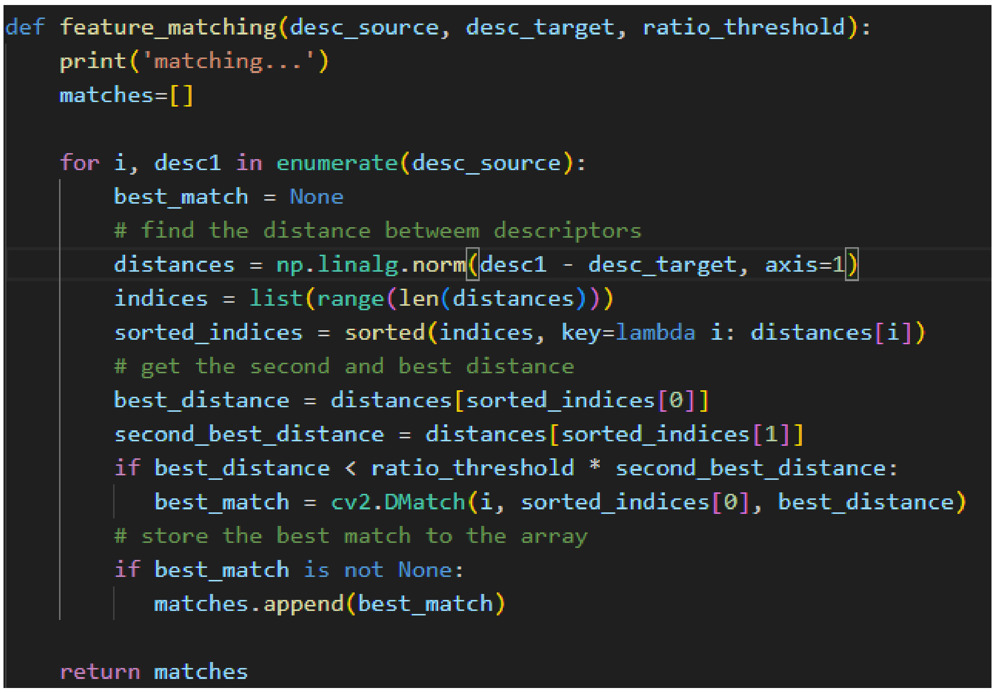
Problem 1. (40%) Image Alignment with RANSA

Step of A

- Code

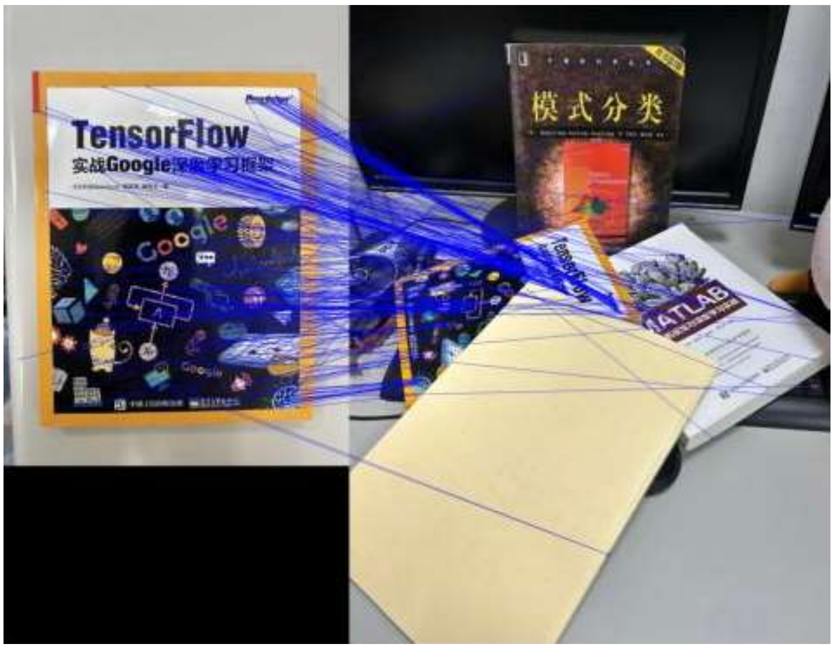


```python
def feature_matching(desc_source, desc_target, ratio_threshold):
    print('matching...')
    matches=[]

    for i, desc1 in enumerate(desc_source):
        best_match = None
        # find the distance betweem descriptors
        distances = np.linalg.norm(desc1 - desc_target, axis=1)
        indices = list(range(len(distances)))
        sorted_indices = sorted(indices, key=lambda i: distances[i])
        # get the second and best distance
        best_distance = distances[sorted_indices[0]]
        second_best_distance = distances[sorted_indices[1]]
        if best_distance < ratio_threshold * second_best_distance:
            best_match = cv2.DMatch(i, sorted_indices[0], best_distance)
        # store the best match to the array
        if best_match is not None:
            matches.append(best_match)

    return matches
```

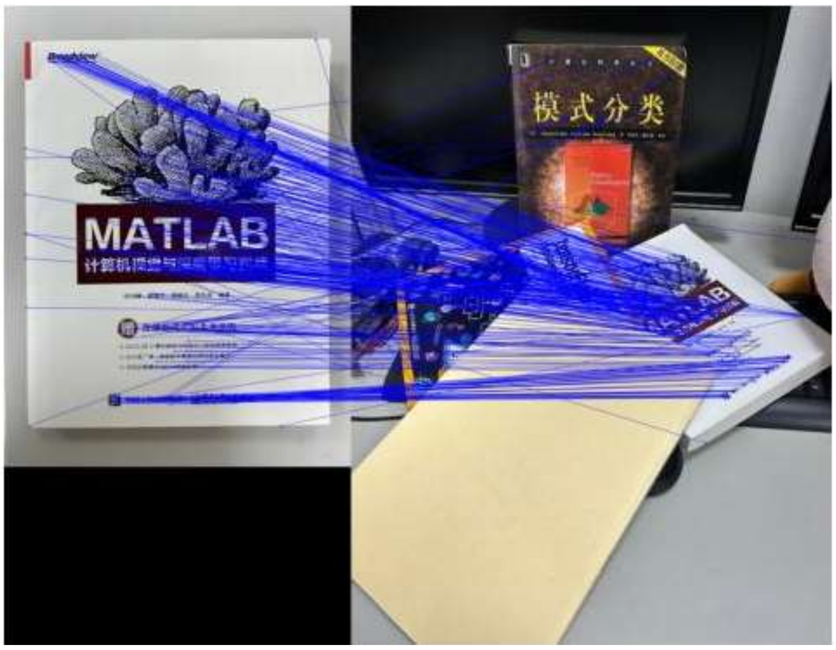
- Explanation

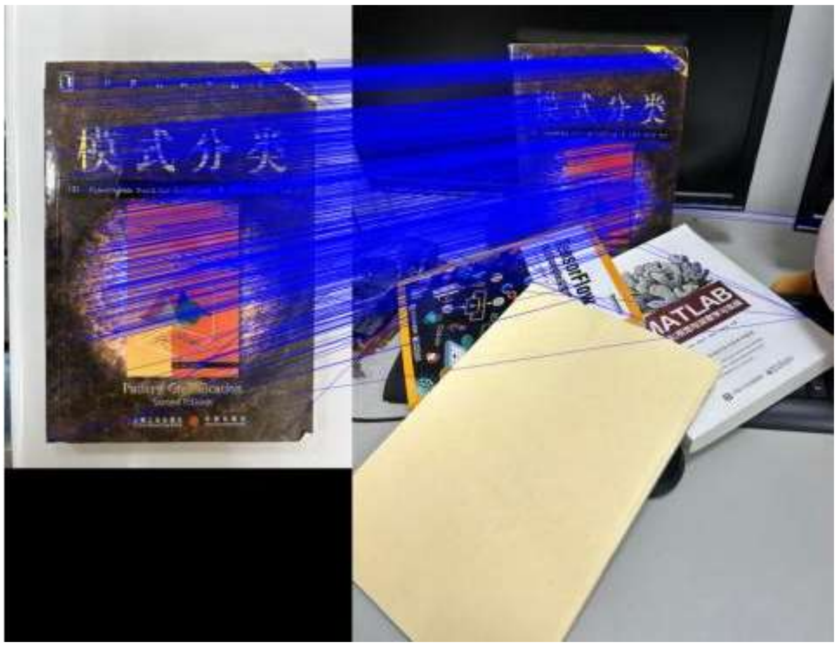
這個部分是沿用作業一的程式。利用 for loop 針對每個 source descriptor 和所有 target descriptor 計算 distance 並找出最短的兩個距離，若最短的距離又小於 threshold *第二短距離，視為 good match。最後回傳所有找到的 good matches，再利用 drawMatchesKnn 把 matching 畫到圖上。

- Results



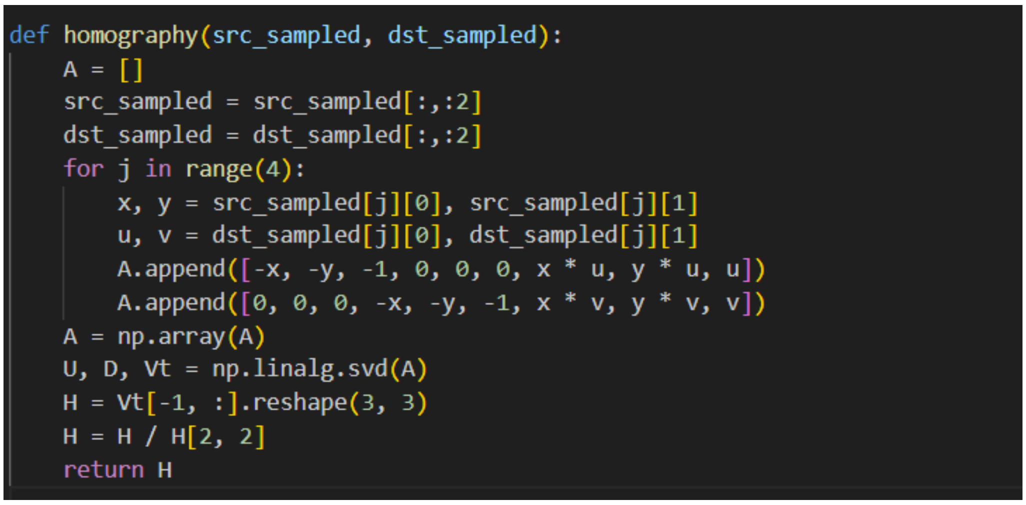
- Results


MATLAB
计算机视觉与深度学习实战
模式分类
MATLAB

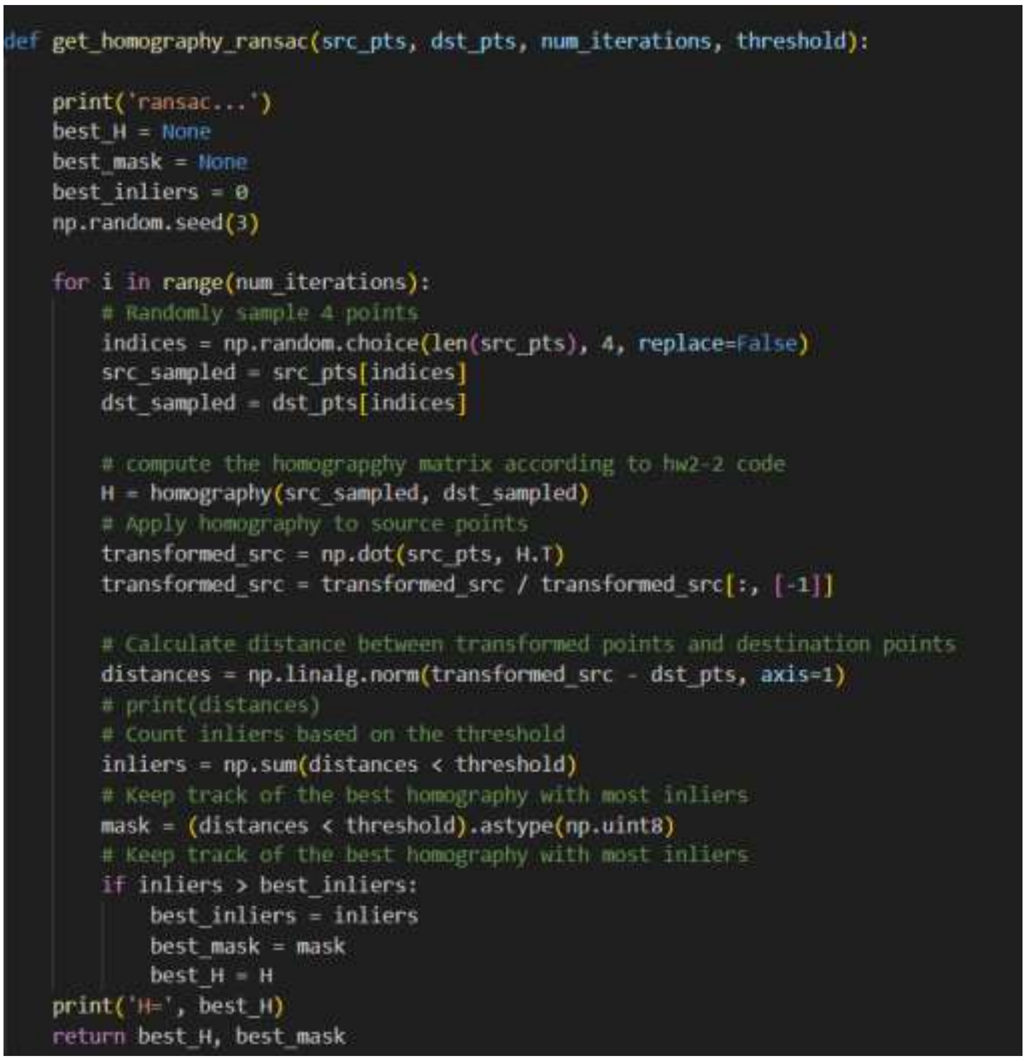

模式分类
Pattern Classification
Second Edition
模式分类
MATLAB

Step of B (RANSAC)

