# COMPUTER VISION

# ECE 661

# HOMEWORK 10

Tanzeel U. Rehman

Email: *trehman@purdue.edu*

## 1. Introduction:

This homework performs the 3D image reconstruction using stereo pair. Th reconstruction is related to world 3D coordinates through projective distortion and therefore can be called as projective reconstruction. The following provides the detailed procedure.

### 1.1. Description of Methods:

#### A. Estimation of Fundamental matrix through linear least squares:

Let a pair of corresponding points between a stereo pair can be represented as (*x*, *x'*), then from theory of epipolar geometry, these point can be related as given by Eq 1. Now, we need atleast 8 correspondences between a pair of images in order to solve for the fundamental matrix (F). Using these points, we can rewrite Eq. 1 as Eq. 2:

$$x'^T F x = 0 \quad \text{(Eq.1)}$$

$$Af = 0 \quad \text{(Eq.2)}$$

Where,

$$A = [x'_i x_i \quad x'_i y_i \quad x'_i \quad y'_i x_i \quad y'_i y_i \quad x_i \quad y_i \quad 1\ ]$$

$$f = [F_{11} \quad F_{12} \quad F_{13} \quad F_{21} \quad F_{22} \quad F_{23} \quad F_{31} \quad F_{32} \quad F_{33}\ ]$$

The following is detailed procedure for estimating the fundamental matrix using linear least squares (LLS):

- Given atleast 8 correspondences between an image pair, the pixel coordinates were normalized to have zero mean and the average distance from the center of $\sqrt{2}$. These transformations were referred as $T_1$ and $T_2$ for image 1 and image 2 of a stereo pair.

- Using the correspondences, we can arrange matrix A and then F matrix was solved using SVD with solution being the eigen vector corresponding to the smallest eigen value.

- The F matrix is then conditioned to make its rank as 2. This can be ensured by enforcing that the last eigen value to be zero.

- The F matrix was then denormalized by using the relation: $F = {T_2}^T F T_1$

- We can now estimate the epipoles *e* and *e'* by computing the right and left null vectors of matrix *F*.

- Finally, the projection matrices of the canonical form for the two cameras can be estimated as:

$$[e'|_x xF|e'] = P'$$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

**B. Refining the fundamental matrix:**

Now using the F matrix obtained from LLS, we can obtain the world coordinates from 2D points ($x$, $x'$) of a stereo pair by solving Eq.3 with a constraint that $||X_{world} = 1||$.

$$AX_{world} = 0 \qquad \text{(Eq.3)}$$

Where,

$$A = \begin{bmatrix} x_i P_3{}^T - P_1{}^T \\ y_i P_3{}^T - P_2{}^T \\ x_i{}' P_3{}^T - P_1{}^T \\ y'{}_i P_3{}^T - P_2{}^T \end{bmatrix}$$

Now, using the world coordinates of the points can be used to optimize the F with the help of LM algorithm via the cost function given by Eq.4.

$$d_{geom}^2 = \sum_i \|x_i - \hat{x}_i\|^2 + \|x_i{}' - \hat{x}_i{}'\|^2 \qquad \text{(Eq.4)}$$

Where, $\hat{x}_i$ and $\hat{x}_i{}'$ are the world coordinates in the first and second image, respectively.

**C. Image rectification:**

To rectify the images, we first need to shift the second image by using $T_1$ matrix. Then, we computed the angle that the epipole makes with x-axis and rotate the image so that the epipoles become parallel to x-axis:

$$e = \begin{bmatrix} f \\ 0 \\ 1 \end{bmatrix}$$

We, then computed the $G$ matrix, that sends the epipoles to infinity as:

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

Now, the image is translated back to the original center using the $T_2$ matrix. So, the overall homography ($H_2$) that is needed to rectify the second image can be given by $H_2 = T_2GRT_1$. Finally, the $H_1$ homography for rectifying the image 1 of a stereo pair is computed through least squares minimization of $min_{H1} \sum_d ( H_1 x_i, H_2 x_i{}')$. This will force the corresponding epipolar lines to be on the same row of images. The details can be seen in the book titled “Multiple View Geometry”. After estimating the homographies the images can be rectified.

### D. Interest point detection:

The interest points were detected by finding the edges in the rectified images using the Canny edge detector. After, estimating the correspondences between the edges of two rectified images, we refined the fundamental matrix and all other results . Finally, the world coordinates were found using triangulation.

