CS6550 Computer Vision Homework 2

112062673 吳文雯

Q1.Fundamental Matrix Estimation from Point Correspondences

1.Implementation

step 1

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import random

#read input files and record the points
def read_points(filename):
    points = []
    with open(filename, 'r') as file:
        num = int(file.readline().strip())
        for i in range(num):
            x, y = map(float, file.readline().strip().split())
            points.append([x, y, 1])
    return np.array(points)

points1 = read_points("./assets/pt_2D_1.txt")  #shape(59,3)
points2 = read_points("./assets/pt_2D_2.txt")
...
```

◆ Read the input file “pt_2D_1.txt” and “pt_2D_2.txt”, and add the third dimension of the points as 1 to make them homogeneous coordinates. Then store the points in points1 and point2 respectively.

step 2

```
...
def compute_f(points1, points2):  #p2->p1的F
    #construct matrix A
    x1, y1, _ = points1.T
    x2, y2, _ = points2.T
    print(x1,y1,x2,y2)
    A = np.column_stack((x1 * x2, x1 * y2, x1, y1 * x2, y1 * y2, y1, x2, y2,
np.ones(len(x1))))

    # Solve for the fundamental matrix using SVD
    _, _, Vt = np.linalg.svd(A)
    F = Vt[-1].reshape(3, 3) #最小 singular value 所對向量

    # Enforce the rank-2 constraint by making the last singular value zero
    U, S, Vt = np.linalg.svd(F)
    S[-1] = 0  #讓 rank=2
    F = U @ np.diag(S) @ Vt

    return F

F = compute_f(points1, points2)
print("Fundamental Matrix:\n", F)
...
```

◆ Define a function to compute Fundamental matrix F:

◆ Construct matrix A by the x and y coordinates of the points, the formula is on Unit4 p.9.

◆ Solve the equation AF=0 by performing SVD on A, pick the eigen vector of the smallest eigen value of A as the result F.

◆ Since we need F to have rank 2, I perform SVD on F again. Then let the third singular value to be 0, and reconstruct F by U @ np.diag(S) @ Vt.

The Fundamental matrix is:

[[5.73019754e-07 1.84317952e-06 -3.95680928e-04]

 [3.49579399e-06 1.02102047e-06 5.43607666e-03]

 [-2.91687458e-03 -8.06156716e-03 9.99948396e-01]]

step 3

```
def nor_f(points1, points2):
    n = points1.shape[0]
    points1_xy = points1[:, 0:2]
    points2_xy = points2[:, 0:2]

    #取得座標平均
    points1_mean = np.mean(points1_xy, axis=0)
    points2_mean = np.mean(points2_xy, axis=0)

    #計算距離
    points1_dist = points1_xy - points1_mean
    points2_dist = points2_xy - points2_mean

    # 計算 scale
    scale1 = np.sqrt(2 / (np.sum(points1_dist**2)/n))
    scale2 = np.sqrt(2 / (np.sum(points2_dist**2)/n))

    T1 = np.array([
        [scale1, 0, -points1_mean[0] * scale1],
        [0, scale1, -points1_mean[1] * scale1],
        [0, 0, 1]
    ])

    T2 = np.array([
        [scale2, 0, -points2_mean[0] * scale2],
        [0, scale2, -points2_mean[1] * scale2],
        [0, 0, 1]
```

```
    ])

    #轉換到新座標
    q1 = (T1 @ points1.T).T   #59*3
    q2 = (T2 @ points2.T).T

    # 計算 fundamental matrix
    F = compute_f(q1, q2)

    #denormalize
    Fn = T1.T @ F @ T2

    return Fn

Fn = nor_f(points1, points2)
print("Normalized Fundamental Matrix:\n", Fn)
```

◆ Define a function to compute Normalized Fundamental matrix Fn :

(the method is on Unit4 p.12.)

◆ Compute the mean of the x and y coordinate for points1 and points2, and the distance from each point to the mean.

◆ Compute the scale for points1 and points2 so that the mean square distance between the centroid and the points is 2 pixel.

◆ Construct translation matrix T for points1 and points2, and convert the points to the new coordinate q1 and q2.

◆ Compute Fundamental matrix F by q1 and q2, and the Normalized Fundamental matrix Fn can be obtained by T1.T @ F @ T2.

The Fundamental matrix is:

```
[[ 1.09764411e-07  5.84388058e-07  2.64451656e-04]
 [ 9.87746602e-07 -3.93098177e-08  4.52537953e-03]
 [-4.43597264e-04 -5.14958068e-03  6.02905048e-02]]
```

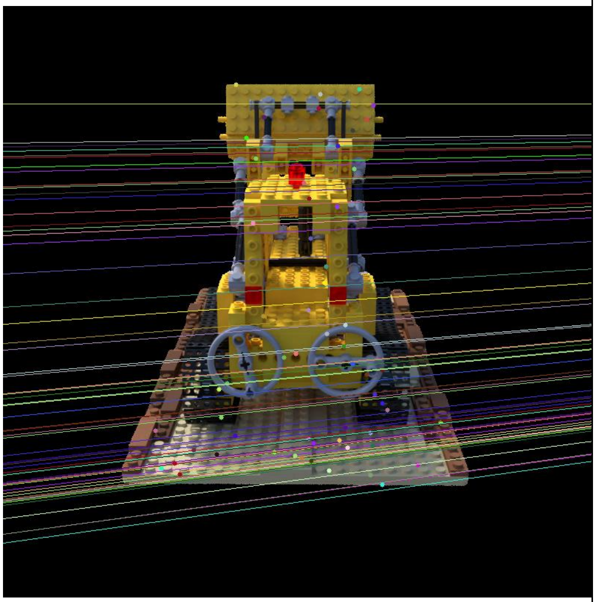
Step 4

```
...
# calculate the epipolar lines
lines1 = F @ points2.T
lines2 = F.T @ points1.T
numl =lines1.shape[1]
#print(numl)
image1 = cv2.imread('./assets/image1.jpg')
image2 = cv2.imread('./assets/image2.jpg')

colors = [(random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))
for i in range(numl)]

for i in range(numl):
    A = lines1[0, i]
    B = lines1[1, i]
    C = lines1[2, i]
    W = image1.shape[1]
    y1 = -C/B
    y2 = -(A * W + C) / B
    cv2.line(image1, (0, int(y1)), (W, int(y2)), colors[i], 1)
    cv2.circle(image1, (int(points1[i, 0]), int(points1[i, 1])), 3, colors[i], -1)
file_path="./output"
cv2.imwrite(os.path.join(file_path,"wo_normalized_img1.png"),image1)
cv2.imshow('wo_normalized version on image 1', image1)
cv2.waitKey(0)
...
```

- Calculate the epipolar lines for points1 and points2, the formula is on Unit4 p.8.
- Generate a list of random colors so that each line could have different color, but the color of the corresponding points will be the same.
- Draw the epipolar lines and the points on the image and show the image.(The code for drawing lines on image2 is similar, so I do not show it here.)
- The code for drawing lines with normalized fundamental matrix is similar, just

replace F by Fn.

Image 1 and image 2 without normalized fundamental matrix:



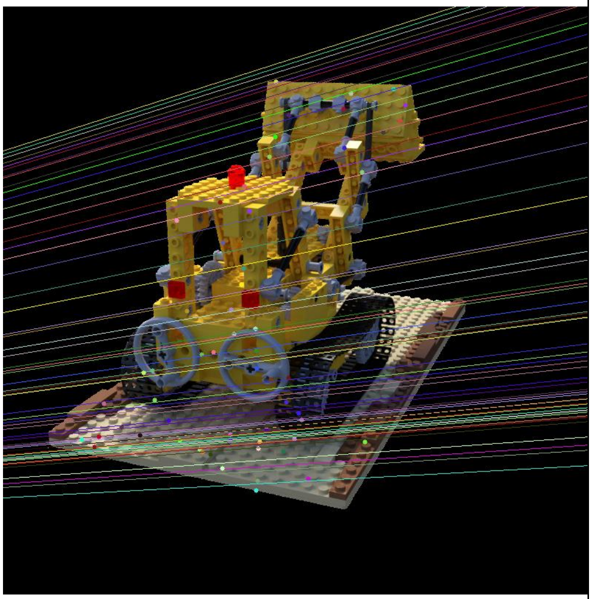
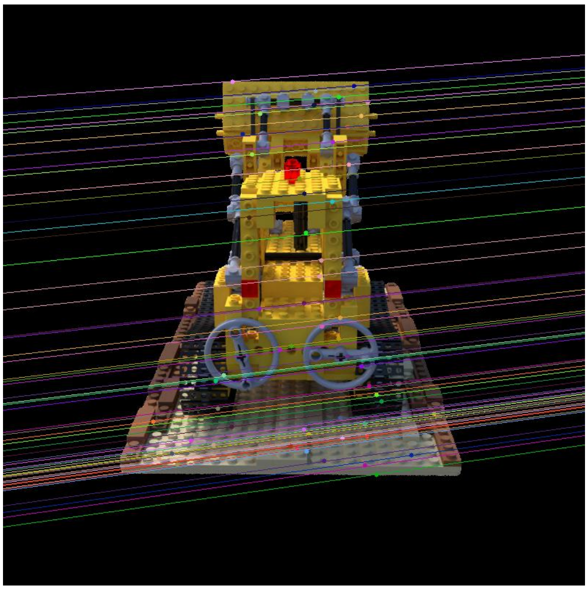
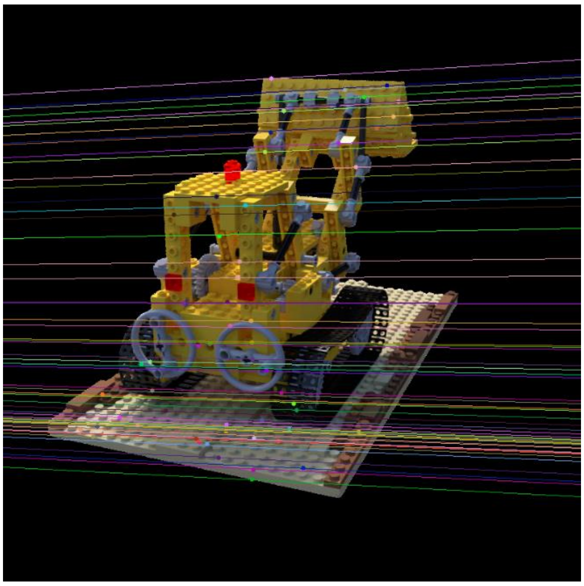


Image 1 and image 2 with normalized fundamental matrix:





Step 5

