

# Homework 3 Automatic Panoramic Image Stitching

## Introduction

The goal of this assignment is to implement Automatic Panoramic Image Stitching, aiming to merge two images into a single panorama. This process includes feature point detection and matching, calculating the homography matrix using RANSAC, and image warping and blending. This technique has extensive applications in the field of image processing, such as generating street view images and displaying tourist attractions.

## Implementation procedure

### 1. Interest points detection & feature description by SIFT or MSER

```
# step 1
if feature == "SIFT":
    print("start run SIFT")
    detector = cv2.SIFT_create()

    print("processing image1")
    keypoints1, descriptors1 = detector.detectAndCompute(img1, None)

    print("processing image2")
    keypoints2, descriptors2 = detector.detectAndCompute(img2, None)

    print("processing complete")

elif feature == "MSER":
    print("start run MSER")
    msr = cv2.MSER_create(min_area=20, max_area=14400)
    sift = cv2.SIFT_create()

    print("processing image1")
    keypoints1 = msr.detect(img1)
    keypoints1, descriptors1 = sift.compute(img1, keypoints1)

    print("processing image2")
    keypoints2 = msr.detect(img2)
    keypoints2, descriptors2 = sift.compute(img2, keypoints2)

    print("processing complete")

else:
    raise ValueError(f"暫不支持{feature}方法，請選擇 'SIFT' or 'MSER'。")
```

For the SIFT part, we use the built-in SIFT model in OpenCV (cv2) to detect and compute the features of two images.

As for MSER, we use the built-in MSER model in OpenCV to detect features. However, since MSER only provides detection without feature computation, we still rely on SIFT for the computation. Thus, we use OpenCV's SIFT model here as well to compute the features.

## 2. Feature matching by above features

```
def ratio_distance(descriptor1, descriptor2):  
    return np.linalg.norm(descriptor1 - descriptor2) / (np.linalg.norm(descriptor1) + np.linalg.norm(descriptor2) + 1e-10)  
  
def match(descriptors1, descriptors2):  
    matches = []  
    tmp = []  
    for li in enumerate(descriptors1):  
        tmp.append(li)  
        feature_size = li[0]  
        print(f'the feature number of descriptors1 is {li[0]}')  
        tmp = []  
    for li in enumerate(descriptors2):  
        tmp.append(li)  
        print(f'the feature number of descriptors2 is {li[0]}')  
        print('start matching feature...')  
        for bar, (i, desc1) in zip(tqdm(range(1, feature_size)), enumerate(descriptors1)):  
            distances = np.array([ratio_distance(desc1, desc2) for desc2 in descriptors2])  
            sorted_indices = np.argsort(distances)[:2]  
            matches.append([cv2.DMatch(_queryIdx=i, _trainIdx=idx, _imgIdx=0, _distance=distances[idx]) for idx in sorted_indices])  
  
    return matches  
  
matches = match(descriptors1, descriptors2)  
good_matches = [m for m, n in matches if m.distance < 0.75 * n.distance]  
print(f'the feature number that is good matching is {len(good_matches)}')
```

First, I replaced the original formula  $|f1-f2|/|f1-f2'|$  with  $(|f1-f2|)/(|f1|+|f2|+1e-10)$ . This approach simplifies the distance calculation and enhances stability.

The original formula used a reference vector  $f2'$  as a standard to comparatively measure the distance between  $f1$  and  $f2$ . In contrast, the new formula uses the sum of the norms  $|f1|+|f2|$  as the denominator for normalization. This change not only simplifies the computation but also removes the dependency on the reference term  $f2'$ .

This normalization method further improves the fairness of comparison: by dividing  $|f1-f2|$  by the total norm  $|f1|+|f2|$ , we effectively reduce the influence of differences in vector magnitudes. This ensures that the distance more accurately reflects the relative differences between the vectors without being affected by their absolute sizes. Additionally, the inclusion of  $1e-10$  in the denominator ensures stability, preventing unreasonable results even when norms approach zero.

For this part, we first pair each descriptor and compute their distances using the revised formula. For each descriptor, we retain only the closest (or most similar) pair as a candidate match. We then perform a ratio test on these matches to obtain the "good matches" by setting a ratio of 0.75. This means the closest distance must be less than 0.75 times the next closest one, effectively filtering out most noise.