## 1.2. Results:


**(a)**


**(b)**

**Fig 1: Stereo images of a scene, (a) left image and (b) right image**


**Fig 2: The manually selected points used for image rectification**

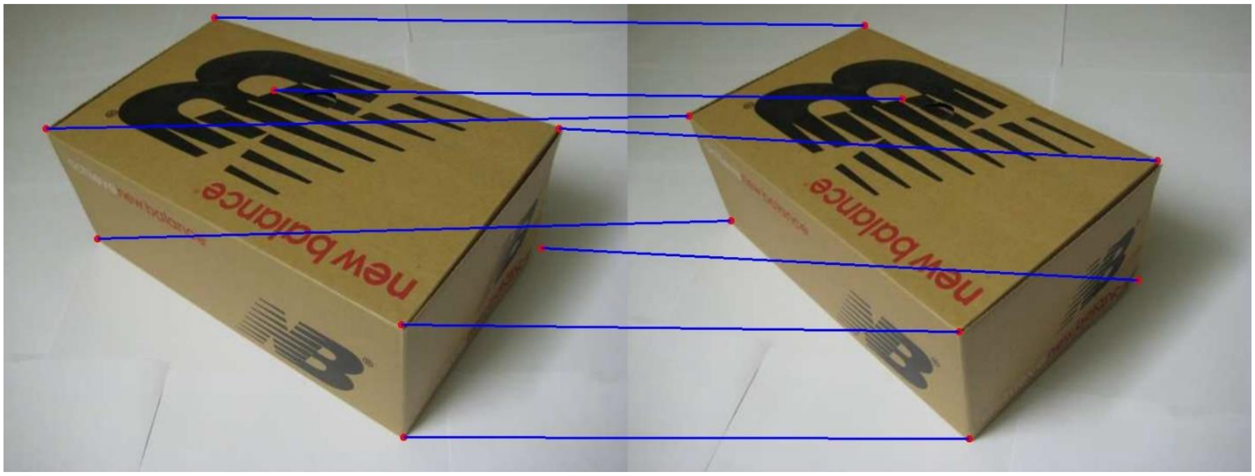
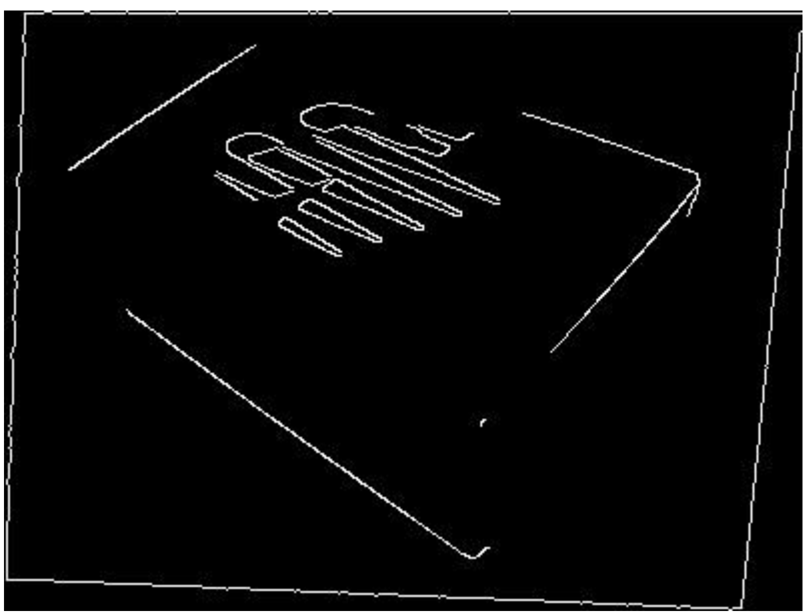


