ECE 661 Computer Vision

Homework 9

Yi-Hong Liao | liao163@purdue.edu | PUID 0031850073

1. Logic

Implement projective stereo reconstruction, run the Loop and Zhang algorithm and calculate the disparity map of the given left and right image.

2. Step

Projective Stereo Reconstruction

- **Image Rectification**

i Manually select a set of corresponding points between the two images and estimate the fundamental matrix F using

$$A_i = (xx' \quad x'y \quad x' \quad y'x \quad y'y \quad y' \quad x \quad y \quad 1)$$
$$Af = 0$$

where f is the eight unknowns in F. We use homogeneous liner least square to solve f.

ii Estimate the left and right epipoles: e and e' (right and left null vector of F).

iii Obtain the initial estimate of the projection matrices in canonical form.

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

$$P' = \left[[\overrightarrow{e'}]_x F \middle| \overrightarrow{e'} \right], where\ [\overrightarrow{e'}]_x = \begin{bmatrix} 0 & -e'_z & e'_y \\ e'_z & 0 & -e'_x \\ -e'_y & e'_x & 0 \end{bmatrix}$$

iv Refine the matrix P' using the nonlinear optimization. Which is minimizing

$$cost = \sum_i \left(\left| \left| \overrightarrow{x_i} - \overrightarrow{\widehat{x_i}} \right| \right|^2 + \left| \left| \overrightarrow{x'_i} - \overrightarrow{\widehat{x'_i}} \right| \right|^2 \right)$$

where $\overrightarrow{\widehat{x'_i}}$ are the reprojected points. Projection reprojection will be explained later.

v Obtain the refine fundamental matrix F using the refined P' from last step using the relation in step iii.

vi Estimate the homography of the second image H' using the refined epipoles.

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

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

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

$$f = (e'_x - w/2) * cos(\theta) - \left(e'_y - h/2\right) * sin(\theta)$$
$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix}$$
$$T_2 = \begin{bmatrix} 1 & 0 & w/2 \\ 0 & 1 & h/2 \\ 0 & 0 & 1 \end{bmatrix}$$
$$H' = T_2 GRT$$

vii Compute initial $H_{initial} = GRT$ on epipole e. Then, calculate $H_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ by minimizing $cost = \sum_i (ax_i + by_i + c - x_i^2)$. Finally, $H = T_2 H_a H_{initial}$

viii Apply the homgraphies to the stereo pair to rectify images.

- **Interest Point Detection**
  i Use canny edge detector to extract edges in the images.
  ii Find the corresponding points in two images by searching on the same row and in a small number of adjoining rows with NCC metric.
- **Projective Reconstruction**
  Given corresponding image points $\vec{x} = (x, y)$ and $\overrightarrow{x'} = (x', y')$ and the camera projection matrices P and P'

$$P = \begin{bmatrix} \overrightarrow{P_1}^T \\ \overrightarrow{P_2}^T \\ \overrightarrow{P_3}^T \end{bmatrix}$$
$$A = \begin{bmatrix} x\overrightarrow{P_3}^T - \overrightarrow{P_1}^T \\ y\overrightarrow{P_3}^T - \overrightarrow{P_2}^T \\ x'\overrightarrow{P'_3}^T - \overrightarrow{P'_1}^T \\ y'\overrightarrow{P'_3}^T - \overrightarrow{P'_2}^T \end{bmatrix}$$

Solve the homogeneous equation $A\vec{X} = 0$.

Dense stereo matching

- Census Transform
  In the left image we have a $M \times M$ window center at pixel P. On the right we have the corresponding pixel $q = \left[p_x - d, p_y\right]$ at disparity d, $d \in 0, \ldots, d_{max}$. We compute a bitvector of size M^2, if the pixel intensity value is strictly greater than the center pixel value, we make that vector element 1. This gives us two bitvectors. Then, we perform XOR operation between the two bitvectors and compute the number of ones in the result bitvector. This is the data cost between the two pixels. We pick the disparity value that has the smallest cost. The disparity of each pixel form the disparity map.

