CS6550 Computer Vision Homework 3

112062673 吳文雯

Q1. Alignment with RANSAC:

(A)SIFT

1.Implementation

step 1

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
import math
import random

#讀圖片且轉成灰階 要跟 code 放在同一個資料夾內
img01 = cv2.imread('1-book1.jpg')
img02 = cv2.imread('1-image.jpg')
img03 = cv2.imread('1-book2.jpg')
img04 = cv2.imread('1-book3.jpg')

img01g = cv2.imread('1-book1.jpg',cv2.IMREAD_GRAYSCALE)
img02g = cv2.imread('1-image.jpg',cv2.IMREAD_GRAYSCALE)
img03g = cv2.imread('1-book2.jpg',cv2.IMREAD_GRAYSCALE)
img04g = cv2.imread('1-book3.jpg',cv2.IMREAD_GRAYSCALE)

#建立 sift
sift = cv2.SIFT_create(nfeatures=0, nOctaveLayers=3, contrastThreshold=0.04,
edgeThreshold=10, sigma=1.6)

#取得特徵點及 descriptor
keypoints01, descriptors01 = sift.detectAndCompute(img01g, None)
keypoints02, descriptors02 = sift.detectAndCompute(img02g, None)
keypoints03, descriptors03 = sift.detectAndCompute(img03g, None)
```

```
keypoints04, descriptors04 = sift.detectAndCompute(img04g, None)
n_keypoints01 = len(keypoints01)
n_keypoints02 = len(keypoints02)
n_keypoints03 = len(keypoints03)
n_keypoints04 = len(keypoints04)
...
```

- Read the input images in its original color and with grayscale version.
- Use the grayscale images to get the key points and descriptors from "sift.detectAndCompute" function.

step 2

```
...
#在圖 2 中找跟 k1 的 descriptor 最相近的 keypoint
def find_match(n_keypoints01,descriptors01,n_keypoints02,descriptors02,t):
        min_1 = {}
    min_2 = {}
    match = {}
    for k1, A in zip(range(n_keypoints01), descriptors01):
        min_dist1 = float('inf')
        min_b1 = None

        for k2, B in zip(range(n_keypoints02), descriptors02):
            dist = np.linalg.norm(A - B)
            if dist < min_dist1:
                min_dist1 = dist
                min_b1 = k2
        min_1[k1] = min_b1

        min_dist2 = float('inf')
        min_b2 = None
        for k2, B in zip(range(n_keypoints02), descriptors02):
            if k2 != min_b1:
                dist = np.linalg.norm(A - B)
                if dist < min_dist2:
                    min_dist2 = dist
                    min_b2 = k2
```

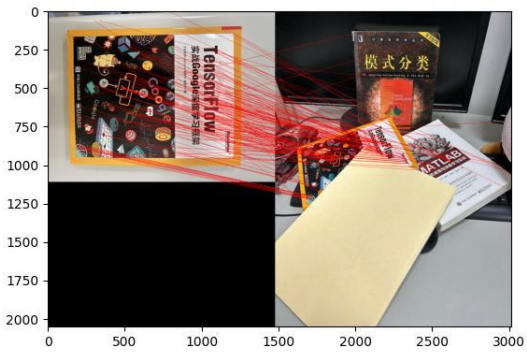
```
        min_2[k1] = min_b2
        #只接受 d(v,b1)/d(v,d2)<1/2 的點
        if (min_dist1/min_dist2)<t:
            match[k1] = [min_b1, min_dist1]
    return match

match1 = find_match(n_keypoints01,descriptors01,n_keypoints02,descriptors02)

