COMPUTER VISION
ECE:66100

Homework Assignment 9

Name: Amruthavarshini Talikoti
Email ID: atalikot@purdue.edu
Student ID: 0029949176
Purdue University
Department of Electrical and Computer Engineering

Introduction

The aim of the assignment is to reconstruct a 3D scene using stereo images. The reconstruction is projective and hence has projective, affine and similarity distortions. The math involved is briefed below.

Estimating the Fundamental Matrix

Initially 8 pairs of corresponding points are chosen from the two stereo images by manually clicking on the pixels. These points are further normalized to center them around the mean of 0 and standard deviation of $\sqrt{2}$.This is done by multiplying the points using a transformation matrix T defined as :

$$T = \begin{bmatrix} scale & 0 & x \\ 0 & scale & y \\ 0 & 0 & 1 \end{bmatrix}$$

wherein

$$scale = \frac{\sqrt{2}}{mean.of.variances.of.distance.between.the.chosen.points}$$
$$x = -scale * (mean.of.x.coordinates)$$
$$y = -scale * (mean.of.y.coordinates)$$

The fundamental matrix is obtained by solving for linear least squares and using the last eigen vector column of the SVD of matrix A in the equation below.

$$Af = 0$$

. Here matrix is constructed by stacking the equations for the correspondences as below wherein i stands for the $i-th$ correspondence.

$$A = \begin{bmatrix} x'_i x_i & x'_i y_i & x'_i & y'_i x_i & y'_i y_i & x_i & y_i & 1 \end{bmatrix}$$

where

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

$UDV^T = A$. To condition the matrix F to make it of rank 2, the last eigen value of the D matrix is set to zero. Let this new diagonal matrix be D'. The final F is obtained as the matrix $F = UD'V^T$.
Let the transformation matrices for the two stereo images be T_1 and T_2. The final estimate for F is evaluated then as

$$F = T_2^T F T_1$$

Further normalize the F matrix. The elements of the matrix must be refined using non linear least squares estimation, Levenberg Marquardt. For this the epipoles e_1 and e_2 are determined as the basis of the right and left null spaces of the matrix F. And the projection matrices for the two cameras, P_1 and P_2 are constructed as follows,

$$P1 = [I_{3X3}|0]$$
$$P2 = [[e_2]_x F|e_2]$$

The world points are evaluated by solving the below system of equations using linear least squares.

$$A = \begin{bmatrix} x_i P1^{3^T} - P1^{1^T} \\ y_i P1^{3^T} - P1^{2^T} \\ x'_i P2^{3^T} - P2^{1^T} \\ y'_i P2^{3^T} - P2^{2^T} \end{bmatrix}$$

where

$$P1 = \begin{bmatrix} P1^{1^T} \\ P1^{2^T} \\ P1^{3^T} \end{bmatrix} P2 = \begin{bmatrix} P2^{1^T} \\ P2^{2^T} \\ P2^{3^T} \end{bmatrix}$$

The error function is defined as below. LM tries to minimize the distance between the two measured image points (on the two images) and the two projected point estimates obtained using the Projection matrices P1 and P2.

$$d^2_{geom} = \sum(||x_i - xP1_i||^2 + ||x'_i - xP2_i||^2)$$

wherein $xP1_i$ and $xP2_i$ are got by multiplying respective P1 and P2 with the world point. This will ultimately give the F matrix refined using LM non-linear least squares optimization.

Image Rectification

Images must be rectified so that the search for a point in one image becomes easier as its corresponding point in the other image is found in the same row as the first. By doing so, the epipoles are sent to infinity. Firstly second image is rotated using T_1 matrix. The rotation natrix R is used to send epipole to x-axis. The epipole is then sent to infinity along x-axis using G matrix. T_2 matrix brings the epipoles back to center. Thus, the homography matrix of second image is given by

$$H = T_2 GRT_1$$

The homography for first image is found by using a linear least squares minimization to minimize sum of squares of distances given by

$$min_{H_1} \sum d(H_1 x_i, Hx'_i)$$

This is equivalent to minimizing

$$\sum (ax_i + by_i + c - x'_i)^2$$

Performing linear least squares optimization, the rectified F matrix and epipoles are obtained. The procedure for implementation was followed as per the reference
$(https://web.stanford.edu/class/cs231a/course\_notes/03-epipolar-geometry.pdf)$

SIFT and Canny Edge Detection

SIFT operators as explained in the previous assignments is used to find correspondences by thresholding using the Sum of Squared Distances (SSD) metric.
For the canny operator, inbuilt functions are used to detect the canny edges. Then the images are searched for in the surrounding rows and in a window of determined size (chosen as 7 for this assignment). The SSD or equivalently the euclidean norm is used to threshold the distance between points for them to be chosen as correspondences.
These correspondences are used in place of the original world points and the entire procedure is repeated again to find the better estimates of F, P1, P2, e1 and e2. Finally these parameters

are used to project the boundary points of the box considered onto the 3D world coordinates and the points are connected to give a final reconstruction. This final step was carried out in MATLAB due to the ease and familiarity of 3D plots in the environment.

Observations

Since the procedure followed does not involve complicated RANSAC rejections, the results obtained are average. Also, some SIFT correspondences are with points outside object of interest. Hence reconstruction may have outliers. This can be further refined. The rank 2 of matrix must be maintained throughout, else the algorithm breaks. The canny reconstruction seems to perform quite well. The manual selection of points is always subjected to human error and results can be better with automation.

Improvements in LM Parameters

Before refinement:
The F matrix:

$$F = \begin{bmatrix} 1.22071449e-05 & -1.06867441e-01 & 4.67160705e-01 \\ 1.06854233e-01 & 5.09060196e-06 & -5.19963727e-01 \\ -4.67154065e-01 & 5.19929995e-01 & 9.80935912e-06 \end{bmatrix}$$

The epipole on Image 1:

$$e1 = \begin{bmatrix} -0.7008923 \\ -0.713266 \\ -0.0012382 \end{bmatrix}$$

The epipole on Image 2:

$$e2 = \begin{bmatrix} 0.70090266 \\ 0.71325586 \\ 0.00123825 \end{bmatrix}$$

The Projection matrix for Image 1:

$$P1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The Projection matrix for Image 2:

$$P2 = \begin{bmatrix} -1.53766941e+02 & 1.51097715e+02 & 9.75568310e-01 & 7.00902661e-01 \\ 1.51103326e+02 & -1.48481248e+02 & -4.33969222e-01 & 7.13255864e-01 \\ 2.62262784e-01 & 2.66961607e-01 & -3.02239675e+02 & 1.23824687e-03 \end{bmatrix}$$

After Refinement:

The F matrix:

$$F = \begin{bmatrix} -6.90931561e-05 & 7.39920345e-02 & -4.30783390e+01 \\ -7.39720920e-02 & -1.18898651e-05 & 4.21561415e+01 \\ 4.30889553e+01 & -4.21614112e+01 & 1.00000000e+00 \end{bmatrix}$$

The epipole on Image 1:

$$e1 = \begin{bmatrix} 569.98712123 \\ 582.55046375 \\ 1. \end{bmatrix}$$

The epipole on Image 2:

$$e2 = \begin{bmatrix} 569.71557969 \\ 582.15481847 \\ 1. \end{bmatrix}$$

The Projection matrix for Image 1:

$$P1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The Projection matrix for Image 2:

$$P2 = \begin{bmatrix} 2.50845169e+04 & -2.45444687e+04 & 5.39998677e+02 & 5.69715580e+02 \\ -2.45484492e+04 & 2.40200868e+04 & -6.12793919e+02 & 5.82154818e+02 \\ -4.21028304e+01 & -4.30815933e+01 & 4.90952732e+04 & 1.00000000e+00 \end{bmatrix}$$

