Problem 1. Image Alignment with RANSAC:

A.

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

img_book1 = cv2.imread('1-book1.jpg')
img_book2 = cv2.imread('1-book2.jpg')
img_book3 = cv2.imread('1-book3.jpg')
img_image = cv2.imread('1-image.jpg')


def SIFT_object_recognition(img_1, img_2): # img1->image, img2->book1, book2, book3
    sift = cv2.SIFT_create() # construct a SIFT object
    kps_1, des_1 = sift.detectAndCompute(img_1, None) # Given a set of Keypoint,compute descriptors.
    kps_2, des_2 = sift.detectAndCompute(img_2, None)

    kps_1_len = len(kps_1)
    kps_2_len = len(kps_2)


    # Brute-force matching and ratio test
    def Brute_force_matching(): #keypoint_num, descriptor_num, descriptor_num_idx):
        distance_num = ()

        for i in range(kps_1_len): # image
            # init tmp_set
            tmp_set = []

            for j in range(kps_2_len): # book
            for j in range(kps_2_len): # book
                # consider the min distance between des_1[i] and des_2[j]
                dist_compute = np.linalg.norm(des_1[i]-des_2[j]) #descriptor_num[j]) # init
                # tmp_set for collecting all (idx_j, distance_between_i_and_j)
                tmp_set.append((cv2.DMatch(i, j, dist_compute)))
            # for i, append the nearest one B and the second nearest one C from A
            # select in tmp_set
            tmp_set = sorted(tmp_set, key = lambda x:x.distance)
            tmp_set = tuple(tmp_set)
            distance_num = ((tmp_set[0], tmp_set[1]), ) + distance_num

        return distance_num #keypoint_num, descriptor_num, descriptor_num_idx, distance_num


    """keypoint_book, descriptor_book, descriptor_book_idx, """
    distance_book = Brute_force_matching()#kps_2, des_2,)
    print("len", len(distance_book))

    # apply ratio test
    def ratio_test(distance_num):
        matches_num = []
        for m, n in distance_num:
            if m.distance < 0.85*n.distance: # 0.5
                # threshold = 600
                if m.distance <= 600:
                    matches_num.append([m])
        return matches_num

    matches = ratio_test(distance_book)



    print("len", len(matches))


    # sort in the order of the distance
    matches = sorted(matches, key = lambda x:x[0].distance) #x[0].distance)

    matchesall = matches[:510] #[:510]

    img_match = cv2.drawMatchesKnn(img_1, kps_1, img_2, kps_2, matchesall, outImg=None, matchColor = (255, 0, 0))
    plt.imshow(img_match), plt.show()

    return kps_1, kps_2, matchesall, img_match


#SIFT_object_recognition(test1, test2)
image1_kps, book1_kps, book1_image_match, img_book1_1a_ans = SIFT_object_recognition(img_image, img_book1)
image2_kps, book2_kps, book2_image_match, img_book2_1a_ans = SIFT_object_recognition(img_image, img_book2)
image3_kps, book3_kps, book3_image_match, img_book3_1a_ans = SIFT_object_recognition(img_image, img_book3)

cv2.imwrite('output/1a_ans_img_book1.jpg', img_book1_1a_ans)
cv2.imwrite('output/1a_ans_img_book2.jpg', img_book2_1a_ans)
cv2.imwrite('output/1a_ans_img_book3.jpg', img_book3_1a_ans)
```

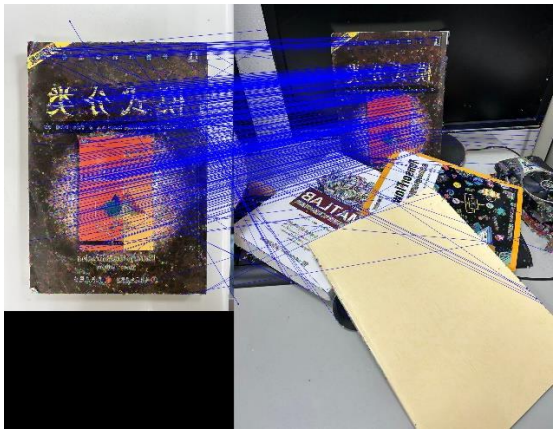
先用 `sift.detectAndCompute` 取出 `keypoint` 和 `descriptor`。

-> 再用 Brute_force_matching()找出:對於第一張圖片中每個 descriptor，第二張圖片中的那兩個 descriptor 是最靠近和第二靠近的。

-> 然後用 ratio_test()找出足夠靠近的 match。

-> 再把這些 match 以 descroptor 之間的距離由小到大做排序。

-> 排序之後，再選出前 510 的 match 出來。



B.

