CV HW3 Report

-111065524 資應二 李懷

Problem 1. Image Alignment with RANSAC

(A) SIFT feature points detection and matching:

1. Implementation:

Step 1. Sift matching:

First, define sift function that extract the SIFT feature points from 2 input image

```python
def sift(image1, image2, threshold_d=None):
    sift = cv2.SIFT_create()
    keypoints1, descriptors1 = sift.detectAndCompute(image1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(image2, None)
```

Implement Brute-force matching(detailed explanation in HW1 report), we perform ratio test or fixed distance threshold if distance specified to filter the matches, return match points and image with matches drawn on them:

```python
    ratio = 0.7
    matches = []
    #  for each 128D vector in desc1, find a match in desc2
    for i, desc1 in enumerate(tqdm(descriptors1)):
        # i used as queryIdx(first image descriptor vector index)

        # j used as trainIdx(second image descriptor vector index)
        # Calculate distance between desc1 and desc2, L2 norm
        distances = [(j, np.linalg.norm(desc1 - desc2)) for j, desc2 in enumerate(descriptors2)]
        distances.sort(key = lambda x:x[1]) # sort by distance
        best_match, second_best_match = distances[0], distances[1]

        if not threshold_d:
        # pass the ratio test to get matched
            if best_match[1] < ratio * second_best_match[1]:
                matches.append((i, best_match[0], best_match[1])) # image index pair
        else:
            if best_match[1] < threshold_d:
                matches.append((i, best_match[0], best_match[1])) # image index pair

    matches.sort(key = lambda x:x[2])
    print(f"number of matches: {len(matches)}")
    matches = [(cv2.DMatch(i, j, 0)) for i,j,d in matches] # convert to DMatch object, imgIdx = 0

    matched_points = [(keypoints1[match.queryIdx].pt, keypoints2[match.trainIdx].pt) for match in matches]
    # matched_image = cv2.drawMatchesKnn(image1, keypoints1, image2, keypoints2, matches[:60], None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_P
    matched_image = cv2.drawMatches(image1, keypoints1, image2, keypoints2, matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    return np.asarray(matched_points), matched_image
```

Draw the matched images for each pair of book image and input image and save:

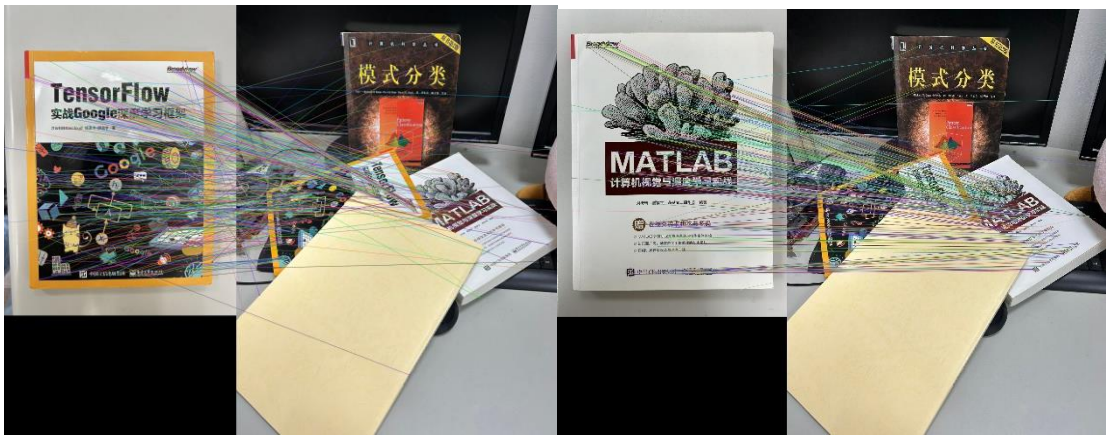
```python
    matches1, matched_image1 = sift(image1, input_image, 0.8)
    draw_and_save(matched_image1, os.path.join(img_path,"output/2(a)sift_match1.jpg"))
```

```python
    matches2, matched_image2 = sift(image2, input_image, 0.75)
    draw_and_save(matched_image2, os.path.join(img_path,"output/2(a)sift_match2.jpg"))
```

```
220     matches3, matched_image3 = sift(image3, input_image, threshold_d=110)
221     draw_and_save(matched_image3, os.path.join(img_path,"output/2(a)sift_match3.jpg"))
```

2. Result:

We can see that there are a lot of outlier matches, we tuned the ratio/distance threshold for each image to make the number of matches around 500 for each, can increase distance threshold to have more matches:



(B) RANSAC Homography

1. Implementation:

Step 1. Define Homography Compute function:

First, we define a compute_homography function that takes into matches of points and compute the homography, the detailed explanation of the code is already discussed in HW2 report:

```
48  def compute_homography(matches):
49      '''
50      given corresponding point and return the homagraphic matrix
51      '''
52      matches = list(matches)
53      A = []
54      for p1,p2 in matches:
55          row1 = [p1[0], p1[1], 1, 0, 0, 0, -p2[0]*p1[0], -p2[0]*p1[1], -p2[0]]
56          row2 = [0, 0, 0, p1[0], p1[1], 1, -p2[1]*p1[0], -p2[1]*p1[1], -p2[1]]
57          A.append(row1)
58          A.append(row2)
59
60      A = np.array(A)
61
62      U, S, Vt = np.linalg.svd(A)
63      # print(U.shape, S.shape, Vt.shape)
64
65      # row of V that corresponds to the smallest singular value of S (that is e
66      H = Vt.T[:,-1].reshape(3, 3)
67      return H
```