Results

Input Images

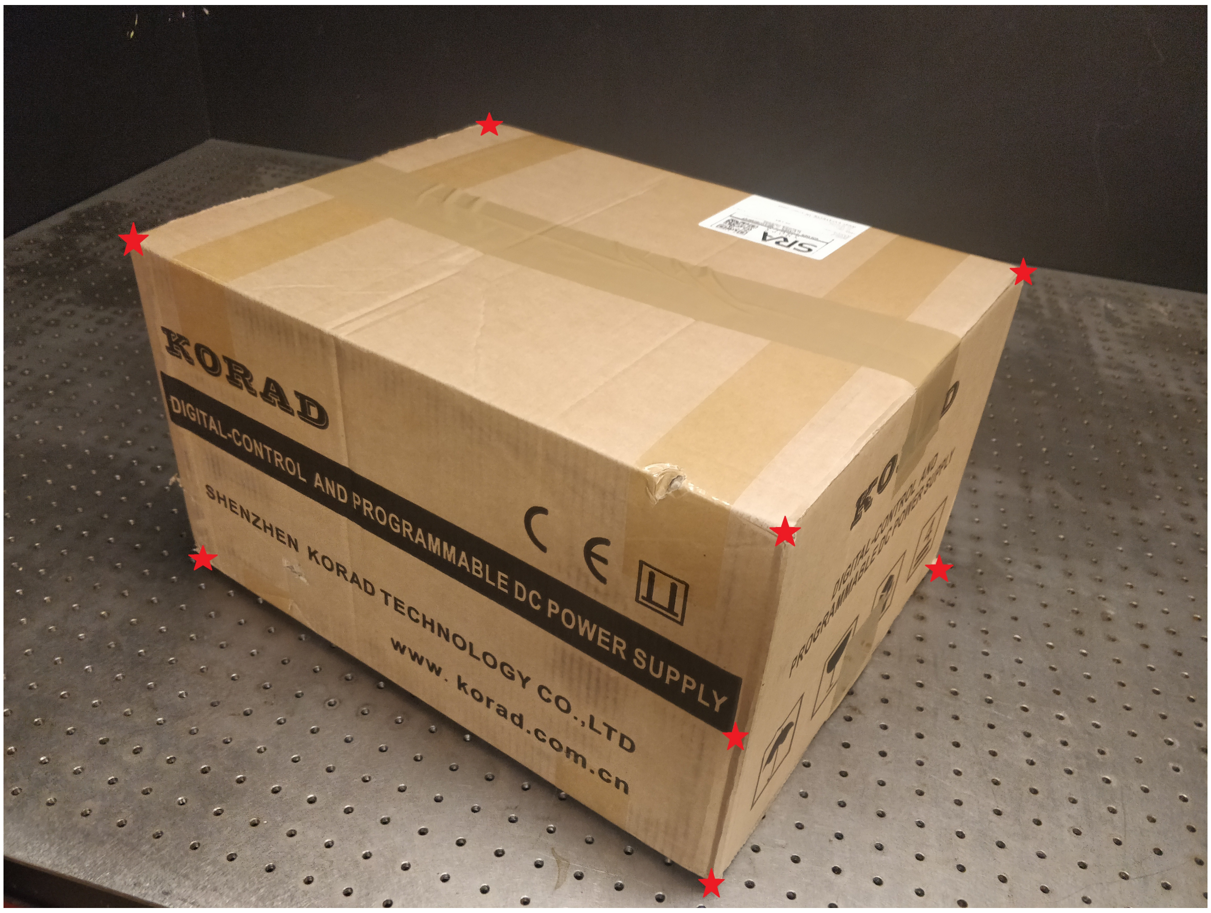
KORAD
DIGITAL-CONTROL AND PROGRAMMABLE DC POWER SUPPLY
SHENZHEN KORAD TECHNOLOGY CO.,LTD
www.korad.com.cn
CE

Figure 1: Input Image 1 with selected points

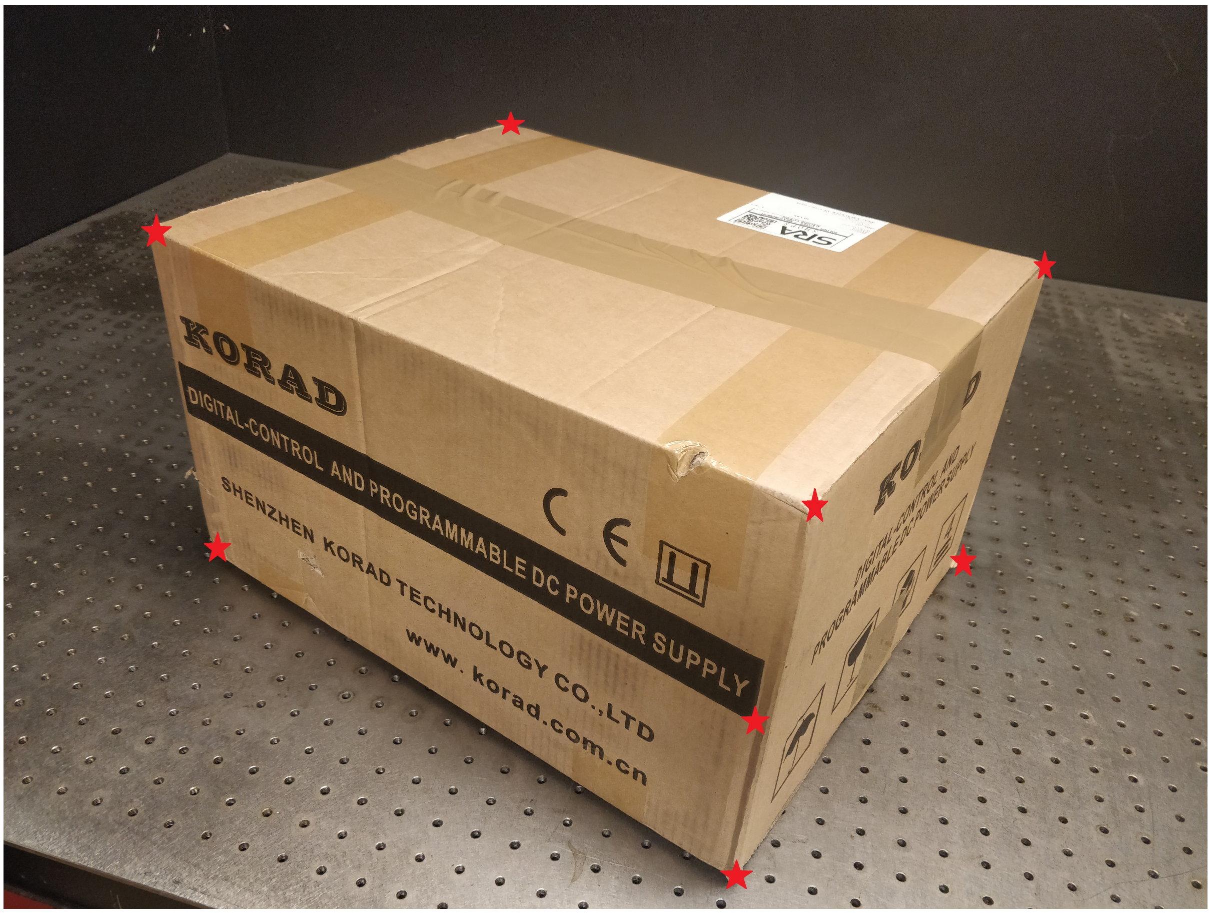
KORAD
DIGITAL-CONTROL AND PROGRAMMABLE DC POWER SUPPLY
SHENZHEN KORAD TECHNOLOGY CO.,LTD
www. korad.com.cn

