Computer Vision

Group 12 | 0786031 廖俊凱 0516044 陳思妤 0856733 黃明翰

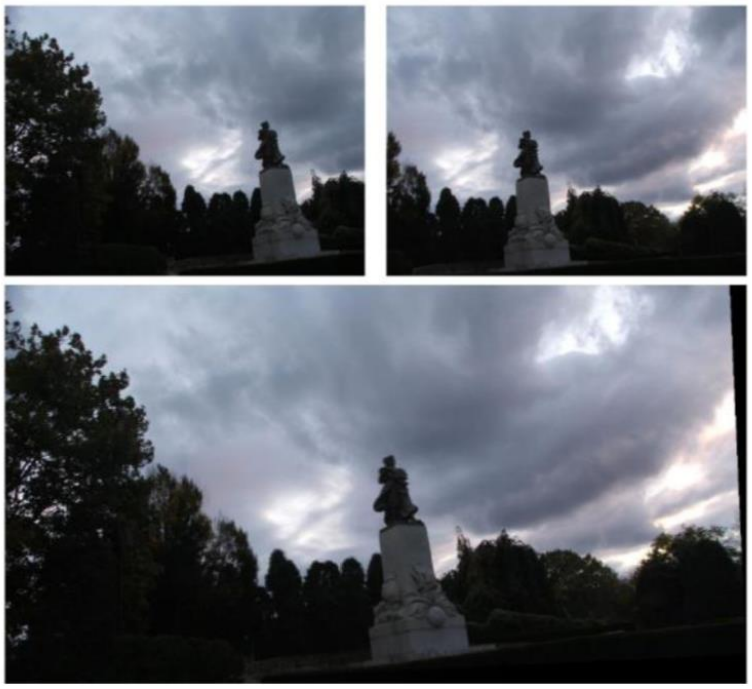
Introduction

Image stitching, in simple terms, is for an input there should be a group of images, the output is a composite image such that it is a culmination of image scenes. At the same time, the logical flow between the images must be preserved.

For example, think about sea horizon while you are taking few photos of it. From a group of these images, we are essentially creating a single stitched image, that explains the full scene in detail. It is quite an interesting algorithm.

Let's first understand the concept of image stitching. Basically, if you want to capture a big scene and your camera can only provide an image of a specific resolution and that resolution is 640 by 480, it is certainly not enough to capture the big panoramic view. So, what we can do is to capture multiple images of the entire scene and then put all bits and pieces together into one big image. Such photos of ordered scenes of collections are called panoramas. The entire process of acquiring multiple image and converting them into such panoramas is called as image stitching. And finally, we have one beautiful big and large photograph of the scenic view.

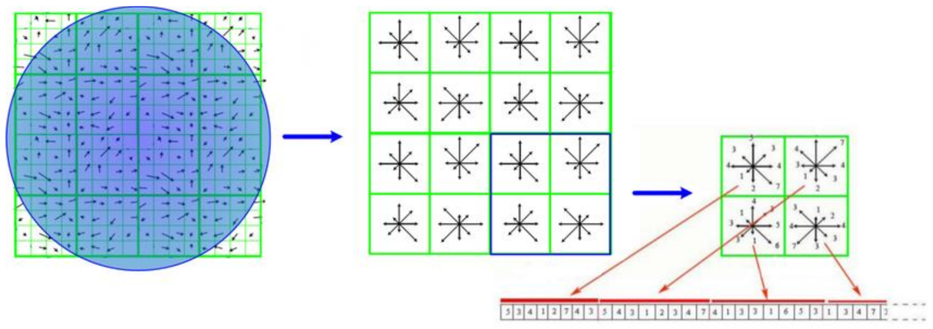
The basic geometry of the problem is well understood, and consists of estimating a 3×3 camera matrix or homography for each image. This estimation process needs an initialization, which is typically provided by user input to approximately align the images, or a fixed image ordering. In this assignment, we formulate stitching as a multi-image matching problem, and use invariant local features to find matches between all of the images.



Implementation Procedure

A. Interest points detection & feature description by SIFT

1. Take 16x16 square window around detected feature
2. Compute edge orientation for each pixel
3. Throw out weak edges (threshold gradient magnitude)
4. Create histogram of surviving edge orientations
5. 16 cells * 8 orientations = 128 dimensional descriptors



```python
def sift(input_img, img_name):
    img = np.uint8(input_img)
    descriptor = cv2.xfeatures2d.SIFT_create()
    (kps, des) = descriptor.detectAndCompute(img, None)
    cv2.drawKeypoints(img, kps, img, (0,255,255), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    cv2.imwrite('./results/sift/sift_{}.jpg'.format(img_name.split('.')[0]),img)
    return kps, des
```

B. Feature matching by SIFT features

1. Exact and match SIFT features between all of the images
2. SIFT features are located at scale-space maxima/minima of a difference of Gaussian function
3. At each feature location, a characteristic scale and orientation is established
4. Give a similarity-invariant frame in which to make measurements

