ECE661: Homework 9

Rahul Deshmukh
deshmuk5@purdue.edu
PUID: 0030004932

November 22, 2018

In this homework we are required ti carry out projective reconstruction of a scene using single pair of stereo images. We firs pick point correspondences manually ans then compute the fundamental matrix using the 8-point algorithm (ie linear least squares). Using the estimate of F we construct the two projection Matrices P and P'.

After obtaining the Linear estimate, we refine our estimate using non-linear least squares (using Gold-standard algorithm). After obtaining the refined estimates we then proceed to rectify the images to build a larger set of correspondences. Using row-scans on the rectified images with we build point pairs (using descriptor vector of SIFT). Finally, we triangulate the final set of pairs to get world points.

1 Logic and Methodology

The task of this homework can be further divided into the following sub-tasks:

a) Linear estimate of F (using 8-point algorithm)

b) Obtaining Projection Matrix

c) Triangulation

d) Refinement using LM

e) Image rectification and row-scans

f) Projective Reconstruction

2 Linear estimate of F (using 8-point algorithm)

After picking manual correspondences (x, x'), we are now in a position to estimate the fundamental Matrix using the epipolar constraint:

$$x'^{T} F x = 0$$

We can break this equation for one pair as:

$$Af = \begin{bmatrix} x'x & x'y & x' & y'x & y'y & y' & x & y & 1 \end{bmatrix} f = 0$$

where $f = [f_{11}, f_{12}, f_{13}, f_{21}...., f_{33}]^T$. We can solve this system of equations using $n \geq 8$ point pairs using SVD.

Note: In order to get better estimate for F we first Normalize the points using a transformations T and T' for the two images and then find the linear estimate with these normalized coordinates using the procedure discussed above. After obtaining the linear estimate of F we then de-normalize it to get F in the original coordinate frame. Where the transformation T of each image is such that it shifts the image origin to the centroid of our correspondences and scales the mean RMS distance of all correspondences from the centroid to a value of 1.

After obtaining the Linear estimate of F we still need to enforce the condition that det(F) = 0. We do this by carrying out SVD of F and then setting the smallest singular value to 0.

The steps for the whole procedure is:

i) Normalize (x, x') : $\hat{x} = Tx$ and $\hat{x'} = T'x'$

ii) Find Linear estimate $\hat{F}$ using $A\hat{f} = 0$

iii) Enforce condition: det(F) = 0

iv) Denormalize $\hat{F}$: F = $T^T \hat{F} T'$

3 Obtaining Projection Matrix

The next step after finding F is to estimate the Projection matrices from F. For this task we first need to estimate the left null vector(e') of F (which is the right image epipole e'). We then set the projetion matrices as:

$$P = [I_{3x3}|\vec{0}]$$
$$P' = [[e']_X F|e']$$

where $[e']_X$ is the cross-representation matrix of e'.

4 Triangulation

Now, that we have the camera projection matrices we can find the world points using the point correspondences and the projection matrices as follows:
Let us write the projection matrix as

$$P = \begin{bmatrix} P_1^T \\ P_2^T \\ P_3^T \end{bmatrix}$$

where P_i^T would be the i^{th} row of the Projection matrix. We then estimate the world point (X) using the following equations:

$$AX = 0$$

where A is as follows:

$$A = \begin{bmatrix} xP_3^T - P_1^T \\ yP_3^T - P_2^T \\ x'P_3'^T - P_1'^T \\ y'P_3'^T - P_2'^T \end{bmatrix}$$

such that the two corresponding points in the two images were (x, y) and (x', y').

5 Refinement using LM

Now, that we have linear estimates of F,P,P' we can now carry out their refinement using non-linear least squares(LM). I am using the **Gold-standard** method for this refinement. In this method our parameters for optimization will be the the 3 physical coordinates of all of the N world points and 12 dof of P'. So in total we have $3n + 12$ parameters. In my code I am setting my parameter vector p as :

$$\begin{aligned} p &= [M, t, X_1, X_2..., X_n] \\ M &= [M_{11}, M_{12}, M_{13}, M_{21}, ..., M_{33}] \\ t &= [t_x, t_y, t_z] \\ X_i &= [x_i, y_i, z_i] \end{aligned}$$

where M and t come from P' such that $P' = [M|t]$

We then set-up a cost function which gives us the geometric distance as follows:

$$d^2_{geom} = \Sigma_i ||x_i - \hat{x}_i||^2 + ||x'_i - \hat{x'}_i||^2$$

where $\hat{x}$ are the the estimates of the points found using the following steps:

i) Build world points (X) from the parameter vector

ii) Project the world points using P and P' (which again needs to be built using the parameters corresponding to M and t) to get $\hat{x} = PX$ and $\hat{x'} = P'X$.

6 Image rectification and row-scans

After Refinement we have the correct estimates of the fundamental Matrix and projection matrices. We can now rectify the images to get larger number of correspondences. To rectify the images the procedure is as follows:

6.1 Rectifying the Right Image

We first rectify the Right image using the following steps:

1. Compute the Translation Matrix which sends the image center to origin.

$$T = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

   where (x_0, y_0) is the image center location.

2. Compute the Rotation Matrix which rotates the translated right epipole to the location $[f, 0, 1]^T$.

$$R \begin{bmatrix} (e'_x - x_0) \\ (e'_y - y_0) \\ 1 \end{bmatrix} = \begin{bmatrix} f \\ 0 \\ 1 \end{bmatrix}$$

Also, The rotation matrix(in-plane rot) is given by:

$$R = \begin{bmatrix} cos(t) & -sin(t) & 0 \\ sin(t) & cos(t) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Which gives:

$$(e_x - x_0)cos(t) - (e_y - y_0)sin(t) = f$$
$$(e_x - x_0)sin(t) + (e_y - y_0)cos(t) = 0$$
$$=> tan(t) = -(e_y - y_0)/(e_x - x_0)$$

and then f can be found

3. Compute the G matrix which would send the epipole to infinity. That is from $[f, 0, 1]^T$ to $[f, 0, 0]^T$.

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ (-1/f) & 0 & 1 \end{bmatrix}$$

4. Compute the Translation Matrix which makes the image upper left corner as origin.

$$T_2 = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $(x0, y0)$ is the image center location.

