CV HW3 Group 19 Report

0756079 陳冠聞, 0756138 黃伯凱

April 23, 2019

1 Introduction

Image stitching is the process of combining multiple photographic images with overlapping fields of view to produce a segmented panorama or high-resolution image. Nowadays, most of the smartphones have the application of panoramic image stitching, and it's widely used in daliy life.

In this homework, we will implement the task of panoramic image stitching. Except for the SIFT, image I/O function and matrix operation function, all the other functions are implemented by ourselves.

2 Implementation

To do the task of Panoramic Image Stitching, we first use SIFT to detect and describe keypoints in different images, then do feature matching to find the correspondences between images, and compute the homography matrix based on those correspondences. Finally, we project the image and warp them to get the stitched panoramic image. And for better result, we use the multi-band blending approch to get the result image.

The details of each step is elaborated below, but be careful that we only show the code snippets which are the major components of our algorithm. To see more details and results, please refer to the folder "codes/" and "results/".

2.1 Keypoints detection and feature description

SIFT (scale-invariant feature transform) is a key point detection and description algorithm. The algoirhtm of SIFT is to find local extrema over scale and space in DoG (difference of Gaussians) as keypoints, and compute the gradient magnitude and direction of neigbourhood around the keypoint location as the description of the key point. SIFT is robust keypoint detector and feature descriptor, and it is invariant to rotation, scale and partially invariant to affine distortion.

In this homework, we use SIFT implemented by OpenCV as our key points detector and feature descriptor. From the Figure 10 we can see that SIFT can detect many interest points, and the keypoints of the same object are very consistent cross different images.

(a) Original Image

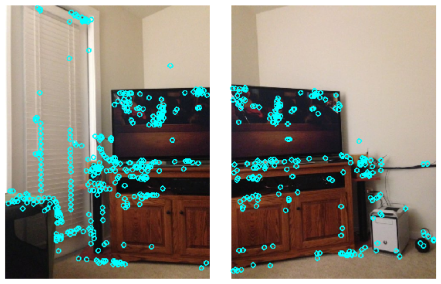
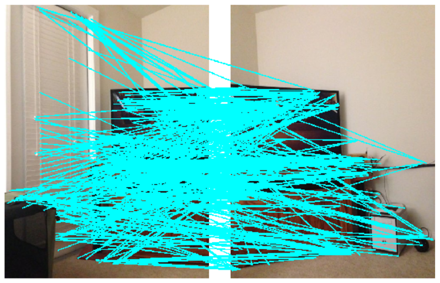

(b) keypoint

Figure 1: keypoints Detection (by SIFT)

2.2 Feature matching

To compute the homography matrix, we must find the correspondences between two image, and the correspondences can be acquired by matching the keypoints across different image. Sum of the square distance (SSD) is the metrics we used to find the matching between keypoints. The smaller a distance between two feature vector is, the more likely they are the same keypoint. So we find the match least SSD between feature vectors across two image as correspondence candidate. Nevertheless, some key point may only appear in one image because the two images are different, so we must threshold out bad matches. Ratio distance is the distance of best match divided by the distance of second best match. Lower ratio distance means the match is more unique, which represents it's a better match. We consider matches with ratio distance lower than 0.2 as bad matches, and remove them. As shown in Figure 2, we can see that the original matches (a) using pure SSD have a lot of incorrect correspondence, and the one with ratio distance (b) remove most of the incorrect correspondence. The implementation details can be seen in Figure 3.


(a) SSD Matching

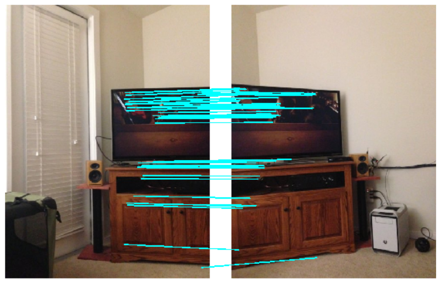

(b) Thresholded by Ratio Distance

Figure 2: keypoints Matching

2.3 Find homography matrix with RANSAC

RANSAC (random sample consensus) is a robust estimation procedure that estimate parameters of a model from a set of observed data that may contains outliers. To be robust to noise, RANSAC will sample many times, and select the solution that are estimated by the inliers in the round, which has maximum numbers of inliers among all rounds.