Once features have been extracted from all n images (linear time), they must be matched. Since multiple images may overlap a single ray, each feature is matched to its k nearest neighbors in feature space (we use k = 4). This can be done in O(n log n) time by using a k-d tree to find approximate nearest neighbors. A k-d tree is an axis aligned binary space partition, which recursively partitions the feature space at the mean in the dimension with highest variance.

```python
def find_match(des1, des2):
    dist_matrix = np.zeros((len(des1), len(des2)))
    # find the distance between each and every descriptors
    for index1 in range(len(des1)):
        for index2 in range(len(des2)):
            dist = np.linalg.norm(des1[index1] - des2[index2])
            dist_matrix[index1][index2] = dist
    #sort the vector to get the two best match with the smallest distance
    #match = [descriptor's number(descriptor i), the two best match with descriptor i, distance with the first match, destance with the second match]
    match = []
    for i in range(dist_matrix.shape[0]):
        two_best_match = np.argsort(dist_matrix[i])[:2]
        match.append ([i, two_best_match, dist_matrix[i][two_best_match[0]], dist_matrix[i][two_best_match[1]]])
    #perform ratio distance with ratio = 0.75
    final_match = []
    for i in range(len(match)):
        if match[i][2] < match[i][3] * 0.75:
            final_match.append([match[i][0], match[i][1][0]])
    return final_match

def draw_match(img1, img2, kps1, kps2, match, img_name):
    height1 = img1.shape[0]
    width1 = img1.shape[1]
    height2 = img2.shape[0]
    width2 = img2.shape[1]
    output_img = np.zeros((max(height1,height2), width1+width2, 3), dtype = 'uint8')
    output_img[0:height1, 0:width1] = img1
    output_img[0:height2, width1:] = img2

    for index1, index2 in match:
        color = list(map(int, np.random.randint(0, high=255, size=(3,))))
        pts1 = (int(kps1[index1].pt[0]), int(kps1[index1].pt[1]))
        pts2 = (int(kps2[index2].pt[0] + width1), int(kps2[index2].pt[1]))
        cv2.line(output_img, pts1, pts2, color, 1)
    cv2.imwrite('./results/match/matchline_'+img_name+'.jpg', output_img)
```

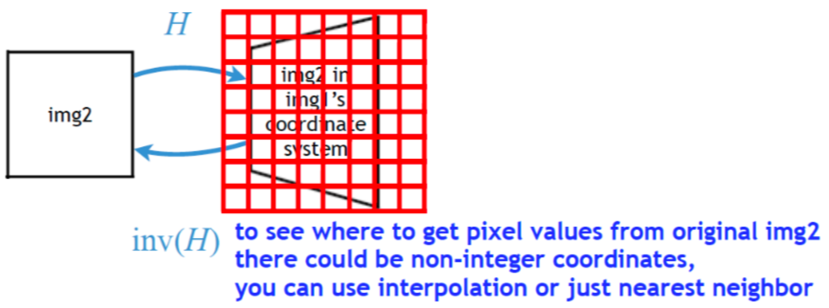
C. RANSAC to find homography matrix H

1. Randomly select a RANSAC sample from the sample set, that is, 4 matching point pairs
2. Calculate the homography matrix H according to these 4 matching point pairs
3. According to the sample set, the homography matrix H, and the error metric function, calculate the consistent set consensus that satisfies the current the homography matrix H and return the number of elements in the set
4. Determine whether the optimal (maximum) consistent set is based on the number of elements in the current consistent set, and if so, update the current optimal consistent set
5. Update the current error probability p, if p is greater than the minimum allowed error probability, repeat (1) to (4) to continue iterating until the current error probability p is less than the minimum error probability

```python
def homomat(min_match_count: int, src, dst):
    A = np.zeros((min_match_count * 2, 9))
    # construct the two sets of points
    for i in range(min_match_count):
        src1, src2 = src[i, 0, 0], src[i, 0, 1]
        dst1, dst2 = dst[i, 0, 0], dst[i, 0, 1]
        A[i * 2, :] = [src1, src2, 1, 0, 0, 0, -src1 * dst1, - src2 * dst1, -dst1]
        A[i * 2 + 1, :] = [0, 0, 0, src1, src2, 1, -src1 * dst2, - src2 * dst2, -dst2]

    # compute the homography between the two sets of points
    [_, S, V] = np.linalg.svd(A)
    m = V[np.argmin(S)]
    m *= 1 / m[-1]
    H = m.reshape((3, 3))
    return H

def ransac(final_match, kps_list, min_match_count, num_test: int, threshold: float):
    if len(final_match) > min_match_count:
        src_pts = np.array([kps_list[1][m[1]].pt for m in final_match]).reshape(-1, 1, 2)
        dst_pts = np.array([kps_list[0][m[0]].pt for m in final_match]).reshape(-1, 1, 2)
        min_outliers_count = math.inf

        while(num_test != 0):
            indexs = np.random.choice(len(final_match), min_match_count, replace=False)

            homography = homomat(min_match_count, src_pts[indexs], dst_pts[indexs])

            # Warp all left points with computed homography matrix and compare SSDs
            src_pts_reshape = src_pts.reshape(-1, 2)
            one = np.ones((len(src_pts_reshape), 1))
            src_pts_reshape = np.concatenate((src_pts_reshape, one), axis=1)
            warped_left = np.array(np.mat(homography) * np.mat(src_pts_reshape).T)
            for i, value in enumerate(warped_left.T):
                warped_left[:, i] = (value * (1 / value[2])).T

            # Calculate SSD
            dst_pts_reshape = dst_pts.reshape(-1, 2)
            dst_pts_reshape = np.concatenate((dst_pts_reshape, one), axis=1)
            inlier_count = 0
            inlier_list = []
            for i, pair in enumerate(final_match):
                ssd = np.linalg.norm(np.array(warped_left[:, i]).ravel() - dst_pts_reshape[i])
                if ssd <= threshold:
                    inlier_count += 1
                    inlier_list.append(pair)

            if (len(final_match) - inlier_count) < min_outliers_count:
                min_outliers_count = (len(final_match) - inlier_count)
                best_homomat = homography
                best_matches = inlier_list
            num_test -= 1
        return best_homomat, best_matches
    else:
        raise Exception("Not much matching keypoints exits!")
```

D. Warp image to create panoramic image

Once we have established a homography, i.e. we know how the second image will look from the current image's perspective, we need to transform it into a new space. That is, the slightly distorted, and altered image that we see from our periphery. This process is called warping. We are converting an image, based on a new transformation.