We then set the rectification homography for the second image as

$$H' = T_2GRT$$

6.2 Rectifying the Left Image

For the Left image we need to find a rectification homography which sends the left epipole to infinity and as well as scale the left image in such a way that after rectification, the corresponding points come to the same row as the right image.The steps involved are as follows:

1. Find $M = P'P^+$ where P^+ is the right Psuedo inverse of P.
2. Set H_0 as $H_0 = H'M$
3. Transform the points (x, x') using the homographies H_0 and H' such that:

$$\hat{x} = H_0 x$$
$$\hat{x'} = H' x'$$

4. Then using the transformed coordinates we setup a distance minimization problem to address the issue of scaling the left image to match up to right image as follows:
We need to find H_a of the form:

$$H_a = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the minimization problem which finds a,b,c is:

$$min_{(a,b,c)}\Sigma_i(a\hat{x}_i + b\hat{y}_i + c - \hat{x'}_i)^2$$

The above minimization problem can be reduced to a linear least squares formulation of the form:

$$min_x||Ax - b||^2$$

Where

$$A = \begin{bmatrix} \hat{x}_1 & \hat{y}_1 & 1 \\ \hat{x}_2 & \hat{y}_2 & 1 \\ .. & .. & .. \end{bmatrix}$$

$$b = \begin{bmatrix} \hat{x'}_1 \\ \hat{x'}_2 \\ .. \end{bmatrix}$$

And we can solve for $x = [a, b, c]^T$ using the left pseudo inverse of A.

5. We then set the rectification Homography as $H = H_a H_0$

After Rectification of the images we are now in a position to build a larger number of correspondences using row-scans. The procedure is as follows:

1. Find SIFT Interest points on the original image.
2. Find the transformed location of the interest points on the rectified image using rectification homographies H and H'.
3. For every interest point in the left image(x_i) look for a corresponding point in the window of row number of $Hx_i \pm$ scan tolerance(user defined).
4. for all the points in this window find the best match based on the descriptor vector(of the original image not the rectified image).
5. store the matched pair coordinates (original coordinates and not the transformed coordinates).
6. Keep a record of the used pair so that its not used in the next iteration.

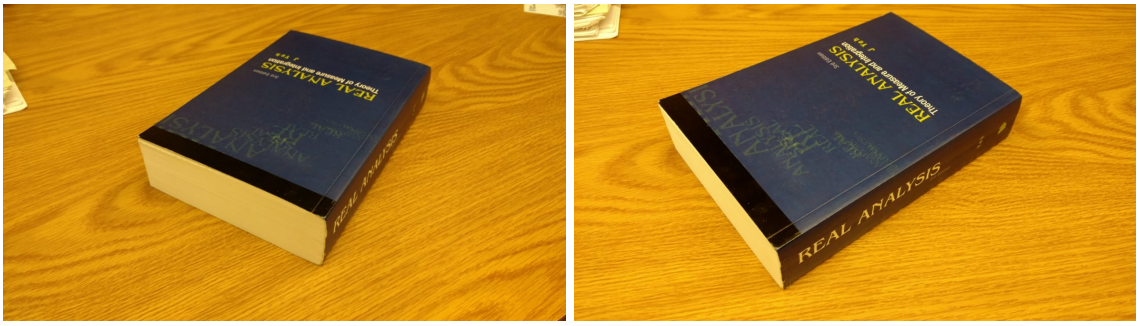
7 Projective Reconstruction

Now we have to Triangulate the whole list of matched pairs to get corresponding world points using the method explained in section 4. The 3D Reconstruction will be upto projective Homography.

8 Observations

As we have picked the Points manually, The initial estimate of F using linear least squares gives a good estimate and LM essentially does very small refinement.