# Draw the matches
Matches01 = [[cv2.DMatch(k, v[0], 0) for k, v in match1.items()]]
img_match = cv2.drawMatchesKnn(img01, keypoints01, img02, keypoints02,
matches01,None, matchColor=(0, 0,
255),flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Convert BGR image to RGB
img_match_rgb = cv2.cvtColor(img_match, cv2.COLOR_BGR2RGB)
plt.imshow(img_match_rgb)
plt.savefig('./output/hw3-1t1.jpg')
plt.show()
...
```

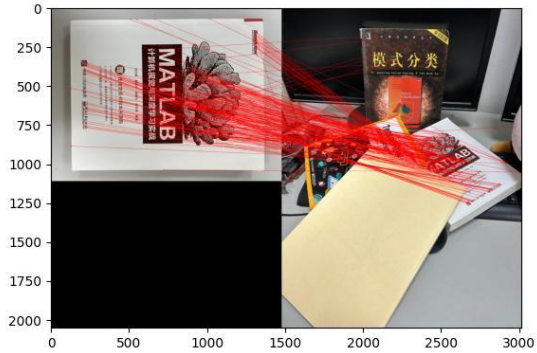
- Define a function to find the match points in "1-image.jpg":
  - For each key points in the image with only 1 book(left image), scan through all the key points in "1-image.jpg" and get the key points with smallest and second smallest distance to the points in the left image.
  - If the ratio between the two key points chosen on the right image is lower than the threshold, which is set to 0.7 for book1 and book2, 0.5 for book3, this pair will be added into the match library. (The reason why I set different threshold for book 3 is to prevent to many matching lines drawn on the image.)
  - Finally return the match library.

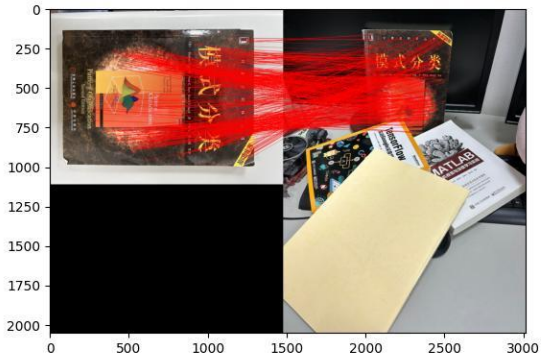
- Print the number of matches in the 2 images, and

  use “cv2.drawMatchesKnn” function to draw the matching lines on the

  original image with BRG color.
- Convert the image into RBG version and show the image by “plt.show.”
- The output images are as below:







(B) RANSAC

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]

def transform(points,H):
    num = len(points)
    transp = np.zeros_like(points[:, :2])
    for i in range(num):
        hpoints = np.hstack((points[i, :2],1))
        thpoints = np.dot(H, hpoints)
        transp[i] = thpoints[:2]/thpoints[2]
    return transp

def get_dist(matches, H,keypoints01,keypoints02):
    num = len(matches)
    p1 = np.array([[keypoints01[m.queryIdx].pt[0], keypoints01[m.queryIdx].pt[1]]
for m in matches])
    p2 = np.array([[keypoints02[m.trainIdx].pt[0], keypoints02[m.trainIdx].pt[1]]
for m in matches])

    t_p1 = transform(p1, H)
    dist = np.linalg.norm(p2 - t_p1, axis=1) ** 2

    return dist

def RANSAC(matches,keypoints01,keypoints02,iters,threshold):
    best_num = 0
    n = 4
     for i in range(iters):
        indices = random.sample(range(len(matches)), n)
```

```
        sample = [matches[idx] for idx in indices]
        src = np.float32([keypoints01[m.queryIdx].pt for m in sample]).reshape(-1,
2)
        dst = np.float32([keypoints02[m.trainIdx].pt for m in sample]).reshape(-1,
2)
        H = Find_Homography(src,dst)

        dist = get_dist(matches,H,keypoints01,keypoints02)
        inlier = np.array(matches)[np.where(dist < threshold)[0]]
        num = len(inlier)

        if num > best_num:
            best_num = num
            best_H = H.copy()
            best_inlier = inlier.copy()
            best_inlier = best_inlier.tolist()

    return best_H, best_inlier
```

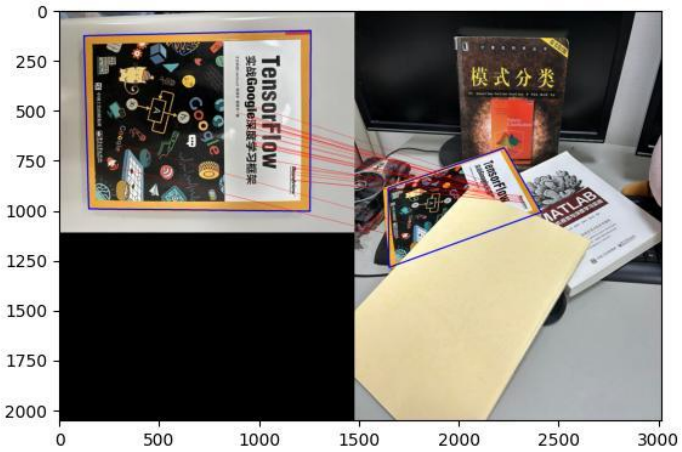
- Define "`Find_Homography`" function to compute homography from the corresponding points.(This is same as previous homework so the details are omitted here.)
- Define "`transform`" function to transform the input points by the given homography. Each points will be multiplied by the homography and the returned data is the coordinates of the transformed points.
- Define "`get_dist`" function. In this function, all the points in the left image will be transformed by the homography to the right image, and the distance between the transformed points and the original key points will be computed.
- Define "`RANSAC`" function:
  - In each iteration, random sample 4 corresponding points from the input matches.
  - Compute the homography by these 4 pair of points.
  - Get the distance from "`get_dist`" function, and for those key points in the right image with a distance lower than the threshold, they are inliers.
  - If the number of inliers is the max, the inliers and the corresponding H will be kept and returned after the iterations finish.

