```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img1 = cv2.imread("paisaje1.jpg", cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread("paisaje2.jpg", cv2.IMREAD_GRAYSCALE)
if img1 is None or img2 is None:
    print('No se pudieron abrir las imágenes')
    exit(0)
sift = cv2.SIFT_create()
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints2, descriptors2 = sift.detectAndCompute(img2, None)

print("Number of keypoints for paisaje1:",len(keypoints1))
print("Number of keypoints for paisaje2:",len(keypoints2))
```

```
Number of keypoints for paisaje1: 6540
Number of keypoints for paisaje2: 6888
```

The descriptors are vectors that captures information about the 16x16 neighbourhood around the keypoint. This neighbourhood is then divided into 16 blocks, for each block an orientation histogram with 8 bins is created. This histogram gives information about the direction of the gradient.

```
orb = cv2.ORB_create(nfeatures=7000)
keypoints1_orb, descriptors1_orb = orb.detectAndCompute(img1, None)
keypoints2_orb, descriptors2_orb = orb.detectAndCompute(img2, None)

print("Number of keypoints for paisaje1:",len(keypoints1_orb))
print("Number of keypoints for paisaje2:",len(keypoints2_orb))
```

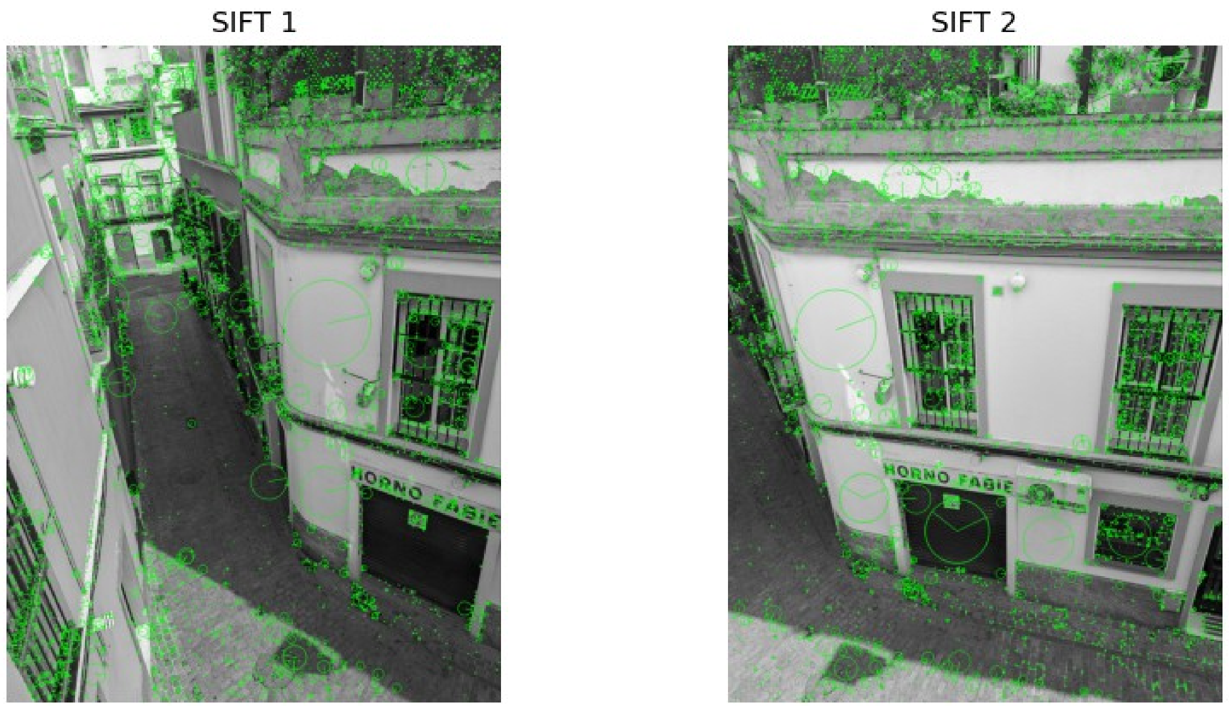
```
Number of keypoints for paisaje1: 7000
Number of keypoints for paisaje2: 7000
```

ORB has a max number of keypoints to be found (default is 500). But i noticed that if I increase this value, ORB tries to reach that number. So increasing this maximum, the keypoints found always increase (even if it doesn't reach the maximum).