9 Results



(a) Left (b) Right

Figure 1: Original images

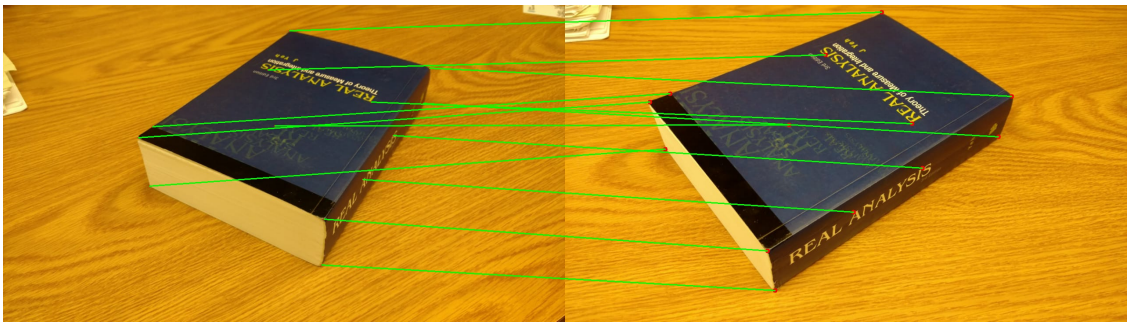


Figure 2: Manual Correspondences

Figure 3: Rectified Images

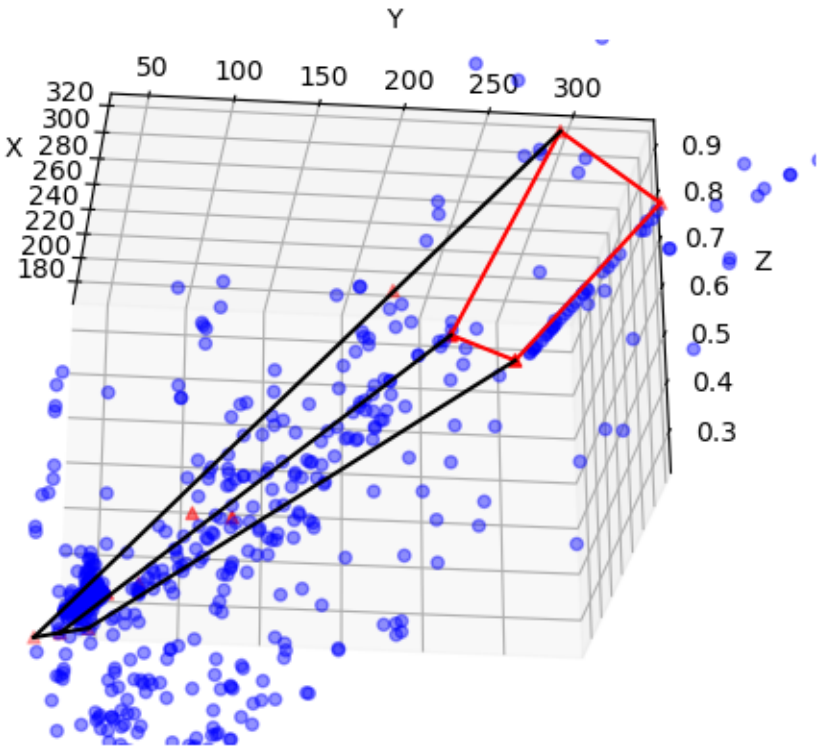


Figure 4: Projective Reconstruction

10 Source Code:

10.1 main file

```
"""
ECE661: hw9 main file
@author: Rahul Deshmukh
email: deshmuk5@purdue.edu
PUID: 0030004932
"""
#%% ---------------------import libraries ---------------------------------------#
import cv2
import numpy as np
import sys
sys.path.append('../../')
import MyCVModule as MyCV
#%% ---------------------define readpath-----------------------------------------#
readpath='../files/original/'
savepath='../files/'
#read images
img1= cv2.imread(readpath+'1 (1).jpeg')
img2= cv2.imread(readpath+'1 (2).jpeg')
img1_gray= cv2.imread(readpath+'1 (1).jpeg',0)
img2_gray= cv2.imread(readpath+'1 (2).jpeg',0)

#%%------------------------- define coordinates ------------------------------#
pts1=[
[731,   486], #1
[725,590], #2
[308,301], #3
[333,411], #4
[971,143], #5
[652,58], #6
[838,220],
[819,396],
[603,147],
[340,276],
[888,296],
[619,274],
[957,223]]#7

pts2=[
[461,   558],
[478,   648],
[193,   220],
[228,   327],
[1017,207],
[721,16],
[790,269],
[657,470],
[593,113],
[239,201],
[814,370],
[510,272],
[987,299]]
#-------------------------------------RANSAC-----------------------------------#
## find pts using SIFT on both the images
## find sift points of images
#pts1,des1=MyCV.SIFT_detector(img1_gray)
#pts2,des2=MyCV.SIFT_detector(img2_gray)
## find euclidean matches between images
#match_pts1,match_pts2=MyCV.Euclidean_sift_match(pts1,des1,pts2,des2,3) # returns list of
    lists
#in_indices= MyCV.F_Ransac(match_pts1,match_pts2)
```

```
#pts1 = np.array(pts1)
#pts2 = np.array(pts2)
#pts1 = pts1[in_indices,:]
#pts2 = pts2[in_indices,:]
#%%------plot images with points and lines connecting correspondences----------#
pt_pairs=[]
for i in range(len(pts1)): pt_pairs.append( [tuple(pts1[i]),tuple(pts2[i])] )
img= MyCV.plot_matching_points(img1,pts1,img2,pts2,pt_pairs)
cv2.imwrite(savepath+'plots/'+'correspondences'+'.jpg',img)

#%%------------------- find F using Linear Least Squares-------------------------#
# convert points to HC
pts1_hc = MyCV.convert2HC(pts1)
pts2_hc = MyCV.convert2HC(pts2)
F0 = MyCV.Funda_using_pts(pts1_hc,pts2_hc)
P1,P2 =MyCV.get_Proj_mat(F0)
# refine F using LM
F_ref,P1_ref,P2_ref,M_ref=MyCV.Refine_F(F0,pts1,pts2)
e2 = MyCV.get_left_nullvec(F_ref)
#%%-----------------------------------Image Rectification----------------------------------#
H1,H2 = MyCV.get_rectify_homo(e2,P1_ref,P2_ref,M_ref,img1,img2,pts1,pts2)

#plot of correspondences in rectified images
pts1_rec_hc = H1@pts1_hc.T
pts2_rec_hc = H2@pts2_hc.T
pts1_rec = MyCV.convert2phy(pts1_rec_hc)
pts2_rec = MyCV.convert2phy(pts2_rec_hc)

img=MyCV.plot_rectify_img(img1,img2,H1,H2)
cv2.imwrite(savepath+'plots/'+'correspondences_Rec'+'.jpg',img)
#%%-----------------------Interest Point Detection-----------------------------------#
pts1_sift,des1=MyCV.SIFT_detector(img1_gray)
pts2_sift,des2=MyCV.SIFT_detector(img2_gray)


# now we need to do row scans on the rectified images to make pairs with
# descriptor vectors as my metric

# find pt pairs using row-scans in the rectified images
scan_tol = 2 # pm row scan window
pt_pairs = MyCV.find_pairs_rectifed_img(img1,pts1_sift,des1,img2,pts2_sift,des2,H1,H2,
    scan_tol)

#%%---------------------------Projective Reconstruction-------------------------#
# separate pt_pairs for each image
pt_pair1 = [] # for pair poin on image1
pt_pair2 = [] # for pair poin on image12
for i in range(len(pt_pairs)):
    pt_pair1.append(pt_pairs[i][0])
    pt_pair2.append(pt_pairs[i][1])

# convert pt_pairs to HC for triangulation
pt_pair1_hc = MyCV.convert2HC(pt_pair1)
pt_pair2_hc = MyCV.convert2HC(pt_pair2)
# Triangulate the pt_pairs to world pts for projective reconstruction
world_pt_hc =[]
for i in range(len(pt_pairs)):
    world_pt_hc.append(MyCV.Triangulate(pt_pair1_hc[i,:],pt_pair2_hc[i,:],P1_ref,P2_ref))

