Image Alignment with SIFT

WZX
13 min read · 2 hours ago

圖像對齊

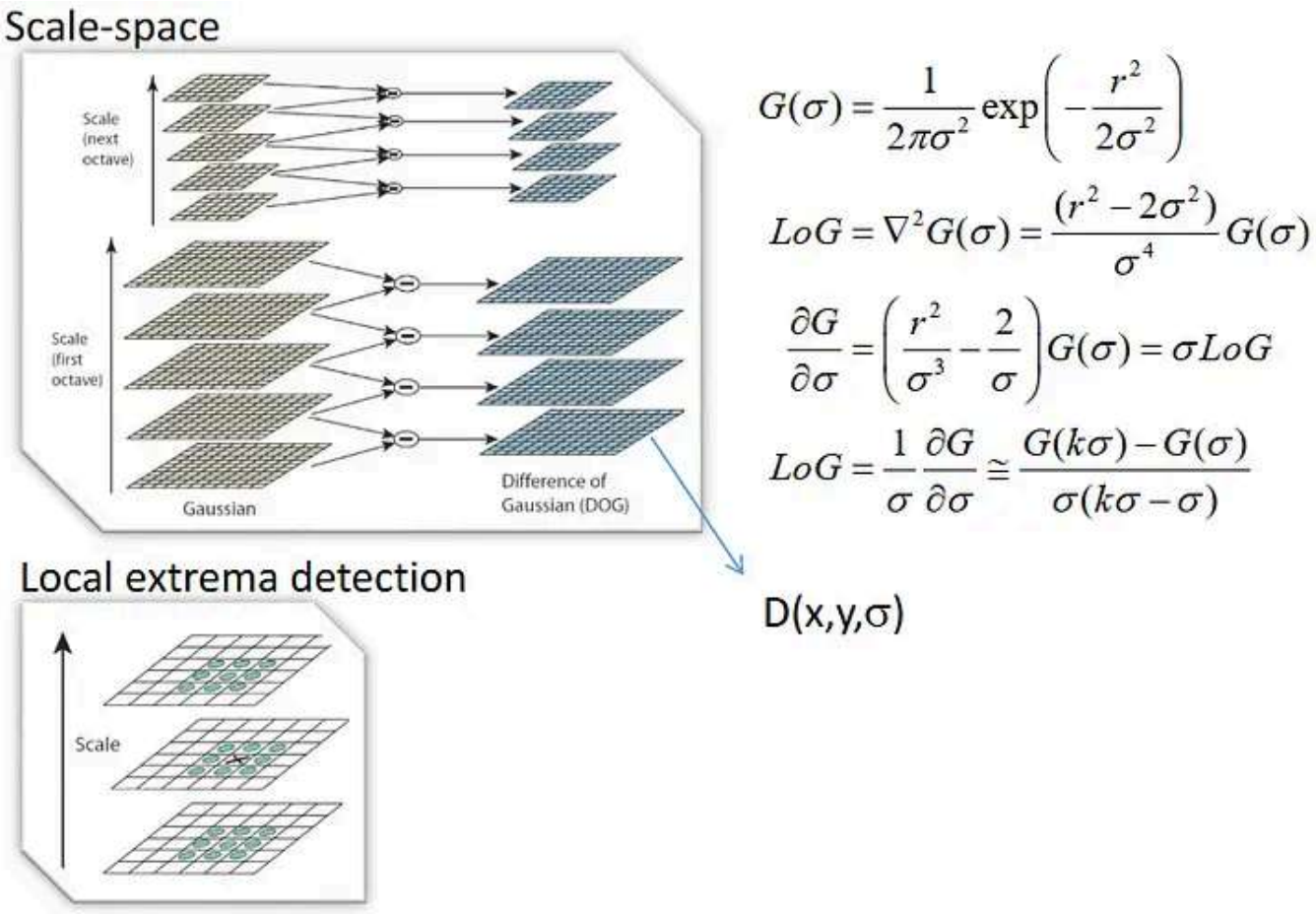
目錄

關鍵點偵測與描述 (Keypoint Detection and Description)