3. Result

- **Task 1**

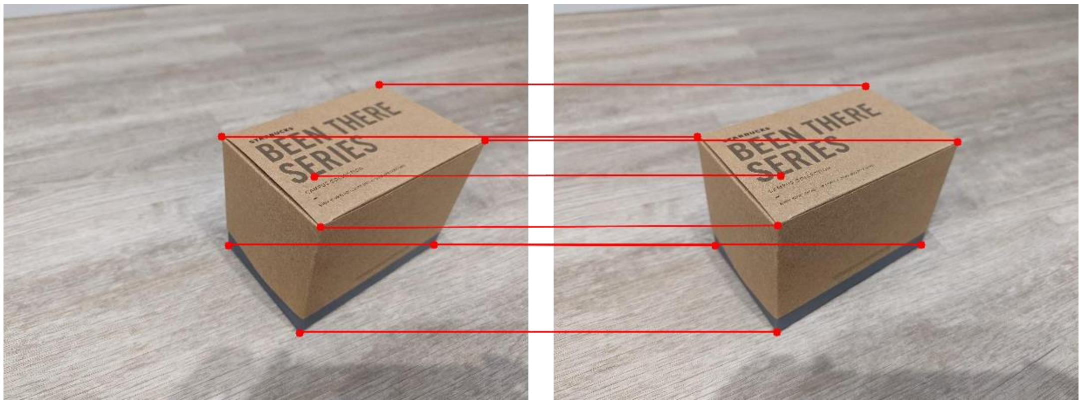

Figure 1: Manually selected points and the correspondence (hand draw).

Figure 2: Stereo images before image rectification.

Figure 3: Stereo images after image rectification.

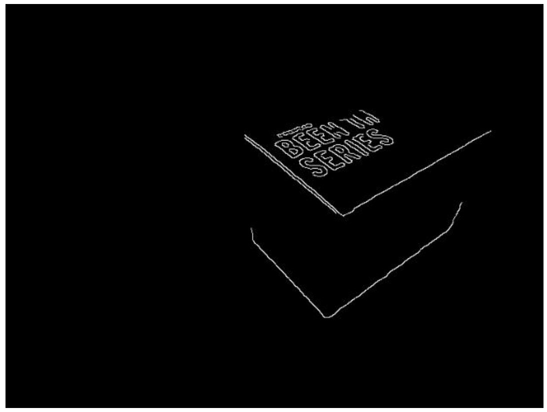

Figure 4: Canny extracted edges.

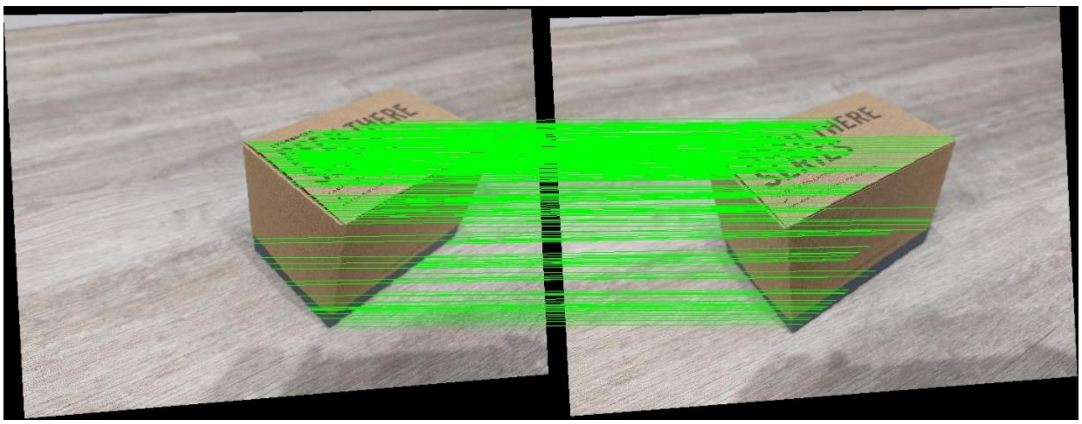

Figure 5: Correspondences based on the Canny points on the rectified pair.

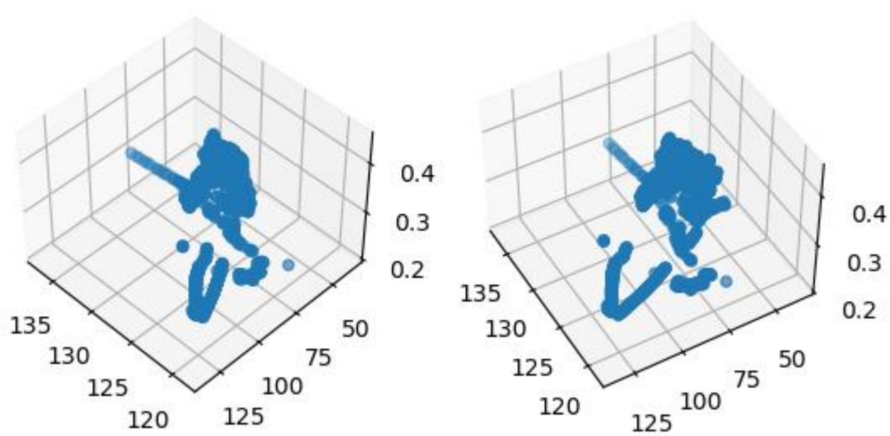

Figure 6: 3D reconstruction from two different views.