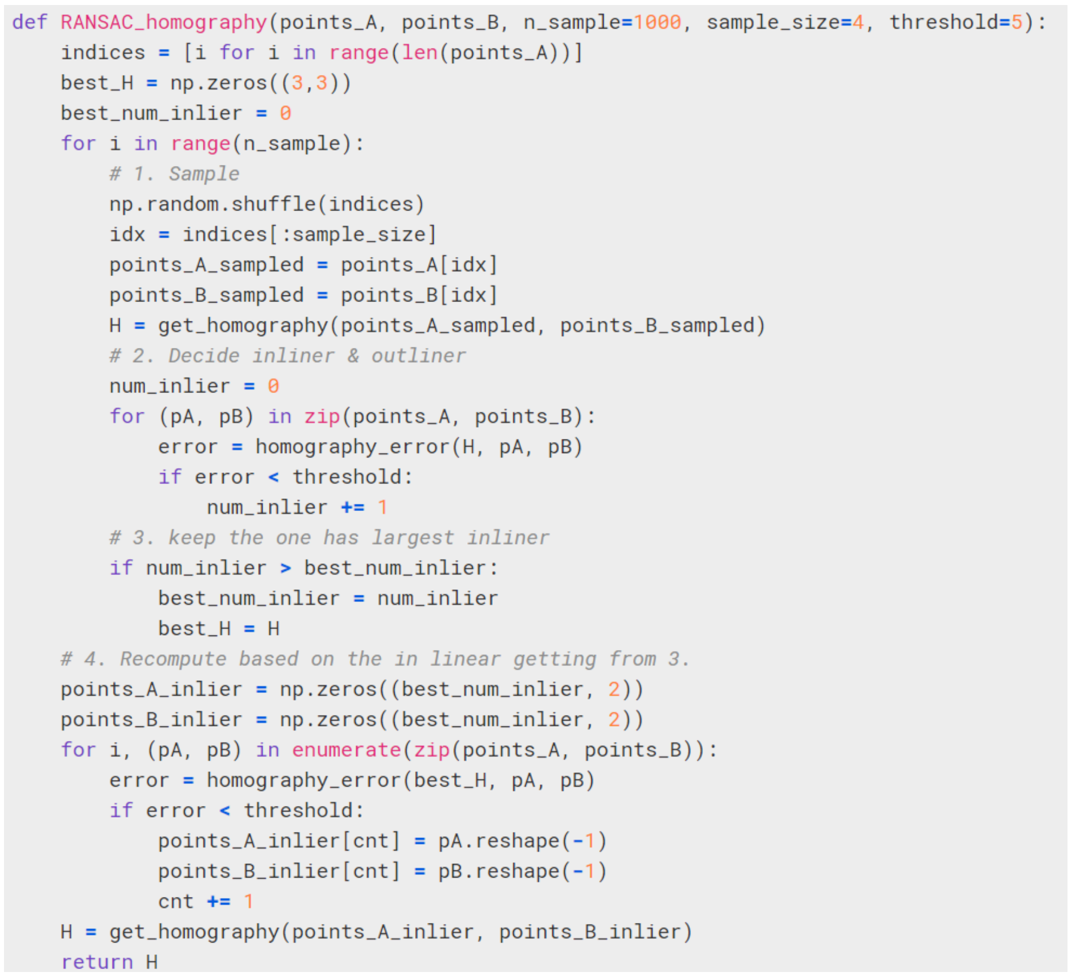
In the case of estimation of homography, we select the number of correspondences sampled to be 4 as well as the number of sample to be 1,000. The metric to decide whether a correspondence is outlier is the Euclidean distance between the true image point p_A and the projected point $\hat{p_A} = Hp_B$ greater than a threshold or not. In here, we set the threshold to be 5, which means the correspondence is considered as outlier if the Euclidean distance is greater than 5. Let (p_A, p_B) be a correspondences, the outlier decision with estimated homography matrix H can be formulated as following

$$(p_A, p_B) \in \begin{cases} inliers, & \text{if } \|p_A - Hp_B\| < 5 \\ outliers, & \text{otherwise} \end{cases} \tag{1}$$