SIFT (Scale-Invariant Feature Transform):

尺度空間極值偵測:

SIFT 首先建立影像的**高斯尺度空間** (Scale-Space)，也就是對原始影像進行不同程度的高斯模糊。接著計算相鄰尺度影像之間的**高斯差分** (Difference-of-Gaussians, DoG)，這個 DoG 影像可以有效地近似**拉普拉斯高斯** (Laplacian of Gaussian, LoG)。SIFT 在這些 DoG 影像的不同尺度和空間位置上尋找局部極值點（最大值或最小值），這些點是潛在的關鍵點。



Scale-space extrema detection

精確關鍵點定位:

對於找到的潛在關鍵點，進行更精確的定位，透過 Taylor expansion 去除低對比度的點(D絕對值低)以及邊緣響應點（因為邊緣上的點定位不穩定），確保關鍵點的穩定性。

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^{\mathbf{T}}\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

$$\Delta\mathbf{x} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1}\frac{\partial D}{\partial \mathbf{x}}$$

泰勒展開式（至二階項）

$$\frac{Tr(\mathbf{H})^2}{Det(\mathbf{H})} \geq \frac{(r+1)^2}{r}$$

移除極值滿足以上式子

方向分配 (Orientation Assignment):

為了實現旋轉不變性，SIFT 計算每個關鍵點周圍鄰域像素的梯度方向和幅度。根據梯度方向的直方圖，找出一個或多個主方向分配給該關鍵點。後續的描述符計算會相對於這個主方向進行。