- **Task 2**
  Loop and Zhang algorithm decomposes the rectifying homographies into purely projective, similarity and shear homographies in order to minimize the distortion to the images.

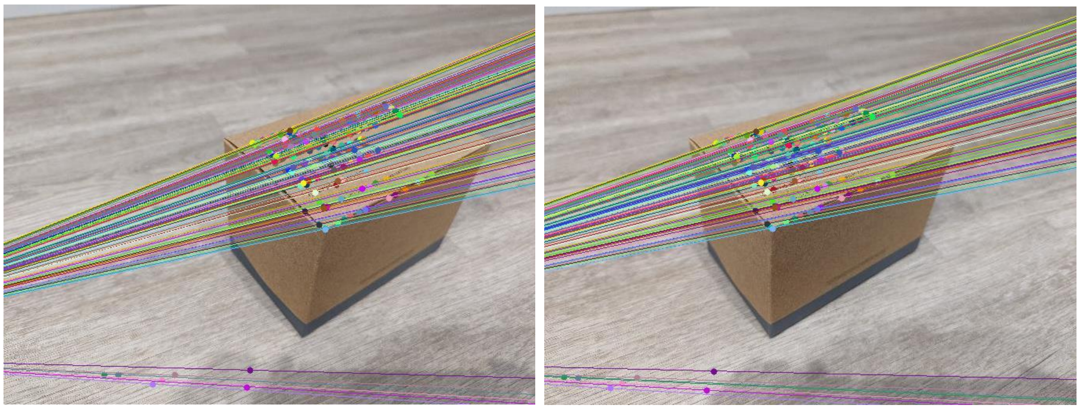

Figure 7: Stereo images before rectification.

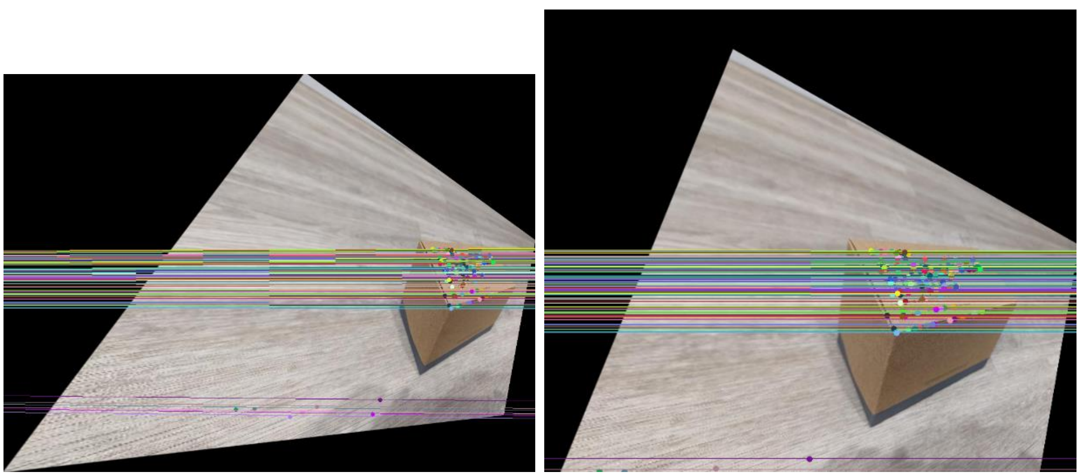

Figure 8: Stereo images after image rectification.

The images have been warpped so that the corresponding points are on the same row in both images, which means that the epipolar lines are parallel. The quality of the correspondences are better in the Loop and Zhang algorithm. The reason may be the number of correspondences used for image rectification and the correspondence matching criterion.

