Homework 2 : Image Stitching

109550135　范恩宇

I. Implementation :

A. Cylindrical Projection

If we don't do cylindrical projection to the images, it may be distorted like the following. For "Base" part, the left most one could be distorted a lot.

As for cylindrical projection, I extract height & width of the images, create an intrinsic matrix, and get cylindrical coordinates. In the end, warp the images according to cylindrical coordinates we just get.

```python
# function for returning cylindrically-projected imgs
def cylindrical_proj(img):
    h, w = img.shape[:2]
    K = np.array([[1600,0,w/2],[0,1600,h/2],[0,0,1]]) # intrinsic matrix
    h_,w_ = img.shape[:2]
    y_i, x_i = np.indices((h_,w_)) # pixel coordinates

    X = np.stack([x_i,y_i,np.ones_like(x_i)],axis=-1).reshape(h_*w_,3) # to homog
    Kinv = np.linalg.inv(K)
    X = Kinv.dot(X.T).T # normalized coords

    # to get cylindrical coords (sin\theta, h, cos\theta)
    A = np.stack([np.sin(X[:,0]),X[:,1],np.cos(X[:,0])],axis=-1).reshape(w_*h_,3)
    B = K.dot(A.T).T # project back to image-pixels plane
    # back from homog coords
    B = B[:,:-1] / B[:,[-1]]
    # make sure warp coords only within image bounds
    B[(B[:,0] < 0) | (B[:,0] >= w_) | (B[:,1] < 0) | (B[:,1] >= h_)] = -1
    B = B.reshape(h_,w_,-1)

    #img_rgba = cv2.cvtColor(img,cv2.COLOR_BGR2BGRA) # for transparent borders...
    # warp img according to cylindrical coords
    return cv2.remap(img, B[:,:,0].astype(np.float32), B[:,:,1].astype(np.float32), cv2.INTER_AREA, borderMode=cv2.BORDER_TRANSPARENT)
```

The level of image-bending depends on the constant in intrinsic matrix, the smaller the constant, the more it bends. After several trials, 1600 seems to be the best choice.

Constant = 1000	Constant = 1600
	

B. SIFT

Here I just followed the slides, using OpenCV functions to extract SIFT features. In this case, "kp" means OpenCV key points, which stores the point information. While "descript" is an array storing gradient information.

```
# 1.SIFT
print("Step 1: SIFT ...")
kp_left, des_left = SIFT(img_left_g)
kp_right, des_right = SIFT(img_right_g)
```

```
# Step 1: SIFT
def SIFT(img):
    SIFT_Detector = cv2.SIFT_create()
    kp, descript = SIFT_Detector.detectAndCompute(img, None)
    return kp, descript
```

C. KNN Feature Matching

Here I apply KNN algorithm (with brutal force) & Lowe's Ratio test to find good matches, then debug with cv2.BFMatcher( ) as the slide suggests. Motivated by "dmatch" in OpenCV which stores match information, I create some dictionaries for the same purpose.

```
#Step 2: KNN feature matching
print("Step 2: KNN Feature Matching ...")
good_match = matcher(kp_left, des_left, kp_right, des_right, 0.7)
matches = get_matches(good_match, kp_left, kp_right)
```

In order to find the least and 2nd least distance, I make two dictionaries(dmatch & dmatch_2nd) store information and use np.linalg.norm( ) to accelerate the calculation.

After getting the distance, do Lowe's Ratio test. In the end, store information of matches in an array and obtain information of it through get_matches( ).

```python
# Step 2: KNN feature matching
def matcher(kp1, des1, kp2, des2, threshold):
    # Brutal Force KNN
    matches = []
    for i in tqdm(range(len(kp1))):
        dmatch = {"distance":1e7, "queryIdx":0, "trainIdx":0} # smallest
        dmatch_2nd = {"distance":1e7, "queryIdx":0, "trainIdx":0} # 2nd smallest
        v1 = des1[i,:]
        for j in range(len(kp2)):
            v2 = des2[j,:]
            dist = 0
            dist = np.linalg.norm(v1-v2)
            if dist < dmatch["distance"]:
                dmatch_2nd["distance"] = dmatch["distance"]
                dmatch_2nd["queryIdx"] = dmatch["queryIdx"]
                dmatch_2nd["trainIdx"] = dmatch["queryIdx"]
                dmatch["distance"] = dist
                dmatch["queryIdx"] = i
                dmatch["trainIdx"] = j
            elif dist < dmatch_2nd["distance"]:
                dmatch_2nd["distance"] = dist
                dmatch_2nd["queryIdx"] = i
                dmatch_2nd["trainIdx"] = j
        matches.append((dmatch, dmatch_2nd))

    good_match = []
    # Lowe's ratio test
    for m,n in matches:# m, n are DMatch
        if m["distance"] < threshold*n["distance"]:
            good_match.append([m])

    return good_match
```

```python
def get_matches(good_match, kp1, kp2):
    # matches [[x1, y1, x1', y1'], ....., [xn, yn, xn', yn']]
    matches = []
    for pair in good_match:
        matches.append(list(kp1[pair[0]["queryIdx"]].pt + kp2[pair[0]["trainIdx"]].pt))  # for self define pair
    matches = np.array(matches)
    return matches
```