Step 2. Define Ransac function for homography:

Define a ransac_homography() function that takes in matches, iterations of ransac, threshold(for counting inliners) and k(which is the minimum number of matches for computing H):

```
69  def ransac_homography(matches, iterations, threshold, k=4):
```

Define select_seed() for randomly selecting 4 matches, and count_inliers() for counting inliers for H computed from random select, it will count the matches whose estimate points(computed from H) are less than the threshold distance of original(sift detected) points as inliers, finally, return the inlier indeices:

```
70      def select_seed(matches, k):
71          '''
72          randomly select matched points
73          '''
74          # k = 4 because homography requires 4
75          index = random.sample(range(len(matches)), k)
76          # print(f"random: {index}")
77          points = [matches[i] for i in index]
78          return np.array(points)
79      def count_inliers(matches, H, threshold):
80          '''count inliers for computed Homography'''
81          inliers = 0
82          num_points = len(matches)
83          pts1 = np.hstack((matches[:, 0, :], np.ones((num_points, 1))))
84          pts2 = matches[:, 1, :]
85          pts2_estimate = np.zeros((num_points, 2))
86          temp = np.dot(H, pts1.T).T
87          pts2_estimate = temp[:, :-1] / temp[:, -1][:, None]
88          # Compute error
89          errors = np.linalg.norm(pts2 - pts2_estimate , axis=1) ** 2
90          inliers = np.where(errors<threshold)[0]
91          return inliers
```

Continue on ransac_homography(), it will loop through number of iterations and random select seed, compute H, and count inliners for that H, and it will store the best H(that has the highest number of inliners) after iterations:

```
93          best_H = 0
94          max_inliers = 0
95          best_inliers = 0
96          best_inliers_indices = 0
97          for i in range(iterations):
98              # print(f"iterations: {i}")
99              random_matches = select_seed(matches, k)
100             H = compute_homography(random_matches)
101             inliers = count_inliers(matches, H, threshold)
102             # print(f"number of inliers: {len(inliers)}")
103             if len(inliers) >= max_inliers:
104                 max_inliers = len(inliers)
105                 best_inliers = matches[inliers]
106                 best_inliers_indices = inliers
107                 best_H = H
108             # print("==============")
109
110         print(f"best H: {best_H}")
111         print(f"max number of inliers: {max_inliers}")
112         print(f"inliers: {best_inliers_indices}")
113         # recompute H using inliers
114         H = compute_homography(best_inliers)
115         return H
```

Step 3. Compute estimate points and draw:

We can then use ransac_homography(iteration set to 1000, inlier threshold set to 50) to compute the best H, and use this H to estimate the points:

```
182     H1 = ransac_homography(matches1, 1000, 50)
183     pts1 = matches1[:, 0, :]
184     pts2_estimate = estimate_points(pts1, H1)
185     pts2_original = [(pt[0], pt[1]) for pt in matches1[:, 1, :]]
186     corner_points = np.array([[97,219],[996,224],[985,1329],[121,1350]])
187     corner_estimate = estimate_points(corner_points, H1)
```

(*4 corner points for drawing boxes are manually specified as TA said we can manually select them.)

*estimate_points() function:

```
165 def estimate_points(pts1, H):
166     pts1 = np.hstack((pts1, np.ones((len(pts1), 1))))
167     temp = np.dot(H, pts1.T).T
168     pts2_estimate = temp[:, :-1] / temp[:, -1][:, None]
169     return pts2_estimate
```

We then draw the points and matches onto the image and save:

```
190 matched_image1 = concat_image(image1, input_image)
191 points_match(matched_image1, pts1, tuple(map(lambda p: (int(p[0] + image1.shape[1]), int(p[1])), pts2_estimate)))
192 draw_box(matched_image1, corner_points, tuple(map(lambda p: (int(p[0] + image1.shape[1]), int(p[1])), corner_estimate)))
193 draw_and_save(matched_image1, os.path.join(img_path,"output/2(b)sift_match1_ransac.jpg"))
```

*points_match() for drawing points and match lines on the image, draw_box() will draw bounding box for the book given 4 corner points:

```
120 def points_match(img, keypoints1, keypoints2, color=(0, 0, 255), radius=5, mode=None):
121     '''draw colored points and match them on image1 and image2'''
122     color2 = (color[1], color[2], color[0])
123     color3 = (color[2], color[1], color[0])
124     for kp1,kp2 in zip(keypoints1, keypoints2):
125         pt1 = tuple(map(int, kp1))
126         cv2.circle(img, pt1, radius, color, -1)
127         pt2 = tuple(map(int, kp2))
128         cv2.circle(img, pt2, radius, color2, -1)
129         if mode:
130             cv2.arrowedLine(img, pt1, pt2, color3, 1)
131         else:
132             cv2.line(img, pt1, pt2, color3, 1)
133
134 def draw_box(img, corners1, corners2, color=(255, 0, 0)):
135     cv2.line(img, corners1[0], corners1[1], color, 5)
136     cv2.line(img, corners1[1], corners1[2], color, 5)
137     cv2.line(img, corners1[2], corners1[3], color, 5)
138     cv2.line(img, corners1[3], corners1[0], color, 5)
139
140     cv2.line(img, corners2[0], corners2[1], color, 5)
141     cv2.line(img, corners2[1], corners2[2], color, 5)
142     cv2.line(img, corners2[2], corners2[3], color, 5)
143     cv2.line(img, corners2[3], corners2[0], color, 5)
144
```