## 3. RANSAC to find homography matrix H

This step uses RANSAC to calculate the homography matrix  $H$  between feature points. The steps are as follows:

I. Sample  $S$  correspondences from the feature matching results

First, from the previously computed `good_match` results, randomly select 4 pairs of correspondences. These pairs will be used in subsequent calculations.

```
def sample_correspondences(matches, keypoints1, keypoints2, S=4):
    sampled_matches = random.sample(matches, S)
    pts1 = np.float32([keypoints1[m.queryIdx].pt for m in sampled_matches])
    pts2 = np.float32([keypoints2[m.trainIdx].pt for m in sampled_matches])
    return pts1, pts2
```

II. Compute the homography matrix based on these sampled correspondences  
Then, calculate the corresponding homography matrix using the selected N points. Using the following transformation formulas, we can compute the homography matrix:

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} \bar{x}_d \\ \bar{y}_d \\ \bar{z}_d \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

$$x_d^n = \frac{\bar{x}_d^n}{\bar{z}_d^n} = \frac{h_{11}x_s^n + h_{12}y_s^n + h_{13}}{h_{31}x_s^n + h_{32}y_s^n + h_{33}}$$

$$y_d^n = \frac{\bar{y}_d^n}{\bar{z}_d^n} = \frac{h_{21}x_s^n + h_{22}y_s^n + h_{23}}{h_{31}x_s^n + h_{32}y_s^n + h_{33}}$$

$$x_d^n(h_{31}x_s^n + h_{32}y_s^n + h_{33}) = h_{11}x_s^n + h_{12}y_s^n + h_{13}$$

$$y_d^n(h_{31}x_s^n + h_{32}y_s^n + h_{33}) = h_{21}x_s^n + h_{22}y_s^n + h_{23}$$

$$\begin{bmatrix} x_s^n & y_s^n & 1 & 0 & 0 & 0 & -x_d^n x_s^n - x_d^n y_s^n - x_d^n \\ 0 & 0 & 0 & x_s^n & y_s^n & 1 & -y_d^n x_s^n - y_d^n y_s^n - y_d^n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
def compute_homography(pts1, pts2):
    A = []
    for i in range(4):
        X, Y = pts1[i][0], pts1[i][1]
        u, v = pts2[i][0], pts2[i][1]
        A.append([-X, -Y, -1, 0, 0, 0, X * u, Y * u, u])
        A.append([0, 0, 0, -X, -Y, -1, X * v, Y * v, v])

    A = np.array(A)
    _, _, Vt = np.linalg.svd(A)
    H = Vt[-1].reshape(3, 3)
    return H / H[2,2]
```

III. Check the number of inliers/outliers by a threshold

Next, project the feature points from the first image onto the coordinate system of the second image using HH, and calculate the projection error for each point. If the error is less than the threshold, the point is considered an inlier.

```
def count_inliers(H, keypoints1, keypoints2, matches, threshold=5.0):
    inliers = 0
    for match in matches:
        pt1 = np.array([*keypoints1[match.queryIdx].pt, 1.0])
        pt2 = np.array([*keypoints2[match.trainIdx].pt, 1.0])

        projected_pt1 = H @ pt1
        projected_pt1 /= projected_pt1[2]
        error = np.linalg.norm(projected_pt1[:2] - pt2[:2])
        if error < threshold:
            inliers += 1
    return inliers
```

IV. Iterate for N times and get the best homography matrix with smallest number of outliers  
Repeat the above steps for max\_iters iterations, recording the number of inliers each time.  
Finally, select the homography matrix H with the largest number of inliers as the optimal result.