D. Homography Matrix

3.Homography

$$A = U\Sigma V^T$$

- Using SVD decomposition to find Least Squares error solution of Ah = 0
- the solution = eigenvector of A^TA associated with the smallest eigenvalue (V stores the eigenvector of A^TA ,Σ stores the singular value (root of eigen value))
  find the smallest number in Σ and H = corresponding vector in V^T
- Remember to normalize h33 to 1

$$\begin{aligned}\mathbf{h} &= (H_{11}, H_{12}, H_{13}, H_{21}, H_{22}, H_{23}, H_{31}, H_{32}, H_{33})^T \\ \mathbf{a}_x &= (-x_1, -y_1, -1, 0, 0, 0, x_2'x_1, x_2'y_1, x_2')^T \\ \mathbf{a}_y &= (0, 0, 0, -x_1, -y_1, -1, y_2'x_1, y_2'y_1, y_2')^T.\end{aligned}$$

$$A = \begin{pmatrix} \mathbf{a}_{x1}^T \\ \mathbf{a}_{y1}^T \\ \vdots \\ \mathbf{a}_{xN}^T \\ \mathbf{a}_{yN}^T \end{pmatrix}.$$

You can multiply a minus to match the form in previous slide

A is a 9 by 9 matrix (It's similar to A in previous slide)

Reference :

$$\begin{bmatrix} \hat{x}_i z_a \\ \hat{y}_i z_a \\ z_a \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

H

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1\hat{x}_1 & -y_1\hat{x}_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2\hat{x}_2 & -y_2\hat{x}_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3\hat{x}_3 & -y_3\hat{x}_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4\hat{x}_4 & -y_4\hat{x}_4 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1\hat{y}_1 & -y_1\hat{y}_1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2\hat{y}_2 & -y_2\hat{y}_2 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3\hat{y}_3 & -y_3\hat{y}_3 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4\hat{y}_4 & -y_4\hat{y}_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = h_{33} \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \hat{x}_3 \\ \hat{x}_4 \\ \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \hat{y}_4 \end{bmatrix}$$

A h b

Through the hints we get from slides, I use np.linalg.svd( ) to solve the least squares problem by pseudo inverse. Of course, also normalize the matrix after getting the first result.

```
# Step 3: RANSAC
def homography_matrix(pairs):
    # pairs: [[(x1, y1), (x1', y1')],[(x2, y2), (x2', y2')] [(x3, y3), (x3', y3')], [(x4, y4), (x4', y4')]]
    rows = []
    for i in range(pairs.shape[0]):
        p1 = np.append(pairs[i][0:2], 1)
        p2 = np.append(pairs[i][2:4], 1)
        row1 = [0, 0, 0, p1[0], p1[1], p1[2], -p2[1]*p1[0], -p2[1]*p1[1], -p2[1]*p1[2]]
        row2 = [p1[0], p1[1], p1[2], 0, 0, 0, -p2[0]*p1[0], -p2[0]*p1[1], -p2[0]*p1[2]]
        rows.append(row1)
        rows.append(row2)
    rows = np.array(rows)
    U, s, V = np.linalg.svd(rows)
    H = V[-1].reshape(3, 3)   # use the last vector(np.linalg.svd has vector sorted in descending order)
    H = H/H[2, 2] # normalize homography matrix
    return H
```

E. RANSAC

Here I basically follow the pseudo code in slide, while the iterations and error threshold are decided through several tests. To get random pair of matches and error value, I also write functions for them.

```
# 3. Homography
print("Step 3: RANSC ...")
inliers, H = ransac(matches, 0.4, 8000)
```

```
def ransac(matches, threshold, iters):
    best_inlier_num = 0
    best_H = np.zeros((3, 3))
    for i in tqdm(range(iters)):
        chosen_points = random_point(matches)
        H = homography_matrix(chosen_points)
        if np.linalg.matrix_rank(H) < 3:# avoid dividing by zero
            continue

        errors = get_error(matches, H)
        inliers = matches[np.where(errors < threshold)[0]]

        inlier_num = len(inliers)
        if inlier_num > best_inlier_num:
            best_inliers = inliers.copy()
            best_inlier_num = inlier_num
            best_H = H.copy()

    print("Inliers/Matches: {}/{}".format(best_inlier_num, len(matches)))
    return best_inliers, best_H
```

```
def random_point(matches, k=4):
    # get k pairs from the matches (used in ransac)
    idx = random.sample(range(len(matches)), k)
    point = [matches[i] for i in idx ]
    return np.array(point)

def get_error(matches, H):
    # matches: [[x1, y1, x1', y1'], ..., [xn, yn, xn', yn']]
    num_points = len(matches)
    total_pair_1 = np.concatenate((matches[:, 0:2], np.ones((num_points, 1))), axis=1) # change (x1, y1) to (x1, y1, 1)
    total_pair_2 = matches[:, 2:4]
    estimate_pair_2 = np.zeros((num_points, 2))
    for i in range(num_points):
        tmp = np.dot(H, total_pair_1[i])
        estimate_pair_2[i] = (tmp/tmp[2])[0:2] # set index 2 to 1 and slice the index 0, 1

    errors = np.linalg.norm(total_pair_2 - estimate_pair_2 , axis=1) ** 2
    return errors
```