Figure 2: Input Image 2 with selected points

Rectified Images

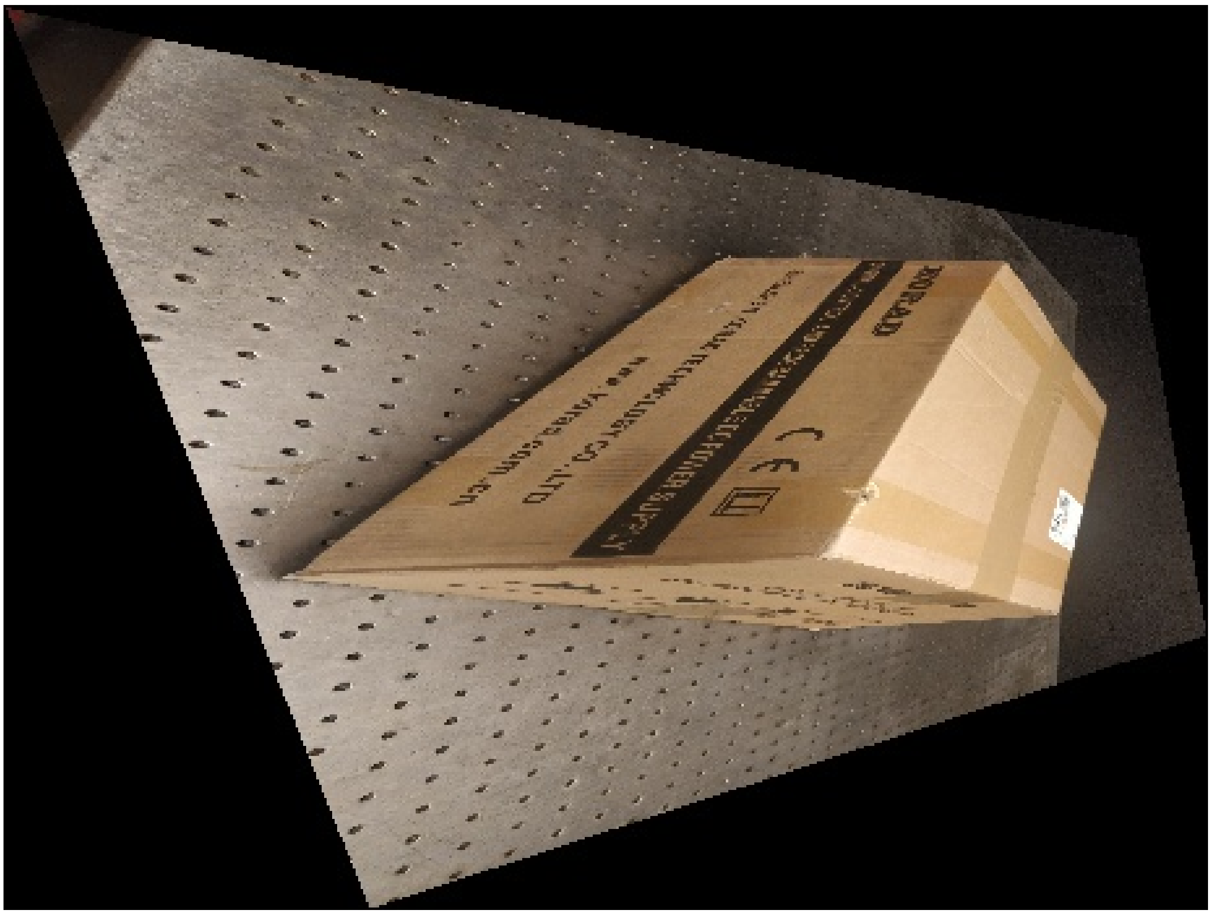

Figure 3: Rectified Image 1

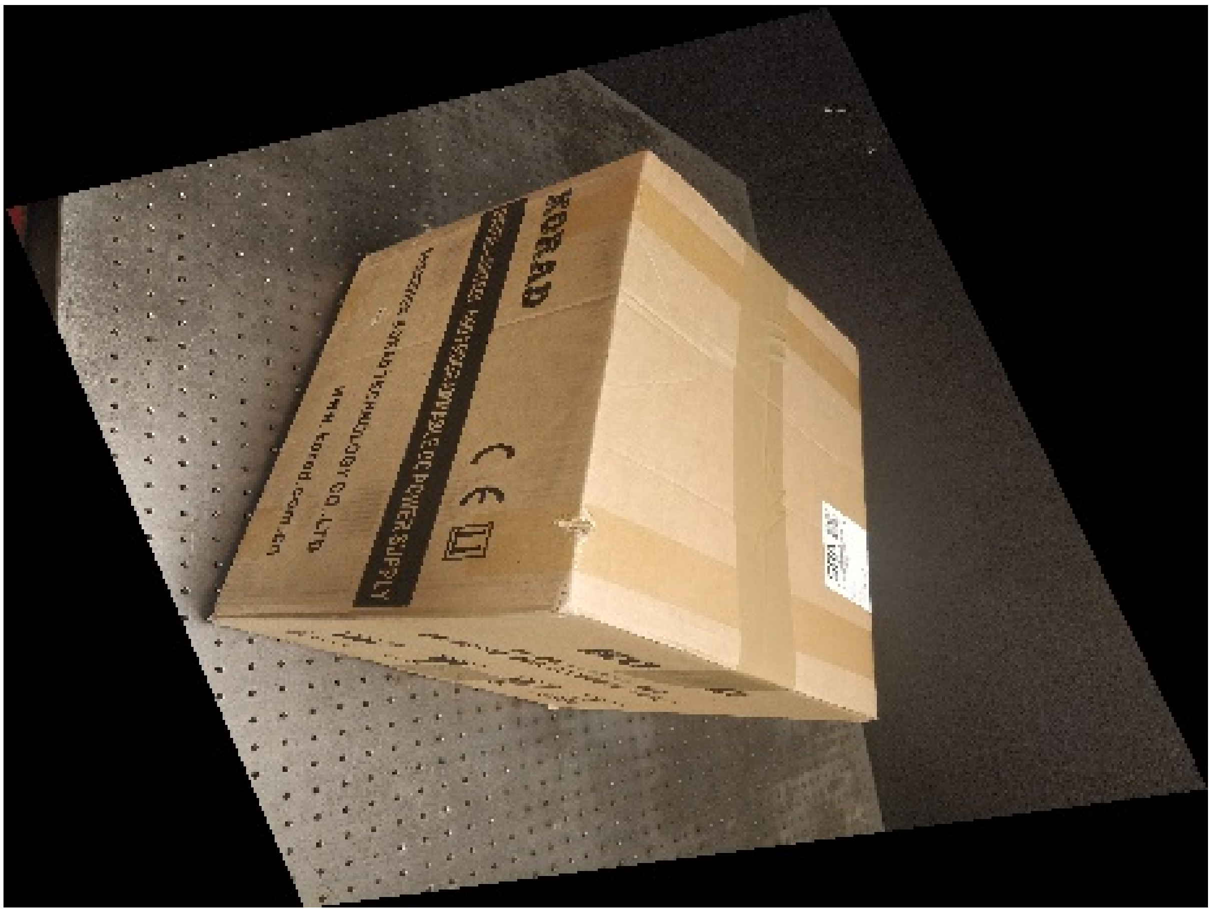

Figure 4: Rectified Image 2

SIFT Correspondences

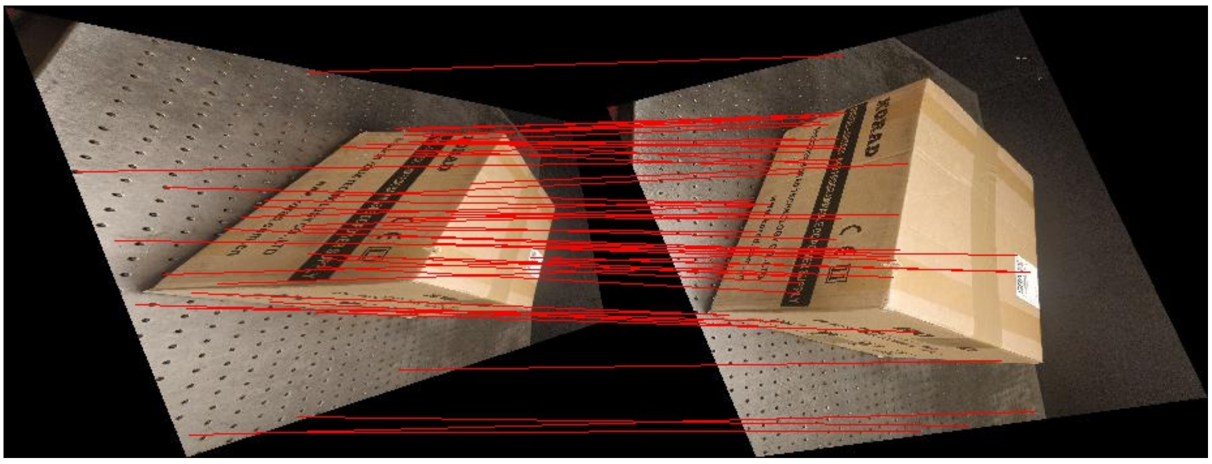

Figure 5: Selected SIFT Correspondences between Image 1 and 2

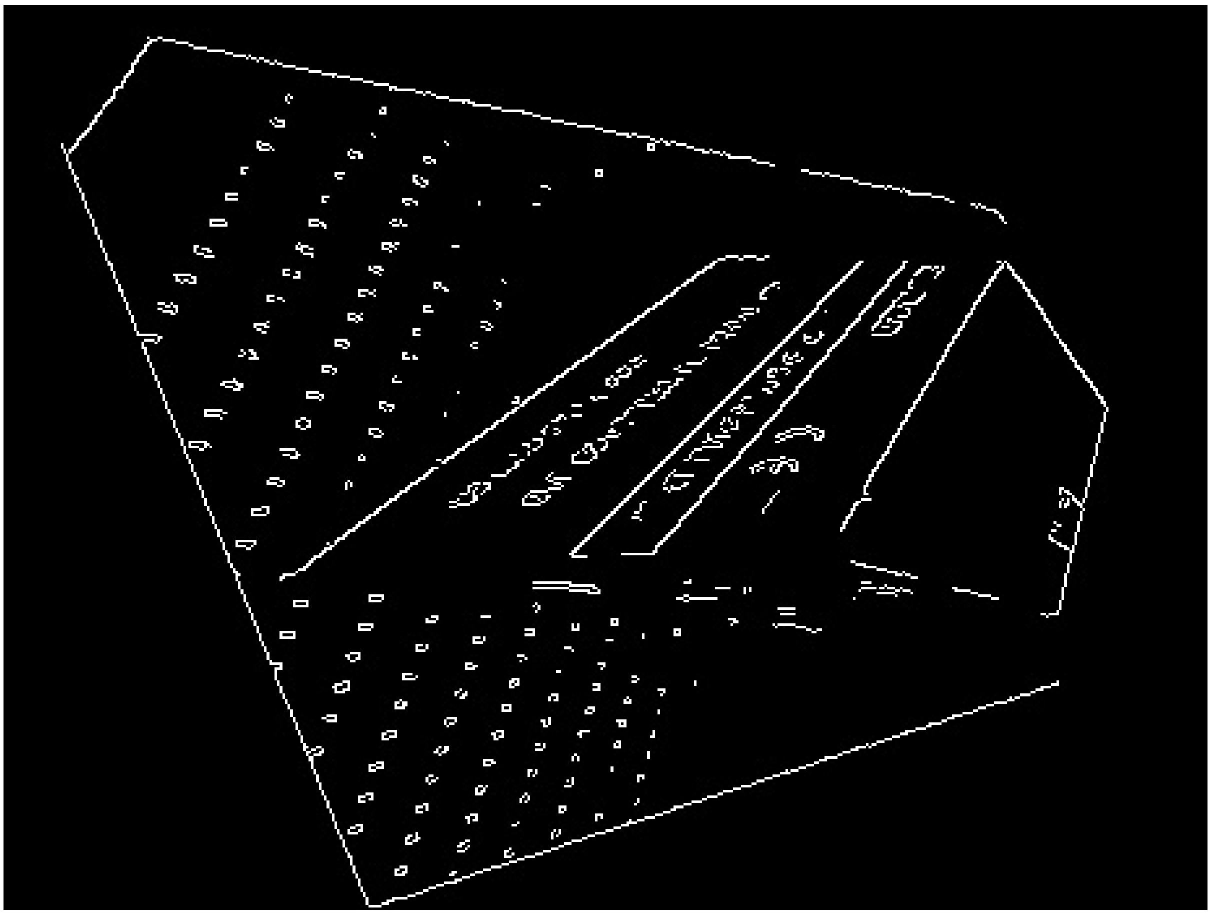

Figure 6: Selected Canny Edged for Image 1

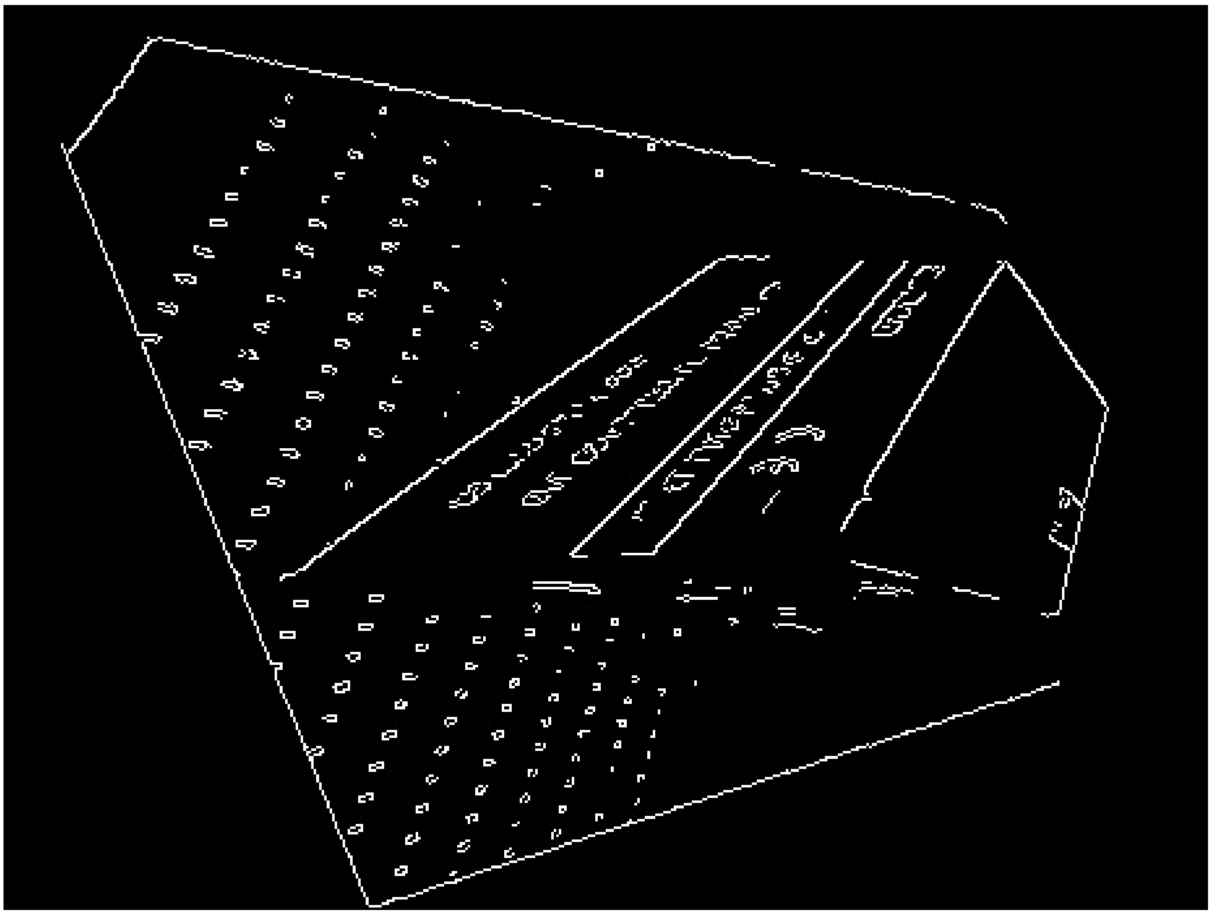

Figure 7: Selected Canny Edged for Image 2

Final 3D Reconstructions

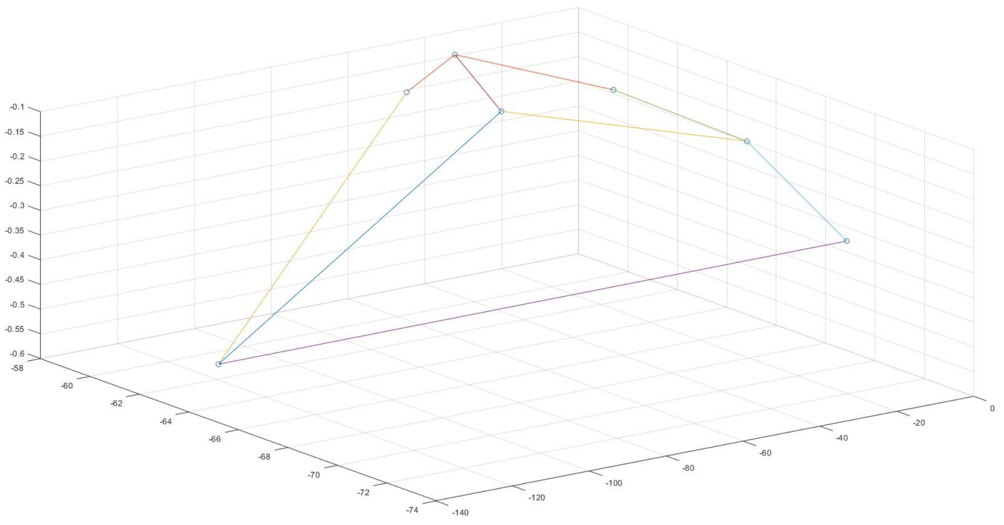

Figure 8: Reconstruction with SIFT with LM optimization

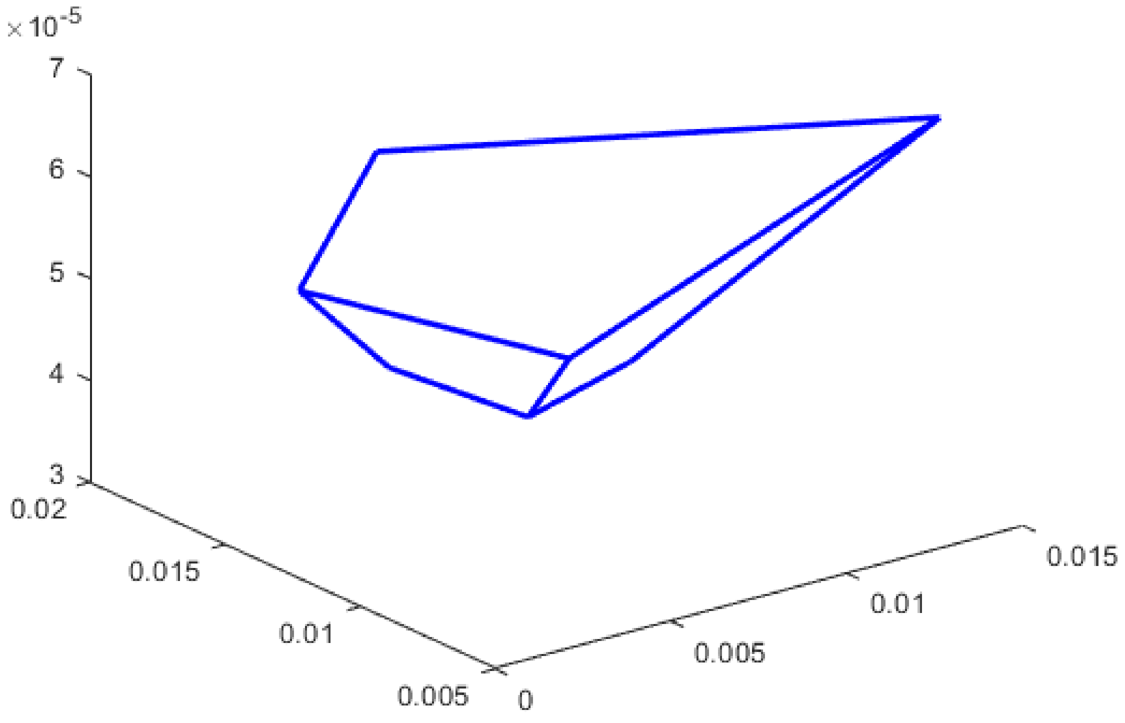

Figure 9: Reconstruction with Canny with LM optimization

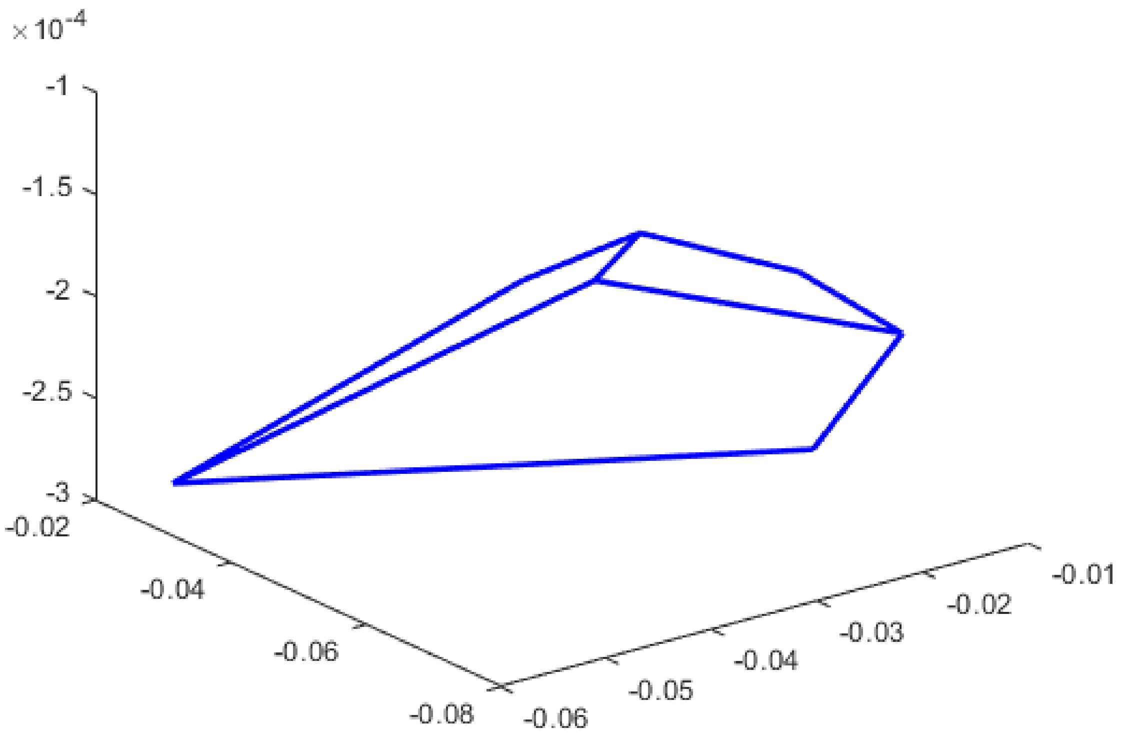

Figure 10: Reconstruction with SIFT without LM optimization

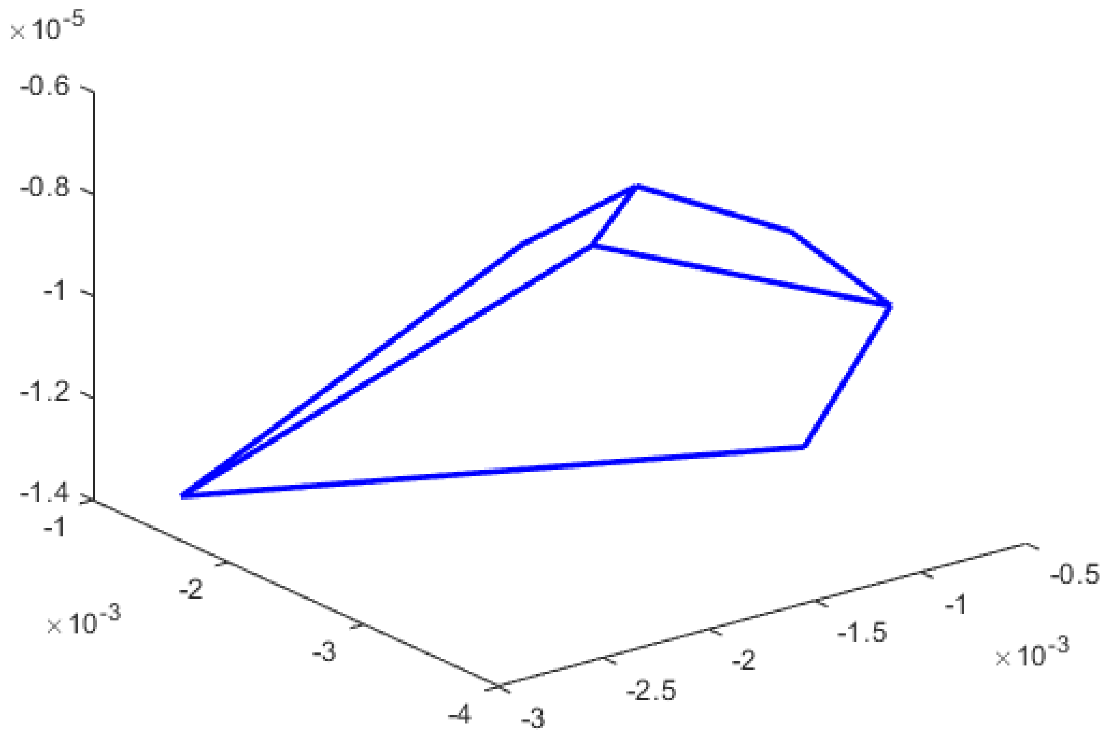

Figure 11: Reconstruction with Canny without LM optimization

3D Visual Inspection

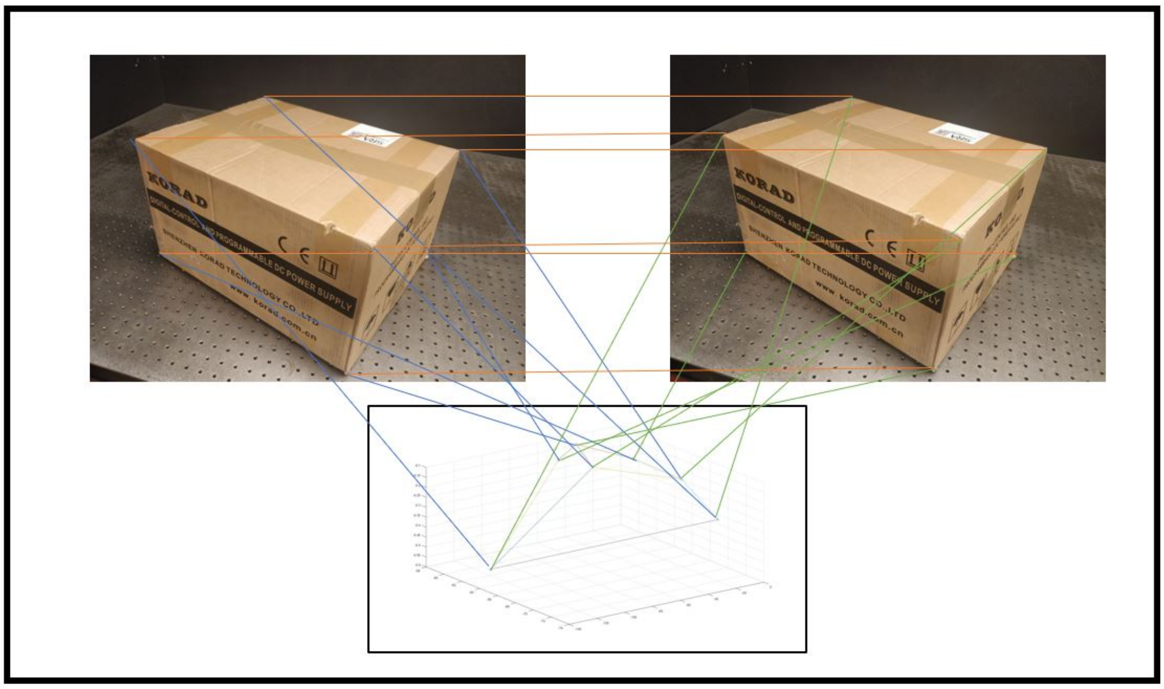

Figure 12: Reconstruction with Canny without LM optimization

Source Code

For better clarity please refer to attached code as reference.

```
import math
import numpy as np
import cv2 as cv
from scipy.optimize import least_squares
from PIL import Image
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Canny Edge Dectector
def do_canny():
    img1_c=cv.imread('Rectified_image_H1.jpg')
    img1_g=cv.cvtColor(img1_c,cv.COLOR_BGR2GRAY)
    img2_c=cv.imread('Rectified_image_H2.jpg')
    img2_g=cv.cvtColor(img2_c,cv.COLOR_BGR2GRAY)
    e1=cv.Canny(img1_g,255*1.5,255)
    e2=cv.Canny(img2_g,255*1.5,255)
    cv.imwrite('canny1.jpg',e1)
    cv.imwrite('canny2.jpg',e2)
    corr_list=list()
    for x in range(5,np.shape(e1)[0]-5):
        for y in range(5,np.shape(e1)[1]-5):
            if e1[x,y]>0:
                largg=10e8
                for pixel in range(5,np.shape(e2)[1]-5):
                    if (np.linalg.norm(img1_g[x-3:x+4,y-3:y+4]-img2_g[x-3:x+4,pixel
                        temp=pixel
                        largg=np.linalg.norm(img1_g[x-3:x+4,y-3:y+4]-img2_g[x-3:x+4
                corr_list.append([np.asarray([x,y]),np.asarray([x,temp])])
    return corr_list
# Function code to connect the corresponding points between images
def connect_points(corr_list,comb1,img1_c):
    for i in range(len(corr_list)):
        var_x=corr_list[i][1][1]+img1_c.shape[1]
        cv.line(comb1,(corr_list[i][0][1],corr_list[i][0][0]),(var_x,corr_list[i][1
    return comb1
# Function code to combine images horizontally
def combine_images(imges):

    images = map(Image.open, imges)
    widths, heights = zip(*(i.size for i in images))

    total_width = sum(widths)
    max_height = max(heights)

    new_im = Image.new('RGB', (total_width, max_height))

    x_offset = 0
    for im in images:
      new_im.paste(im, (x_offset,0))
```

```
            x_offset += im.size[0]

        new_im.save('comb12.jpg')
# Function that determines the euclidean distance between each pair
# of points in desc1 and desc2
def calc_eucl(desc1,desc2):
    dist_matrix=np.zeros((len(desc1),len(desc2)))
    for i in range(len(desc1)):
        for j in range(len(desc2)):
            dist_matrix[i][j]=np.linalg.norm(desc1[i]-desc2[j])
    return dist_matrix
# Choosing the best points pairs, the euclidean distance of which
# is lesser than the threshold set
def choose_best_points(dist_matrix,kp1,kp2):
    thr=200
    print np.min(dist_matrix)
    print dist_matrix
    cp1=list()
    cp2=list()
    j_done=list()
    for i in range(np.shape(dist_matrix)[0]):
        x=np.min(dist_matrix[i])
        j=np.argmin(dist_matrix[i])
        #print j
        if ((x<thr)and(not(j in j_done))):
