Computational Photography Homework 3
– Panorama Image Generation

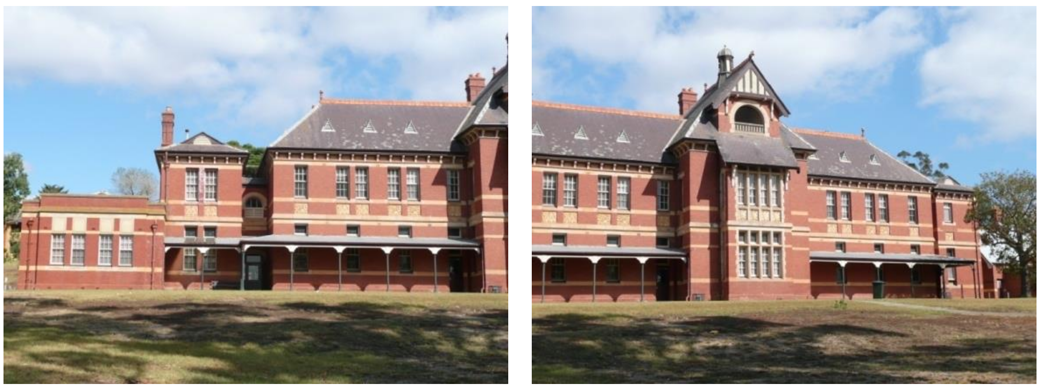
20170243 SimJaeYoon

1. Overview

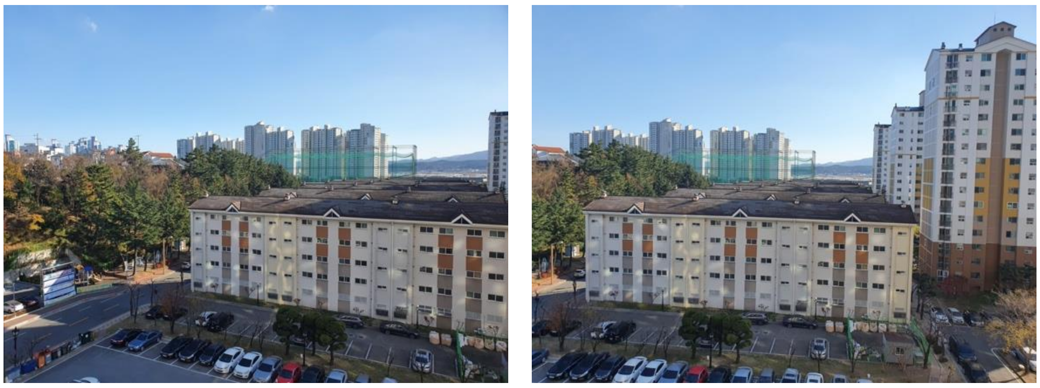
This assignment 3 is about automatic panorama stitching algorithm. I should be able to take not only two but an arbitrary number of images (N>=2) and generate a panorama image from them. So what I need to implement is the following panorama pipeline.

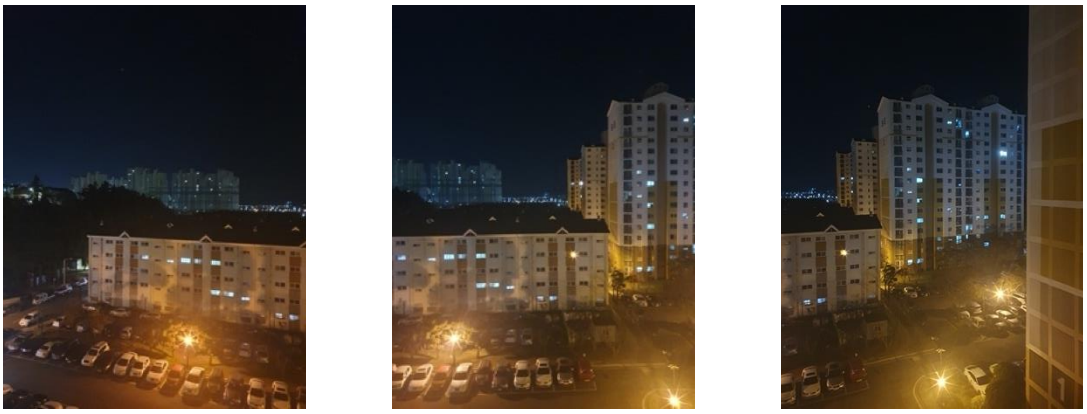
① Given N input images, set one image as a reference
② Detect feature points from images and correspondences between pairs of images
③ Estimate the homographies between images using RANSAC
④ Warp the images to the reference image
⑤ Composite them with blending

I used the same version of the library as python 3.7.11, numpy 1.20.3, opencv 3.4.2.16 to implement this assignment. And I used visual studio code as IDE. I used my test images as follow.



