1. Fundamental Matrix Estimation from Point Correspondences

(a)

```
# hw2_1
# A = x1x'1 x1y'1 x1 y1x'1 y1y'1 y1 x'1 y'1 1
#      :                                      :
#      xmx'm xmy'm xm ymx'm ymy'm ym x'm y'm 1

import cv2
import numpy as np
import matplotlib.pyplot as plt
import math


file1 = open("assets\pt_2D_1.txt", 'r')
file2 = open("assets\pt_2D_2.txt", 'r')
img1 = cv2.imread('assets\image1.jpg')
img2 = cv2.imread('assets\image2.jpg')


point_1_list = []
point_2_list = []
row_num_1 = file1.readline()
row_num_2 = file2.readline()
```

```
# known point, find the fundamental matrix
def find_fundamental_matrix(img1_point_list, img2_point_list):
    A = []
    for i in np.arange(int(row_num_1)):
        x_1, y_1 = img1_point_list[i][0], img1_point_list[i][1]
        x_2, y_2 = img2_point_list[i][0], img2_point_list[i][1]
        add_row = np.array([x_1*x_2, x_1*y_2, x_1, y_1*x_2, y_1*y_2, y_1, x_2, y_2, 1])
        if i==0:
            A = np.hstack((A, add_row))
        else:
            A = np.vstack((A, add_row))

    # find SVD of AT A
    # A:MxN, full_matrices=1 means U:MxM and V:NxN. Otherwise, U:MxK, V:KxN, K=min(M,N)
    #  compute_uv=1 means compute U, sigma, VT. Otherwise, only compute sigma.
    U, sigma, VT = np.linalg.svd(A, full_matrices=1, compute_uv=1)
    # Entries of F are the elements of column of V corresponding to the least singular value
    # A = UxSxVT, where V = (v1, v2, ...vn), the column is vn, VT is the transport of V
    F = (VT[8]).reshape(3, 3)
    # Enforce rank2 constraint
    F_U, F_S, F_VT = np.linalg.svd(F)
    F = F_U@np.diag([F_S[0], F_S[1], 0])@F_VT
    return F
```

```python
# read points from the files
for i in np.arange(int(row_num_1)):
    line1 = file1.readline() # type str
    line2 = file2.readline()

    point1 = line1.split() #np.char.split(line1) # point1 is a list
    point2 = line2.split() #np.char.split(line2) # point2 is a list

    x_1, y_1 = float(point1[0]), float(point1[1])
    x_2, y_2 = float(point2[0]), float(point2[1])

    if i == 0:
        point_1_list = np.hstack((point_1_list, np.array([x_1, y_1])))
        point_2_list = np.hstack((point_2_list, np.array([x_2, y_2])))
    else:
        point_1_list = np.vstack((point_1_list, np.array([x_1, y_1])))
        point_2_list = np.vstack((point_2_list, np.array([x_2, y_2])))
```

```python
print("non normal F")
print(non_normalized_F)
normalized_F = T1.T@normalized_F@T2
print("normalized F")
print(normalized_F)
```

我先把兩個 txt 檔中的點分別存進 point_1_list 和 point_2_list 中，然後呼叫 find_fundamental_matrix 找 fundamental matrix。

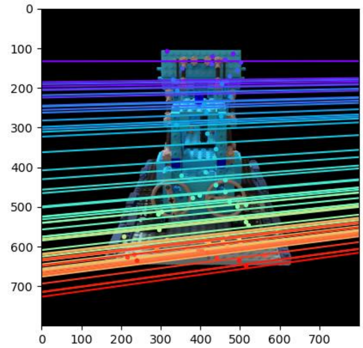
在 find_fundamental_matrix 這個函式中，先找到矩陣 A

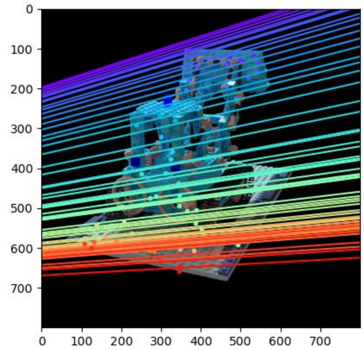
$$A = \begin{bmatrix} x1x'1 & x1y'1 & x1 & y1x'1 & y1y'1 & y1 & x'1 & y'1 & 1 \\ \vdots & & & & & & & & \vdots \\ xmx'm & xmy'm & xm & ymx'm & ymy'm & ym & x'm & y'm & 1 \end{bmatrix}$$