            if abs(kp1[i].pt[1]-kp2[j].pt[1])<20:
                cp1.append([kp1[i].pt[1],kp1[i].pt[0]])
                cp2.append([kp2[j].pt[1],kp2[j].pt[0]])
                j_done.append(j)
    return cp1,cp2
# Performing SIFT and extracting best set of correspondences
def do_sift():
    # Loading the images
    img1_c=cv.imread('Rectified_image_H1.jpg')
    img1_g=cv.cvtColor(img1_c,cv.COLOR_BGR2GRAY)
    img2_c=cv.imread('Rectified_image_H2.jpg')
    img2_g=cv.cvtColor(img2_c,cv.COLOR_BGR2GRAY)

    # Performing SIFT using opencv inbuilt functions
    sift=cv.xfeatures2d.SIFT_create()
    kp1,desc1=sift.detectAndCompute(img1_g,None)
    kp2,desc2=sift.detectAndCompute(img2_g,None)

    # Code that calls function to evaluate euclidean distance each
    # respective row of  desc1, desc2
    dist_matrix=calc_eucl(desc1,desc2)
    #print dist_matrix
    chosen_points1,chosen_points2=choose_best_points(dist_matrix,kp1,kp2)
    # To combine images and read it
    combine_images(['Rectified_image_H1.jpg', 'Rectified_image_H2.jpg'])
    comb1=cv.imread('comb12.jpg')
    # To construct the corr_list
    corr_list=list()
    for i in range(len(chosen_points1)):
        corr_list.append([np.asarray(chosen_points1[i]).astype('int'),np.asarray(ch

    comb1_sift=connect_points(corr_list,comb1,img1_c)
    cv.imwrite('comb12_sift.jpg',comb1_sift)
    return corr_list
# Function to use homographies to rectify the images
def rectified_images(H1,bound_points):
    H=H1
    # Image input.
    i1=cv.imread('1.jpg')
    i2=cv.imread('2.jpg')
    dist_img=i1
    D=list()
    D.append((0.0,0.0,1.0))
    D.append((np.shape(i1)[1],0.0,1.0))
    D.append((0.0,np.shape(i1)[0],1.0))
```

```
D.append((np.shape(i1)[1],np.shape(i1)[0],1.0))

# Finding the coordinates of the undistorted image required using transform ma
U=[0,0,0,0]
for i in range(len(D)):
    U[i]=np.matmul(H,D[i])
    U[i][0]=U[i][0]/U[i][2]
    U[i][1]=U[i][1]/U[i][2]
    U[i][2]=U[i][2]/U[i][2]

# Finding the boundary box of undistorted image required using max of coordinat
x_l=int(np.floor(min(U[0][0],U[1][0],U[2][0],U[3][0])))
x_r=int(np.ceil(max(U[0][0],U[1][0],U[2][0],U[3][0])))
y_u=int(np.floor(min(U[0][1],U[1][1],U[2][1],U[3][1])))
y_d=int(np.ceil(max(U[0][1],U[1][1],U[2][1],U[3][1])))
# Scaling
H_s=np.eye((3))
H_s[0][0]=float(np.shape(i1)[1])/(x_r-x_l)/10
H_s[1][1]=float(np.shape(i1)[0])/(y_d-y_u)/10
H=np.matmul(H_s,H)

D=list()
D.append((0.0,0.0,1.0))
D.append((np.shape(i1)[1],0.0,1.0))
D.append((0.0,np.shape(i1)[0],1.0))
D.append((np.shape(i1)[1],np.shape(i1)[0],1.0))

# Finding the coordinates of the undistorted image required using transform ma
U=[0,0,0,0]
for i in range(len(D)):
    U[i]=np.matmul(H,D[i])
    U[i][0]=U[i][0]/U[i][2]
    U[i][1]=U[i][1]/U[i][2]
    U[i][2]=U[i][2]/U[i][2]
# Finding the boundary box of undistorted image required using max of coordinat
```

```
x_l=int(np.floor(min(U[0][0],U[1][0],U[2][0],U[3][0])))
x_r=int(np.ceil(max(U[0][0],U[1][0],U[2][0],U[3][0])))
y_u=int(np.floor(min(U[0][1],U[1][1],U[2][1],U[3][1])))
y_d=int(np.ceil(max(U[0][1],U[1][1],U[2][1],U[3][1])))

# Traversing through image plane and finding corresponding pixel in distorted i
# Taking weighted value of pixel from distorted image
H_inv=np.linalg.pinv(H)
dist_rmv=np.zeros((y_d-y_u,x_r-x_l,3))
for x in range(dist_rmv.shape[0]):
    for y in range(dist_rmv.shape[1]):
        f_coord=(float(y+x_l),float(x+y_u),1.0)
        p_coord=np.matmul(H_inv,f_coord)
        factor=p_coord[2]
        p_coord=p_coord/factor
        if p_coord[1]<0 or p_coord[1]>(dist_img.shape[0]-1) or p_coord[0]<0 or
            dist_rmv[x][y]=[0.0,0.0,0.0]
        else:
            n1=dist_img[int(np.floor(p_coord[1]))][int(np.floor(p_coord[0]))]