My Test Image Set 1


My Test Image Set 2 (My House View)


My Test Image Set 3 (My House Night View)

2. Implements and Results

The following is a function that contains the overall process of creating a panorama image. I made the process of creating a panorama image easy to manage by making it into one function as follows.

```python
# Overall process of making panorama
def make_panorama(src_image1, src_image2):
    image1 = src_image1.copy()
    image2 = src_image2.copy()

    homography_matrix = get_homography(image1, image2, "ORB")

    warped_image = warp_image(image1, image2, homography_matrix)

    stitched_image = stitch_images_with_blending(image1, warped_image)

    panorama_result = crop_black_area(stitched_image)

    return panorama_result
```

And whenever I made a panorama image, I managed it as a list. So the following function is used to invoke the final result.

```
# Get final panorama image result
def get_panorama(src_list):
    dst_image = src_list[-1]
    return dst_image
```

2.1. Given N input images, set one image as a reference

From now on, I will explain the process of creating a panorama image. I tried to make a panorama image by implementing the functions according to the pipelines I learned in class.

```
### Load input images
input_folder = 'image3'
input_path = '../images/' + input_folder
input_images_name = sorted(glob.glob(input_path + '/*.jpg'))
input_images_num = len(input_images_name)

input_images = []
for idx in range(0, input_images_num):
    image = cv2.imread(input_images_name[idx], cv2.IMREAD_COLOR)
    image = cv2.resize(image, dsize = (400, 600))
    input_images.append(image)
```

First of all, I brought several pictures that would be used to create panoramic images. The imported images were also used by modifying the image size for high speed. This is because the resolution of photos taken with smartphones is very high. Of course, there are also results made because it takes a little time according to the original resolution. I tried to manage the images as a list.

```
### Get panorama result image
stitched_images = []
for idx in range(0, input_images_num):
    image = input_images[idx]

    if(idx == 0):
        stitched_images.append(image)
    else:
        panorama_image = make_panorama(stitched_images[idx - 1], image)
        stitched_images.append(panorama_image)
```

Panorama images will be created using images one by one from the list of images stored through repetition statements. It is to set one as a reference image and proceed with a kind of calculation with a neighboring image.

```
### Save the result
output_image = get_panorama(stitched_images)
output_path = '../images/results/'
cv2.imwrite(output_path + input_folder + '_result.jpg', output_image)
```

This is a function of storing a panorama image in a separate folder after it is created.

2.2. Detect feature points from images and correspondences between pairs of image

The panorama image is a feature-based image alignment. First, feature detection should be performed to find feature points for two given images. The feature points to be found here must be accurate and reliable. So the feature point must satisfy the following properties.

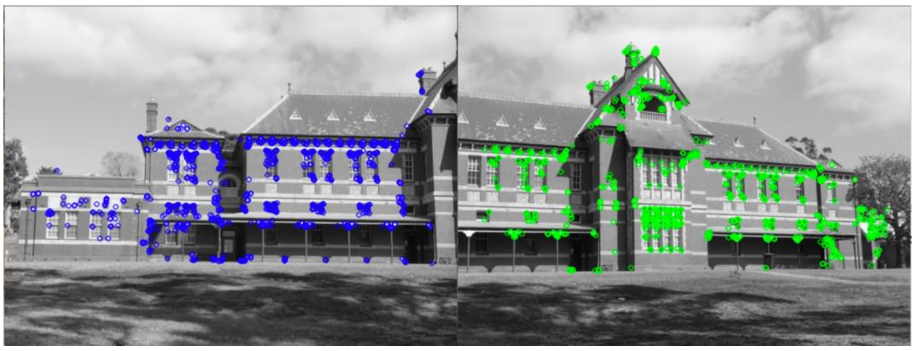
1. Found in other images: We want a feature point that images often naturally have, for example, edges or corners.
2. Found precisely - well localized: Let's say we detected two points above a particular edge from an image. It is difficult to say whether this feature point is in the same position. Because the other points present on the edge are seemingly almost identical. Corner, on the other hand, has a feature that is easier to localize. So it can be said that the points in the corner are in the same position. So these points can be matched.
3. Found reliably - well matched: We want feature points to be reliable. For example, if we have a very dark image with a low contrast. In this image, image features such as corner may or may not be detected depending on noise or lighting condition. Therefore, feature detection will not be reliable.

```
# Detect and describe the key points from the image
def detect_and_describe(src_image, method):
    image = src_image.copy()

    if method == 'SIFT':
        descriptor = cv2.xfeature2d.SIFT_create()
    elif method == 'SURF':
        descriptor = cv2.xfeature2d.SURF_create()
    elif method == 'BRISK':
        descriptor = cv2.BRISK_create(1000)
    elif method == 'ORB':
        descriptor = cv2.ORB_create(1000)

    (keypoints, descriptors) = descriptor.detectAndCompute(image, None)

    return (keypoints, descriptors)
```

I will find features and explain what features the corresponding point has. Therefore, the function was implemented to use SIFT, SURF, BRISK, and ORB, which are widely used as feature detectors and descriptors. Using each method, we will find features and description of them from the image, and try to use the key points and feature descriptor later. The feature point I wanted to find could be modified by hand, and I set it with a goal of about 1000.

```
# Create matcher
def create_matcher(method):
    if method == 'SIFT' or method == 'SURF':
        bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    elif method == 'BRISK' or method == 'ORB':
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

    return bf
```

