CV2024 HW3 Automatic Panoramic Image Stitching

Group10 - 313551073 顏琦恩, 313581027 葉彥谷, 313552001 黃梓濤

I. Introduction

Image stitching is a technique used to combine multiple images into a single, larger view or panorama by matching features and computing transformations. This technology is widely applied in fields like photography, Geographic Information Systems (GIS), medical imaging, and virtual reality (VR). The process is separated into 4 steps: feature detection, feature matching, homography estimation, and image wrapping. Our task in this homework is to implement the last 3 steps from scratch. For the first step, we utilized libraries from OpenCV to achieve our goal.

II. Implementation Procedure

Step 1. Interest Points Detection and Feature Description by SIFT

In this section, we applied 3 different feature detection functions either from OpenCV or from scratch to detect the interest points and feature description for input images. The detailed code is shown in the following code block.

```
def get_keypoint(img1, img2, method='SIFT'):
    img1 = cv2.cvtColor(img1, cv2.COLOR_RGB2GRAY)
    img2 = cv2.cvtColor(img2, cv2.COLOR_RGB2GRAY)

    if method == 'SIFT':
        detector = cv2.SIFT_create()
        keypoints1, descriptors1 = detector.detectAndCompute(img1, None)
        keypoints2, descriptors2 = detector.detectAndCompute(img2, None)
    elif method == 'MSER':
        detector = cv2.MSER_create()
        keypoints1 = detector.detect(img1)
        keypoints2 = detector.detect(img2)

        sift = cv2.SIFT_create()
        keypoints1, descriptors1 = sift.compute(img1, keypoints1)
        keypoints2, descriptors2 = sift.compute(img2, keypoints2)
    elif method == 'HARRIS':
        keypoints1 = harris_corner_detector(img1)
        keypoints2 = harris_corner_detector(img2)

        sift = cv2.SIFT_create()
        keypoints1, descriptors1 = sift.compute(img1, keypoints1)
        keypoints2, descriptors2 = sift.compute(img2, keypoints2)
    else:
```

```
        raise ValueError(
            f"Unsupported method: {method}. Choose from 'SIFT', 'MSER', or
'HARRIS'.")

    img1_keypoints = cv2.drawKeypoints(
        img1, keypoints1, None, flags=cv2.DrawMatchesFlags_DRAW_RICH_KEYPOINTS)
    img2_keypoints = cv2.drawKeypoints(
        img2, keypoints2, None, flags=cv2.DrawMatchesFlags_DRAW_RICH_KEYPOINTS)

    combined_img = cv2.hconcat([img1_keypoints, img2_keypoints])
    cv2.imwrite('img_keypoint.jpg', combined_img)
    print("img_keypoint.jpg saved")

    return [keypoints1, descriptors1], [keypoints2, descriptors2]


def harris_corner_detector(img):
    dst = cv2.cornerHarris(img, blockSize=2, ksize=3, k=0.04)
    dst = cv2.dilate(dst, None)

    keypoints = []
    threshold = 0.01 * dst.max()
    for y in range(dst.shape[0]):
        for x in range(dst.shape[1]):
            if dst[y, x] > threshold:
                keypoints.append(cv2.KeyPoint(x, y, 1))
    return keypoints
```

Step 2. Feature Matching by SIFT Features

We match feature descriptors between two images using a distance-based approach and the ratio test to improve match accuracy. The key point of our feature matching function is that to ensure reliable matching, we find two nearest descriptors for each descriptor in the first set, and calculate the ratio of the closest distance to the second-closest. If the ratio is below the threshold, the match is considered reliable and will be returned for further analysis. The detailed code is presented in the following code block.

```
def match_keypoints(descriptors1, descriptors2, threshold=0.1):
    best_score_pairs = []
    best_index_pairs = []
    for i in range(len(descriptors1)):
        best_scores = [float('inf'), float('inf')]
        best_index = [0, 0]
        for j in range(len(descriptors2)):
            score = np.linalg.norm(descriptors1[i] - descriptors2[j])
            if score < best_scores[0]:
                best_scores[1] = best_scores[0]
                best_index[1] = best_index[0]
                best_scores[0] = score
```

```
                best_index[0] = j
            elif score < best_scores[1]:
                best_scores[1] = score
                best_index[1] = j
        best_score_pairs.append(best_scores)
        best_index_pairs.append(best_index)

    final_index_pairs = []
    for i in range(len(best_index_pairs)):
        score = best_score_pairs[i][0] / best_score_pairs[i][1]
        if score < threshold:
            final_index_pairs.append((i, best_index_pairs[i][0]))

    return final_index_pairs
```

