HW1	4109037012 資工四 李曉旻

作業描述：

這份作業實現 Stitching Image，包含提取圖像特徵點、匹配不同圖像間的特徵點、進行圖像變形和融合。

Coding part :

```
def extract_features(image):
    gradient = np.sqrt(ndimage.sobel(image, axis=0)**2 + ndimage.sobel(image, axis=1)**2)
    keypoints = []
    threshold = 0.1 * gradient.max()
    for y in range(1, gradient.shape[0] - 1):
        for x in range(1, gradient.shape[1] - 1):
            if (gradient[y, x] > threshold).all() and \
                (gradient[y, x] > np.array([gradient[y-1, x], gradient[y+1, x], gradient[y, x-1], gradient[y, x+1]])).all():
                keypoints.append((x, y))

    features = []
    for kp in keypoints:
        x, y = kp
        patch = image[max(0, y-8):min(image.shape[0], y+8), max(0, x-8):min(image.shape[1], x+8)]
        features.append(patch.flatten())

    return keypoints, features
```

特徵提取

計算影像的梯度，將其轉化為梯度幅度，接著在梯度幅度圖中找出大於閾值且比周圍點大的點作為特徵點，提取其周圍 16x16 區域的特徵向量，並將這些特徵向量展平為一維向量。

```
def matchKeyPoint(kps_l, kps_r, features_l, features_r, ratio):
    Match_idxAndDist = []
    for i in range(len(features_l)):
        min_IdxDis = [-1, np.inf]
        secMin_IdxDis = [-1, np.inf]
        for j in range(len(features_r)):
            if features_l[i].shape != features_r[j].shape:
```

```
            continue
        dist = np.linalg.norm(features_l[i] - features_r[j])
        if min_IdxDis[1] > dist:
            secMin_IdxDis = np.copy(min_IdxDis)
            min_IdxDis = [j, dist]
        elif secMin_IdxDis[1] > dist and secMin_IdxDis[1] != min_IdxDis[1]:
            secMin_IdxDis = [j, dist]

    Match_idxAndDist.append([min_IdxDis[0], min_IdxDis[1],
secMin_IdxDis[0], secMin_IdxDis[1]])

goodMatches = []
for i in range(len(Match_idxAndDist)):
    if Match_idxAndDist[i][1] <= Match_idxAndDist[i][3] * ratio:
        goodMatches.append((i, Match_idxAndDist[i][0]))

goodMatches_pos = []
for (idx, correspondingIdx) in goodMatches:
    psA = (int(kps_l[idx][0]), int(kps_l[idx][1]))
    psB = (int(kps_r[correspondingIdx][0]),
int(kps_r[correspondingIdx][1]))
    goodMatches_pos.append([psA, psB])

return goodMatches_pos
```

配對兩張影像中的特徵點

計算每個特徵向量之間的歐氏距離，找出最小距離和次小距離，根據比值測試過濾掉不良配對，只保留比值小於設定閾值的。最後將好的配對轉換成特徵點對應的座標位置 Return。

```
def drawMatches(imgs, matches_pos):
    img_left, img_right = imgs
    (hl, wl) = img_left.shape[:2]
    (hr, wr) = img_right.shape[:2]
    vis = np.zeros((max(hl, hr), wl + wr, 3), dtype="uint8")
    vis[0:hl, 0:wl] = img_left
    vis[0:hr, wl:] = img_right

    for (img_left_pos, img_right_pos) in matches_pos:
        pos_l = img_left_pos
```

```
        pos_r = img_right_pos[0] + w1, img_right_pos[1]
        plt.plot([pos_l[0], pos_r[0]], [pos_l[1], pos_r[1]], color='blue')
        plt.scatter([pos_l[0], pos_r[0]], [pos_l[1], pos_r[1]], color=['red',
'green'])
    plt.imshow(vis)
    plt.title("Matching points")
    plt.show()
```