- Code



```
def homography(src_sampled, dst_sampled):
    A = []
    src_sampled = src_sampled[:,:2]
    dst_sampled = dst_sampled[:,:2]
    for j in range(4):
        x, y = src_sampled[j][0], src_sampled[j][1]
        u, v = dst_sampled[j][0], dst_sampled[j][1]
        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)
    U, D, Vt = np.linalg.svd(A)
    H = Vt[-1, :].reshape(3, 3)
    H = H / H[2, 2]
    return H
```



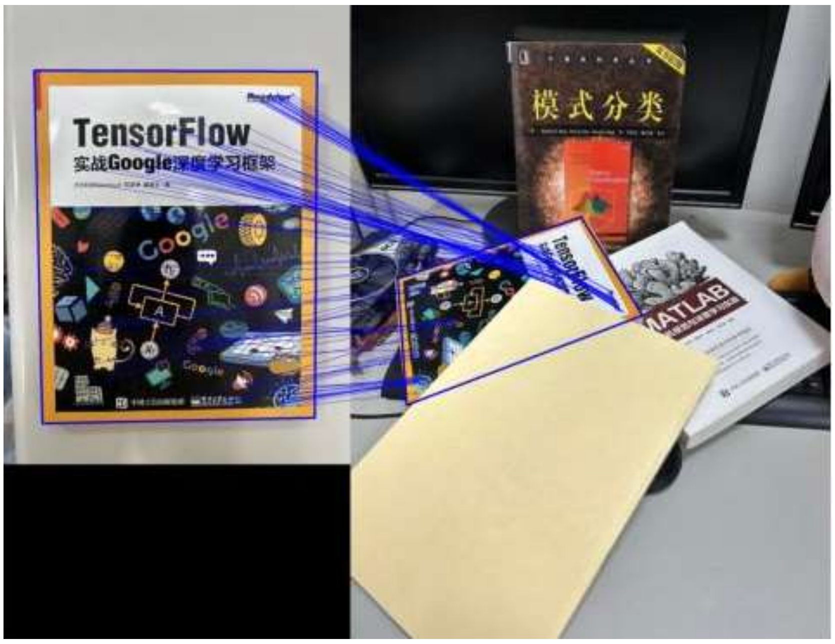
```
def get_homography_ransac(src_pts, dst_pts, num_iterations, threshold):

    print('ransac...')
    best_H = None
    best_mask = None
    best_inliers = 0
    np.random.seed(3)

    for i in range(num_iterations):
        # Randomly sample 4 points
        indices = np.random.choice(len(src_pts), 4, replace=False)
        src_sampled = src_pts[indices]
        dst_sampled = dst_pts[indices]

        # compute the homograpghy matrix according to hw2-2 code
        H = homography(src_sampled, dst_sampled)
        # Apply homography to source points
        transformed_src = np.dot(src_pts, H.T)
        transformed_src = transformed_src / transformed_src[:, [-1]]

        # Calculate distance between transformed points and destination points
        distances = np.linalg.norm(transformed_src - dst_pts, axis=1)
        # print(distances)
        # Count inliers based on the threshold
        inliers = np.sum(distances < threshold)
        # Keep track of the best homography with most inliers
        mask = (distances < threshold).astype(np.uint8)
        # Keep track of the best homography with most inliers
        if inliers > best_inliers:
            best_inliers = inliers
            best_mask = mask
            best_H = H
    print('H=', best_H)
    return best_H, best_mask
```

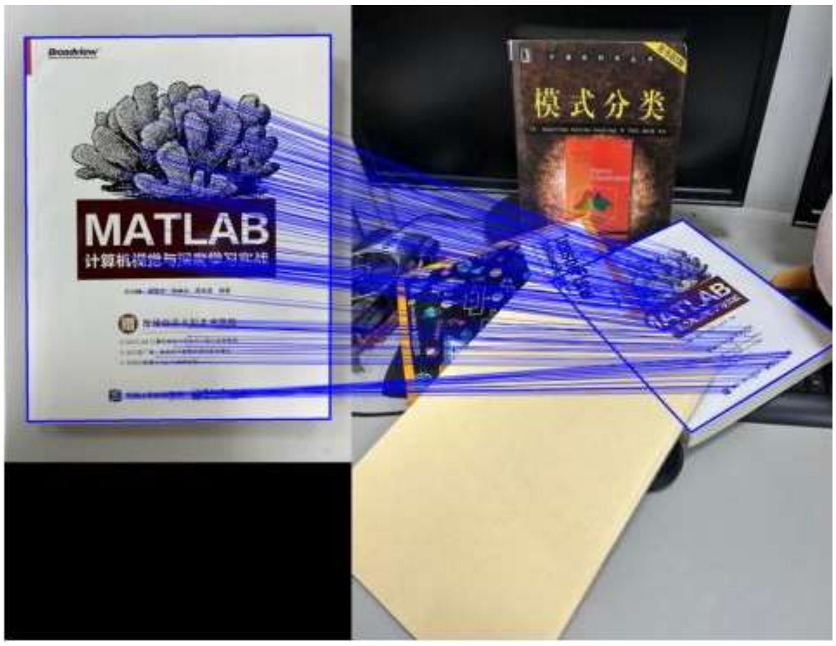
- Explanation

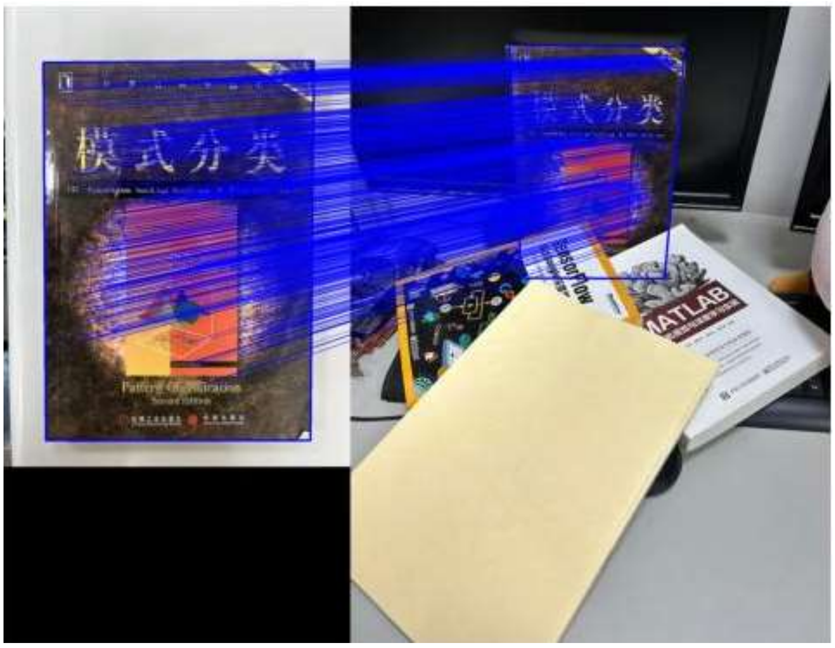
從 src_pts 、dst_ptS）中隨機選擇四對對應點利用 homography function (code 第一張圖)去計算 homography matrix H。利用 H 對所有的 scr_pts，並計算轉換後的點 transfomed_scr。通過計算 transfomed_scr 和 dst_pts 之間的 distance，distance < threshold，則視為 inliers。利用 for loop 重複上述 process 持續計算出新的 H，若 inliers 較多就持續更新為 best _H 以及

對應的 best_mask，後續用來計算來選取 inlier 的 matching。

- Results

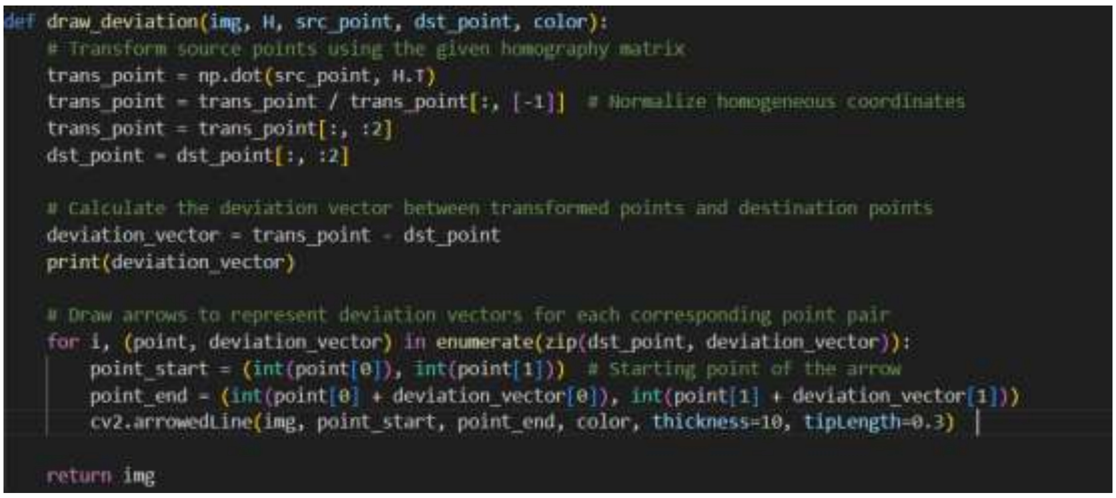






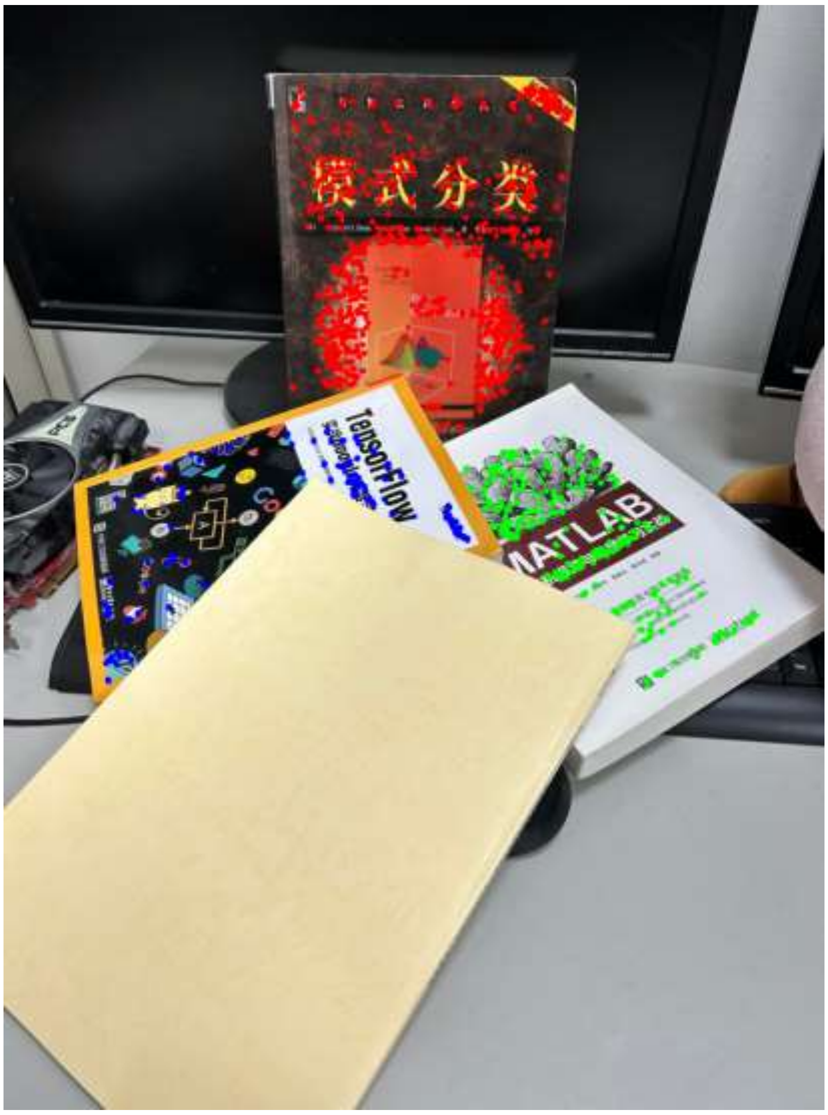
Step of B (Deviation)

