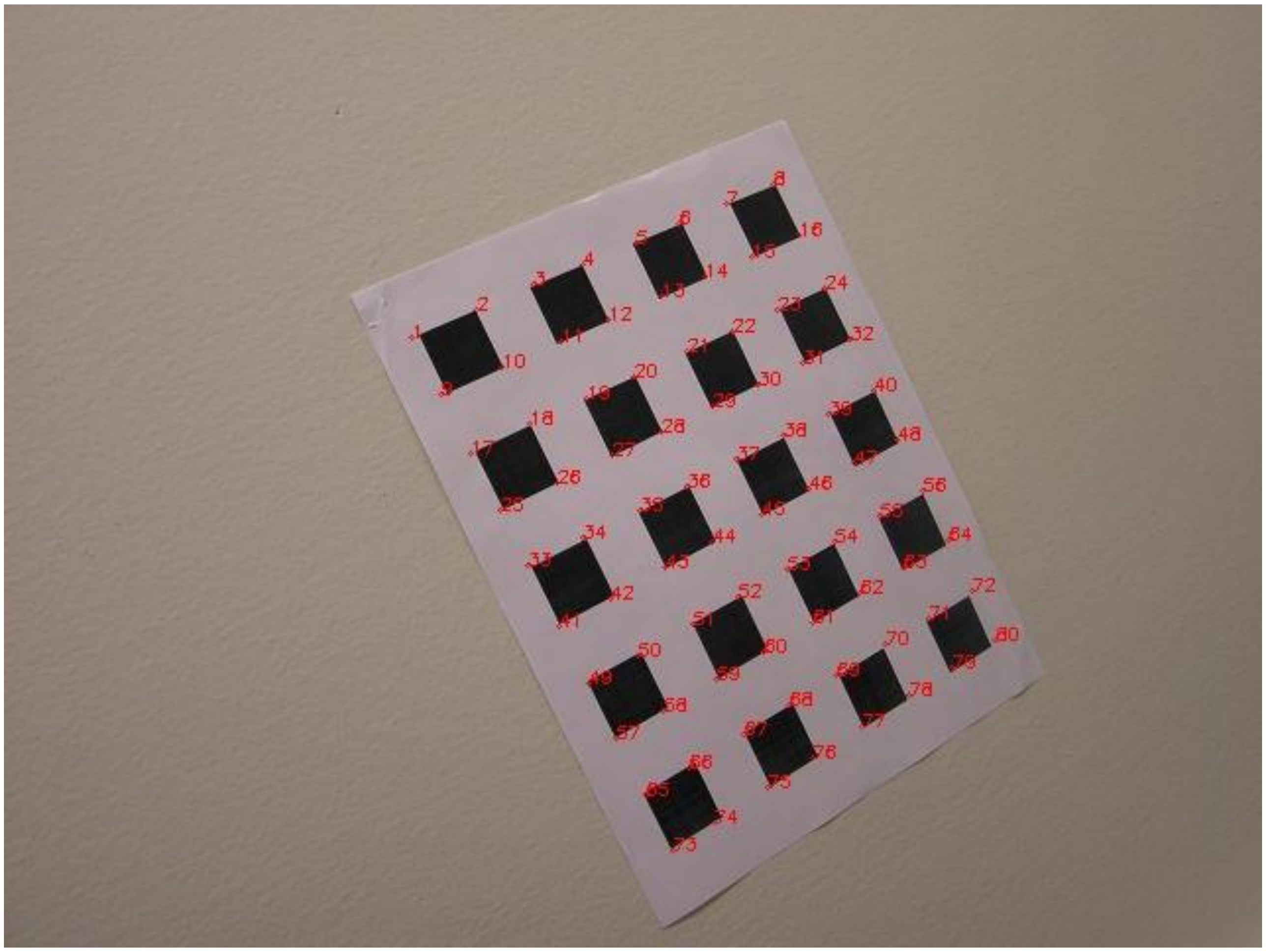
ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

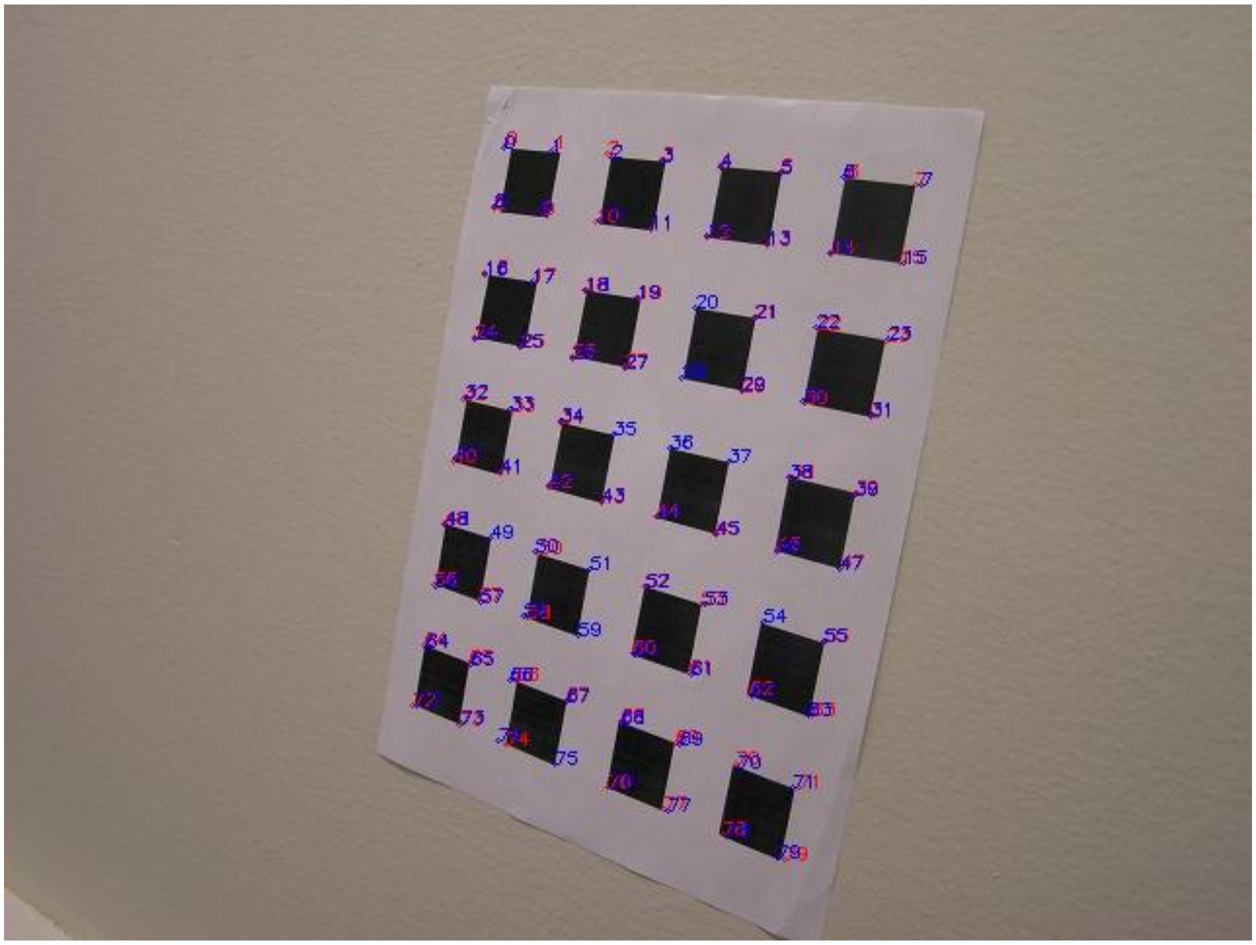


Reprojected corners for Images 1, 2, 3 and 4:

Blue points and text show the actual corners. Red points and text show the reprojected corners. Sometimes only one color may be visible because of overlap.

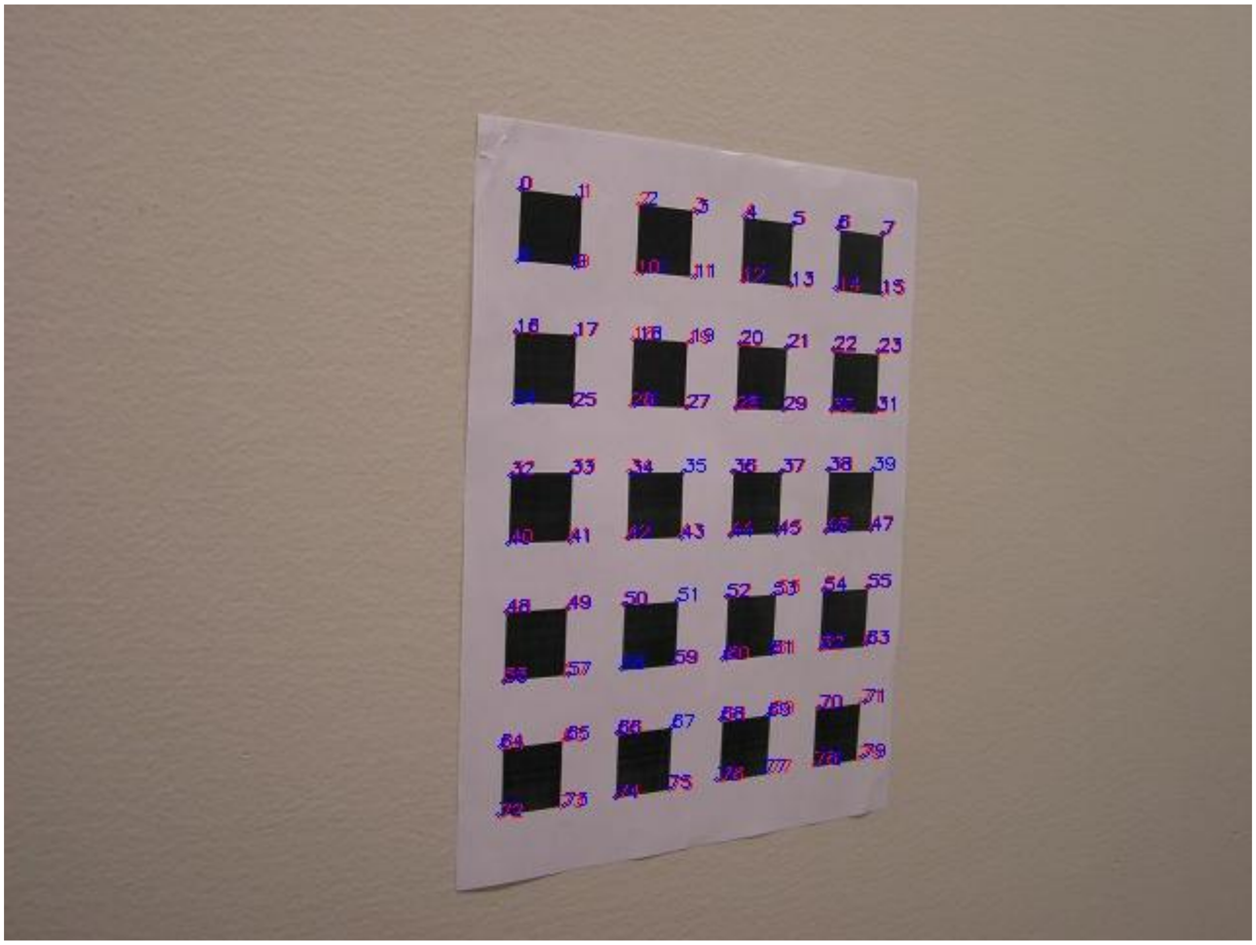
ECE 661 HW08

Rehana Mahfuz (rmahfuz@purdue.edu)