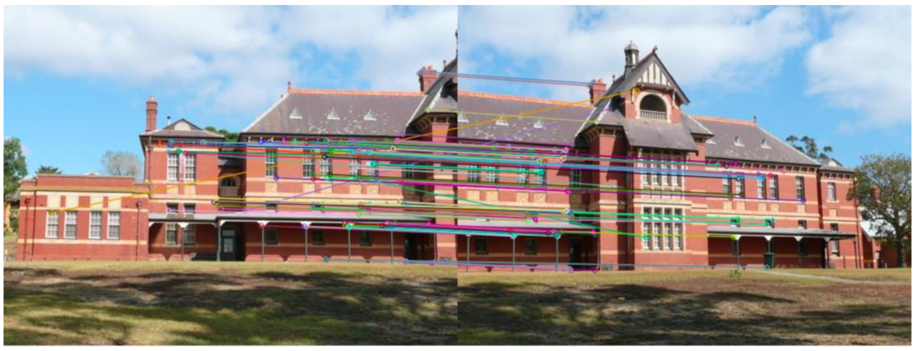
After finding features, we have to know which feature in source image corresponds to feature point in target image. I used brute-force algorithm to find corresponding pairs, which is BFMatcher in OpenCV. The following image deliberately shows features in different colors by transforming the original image into gray scale. As can be seen from the results, it can be seen that the edge or corner was found as a feature.



Feature Detection and Descriptors

```
# Match the keypoints from two images
def match_keypoints(bf, descriptors1, descriptors2, method):
    if method == 'SIFT' or method == 'SURF':
        initial_matches = bf.match(descriptors1, descriptors2)
        good_matches = sorted(initial_matches, key = lambda x:x.distance)
    elif method == 'BRISK' or method == 'ORB':
        initial_matches = bf.knnMatch(descriptors1, descriptors2, 2)
        good_matches = []
        ratio = 0.75
        for first, second in initial_matches:
            if first.distance < second.distance * ratio:
                good_matches.append(first)

    return good_matches
```

Now, the feature points found as above must be matched between those that match each other. For the matched results, the knnMatch function is useful when we return the most matching k by k defined by the user and do something additional. Here, we set it to 2 and set a certain ratio for the two matching, so among the selected matching results, nothing too absurd was used.



Feature Matching (Source Image(L) / Target Image(R))

Looking at the above results, there are many outliers. They interfere with calculating homography in the next step. To this end, the next step attempts to accurately find homography matrix by implementing RANSAC algorithm directly.

2.3. Estimate the homographies between images using RANSAC

The following is the pseudocode of RANSAC to obtain homography.

```
==============================================================
For iter in range(0,N):
        Randomly select 4 feature pairs
        Compute homography H
        Compute inliers where $SSD(p_i', Hp_i) < \epsilon$
        Keep largest set of inliers
Re-compute least-squares H estimate on all of the inliers
```

We repeat the entire process N times, and we randomly choose four feature point pairs for each iteration. This is because four correspondences are required to obtain homography. For four pairs, homography can be calculated and the number of inliers can be counted. After a total of N iterations, the inlier has the most included set, and the optimal homography H can be calculated by solving the last-square program from this set.

```python
# Compute homography from 4 correspondences
def get_homography(src_image1, src_image2, method):
    image1 = src_image1.copy()
    image2 = src_image2.copy()
    image1_gray = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    image2_gray = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

    keypoints1, descriptors1 = detect_and_describe(image1_gray, method)
```

```
keypoints2, descriptors2 = detect_and_describe(image2_gray, method)
"""
featured_image1 = cv2.drawKeypoints(image1_gray, keypoints1, descriptors1, (255, 0, 0))
featured_image2 = cv2.drawKeypoints(image2_gray, keypoints2, descriptors2, (0, 255, 0))

cv2.imshow('image1', featured_image1)
cv2.imshow('image2', featured_image2)
cv2.waitKey(0)
cv2.destroyAllWindows()
"""
bf = create_matcher(method)
matches = match_keypoints(bf, descriptors1, descriptors2, method)
"""
feature_matched_image = cv2.drawMatches(image1, keypoints1, image2, keypoints2, matches, None, flags = 2)
cv2.imshow('feature_matched_image', feature_matched_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
"""

N = 1000

matches_num = len(matches)
max_inlier = 0
inlier_candidate = []

# RANSAC algorithm
for iteration in range (0, N):
    # Select randomly 4 feature pairs
    index = random.sample(range(0, matches_num), 4)
    src = np.float32([keypoints1[matches[i].queryIdx].pt for i in index])
    dst = np.float32([keypoints2[matches[i].trainIdx].pt for i in index])

    # Compute homography H by using 4 selected random pairs
    H, status = cv2.findHomography(src, dst, method = 0)
    inliers = 0
    del inlier_candidate[:]

    # Compute inliers where SSD is smaller than epsilon
    for match in matches:
        src_pt = np.empty((3,1), dtype = np.float32)
        src_pt[0] = np.float32(keypoints1[match.queryIdx].pt[0])
        src_pt[1] = np.float32(keypoints1[match.queryIdx].pt[1])
        src_pt[2] = 1
```

```
            dst_pt = np.empty((3,1), dtype = np.float32)
            dst_pt[0] = np.float32(keypoints2[match.trainIdx].pt[0])
            dst_pt[1] = np.float32(keypoints2[match.trainIdx].pt[1])
            dst_pt[2] = 1

            exp_pt = np.matmul(H, src_pt)
            exp_pt = exp_pt / exp_pt[2]

            ssd = math.sqrt(np.sum((exp_pt - dst_pt) ** 2))
            if (ssd < 1.0):
                inliers = inliers + 1
                inlier_candidate.append(match)

        # Keep largest set of inliers
        if (inliers > max_inlier):
            max_inlier = inliers
            inlier_final = copy.copy(inlier_candidate)

    # Re-compute least-squares H estimate on all of the inliers
    src_final = np.float32([keypoints2[match.trainIdx].pt for match in inlier_final])
    dst_final = np.float32([keypoints1[match.queryIdx].pt for match in inlier_final])
    homography_matrix, status = cv2.findHomography(src_final, dst_final, method = 0)

    return homography_matrix
```