H
img2
img2 in img1's coordinate system
inv(H)
to see where to get pixel values from original img2
there could be non-integer coordinates,
you can use interpolation or just nearest neighbor

```
def forward_warp(h, w, warped_img, img_grid):
    res = np.zeros((h, w, 3), dtype='uint8')
    for x, y, im in zip(warped_img[0], warped_img[1], img_grid):
        res[math.floor(y)+top, math.floor(x)+left] = img2[im[1], im[0], :]
    cv2.imwrite('./results/warp/warp_res/forward_'+img_name+'.jpg', res)
    blend(res, 'forward_')

def inverse_warp(h, w, corners, homography):

    res_nn = np.zeros((h, w, 3), dtype='uint8')
    res_bi = np.zeros((h, w, 3), dtype='uint8')

    # Create image 2 grid in image 1 coordinate
    b, t, r, l = math.ceil(max(corners[:, 1])),math.floor(min(corners[:, 1])),math.ceil(max(corners[:, 0])), math.floor(min(corners[:, 0]))

    img2_trans_grid = [[n, m, 1] for n in range(l, r) for m in range(min(t, 1), b)]

    # Inverse mapping points on image 1 to image 2
    img2_trans_inv = np.array(np.mat(np.linalg.inv(homography)) * np.mat(img2_trans_grid).T)
```

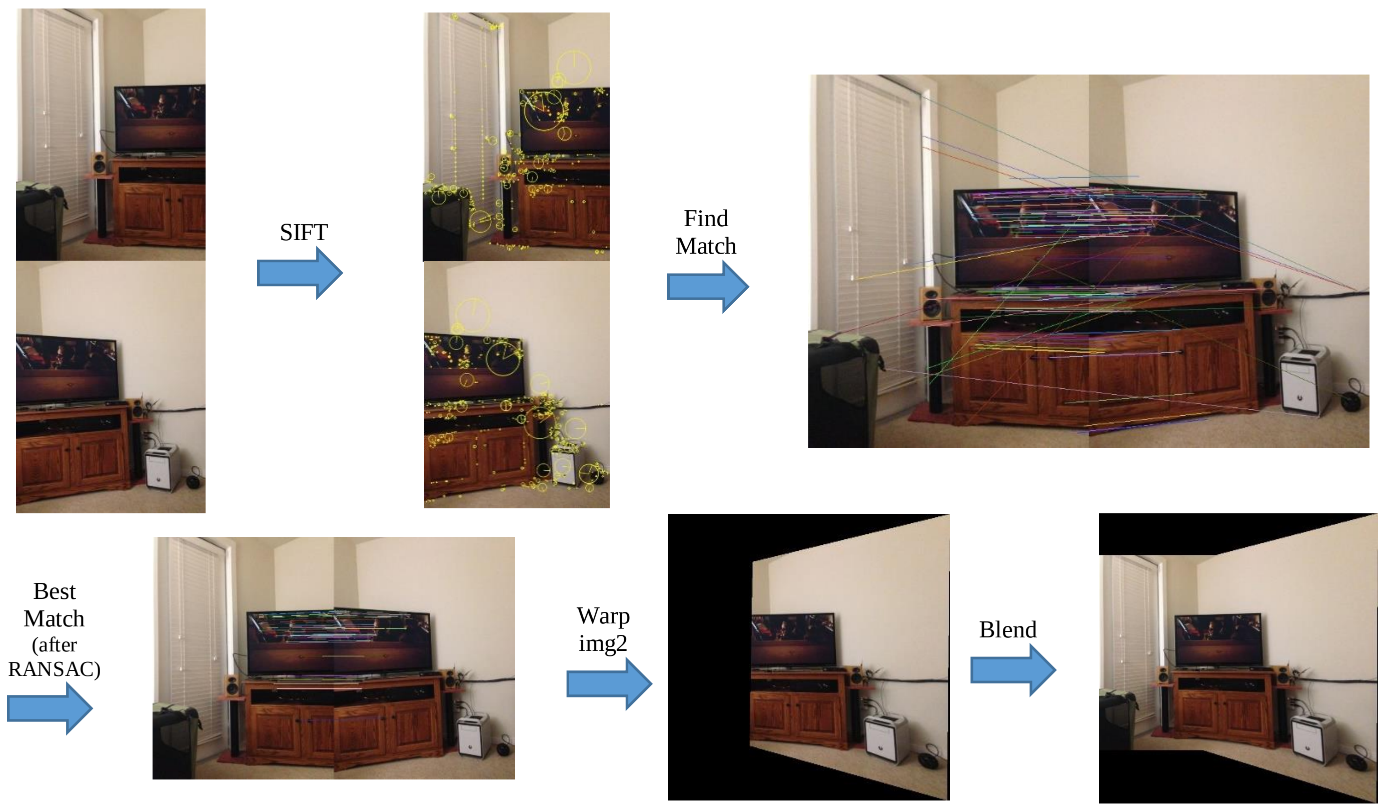
```python
    img2_trans_inv /= img2_trans_inv[2]

    for x, y, im in zip(img2_trans_inv[0], img2_trans_inv[1], img2_trans_grid):
        if math.ceil(y) < img2.shape[0] and math.ceil(y)>0 and math.ceil(x) < img2.shape[1] and math.ceil(x)>0:
            # Nearest Neighbor
            res_nn[im[1]+top, im[0]+left] = img2[int(y + 0.5), int(x + 0.5), :]

            # Bilinear interpolation
            res_bi[im[1]+top,im[0]+left] = (img2[math.ceil(y), math.ceil(x), :]*((y-math.floor(y))*(x-math.floor(x)))+
                                                 img2[math.floor(y), math.floor(x), :]*((math.ceil(y)-y)*(math.ceil(x)-x))+
                                                 img2[math.ceil(y), math.floor(x), :]*((y-math.floor(y))*(math.ceil(x)-x))+
                                                 img2[math.floor(y), math.ceil(x), :]*((math.ceil(y)-y)*(x-math.floor(x))))
    cv2.imwrite('./results/warp/warp_res/backward_nn_'+img_name+'.jpg', res_nn)
    cv2.imwrite('./results/warp/warp_res/backward_bi_'+img_name+'.jpg', res_bi)
    #blend(res_nn, 'inverse_nn_', 0.5)
    #blend(res_bi, 'inverse_bi_', 0.5)
    return linear_blend(res_bi, window_size)

def warp(homography):

    # Transform image 2 with homography matrix
    img2_grid = [[n, m, 1] for n in range(img2.shape[1]) for m in range(img2.shape[0])]
    img2_trans = np.array(np.mat(homography) * np.mat(img2_grid).T)
    img2_trans /= img2_trans[2]

    # Find transformed four corners of image 2 on image 1 coordinate system

    corners = np.zeros((4, 3))
    for p, im in zip(img2_trans.T, img2_grid):
        if im[0] == 0 and im[1] == 0:
            corners[0] = p
        elif im[0] == 0 and im[1] == img2.shape[0] - 1:
            corners[1] = p
        elif im[0] == img2.shape[1] - 1 and im[1] == 0:
            corners[2] = p
        elif im[0] == img2.shape[1] - 1 and im[1] == img2.shape[0] - 1:
            corners[3] = p

    # Blended image size
    global top,left,bottom,r
    top = max(0, math.ceil(-min(corners.T[1])))
    bottom = max(img1.shape[0], math.ceil(max(corners.T[1])))+top
    left = max(0, math.ceil(-min(corners.T[0])))
    right = max(img1.shape[1], math.ceil(max(corners.T[0])))+left
    r = max(img1.shape[1], math.ceil(min(corners[2][0],corners[3][0])))+left

    #Forward warping
    #forward_warp(bottom, right, img2_trans, img2_grid)

    # Inverse warping
    return inverse_warp(bottom, right, corners, homography)
```