            n2=dist_img[int(np.floor(p_coord[1]))][int(np.ceil(p_coord[0]))]
            n3=dist_img[int(np.ceil(p_coord[1]))][int(np.ceil(p_coord[0]))]
            n4=dist_img[int(np.ceil(p_coord[1]))][int(np.floor(p_coord[0]))]
            w1=1/np.linalg.norm(n1-p_coord)
            w2=1/np.linalg.norm(n2-p_coord)
            w3=1/np.linalg.norm(n3-p_coord)
            w4=1/np.linalg.norm(n4-p_coord)
            dist_rmv[x][y]=((w1*n1)+(w2*n2)+(w3*n3)+(w4*n4))/(w1+w2+w3+w4)

#cv.imwrite("distrmv1.jpg",dist_rmv)
#cv.imwrite("distrmv2.jpg",dist_rmv)
#cv.imwrite("distrmv3.jpg",dist_rmv)
cv.imwrite("Rectified_image_H2.jpg",dist_rmv)

boundaries=np.zeros((7,3))
for j in range(0,len(boundaries)):
```

```
            boundaries[j]=np.matmul(H,bound_points[j])
            boundaries[j]=boundaries[j]/boundaries[j][2]
            boundaries[j][0]=boundaries[j][0]-x_l
            boundaries[j][1]=boundaries[j][1]-y_u
    return boundaries
# Function to compute homographies from F that rectify images and
# send the epipoles to infinity
def rectify_images(F,e1,e2,P1,P2):
    global Img1
    global Img2
    i1=cv.imread('1.jpg')
    i2=cv.imread('2.jpg')
    height=np.shape(i2)[0]
    width=np.shape(i2)[1]
    # Angle of epipole wrt x axis for rotating image
    theta=np.arctan((e2[1]-height/2)/(width/2-e2[0]))
    focal=np.cos(theta)*(e2[0]-width/2)-np.sin(theta)*(e2[1]-height/2)
    G=np.zeros((3,3))
    G[0][0]=1.0
    G[1][1]=1.0
    G[2][2]=1.0
    G[2][0]=-1/focal
    # Rotation Matrix to rotate the image so that epipole goes to x axis
    R=np.zeros((3,3))
    R[0][0]=np.cos(theta)
    R[0][1]=-np.sin(theta)
    R[1][0]=np.sin(theta)
    R[1][1]=np.cos(theta)
    R[2][2]=1.0
    # Homography to translate the second image center to origin
    T=np.zeros((3,3))
    T[0][0]=1.0
    T[0][2]=-width/2
    T[1][1]=1.0
    T[1][2]=-height/2
```

```
T[2][2]=1.0
H2 = np.matmul(np.matmul(G,R),T)
image_center = np.asarray([width/2,height/2,1.0])
center_rectified = np.matmul(H2,image_center)
center_rectified = center_rectified/center_rectified[2]
# Homography to translate second image center back to original center
T2=np.zeros((3,3))
T2[0][0]=1.0
T2[0][2]=width/2-center_rectified[0]
T2[1][1]=1.0
T2[1][2]=height/2-center_rectified[1]
T2[2][2]=1.0
# Overall Homography for rectification of second image
H2 = np.matmul(T2,H2)
# Finding the overall Homography for rectification of first image
# Find M
Ex=np.zeros((3,3))
#print e2
Ex[0][1]=-e2[2]
Ex[0][2]=e2[1]
Ex[1][0]=e2[2]
Ex[1][2]=-e2[0]
Ex[2][0]=-e2[1]
Ex[2][1]=e2[0]
E=np.zeros((3,3))
E[:,0]=e2
E[:,1]=e2
E[:,2]=e2
F=F/F[2][2]
M=np.matmul(Ex,F)+E
#M = np.matmul(P2,np.linalg.pinv(P1))
# Find H0
H0 = np.matmul(H2,M)
#print(H0)
# Project the manual correspondences and find pixel coordinates
```

```
img1_new=np.zeros((len(Img1),3))
img2_new=np.zeros((len(Img2),3))
for i in range(len(Img1)):
    img1_new[i]=np.matmul(H0,Img1[i])
    img2_new[i]=np.matmul(H2,Img2[i])
    img1_new[i]=img1_new[i]/img1_new[i][2]
    img2_new[i]=img2_new[i]/img2_new[i][2]
# Solve for a,b,c for H_A using Linear Least Squares that minimizes
# function
A = img1_new
b = img2_new[:,0]
#print('A,b')
#print(A)
#print(b)
x = np.matmul(np.linalg.pinv(A),b)
#print (x)
HA=np.zeros((3,3))
HA[0][0]=x[0]
HA[0][1]=x[1]
HA[0][2]=x[2]
HA[1][1]=1.0
HA[2][2]=1.0
H1 = np.matmul(HA,H0)
center_rectified = np.matmul(H1,image_center)
center_rectified = center_rectified/center_rectified[2]
# Homography to translate first image center back to original center
T1=np.zeros((3,3))
T1[0][0]=1.0
T1[0][2]=width/2-center_rectified[0]
T1[1][1]=1.0
T1[1][2]=height/2-center_rectified[1]
T1[2][2]=1.0
# Overall Homography for rectification of first image
H1 = np.matmul(T1,H1)