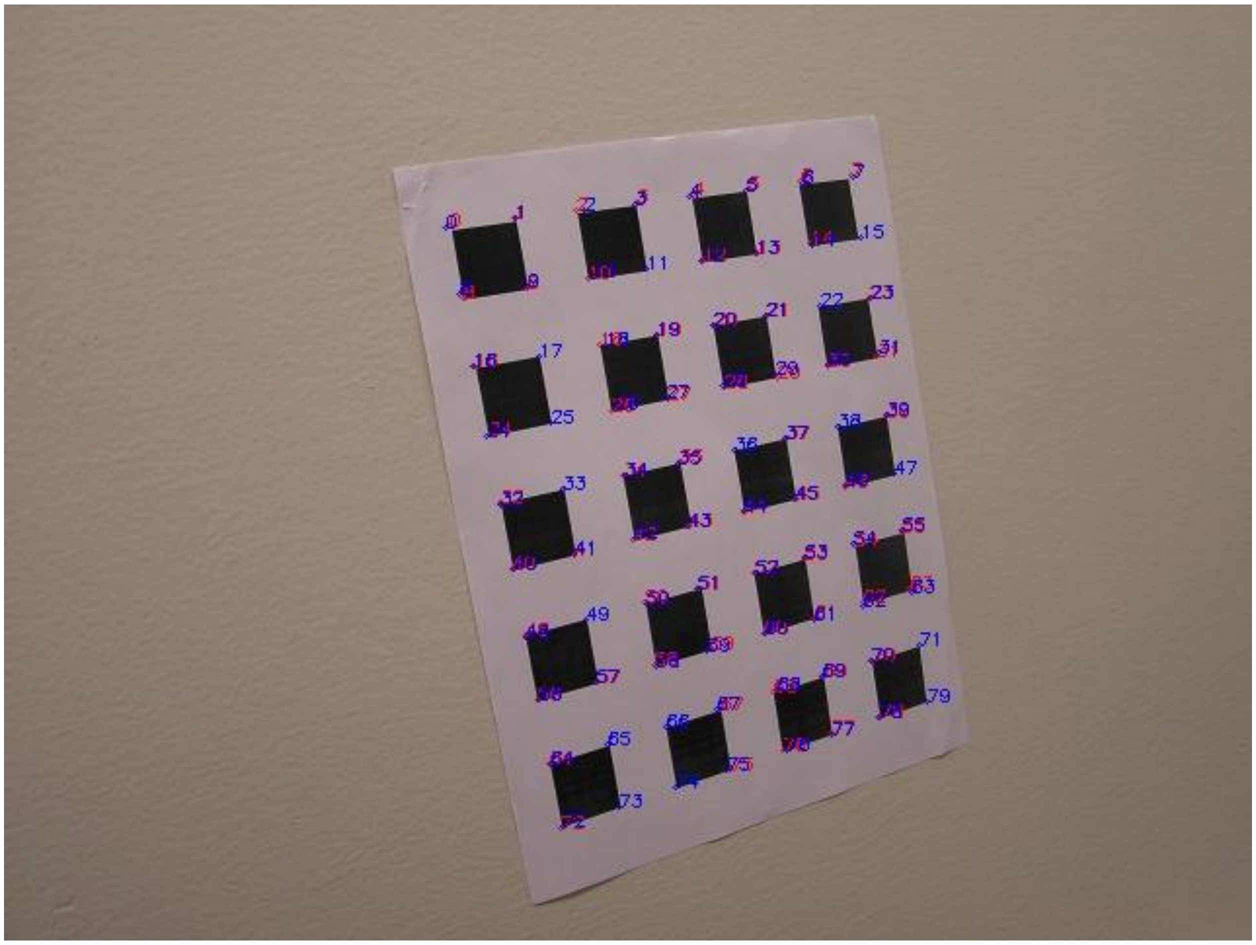


ECE 661 HW08

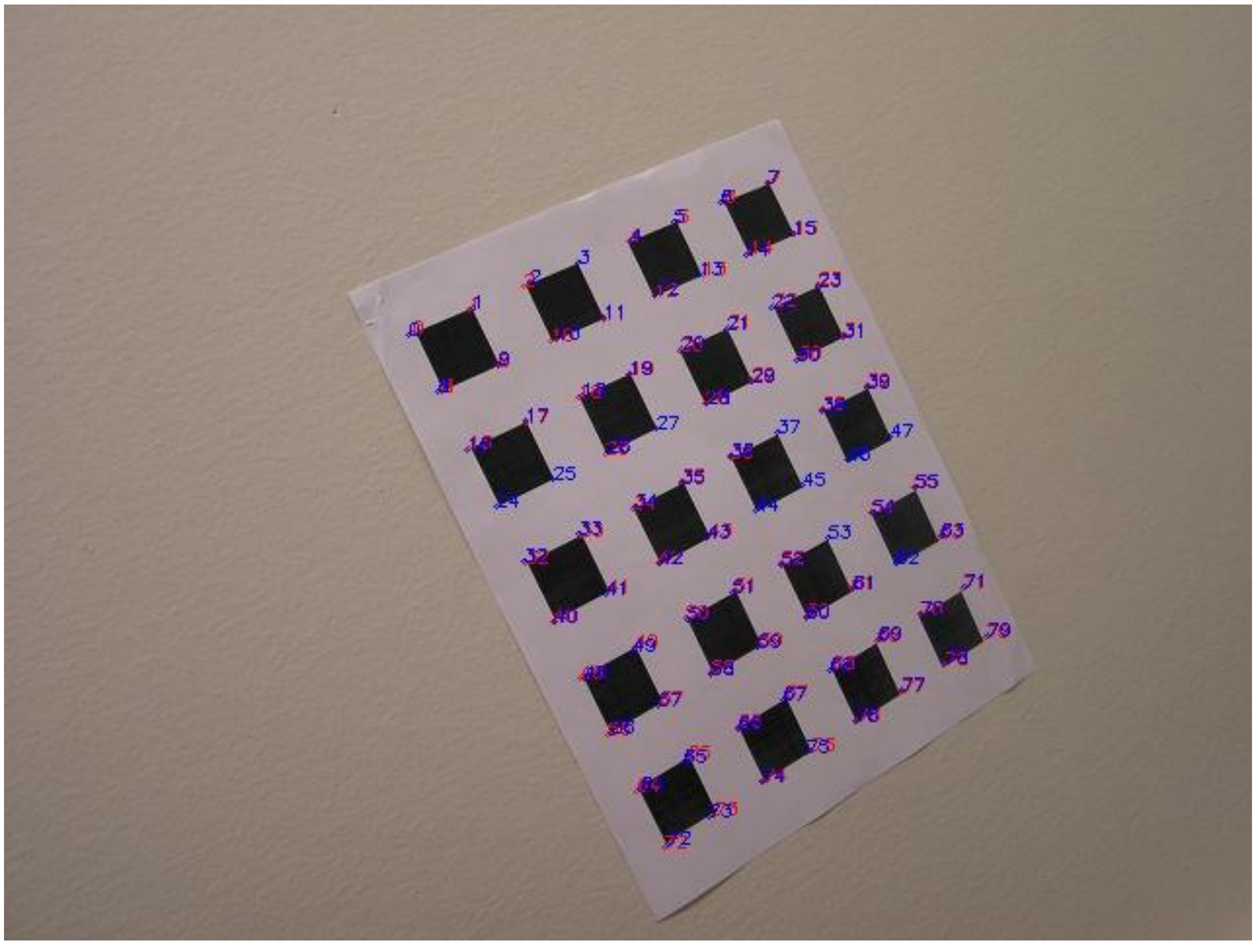
Rehana Mahfuz (rmahfuz@purdue.edu)



ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



Gain using LM for images 1, 2, 3 and 4:

Green shows actual corners. Blue shows optimized corners. Red shows unoptimized corners. Sometimes a color may be eclipsed by other colors because of overlap.

For image 1

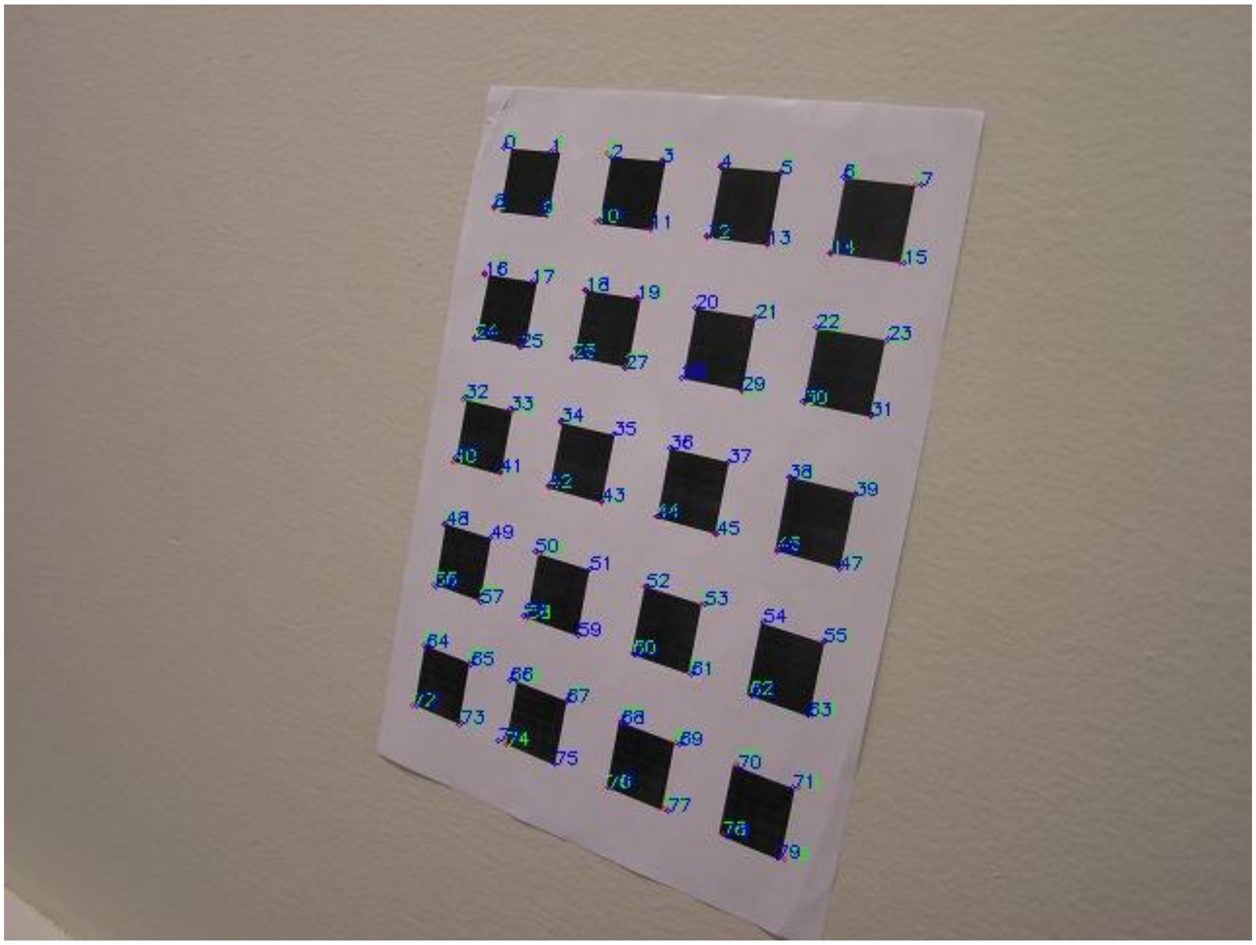
Mean of euclidean distance between unoptimized corners and actual corners = 1.3565302934373897

Variance of euclidean distance between unoptimized corners and actual corners = 0.6976034932102673

Mean of euclidean distance between optimized corners and actual corners = 1.0792225348358464

Variance of euclidean distance between optimized corners and actual corners = 0.6010614726704785