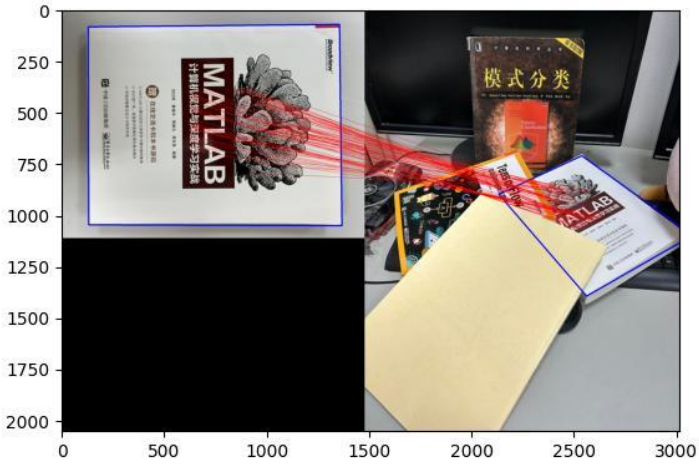
step 2

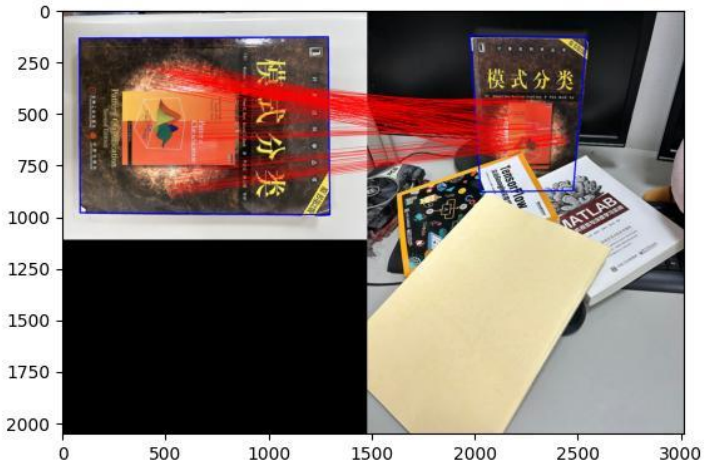
```
def transform_p(points, H):
    num = len(points)
    transp = np.zeros_like(points[:, :2])

    for i in range(num):
        hpoints = np.hstack((points[i, :2], 1))
        thpoints = np.dot(H, hpoints)
        transp[i] = thpoints[:2] / thpoints[2]

    return transp

def draw_rectangle(img, corners):
    for i in range(4):
        cv2.line(img, tuple(corners[i]), tuple(corners[(i + 1) % 4]), color=(255,
0, 0), thickness=5)
    return img

H1, inlier1 = RANSAC(matches1,keypoints01,keypoints02,500,0.5)
inlier_matches1 = [cv2.DMatch(m.queryIdx, m.trainIdx, m.distance) for m in
inlier1]

corners1 = np.array([[124,124], [1260,98], [1256,1002], [148,990]],
dtype=np.int32)
hcorners1 = transform(corners1,H1)

img02_1 = cv2.imread('1-image.jpg')
img01 = draw_rectangle(img01, corners1)
img02_1 = draw_rectangle(img02_1, hcorners1)

img_matches = cv2.drawMatchesKnn(img01, keypoints01, img02_1, keypoints02,
[inlier_matches1],None, matchColor=(0, 0,
255),flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

img_match_rgb = cv2.cvtColor(img_matches, cv2.COLOR_BGR2RGB)

plt.imshow(img_match_rgb)
```

```
plt.savefig('./output/hw3-1-B1.jpg')
plt.show()
```

- Get the homography H and the inlier from RANSAC, and construct list of DMatch objects for the inliers.
- Indicate the corners on the left image and get the transformed corners by the homography using "`transform_p`" function. Then draw the two rectangles on the certain image using "`draw_rectangle`" function.
- Draw the matching lines between the two images, and then show and save the image.
- The output images are shown below:



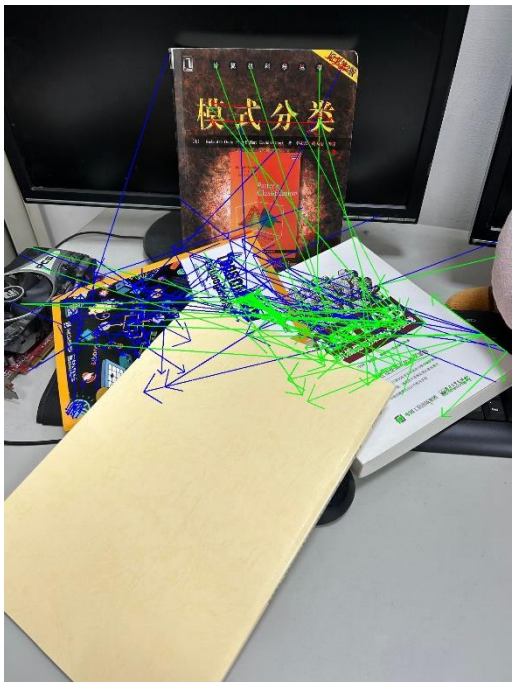




step 3

```
img02_p = cv2.imread('1-image.jpg')
points1 =  np.float32([keypoints01[m.queryIdx].pt for m in matches1]).reshape(-1,
2)
t_points1 = transform_p(points1, H1)
o_points1 = np.float32([keypoints02[m.trainIdx].pt for m in matches1]).reshape(-1,
2)
...
(similar codes for book 2 and 3)
...
for i in range(len(o_points1)):
    pt1 = (int(o_points1[i, 0]), int(o_points1[i, 1]))
    pt2 = (int(t_points1[i, 0]), int(t_points1[i, 1]))

    cv2.arrowedLine(img02_p, pt1,pt2,(255,0, 0), 2)
...
(similar codes for book 2 and 3)
...
filename3 = f'./output/hw3-1-B4.jpg'
cv2.imwrite(filename3, img02_p)
cv2.imshow('arrows', img02_p)
cv2.waitKey(0)
```

- Read the image “1-image.jpg” again and name it as image_02p.
- Transform all the key points in the matches get from SIFT on the left image to the right image by its corresponding homography.
- Draw all the arrow lines between the transformed points and the original key points, and then show and save the image.
- The output image shows difference of result get by SIFT and RANSAC:
  (blue lines for book1, green lines for book 2, red lines for book 3)

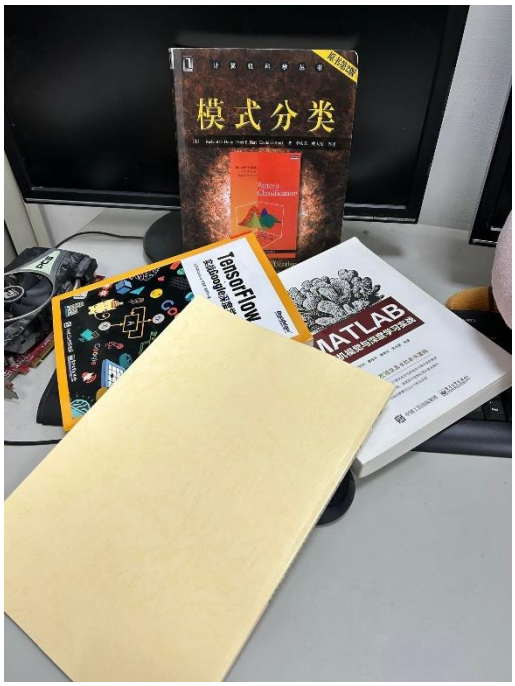


step 4

```
img02_p = cv2.imread('1-image.jpg')
points1 =  np.float32([keypoints01[m.queryIdx].pt for m in inlier1]).reshape(-1, 2)
t_points1 = transform_p(points1, H1)
o_points1 = np.float32([keypoints02[m.trainIdx].pt for m in inlier1]).reshape(-1, 2)
...
(similar codes for book 2 and 3)
...
for i in range(len(o_points1)):
    pt1 = (int(o_points1[i, 0]), int(o_points1[i, 1]))
    pt2 = (int(t_points1[i, 0]), int(t_points1[i, 1]))

    cv2.arrowedLine(img02_p, pt1,pt2,(255,0, 0), 2)
```

```
...
(similar codes for book 2 and 3)
...
filename3 = f'./output/hw3-1-B5.jpg'
cv2.imwrite(filename3, img02_p)
cv2.imshow('arrows', img02_p)
cv2.waitKey(0)
```