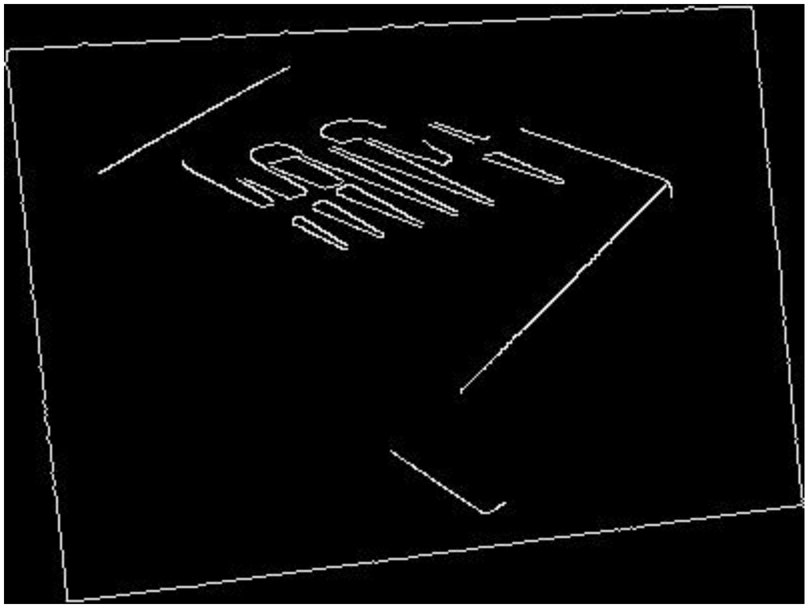

**Fig 3: Correspondence among the Manually selected points**


**(a)**


**(b)**

**Fig 4: Rectified stereo images of a scene, (a) left image and (b) right image**





**(b)**

**Fig 5: Edges of rectified stereo images, (a) left image and (b) right image**

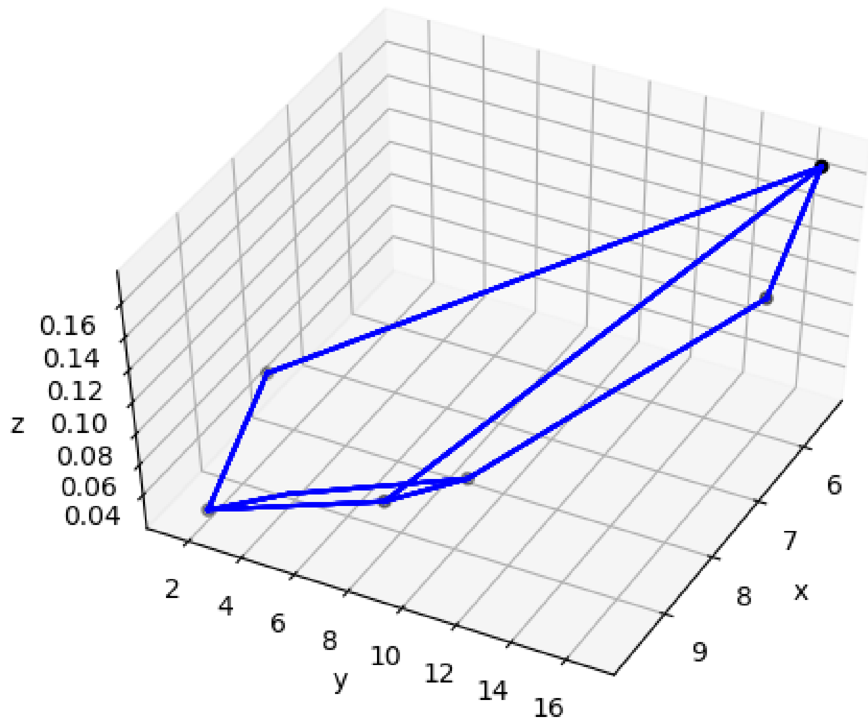



**Fig 6: The boundary of 3D projected points with manually selected points from LLS**

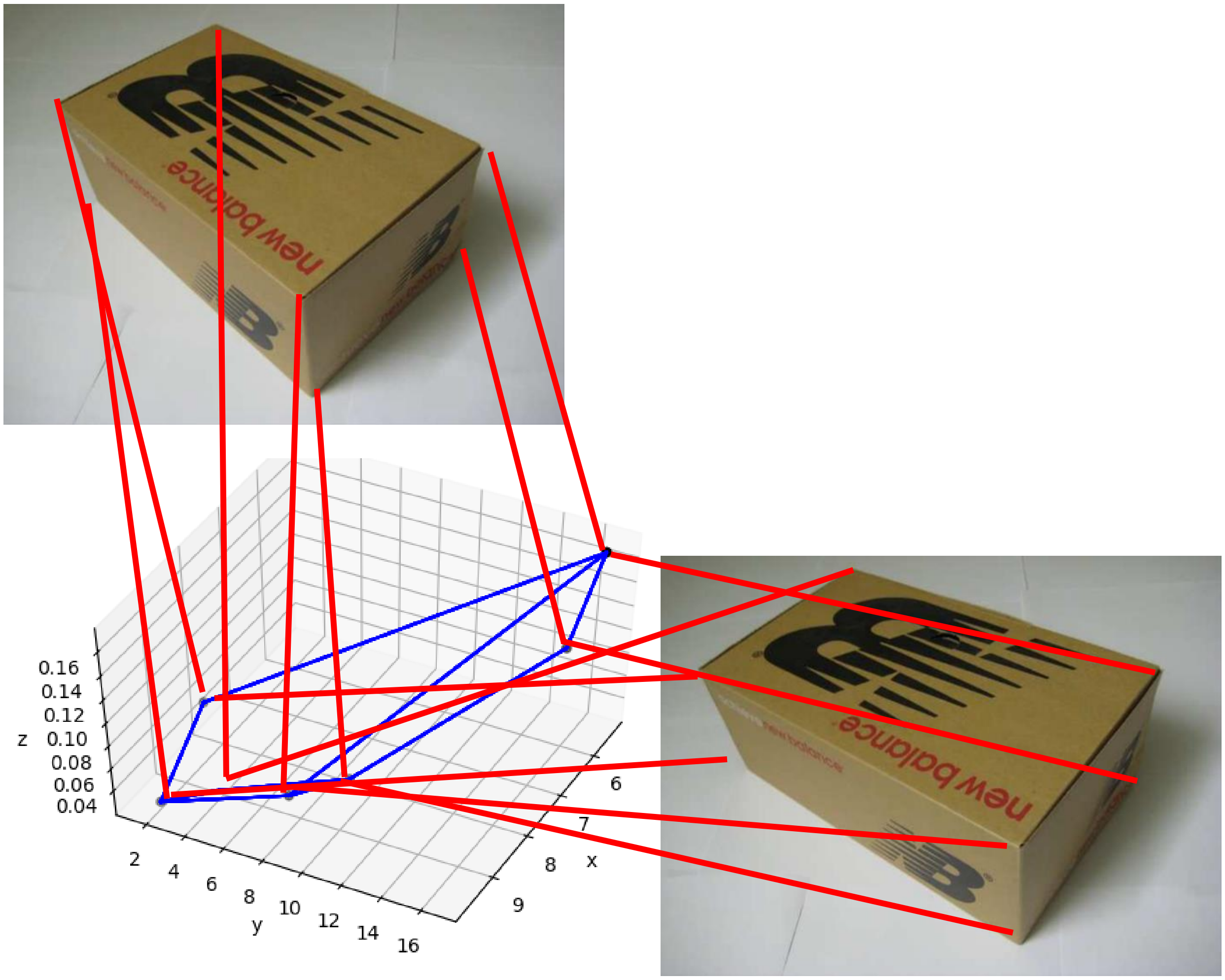



