Computer Vision HW2 report

Author: 109550093 黃得誠

I. Implementation

In this homework, I learnt how to stitch images using SIFT features and homography matrix with RANSAC. I implemented the algorithms using Python and OpenCV. I started from stitching 2 images. I used the right image as base, project left image onto right image coordinates. For baseline image cases, I stitch the rightmost first, from right to left, 2 images at a time. For the bonus images, I stitched m3 and m4, m1 and m2 first, then stitch the two stitched image together to form the final image.

1. SIFT

For SIFT, I simply followed the spec, using OpenCV functions to extract SIFT features. “kp” is OpenCV keypoints, which stores the point information. “descript” is an array, which stores the gradient information.

```
# Step 1 SIFT
print("Step 1 SIFT ...")
kp_left, des_left = SIFT(img_left_g)
kp_right, des_right = SIFT(img_right_g)
```

```
24  # SIFT
25  def SIFT(img):
26      SIFT_Detector = cv2.SIFT_create()
27      kp, descript = SIFT_Detector.detectAndCompute(img, None)
28      return kp, descript
```

2. Feature Matching-kNN and Lowe's Ratio test

In step 2, I apply kNN algorithm (in a brutal Force way) and Lowe's Ratio test to find the good matches. I used OpenCV.BFMatcher() to debug. Since I found that OpenCV uses a class called dmatch to store the match information, I created dictionaries to store the information in a similar way.

```
440     # Step 2 Feature matching
441     print("Step 2 Feature Matching ...")
442     good, good1 = matcher(kp_left, des_left, kp_right, des_right, 0.7)
443     matches = get_matches(good1, kp_left, kp_right)
```

```
36  def matcher(kp1, des1, kp2, des2, threshold):
```

To find the least and second least distance, I created two dictionaries, dmatch and dmatch1 to store. I used np.linalg.norm to calculate the distance faster.

```
50      # Brutal Force
51      matches1 = []
52      for i in tqdm(range(len(kp1))):
53          dmatch = {"distance":1e7, "queryIdx":0, "trainIdx":0} # smallest
54          dmatch1 = {"distance":1e7, "queryIdx":0, "trainIdx":0} # 2nd smallest
55          v1 = des1[i,:]
56          for j in range(len(kp2)):
57              v2 = des2[j,:]
58              distance = 0
59              distance = np.linalg.norm(v1-v2)
60              if distance < dmatch["distance"]:
61                  dmatch1["distance"] = dmatch["distance"]
62                  dmatch1["queryIdx"] = dmatch["queryIdx"]
63                  dmatch1["trainIdx"] = dmatch["queryIdx"]
64                  dmatch["distance"] = distance
65                  dmatch["queryIdx"] = i
66                  dmatch["trainIdx"] = j
67              elif distance<dmatch1["distance"]:
68                  dmatch1["distance"] = distance
69                  dmatch1["queryIdx"] = i
70                  dmatch1["trainIdx"] = j
71          matches1.append((dmatch, dmatch1))
```

Finally, the ratio test.

```
73      good1 = []
74      # Apply Lowe's ratio test
75      for m,n in matches1:
76          # m, n are DMatch
77          if m["distance"] < threshold*n["distance"]:
78              good1.append([m])
79
80      return good, good1
```

I stored the matches information in np.array "matches" for later use.

```
83  def get_matches(good, kp1, kp2):
84      '''
85      out: matches [[x1, y1, x1', y1'], ....., [xn, yn, xn', yn']]
86      '''
87      matches = []
88      for pair in good:
89          matches.append(list(kp1[pair[0]["queryIdx"]].pt + kp2[pair[0]["trainIdx"]].pt))
90          # matches.append(list(kp1[pair[0].queryIdx].pt + kp2[pair[0].trainIdx].pt))
91      matches = np.array(matches)
92      return matches
```

3. Homography matrix with RANSAC

For RANSAC, I followed the pseudocode in the spec. I decided the iterations and the error threshold by testing serval times. I didn't applied a stopping criterion.

```
452         # Step 3 Homography
453         print("Step 3 RANSC ...")
454         inliers, H = ransac(matches, 0.4, 8000)
```

```
149     def ransac(matches, threshold, iters):
150         num_best_inliers = 0
151         best_H = np.zeros((3, 3))
152         for i in tqdm(range(iters)):
153             points = random_point(matches)
154             H = homography(points)
155 
156             #  avoid dividing by zero
157             if np.linalg.matrix_rank(H) < 3:
158                 continue
159 
160             errors = get_error(matches, H)
161             idx = np.where(errors < threshold)[0]
162             inliers = matches[idx]
163 
164             num_inliers = len(inliers)
165             if num_inliers > num_best_inliers:
166                 best_inliers = inliers.copy()
167                 num_best_inliers = num_inliers
168                 best_H = H.copy()
169 
170         print("inliers/matches: {}/{}".format(num_best_inliers, len(matches)))
171         return best_inliers, best_H
```