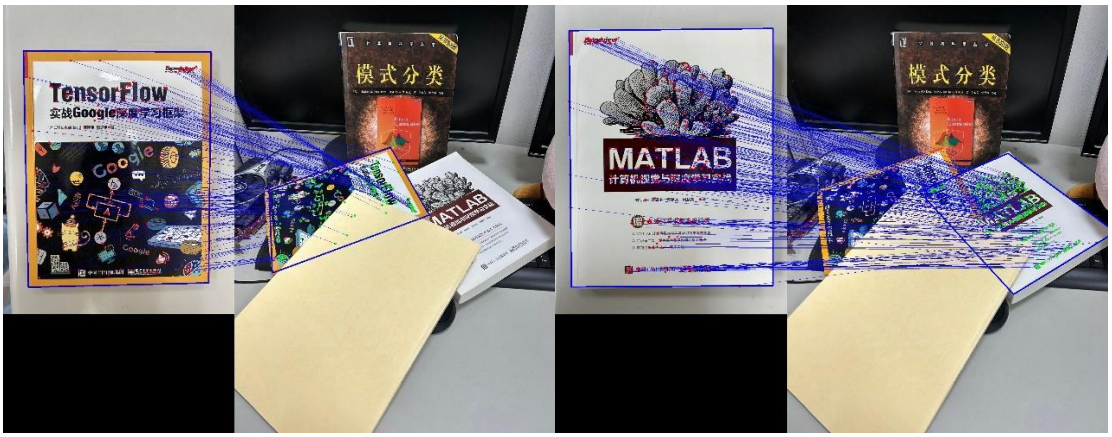
We also want to draw the deviation vectors between the transformed(estimated) feature points and the corresponding feature points(original sift detected ones):

```
195     dv_image1 = input_image.copy()
196     points_match(dv_image1, pts2_original, pts2_estimate, mode=1)
197     draw_and_save(dv_image1, os.path.join(img_path,"output/2(b)dv1.jpg"))
```

※We repeat these processes for image2 and image3(iterations and threshold remains the same).

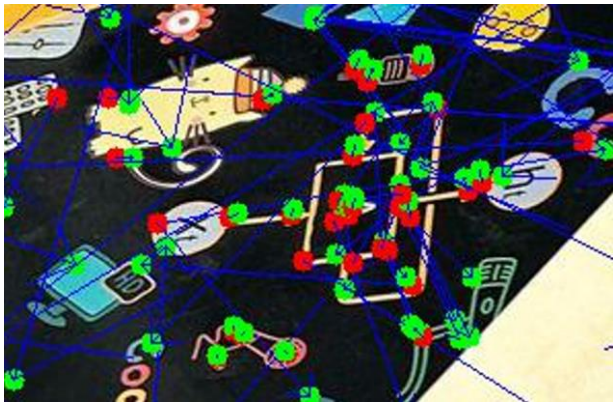
2. Result:

We can see the results using ransac homography are mcuh better than the original sift ones, because it significantly reduces the outliers



Results for deviation vectors(I draw them separately as it is easier to see):

Zoom in to better see the transition between estimate and original points, green points are the estimate, and red points are the original:



3. Discussion

We can see using ransac homography can much better estimate the points than the original sift ones, because it significantly eliminates the outliers.

For sift feature matching, we can set the distance threshold high to allow more matches, however, it will introduce more outlier matches as well, some keypoints 1 will even be matched to the same keypoint 2, so we can also use ratio test to better filter out some outlier matches before RANSAC.

If we use high fixed threshold, it could potentially affect the result of RANSAC, because when there are too many outliers, we have higher chance selecting these outliers in RANSAC select seed step, counting outliers as inliers and compute inaccurate homography, we can also increase the iteration parameters in RANSAC to have more consistent result and increasing the threshold for RANSAC should allow the sensitivity to outliers to reduce.