```python
# RANSAC homography , parameter:loop_time, inlier_threshold
# homography
# A = [[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]]
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

def dist(img_point, H_matrix, book_point): # the distance between the estimate point(H@book_point) and real point(image_point)
        esti_point_in_img = H_matrix@(np.array([book_point[0], book_point[1], 1]).T)
        esti_point_in_img[0] = esti_point_in_img[0] / esti_point_in_img[2]
        esti_point_in_img[1] = esti_point_in_img[1] / esti_point_in_img[2]
        esti_point_in_img[2] = 1
        dist = np.linalg.norm([img_point[0], img_point[1], 1] - esti_point_in_img.T)
        return dist

len_book1_image_match = len(book1_image_match)
len_book2_image_match = len(book2_image_match)
len_book3_image_match = len(book3_image_match)


# initial
corner_point_in_book1 = [[96, 220],[999, 224], [987, 1333], [119, 1349]]
corner_point_in_book2 = [[65, 122], [1045, 110], [1043, 1344], [74, 1352]]
corner_point_in_book3 = [[124, 182], [996, 177], [983, 1392], [129, 1397]]


def point_H_line(corner_point_in_book, H, img_image, img_book): # point -> H -> point(image)
    img_with_corner = img_image.copy()
    img_book_with_corner = img_book.copy()
    corner_list_in_img = [] #leftup, rightup, rightdown, leftdown
    corner_list_in_book = []
    for i in np.arange(4):
        corner_point_in_img = H@(np.array([corner_point_in_book[i][0], corner_point_in_book[i][1], 1]).T)
        corner_point_in_img[0] = corner_point_in_img[0] / corner_point_in_img[2]
        corner_point_in_img[1] = corner_point_in_img[1] / corner_point_in_img[2]
        corner_list_in_img.append((int(corner_point_in_img[0]), int(corner_point_in_img[1])))
        corner_list_in_book.append((int(corner_point_in_book[i][0]), int(corner_point_in_book[i][1])))
    img_with_corner = cv2.line(img_with_corner, corner_list_in_img[0], corner_list_in_img[1], color=(255, 0, 0), thickness=5)
    img_with_corner = cv2.line(img_with_corner, corner_list_in_img[1], corner_list_in_img[2], color=(255, 0, 0), thickness=5)
    img_with_corner = cv2.line(img_with_corner, corner_list_in_img[2], corner_list_in_img[3], color=(255, 0, 0), thickness=5)
    img_with_corner = cv2.line(img_with_corner, corner_list_in_img[3], corner_list_in_img[0], color=(255, 0, 0), thickness=5)

    img_book_with_corner = cv2.line(img_book_with_corner, corner_list_in_book[0], corner_list_in_book[1], color=(255, 0, 0), thickness=5)
    img_book_with_corner = cv2.line(img_book_with_corner, corner_list_in_book[1], corner_list_in_book[2], color=(255, 0, 0), thickness=5)
    img_book_with_corner = cv2.line(img_book_with_corner, corner_list_in_book[2], corner_list_in_book[3], color=(255, 0, 0), thickness=5)
    img_book_with_corner = cv2.line(img_book_with_corner, corner_list_in_book[3], corner_list_in_book[0], color=(255, 0, 0), thickness=5)
    return img_with_corner, img_book_with_corner
```

```python
def ransac_homography(len_book_image_match, book_image_match, image_kps, book_kps, img_book, inlier_threshold, corner_point_in_book):
    loop_time = 0
    best_H = []
    best_inlier_of_H = []
    best_inlier_len = 0 # the length of best_inlier_of_H
    while loop_time<=2000: # 2000 # find the best homography matrix among loop_time.
        loop_time = loop_time + 1
        # randomly choose 4 points
        ran_num = np.random.randint(low=0, high=len_book_image_match, size=4)
        #print(ran_num)
        image_4_point = [] # the four point we randomly choose in img image
        book_4_point = [] # the four point we randomly choose in img book1
        for i in np.arange(4):
            queryidx = book_image_match[ran_num[i]][0].queryIdx # img1 -> image
            trainidx = book_image_match[ran_num[i]][0].trainIdx # img2 -> book
            image_kps_x, image_kps_y = image_kps[queryidx].pt
            book_kps_x, book_kps_y = book_kps[trainidx].pt
            #if i == 0:
            image_4_point.append([image_kps_x, image_kps_y])
            book_4_point.append([book_kps_x, book_kps_y])

        # find homography matrix H
        H = Find_Homography(image_4_point, book_4_point) # book - H > image

        # count inlier
        #inlier_threshold = 50
        inlier_match = []
        for i in np.arange(len_book_image_match): # check all match
            queryidx = book_image_match[i][0].queryIdx # queryidx is the index of image's kps
            trainidx = book_image_match[i][0].trainIdx # trainidx is the index of book's kps
            image_kps_x, image_kps_y = image_kps[queryidx].pt # the position of image's kps
            book_kps_x, book_kps_y = book_kps[trainidx].pt # the position of book's kps

            # compute the distance of match points
            distance = dist([image_kps_x, image_kps_y], H, [book_kps_x, book_kps_y])
            print("distance", distance)
            if distance < inlier_threshold : # True->is inlier
                inlier_match.append(book_image_match[i])

        # check whether or not to update best_H and best_inlier_of_H
        if(best_inlier_len < len(inlier_match)):
            # inlier become more, need to update
            best_H = H
            best_inlier_of_H = inlier_match
            best_inlier_len = len(best_inlier_of_H)
    print(best_H)