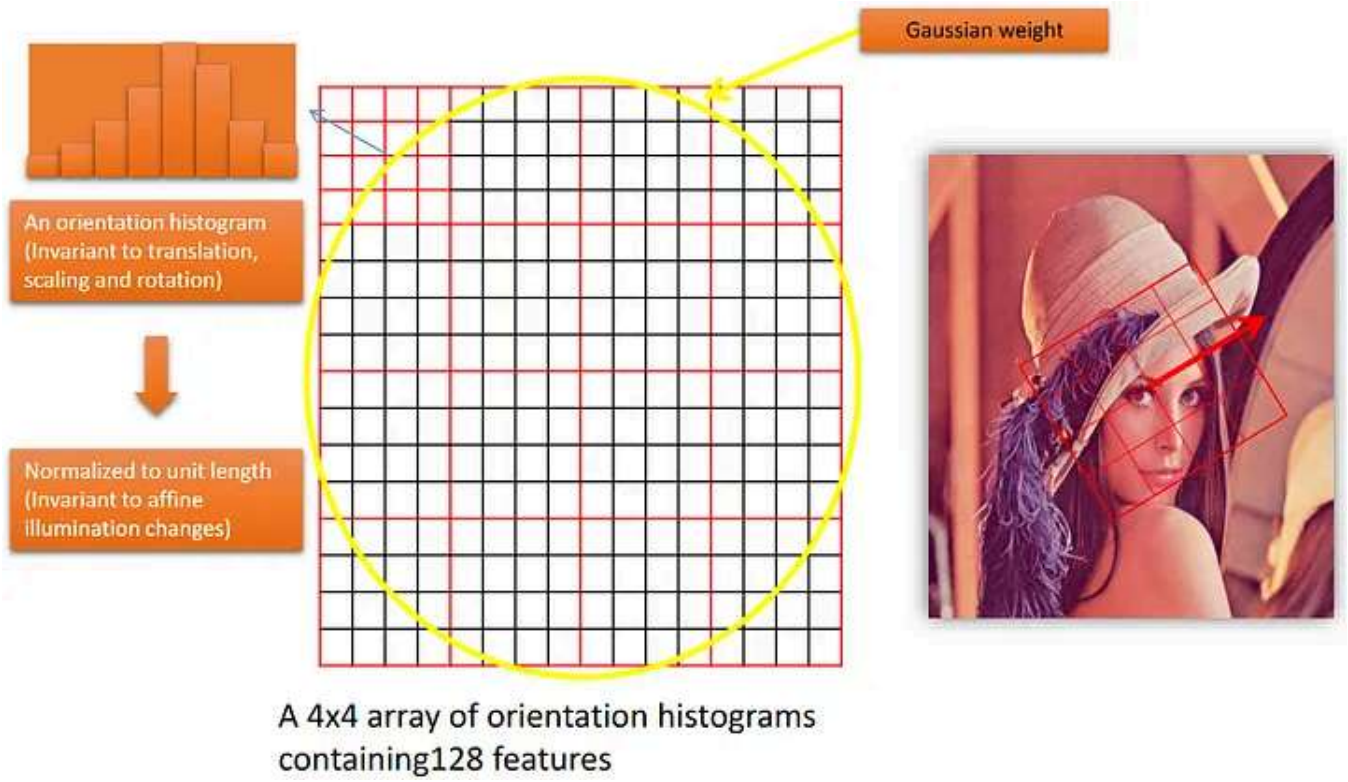
梯度方向直方圖

描述符 (Descriptor):

在確定了關鍵點的位置、尺度和主方向後，SIFT 在關鍵點周圍（根據其尺度和主方向旋轉對齊）的一個鄰域內計算**梯度方向直方圖 (Histogram of Oriented Gradients, HOG)**。

通常這個鄰域被劃分成 4x4 的子區域，每個子區域計算 8 個方向的梯度直方圖。

將這 16 個子區域的直方圖串接起來，形成一個 **128 維的向量**，就是 SIFT 描述符。這個描述符對光照變化具有一定的魯棒性（因為基於梯度），且由於主方向的對齊，它對旋轉是不變的。



局部影像描述符

程式:

使用 RootSIFT： 這是在 SIFT 描述符基礎上做的一個簡單但通常效果更好的改進。對光照變化更不敏感。計算方法是對 L1 歸一化後的 SIFT 描述符取平方根。

```
# 讀取圖片
img1_color = cv2.imread(img1_path)
img2_color = cv2.imread(img2_path)

img1_gray = cv2.cvtColor(img1_color, cv2.COLOR_BGR2GRAY)
img2_gray = cv2.cvtColor(img2_color, cv2.COLOR_BGR2GRAY)
h1, w1 = img1_gray.shape
h2, w2 = img2_gray.shape

# 初始化 SIFT 偵測器
sift = cv2.SIFT_create()

# RootSIFT 轉換函數
def rootsift(des):
    if des is None: return None
    # L1 歸一化 + 平方根 (添加一個小值避免除以零)
    des = np.float32(des) # 確保是 float32
    des /= (np.sum(np.abs(des), axis=1, keepdims=True) + 1e-7)
    des = np.sqrt(des)
    return des

des1 = rootsift(des1)
```

```
des2 = rootsift(des2)

# 視覺化
img1_keypoints = cv2.drawKeypoints(img1_color, kp1, None, flags=cv2.DRAW_MATCHE
img2_keypoints = cv2.drawKeypoints(img2_color, kp2, None, flags=cv2.DRAW_MATCHE
plt.figure(figsize=(20, 10))
plt.subplot(1, 2, 1); plt.imshow(cv2.cvtColor(img1_keypoints, cv2.COLOR_BGR2RGB
plt.subplot(1, 2, 2); plt.imshow(cv2.cvtColor(img2_keypoints, cv2.COLOR_BGR2RGB
```

特徵匹配 (Feature Matching)

Brute-Force Matcher (cv2.BFMatcher):

暴力匹配器。拿第一張圖片（參考圖）中的每個 SIFT 描述符，去和第二張圖片（待對齊圖）中的**所有** SIFT 描述符計算距離（相似度）。

FLANN (Fast Library for Approximate Nearest Neighbors):

使用 k-d 樹或 LSH 等數據結構，可以**顯著加速**高維度描述符（如 SIFT 的 128 維）的最近鄰搜索過程。它找到的是「近似」最近鄰，但在實踐中精度損失很小，速度提升卻非常大。對於非二元描述符，FLANN 通常是比 BF Matcher 更好的選擇。