- **Task 3**

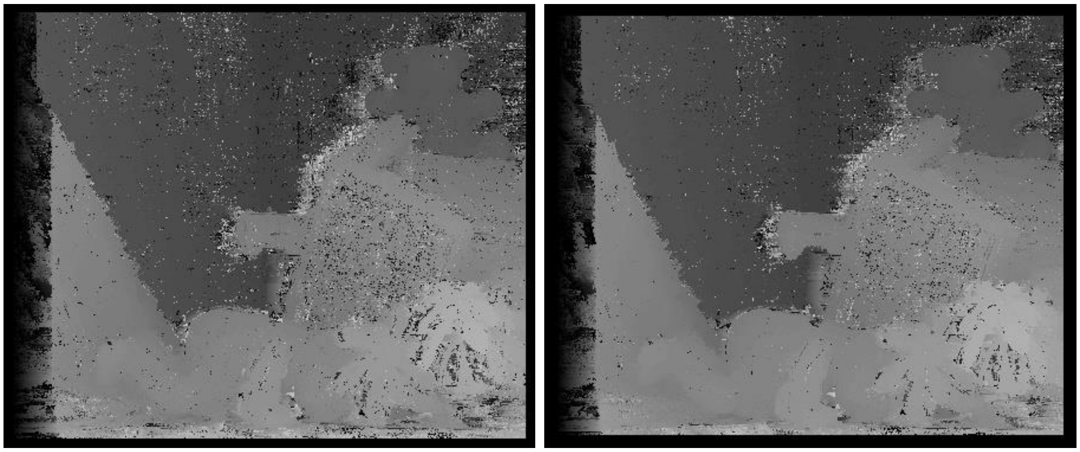

Figure 9: Estimated disparity maps for window size 15 (left) and 21 (right).

Figure 10: Error masks for window size 15 (left) and 21 (right).

The accuracy of window size = 15 is 0.7127, and the accuracy of window size = 21 is 0.7131. The output quality is slightly higher for larger local window size.

4. Source Code

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

# In[214]:


import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
from scipy.spatial import distance
from scipy import optimize
from scipy.linalg import null_space
from pathlib import Path


# In[215]:


def linear_est_F(pts1, pts2):
    n = len(pts1)
    A = np.zeros((n, 9))

    for i in range(n):
        x = pts1[i][0]
        y = pts1[i][1]
        xp = pts2[i][0]
        yp = pts2[i][1]
        A[i] = np.array ([xp*x, xp*y, xp, yp*x, yp*y, yp, x, y, 1])

    AT_A = np.transpose(A).dot(A)

    _, _, v = np.linalg.svd(AT_A)
    f = v[-1]
    F0 = np.reshape(f, (3, 3))

    u, s, vh = np.linalg.svd(F0)

    D = np.array ([[s[0] , 0, 0] , [0 , s[1] , 0], [0, 0, 0]])
    F = u.dot(D).dot(vh)

    return F

def find_epipole(F):
    e = null_space(F)
    ep = null_space(np.transpose(F))

    return e[:, 0]/ e[2, 0], ep[:, 0]/ ep[2, 0]

def triangulation(pt1, pt2, P, Pp):
    A = np.zeros((4, 4))
    x, y = pt1[0], pt1[1]
```

```
    xp, yp = pt2[0], pt2[1]

    A[0] = x*P[2, :] - P[0, :]
    A[1] = y*P[2, :] - P[1, :]
    A[2] = xp*Pp[2, :] - Pp[0, :]
    A[3] = yp*Pp[2, :] - Pp[1, :]

    AT_A = np.transpose(A).dot(A)

    _, _, v = np.linalg.svd(AT_A)
    X = v[-1]
    X = X/X[3]

    return X


def cost_func(params, P, pts1, pts2):

    Pp = np.reshape(params, (3, 4))

    errors = np.array([])
    for pt1, pt2 in zip(pts1, pts2):
        X = triangulation(pt1, pt2, P, Pp)

        pt1_hat = P.dot(X)
        pt2_hat = Pp.dot(X)

        pt1_hat = pt1_hat[0:2]/pt1_hat[2]
        pt2_hat = pt2_hat[0:2]/pt2_hat[2]

        error1 = pt1 - pt1_hat
        error2 = pt2 - pt2_hat

        errors = np.concatenate((errors, error1.flatten(),
error2.flatten()))

    return errors


def nonlinear_est_P_Pp(F, e, ep, pts1, pts2):
    #Compute camera projection matrices
    P = np.hstack((np.identity(3), np.zeros((3, 1))))

    epx = np.array([[0, -ep[2], ep[1]], [ep[2], 0, -ep[0]], [-ep[1],
ep[0], 0]])
    Pp0 = np.hstack((epx.dot(F), np.transpose([ep])))