    # draw line
    img_image_with_line, img_book_with_line = point_H_line(corner_point_in_book, best_H, img_image, img_book)

    img_match = cv2.drawMatchesKnn(img_image_with_line, image_kps, img_book_with_line, book_kps, best_inlier_of_H, outImg=None, matchColor = (255, 0, 0))

    plt.imshow(img_match), plt.show()
    return best_inlier_of_H, best_inlier_len, best_H, img_match
```

```python
best_inlier_of_H1, best_inlier_len_book1, best_H1, img_match_book1_ransac = ransac_homography(len_book1_image_match, book1_image_match, image1_kps, book1_kps, img_book1, 50, corner_point_in_book1)
best_inlier_of_H2, best_inlier_len_book2, best_H2, img_match_book2_ransac = ransac_homography(len_book2_image_match, book2_image_match, image2_kps, book2_kps, img_book2, 20, corner_point_in_book2)
best_inlier_of_H3, best_inlier_len_book3, best_H3, img_match_book3_ransac = ransac_homography(len_book3_image_match, book3_image_match, image3_kps, book3_kps, img_book3, 30, corner_point_in_book3)


cv2.imwrite('output/1b_ans_img_book1.jpg', img_match_book1_ransac)
cv2.imwrite('output/1b_ans_img_book2.jpg', img_match_book2_ransac)
cv2.imwrite('output/1b_ans_img_book3.jpg', img_match_book3_ransac)

img_image_b = img_image.copy()

def draw_deviation_vector(img_image_b, best_inlier_of_H, best_inlier_len, best_H, book_kps, image_kps):
    # show the deviation vector
    for i in np.arange(best_inlier_len):
        # find the position of P in book
        trainidx = best_inlier_of_H[i][0].trainIdx # trainidx is the index of book's kps
        book_kps_x, book_kps_y = book_kps[trainidx].pt # the position of book's kps
        # find the position in image, P's corresponding feature point
        queryidx = best_inlier_of_H[i][0].queryIdx # queryidx is the index of image's kps
        image_kps_x, image_kps_y = image_kps[queryidx].pt # the position of image's kps
        # find the position in image, HP (transformed feature point)
        transformed_point_in_img = best_H@(np.array([book_kps_x, book_kps_y, 1]).T)
        transformed_point_in_img[0] = transformed_point_in_img[0] / transformed_point_in_img[2]
        transformed_point_in_img[1] = transformed_point_in_img[1] / transformed_point_in_img[2]
        # draw arrowed line
        img_image_b = cv2.arrowedLine(img_image_b, (int(image_kps_x), int(image_kps_y)), (int(transformed_point_in_img[0]), int(transformed_point_in_img[1])), color=(0, 0, 255), thickness=2)
    return img_image_b

img_image_b = draw_deviation_vector(img_image_b, best_inlier_of_H1, best_inlier_len_book1, best_H1, book1_kps, image1_kps)
img_image_b = draw_deviation_vector(img_image_b, best_inlier_of_H2, best_inlier_len_book2, best_H2, book2_kps, image2_kps)
img_image_b = draw_deviation_vector(img_image_b, best_inlier_of_H3, best_inlier_len_book3, best_H3, book3_kps, image3_kps)
cv2.imwrite('output/1b_ans_image.jpg', img_image_b)
```

RANSAC 的做法：

Step1. 我先從 A.小題挑出來的 match 中隨機取出 4 個 match。

Step2. 用 4 個 match 算 homography matrix H。

Step3. 對於每個點 P，算出 HP，令 P’ = HP。

->如果 P’和”P 在 match 中所對應的點”距離小於 threshold(這裡把這個 threshold 稱為 inlier_threshold)，那麼 inlier 的個數就加一。

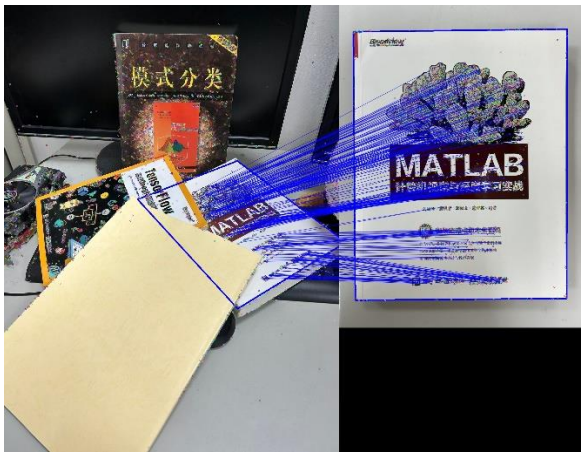
Step1~3 做很多次(這裡做了 2000 次)，找出 inlier 個數最多的 H。

最後，畫出最好的 H 所對應的 inlier match 的圖片 。

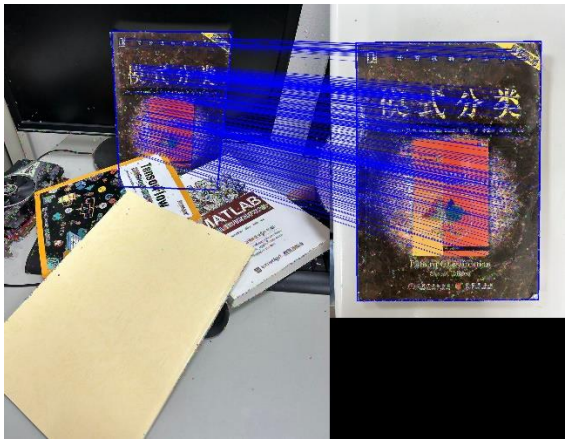
此時，inlier_threshold: book1->50, book2->20, book3->30。