**Fig 7: The boundary of 3D projected points with manually selected points from LLS for scene understanding**

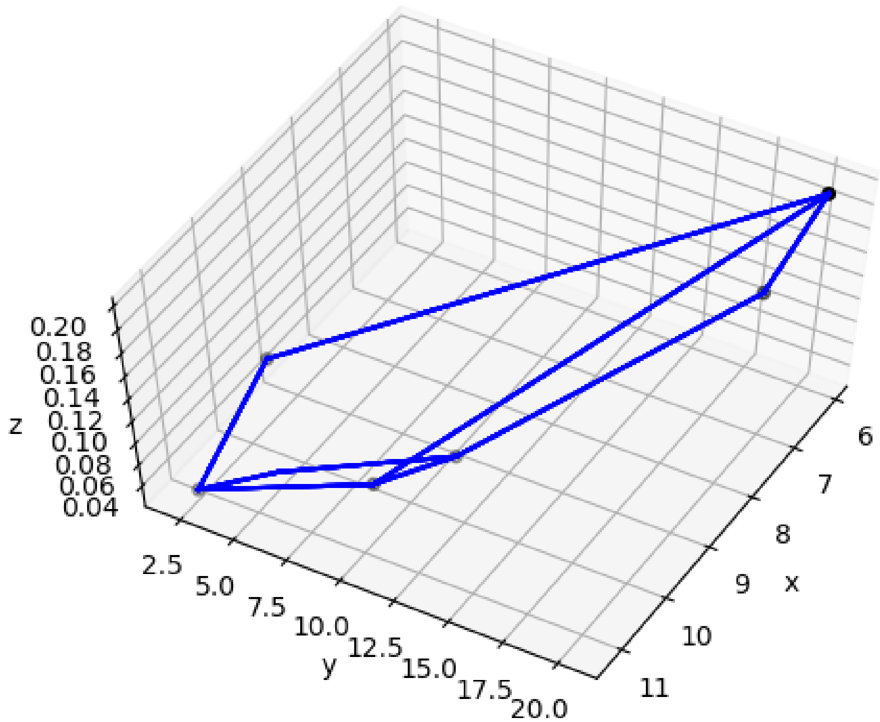



**Fig 8: The boundary of 3D projected points with manually selected points using LM optimization**

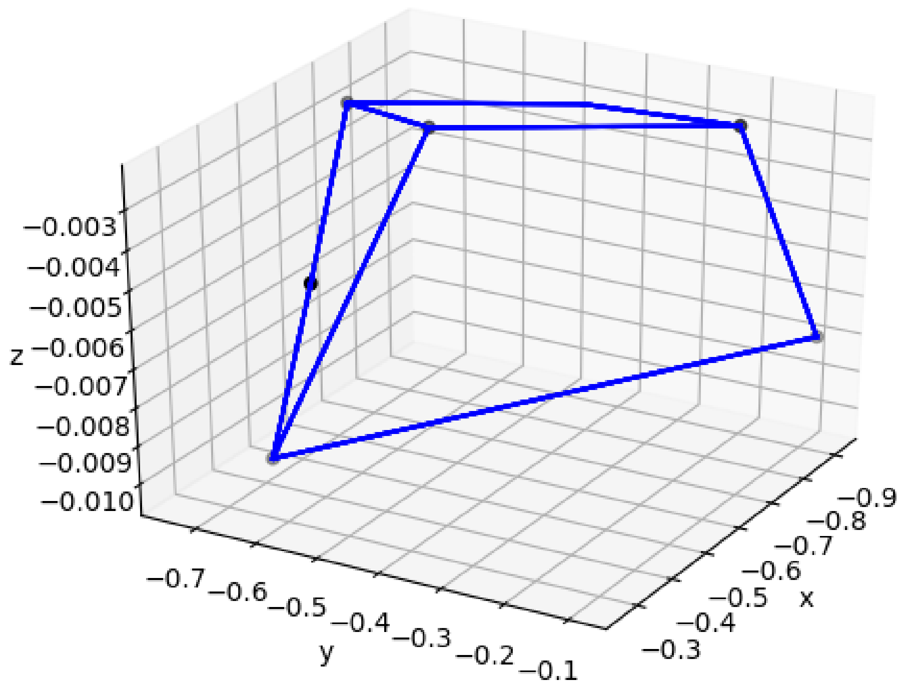



**Fig 9: The boundary of 3D projected points with Canny edges using LM optimization**

### 1.3. Observations:

1) The results achieved by Canny edges did not yield the expected results. There seems to be some improvement needed at the edge detection task.