程式:

```
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50) # checks 值越高越準確但越慢

flann = cv2.FlannBasedMatcher(index_params, search_params)
# 確保描述符是 float32 for FLANN
```

```
if des1.dtype != np.float32: des1 = np.float32(des1)
if des2.dtype != np.float32: des2 = np.float32(des2)

matches = flann.knnMatch(des1, des2, k=2)
```

匹配篩選 (Match Filtering)

Lowe's Ratio Test:
透過比較最佳匹配 (m) 和次佳匹配 (n) 的距離，if m.distance < RATIO_THRESHOLD * n.distance:

只有當最佳匹配顯著優於次佳匹配時（即兩者距離之比較小於一個閾值），才認為這個匹配是可靠的、具有區分度的 "良好匹配" (good_matches)。能有效濾除由重複紋理或噪聲引起的模糊或錯誤匹配。

程式:

```
RATIO_THRESHOLD = 0.5
good_matches = []
# 檢查 knnMatch 是否返回了成對的結果
valid_matches_count = sum(1 for pair in matches if len(pair) == 2)
if valid_matches_count > 0:
    for m, n in (pair for pair in matches if len(pair) == 2): # 只處理成對的匹配
        if m.distance < RATIO_THRESHOLD * n.distance:
            good_matches.append(m)

# 顯示良好的匹配點
if good_matches:
    img_good_matches = cv2.drawMatches(img1_color, kp1, img2_color, kp2, good_m
    plt.figure(figsize=(20, 10)); plt.imshow(cv2.cvtColor(img_good_matches, cv2
```

幾何變換估計 (Geometric Transformation Estimation)

Homography (單應性矩陣):

假設兩張圖片之間的主要差異可以透過一個**平面透視變換**來描述（適用於平面場景、相機純旋轉或物體很遠的情況）。Homography 是一個 3x3 矩陣 M，可以將一張圖片上的點座標 (x, y) 映射到另一張圖片上對應的點座標 (x', y')。

RANSAC (Random Sample Consensus):

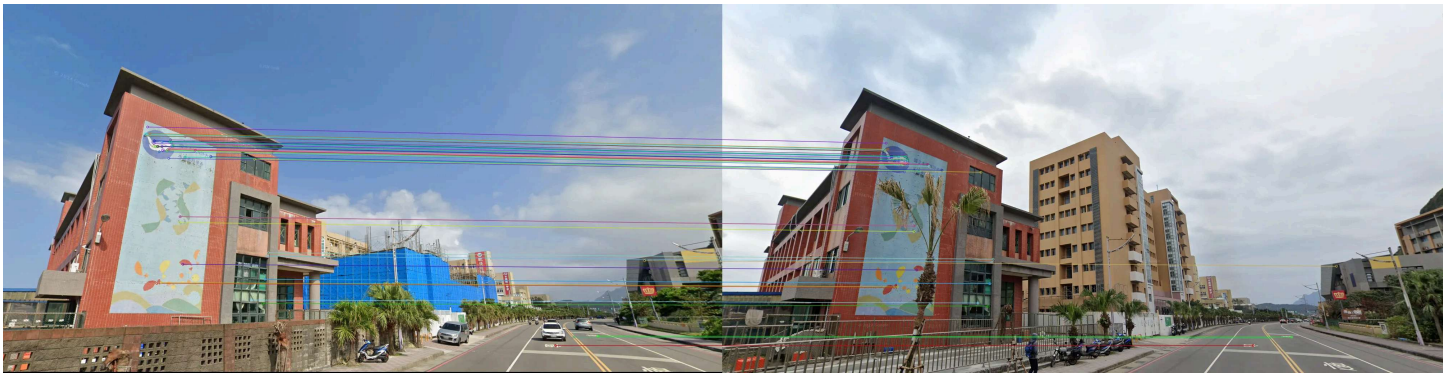
使用 cv2.findHomography 計算 Homography 矩陣時，指定使用 RANSAC 方法。隨機地從 good_matches 中選取一小部分樣本點（計算 Homography 至少需要 4 對點）基於這些樣本計算一個候選的 Homography 矩陣。用這個候選矩陣去測試**所有** good_matches，統計有多少匹配點對符合這個變換（它們之間的誤差在一個可接受的閾值內），這些符合的點稱為**內點 (inliers)**。重複以上步驟多次。最終選取那個能夠獲得最多內點的 Homography 矩陣作為結果。