world_pt_hc = np.array(world_pt_hc)
world_pt = MyCV.convert2phy(world_pt_hc.T)
#%%------------------------------ 3D Visual Inspection----------------------------#
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
fig = plt.figure(1)
ax = fig.add_subplot(111, projection='3d')
#plots 3D points from sift in blue
ax.scatter(world_pt[:,0], world_pt[:,1], world_pt[:,2], c='b', marker='o')
# plot Triangulation of original inliers in red
inlier_3D_hc = []
for i in range(len(pts1)):
    inlier_3D_hc.append(MyCV.Triangulate(pts1_hc[i,:],pts2_hc[i,:],P1_ref,P2_ref))
inlier_3D_hc = np.array(inlier_3D_hc)
inlier_3D = MyCV.convert2phy(inlier_3D_hc.T)
ax.scatter(inlier_3D[:,0], inlier_3D[:,1], inlier_3D[:,2], c='r', marker='^')
# draw line joininG the vertices which were picked manually
# LINES: 12,24,43,31,15,56,63,57,72
lookup = np.array([[1,2,3,4,5,6, 7],
                   [0,1,2,3,4,5,-1]])
#edges = np.array([[1,2],[2,4],[4,3],[3,1],[1,5],[5,6],[6,3],[5,7],[7,2]])
face1= np.array([[1,2],[2,4],[4,3],[3,1]])
for i in range(np.shape(face1)[0]):
    ip1x,ip1y,ip1z = inlier_3D[lookup[1,(face1[i,0])-1],:]
    ip2x,ip2y,ip2z = inlier_3D[lookup[1,(face1[i,1])-1],:]
    ax.plot([ip1x,ip2x],[ip1y,ip2y],[ip1z,ip2z],c='r')

face2=np.array([[1,5],[5,6],[6,3],[5,7],[7,2]])
for i in range(np.shape(face2)[0]):
    ip1x,ip1y,ip1z = inlier_3D[lookup[1,(face2[i,0])-1],:]
    ip2x,ip2y,ip2z = inlier_3D[lookup[1,(face2[i,1])-1],:]
    ax.plot([ip1x,ip2x],[ip1y,ip2y],[ip1z,ip2z],c='black')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.xlim(min(inlier_3D[:,0]),max(inlier_3D[:,0]))
plt.ylim(min(inlier_3D[:,1]),max(inlier_3D[:,1]))
ax.set_zlim(min(inlier_3D[:,2]),max(inlier_3D[:,2]))
plt.show()

# write a ply file for visualization in openmesh
all_3D = np.vstack((world_pt,inlier_3D))
MyCV.write_ply_file(all_3D,savepath+'Point_cloud')
```

10.2 Functions

```
#%%
#============================ V Imp functions ================================#
# most called functions
#Take note of the format of input/output while using them

#function to convert the Physical coordinates into HC coordinates
def convert2HC(pts):
    """
    Input: pts: list of list/np.array of coordinate [[x,y],...]
    Output: pts_hc: HC coordinates of pts [[x,y,1],....] np.array
    """
    pts_hc = np.ones((np.shape(pts)[0],np.shape(pts)[1]+1))
    pts_hc[:,:-1]=pts
    return(pts_hc)
#function for converting the HC coordinates to Physical coordinates
def convert2phy(pts_hc):
    """
    Input: pts_hc: HC coordinates format: np.array !!! format: [[x,y,1]^T,...] !!!
    Output: pts: array of pts format: [[x,y],..]
    """
    pts = np.linalg.inv(np.diag(pts_hc[-1,:]))@pts_hc.T
    pts= pts[:,:-1]
    return(pts)
```

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

#%%
#=============================HW9=============================================#
#%%
#function for writing a ASCII PLY file for the 3D points for visualization
def write_ply_file(pts,filename):
    """
    Input: pts: physical cooordinates of 3D pt np.array[[X,Y,Z],..]
           filename: full filename wo file extension
    Ouput: asc file will be printed with the coordinates
    """
    file = open(filename+'.asc','w')
    # write in asc format
    file.write('ply\n')
    file.write('format ascii 1.0\n')
    file.write('comment Rahul Deshmukh generated\n')
    file.write('element vertex '+str(np.shape(pts)[0])+'\n')
    file.write('property float x\n')
    file.write('property float y\n')
    file.write('property float z\n')
    file.write('element face 0\n')
    file.write('property list uchar int vertex_indices\n')
    file.write('end_header\n')
    for i in range(np.shape(pts)[0]):
        file.write(str(pts[i,0])+'\t'+str(pts[i,1])+'\t'+str(pts[i,2])+'\n')
    return()
#%% ---------------------Interest Pt Pairs on Rectfied Images---------------%
# pts detected using sift and mathcing is done using descriptor vector
# scanning done on rectified image row by row with a user def tolerance