```
...
def get_distance(points1, points2, F):
    lines = F @ points2.T
    n = points1.shape[0]

    A = lines[0]
    B = lines[1]
    C = lines[2]
    x = points1[:, 0]
    y = points1[:, 1]

    # Compute distances
    distances = np.abs(A * x + B * y + C) / np.sqrt(A**2 + B**2)

    dis_sum = np.sum(distances)

    return dis_sum/n

F_image1 = get_distance(points1, points2, F)
print(F_image1,"\n" )
F_image2 = get_distance(points2, points1, F.T)
print(F_image2,"\n" )
Fn_image1 = get_distance(points1, points2, Fn)
print(Fn_image1,"\n" )
Fn_image2 = get_distance(points2, points1, Fn.T)
print(Fn_image2,"\n" )...
```

- Define a function to compute the average distance between the feature

points and their corresponding epipolar lines:

- The epipolar lines for points2 can be obtained by F @ points2.T. Then compute the

distance according to the formula $d=|Ax+By+C/(A^2+B^2)^{1/2}|$.

- The average distance is the sum of the distance divided by the number of points.

- The way to calculate the distance for points1 and for different Fundamental matrix are similar. Just to change the parameters passed into the function. Since F is the matrix for points2 to points1, if we want the calculate the lines for points1, we have to use F.T.
- **Results:**

Average distance on image 1 for F:	25.26065725927166
Average distance on image 1 for Fn:	0.9153900496400987
Average distance on image 2 for F:	25.64771846803731
Average distance on image 2 for Fn:	0.9005038459924034

The accuracy of the Normalized Fundamental Matrix is better than the one without normalizing.

Q2.Homography transform

1.Implementation

step 1

```
def Find_Homography(src,tar):
    A = []
    for i in range(len(src)):
        x, y = src[i]
        u, v = tar[i]
        A.append([x, y, 1, 0, 0, 0, -x*u, -y*u, -u])
        A.append([0, 0, 0, x, y, 1, -x*v, -y*v, -v])

    A = np.array(A)
    _, _, VT = np.linalg.svd(A)
```

```
    H = VT[-1].reshape(3, 3)

    return H / H[2,2]
```

◆ Implement a function to compute homography matrix H:

(this method is based on Unit3 p.29&30)

◆ Construct matrix A from the source points and the target points, and solve the

equation AH=0 by using SVD. The solution is the eigen vector of the smallest eigen value of A, so I take VT[-1] as H and reshape it into 3*3 matrix. Also, I let the final element of H become 1 for homogeneous coordinates.

step 2

```
    ...
    img_src = cv2.imread("./assets/post.png")
    src_H,src_W,_=img_src.shape
    corner_src=[(0,0),(src_W-1,0),(src_W-1,src_H-1),(0,src_H-1)]
    print("src=",corner_src)
    #print(src_H,src_W)
    #file_path="./output"
    img_tar = cv2.imread("./assets/display.jpg")
    tar_H,tar_W,_=img_tar.shape
    re_h = int(img_tar.shape[0] * 0.5)
    re_w = int(img_tar.shape[1] * 0.5)
    img_tar = cv2.resize(img_tar, (re_w, re_h))
    cv2.namedWindow("Interative window")
    cv2.setMouseCallback("Interative window", mouse_callback)
    cv2.setMouseCallback("Interative window", mouse_callback)
    ...
    H = Find_Homography(corner_src,corner_tar)
    print(H)
```

◆ Read the image “post.png” and stored it as img_src. The interest points are the 4

corners of the image, so I set them in corner_src.

◆ Read the image “display.png” and stored it as img_tar. Then use the function provided by TA to get the 4 corners of the screen. The return list of points is corner_tar.

◆ Use the previous function "Find_Homography()" to compute the homography. The selected line pairs are the green lines in the image shown later, and the homography is :

[[1.09724917e+00 7.11284984e-03 2.72000000e+02]
[3.34938133e-01 8.18009402e-01 6.00000000e+01]
[1.31113554e-03 4.74050096e-05 1.00000000e+00]]

step 3