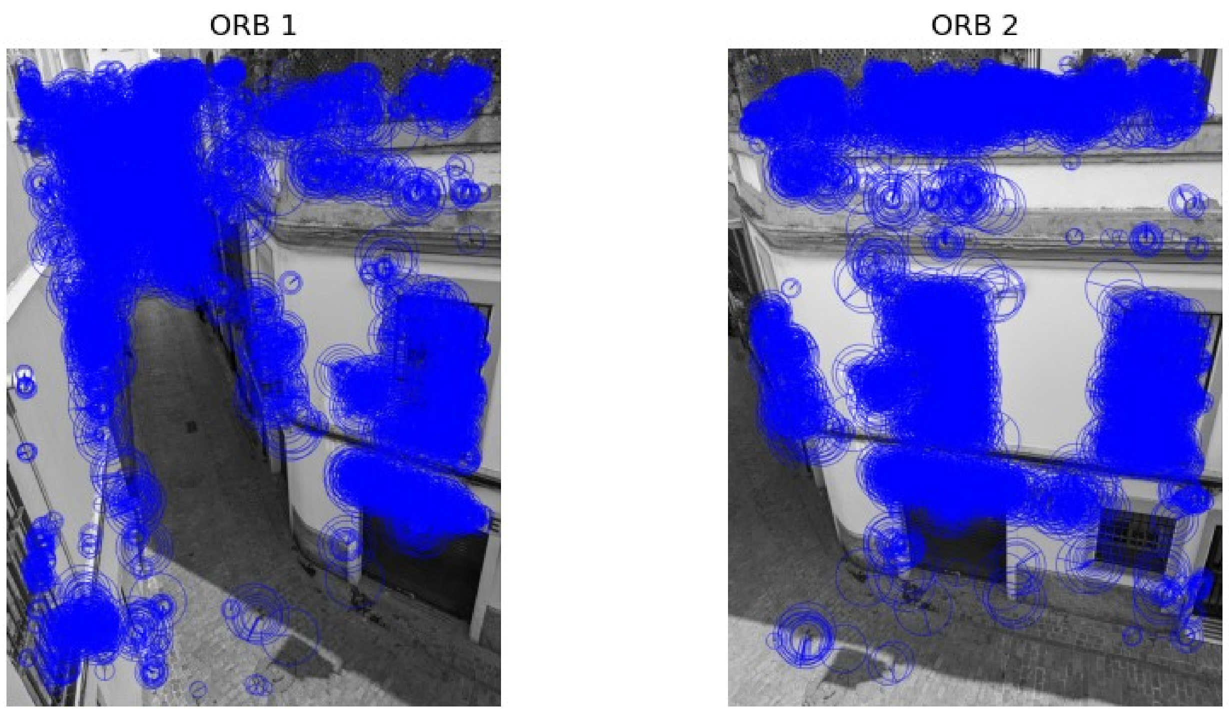
I decided to cap the number of features to 7000, looking at the number of features found with SIFT.

```
image_sift1 = cv2.drawKeypoints(img1, keypoints1, None,
color=(0,255,0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
image_sift2 = cv2.drawKeypoints(img2, keypoints2, None,
color=(0,255,0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

image_orb1 = cv2.drawKeypoints(img1, keypoints1_orb, None,
color=(0,0,255), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
image_orb2 = cv2.drawKeypoints(img2, keypoints2_orb, None,
color=(0,0,255), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