random_point and get_error are some helper functions that I created.

```
127 # Helper functions of RANSEC
128 def random_point(matches, k=4):
129     # Get k pairs from the matches (Used in RANSEC)
130     idx = random.sample(range(len(matches)), k)
131     point = [matches[i] for i in idx ]
132     return np.array(point)
133 
134 def get_error(matches, H):
135     # matches: [[x1, y1, x1', y1'], ..., [xn, yn, xn', yn']]
136     num_points = len(matches)
137     all_p1 = np.concatenate((matches[:, 0:2], np.ones((num_points, 1))), axis=1) # change (x1, y1) to (x1, y1, 1
138     all_p2 = matches[:, 2:4]
139     estimate_p2 = np.zeros((num_points, 2))
140     for i in range(num_points):
141         temp = np.dot(H, all_p1[i])
142         estimate_p2[i] = (temp/temp[2])[0:2] # set index 2 to 1 and slice the index 0, 1
143     # Compute error
144     errors = np.linalg.norm(all_p2 - estimate_p2 , axis=1) ** 2
145 
146     return errors
```

Finally, the homography matrix.

```python
# Step 3 Homography
def homography(pairs):
    # pairs: [[(x1, y1), (x1', y1')],[(x2, y2), (x2', y2')] [(x3, y3), (x3', y3')], [(x4, y4), (x4', y4')]]
    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, p1[0], p1[1], p1[2], -p2[1]*p1[0], -p2[1]*p1[1], -p2[1]*p1[2]]
        row2 = [p1[0], p1[1], p1[2], 0, 0, 0, -p2[0]*p1[0], -p2[0]*p1[1], -p2[0]*p1[2]]
        rows.append(row1)
        rows.append(row2)
    rows = np.array(rows)     # A in spec, rows.shape is (8, 9)
    U, s, V = np.linalg.svd(rows)
    H = V[-1].reshape(3, 3)   # use the last vector since np.linalg.svd has vector sorted in descending order
    H = H/H[2, 2] # normalize h33 to 1
    return H
```

I followed this page (with a minus sign) in the spec to calculate my homograpy matrix.

3.Homography

$$A = U\Sigma V^T$$

- Using SVD decomposition to find Least Squares error solution of Ah = 0
- the solution = eigenvector of A^TA associated with the smallest eigenvalue (V stores the eigenvector of A^TA ,Σ stores the singular value (root of eigen value))
  find the smallest number in Σ and H = corresponding vector in V^T
- Remember to normalize h33 to 1

$$\begin{aligned}
\mathbf{h} &= (H_{11}, H_{12}, H_{13}, H_{21}, H_{22}, H_{23}, H_{31}, H_{32}, H_{33})^T \\
\mathbf{a}_x &= (-x_1, -y_1, -1, 0, 0, 0, x_2'x_1, x_2'y_1, x_2')^T \\
\mathbf{a}_y &= (0, 0, 0, -x_1, -y_1, -1, y_2'x_1, y_2'y_1, y_2')^T .
\end{aligned}$$

$$A = \begin{pmatrix} \mathbf{a}_{x1}^T \\ \mathbf{a}_{y1}^T \\ \vdots \\ \mathbf{a}_{xN}^T \\ \mathbf{a}_{yN}^T \end{pmatrix}.$$

You can multiply a minus to match the form in previous slide

A is a 9 by 9 matrix
(It's similar to A in previous slide)

Reference :

By the way, I think that A is 8x9 (since there should be even rows).
I used np.linalg.svd to solve the Least squares problem. Since the function give us eigenvalues in a descending order, I get my H through the last row.

Returns:

u : *{(..., M, M), (..., M, K)} array*

Unitary array(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

s : *(..., K) array*

Vector(s) with the singular values, within each vector sorted in descending order. The first `a.ndim - 2` dimensions have the same size as those of the input *a*.

vh : *{(..., N, N), (..., K, N)} array*

Unitary array(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

4. Image Stitching

Last but not least, the image stitching part. I followed the spec as well.

```
457     # Step 4 Stitch Image
458     print("Step 4 stiching image ...")
459     img_stitched = stitch_img(img_left, img_right, H)
```

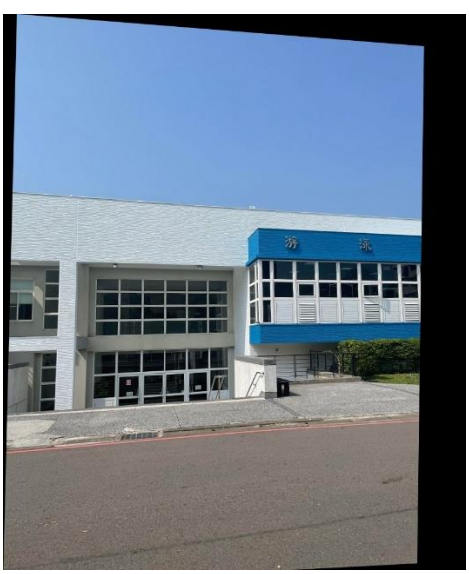
First, convert the image to double for further multiplication.