- Code



```python
def draw_deviation(img, H, src_point, dst_point, color):
    # Transform source points using the given homography matrix
    trans_point = np.dot(src_point, H.T)
    trans_point = trans_point / trans_point[:, [-1]]  # Normalize homogeneous coordinates
    trans_point = trans_point[:, :2]
    dst_point = dst_point[:, :2]

    # Calculate the deviation vector between transformed points and destination points
    deviation_vector = trans_point - dst_point
    print(deviation_vector)

    # Draw arrows to represent deviation vectors for each corresponding point pair
    for i, (point, deviation_vector) in enumerate(zip(dst_point, deviation_vector)):
        point_start = (int(point[0]), int(point[1]))  # Starting point of the arrow
        point_end = (int(point[0] + deviation_vector[0]), int(point[1] + deviation_vector[1]))
        cv2.arrowedLine(img, point_start, point_end, color, thickness=10, tipLength=0.3)

    return img
```

- Explanation
  利用 H 去計算出 transformed source point 和 destination 之間的 deviation vector，利用 cv2.arrawedLine 畫出結果。
- Results



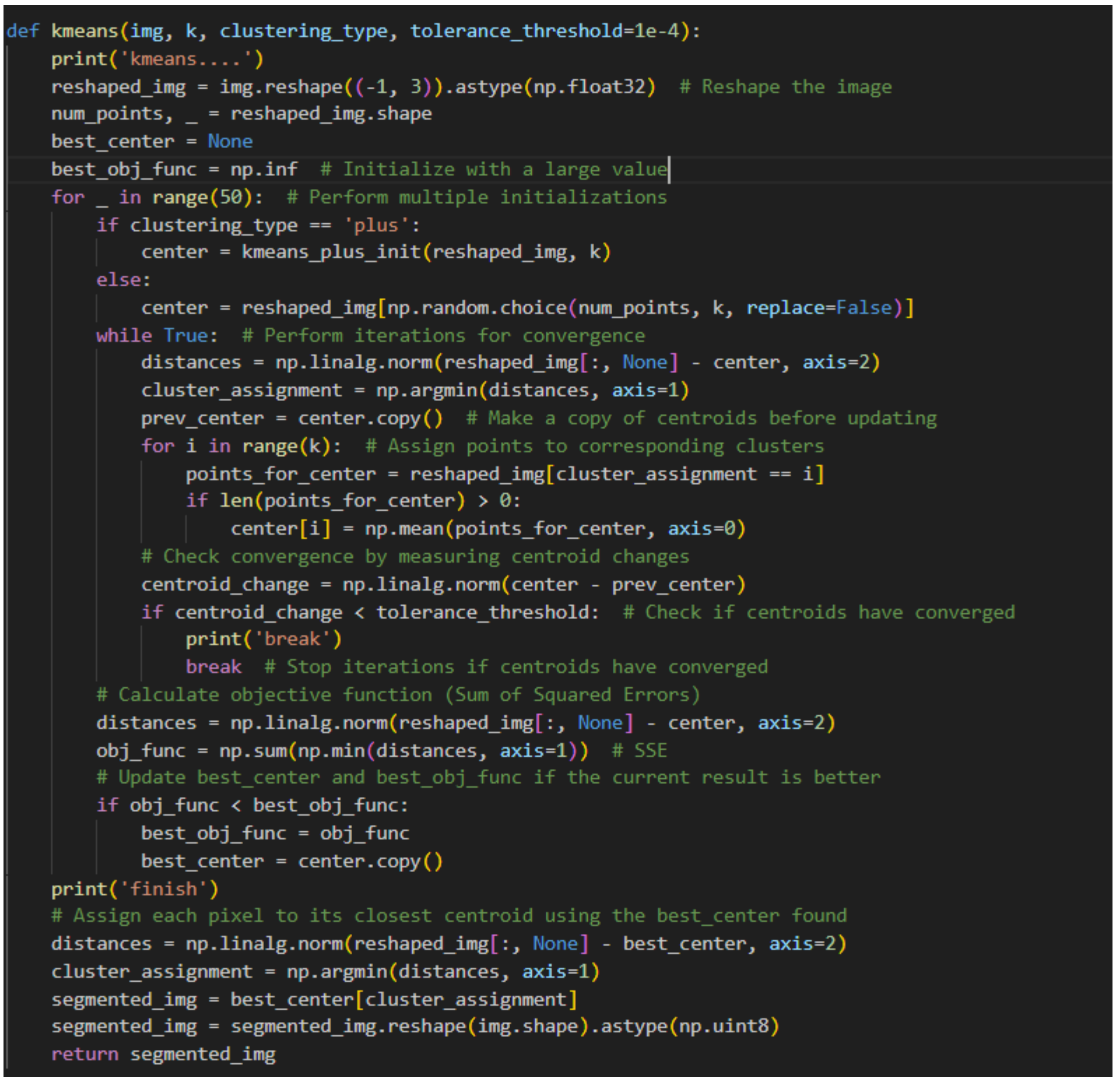
- Discussion--Compare the parameter settings in SIFT feature and RANSAC and discuss the result

如果 threshold 設定得太小，可能會將一些 good matching point 誤判為外 outliers，導致最終估計的 homography matrix H 不夠精確。相反，如果 threshold 設定得太大，則可能會將太多的雜訊或異常值納入 inlier，導致估計的 H 一樣會不 robust。這兩個計算出來的 matrix 對 scr_pts 做 transform 和 dst_pts 做 distance 的比較容易誤差很大。所以需要選取適當的 threshold，來保證 homograpy matrix 的正確性。

Problem 2. (60%) Image segmentation:

Step of A -- Kmeans

- Code



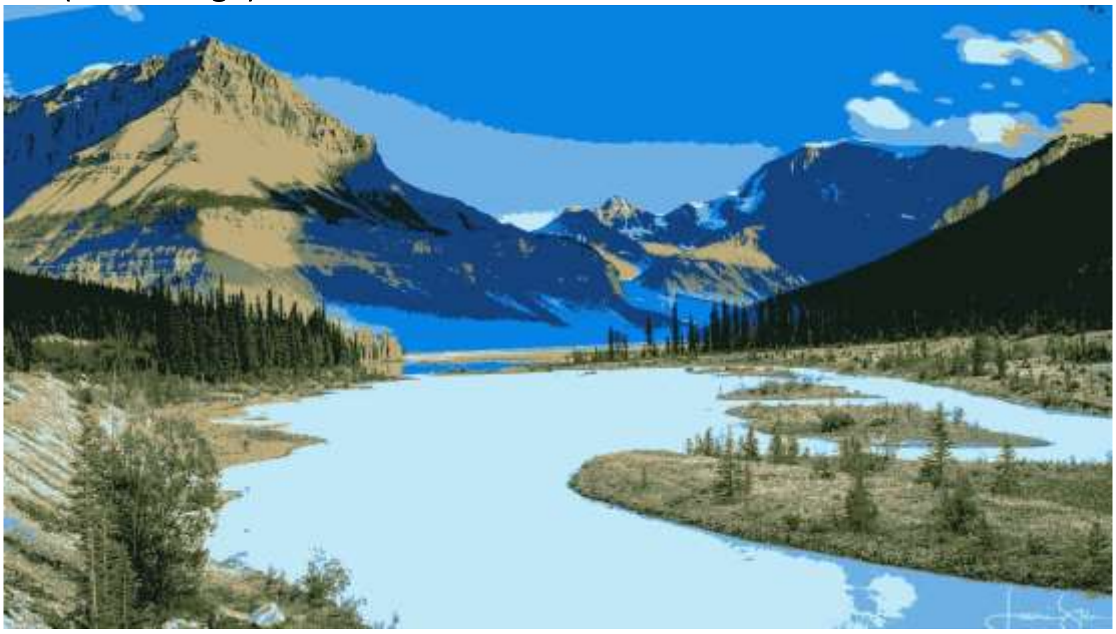
```python
def kmeans(img, k, clustering_type, tolerance_threshold=1e-4):
    print('kmeans....')
    reshaped_img = img.reshape((-1, 3)).astype(np.float32)  # Reshape the image
    num_points, _ = reshaped_img.shape
    best_center = None
    best_obj_func = np.inf  # Initialize with a large value
    for _ in range(50):  # Perform multiple initializations
        if clustering_type == 'plus':
            center = kmeans_plus_init(reshaped_img, k)
        else:
            center = reshaped_img[np.random.choice(num_points, k, replace=False)]
        while True:  # Perform iterations for convergence
            distances = np.linalg.norm(reshaped_img[:, None] - center, axis=2)
            cluster_assignment = np.argmin(distances, axis=1)
            prev_center = center.copy()  # Make a copy of centroids before updating
            for i in range(k):  # Assign points to corresponding clusters
                points_for_center = reshaped_img[cluster_assignment == i]
                if len(points_for_center) > 0:
                    center[i] = np.mean(points_for_center, axis=0)
            # Check convergence by measuring centroid changes
            centroid_change = np.linalg.norm(center - prev_center)
            if centroid_change < tolerance_threshold:  # Check if centroids have converged
                print('break')
                break  # Stop iterations if centroids have converged
        # Calculate objective function (Sum of Squared Errors)
        distances = np.linalg.norm(reshaped_img[:, None] - center, axis=2)
        obj_func = np.sum(np.min(distances, axis=1))  # SSE
        # Update best_center and best_obj_func if the current result is better
        if obj_func < best_obj_func:
            best_obj_func = obj_func
            best_center = center.copy()
    print('finish')
    # Assign each pixel to its closest centroid using the best_center found
    distances = np.linalg.norm(reshaped_img[:, None] - best_center, axis=2)
    cluster_assignment = np.argmin(distances, axis=1)
    segmented_img = best_center[cluster_assignment]
    segmented_img = segmented_img.reshape(img.shape).astype(np.uint8)
    return segmented_img
```

- Explanation

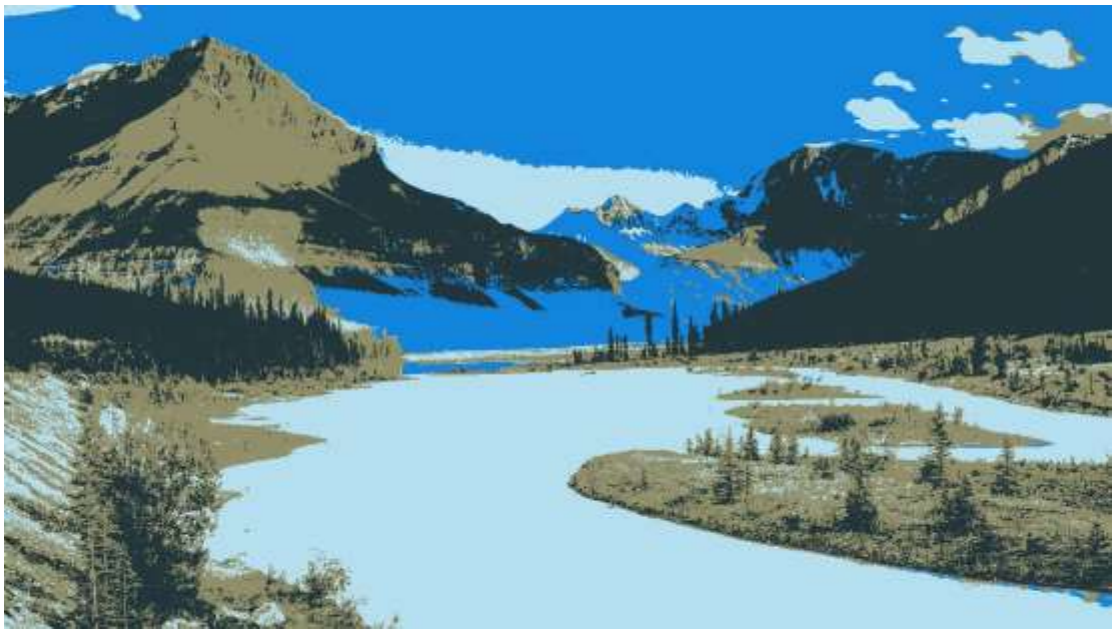
總共會對隨機選擇不同的初始 cluster center 做 50 次。每一次 iteration 是透過計算每個 pixel data 到 cluster center 的距離，將每個 pixel 分配給 distance 最小的 cluster center。新的 cluster center 由 cluster 中所有 pixel data 的 mean 做更新。持續新的 cluster center 直至 cluster center 達到設定的收斂 threshold。在每次 iteration 後去利用 sum of square errors 計算 obj_func，並選擇 obj_func 值最小的 clustering result 作為最終分割。