Step 3. Homography Estimation by RANSAC

The estimation of homography is mostly the same as calculating the homography for camera calibration, the only difference is that after finishing the estimation, we project points in the first image and measure its distance to the corresponding point in the second image. We count inliers and outliers for each homography matrix to find the best homography for the given point sets. The detailed code is shown in the following code block.

```
def homomat(points_in_img1, points_in_img2, sampling_point=4, threshold=5,
S=1500):
    max_inliers = 0
    best_H = None

    for _ in range(S):
        if len(points_in_img1) > sampling_point:
            sample_indices = np.random.choice(
                len(points_in_img1), sampling_point, replace=False)
            pts1 = points_in_img1[sample_indices]
            pts2 = points_in_img2[sample_indices]
        else:
            pts1 = points_in_img1
            pts2 = points_in_img2

        H = homography(pts1, pts2)

        inliers = 0
        for (x1, y1), (x2, y2) in zip(points_in_img1, points_in_img2):
            pred = np.dot(H, np.array([x1, y1, 1]))
            pred /= pred[2]  # normalize
            error = np.linalg.norm(np.array([x2, y2]) - pred[:2])
            if error < threshold:
                inliers += 1

        if inliers > max_inliers:
```

```
            max_inliers = inliers
            best_H = H
    print("Inliers:", inliers)

    return best_H


def homography(pts1, pts2):
    A = []
    for (x1, y1), (x2, y2) in zip(pts1, pts2):
        A.append([-x1, -y1, -1, 0, 0, 0, x1 * x2, y1 * x2, x2])
        A.append([0, 0, 0, -x1, -y1, -1, x1 * y2, y1 * y2, y2])
    A = np.array(A)
    _, _, V = np.linalg.svd(A)
    H = V[-1].reshape(3, 3)
    H = H/H[2, 2]   # Normalize to ensure H[2,2] is 1
    return H
```

Step 4. Image Wrapping

The last part of image stitching is wrapping two images together. We first create an empty canvas to fit images side-by-side, then we use the inverse of the homography matrix to map each pixel in one image to its corresponding pixel in the other image. For overlapping regions, we apply alpha blending based on the distance from the overlap edge. The final edge_blending function further smooths any visible seams using a Gaussian filter. The detailed code is shown in the following code block.

```
def wrapping(img1, img2, H):
    (h1, w1) = img1.shape[:2]
    (h2, w2) = img2.shape[:2]
    wrap_img = np.zeros( (max(h1, h2), w1+w2, 3), dtype="int")
    wrap_img[:h1, :w1] = img1

    # Transform left corr to right, and reproject to warped image
    inv_H = np.linalg.inv(H)
    for i in range(wrap_img.shape[0]):
        for j in range(wrap_img.shape[1]):
            coor = np.array([j, i, 1])
            img_right_coor = inv_H @ coor # the coor of right image
            img_right_coor /= img_right_coor[2]

            # interpolation
            y, x = int(round(img_right_coor[0])), int(round(img_right_coor[1]))
            if (x < 0 or x >= h2 or y < 0 or y >= w2):
                continue
            # wrap pixel
            # wrap_img[i, j] = img2[x, y]
            alpha = max(0, min(1, (w1 - j) / float(w1)))
            if(wrap_img[i, j][0]==0) and (wrap_img[i, j][1]==0) and (wrap_img[i,
```

```
j][2]==0):
                wrap_img[i, j] = img2[x, y]
            else:
                wrap_img[i, j] = (wrap_img[i, j] * alpha + img2[x, y] * (1 -
alpha)).astype(int)

    wrap_img = edge_blending(wrap_img, H)
    return wrap_img

def edge_blending(wrap_img, H):
    inv_H = np.linalg.inv(H)
    for i in range(wrap_img.shape[0]):
        for j in range(wrap_img.shape[1]):
            coor = np.array([j, i, 1])
            img_right_coor = inv_H @ coor  # the coor of right image
            img_right_coor /= img_right_coor[2]

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

            if x == 0:
                # Apply Gaussian filter for edge blending
                wrap_img[i, j] = gaussian_filter(wrap_img[i, j], sigma=2)
    return wrap_img
```

Experimental Results

We conducted the experiment with two different data setups, which are images from TA and images captured ourselves. The following figures present the feature matching and the final results of image stitching using data provided either by TA or by us. Note: The description of the output file is provided in Appendix I.