```
def warp(img_src, H, tar_h,tar_w):
    dst = np.zeros((tar_h, tar_w, img_src.shape[2]), dtype=img_src.dtype)

    H_inv = np.linalg.inv(H)

    for y in range(tar_h):
        for x in range(tar_w):
            # calculate the src coordinates
            src_co = H_inv @ np.array([x, y, 1])
            src_x, src_y, src_w = src_co / src_co[2]

            # check boundary
            if 0 <= src_x < img_src.shape[1] - 1 and 0 <= src_y < img_src.shape[0]
- 1:
                x0, y0 = int(src_x), int(src_y)
                x1, y1 = x0 + 1, y0 + 1
                dx, dy = src_x - x0, src_y - y0

                # bilinear interpolation
                pixel = (
                (1 - dx) * (1 - dy) * img_src[y0, x0] +
                dx * (1 - dy) * img_src[y0, x1] +
                (1 - dx) * dy * img_src[y1, x0] +
                dx * dy * img_src[y1, x1]
                )
```

```
                dst[y, x] = pixel

        return dst

    src_h,src_w,_=img_src.shape
    tar_h,tar_w,_=img_tar.shape

    result = warp(img_src, H, tar_h , tar_w)
    cv2.fillConvexPoly(img_tar, np.array([corner_tar], dtype=np.int32) ,(0,0,0))
    image = img_tar+result
```

◆ Implement a function to perform backward warping:

◆ First construct a new image "dst" to store the collected value from src, and its size is the same as image_src.

◆ Compute the inverse homography by np.linalg.inv(H).

◆ Convert each point in img_tar to the coordinate in img_src by multiplying with inverse homography. And since they were formed as homogeneous coordinates before, their coordinates then divided by their third element to become cartesian coordinates on the image_src.

◆ Perform bi-linear interpolation only on those points in the range of img_tar, and the value of each pixel is stored in "dst" and later return to the "result".

◆ Use cv2.fillConvexPoly() to make the area of the screen become black so that the image obtained from backward warping can be shown clearly.

◆ Then use image_tar + result to combine the original image and one projected on the screen.

step 4

```
#draw 4 lines
p1, p2, p3, p4 = corner_tar
```

```
l1 = np.array([p1, p2], dtype=np.int32).reshape((-1, 1, 2))
l2 = np.array([p2, p3], dtype=np.int32).reshape((-1, 1, 2))
l3 = np.array([p3, p4], dtype=np.int32).reshape((-1, 1, 2))
l4 = np.array([p4, p1], dtype=np.int32).reshape((-1, 1, 2))

image = cv2.polylines(image, [l1, l2, l3, l4], isClosed=True, color=(0, 255, 0),
thickness=2)

#compute vanishing point
line1 = np.cross([p1[0], p1[1],1], [p2[0], p2[1],1])
line2 = np.cross([p4[0], p4[1],1], [p3[0], p3[1],1])
vp = np.cross(line1,line2)
#print(vp)
vp = (vp/vp[2])[:2]
print(vp)

vp = vp.astype(int)
cv2.circle(image, vp, 5, (0, 255, 0), -1)
cv2.imwrite(os.path.join(file_path,"homography.png"),image)
cv2.imshow("image",image)
cv2.waitKey(0)
```

◆ Construct 4 lines from the 4 points we previously click and show them on the image by cv2.polylines. **The lines are shown in green.**

◆ Compute the vanishing points from the two parallel line. **I choose the same lines as the upper and bottom lines among the previous 4 lines, so I do not draw other lines on the image.** The 2 lines can be obtained by performing cross product on the 2 points in the line, which are p1, p2 and p4, p3 respectively. Finally, the vanishing point is result of the cross product of the 2 parallel lines.
The vanishing point is the green dot shown on the image and it: coordinate is:

[842 256]

(it will differ from the points you click. Also, I couldn't view the whole image in the interactive window before, so I resize "display.png" to half its size. Thus, the coordinate may be different, but the location it is shown on the image is similar.)

◆ Draw the vanishing point on the image and show it, the result is as below.