FALL 22 ECE 661 – COMPUTER VISION

Homework 9
Sahithi Kodali – 34789866
kodali1@purdue.edu

This HW involves three tasks which are explained below followed by showcasing the results.

TASK – 1: Projective Stereo Rectification

This ask involves obtaining a 3D reconstruction from pair of images captured using an uncalibrated camera like a cellphone. The following steps are followed in this task.

1.1 Image Rectification

The goal of Image rectification is to transform pair of stereo images such that an image physical points are in the same rows as the corresponding points in the other image.

First pick corresponding points manually. 8 corresponding points for each image are picked using GIMP for this HW. This is followed by normalizing the points. To normalize the points the mean of the x- coordinates (x_m) and y-coordinates(y_m) are estimated to calculate the distance (d) between the coordinates and their mean respectively. Also find the mean of these distances (d_m). Using these the transformation matrix (T) to normalize is written as below.

$$T = \begin{bmatrix} c & 0 & -cx \\ 0 & c & -cy \\ 0 & 0 & 1 \end{bmatrix} \text{ where } c = \sqrt{2}/d_m$$

Compute the normalized coordinates $x_n = Tx$ and convert these to the physical coordinated once normalized. Here, x` and x are the homogenous 3 vector form of the pixel coordinates.

Using the normalized coordinates of two images, compute the initial homogenous Fundamental Matrix (F) that can explain the epipolar geometry of the image scene. The matrix A can be given as,

$$A = (x_n x_n` \quad x_n` y_n \quad x_n` \quad y_n` x_n \quad y_n`y_n \quad y_n` \quad x_n \quad y_n \quad 1)$$

Here, (x_n , y_n) and (x_n` , $y_n$`) are the normalized coordinates of the image pairs respectively. Using the Linear least squares, we can now solve for F with 8 unknowns by finding the smallest eigen vector of SVD of A. It is to be conditioned that rank (F) = 2 so that the epipolar lines are corresponded among the images. SVD of F can be performed and the smallest eigen vector of D is set to zero to recompute the conditioned F. Finally, F is denormalized using the transformation matrices T_1 and T_2 that normalized the images as below,

$$F = T_1^T F_{conditioned} T_2$$

Next, obtain the epipoles e and e` that can be used to compute projection matrices P and P` by canonical camera configuration approach as below,

$$Fe = 0 \text{ and } e^{`T} F = 0$$

$$P = [I \mid 0]$$

$$P` = [[e`]_x F \mid e`]$$

Here, I is the identity matrix and $[e`]_x$ is the cross-product equivalent of e` given as,

$$[e`]_x = \begin{bmatrix} 0 & -e`_z & e`_y \\ e`_z & 0 & -e`_z \\ -e`_y & e`_z & 0 \end{bmatrix} \text{ and } e` = \begin{pmatrix} e`_x \\ e`_y \\ e`_z \end{pmatrix}$$

Now using this information, we can rectify the images. However, these results may turn out inaccurate as the initially selected correspondences may not fall in the same respective rows. To minimize this error, Levenberg – Marquardt (LM) algorithm can be used to recompute F and thus the projection matrices P, P`. A cost function that compute the 3D world coordinates of the points and reproject these into the image plane thus finding an error metric between the initial correspondences ($x_i$ , $x_i$`) and (x_p , x_p`) and the reprojected correspondences given as,

$$\text{Cost function (LM)} = \sum_i \left(\| x_i - x_p \|^2 + \| x_p` - x_i` \|^2 \right)$$

Using the refined F, P, P` and the epipoles, the Homography matrices H (left image) and H`(right image) used to rectify the images can be estimated.

Calculating H`:

i. First the translation matrix (T) that send the center of the image to origin is calculated as below using image width (w) and height (h).

$$T = \begin{bmatrix} 1 & 0 & -w/2 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1 \end{bmatrix}$$

ii. Find the angle (θ) that rotates the epipole (e`) such that they are parallel to x-axis and compute the Rotation matrix (R) and the translation matrix(G) that send epipoles to infinity using a scale factor (sf).

$$\theta = \tan^{-1}\left(\frac{e`_y - h/2}{-e`_x - w/2}\right)$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$sf = (e`_x - w/2)\cos\theta - (e`_y - h/2)\sin\theta$$

iii. The compute the initial Homography that rectifies the image center H_c` = GRT.

iv. Noe using H_c` find the rectified centre of image ($x_c$, $y_c$). Find the translation matrix $T_2$ that moves back the rectifies centre to image centre and find the final Homography H`.

$$T_2 = \begin{bmatrix} 1 & 0 & \frac{w}{2} - x_c \\ 0 & 1 & \frac{h}{2} - y_c \\ 0 & 0 & 1 \end{bmatrix}$$

$$H` = T_2GRT$$

Calculating H:

i. To get an initial H_{ini}, the steps (i – iii) are followed. Using the H` estimated above and the initial Homography, the Homography that minimizes the distance between the transformed points is computed to solve for a,b,c using linear-least squares.

$$\text{Cost} = \sum_i (\text{dist}(H_{ini}\, x_i, H`\, x`_i)) = \underset{a,b,c}{\arg\min} \sum_i (a\, x_i + b\, y_i + c - x_i^2)$$

ii. Now, compute the Homography that rectifies the center of image (H_c) and estimate T_1 that brings this center to its original image center like step (iv) above. Then compute the final Homography of image H = T_1H_c.

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

The Homographies H and H` can now be applied on the images respectively to rectify. This rectification should make the images in such a way that the pixel of a 3D world point is located in the same row of both the images.

1.2 Interest Point Detection

To obtain large number of correspondences between the rectified images, canny edge detection is used to get the interest points which are non-zero pixels. If the rectification was good enough, the pixels of a point will exist in the same row in both images and the correspondences can be matched accordingly using this assumption of correctness also showcasing the efficiency of the rectified images. Since an interest point in the first image can have multiple points in the second image, a NCC or SSD distance metrics are used to filter the corresponding points.

1.3 Projective Reconstruction and 3D visualization

Using projective reconstruction, the images can be plotted in 3D using the stereo images pair. The projection matrices are to be estimated under the canonical configuration of camera assumption.

Then extract the world point coordinates (X) using both the image coordinates and projection matrices P and P`. The solution for X can be give by solving the equation AX = 0 using linear least squares method. The smallest eigen vector obtained for A^TA after SVD is the required world coordinates X.

$$A = \begin{bmatrix} xP_3^T - P_1^T \\ yP_3^T - P_2^T \\ x`P_3^{`T} - P_1^{`T} \\ y`P_3^{`T} - P_2^{`T} \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} P_1^T \\ P_2^T \\ P_3^T \end{bmatrix}$$

Finally, the 3D point coordinates are used to reconstruct the actual image. This 3D construction involves projective distortion due to the use of uncalibrated cameras and hence doesn't appear as to be in the real world but is good for visualization of the real object in image.

TASK – 2: Loop and Zhang Algorithm

Loop and Zhang algorithm works by decomposing the Homographies (H, H`) as product of purely projective homographies ($H_p$, $H_p$`), similarity homographies (H_{sim}, H_{sim}`) and the shearing homographies ($H_{sh}$, $H_{sh}$`) as below.

$$H = H_p \, H_{sim} \, H_{sh}$$

$$H` = H_p` \, H_{sim}` \, H_{sh}`$$

Here, (H_p, H_p`) send the epipoles to infinity in the direction that causes minimal distortion to images. ($H_{sim}$, $H_{sim}$`) can rotate, translate and scale image and hence they rotate the epipoles such that they will be on world x-axis after projective Homography. (H_{sh}, H_{sh}`) play a role in adding additional degrees of freedom to reduce the distortion as much as possible as the projective and similarity though makes the epipoles parallel and to infinity the distortion by projective homography still exists and hence needs to be minimized. To avoid this distortion an affine part is introduced. Further, a valid choice for the matrices is made in such a way that the distortion is reduced. This is an overview of the Loop and Zhang algorithm working.