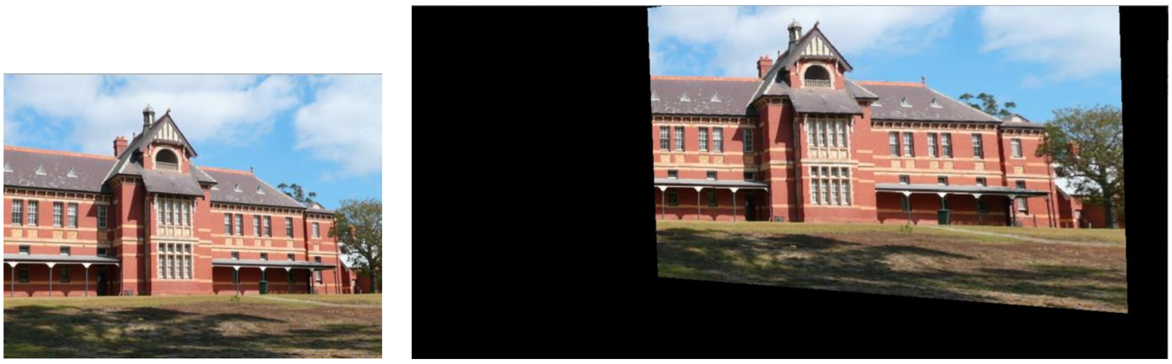
We can find out some most of pairs look similar, but some ourliers are observed. So, we need to remove outliers by finding out which homography can represent image transformation. We will use RANSAC algorithm which is explaine above. I implemented RANSAC algorithm without using a function provide by image processing library. Using 4 randomly selected points, we can find homography first. And using this homogray to count inliers whose SSD between expected and real location in target image. So, we will be able to use largest inliers set to recalculate homography.

2.4. Warp the images to the reference image

After finding the homography, the image must be warped to create a panorama image. This is because it becomes a natural connection when it is attached later.

```
# Warp the second image by using homography matrix
def warp_image(src_image1, src_image2, homography_matrix):
    image1 = src_image1.copy()
    image2 = src_image2.copy()
    h1, w1 = image1.shape[:2]
    h2, w2 = image2.shape[:2]

    warp_result = cv2.warpPerspective(image2, homography_matrix, (w1 + w2, h1 + h2), cv2.INTER_CUBIC, 0)

    return warp_result
```

The desired image is warped using the homography matrix obtained from the two images. And the results are as follows.


Warp Target Image (Before Warping(L) / After Warping(R))

2.5. Composite them with blending

This is the last step in the process of creating a panorama image. We can attach the modified image together, and it is important to note that the process of attaching it should not be unnatural. That is, if the values of the boundary line or pixel are compared when the images are connected, it seems to be connected unnaturally. So, in the process of combining images, we will naturally connect the images using blending techniques, and in particular, I tried to create Gaussian pyramids and Laplacian pyramids to use multi-band blending techniques.

```
# Find center value of overlapping area of two input images
def find_overlap_center(src_image1, src_image2):
    image1 = src_image1.copy()
    image2 = src_image2.copy()

    image1_gray = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    _, image1_mask = cv2.threshold(image1_gray, 0, 255, cv2.THRESH_BINARY)
    image2_gray = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    _, image2_mask = cv2.threshold(image2_gray, 0, 255, cv2.THRESH_BINARY)

    h1, w1 = image1.shape[:2]
    h2, w2 = image2.shape[:2]

    right_edge = 0
    left_edge = w2

    for y in range(0, h1):
        for x in range(0, w1):
            if image1_mask[y, x] == 255:
                if right_edge < x:
                    right_edge = x

    for y in range(0, h2):
        for x in range(0, w2):
            if image2_mask[y, x] == 255:
                if left_edge > x:
                    left_edge = x

    dst_center = int(left_edge + right_edge / 2)

    return dst_center

# Fine left most edge value of overlapping area of two input images
def find_left_overlap_edges(src_image1, src_image2):
    image1 = src_image1.copy()
    image2 = src_image2.copy()

    image1_gray = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    _, image1_mask = cv2.threshold(image1_gray, 0, 255, cv2.THRESH_BINARY)
    image2_gray = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    _, image2_mask = cv2.threshold(image2_gray, 0, 255, cv2.THRESH_BINARY)

    h1, w1 = image1.shape[:2]
    h2, w2 = image2.shape[:2]

    left_edge = w2

    for y in range(0, h1):
        for x in range(0, w1):
            if image2_mask[y, x] == 255:
                if left_edge > x:
                    left_edge = x

    dst_left = left_edge

    return dst_left

# Find right most edge value of overlapping area of two input images
```

```
def find_right_overlap_edges(src_image1, src_image2):
    image1 = src_image1.copy()
    image2 = src_image2.copy()

    image1_gray = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    _, image1_mask = cv2.threshold(image1_gray, 0, 255, cv2.THRESH_BINARY)
    image2_gray = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    _, image2_mask = cv2.threshold(image2_gray, 0, 255, cv2.THRESH_BINARY)

    h1, w1 = image1.shape[:2]
    h2, w2 = image2.shape[:2]

    right_edge = 0

    for y in range(0, h1):
        for x in range(0, w1):
            if image1_mask[y, x] == 255:
                if right_edge < x:
                    right_edge = x

    dst_right = right_edge

    return dst_right
```

