ECE 661: Homework 8

Due on Tuesday, November 13st, 2018

Dr. Avinash Kak

Naveen Madapana

1. Extracting corners of checkerboard pattern

Identifying the corners of the black squares in the given checkerboard pattern is a key step in camera calibration. The procedure to estimate the location of corners is two fold: first, we need to apply Canny edge detection algorithm to identify the edges and second, we need to apply Hough Transform to identify the straight lines. Next, the obtained the lines are pre-processed to remove the false positives, duplicates (multiple lines close to the true line) and the outliers.

1.1 Canny Edge Detection

In this homework, the Canny edge detection implementation of OpenCV was used. This function consists of two hyperparameters: 1. (***threshold_ min*** and 2. ***threshold_ max***), where ***threshold_ min*** determines how the edges are linked and ***threshold_ max*** helps determine the strong edges. In both the datasets (given and self made), the values are 100 and 200 respectively.

1.2 Hough Lines

Once the Canny algorithm is applied to identify the edges, the next step is to apply the Hough transform to detect the straight lines. Specifically, we want to detect 18 lines (10 horizontal and 8 vertical). Further, there are multiple lines that are close to the true line. Hence, we want to eliminate the false positives and duplicates. Appropriate thresholding has been used to remove the unnecessary lines.

In Cartesian plane, a line is described by two points $l(x_1, x_2)$, where x_1 and x_2 are the two points that lie on the line. The poloar representation of this line is given as $l(\rho, \theta)$, where ρ and θ follow the equation below:

$$\rho = x_1 sin(\theta) + x_2 cos(\theta)$$

ρ is the shortest distance between the line and the origin and θ is the counterclockwise angle between the X axis and the line.

The values of ρ ranges between 0 and the maximum of no. of rows and no. of columns (D) in the image. And the value of θ ranges between 0 and π. Hence the sampling distance of ρ is $1/D$, while the sampling distance for θ is $1/\pi$.

In Hough transform, for each edge detected in the Canny, we compute the ρ and θ and update the corresponding bins. The values of the bins are thresholded to identify the lines.

OpenCV implementation of Hough Transform is used to identify the lines. The parameters set are: resolution of ρ is 1, resolution of θ is $1/\pi$, and the bin threshold is 50 or 40.

These lines are used to compute the corners. As expected, it gives a large number of corners due to the presence of duplicate lines. Such lines are eliminated using another thresholding step.

1. For each line $l(\rho, \theta)$:
    - Horizontal line if $abs(cos(\theta)) > abs(sin(\theta))$

- Vertical line if $abs(cos(\theta)) \leq abs(sin(\theta))$

2. select a set of vertical lines $l_{v1}, l_{v2}, \ldots$
3. for each horizontal line l_{h_i}:
   - find the intersection between l_{vj} and l_{h_i}
4. Delete l_{h_i} if it yields intersection points very close to another horizontal line. This threshold is set to 10 pixels.
5. Repeat the same process by interchanging horizontal and vertical lines.

2. Generating Correspondences

Next important step in the calibration is to define the world coordinates of the corners of the calibration pattern. There are a set of 80 coordinates. OpenCV convention of x and y axes is used to define the coordinates. The size of the square blocks is found to be 25 mm. This value is used to give a world coordinate to each corner with respective to the top left corner of the pattern.

The points are depicted in the matrix below

$$\begin{bmatrix} (0,0) & (25,0) & (50,0) & \ldots & (175,0) \\ (0,25) & (25,25) & (50,25) & \ldots & (175,25) \\ & & \vdots & & \\ (0,225) & (25,225) & (50,225) & \ldots & (175,225) \end{bmatrix}$$

Now we need to find a ground truth correspondence between he world coordinate and the corresponding image corner. The lines obtained using Hough transform are cleaned to obtain 10 horizontal lines and 8 vertical lines. Following procedure is adopted to obtain the image corner points in the right order.

1. Sort the horizontal lines based on the y intercept and sort the vertical lines based on x intercept.
2. Now intersect the first horizontal line (the one with lowest y intercept) with all of the vertical lines and find the intersection points.
3. Repeat this procedure for all horizontal lines in the same order (ascending).

Now, we have point to point correspondences between the image corners and their respective world coordinates.

3. Camera Calibration - Zhang's Algorithm

Zhang's algorithm is described in this section. Once, we have point to point correspondences between the image corners and their respective world coordinates, we need to estimate a homography between the world coordinates and image corner pixels. Next an initial estiamte of the parameters K, P and t that describe the camera matrix P, where $P = K[R|\vec{t}]$ was made.

3. Camera Calibration - Zhang's Algorithm continued on next page...

Homography estimation

Let $c = \begin{bmatrix} u_1 & v_1 & w_1 \end{bmatrix}^T$ and $c' = \begin{bmatrix} u_2 & v_2 & w_2 \end{bmatrix}^T$ be the homogenous coordiantes of image pixel and world coordinates.

In this approach, it is assumed that we have a one to one correspondence between pixels in the domain plane to the pixels in the range plane. Overall, we need atleast four correspondences (pairs of points) in order to compute the homography in this method.
Without the loss of generality, it can be assumed that $w_1 = 1$ as we are working with homogeneous coordinates. By simplifying the equation further and rearranging the equation in the form of $Ah = 0$, where h is a 9×1 vector consisting of elements in the homography H, we will get,

$$u_2(H_{31}u_1 + H_{32}v_1 + H_{33}) = H_{11}u_1 + H_{12}v_1 + H_{13}w_1$$
$$v_2(H_{31}u_1 + H_{32}v_1 + H_{33}) = H_{21}u_1 + H_{22}v_1 + H_{23}w_1$$

Hence, each pair of matching point will yield two equations (for x and y). where h is a 9×1 vector consisting of elements in the homography H and A is a $2m \times 9$ matrix (m being the number matching pairs of points selected to estimate h). Let a_u and a_v be the coefficients corresponding to x and y dimensions.

$$Ah = 0$$

Where:

$$\begin{bmatrix} -u_1 & -v_1 & -1 & 0 & 0 & 0 & u_1u_2 & v_1u_2 & u_2 \\ 0 & 0 & 0 & -u_1 & -v_1 & -1 & u_1v_2 & v_1v_2 & v_2 \end{bmatrix} h = 0$$
$$h = \begin{bmatrix} H_{11} & H_{12} & H_{13} & H_{21} & H_{22} & H_{23} & H_{31} & H_{32} & H_{33} \end{bmatrix}^T$$

Now if we choose m pairs of points we will get the system $Ah = 0$:

$$\begin{bmatrix} -u_{11} & -v_{11} & -1 & 0 & 0 & 0 & u_{11}u_{12} & v_{11}u_{12} & u_{12} \\ 0 & 0 & 0 & -u_{11} & -v_{11} & -1 & u_{11}v_{12} & v_{11}v_{12} & v_{12} \\ -u_{21} & -v_{21} & -1 & 0 & 0 & 0 & u_{21}u_{22} & v_{21}u_{22} & u_{22} \\ 0 & 0 & 0 & -u_{21} & -v_{21} & -1 & u_{21}v_{22} & v_{21}v_{22} & v_{22} \\ & & & & \vdots & & & & \\ -u_{n1} & -v_{n1} & -1 & 0 & 0 & 0 & u_{n1}u_{n2} & v_{n1}u_{n2} & u_{n2} \\ 0 & 0 & 0 & -u_{n1} & -v_{n1} & -1 & u_{n1}v_{n2} & v_{n1}v_{n2} & v_{n2} \end{bmatrix} \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \\ H_{33} \end{bmatrix} = 0$$

Hence, h is the eigenvector of A with lowest eigenvalue. Therefore, we compute SVD of A i.e. $A = USV^T$ and choose the last column of V^T. Now, we reshape h in order to get a $H_{3\times 3}$.