1. Results of TA's dataset

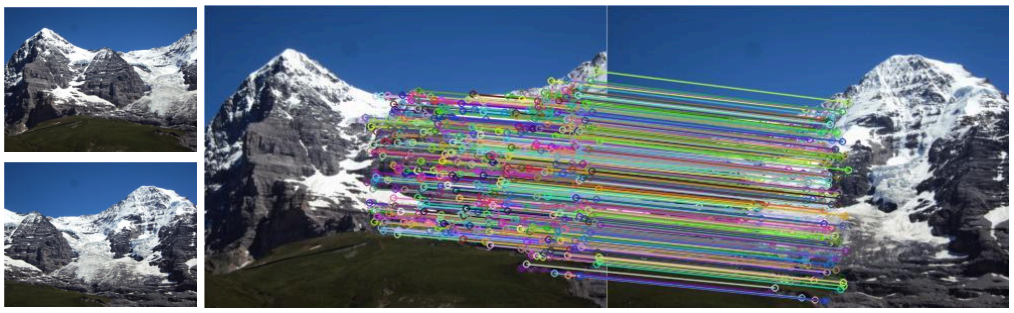
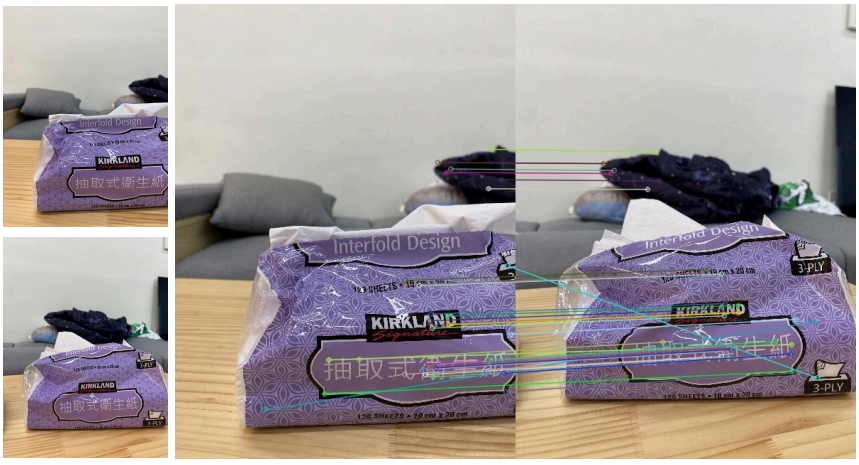


Figure. 1 Results of feature matching using TA's data.

Figure. 2 Results of stitched images using TA's data.

2. Results of our dataset



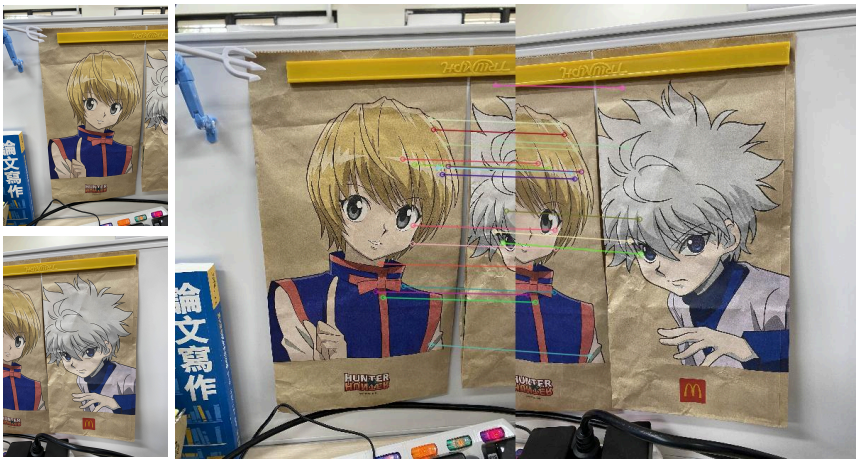


Figure. 3 Results of feature matching using our data.