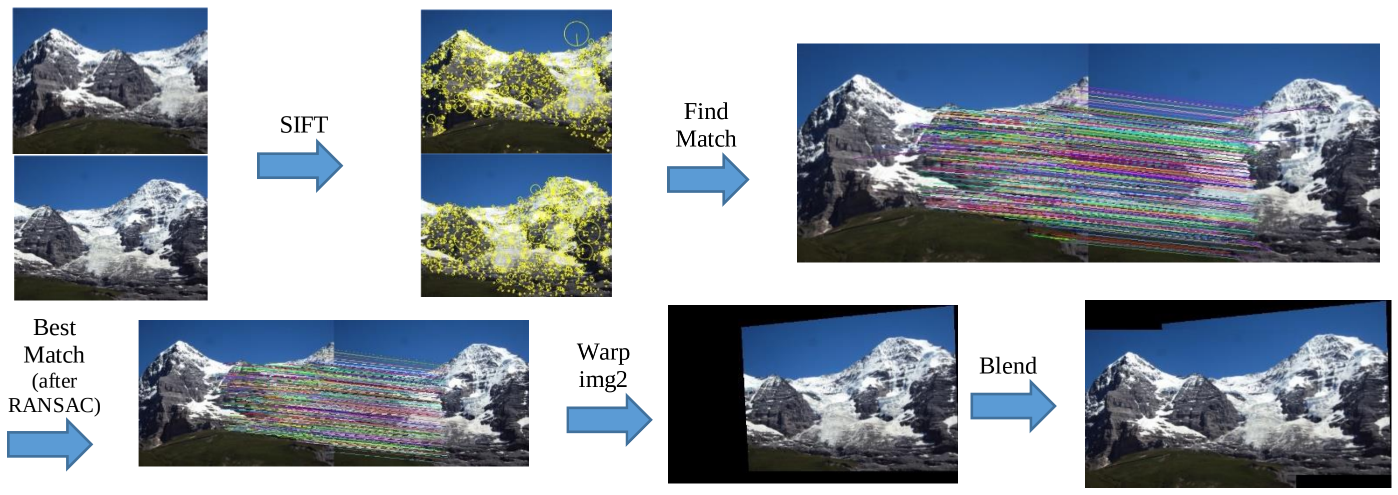
E. Blending function

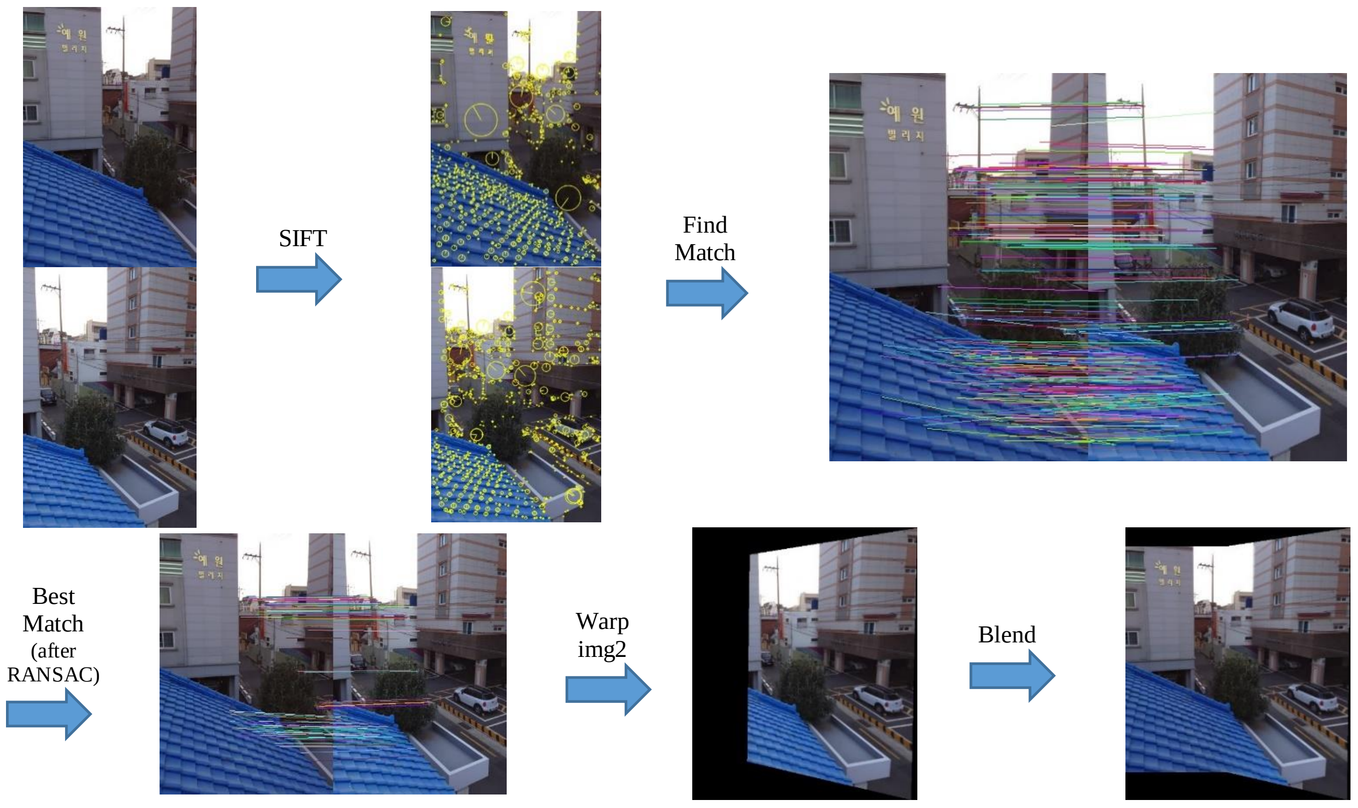
Ideally each sample (pixel) along a ray would have the same intensity in every image that it intersects, but in reality, this is not the case. Even after gain compensation some image edges are still visible due to a number of unmodelled effects, such as vignetting (intensity decreases towards the edge of the image), parallax effects due to unwanted motion of the optical center, mis-registration errors due to mismodelling of the camera, radial distortion and so on. Because of this, a good blending strategy is important. Here we use the linear blending technique with a flexible window size.