Estimating ω

Let ω be the image of the absolute conic Ω_∞ in the image plane. Let $H = [\overrightarrow{h_1}, \overrightarrow{h_2}, \overrightarrow{h_3}]$ be the homography that maps the points in the world coordinates to the corners in the image pixels. Then we can establish the following relationship between H and ω:

$$\overrightarrow{h_1} W \overrightarrow{h_2} = 0$$
$$\overrightarrow{h_1} W \overrightarrow{h_1} = \overrightarrow{h_2} W \overrightarrow{h_2}$$

The 3 x 3 matrix, ω is symetric, and there are six independent parameters. Let $b = [w_{11}, w_{12}, w_{22}, w_{13}, w_{23}, w_{33}]$. Now, we can rewrite the above two equations as a matrix vector product. The image of the calibration pattern is taken from different camera positions. Each position will yield a set of two equations. If there are n camera positions, then there will be $2n$ equations and six variables.

$$\begin{bmatrix} V_{12_1}^T \\ V_{11_1}^T - V_{22_1}^T \\ \vdots \\ V_{12_n}^T \\ V_{11_n}^T - V_{22_n}^T \\ \end{bmatrix} b = 0$$

The value of V_{ij} is given by the following:

$$V_{ij} = \begin{bmatrix} h_{1i}h_{1j} \\ h_{1i}h_{2j} + h_{2i}h_{1j} \\ h_{2i}h_{2j} \\ h_{3i}h_{1j} + h_{1i}h_{3j} \\ h_{3i}h_{2j} + h_{2i}h_{3j} \\ h_{3i}h_{1j3} \end{bmatrix}$$

Now the system of linear equations $Vb = 0$ needs to be solved to find ω.

Estimating intrinsic Parameters

Once we obtained b, the matrix ω is constructed. We know the relation between ω and the camera intrinsic parameter matrix K (5 degrees of freedom) as:

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

Where,

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

Simplifying the relation between ω and K, followed by further estimation of scale λ from the matrix ω gives the following closed form equations.

$$x_0 = \frac{w_{12}w_{13} - w_{11}w_{23}}{w_{11}w_{22} - w_{12}^2},$$

$$\lambda = w_{33} = \frac{w_{13}^2 + x_o(w_{12}w_{13} - w_{11}w_{23})}{w_{11}},$$
$$\alpha_x = \sqrt{\frac{\lambda}{w_{11}}},$$
$$\alpha_y = \sqrt{\frac{\lambda w_{11}}{w_{11}w_{22} - w_{12}^2}},$$
$$s = -\frac{w_{12}\alpha_x^2\alpha_y}{\lambda},$$
$$y_o = \frac{sx_0}{\alpha_y} - \frac{w_{13}\alpha_x^2}{\lambda}$$

Estimating extrinsic Parameters

Once we estimated K, we need to estimate R (rotation matrix) and t (translation vector). The relationship between K, and $R = [\overrightarrow{r_1}, \overrightarrow{r_2}, \overrightarrow{r_3}]$ and $\overrightarrow{t}$ is described as:

$$K^{-1}H = K^{-1}[\overrightarrow{h_1}, \overrightarrow{h_2}, \overrightarrow{h_3}] = [\overrightarrow{r_1}, \overrightarrow{r_2}, \overrightarrow{t}],$$

Without the loss of generality, we assume that, Z = 0 on the calibration patter.

From the equation above we can determine the following formulas for the values of R and $\overrightarrow{t}$:

$$\overrightarrow{r_1} = \zeta K^{-1}\overrightarrow{h_1}$$
$$\overrightarrow{r_2} = \zeta K^{-1}\overrightarrow{h_2}$$
$$\overrightarrow{r_3} = \overrightarrow{r_1} \times \overrightarrow{r_2} \text{ , since R is orthogonal}$$
$$\overrightarrow{t} = \zeta K^{-1}\overrightarrow{h_3}$$

$$\zeta = \frac{1}{\|K^{-1}\overrightarrow{h_1}\|} \text{ , since R is orthonormal}$$

The rotation matrix obtained using this process may not be orthogonal. Hence SVD decomposition is used to obtain the conditioned rotation matrix.

4. Levenberg Marquadt Algorithm

Nonlinear refinement of the estimated parameters is essential in the robust estimation of the camera intrinsic and extrinsic parameters. In LM, we try to minimize the geometric error.

$$\text{Minimize } d^2_{geometric} = \|\overrightarrow{X}_{img} - \overrightarrow{f}(\overrightarrow{p})\|$$

$\vec{X}_{img}$ is the image pixel coordinate of a corner, let $\vec{f}(\vec{p})$ be the estimate of that point computed using re-projection and p is a function of $\vec{r_1}$, $\vec{r_2}$ and t.

The above equation can be re-written as: $\vec{x}_{ij}$ - image pixel coordinates and real world corner point $\vec{x}_{ij_world}$

$$\text{Min } d^2_{geometric} = \sum_i \sum_j \|\vec{x}_{ij} - K[R_i|\vec{t}_i]\vec{x}_{ij_world}\|$$

The 3 x 3 rotation matrix has only three degrees of freedom. Hence the Rodrigues representation is used to convert the nine rotation numbers to three angle values. Also, it is very easy to go back and forth between the r3 x 3 matrix and angle representations. The equations are given below as follows:

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

$$R = I_{3\times3+\frac{sin(\phi)}{\phi}}[\vec{w}]_x + \frac{1-cos(\phi)}{\phi^2}[\vec{w}]^2_x, \quad \text{Given } \vec{w}$$

$$\phi = cos^{-1}\frac{tr(R)-1}{2}, \ \vec{w} = \frac{\phi}{2sin(\phi)}\begin{bmatrix} r_{32} & -r23 \\ r_{13} & -r31 \\ r_{21} & -r12 \end{bmatrix}, \quad \text{For a given R}$$

6. Results

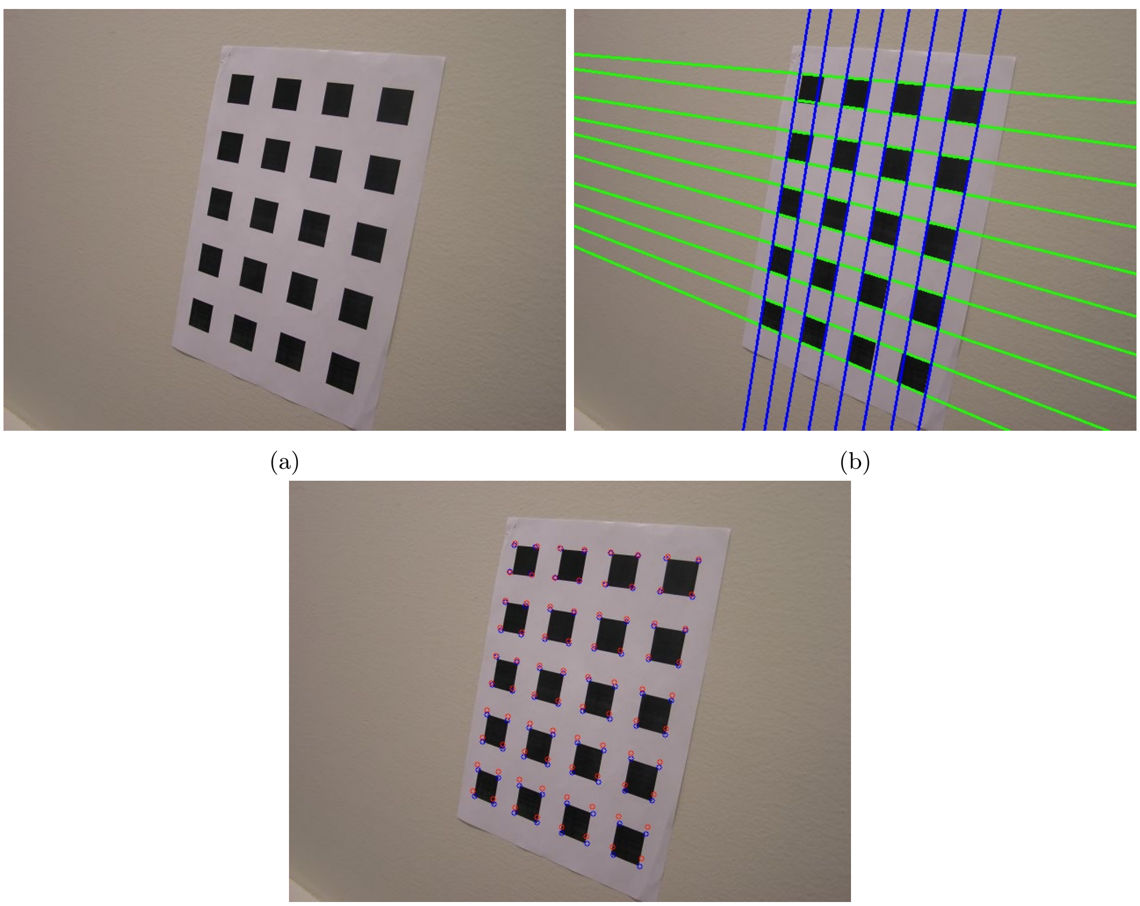
In this section, sample results are presented.