Problem 2. Image segmentation

(A) K-means on RGB

1. Implementation:

Step 1. Define K-means function:

First, we define a function k_means() that takes in points(image 2d array flatten to 1d), threshold(for centroid update stop criteria), random_seed and mode(for 2(b), discussed later), we randomly select k points as initial centroids(*k_means_init()), then we will repeat this random initial guess processes 50 times until we find the best starting centroids, we compute WCSS (objective function, *kmeans_objective()) and save the initial centroids and clusters with lowest WCSS:

```
32  def k_means(points, k, threshold=1, random_seed=42, mode=None):
33      points = points.astype(np.float32)
34      if not mode:
35          centroids = k_means_init(points, k)
36      else:
37          centroids = k_means_pp_init(points, k)
38      clusters, colors = group(points, centroids)
39
40      if not mode:
41          min_wcss = float("inf")
42          best_clusters = 0
43          best_colors = 0
44          best_centroids = 0
45          for i in tqdm(range(49)):
46              centroids = k_means_init(points, k)
47              clusters, colors = group(points, centroids)
48              wcss = kmeans_objective(points, centroids, clusters)
49              if wcss < min_wcss:
50                  min_wcss = wcss
51                  best_clusters = clusters
52                  best_colors = colors
53                  best_centroids = centroids
54          centroids, clusters, colors = best_centroids, best_clusters, best_colors
```

We then repeat k-means process where we group(*group()) the points to their centroids forming clusters and compute the mean point of each cluster, update the centroid to the new mean if the difference is larger than the threshold, if not, that centroid stops moving, and the whole process terminates when all centroids stop moving, we then return the clusters and its WCSS:

```
    fixed_centroids = [np.array([])]*k
    converges = False
    while True:
        for i, (key, value) in enumerate(clusters.items()):
            new_centroid = np.mean(points[value], axis=0)
            if np.linalg.norm(centroids[i] - new_centroid) > threshold:
                centroids[i] = new_centroid
            else:
                if fixed_centroids[i].size == 0:
                    fixed_centroids[i] = centroids[i]
                # print(f"fixed_centroids: {len([each for each in fixed_centroids if each.size!=0])}")
                converges = all(each.size!=0 for each in fixed_centroids)
                if converges:
                    break
        if converges:
            break
        centroids = [each if each.size!=0 else centroids[i] for i, each in enumerate(fixed_centroids)]
        clusters, colors = group(points, centroids)
    total = 0
    # for each in clusters:
    #     print(len(clusters[each]))
    #     total += len(clusters[each])
    # print(total)

    wcss = kmeans_objective(points, centroids, clusters)
    return clusters, colors, wcss
```

*k_means_init(): for randomly select k initial centroids from points

```
def k_means_init(points, k, random_seed=None):
    '''
    Randomly choose k data points as centroids
    '''
    # random.seed(random_seed)
    centroid_indices = random.sample(range(len(points)), k)
    centroids = points[centroid_indices]
    return centroids
```

*group(): for grouping the points to their centroids forming clusters

```
def group(points, centroids):
    # distances of each pixel to each centroid
    distances = np.sum((points[:, None] - centroids) ** 2, axis=2)
    # index of closest centroid for each pixel
    closest_centroids = np.argmin(distances, axis=1)
    clusters = {centroid_index: [] for centroid_index in range(len(centroids))}
    colors = {centroid_index: centroids[centroid_index] for centroid_index in range(len(centroids))}
    # assign each pixel to the closest cluster
    for i, centroid_index in enumerate(closest_centroids):
        clusters[centroid_index].append(i)

    return clusters, colors
```

*kmeans_objective(): for computing total square-sum distance of each cluster

```
def kmeans_objective(points, centroids, clusters):
    wcss = 0
    for key, cluster in clusters.items():
        centroid = centroids[key]
        points_of_ci = points[cluster]
        distances_to_ci = np.sum(np.linalg.norm(centroid - points_of_ci, axis=1)**2)
        wcss += distances_to_ci
    # print(f"Within-Cluster Sum of Square: {wcss}")

    return wcss
```

Step 2. Image Segmentation:

After obtaining the clusters(a dictionary that stores the indices of pixel) for RGB pixels of the image, we can segment the image then draw and save it, and do the same for k=10 and k=15 and masterpiece image:

```
253     # 2(a) 2-image
254     clusters, colors, wcss = k_means(image1_flatten, 5)
255     print(f"2-image kmeans best wcss(k=5): {wcss}")
256     image1_segemented = segmentation(image1, clusters, colors)
257     draw_and_save(image1_segemented, os.path.join(img_path,"output/2(a)2-image_kmeans_k5.jpg"))
```

*segmentation():takes in image array, clusters dict., and colors(for each cluster), it iterates through the clusters dict. and assign the according color for each centroid

```
83  def segmentation(img, clusters, colors=None):
84      img_segemented = img.copy()
85      k = len(clusters)
86      print(len(clusters))
87      i = 0
88      for key, value in clusters.items():
89          if colors:
90              color = colors[key]
91          if isinstance(clusters, defaultdict):
92              color = key
93              key = clusters[key]
94          img_segemented[np.unravel_index(value, img.shape[:2])] = color
95          i+=1
96      return img_segemented
```

(*some parts of this function are used for later on)

2. Results:

Results of different K for 2-image and 2-masterpiece(K=5, K=10, K=15 from top to bottom)

2(a)2-image_kmeans_k5.jpg

2(a)masterpiece_kmeans_k5.jpg

2(a)2-image_kmeans_k10.jpg

2(a)masterpiece_kmeans_k10.jpg

2(a)2-image_kmeans_k15.jpg

2(a)masterpiece_kmeans_k15.jpg

3. Discussion:

As we increase the number of K for our k-means segmentation, the segmented image will start to look more alike the original image as we increase the number of cluster count, more colors can be presented, so the color gradient transitions (e.g. in the blue sky) will become smoother(color banding).