創建一張空白影像，將左圖和右圖分別放在空白影像的左側和右側，將匹配點之間的連接線繪製出來，左圖特徵點用紅色標記，右圖特徵點用綠色標記。

```
def solve_homography(src_points, dst_points):
    A = []
    for i in range(src_points.shape[0]):
        x, y = src_points[i, 0], src_points[i, 1]
        u, v = dst_points[i, 0], dst_points[i, 1]
        A.append([-x, -y, -1, 0, 0, 0, u * x, u * y, u])
        A.append([0, 0, 0, -x, -y, -1, v * x, v * y, v])

    A = np.array(A)
    U, S, Vt = np.linalg.svd(A)
    H = Vt[-1].reshape((3, 3))

    return H
```

計算 homography matrix

先構建方程系統的矩陣 A，其來源點和目標點之間的關係被轉換成一組線性方程，然後對矩陣 A 進行 SVD 分解，並從分解結果中提取最後一個向量，將其重構為 3x3 的同質矩陣 H。

```
def estimate_homography_ransac(matches_pos):
    dstPoints = []
    srcPoints = []
    for dstPoint, srcPoint in matches_pos:
        dstPoints.append(list(dstPoint))
        srcPoints.append(list(srcPoint))
    dstPoints = np.array(dstPoints)
    srcPoints = np.array(srcPoints)

    NumSample = len(matches_pos)
    threshold = 5.0
```

```
    NumIter = 8000
    NumRamdomSubSample = 4
    MaxInlier = 0
    Best_H = None
    for run in range(NumIter):
        SubSampleIdx = random.sample(range(NumSample), NumRamdomSubSample)
        H = solve_homography(srcPoints[SubSampleIdx], dstPoints[SubSampleIdx])

        NumInlier = 0
        for i in range(NumSample):
            if i not in SubSampleIdx:
                concateCoor = np.hstack((srcPoints[i], [1]))
                dstCoor = H @ concateCoor.T
                if dstCoor[2] <= 1e-8:
                    continue
                dstCoor = dstCoor / dstCoor[2]
                if np.linalg.norm(dstCoor[:2] - dstPoints[i]) < threshold:
                    NumInlier += 1
        if MaxInlier < NumInlier:
            MaxInlier = NumInlier
            Best_H = H

    print("The Number of Maximum Inlier:", MaxInlier)
    return Best_H
```

使用 RANSAC 算法估計兩張圖片間的單應性矩陣，為了找出最佳的單應性矩陣，處理錯誤匹配，提高拼接準確性。

```
def warp(imgs, HomoMat):
    img_left, img_right = imgs
    (hl, wl) = img_left.shape[:2]
    (hr, wr) = img_right.shape[:2]
    stitch_img = np.zeros((max(hl, hr), wl + wr, 3), dtype="int")

    inv_H = np.linalg.inv(HomoMat)
    for i in range(stitch_img.shape[0]):
        for j in range(stitch_img.shape[1]):
            coor = np.array([j, i, 1])
            img_right_coor = inv_H @ coor
            img_right_coor /= img_right_coor[2]
```

```
            y, x = int(round(img_right_coor[0])), int(round(img_right_coor[1]))

            if x < 0 or x >= hr or y < 0 or y >= wr:
                continue
            stitch_img[i, j] = img_right[x, y]

    stitch_img = linearBlending([img_left, stitch_img])
    stitch_img = removeBlackBorder(stitch_img)

    return stitch_img
```

實現圖像拼接的扭曲和融合過程

首先創建一個大畫布，然後使用逆單應性矩陣將右圖像的像素映射到新的位置，遍歷每個像素，計算其在原始右圖中的對應位置，並將顏色值複製到新畫布上。這個反向映射過程是程式的核心，確保了圖像的正確變形。最後，函數應用線性融合算法來平滑過渡區域，並移除黑色邊框，得到最終的拼接結果。

```
def removeBlackBorder(img):
    h, w = img.shape[:2]
    reduced_h, reduced_w = h, w
    for col in range(w - 1, -1, -1):
        all_black = True
        for i in range(h):
            if np.count_nonzero(img[i, col]) > 0:
                all_black = False
                break
        if all_black:
            reduced_w -= 1
    for row in range(h - 1, -1, -1):
        all_black = True
        for i in range(reduced_w):
            if np.count_nonzero(img[row, i]) > 0:
                all_black = False
                break
        if all_black:
            reduced_h -= 1
    return img[:reduced_h, :reduced_w]
```