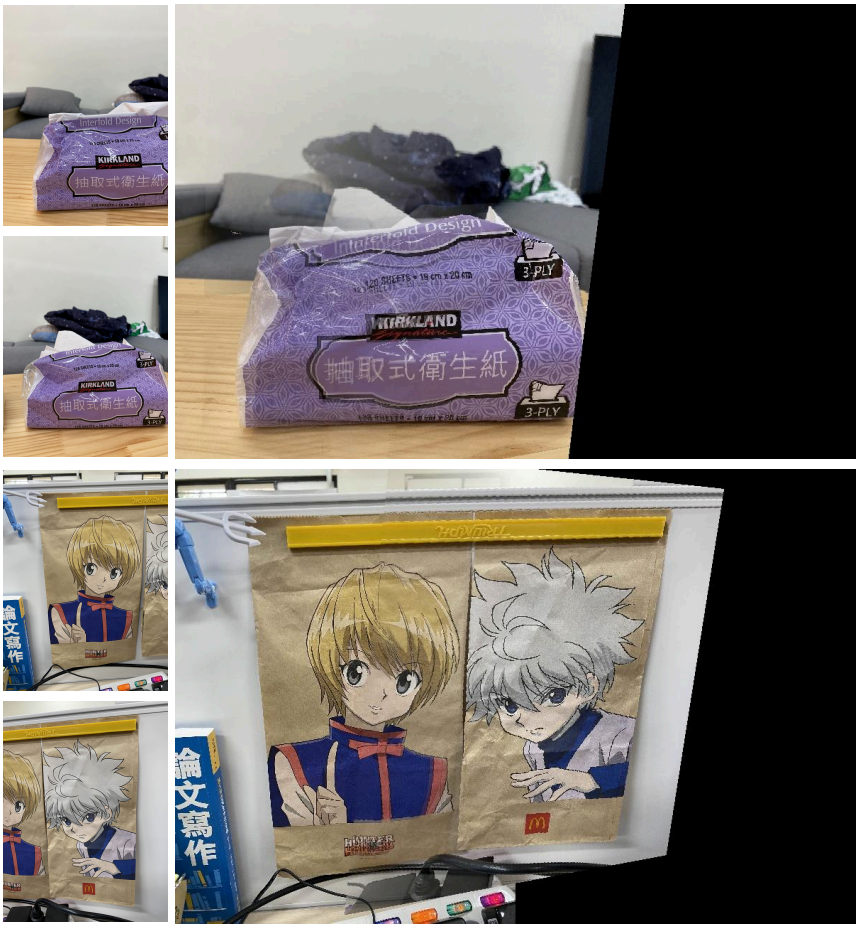


Figure. 4 Results of stitched images using our data.

We've further implemented multiple images stitching in our experiment. The following figure shows the results of matches and stitched images.