- Results

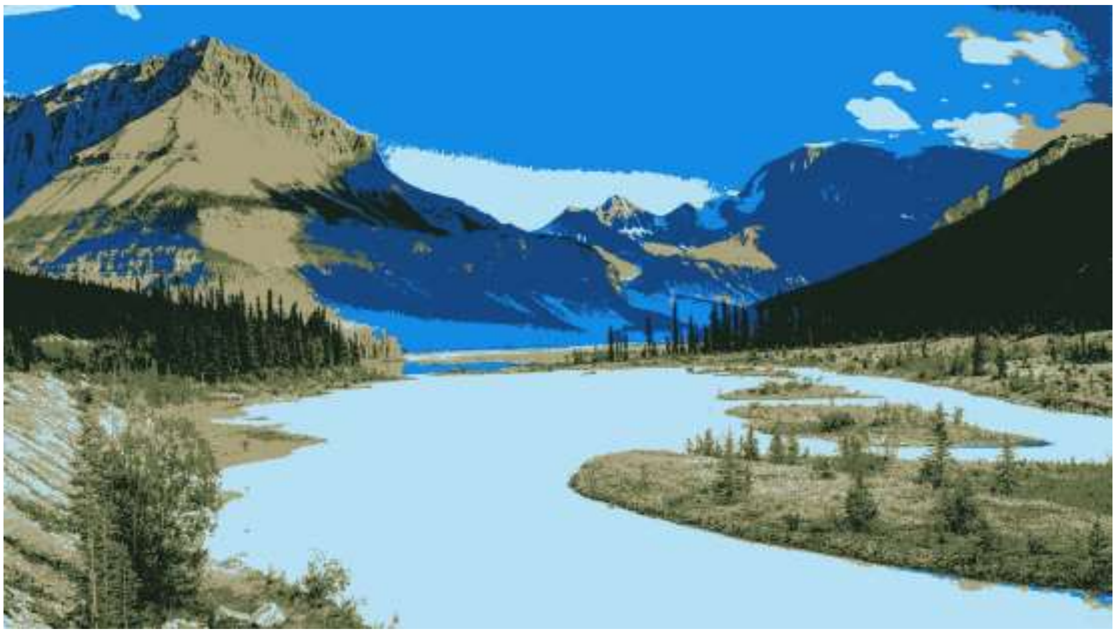
K=4(for 2-image)



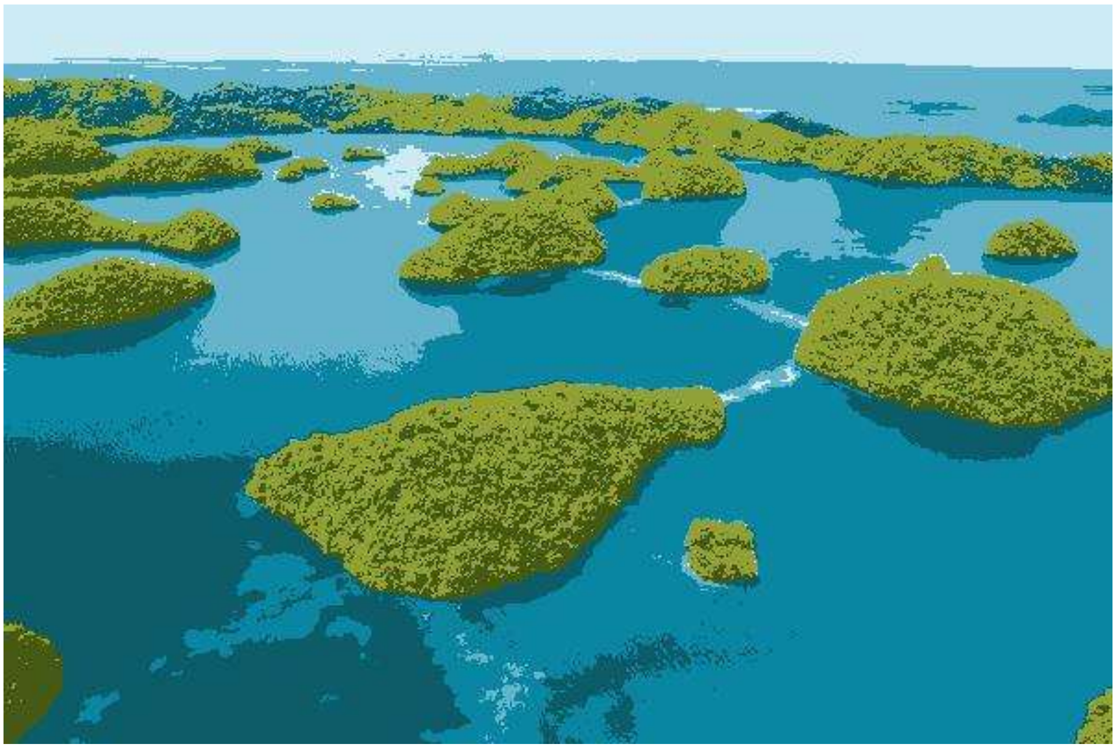
K=6



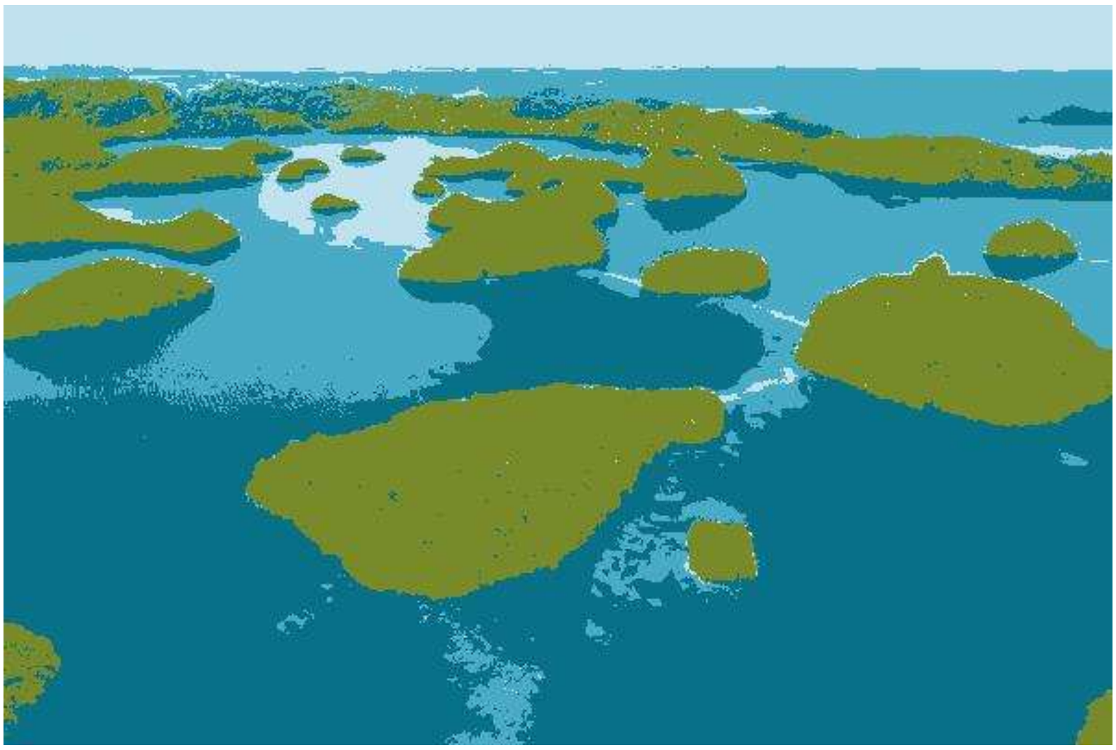
K=8



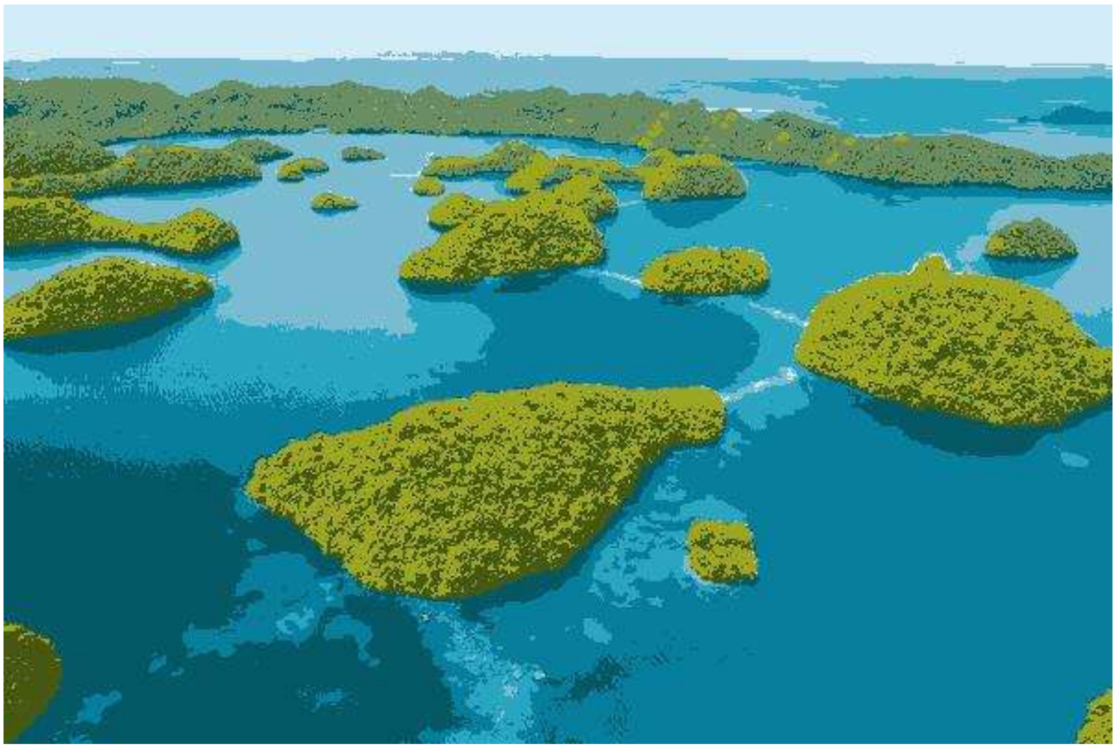
K=4(for 2-masterpiece)