## 3. Source Code:

### 3.1.Function calls for Task 1.1:

```
'''-------------Main function for image rectification-----------'''
im1 = cv2.imread('1.png')
im2 = cv2.imread('2.png')
```

```
#Manually selected points for image 1 and image 2
corners_1                                                                                          =
np.array([[54,159,1],[270,17,1],[509,410,1],[711,159,1],[120,300,1],[512,554,1],[689,312,1],[346,11
0,1]])
corners_2                   =                   np.array([[78,143,1],[303,27,1],[424,419,1],[678,
200,1],[132,277,1],[437,556,1],[653,353,1],[351,121,1]])
#----Show the correspondences between the image points-----
plot_img = Show_Correspondences(im1,im2,corners_1,corners_2)
cv2.imwrite('correspondences_manual.jpg',plot_img)
#Normalize the corner points
T1 = normalize_points(corners_1)
T2 = normalize_points(corners_2)
#Compute the fundamental amtrix
F = Find_F_LLS(corners_1,corners_2,T1,T2)
#Denormalize the F matrix
F = T2.T@(F@T1)
# HC to physical
F = F/F[2,2]
#Get the epipoles
e1,e2 = compute_epipoles(F)
e1,e2 = e1/e1[2],e2/e2[2]
P,P_dash = Compute_P_P_dash(e2,F)
#Optimize the F matrix using LM
F_LM=LM_Fundamental(F,corners_1,corners_2)
F_LM = F_LM/F_LM[2,2]
#Get the epipoles after LM
e1_LM,e2_LM = compute_epipoles(F_LM)
e1_LM,e2_LM = e1_LM/e1_LM[2],e2_LM/e2_LM[2]
P_LM,P_LM_dash = Compute_P_P_dash(e2_LM,F_LM)
#Find the homographies needed for the rectification
H1,H2,F_rec,x1_rec,x2_rec,e1_rec,e2_rec=Find_Homographies(im1,e1_LM,e2_LM,P_LM,P_LM_das
h,F_LM,corners_1,corners_2)
#Rectify the two images
rectified_im1,corners_1_rectified=Rectify_imgae(H1,im1,corners_1,'rectified_img_1.jpg')
rectified_im2,corners_2_rectified=Rectify_imgae(H2,im2,corners_2,'rectified_img_2.jpg')

x_world,y_world,z_world= triangulate(corners_1_rectified,corners_2_rectified,P_LM,P_LM_dash)
fig = plt . figure ()
ax = fig . add_subplot ( 111 , projection = '3d' )
ax.scatter(x_world[:6],y_world[:6],z_world[:6],color = 'k')
#plt.savefig('border_pts.png')

for i in range(7):
   ax.text(x_world[i],z_world[i],y_world[i],str(i+1))
   ax.plot([x_world[0],x_world[1]],[y_world[0],y_world[1]],[z_world[0],z_world[1]],color='g')
   ax.plot([x_world[0],x_world[2]],[y_world[0],y_world[2]],[z_world[0],z_world[2]],color='b')
   ax.plot([x_world[0],x_world[4]],[y_world[0],y_world[4]],[z_world[0],z_world[4]],color='r')
   ax.plot([x_world[4],x_world[5]],[y_world[4],y_world[5]],[z_world[4],z_world[5]],color='k')
```

```
    ax.plot([x_world[2],x_world[5]],[y_world[2],y_world[5]],[z_world[2],z_world[5]],color='c')
    ax.plot([x_world[1],x_world[3]],[y_world[1],y_world[3]],[z_world[1],z_world[3]],color='y')
    ax.plot([x_world[5],x_world[6]],[y_world[5],y_world[6]],[z_world[5],z_world[6]],color='b')
    ax.plot([x_world[3],x_world[6]],[y_world[3],y_world[6]],[z_world[3],z_world[6]],color='b')
    ax.plot([x_world[2],x_world[3]],[y_world[2],y_world[3]],[z_world[2],z_world[3]],color='b')
    ax.view_init(None,30)
    ax.set_xlabel('x');ax.set_ylabel('y');ax.set_zlabel('z')
   plt.savefig('border_pts.png')
'''
'''-----------Canny Edge based refinement-----------'''
'''
# Start improving the rectification using canny edges
image1gray = cv2.cvtColor (rectified_im1, cv2.COLOR_BGR2GRAY)
edges1 = cv2.Canny ( image1gray , 255 * 1.75 , 255 )
image2gray = cv2.cvtColor (rectified_im2, cv2.COLOR_BGR2GRAY)
edges2 = cv2.Canny ( image2gray , 255 *1.75, 255 )
corners_1_ed,corners_2_ed = return_correspondences(image1gray,image2gray,edges1,edges2)
corners_1_ed = np.column_stack((corners_1_ed,np.ones(len(corners_1_ed))))
corners_2_ed = np.column_stack((corners_2_ed,np.ones(len(corners_2_ed))))

#Normalize the corner points
T1 = normalize_points(corners_1_ed)
T2 = normalize_points(corners_2_ed)
#Compute the fundamental amtrix
F = Find_F_LLS(corners_1_ed,corners_2_ed,T1,T2)
#Denormalize the F matrix
F = T2.T@(F@T1)
# HC to physical
F = F/F[2,2]
#Get the epipoles
e1,e2 = compute_epipoles(F)
P,P_dash = Compute_P_P_dash(e2,F)

#Optimize the F matrix using LM
#F_LM=LM_Fundamental(F,corners_1_ed,corners_2_ed)
#F_LM = F_LM/F_LM[2,2]
#Get the epipoles after LM
e1_LM,e2_LM = compute_epipoles(F)
e1_LM,e2_LM = e1_LM/e1_LM[2],e2_LM/e2_LM[2]
P_LM,P_LM_dash = Compute_P_P_dash(e2_LM,F)
x_world,y_world,z_world= triangulate(corners_1_rectified,corners_2_rectified,P_LM,P_LM_dash)
fig = plt . figure ()
ax = fig . add_subplot ( 111 , projection = '3d' )
ax.scatter(x_world[:6],y_world[:6],z_world[:6],color = 'k')
#plt.savefig('border_pts.png')

for i in range(7):
    ax.text(x_world[i],z_world[i],y_world[i],str(i+1))
```

```
        ax.plot([x_world[0],x_world[1]],[y_world[0],y_world[1]],[z_world[0],z_world[1]],color='g')
        ax.plot([x_world[0],x_world[2]],[y_world[0],y_world[2]],[z_world[0],z_world[2]],color='b')
        ax.plot([x_world[0],x_world[4]],[y_world[0],y_world[4]],[z_world[0],z_world[4]],color='r')
        ax.plot([x_world[4],x_world[5]],[y_world[4],y_world[5]],[z_world[4],z_world[5]],color='k')
        ax.plot([x_world[2],x_world[5]],[y_world[2],y_world[5]],[z_world[2],z_world[5]],color='c')
        ax.plot([x_world[1],x_world[3]],[y_world[1],y_world[3]],[z_world[1],z_world[3]],color='y')
        ax.plot([x_world[5],x_world[6]],[y_world[5],y_world[6]],[z_world[5],z_world[6]],color='b')
        ax.plot([x_world[3],x_world[6]],[y_world[3],y_world[6]],[z_world[3],z_world[6]],color='b')
        ax.plot([x_world[2],x_world[3]],[y_world[2],y_world[3]],[z_world[2],z_world[3]],color='b')
        ax.view_init(None,30)
        ax.set_xlabel('x');ax.set_ylabel('y');ax.set_zlabel('z')
       plt.savefig('border_pts_Canny.png')
```