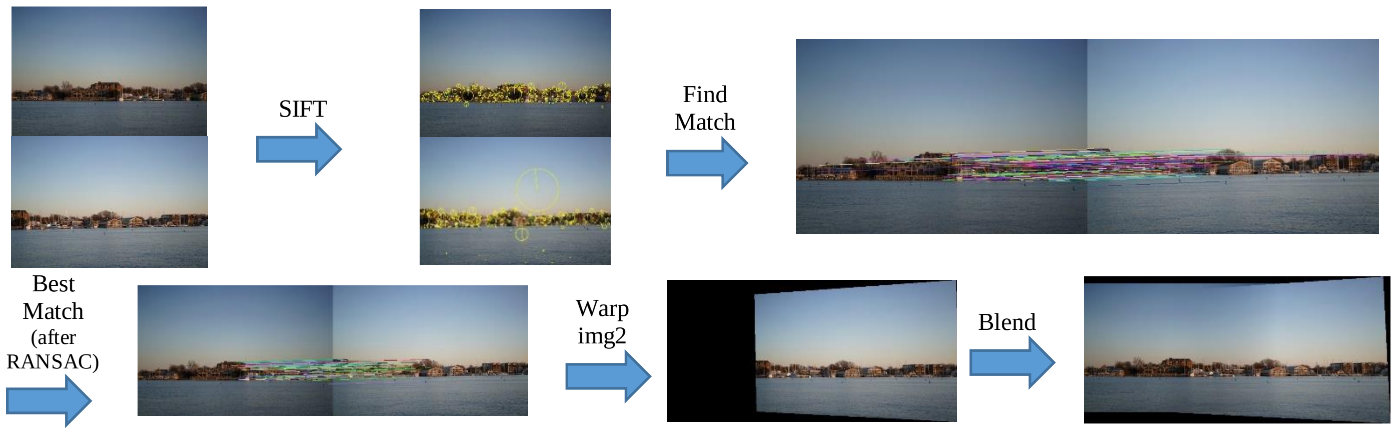
```python
def linear_blend(res, window_size=0):
    res = res.copy()
    alpha = step_a = window = step_w = 0
    for m in range(img1.shape[1]):
        alpha += step_a
        window += step_w
        for n in range(img1.shape[0]):
            if sum(res[n+top, m+left]) != 0:
                if window==0:
                    step_w = 1/(img1.shape[1]-m)
                    window+=step_w
                if window > window_size and alpha==0:
                    step_a = 1/(img1.shape[1]-m)
                    alpha+=step_a
                res[n+top, m+left] = alpha * res[n+top, m+left] + (1 - alpha) * img1[n, m]
            else:
                res[n+top, m+left] = img1[n, m]
    cv2.imwrite('./results/warp/linear_window_warp_'+img_name+'.jpg', res)
    return res

def blend(res, name, alpha=0.5):
    res = res.copy()
    for m in range(img1.shape[1]):
        for n in range(img1.shape[0]):
            if sum(res[n+top, m+left]) != 0:
                res[n+top, m+left] = alpha * res[n+top, m+left] + (1 - alpha) * img1[n, m]
            else:
                res[n+top, m+left] = img1[n, m]
    cv2.imwrite('./results/warp/warp_'+name+img_name+'.jpg', res)
```