TASK – 3: Dense Stereo Matching

Dense Stereo matching algorithm assist in finding the accuracy of the pixel correspondences and marks the occluded regions as invalid. This process is performed using Census transform method as described below.

a. Firstly, find the intensity of the pixels in M x N neighborhood of a pixel in left image and create such neighborhood around the pixel in right image.
b. A binary masking is applied such that the pixels greater than the center of neighborhood are marked as 1 and the remaining as 0. Now perform a XOR between both the neighborhoods that are binarized and the number of 1s are considered as the cost value.
c. Now consider the pixels at a distance d in the same row and repeat the same steps above for d_{max} times. This produces the cost for the pixel in the left image.
d. Find the minimum cost and the index corresponding to it is associated to the disparity value of the pixel in left image. Similarly, all the disparity values of pixels are computed to plot the disparity map.

After computing the disparity map, the accuracy with the ground truth and the error masks are computed for different error values.

RESULTS & OBSERVATIONS:

Task- 1:

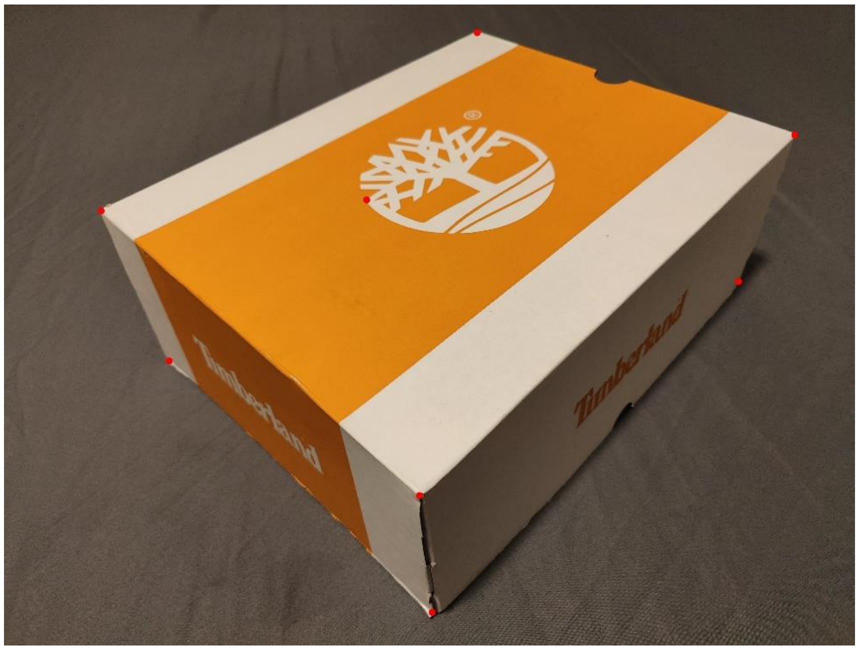
Figure 1: Image showing the correspondences chosen for Image 1

