ECE661 Computer Vision HW9

Chengjun Guo
guo456@purdue.edu

December 2022

1 Logic

1.1 Projective Reconstruction

1.1.1 Image Rectification

1. For the two images, select 8 points in each image for the initial F calculation.
2. Normalize points. 8-points algorithm is sensitive to origin of coordinates and scale. It will translate so the centroid is at origin. The image coordinates will be transformed by:

$$\hat{x_i} = Tx_i$$

For denormalization, the F will become:

$$F = T'\hat{F}'T$$

where $\hat{x_i}'\hat{F}\hat{x_i} = 0$.
T can be calculated by:

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

where $scale = \frac{\sqrt{2}}{\bar{D}}$.
$\bar{D}$ can be calculated by finding the mean of list D where each element can be calculated by $D_i = \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2}$.
3. With the normalized points, we can form the matrix A for the calculation of f vector. f is constructed by reshaping $\hat{F}$.

$$Af = \begin{bmatrix} \hat{x_1}\hat{x_1}' & \hat{x_1}'\hat{y_1} & \hat{x_1}' & \hat{y_1}'\hat{x_1} & \hat{y_1}'\hat{y_1} & \hat{y_1}' & \hat{x_1} & \hat{y_1} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hat{x_n}\hat{x_n}' & \hat{x_n}'\hat{y_n} & \hat{x_n}' & \hat{y_n}'\hat{x_n} & \hat{y_n}'\hat{y_n} & \hat{y_n}' & \hat{x_n} & \hat{y_n} & 1 \end{bmatrix} \begin{bmatrix} \hat{F}_{11} \\ \hat{F}_{12} \\ \hat{F}_{13} \\ \hat{F}_{21} \\ \hat{F}_{22} \\ \hat{F}_{23} \\ \hat{F}_{31} \\ \hat{F}_{32} \\ \hat{F}_{33} \end{bmatrix} = 0$$

where n is the number of corresponding pairs.
4. Constrain F: In order to make F rank 2, use SVD to decompose F and set the last eigenvalue to 0.
5. The initial guess of $\overrightarrow{e}$ and $\overrightarrow{e}'$ are the left and right null vector of F. It can be calculated since the F is conditioned.
6. we set the projection of left image P as $[I_{3\times3}|0]$. The right image projective matrix P' will be $[[\overrightarrow{e}']_x F|\overrightarrow{e}']$ where

$$[\overrightarrow{e}']_x = \begin{bmatrix} 0 & -\overrightarrow{e_3}' & \overrightarrow{e_2}' \\ -\overrightarrow{e_3}' & 0 & -\overrightarrow{e_1}' \\ -\overrightarrow{e_2}' & \overrightarrow{e_1}' & 0 \end{bmatrix}$$

.
6. With the Levenberg-Marquardt method, we can refine the P' with the cost function:

$$d^2_{\text{geom}} = \sum \left(\|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 + \|\mathbf{x}'_i - \hat{\mathbf{x}}'_i\|^2 \right)$$

where $\mathbf{x}_i$ and $\mathbf{x}'_i$ is the ground truth, $\hat{\mathbf{x}}_i$ and $\hat{\mathbf{x}}'_i$ is the projected points.
7. With the refined P', F can be calculated by

$$F = [\overrightarrow{e}']_x P' P^T (PP^T)^{-1}$$

and condition F as the previous step.
8. Then the homography for both left and right image would be calculated. The right image homography would be calculated first by $H' = T_2 G R T_1$.
T_1 and T_2 are the translation matrices that send the center to origin and send it back to the center.

$$T_1 = \begin{bmatrix} 1 & 0 & -w/2 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1 \end{bmatrix} T_2 = \begin{bmatrix} 1 & 0 & w/2 \\ 0 & 1 & h/2 \\ 0 & 0 & 1 \end{bmatrix}$$

R is rotation matrix.

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

where θ can be calculated by:

$$\theta = tan^{-1}(\frac{\overrightarrow{e_2'} - h/2}{-\overrightarrow{e_1'} - w/2})$$

G is the transformation matrix to send the epipole to infinity:

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

where f is calculated by:

$$f = (-\overrightarrow{e_1'} - w/2)cos(\theta) - (\overrightarrow{e_2'} - h/2)sin(\theta)$$

Then we will calculate the homography for the right image. We first get $H_0 = H'P'P^T(PP^T)^{-1}$. Then we calculate the a,b,c by minimizing

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

Then we form H_A by

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

Then we will have H for the left image by $H = H_A H_0$.

1.1.2 Interest Point Detection

For the two images, we first convert them to gray and gaussian blur them. Then we use canny to extract the rectified edge. In case of the error, I search pixels less than 10 pixels far from the point in the left in the points in the right for the correspondences. With SSD, we filter the corresponding points in each images. Then we can find the point pairs.