```
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image_sift1,cmap='gray')
plt.title('SIFT 1')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(image_sift2,cmap='gray')
plt.title('SIFT 2')
plt.axis('off')

plt.show()
```



```
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image_orb1,cmap='gray')
plt.title('ORB 1')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(image_orb2,cmap='gray')
plt.title('ORB 2')
plt.axis('off')

plt.show()
```



Now it's possible to find the matching of keypoints in the two images, using Brute Force matching With SIFT descriptors.

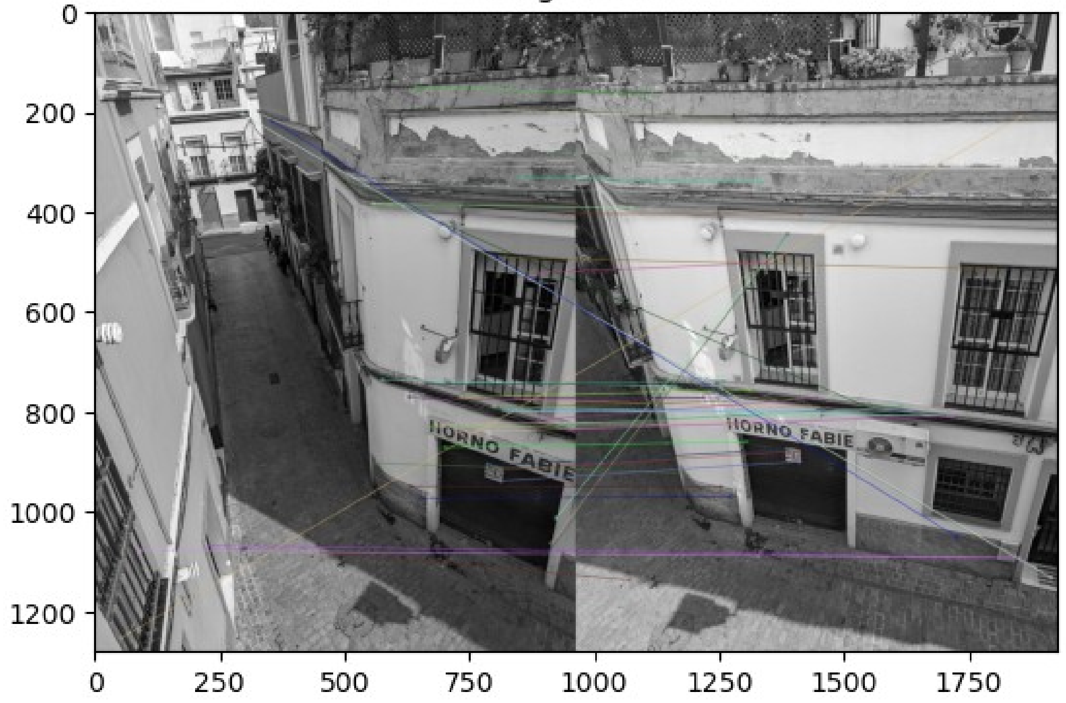
I couldn't find cv2.DescriptorMatcher_create documentation, so I use the method suggested by openCV