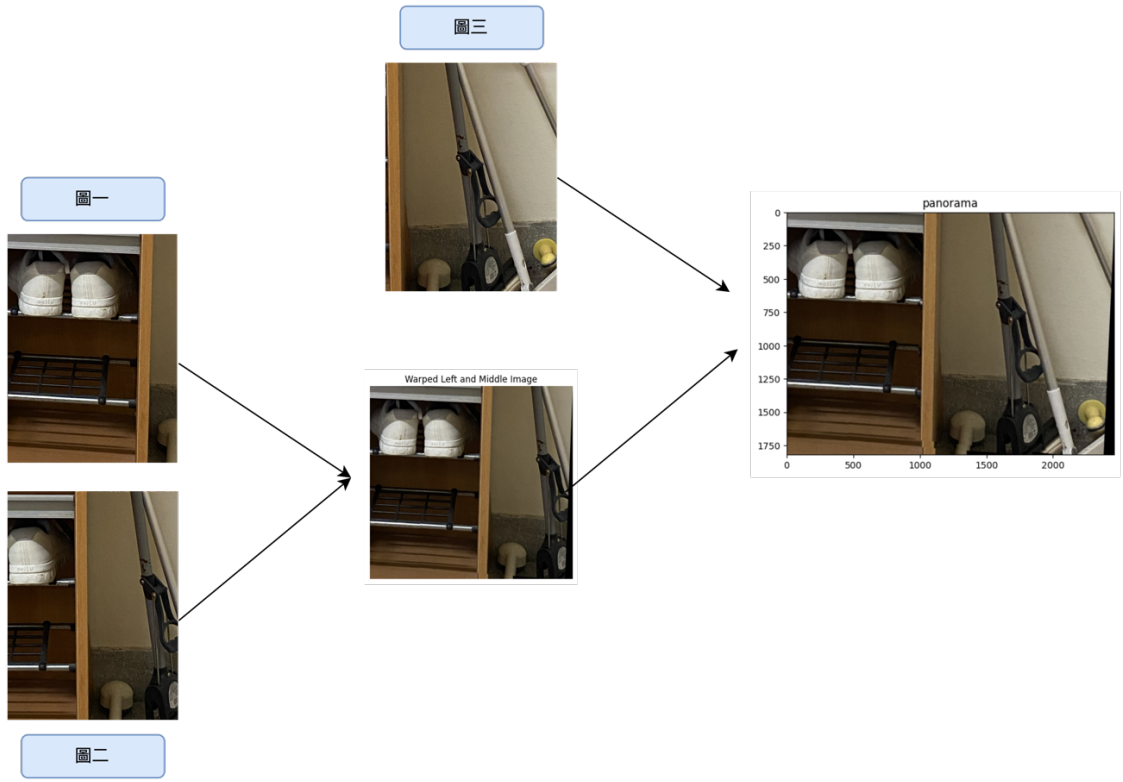
移除圖像中的黑色邊框，如果發現全黑的列或行，就將圖像的寬度或高度減少。

```
def linearBlending(imgs):
    img_left, img_right = imgs
    (hl, wl) = img_left.shape[:2]
    (hr, wr) = img_right.shape[:2]

    centerX = wl // 2
    for i in range(hl):
        for j in range(centerX, wl):
            if np.count_nonzero(img_right[i, j]) > 0:
                alpha = 1 - ((j - centerX) / (wl - centerX))
                img_left[i, j] = img_left[i, j] * alpha + img_right[i, j] * (1 -
alpha)

    return img_left
```

實作線性融合，用於平滑兩張圖像的過渡區域，對於重疊區域的每個像素，函數根據 alpha 值對左右圖像的顏色進行加權平均，消除了拼接處的明顯邊界，創造更自然、無縫的過渡效果，提高了拼接圖像的視覺質量。