1.1.3 Projective Reconstruction

For the Projective reconstruction, the world coordinate X can be calculated by:

$$AX = \begin{bmatrix} x\vec{P_3}^T - \vec{P_1}^T \\ y\vec{P_3}^T - \vec{P_2}^T \\ x'\vec{P_3}^T - \overrightarrow{P_1'}^T \\ y'\vec{P_3'} - \overrightarrow{P_2'}^T \end{bmatrix} X = 0$$

X can be calculated by the smallest eigenvector of $A^T A$.

1.1.4 3D Visual Inspection

In this section, we can plot the 3D with matplotlib function.

1.2 Loop and Zhang Algorithm

In the loop and zhang algorithm, there will be 3 homography matrices for different purpose to make the lines parallel to each other. One of the homography is for rotate and translate the image in the same scale. Then another homography is to make the epipolar lines parallel to each other. The rest of homography is to send the epipoles to infinity as we did in the normal 8point algorithm. With the extra degree of freedom, we can reduce the projection distortion.

1.3 Dense Stereo Matching

For pixel in the left image, we get the values of the pixels in M by M window around it.(It doesn't have to be Square, M by N is fine). Then we can get a list of M by M windows around the pixel from (x,y) to (x-dMax,y) where x,y is the coordinate of the pixel selected in the left image. Flatten the windows and denote 1 for each pixel have larger value than the pixel in the center while 0 means smaller. Do a element wise XOR operation to the windows of left with the list of windows in right image and the cost is the number of the 1s. The index of the minimum value of the cost vector will be the d associated with the pixel in the left image. The disparity map will be filled with value d while we loop through each point. In case of out of the indices, we pad the image first. For the error less than $\delta = 2$, we fill 255 to the mask.

2 Image and Data

Original inputs:

(a) left

(b) right

8 points:

(a) left

(b) right

Rectified images:

(a) left

(b) right

Canny edges:

(a) left (b) right

Correspondences of rectified images:

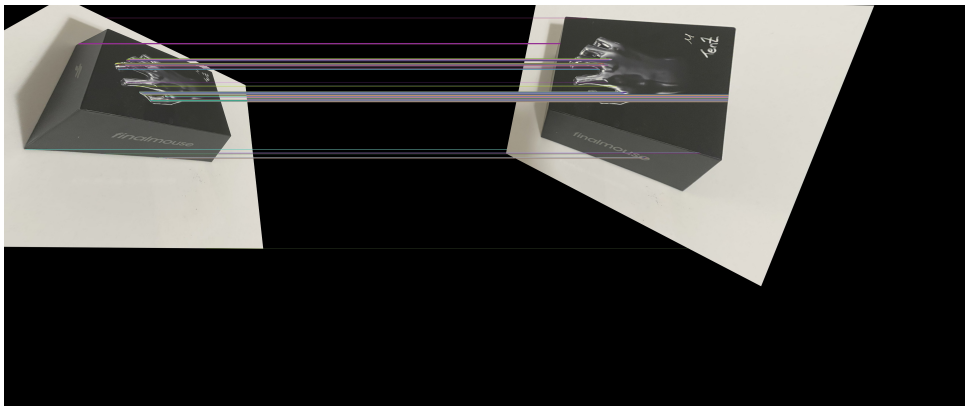
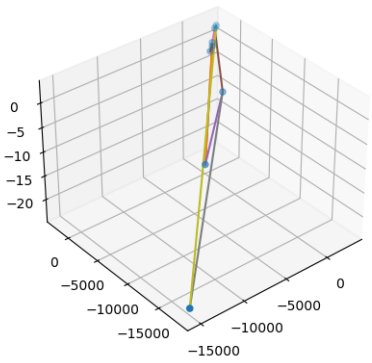
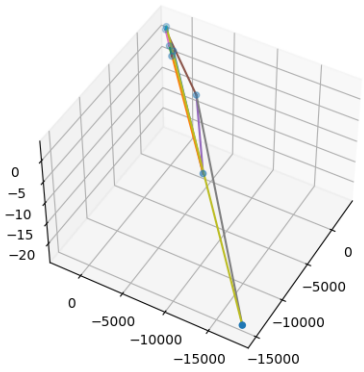


Figure 5: interesting point pairs

3D reconstruction without interesting points:

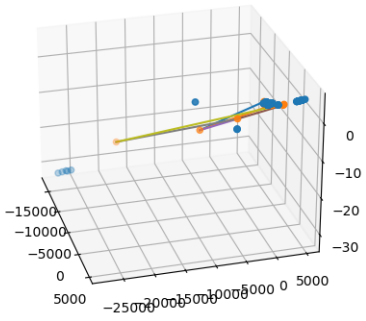


(a) 1

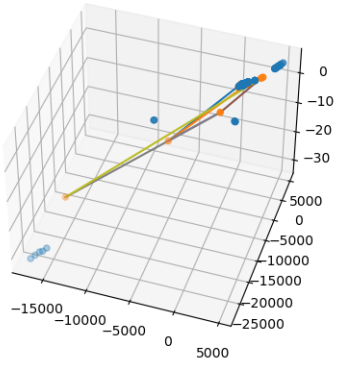


(b) 2

3D reconstruction with interesting points: Some of the points of the edge transferring from white image to the black padding are transformed to world coordinates



(a) 1



(b) 2

Loop and zhang algorithm output:

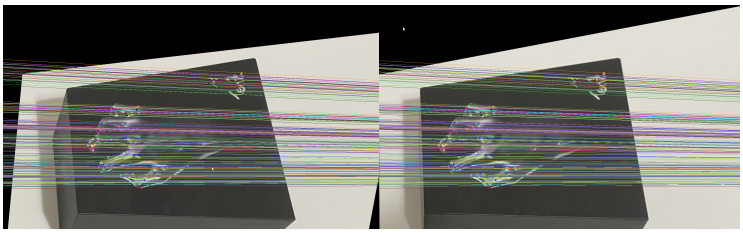


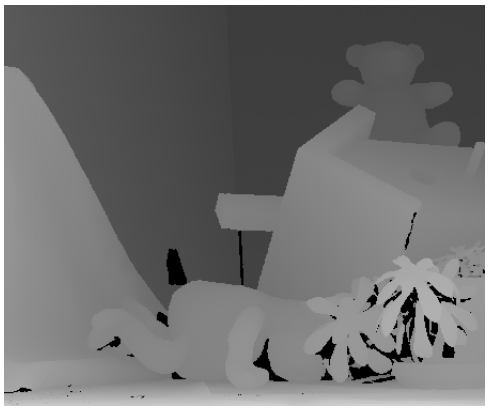
Figure 8: loop and zhang output

Dense Stereo Matching original input:

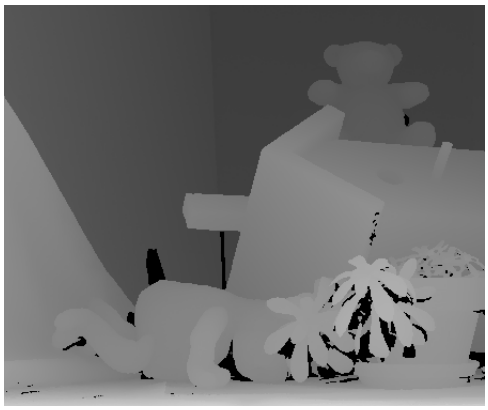
(a) left

(b) right

Dense Stereo Matching ground truth:



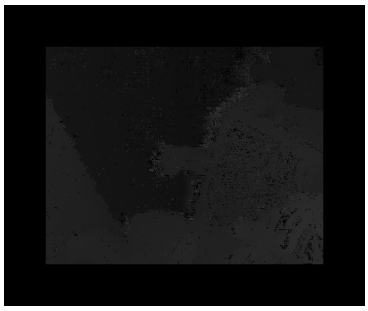
(a) left



(b) right

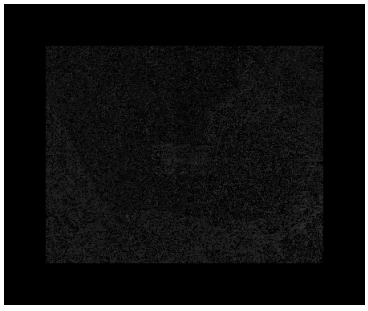
Disparity map:

(a) M=31



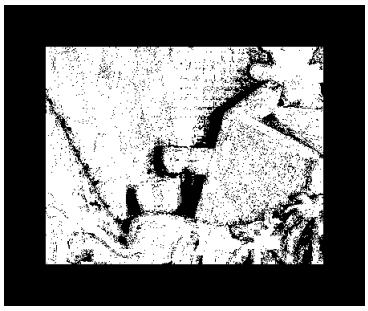
(b) M=15

(c) M=7

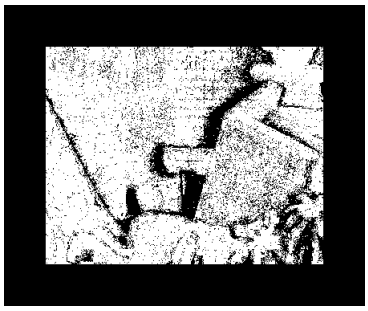


(d) M=3

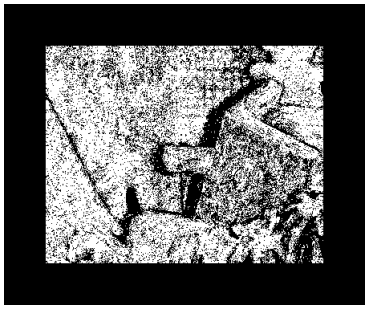
Mask:



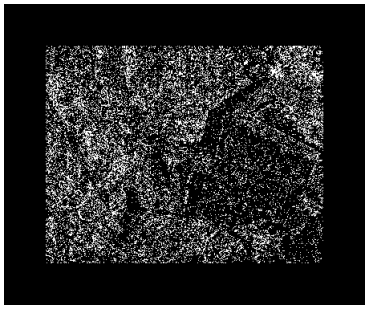
(a) M=31



(b) M=15



(c) M=7



(d) M=3

The accuracy with $\delta <= 2$:

M	accuracy
3	86.91%
7	63.53%
15	55.68%
31	54.33%

3 Observation

3.1 Discussion on loop and zhang algorithm

The result of Loop and Zhang algorithm is better than the normalized 8 point algorithm. With more degree of freedom, the projection have less error. In the practice output, the filter algorithm of their implementation filtered out the edges of the padded rectified image. Also, Comparing to the rectified left image of loop and zhang algorithm, the output of normalized 8 point algorithm is projective to a different way which even if it make the epipoles parallel, it looks not as good as the loop and zhang algorithm.

3.2 discussion on dense stereo matching

With larger and larger window size get, the image is less noisy and closer to the original ground truth. The accuracy change slower as the window size becomes larger.

4 Code

```
import math

import cv2
import numpy as np
import os
os.environ["OMP_NUM_THREADS"] = '1'
import sklearn
import skimage
from sklearn import metrics
from sklearn.cluster import KMeans,DBSCAN
from skimage import data, io, filters, transform,color
from matplotlib import pyplot as plt
from scipy.optimize import least_squares
```

```
def normalize_points(pts):
    pts = np.array(pts)
    mean_x = np.mean(pts[:,0])
    mean_y = np.mean(pts[:,1])
    dist = []
    np.mean(np.linalg.norm(pts-[mean_x,mean_y],axis=1))
    for i in range(len(pts)):
        dist.append(np.sqrt((pts[i][0]-mean_x)**2 + (pts[
            i][1]-mean_y) ** 2))
    mean_dist = np.mean(dist)
    scale = np.sqrt(2) / mean_dist
    T = np.zeros((3,3))
    T[0,0] = scale
    T[0,2] = -scale * mean_x
    T[1,1] = scale
    T[1,2] = -scale * mean_y
    T[2,2] = 1
    return (pts-[mean_x,mean_y]) * scale, T

def calculate_F(pts1,pts2):
    A = np.ones((len(pts1), 9))
    for i in range(len(pts1)):
        A[i][0] = pts2[i][0] * pts1[i][0]
        A[i][1] = pts2[i][0] * pts1[i][1]
        A[i][2] = pts2[i][0]
        A[i][3] = pts2[i][1] * pts1[i][0]
        A[i][4] = pts2[i][1] * pts1[i][1]
        A[i][5] = pts2[i][1]
        A[i][6] = pts1[i][0]
        A[i][7] = pts1[i][1]
    _, _, v = np.linalg.svd(A)
    f = v[-1].reshape((3,3))
    #condition F
    U, s, VT = np.linalg.svd(f)
    s[2] = 0
    F = np.dot(U, np.dot(np.diag(s), VT))
    return F


def get_epipoles(F):
    u, _, v = np.linalg.svd(F)
    e1 = v[-1].T
    e2 = u[:,-1]
    return e1/e1[2], e2/e2[2]
```

```
def cost_function(p,pts1,pts2):
    P1 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1,
        0]])
    P2 = p.reshape((3,4))
    pts1 = np.hstack((np.array(pts1), np.ones((8, 1))))
    pts2 = np.hstack((np.array(pts2), np.ones((8, 1))))
    gt = np.hstack((pts1.T, pts2.T))
    error = []
    for i in range(8):
        A = np.zeros((4,4))
        A[0] = pts1[i, 0] * P1[2] - P1[0]
        A[1] = pts1[i, 1] * P1[2] - P1[1]
        A[2] = pts2[i, 0] * P2[2] - P2[0]
        A[3] = pts2[i, 1] * P2[2] - P2[1]
        _, _, vT = np.linalg.svd(A)
        v = vT[-1]
        vec = v/v[-1]
        x1 = np.dot(P1, vec)
        x2 = np.dot(P2, vec)
        x1 = x1 / x1[2]
        x2 = x2 / x2[2]
        error.append(np.linalg.norm((x1 - pts1[i]) ** 2))
        error.append(np.linalg.norm((x2 - pts2[i]) ** 2))
    error = np.array(error)
    return error.flatten()



def calc_homography(img1,img2,e1,e2,P1,P2,pts1,pts2):
    # reference: https://engineering.purdue.edu/RVL/
        ECE661_2020/Homeworks/HW10/2BestSolutions/2.pdf