K=6



K=8

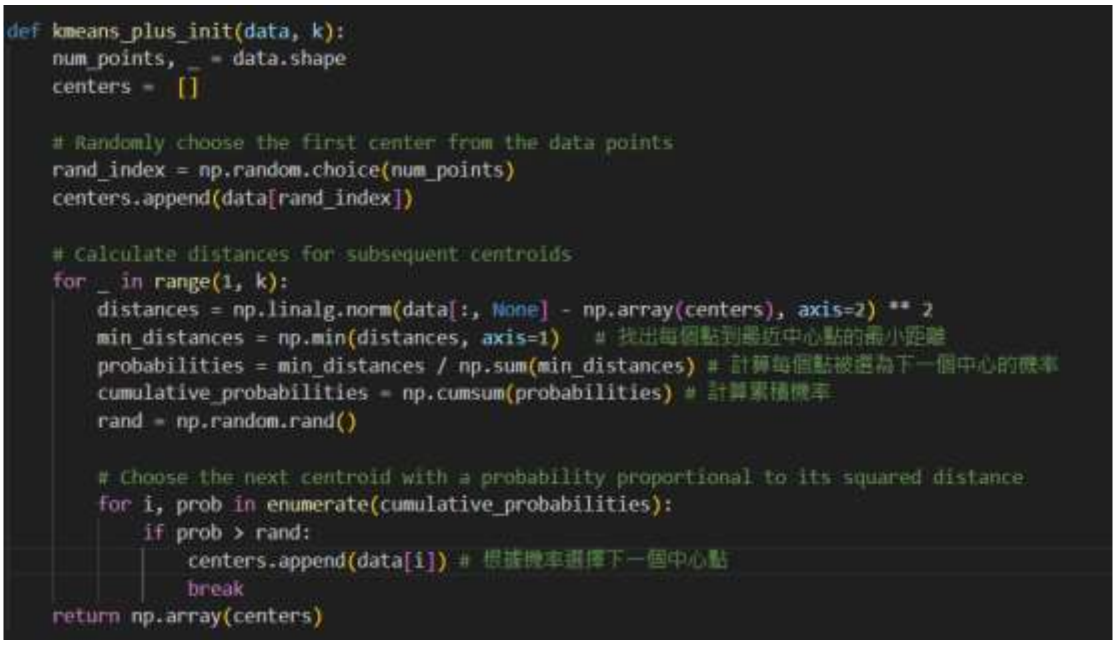


- Discussion

  中心點數量越小可能導致模型複雜度不足以有效捕捉數據特徵，而數量越大則可能增加計算成本且容易產生過度分類的情況。

Step of B– Kmeans++

- Code



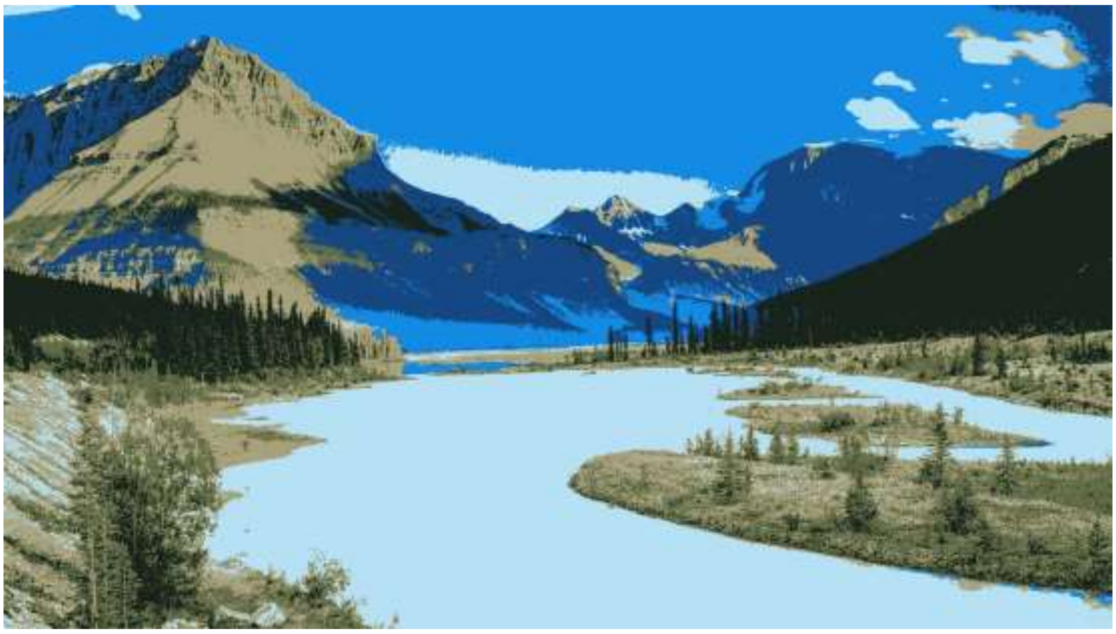
```python
def kmeans_plus_init(data, k):
    num_points, _ = data.shape
    centers = []

    # Randomly choose the first center from the data points
    rand_index = np.random.choice(num_points)
    centers.append(data[rand_index])

    # Calculate distances for subsequent centroids
    for _ in range(1, k):
        distances = np.linalg.norm(data[:, None] - np.array(centers), axis=2) ** 2
        min_distances = np.min(distances, axis=1)   # 找出每個點到最近中心點的最小距離
        probabilities = min_distances / np.sum(min_distances) # 計算每個點被選為下一個中心的機率
        cumulative_probabilities = np.cumsum(probabilities) # 計算累積機率
        rand = np.random.rand()

        # Choose the next centroid with a probability proportional to its squared distance
        for i, prob in enumerate(cumulative_probabilities):
            if prob > rand:
                centers.append(data[i]) # 根據機率選擇下一個中心點
                break
    return np.array(centers)
```

- Explanation

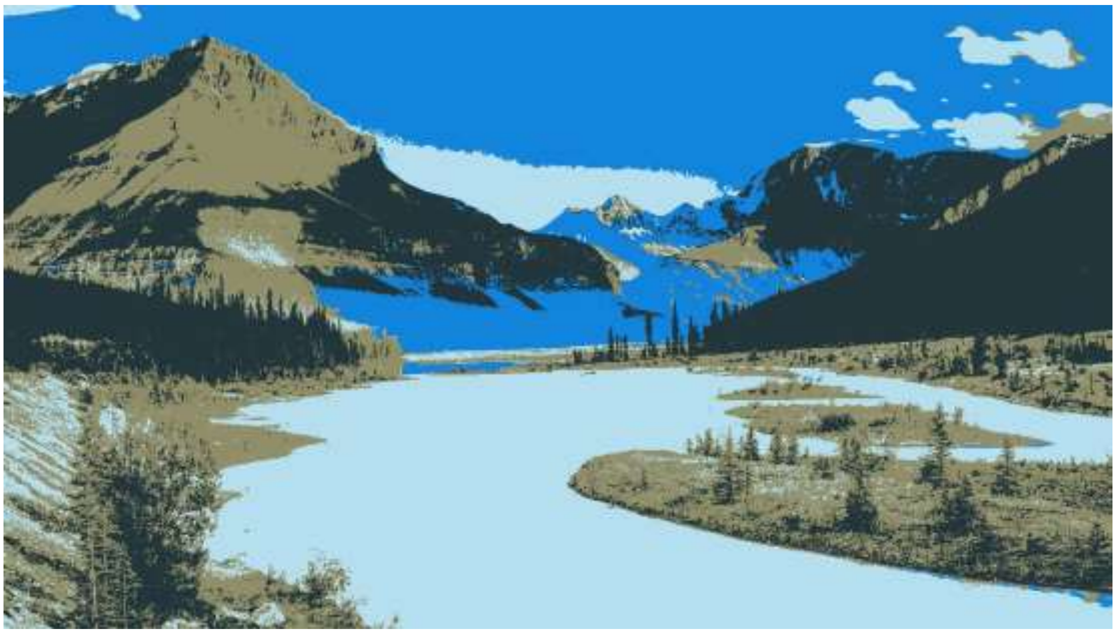
先隨機選擇一個資料點作為第一個 cluster center，然後依據 data point 到最近 cluster center 的 distance 平方，以機率的方式選擇下一個 cluster center，直到選擇出所需數量的 cluster center。再把 cluster center 餵入 keams 中用 A 小題的方法計算最後 clustering 的 result。

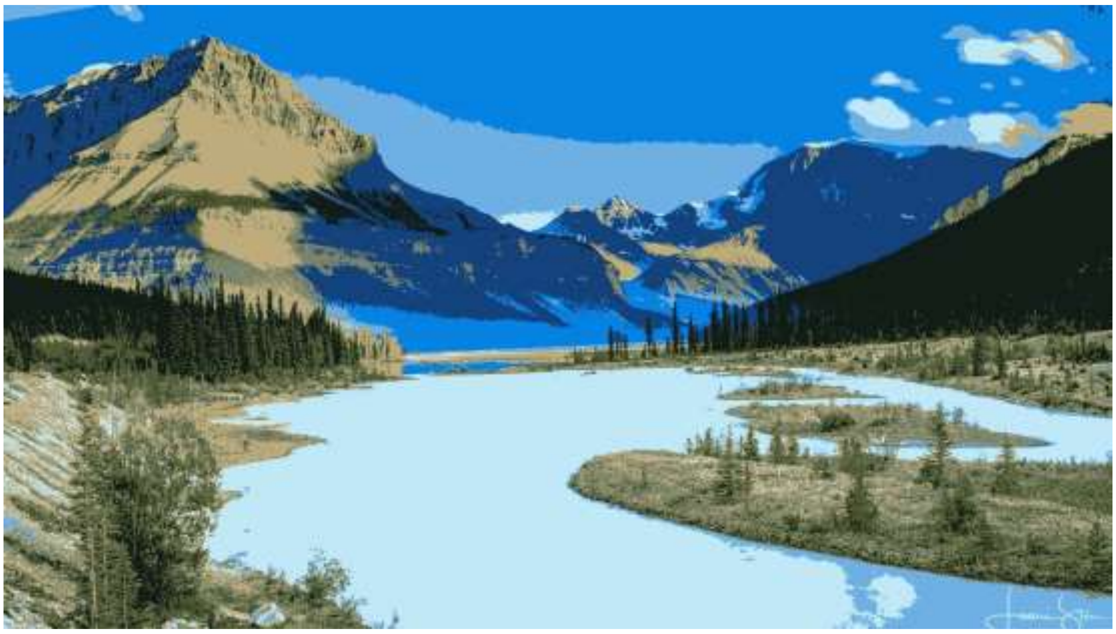
- Results

K=4(for 2-image)