Dr. Avinash Kak's LM Module is used to refine the homogprahy parameters in the non linear manner. However, it was found that the error in the number of pixels is very similar. It is better than 2-3 pixels when compared to when we do not use the LM at all.

6.1 Given Images

$$K = \begin{bmatrix} 713.807 & 10.324 & 319.731 \\ 0. & 710.177 & 240.384 \\ 0. & 0. & 1. \end{bmatrix}$$

Image 1



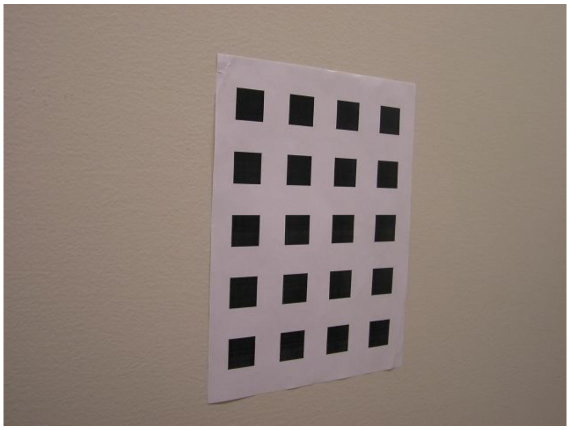
(a) (b)

(c)

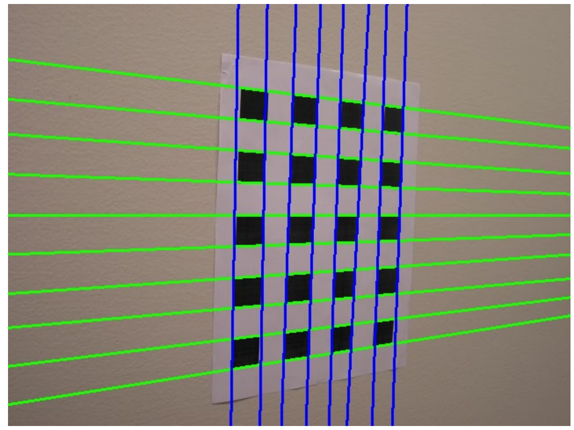
Figure 1: Original image, image with horizontal and vertical lines and results of reprojection.

6. Results continued on next page...

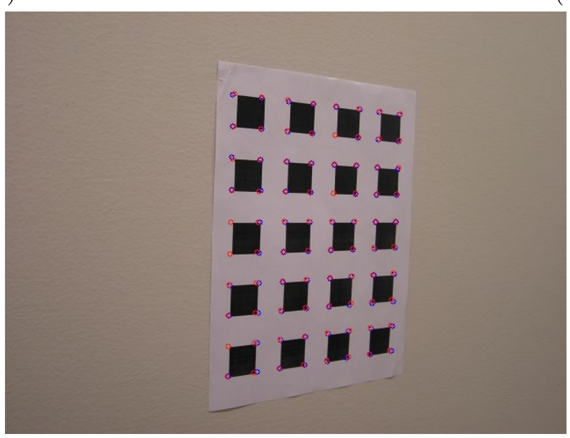
Image 2



(a)



(b)

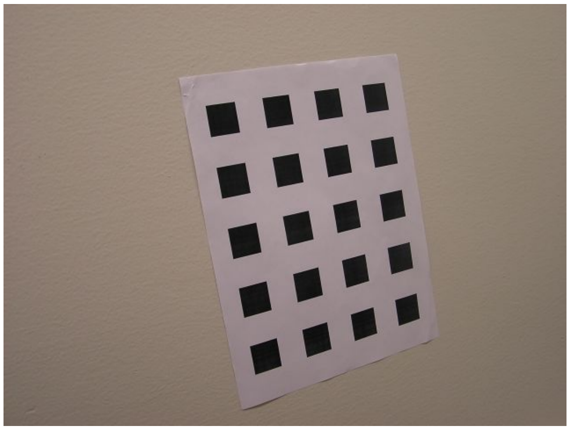
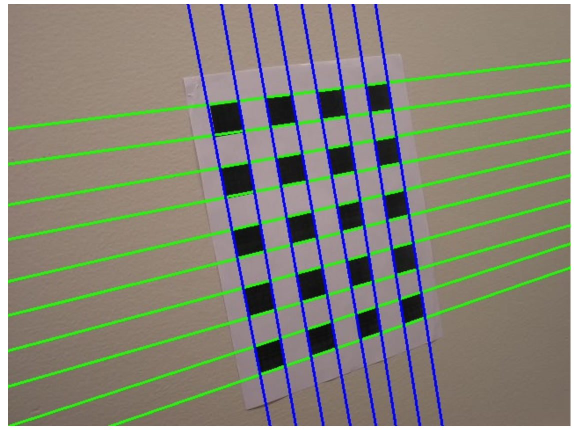


(c)

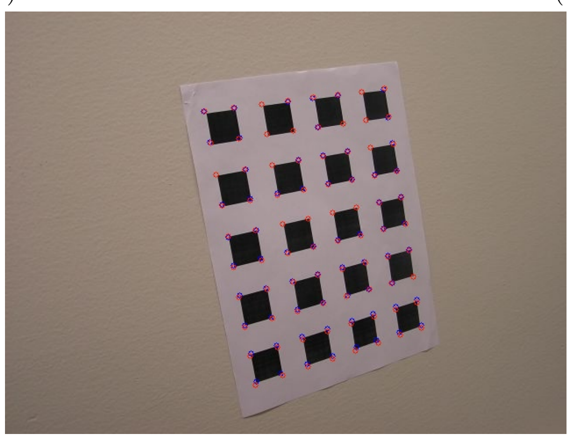
Figure 2: Original image, image with horizontal and vertical lines and results of reprojection.

6. Results continued on next page...

Image 3

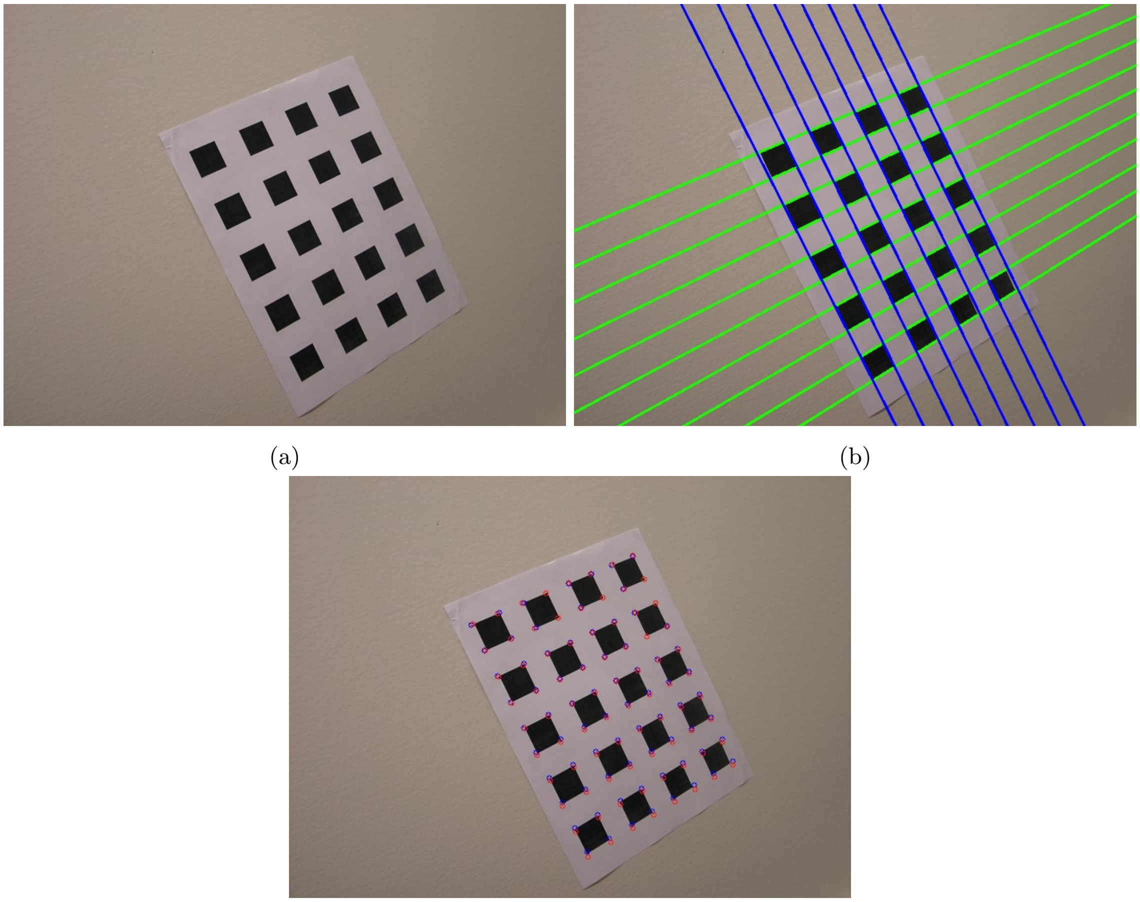
(a) (b)