    h2, w2 = img2.shape[0],img2.shape[1]
    T1 = np.array([[1,0,-w2/2],[0,1,-h2/2],[0,0,1]])
    # rotation matrix R
    angle = -np.arctan((e2[1]-h2/2) / (e2[0]-w2/2))
    R = np.array([[math.cos(angle), -math.sin(angle),
        0],[math.sin(angle), math.cos(angle), 0],[0,0,1]])
    # G epipoles to infinity
    f = np.linalg.norm(np.array([e2[1] - h2/2,e2[0] - w2
        /2]))
    G = np.array([[1,0,0],[0,1,0],[-1/f,0,1]])
    # T2
    # img2center = np.dot(np.dot(G,np.dot(R,T1)), np.
        array([w2/2, h2/2, 1]))
    # img2center = img2center/img2center[2]
```

```
    # T2 = np.array([[1,0,w2/2 - img2center[0]],[0,1,h2/2
        - img2center[1]],[0,0,1]])
    T2 = np.array([[1, 0, w2 / 2], [0, 1, h2 / 2 ], [0,
        0, 1]])
    H2 = np.dot(T2,np.dot(G,np.dot(R,T1)))
    H2 = H2/H2[2,2]
    #homography of left
    M = np.dot(P2,np.dot(P1.T,np.linalg.inv(np.dot(P1,P1.
        T))))
    H0 = np.dot(H2, M)
    pts1 = np.hstack((np.array(pts1), np.ones((8, 1))))
    pts2 = np.hstack((np.array(pts2), np.ones((8, 1))))
    pts1h = np.dot(H0, pts1.T).T
    pts2h = np.dot(H2, pts2.T).T
    pts1h = pts1h / pts1h[:,2].reshape(-1,1)
    pts2h = pts2h / pts2h[:,2].reshape(-1,1)
    ha = np.dot(np.linalg.pinv(pts1h),pts2h[:,0])
    HA = np.array([[ha[0],ha[1],ha[2]],[0,1,0],[0,0,1]])
    H1 = np.dot(HA,H0)
    # H1 = np.dot(T2,H1)
    H1 = H1/H1[2,2]
    return H1,H2


def bilinear_interpolation(y, x, image):
    # based on wikipedia formula
    X_Q11 = image[int(math.floor(y)), int(math.floor(x))]
    X_Q12 = image[int(math.floor(y)), int(math.ceil(x))]
    X_Q22 = image[int(math.ceil(y)), int(math.ceil(x))]
    X_Q21 = image[int(math.ceil(y)), int(math.floor(x))]
    X_x_y1 = (math.ceil(y) - y) * X_Q11 + (y - math.floor
        (y)) * X_Q21
    X_x_y2 = (math.ceil(y) - y) * X_Q12 + (y - math.floor
        (y)) * X_Q22

    result = (math.ceil(x) - x) * X_x_y1 + (x - math.
        floor(x)) * X_x_y2
    return result


def point_mapping_complementary(Image_X, Image_X_prime, H
    ):
    # finishing the full picture, includes blank area
    H_inv = np.linalg.inv(H)
    tl_prime = np.dot(H, np.array([0, 0, 1]))
```

```
tr_prime = np.dot(H, np.array([len(Image_X_prime[0])
    - 1, 0, 1]))
bl_prime = np.dot(H, np.array([0, len(Image_X_prime)
    - 1, 1]))
br_prime = np.dot(H, np.array([len(Image_X_prime[0])
    - 1, len(Image_X_prime) - 1, 1]))
tl_point = [int(tl_prime[0] / tl_prime[2]), int(
    tl_prime[1] / tl_prime[2])]
tr_point = [int(tr_prime[0] / tr_prime[2]), int(
    tr_prime[1] / tr_prime[2])]
bl_point = [int(bl_prime[0] / bl_prime[2]), int(
    bl_prime[1] / bl_prime[2])]
br_point = [int(br_prime[0] / br_prime[2]), int(
    br_prime[1] / br_prime[2])]
x_max = max(tr_point[0], br_point[0])
x_min = min(tl_point[0], bl_point[0])
y_max = max(bl_point[1], br_point[1])
y_min = min(tl_point[1], tr_point[1])
print(x_max, x_min, y_max, y_min)
x_offset = int(0 - x_min)
y_offset = int(0 - y_min)
shape = (int(y_max - y_min), int(x_max - x_min),
    Image_X_prime.shape[2])
area = np.zeros(shape)
points = [tl_point, bl_point, br_point, tr_point]
for pt in points:
    pt[0] += x_offset
    pt[1] += y_offset
image_to_fill = cv2.fillPoly(area, pts=[np.array(
    points, dtype=np.int32)], color=(255, 255, 255))
print(tl_point, bl_point, br_point, tr_point)
# cv2.imshow("image_X_prime filled", image_to_fill)
for i in range(len(image_to_fill)):
    for j in range(len(image_to_fill[i])):
        if np.all(image_to_fill[i, j] == 255):
            point_in_x_prime = np.array([j - x_offset
                , i - y_offset, 1])