- Read the image “1-image.jpg” again and name it as image_02p.
- Transform all the inliers in the left points get from RANSAC on the left image to the right image by its corresponding homography.
- Draw all the arrow lines between the transformed points and the corresponding key points, and then show and save the image.
- The output image shows that RANSAC can get proper homography so that the arrow lines are too close and look like dots:
  (blue lines for book1, green lines for book 2, red lines for book 3)



Discussion:

When using SIFT, there are lots of mismatches even though the ratio test has performed and the parameter of ratio threshold is not large. This may be because the book is rotated in a larger angle, so SIFT cannot make proper matching on the 2 images. However, the result from RANSAC is much better. The parameters in RANSAC are the number of iterations and the threshold. The number of iterations

should not be too small so that it's possible to get better solution. Also, the threshold for finding inliers should not be set too large, which can prevent including outliers in the computation. Therefore, RANSAC can eliminate the influence of some outliers and get the better homography. The result shown on the image in step 2 are only inliers which are matched more properly.

Q2.Image segmentation:

(A)K-means

1.Implementation

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

image = cv2.imread('2-image.jpg')
data = image.reshape((-1,3))
data = data.astype(np.float32)

k_list = [4,6,8]

#randomly initialize centers
def init_center(data,k):
    index = np.random.choice(len(data),k,replace=False)
    return data[index]

def assign_cluster(data, center):
    cluster = np.argmin(np.linalg.norm(data[:,np.newaxis]-
center,ord=2,axis=2),axis=1)
    return cluster

def update_center(data, clusters, k):
    new_center = np.array([data[clusters == j].mean(axis=0) for j in range(k)])
    return new_center
```

```
def kmeans(data,k,iters,initial_num):
    best_center = best_cluster = 0
    best_dist = float('inf')
    for i in range(initial_num):
        center = init_center(data,k)
        for i in range(iters):
            cluster = assign_cluster(data,center)
            new_center = update_center(data,cluster,k)
            if np.all(center == new_center):
                break
            center = new_center
        #compute the sum of distance to the closest center
        dist = np.sum(np.min(np.linalg.norm(data[:, np.newaxis]-center,
axis=2),axis=1))
        if dist < best_dist:
            best_dist = dist
            best_center = new_center
            best_cluster = cluster
    return best_center, best_cluster

for k in k_list:
    #max iteration number=100, initial guess = 50
    center, cluster = kmeans(data,k,100,50)
    center = np.uint8(center)
    seg_data = center[cluster.flatten()]
    seg_img = seg_data.reshape(image.shape)

    filename = f'./output/hw3-2-Am_k={k}.jpg'
    cv2.imwrite(filename, seg_img)
    cv2.imshow(f'hw3-2-Am_k={k}.jpg', seg_img)
    cv2.waitKey(0)
```

- Read the image and reshape it into a 2-d array with np.float32 data type.
- Set the value of k to be 4, 6 and 8, and set the number of max iterations in k-means as 100 and the number of initial guesses is 50.
- Implement K-means function:
  - First get the initial center by “init_center”, and this function randomly

choose k numbers as the indices of the centers.

- Assign each pixel in the image to their nearest center by “assign_cluster.” In this function, the distance between each pixel and each center is calculated. Then the indices of the center with smallest distance will be returned to be the cluster assigned to the pixel.
- Update the centers of each cluster by the pixels in the cluster, the “update_center” function will return the new mean of those pixels as the new center.
- If the new centers are the same as the previous centers, the iteration will be stopped. Otherwise, it will end after finishing 100 iterations.
- Computer the distance of each pixel to the centers of the cluster they belong to, and keep the record of the minimal distance and the corresponding centers and clusters.
- Since the number of initial guesses is 50, it will return the best result among the 50 times of k-means operation.

- Convert the type of the center into np.uint8 and map each pixel to their corresponding center. Then show the segmented image.
  (The code for “2-masterpiece.jpg” is the same, so it’s omitted here.)
- The output images will be put together with the output from k-means++ for comparison.

(B) K-means++

1.Implementation

```
image = cv2.imread('2-image.jpg')
data = image.reshape((-1,3))
data = data.astype(np.float32)

k_list = [4,6,8]

#kmeans ++
def kmeans_pp_init(data,k):
    center = [data[np.random.choice(len(data))]]

    while len(center) < k:
        dist = np.min(np.square(np.linalg.norm(data - np.array(center)[:,
np.newaxis], axis=2)), axis=0)
```