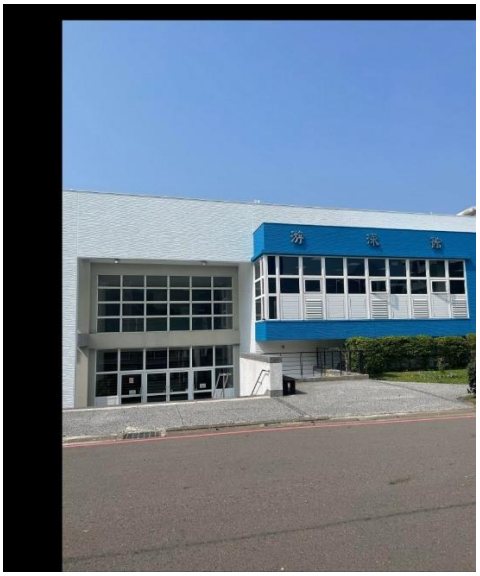
```
343 def stitch_img(left, right, H):
344 
345     # Convert to double and normalize. Avoid noise.
346     left = cv2.normalize(left.astype('float'), None,
347                          0.0, 1.0, cv2.NORM_MINMAX)
348     # Convert to double and normalize.
349     right = cv2.normalize(right.astype('float'), None,
350                          0.0, 1.0, cv2.NORM_MINMAX)
```

Second, calculate the new corners. Calculate the new size and the affine matrix mentioned in the spec. Then do matrix multiplications by cv2.warpPerspective.

```
352     # left image
353     height_l, width_l, channel_l = left.shape
354     corners = [[0, 0, 1], [width_l, 0, 1], [width_l, height_l, 1], [0, height_l, 1]]
355     corners_new = [np.dot(H, corner) for corner in corners]
356     corners_new = np.array(corners_new).T
357     x_news = corners_new[0] / corners_new[2]
358     y_news = corners_new[1] / corners_new[2]
359     y_min = min(y_news)
360     x_min = min(x_news)
361     y_min = min(y_min, 0)
362     x_min = min(x_min, 0)
363 
364     translation_mat = np.array([[1, 0, -x_min], [0, 1, -y_min], [0, 0, 1]])
```

Finally, concatenate the two images.





baseline/m4 and baseline/m5

For concatenating, I used blender method. I applied “linear blending” on the baseline images. Linear blending means that in the overlap area, I give each pixel different weights, if the pixel is closer to the left picture, the left picture weight will be higher. I eliminated some strange boundaries after applying blender method.

With and without blender.

I also wrote a removeBlackBorder function to eliminate the black border by going through the pixels.

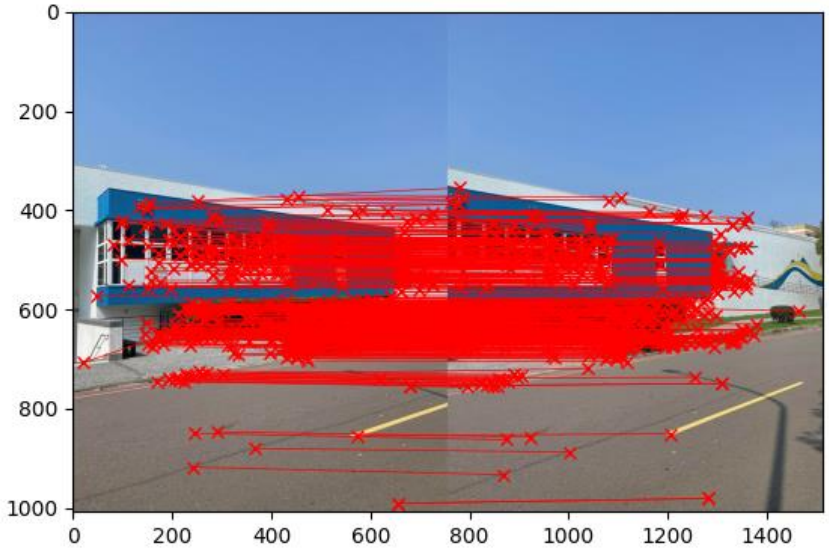
By the way, in the baseline images, to make the result looks better, I cut the top of the image to avoid this kind of result happens.

However, I didn’t do it in the bonus, since the way that bonus images have taken is not in the same order. I change the stitching order instead in the bonus, as I mentioned earlier.