，把 A 用 np.linalg.svd 分成 U, sigma, V^T，然後取出 the column of V corresponding to the least singular value，也就是 VT[8]，並轉成 3x3 矩陣當作 F，之後再做”enforce the rank-2 constraint”，把此時的 F 用 np.linalg.svd 分成 F_U, F_S, F_VT，再把 F_U、新的 F_S、F_VT 相乘起來，而此時的新的 F_S 是原本 F_S 把第三列變成[0, 0, 0]，相乘起來的矩陣就是 fundamental matrix。

The returned fundamental matrix:

```
non normal F
[[ 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]]
```





(b)

```python
def Sum_of_Euclidean_Distance(point, center_point):
    dist = 0
    for i in np.arange(int(row_num_1)):
        dist = dist + (point[i]-center_point)@(point[i]-center_point).T
    return dist
```

```python
# normalized
x, y, channel = img1.shape
image1_center_x, image1_center_y = x/2, y/2

# mean squared distance between center and the data points is 2 pixels
dist1 = Sum_of_Euclidean_Distance(point_1_list, [image1_center_x, image1_center_y])
dist2 = Sum_of_Euclidean_Distance(point_2_list, [image1_center_x, image1_center_y])

s1 = np.sqrt(2/dist1)
s2 = np.sqrt(2/dist2)
T1 = np.diag([s1, s1, 1])@[[1, 0, -image1_center_x],
                           [0, 1, -image1_center_y],
                           [0, 0, 1]]
T2 = np.diag([s2, s2, 1])@[[1, 0, -image1_center_x],
                           [0, 1, -image1_center_y],
                           [0, 0, 1]]

normalized_point_1_list = []
normalized_point_2_list = []

for i in np.arange(int(row_num_1)):
    add_normalized_point_1 = T1@[point_1_list[i][0], point_1_list[i][1], 1]
    add_normalized_point_2 = T2@[point_2_list[i][0], point_2_list[i][1], 1]
    if i == 0:
        normalized_point_1_list = np.hstack((normalized_point_1_list, add_normalized_point_1)) #test array
        normalized_point_2_list = np.hstack((normalized_point_2_list, add_normalized_point_2))
    else:
        normalized_point_1_list = np.vstack((normalized_point_1_list, add_normalized_point_1)) #test array
        normalized_point_2_list = np.vstack((normalized_point_2_list, add_normalized_point_2))
```

```python
non_normalized_F = find_fundamental_matrix(point_1_list, point_2_list)
normalized_F = find_fundamental_matrix(normalized_point_1_list, normalized_point_2_list)
```

要做 normalized eight-point algorithm，就是把原本在 image1 和 image2 上的點，分別透過 T1 和 T2 的轉換，讓新的點的 mean squared distance between the origin and the data points 是 2 pixels，並分別儲存在 normalized_point_1_list 和 normalized_point_2_list 中，再呼叫 find_fundamental_matrix，並傳入 normalized_point_1_list 和 normalized_point_2_list，且傳回一個 matrix F，最後

的 fundamental matrix 會是 T1.T@F@T2。

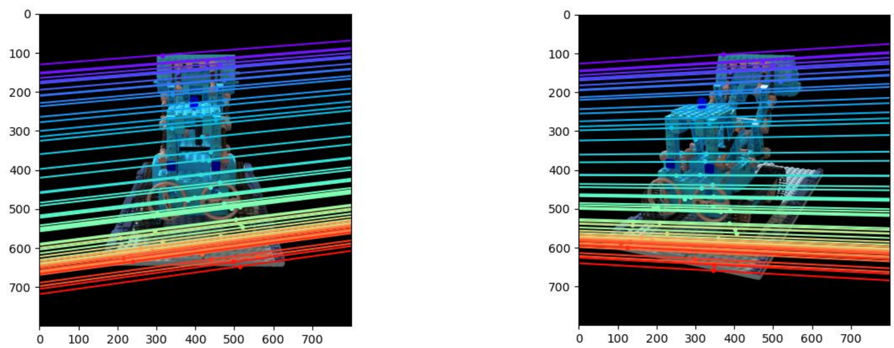
新的點的尋找方法是：利用 Sum_of_Euclidean_Distance 找到 s

$$s = \sqrt{\frac{2}{\sum_{i=1}^{n} \|x_i - c\|^2}}$$

，並計算每一點以圖片正中心為原點的座標，再乘上 s，就可以得到新的點。

The returned fundamental matrix:

```
normalized F
[[ 1.38911301e-08  6.56063741e-08  3.36993885e-05]
 [ 1.26590647e-07 -2.79939783e-09  5.47867656e-04]
 [-5.62447685e-05 -6.25262920e-04  7.69761279e-03]]
```