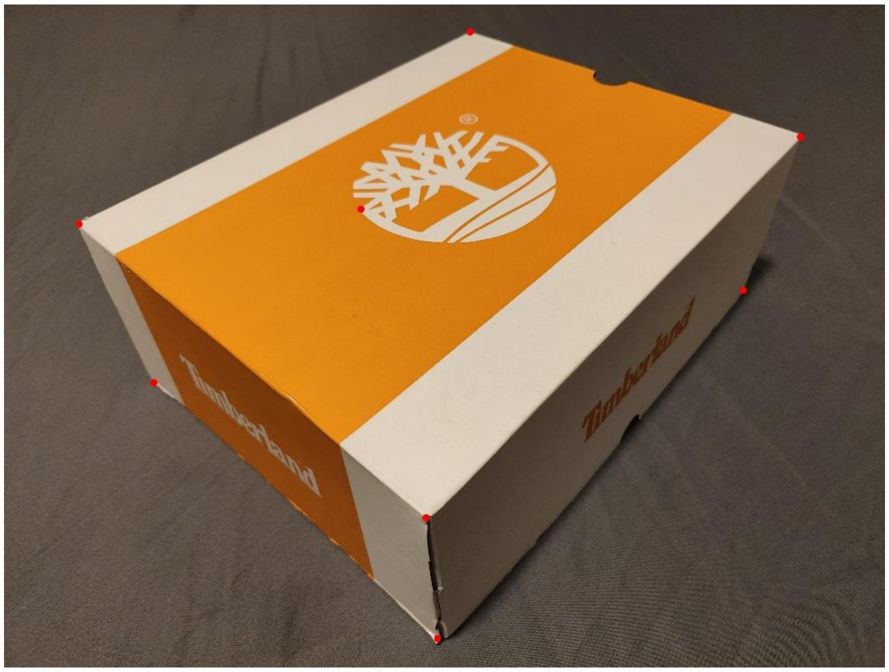

Figure 2: Image showing the correspondences chosen for Image 2

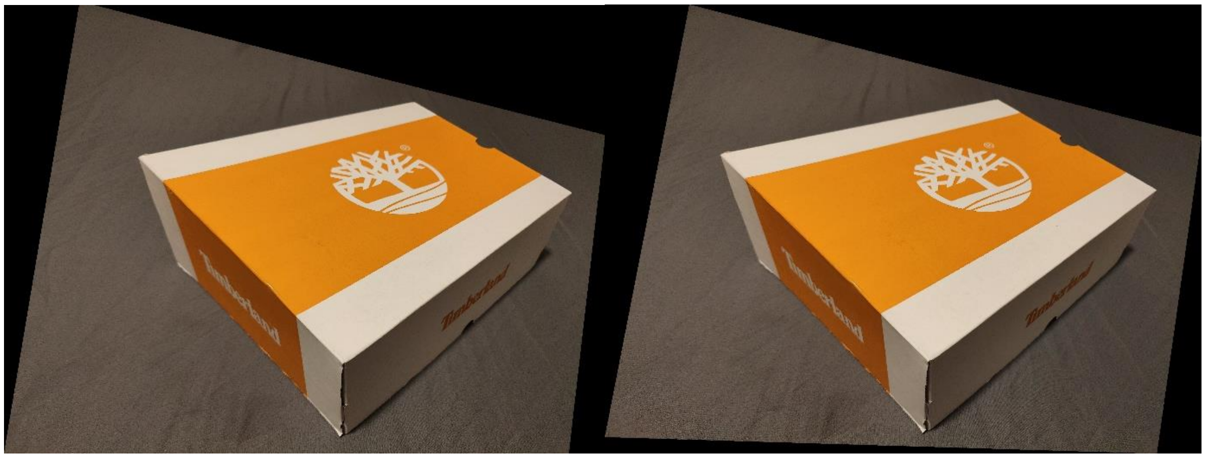

Figure 3: Rectified images of Image 1 and Image 2

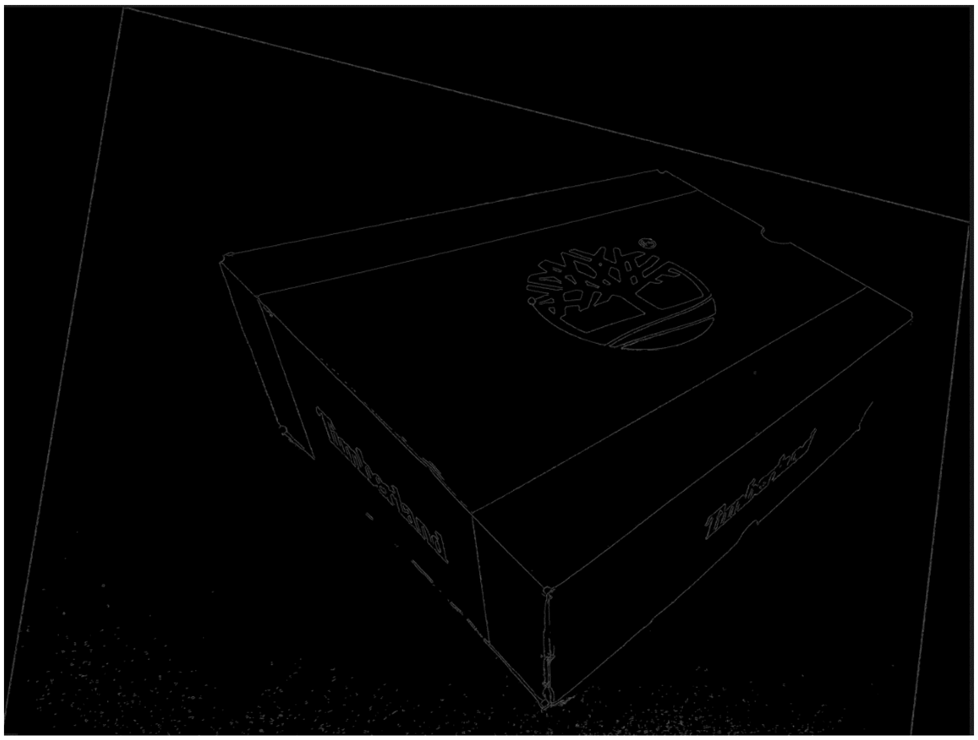

Figure 4: Canny edge detection of rectified image of Image 1

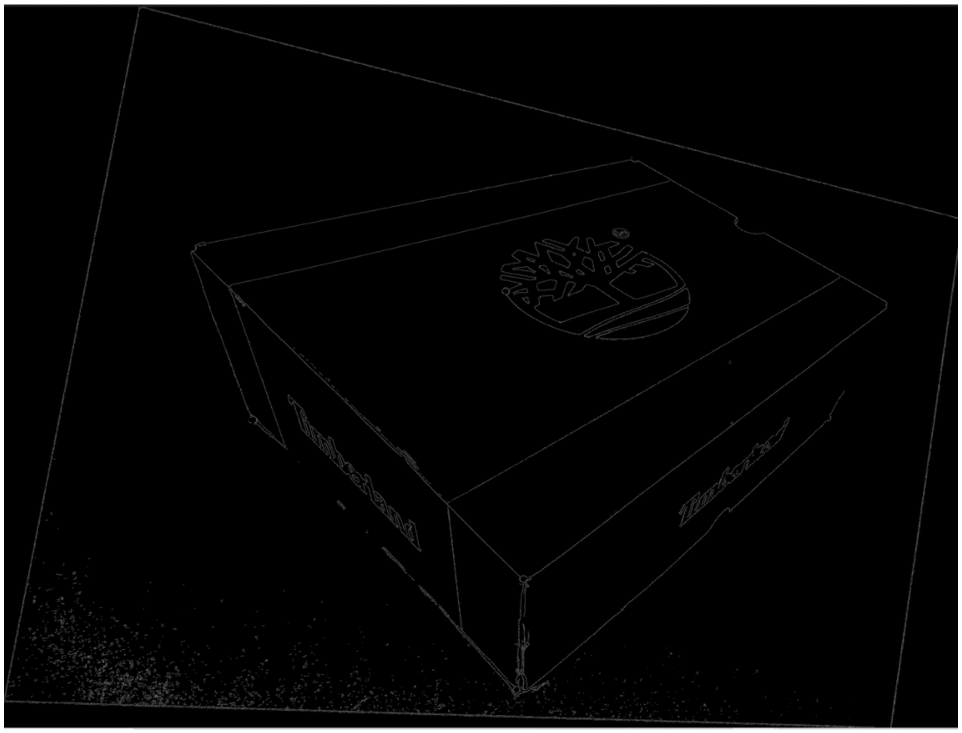

Figure 5: Canny edge detection of rectified image of Image 2

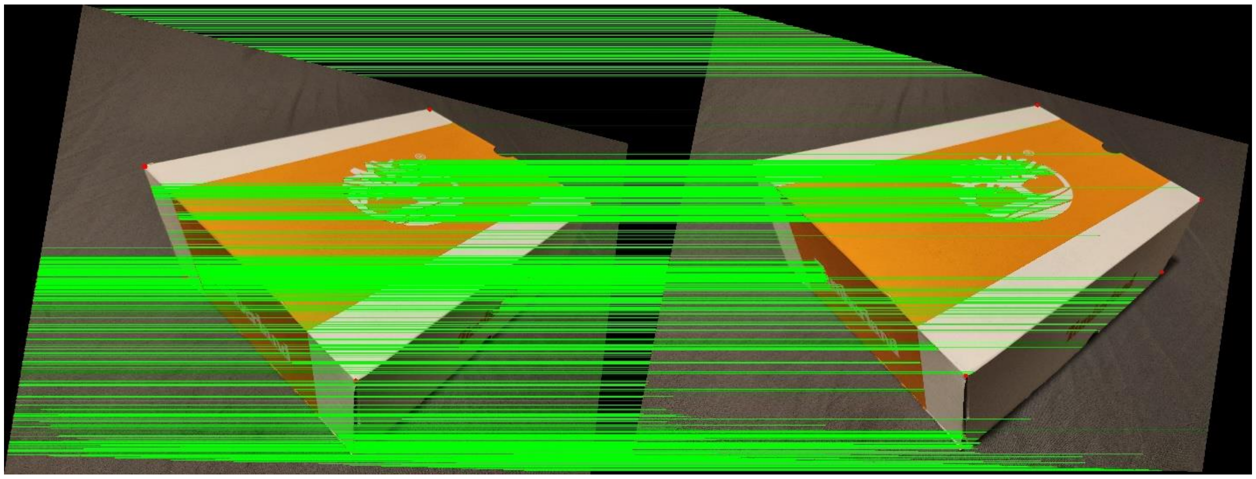
Figure 6: Plotted correspondences between the Image 1 and Image 2

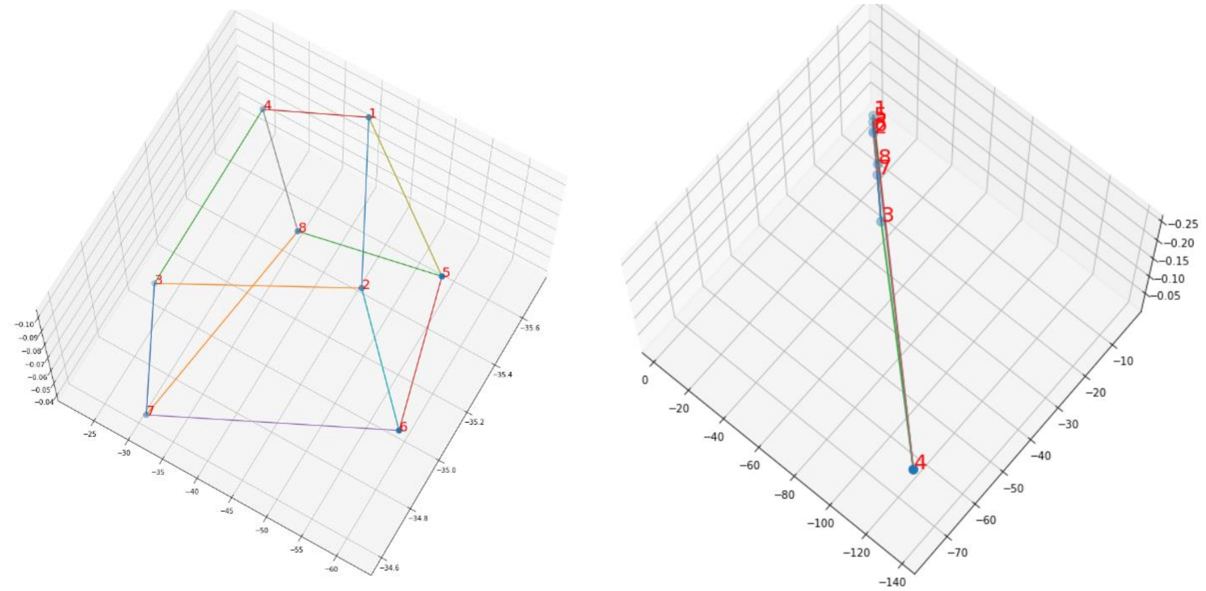
Figure 7,8: The 2 views of the 3D reconstruction of the scene structure in Images 1 and 2

Task- 2:

Implementing the Loop and Zhang algorithm code as given the following outputs were obtained for the pair of stereo images captured.