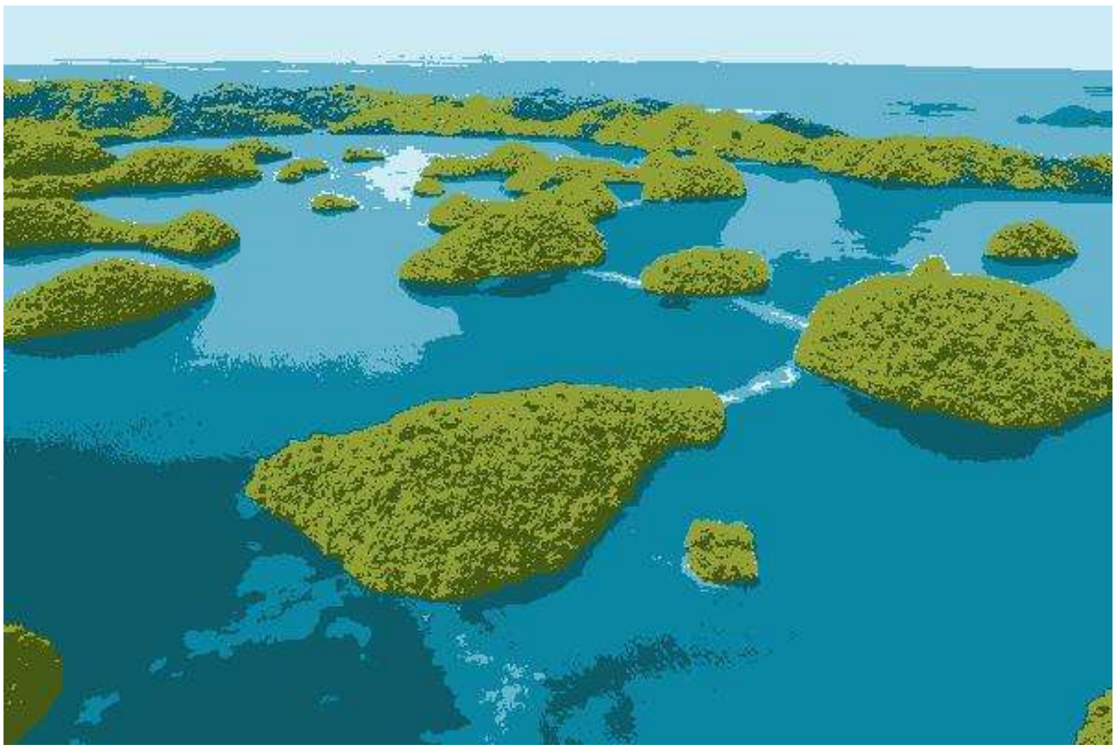


K=6

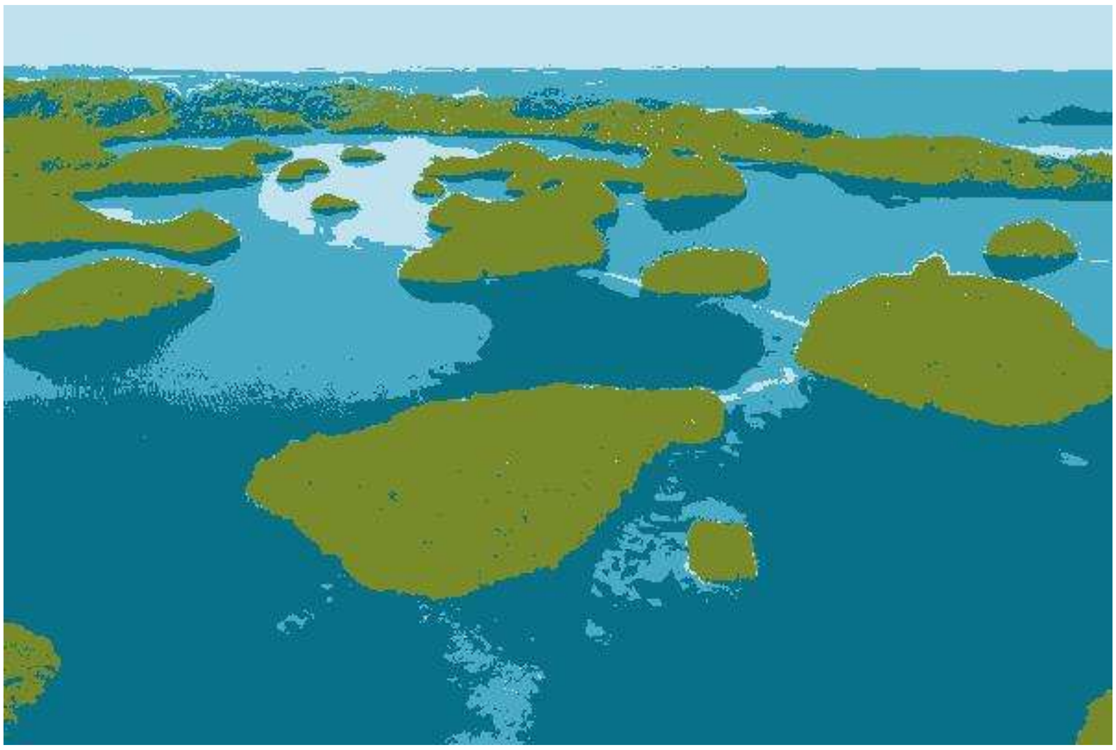

K=8



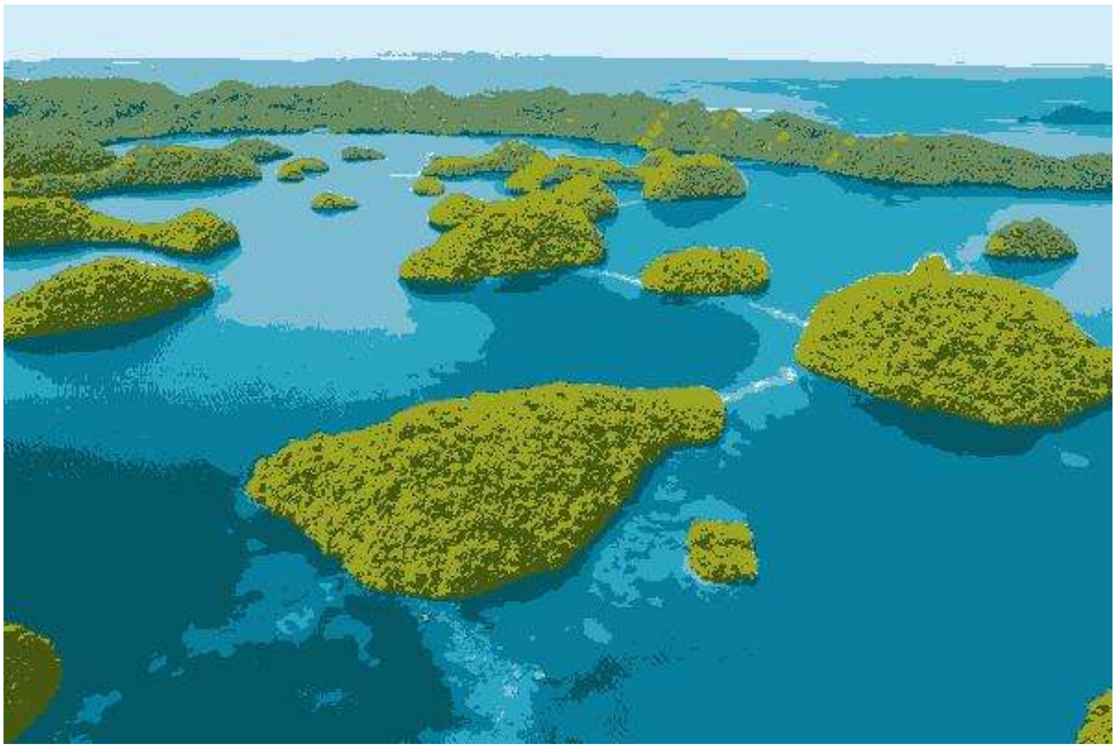
K=4(for 2-masterpiece)



K=6



K=8

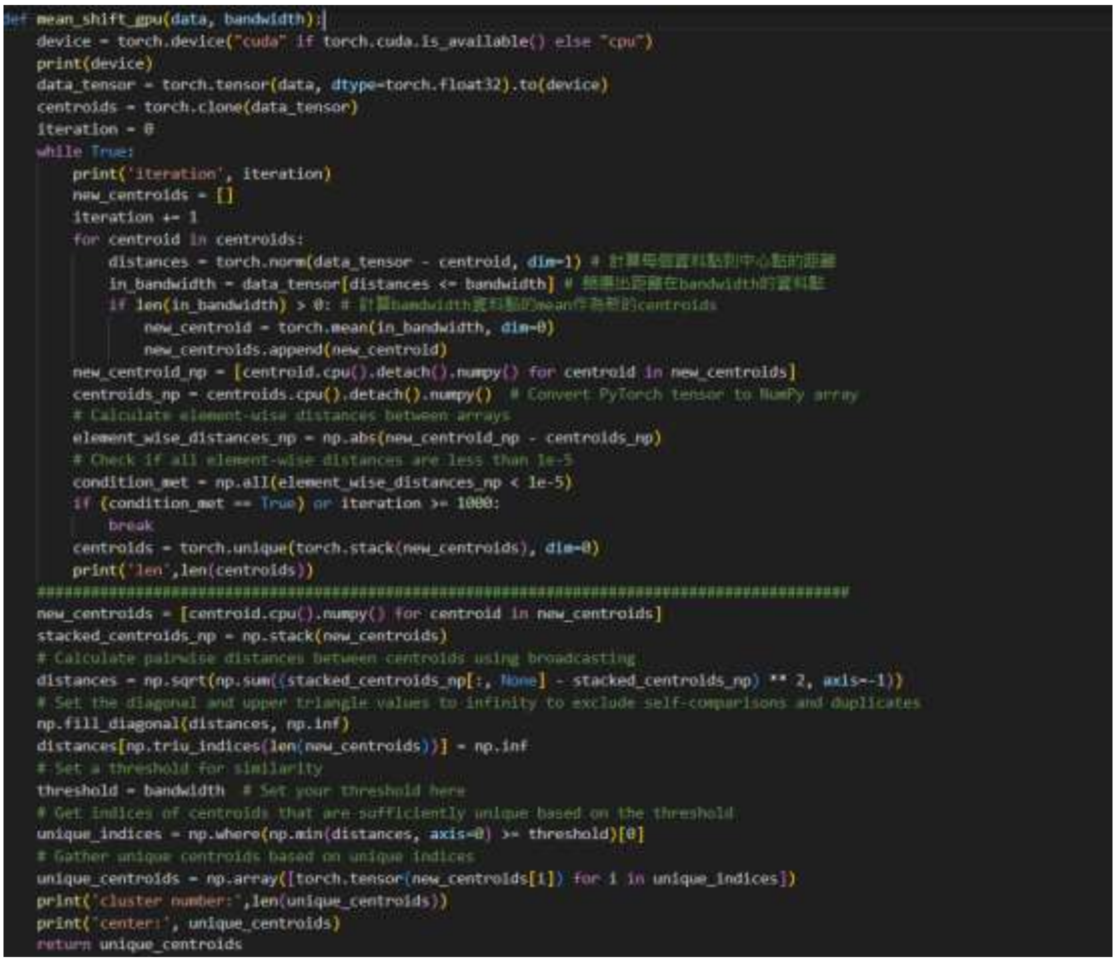


- Discussion

  K-means++相較於 K-means 能更有效地選擇初始聚類中心，因此在收斂速度和最終結果的 cluster 上較優於傳統 K-means。

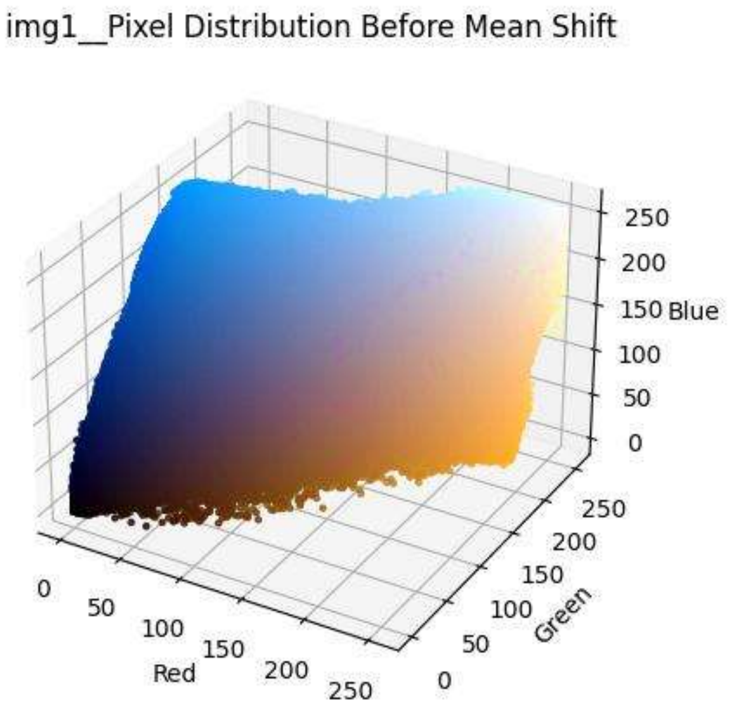
Step of C – mean shift

- Code

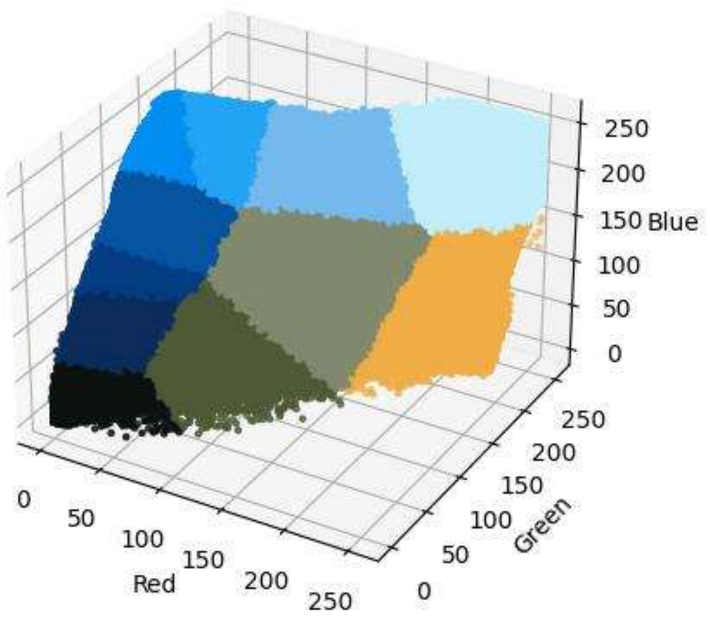


```python
def mean_shift_gpu(data, bandwidth):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(device)
    data_tensor = torch.tensor(data, dtype=torch.float32).to(device)
    centroids = torch.clone(data_tensor)
    iteration = 0
    while True:
        print('iteration', iteration)
        new_centroids = []
        iteration += 1
        for centroid in centroids:
            distances = torch.norm(data_tensor - centroid, dim=1) # 計算每個資料點到中心點的距離
            in_bandwidth = data_tensor[distances <= bandwidth] # 篩選出距離在bandwidth的資料點
            if len(in_bandwidth) > 0: # 計算bandwidth資料點的mean作為新的centroids
                new_centroid = torch.mean(in_bandwidth, dim=0)
                new_centroids.append(new_centroid)
        new_centroid_np = [centroid.cpu().detach().numpy() for centroid in new_centroids]
        centroids_np = centroids.cpu().detach().numpy()  # Convert PyTorch tensor to NumPy array
        # Calculate element-wise distances between arrays
        element_wise_distances_np = np.abs(new_centroid_np - centroids_np)
        # Check if all element-wise distances are less than 1e-5
        condition_met = np.all(element_wise_distances_np < 1e-5)
        if (condition_met == True) or iteration >= 1000:
            break
        centroids = torch.unique(torch.stack(new_centroids), dim=0)
        print('len',len(centroids))
    ###############################################################################
    new_centroids = [centroid.cpu().numpy() for centroid in new_centroids]
    stacked_centroids_np = np.stack(new_centroids)
    # Calculate pairwise distances between centroids using broadcasting
    distances = np.sqrt(np.sum((stacked_centroids_np[:, None] - stacked_centroids_np) ** 2, axis=-1))
    # Set the diagonal and upper triangle values to infinity to exclude self-comparisons and duplicates
    np.fill_diagonal(distances, np.inf)
    distances[np.triu_indices(len(new_centroids))] = np.inf
    # Set a threshold for similarity
    threshold = bandwidth  # Set your threshold here
    # Get indices of centroids that are sufficiently unique based on the threshold
    unique_indices = np.where(np.min(distances, axis=0) >= threshold)[0]
    # Gather unique centroids based on unique indices
    unique_centroids = np.array([torch.tensor(new_centroids[i]) for i in unique_indices])
    print('cluster number:',len(unique_centroids))
    print('center:', unique_centroids)
    return unique_centroids
```