```
        next_center = data[np.random.choice(len(data), p=dist / np.sum(dist))]
        center.append(next_center)

    return np.array(center)

def kmeans_pp(data,k,iters):
    center = kmeans_pp_init(data,k)

    for i in range(iters):
        cluster = assign_cluster(data,center)
        new_center = update_center(data,cluster,k)

        if np.all(center == new_center):
            break
        center = new_center

    return center, cluster

for k in k_list:
    #max iteration number=100
    center, cluster = kmeans_pp(data,k,100)
    center = np.uint8(center)
    seg_data = center[cluster.flatten()]
    seg_img = seg_data.reshape(image.shape)
    ...
```

- K-means++ does not randomly select the initial centers, here I implement "kmeans_pp_init" to get the initial centers:
  - First randomly choose a number as the center index.
  - Compute the distance from the current center point to each pixel in the image, and these distance will be used to form the probability for them to be chosen as the next center.
  - The new center will be appended to the center list, and the previous steps will keep going until we have enough center points.
- The following steps are similar to k-means, the iterations will end until the centers are not changed or finish 100 iterations. But here it's not necessary to keep track of the best result, it will only return the result from the centers chosen from k-means++ since the initialization of the centers is good enough.

- The output images are shown below:

2-image.jpg

K=4

K-means:	K-means++:
	

K=6

K-means:	K-means++:
	

K=8

K-means:	K-means++:
	

2-masterpiece.jpg

(The upper images are from k-means and the lower images are from k-means++)

K=4	K=6	K=8
		
		

Discussion:

For both “2-image.jpg” and “2-masterpiece.jpg”, the result images from k-means and k-means++ are almost the same. This may be because when doing k-means, we choose the best result from the 50 initial guesses and the number of trials is large enough for k-means to achieve performance similar to k-means++.

(C) Mean shift

1.Implementation

step 1

```
---------------------------------------------------------------------------------------------------------------------
import cv2
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

image = cv2.imread('2-image.jpg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

r = image_rgb[:,:,0].ravel()
g = image_rgb[:,:,1].ravel()
b = image_rgb[:,:,2].ravel()

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(r, g, b, c=np.array([r, g, b]).T / 255.0, marker='o', alpha=0.5)

ax.set_xlabel('R')
ax.set_ylabel('G')
ax.set_zlabel('B')

plt.savefig('./output/hw3-2_C1.jpg')
plt.show()    ...
---------------------------------------------------------------------------------------------------------------------
```

- Read the image and convert it into “RGB” form.
- Get the RGB value of each pixel, and render the pixels in RBG space, the color of each point is the same as their RGB value.

step 2

```
---------------------------------------------------------------------------------------------------------------------
...
def uni_kernel(d,bandwidth):
    c = 1
```

```
     return c if d<=bandwidth else 0

def find_closest_center(pixel, centers):
    distances = np.linalg.norm(centers - pixel, axis=1)
    return np.argmin(distances)

def meanshift(image, x_seg, y_seg, bandwidth):
    rows, cols, _ = image.shape
    result = np.zeros_like(image, dtype=np.uint8)
    idx = np.zeros(rows * cols, dtype=np.int32)
    idx = idx.reshape((rows, cols))
    x_size = cols // x_seg
    y_size = rows // y_seg

    # Collect RGB values of center points
    center_points = []
    for i in range(0, rows, y_size):
        for j in range(0, cols, x_size):
            block = image[i:i + y_size, j:j + x_size].reshape(-1, 3)
            center = np.mean(block, axis=0)
            center_points.append(center)

    center_points = np.array(center_points)
    print(center_points)
    print(len(center_points))

    #assgin center
    for i in range(rows):
        for j in range(cols):
            pixel = image[i, j]
            closest_center_idx = find_closest_center(pixel, center_points)
            idx[i,j] = closest_center_idx
    # Apply mean shift in RGB space
    for i in range(center_points.shape[0]):
        shifting = 1
        ncenter = center_points[i]
        center = center_points[i]
        while True :
```

```
            center = ncenter
            shift = np.zeros_like(center, dtype=np.float32)
            ncenter = shift
            weight_sum = 0

            for j in range(center_points.shape[0]):
                d = np.sqrt(np.sum((center - center_points[j]) ** 2))
                w = uni_kernel(d, bandwidth)
                shift += w * center_points[j]
                weight_sum += w

            shift /= weight_sum
            ncenter = shift

            if np.linalg.norm(ncenter - center) < 0.001:
                center_points[i] = ncenter
                for k in range(i-1):
                    if np.linalg.norm(center_points[k] - ncenter) < 0.1:
                        center_points[i] = center_points[k]

                break

    # Assign each pixel in the image to its representing center
    for i in range(rows):
        for j in range(cols):
            pixel = image[i, j]
            #closest_center_idx = find_closest_center(pixel, center_points)
            closest_center_idx = idx[i, j]
            closest_center = center_points[closest_center_idx]
            result[i, j] = closest_center

    return result

bandwidth = 100.0
x_seg = 20
y_seg = 20
seg_img = meanshift(image, x_seg, y_seg, bandwidth)
filename = f'./output/hw3-2-C2.jpg'
```