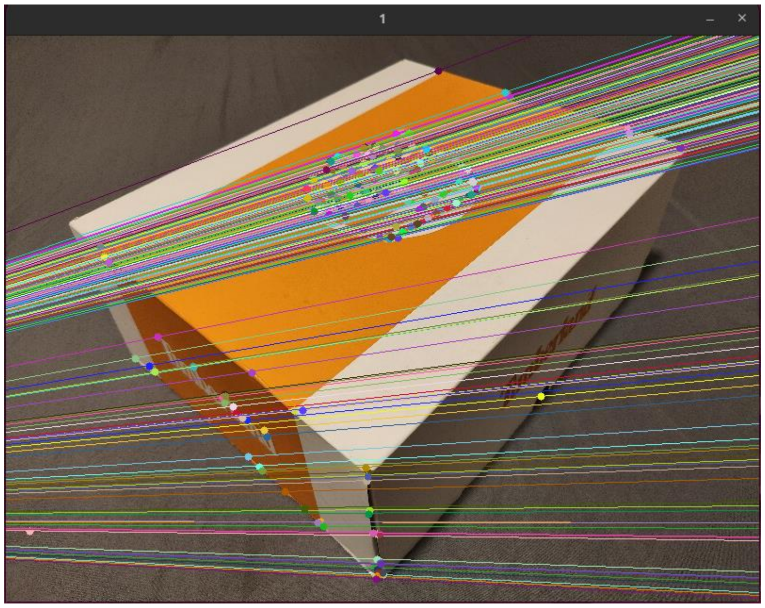

Figure 9: Correspondences of image 1 using Loop and Zhang algorithm

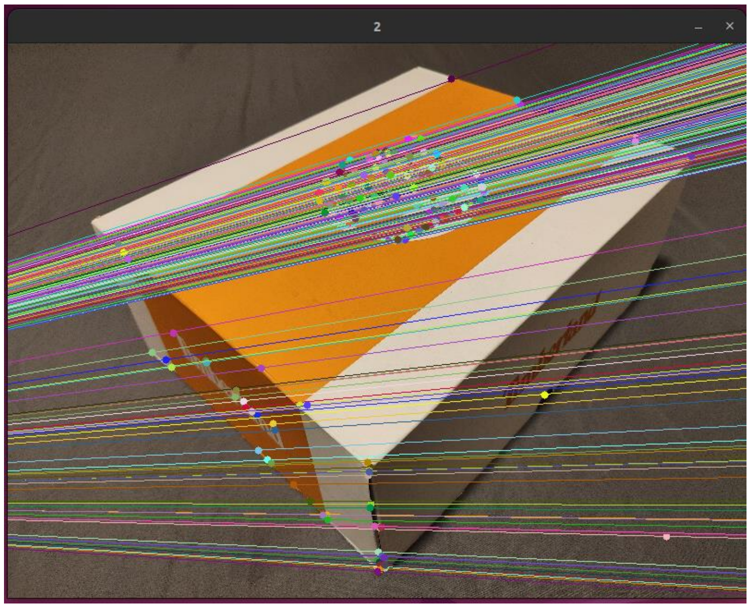

Figure 10: Correspondences of image 2 using Loop and Zhang algorithm

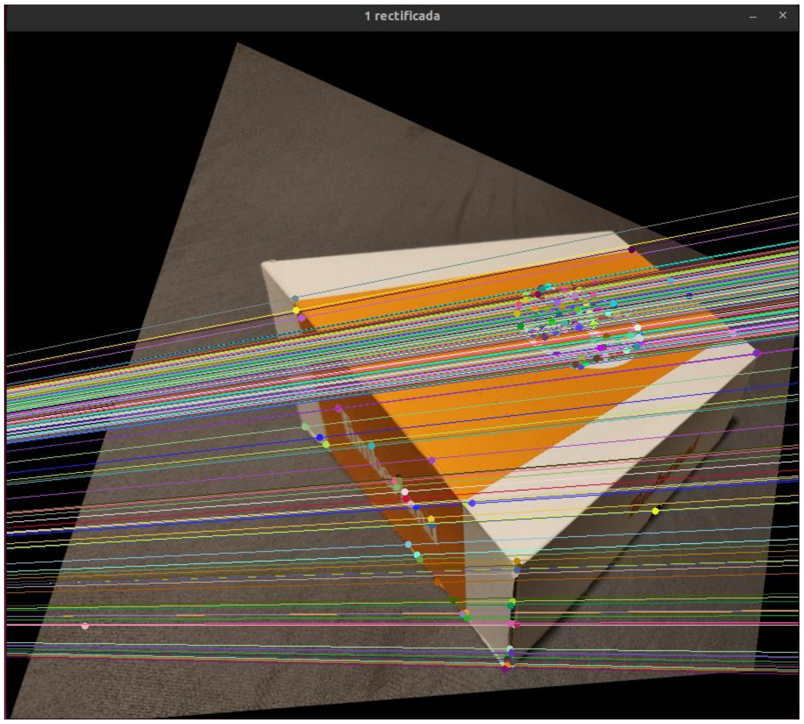

Figure 11: Rectified image of the image 1 using Loop and Zhang algorithm

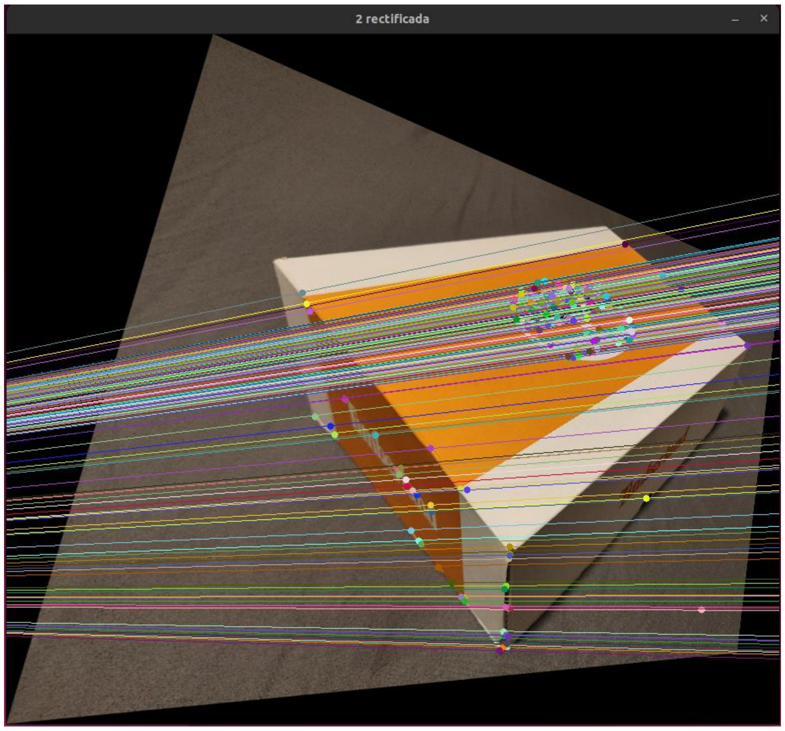

Figure 12: Rectified image of the image 2 using Loop and Zhang algorithm

Observations:

We can observe that the images warped using Loop and Zhang algorithm have less projective distortion as compared to the method used by my own pipeline in Task 1. The corresponding pixels are also plotted with very minimal error and better. Hence, we can clearly say that, Loop and Zhang algorithm can successfully minimize the projective error as much as possible when compared to the own pipeline in Task 1.

Task- 3:

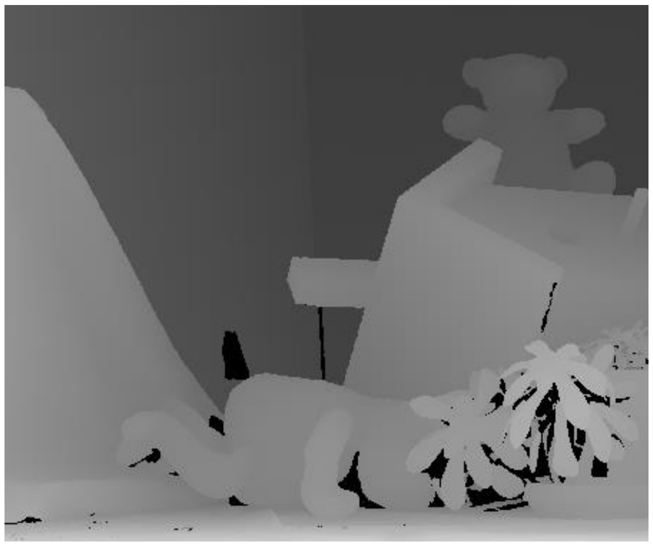

Figure 13: Ground Truth Disparity map

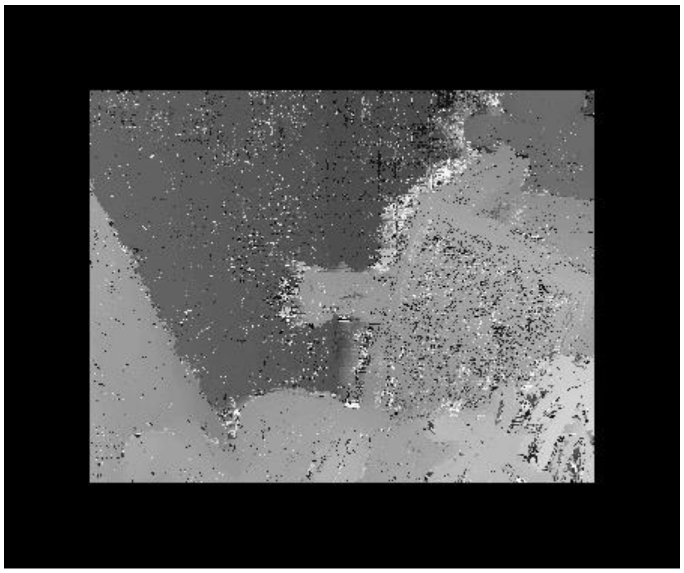

Figure 14: Estimated disparity map for window = 10 and dmax = 52

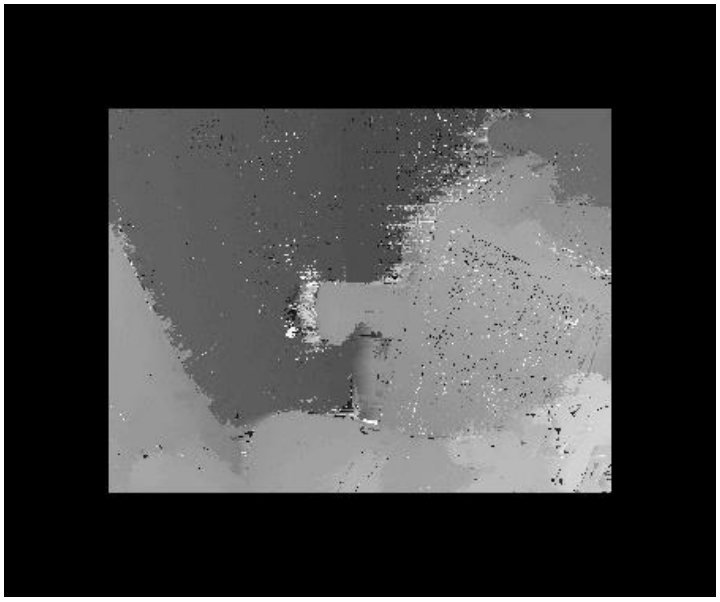

Figure 15: Estimated disparity map for window = 29 and dmax = 52

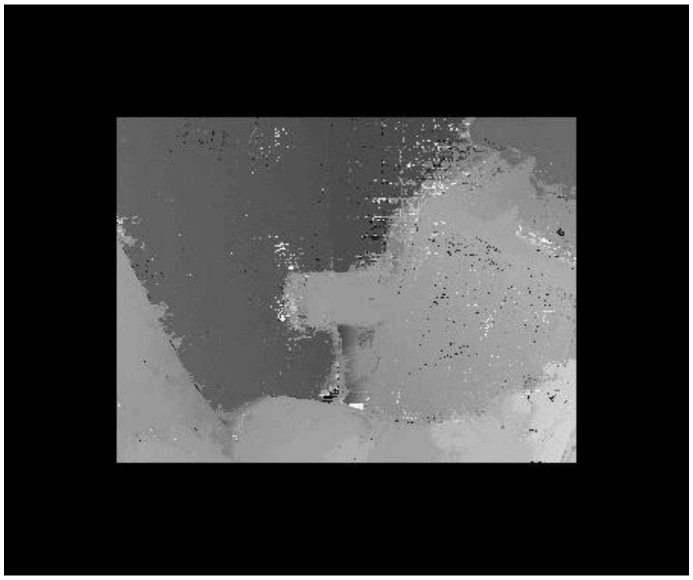

Figure 16: Estimated disparity map for window = 45 and dmax = 52

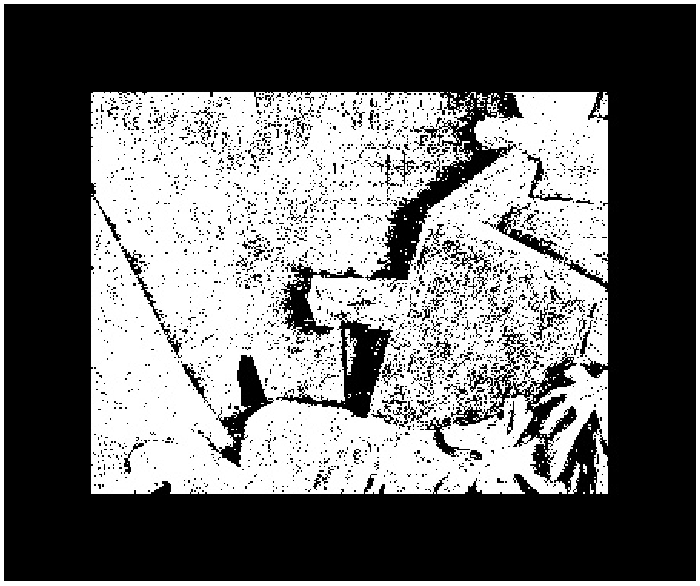

Figure 17: Error mask for disparity map with window = 10 and dmax = 52. Accuracy obtained is 0.8016 (80%).

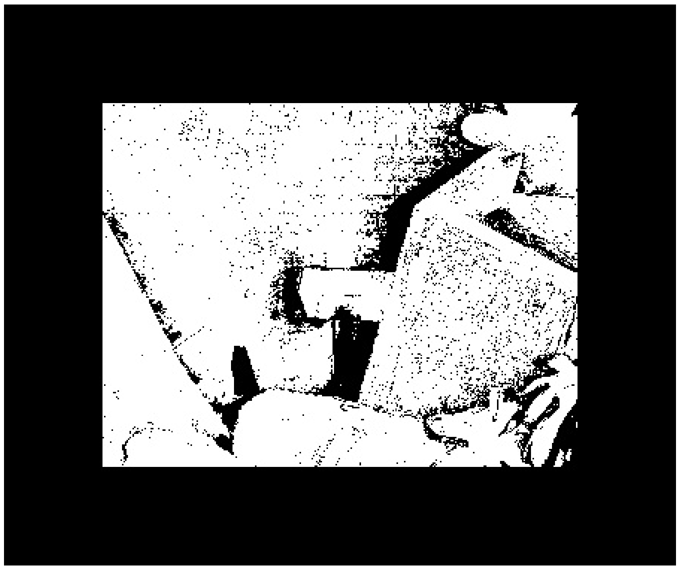

Figure 18: Error mask for disparity map with window = 29 and dmax = 52. Accuracy obtained is 0.8545 (85%)

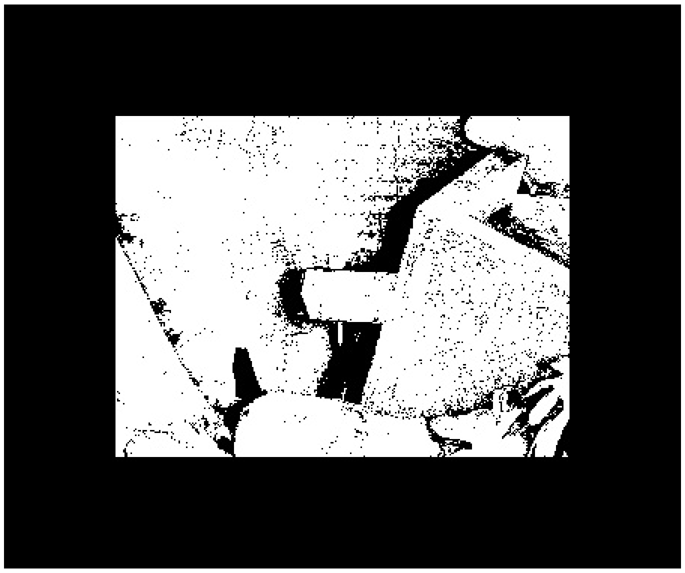

Figure 19: Error mask for disparity map with window = 45 and dmax = 52.
Accuracy obtained is 0.850 (85%)

Observations:

We can observe that as the window size increase the disparity map is better with less error pixels, and is closer to the ground truth disparity map shown. We can also see that the black pixels are less visible as window increases and hence mask turns white. The numerical comparison of accuracy is also reported below the error mask images showing the increase in accuracy. It can be noted that the accuracy didn't improve from window 29 to 45 which can be due to an already better map or window size that is working good enough.