```
matcher = cv2.BFMatcher()
matches = matcher.match(descriptors1, descriptors2)
```

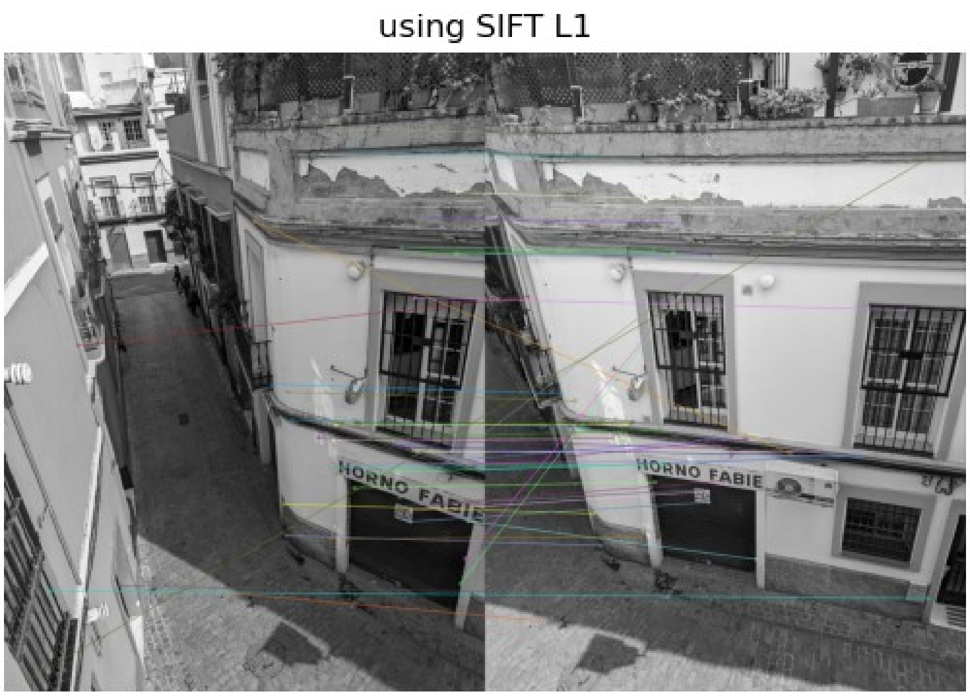
By default, the distance used is NORM_L2, the euclidean distance. According to openCV documentation, NORM_L2 is good for SIFT, since the descriptors of the algorithm are vectors of float. NORM_L1, manhattan distance, is also an option.

```
matcher2 = cv2.BFMatcher(cv2.NORM_L1)
matches2 = matcher2.match(descriptors1, descriptors2)

matches = sorted(matches, key = lambda x:x.distance)
matches2 = sorted(matches2, key = lambda x:x.distance)

img_matches =
cv2.drawMatches(img1,keypoints1,img2,keypoints2,matches[:50],None,flag
s=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(img_matches)
plt.title("using SIFT L2")
plt.show()
```

using SIFT L2



```
img_matches2 =
cv2.drawMatches(img1,keypoints1,img2,keypoints2,matches2[:50],None,fla
gs=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(img_matches2)
plt.title("using SIFT L1")
plt.axis('off')
plt.show()
```



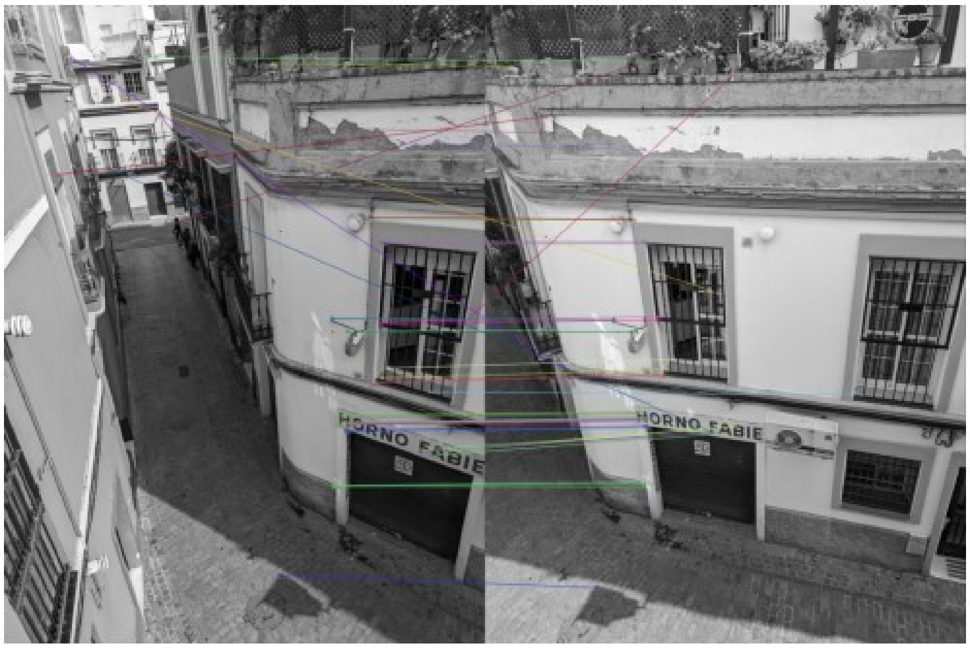
Drawing the matches, I sorted the matches by distance and drawn just the first 50, for a better visibility.

I can't see a big difference between the 2 different distances used. In general there are a lot of positive matching but also some errors, like matching keypoints that are not in both images.

Same procedure with ORB.