            point_in_x = np.dot(H_inv,
                point_in_x_prime)
            x = point_in_x[0] / point_in_x[2]
            y = point_in_x[1] / point_in_x[2]
            if y < 0:
                y = 0
            if y > len(Image_X) - 1:
                y = len(Image_X) - 1
            if x < 0:
```

```
                    x = 0
                if x > len(Image_X[0]) - 1:
                    x = len(Image_X[0]) - 1
                image_to_fill[i, j] =
                    bilinear_interpolation(y, x, Image_X)

    return image_to_fill


def correspondences(img1,img2,edges1,edges2,dist,metric,N
    =10):
    point_pair = []
    img1 = img1 / 255
    img2 = img2 / 255
    pairs = []
    for row in range(edges1.shape[0]):
        x_index = np.flatnonzero(edges1[row])
        if np.size(x_index) == 0:
            continue
        for x in x_index:
            if x+dist>len(edges2[0])-1:
                end = len(edges2)-1
            else:
                end = x+dist
            window = edges2[row,x:end+1]
            candidate = np.flatnonzero(window)
            if np.size(candidate) == 0:
                continue
            corr_x = x + candidate[0]
            edges2[row,corr_x] = 0
            pairs.append([[row,x],[row,corr_x]])
    distance_list = []
    for pair in pairs:
        pt1 = pair[0]
        pt2 = pair[1]
        distance_list.append((compute_distance(img1, img2
            , pt1, pt2, metric, N),pair))
    sort_dist = sorted(distance_list, key=lambda distance
        :distance[0])
    return [coord for _,coord in sort_dist]


def compute_distance(img1,img2,point1,point2,metric,N=10)
    :
    img1 = cv2.copyMakeBorder(img1,N,N,N,N,cv2.
        BORDER_REPLICATE)
```

```
    img2 = cv2.copyMakeBorder(img2,N,N,N,N,cv2.
        BORDER_REPLICATE)
    window1 = img1[point1[0]:point1[0]+int(2*N+1),point1
        [1]:point1[1]+int(2*N+1)]
    window2 = img2[point2[0]:point2[0]+int(2*N+1),point2
        [1]:point2[1]+int(2*N+1)]
    distance = np.inf
    if metric == 'SSD':
        distance = np.sum((window1 - window2) ** 2)
    elif metric == 'NCC':
        mean1 = window1.mean()
        mean2 = window2.mean()
        corr = np.sum((window1-mean1)*(window2-mean2))
        norm1 = np.sum((window1-mean1) ** 2)
        norm2 = np.sum((window2-mean2) ** 2)
        distance = 1 - corr/np.sqrt(norm1*norm2)
    return distance


def show_correspondences(img1,img2,pairs):
    offset = len(img1[0])
    img = cv2.hconcat([img1,img2])
    for pair in pairs:
        color = list(np.random.random(size=3) * 256)
        cv2.line(img,(pair[0][1],pair[0][0]),(pair[1][1]+
            offset,pair[1][0]),color,1)
    return img

def projective_reconstruction(img1pts,img2pts, P1,P2):
    # input pts in x,y
    worldpts = []
    for i in range(len(img1pts)):
        A = np.zeros((4,4))
        A[0] = img1pts[i][0] * P1[2] - P1[0]
        A[1] = img1pts[i][1] * P1[2] - P1[1]
        A[2] = img2pts[i][0] * P2[2] - P2[0]
        A[3] = img2pts[i][1] * P2[2] - P2[1]
        u,_,v = np.linalg.svd(np.dot(A.T,A))
        world = v[-1].flatten()
        worldpts.append(world/world[3])
    return worldpts


def apply_homography_to_point(pts, H):
    points = np.array(pts)
```

```
    points = np.hstack((points,np.ones(len(points)).
        reshape(-1,1)))
    ans = []
    for p in points:
        new = np.dot(H,p)
        new = new/new[-1]
        ans.append([new[0],new[1]])
    return ans



def main_task_2():
    img_name1 = 'box1'
    img1 = cv2.imread(img_name1 + '.jpg', cv2.
        COLOR_BGR2RGB) # left
    img1_pts =