The above functions are auxiliary functions to be used for image blending. Image blending is used to remove the original seam and adjust the pixel values that contrasts severely between the two images. The multi-band blending I implemented reduces and increases the resolution of the image in the process of creating Gaussian pyramid and Laplacian pyramid. In this case, even if the interpolation technique is used, the image may appear blurred as a whole.

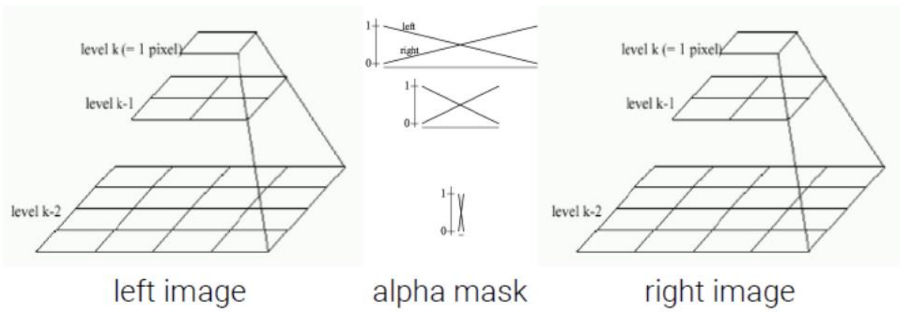
So, the idea is to use the original image for the image of the surrounding outer part, assuming that the part connecting between the two images is naturally changed using multi-band blending. So, in order to set the range, a kind of masking technique was introduced to use the original image except for the overlapping part between the two images.

```
# Blend unnatural overlapping parts of two input images by using laplacian pyramid
def stitch_images_with_blending(src_image1, warped_image):
    image1_t = src_image1.copy()
    image2 = warped_image.copy()

    h1, w1 = image1_t.shape[:2]

    image1 = np.zeros(image2.shape, dtype = np.uint8)
    image1[0:h1, 0:w1] = image1_t
```

```
# Generate Gaussian pyramid
image1_gaussian_pyramid = []
image2_gaussian_pyramid = []
center_x = []

image1_gaussian_pyramid.append(image1)
image2_gaussian_pyramid.append(image2)

image1_down = image1.copy()
image2_down = image2.copy()

center = find_overlap_center(image1_t, image2)
center_r = center

center_x.append(center)

# Level of pyramid
level = 2
level_ = level - 1

for i in range(1, level):
    image1_down = cv2.pyrDown(image1_down)
    image2_down = cv2.pyrDown(image2_down)

    image1_gaussian_pyramid.append(image1_down)
    image2_gaussian_pyramid.append(image2_down)

    center = find_overlap_center(image1_down, image2_down)
    center_x.append(center)

# Generate Laplacian pyramid
image1_laplacian_pyramid = []
image2_laplacian_pyramid = []

image1_laplacian_pyramid.append(image1_gaussian_pyramid[level_])
image2_laplacian_pyramid.append(image2_gaussian_pyramid[level_])

for i in range(level_, 0, -1):
    image1_up = cv2.pyrUp(image1_gaussian_pyramid[i])
    image2_up = cv2.pyrUp(image2_gaussian_pyramid[i])

    hg1, wg1 = image1_gaussian_pyramid[i - 1].shape[:2]
    hg2, wg2 = image2_gaussian_pyramid[i - 1].shape[:2]
    cv2.resize(image1_up, (hg1, wg1), image1_up, interpolation=cv2.INTER_CUBIC)
    cv2.resize(image2_up, (hg2, wg2), image2_up, interpolation=cv2.INTER_CUBIC)

    image1_sub = cv2.subtract(image1_gaussian_pyramid[i-1], image1_up)
    image2_sub = cv2.subtract(image2_gaussian_pyramid[i-1], image2_up)

    image1_laplacian_pyramid.append(image1_sub)
    image2_laplacian_pyramid.append(image2_sub)

concat_result = []
for i in range(0, level):
    lh1, lw1 = image1_laplacian_pyramid[level_ - i].shape[:2]
    lh2, lw2 = image2_laplacian_pyramid[level_ - i].shape[:2]

    left_image = image1_laplacian_pyramid[level_ - i][0:lh1, 0:center_x[i]]
    right_image = image2_laplacian_pyramid[level_ - i][0:lh2, center_x[i]:lw2]

    concat_image = cv2.hconcat([left_image, right_image])

    concat_result.append(concat_image)

# Reconstruct
reconstruct_set = []
reconstruct_image = concat_result[level_]
reconstruct_set.append(reconstruct_image)

for i in range(1, level):
    reconstruct_image = cv2.pyrUp(reconstruct_set[i - 1])

    h, w = reconstruct_image.shape[:2]
    cv2.resize(concat_result[level_ - i], (h, w), concat_result[level_ - i])

    cv2.add(reconstruct_image, concat_result[level_ - i], reconstruct_image)

    reconstruct_set.append(reconstruct_image)

stitched_result = reconstruct_set[level-i].copy()

w1, h1 = image1.shape[:2]
w2, h2 = image2.shape[:2]

left_result = find_left_overlap_edges(image1_t, image2)
right_result = find_right_overlap_edges(image1_t, image2)
```

```
cv2.resize(stitched_result, image2.shape[:2], stitched_result)

# Crop original images and stitch with reconstrued image
cropped_image1 = image1[0:h1, 0:left_result]
cropped_image2 = image2[0:h2, right_result:w2]

stitched_result[0:h1, 0:left_result] = cropped_image1
stitched_result[0:h2, right_result:w2] = cropped_image2

return stitched_result
```