```
cv2.imwrite(filename, seg_img)
cv2.imshow(f'hw3-2-C_h=20.jpg', seg_img)
cv2.waitKey(0)
```

- Implement the function “meanshift”:
  - To decrease the run time of the program, I partition the image into 20*20 blocks. Then take the mean RGB value of the block to be the RGB value of the center points.
  - Assign each pixel in the image to the closest center by “find_closest_center“, and those pixels will get the same value of its assigned center after finishing mean-shift. After collecting total 400 points, I then apply mean-shift on them to get the local maximum.
  - For each point in the “center_points” array, it will scan through all the points and compute the distance based on the RGB value. The function “uni_kernel” will return the weight, if the distance is lower than the bandwidth, the value is 1, otherwise, 0.
  - After sum up all the valid RGB value, this vector will be divided by the number of weight to get the mean, and this will be new center and also the new value for this point. When the updated center is quite similar to the previous center(distance within 0.001), the loop ends and record the center for this point.
  - Each point will go through the same process to get their final center. If their final center is close to previous center, it will be clustered with previous one and assign the same value of that center.
  - After finishing mean-shift, each pixel in the image will get the same value as its representing center assigned initially.

step 3

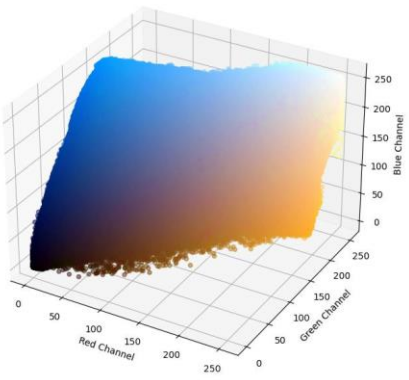
```
...
seg_image_rgb = cv2.cvtColor(seg_img, cv2.COLOR_BGR2RGB)

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

R = seg_image_rgb[:,:,0].ravel()
G = seg_image_rgb[:,:,1].ravel()
B = seg_image_rgb[:,:,2].ravel()
```

```
ax.scatter(r, g, b, c=np.array([R, G, B]).T / 255.0, marker='o', alpha=0.5)

ax.set_xlabel('R')
ax.set_ylabel('G')
ax.set_zlabel('B')

plt.savefig('./output/hw3-2_C3.jpg')
plt.show()
```

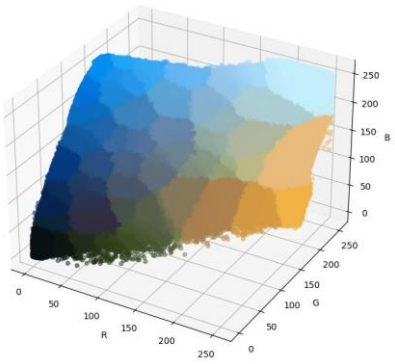
- Convert the segmented image into RGB, and get the RGB value of each pixel.
- Render the points in RGB space, here it uses the original coordinates in RGB space, but the color of each point in their current RGB value.
- The output images are as below:

2-image.jpg:

Before mean-shift

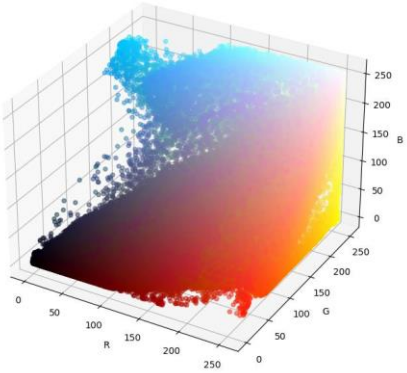


After mean-shift

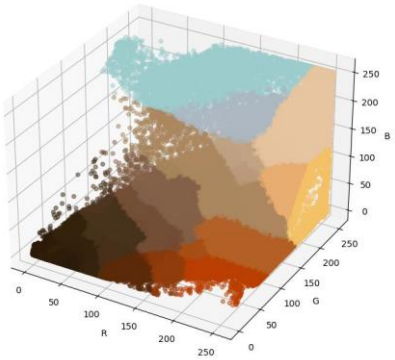


2-masterpiece.jpg:

Before mean-shift



After mean-shift



(D)Mean shift (combine the color and spatial information)

1.Implementation

step 1

