Report

1. explain your implementation.

本次作業使用講義所教之方法來實現圖片的拼接任務，實施步驟大略如下：

- SIFT 尋找特徵點
- KNN 做兩張圖片之特徵點配對
- RANSAC 多次迭代尋找兩張圖片間之最佳轉移矩陣 H
- 使用最佳轉移矩陣完成圖片拼接

接下來將解釋我在程式中如何實現這些步驟。

(1) SIFT 尋找特徵點

SIFT 主要用來偵測並描述影像中的局部特徵，在不同的 scale space 中尋找極值點後，取得這些極值點的位置、尺度、旋轉等資訊，最後計算出所有局部特徵點的方向。

```
def SIFT(img):
    siftDetector = cv2.SIFT_create() # limit 1000 points
    kp, des = siftDetector.detectAndCompute(img, None)

    return kp, des
```

此步驟使用 cv 提供的 operation 實現，首先將圖片轉至灰階作為輸入，接著透過 cv 提供的 operation 取得特徵點的座標、特徵向量兩項資訊，每個特徵向量都會是 128 維的形式。

每張欲拼接的照片都會透過此 SIFT 函式取得局部特徵點資訊。

(2) KNN 特徵點配對

該步驟透過 KNN 將每個特徵點與另一張圖片的每個特徵點 descriptor 做 2-norm 找出最小值(即相似度最高)，以此找出每個特徵點的匹配點。

由於特徵向量的維度高達 128 維，若使用暴力法直接將兩個 128 維的矩陣做 2-norm 距離計算，計算量將會非常大。因此我使用 KDTree 方法做 KNN，善用 KDTree 的二元樹特性降低資料和時間複雜度，提高搜尋鄰近點的效率。

```
# Use kdtree to search 128-D space of descripter
tree = KDTree(des1, leaf_size = 30)
matches = []
for i, x in enumerate(des2):
    distances, indices = tree.query([x], k = 2)
    matches.append([distances[0], indices[0][0], i])
```

首先將第一張圖片之特徵點轉換成 KDTree 之資料結構，接著使用 sklearn 提供之 KDTree 功能:tree.query，比較另一張圖中每個特徵點並取得最近點及次近點，存入 match list。

```
# Apply ratio test
good_index = []
for m in matches:
    if m[0][0] < threshold * m[0][1]:
        good_index.append([m[1], m[2]])
```

為避免最近點因其他點特徵相近導致錯誤匹配，Lowe's Ratio test 可以確保匹配點之準確性，比較最近點與次近點的比例，若小於閾值則判定為良好配對。此外，Lowe 的論文中提到，經過他測試後的合適閾值建議在 0.4~0.6 間，由於 baseline 的匹配難度較低，設定較高的匹配閾值可以增加匹配點數量，提升匹配效果，因此我在 baseline 設定閾值為 0.6。最後將良好的匹配組合記錄下來並回傳。

(3) RANSAC 與 Homography

RANSAC 在每次迭代中隨機採樣匹配點，計算出轉移矩陣 H 後，將第一張照片的每個特徵點經過 H 轉移到第二張照片的 frame，比較每個轉移後的點與第二張照片的對應點之距離誤差值，小於指定閾值則為 inlier，inlier 數量越高，代表 H 越貼近理想轉移情形，記錄此 H；經過多次迭代將記錄較佳的轉移矩陣，該轉移矩陣將用來進行最後拼接。

```
for i in range(iters):
    points = random_point(matches)
    H = homography(points) # img2 to img1

    #  avoid dividing by zero
    if np.linalg.matrix_rank(H) < 3:
        continue

    errors = get_error(matches, H)
    idx = np.where(errors < threshold)[0]
    inliers = matches[idx]

    num_inliers = len(inliers)
    if num_inliers > num_best_inliers:
        best_inliers = inliers.copy()
        num_best_inliers = num_inliers
        best_H = H.copy()
```

此步驟程式主要實現方法如下：

- Random_point 函式在所有匹配點中隨機提取四組匹配點

- Homography 函式使用該四組匹配點計算出轉移矩陣中之所有元素值，目的在於解出以下矩陣方程：

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1\hat{x_1} & -y_1\hat{x_1} \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2\hat{x_2} & -y_2\hat{x_2} \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3\hat{x_3} & -y_3\hat{x_3} \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4\hat{x_4} & -y_4\hat{x_4} \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1\hat{y_1} & -y_1\hat{y_1} \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2\hat{y_2} & -y_2\hat{y_2} \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3\hat{y_3} & -y_3\hat{y_3} \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4\hat{y_4} & -y_4\hat{y_4} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = h_{33} \begin{bmatrix} \hat{x_1} \\ \hat{x_2} \\ \hat{x_3} \\ \hat{x_4} \\ \hat{y_1} \\ \hat{y_2} \\ \hat{y_3} \\ \hat{y_4} \end{bmatrix}$$