(c)

Figure 3: Original image, image with horizontal and vertical lines and results of reprojection.

Image 4



(a) (b)

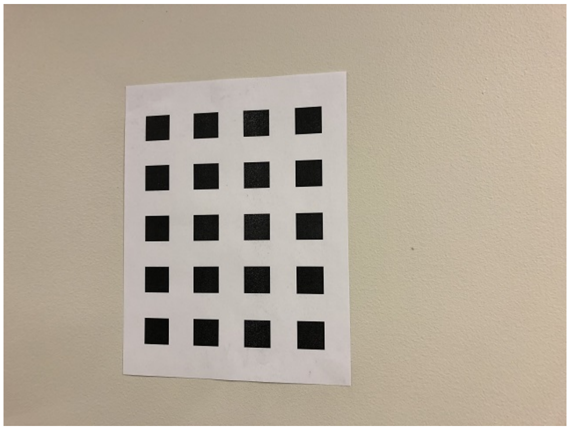
(c)

Figure 4: Original image, image with horizontal and vertical lines and results of reprojection.

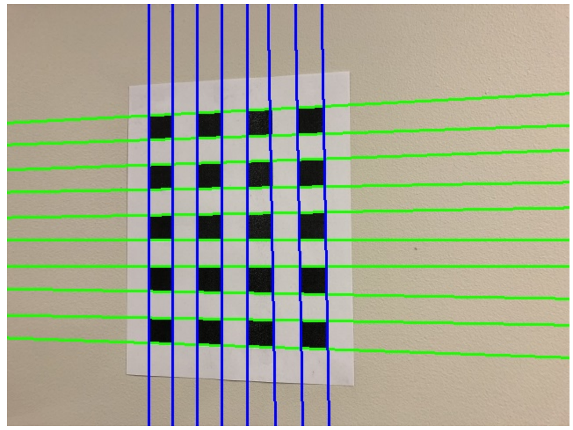
6.1 Self taken images

$$K = \begin{bmatrix} 534.879 & 6.184 & 321.109 \\ 0. & 534.478 & 224.717 \\ 0. & 0. & 1. \end{bmatrix}$$

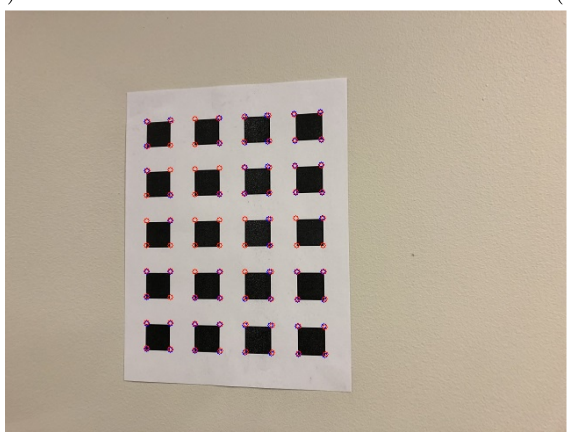
Image 1



(a)



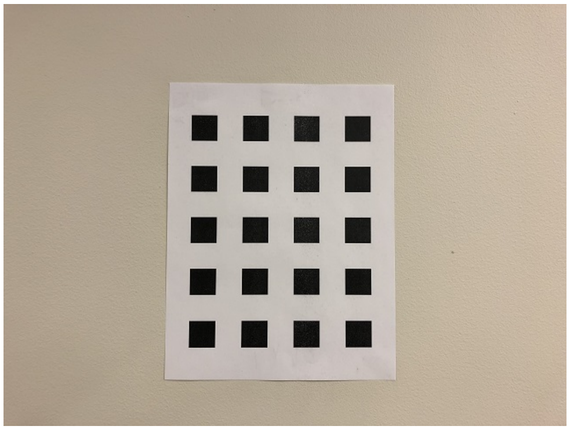
(b)



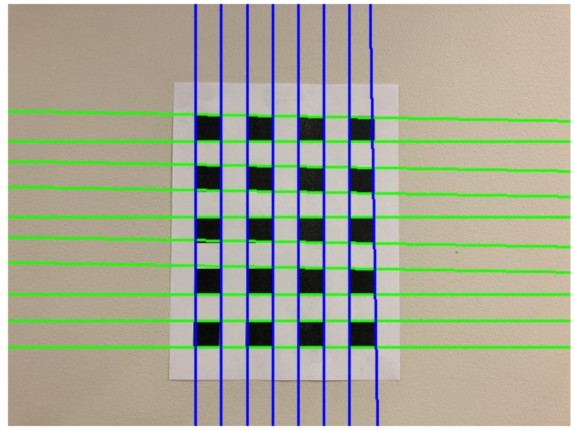
(c)

Figure 5: Original image, image with horizontal and vertical lines and results of reprojection.

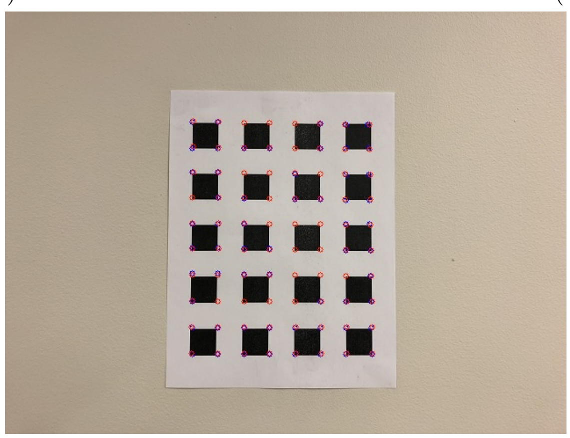
Image 2



(a)