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



For image 2

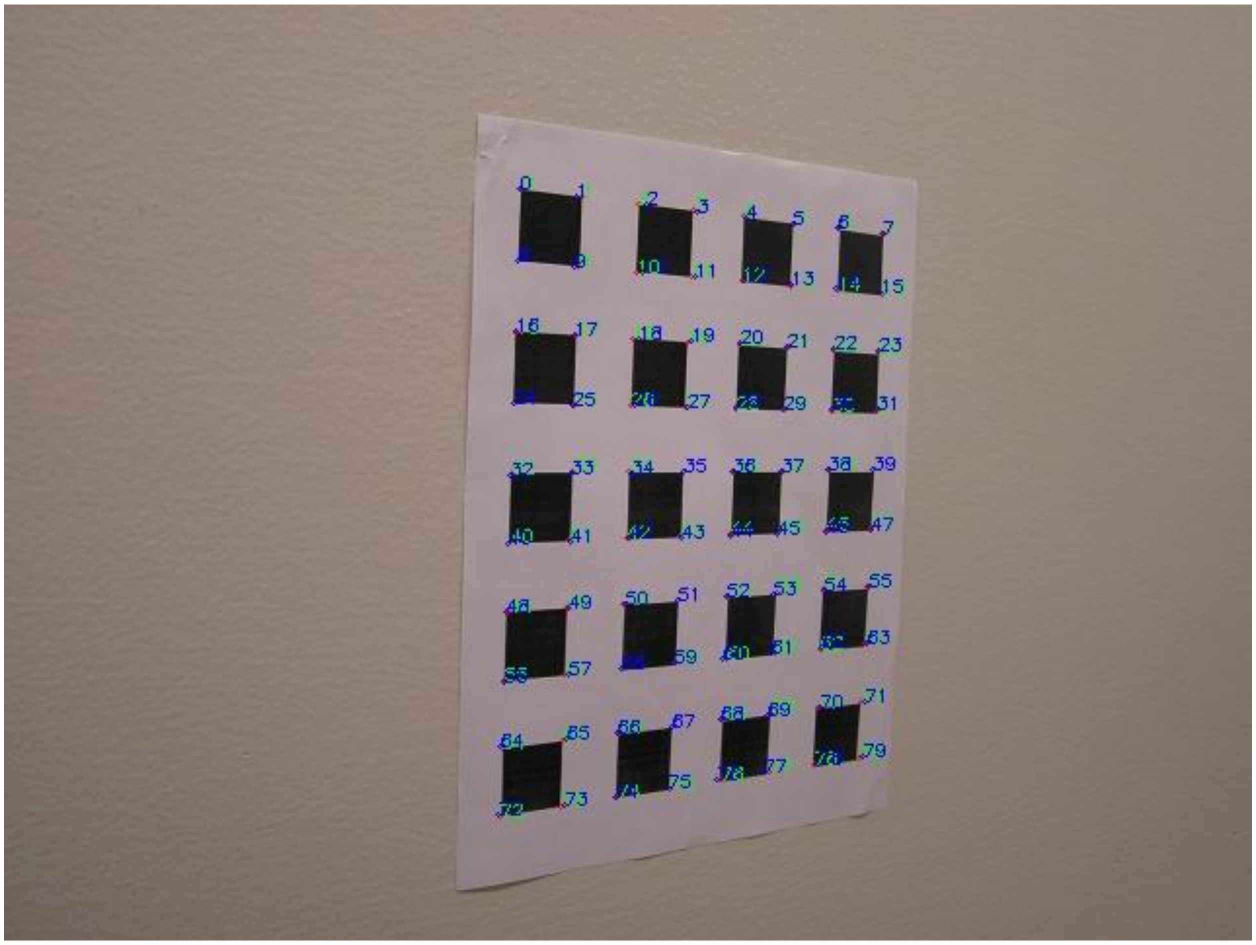
Mean of euclidean distance between unoptimized corners and actual corners = 1.2941647807487013

Variance of euclidean distance between unoptimized corners and actual corners = 0.529217975109441

Mean of euclidean distance between optimized corners and actual corners = 1.09505663464601

Variance of euclidean distance between optimized corners and actual corners = 0.463208373143824

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



For image 3

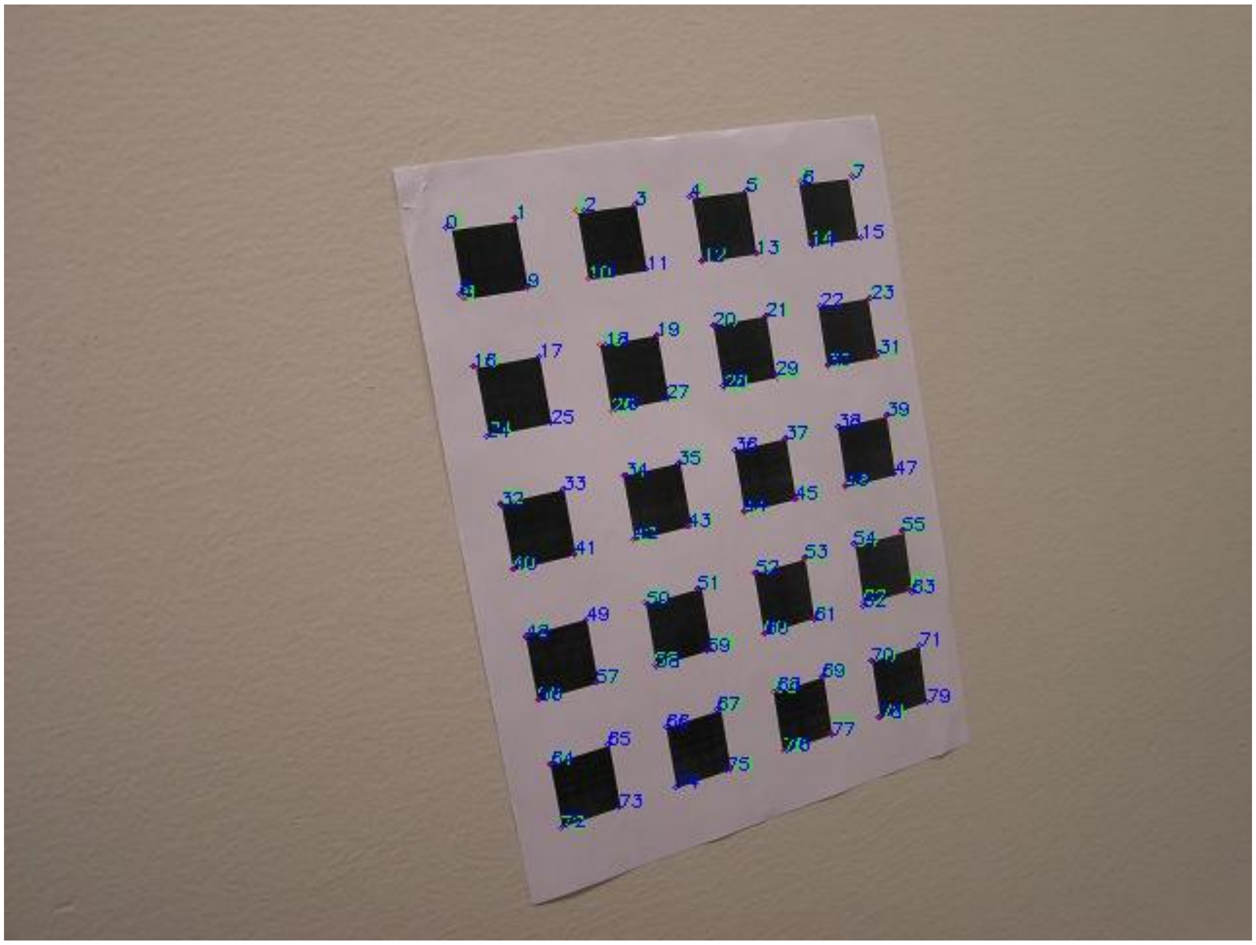
Mean of euclidean distance between unoptimized corners and actual corners = 1.1140433299120829

Variance of euclidean distance between unoptimized corners and actual corners = 0.5157228671625466

Mean of euclidean distance between optimized corners and actual corners = 0.8674226319396673

Variance of euclidean distance between optimized corners and actual corners = 0.4320076333319777

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



For image 4

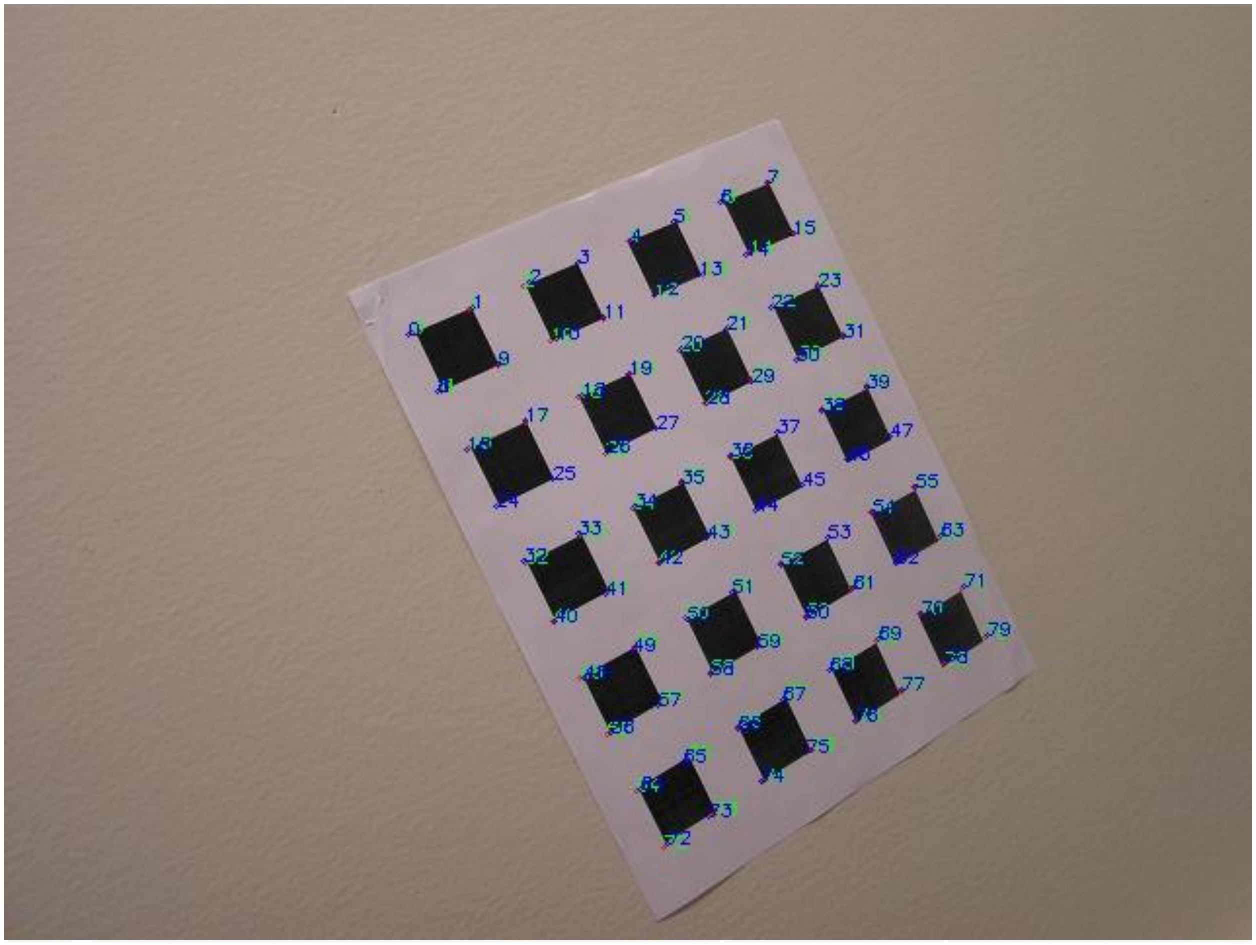
Mean of euclidean distance between unoptimized corners and actual corners = 1.1616921475029525

Variance of euclidean distance between unoptimized corners and actual corners = 0.5244663912064448

Mean of euclidean distance between optimized corners and actual corners = 0.8780591518121903

Variance of euclidean distance between optimized corners and actual corners = 0.44719589145198

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



Dataset 2

K =

```
 [[454.90868982   2.29894721 233.79498543]
 [  0.         442.00958493 266.13285637]
 [  0.           0.           1.        ]]
```

For image 2:

```
r1 = [ 0.97200347 -0.12798057  0.19705386]

r2 = [0.07124315 0.91342239 0.30235505]

r3 = [-0.21868898 -0.27985143  0.89696748]

t = [-2.67061804 -3.58763782  8.75902116]
```

For image 8:

```
r1 = [0.95973932 0.00690542 0.28080731]

r2 = [ 0.0102364   0.97873155 -0.16496577]
```

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

r3 = [-0.27597413 0.16119859 0.93925647]

t = [-0.72731578 -4.14203143 9.44551375]

For image 11:

r1 = [0.96343982 -0.08777046 0.25314038]

r2 = [-5.46126679e-04 8.94822877e-01 3.40967879e-01]

r3 = [-0.25644271 -0.32864028 0.86206006]

t = [-2.03289703 -4.1148753   9.02893818]

For image 12:

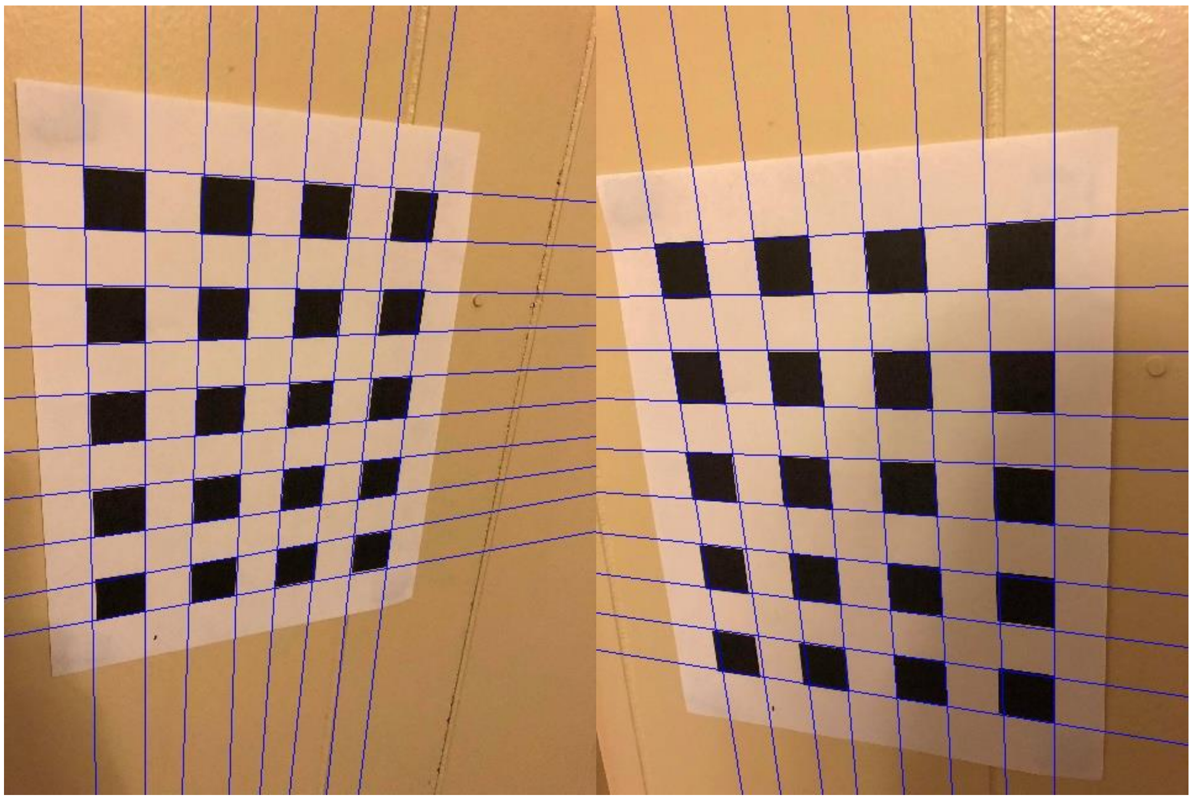
r1 = [0.97543719 0.05358166 0.21366162]

r2 = [-0.03241356 0.97278328 -0.24787261]

r3 = [-0.22112788 0.23485863 0.95062576]

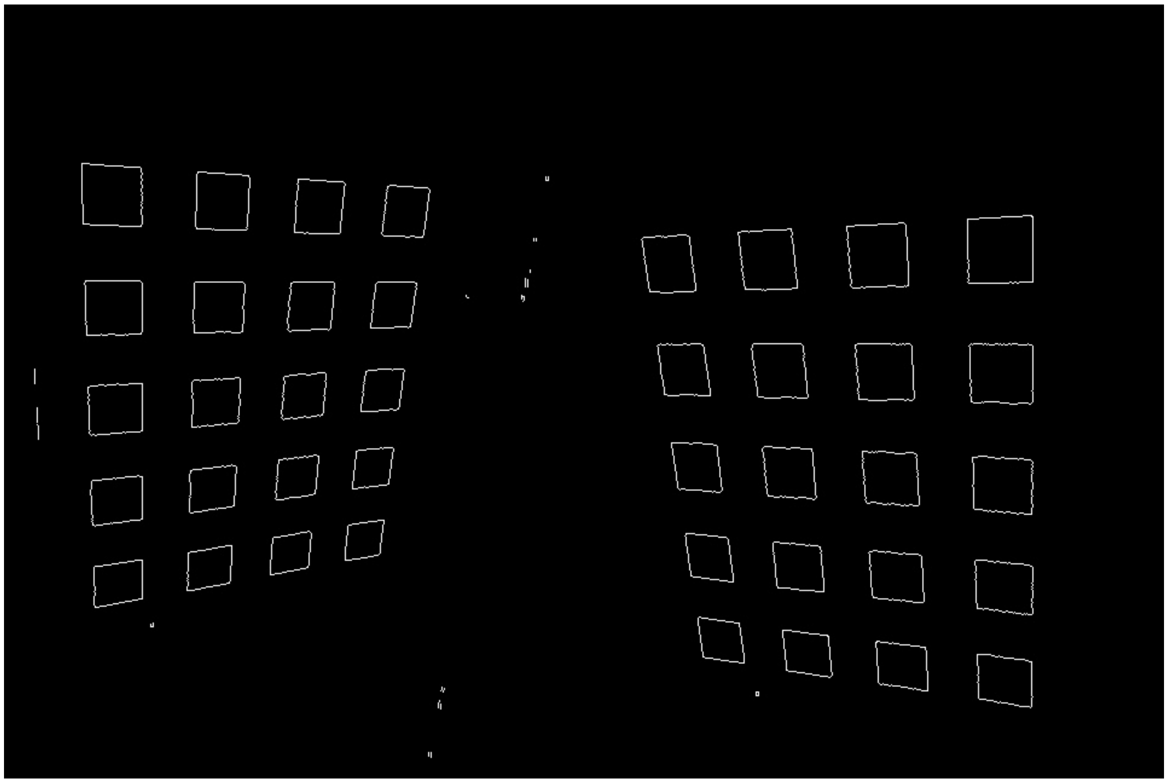
t = [-0.85691631 -4.88352805 9.86735105]

Hough lines in 2 images:



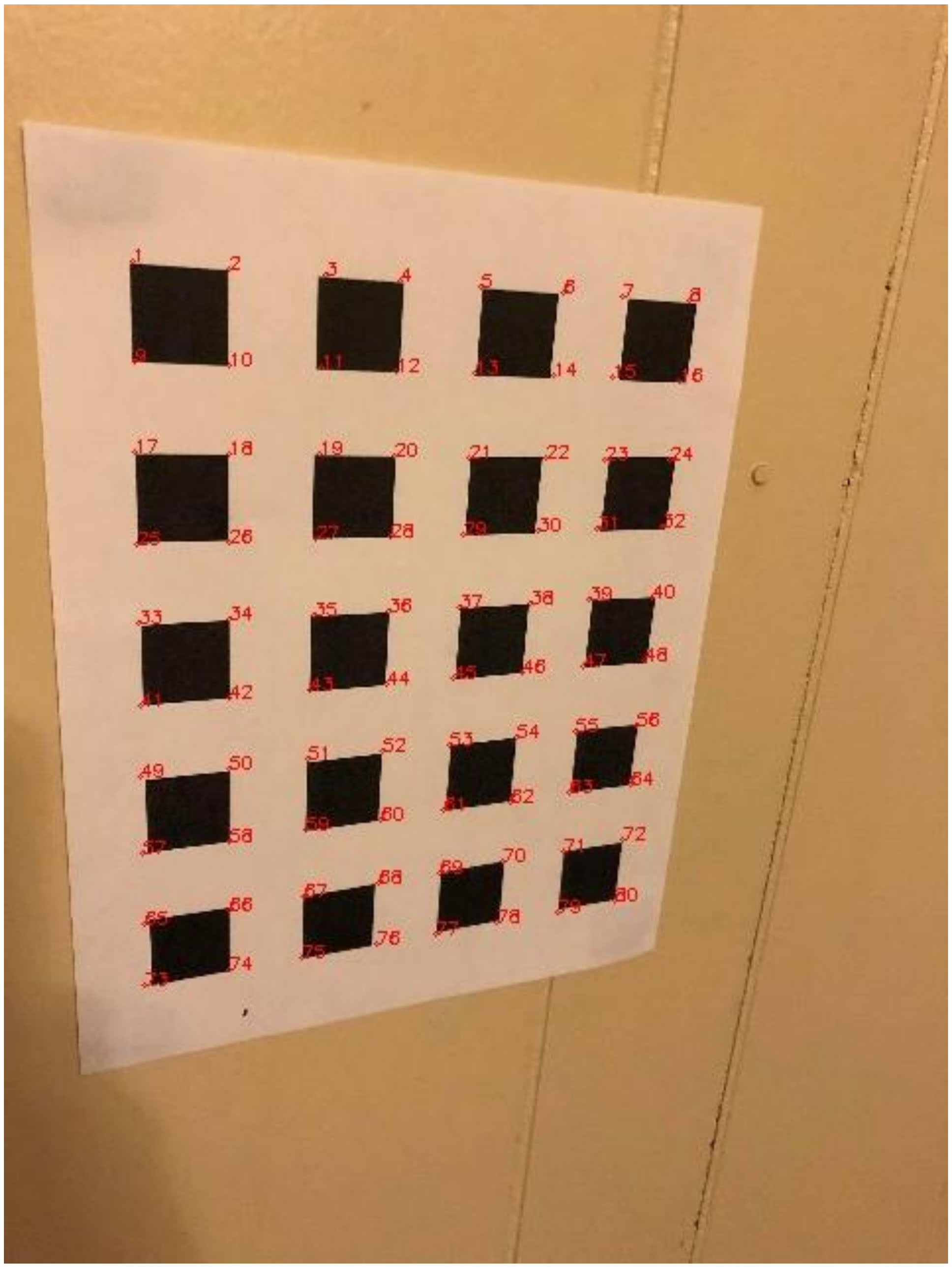
Edges in 2 images:

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

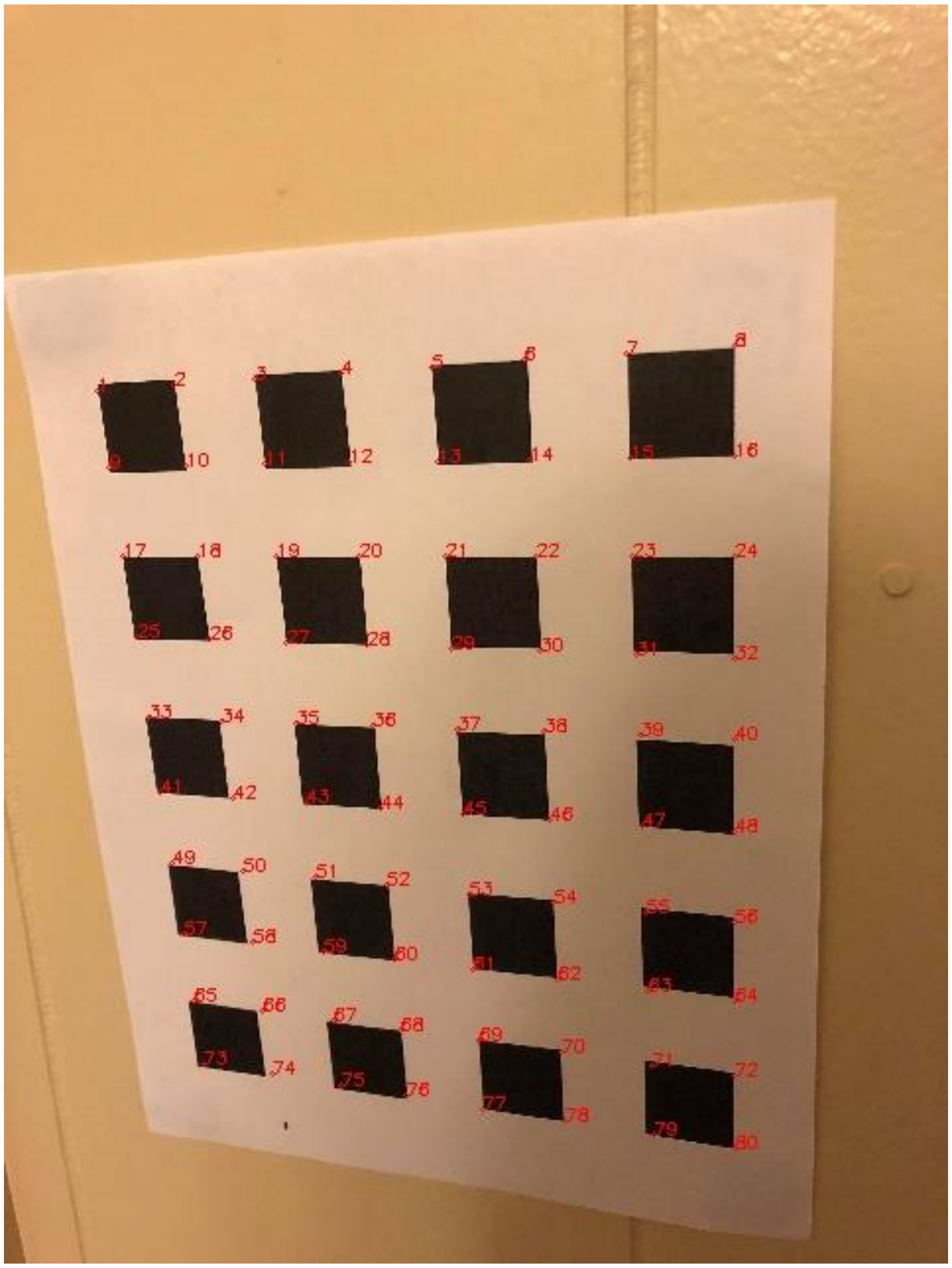


Corners in 2 images:

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



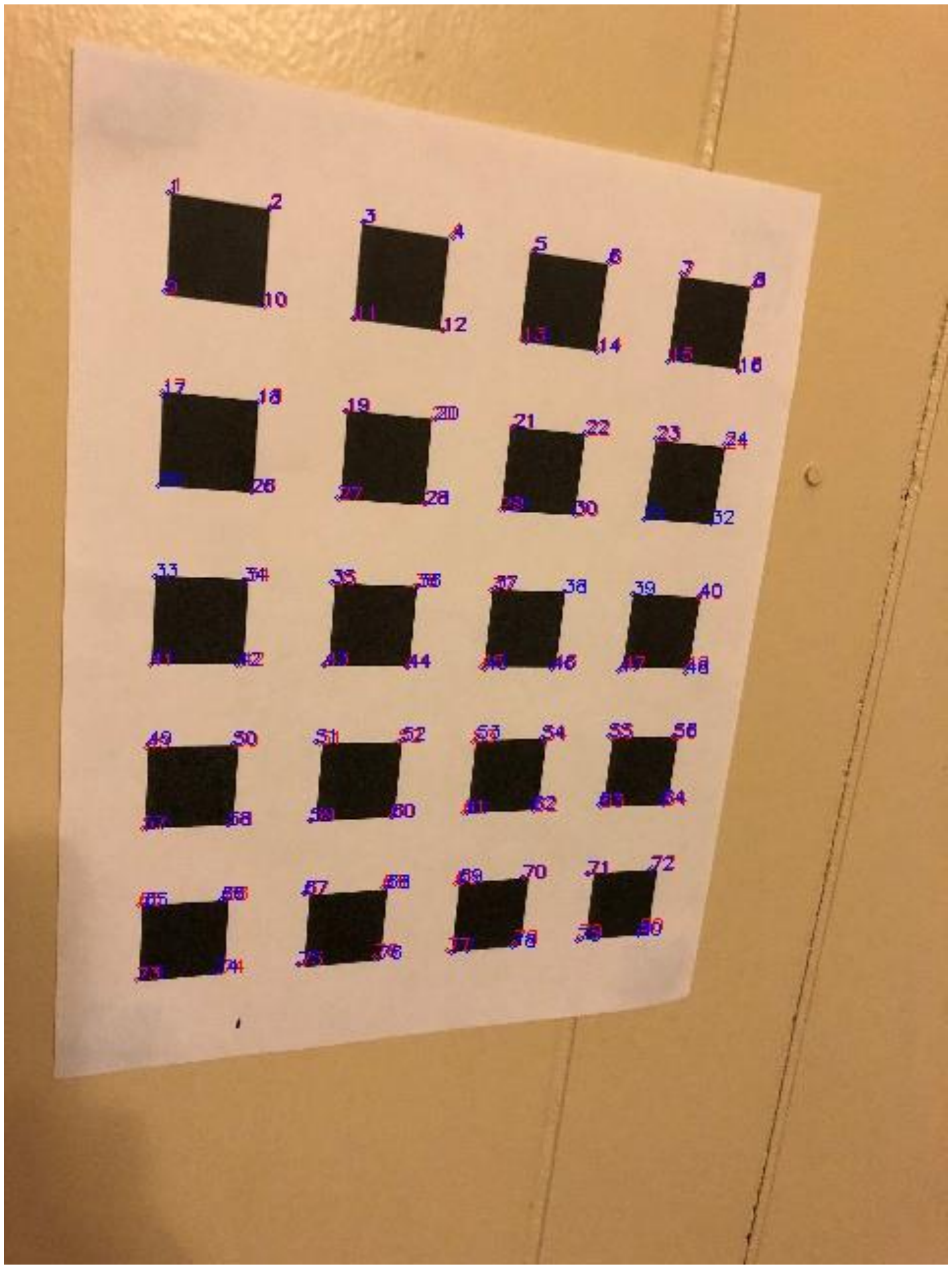
ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



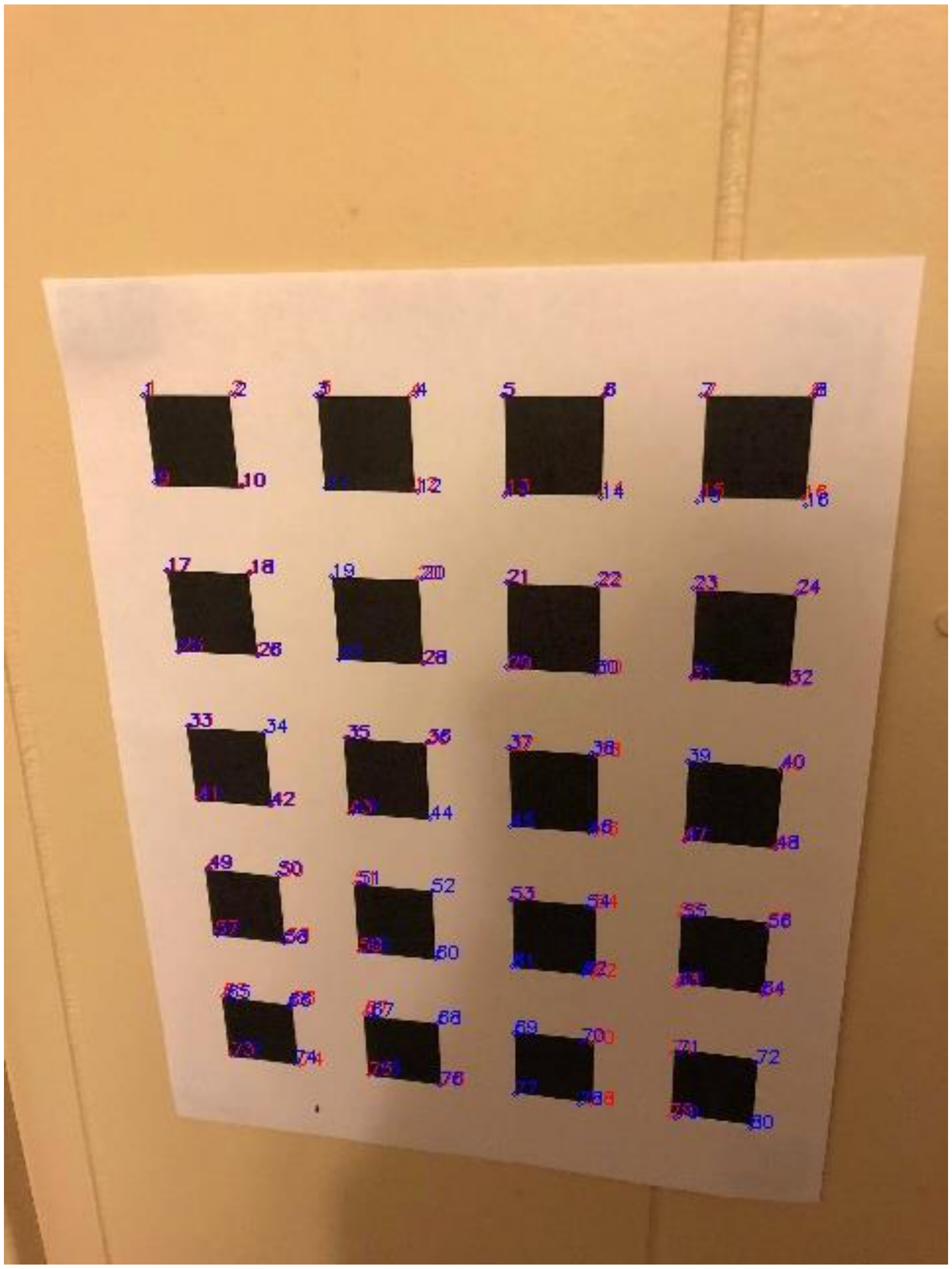
Reprojected corners for Images 2, 8, 11 and 12:

Blue points and text show the actual corners. Red points and text show the reprojected corners. Sometimes only one color may be visible because of overlap.

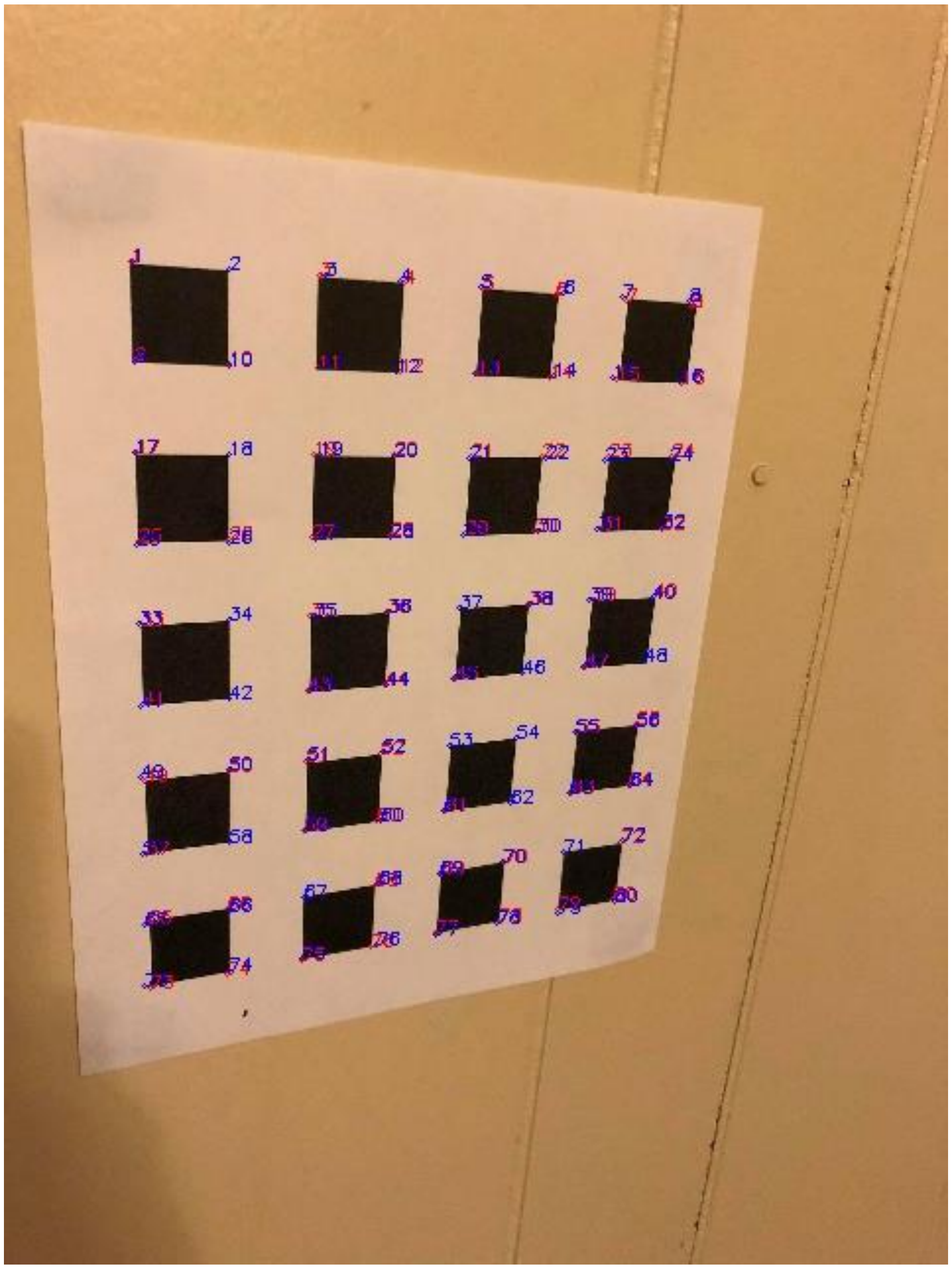
ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