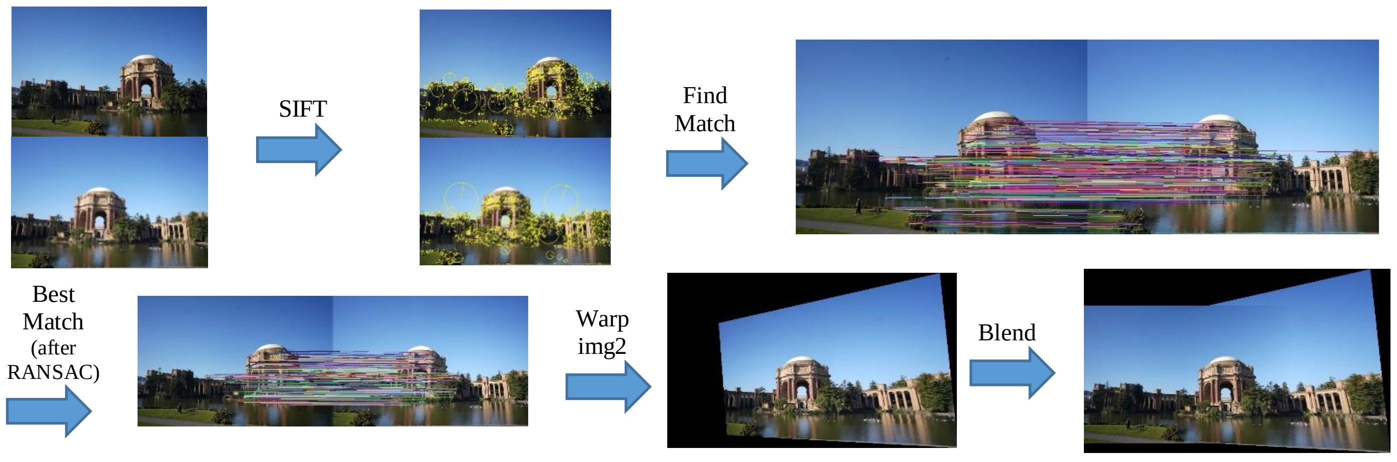
Experiment Result

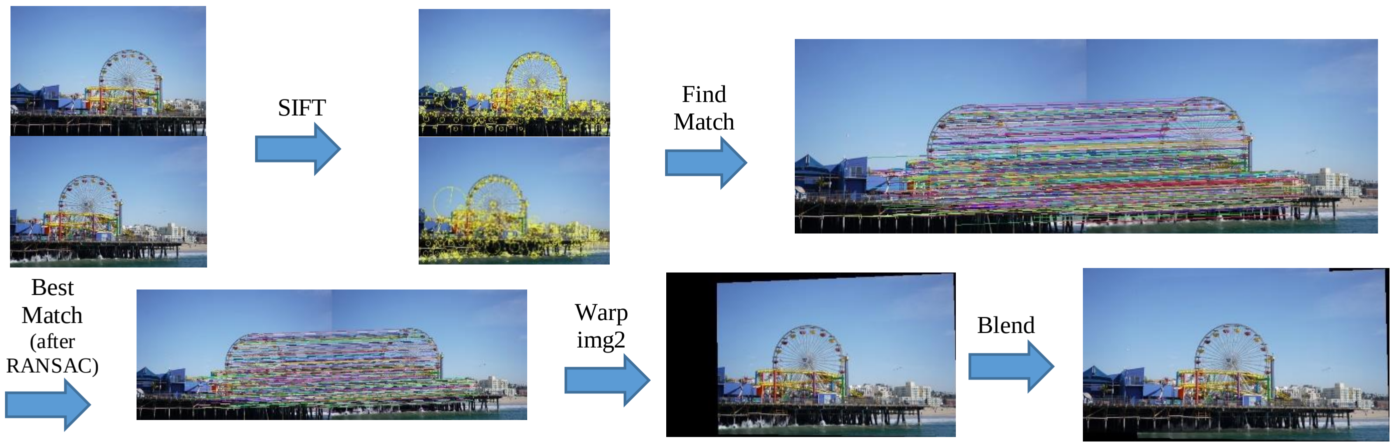




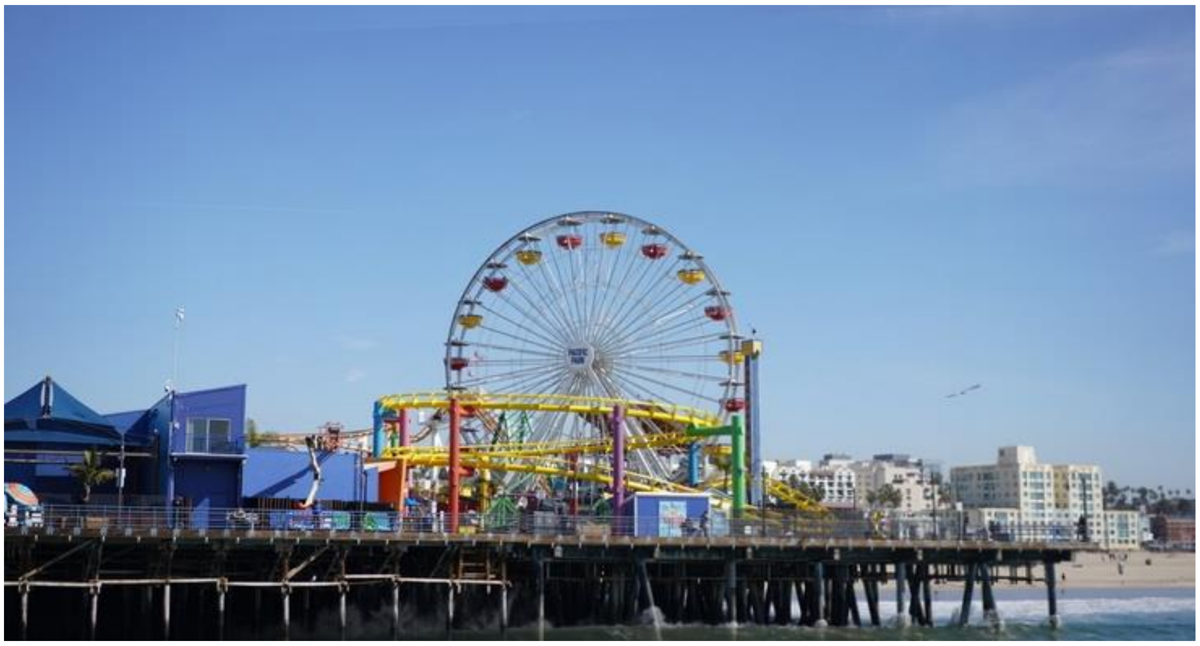

SIFT
Find Match
Best Match (after RANSAC)
Warp img2
Blend

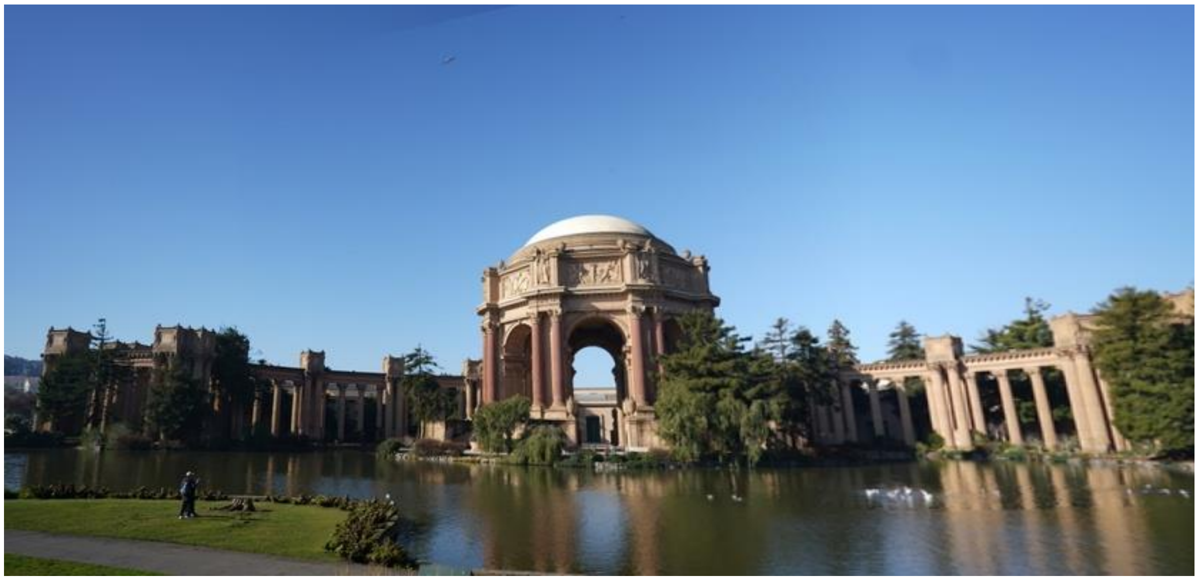

SIFT
Find Match
Best Match (after RANSAC)
Warp img2
Blend


SIFT
Find Match
Best Match (after RANSAC)
Warp img2
Blend



Cropped Result (Big Picture)





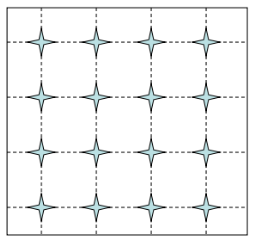
예원
빌리지

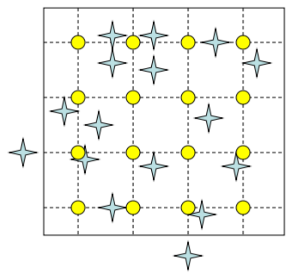
Discussion

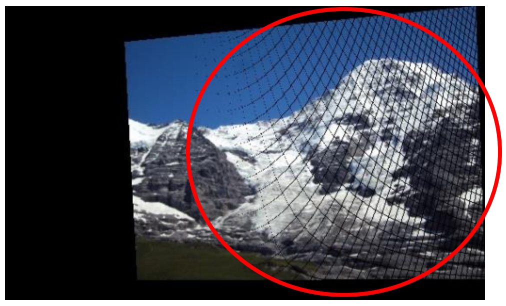
1. Forward & Inverse Warping

A. Forward Warping

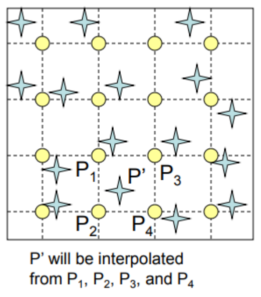