(c)

```
# epipolar line
def plot_epipolar_line(img, w, F, point_list, plot_point_list):
    colormap = plt.get_cmap("rainbow")
    for i in np.arange(int(row_num_1)):
        color = colormap(i/len(point_list))
        # point1(0, -c/b),point2(w, -(a*w+c)/b) are on line:ax+by+c=0
        a, b, c = F@(np.array([point_list[i][0], point_list[i][1], 1]).T)
        point1 = (0, -c/b)
        point2 = (w, -(a*w+c)/b)
        plt.plot(*zip(point1, point2), color=color)
        plt.plot(plot_point_list[i][0], plot_point_list[i][1], ".", color=color)
    plt.imshow(img)
    plt.show()
    return
```

```python
# shortest distance between a point:(p1, p2) and a line:ax+by+c=0
# point are point_on_img, line created by F@point_create_line
def total_dist(F, point_create_line, num_of_point, point_on_img):
    total_dist_in_accu = 0
    for i in np.arange(num_of_point):
        a, b, c = F@(np.array([point_create_line[i][0], point_create_line[i][1], 1]).T)
        p1, p2 = point_on_img[i][0], point_on_img[i][1]
        d = abs((a*p1 + b*p2 + c)) / (np.sqrt(a*a + b*b))
        total_dist_in_accu = total_dist_in_accu + d
    return total_dist_in_accu
```

```python
# plot epipolar line
# on img2, epipolar line is F.T@P
plot_epipolar_line(img2, y, non_normalized_F.T, point_1_list, point_2_list)
plot_epipolar_line(img1, y, non_normalized_F, point_2_list, point_1_list)
plot_epipolar_line(img2, y, normalized_F.T, point_1_list, point_2_list)
plot_epipolar_line(img1, y, normalized_F, point_2_list, point_1_list)

# the accuracy
avg_dist_between_non_normalized_f_and_point = (total_dist(non_normalized_F.T, point_1_list, int(row_num_1), point_2_list)+total_dist(non_normalized_F, point_2_list, int(row_num_1), point_1_list))/(2*int(row_num_1))
avg_dist_between_normalized_f_and_point = (total_dist(normalized_F.T, point_1_list, int(row_num_1), point_2_list)+total_dist(normalized_F, point_2_list, int(row_num_1), point_1_list))/(2*int(row_num_1))
print("accuracy of non_normalized F", avg_dist_between_non_normalized_f_and_point)
print("accuracy of normal F", avg_dist_between_normalized_f_and_point)

file1.close()
file2.close()
```

Plot the epipolar line：

F.T@p = [a, b, c].T，image1 的點 p 對應到的 epipolar line 是 ax+by+c=0。

F@p = [a, b, c].T，image2 的點 p 對應到的 epipolar line 是 ax+by+c=0。

計算完 epipolar line 之後，再把它們顯示在圖上。

計算 the accuracy of the fundamental matrix：

先用 `total_dist(F, point_create_line, num_of_point, point_on_img)`

計算 image1 中的點和”image2 的點對應到的 epipolar line”的距離 加上 image2 中的點和”image1 的點對應到的 epipolar line”的距離 的總和，再把總合除以全部的點的數量，就可以得到 accuracy。

The accuracy of the fundamental matrix in (a), (b) are

```
accuracy of non_normalized F 25.45418786368739
accuracy of normal F 0.9080341346800845
```