#function for finding pairs
def find_pairs_rectifed_img(img1,pts1,des1,img2,pts2,des2,H1,H2,scan_tol):
    """
    Input: img1,img2: image matrices mxnx3
           pts1,pts2: list of list of coordinates of N interest pts [[x,y],[x,y],..]
           des1,des2: descriptor vectors stacked as row vec in a np.array Nx128
           H1,H2: Homography for rectification of image1 and image 2
           scan_tol: row tolerance for finding pairs
    Output: pt_pairs: list of list of pt pairs [ [[x,y] , [x,y]] ,...]
    """
    # find m,n,corners of transformed second image
    tr_m2,tr_n2,cor2 = get_size_after_homo(img2,H2)
    # convert pts to HC
    pts1_hc = convert2HC(pts1) #[[x,y,1],..]
    pts2_hc = convert2HC(pts2)
    # find transformed sift pts using H1 and H2: both HC and phy
    tr_pts1_hc = H1@pts1_hc.T # hc vec as cols
    tr_pts1 = convert2phy(tr_pts1_hc)
    tr_pts2_hc = H2@pts2_hc.T
    tr_pts2 = convert2phy(tr_pts2_hc)
    # loop over pts
    pt_pairs = []
    record= np.zeros((np.shape(tr_pts2)[0],1)) # stores 1 or 0 if 1 then used
    for i in range(len(pts1)):
        ipt = tr_pts1[i,:]
        ides1 = des1[i,:]
        # find window of search for row scans in the second image
        irow1 = ipt[1]
        iwin = give_window(irow1,scan_tol,(tr_m2,tr_n2))
        # find pts in the second transfromed image which lie in  this window
        ind = pts_in_win(tr_pts2,iwin,record)
        if len(ind)>0:
            # make temp descriptor-vector packet for these indices
            temp_des = []
```

```
            for j in range(len(ind)): temp_des.append(des2[ind[j],:])
            # find euclidean distance of ides1 with temp_des
            d = np.array(temp_des)-ides1
            d = np.linalg.norm(d,axis=1)
            imin = np.argmin(d)
            i_pair = ind[imin]
            pt_pairs.append([pts1[i],pts2[i_pair]])
            #update record
            record[i_pair]=1
    return(pt_pairs)
#function for finding pts in a row window
def pts_in_win(tr_pts2,win,record):
    """
    Input: tr_pts2: Nx2 array of physical coordinates of interest pts in transformed img2
           format: [[x,y],...]
           win: size of row scan window [lb,ub]: indices of rows
           record: vector os size N, has 1 or 0
           1: element is used dont send index
    Output: ind: indices of pts in this window
    """
    lb_ind = list(np.where(tr_pts2[:,1]>win[0])[0])
    ub_ind = list(np.where(tr_pts2[:,1]<win[1])[0])
    usable = list(np.where(record<1)[0])
    ind = list(set(lb_ind)&set(ub_ind)&set(usable))
    return(ind)
# function for finding window using scan_tol
def give_window(irow1,scan_tol,img_size):
    """
    Input:  irow1: row number of current sift pt on left image
            scan_tol: user defined scan tol on row
            img_size: tuple m,n: row col of transformed image: using corners
    Output: window: [lb,ub] lower bound and upper bounds for row index
    """
    win = int(irow1)*np.ones(2,dtype=int)
    win = win +scan_tol*np.array([-1,1])
    # check if win is not outside img_size
    if win[0]<0:
        win[0]=0
    if win[1]>=img_size[1]:
        win[1]=img_size[1]-1 # row number index
    return(win)
#%% ---------------- Image Rectification ----------------------------%
# pg 302 of text section 11.12
#function for plotting image with interest points of two scenes and lines joining matching
    points
def plot_rectify_img(img1,img2,H1,H2):
    """
    Input: img1,img2: original images of same sizes
           H1,H2: homography for the two images respectively st H1@img1 = rectified img
    OutPut: img3: image matrix with both scenes placed side by side horizontally
    """
    m=np.shape(img1)[0];n=np.shape(img1)[1];#n2=np.shape(img2)[1]
    # find transformed corners of both the images to determine the size of the
    # cnavas
    tr_m1,tr_n1,cor1 = get_size_after_homo(img1,H1)
    tr_m2,tr_n2,cor2 = get_size_after_homo(img2,H2)
    # for canvas
    canvas_y = np.max([tr_m1,tr_m2])
    ind = np.argmax([tr_m1,tr_m2]) # which image had bigger size in y
    canvas = np.zeros((int(canvas_y),int(tr_n1+tr_n2),3))

    # get transformed images
    distype = 'BiLinear'
    rec_img1 = mapFitToCanvas(img1,cor1,[],np.linalg.inv(H1),distype)
    rec_img2 = mapFitToCanvas(img2,cor2,[],np.linalg.inv(H2),distype)
```