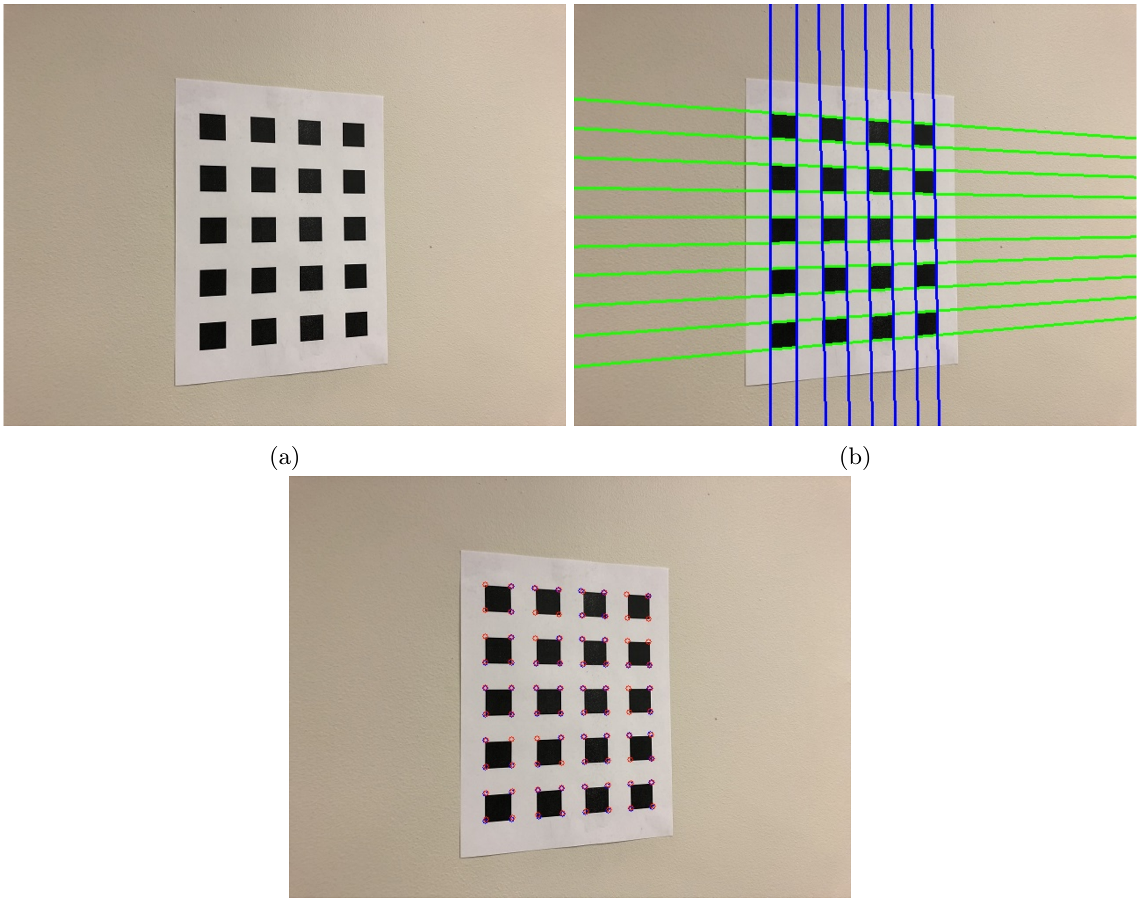
(b)



(c)

Figure 6: Original image, image with horizontal and vertical lines and results of reprojection.

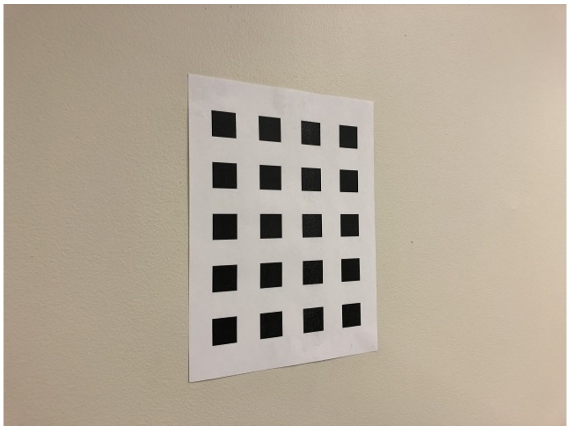
Image 3



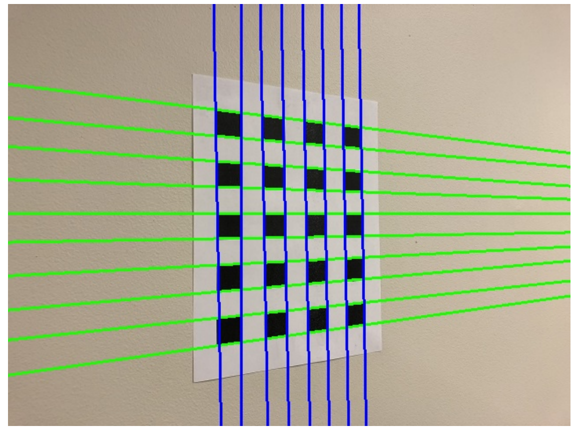
(a) (b) (c)

Figure 7: Original image, image with horizontal and vertical lines and results of reprojection.

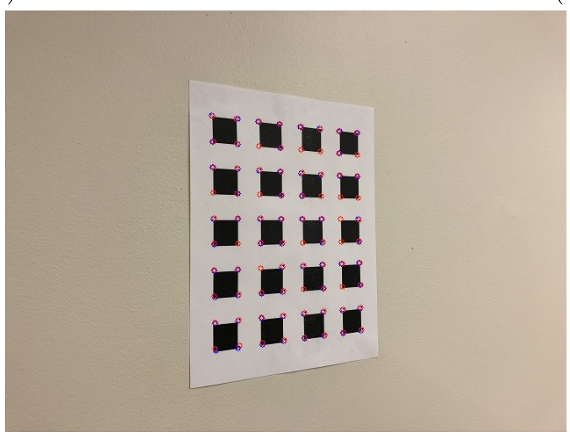
Image 4



(a)



(b)



(c)

Figure 8: Original image, image with horizontal and vertical lines and results of reprojection.

6. Code

Listing 1: HW8 code

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from scipy.spatial.distance import pdist
import sys, os, time
from helpers import *
import glob
from copy import deepcopy
from NonlinearLeastSquares import NonlinearLeastSquares as NLS

##############################
########### HELPERS ##########
##############################

def get_v_rep(H, p, q):
  '''
  Description:
    V(H, p, q) is computed. It is a part of Zhang's algorithm for camera callibration.
  Input argumnets:
    * H - 2D square np.ndarray of size 3 x 3
    * p - {0, 1, 2}
    * q - {0, 1, 2}
  Return:
    V(H, p, q)
  '''
  assert isinstance(H, np.ndarray), 'H should be a numpy array'
  assert H.ndim == 2, 'H should be a 2D np.ndarray'
  assert H.shape[0] == H.shape[1], 'H should be a square matrix'
  assert p < 3 and p >= 0, 'p should be 0, 1, or 2'
  assert q < 3 and q >= 0, 'q should be 0, 1, or 2'

  # v0 = H[p, 0] * H[q, 0]
  # v1 = H[p, 0] * H[q, 1] + H[p, 1] * H[q, 0]
  # v2 = H[p, 1] * H[q, 1]
  # v3 = H[p, 2] * H[q, 0] + H[p, 0] * H[q, 2]
  # v4 = H[p, 2] * H[q, 1] + H[p, 1] * H[q, 2]
  # v5 = H[p, 2] * H[q, 2]

  v0 = H[0, p] * H[0, q]
  v1 = H[0, p] * H[1, q] + H[1, p] * H[0, q]
  v2 = H[1, p] * H[1, q]
  v3 = H[2, p] * H[0, q] + H[0, p] * H[2, q]
  v4 = H[2, p] * H[1, q] + H[1, p] * H[2, q]
  v5 = H[2, p] * H[2, q]

  return np.array([v0, v1, v2, v3, v4, v5])

def get_world_coordinates(pattern_size, unit_size, homo = False, display = False):
  '''
```

```
  Description:
    Compute the world coordinates of the checkerboard pattern of given size.
    Returns the word coo. in raster scan order i.e. row by row starting from the first row.
  Input arguments:
    * pattern_size = (height, width). For instance (9, 7) ==> (NUM_HORZ_LINES-1, NUM_VERT_LINES-
      * width: width of the pattern in terms of no. of blocks along x - axis.
      * height: height of the pattern in terms of no. of blocks along y - axis.
    * unit_size: A floating point value indicating the size of the block.
  Return:
    * mat: A 2D np.ndarray. Each row is the world point (x, y) either in physical or homogeneous
      * It looks like [[x1, y1, 1], [x2, y2, 1], ...]
  '''
  height, width = pattern_size
  height = height + 1
  width = width + 1

  mat = []
  for yidx in range(height):
    for xidx in range(width):
      if(homo):
        mat.append([xidx*unit_size, yidx*unit_size, 1])
      else:
        mat.append([xidx*unit_size, yidx*unit_size])