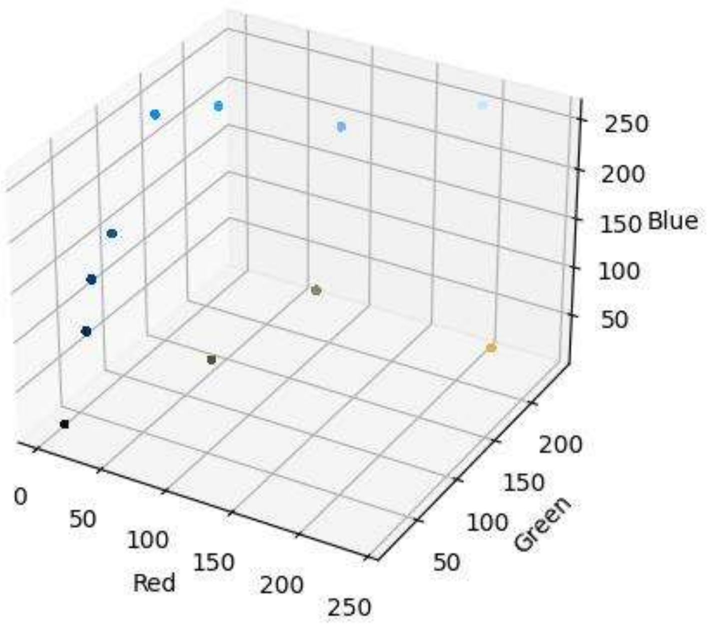
- Explanation
  進入 while，迴圈將持續計算新的 cluster center 直到達到停止條件（在此是 lopp 次數超過 1000 次或新中心點與舊中心點的距離變化小於 1e-5）。 在每個迴圈中： 對於每個 cluster center，計算其與資料點的距離，篩選出 bandwidth 內的 data point，然後計算這些點的 mean 作為新的中心點。 將新的中心點與舊的中心點進行比較，檢查它們之間的距離變化是否足夠小，若是則結束迭代，若沒有則利用 unique 來去除重複的點，可以加快程是運行。最後計算結果 cluster center data 之間的兩兩距離，並根據 threshold 排除 similar 的 cluster center point。 再將 data 和 cluster center 餵入 predict function 得到每個 pixel data 的屬於哪個 class 的 label。
- Results
  Bandwidth = 30(for 2-image)



img1_30_Pixel Distribution After Mean Shift

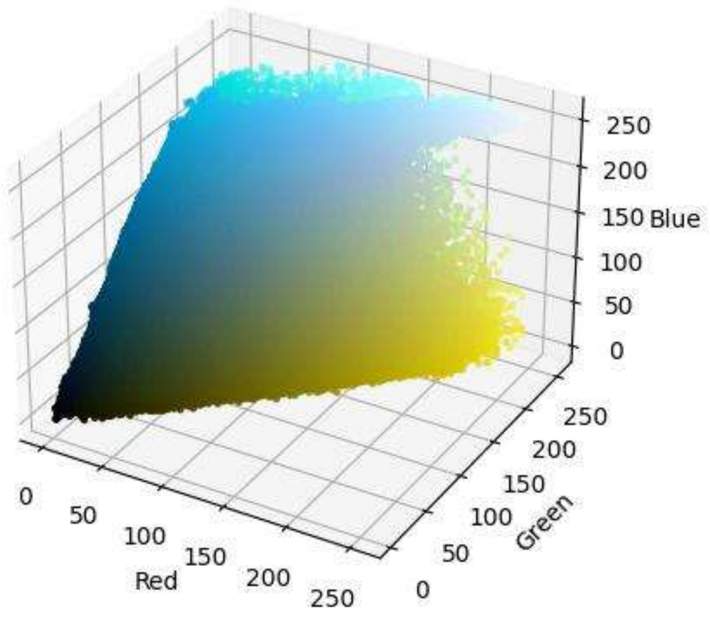


img1_30_clustering resulting

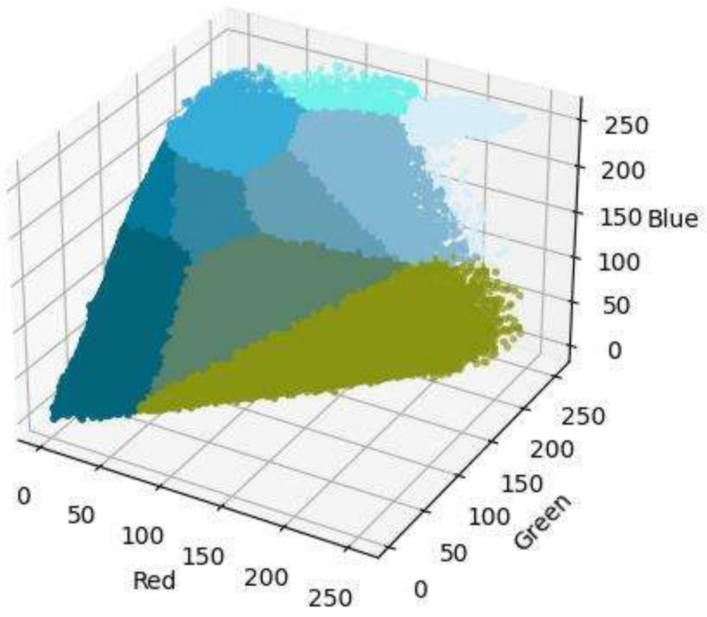


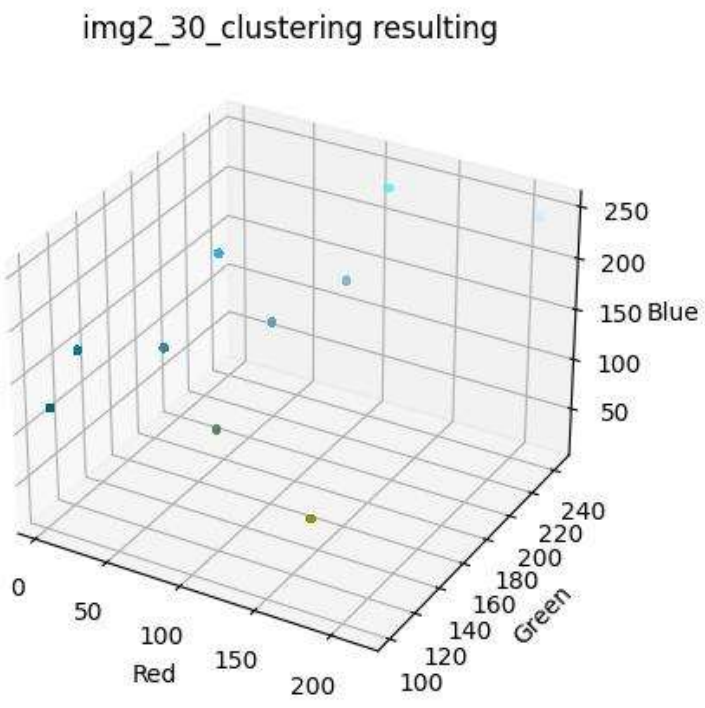
Bandwidth = 30(for 2-masterpiece)

img2__Pixel Distribution Before Mean Shift



img2_30_Pixel Distribution After Mean Shift





Step of D – spatial

- Code

```
print('channels_5')
image_5channel = np.zeros((img.shape[0], img.shape[1], 5), dtype=np.uint8)
image_5channel[:, :, :3] = img
# 創建 x_coords 和 y_coords 網格，表示圖像的 x 和 y 座標
x_coords, y_coords = np.meshgrid(np.arange(img.shape[1]), np.arange(img.shape[0]))
# Assign the spatial information to the new channels
image_5channel[:, :, 3] = x_coords.astype(np.uint8)
image_5channel[:, :, 4] = y_coords.astype(np.uint8)
print('spatial_mean_shifting...')
spatial_data = image_5channel.reshape(-1,5).astype(np.float64)
spatial_center = mean_shift_gpu(spatial_data, 90)
spatial_label = predict(spatial_data, spatial_center)
segmented_data = spatial_center[spatial_label]
segmented_img = segmented_data[:,:3].reshape(img.shape).astype(np.uint8)
filepath =  './output/' + 'img' + str(count) + '_' + 'spatial_mean_shift.jpg'
plt.imsave(filepath, segmented_img)
```

- Explanation

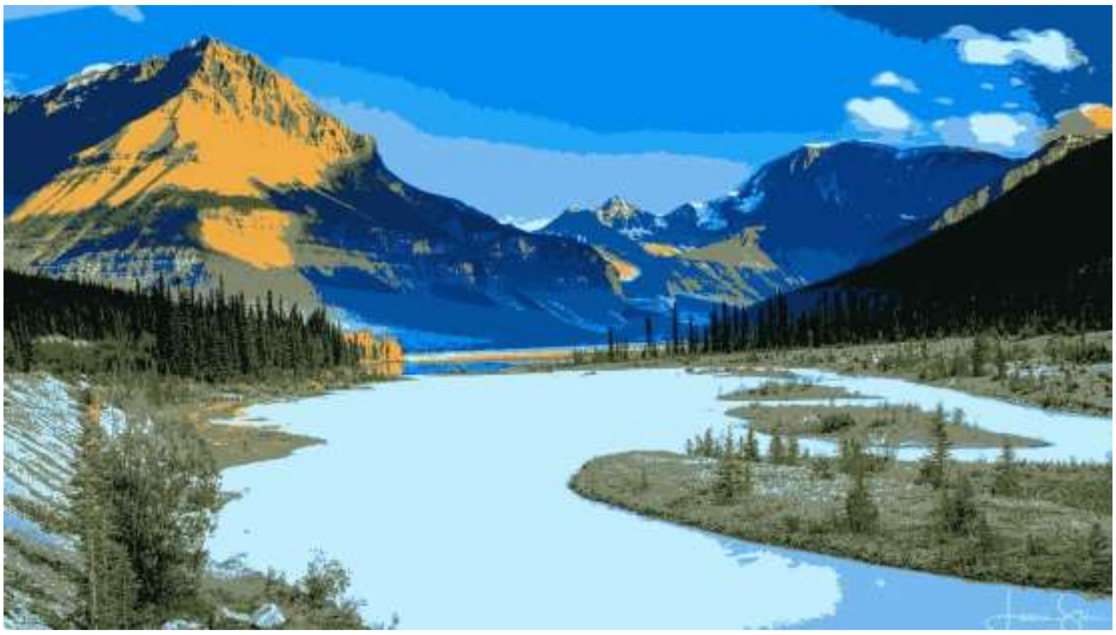
  一張圖像轉換成具有 5 個 channel 的新圖像，其中前三個 channel 是原始圖像的 RGB 通道，而後兩個通道分別表示像素點的 x 和 y 座標。再餵入 mean_shift_gpu function 和 step C 一樣的方式產生 mean shift 的結果。再透過前三個 channel 去還原 segmented image。

- Results(這個部分來不及對原始影像產生結果，所以我用五分之一大小作代替)