```
    # now assign image pixels to canvas
    if ind == 0:
        # left image is bigger
        # assign left part as rec_img1
        canvas[:,:int(tr_n1),:]=rec_img1
        # for right image we need to displace its y such that it comes into the midddle
        displacement = canvas_y - (tr_m2); displacement = displacement/2
        canvas[int(displacement):int(displacement)+rec_img2.shape[0],int(tr_n1):,:]=rec_img2
    else:
        # right image is bigger
        # assign right part as rec_img1
        canvas[:,int(tr_n1):,:]=rec_img2
        # for left image we need to displace its y such that it comes into the midddle
        displacement = canvas_y - (tr_m1); displacement = displacement/2
        canvas[int(displacement):int(displacement)+rec_img1.shape[0],:int(tr_n1),:]=rec_img1
    return(canvas)
# function for finding transfromed image size
def get_size_after_homo(img,H):
    """
    Input:img: image mat mxnx3
            H: Homography by which this image will transform
            ie H*img = tr_img
    Output: m,n: row, col new image size using information of corners
            cor: original corners of image
    """
    # find corners:
    cor = [[0,0],
           [img.shape[1]-1,0],
           [img.shape[1]-1,img.shape[0]-1],
           [0,img.shape[0]-1]]
    cor_hc = convert2HC(cor)
    tr_cor_hc = H@cor_hc.T
    tr_cor = convert2phy(tr_cor_hc)
    x_min = min(tr_cor[:,0]); x_max = max(tr_cor[:,0]);
    y_min = min(tr_cor[:,1]); y_max = max(tr_cor[:,1]);
    m = y_max-y_min # height of image
    n = x_max-x_min # width of image
    return(m,n,cor)
#%%
# function for finding H and H' for image rectification
def get_rectify_homo(e,P1,P2,M,img1,img2,pts1,pts2):
    """
    Input:  e: right image epipole
            M: rotation part of P2 = [M|t]
            img1,img2: image matrices
            pts1,pts2: Matching points
    Output: H1 and H2: image rectification Matrices
    """
    # convert pts to HC
    pts1_hc = convert2HC(pts1) #[[x,y,1],...]
    pts2_hc = convert2HC(pts2) #[[x,y,1],...]
    # find image sizes for second image
    m=img2.shape[0]; n=img2.shape[1];
    # get T,R,G to make H2
    T=get_T(m,n)
    T2 = get_T(-1*m,-1*n)
    R,f=get_Rot(e,m,n)
    G=get_G(f)
    H2=T2@G@R@T
    # find M=P'P+
    P1_psinv = P1.T@(np.linalg.inv(P1@P1.T)) # right psuedo inv
    M = P2@P1_psinv
    # Now find Matching H1
    H0= H2@M
    x2_hat_hc = H2@pts2_hc.T
```

```
    x1_hat_hc = H0@pts1_hc.T
    # find physical coordinates of x1_hat and x2_hat
    x2_hat = convert2phy(x2_hat_hc) #[[x,y],..]
    x1_hat = convert2phy(x1_hat_hc) #[[x,y],..]
    # Find Ha
    Ha = get_Ha(x1_hat,x2_hat)
    H1= Ha@H0
    return(H1,H2)
# function for solving Ha for first image
def get_Ha(x1_hat,x2_hat):
    """
    Input: x1_hat, x2_hat: transformed physical coordinates
    Output: Ha homography using Linear solution
    """
    A = np.ones((x1_hat.shape[0],3))
    A[:,:-1]= x1_hat
    b = x2_hat[:,0]
    # solve Ax-b = 0 using left Psuedo inverse
    a = (np.linalg.inv((A.T)@A)@A.T)@b
    Ha = np.eye(3)
    Ha[0,:]=a
    return(Ha)
#function for getting G matrix for image rectification
def get_G(f):
    """
    Input: f: x coordinate of transformed epipole
    Output: G:matrix that take epipole from (f,0,1) to (f,0,0)
    """
    G= np.array([[1    ,0,0],
                 [0    ,1,0],
                 [-1/f,0,1]])
    return(G)
# function for finding Rotation matrix for epipolar lines
def get_Rot(e,m,n):
    """
    Input: e: the epipole e_prime for right image in HC
    m,n= image size row,col

    Output: R:Rotation MAtrix  , f: new epipole location
    Finds the Rotation Matrix which takes the epipole from
    ((ex-x0),(ey-y0),1) to (f,0,1)
    ------------------------------------------------------

    given:
    R@[(ex-x0),(ey-y0),1]^T =[f,0,1]
    Also, The rotation matrix(in-plane rot) is given by:
    R=[[cos(t),-sint(t),0],
       [sin(t),cos(t),0],
       [0    , 0,       1]]
    Which gives:
        (ex-x0)cos(t)-(ey-y0)sin(t) = f   &
        (ex-x0)sin(t)+(ey-y0)cos(t) = 0
        => tan(t) = -(ey-y0)/(ex-x0)
        and then f can be found
    ------------------------------------------------------
    """
    x0=n/2; y0=m/2;
    e = e/e[-1]
    ex= e[0]; ey=e[1];
    t = np.arctan(-(ey-y0)/(ex-x0)) # theta
    R=np.array([[np.cos(t),-1*np.sin(t),    0],
                [np.sin(t),   np.cos(t),    0],
                [0         ,   0        ,    1]])
    f=(ex-x0)*np.cos(t)-(ey-y0)*np.sin(t)
    return(R,f)
# function for Translating image origin to image center
```