  ## Display the world coordinates for debugging purposes.
  if(display):
    scale = 20
    offset_x = 50
    offset_y  = 50
    img_width = int(width*unit_size*scale) + offset_x
    img_height = int(height * unit_size*scale) + offset_y
    img = 255 * np.ones((img_height, img_width, 3), dtype = np.uint8)
    for row in mat:
      x, y = int(row[0]*scale+offset_x), int(row[1]*scale+offset_y)
      cv2.circle(img, (x, y), 5, color = (255, 0, 0))
      cv2.imshow('World points', img)
      cv2.waitKey(0)

  return np.array(mat)

def order_lines(lines, type = 'h'):
  '''
  lines: list of elements. Each element is of form [x1, y1, x2, y2]
  type: 'h' for horizontal lines and 'v' for vertical lines
  '''
  intercept_list = []
  for idx, line in enumerate(lines):
    x1, y1, x2, y2 = tuple(line.tolist())
    line = np.cross([x1, y1, 1], [x2, y2, 1])
    if(type == 'h'):
      intercept = -1 * line[2] / line[1]
    elif(type == 'v'):
      intercept = -1 * line[2] / line[0]
```

```
    else:
      raise Exception('type should be "h" or "v"')
    intercept_list.append(intercept)
  argsort = np.argsort(intercept_list)
  return lines[argsort, :]

def plot_lines(img, lines, color = (0,255,25)):
  '''
  img: 2D or 3D np.ndarray
  lines: list of elements. Each element is a list: it looks like [x1, y1, x2, y2]
  '''
  img = np.copy(img)
  for line in lines:
    x1, y1, x2, y2 = tuple(line)
    cv2.line(img,(x1,y1),(x2,y2),color,2)
  cv2.imshow('Lines', img)
  cv2.waitKey(5)
  return img

def filter_white_points(img, points, kernel_sz = 10, thresh = 150, debug = False):
  '''
  points: 2D/3D np.ndarray. Rows look like [x1, y1].
  '''
  img = np.copy(img)
  flags = np.zeros(points.shape[0]).astype(int) == 0
  for idx, point in enumerate(points):
    point = point.astype(int)
    x, y = point[0], point[1]
    if(img.ndim == 2):
      temp = img[y-kernel_sz:y+kernel_sz, x-kernel_sz:x+kernel_sz].flatten()
    else:
      temp = img[y-kernel_sz:y+kernel_sz, x-kernel_sz:x+kernel_sz, :].flatten()
    max_min_diff = np.max(temp) - np.min(temp)
    if(debug):
      cv2.circle(img, (x, y), 10, color = [255, 0, 0])
      print max_min_diff
    if(max_min_diff < thresh):
      flags[idx] = False
    if(debug):
      cv2.imshow('', img)
      cv2.waitKey(0)
  return flags

def intersect_lines(pair1, pair2):
  '''
  pair1: A list [x1, y1, x2, y2], where (x1, y1) and (x2, y2) are start and end points of the li
  pair2: A list [x1, y1, x2, y2], where (x1, y1) and (x2, y2) are start and end points of the li
  '''
  line1 = np.cross([pair1[0], pair1[1], 1], [pair1[2], pair1[3], 1])
  line2 = np.cross([pair2[0], pair2[1], 1], [pair2[2], pair2[3], 1])
  point = np.cross(line1, line2)
  point = point / point[-1]
  return point[:-1].tolist()
```

```
def filter(M, thresh):
  '''
  Description:
    * Filters the rows in M. Eliminate the rows in M are
     very close according to euclidean norm between rows.
  M: 2D np.ndarray. Rows are features
  '''
  M = deepcopy(M)
  if(M.ndim == 1):
    M = M.reshape(-1, 1)
  flags = np.zeros(M.shape[0]).astype(int) == 0
  dist_mat = dist_mat_mat(M, M)
  max_val = 2 * np.max(M.flatten()) # Some big value
  for idx, row in enumerate(dist_mat):
    row[:idx+1] = max_val
    nz_ids = np.nonzero(row < thresh)[0]
    flags[nz_ids] = False
  return flags, M[flags, :]

def rth_to_xy(rth_arr):
  '''
  rth_arr: 2D np.ndarray. Each row has two elements (rho and theta).
  '''
  xy_arr = []
  for line in rth_arr:
    rho, theta = line[0], line[1]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    nline = np.cross([x1, y1, 1],[x2, y2, 1])
    nline = nline / np.max(nline)
    xy_arr.append(nline)
  return np.array(xy_arr)

def find_checkerboard_points(img_path, pattern_size, unit_size, display = False, out_dir = ''):
  '''
  Description:
    Return checker board edges
  Input arguments:
    img_path: Absolute path to the image of a checker board pattern
    * pattern_size = (height, width). For instance (9, 7) ==> (NUM_HORZ_LINES-1, NUM_VERT_LINES-
      * width: width of the pattern in terms of no. of blocks along x - axis.
      * height: height of the pattern in terms of no. of blocks along y - axis.
  Return:
    Return lines of checker board pattern in the form of rho and theta
  '''
```

```
  if(not os.path.isfile(img_path)):
    raise IOError('ERROR! ' + img_path + ' does NOT exists !!')

  num_horz_lines = pattern_size[0] + 1
  num_vert_lines = pattern_size[1] + 1

  color_img = cv2.imread(img_path)
  img = cv2.imread(img_path, 0) # Read image as grayscale
  height, width = img.shape
  diag_length = np.max([width, height])

  ## Apply Canny edge detector
  edges = cv2.Canny(img, 100, 200)

  ## Apply Hough transform to detect the edges
  lines = cv2.HoughLines(edges, 1, np.pi/180, 50) # (_ x 1 x 2)
  lines = np.reshape(lines, (lines.shape[0], lines.shape[2])) # (_ x 2)

  h_lines = []
  v_lines = []

  for line in lines:
    rho, theta = line[0], line[1]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    # if(display): cv2.line(color_img,(x1,y1),(x2,y2),(0,255,25),2)
    if np.abs(np.cos(theta)) > np.abs(np.sin(theta)):
      v_lines.append([x1, y1, x2, y2])
    else:
      h_lines.append([x1, y1, x2, y2])

  h_lines = order_lines(np.array(h_lines), type = 'h')
  v_lines = order_lines(np.array(v_lines), type = 'v')

  # print 'NO. of hlines: ', len(h_lines)
  # print 'NO. of vlines: ', len(v_lines)

  for rep_idx in range(20):
    if(rep_idx == 0):
      hidx = len(h_lines)/2
      vidx = len(v_lines)/2
    else:
      hidx = np.random.randint(0, len(h_lines))
      vidx = np.random.randint(0, len(v_lines))
    ## Intersect first vertical line with all horizontal lines
    h_points = np.array([intersect_lines(line, v_lines[vidx, :]) for line in h_lines])
    ## Intersect first horizontal line with all vertical lines
```

```
    v_points = np.array([intersect_lines(line, h_lines[hidx, :]) for line in v_lines])

    flags, _ = filter(h_points[:, 1], thresh = 10.0)
    h_points = h_points[flags, :]
    h_lines = h_lines[flags, :]
    flags = filter_white_points(color_img, h_points, debug = False)
    new_h_lines = h_lines[flags, :]
    new_h_lines = order_lines(np.array(new_h_lines), type = 'h')
    h_lines = new_h_lines

    flags, _ = filter(v_points[:, 0], thresh = 10.0)
    v_points = v_points[flags, :]
    v_lines = v_lines[flags, :]
    flags = filter_white_points(color_img, v_points, debug = False)
    new_v_lines = v_lines[flags, :]
    new_v_lines = order_lines(np.array(new_v_lines), type = 'v')
    v_lines = new_v_lines

    if(len(h_lines) == num_horz_lines and len(v_lines) == num_vert_lines):
      break
    else:
      print 'Cleaning: ',
      print len(h_lines), len(v_lines)

  if(display):
    img = plot_lines(color_img, new_h_lines, color = (0, 255, 25))
    img = plot_lines(img, new_v_lines, color = (255, 0, 0))
    fname = os.path.basename(img_path)
    fname = os.path.splitext(fname)[0] + '_' + os.path.splitext(fname)[1]
    print 'Writing to: ', os.path.join(out_dir, fname)
    cv2.imwrite(os.path.join(out_dir, fname), img)