SOURCE CODE:

```
#!/usr/bin/env python
# coding: utf-8

# <h2><center>ECE661 COMPUTER VISION</center></h2>
# <h3><center>Homework - 9 </center></h3>
# <h3><center>Sahithi Kodali - 34789866</center></h3>
# <h3><center>kodali1@purdue.edu</center></h3>

# ### TASK1

# In[2]:


# Import libraries needed
```

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
import copy
import pandas as pd
import os
import PIL
get_ipython().run_line_magic('matplotlib', 'inline')

from scipy import optimize


# In[3]:


#All the functions below have been inspired by 2020 Hw1 sample and my past
homeworks

#condition F to rank(F) = 2
def F_conditioned(F):
    u,s,v = np.linalg.svd(F)
    s_mat = np.array([[s[0], 0, 0], [0, s[1], 0], [0, 0, 0]])
    F_cond = np.dot(np.dot(u, s_mat), v)
    return F_cond


# In[4]:


#normalize the points chosen

def normalize_pts(pts):
    x = pts[:,0]
    y = pts[:,1]

    x_m = np.mean(x)
    y_m = np.mean(y)

    d = np.sqrt((x - x_m)**2 + (y - y_m)**2)
    d_m = np.mean(d)

    sf = np.sqrt(2)/d_m
    sf_mat = np.array([[sf, 0, -sf*x_m], [0, sf, -sf*y_m], [0, 0, 1]])
    pts_HC = np.hstack((pts, np.ones((len(pts), 1))))
    norm_pts = np.transpose(np.dot(sf_mat, np.transpose(pts_HC)))

    return sf_mat, norm_pts


# In[5]:


#calculate F using normalized points
def compute_F(pts1, pts2):

    A = np.zeros((8,9))

    for i in range(8):
        x1 = pts1[i][0]
        x2 = pts2[i][0]
        y1 = pts1[i][1]
        y2 = pts2[i][1]
        A[i] = np.array([x1*x2, x2*y1, x2, y2*x1, y2*y1, y2, x1, y1, 1])

    u, s, v = np.linalg.svd(A)
    F = v[-1]
    F = np.reshape(F, (3,3))
    F_cond = F_conditioned(F)

    return F_cond


# In[6]:


#compute epipoles
def compute_e(F):
    u,s,v = np.linalg.svd(F)
    e = np.transpose(v[-1, :])
    e = e/e[2]

    e2 = u[:, -1]
    e2 = e2/e2[2]
    e2_x = np.array([[0, -e2[2], e2[1]], [e2[2], 0, -e2[0]], [-e2[1], e2[0], 0]])

    return e, e2, e2_x


# In[7]:
```

```


# compute the projection matrices
def compute_P(F, e2, e2_x):
    P = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
    P2 = np.hstack((np.dot(e2_x, F), np.transpose([e2])))
    return P, P2


# In[8]:


#cost function using LM non-linear optimization and world coordinates of point
correspondences

def cost(F, pts):
    F = np.reshape(F, (3,3))
    [e, e2, e2_x] = compute_e(F)
    [P, P2] = compute_P(F, e2, e2_x)

    pts1 = np.hstack((pts[0], np.ones((len(pts[0]), 1))))
    pts2 = np.hstack((pts[1], np.ones((len(pts[1]), 1))))

    err = []

    for i in range(np.size(pts, 1)):
        A = np.zeros((4, 4))
        A[0] = pts1[i][0]*P[2, :] - P[0, :]
        A[1] = pts1[i][1]*P[2, :] - P[1, :]
        A[2] = pts2[i][0]*P2[2, :] - P2[0, :]
        A[3] = pts2[i][1]*P2[2, :] - P2[1, :]

        u,s,v = np.linalg.svd(A)
        ev = np.transpose(v[-1, :])
        ev_norm = ev/ev[3]

        x1 = np.dot(P, ev_norm)
        x1 = x1/x1[2]

        x2 = np.dot(P2, ev_norm)
        x2 = x2/x2[2]

        err.append(np.linalg.norm(x1 - pts1[i])**2)
        err.append(np.linalg.norm(x2 - pts2[i])**2)
    return np.ravel(err)


# In[9]:


#compute homogaphy

def compute_homo(img, pts1, pts2, F, e, e2, P, P2):
    h = img.shape[0]
    w = img.shape[1]

    #compute H2
    T_ini = np.array([[1, 0, -w/2], [0, 1, -h/2], [0, 0, 1]])

    theta2 = math.atan2(e2[1] - h/2, -(e2[0] - w/2))
    R2 = np.array([[math.cos(theta2), -math.sin(theta2), 0], [math.sin(theta2),
math.cos(theta2), 0], [0, 0, 1]])

    f2 = (e2[0] - w/2)*math.cos(theta2) - (e2[1] - h/2)*math.sin(theta2)
    G2 = np.array([[1, 0, 0], [0, 1, 0], [-1/f2, 0, 1]])

    H2_centre = np.dot(G2, np.dot(R2, T_ini))
    img_centre2 = np.dot(H2_centre, np.array([w/2, h/2, 1]))
    img_centre2 = img_centre2/img_centre2[2]

    T2 = np.array([[1, 0, w/2 - img_centre2[0]], [0, 1, h/2 - img_centre2[1]],
[0, 0, 1]])

    H2 = np.dot(np.dot(T2, G2), np.dot(R2, T_ini))
    H2 = H2/H2[2, 2]

    #compute H
    theta = math.atan2(e[1] - h/2, -(e[0] - w/2))
    R = np.array([[math.cos(theta), -math.sin(theta), 0], [math.sin(theta),
math.cos(theta), 0], [0, 0, 1]])


    f = (e[0] - w/2)*math.cos(theta) - (e[1] - h/2)*math.sin(theta)
    G = np.array([[1, 0, 0], [0, 1, 0], [-1/f, 0, 1]])

    H_ini = np.dot(G, np.dot(R, T_ini))

    pts1 = np.hstack((pts1, np.ones((len(pts1), 1))))
    pts2 = np.hstack((pts2, np.ones((len(pts2), 1))))
    c1 = np.transpose(np.dot(H_ini, np.transpose(pts1)))
    c2 = np.transpose(np.dot(H2, np.transpose(pts2)))

    c1[:, 0] = c1[:, 0]/c1[:, 2]
    c1[:, 1] = c1[:, 1]/c1[:, 2]
    c1[:, 2] = c1[:, 2]/c1[:, 2]

    c2[:, 0] = c2[:, 0]/c2[:, 2]

    abc = np.dot(np.linalg.pinv(c1), c2[:, 0])

    H_abc = np.array([[abc[0], abc[1], abc[2]], [0, 1, 0], [0, 0, 1]])
    H_centre = np.dot(H_abc, H_ini)
    img_centre = np.dot(H_centre, np.array([w/2, h/2, 1]))
    img_centre = img_centre/img_centre[2]

    T = np.array([[1, 0, w/2 - img_centre[0]], [0, 1, h/2 - img_centre[1]], [0,
0, 1]])

    H = np.dot(T, H_centre)
    H = H/H[2, 2]

    return H, H2


# In[141]:


#apply H to the images
def apply_homo(dimg, H):
    [w, h] = [dimg.shape[1], dimg.shape[0]]
    H_inv = np.linalg.pinv(H)

    wc = ((0, 0), (0, h-1), (w-1, h-1), (w-1, 0))
    x_img = []
    y_img = []

    for i in range(4):
        wx = wc[i][0]
        wy = wc[i][1]
        [ix, iy, iz] = np.dot(H, [wx, wy, 1.0])
        x_img.append(round(ix/iz))
        y_img.append(round(iy/iz))
    ic_w = max(x_img) - min(x_img)
    ic_h = max(y_img) - min(y_img)