Directly transform image 2 to the coordinate system of image 1. If the result is not an integer, take the nearest integer as result. Since the warping points are often non-integer sample, many samples “o” are **not assigned values**.

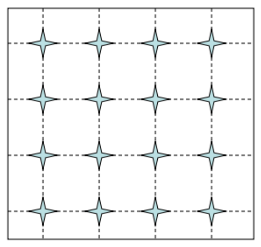






B. Inverse Warping





Using inverse warping, we can avoid the problem in forward warping since every point in the new coordinate system (image1) has a value. The different interpolation technology will get different result. We use **bilinear interpolation** for better result.

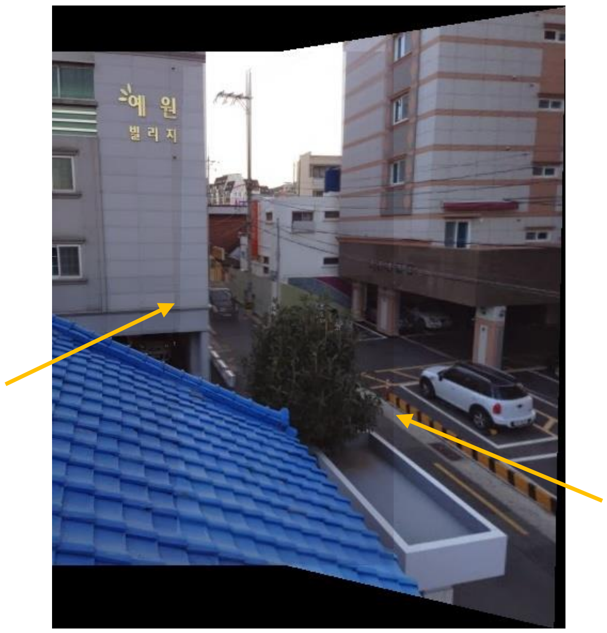
Nearest-Neighbor Interpolation (TOP) looks more **crufty** than Bilinear Interpolation (BOT)