  ## Obtain final list of points in raster scan order.
  final_points = []
  for hidx, hline in enumerate(h_lines):
    for vidx, vline in enumerate(v_lines):
      final_points.append(intersect_lines(hline, vline))
  final_points = np.array(final_points)

  world_points = get_world_coordinates(pattern_size, unit_size)

  assert final_points.shape[0] == world_points.shape[0], \
      'Error! No. of image and world points should be same '

  return final_points, world_points

def get_pts(shap, corners = False, H=None, targ_shap = None):
  #####
  #
  # Description:
  #
  # Input:
  #   shap: tuple (num_rows, num_cols, optional) - for given image
```

```
    #   corners: if True, only corner points, if False, all points
    #   H: 3 x 3 ndarray - given image to target image
    #   targ_shap: tuple (num_rows, num_cols, optional) - for target image
    #
    #   if H is not None, apply the homography
    #   if targ_shap is not None, clip the transformed pts accordingly.
    #
    # Return:
    #   trasformed points. In (x, y) format. It depends on if H, targ_shap are None
    #
    #####

    M, N = shap[0], shap[1]
    if(corners):
      pts = np.array([[0, 0],[N-1, 0],[N-1, M-1], [0, M-1]])
    else:
      xv, yv = np.meshgrid(range(N), range(M))
      pts = np.array([xv.flatten(), yv.flatten()]).T
    if H is None: return pts, None

    # else
    t_pts = np.dot(H, real_to_homo(pts).T)
    t_pts = homo_to_real(t_pts.T).astype(int)
    if(targ_shap is None): return pts, t_pts

    #else
    t_pts[:,0] = np.clip(t_pts[:,0], 0, targ_shap[1]-1)
    t_pts[:,1] = np.clip(t_pts[:,1], 0, targ_shap[0]-1)
    return pts, t_pts

  def find_homography_2d(pts1, pts2):
    # H: 2 --> 1

    # Assertion
    assert pts1.shape[1] == 2, 'pts1 should have two columns'
    assert pts2.shape[1] == 2, 'pts2 should have two columns'
    assert pts1.shape[0] == pts2.shape[0], 'pts1 and pts2 should have same number of rows'

    # Forming the matrix A (8 x 9)
    A = []
    for (x1, y1), (x2, y2) in zip(pts1, pts2):
      A.append([x2, y2, 1, 0, 0, 0, -1*x1*x2, -1*x1*y2, -1*x1])
      A.append([0, 0, 0, x2, y2, 1, -1*y1*x2, -1*y1*y2, -1*y1])
    A = np.array(A)

    [U, S, V] = np.linalg.svd(A, full_matrices = True)
    h = V.T[:,-1]
    h = h / h[-1]

    # # Finding the homography. H[3,3] is assumed 1.
    # h = np.dot(np.linalg.pinv(A[:,:-1]), -1*A[:,-1])
    # h = np.append(h, 1)
```

```
    H = np.reshape(h, (3, 3))
    return H, hinv(H)

hvars = ['h11', 'h12', 'h13', 'h21', 'h22', 'h23', 'h31', 'h32']
Nx = '(h11*{0}+h12*{1}+h13)'
Ny = '(h21*{0}+h22*{1}+h23)'
D = '(h31*{0}+h32*{1}+1)'

def senc(value): return '('+str(value)+')'

def fvec_row(x, y, axis = 'x'):
    if(axis == 'x'):
        fvec = Nx + '/' + D
    else:
        fvec = Ny + '/' + D
    return fvec.format(senc(x), senc(y))

def jac_row(x, y, axis = 'x'):
    d = [0]*8
    if(axis == 'x'):
        d[0] = '{0}'+'/'+D
        d[1] = '{1}'+'/'+D
        d[2] = '1'+'/'+D
        d[3] = '0'
        d[4] = '0'
        d[5] = '0'
        d[6] = '(-'+Nx+'*'+'{0})/' + '(' + D + '**2)'
        d[7] = '(-'+Nx+'*'+'{1})/' + '(' + D + '**2)'
        # d[8] = '(-'+Nx+'*'+'1)/' + '(' + D + '**2)'
    else:
        d[0] = '0'
        d[1] = '0'
        d[2] = '0'
        d[3] = '{0}'+'/'+D
        d[4] = '{1}'+'/'+D
        d[5] = '1'+'/'+D
        d[6] = '(-'+Ny+'*'+'{0})/' + '(' + D + '**2)'
        d[7] = '(-'+Ny+'*'+'{1})/' + '(' + D + '**2)'
        # d[8] = '(-'+Ny+'*'+'1)/' + '(' + D + '**2)'

    for idx, _ in enumerate(d):
        d[idx] = d[idx].format(senc(x), senc(y))

    return d

def LM_Minimizer(point_corresps, H_init, max_iter = 200, \
            delta_for_jacobian = 0.000001, \
            delta_for_step_size = 0.0001, debug = False):
    '''
        H: 2 --> 1
    '''
    nls = NLS(max_iterations = max_iter, \
            delta_for_jacobian = delta_for_jacobian, \
```

```
            delta_for_step_size = delta_for_step_size, debug = debug)

    pts1 = point_corresps[:,:2]
    pts2 = point_corresps[:,2:]

    X = pts1.flatten().reshape(-1, 1) # [x1, y1, x2, y2, ...]

    Jac = []
    Fvec = []
    for x, y in pts2:
      fx = fvec_row(x, y, 'x')
      fy = fvec_row(x, y, 'y')
      dfx = jac_row(x, y, 'x')
      dfy = jac_row(x, y, 'y')
      Jac.append(dfx)
      Jac.append(dfy)
      Fvec.append(fx)
      Fvec.append(fy)

    Fvec = np.array(Fvec).reshape(-1, 1)
    Jac = np.array(Jac)

    nls.set_Fvec(Fvec)
    nls.set_X(X)
    nls.set_jacobian_functionals_array(Jac)
    nls.set_params_ordered_list(hvars)
    nls.set_initial_params(dict(zip(hvars, H_init.flatten().tolist())))

    # print Jac
    # print ''
    # print Fvec

    return nls.leven_marq()

def apply_trans_patch(base_img_path, template_img_path, H, suff = '_fnew'):
    ## Read images
    if(isinstance(base_img_path, str)): base_img = cv2.imread(base_img_path)
    else: base_img = np.copy(base_img_path)

    if(isinstance(template_img_path, str)): temp_img = cv2.imread(template_img_path)
    else: temp_img = np.copy(template_img_path)

    ## Find corners in base that correspond to corners in template
    temp_cpts, trans_temp_cpts = get_pts(temp_img.shape, corners=True, H=H, targ_shap = base_img.s

    _cpts = real_to_homo(trans_temp_cpts) # homo. representation
    _cent_cpts = np.mean(_cpts, axis = 0)# centroid of four points

    # Find the four lines of quadrilateral
    lines = [np.cross(_cpts[0], _cpts[1]), np.cross(_cpts[1], _cpts[2]), np.cross(_cpts[2], _cpts[

    ## Finding points in the base that are present in the quadrilateral
    base_bool = np.zeros(base_img.shape[:-1]).flatten() == 0 # True -> inside the quadrilateral
```

```
  base_all_pts, _ = get_pts(base_img.shape) # get all pts
  for line in lines:
    line = line / line[-1]
    sn = int(np.sign(np.dot(_cent_cpts, line)))
    nsn = np.int8(np.sign(np.dot(real_to_homo(base_all_pts), line)))
    base_bool = np.logical_and(base_bool, nsn==sn)
  base_bool = base_bool
  base_bool = np.reshape(base_bool, (base_img.shape[0], base_img.shape[1]))
  row_ids, col_ids = np.nonzero(base_bool)
  des_base_pts = np.array([col_ids, row_ids])

  # Find corresponding points in the template image
  trans_des_base_pts = homo_to_real(np.dot(hinv(H), real_to_homo(des_base_pts.T).T).T).astype(in

  # Clip the points
  trans_des_base_pts[:, 0] = np.clip(trans_des_base_pts[:, 0], 0, temp_img.shape[1]-1)
  trans_des_base_pts[:, 1] = np.clip(trans_des_base_pts[:, 1], 0, temp_img.shape[0]-1)

  base_img[des_base_pts[1].tolist(), des_base_pts[0].tolist(), :] = temp_img[trans_des_base_pts[