        [[347,110],[515,764],[1490,648],[1266,42],[708,867],[431,992],[1325,875],

    img_name2 = 'box2'
    img2 = cv2.imread(img_name2 + '.jpg', cv2.
        COLOR_BGR2RGB) # right
    img2_pts =
        [[296,116],[326,789],[1412,764],[1293,117],[578,914],[318,1020],[1299,992

    # # draw points
    # for pt in img1_pts:
    #     cv2.drawMarker(img1, (pt[0],pt[1]), (0, 0, 255)
        , markerType=1, markerSize=20, thickness=3)
    # cv2.imwrite('box1_selected_points.jpg',img1)
    # for pt in img2_pts:
    #     cv2.drawMarker(img2, (pt[0], pt[1]), (0, 0,
        255), markerType=1, markerSize=20, thickness=3)
    # cv2.imwrite('box2_selected_points.jpg', img2)
    img1_norm_pts, T1 = normalize_points(img1_pts)
    img2_norm_pts, T2 = normalize_points(img2_pts)
    F_norm = calculate_F(img1_norm_pts, img2_norm_pts)
    F = np.dot(T2.T, np.dot(F_norm, T1))
    # F = F / F[2, 2]
    e1,e2 = get_epipoles(F)
    e2_cross = np.array([[0, -e2[2], e2[1]], [e2[2], 0, -
        e2[0]], [-e2[1], e2[0], 0]])
    P1 = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])
    P2 = np.hstack((np.dot(e2_cross, F), np.array([e2]).T
        ))
    sol = least_squares(cost_function, P2.flatten(), args
        =[img1_pts, img2_pts], method='lm')
```

```
refined_P2 = sol.x
refined_P2 = refined_P2.reshape((3,4))
# refined_P2 = refined_P2/refined_P2[-1,-1]
e2_refined = refined_P2[:,3]
e2_refined_cross = np.array([[0, -e2_refined[2],
    e2_refined[1]], [e2_refined[2], 0, -e2_refined
    [0]], [-e2_refined[1], e2_refined[0], 0]])
F_refined = np.dot(e2_refined_cross,np.dot(refined_P2
    ,np.dot(P1.T, np.linalg.inv(np.dot(P1, P1.T)))))
U, s, VT = np.linalg.svd(F_refined)
s[2] = 0
F_refined = np.dot(U, np.dot(np.diag(s), VT))
F_refined = F_refined/F_refined[-1,-1]
e1_refined, e2_refined = get_epipoles(F_refined)
H1,H2 = calc_homography(img1,img2,e1_refined,
    e2_refined,P1,refined_P2,img1_pts,img2_pts)
print(H1,H2)

# left_img = point_mapping_complementary(img1,img1.
    copy(),H1)
# right_img = point_mapping_complementary(img2, img2.
    copy(), H2)
# right_img = cv2.resize(right_img, (left_img.shape
    [1],left_img.shape[0]), interpolation = cv2.
    INTER_AREA)
w,h = 3000, 2500
# negative coordinate: 1847 -118 724 -897   1781 -75
    1381 -109 new: 2355 -112 707 -1171 2265 105 1311
    -406
# H1[0, 2] += 112
# H1[1, 2] += 720
# H2[0, 2] += -90
# H2[1, 2] += 246
left_img = cv2.warpPerspective(img1, H1, (w,h))
right_img = cv2.warpPerspective(img2, H2, (w,h))


cv2.imwrite("img1_rectified.jpg", left_img)
cv2.imwrite("img2_rectified.jpg", right_img)

# interest point detection

img1 = cv2.cvtColor(left_img.astype(np.uint8), cv2.
    COLOR_BGR2GRAY)
img1 = cv2.GaussianBlur(img1, (5, 5), 3)
edges1 = cv2.Canny(img1, 300, 100, 3)
```

```
cv2.imwrite('img1_edge.jpg',edges1)
img2 = cv2.cvtColor(right_img.astype(np.uint8), cv2.
    COLOR_BGR2GRAY)
img2 = cv2.GaussianBlur(img2, (5, 5), 3)
edges2 = cv2.Canny(img2, 30, 100, 3)
cv2.imwrite('img2_edge.jpg', edges2)
pairs = correspondences(img1,img2,edges1,edges2,10,'
    SSD')
img_corr = show_correspondences(left_img.astype(np.
    uint8),right_img.astype(np.uint8),pairs[:500])
cv2.imwrite('img_corr.jpg', img_corr)

# projective reconstruction
# img1_pts_rect =
    [[231,540],[450,1128],[1710,1095],[1122,357],[657,1257],[411,1338],[1503,

# img2_pts_rect =
    [[204,594],[342,1149],[1404,1020],[1056,312],[555,1251],[372,1359],[1329,

img1_pts_rect = apply_homography_to_point(img1_pts,H1
    )
img2_pts_rect = apply_homography_to_point(img2_pts,H2
    )
world_coord = projective_reconstruction(img1_pts_rect