F_rectified = np.matmul(np.matmul(np.linalg.pinv(np.transpose(H2)),F),np.linalg
    # Find the pixel coordinates rectified manually selected points
    x1_rectified=np.zeros((len(Img1),3))
    x2_rectified=np.zeros((len(Img2),3))
    for i in range(len(Img1)):
        x1_rectified[i]=np.matmul(H1,Img1[i])
        x2_rectified[i]=np.matmul(H2,Img2[i])
        x1_rectified[i]=img1_new[i]/img1_new[i][2]
        x2_rectified[i]=img2_new[i]/img2_new[i][2]
    # Finding new epipoles
    # Evaluating e1 and e2
    u, d, vt = np.linalg.svd(F_rectified)
    e1_rectified=np.transpose(vt)[:,-1]
    e2_rectified=u[:,-1]
    return [x1_rectified,x2_rectified,H1,H2,F_rectified, e1_rectified,e2_rectified]
# Cost Function for Non linear square optimization
def cost(FF):
    F=np.zeros((3,3))
    F[0][0]=FF[0]
    F[0][1]=FF[1]
    F[0][2]=FF[2]
    F[1][0]=FF[3]
    F[1][1]=FF[4]
    F[1][2]=FF[5]
    F[2][0]=FF[6]
    F[2][1]=FF[7]
    F[2][2]=FF[8]
    global Img1
    global Img2
    # Evaluating e1 and e2
    u, d, vt = np.linalg.svd(F)
    e1=np.transpose(vt)[:,-1]
    e2=u[:,-1]
    # Evaluating P1 and P2 (proj matrices) for two cameras
    Ex=np.zeros((3,3))
    Ex[0][1]=-e2[2]
    Ex[0][2]=e2[1]
    Ex[1][0]=e2[2]
    Ex[1][2]=-e2[0]
    Ex[2][0]=-e2[1]
    Ex[2][1]=e2[0]
    P1=np.zeros((3,4))
    P1[0][0]=1.0
    P1[1][1]=1.0
    P1[2][2]=1.0
    P2=np.zeros((3,4))
    P2[:,0:3]=np.matmul(Ex,F)
    P2[:,3]=e2
    # Doing non linear square optimization. Finding the X_world
    X_world=np.zeros((len(Img1),4))
    cost_array=list()
    for i in range(len(Img1)):
        A=np.zeros((4,4))
        A[0]=Img1[i][0]*P1[2,:]-P1[0,:]
        A[1]=Img1[i][1]*P1[2,:]-P1[1,:]
        A[2]=Img2[i][0]*P2[2,:]-P2[0,:]
        A[3]=Img2[i][1]*P2[2,:]-P2[1,:]
        u, d, vt = np.linalg.svd(A)
        X_world[i]=np.transpose(vt)[:,-1]
        X_world[i]=X_world[i]/np.linalg.norm(X_world[i])
        pred1=np.matmul(P1,X_world[i])
        pred1=pred1/pred1[2]
        pred2=np.matmul(P2,X_world[i])
        pred2=pred2/pred2[2]
        cost_array.append(np.square(np.linalg.norm(Img1[i]-pred1)))
        cost_array.append(np.square(np.linalg.norm(Img2[i]-pred2)))
    return np.asarray(cost_array)
# Normalize function to calculate transformation matrix T
def normalize_T(Img):
    x_coord=list()
    y_coord=list()
    for i in range(len(Img)):
        x_coord.append(Img[i][0])
        y_coord.append(Img[i][1])
    mean_x=np.mean(x_coord)
    mean_y=np.mean(y_coord)
    dist_x=(x_coord-mean_x)*(x_coord-mean_x)
    dist_y=(y_coord-mean_y)*(y_coord-mean_y)
    var_xy=np.sqrt(dist_x+dist_y)
    mean_dist=np.mean(var_xy)
    scale=math.pow(2,0.5)/mean_dist
    x=-scale*mean_x
    y=-scale*mean_y
    T=np.zeros((3,3))
    T[0][0]=scale
    T[0][1]=0.0
    T[0][2]=x
    T[1][0]=0.0
    T[1][1]=scale
    T[1][2]=y
    T[2][0]=0.0
    T[2][1]=0.0
    T[2][2]=1.0
    return T

# Function to compute Fundamental matrix F by linear least squares
def linear_least_squares(img1,img2):
    A=np.zeros((len(img1),9))
    for i in range(len(img1)):
        A[i][0]=img2[i][0]*img1[i][0]
        A[i][1]=img2[i][0]*img1[i][1]
        A[i][2]=img2[i][0]
        A[i][3]=img2[i][1]*img1[i][0]
        A[i][4]=img2[i][1]*img1[i][1]
        A[i][5]=img2[i][1]
        A[i][6]=img1[i][0]
```

```
            A[i][7]=img1[i][1]
            A[i][8]=1.0
        # Computing SVD of A
        u, d, vt = np.linalg.svd(A)
        # Choosing the last column (smallest eigen value vector )
        # of vt as solution
        F_col=np.transpose(vt)[:,-1]
        # Converting to a 3X3 structure
        F=np.zeros((3,3))
        F[0][0]=F_col[0]
        F[0][1]=F_col[1]
        F[0][2]=F_col[2]