```
matcher_orb = cv2.BFMatcher(cv2.NORM_HAMMING,crossCheck=True)
matches_orb = matcher.match(descriptors1_orb, descriptors2_orb)
matches_orb = sorted(matches_orb, key = lambda x:x.distance)

img_matches_orb =
cv2.drawMatches(img1,keypoints1_orb,img2,keypoints2_orb,matches_orb[:5
0],None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(img_matches_orb)
plt.title("using ORB")
plt.axis('off')
plt.show()
```

using ORB



For bruteforce matching with ORB, it's better to use NORM_HAMMING, that is the Hamming distance. This is because the ORB descriptors are binary vectors.

Also with ORB we can find some matches around the common area but seems less than SIFT. There are a lot of matches in keypoints that are not in both images.

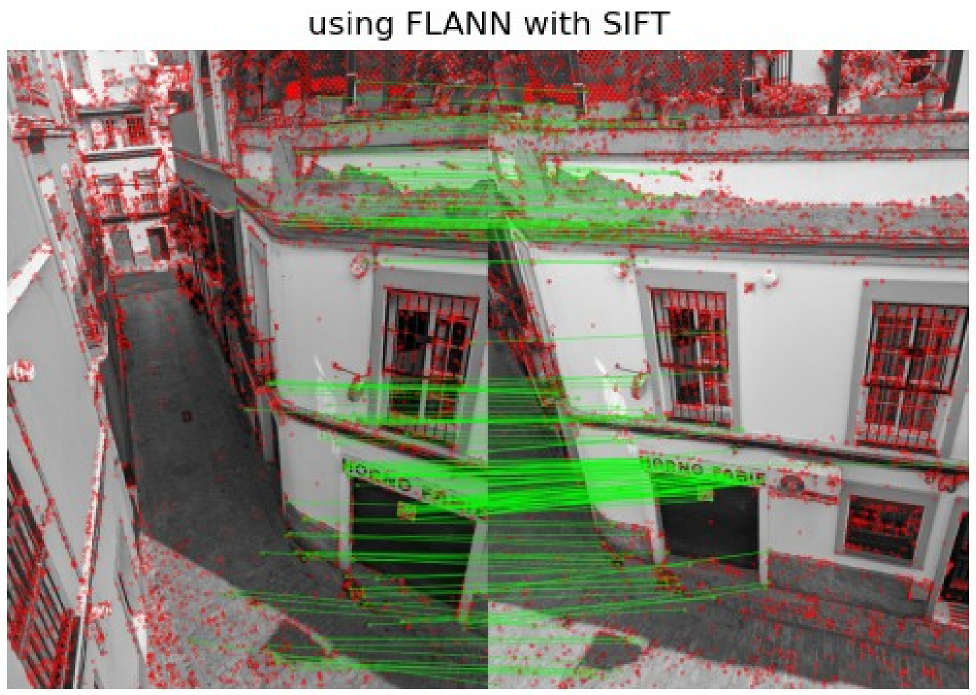
Overall SIFT seems to have better results.

Using FLANN with SIFT