```
def ransac_homography(matches, keypoints1, keypoints2, threshold=5.0, max_iters=1000):
    best_H = None
    max_inliers = 0

    for _ in range(max_iters):
        pts1, pts2 = sample_correspondences(matches, keypoints1, keypoints2)
        H = compute_homography(pts1, pts2)

        inliers = count_inliers(H, keypoints1, keypoints2, matches, threshold)

        if inliers > max_inliers:
            max_inliers = inliers
            best_H = H

    return best_H
```

4. Warp image to create panoramic image

In this part, we warp and stitch the images into a panoramic image using the homography matrix  $H$ . We implemented a function named `warp_and_stitch` to complete this process, as described below:

**I. Calculating the Warped Positions of Image Corners**

First, we need to determine the four corner points of the first image (`img1`). These corner points are represented by coordinates such as (0, 0) and (`img1.shape[1]`, 0), and are stored in a matrix called `corners_img1`. Next, these corner points are warped to the perspective coordinates of the second image, allowing us to determine the spatial distribution of the stitched image.

We calculate the warped coordinates by converting each corner point into homogeneous coordinates (i.e., appending a 1) and then applying the homography matrix  $H$ . The transformed results are then normalized so that the third homogeneous coordinate remains 1.

**II. Calculating the Size and Translation of the New Image**

The size of the stitched image depends on the range of warping between the two images. To ensure that the stitched image can accommodate both images, we calculate the minimum and maximum values of all the corner points, determining the boundaries of the new image (`x_min`, `y_min`, `x_max`, `y_max`).

Based on the minimum boundary values, we compute a translation matrix  $H^T$  to ensure that the entire stitched result falls within the positive coordinate space. This avoids issues where parts of the warped image would be outside the visible area due to negative coordinates.

**III. Warping and Stitching the Image**

In the warping process, we apply the perspective transformation matrix  $H$  to warp the first image into the new coordinate system and write the warped pixel values into the resulting image.

During implementation, for each pixel, we use the homography matrix to calculate its position in the resulting image and fill in the corresponding RGB values. When handling empty pixels in the resulting image (i.e., places with no corresponding pixel), we designed a simple averaging strategy. This strategy checks the valid pixels in a 3x3 neighborhood around the empty pixel and fills the empty area with their average value. This helps reduce visible seams during the image stitching process.

**IV. Stitching the Second Image**

The final step is to overlay the second image onto the already warped first image. We copy the pixel values of the second image directly into the appropriate position in the resulting image (considering the translation distance). This way, we obtain a complete panoramic image.

**V. Image Blending and Enhancement**

In this part, we also performed a simple blending operation, using the average value of neighboring pixels to fill in the empty regions after stitching. This reduces the boundary effects caused by geometric warping, making the final panoramic image appear more natural.

```

def warp_and_stitch(img1, img2, H):
    h2, w2 = img2.shape[:2]
    corners_img1 = np.float32([[0, 0], [0, img1.shape[0]], [img1.shape[1], img1.shape[0]], [img1.shape[1], 0]]).reshape(-1, 1, 2)

    corners_img1_enpend = np.concatenate((corners_img1, np.ones((4, 1, 1))), axis=2)
    warped_corners_img1 = []
    for i in range(len(corners_img1_enpend)):
        temp = H @ corners_img1_enpend[i].T
        temp /= temp[2,0]
        warped_corners_img1.append(temp[0:2].T)

    corners = np.concatenate((warped_corners_img1, np.float32([[0, 0], [0, h2], [w2, h2], [w2, 0]]).reshape(-1, 1, 2)), axis=0)
    [x_min, y_min] = np.int32(corners.min(axis=0).ravel() - 0.5)
    [x_max, y_max] = np.int32(corners.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]])

    result = np.zeros((y_max - y_min, x_max - x_min, 3))
    temp = H_translation @ H
    for i in range(len(img1)):
        for j in range(len(img1[0])):
            newtemp = temp @ np.array([[j],[i],[1]])
            newtemp /= newtemp[2,0]
            result[int(newtemp[1,0]),int(newtemp[0,0])] = img1[i,j]
    temp = result