RANSAC 能夠有效地從包含大量**外點 (outliers)**（錯誤的匹配點或不符合單一平面變換的點）的數據中，找出最優的幾何模型。使得即使在 Lowe's Ratio Test 之後仍存在一些錯誤匹配的情況下，也能計算出準確的 Homography。

MAGSAC++ (cv2.USAC_MAGSAC): 目前被認為是 RANSAC 的一個非常優秀的 SOTA 變種。通常比標準 RANSAC 更快收斂，對高比例的外點更穩健，並且能提供更準確的模型估計。

```
MIN_GOOD_MATCH_COUNT = 10       # Ratio Test 後至少需要多少個匹配點才能繼續
# 計算 Homography 矩陣 (使用 MAGSAC++) ---
if len(good_matches) >= MIN_GOOD_MATCH_COUNT:
    pts1 = np.float32([ kp1[m.queryIdx].pt for m in good_matches ]).reshape(-1,
```

```
    pts2 = np.float32([ kp2[m.trainIdx].pt for m in good_matches ]).reshape(-1,

    H = None
    mask = None
    homography_method_str = ""
    find_homography_success = False

    H, mask = cv2.findHomography(pts1, pts2, cv2.USAC_MAGSAC, RANSAC_REPROJ_THR
    if H is not None: find_homography_success = True

    if find_homography_success and mask is not None:
        inlier_count = np.sum(mask)

        if inlier_count >= MIN_INLIER_COUNT:
            matchesMask = mask.ravel().tolist()

            img_ransac_inliers = cv2.drawMatches(img1_color, kp1, img2_color, k
            plt.figure(figsize=(20, 10)); plt.imshow(cv2.cvtColor(img_ransac_ir
```



圖像變換 (Image Transformation)

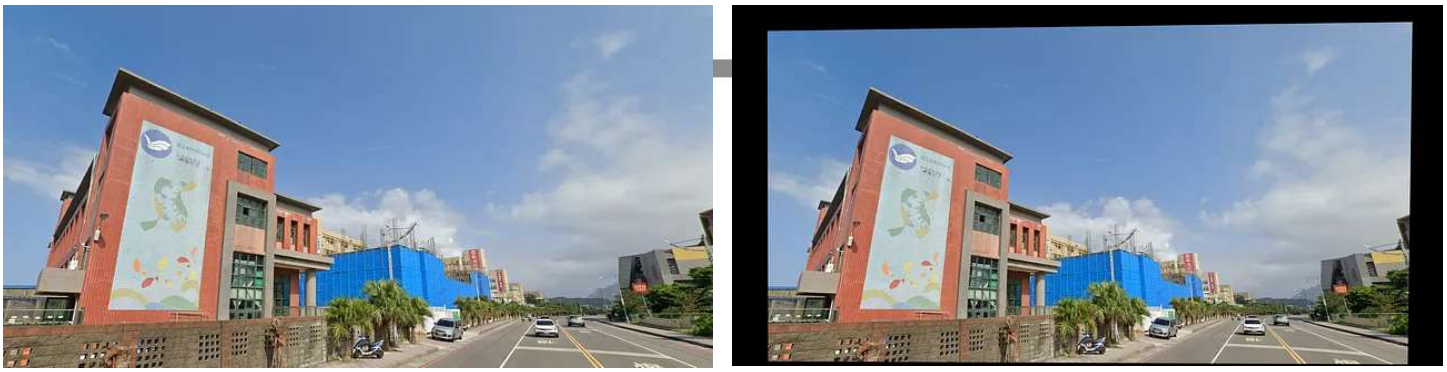
透視變換 (cv2.warpPerspective):