Stitching Functions

這邊的做法是先將 1 和 2 圖做一次拼接，將產出的結果再與 3 做一次拼接，如下圖所示：

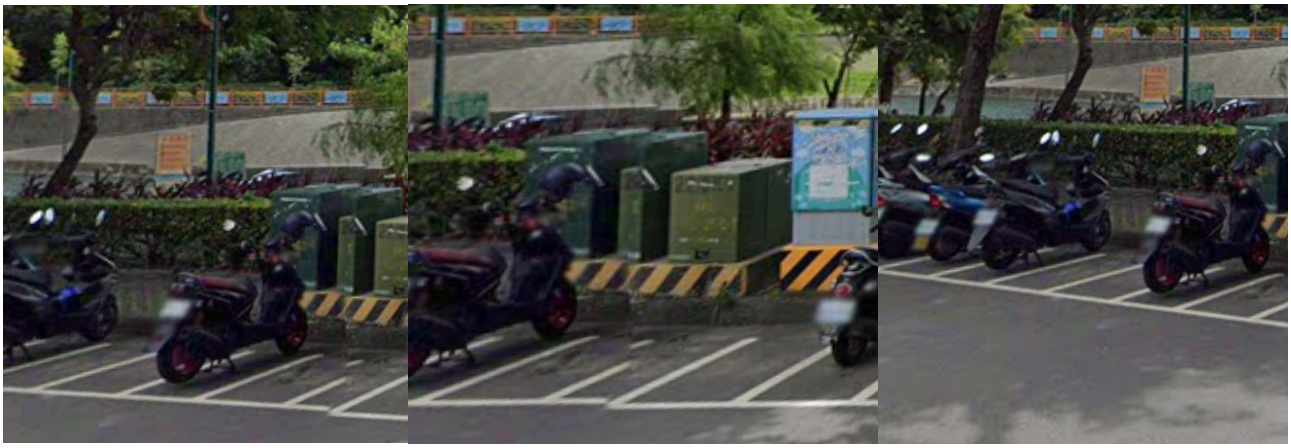


Result:

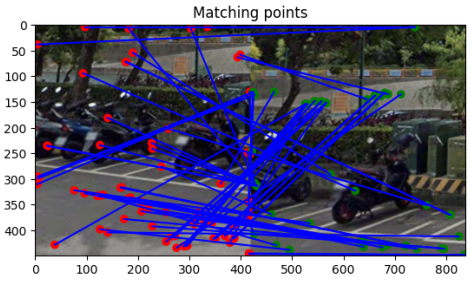
（我做了 2 種實驗，分別為有用到額外套件的以及全部手刻的程式）

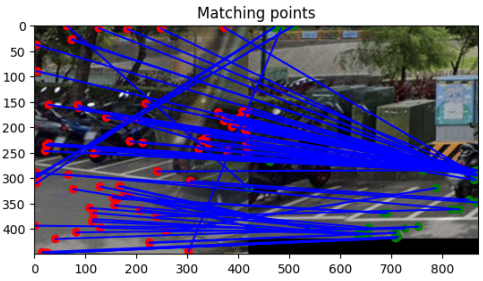
----------手動找特徵點、拼接----------

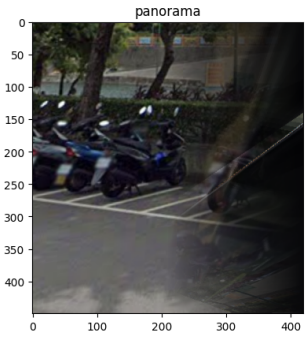
原圖 1(三張)