```
# initialization parameters
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params,search_params)
```

As suggested in the documentation, I define a mask for drawing just "good matches". Since knnMatch with k=2 founds the 2 closest point for each descriptors, a match can be defined good when these 2 points are close.

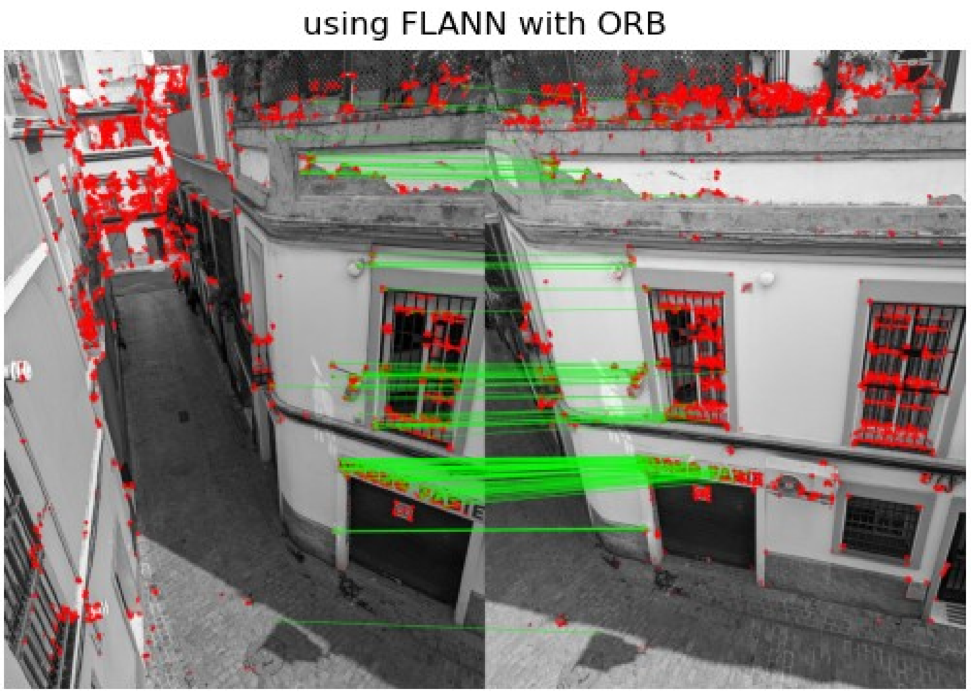
```
matches_flann = flann.knnMatch(descriptors1,descriptors2,k=2)
matchesMask = [[0,0] for i in range(len(matches_flann))]
for i,(m,n) in enumerate(matches_flann):
    if m.distance < 0.5*n.distance:
        matchesMask[i]=[1,0]
draw_params = dict(matchColor = (0,255,0),
                   singlePointColor = (255,0,0),
                   matchesMask = matchesMask,
                   flags = cv2.DrawMatchesFlags_DEFAULT)

img_flann =
cv2.drawMatchesKnn(img1,keypoints1,img2,keypoints2,matches_flann,None,
**draw_params)
plt.imshow(img_flann)
plt.title("using FLANN with SIFT")
plt.axis('off')
plt.show()
```



FLANN with ORB

```
FLANN_INDEX_LSH = 6
index_params2= dict(algorithm = FLANN_INDEX_LSH,
                   table_number = 6,
                   key_size = 12,
                   multi_probe_level = 1)
flann2 = cv2.FlannBasedMatcher(index_params2,search_params)

matches_flann2 =
flann2.knnMatch(descriptors1_orb,descriptors2_orb,k=2)
matchesMask2 = [[0,0] for i in range(len(matches_flann2))]

for i,(m,n) in enumerate(matches_flann2):
    if m.distance < 0.6*n.distance:
        matchesMask2[i]=[1,0]
```

```
draw_params2 = dict(matchColor = (0,255,0),
                    singlePointColor = (255,0,0),
                    matchesMask = matchesMask2,
                    flags = cv2.DrawMatchesFlags_DEFAULT)

img_flann2 =
cv2.drawMatchesKnn(img1,keypoints1_orb,img2,keypoints2_orb,matches_fla
nn2,None,**draw_params2)
plt.imshow(img_flann2)
plt.title("using FLANN with ORB")
plt.axis('off')
plt.show()
```



Since I used a mask, as suggested in the documentation, only certain keypoints are shown ("good matches").

I would say that with SIFT the result are quite nice, because it found a lot of good matches in different areas of the image. ORB seems to find more concentrates matches in the same area, and less matches overall.