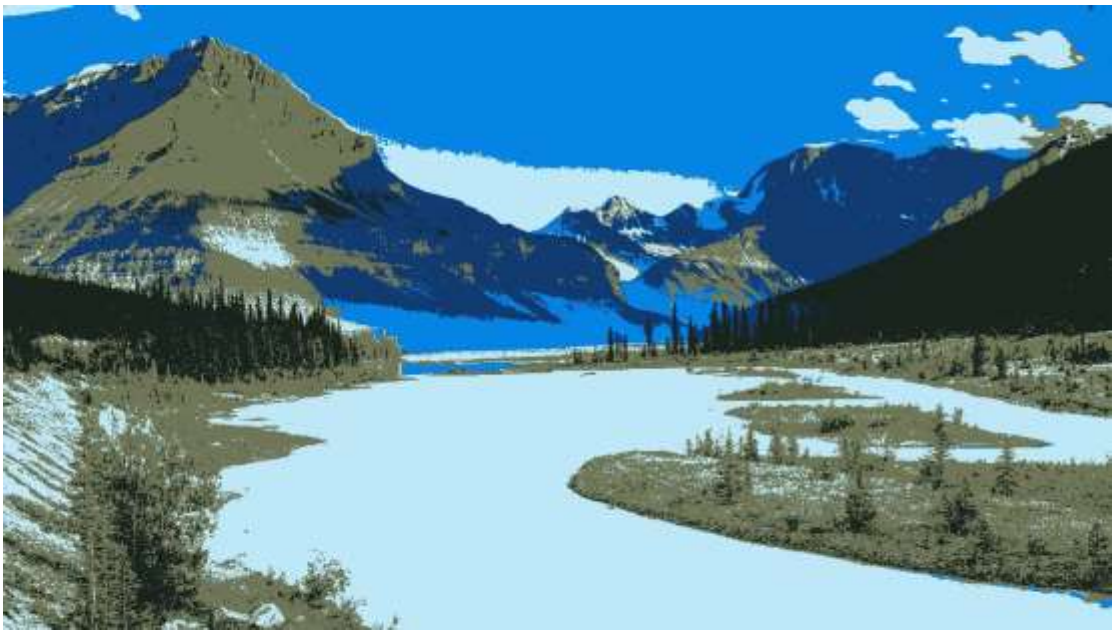
Step of E – different bandwidth

- Results

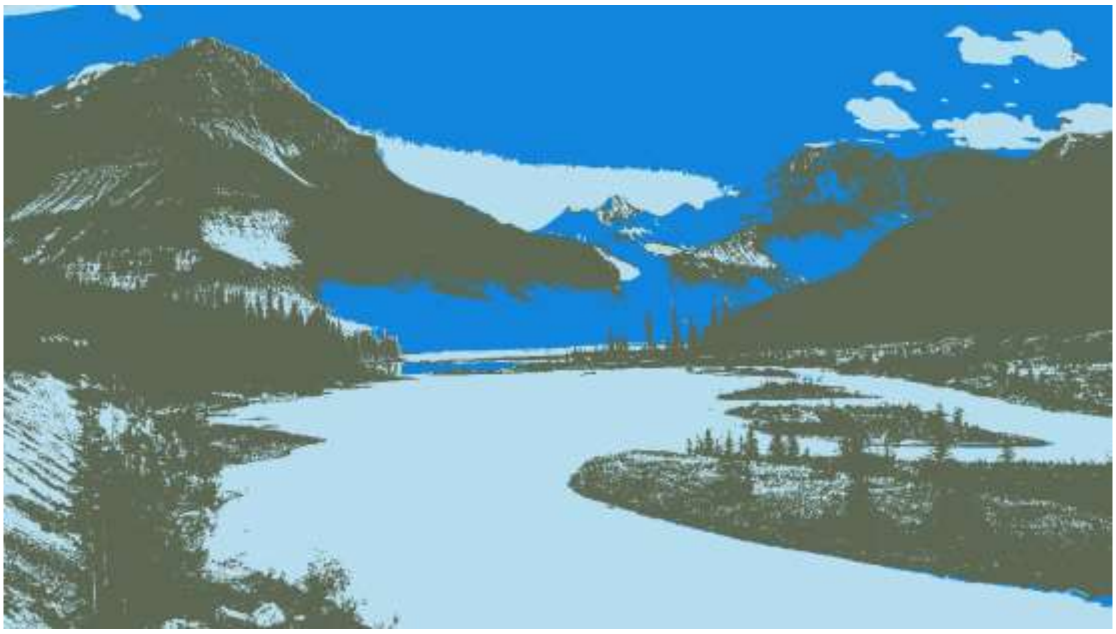
  Bandwidth = 30 (for 2-image)



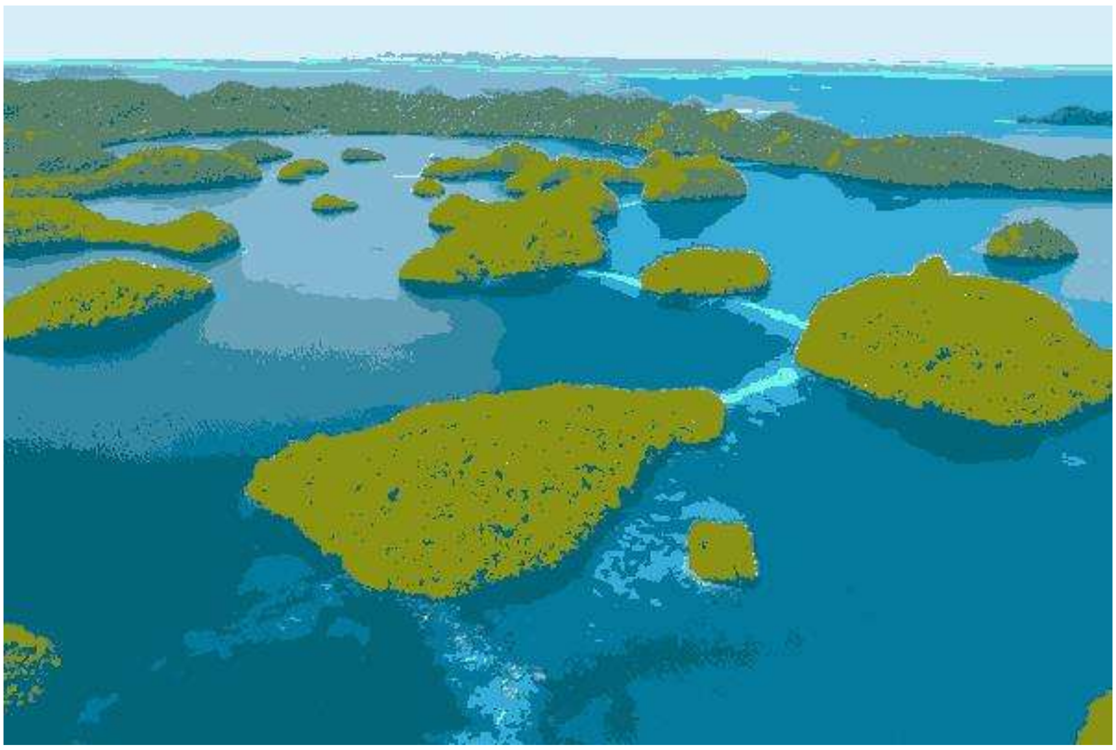
Bandwidth = 60



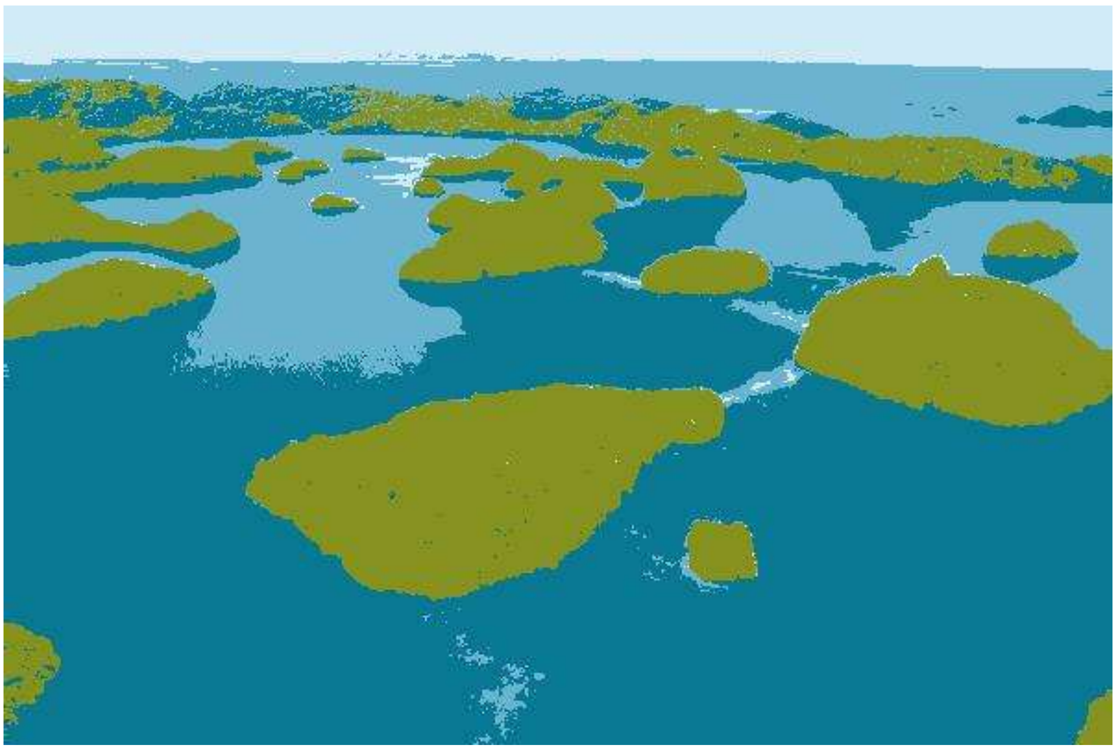
Bandwidth = 90



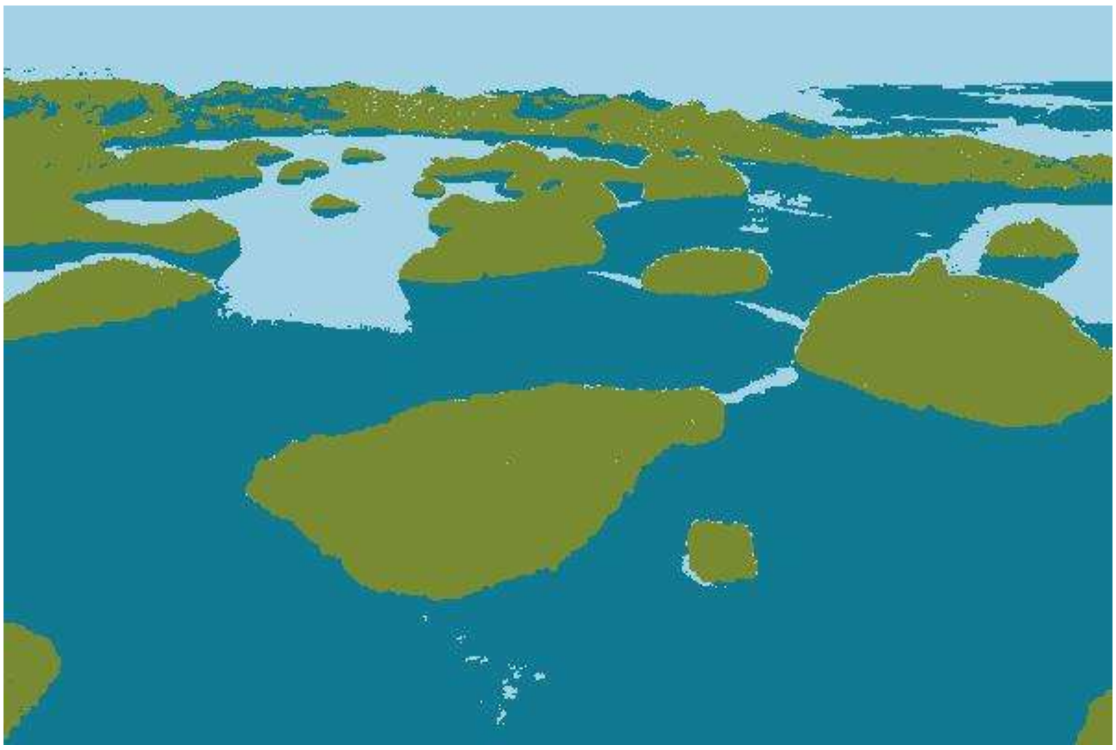
Bandwidth = 30 (for 2-masterpiece)



Bandwidth = 60



Bandwidth = 90



- Discussion
  由圖片的結果可以發現到說，較大的 bandwidth parameter 會導致過度平滑，造成特徵細節的丟失，像是在 bandwidth=90 的情況下，可以看到說色塊較少。若 bandwidth 太小，圖像容易出現過度分割，細微特徵會被切分成更多小區塊，可能導致細節過多且難以識別。

Step of F

- Discussion

  1. For the segmented results
  Mean shift 的穩定性較 k mean 高，clustering 的結果較準確由屠圖可以之，相對來說可以處理相對較複雜的影像。K mean 容易會受初始點影響相對比較適合處理簡單且具有明確分界的影像。
  2. For computation loss
  Mean shift 通常在處理大型數據時計算成本較高，因為它需要遍歷數據空間來找尋密度峰值。而 K 均值在計算上較為快速。再跑實驗的過程中可以明顯感受到 mean shift 相對於 k mean 需要花費較多的時間。