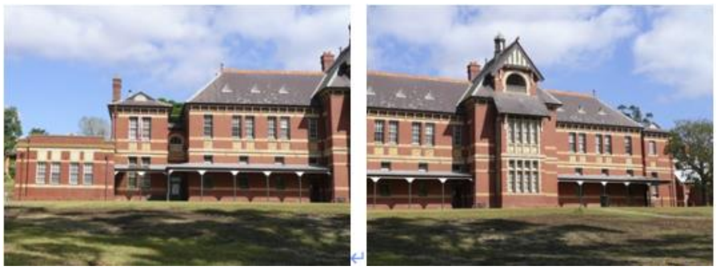
The above process is image blending and composition. The multi-band blending I used emerged to solve the problem of existing alpha blending. In the case of Alpha blending, it is difficult to set the size of the transition area. If this part is set too narrow, it is unnatural, and if it is set too large, a ghosting effect occurs in which transparent objects overlap. Therefore, to solve this phenomenon, multi-band blending was used.

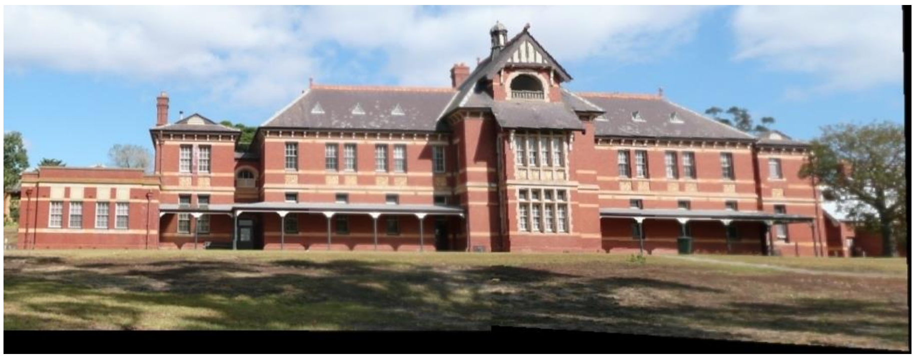


If there is a large scale structure in the two input images, this means that there is a low frequency component, and in this case, if we want to avoid the seam, we should slowly blend or use a wide window. Conversely, when there is a high frequency component corresponding to a structure with a small scale in two input images, it is necessary to quickly blend or use a narrow window to avoid ghosting. Low frequency component means that the scale has a large object, and in this case, a wide transition window should be used to avoid the seam. For small objects, ghosting should be avoided using a narrow transition window. This is the basic idea. When there are two pyramids, the high level corresponds to the low frequency, so the wide window should be used, and the low level corresponds to the high frequency, so the narrow window should be used.

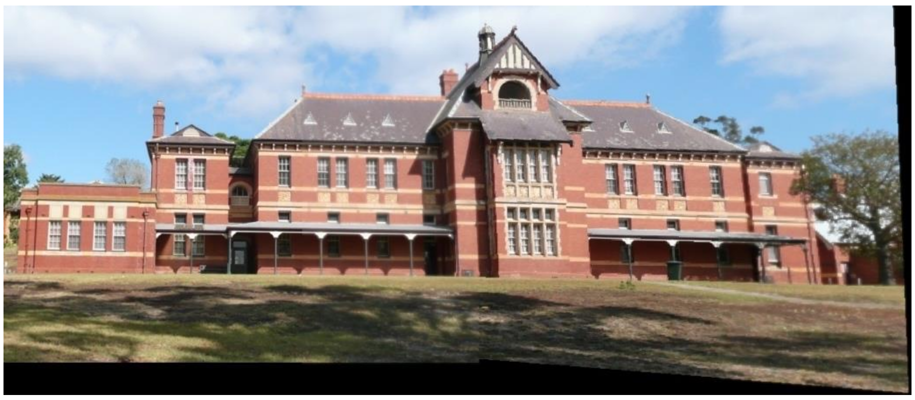
In fact, blending uses Laplacian pyramid. Specifically, in the first step, two Laplacian pyramids are created for each image. And in the second step, blending is performed at each level from the pyramid using a region mask. We need a region mask for each level, which becomes an alpha mask. Finally, to obtain the final blended image, the pyramid can be overlapped.