  由於矩陣並非 n*n 形式，因此該處我使用 pseudo inverse，以 SVD 解出 h 矩陣，並將 h33 標準化為 1，對每個 h 元素值除以 h33。函式實現如下：

```
def homography(pairs):
    rows = []
    for i in range(pairs.shape[0]):
        p1 = np.append(pairs[i][0:2], 1)
        p2 = np.append(pairs[i][2:4], 1)
        row1 = [0, 0, 0, p2[0], p2[1], p2[2], -p1[1]*p2[0], -p1[1]*p2[1], -p1[1]*p2[2]]
        row2 = [p2[0], p2[1], p2[2], 0, 0, 0, -p1[0]*p2[0], -p1[0]*p2[1], -p1[0]*p2[2]]
        rows.append(row1)
        rows.append(row2)
    rows = np.array(rows)

    U, s, V = np.linalg.svd(rows)
    H = V[-1].reshape(3, 3)
    H = H/H[2, 2] # standardize to let w*H[2,2] = 1

    return H
```

- Get_error 函式計算所有「照片一經 H 轉移到照片二之特徵點」與「照片二之匹配特徵點」做距離誤差比較，並記錄這些誤差值回傳給 RANSAC 做 inlier 數量比較。函式如下：

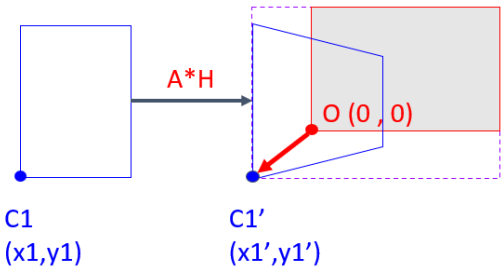
```
def get_error(points, H):
    num_points = len(points)
    all_p2 = np.concatenate((points[:, 2:4], np.ones((num_points, 1))), axis=1)
    all_p1 = points[:, 0:2]
    estimate_p1 = np.zeros((num_points, 2))
    for i in range(num_points):
        temp = np.dot(H, all_p2[i])
        estimate_p1[i] = (temp/temp[2])[0:2] # set index 2 to 1 and slice the index 0, 1
    # Compute error
    errors = np.linalg.norm(all_p1 - estimate_p1 , axis=1) ** 2

    return errors
```

- 若這些匹配距離誤差值小於指定閥值，則加入 inlier 點，此閥值因圖片拼接難度設定不一，我在此設定為 0.5。在每次迭代中比較該次的 inlier 數量是否比當前最高的還要多，以此紀錄最佳之 H 轉移矩陣。

(4) 圖片拼接(stitching)

此步驟主要實現三件事情：重設圖片原點、圖片映射、邊界優化。

由於圖片在映射到目標圖片後，其圖片長寬值將會改變，因此須設定一個新的原點，讓圖片可以完整的將每個部份顯示出來並輸出，原理如下圖。



此外，圖片拼接後，邊界因原始圖片 rgb 差異導致非常明顯，因此需對其做後處理，處理方法將會於後續說明。

我將整個拼接流程以一個函式實現，方法如下：

- 先將圖片二的 8 個轉角經由 H 映射到目標圖片，並從兩張圖共 8 個轉角找出 x 與 y 的最大與最小值。透過 x、y 的最小值當作兩張圖片之平移數值，取得平移矩陣，同時利用 x、y 的最大最小值設定圖片長寬值；接著將兩張圖片進行映射、平移及範圍調整，以取得接下來要用來合併的兩張輸入圖片。

```
# image corner deciding
height_l, width_l, channel_l = left.shape
height_r, width_r, channel_r = right.shape
corners = [[0, 0, 1], [width_r, 0, 1], [width_r, height_r, 1], [0, height_r, 1]]
corners_new = [np.dot(H, corner) for corner in corners]
corners_new = np.array(corners_new).T
x_news = corners_new[0] / corners_new[2]
y_news = corners_new[1] / corners_new[2]

# get the maximum and minimum points of two images
y_max = max(y_news)
x_max = max(x_news)
y_min = min(y_news)
x_min = min(x_news)

y_max = max(y_max, height_l)
x_max = max(x_max, width_l)
y_min = min(y_min, 0)
x_min = min(x_min, 0)

translation_mat = np.array([[1, 0, -x_min], [0, 1, -y_min], [0, 0, 1]]).astype('float')
H = np.dot(translation_mat, H)

# Get height, width
height_new = round(abs(y_max) - y_min)
width_new = round(abs(x_max)- x_min)
size = (width_new, height_new)

# warped image
warped_l = cv2.warpPerspective(src=left, M=translation_mat, dsize=size)
warped_r = cv2.warpPerspective(src=right, M=H, dsize=size)
```