The detail of implementation of RANSAC is shown in Figure 4. As for the estimation of homography matrix with given correspondences have been mentioned in homework 1, so we will leave it in appendix for conciseness.

```python
class KeypointMatcher:
    def __init__(self, detector='SIFT'):
        if detector == "SIFT":
            self.detector = cv2.xfeatures2d.SIFT_create()
        elif detector == "SURF":
            self.detector = cv2.xfeatures2d.SURF_create()
    def find_best_match(self, f, fs):
        ssds = []
        for f_ in fs:
            ssds.append(SSD(f, f_))
        index_sorted = np.argsort(ssds)
        f1 = fs[index_sorted[0]]
        f2 = fs[index_sorted[1]]
        ratio_distance = SSD(f, f1) / SSD(f, f2)
        f1_index = index_sorted[0]
        return f1_index, ratio_distance

    def find_best_matches(self, img1, img2, threshold=0.6):
        (keypoints_1, features_1) = self.detector.detectAndCompute(img1, None)
        (keypoints_2, features_2) = self.detector.detectAndCompute(img2, None)
        matches = [] #(index, ratio_dist)
        for f in features_1:
            matches.append(self.find_best_match(f, features_2))
        indices = [match[0] for match in matches]
        dists = [match[1] for match in matches]
        match_points = []
        for i in range(len(matches)):
            if dists[i] < threshold:
                k1 = keypoints_1[i].pt
                k2 = keypoints_2[indices[i]].pt
                k1 = int(k1[0]), int(k1[1])
                k2 = int(k2[0]), int(k2[1])
                match_points.append([k1, k2])
```

Figure 3: Code Snippet for Keypoints Detection and Feature Matching



```python
def RANSAC_homography(points_A, points_B, n_sample=1000, sample_size=4, threshold=5):
    indices = [i for i in range(len(points_A))]
    best_H = np.zeros((3,3))
    best_num_inlier = 0
    for i in range(n_sample):
        # 1. Sample
        np.random.shuffle(indices)
        idx = indices[:sample_size]
        points_A_sampled = points_A[idx]
        points_B_sampled = points_B[idx]
        H = get_homography(points_A_sampled, points_B_sampled)
        # 2. Decide inliner & outliner
        num_inlier = 0
        for (pA, pB) in zip(points_A, points_B):
            error = homography_error(H, pA, pB)
            if error < threshold:
                num_inlier += 1
        # 3. keep the one has largest inliner
        if num_inlier > best_num_inlier:
            best_num_inlier = num_inlier
            best_H = H
    # 4. Recompute based on the in linear getting from 3.
    points_A_inlier = np.zeros((best_num_inlier, 2))
    points_B_inlier = np.zeros((best_num_inlier, 2))
    for i, (pA, pB) in enumerate(zip(points_A, points_B)):
        error = homography_error(best_H, pA, pB)
        if error < threshold:
            points_A_inlier[cnt] = pA.reshape(-1)
            points_B_inlier[cnt] = pB.reshape(-1)
            cnt += 1
    H = get_homography(points_A_inlier, points_B_inlier)
    return H
```

Figure 4: Code Snippet for RANSAC

2.4 Warp image to create panoramic image

With the homography matrix we got from last section, we obtain the relationship between two images. That is, we are able to stitch one of the images to another by warping perspective.

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = H \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Because the coordinate of pixel is discrete, instead of compute the warped image by project original image directly, we choose to loop through all possible discrete pixel coordinate (x', y') and get the correspond pixel value by

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = H^{-1} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$$

This is, we get the pixel value $I^{'}(x', y')$ in warped image by sampling the pixel value $I(x, y)$ in the original image if the (x, y) is within valid image range. Again, the value of (x, y) might be float, and the pixel value only appear in integer coordinate, so we use **bilinear interpolation** to get the approximate value.

After warping perspective, there must be some part of the image overlapping. The overlapping region might be totally white if we do nothing but add the images together. To keep away from the terrible result, we must apply blending on both images. There are multiple ways to blend the images, but some of them are also terrible. As a result, we implement and compare four methods in this homework, and will explain them in detail in this section.

2.4.1 Without Blending

In first method, we just replace every pixels in the overlapping part with one image. Since there is no any blending but only image replacing in this method, there is an obvious artificial edge at the boundary of two images. In our implementation, we replace image on the right with the left one, the result is Figure 6(a).

2.4.2 Average Blending

The simplest way to blend is **Average Blending**. After warping perspective, we add two images together and get the overlapping region. Then we average each pixels in the overlapping area and we will get the result as Figure 6(b). In this method, image indeed looks smoother and better than the result without blending. However, there will be two blurred artificial lines at boundaries of both images, and the content in the overlapping region become blurred. It sure gives better blending, but still not good enough.