    # refine Pp
    params = Pp0.flatten()
#     cost_func(params, P, pts1, pts2)
    params_nonlinear = optimize.least_squares(cost_func, params, args =
(P, pts1, pts2), method = 'lm').x

    Pp = np.reshape(params_nonlinear, (3, 4))

    return P, Pp
```

```
def nonlinear_est_F(F, e, ep, pts1, pts2):

    P, Pp = nonlinear_est_P_Pp(F, e, ep, pts1, pts2)

    ep = Pp[:, 3]
    epx = np.array([[0, -ep[2], ep[1]], [ep[2], 0, -ep[0]], [-ep[1],
ep[0], 0]])
    P_plus = np.transpose(P).dot(np.linalg.inv(P.dot(np.transpose(P))))
    F = epx.dot(Pp).dot(P_plus)

    return F

def find_rectify_H_Hp(img1, img2, pts1, pts2):
    h, w = img1.shape[0], img1.shape[1]

    F_l = linear_est_F(pts1, pts2)
    e_l, ep_l = find_epipole(F_l)


    F_nl = nonlinear_est_F(F_l, e_l, ep_l, pts1, pts2)
    e_nl, ep_nl = find_epipole(F_nl)
#     print(ep_nl)
#     print(e_nl)

    # compute H prime
############################################################
    T = np.array([[1, 0, -w/2],                   [0, 1, -h/2],
[0, 0,    1]])

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

#     print(R.dot(T).dot(ep_nl))

    f = (ep_nl[0] - w/2)*math.cos(theta) - (ep_nl[1] -
h/2)*math.sin(theta)

    G = np.array([[   1, 0, 0],                   [   0, 1, 0],
[-1/f, 0, 1]])
#     print(G.dot(R).dot(T).dot(ep_nl))

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

    Hp = T2.dot(G).dot(R).dot(T)
    Hp = Hp / Hp[2, 2]
#     print(Hp.dot(ep_nl))

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

    theta = math.atan2(e_nl[1] - h/2, -(e_nl[0] - w/2))
```

```
    R = np.array([[math.cos(theta), -math.sin(theta), 0],
[math.sin(theta), math.cos(theta), 0],                   [              0,
0, 1]])

#     print(R.dot(T).dot(e_nl))

    f = (e_nl[0] - w/2)*math.cos(theta) - (e_nl[1] - h/2)*math.sin(theta)

    G = np.array([[   1, 0, 0],                   [   0, 1, 0],
[-1/f, 0, 1]])

    H_hat = G.dot(R).dot(T)
#     print(H_hat.dot(e_nl))

    pts1_h = np.transpose(np.insert(pts1, 2, 1, axis=1))
    pts2_h = np.transpose(np.insert(pts2, 2, 1, axis=1))

    pts1_h_hat = H_hat.dot(pts1_h)
    pts2_h_hat = Hp.dot(pts2_h)

    pts1_h_hat = pts1_h_hat/pts1_h_hat[2, :]
    pts2_h_hat = pts2_h_hat/pts2_h_hat[2, :]

    pts1_h_hat = pts1_h_hat.T
    pts2_h_hat = pts2_h_hat.T

    abc = np.dot(np.linalg.pinv(pts1_h_hat), pts2_h_hat[: , 0])
    HA = np.array([[abc[0], abc[1], abc[2]], [0, 1, 0], [0, 0, 1]])
    H0 = HA.dot(H_hat)

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

    H = T2.dot(H0)
    H = H / H[2, 2]

    return H, Hp

def create_transformed_image(img, ROI_p, H, ds):
    h = img.shape[0]
    w = img.shape[1]

    new_ROI_p = np.zeros(shape=(ROI_p.shape[0], 3), dtype='int32')

    for i in range(ROI_p.shape[0]):
        pointH = np.array([ROI_p[i, 0], ROI_p[i, 1], 1])
        pointHp = H.dot(pointH)
        pointHp = pointHp/pointHp[2]
        pointHp = np.around(pointHp).astype(int)
        new_ROI_p[i, :] = pointHp

    ROI_max = np.amax(new_ROI_p, axis=0)
    ROI_min = np.amin(new_ROI_p, axis=0)

    new_ROI_p_offset = new_ROI_p
    new_ROI_p_offset[:, 0] = new_ROI_p_offset[:, 0]-ROI_min[0]
    new_ROI_p_offset[:, 1] = new_ROI_p_offset[:, 1]-ROI_min[1]
```