Also, we calculated the WCSS(within-cluster sum of squares) for different K, as you can see, as we increase K, the WCSS decreases, which is very obvious as there are the more centroids the finer the clusters are.

```
2-image kmeans best wcss(k=5): 2365973480.003558
2-image kmeans best wcss(k=10): 9485637240.469538
2-image kmeans best wcss(k=15): 655679512.3062412
```

(B) K-means++

1. Implementation:

Step 1. Define K-means++ function:

We use the k_means() function discussed in the previous part for k_means++ as they share the steps after selecting the initial centroids, we just have to input mode=1 when we want to call for k_means++, the initial centroids are selected by using *k_means_pp_init() function:

```
32  def k_means(points, k, threshold=1, random_seed=42, mode=None):
33      points = points.astype(np.float32)
34      if not mode:
35          centroids = k_means_init(points, k)
36      else:
37          centroids = k_means_pp_init(points, k)
38      clusters, colors = group(points, centroids)
39
40      if not mode:
41          min_wcss = float("inf")
42          best_clusters = 0
43          best_colors = 0
44          best_centroids = 0
45          for i in tqdm(range(49)):
46              centroids = k_means_init(points, k)
47              clusters, colors = group(points, centroids)
48              wcss = kmeans_objective(points, centroids, clusters)
49              if wcss < min_wcss:
50                  min_wcss = wcss
51                  best_clusters = clusters
52                  best_colors = colors
53                  best_centroids = centroids
54          centroids, clusters, colors = best_centroids, best_clusters, best_colors
```

*k_means_pp_init(): this function is the essential part of k-means++, we first randomly select 1 point as our initial cluster centroid c1, we then compute the distance squares D^2(x) of each point to the c1, then we select the next centroid based on the probability $D^2(x)/\sum x \in c1\, D^2(x)$ (or we can select the furthest one as it has the highest probability to be selected), the selected one will be the next centroid c2, then we the process, compute the distance of point to its belong cluster(that is the shortest centroid to the points, so we do group() here), then select the point that has the highest distance(or probability to be selected) to its centroid as the next centroid, repeat the process until K centroids are selected:

```python
def k_means_pp_init(data, k, random_seed=None):
    data = data.astype(np.float32)
    centroids = []
    c1_index = random.sample(range(len(data)), 1)
    c1 = data[c1_index]
    centroids.append(c1[0])
    for i in range(k-1):
        clusters, colors = group(data,centroids)

        max_distances = 0
        furthest_centroid = 0
        for i, (key, cluster) in enumerate(clusters.items()):
            distances_to_ci = np.linalg.norm((data[cluster][:, None] - centroids[i]), axis=2)
            temp = np.max(distances_to_ci)
            temp_index = np.argmax(distances_to_ci)
            if temp > max_distances:
                max_distances = temp
                furthest_centroid = cluster[temp_index]

        # furthest_centroid = np.argmax(np.max(distances, axis=1))
        # print(distances[furthest_centroid])
        # print(furthest_centroid)
        # print(data[furthest_centroid])
        centroids.append(data[furthest_centroid])
    # print(centroids)
    return centroids
```

We then can continue on the rest of the k-means process (“50 random initial guesses to select the best result” will not be triggered when in k-means++ mode as it is not stated in (B).).

Step 2. Image Segmentation:

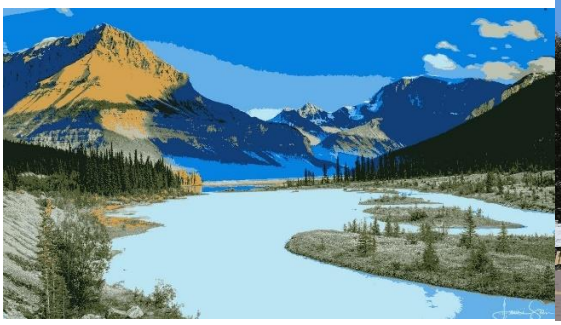
Obtain clusters by calling k_means with mode = 1, segment the image into clusters and draw the images (repeat for k=10, k=15 and masterpiece image)

```python
    # 2(b) 2-image
    clusters, colors, wcss = k_means(image1_flatten, 5, mode=1)
    print(f"2-image kmeans++ wcss(k=5): {wcss}")
    image1_segemented = segmentation(image1, clusters, colors)
    draw_and_save(image1_segemented, os.path.join(img_path,"output/2(b)2-image_kmeans++_k5.jpg"))
```

2. Results:

2(b)2-image_kmeans++_k5.jpg

2(b)masterpiece_kmeans++_k5.jpg



2(b)2-image_kmeans++_k10.jpg

2(b)masterpiece_kmeans++_k10.jpg

2(b)2-image_kmeans++_k15.jpg

2(b)masterpiece_kmeans++_k15.jpg

3. Discussion:

Compare results in (A) and (B):

although we can see some minor differences in the segmented images between kmeans and kmeans++ (with the same K same image), however, it is not easy for the human eyes to judge whether the results are much better as it is very subjective, so we can instead compare the two by comparing WCSS:

```
2-image kmeans best wcss(k=5): 2239554617.785165
2-image kmeans++ wcss(k=5): 2366270775.6869907
2-image kmeans best wcss(k=10): 1435222224.4571865
2-image kmeans++ wcss(k=10): 977024295.7263659
2-image kmeans best wcss(k=15): 630558973.4327586
2-image kmeans++ wcss(k=15): 630137041.9604331
```

We can see kmeans++ has better WCSS when K=10 and 15, but slightly worse when K=5, so it means that kmeans++ does not always guarantee the better initial guesses than kmeans, but it should be more consistent, and since we do the 50 initial guesses process for kmeans here, the result should not be that bad as with only one guess.