```
the best matching homography transformation
[[-2.07632069e-05 -4.93472970e-04  7.76321557e-01]
 [ 6.92010680e-05  2.23107679e-05  6.30336147e-01]
 [-2.35127395e-07 -1.27130114e-07  9.65716222e-04]]
```

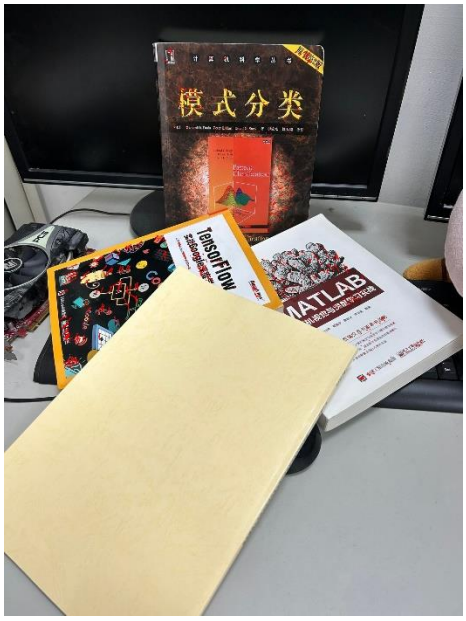

```
the best matching homography transformation
[[ 7.64439186e-04  1.21102678e-04  5.29481892e-01]
 [ 6.44096594e-05  1.04639815e-04  8.48320287e-01]
 [ 2.65817803e-07 -2.34604968e-07  1.00137358e-03]]
```




```
the best matching homography transformation
[[ 1.64874897e-03  3.18697488e-04  9.98708011e-01]
 [ 4.03784602e-05  1.93608488e-03 -5.06940692e-02]
 [ 1.73440761e-08  4.35298531e-07  2.41880910e-03]]
```



Showing the deviation vectors between the transformed feature points and the corresponding feature points.:



Compare the parameter settings in SIFT feature and RANSAC and discuss the result.
當 inlier_threshold 越大時，就會有越多明顯不 match 的線出現、留在圖上。

Problem 2. Image segmentation:
換成另一張圖片 2-masterpiece.jpg 的做法，是直接從 code 中改。
跟 mean shift 有關的，我把圖片 2-image.jpg、2-masterpiece.jpg 做 resize 之後才跑。
Output 資料夾中，檔名後有_byimage、_bymasterpiece，分別代表用 2-image.jpg、2-masterpiece.jpg 跑出來的檔案。

A. kmean

shape

```python
# k-means

# determine points in each cluster
def select_point_to_center(k, point, center): # point, center is array with element = 1x3 array.
    dist_matrix = []
    for i in np.arange(k):
        # 3d dist_matrix
        dist_matrix.append(point-center[i]) # dist_matrix: kx(# of pixel)x3
    #dist_matrix[i][j][k]: [dist_R, dist_G, dist_B], where i:center[i], img_image[j][k]
    dist_matrix = np.square(dist_matrix)
    dist_matrix = np.sum(dist_matrix, axis=2) # [dist_R, dist_G, dist_B] -sum-> R #size:kx(# of pixel)
    # img_with_cluster_center: size:(# of pixel)(Ximg_image), element:index of center
    img_with_cluster_center = np.argmin(dist_matrix, axis=0) # 2d, axis=0, fix j, see which i(center i) is the index of min.
    return img_with_cluster_center

# return new_center = the mean of all points in clusteri.
def find_new_center(k, point, img_with_cluster_center): #(center_with_point):
    new_center = []
    for i in np.arange(k):
        # point: kx(# of pixel)x3
        # point[img_with_cluster_center==i]: (# of pixel)x3
        new_center.append(point[img_with_cluster_center==i].mean(axis=0)) #3d, axis=1 fix j(3: RGB), compute mean
    return new_center


def kmean_step23(k, point, center):
    while 1:
        img_with_cluster_center = select_point_to_center(k, point, center)
        new_center = find_new_center(k, point, img_with_cluster_center)

        find_new_center_again = 0
        for j in np.arange(k):
            if not (np.array_equal(center, new_center)): # center[j] != new_center[j]: # check
                find_new_center_again = 1
            center = new_center
        if find_new_center_again == 0:
            break # have already found the center we want.
    return center, img_with_cluster_center # now, center is the new_center.