- 合併過程同時也會處理邊界問題，處理方法透過兩張相同 pixel 的 rgb 資訊做 2-norm 比較，以 rgb 差異做不同的合併處理。

  此方法分為三種情況：前兩種為當其中一張圖片為黑色，則該 pixel 之 rgb 值以有顏色之圖片為主；當兩張圖片皆有 rgb 值，則先將兩者的 rgb 值取 2-norm，若數值大於所設定之閥值，則讓 rgb 比例傾向

其中一張，反之則兩個 rgb 值比例各半。此方法實測後，邊界問題有顯改善。

```
black = np.zeros(3)  # Black pixel.
beta = 0.15
for i in range(warped_r.shape[0]):
    for j in range(warped_r.shape[1]):
        pixel_l = warped_l[i, j, :]
        pixel_r = warped_r[i, j, :]

        if not np.array_equal(pixel_l, black) and np.array_equal(pixel_r, black):
            warped_l[i, j, :] = pixel_l
        elif np.array_equal(pixel_l, black) and not np.array_equal(pixel_r, black):
            warped_l[i, j, :] = pixel_r
        elif not np.array_equal(pixel_l, black) and not np.array_equal(pixel_r, black):
            # warped_l[i, j, :] = pixel_l * 0.5 + pixel_r * 0.5
            alpha = np.linalg.norm(pixel_l - pixel_r)
            if(np.linalg.norm(pixel_l) >= np.linalg.norm(pixel_r)):
                if (alpha > beta):
                    warped_l[i, j, :] = pixel_l * 0.98 + pixel_r*0.02
                else:
                    warped_l[i, j, :] = pixel_l*0.5 + pixel_r*0.5
            else:
                if (alpha > beta):
                    warped_l[i, j, :] = pixel_l * 0.02 + pixel_r*0.98
                else:
                    warped_l[i, j, :] = pixel_l*0.5 + pixel_r*0.5
```

2. show the result of stitching 2 images.

(1) baseline

(2) bonus

3. try to stitch more images as you can and compare with them.

我將 6 張 baseline 完成拼接，以及 4 張 bonus 的拼接。此部分我將說明如何完成多張圖的拼接，以及 bonus 的優化。

(1) 多張圖的拼接

首先，我同樣對每張圖片進行以 SIFT 找特徵點、KNN 特徵點匹配的動作。接著透過 ransac 函式，找出所有圖片兩兩間的轉移矩陣 H，如下圖：

```
inliers, H = ransac(matches, 0.5, 1000)
inliers2, H2 = ransac(matches2, 0.5, 1000)
inliers3, H3 = ransac(matches3, 0.5, 1000)
```

接下來將進行重複拼接，如下圖：

```
merge_img1, A1 = stitch_img(img1_rgb, img2_rgb, H)
merge_img2, A2 = stitch_img(merge_img1, img3_rgb, A1@H@H2)
merge_img3, A3 = stitch_img(merge_img2, img4_rgb, A1@H@H2@H3)
```

此拼接方法的概念是，**將每張圖片都映射到第一張圖片做拼接**。當進行到第三張、第四張圖時，映射的輸入矩陣應為前述所有轉移矩陣的連乘，以及乘上第一張圖的平移 A1，如此才能將後續的圖片完整映射到第一張圖。進行多次拼接後，最後一次的圖片輸出即為合併後的最終結果。

Baseline 之 6 張圖片也是如此，然而由於實測發現其加入平移後的拼接效果稍差，因此最佳的版本中並未使用到平移矩陣 A，僅使用 H 的連乘。程式實現如下：

```
merge_img1 = stitch_img(img1_rgb, img2_rgb, H)
merge_img2 = stitch_img(merge_img1, img3_rgb, H@H2)
merge_img3 = stitch_img(merge_img2, img4_rgb, H@H2@H3)
merge_img4 = stitch_img(merge_img3, img5_rgb, H@H2@H3@H4)
merge_img5 = stitch_img(merge_img4, img6_rgb, H@H2@H3@H4@H5)
```

拼接結果如下：

可以發現越後面拼接的圖片，其拼接效果會略為下降且扭曲情況較大，此部分由於本身拍攝角度之旋轉差異，導致後面的圖本來就會相對有扭曲情況，因此可以合理解釋。

(2) Bonus 優化

因 bonus 每張圖片之差異性較大，因此我在特徵點匹配部分提升了匹配

的難度，Lowe’s Ratio test 的閥值從 baseline 的 0.6 調整為 0.4；在 ransac 的 inlier 判定中，也將其閥值由 0.5 降低為 0.4，提升 inlier 之判定難度，同時將迭代次數增加，嘗試提升 H 準確性。
此外，也修正在拼接後之邊界處理方法，增加兩張圖 pixel 間的 rgb 差異範圍處理條件，以適應不同差異的 rgb 之融合比例。
經過上述調整前後比較如下：

調整前↓

調整後↓

可以觀察到「館」字的疊合以及樓梯部分有明顯改善，然而其邊界仍較為明顯，相對於原本之處理方法，已有稍微改善。