(C) Mean-shift on RGB

1. Implementation:

Step 1. Define mean shift function:

First, we define mean_shift() function which takes image array, bandwidth(rgb space), c(default to 1 for uniform kernel, does not affect the result), spatial and bandwidth2(for spatial space):

```
136  def mean_shift(img, bandwidth, c=1, spatial=False, bandwidth2=None):
137      if spatial and bandwidth2 is None:
138          raise ValueError("Bandwidth2 is required when add in spatial.")
139
140      MIN_DISTANCE = 1
141      if spatial:
142          m, n, _ = img.shape
143          # Create x and y coordinate grids
144          x, y = np.meshgrid(np.arange(n), np.arange(m))
145          img_copy = np.zeros((m, n, 5), dtype=np.uint8)
146          img_copy[:, :, :3] = img
147          img_copy[:, :, 3] = x
148          img_copy[:, :, 4] = y
149          img = img_copy
150      img_faltten = img.reshape((-1,img.shape[-1]))
151      img_faltten = img_faltten.astype(np.float32)
```

Create a shifting array which stores the shifting status of each point, True means the point has not yet reached its mode and will continue to shift, False means it has reached its mode and will stop shifting, so it will be our convergence condition here, and while there is still pixel shifting, we will iterate through those pixels that have not yet stopped, and compute its(current pixel) distance(RGB value distance) to every other pixels, and filter the pixels that are within the bandwidth, we then can construct the uniform kernel, those within the bandwidth will each have equal amount of weight to the mean, finally, calculate the mean and the mean shift vector for the current pixel, update the current pixel to the new mean and check if the mean shift vector magnitude(that is the distance between the mean and the point) is smaller than the set MIN_DISTANCE, if so, the current pixel has reached its top mode and will not shift in the rest of the iterations:

```
153     shifting = np.array([True] * img_faltten.shape[0])
154     while np.sum(shifting)>0:
155         sum_distance = 0
156         for i in tqdm(range(0, len(img_faltten))):
157             if not shifting[i]:
158                 continue
159             centroid = img_faltten[i]
160 
161             # if not spatial:
162             distances = np.linalg.norm(img_faltten - centroid, axis=1)
163             filtered_pixels = img_faltten[distances <= bandwidth]
164             # distances_squared = np.sum((filtered_pixels - centroid)**2, axis=1)
165             distances = distances[distances <= bandwidth]
166             uniform_kernel = np.where(distances <= bandwidth, c, 0)
167             # n_kernel = np.exp(-distances_squared / (2 * bandwidth**2))
168 
169             mx = np.sum(filtered_pixels * uniform_kernel[:, np.newaxis], axis=0) / np.sum(uniform_kernel)
```

After all pixels have done shifting, their values are their mode, so pixels with the same mode value will be in the same attraction basin(aka cluster), we then just have to reshape the flatten image array back to 2D and then the segmentation is done:

```
184                 dist = np.linalg.norm(mx - centroid)
185                 sum_distance += dist
186                 img_faltten[i] = mx
187                 if dist < MIN_DISTANCE:
188                     shifting[i] = False
189         print(np.sum(shifting))
190 
191     img_faltten = img_faltten.astype(np.uint8)
192     return img_faltten.reshape((img.shape[0],img.shape[1], img.shape[-1]))
```

Step 2. Compute mean shift and draw the image:

Because the original size of the image is too large for mean shift to perform, we first downsample it to 1/16 of the original area(as the size gets larger, the computational cost increases exponentially, discussed in (F)), we can also cut the image into smaller blocks(pages), but it is doing the similar thing to downsmapling, we then call our mean_shift() function and set the RGB bandwidth to be 50, after segmented the image, we can draw its pixel distribution in RGB space and show its before and after segmentation (do the same for masterpiece image) :

```
326     image1_resized = cv2.resize(image1, (image1.shape[1]//4, image1.shape[0]//4), interpolation=cv2.INTER_AREA)
327     image1_segemented = mean_shift(image1_resized.copy(), 50, 1)
328     print(f"number of clusters: {np.unique(image1_segemented.reshape((-1,3)), axis=0).shape[0]}")
329     draw_and_save(image1_segemented, os.path.join(img_path,"output/2(c)2-image_msrgb_h50.jpg"))
330     draw_rgb(cv2.cvtColor(image1_resized, cv2.COLOR_BGR2RGB), cv2.cvtColor(image1_segemented, cv2.COLOR_BGR2RGB), os.pat
```