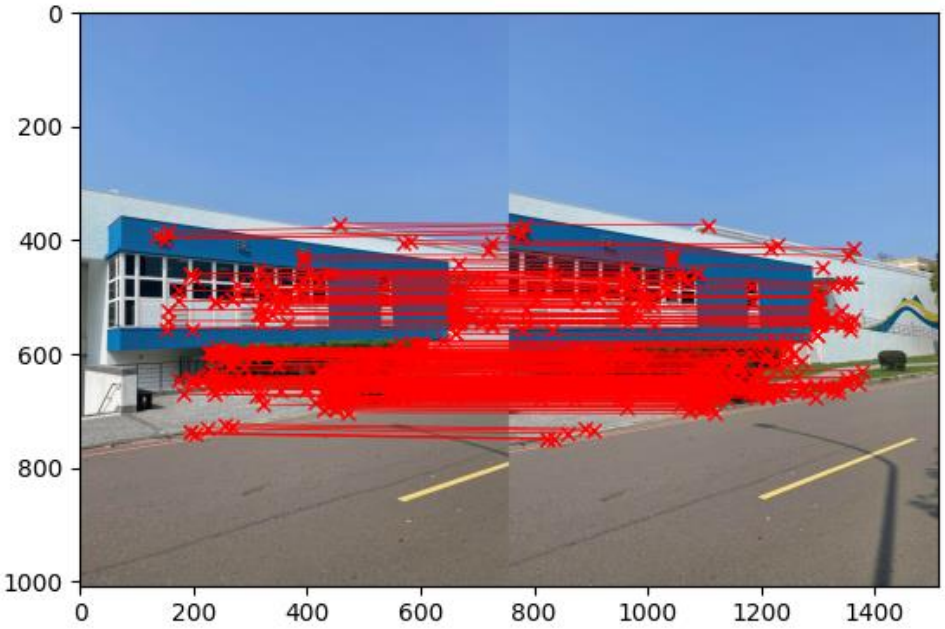
This is the end of the introduction of my implementation. Here are some figures that could show the results of each step.

SIFT:

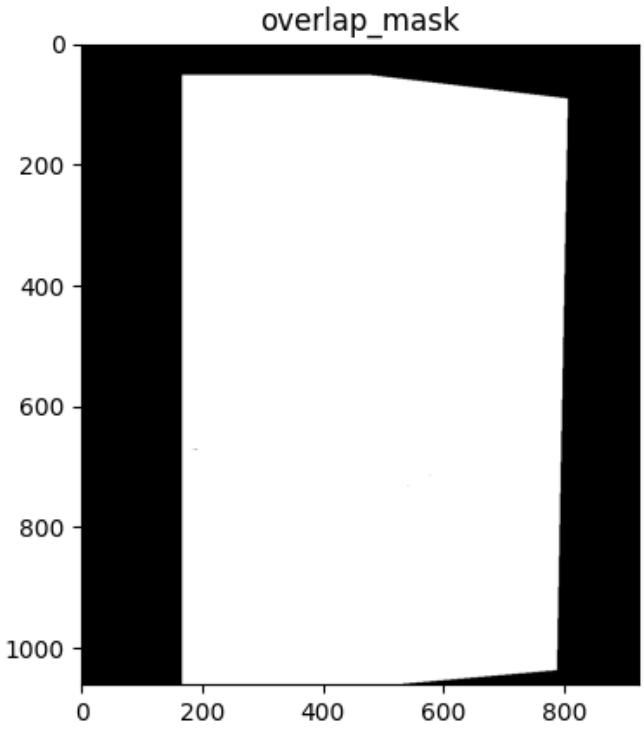
Feature matching



RANSAC



Mask that blending method used.



II. Result of 2 images

1. Baseline

2. Bonus

III. Stitching more images

1. Baseline

2. Bonus

IV. Conclusion

In this homework, I implemented an image stitching method that have been taught in class. Although taught in class, I still met some problems when implementing.

The first problem is the boundary problem.

The alias problem and inverse mapping came to my mind when I met this problem, but I didn't use the inverse mapping to cope with it. Because after blending, the problem seems to be solved.

The second problem occurred when I’m stitching the bonus images.

There are still some boundaries that can be clearly seen after linear blending. I tried “linear blending with constant width” instead, which only blend constant width instead of all the overlapped parts.

It kind of solve some part of the problem but new boundaries occurred too. I think I need to spend more time to test with the constant to have better results, which I didn't continued finally. Since stitching 4 pictures could somehow cover those boundaries.

Last thing that I discovered is that linear blending with constant width can solve another problem, called ghost.

The ghost problem means that there are some scene that are doubled, due to the blending process. Linear blending with constant width doesn't blend that much so the ghost problem would not occur.

Finally, I learnt a lot in this homework.