    #get the scaling factors and offset
    asp_ratio = h/w
    h_adj = asp_ratio * ic_w

    x_scale = 1.0
    y_scale = h_adj/ic_h

    x_offset = min(x_img)
    y_offset = min(y_img)

    rimg = np.zeros((round(h_adj), round(ic_w), 3), np.uint8)

    h_rimg = rimg.shape[0]
    w_rimg = rimg.shape[1]

    #perform homography
    xyidxs_rimg = np.indices((w_rimg, h_rimg))
    xidxs_rimg = xyidxs_rimg[0].reshape(w_rimg*h_rimg, 1)
    yidxs_rimg = xyidxs_rimg[1].reshape(w_rimg*h_rimg, 1)
    zidxs_rimg = np.ones((w_rimg*h_rimg, 1))
    idxs_rimg = np.ndarray.astype(np.concatenate((xidxs_rimg, yidxs_rimg,
zidxs_rimg), 1), int)

    sc_idxs = idxs_rimg.copy()
    sc_idxs[:, 0] = sc_idxs[:, 0]/x_scale + x_offset
    sc_idxs[:, 1] = sc_idxs[:, 1]/y_scale + y_offset

    homo_idxs = np.transpose(np.dot(H_inv, np.transpose(sc_idxs)))
    homo_idxs[:, 0] = homo_idxs[:, 0]/homo_idxs[:, 2]
    homo_idxs[:, 1] = homo_idxs[:, 1]/homo_idxs[:, 2]
    homo_idxs = np.ndarray.astype(np.round(homo_idxs), int)

    #refine points to get in the domain
    idxs_rimg = idxs_rimg[homo_idxs[:, 0] >= 0]
    homo_idxs = homo_idxs[homo_idxs[:, 0] >= 0]

    idxs_rimg = idxs_rimg[homo_idxs[:, 1] >= 0]
    homo_idxs = homo_idxs[homo_idxs[:, 1] >= 0]

    idxs_rimg = idxs_rimg[homo_idxs[:, 0] < w]
    homo_idxs = homo_idxs[homo_idxs[:, 0] < w]
    idxs_rimg = idxs_rimg[homo_idxs[:, 1] < h]
    homo_idxs = homo_idxs[homo_idxs[:, 1] < h]

    #plot the pixel coordinates
    for r in range(np.size(homo_idxs, 0)):
        rimg[idxs_rimg[r,1]][idxs_rimg[r,0]] = 
dimg[homo_idxs[r,1]][homo_idxs[r,0]]

    return rimg


# In[11]:


#Canny edge detection
def canny_edges(img):
    if(len(img.shape) > 2):
        g_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    g_img = cv2.blur(g_img, (9,9))
    canny = cv2.Canny(g_img, 30, 20, 5)
    return canny


# In[12]:


#get correspondences from the canny plot above. Harris has also been executed.
def get_corr(canny1, canny2, d_thresh):
    if(len(canny1.shape) > 2):
        canny1 = cv2.cvtColor(canny1, cv2.COLOR_BGR2GRAY)
    if(len(canny2.shape) > 2):
        canny2 = cv2.cvtColor(canny2, cv2.COLOR_BGR2GRAY)

    corr = []

    for r in range(canny1.shape[0]):
        cidxsL = np.where(canny1[r] > 0)[0]
        if np.size(cidxsL) == 0:
            continue

        for cidxL in cidxsL:
            rowR = canny2[r, cidxL:cidxL + d_thresh + 1]
            cidxsR = np.where(rowR > 0)[0]
            if np.size(cidxsR) == 0:
                continue
            cR = cidxsR[0] + cidxL
            canny2[r, cR] = 0
            corr.append([[cidxL, r], [cR, r]])

    return corr


# In[13]:


#filter correspondences
def distFilt_corr(img1, img2, corrs, w_size, corrs_thresh):
    win_size = int(w_size/2)

    if(corrs_thresh > len(corrs)):
        corrs_thresh = len(corrs)

    ssd_corr = []

    for c in corrs:
        c1 = c[0]
        c2 = c[1]
        w1 = img1[c1[1] - win_size : c1[1] + win_size + 1, c1[0] - win_size :
c1[0] + win_size + 1]
        w2 = img2[c2[1] - win_size : c2[1] + win_size + 1, c2[0] - win_size :
c2[0] + win_size + 1]
        ssd = np.sum((w1 - w2) ** 2)
        ssd_corr.append(ssd)

    sort_corrs = [corr for d,corr in sorted(zip(ssd_corr, corrs), key = lambda
pt: pt[0])]

    return sort_corrs[0 : corrs_thresh]


# In[118]:


#plot the correspondences
def plot_corr(img1, img2, corr):

    h1 = img1.shape[0]
    h2 = img2.shape[0]

    w1 = img1.shape[1]
    w2 = img2.shape[1]
    if(h1 < h2):
        img1 = np.concatenate((img1, np.zeros((h2 - h1, w1, 3), np.uint8)), 0)
    elif (h1 > h2):
        img2 = np.concatenate((img1, np.zeros((h1 - h2, w2, 3), np.uint8)), 0)

    concat_img = np.concatenate((img1, img2), 1)

    for i in range(len(corr)):
        c = corr[i]
        p1 = tuple(c[0])
        p2 = tuple(np.array(c[1]) + [w1, 0])
        cv2.line(concat_img, p1, p2, (0, 255, 0), 1)
        cv2.circle(concat_img, p1, 1, (0, 0, 255), 1)
        cv2.circle(concat_img, p2, 1, (0, 0, 255), 1)

    return concat_img


# In[15]:


#to project the world coordinates.
def proj_wc(P, P2, corr):
    wc = []

    for i in range(len(corr)):
        A = np.zeros((4,4))
        pt1 = corr[i][0]
        pt2 = corr[i][1]
        A[0] = pt1[0] * P[2, :] - P[0, :]
        A[1] = pt1[1] * P[2, :] - P[1, :]
        A[2] = pt2[0] * P2[2, :] - P2[0, :]
        A[3] = pt2[1] * P2[2, :] - P2[1, :]

        u,s,v = np.linalg.svd(np.dot(np.transpose(A), A))
        ev = np.transpose(v[-1, :])
        wc.append(ev/ev[3])

    wc = np.reshape(wc, (len(corr), 4))
    return wc


# In[203]:
```

```
img1 = cv2.imread('NP3.jpg')
img2 = cv2.imread('NP4.jpg')


# In[205]:


im1 = img1.copy()
im2 = img2.copy()


# In[147]:


coords = np.array([[[91,193], [390,458], [743,121], [445,26], [154,334],
[402,568], [692,256], [362,131]],
                   [[68,200], [383,463], [723,118], [424,25], [135,340],
[394,573], [672,255], [344,132]]])


# In[206]:


# coords = np.array([[[456,968], [1952,2304], [3716,612], [2224,132], [776,1672],
[2012,2852], [3452,1300], [1704,916]],
#             [[340,996], [1916,2328], [3616,600], [2116, 124],  [680,1716],
[1968,2876], [3356,1296], [1620,928]]])


# In[208]:


#plot the points

for i in range(8):
    cv2.circle(img1, tuple(coords[0][i]), 8, (0,0, 255), 15)
    cv2.circle(img2, tuple(coords[1][i]), 8, (0,0, 255), 15)

cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/corrPoints1.jpg", img1)
cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/corrPoints2.jpg", img2)


# In[209]:


#normalize points
sf_mat1, norm_pts1 = normalize_pts(coords[0])
```

```python
sf_mat2, norm_pts2 = normalize_pts(coords[1])


# In[210]:


#compute F without LM optimization
F_ini = compute_F(norm_pts1, norm_pts2)
F = np.dot(np.dot(np.transpose(sf_mat2), F_ini), sf_mat1)
F = F/F[2,2]


# In[211]:


#compute F with LM optimization (non linear least square minimization)
F_vect = np.ravel(F)
F_LM = optimize.least_squares(cost, F_vect, args = [coords], method = 'lm').x


# In[212]:


#compute epipoles (e,e`) and projections matrices (P, P`) using refined F
F_LM = np.reshape(F_LM, (3,3))
F_cond = F_conditioned(F_LM)
F_cond = F_cond/F_cond[2,2]