    ,img2_pts_rect,P1,refined_P2)
interestpts1 = [[pt1[1],pt1[0]] for [pt1,pt2] in
    pairs]
interestpts2 = [[pt2[1], pt2[0]] for [pt1, pt2] in
    pairs]
interestpts1 = apply_homography_to_point(interestpts1
    ,H1)
interestpts2 = apply_homography_to_point(interestpts2
    , H2)

world_interest = projective_reconstruction(
    interestpts1,interestpts2,P1,refined_P2)
world_interest = np.array(world_interest)
world_coord = np.array(world_coord)
# reference: https://matplotlib.org/2.0.2/
    mpl_toolkits/mplot3d/tutorial.html
fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
# ax.scatter(world_interest[:,0],world_interest[:,1],
     world_interest[:,2])
ax.scatter(world_coord[:,0],world_coord[:,1],
    world_coord[:,2])
```

```
    # connect 10 lines
    lines =
        [[0,1],[1,2],[2,3],[0,3],[1,5],[5,6],[2,6],[4,5],[4,7],[6,7]]

    for line in lines:
        ax.plot([world_coord[line[0]][0], world_coord[
            line[1]][0]], [world_coord[line[0]][1],
            world_coord[line[1]][1]], [world_coord[line
            [0]][2], world_coord[line[1]][2]])
    plt.show()


def census_transform(d,img1,img2,M):
    w_half = int(M / 2)
    Map = np.zeros(img1.shape)
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    img1 = cv2.copyMakeBorder(img1, w_half, w_half,
        w_half, w_half, cv2.BORDER_CONSTANT, None, value
        =0)
    img2 = cv2.copyMakeBorder(img2, w_half, w_half,
        w_half, w_half, cv2.BORDER_CONSTANT, None, value
        =0)

    for row in range(d+w_half, len(img1) - d - w_half):
        for col in range(d+w_half, len(img1[0])- d -
            w_half):
            dd = np.zeros(d)
            window_left = img1[row-w_half:row+w_half+1,
                col-w_half:col+w_half+1]
            left = (window_left>img1[row,col])*1
            for dist in range(d):
                #shift by dist
                window_right = img2[row-w_half:row+w_half
                    +1, col-dist-w_half:col-dist+w_half+1]
                right = (window_right > img2[row, col-
                    dist]) * 1
                dd[dist] = np.sum(np.logical_xor(left,
                    right))
            Map[row-w_half,col-w_half] = np.argmin(dd)
    return Map.astype(np.uint8)



def main_task_3():
    gt = cv2.imread('disp2.png',cv2.COLOR_BGR2GRAY)
```

```
    gt = (gt.astype(np.float32) / 4).astype(np.uint8)
    cv2.imwrite('gt.jpg',gt)
    d_max = np.max(gt)
    M = 3
    img1 = cv2.imread('im2.png')
    img2 = cv2.imread('im6.png')
    Map = census_transform(d_max,img1,img2,M)
    cv2.imwrite('M'+str(M)+'.png',Map)
    gt = cv2.imread('gt.jpg', 0)
    Map = cv2.imread('M'+str(M)+'.png', 0)
    error = abs(np.subtract(Map,gt))
    Mask = np.array(gt,dtype=bool)
    Mask2 = ((error<=2) & Mask) * 255
    print(Mask2)
    print(np.count_nonzero(Mask2) / np.count_nonzero(Mask
        ))
    cv2.imwrite('Error_mask_M'+str(M)+'.png',Mask2)



if __name__ == '__main__':
    # main_task_2()
    # img_name1 = 'box1'
    # img1 = cv2.imread(img_name1 + '.jpg') # left
    # img_name2 = 'box2'
    # img2 = cv2.imread(img_name2 + '.jpg') # right
    # scale_percent = 75  # percent of original size
    # width = int(img1.shape[1] * scale_percent / 100)
    # height = int(img1.shape[0] * scale_percent / 100)
    # dim = (width, height)
    # resized1 = cv2.resize(img1, dim, interpolation=cv2.
        INTER_AREA)
    # resized2 = cv2.resize(img2, dim, interpolation=cv2.
        INTER_AREA)
    # cv2.imwrite('box1.jpg', resized1)
    # cv2.imwrite('box2.jpg', resized2)
    main_task_3()
```