## A. Different functions for image rectification:

```
 from mpl_toolkits.mplot3d import Axes3D

def Show_Correspondences(img_1_color,img_2_color,corners_1,corners_2):
   '''
   Function for plotting the corresponding points on the image
   '''
   #Shape of input images
   h1,w1=img_1_color.shape[0:2]
   h2,w2=img_2_color.shape[0:2]
   #Find the maximum height from 2 images. This will be the height of output image
   max_height = max(h1,h2)
   #create an empty image having size of max_height x w1+w2
   plot_img = np.zeros((max_height,(w1+w2),3))
   #Fill empty image with image1 and 2, this will leave empty border on the
   #image of least height
   plot_img [0:h1,0:w1,:] = img_1_color
   plot_img [0:h2,w1:,:] = img_2_color
   for i in range(len(corners_1)):
      #Plot a circle of red color with a radius of 3 to mark the corner points on img1.
      cv2.circle(plot_img,tuple(corners_1[i,0:2].astype(int)),3,(0,0,255),2)
      #Plot a circle of red color with a radius of 3 to mark the corner points on img2.
      cv2.circle(plot_img,tuple([int(corners_2[i,0]+w1),int(corners_2[i,1])]),3,(0,0,255),2)  #shift the
pointer by width of img1
      #Plot the blue line joining corresponding points on images

cv2.line(plot_img,tuple(corners_1[i,0:2].astype(int)),tuple([int(corners_2[i,0]+w1),int(corners_2[i,1])]),(2
55,0,0),2)
   return plot_img

def draw_and_save(img_color,savename,corners):

   for i in range (len(corners)):
      #Plot a circle of red color with a radius of 3 to mark the corner points.
```

```python
        cv2.circle(img_color, tuple(corners[i,0:2]), 3, (0,0,255), 2)
        #cv2.imshow(savename,img_color)
        cv2.imwrite(savename,img_color)

def normalize_points(corners):
    '''Function for computing the T matrix from an array of input points'''
    mean_x = np.mean(corners[:,0])
    mean_y = np.mean(corners[:,1])
    mean_dist = np.sqrt((corners[:,0]- mean_x)**2+(corners[:,1]- mean_y)**2)

    mean_dist= np.sum(mean_dist)/len(corners)
    scale = np.sqrt(2)/mean_dist
    xtr = -scale*mean_x
    ytr = -scale*mean_y
    T = np.array([[scale,0,xtr],[0,scale,ytr],[0,0,1]])
    return T

def Find_F_LLS(corners_1,corners_2,T1,T2):
    ncorners_1 = (T1@corners_1.T).T
    ncorners_2 = (T2@corners_2.T).T
    #Initialize an empty Fundamental matrix
    F = np.zeros((3,3))
    # Find num of points provided
    n = ncorners_1.shape[0]
    #Initialize A Design matrix having size of n x 9
    A = np.zeros((n,9))
    #Loop through all the points provided and stack them vertically, this will result in 2n x 9 Design matrix
    for i in range (n):
        A[i]=Get_A_matrix(ncorners_1[i],ncorners_2[i])
    #Decompose the A matrix and obtain the
    U,D,V = np.linalg.svd(A)
    h = V.T[:,8] #Eigen vector corresponding to the smallest eigen value of D
    # Rearrange the vector h to fundamental matrix F
    F[0] = h[0:3]
    F[1] = h[3:6]
    F[2] = h[6:9]
    #Condition the F to make it rank 2 matrix
    U,D,V = np.linalg.svd(F)
    D[2]=0
    F = U@(np.diag (D)@V)
    return F

def Get_A_matrix(ncorners_1,ncorners_2):
    # Extract the x and y coordinates from a point pair
    x1,y1=ncorners_1[0], ncorners_1[1]
    x2,y2=ncorners_2[0], ncorners_2[1]
    #  Make A matrix
    A=np.array([[x2*x1,x2*y1,x2,y2*x1,y2*y1,y2,x1,y1,1]])
```

```
    return A

def compute_epipoles(F):
    U,D,V = np.linalg.svd(F)
    e1 = V [2].T
    e2 = U [:,2]
    return e1,e2

def Compute_P_P_dash(e2,F):
    e2x = np.array([[0,-e2[2],e2[1]],[e2[2],0,-e2[0]],[-e2[1],e2[0],0]])
    P_dash = np.column_stack((e2x@F,e2))
    P = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])
    return P,P_dash

def func_LM_Fundamental(F, crns_1_x,crns_1_y,crns_2_x,crns_2_y):
    '''
    Function that need to be supplied to the scipy optimize module. Requires all inputs as 1d array
    The first argument will be optimized as a result. Optimization will be done based on the Eucledian
distance
    '''
    #Combine the x and y from the domain and range points
    corners_1 = (np.array([crns_1_x,crns_1_y]))
    corners_2 = (np.array([crns_2_x,crns_2_y]))
    #Reshape the F matrix to be 3 x 3 matrix
    F = F.reshape((3,3))

    # Get the epipoles
    e1,e2 = compute_epipoles(F)
    #e1,e2 = e1/e1[2],e2/e2[2]
    P,P_dash = Compute_P_P_dash(e2,F)
    Cost = []
    for i in range(len(crns_1_x)):
        A = np.array([[crns_1_x[i]*P[2,:]-P[0,:]],[crns_1_y[i]*P[2,:]-P[1,:]],
                    [crns_2_x[i]*P_dash[2,:]-P_dash[0,:]],
                    [crns_2_y[i]*P_dash[2,:]-P_dash[1,:]]])

        U,D,V = np.linalg.svd(A.squeeze())
        X_world = V[3].T
        X_world = X_world / np.linalg.norm(X_world)

        Estimate_corners_1 = P @ X_world
        Estimate_corners_1 = Estimate_corners_1/Estimate_corners_1[2]
        Estimate_corners_2 = P_dash @ X_world
        Estimate_corners_2 =Estimate_corners_2/Estimate_corners_2[2]
        #Compute the residuals based on Eucledian distance