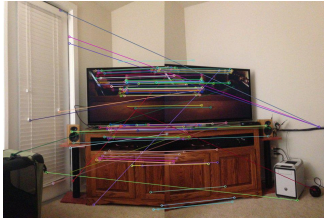
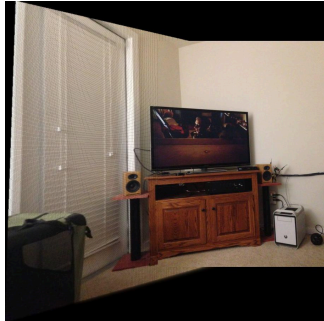

    for i in range(len(temp)):
        for j in range(len(temp[0])):
            if temp[i,j].sum() == 0:
                count = 0
                for x in range(-1,2):
                    for y in range(-1,2):
                        if result[min(len(temp)-1,max(0,i+x)),min(len(temp[0])-1,max(0,j+y))].sum() != 0 :
                            count += 1
                        result[i,j] += result[min(len(temp)-1,max(0,i+x)),min(len(temp[0])-1,max(0,j+y))]/25
            if count != 0:
                result[i,j] /= count
                result[i,j] *= 25

    result[translation_dist[1]:h2 + translation_dist[1], translation_dist[0]:w2 + translation_dist[0]] = img2
    return result.astype("uint8")