```
def meanshift_spatial(image, x_seg, y_seg, hs , hr):
    rows, cols, _ = image.shape
    result = np.zeros_like(image, dtype=np.uint8)
    idx = np.zeros(rows * cols, dtype=np.int32)
```

```
    idx = idx.reshape((rows, cols))
    x_size = cols // x_seg
    y_size = rows // y_seg

    center_points2=[]
    for i in range(0, rows, y_size):
        for j in range(0, cols, x_size):
            block = image[i:i + y_size, j:j + x_size].reshape(-1, 3)
            center_rgb = np.mean(block, axis=0)
            center_coords = (i + y_size // 2, j + x_size // 2)
            center_points2.append((center_rgb, center_coords))
            center_points = np.array(center_points2, dtype=[('rgb', '3float32'),
('coords', '2float32')])

    for i in range(rows):
        for j in range(cols):
            pixel = image[i, j]
            closest_center_idx = find_closest_center(pixel, center_points['rgb'])
            idx[i,j] = closest_center_idx

    #print(len(center_points))

    # Apply mean shift in RGB space
    for i in range(center_points.shape[0]):
        ncenter= center_points[i]
        center = center_points[i]
        while True :
            center = ncenter
            shift = np.zeros_like(center)
            ncenter = shift
            weight_sum = 0

            for j in range(center_points.shape[0]):
                dr = np.sqrt(np.sum((center['rgb'] - center_points[j]['rgb']) **
2))
                if dr <= hr:
                    ds = np.linalg.norm(center['coords'] -
center_points[j]['coords'])
```

```
                w = uni_kernel(ds, hs)
                if w == 1:
                    shift['rgb'] += center_points[j]['rgb']
                    shift['coords'] += center_points[j]['coords']
                    weight_sum += w

            shift['rgb']  /= weight_sum
            shift['coords']  /= weight_sum
            ncenter = shift

            if np.linalg.norm(ncenter['rgb'] - center['rgb']) < 0.001 and
np.linalg.norm(ncenter['coords'] - center['coords']) < 0.001:
                center_points[i] = ncenter
                for k in range(i-1):
                    if np.linalg.norm(center_points[k]['rgb'] - ncenter['rgb'])
<0.1 and np.linalg.norm(center_points[k]['coords'] - ncenter['coords']) < 0.1:
                        center_points[i] = center_points[k]

                break

    # Assign each pixel in the image its repesenting center
    for i in range(rows):
        for j in range(cols):
            pixel = image[i, j]
            #closest_center_idx = find_closest_center(pixel, center_points['rgb'])
            closest_center_idx = idx[i, j]
            closest_center= center_points[closest_center_idx]
            result[i, j] = closest_center['rgb']

    return result
```

- Implement the function “meanshift_spatial”:
  - This is similar to “meanshift” but it also records the x and y coordinates of each center points.
  - For each point in the “center_points” array, it will scan through all the points and compute the distance based on the x and y coordinates first. If the distance is within the spatial bandwidth, the distance of regarding RGB value will then be calculated.

- The function “uni_kernel” will return the weight, if the distance is lower than the color bandwidth, the value is 1, otherwise, 0.
- The RGB value of the new center will be the mean of the points within both spatial and color bandwidth, and the x and y coordinates of the new center are computed similarly.
- When the spatial distance and color distance of the new center and the previous center is close enough, the loop will end. Each pixel in the image will be assigned to the color as its representing centers.

- The output images are as below:

2-image.jpg:

2-masterpiece.jpg

(E) Mean-shift segmentation with different bandwidth parameters

2-image.jpg:

Bandwidth=20

Bandwidth=30

Bandwidth=50

2-masterpiece.jpg:

Bandwidth=20 **Bandwidth=30** **Bandwidth=50**



Discussion:

When the bandwidth becomes larger, more colors will be combined during the shifting process. Therefore, the kinds of colors on the image will be less and making the result image more coarse and different from the original image.

(F) Compare K-means and mean-shift algorithms

The results of K-means and mean-shift are highly depending on the parameters. If the bandwidth is suitable, mean-shift can generate a result quite similar to the input image. However, with proper k, K-means still can generate results better than the results from mean-shift with a high bandwidth.

As for the computational cost, mean-shift algorithm needs to perform the shifting process on all the points, and in every shifting iteration, the point needs to be computed with all the points in the image. On the other hand, K-means algorithm only needs to update those k centers, and in each iteration it only needs to compute every point once. Thus, the computational cost for mean-shift algorithm is much higher than k-means.