        F[1][0]=F_col[3]
        F[1][1]=F_col[4]
        F[1][2]=F_col[5]
        F[2][0]=F_col[6]
        F[2][1]=F_col[7]
        F[2][2]=F_col[8]
        # Conditioning F to make it rank 2 matrix
        # Finding SVD of F
        u, d, vt = np.linalg.svd(F)
        d[2]=0.0
        # Convert d to a 3X3 diagonal matrix
        D=np.zeros((3,3))
        D[0][0]=d[0]
        D[1][1]=d[1]
        F=np.matmul(np.matmul(u,D),vt)
        return F

# Manual Selection of Image Coordinates
global Img1
Img1=list()
Img1.append([1859.0,447.0,1.0])
Img1.append([3905.0,1011.0,1.0])
Img1.append([481.0,897.0,1.0])
```

```
Img1.append([3001.0,2033.0,1.0])
Img1.append([737.0,2093.0,1.0])
Img1.append([2701.0,3341.0,1.0])
Img1.append([3567.0,2133.0,1.0])
#Img1.append([2073.0,1937.0,1.0])
Img1.append([2777.0,2785.0,1.0])
Img1_old=np.asarray(Img1).copy()
# Normalizing x and y coordinates for image 1
T1=normalize_T(Img1)
norm_img1=np.zeros((8,3))
# Normalizing the points with T matrix
for i in range(len(Img1)):
    norm_img1[i]=np.matmul(T1,np.asarray(Img1[i]))

global Img2
Img2=list()
Img2.append([1921.0,459.0,1.0])
Img2.append([3977.0,985.0,1.0])
Img2.append([557.0,853.0,1.0])
Img2.append([3093.0,1929.0,1.0])
Img2.append([793.0,2077.0,1.0])
Img2.append([2769.0,3329.0,1.0])
Img2.append([3643.0,2127.0,1.0])
#Img2.append([2153.0,1869.0,1.0])
Img2.append([2857.0,2725.0,1.0])
Img2_old=np.asarray(Img2).copy()
# Normalizing x and y coordinates for image 2
T2=normalize_T(Img2)
norm_img2=np.zeros((8,3))
# Normalizing the points with T matrix
for i in range(len(Img2)):
    norm_img2[i]=np.matmul(T2,np.asarray(Img2[i]))
# Evaluating the fundamental matrix using linear least squares
F_old=linear_least_squares(norm_img1,norm_img2)
# Denormalizing the F matrix with T1 and T2
```

```
F=np.matmul(np.matmul(np.transpose(T2),F_old),T1)

# Finding old epipoles and projection matrices before lm
# Evaluating e1 and e2
u, d, vt = np.linalg.svd(F)
e1_old=np.transpose(vt)[:,-1]
e2_old=u[:,-1]
# Evaluating P1 and P2 (proj matrices) for two cameras
Ex=np.zeros((3,3))
Ex[0][1]=-e2_old[2]
Ex[0][2]=e2_old[1]
Ex[1][0]=e2_old[2]
Ex[1][2]=-e2_old[0]
Ex[2][0]=-e2_old[1]
Ex[2][1]=e2_old[0]
P1_old=np.zeros((3,4))
P1_old[0][0]=1.0
P1_old[1][1]=1.0
P1_old[2][2]=1.0
P2_old=np.zeros((3,4))
F=F/F[2][2]
P2_old[:,0:3]=np.matmul(Ex,F)
P2_old[:,3]=e2_old

# Non linear squares optimization
f=np.zeros((9))
f[0]=F[0][0]
f[1]=F[0][1]
f[2]=F[0][2]
f[3]=F[1][0]
f[4]=F[1][1]
f[5]=F[1][2]
f[6]=F[2][0]
f[7]=F[2][1]
f[8]=F[2][2]
```

```
f=f/f[8]
F_new=least_squares(cost, f, method='lm')
#Converting to 3X3 matrix
F_newm=np.zeros((3,3))
F_newm[0][0]=F_new.x[0]
F_newm[0][1]=F_new.x[1]
F_newm[0][2]=F_new.x[2]
F_newm[1][0]=F_new.x[3]
F_newm[1][1]=F_new.x[4]
F_newm[1][2]=F_new.x[5]
F_newm[2][0]=F_new.x[6]
F_newm[2][1]=F_new.x[7]
F_newm[2][2]=F_new.x[8]
# Conditioning F_newm to make it rank 2 matrix
# Finding SVD of F
#u, d, vt = np.linalg.svd(F_newm)
#d[2]=0.0
## Convert d to a 3X3 diagonal matrix
#D=np.zeros((3,3))
#D[0][0]=d[0]
#D[1][1]=d[1]
#F_newm=np.matmul(np.matmul(u,D),vt)
# Finding new epipoles
# Evaluating e1 and e2
u, d, vt = np.linalg.svd(F_newm)
e1=np.transpose(vt)[:,-1]
e2=u[:,-1]
e1=e1/e1[2]
e2=e2/e2[2]
F_newm=F_newm/F_newm[2][2]
# Evaluating P1 and P2 (proj matrices) for two cameras
Ex=np.zeros((3,3))
Ex[0][1]=-e2[2]
Ex[0][2]=e2[1]
Ex[1][0]=e2[2]
```

```
Ex[1][2]=-e2[0]
Ex[2][0]=-e2[1]
Ex[2][1]=e2[0]
P1=np.zeros((3,4))
P1[0][0]=1.0
P1[1][1]=1.0
P1[2][2]=1.0
P2=np.zeros((3,4))
P2[:,0:3]=np.matmul(Ex,F_newm)
P2[:,3]=e2

F_int=F_newm.copy()
e1_int=e1.copy()
e2_int=e2.copy()
P1_int=P1.copy()
P2_int=P2.copy()
## Finding Homographies from F after lm
#print("F_newm")
#print(F_newm)
rectified=rectify_images(F_newm,e1,e2,P1,P2)