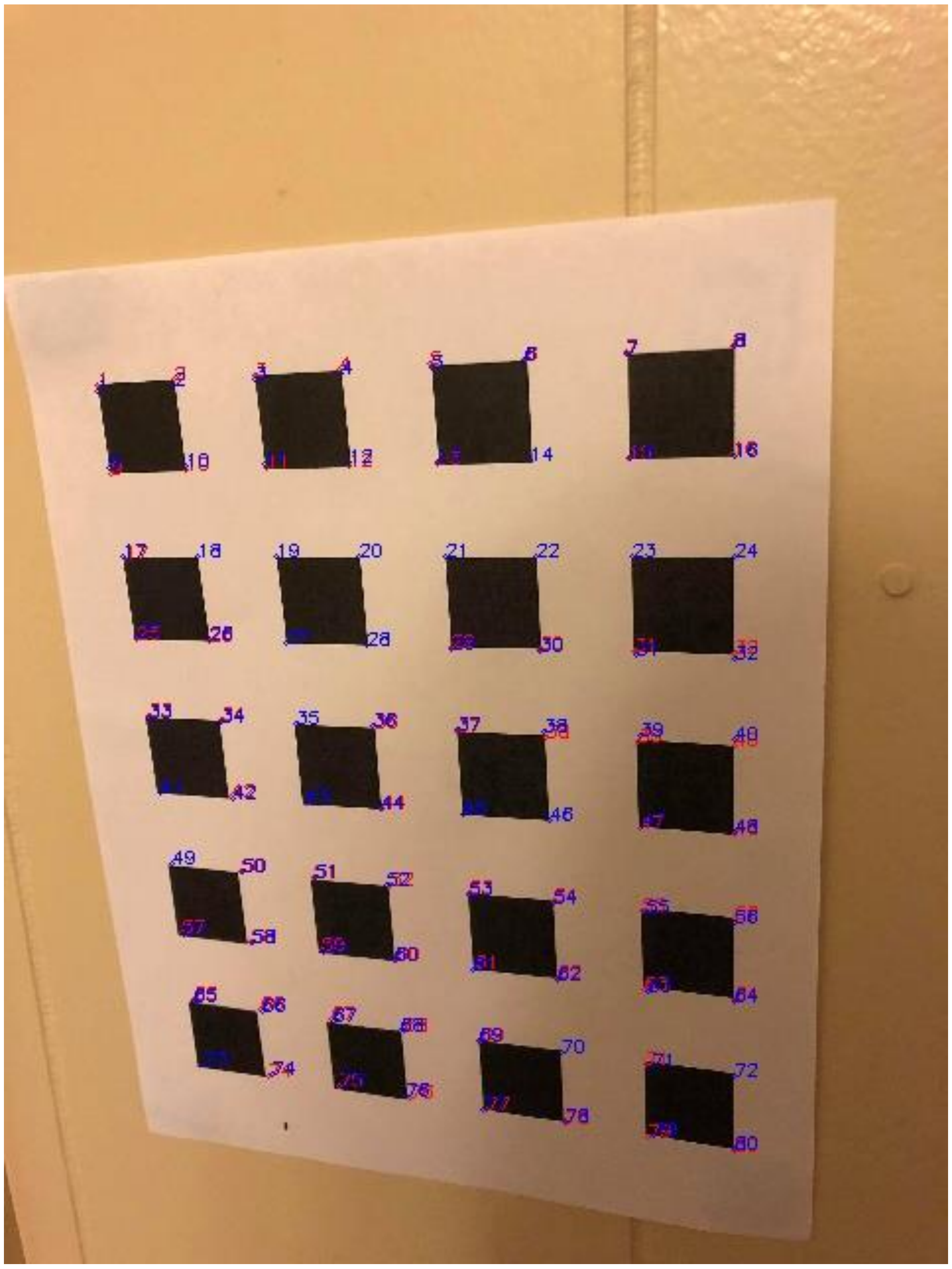
ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



Gain using LM for images 2, 8, 11 and 12:

Green shows actual corners. Blue shows optimized corners. Red shows unoptimized corners. Sometimes a color may be eclipsed by other colors because of overlap.

For image 2

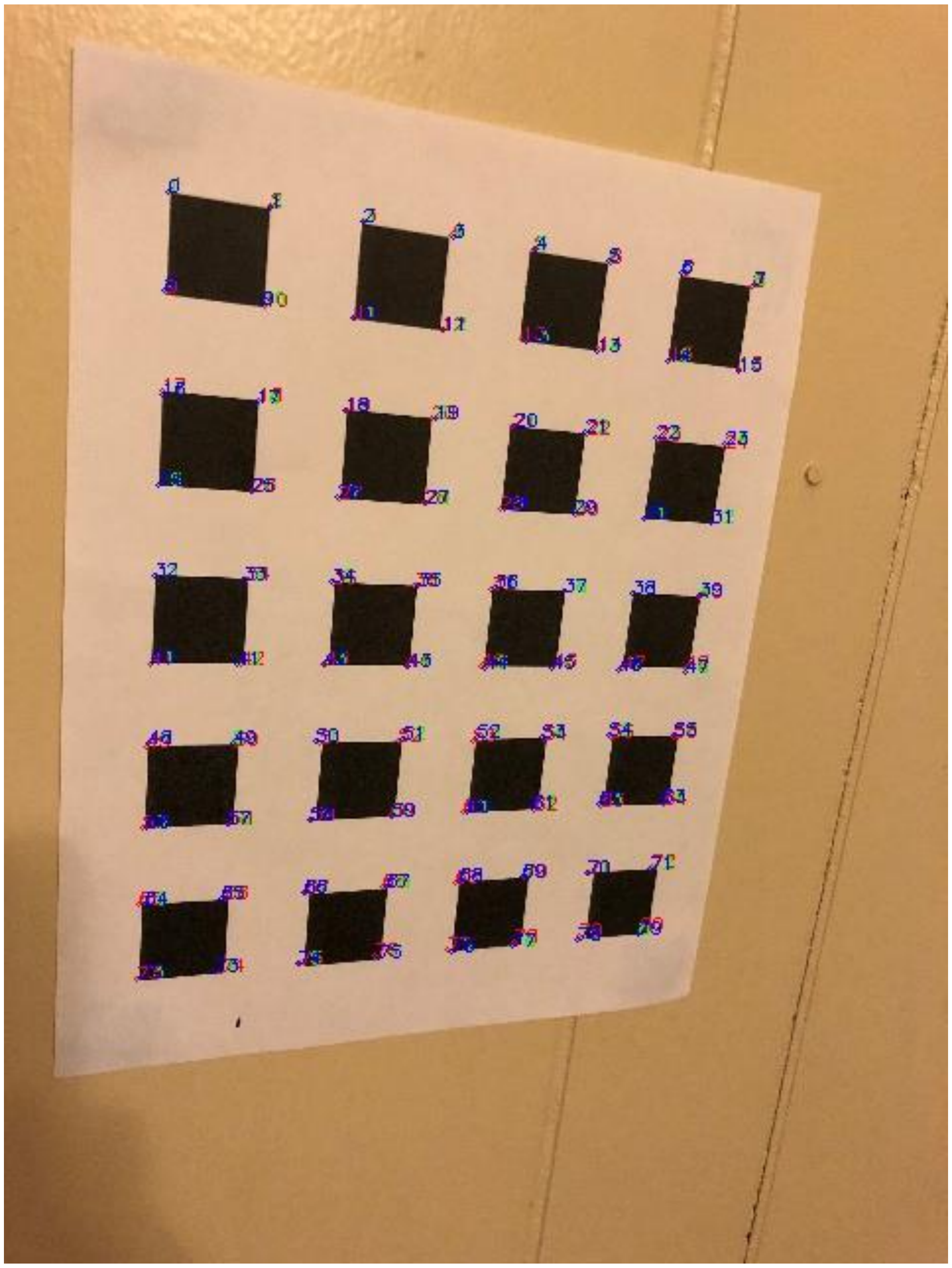
Mean of euclidean distance between unoptimized corners and actual corners = 1.4741333925800377

Variance of euclidean distance between unoptimized corners and actual corners = 0.2535122739413193

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

Mean of euclidean distance between optimized corners and actual corners = 1.153736394440396

Variance of euclidean distance between optimized corners and actual corners = 0.112629942974534



For image 8

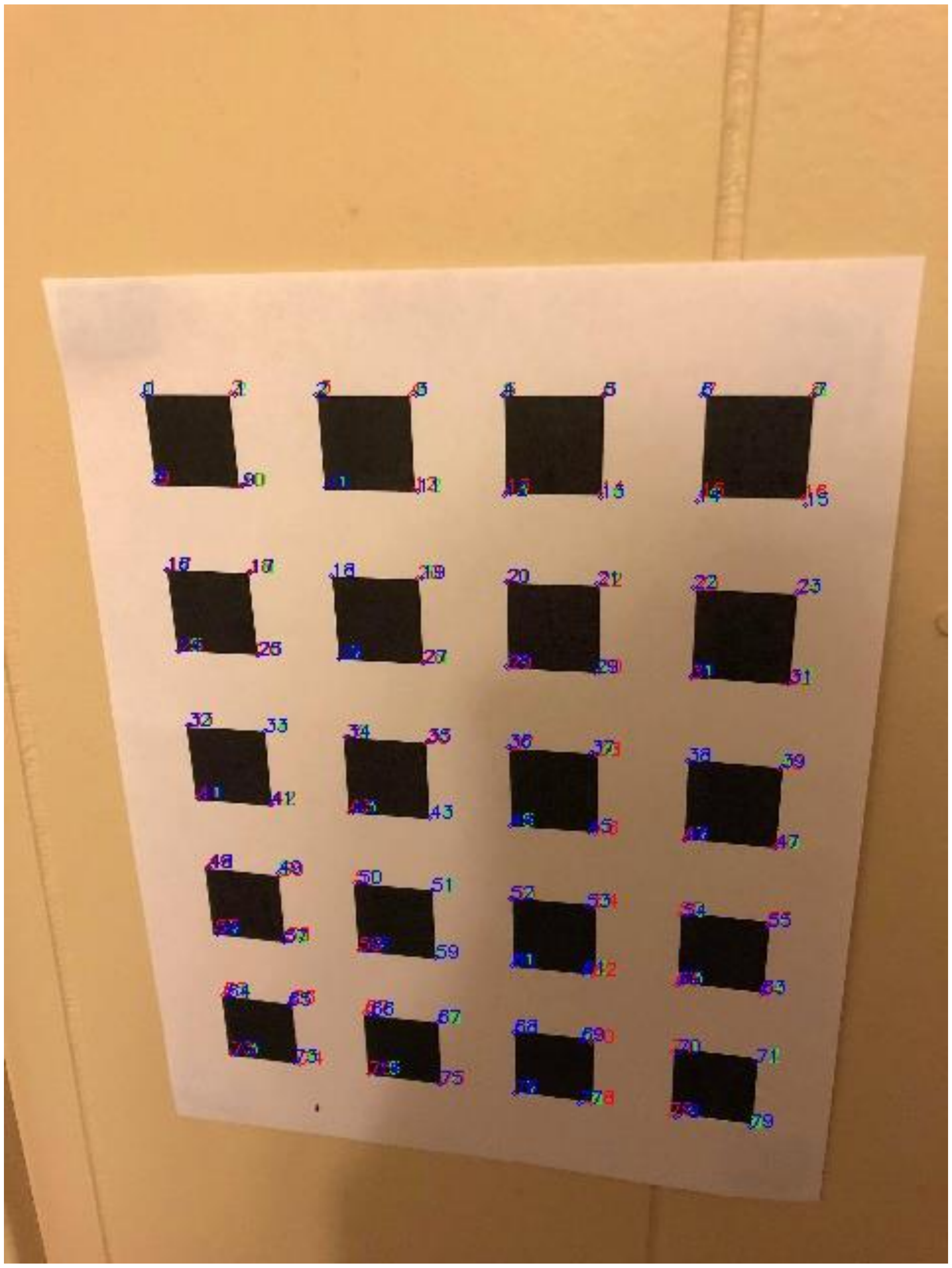
Mean of euclidean distance between unoptimized corners and actual corners = 1.63667537217414

Variance of euclidean distance between unoptimized corners and actual corners = 1.3698492269428972

Mean of euclidean distance between optimized corners and actual corners = 1.31640848785516

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

Variance of euclidean distance between optimized corners and actual corners = 1.1295586506841506



For image 11

Mean of euclidean distance between unoptimized corners and actual corners = 1.3140185360385044