2.4.3 Linear Blending

To get rid of the previous artificial boundary problem, we can use **Linear Blending**. It is actually weighting two images in the overlapping area, and multiplies the weight of each pixels in the area. Briefly speaking, in the overlapping area, pixels on the left hand side will be more similar to the left image, while pixels on the right hand side will be more similar to the right image. The result is Figure 6(c), it is much better than previous methods. However, if you pay more attention to the air conditioner controller in the image, you will find out ghost shadow. That is, we will keep looking for a better solution.

2.4.4 Multi-Band Blending

Multi-Band Blending is quite similar to *Linear Blending*. The difference is that Multi-Band Blending uses **Laplacian Pyramid** to blend images. First, we construct Laplacian Pyramid of both images, and initial a blending window size. Then, we will weight all images at the same level of the pyramid in the blending window, and sum them up. Finally, reconstruct from the coarsest layer to the finest layerthe pyramid and we will get the result as Figure 6(d). It has natural color and has no ghost shadow anymore. In our implementation, we use 6-Band Blending, and we consider multi-band blending is the best method of image blending among four method we have mentioned above. The details of multi-band blending can refer to Figure 7, but note the implementation of image pyramid is the same as the one in homework 2, so we don't elaborate it here but leave it in the appendix.

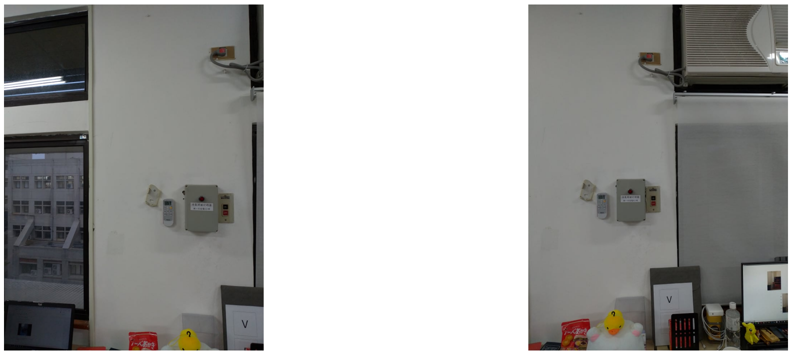
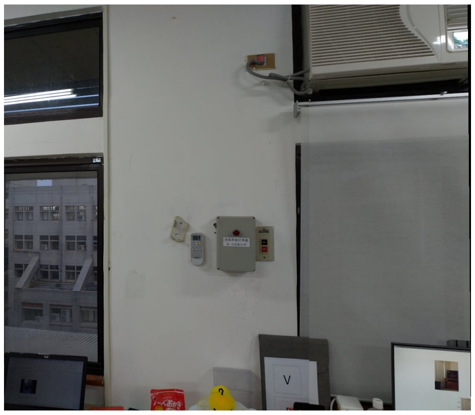
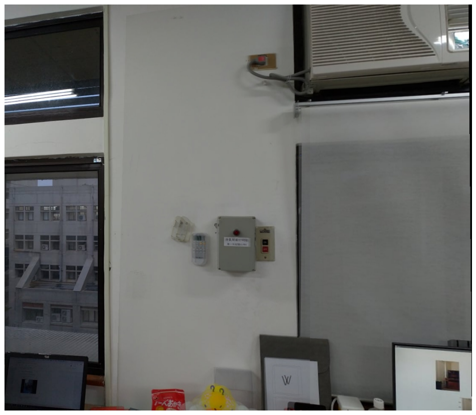