第一次特徵點連接　　第二次特徵點連接　　結果圖

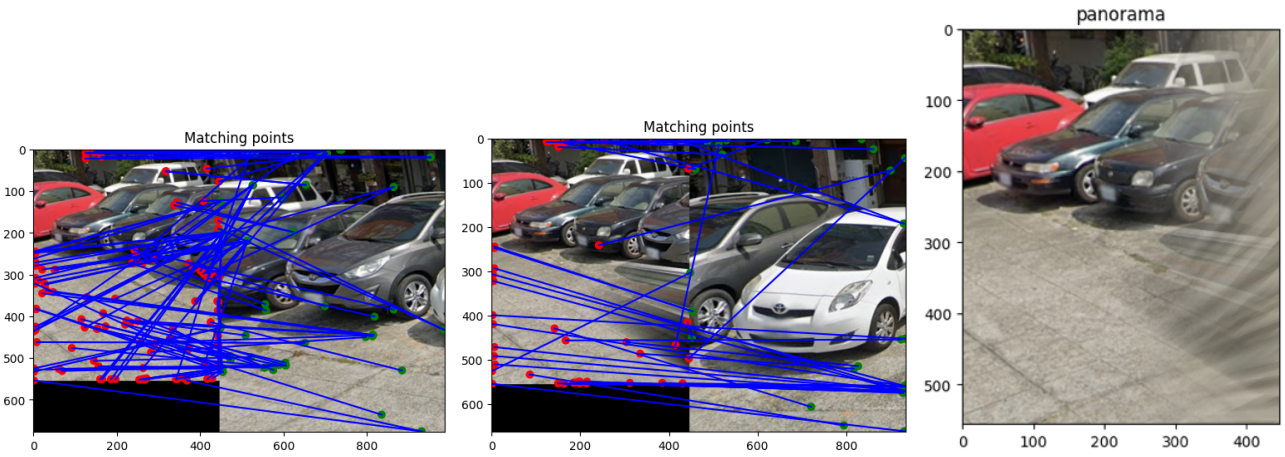






原圖 2(三張)

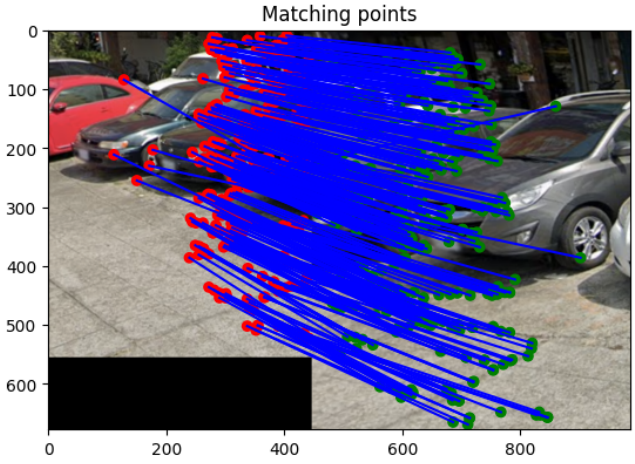
第一次特徵點連接　　第二次特徵點連接　　結果圖



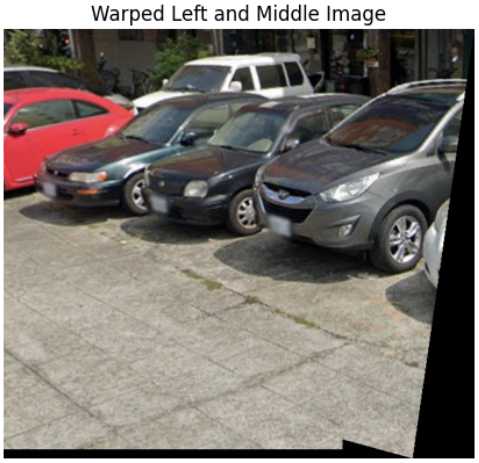
----------有用套件找特徵點----------

原圖 1(三張)