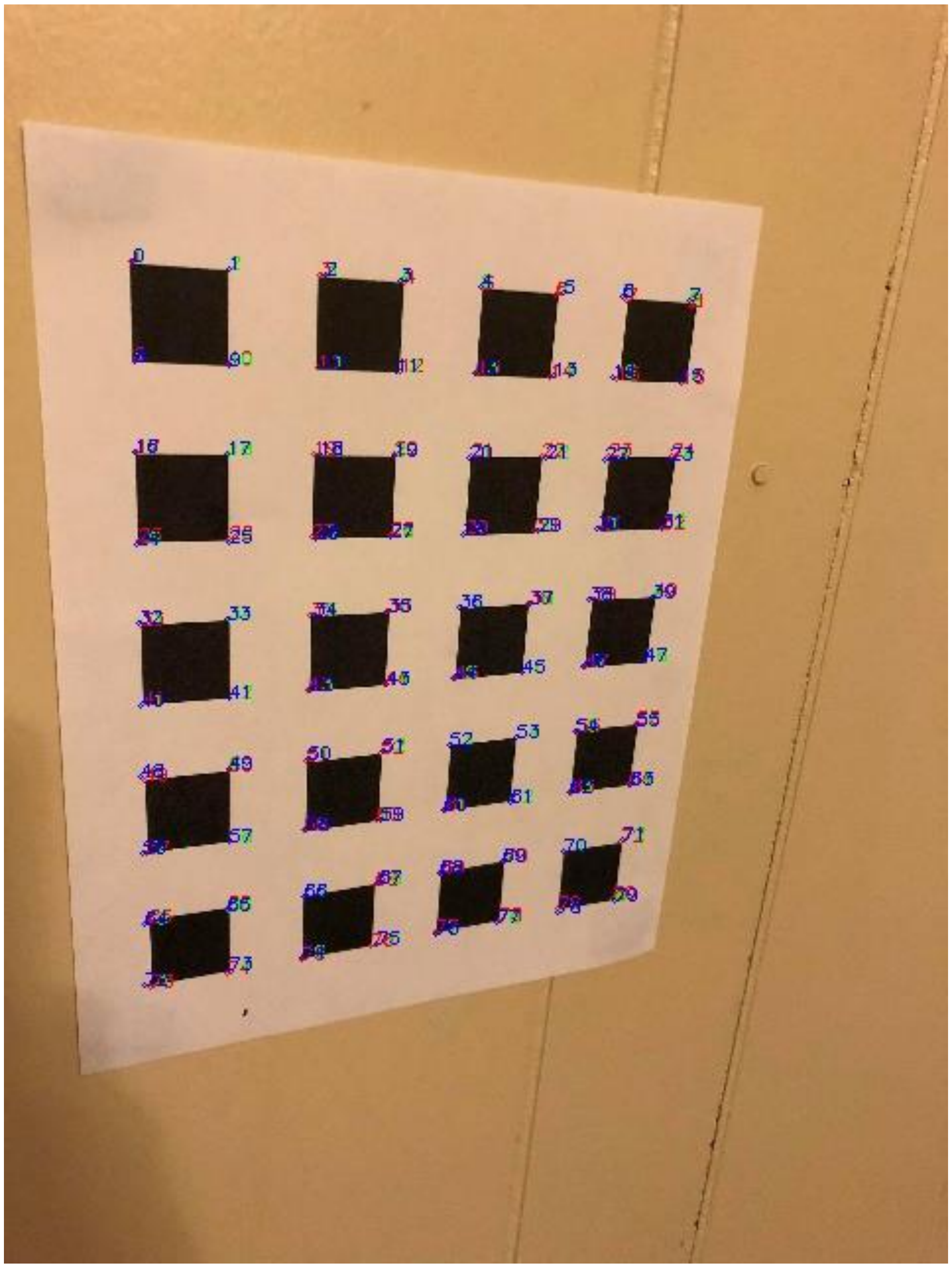
Variance of euclidean distance between unoptimized corners and actual corners = 0.7512395271874656

Mean of euclidean distance between optimized corners and actual corners = 0.983039569203729

Variance of euclidean distance between optimized corners and actual corners = 0.6673266565972996

ECE 661 HW08

Rehana Mahfuz (rmahfuz@purdue.edu)



For image 12

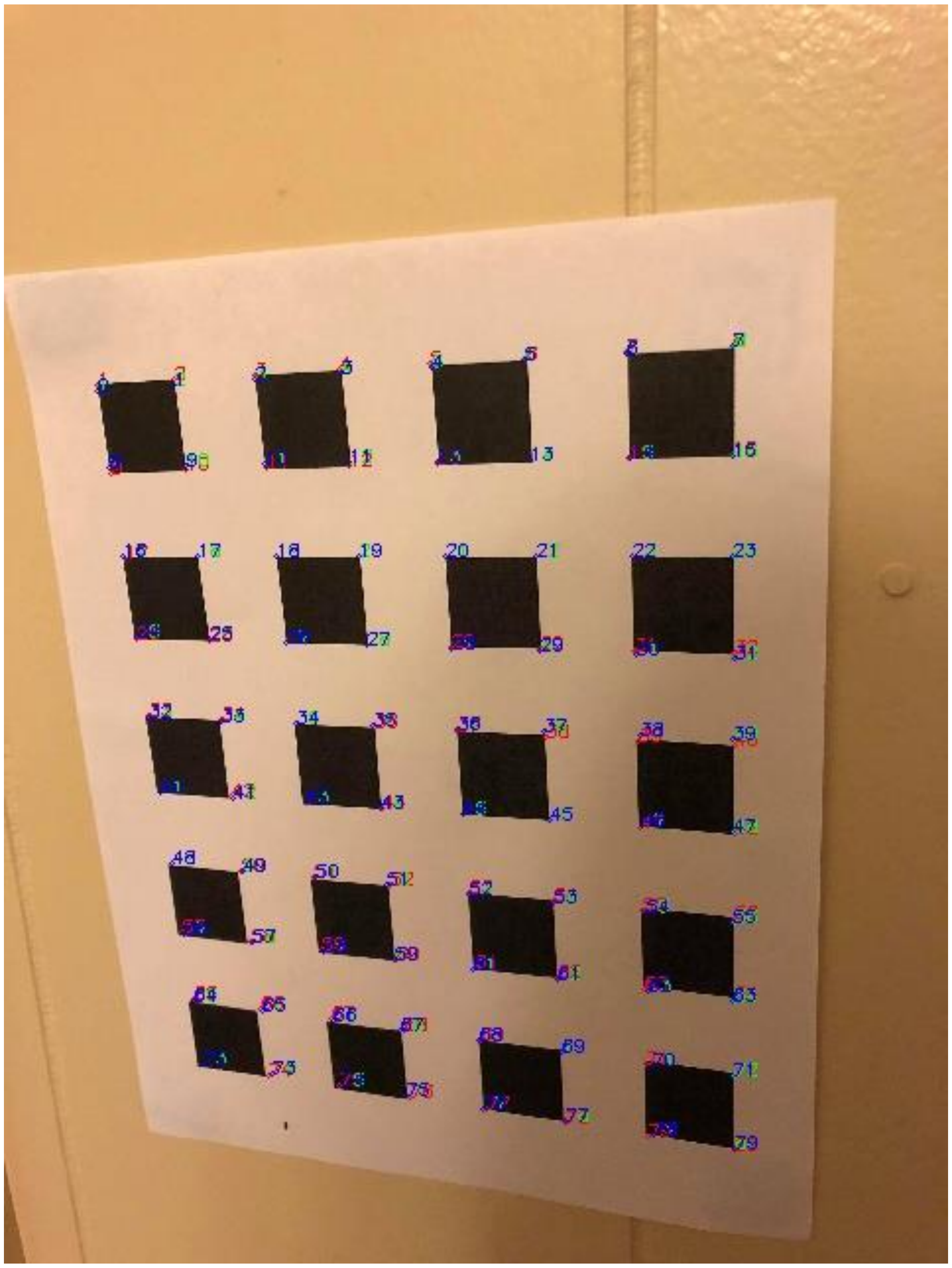
Mean of euclidean distance between unoptimized corners and actual corners = 1.328314675999981

Variance of euclidean distance between unoptimized corners and actual corners = 0.642709190268285

Mean of euclidean distance between optimized corners and actual corners = 1.02371947814936

Variance of euclidean distance between optimized corners and actual corners = 0.941586179039393

ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)



ECE 661 HW08
Rehana Mahfuz (rmahfuz@purdue.edu)