[e, e2, e2_x] = compute_e(F_cond)
[P, P2] = compute_P(F_cond, e2, e2_x)


# In[213]:


#compute Homographies (H, H`)
H, H2 = compute_homo(img1, coords[0], coords[1], F_cond, e, e2, P, P2)


# In[239]:


#rectify the images
img1_rect = apply_homo(img1, H)
img2_rect = apply_homo(img2, H2)
```

```
# In[240]:


cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/rect_img1_dots.jpg", img1_rect)
cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/rect_img2_dots.jpg", img2_rect)


# In[243]:


# img2_rect_pad = np.pad(img2_rect, pad_width=[(35, 36),(47, 48),(0, 0)],
mode='constant')
# # img2_rect_pad = np.pad(img2_rect, pad_width=[(20, 21),(27, 27),(0, 0)],
mode='constant')


# In[247]:


cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/rect_img2_pad_dots.jpg",
img2_rect_pad)


# In[263]:


#obtain canny edges
img1_canny = canny_edges(img1_rect)
img2_canny = canny_edges(img2_rect_pad)

cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/canny_img1.jpg", img1_canny)
cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/canny_img2.jpg", img2_canny)


# In[265]:


canny_corr = get_corr(img1_canny, img2_canny, 40)
dist_corr = distFilt_corr(img1_rect, img2_rect_pad, canny_corr, 30, 1e6)
plor_corr = plot_corr(img1_rect, img2_rect_pad, dist_corr)

cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/plot_corres.jpg", plor_corr)


# In[253]:
```

```
# coords_rect = [[[1034, 1177], [919, 1177]],
#                 [[2585, 2756], [2508, 2756]],
#                 [[4334, 1458], [4219, 1458]],
#                 [[3141, 754], [3025, 754]],
#                 [[1320, 1991], [1205, 1991]],
#                 [[2580, 3289], [2458, 3289]],
#                 [[4070, 1986], [3938, 1986]],
#                 [[2525, 1381], [2426, 1381]]]

# coords_1plt = np.array([[[91,193], [390,458], [743,121], [445,26], [154,334],
[402,568], [692,256], [362,131]],
#                     [[68,200], [383,463], [723,118], [424,25], [135,340],
[394,573], [672,255], [344,132]]])

coords_rect = [[[555, 392], [495, 392]],
               [[1161, 659], [1113, 659]],
               [[1415, 470], [1359, 470]],
               [[1223, 369], [1173, 369]],
               [[752, 576], [692, 576]],
               [[1172, 734], [1116, 734]],
               [[1391, 528], [1331, 528]],
               [[1133, 419], [1085, 419]]]

wc_rect = proj_wc(P, P2, coords_rect)
wc_dist = proj_wc(P, P2, dist_corr)


# In[255]:


fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.scatter(wc_dist[:, 0], wc_dist[:, 1], wc_dist[:, 2])
ax.scatter(wc_rect[:, 0], wc_rect[:, 1], wc_rect[:, 2])

pairs = [[0,2], [0,1], [2,3], [2,4], [4,5], [5,3], [1,6], [5,6], [1,3]]

for p in pairs:
    ax.plot([wc_rect[p[0]][0], wc_rect[p[1]][0]], [wc_rect[p[0]][1],
wc_rect[p[1]][1]], [wc_rect[p[0]][2], wc_rect[p[1]][2]])

plt.show()


# In[196]:
```

```
ax1 = fig.add_subplot(221)
ax1.set_aspect('equal')
ax1.imshow(im1)

ax2 = fig.add_subplot(222)
ax2.set_aspect('equal')
ax2.imshow(im2)


# In[200]:


for p in pairs:
    ax1.plot([coords_1plt[p[0]][0], coords_1plt[p[1]][0]], [coords_1plt[p[0]][1],
coords_1plt[p[1]][1]], [coords_1plt[p[0]][2], coords_1plt[p[1]][2]]
    ax2.plot([coords_2plt[p[0]][0], coords_2plt[p[1]][0]], [coords_2plt[p[0]][1],
coords_2plt[p[1]][1]], [coords_2plt[p[0]][2], coords_1plt[p[1]][2]]

plt.show()


# ### TASK - 3

# In[256]:


#Estimate the disparity map using census transform
def census_transform (l_img, r_img, W, d_max) :

    l_h, l_w = l_img.shape[0], l_img.shape[1]
    d_map = np.zeros((l_h,l_w))

    for i in range ( d_max + int(W/2), l_h - int(W/2) - d_max):
        for j in range (l_w-int(W/2)-d_max-1, d_max + int(W/2)-1, -1):

            cost = []

            l_img_win = l_img[i-int(W/2) : i+1+int(W/2), j-int(W/2) :
j+1+int(W/2)]
            l_temp = l_img_win > l_img[i,j]
            l_bitvec = np.ravel(l_temp.astype(int))

            for d in range (0, d_max+1) :
                r_img_win = r_img [i-int(W/2) : i+1+int(W/2), j-d-int(W/2) : j-
d+1+int(W/2)]
                r_temp = r_img_win > r_img [i,j-d]
                r_bitvec = np.ravel(r_temp.astype(int))

                bits_XOR = np.logical_xor(l_bitvec, r_bitvec)
                unique, count = np.unique(bits_XOR, return_counts = True)
                uc = dict(zip(unique, count))

                if 1 in uc :
                    cost.append(uc[1])
                else :
                    cost.append(0)

            d_map[i,j] = np.argmin(cost)

    d_map = d_map.astype(np.uint8)
    d_map_sc = ((d_map / np.max(d_map))*255).astype(np.uint8)

    return d_map, d_map_sc


# In[257]:


#find the accuracy
def acc_dmap(gt_dmap, est_dmap):

    #compute the error
    comp_dmap = est_dmap.astype(np.int16)
    gt_dmap = gt_dmap.astype(np.int16)
    err = (abs(comp_dmap - gt_dmap)).astype(np.uint8)

    #get the mask for pixels with disparity error less than 2.
    mask = ((err <= 2) * 255).astype(np.uint8)

    #calculate the accuracy
    err_mask = np.bitwise_and(comp_dmap, mask)
    N = np.count_nonzero(comp_dmap)
    acc = np.count_nonzero(err_mask)/N

    for i in range(err_mask.shape[0]):
        for j in range(err_mask.shape[1]):
            if err_mask[i,j] !=0:
                err_mask[i,j] = 255
    return acc, err_mask


# In[258]:


#load the images, gt_map and convert to gray image
l_img = cv2.imread('im2.png')
l_img = cv2.cvtColor(l_img, cv2.COLOR_BGR2GRAY)

r_img = cv2.imread('im6.png')
r_img = cv2.cvtColor(r_img, cv2.COLOR_BGR2GRAY)

gt_img = cv2.imread('disp2.png')
gt_img = cv2.cvtColor(gt_img,cv2.COLOR_BGR2GRAY)


# In[259]:


sc_gt_img = gt_img.astype(np.float32)
sc_gt_img = (gt_img/4).astype(np.uint8)


# In[260]:


#Using window 10 and dmax = 52
dmap_w10, dmap_w10_sc = census_transform(l_img, r_img, 10, 52)
acc, mask = acc_dmap(sc_gt_img, dmap_w10)
print("Accuracy",acc)

cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/mask_w10.jpg", mask)
cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/dmap_w10.jpg", dmap_w10_sc)


# In[261]:


#Using window 29 and dmax = 52
dmap_w29, dmap_w29_sc = census_transform(l_img, r_img, 29, 52)
acc, mask_29 = acc_dmap(sc_gt_img, dmap_w29)
print("Accuracy",acc)

cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/mask_w29.jpg", mask_29)
```

```
cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/dmap_w29.jpg", dmap_w29_sc)


# In[262]:


#Using window 45 and dmax = 52
dmap_w45, dmap_w45_sc = census_transform(l_img, r_img, 45, 52)
acc, mask_45 = acc_dmap(sc_gt_img, dmap_w45)
print("Accuracy",acc)

cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/mask_w45.jpg", mask_45)
cv2.imwrite("D:/Purdue/ECE661_CV/HW9/Outputs/dmap_w45.jpg", dmap_w45_sc)
```

*****