Let's say there are two images. For each image, we can make a Laplacian pyramid. For each level, two images are blended together using alpha blending. Then, for each level, one blended image comes out. Then we will get a new Gaussian image pyramid from Laplacian pyramid. That's how it created a new image pyramid. And if we continue to add this, we can finally make an output. The following are the results of panoramic images made using my test images. For fast computation of computers, the size of the image was reduced from the beginning and also made with the original size.

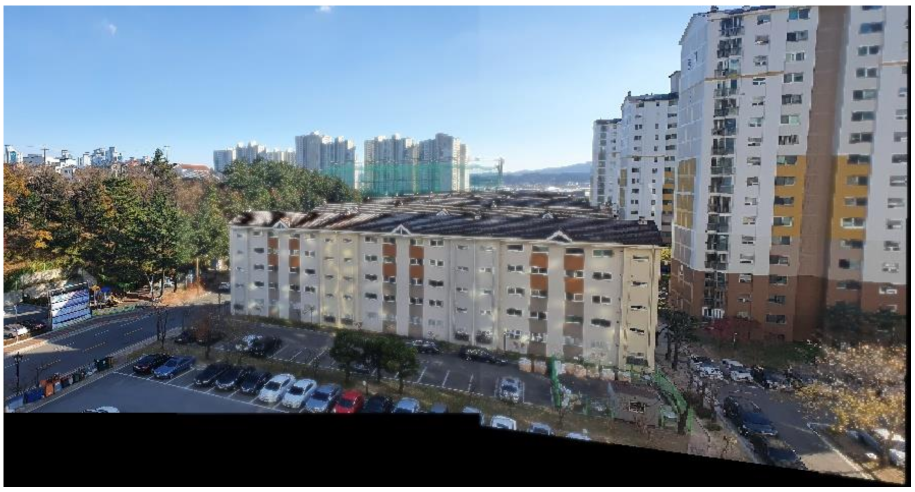


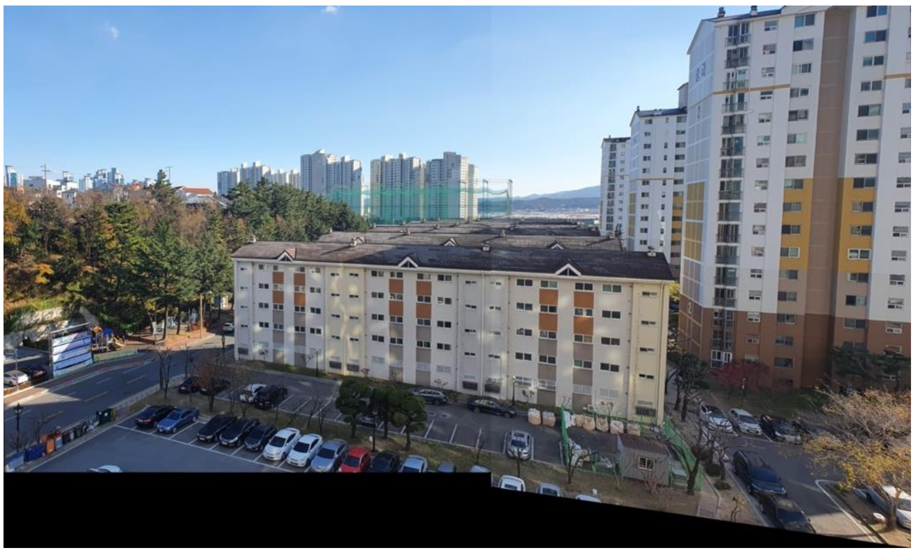


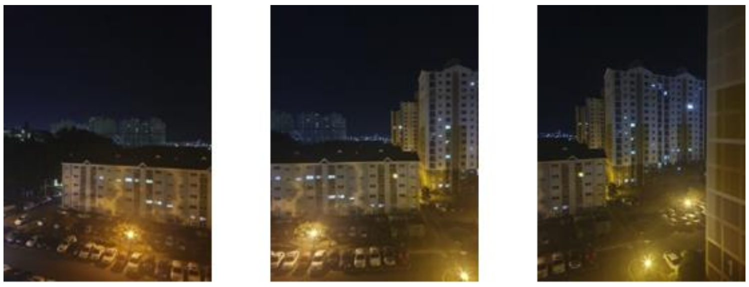
Panorama Result 1 (N = 2, resized)