  # Write the resulting image to a file
  fname, ext = tuple(os.path.basename(base_img_path).split('.'))
  write_filepath = os.path.join(os.path.dirname(base_img_path), fname+suff+'.'+ext)
  print write_filepath
  cv2.imwrite(write_filepath, base_img)


def dist_mat_vec(M, vec):
  # Compute distance between each row of 'M' with 'vec'
  # method: 'ncc', 'dot', 'ssd'
  # M : ndarray ( _ x k); vec: (1 x k)
  # Returns a 1D numpy array of distances.
  return np.linalg.norm(M - vec, axis = 1)

def dist_mat_mat(M1, M2):
  # M1, M2 --> ndarray (y1 x k) and (y2 x k)
  # Returns y1 x y2 ndarray with the distances.
  # If y1 and y2 are huge, it might run into MemoryError
  D = np.zeros((M1.shape[0], M2.shape[0]))
  for idx2 in range(M2.shape[0]):
    D[:, idx2] = dist_mat_vec(M1, M2[idx2, :])
  return D

def real_to_homo(pts):
  # pts is a 2D numpy array of size _ x 2/3
  # This function converts it into _ x 3/4 by appending 1
  if(pts.ndim == 1):
    return np.append(pts, 1)
  else:
    return np.concatenate((pts, np.ones((pts.shape[0], 1))), axis = 1)

def homo_to_real(pts):
  # pts is a 2D numpy array of size _ x 3/4
```

```
  # This function converts it into _ x 2/3 by removing last column
  if(pts.ndim == 1):
    pts = pts / pts[-1]
    return pts[:-1]
  else:
    pts = pts.T
    pts = pts / pts[-1,:]
    return pts[:-1,:].T

def hinv(H):
  assert isinstance(H, np.ndarray), 'H should be a numpy array'
  assert H.ndim == 2, 'H should be a numpy array of two dim'
  assert H.shape[0] == H.shape[1], 'H should be a square matrix'
  Hinv = np.linalg.inv(H)
  return Hinv / Hinv[-1,-1]

def nmlz(x):
  assert isinstance(x, np.ndarray), 'x should be a numpy array'
  assert x.ndim > 0 and x.ndim < 3, 'dim of x >0 and <3'
  if(x.ndim == 1 and x[-1]!=0): return x/float(x[-1])
  if(x.ndim == 2 and x[-1,-1]!=0): return x/float(x[-1,-1])
  return x

##############################
########### MAIN #############
##############################

NUM_HORZ_LINES = 10
NUM_VERT_LINES = 8
SQUARE_SZ = 25

dataset = 'dataset2'
base_dir = os.path.join('.\\Data', dataset)
out_dir = os.path.join('.\\Data\\results', dataset)

img_paths = glob.glob(os.path.join(base_dir, '*.jpg'))

if os.path.isfile('homographies_'+os.path.basename(base_dir)+'.pkl'):
  with open('homographies_'+os.path.basename(base_dir)+'.pkl', 'rb') as fp:
    temp = pickle.load(fp)
  homographies, img_points_list, world_points_list = zip(*temp['homographies'])
else:
  homographies = []
  print 'Processing'
  for img_path in img_paths:
    print img_path
    img_points, world_points = find_checkerboard_points(img_path, \
      (NUM_HORZ_LINES-1, NUM_VERT_LINES-1), unit_size = SQUARE_SZ,\
      display = True, out_dir = out_dir)
    ## Find homography from 2 --> 1
    H, Hinv = find_homography_2d(img_points, world_points)
    ## Uncomment to apply the transformed patch on the original image.
    # template_img = 255 * np.ones((SQUARE_SZ*(NUM_HORZ_LINES-1), SQUARE_SZ*(NUM_VERT_LINES-1),
```

```
    # apply_trans_patch(img_path, template_img, H, suff = '_fnew')

    lmres = LM_Minimizer(np.concatenate([img_points, world_points], axis = 1), H)

    new_H = np.squeeze(np.asarray(lmres['parameter_values']))
    new_H = np.append(new_H, np.array([1])).reshape(3, 3)
    new_H = nmlz(new_H)

    # pred_img_points = homo_to_real(np.dot(new_H, real_to_homo(world_points).T).T)
    # print np.append(img_points, pred_img_points, axis = 1)
    # sys.exit()

    homographies.append((new_H, img_points, world_points))

  with open('homographies_'+os.path.basename(base_dir)+'.pkl', 'wb') as fp:
    pickle.dump({'homographies': homographies}, fp)
  homographies, img_points_list, world_points_list = zip(*homographies)

V = []
for H in homographies:
  V12 = get_v_rep(H, 0, 1)
  V11 = get_v_rep(H, 0, 0)
  V22 = get_v_rep(H, 1, 1)
  V.append(V12)
  V.append(V11 - V22)
V = np.array(V)

[U, S, E] = np.linalg.svd(V, full_matrices = True)
b = E.T[:,-1]

# ww = b[0]*b[2]*b[5] - (b[1]**2)*b[5] - b[0]*(b[4]**2) + 2*b[1]*b[3]*b[4] - b[2]*(b[3]**2)
# dd = b[0]*b[2] - b[1]**2
# alph = np.sqrt(ww/(dd*b[0])) # alpha_x
# bet = np.sqrt(b[0]*ww/(dd**2)) # alpha_y
# gam = -1 * b[1] * np.sqrt(ww/((dd**2)*b[0])) # s # I added negative sign.
# uc = (b[1]*b[4] - b[2]*b[3]) / dd # x0
# vc = (b[1]*b[3] - b[0]*b[4]) / dd # y0

# K = np.array([[alph, gam, uc],
#       [0, bet, vc],
#       [0, 0, 1]])

W11, W12, W22, W13, W23, W33 = b[0], b[1], b[2], b[3], b[4], b[5]
# Estimate intrinstic parameters
y0 = (W12*W13 - W11*W23)/(W11*W22 - (W12**2))
lambd = W33 - (W13**2 + y0*(W12*W13 - W11*W23))/W11
alpha_x = np.sqrt(lambd/W11)
alpha_y = np.sqrt(lambd*W11/(W11*W22 - W12**2))
s = -((W12*(alpha_x**2)*alpha_y)/lambd)
x0 = s*y0/alpha_y - W13*(alpha_x**2)/lambd

# define K matrix
K = np.array([[alpha_x, s, x0],
```

```
      [0, alpha_y, y0],
      [0, 0, 1]], dtype=np.float64)

Kinv = np.linalg.inv(K)
print 'K'
print K

for hidx, H in enumerate(homographies):
  img_path = img_paths[hidx]
  # H = [h1, h2, h3]
  H = homographies[hidx]
  print 'H'
  print H
  img_points = img_points_list[hidx]
  world_points = world_points_list[hidx]
  h1 = H[:, 0]
  h2 = H[:, 1]
  h3 = H[:, 2]
  t = np.dot(Kinv, h3)
  zeta = 1.0 / np.linalg.norm(np.dot(Kinv, h1))
  if(t[2] < 0): zeta = -1 * zeta
  # print zeta

  R = np.zeros((3, 3))
  Z = np.zeros((3, 4))
  r1 = zeta * np.dot(Kinv, h1)
  r2 = zeta * np.dot(Kinv, h2)
  r3 = np.cross(r1, r2)
  t = zeta * t
  R[:, 0] = r1
  R[:, 1] = r2
  R[:, 2] = r3

  [U, S, E] = np.linalg.svd(R, full_matrices = True)
  R = np.dot(U, E)

  Z[0:3,0:3] = R
  Z[:,-1] = t

  # print Z

  image = np.copy(cv2.imread(img_path))

  ## Compute the error
  C_mat = np.dot(K, Z)
  for row in img_points.astype(int):
    cv2.circle(image, tuple(row.tolist()), 3, color = [250, 0, 0], thickness = 1)
  world_points = np.concatenate([world_points, np.zeros((world_points.shape[0], 1))], axis = 1)
  world_points = np.concatenate([world_points, np.ones((world_points.shape[0], 1))], axis = 1)
  pred_img_points = homo_to_real(np.dot(C_mat, world_points.T).T)
  for row in pred_img_points.astype('int'):
    cv2.circle(image, tuple(row.tolist()), 3, color = [0, 25, 255], thickness = 1)
  # print np.append(world_points, np.append(img_points, pred_img_points, axis = 1), axis = 1).as
```

```
out_path = os.path.join(out_dir, os.path.basename(img_path))
print out_path
cv2.imwrite(out_path, image)
```