2. Blending

There are many ways to blend image1 and image2:

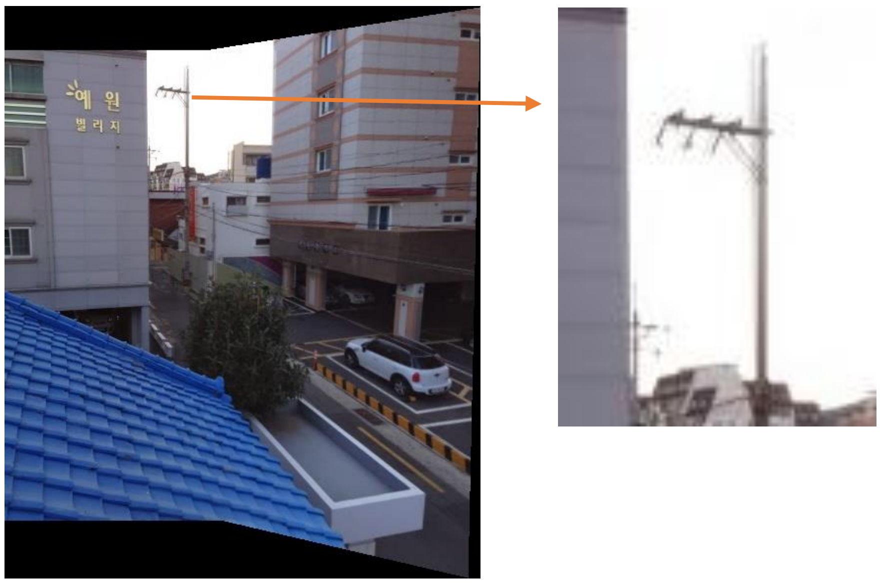
A. Naïve Blending

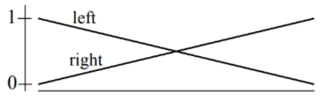
For $\alpha \in [0,1]$, new image $= \alpha * \text{image1} + (1-\alpha) * \text{image2}$. The result is not good since there are obvious edges in the intersection of two images.



B. Linear Blending

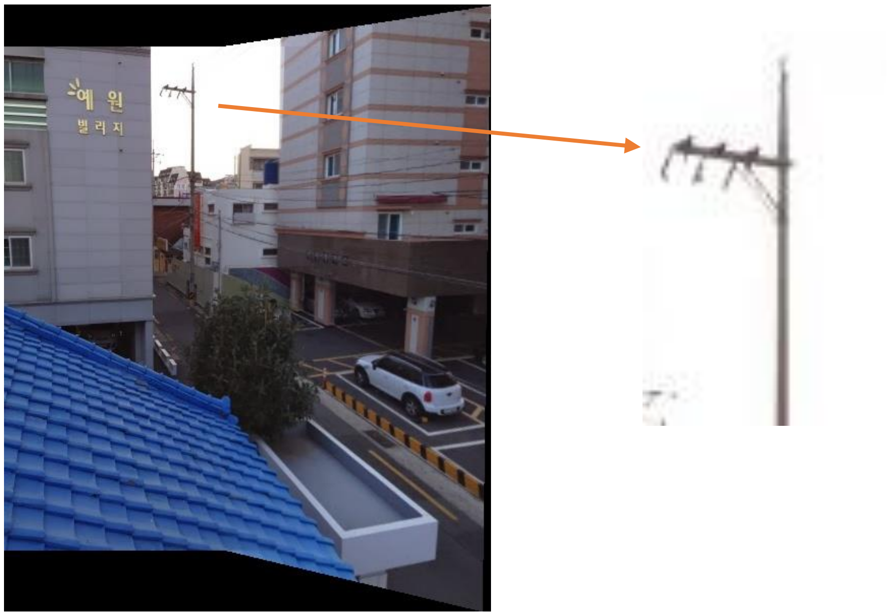
Linearly feather the intersection part of two image. The result is better now since the edges dispear. But there still some problem.

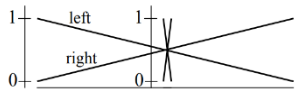




C. Linear Blending with a window

Take a closed look at the pole in the result of the linear blending, there is a **ghosting problem** since two images are not perfectly matched. We can narrow the blending part by using a proper window to solve this problem.





3. Comparison with CV2 built-in function

In this assignment, we were manually performing keypoint detection, feature matching, giving us access to the homography matrix used to warp our two input images into a panorama.

And while OpenCV's built-in cv2.createStitcher and cv2.Stitcher_create functions are certainly capable of constructing accurate, aesthetically pleasing panoramas, one of the primary drawbacks of the method is that it abstracts away any access to the homography matrices.

One of the assumptions of real-time panorama construction is that the scene itself is not changing much in terms of content. Once we compute the initial homography estimation we should only have to occasionally recompute the matrix.

Not having to perform a full-blown key-point matching and RANSAC estimation gives us a tremendous boost of speed when building our panorama, so without access to the raw homography matrices, it would be challenging to take OpenCV's built-in image stitching algorithm and convert it to real-time.

Conclusion

In this assignment, we learned how to perform multiple image stitching using OpenCV and Python. Using both OpenCV and Python we were able to stitch multiple images together and create panoramic images. First, we implement feature matching with interest points detection and feature description by SIFT. Then, we warp images to create panoramic image after establishing homography matrix H according to RANSAC. Finally, our output panoramic images were not only accurate in their stitching placement but also aesthetically pleasing as well.

Reference

1. https://www.pyimagesearch.com/2018/12/17/image-stitching-with-opencv-and-python/
2. https://kushalvyas.github.io/stitching.html
3. http://matthewalunbrown.com/papers/ijcv2007.pdf
4. https://blog.csdn.net/zhangjunhit/article/details/83088627

Work Assignment Plan Between Team Members

0516044 陳思妤	SIFT + Feature Matching
0786031 廖俊凱	RANSAC + Homography
0856733 黃明翰	Wrap + Blending