2. Homography transform

(a)

```python
def Find_Homography(world,camera):
    '''
    given corresponding point and return the homagraphic matrix
    '''
    # world(left) x_2 y_2 <- H -- camera(right) x_1 y_1
    # x_1 y_1 1 0   0   0 -x_2*x_1 -x_2*y_1 -x_2
    # 0   0   0 x_1 y_1 1 -y_2*x_1 -y_2*y_1 -y_2
    A = []
    for i in np.arange(4):
        x_1, y_1 = camera[i][0], camera[i][1]
        x_2, y_2 = world[i][0], world[i][1]
        add_row = [[x_1, y_1, 1, 0, 0, 0, -x_2*x_1, -x_2*y_1, -x_2],
                   [0, 0, 0, x_1, y_1, 1, -y_2*x_1, -y_2*y_1, -y_2]]
        if i==0:
            A = add_row
        else:
            A = np.vstack((A, add_row))
    A_U, A_S, A_VT = np.linalg.svd(A)
    H = (A_VT[8]).reshape(3, 3)
    return H
```

`Find_Homography(world,camera)`

這個函數中要找的 homography matrix H 滿足[x_2, y_2, 1] = H@[x_1, y_1, 1]且(x_1, y_1)和(x_2, y_2)分別來自 camera 和 world。

$$\underbrace{\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ & & & & \vdots & & & & \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n & -y'_n \end{bmatrix}}_{\substack{\mathbf{A} \\ 2n \times 9}} \underbrace{\begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix}}_{\substack{\mathbf{h} \\ 9}} = \underbrace{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}}_{\substack{\mathbf{0} \\ 2n}}$$

我先找出這個 matrix A，然後再用 np.linalg.svd 把 A 分成 A_U, A_S, A_VT，之後取出 A_VT 的最後一個 row 當作 h，再把 h 轉成 H，就得到 homography matrix H。

(b)

```python
if(len(corner_list)==4):
    # implement the inverse homography mapping and bi-linear interpolation
    world = corner_list
    print("corner_list")
    print(corner_list)
    camera = [[0, 0],
              [src_W, 0],
              [src_W, src_H],
              [0, src_H]]
    H_map_CVimage_to_screen = Find_Homography(world, camera) # world <-H- camera
    inverse_H = Find_Homography(camera, world) # world -inverse_H -> camera
    print("H(map CV image to screen)")
    print(H_map_CVimage_to_screen)
    print("inverse H")
    print(inverse_H)

    array_corner = np.array(corner_list)
    min_x = np.min(array_corner[:, 0])
    max_x = np.max(array_corner[:, 0])
    min_y = np.min(array_corner[:, 1])
    max_y = np.max(array_corner[:, 1])

    for i in np.arange(min_x, max_x+1): # [min_x, max_x]
        for j in np.arange(min_y, max_y+1): # [min_y, max_y]
            point_in_camera = inverse_H@(np.array([i, j, 1]).T)
            point_in_camera[0] = point_in_camera[0]/point_in_camera[2]
            point_in_camera[1] = point_in_camera[1]/point_in_camera[2]

            if (0<=point_in_camera[0] and point_in_camera[0]<src_W-1 and
                0<=point_in_camera[1] and point_in_camera[1]<src_H-1):
                # point is in img of camera
                # find color in (point_in_camera[0], point_in_camera[1]) by bi-linear interpolation
                # tmp1:(int(point_in_camera[0]), point_in_camera[1]) tmp2:(int(point_in_camera[0])+1, point_in_camera[1])
                # imgsrc00(img[int(point[1])][int(point[0])])     tmp1   imgsrc01(img[int(point[1])+1][int(point[0])])
                #                                                  p
                # imgsrc10(img[int(point[1])][int(point[0])+1])   tmp2   imgsrc11(img[int(point[1])+1][int(point[0])+1])
                # img[j][i] = color
                imgsrc00 = img_src[int(point_in_camera[1])][int(point_in_camera[0])]
                imgsrc01 = img_src[int(point_in_camera[1])+1][int(point_in_camera[0])]
                imgsrc10 = img_src[int(point_in_camera[1])][int(point_in_camera[0])+1]
                imgsrc11 = img_src[int(point_in_camera[1])+1][int(point_in_camera[0])+1]
                tmp1 = imgsrc00*(int(point_in_camera[1])+1 - point_in_camera[1]) + imgsrc01*(point_in_camera[1] - int(point_in_camera[1]))
                tmp2 = imgsrc10*(int(point_in_camera[1])+1 - point_in_camera[1]) + imgsrc11*(point_in_camera[1] - int(point_in_camera[1]))
                color_value = tmp1*(int(point_in_camera[0])+1 - point_in_camera[0]) + tmp2*(point_in_camera[0] - int(point_in_camera[0]))
                fig[j][i] = color_value
```