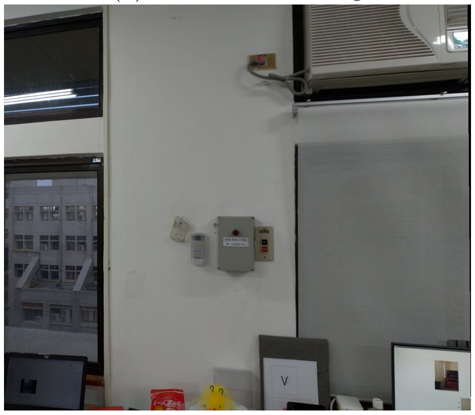
Figure 5: Original Image



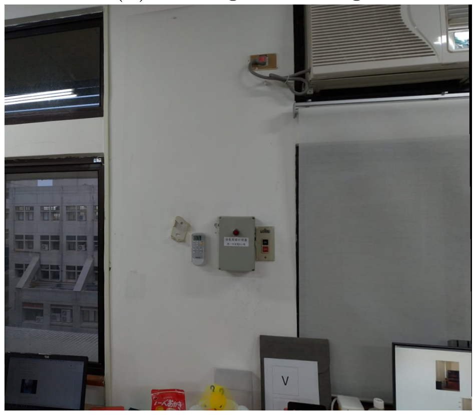
(a) Without Blending



(b) Average Blending

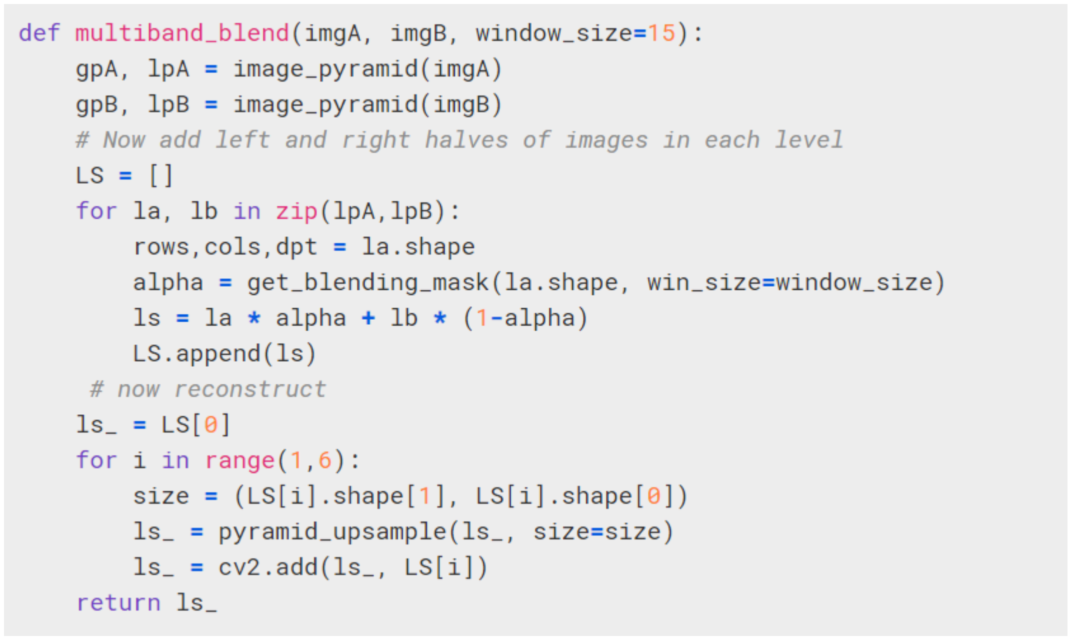


(c) Linear Blending



(d) Multi-band Blending

Figure 6: Blending Result



```python
def multiband_blend(imgA, imgB, window_size=15):
    gpA, lpA = image_pyramid(imgA)
    gpB, lpB = image_pyramid(imgB)
    # Now add left and right halves of images in each level
    LS = []
    for la, lb in zip(lpA,lpB):
        rows,cols,dpt = la.shape
        alpha = get_blending_mask(la.shape, win_size=window_size)
        ls = la * alpha + lb * (1-alpha)
        LS.append(ls)
     # now reconstruct
    ls_ = LS[0]
    for i in range(1,6):
        size = (LS[i].shape[1], LS[i].shape[0])
        ls_ = pyramid_upsample(ls_, size=size)
        ls_ = cv2.add(ls_, LS[i])
    return ls_
```

Figure 7: Code Snippet for Multi-band Blending

3 Discussion

3.1 SIFT v.s. SURF

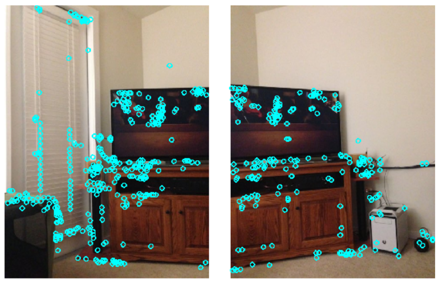
SURF (Speeded Up Robust Feature) was proposed mainly to improve the speed of SIFT. For the keypoint detection, SIFT use DoG to approximate LoG while SURF use different size of box filter to approximate, which is faster and easier to compute parallelly. For the feature descriptor, SIFT use 128-D vector while SURF uses 64-D vector as descriptor.

SIFT and SURF are neck and neck in term of detection performance. SURF performs better for blurred or rotated image while SIFT deal with scale problem better. However, in term of speed, SURF is definitely superior to SIFT. SURF is about three times the speed of SIFT, which make SURF become an better choice for near-realtime application.