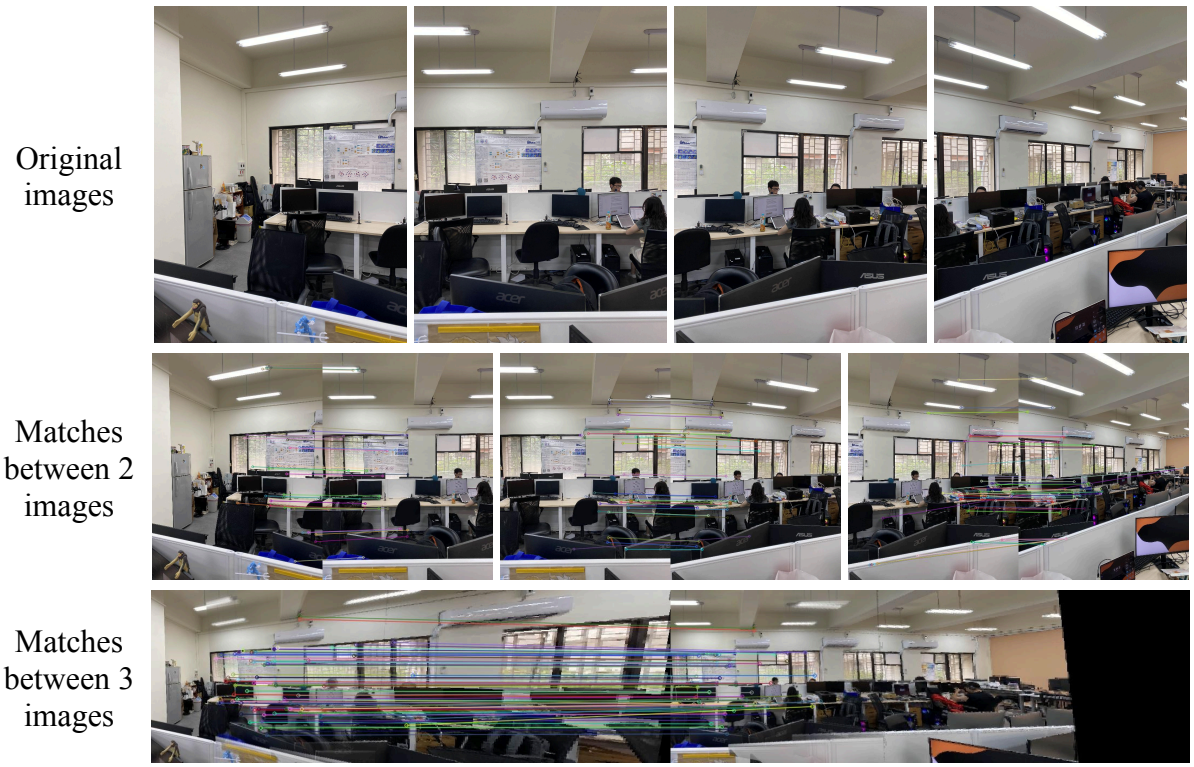


Figure. 5 Matching results of multiple images stitching.

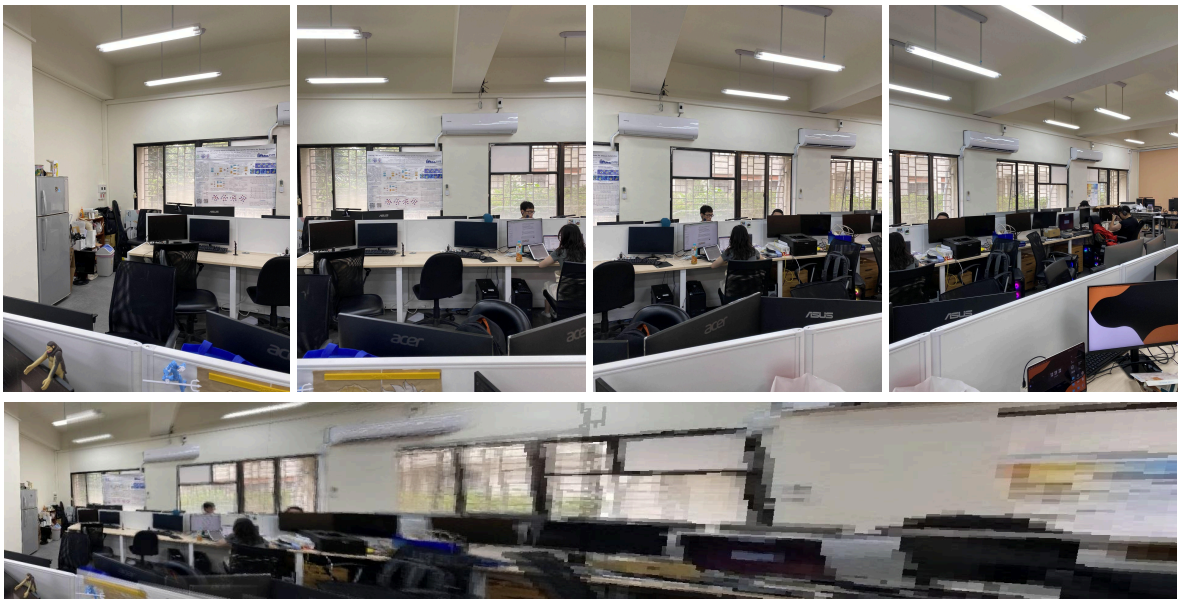


Figure. 6 Results of multiple images stitching.

III. Discussion

1. Comparison of different feature detection methods

We compare 3 different feature detection methods in this homework: SIFT, MSER, and Harris corner. The comparison from Table. 1 shows that SIFT is the most effective for image stitching, detecting stable and distinctive points that yield reliable matches and high-quality results due to its scale- and rotation-invariant keypoints. Harris Corner also detects stable

points but lacks SIFT's scale invariance, making it more sensitive to perspective changes. However, MSER, which detects high-contrast regions rather than distinct points, detects far fewer points, achieving only 6 matches, making it unsuitable for stitching as it lacks enough distinctive points for reliable alignment. Figure. 7 shows the visualization of the comparisons.