def kmean_function(k, image):
    print("k", k)
    # random50
    kmean_time = 0
    min_obj_function = sys.maxsize # set as maxint
    min_obj_func_center = []
    min_obj_func_img_with_cluster_center = []
    while kmean_time<50:
        kmean_time = kmean_time + 1
        # randomly initialize the cluster centers
        img_h, img_w = image.shape[:2] # img_image[img_h][img_w]
        img_h_random = np.random.randint(0, img_h, size=k) # the random index of img_h
        img_w_random = np.random.randint(0, img_w, size=k) # the random index of img_w
        center = []

        for i in np.arange(k):
            center.append(image[img_h_random[i]][img_w_random[i]])

        point = image.reshape(-1, 3) # point: row x 3

        new_center, img_with_cluster_center = kmean_step23(k, point, center)

        # choose the best result based on the objective function for each k.
        obj_function = 0 # for computing the objective function
        for i in np.arange(k):
            # point[img_with_cluster_center==i]-new_center[i]: a matrix, element:(point p(in clusteri)-new_center[i])
            # tmp_for_obj_function contains cluster i=0, ..(k-1), element: the square of (point p(in clusteri)-new_center[i])
            tmp = np.square(point[img_with_cluster_center==i]-new_center[i])
            tmp = np.sum(tmp)
            obj_function = obj_function + tmp
        # square: every element, sum all elements to a real number
        print("min_obj_function, obj_function", min_obj_function, obj_function)
        if min_obj_function > obj_function:
            # if True, update min_obj_function, new_center, img_with_cluster_center
            min_obj_function = obj_function
            min_obj_func_center = new_center
            min_obj_func_img_with_cluster_center = img_with_cluster_center

    # show image
    # index i -> min_obj_func_img_with_cluster_center[i](which cluster j of point i) -> min_obj_func_center[j](the color)
    kmean_img = np.array(min_obj_func_center)[min_obj_func_img_with_cluster_center]
    kmean_img = np.array(kmean_img.reshape(image.shape), dtype=np.uint8)

    return kmean_img
```

```
91
92  #"""
93  kmean_5_image = kmean_function(5, img_image)
94  cv2.imwrite('output/2a_k_5.jpg', kmean_5_image)
95  cv2.imshow('2a_kmean_5_image', kmean_5_image)
96
97  kmean_7_image = kmean_function(7, img_image)
98  cv2.imwrite('output/2a_k_7.jpg', kmean_7_image)
99  cv2.imshow('2a_kmean_7_image', kmean_7_image)
100
101 kmean_9_image = kmean_function(9, img_image)
102 cv2.imwrite('output/2a_k_9.jpg', kmean_9_image)
103 cv2.imshow('2a_kmean_9_image', kmean_9_image)
104 cv2.waitKey(0)
105 #"""
```

對於每一個 k，重複做 50 次 step1~4，然後取最小的 objective function 當結果。

step1. 隨機挑選圖上的 k 個點當 cluster center。

Step2. 決定每一個點應該在哪個 cluster 中。用 select_point_to_center()做。

Step3. 對於每個 cluster，用 mean 決定新的 cluster center。用 find_new_center()做。

Step4. 如果 cluster center 有改變，那麼 repeat step2。

k=5

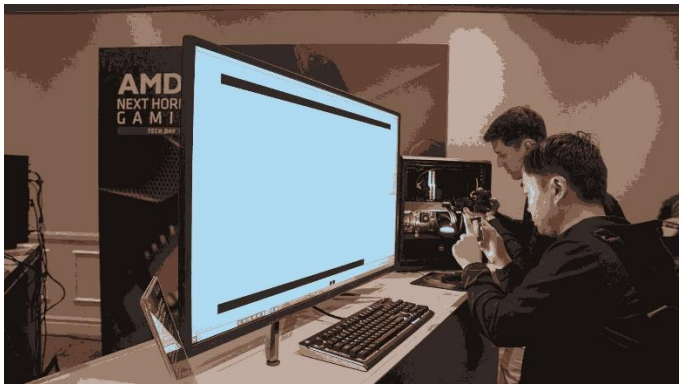
2-image.jpg

2-masterpiece.jpg

k=7

2-image.jpg

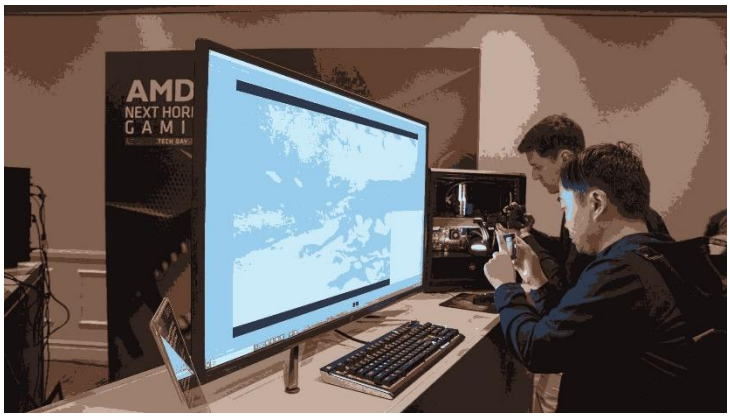
2-masterpiece.jpg



k=9

2-image.jpg

2-masterpiece.jpg



Please discuss the difference between the results for different K's.:

當 k 越小，圖片中的顏色越少。當 k 越大，圖片中的顏色越多，也比 k 比較小的時候更清楚、更有細節一些。

B.