Beside the characteristic mentioned above, the keypoints found by SIFT and SURF are also different, so we want to compare the keypoints they find. As shown in Figure 8 and Table 1, we try both SIFT and SURF to the same image. The result shows that SURF is more repeatable and the SIFT is more distinctive.

Table 1: Comparison Between SIFT and SURF

	# Keypoints	# Matches (thresholded)
SIFT	(410, 424)	51
SURF	(1029, 981)	81


(a)SIFT

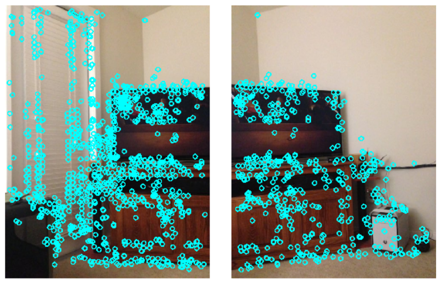

(b)SURF

Figure 8: Keypoints Detection

3.2 Ratio Distance with RANSAC

Ratio distance is used to eliminate the incorrect matches in feature matching. The higher threshold value is, the less the incorrect matches will appear. However, higher threshold value will also reduce the number of correct matches, which may lead to less accurate estimation of a homography. It's hard to determine the best threshold. Fortunately, since homography is estimated using the RANSAC algorithm, which is capable of removing outliers, we can loosen the threshold of ratio distance, making the matches include more correct matches, and eliminate the outliers in RANSAC.

We use the same image above as well as SIFT as the detector, and set the threshold to 0.25, 0.5 and 0.75 to see the difference between them. From table 2 we can see that although loose threshold indeed includes more incorrect matches, but number of inliers is increased, which could let the estimation of the homography become more stable as well as more accurate.

Table 2: Comparison of different thresholds

	# Matches	# Inliers	# Outliers
T=0.25	58	56	2
T=0.50	100	72	28
T=0.75	164	84	80

4 Result

In this section, we show the results of panoramic image stitching both given data and our data. We use the parameters setting by the experience we learn from the implementation and discussion section. That is, SIFT as the keypoints detector and descriptor, threshold is set to 0.75 for ratio distance, and multi-band blending method to produce the results. For higher resolution images, please refer to the folder "results/".