Source code:

```
import cv2, os

import numpy as np

from scipy import optimize

path = '../Users/rmahfuz/Desktop/661/HW08/'

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

def homogeneous_from_polar(x):

    pt1 = [x[0]*np.cos(x[1]), x[0]*np.sin(x[1]), 1.0]

    pt2 = [pt1[0]+100*np.sin(x[1]), pt1[1]-100*np.cos(x[1]), 1.0]

    return np.cross(pt1, pt2)

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

def gen_world_corners():

    world_cor = []

    x_li = [0, 1, 2, 3, 4, 5, 6, 7]

    y_li = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    for j in range(10):

        for i in range(8):

            world_cor.append((x_li[i], y_li[j]))

    #print('world_cor = ', world_cor)

    return world_cor

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

def find_corners(fileName):

    #print('fileName = ', fileName)

    color_img = cv2.imread(path + 'Dataset2/' + fileName)

    img = cv2.cvtColor(color_img, cv2.COLOR_BGR2GRAY)

    #Finding edges----------------------------

    edges = cv2.Canny(img, 255*1.5, 255)
    cv2.imwrite(path + 'edges/' + fileName, edges)

    #Finding lines----------------------------

    lines = cv2.HoughLines(edges, 1, np.pi/180, 50)

    lines = list(map(lambda x: x[0], lines))

    #Removing duplicate lines

    idx = 0

    to_del = []

    for i in lines:

        for j in lines[idx+1:]:

            #print(abs(i[0] - j[0]))

            if abs(i[0] - j[0]) < 15 and abs(i[1] - j[1]) < 1:

                #print('removed {}'.format(i))

                to_del.append(idx)

        idx += 1

    #print('to_del = ', to_del)

    all_idx = list(range(len(lines)))

    lines = np.array(lines)

    new_lines = lines[list(set(all_idx)-set(to_del))]

    lines = new_lines

    #print(lines)

    lines_img = color_img.copy()

    for line in lines:

        #print('line = ', line)

        r = line[0]

        theta = line[1]

        x = np.cos(theta)

        y = np.sin(theta)

        x0 = r*x

        y0 = r*y
        x1 = int(x0 + 1000*(-1*np.sin(theta)))
        y1 = int(y0 + 1000*np.cos(theta))
        x2 = int(x0 - 1000*(-1*np.sin(theta)))
        y2 = int(y0 - 1000*np.cos(theta))
        cv2.line(lines_img, (x1,y1), (x2,y2), (255, 0, 0), 1)
    cv2.imwrite(path + 'lines/' + fileName, lines_img)
    #cv2.imshow('lines', lines_img)
    #Finding corners----------------------------
    horizontal = []; vertical = []
    idx = 0; h_idx = []; v_idx = []
    for line in lines:
        #print(idx, ') line = ', line)
        if abs(line[1]-(np.pi/2)) < (np.pi/4):
            horizontal.append(line)
            h_idx.append(idx)
        else:
            vertical.append(line)
            v_idx.append(idx)
        idx += 1
    assert(len(horizontal) + len(vertical) == len(lines)) #make sure all lines are classified
    #assert(len(horizontal) == 10 and len(vertical) == 8)
    horizontal = sorted(horizontal, key = lambda x: x[0]*np.sin(x[1]))
    vertical = sorted(vertical, key = lambda x: x[0]*np.cos(x[1]))

    corners = []
    for hline in horizontal:
        hc_horiz = homogeneous_from_polar(hline)
        for vline in vertical:
            hc_vert = homogeneous_from_polar(vline)
            corner = np.cross(hc_horiz, hc_vert)
            corner /= corner[2]
            corners.append(corner)
    #print('corners = ', corners)
    corners_img = color_img.copy()
    #--------------------------------------------
    #removing redundant corners:
    #new_corners = list(map(lambda x: tuple(x), corners));
    if fileName == 'Pic_13.jpg' or fileName == 'pic_13.jpg':
        new_corners=list(map(lambda x: tuple(x),corners))
        #print('len(new_corners) = ',len(new_corners))
        for i in range(len(corners)):
            for j in range(i+1,len(corners)):
                if (abs(corners[i][0]-corners[j][0]) <=20 and abs(corners[i][1]-corners[j][1]) <=20):
                    if tuple(corners[i]) in new_corners:
                        new_corners.remove(tuple(corners[i]))

        #print('len(new_corners) = ',len(new_corners))
        corners = list(set(new_corners))
        corners = sorted(corners, key=lambda x:x[1])
    #-----------------------------------------------
    idx=0
    #print('len(corners) =', len(corners))
    #print('corners = ',corners)
    for corner in corners:
        idx += 1
        pt = tuple(map(int, corner))[:2]
        cv2.circle(img = corners_img, center = pt, radius = 1, color = (0, 0, 255))
        cv2.putText(img = corners_img, text = str(idx), org = pt, fontFace = cv2.FONT_HERSHEY_SIMPLEX,
fontScale = 0.3, color = (0, 0, 255))

    cv2.imwrite(path + 'corners/' + fileName, corners_img)

    #print(idx)

    return corners

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

def find_result(im1, im2, pts1, H):

    '''Returns the resulting image'''

    #creating the 'dummy' image which is completely blacked out except at the pixels which need to be
replaced

    '''dummy = np.zeros((im1.shape[0],im1.shape[1],3),dtype='uint8') #completely blacked out

    pts = np.array([[pts1[0][1], pts1[0][0]], [pts1[1][1], pts1[1][0]], [pts1[2][1], pts1[2][0]], [pts1[3][1],
pts1[3][0]]], np.int32) #pixels that need to be whitened

    pts = pts.reshape((-1,1,2))

    cv2.fillPoly(dummy,[pts],(255,255,255)) #whitening those pixels'''

    dummy = np.zeros((640,480,3),dtype='uint8') #completely blacked out


    cv2.imwrite(path + 'dummy.jpg', dummy)

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

    #Filling im1 with distorted im2:

    for i in range(im1.shape[0]): #till 2709

        for j in range(im1.shape[1]): #till 3612

            if dummy[i][j][0] == 0: #change the contents

                source = np.matmul(H, [[i], [j], [1]]);#print(source, '\n')

                source /= source[2][0]#; print(source)

                #print('source = ', source)

                if source[0][0] > 0 and source[1][0] > 0 and source[0][0] < im2.shape[0] and source[1][0] <
im2.shape[1]:

                    im1[i][j] = im2[int(source[0][0]), int(source[1][0])]
                #print('changing')

    return im1

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

def find_homography(x, x_dash):

    '''x and x_dash are lists of four lists, each list containing two coordinates: [x,y]

    returns H = inv(P)*t'''

    assert len(x) == len(x_dash)

    num_pts = len(x)

    t = np.array(x_dash[:num_pts], dtype = float).flatten().reshape((num_pts*2,1))

    P = []

    for i in range(num_pts):

        P.append([x[i][0], x[i][1], 1, 0, 0, 0, -1*x[i][0]*x_dash[i][0], -1*x[i][1]*x_dash[i][0]])

        P.append([0, 0, 0, x[i][0], x[i][1], 1, -1*x[i][0]*x_dash[i][1], -1*x[i][1]*x_dash[i][1]])

    P = np.array(P, dtype = float)#; print(P)

    P_inv = np.linalg.pinv(P)

    H = np.matmul(P_inv, t) # H = inv(P)*t

    H = np.insert(H, 8, 1).reshape((3,3))

    return H

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

def find_omega(H):

    def find_v(h):

        def find_vij(i, j):

            return np.array([h[0,i]*h[0,j],

                        h[0,i]*h[1,j]+h[1,i]*h[0,j],

                        h[1,i]*h[1,j],

                        h[2,i]*h[0,j]+h[0,i]*h[2,j],

                        h[2,i]*h[1,j]+h[1,i]*h[2,j],
                       h[2,i]*h[2,j]])
        v = np.array([find_vij(0,1), find_vij(0,0) - find_vij(1,1)])
        assert v.shape == (2,6)
        return v
    to_stack = []
    for i in range(len(H)):
        to_stack.append(find_v(H[i]))
    V = np.vstack(tuple(to_stack))
    #V = np.vstack((find_v(H[0]), find_v(H[1]), find_v(H[2])))
    #print(V)
    assert V.shape == (len(H)*2,6)
    #print('V.shape = ', V.shape)
    #Linear least squares
    u,d,v_t = np.linalg.svd(V)
    v = v_t.transpose()
    #print('v_t.shape = ', v_t.shape)
    b = v[:,v.shape[1]-1] #last col of v
    #print('b = ', b)
    omega = np.array([[b[0], b[1], b[3]], [b[1], b[2], b[4]], [b[3], b[4], b[5]]])
    assert omega.shape == (3,3)
    #print('omega = ', omega)
    return omega
#=====================================================================================
==================
def find_k(omega):
    x0 = (omega[0,1]*omega[0,2] - omega[0,0]*omega[1,2])/(omega[0,0]*omega[1,1] -
omega[0,1]*omega[0,1])
    lambdaa = omega[2,2] - ((omega[0,2]**2 + x0*(omega[0,1]*omega[0,2] -
omega[0,0]*omega[1,2]))/omega[0,0])
    alpha_x = np.sqrt(lambdaa/omega[0,0])
    alpha_y = np.sqrt(lambdaa*omega[0,0]/(omega[0,0]*omega[1,1] - (omega[0,1]**2)))

    s = -1*omega[0,1]*alpha_x*alpha_x*alpha_y/lambdaa

    y0 = (s*x0/alpha_y) - (omega[0,2]*alpha_x*alpha_x/lambdaa)

    K = np.array([[alpha_x, s, x0], [0, alpha_y, y0], [0, 0, 1]])

    print('K = \n', K)

    return K

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

def get_extrinsic(K, h):

    K_inv = np.linalg.inv(K)

    epsilon = 1/(np.linalg.norm(np.matmul(K_inv, h[:,0])))

    r1 = epsilon*np.matmul(K_inv, h[:,0])

    r2 = epsilon*np.matmul(K_inv, h[:,1])

    t  = epsilon*np.matmul(K_inv, h[:,2])

    r3 = np.cross(r1, r2)

    print('r1 = {}\nr2 = {}\nr3 = {}\nt = {}'.format(r1,r2,r3,t))

    '''R = np.vstack(r1,r2,r3).T

    u,d,v_t = np.linalg.svd(R); R = np.matmul(u, v_t.T)

    r1 = R[:,0]; r2 = R[:,1]; r3 = R[:,2]'''

    return (r2, r1, r3, t)

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

def rodr(R):

    phi = np.acos((np.trace(R)-1)/2)

    w = (phi/(2*np.sin(phi)))*np.array(R[3,2]-R[2,3], R[1,2]-R[2,1], R[2,1]-R[1,2])

    return (w,phi)

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

def anti_rodr(w,phi):

    W=np.array([[0,-1*w[2],w[1]],[w[2],0,-1*w[0]],[-1*w[1],w[0],0]])
    R=np.identity(3) + (np.sin(phi)/phi)*W + ((1-np.cos(phi))/phi)*np.square(W)
    return R
#==================================================================================================
def main():
    world_corners = gen_world_corners()
    img13_corners=find_corners('Pic_13.jpg')
    #image 13 is the fixed image in both datasets
    H = [] #list of homographies
    fn_list = []
    #for i in [1,2,3,4,6,13,15,17,29,33,34,35,36,38]:
    #   fn_list.append('Pic_{}.jpg'.format(i))
    for i in [2,8,11,12,13]:
        fn_list.append('Pic_{}.jpg'.format(i))
    fn_list = np.array(fn_list)
    for fileName in fn_list:
        img = cv2.imread(path + 'Dataset2/' + fileName)
        corners = find_corners(fileName)
        def switch(x):
            return (x[1], x[0])
        switched_corners = list(map(switch, corners))
        switched_world_corners = list(map(lambda x: (x[1], x[0]), world_corners))
        h = find_homography(switched_corners, switched_world_corners) #from world to pixels
        #print('h = ' , h)
        if fileName != 'Pic_13.jpg' and fileName != 'pic_13.jpg':
            H.append(np.linalg.pinv(h))

    H = np.array(H)
    omega = find_omega(H)
    K = find_k(omega)

    #finding intrinsic parameters for img 13

    (r1, r2, r3, t) = get_extrinsic(K, H[3]) #for pic 13

    P_13 = np.vstack((r1, r2, r3, t)).T #3x4

    P_13 = np.matmul(K, P_13) #3x4


    #K = np.array([[943.53,1.77,319.8], [0,942.89,235.3], [0,0,1]])

    '''#Checking if homography is correct:

    blank = np.zeros((640,480,3),dtype='uint8') #completely blacked out

    pts1      = np.array([[0   ,0   ], [0,640    ], [480,640  ], [480,0    ]], dtype = float)

    cv2.imwrite(path + 'result4.jpg', find_result(blank, img, pts1, np.linalg.pinv(h)))'''

    #---------------Without LM---------------------------

    idx = 0

    P_long = [K]; corners_long = []

    #for i in [1,2,3,4,6,15,17,29,33,34,35,36,38]:

    for i in [2,8,11,12]:

        img = cv2.imread(path + 'Dataset2/' + 'pic_{}.jpg'.format(i))

        corners = find_corners('pic_{}.jpg'.format(i)) #corners of this picture

        (r1, r2, r3, t) = get_extrinsic(K, H[idx]) #getting extrinsic params for i-th picture

        P = np.vstack((r1, r2, r3, t)).T #3x4

        P = np.matmul(K, P) #3x4

        R = np.vstack(r1,r2,r3).T

        (w,phi) = rodr(R)

        P_long.extend(R); P_long.extend(t)

        corners_long.append(corners)

        euclid_unopt = []; euclid_opt = []

        #Projection:

        for j in range(80):

            #print('corners[j] = ', corners[j])
            cam_cor = np.matmul(P, [world_corners[j][0], world_corners[j][1], 0, 1]) #3x1

            cam_cor /= cam_cor[2]

            actual_c = (int(corners[j][0]), int(corners[j][1]))


            c_c = (int(cam_cor[1]), int(cam_cor[0]))




            cv2.circle(img = img, center = c_c, radius = 1, color = (0, 0, 255))

            cv2.putText(img = img, text = str(j+1), org = c_c, fontFace = cv2.FONT_HERSHEY_SIMPLEX,

                        fontScale = 0.3, color = (0,0,255))

            cv2.circle(img = img, center = actual_c, radius = 1, color = (0,255,0))

            cv2.putText(img = img, text = str(j+1), org = actual_c, fontFace = cv2.FONT_HERSHEY_SIMPLEX,

                        fontScale = 0.3, color = (0,255,0))

        cv2.imwrite(path + 'projection/' + 'pic_{}.jpg'.format(i), img)

        idx += 1

    #-------------------------With LM---------------------------

    def fun(P):

        return np.linalg.norm(corners_long.flatten() - P_long)

    P1 = optimize.root(fun, P, method='lm').x

    idx = 1

    for i in [2,8,11,12]:

        w = P1[idx+1:idx+3]

        t = P1[idx+4:idx+6]

        R = anti_rodr(w,phi)

        P_optimized = np.vstack(R,t).T

        idx += 6

        img = cv2.imread(path + 'Dataset2/' + 'pic_{}.jpg'.format(i))

        corners = find_corners('pic_{}.jpg'.format(i)) #corners of this picture

        (r1, r2, r3, t) = get_extrinsic(K, H[idx]) #getting extrinsic params for i-th picture
    P = np.vstack((r1, r2, r3, t)).T #3x4

    P = np.matmul(K, P) #3x4

    R = np.vstack(r1,r2,r3).T

    euclid_unopt = []; euclid_opt = []

    #Projection:

    for j in range(80):

      #print('corners[j] = ', corners[j])

      cam_cor = np.matmul(P, [world_corners[j][0], world_corners[j][1], 0, 1]) #3x1

      cam_cor /= cam_cor[2]


      optimized_cam_cor=np.matmul(P_optimized, [world_corners[j][0], world_corners[j][1], 0, 1])

      actual_c = (int(corners[j][0]), int(corners[j][1]))


      c_c = (int(cam_cor[1]), int(cam_cor[0]))

      o_c = (int(optimized_cam_cor[0]), int(optimized_cam_cor[1]))



      cv2.circle(img = img, center = c_c, radius = 1, color = (0, 0, 255))

      cv2.putText(img = img, text = str(j+1), org = c_c, fontFace = cv2.FONT_HERSHEY_SIMPLEX,

            fontScale = 0.3, color = (0,0,255))

      cv2.circle(img = img, center = actual_c, radius = 1, color = (0,255,0))

      cv2.putText(img = img, text = str(j+1), org = actual_c, fontFace = cv2.FONT_HERSHEY_SIMPLEX,

            fontScale = 0.3, color = (0,255,0))

      cv2.circle(img = img, center = o_c, radius = 1, color = (255, 0, 0))

      cv2.putText(img = img, text = str(j), org = o_c, fontFace = cv2.FONT_HERSHEY_SIMPLEX, fontScale
= 0.3, color = (255, 0, 0))


      #print('actual = {}, unoptimized = {}, optimized = {}, norm = {}'.format(corners[j][:2],cam_cor[:2],
optimized_cam_cor[:2],np.linalg.norm(corners[j][:2]-cam_cor[:2])))
```

```
            euclid_unopt.append(np.linalg.norm(corners[j][:2]-[cam_cor[1],cam_cor[0]]))
            euclid_opt.append(np.linalg.norm(corners[j][:2]-optimized_cam_cor[:2]))


        print('\n\nFor image ', i)
        #print('euclid_unopt = ', euclid_unopt)
        print('Mean of euclidean distance between unoptimized corners and actual corners = ',
              np.mean(euclid_unopt))
        print('Variance of euclidean distance between unoptimized corners and actual corners = ',
              np.var(euclid_unopt))
        print('Mean of euclidean distance between optimized corners and actual corners = ',
              np.mean(euclid_opt))
        print('Variance of euclidean distance between optimized corners and actual corners = ',
              np.var(euclid_opt))
        cv2.imwrite(path + 'projection/' + 'pic_{}_lm.jpg'.format(i), img)
#==================================================================================
==================
if __name__ == '__main__':
    main()
```