```
# k-means++
# randomly choose the first center
h, w, channel = img_image.shape

def kmean_plus_select_center(k, point): # point: (# of pixel)x3
    kmean_plus_center = []
    kmean_plus_center.append(point[np.random.randint(low=0, high=h*w-1, size=1)])
    center_num = 1
    while center_num<=k:
        center_num = center_num + 1
        # pick the next new center from dataset with prob
        D = 0 # initial
        dist = [] # dist[i]: dist[i]: the distance between point[i] and (all exist center).
        for i in np.arange(len(point)):
            for j in np.arange(len(kmean_plus_center)):
                D = D + np.linalg.norm(point[i]-kmean_plus_center[j])
            dist.append(D)
        # randomly choose next center
        #cum
        dist = dist / np.sum(dist)
        dist = np.cumsum(dist)
        dist = np.insert(dist, 0, 0) # np.insert(array, pos, value)
        # dist = [0, ..., 1]
        ran_num = np.random.uniform(0, 1) # randomly choose from [0, 1]
        for i in np.arange(len(dist)-1):
            if dist[i] <= ran_num < dist[i+1]: # find the interval
                kmean_plus_center.append(point[i])
                break
    return kmean_plus_center

def kmean_plus_function(k, image):
    point = image.reshape(-1, 3)
    kmean_plus_center = kmean_plus_select_center(k, point)
    new_center, img_with_cluster_center = kmean_step23(k, point, kmean_plus_center)
    # show image
    # index i -> min_obj_func_img_with_cluster_center[i](which cluster j of point i) -> min_obj_func_center[j](the color)
    kmean_plus_img = np.array(new_center)[img_with_cluster_center]
    kmean_plus_img = np.array(kmean_plus_img.reshape(image.shape), dtype=np.uint8)
    return kmean_plus_img


#"""
kmean_plus_5_image = kmean_plus_function(5, img_image)
cv2.imwrite('output/2b_k_5.jpg', kmean_plus_5_image)
cv2.imshow('2b_kmean_plus_5_image', kmean_plus_5_image)

kmean_plus_7_image = kmean_plus_function(7, img_image)
cv2.imwrite('output/2b_k_7.jpg', kmean_plus_7_image)
cv2.imshow('2b_kmean_plus_7_image', kmean_plus_7_image)

kmean_plus_9_image = kmean_plus_function(9, img_image)
cv2.imwrite('output/2b_k_9.jpg', kmean_plus_9_image)
cv2.imshow('2b_kmean_plus_9_image', kmean_plus_9_image)
cv2.waitKey(0)
#"""
```

挑選完一開始的 k 個 center 的過程：

Step1. 隨機挑選第一個 center。

Step2. 對於尚未被選到的每個點，都計算 $\sum_i \ ||p - c_i||^2$ ，再按照輪盤法選出下一個 center。

Step3. Repeat step2 until k centers are selected.

挑選完一開始的 k 個 center 後，之後的過程都跟 kmean 相同。

k=5

2-image.jpg

2-masterpiece.jpg

k=7

2-image.jpg

2-masterpiece.jpg

k=9

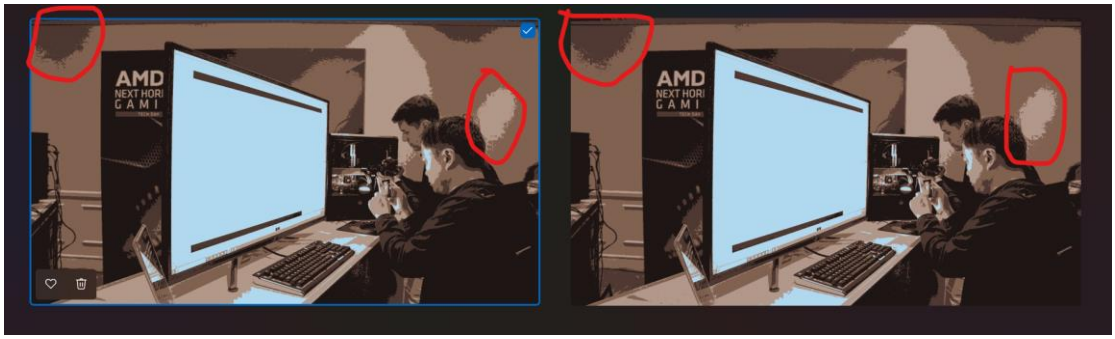
2-image.jpg

2-masterpiece.jpg

Discuss the difference between (A) and (B).:

根據助教給的、自己提供的圖片，k-means 和 k-means++的結果相近。

K=5，只有背景(例如：紅色圈起來的地方)有些許差異。



C.