H1=rectified[2]/rectified[2][2][2]
H2=rectified[3]/rectified[3][2][2]
boundaries1=rectified_images(H1,Img1_old[0:7])
boundaries2=rectified_images(H2,Img2_old[0:7])
# Performing SIFT/Canny and extracting best set of correspondences
corr_list=do_sift()
#corr_list=do_canny()
############################################################################
# Repeating the entire logic for better refinement of
# F, P1, P2, e1 and e2 with the SIFT correspondences
# from the rectified images
Img1=list()
Img2=list()
for i in range(len(corr_list)):
    temp1=list((corr_list[i][0]).astype(float))
    temp1.append(1.0)
    Img1.append(temp1)

    temp2=list((corr_list[i][1]).astype(float))
    temp2.append(1.0)
    Img2.append(temp2)
for j in range(len(boundaries1)):
    Img1.append([boundaries1[j][1],boundaries1[j][0],1.0])
    Img2.append([boundaries2[j][1],boundaries2[j][0],1.0])

# Normalizing x and y coordinates for image 1
T1=normalize_T(Img1)
norm_img1=np.zeros((len(Img1),3))
# Normalizing the points with T matrix
for i in range(len(Img1)):
    norm_img1[i]=np.matmul(T1,np.asarray(Img1[i]))
    # Normalizing x and y coordinates for image 2
T2=normalize_T(Img2)
norm_img2=np.zeros((len(Img1),3))
# Normalizing the points with T matrix
for i in range(len(Img2)):
    norm_img2[i]=np.matmul(T2,np.asarray(Img2[i]))
# Evaluating the fundamental matrix using linear least squares
F_old=linear_least_squares(norm_img1,norm_img2)
# Denormalizing the F matrix with T1 and T2
F=np.matmul(np.matmul(np.transpose(T2),F_old),T1)
# Finding old epipoles and projection matrices before lm
# Evaluating e1 and e2
u, d, vt = np.linalg.svd(F)
e1_old=np.transpose(vt)[:,-1]
e2_old=u[:,-1]
# Evaluating P1 and P2 (proj matrices) for two cameras
Ex=np.zeros((3,3))
Ex[0][1]=-e2_old[2]
Ex[0][2]=e2_old[1]
```

```
Ex[1][0]=e2_old[2]
Ex[1][2]=-e2_old[0]
Ex[2][0]=-e2_old[1]
Ex[2][1]=e2_old[0]
P1_old=np.zeros((3,4))
P1_old[0][0]=1.0
P1_old[1][1]=1.0
P1_old[2][2]=1.0
P2_old=np.zeros((3,4))
F=F/F[2][2]
P2_old[:,0:3]=np.matmul(Ex,F)
P2_old[:,3]=e2_old
# Non linear squares optimization
f=np.zeros((9))
f[0]=F[0][0]
f[1]=F[0][1]
f[2]=F[0][2]
f[3]=F[1][0]
f[4]=F[1][1]
f[5]=F[1][2]
f[6]=F[2][0]
f[7]=F[2][1]
f[8]=F[2][2]
f=f/f[8]
#F_new=least_squares(cost, f, method='lm')
#Converting to 3X3 matrix
#F_newm=np.zeros((3,3))
#F_newm[0][0]=F_new.x[0]
#F_newm[0][1]=F_new.x[1]
#F_newm[0][2]=F_new.x[2]
#F_newm[1][0]=F_new.x[3]
#F_newm[1][1]=F_new.x[4]
#F_newm[1][2]=F_new.x[5]
#F_newm[2][0]=F_new.x[6]
#F_newm[2][1]=F_new.x[7]
```

```
#F_newm[2][2]=F_new.x[8]
F_newm=F
u, d, vt = np.linalg.svd(F_newm)
e1=np.transpose(vt)[:,-1]
e2=u[:,-1]
e1=e1/e1[2]
e2=e2/e2[2]
F_newm=F_newm/F_newm[2][2]
# Evaluating P1 and P2 (proj matrices) for two cameras
Ex=np.zeros((3,3))
Ex[0][1]=-e2[2]
Ex[0][2]=e2[1]
Ex[1][0]=e2[2]
Ex[1][2]=-e2[0]
Ex[2][0]=-e2[1]
Ex[2][1]=e2[0]
P1=np.zeros((3,4))
P1[0][0]=1.0
P1[1][1]=1.0
P1[2][2]=1.0
P2=np.zeros((3,4))
P2[:,0:3]=np.matmul(Ex,F_newm)
P2[:,3]=e2
# Triangulating
X_world=np.zeros((len(Img1),4))
for i in range(len(Img1)):
    A=np.zeros((4,4))
    A[0]=Img1[i][0]*P1[2,:]-P1[0,:]
    A[1]=Img1[i][1]*P1[2,:]-P1[1,:]
    A[2]=Img2[i][0]*P2[2,:]-P2[0,:]
    A[3]=Img2[i][1]*P2[2,:]-P2[1,:]
    u, d, vt = np.linalg.svd(A)
    X_world[i]=np.transpose(vt)[:,-1]
    X_world[i]=X_world[i]/np.linalg.norm(X_world[i])
# To print a scatter plot of the points in 3D
```

```
for i in range(len(X_world)):
    X_world[i]=X_world[i]/X_world[i][3]
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_world[:,0], X_world[:,1], X_world[:,2])
# Finding the corresponding point in rectified images and using
# P1,P2 to find the corresponding world points to find boundary
# Done in matlab
```