利用上一步計算出的 Homography 矩陣 M，將第二張圖片 (img2_color) 進行幾何變換，將其像素重新映射到新的位置，使其視角與第一張參考圖片 (img1_color) 對齊。輸出的 img2_aligned 就是對齊後的圖像。

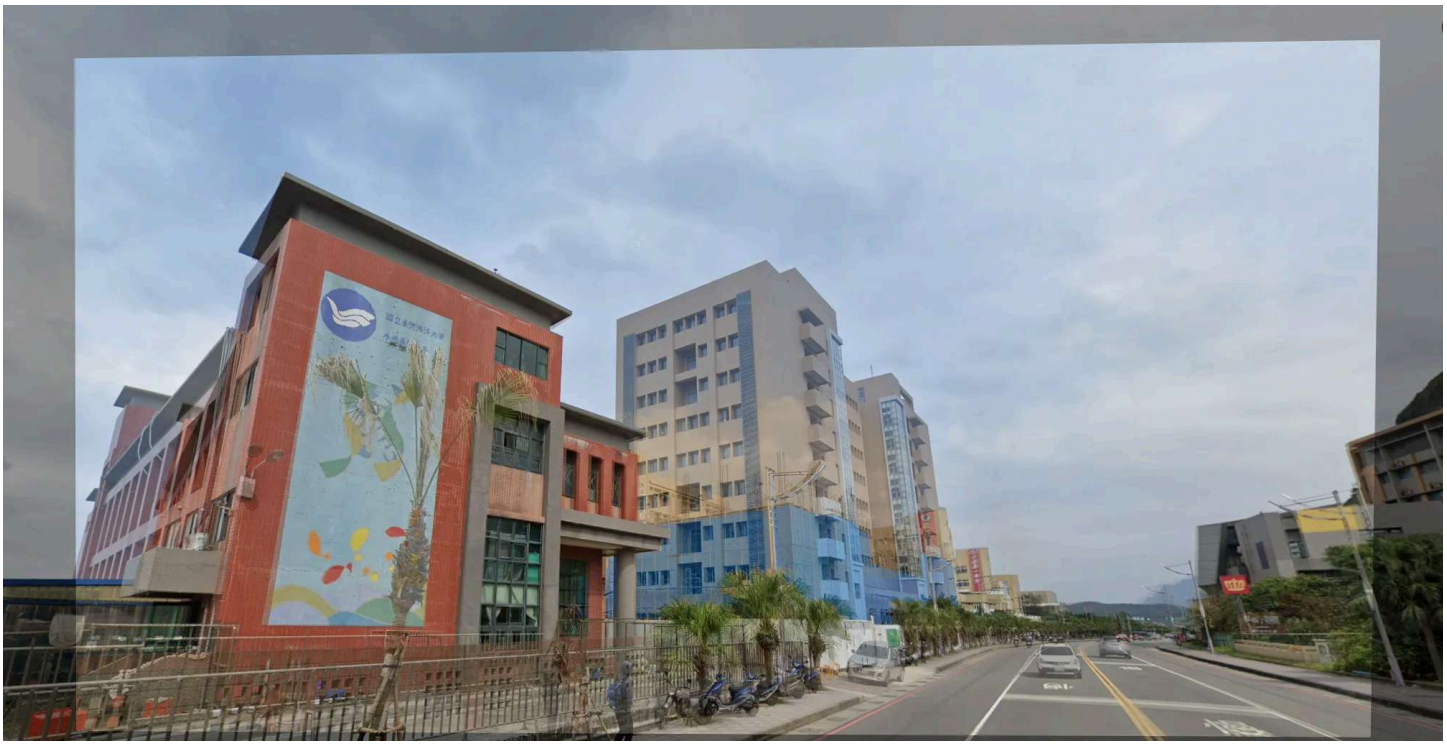
程式:

```
aligned_img1 = cv2.warpPerspective(img1_color, H, (w2, h2), flags=cv2.INTER_CUB
```

```
blended_image = cv2.addWeighted(img2_color, 0.6, aligned_img1, 0.4, 0.0)
```



左:原圖, 右:對齊後



blended_result

Image Alignment Sift Python Opencv

Edit profile