        Cost.append(np.linalg.norm(Estimate_corners_1[0:2]-corners_1[:,i])**2)
        Cost.append(np.linalg.norm(Estimate_corners_2[0:2]-corners_2[:,i])**2)
    return np.array(Cost)
```

```
def LM_Fundamental(F,corners_1,corners_2):
    '''
    Function for computing the LM refined Fundamental matrix
    '''
    #Reshape the input F matrix to be a vector obtained from Linear approach
    F0 = np.reshape(F,9)
    crns_1_x = corners_1[:,0]
    crns_1_y = corners_1[:,1]
    crns_2_x = corners_2[:,0]
    crns_2_y = corners_2[:,1]

    res_lsq = least_squares(func_LM_Fundamental, F0,
args=(crns_1_x,crns_1_y,crns_2_x,crns_2_y),method = 'lm')
    F_LM = res_lsq.x
    F_LM = F_LM.reshape((3,3))
    return F_LM

def Find_Homographies(img,e1,e2,P1,P2,F,x1,x2):
    h,w=img.shape[0:2]
    npts = len(x1)
    #Get the R,T,G matrices
    ang = np.arctan(-(e2[1]- h/2 )/(e2[0]- w/2))
    f = np.cos (ang)*(e2[0]- w/2)-np.sin(ang)*(e2[1]-h/2)
    R = np.array([[np.cos(ang),-np.sin(ang),0],[np.sin(ang),np.cos(ang),0],[0,0,1]])
    T = np.array([[1,0,-w/2],[0,1,-h/2],[0,0,1]])
    G = np.array([[1,0,0],[0,1,0],[-1/f,0,1]])
    #Compute the homogrpahy for second image
    H2= G@(R@T)
    #Preserve the center
    c_pt = np.array([w/2,h/2,1])
    c_rec = H2@c_pt
    c_rec = c_rec/c_rec[2]
    #Translate the center of 2nd image back to original center
    T = np.array([[1,0,w/2-c_rec[0]],[0,1,h/2-c_rec[1]],[0,0,1]])
    #Copute the homoraphy for the second image
    H2 = T @ H2
    #Compute the homography for the first image
    #Compute M matrix
    e2x = np.array([[0,-e2[2],e2[1]],[e2[2],0,-e2[0]],[-e2[1],e2[0],0]])
    E = np.array([e2,e2,e2]).T
    M = (e2x @ F) + E

    #Get the H0 Homographjy
    H0 = H2 @ M
    #project the correspondences
    x1_hat = np.zeros((npts,3))
```

```
    x2_hat = np.zeros((npts,3))
    for i in range(npts):
        temp = H0 @ x1[i]
        x1_hat[i] = temp/temp[2]
        temp = H2 @ x2[i]
        x2_hat[i] = temp/temp[2]
    #Linear least squares for finding the Ha
    A = x1_hat
    b = x2_hat [:,0]
    x = np.linalg.pinv (A)@ b
    Ha = np.array([[x[0],x[1],x[2]],[0,1,0],[0,0,1]])

    H1 = Ha @ H0
    #preserve the center
    c_rec = H1@c_pt
    c_rec = c_rec/c_rec[2]
    #Translate the center of 2nd image back to original center
    T1 = np.array([[1,0,w/2-c_rec[0]],[0,1,h/2-c_rec[1]],[0,0,1]])
    #Copute the homoraphy for the 1st image
    H1 = T1 @ H1
    F_rec = (np.linalg.pinv(H2.T))@(F@np.linalg.pinv(H1))
    #Get the rectified epipoles
    e1_rec,e2_rec=compute_epipoles(F_rec)
    e1_rec,e2_rec = e1_rec/e1_rec[2],e2_rec/e2_rec[2]
    #Compute the rectified coordinates of corners
    x1_rec = np.zeros((npts,3))
    x2_rec = np.zeros((npts,3))
    for i in range(npts):
        temp = H1 @ x1[i]
        x1_rec[i] = temp/temp[2]
        temp = H2 @ x2[i]
        x2_rec[i] = temp/temp[2]
    return H1,H2,F_rec,x1_rec,x2_rec,e1_rec,e2_rec

'''---Code modified from HW 5----'''
def Rectify_imgae(Homography,image,corners,filename):
    h,w=image.shape[0:2]
    #Get the max min and length width of projected image on world plane
    xmin, ymin,xmax,ymax,width_corr,height_corr = Bounds_Undistorted(Homography,image)
    H_scale=np.array([[(w/width_corr)/2, 0, 0],[0, (h/height_corr)/2, 0],[0, 0, 1]])
    Homography=H_scale@Homography;
    # New bounds of the rectified image after scaling it down
    xmin, ymin,xmax,ymax,width_corr,height_corr = Bounds_Undistorted(Homography,image)
    #print (xmin, ymin,xmax,ymax,width_corr,height_corr)
    #Create an empty image
    rectified_image = np.zeros((int(np.round(height_corr)), int(np.round(width_corr)), 3),dtype='uint8')
    #Start rectifying the image by going through all the pixels
    height, width = rectified_image.shape[:2]
```

```
    H_inv = np.linalg.inv(Homography)
    for i in range(height):
        for j in range(width):
            k1 = j + xmin
            k2 = i + ymin
            X_domain = [k1,k2]
            X_domain = np.array(X_domain)
            X_domain = np.append(X_domain,1)
            X_range = np.matmul(H_inv, X_domain)
            X_range = X_range/X_range[-1]
            if(X_range[0] > 0 and X_range[1] > 0 and X_range[0] < image.shape[1]-1 and X_range[1] <
image.shape[0]-1):
                rectified_image[i,j] = RGB_Averaged(image,X_range)
    #Write the rectified image
    cv2.imwrite(filename,rectified_image)
    corners_rec = np.zeros_like(corners)
    for i in range(len(corners)):
        temp = Homography @ corners[i]
        temp=temp/temp[2]
        corners_rec[i] = np.array([temp[0]-xmin,temp[1]-ymin,1])

    return rectified_image,corners_rec

def RGB_Averaged(img,Range_point) :
    x= int(math.floor(Range_point[0]))
    xx= int (math.ceil(Range_point[0]))
    y= int (math.floor(Range_point[1]))
    yy= int (math.ceil(Range_point[1]))

    w1= 1/np.linalg.norm (np.array ([Range_point [0] -x , Range_point [1] -y]))
    w2= 1/np.linalg.norm (np.array ([Range_point [0] -x , Range_point [1] -yy]))