```
    ho = ROI_max[1]-ROI_min[1]+1
    wo = ROI_max[0]-ROI_min[0]+1
    print("new image height and width", ho, wo)
    imgOut = np.zeros([np.int32(np.floor(ho/ds))+1,
np.int32(np.floor(wo/ds))+1, 3], dtype='uint8')
    new_ROI = np.zeros([ho, wo])
    cv2.fillPoly(new_ROI, pts = np.int32([new_ROI_p_offset[:, 0:2]]),
color = 255)

    for i in range(0, wo, ds):
        for j in range(0, ho, ds):
            if new_ROI[j,i] > 0:
                pointH = np.array([i+ROI_min[0], j+ROI_min[1], 1])
                pointHp = np.linalg.inv(H).dot(pointH)
                pointHp = pointHp/pointHp[2]
                pointHp = np.around(pointHp).astype(int)
                if pointHp[0] < w and pointHp[0] >= 0 and pointHp[1] < h
and pointHp[1] >= 0:
                    imgOut[np.int32(np.floor(j/ds)),
np.int32(np.floor(i/ds))] = img[pointHp[1], pointHp[0]]

    return imgOut, -ROI_min[0:2]

def create_rectified_pair(img1, img2, offset1, offset2):
    h1, w1 = img1.shape[0], img1.shape[1]
    h2, w2 = img2.shape[0], img2.shape[1]
    min_y = min(-offset1[1], -offset2[1])
    max_y = max(h1-offset1[1], h2-offset2[1])
    new_h = max_y-min_y+1

    imgOut = np.zeros([np.int32(new_h), np.int32(w1+w2), 3],
dtype='uint8')

    if offset1[1] < offset2[1]:
        imgOut[offset2[1]-offset1[1]:offset2[1]-offset1[1]+h1, 0:w1] =
img1
        imgOut[0:h2, w1:] = img2
    else:
        imgOut[0:h1, 0:w1] = img1
        imgOut[offset1[1]-offset2[1]:offset1[1]-offset2[1]+h2, w2:] = img2

    return imgOut

def extract_edges(image):
    h, w = image.shape[0], image.shape[1]
    img = cv2.GaussianBlur(image, (5, 5), 5/3)

    img[:, 0] = 0
    img[:, -1] = 0
    img[0, :] = 0
    img[-1, :] = 0

    kernel = np.ones((5, 5), np.uint8)
    img_dilation = cv2.erode(img, kernel, iterations=10)

    ret, bw_img = cv2.threshold(img_dilation, 1, 255, cv2.THRESH_BINARY)
```

```
    edges = cv2.Canny(img, 200, 100, 5)

    return edges*bw_img*255

def draw_corr_points(img1, img2, pts1, pts2, offset1, offset2):
    h1, w1 = img1.shape[0], img1.shape[1]
    h2, w2 = img2.shape[0], img2.shape[1]
    min_y = min(-offset1[1], -offset2[1])
    max_y = max(h1-offset1[1], h2-offset2[1])
    new_h = max_y-min_y+1

    imgOut = np.zeros([np.int32(new_h), np.int32(w1+w2), 3],
dtype='uint8')
    color = (0, 255, 0)

    if offset1[1] < offset2[1]:
        imgOut[offset2[1]-offset1[1]:offset2[1]-offset1[1]+h1, 0:w1] =
img1
        imgOut[0:h2, w1:] = img2
        for pt1, pt2 in zip(pts1[::10], pts2[::10]):
            #convert data types int64 to int
            cv2.line(imgOut, (pt1[0]+offset1[0], pt1[1]+offset2[1]),
(pt2[0]+offset2[0]+w1, pt2[1]+offset2[1]), color, 1)
    else:
        imgOut[0:h1, 0:w1] = img1
        imgOut[offset1[1]-offset2[1]:offset1[1]-offset2[1]+h2, w2:] = img2
        for pt1, pt2 in zip(pts1[::10], pts2[::10]):
            #convert data types int64 to int
            cv2.line(imgOut, (pt1[0]+offset1[0], pt1[1]+offset1[1]),
(pt2[0]+offset2[0]+w1, pt2[1]+offset1[1]), color, 1)

    return imgOut

def calculate_NCC(point1, img1, point2, img2, win_size):
    h = img1.shape[0]
    w = img1.shape[1]
    hwin_size = int(np.floor(win_size/2))