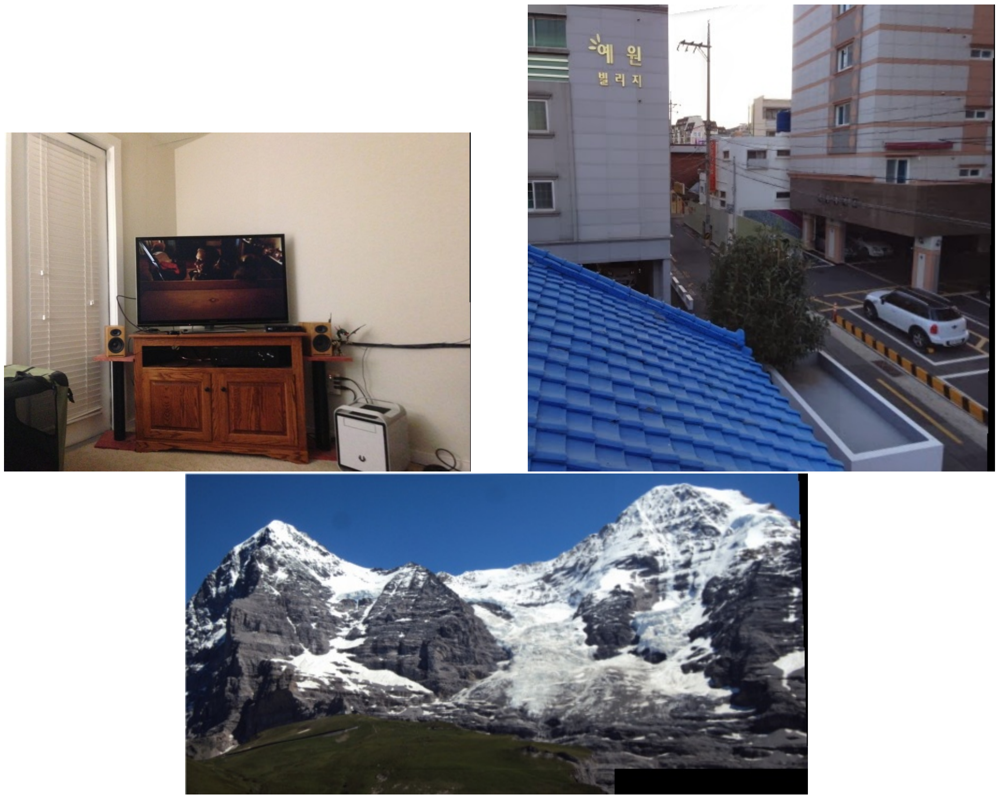


Figure 9: Result of Given Data

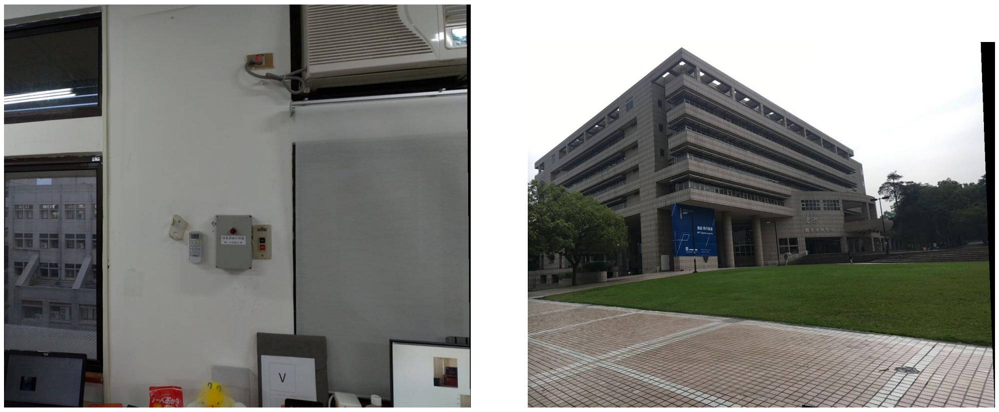


Figure 10: Result of Our Data

5 Conclusion

To fulfill panoramic image stitching, we use **SIFT** to detect keypoints and match the images. Then, we use **RANSAC** to find homography matrix between these images. The homography matrix we found enables us to warp the images together with **Bilinear Interploation**, and we use four different methods to blend the warped images. After doing all of these, we find out **Multi-Band Blending** is the best method for us to blend the warped images (Figure 6(d)). Furthermore, we not only discuss the comparsion between SURF and SIFT, but also talk about the usage of ratio distance with RANSAC in the discussion.

6 Work Assignment

0756138 黃伯凱: Code and report of image stitching and blending .

0756079 陳冠聞: The rest of the code and report.

7 Appendix

7.1 Estimation of homography matrix

We know the correspondences of two image planes. If we rewrite the formula in matrix form with N correspondences

$$\begin{pmatrix} -X_0 & -Y_0 & -1 & 0 & 0 & 0 & u_0 * X_0 & u_0 * Y_0 & u_0 \\ 0 & 0 & 0 & -X_0 & -Y_0 & -1 & v_0 * X_0 & v_0 * Y_0 & v_0 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ -X_N & -Y_N & -1 & 0 & 0 & 0 & u_N * X_N & u_N * Y_N & u_N \\ 0 & 0 & 0 & -X_N & -Y_N & -1 & v_N * X_N & v_N * Y_N & v_N \end{pmatrix} \begin{pmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{pmatrix} = \vec{0} \tag{2}$$

The Equation 2 is actually a homogeneous system $A\vec{h} = 0$, and we want don't want a trivial solution where $\vec{h} = 0$. Because the correspondences are the noisy observation, we can solve $min\|A\vec{h}\|$ to get our best estimation of H. By linear algebra, the solution can be computed using SVD, that is $USV^T = A$. And the solution of $\vec{h}$ is the eigenvector corresponding to the minimum singular value in S. Once get the $\vec{h}$, we can get the H simply by reshape the $\vec{h}$.