```python
# mean shift

def find_next_center_meanshift_RGB(re_image, center_now, bandwidth_square):
    row_re_image, col_re_image = re_image.shape
    diff_re_image_center_now = re_image - center_now
    dist_matrix = np.square(diff_re_image_center_now) # dist_matrix: (# of pixel) x 3(RGB)
    dist_matrix = np.sum(dist_matrix, axis=1) # dist_matrix: (# of pixel)
    # now, element of dist_matrix is (R-Ri)^2+(G-Gi)^2+(B-Bi)^2

    weight_index = np.where(dist_matrix<=bandwidth_square) #, True, False) # 1:distance between image[i][j] and center is small enough, 0:o.w.

    re_weight = weight_index
    dist_multi_weight = re_image[re_weight]
    #dist_multi_weight = np.where(dist_multi_weight>0) # array(size:rowx3) with nonzero element in dist_multi_weight

    next_center = np.mean(dist_multi_weight, axis=0)
    next_center = next_center.astype(int)

    return next_center

img_h, img_w = img_image.shape[:2] # img_image[img_h][img_w]
img_h_resize, img_w_resize = int(img_h/4), int(img_w/4)

img_resize = cv2.resize(img_image, (img_w_resize, img_h_resize))
#cv2.imshow('2c', img_resize)
#cv2.waitKey(0)
re_image = img_resize.reshape(-1, 3)
re_image = np.array(re_image, dtype=np.int32)


def meanshift(re_image, bandwidth_square):
    img_meanshift = img_resize.copy() # initial

    for i in np.arange(img_h_resize): #*img_w_resize):
        for j in np.arange(img_w_resize):
            center = re_image[i*img_w_resize + j]  #[i][j]
            print("i, j, center", i, j, center)
            next_center = center

            while 1 :
                next_center = find_next_center_meanshift_RGB(re_image, center, bandwidth_square)
                print("next_center", next_center)
                if not np.array_equal(center, next_center):
                    center = next_center
                else:
                    break

            add_next_center = next_center[:3]
            img_meanshift[i][j] = add_next_center
            #print(img_meanshift, i) #, j)

    img_meanshift = np.array(img_meanshift, dtype=np.uint8)
    #img_meanshift = np.array(img_meanshift.reshape(re_image.shape), dtype=np.uint8)
    return img_meanshift


#
def plotrgb(ori_image, new_image, img_h_resize, img_w_size):
    fig = plt.figure()
    axis = plt.axes(projection="3d")
    x = []
    y = []
    z = []
    color = []
    for i in np.arange(img_h_resize):
        for j in np.arange(img_w_resize):
            ori_pixel = ori_image[i][j]
            new_pixel = new_image[i][j] #[i, j]
            newcolor = (new_pixel[0]/255, new_pixel[1]/255, new_pixel[2]/255)
            #if not newcolor in color :
            x.append(ori_pixel[0])
            y.append(ori_pixel[1])
            z.append(ori_pixel[2])
            color.append(newcolor)
    axis.scatter(x, y, z, c = color, marker=".")
    #axis.scatter(r.flatten(), g.flatten(), b.flatten(), facecolors=img_meanshift_RGB.tolist(), marker=".")
    axis.set_xlabel("Red")
    axis.set_ylabel("Green")
    axis.set_zlabel("Blue")
    plt.show()
#"""
plotrgb(img_resize, img_resize, img_h_resize, img_w_resize)
img_meanshift_RGB = meanshift(re_image, 1000) #RGB
cv2.imwrite('output/meanshift_rgb1000.jpg', img_meanshift_RGB)
plotrgb(img_resize, img_meanshift_RGB, img_h_resize, img_w_resize)
```

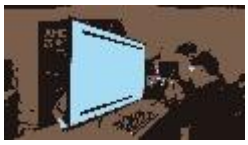
Mean shift 的做法：

把每個點都當作 center，然後計算 bandwidth 內的 mean，把 mean 當作新的

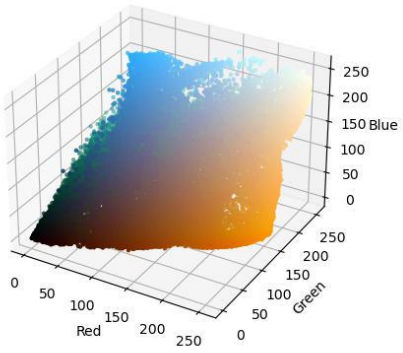
center，然後再計算以新的 center 為中心的 bandwidth 內的 mean，再找出更新的 center，以此類推，一直找出新的 center，直到 center 和新的 center 是同一點。
計算 mean 的方法：(using uniform kernel)
bandwidth 之內的 point 是指：如果 point pi 的(紅, 綠, 藍)=(Ri, Gi, Bi), center 的(紅, 綠, 藍)=(R, G, B)，(R-Ri)^2+(G-Gi)^2+(B-Bi)^2 <= bandwidth_square，那麼 pi 就是 bandwidth 之內的 point。
把 bandwidth 之內的 point 值(R, G, B)相加，然後除以 bandwidth 之內的 point 的數量。

the clustered result:
2-image.jpg: `bandwidth_square=1000, resize->1/4`

2-masterpiece.jpg: `bandwidth_square=1000, resize->1/16`

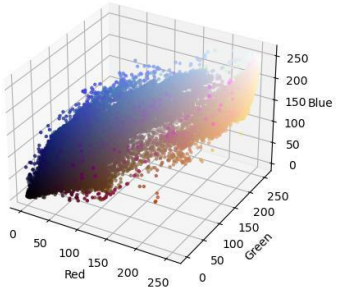


the pixel distributions in the R*G*B feature space before applying mean-shift:
2-image.jpg : `resize->1/4`

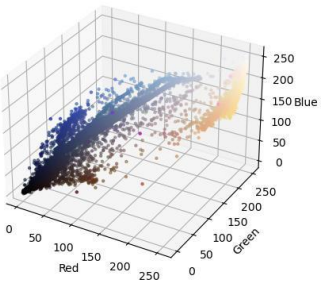


2-masterpiece.jpg:

`resize->1/4`

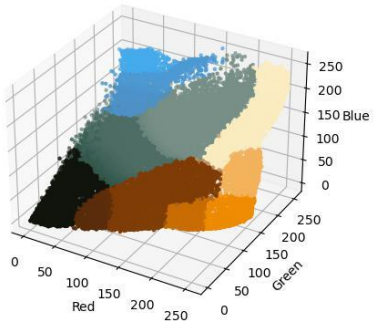


resize->1/16

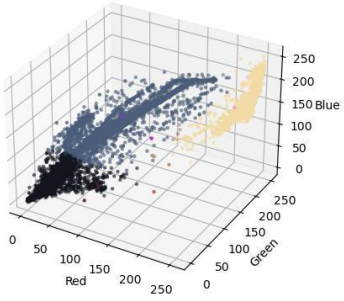


the pixel distributions in the R*G*B feature space after applying mean-shift:

2-image.jpg `:resize->1/4`



2-masterpiece.jpg:`resize->1/16`



D.