```
def get_T(m,n):
    """
    Input: m,n size of image as mxn
      ie center is at x0=n/2 y0=m/2
    Output: T 3x3 transformation matrix
    """
    x0=n/2; y0=m/2;
    T= np.array([[1.0,0.0,-x0],
                 [0.0,1.0,-y0],
                 [0.0,0.0,1.0]])
    return(T)
#%%
# function for LM call for refinement of F
def Refine_F(F0,pts1,pts2):
    """
    Using Gold Standard Algorithm
    Input: F0: initial solution for LM 3x3 matrix
           pts1,pts2: lis of list of coordinates [[x1,y1],[x2,y2],...]
    Output: F_ref,P1_ref,P2_ref : refined properties
    """
    # make P2:[M|t] and Xi from F
    F0=F0/F0[-1,-1]
    P1,P2=get_Proj_mat(F0)
    M=P2[:,:-1]; t=P2[:,-1];
    # find HC rep of pts
    pts1_hc=convert2HC(pts1)
    pts2_hc=convert2HC(pts2)
    # find world pt
    world_pt_hc =[]
    for i in range(len(pts1)):
        world_pt_hc.append(Triangulate(pts1_hc[i,:],pts2_hc[i,:],P1,P2)) #will give HC
    world_pt_hc=np.array(world_pt_hc)
    # get physical coordinate of world pt
    world_pt = convert2phy(world_pt_hc.T) #format [[X,Y,Z],..]
    # make parameter vector p using M, t, X
    p0=np.zeros(3*len(pts1)+12)
    p0[:9]=np.reshape(M,(9))
    p0[9:12]=t
    p0[12:]=np.reshape(world_pt,(3*len(pts1)))
    # call to LM
    optim=least_squares(error_fun_F,p0,
                        method='lm',args=(pts1,pts2))
    p_star = optim['x']
    # get back P2,F
    M_star=p_star[:9]
    M_star= np.reshape(M_star,(3,3))
    t_star = p_star[9:12]
    tx= cross_rep_mat(t_star)
    F_ref = tx@M
    F_ref = F_ref/F_ref[-1,-1]
    P2_ref = np.zeros((3,4))
    P2_ref[:,:-1]=M_star
    P2_ref[:,-1]=t_star
    P1_ref=P1
    return(F_ref,P1_ref,P2_ref,M_star)
#%%
# function for defining cost function for LM
# optimization of Fundamental Matrix
def error_fun_F(p,pts1,pts2):
    """
    Gold-Standard Method cost function
    Input:
    ------------------------------------------------------------------
        p: parameter vector of size 3n+12
            3n: 3D physical coordinates of World_pts
```

```
            12: for Projection Matrix
            p=[M,t,X1,X2...,Xn]
            M=[M11,M12,M13,M21,...]
            t=[tx,ty,tz]
            Xi=[xi,yi,zi]
    ------------------------------------------------------------------------
        pts1,pts2: list of coordinates of pts in format [[x,y],..]

    Output: cost: geometric distance vector
    """
    # Construct P1,P2
    P1=np.hstack((np.eye(3),np.zeros((3,1))))
    M=p[:9]
    M=np.reshape(M,(3,3))
    t=p[9:12]
    P2=np.zeros((3,4))
    P2[:,:-1]=M ; P2[:,-1]=t;
    # construct World_pt from p
    world_pt = []
    for i in range(len(pts1)): world_pt.append(p[12+3*i:12+3*(i+1)])
    world_pt = np.array(world_pt) # rows as physical coord
    #make world pt HC
    world_pt_hc = convert2HC(world_pt) # [[x,y,z,1],....]
    # get extimates of world point using P1 and P2
    x1_hat_hc = P1@world_pt_hc.T
    x2_hat_hc = P2@world_pt_hc.T
    # convert to physical coordinates
    x1_hat = convert2phy(x1_hat_hc)
    x2_hat = convert2phy(x2_hat_hc)
    # find difference between physical coordinates
    diff1 = pts1-x1_hat
    diff2 = pts2-x2_hat
    error = np.hstack((diff1[:,0],diff1[:,1],diff2[:,0],diff2[:,1]))
    return(error)
#function for triangulating the world pt
def Triangulate(x1,x2,P1,P2):
    """
    Input: x1: Hc coordinate in left image
           x2: HC coordinate in right image
           P1,P2: Projection matrix for left and right resp
    Output: world_pt: the cooresponding world point in HC
    """
    x1= x1/x1[-1]; x2= x2/x2[-1]
    A=np.zeros((4,4))
    A[0,:]=x1[0]*P1[2,:]-P1[0,:]
    A[1,:]=x1[1]*P1[2,:]-P1[1,:]
    A[2,:]=x2[0]*P2[2,:]-P2[0,:]
    A[3,:]=x2[1]*P2[2,:]-P2[1,:]
    # solve AX =0 using svd
    u,d,vt=np.linalg.svd(A)
    i=np.argmin(np.abs(d))
    world_pt = vt[i,:]
    world_pt = world_pt/world_pt[-1]
    return(world_pt)
#function for obtaining Projection Matrics from F
def get_Proj_mat(F):
    """
    Input: F:3x3 matrix
    Output: P1=[I|0] P2=[exF|e]
    """
    F=F/F[-1,-1]
    P1= np.hstack((np.eye(3),np.zeros((3,1))))
    e = get_left_nullvec(F)
    ex = cross_rep_mat(e)
    P2= np.zeros((3,4))
```

```
    P2[:,:-1]=ex@F
    P2[:,-1]=e
    return(P1,P2)
# function for finding the right Null vector
def get_right_nullvec(F):
    """
    Input: F: 3x3 matrix
    Output: e: right Null vector of F
    by solving Fe=0
    """
    u,d,vt = np.linalg.svd(F)
    i_null= np.argmin(np.abs(d))
    e=vt[i_null,:]
    return(e)
# function for finding the left Null vector
def get_left_nullvec(F):
    """
    Input: F: 3x3 matrix
    Output: e: left Null vector of F
    by solving e_pr^T F=0 or FT e_pr = 0
    """
    u,d,vt = np.linalg.svd(F.T)
    i_null= np.argmin(np.abs(d))
    e=vt[i_null,:]
    return(e)
# function for getting cross-representation MAtrix
def cross_rep_mat(w):
    """
    Input:w 3x1 vector
    Output: W 3x3 matrix
    """
    # make Wx from w
    Wx=np.array([[0      , -1*w[2],     w[1]],
                 [w[2]   , 0      , -1*w[0]],
                 [-1*w[1], w[0]   ,       0]])
    return(Wx)
#%%
#function for finding F using RANSAC
def F_Ransac(pts1,pts2):
    """
    Input: pts1,pts2: list of list of coordinates of interest pts
    Output: in_indices_store: indices of inlier points of image  1 and image 2 respectively
                            format:[index1,index2,....] list of index numbers
    """
    #----define RANSAC parameters:-----#
    p=0.95  # probability that at least one of the N trials will be free of outliers
    n=9     # minimal set of correspondences chosen randomly for constructing F
    delta=10 # decision threshold to construct inlier set
    e=0.5   #worst case probability that a correspondence is an outlier
    N=int(np.ceil(np.log(1-p)/np.log(1-(1-e)**n))) #number of iterations needed
    #-----------------begin-------------------#
    n_total=len(pts1) # total number of correspondences
    # convert pts lists to array and into homogeneous form
    pts1_hc=convert2HC(pts1)
    pts2_hc=convert2HC(pts2)