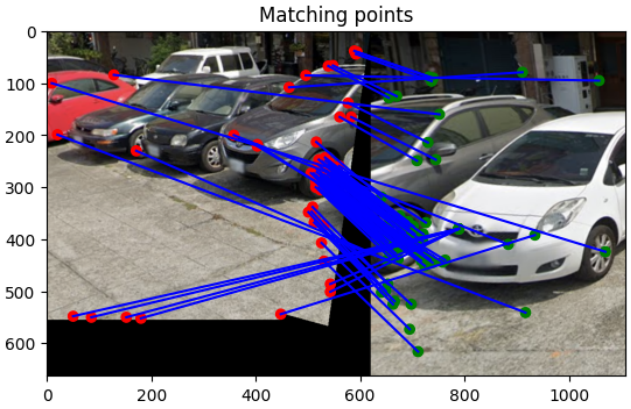
第一次特徵點連接



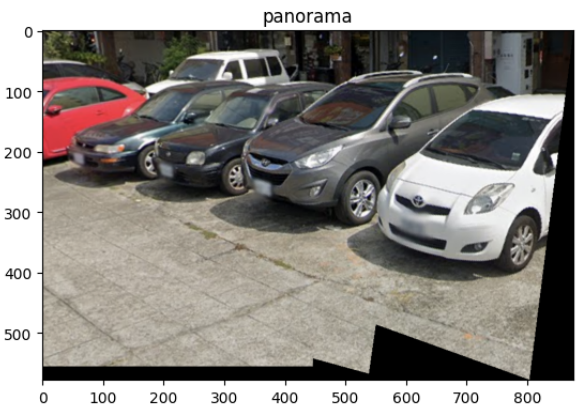
第一次拼接



第二次特徵點連接



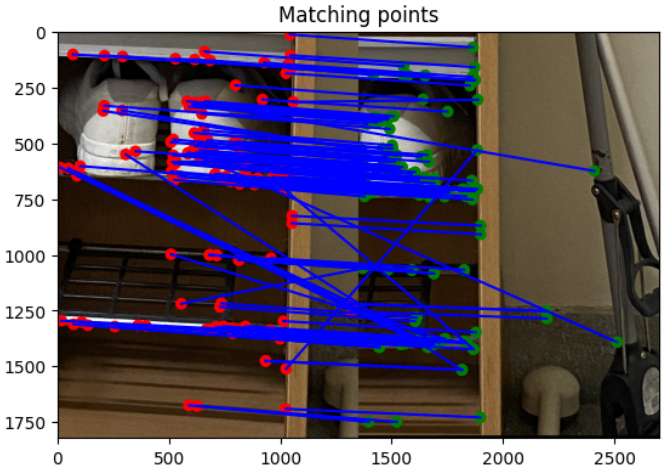
結果圖



原圖 2(三張)