2. Results:

RGB segmentation with bandwidth set to 50:

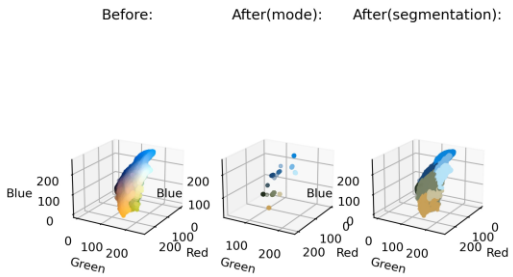
2(c)2-image_msrgb_h50.jpg

```
number of clusters: 129
```

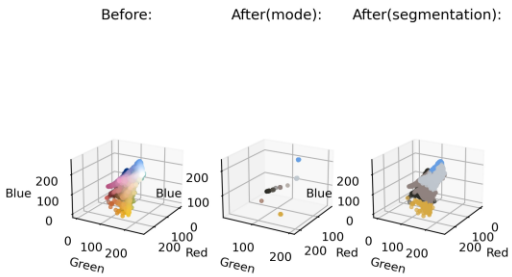
2(c)masterpiece_msrgb_h50.jpg

```
number of clusters: 87
```

pixel distributions in the R*G*B feature space before and after segmentation, (but p.36 is Luv space):



2(c)2-image_rgb_cube.jpg



2(c)masterpiece_rgb_cube.jpg

```
100%|
57586
100%|
56521
100%|
51164
100%|
33937
100%|
18046
100%|
6118
100%|
1352
100%|
1268
100%|
0
```

Number of pixels left shifting decreases.

(D) Mean-shift on spatial and RGB

1. Implementation:

Step 1. Modify mean shift function:

As we already explain the implementation of mean_shift() function, we just have to add pixel coordinates information to rgb array, becomes (m,n,5):

```
141         if spatial:
142             m, n, _ = img.shape
143             # Create x and y coordinate grids
144             x, y = np.meshgrid(np.arange(n), np.arange(m))
145             img_copy = np.zeros((m, n, 5), dtype=np.uint8)
146             img_copy[:, :, :3] = img
147             img_copy[:, :, 3] = x
148             img_copy[:, :, 4] = y
149             img = img_copy
```

And compute mean shift vector with these 5-dimensional values

```
153     shifting = np.array([True] * img_faltten.shape[0])
154     while np.sum(shifting)>0:
155         sum_distance = 0
156         for i in tqdm(range(0, len(img_faltten))):
157             if not shifting[i]:
158                 continue
159             centroid = img_faltten[i]
160 
161             # if not spatial:
162             distances = np.linalg.norm(img_faltten - centroid, axis=1)
163             filtered_pixels = img_faltten[distances <= bandwidth]
164             # distances_squared = np.sum((filtered_pixels - centroid)**2, axis=1)
165             distances = distances[distances <= bandwidth]
166             uniform_kernel = np.where(distances <= bandwidth, c, 0)
167             # n_kernel = np.exp(-distances_squared / (2 * bandwidth**2))
168 
169             mx = np.sum(filtered_pixels * uniform_kernel[:, np.newaxis], axis=0) / np.sum(uniform_kernel)
```

Another approach:

we can separate color and spatial information and each has its own bandwidth parameter, the results are slightly different from above.

```
161             if not spatial:
162                 distances = np.linalg.norm(img_faltten - centroid, axis=1)
163                 filtered_pixels = img_faltten[distances <= bandwidth]
164                 # distances_squared = np.sum((filtered_pixels - centroid)**2, axis=1)
165                 distances = distances[distances <= bandwidth]
166                 uniform_kernel = np.where(distances <= bandwidth, c, 0)
167                 # n_kernel = np.exp(-distances_squared / (2 * bandwidth**2))
168 
169                 mx = np.sum(filtered_pixels * uniform_kernel[:, np.newaxis], axis=0) / np.sum(uniform_kernel)
170             else:
171                 distances = np.linalg.norm(img_faltten - centroid, axis=1)
172                 distances_s = np.linalg.norm(img_faltten[:, 3:] - centroid[3:], axis=1)
173                 distances_r = np.linalg.norm(img_faltten[:, :3] - centroid[:3], axis=1)
174                 filtered_pixels = img_faltten[(distances_r <= bandwidth) & (distances_s <= bandwidth2)]
175                 distances = distances[(distances_r <= bandwidth) & (distances_s <= bandwidth2)]
176                 uniform_kernel = np.where(distances, c, 0)
177                 mx = np.sum(filtered_pixels * uniform_kernel[:, np.newaxis], axis=0) / np.sum(uniform_kernel)
178 
179             dist = np.linalg.norm(mx - centroid)
```

Step 2. Draw the images for both images:

We set bandwidth to be 50 for both images and draw:

```
344     # mean shift rgb+xy on image1
345     image1_segemented = mean_shift(image1_resized.copy(), 50, 1, True)
346     print(f"number of clusters: {np.unique(image1_segemented.reshape((-1,5)), axis=0).shape[0]}")
347     draw_and_save(image1_segemented[:,:,:3], os.path.join(img_path,"output/2(d)2-image_msxy.jpg"))
```

*The returned array is (m,n,5), we have to truncate down the last 2 cols to make it (m,n,3) RGB pixel value, and draw the segmentation result.