Table. 1 Comparison results of good matches and inliners of different feature detection methods.

Method	Good Matches	Inliners
SIFT	147	147
MSER	6	6
Harris corner	242	230

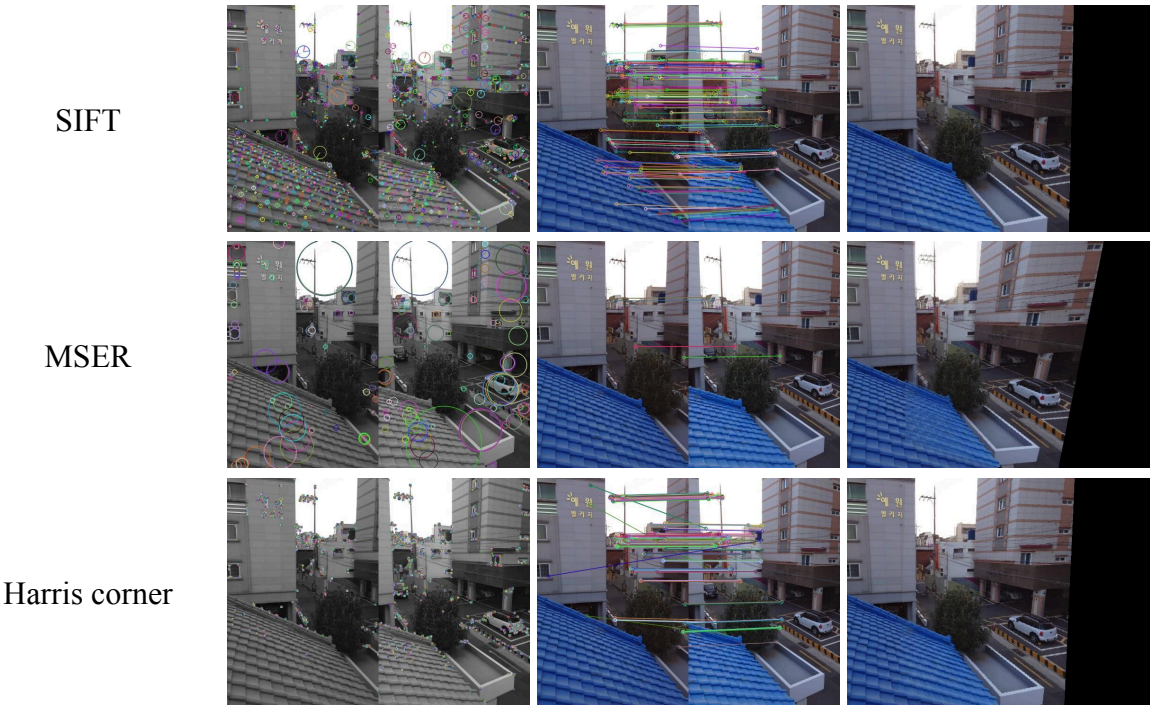


Figure. 7 Comparison results of different feature detection methods: left (Key Points), middle (Matches), and right (Stitched Image).

2. Comparison of different numbers of sampling points

We also conducted an experiment on the effectiveness of different numbers of sampling points during the calculation of the homography matrix. We run the SIFT method for 5 times and calculate the number of average inliners. The average error is calculated between the original key points and the predicted key points of the same image. In Table. 2, we can see that the more points we sampled, the more accurate the predictions are.

Table. 2 Results of different numbers of sampling points for calculating the homography matrix.

Sampling Point	Avg Error
4	1.3266
6	1.1576
8	1.0749
10	0.9491

IV. Conclusion

In conclusion, we've accomplished image stitching from scratch and created our own panorama. Furthermore, we utilized and compared the difference between 3 different feature detection methods.

V. Work Assignment Plan

- 葉彥谷 : Programming and report.
- 顏琦恩 : Programming and report.
- 黃梓濤 : Programming and report.

Appendix I : Filename Description

Path	Description
main.py	Main code
\my_data\	Our dataset
\output\matches\TA_data\	Matches results using TA's data
\output\matches\cus_data\	Matches results using our data
\output\stitched_images\TA_data\	Stitched images using TA's data
\output\stitched_images\cus_data\	Stitched images using our data