    img1p = cv2.copyMakeBorder(img1, hwin_size, hwin_size, hwin_size,
hwin_size, cv2.BORDER_CONSTANT, 0);
    img2p = cv2.copyMakeBorder(img2, hwin_size, hwin_size, hwin_size,
hwin_size, cv2.BORDER_CONSTANT, 0);

    sec1 = img1p[point1[1]:point1[1]+win_size,
point1[0]:point1[0]+win_size]
    sec2 = img2p[point2[1]:point2[1]+win_size,
point2[0]:point2[0]+win_size]

    m1 = np.mean(sec1)
    m2 = np.mean(sec2)

    sec1m = sec1 - m1
    sec2m = sec2 - m2

    NCC = np.sum(sec1m*sec2m)/np.sqrt(np.sum(sec1m**2)*np.sum(sec2m**2))
```

```
    return NCC

def find_corr_point(pt1, img1, img2, offset1, offset2, disp_range):
    h2, w2 = img2.shape[0], img2.shape[1]
    adjs = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
#     adjs = [0]
    offset = offset1-offset2
    max_ncc = -100
    min_ssd = (255**2)*(21**2)+1
    max_pt = [-1, -1]
    lower_bound = max(0, pt1[0]+disp_range[0])
    upper_bound = min(w2, pt1[0]+disp_range[1])

    for i in range(lower_bound, upper_bound):
        for adj in adjs:
            row = pt1[1]+offset[1]+adj
            if row >= 0 and row < h2:
                ncc = calculate_NCC(pt1, img1, [i, row], img2, 21)
                if ncc > max_ncc:
                    max_ncc = ncc
                    max_pt = [i, row]
    return max_pt

def find_correspondences(img1, img2, offset1, offset2, disp_range):
    h1, w1 = img1.shape[0], img1.shape[1]
    h2, w2 = img2.shape[0], img2.shape[1]

    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    points1 = []
    points2 = []
    edges1 = extract_edges(img1)
    cv2.imwrite("edge.jpg", edges1)
    test = 0

    for i in range(w1):
        for j in range(h1):
            if edges1[j, i] > 0:
                point2 = find_corr_point([i, j], img1, img2, offset1,
offset2, disp_range)
                if point2[0] != -1:
                    points1.append([i, j]-offset1)
                    points2.append(point2-offset2)
#                     print([i, j], point2)
#                     test += 1
#                     if test == 100:
#                         return points1, points2

    return points1, points2

def projective_reconstruction(pts1, pts2, H, Hp):
    pts1_h = np.transpose(np.insert(pts1, 2, 1, axis=1))
    pts2_h = np.transpose(np.insert(pts2, 2, 1, axis=1))

    pts1_h_hat = np.linalg.inv(H).dot(pts1_h)
    pts2_h_hat = np.linalg.inv(Hp).dot(pts2_h)
```

```
    pts1_h_hat = pts1_h_hat/pts1_h_hat[2, :]
    pts2_h_hat = pts2_h_hat/pts2_h_hat[2, :]

    pts1_h_hat = pts1_h_hat.T
    pts2_h_hat = pts2_h_hat.T

    pts1_hat = pts1_h_hat[:, 0:2]
    pts2_hat = pts2_h_hat[:, 0:2]

    F_l = linear_est_F(pts1_hat, pts2_hat)
    e_l, ep_l = find_epipole(F_l)

    P, Pp = nonlinear_est_P_Pp(F_l, e_l, ep_l, pts1_hat, pts2_hat)

    Xs = np.empty((0, 3))
    for pt1, pt2 in zip(pts1_hat, pts2_hat):
        X = triangulation(pt1, pt2, P, Pp)
        Xs = np.append(Xs, [X[0:3]], axis=0)

    return Xs

def draw_points(image, pts, color):
    for i, pt in enumerate(pts):
        image = cv2.circle(image, pt, radius=5, color=color, thickness=-1)
    return image

def census_disparity(img1, img2, d_max, win_size):
    h, w = img1.shape[0], img1.shape[1]
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    hw_size = int(np.floor(win_size/2))

    disparity = np.zeros((h , w))
    for j in range(hw_size, h-hw_size-1):
        for i in range(hw_size, w-hw_size-1):
            costs = []
            sec1 = img1[j-hw_size:j+hw_size, i-hw_size:i+hw_size]
            vec1 = np.ravel((sec1 > img1[j, i])*1)
            for m in range(i, i-d_max-1, -1):
                if m >= hw_size:
                    sec2 = img2[j-hw_size:j+hw_size, m-hw_size:m+hw_size]
                    vec2 = np.ravel((sec2 > img2[j, m])*1)
                    costs.append(sum(vec1 ^ vec2))
            disparity[j, i] = np.argmin(costs)
#         print(j)
    disparity.astype(np.uint8)
    return disparity

def calculate_disp_accuracy(disp, gt, delta):
    gt = cv2.cvtColor(gt, cv2.COLOR_BGR2GRAY)
    h, w = disp.shape[0], disp.shape[1]
    gt = gt.astype(float)
    gt = gt/4
    gt = gt.astype(np.uint8)
```