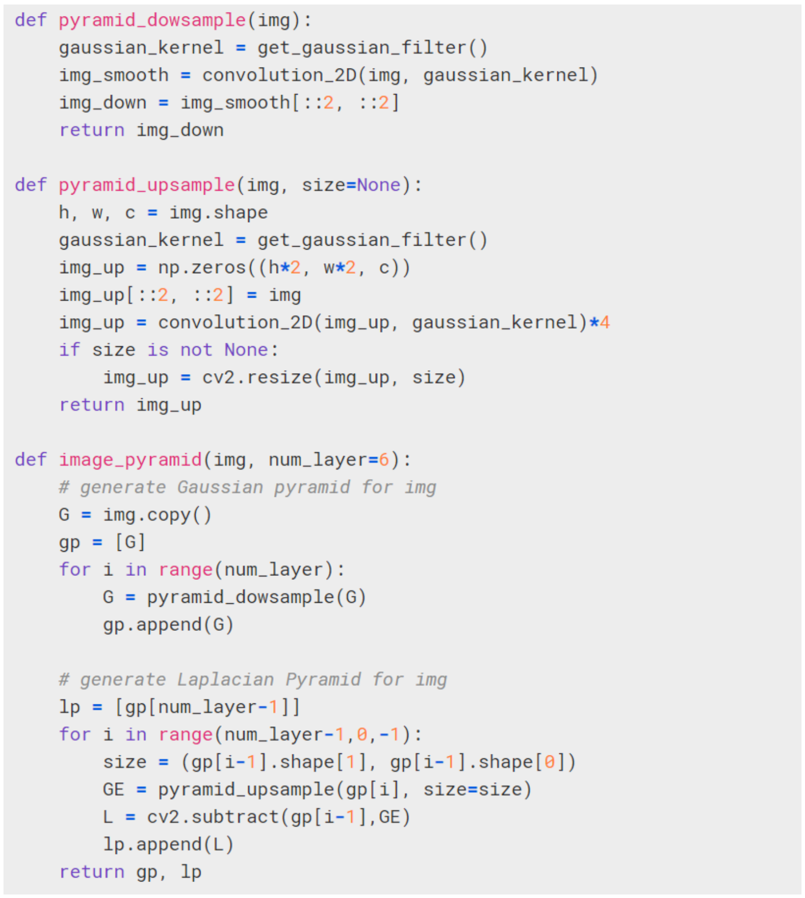
```
def get_homography(points_A, points_B):
    #loop through correspondences and create assemble matrix
    aList = []
    for p1, p2 in zip(points_A, points_B):
        p1 = np.matrix([p1[0], p1[1], 1])
        p2 = np.matrix([p2[0], p2[1], 1])

        a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2),
              p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
        a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2), 0, 0, 0,
              p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
        aList.append(a1)
        aList.append(a2)

    matrixA = np.matrix(aList)

    #svd composition
    u, s, v = np.linalg.svd(matrixA)

    #reshape the min singular value into a 3 by 3 matrix
    h = np.reshape(v[8], (3, 3))

    #normalize and now we have h
    h = (1/h.item(8)) * h
    return np.array(h)
```

Figure 11: Code snippet for computing homography matrix

7.2 Image Pyramid

We build the Gaussian pyramid as following. The first layer (finest layer) is just the input image, and the following layers is apply Gaussian filter to the previous layer then do subsampling, and all the way down do the last layer (coarsest layer).

For the Laplacian pyramid, the n-th layer of Laplacian pyramid L_n is n-th layer of Gaussian pyramid G_n subtract from upsampled result of (n+1)-th layer of Gaussian pyramid, which can be written as $L_n = G_n - upsample(G_{n+1})$.



```
def pyramid_dowsample(img):
    gaussian_kernel = get_gaussian_filter()
    img_smooth = convolution_2D(img, gaussian_kernel)
    img_down = img_smooth[::2, ::2]
    return img_down

def pyramid_upsample(img, size=None):
    h, w, c = img.shape
    gaussian_kernel = get_gaussian_filter()
    img_up = np.zeros((h*2, w*2, c))
    img_up[::2, ::2] = img
    img_up = convolution_2D(img_up, gaussian_kernel)*4
    if size is not None:
        img_up = cv2.resize(img_up, size)
    return img_up

def image_pyramid(img, num_layer=6):
    # generate Gaussian pyramid for img
    G = img.copy()
    gp = [G]
    for i in range(num_layer):
        G = pyramid_dowsample(G)
        gp.append(G)

    # generate Laplacian Pyramid for img
    lp = [gp[num_layer-1]]
    for i in range(num_layer-1,0,-1):
        size = (gp[i-1].shape[1], gp[i-1].shape[0])
        GE = pyramid_upsample(gp[i], size=size)
        L = cv2.subtract(gp[i-1],GE)
        lp.append(L)
    return gp, lp
```

Figure 12: Code snippet for Image Pyramid