```
254
255  # RGBxy
256  re_image_RGBxy = [] # = img_resize.copy()
257  #xx, yy = np.meshgrid(np.arange(img_w_resize), np.arange(img_h_resize))
258  for i in np.arange(img_h_resize):
259      for j in np.arange(img_w_resize):
260          re_image_RGBxy = np.append(re_image_RGBxy, [img_resize[i][j][0], img_resize[i][j][1], img_resize[i][j][2], i, j])
261
262
263
264  re_image_RGBxy = re_image_RGBxy.reshape(-1, 5)
265  re_image_RGBxy = np.array(re_image_RGBxy, dtype=np.int32)
266  img_meanshift_RGBxy = meanshift(re_image_RGBxy, 50)
267
268  cv2.imwrite('output/meanshift_rgbxy.jpg', img_meanshift_RGBxy)
269
270
```

C 小題是把每一個點看成 3 維[R, G, B]的向量，D 小題則是把每一個點看成 5 維[R, G, B, x, y]的向量。其他地方都差不多。

the color and spatial information:

2-image.jpg: `bandwidth_square=50, resize->1/4`

2-masterpiece.jpg: `bandwidth_square=50, resize->1/16`

E.

```
243   #"""
244   plotrgb(img_resize, img_resize, img_h_resize, img_w_resize)
245   img_meanshift_RGB = meanshift(re_image, 1000) #RGB
246   cv2.imwrite('output/meanshift_rgb1000.jpg', img_meanshift_RGB)
247   plotrgb(img_resize, img_meanshift_RGB, img_h_resize, img_w_resize)
248
249   img_meanshift_RGB = meanshift(re_image, 400) #RGB
250   cv2.imwrite('output/meanshift_rgb400.jpg', img_meanshift_RGB)
251   img_meanshift_RGB = meanshift(re_image, 8000) #RGB
252   cv2.imwrite('output/meanshift_rgb8000.jpg', img_meanshift_RGB)
253
```

the clustered result:

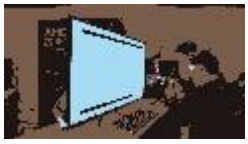
2-image.jpg: `bandwidth_square=400, resize->1/4`

2-masterpiece.jpg: `bandwidth_square=400, resize->1/16`

the clustered result:

2-image.jpg: `bandwidth_square=1000, resize->1/4`

2-masterpiece.jpg: `bandwidth_square=1000, resize->1/16`



the clustered result:

2-image.jpg: `bandwidth_square=8000, resize->1/4`

2-masterpiece.jpg: `bandwidth_square=8000, resize->1/16`

Discuss the segmentation results for different bandwidth parameters.:

不同的 bandwidth parameters 會有不同顏色的區分程度。當 bandwidth parameters 越小時，不同顏色之間會從一個顏色慢慢變化到另一個顏色；當 bandwidth parameters 越大時，不同顏色之間就會區分的越明顯，不再會以慢慢的方式變化。

例如：天空的顏色沿著紅色的線走，左圖(`bandwidth_square=400`)要經過好幾個顏色才變化完，右圖(`bandwidth_square=8000`)只經過 2 個顏色(深藍和淺藍)。

當 bandwidth parameters 越大時，一張圖片中所含的顏色也會越來越少。

F.

Compare the segmentation results by using K-means and mean-shift algorithms and their computational cost.:

跟 k-means 做比較，mean-shift 需要計算的時間更長，也就是 mean-shift 的 time computational cost 更大。