2. Results:

2(d)2-image_msxy.jpg

2(d)masterpiece_msxy.jpg

(This is using 2nd approach(2 bandwidths))

We can see the results don't seem to be that much different from only considering RGB space overall, however, we notice after adding spatial information, the right portion of the segmented 2-image seems to have a darker toned vertical band, this could be due to that a lot more dark colors are located on the right side.

(E) 2(C) with different bandwidth

1. Implementation:

Try three different RGB bandwidths 5, 25, 75 (50 in C) for both images:

```
363     print("=======2(e)=======")
364     # mean shift rgb on image1 with different bandwidth
365     image1_segemented = mean_shift(image1_resized.copy(), 5, 1)
366     print(f"number of clusters: {np.unique(image1_segemented.reshape((-1,3)), axis=0).shape[0]}")
367     draw_and_save(image1_segemented, os.path.join(img_path,"output/2(e)2-image_msrgb_h5.jpg"))
368
369     image1_segemented = mean_shift(image1_resized.copy(), 25, 1)
370     print(f"number of clusters: {np.unique(image1_segemented.reshape((-1,3)), axis=0).shape[0]}")
371     draw_and_save(image1_segemented, os.path.join(img_path,"output/2(e)2-image_msrgb_h25.jpg"))
372
373     image1_segemented = mean_shift(image1_resized.copy(), 75, 1)
374     print(f"number of clusters: {np.unique(image1_segemented.reshape((-1,3)), axis=0).shape[0]}")
375     draw_and_save(image1_segemented, os.path.join(img_path,"output/2(e)2-image_msrgb_h75.jpg"))
376
377     image2_segemented = mean_shift(image2_resized.copy(), 5, 1)
378     print(f"number of clusters: {np.unique(image2_segemented.reshape((-1,3)), axis=0).shape[0]}")
379     draw_and_save(image2_segemented, os.path.join(img_path,"output/2(e)masterpiece_msrgb_h5.jpg"))
380
381     image2_segemented = mean_shift(image2_resized.copy(), 25, 1)
382     print(f"number of clusters: {np.unique(image2_segemented.reshape((-1,3)), axis=0).shape[0]}")
383     draw_and_save(image2_segemented, os.path.join(img_path,"output/2(e)masterpiece_msrgb_h25.jpg"))
384
385     image2_segemented = mean_shift(image2_resized.copy(), 75, 1)
386     print(f"number of clusters: {np.unique(image2_segemented.reshape((-1,3)), axis=0).shape[0]}")
387     draw_and_save(image2_segemented, os.path.join(img_path,"output/2(e)masterpiece_msrgb_h75.jpg"))
```

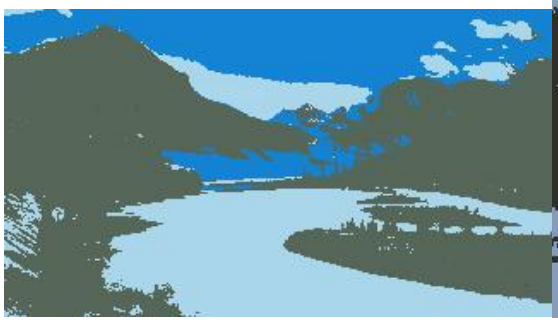
2. Results:

2(e)2-image_msrgb_h5.jpg

2(e)masterpiece_msrgb_h5.jpg

2(e)2-image_msrgb_h25.jpg

2(e)masterpiece_msrgb_h25.jpg



2(e)2-image_msrgb_h75.jpg

2(e)masterpiece_msrgb_h75.jpg

3. Discussion:

We can see, as we increase the RGB bandwidth, the color segmentation becomes more obvious, that is less cluster, if we set the bandwidth low enough, it will look very similar to the original image before segmentation, this is because when bandwidth is low, it is more likely that each pixel stops shifting very soon as pixels within the bandwidth are very close in values, leading to only small changes in mean, hence small mean shift vector, if we set the bandwidth too high, it will look something like in the third result image, less number of cluster, as more pixels will go to the same mode.

(F) K-means v.s. mean-shift

Discussion:

We can see in 2(A) and 2(E) result images segmented using k-means and mean-shift respectively, as we increase the parameters of k-means, that is K, the number of color clusters increases, lead to finer(smoother color transition) segmentation results, as for mean-shift, when we increase the parameter bandwidth, the segmentation becomes less fine as we discussed in 2(E), so the number of cluster is specified for k-means, however, for mean-shift, it is decided by the bandwidth.

And for k-means, initial guesses could affect the final result, so if you rerun the code, the results maybe slightly different from the images shown above, however, for mean-shift, the results should be consistent as there is no guesses in the initialization.

As for computational cost, mean-shift is significantly more costly in terms of time complexity, it requires O(n^2), whereas it is only O(K*n) for k-means, n is the number of pixels, we record the time required for mean-shift to segment our images in our code, it takes a lot longer even for a donwsampled image. We also discovered that as we increase bandwidth, it slightly increases the computation time as more pixels shift longer than small bandwidth.