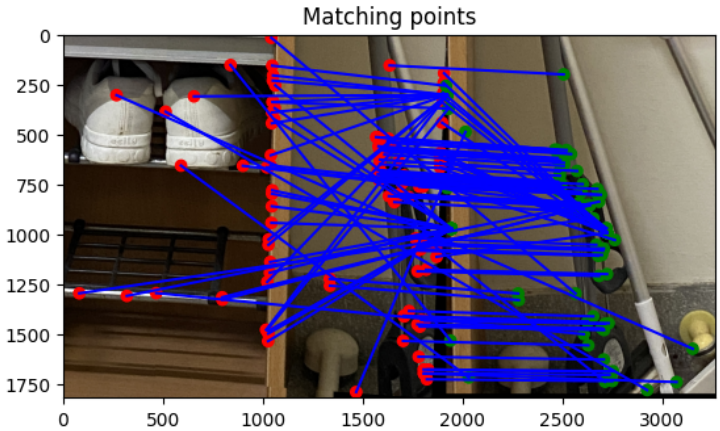
第一次特徵點連接

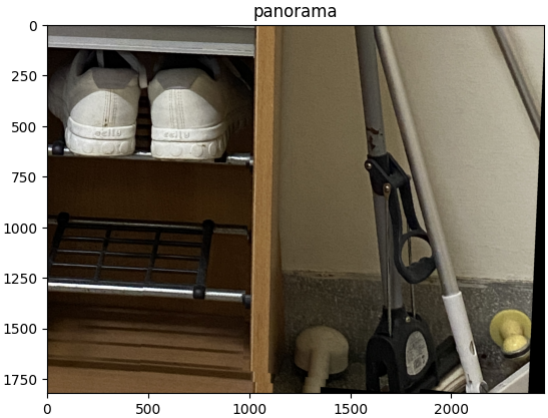
第一次拼接



第二次特徵點連接

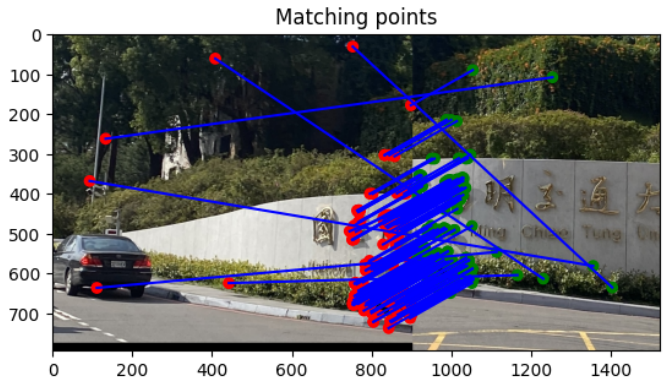
結果圖

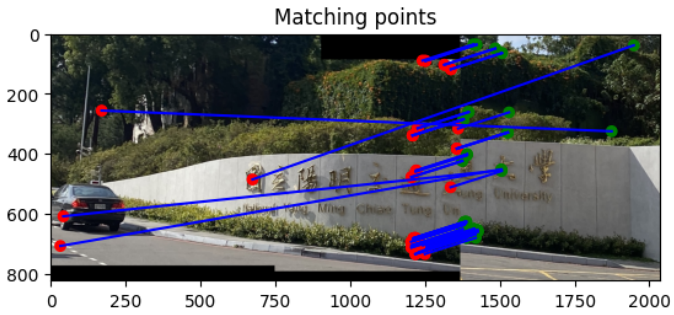




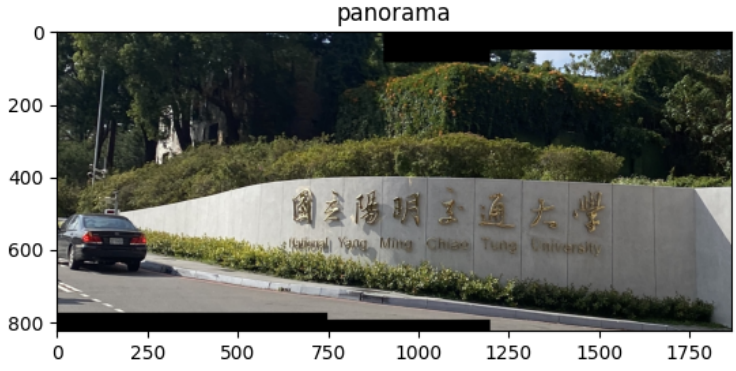
原圖 3(三張)

第一次特徵點連接　　第一次拼接





第二次特徵點連接



結果圖

Discussion:

實驗比較了完全手動實現和使用部分現有函式庫兩種方法，結果顯示手動方法在特徵點檢測方面就表現不佳，導致拼接效果較差，相比之下，使用函式庫的方法在特徵點檢測和匹配上明顯優於手動方法，產生了更精確的拼接結果。

從展示的拼接結果可以看出，對於簡單的場景可以產生流暢、無縫的拼接效果，但對於復雜場景像是圖片大小不一、視角歪斜或是整體圖片較暗的圖，會出現一些拼接重影(ghosting)、照片傾斜等問題，猜測是圖像之間的重疊區域太小、旋轉角度太大等因素導致無法找到足夠的特徵匹配點，無法正確估計單應變換矩陣，影響最終視覺效果。

心得:

交出這份作業也代表結束我大學四年的課業了，很感謝這堂課讓我學到很有趣的影像知識，我之前從未研究過這個領域，沒想到每一張圖片的背後都有這麼多的數學轉換，照片的成像都有其道理，也感謝助教給我補交作業的機會，讓我能真正的能理解這份作業在做的事情。