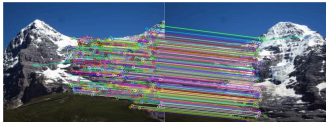

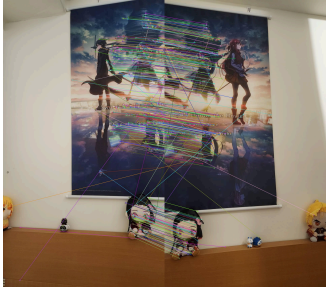
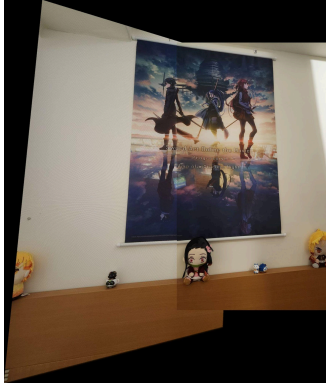
```

## Experimental results

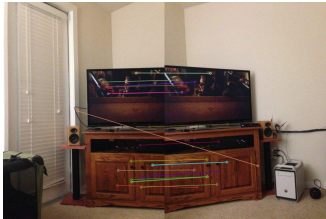
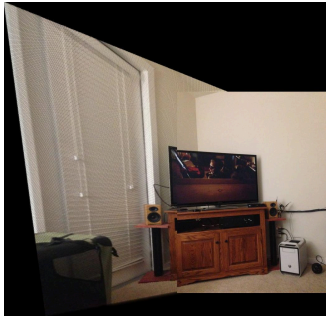
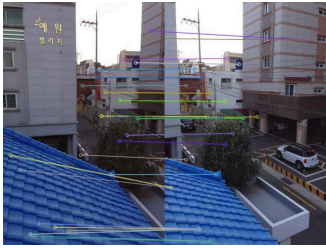

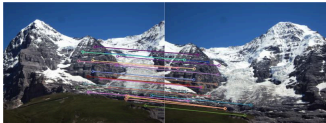
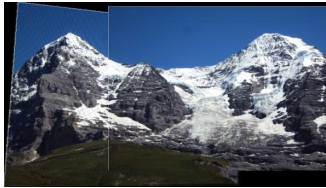
### SIFT



	match	some data value	result
TV		<p>H:</p> <pre>[[ 1.50423173e+00 -9.34092666e-04 -3.99687454e+02]  [ 3.29331352e-01 1.33157390e+00 -1.12660400e+02]  [ 1.11212035e-03 -1.11382871e-04 1.00000000e+00]]</pre> <p>features in img1: 421  features in img2: 428  good matches: 112</p>	
S		<p>H:</p> <pre>[[ 1.32594323e+00 2.75529449e-02 -1.63224654e+02]  [ 1.79666788e-01 1.26269557e+00 -4.27542289e+01]  [ 9.93282701e-04 1.01519440e-04 1.00000000e+00]]</pre> <p>features in img1: 951  features in img2: 1022  good matches : 289</p>	



hill		H: <pre>[ 1.11773546e+00 -7.25069776e-02 -1.60154497e+02] [ 1.12657816e-01 1.04292960e+00 -8.75318268e-01] [ 3.69330172e-04 -1.50265129e-04 1.00000000e+00]</pre> features in img1: 1256 features in img2: 1490 good matches : 534	
mydata		H: <pre>[ 1.41865103e+00 -7.32214564e-02 -8.58467040e+02] [ 4.29855426e-01 1.29177662e+00 -2.71301977e+02] [ 4.20648551e-04 -6.43516147e-05 1.00000000e+00]</pre> features in img1: 2342 features in img2: 2803 good matches : 603	

## MSER

	match	some data value	result
TV		H: <pre>[ 2.04796839e+00 2.13526228e-02 -5.53509036e+02] [ 7.09274502e-01 1.87511230e+00 -2.84976668e+02] [ 1.72048645e-03 4.44206410e-04 1.00000000e+00]</pre> features in img1: 315 features in img2: 268 good matches : 24	
S		H: <pre>[ 1.42809645e+00 5.65517603e-02 -1.86212707e+02] [ 2.30036406e-01 1.30547669e+00 -5.34797522e+01] [ 1.22750710e-03 6.62582392e-05 1.00000000e+00]</pre> features in img1: 317 features in img2: 279 good matches : 33	
hill		H: <pre>[ 1.13336738e+00 -5.98376594e-02 -1.63872940e+02] [ 1.38382735e-01 1.07886145e+00 -7.58760764e+00] [ 4.83966310e-04 -1.44635490e-04 1.00000000e+00]</pre> features in img1: 152 features in img2: 192 good matches : 29	

mydata		H: <pre>[ 1.37946136e+00 -7.49440188e-02 -8.34388675e+02] [ 4.02642414e-01  1.24638145e+00 -2.53047111e+02] [ 3.52769482e-04 -4.91817033e-05  1.00000000e+00]</pre> features in img1: 528 features in img2: 627 good matches : 68	
--------	---	---	---

## Discussion

1. The comparison between image stitching using features detected by SIFT and MSER

From the results above, we compared the image stitching using features detected by SIFT and MSER separately. The effect produced by SIFT is noticeably better in most cases, as it identifies three times more features than MSER—and up to ten times more in the case of the "hill" image. This leads to a significantly higher number of good matches with SIFT, making the difference in quality quite clear.

However, if perfect stitching is not required, the results from MSER are still acceptable. The only downside is minor misalignment at the boundaries, though in some cases (e.g., "hill"), MSER's results are comparable to those of SIFT. Due to the lower number of features detected by MSER, matching is also much faster. For instance, in the "mydata" image, SIFT required 50 seconds for matching, while MSER completed the task in just 2 seconds. In scenarios involving a large volume of computations, this speed can save a considerable amount of processing time.

2. When using RANSAC to compute the homography matrix, finding the optimal threshold and number of iterations was crucial. If the threshold was set too high, it could include too many outliers; conversely, if it was too low, it might not retain enough inliers for a robust estimation. We experimented with different parameters to achieve the best results.
3. Warp image to create panoramic image  
When the first image is warped to the new coordinate system, missing values can occur. Some pixels do not have suitable corresponding points in the new coordinate system, resulting in blank areas. In our implementation, we used the average value of neighboring pixels to fill in these missing regions, averaging the valid pixels in the surrounding 3x3 area to fill the blanks. However, when the edges in the image are complex or the color contrast is strong, this simple averaging strategy cannot completely eliminate visual discontinuities, and there may still be some visible lines in the stitched panoramic image.



## Conclusion

In summary, this project successfully achieved the goal of image stitching using feature detection and matching techniques, validating the effectiveness of different methods in the stitching process. After feature matching, we applied the RANSAC algorithm to compute the optimal homography matrix, filtering out incorrect matches and aligning the images into a seamless panorama. To address the stitching boundaries, we implemented edge blending, enhancing the smoothness of the image transitions. The result was a stable and cohesive panoramic image. These methods and workflows lay a solid foundation for future applications in image processing and computer vision.

## Work assignment plan between team members.

鍾任軒:

- Interest points detection & feature description by SIFT or MSER
- Feature matching by above features

王唯誠:

- RANSAC to find homography matrix H

黃竑睿:

- Warp image to create panoramic image

## File structure

Group[25]\_HW[3]\_code

— data

— some image need to be stitched(imageName[1,2].jpg)

— mydata

— some image myself need to be stitched(mydata[1,2].jpg)

— output

— some image after stitch (method\_imageName\_[match,result].jpg)

— code

— Group[25]\_HW[3]\_report.pdf