    size_old=0
    count=0
    while N>count:
        #randomly sample n points from pts1 and pts2
        rand_indices=np.random.randint(n_total,size=n)
        ipts1=pts1_hc[rand_indices,:]
        ipts2=pts2_hc[rand_indices,:]
        # find Fundamental Matrix using these points: linear least sqaures
        iF = Funda_using_pts(ipts1,ipts2)
```

```
            P1,P2 = get_Proj_mat(iF)
            # find world pt
            world_pt_hc =[]
            for i in range(len(pts1)):
                 world_pt_hc.append(Triangulate(pts1_hc[i,:],pts2_hc[i,:],P1,P2)) #will give HC
            world_pt_hc = np.array(world_pt_hc) #[[X,Y,Z,1],..]
            # find estimated pts
            est_pts1_hc=P1@world_pt_hc.T
            est_pts2_hc=P2@world_pt_hc.T
            # get physical coord of estimate pts
            est_pts1 = convert2phy(est_pts1_hc)#[[x,y],..]
            est_pts2 = convert2phy(est_pts2_hc) #[[x,y],..]
            # find error distance
            error1=pts1_hc[:,:-1]-est_pts1
            error2=pts2_hc[:,:-1]-est_pts2
            distance1=np.linalg.norm(error1,axis=1)
            distance2=np.linalg.norm(error2,axis=1)
            # consensus
            in1 = np.where(distance1<=delta)[0]
            in2 = np.where(distance2<=delta)[0]
            in_indices=list(set(in1)&set(in2))
            size_new=len(in_indices) #size of ith inlier set
            #store the set of indices with largest size
            if size_new>size_old:
                 in_indices_store=in_indices
                 size_old=size_new
                 e = 1-size_old/n_total
                 N=int(np.ceil(np.log(1-p)/np.log(1-(1-e)**n)))
            count+=1
        print('RANSAC_max_inlier_set_was'+str(len(in_indices_store)))
        return(in_indices_store)
#%%
#function for finding Fundamental matrix using point correspondences: Linear Least sqaures
def Funda_using_pts(pts1_hc,pts2_hc):
    """
    Input: pts1_hc,pts2_hc: list of list of correponding points: size Nx3 in HC
           [[X,Y,1],...]
    Output: F: 3x3 array, Linear Least squre estimate
    F needs to be Conditioned to Rank 2
    """
    # convert points to physical coord
    pts1 = convert2phy(pts1_hc.T) #[[x,y],..]
    pts2 = convert2phy(pts2_hc.T) #[[x,y],..]
    # get Normalization transformation Matrix
    T1=Norm_Transform(pts1)
    T2=Norm_Transform(pts2)
    # Normalize Hc coordinates
    pts1_hc=(T1@pts1_hc.T).T
    pts2_hc=(T2@pts2_hc.T).T
    # make A matrix for Af=0
    A1=np.hstack((pts1_hc,pts1_hc,pts1_hc))
    A2=np.kron(pts2_hc,np.ones((1,3)))
    A=np.multiply(A1,A2)
    #solve for f using SVD
    u,d,vt=np.linalg.svd(A)
    f=vt[-1,:]
    F=np.reshape(f,(3,3))
    # apply Rank=2 constraint
    F=make_detF_0(F)
    # de-normalized F
    F=T2.T@F@T1
    F=F/F[-1,-1]
    return(F)
#function to make det(F)=0
def make_detF_0(F):
```

```
    """
    Input: F 3x3 matrix
    Output: F0: F with det(F)=0
    """
    # apply Rank=2 constraint
    U,D,VT=np.linalg.svd(F)
    d_prime=D
    d_prime[np.argmin(D)]=0
    F0=U@np.diag(d_prime)@VT
    return(F0)
# function for finding Transformation for normalization
def Norm_Transform(pts):
    """
    This just shifts the image origin to centroid of pts
    Input: pts: list of list of corrdinates [[x1,y1],[x2,y2],...]
    Output: Transformation T for normalization
    """
    pts=np.array(pts)
    # find centroid coordinataes
    x_mean= np.mean(pts[:,0])
    y_mean= np.mean(pts[:,1])
    # shift the origin to centroid pt
    shifted_pts=np.zeros((pts.shape),dtype=np.float32)
    shifted_pts[:,:]=pts
    shifted_pts[:,0]+= -1*x_mean
    shifted_pts[:,1]+= -1*y_mean
    # find mean distance of all points from new origin
    mean_d = np.mean(np.linalg.norm(shifted_pts,axis=1))
    # scale mean distance to sqrt(2)
    scale= np.sqrt(2)/mean_d
    T = np.array([[scale,0.0,-1*scale*x_mean],
                  [0.0,scale,-1*scale*y_mean],
                  [0.0, 0.0, 1.0]])
    return(T)
#=================================HW9=================================#
```