```
    N = 0
    n = 0
    mask = np.zeros((h, w))
    for i in range(w):
        for j in range(h):
            if gt[j, i] > 0:
                N += 1
                if abs(disp[j, i] - gt[j, i]) <= delta:
                    n += 1
                    mask[j, i] = 255

    return n/N, mask


# In[216]:


if __name__ == '__main__':
    path = Path("C:/Users/yhosc/Desktop/ECE661/HW9/")
    outputPath = Path("C:/Users/yhosc/Desktop/ECE661/HW9")

    img1 = cv2.imread(str(path / "left.jpg"))
    img2 = cv2.imread(str(path / "right.jpg"))

    points1  = [[282, 171], [291, 312], [383, 425], [410, 288], [486, 
104], [623, 176], [557, 311], [402, 223]]
    points2 = [[186, 171], [209, 312], [289, 424], [290, 286], [404, 106], 
[523, 178], [476, 311], [294, 222]]

    H, Hp = find_rectify_H_Hp(img1, img2, points1, points2)

    ROI = np.array([[0, 0, 1], [0, 511, 1], [679, 511, 1], [679, 0, 1]])
    rect1, offset1 = create_transformed_image(img1, ROI, H, 1)
    print(offset1)
    cv2.imwrite("rect1.jpg", rect1)

    rect2, offset2 = create_transformed_image(img2, ROI, Hp, 1)
    print(offset2)
    cv2.imwrite("rect2.jpg", rect2)


# In[217]:


img1_man = draw_points(img1, points1, (0, 0, 255))
cv2.imwrite("img1_man.jpg", img1_man)
img2_man = draw_points(img2, points2, (0, 0, 255))
cv2.imwrite("img2_man.jpg", img2_man)


# In[218]:


rect_pair = create_rectified_pair(rect1, rect2, offset1, offset2)
cv2.imwrite("rect_pair.jpg", rect_pair)
```

```
# In[219]:


corr_pts1, corr_pts2 = find_correspondences(rect1, rect2, offset1,
offset2, [-150, -100])
print("done")
#     print(corr_pts1)
#     print(corr_pts2)


# In[220]:


corr = draw_corr_points(rect1, rect2, corr_pts1, corr_pts2, offset1,
offset2)
cv2.imwrite("correspondence.jpg", corr)


# In[221]:


Xs = projective_reconstruction(corr_pts1, corr_pts2, H, Hp)
#     print(Xs)
fig = plt.figure()
ax = fig.add_subplot(121, projection='3d')
ax.scatter(Xs[:, 0], Xs[:, 1], Xs[:, 2])
ax.view_init(45, 135)
ax2 = fig.add_subplot(122, projection='3d')
ax2.scatter(Xs[:, 0], Xs[:, 1], Xs[:, 2])
ax2.view_init(45, 150)
plt.savefig('3D.png')
plt.show()


# In[222]:


img1_provide = cv2.imread(str(path / "Task3Images/Task3Images/im2.png"))
img2_provide = cv2.imread(str(path / "Task3Images/Task3Images/im6.png"))

disp_15 = census_disparity(img1_provide, img2_provide, 50, 15)
disp_21 = census_disparity(img1_provide, img2_provide, 50, 21)
cv2.imwrite("disp_15.jpg", disp_15*4)
cv2.imwrite("disp_21.jpg", disp_21*4)
print("done")


# In[223]:


disp_gt = cv2.imread(str(path / "Task3Images/Task3Images/disp2.png"))
accuracy_15, mask_15 = calculate_disp_accuracy(disp_15, disp_gt, 2)
accuracy_21, mask_21 = calculate_disp_accuracy(disp_21, disp_gt, 2)
cv2.imwrite("mask_15.jpg", mask_15)
cv2.imwrite("mask_21.jpg", mask_21)

print("window size: 15: ", accuracy_15)
```

```
print("window size: 21: ", accuracy_21)
```