    w3= 1/np.linalg.norm (np.array ([Range_point [0] -xx , Range_point [1] -y]))
    w4= 1/np.linalg.norm (np.array ([Range_point [0] -xx , Range_point [1] -yy]))

    RGBVal = (w1*img [y] [x] + w2*img [yy][x] + w3*img [y] [xx] + w4*img [yy] [xx])/ (w1 + w2 + w3 + w4)
    return RGBVal

def Bounds_Undistorted(Homography,image):
    #Shape of the distorted image
    image_shape = image.shape
    #Distorted Homogeneous Coordinates of image Bounds
    ImgP= np.array([0,0,1])     # Top left corner of image (X,Y,1)
    ImgQ= np.array([image_shape[1],0,1])    # Top right corner
    ImgS = np.array([image_shape[1],image_shape[0],1]) #Bottom right
    ImgR = np.array([0,image_shape[0],1])   #bottom left

    #Apply the homography on the distroted image bounds to obtain the Corrected image bounds
    WorldP = np.dot(Homography,ImgP)
```

```
    WorldP = WorldP/WorldP[2]
    WorldQ = np.dot(Homography,ImgQ)
    WorldQ = WorldQ/WorldQ[2]
    WorldS = np.dot(Homography,ImgS)
    WorldS = WorldS/WorldS[2]
    WorldR = np.dot(Homography,ImgR)
    WorldR = WorldR/WorldR[2]

    #Find the extreme points of the corrected image bounds
    max_point = np.maximum(np.maximum(np.maximum(WorldP ,WorldQ ), WorldS), WorldR)
    min_point = np.minimum (np.minimum (np.minimum(WorldP,WorldQ), WorldS), WorldR)
    #Find the coordinates of cextreme points of corrected image bounds
    xmax,ymax = max_point[0],max_point[1]
    xmin,ymin = min_point[0],min_point[1]
    # New width and height of corrected image
    width_corr = (xmax-xmin)
    height_corr = (ymax-ymin)
    return xmin, ymin,xmax,ymax,width_corr,height_corr

def Find_Correspondences (im1,im2,Edges1,Edges2 ) :
    e_coords = []
    e_coords2 = []
    for i in range (3,len (Edges1)-3):
        for j in range ( 3 , len ( Edges1 [ 0 ] ) -3 ) :
            #Check if this is edge
            if (Edges1 [i,j ] == 255) :
                dist =1e10
                ii = i
                for jj in range ( 3 , len ( Edges2 [ 0 ] ) - 3 ) :
                    if (np.linalg.norm (im1[i-2: i+3 ,j-2: j+ 3 ]-im2 [ii-2:ii+3 , jj-2:jj+3 ] )<dist ):
                        t = jj
                        dist = np.linalg.norm (im1 [ i-2: i+3 ,j-2: j+3 ]-im2 [ii -2:ii +3,jj-2: jj + 3])
                e_coords.append ( [ i,j ] )
                e_coords2.append ( [ ii ,t] )
    return np.array(e_coords) , np.array(e_coords2)

def triangulate (corners_1,corners_2,P,P_dash):
    X_world =[]
    crns_1_x = corners_1[:,0]
    crns_1_y = corners_1[:,1]
    crns_2_x = corners_2[:,0]
    crns_2_y = corners_2[:,1]

    for i in range(len(crns_1_x)):
        A = np.array([[crns_1_x[i]*P[2,:]-P[0,:]],[crns_1_y[i]*P[2,:]-P[1,:]],
                    [crns_2_x[i]*P_dash[2,:]-P_dash[0,:]],
                    [crns_2_y[i]*P_dash[2,:]-P_dash[1,:]]])
```

```
        U,D,V = np.linalg.svd(A.squeeze())
        temp = V[3].T
        temp = temp /temp[3]
        X_world.append(temp)
    X_world= np.array(X_world)
    #return X_world
    return np.array(X_world[:,0]),X_world[:,1],X_world[:,2]
def
Get_NCC_Correspondences(image1,image2,Cornerslist_1,Cornerslist_2,window_dim=21,reject_ratio=0.
9):

    #Convert the corner lists to the arrays
    corner_image1 = (Cornerslist_1)
    corner_image2 = (Cornerslist_2)

    win_half = int(window_dim/2)
    #Initialize an empty list for storing corners
    valid_correspondences = []

    #Initialize 2D matrix to store the distances of a specific point in image 1 with everyother point in
image 2
    #Size will be num_corners_1 x num_corners_2
    F = np.zeros((len(corner_image1),len(corner_image2)))

    for y in range(15,len(corner_image1)-15):
        for x in range(15,len(corner_image2)-15):
            f1 =  Get_window(image1,win_half,corner_image1[y,0],corner_image1[y,1])
            f2 = Get_window(image2,win_half,corner_image2[x,0],corner_image2[x,1])
            mean1 = np.mean(f1)
            mean2 = np.mean(f2)
            numemnator = np.sum((f1- mean1)*(f2- mean2))
            denomenator = np.sqrt((np.sum((f1- mean1)**2))*(np.sum((f2- mean2)**2)))
            F[y,x] = numemnator/denomenator

    #Identify the corresponding corner points in the two images by thresholding
    for y in range(len(corner_image1)):
        x=np.argmax(F[y,:])
        if F[y,x] > reject_ratio:
            F[:,x] = np.NINF   #Mark that this column has been taken to avoid double correspondence (hard
learn't lesson)

valid_correspondences.append([corner_image1[y,0],corner_image1[y,1],corner_image2[x,0],corner_im
age2[x,1]])

    return np.array(valid_correspondences)
def Get_window(image,kernel_size,x,y):
    '''
    Function for finding the maximum inside a kernel.
```

```
Requires the image, kernelsize and current image coordinates
to define the current region occupied by kernel
'''
# Current kernel centered at x and y
Window = image[y-kernel_size : y + kernel_size+1, x-kernel_size : x + kernel_size+1]
#max_val = np.max(Window)
return Window
```