```
# Warp img2 to img1 using the homography matrix H
def warpImages(img1, img2, H):
    rows1, cols1 = img1.shape[:2]
    rows2, cols2 = img2.shape[:2]
    list_of_points_1 = np.float32([[0,0], [0,rows1], [cols1,rows1],
    [cols1,0]]).reshape(-1,1,2)
    temp_points = np.float32([[0,0], [0,rows2], [cols2,rows2],
[cols2,0]]).reshape(-
```

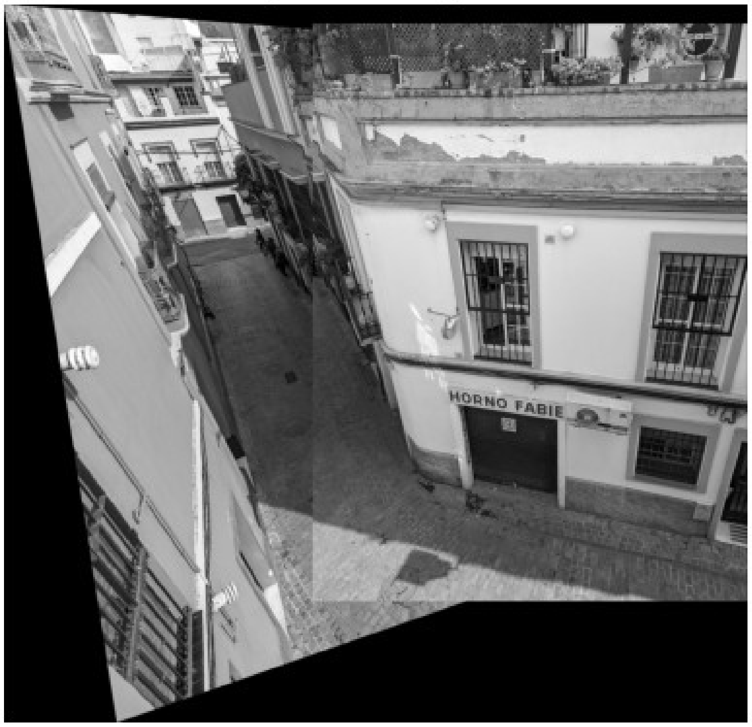
```
    1,1,2)
    list_of_points_2 = cv2.perspectiveTransform(temp_points, H)
    list_of_points = np.concatenate((list_of_points_1,
list_of_points_2), axis=0)
    [x_min, y_min] = np.int32(list_of_points.min(axis=0).ravel() -
0.5)
    [x_max, y_max] = np.int32(list_of_points.max(axis=0).ravel() +
0.5)
    translation_dist = [-x_min, -y_min]
    H_translation = np.array([[1, 0, translation_dist[0]], [0, 1,
translation_dist[1]],
    [0,0,1]])
    output_img = cv2.warpPerspective(img2, H_translation.dot(H),
(x_max-x_min,
    y_max-y_min))
    output_img[translation_dist[1]:rows1+translation_dist[1],
    translation_dist[0]:cols1+translation_dist[0]] = img1
    return output_img
```

I use the FLANN matching with SIFT here, trying with least squared method and RANSAC for the homography.

For the threshold, I already did the good matches test ratio before trying with different values, using 0.5 as a threshold resulted to be the best choice in finding good matches.

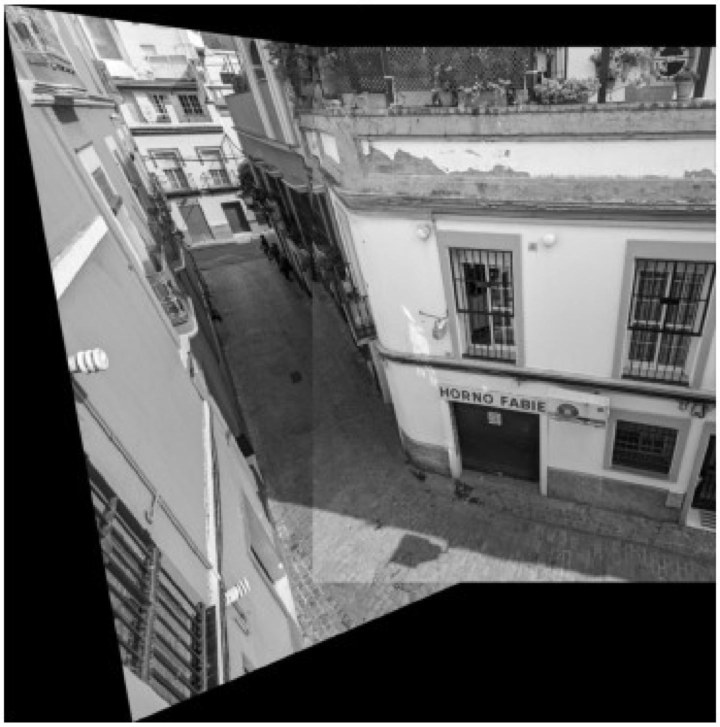
```
good_matches = []
for m1, m2 in matches_flann:
    if m1.distance < 0.5 * m2.distance:
        good_matches.append(m1)