我先用 Find_Homography 找到可以”把我所選的四個點”打到”CV image 上的四個角”的 homography matrix H，然後用 min_x、min_y、max_x、max_y 表示一個比我所選的四個點還大的長方形，之後把長方形中的每一個點 p 都用 H 傳送到另一張圖片 CV image 上，假設傳送到 p’。如果超出 CV image 的範圍，則忽視；如果沒有超出範圍，就用 bilinear interpolation 找到 p 的顏色。

bilinear interpolation:

如果遇到在邊界上的點，則忽視。

假設 imgsrc00、imgsrc01、imgsrc10、imgsrc11 分別是 p’的左上、右上、左下、右下的點，tmp1、tmp2 分別在 imgsrc00 和 imgsrc01、imgsrc10 和 imgsrc11 之間，i.e.

```
imgsrc00 ----- tmp1 ----- imgsrc01
                 |
                 P’
                 |
Imgsrc10 ----- tmp2 ----- imgsrc11
```

先用 linear interpolation 找出 tmp1、tmp2，再用 linear interpolation 找出 p 要的顏色。

```
corner_list
[(541, 111), (1123, 317), (1114, 745), (532, 970)]
H(map CV image to screen)
[[ 3.89781232e-03 -2.20010460e-06  9.79579494e-01]
 [ 1.19468482e-03  2.96989019e-03  2.00985811e-01]
 [ 2.29789889e-06  5.33351124e-08  1.81068298e-03]]
inverse H
[[ 1.82835228e-03  1.91561938e-05 -9.91264918e-01]
 [-5.79612070e-04  1.63754478e-03  1.31802659e-01]
 [-2.30324967e-06 -7.25459029e-08  3.94460543e-03]]
```

(c)

```python
            # compute vanishing point(v1, v2)
            # the intersection of line((corner_list[0]), (corner_list[1])) and
            # line((corner_list[2]), (corner_list[3]))
            L1 = np.cross((corner_list[0][0], corner_list[0][1], 1), (corner_list[1][0], corner_list[1][1], 1))
            L2 = np.cross((corner_list[2][0], corner_list[2][1], 1), (corner_list[3][0], corner_list[3][1], 1))
            kv1, kv2, k = np.cross(L1, L2)
            v1 = kv1/k
            v2 = kv2/k
            print("vanishing point's coordinate (", int(v1), int(v2), ")")
            # print vanishing point
            cv2.circle(fig, center=(int(v1), int(v2)), radius=5, color=(0, 255, 0), thickness=-1)
            # print corner
            cv2.circle(fig, corner_list[0], 5, color=(0, 255, 0), thickness=-1)
            cv2.circle(fig, corner_list[1], 5, color=(0, 255, 0), thickness=-1)
            cv2.circle(fig, corner_list[2], 5, color=(0, 255, 0), thickness=-1)
            cv2.circle(fig, corner_list[3], 5, color=(0, 255, 0), thickness=-1)
            # print line
            cv2.line(fig, corner_list[0], corner_list[1], color=(75, 0, 130), thickness=3)
            cv2.line(fig, corner_list[1], corner_list[2], color=(75, 0, 130), thickness=3)
            cv2.line(fig, corner_list[2], corner_list[3], color=(75, 0, 130), thickness=3)
            cv2.line(fig, corner_list[3], corner_list[0], color=(75, 0, 130), thickness=3)
            # put the image fig in homography.png
            cv2.imwrite('output/homography.png', fig)
            # pass
```

Vanishing point (v1, v2)是由 corner_list[0]、corner_list[1]所形成的線 L1 和由 corner_list[2]、corner_list[3]所形成的線 L2 相交的點。

令(a1, b1, c1) = (corner_list[0][0], corner_list[0][1], 1)x(corner_list[1][0], corner_list[1][1], 1)、

(a2, b2, c2) = (corner_list[2][0], corner_list[2][1], 1)x(corner_list[3][0], corner_list[3][1], 1)。

可得 L1 : a1*x+b1*y+c1 = 0、L2 : a2*x+b2*y+c2 = 0。

因為 vanishing point 在 L1、L2 上，所以

(k*v1, k*v2, k*1) = (a1, b1, c1)x(a2, b2, c2) for some k。

之後，就可以得到 vanishing point (v1, v2)。

Vanishing point's coordinate:

```
vanishing point's coordinate ( 1696 519 )
```


AMD
NEXT HORI
G A M I
TECH DAY