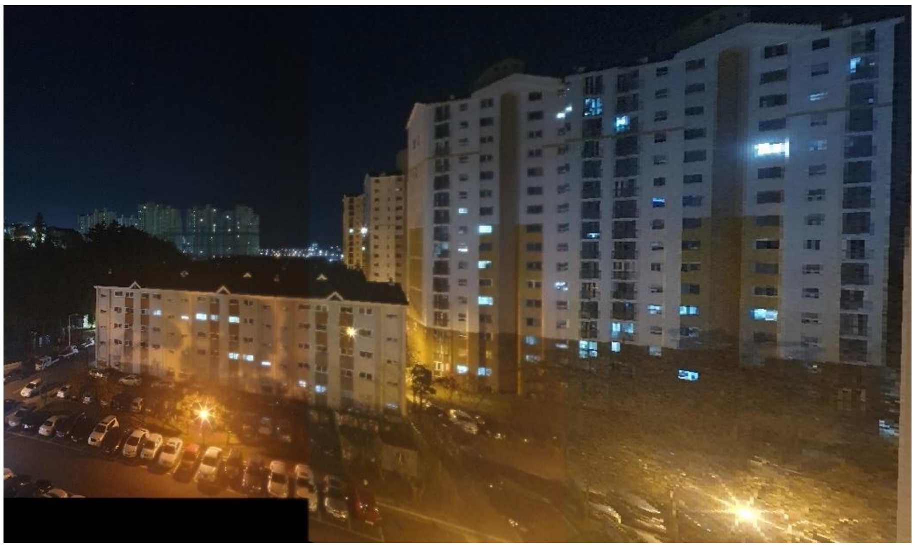


Panorama Result 1 (N = 2)


Panorama Result 2 (N = 2, resized)


Panorama Result 2 (N = 2)

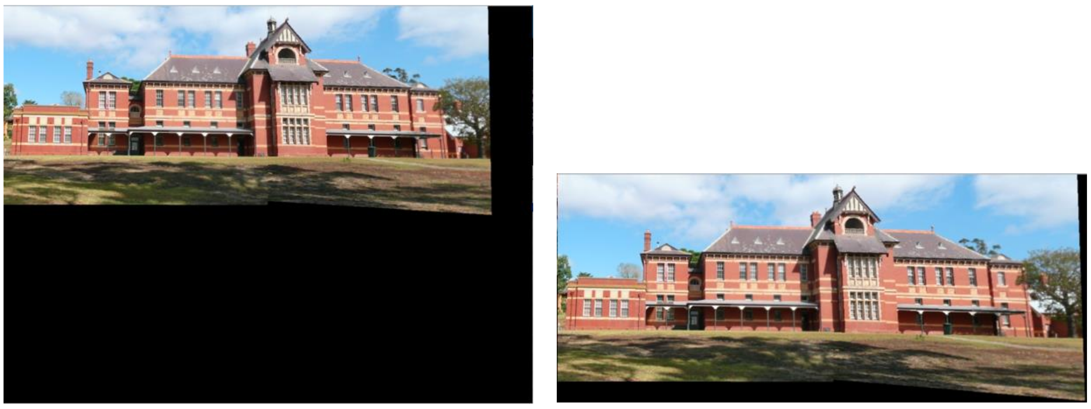




Panorama Result 3 (N = 3)

```
# Remove unnecessary black backgroun after image stithing process
def crop_black_area(src_image):
    image = src_image.copy()
    image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    _, image_mask = cv2.threshold(image_gray, 0, 255, cv2.THRESH_BINARY)

    contours = cv2.findContours(image_mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours = imutils.grab_contours(contours)

    max_contour_area = max(contours, key=cv2.contourArea)
    (x, y, w, h) = cv2.boundingRect(max_contour_area)

    result = image[y:y+h, x:x+w]

    return result
```

Additionally, warping the image creates an unnecessary black background. This is an unnecessary part of the calculation by forcibly increasing the resolution, so we tried to remove the black background whenever we made images in-between. I made the image a binary mask to create a boundary line for the part where the pixel value exists, and based on this, it was implemented as a cutting method.



Crop Black Area (Before Cropping(L) / After Cropping(R))

3. Discussion

This panorama image works properly in a flat field of view with a small number of images. This is because the panorama algorithm created this time simply sets one image as a reference image to obtain homography using correspondences and modify the image to attach it.

The most important problem in this assignment was to implement the RANSAC algorithm without the help of the library and to make blending techniques well so that the team could not be seen. It could have worked well because I focused on these areas, but if the source images are close to the camera or the viewing angle is widened, the distortion will inevitably worsen. Since homography is obtained based on the four feature points, if the viewing angle is widened or the object is close, it will look unnatural even if it is difficult to operate or make during the image stitching process. As a result of the investigation, if the angle is widened or the object is close, the coordinate system can be set differently when creating a panorama image. Now, we have implemented the code so that the seam is not visible using a small image, but if we set the coordinate system with a spherical coordinate or a cylindrical coordinate for a multi-angle panorama image, we can somehow create a panorama image.

And if it is difficult to distinguish feature points in the source image, it seems difficult to create a panorama image. Most of the images have edges or corners, so there is no big difficulty in calculating homography, but if the feature point is not properly visible, it will be difficult to create a panorama image. Perhaps even if we look at the results of matching feature points, there are many outliers. we can use selected matchings such as the RANSAC algorithm, but if we can accurately judge them from the beginning, the results will be better.