if len(good_matches) > 4:
    src_pts = np.float32([keypoints1[good_match.queryIdx].pt for
good_match in good_matches]).reshape(-1, 1, 2)
    dst_pts = np.float32([keypoints2[good_match.trainIdx].pt for
good_match in good_matches]).reshape(-1, 1, 2)
    M, mask = cv2.findHomography(src_pts, dst_pts, 0, 5.0)
    result = warpImages(img2, img1, M)
    plt.imshow(result,cmap="gray")
    plt.title("Stitched output least squared method")
    plt.axis('off')
    plt.show()
else:
    print("No hay suficientes correspondencias entre las dos
imágenes")
    print("Se encontraron sólo %d. Se necesitan al menos %d." %
(len(good_matches), 4))
```

Stitched output least squared method



```
good_matches = []
for m1, m2 in matches_flann:
    if m1.distance < 0.5 * m2.distance:
        good_matches.append(m1)

if len(good_matches) > 4:
    src_pts = np.float32([keypoints1[good_match.queryIdx].pt for
good_match in good_matches]).reshape(-1, 1, 2)
    dst_pts = np.float32([keypoints2[good_match.trainIdx].pt for
good_match in good_matches]).reshape(-1, 1, 2)
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    result_ransac = warpImages(img2, img1, M)
    plt.imshow(result_ransac,cmap="gray")
    plt.title("Stitched output RANSAC")
    plt.axis('off')
    plt.show()
else:
    print("No hay suficientes correspondencias entre las dos
imágenes")
    print("Se encontraron sólo %d. Se necesitan al menos %d." %
(len(good_matches), 4))
```

Stitched output RANSAC



A few considerations:

- Using least squared method, the result is very sensible to the threshold, we obtain good result only with threshold from 0.3 to 0.5.
- With RANSAC, the result seems the same with every threshold.
- Using 0.5 as a threshold, the 2 method give us the same result apparently.

I would say that the results are quite good, the images are not perfectly aligned but that's normal, we have obteined a panoramic image.

Sources:

- https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html
- https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
- https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html
- https://docs.opencv.org/3.4/d2/de8/group__core__array.html#ggad12cefbcb5291cf958a85b4b67b6149fa7bacbe84d400336a8f26297d8e80e3a2
- https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html
- https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html