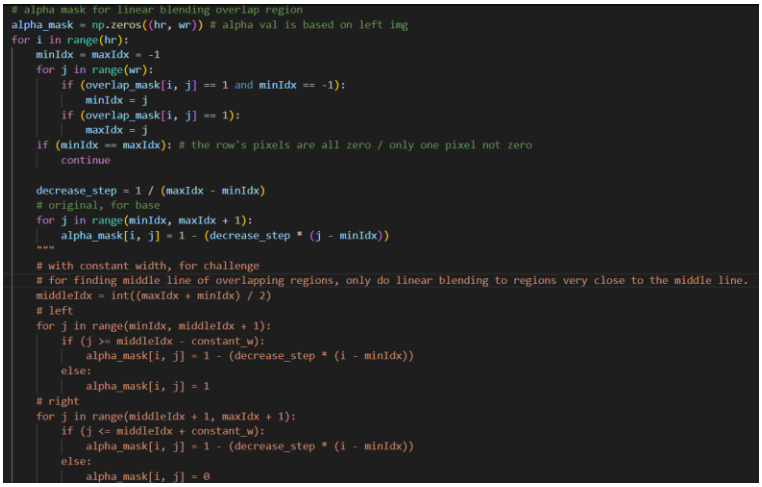
F. Blending Image

When stitching images, I apply linear blending on the images, which means I give each pixel different weights in the overlap region. The corresponding direction weight (left/right) will be larger if the pixel is closer to left/right. And this do remove some boundaries.

So in my implementation, I find left & right image mask region and their overlap mask first.

```
# for blendering the imgs
class Blender:
    def linear_blend(self, imgs):
        img_left, img_right = imgs
        (hl, wl) = img_left.shape[:2]
        (hr, wr) = img_right.shape[:2]
        img_left_mask = np.zeros((hr, wr), dtype="int")
        img_right_mask = np.zeros((hr, wr), dtype="int")
        constant_w = 0.01 # constant width

        # find left & right img mask region(pixels that aren't 0s)
        for i in range(hl):
            for j in range(wl):
                if np.count_nonzero(img_left[i, j]) > 0:
                    img_left_mask[i, j] = 1
        for i in range(hr):
            for j in range(wr):
                if np.count_nonzero(img_right[i, j]) > 0:
                    img_right_mask[i, j] = 1

        # find overlap mask(overlap region of two imgs)
        overlap_mask = np.zeros((hr, wr), dtype="int")
        for i in range(hr):
            for j in range(wr):
                if (np.count_nonzero(img_left_mask[i, j]) > 0 and np.count_nonzero(img_right_mask[i, j]) > 0):
                    overlap_mask[i, j] = 1
        """
```

Then I get alpha mask for linear blending the overlap region, whose alpha value is based on left image. For “Challenge” task, since the original blending method somehow creates more doubled scenes, I check middle lines of overlapping regions and only do linear blending to regions that are close to the middle line (within a custom-constant width).



```
# alpha mask for linear blending overlap region
alpha_mask = np.zeros((hr, wr)) # alpha val is based on left img
for i in range(hr):
    minIdx = maxIdx = -1
    for j in range(wr):
        if (overlap_mask[i, j] == 1 and minIdx == -1):
            minIdx = j
        if (overlap_mask[i, j] == 1):
            maxIdx = j
    if (minIdx == maxIdx): # the row's pixels are all zero / only one pixel not zero
        continue

    decrease_step = 1 / (maxIdx - minIdx)
    # original, for base
    for j in range(minIdx, maxIdx + 1):
        alpha_mask[i, j] = 1 - (decrease_step * (j - minIdx))
    """
    # with constant width, for challenge
    # for finding middle line of overlapping regions, only do linear blending to regions very close to the middle line.
    middleIdx = int((maxIdx + minIdx) / 2)
    # left
    for j in range(minIdx, middleIdx + 1):
        if (j >= middleIdx - constant_w):
            alpha_mask[i, j] = 1 - (decrease_step * (i - minIdx))
        else:
            alpha_mask[i, j] = 1
    # right
    for j in range(middleIdx + 1, maxIdx + 1):
        if (j <= middleIdx + constant_w):
            alpha_mask[i, j] = 1 - (decrease_step * (i - minIdx))
        else:
            alpha_mask[i, j] = 0
```

In the end, do linear blending through things I just get in this part.

```
linear_blend_img = np.copy(img_right)
# linear blending
for i in range(hr):
    for j in range(wr):
        if ( np.count_nonzero(overlap_mask[i, j]) > 0):
            linear_blend_img[i, j] = alpha_mask[i, j] * img_left[i, j] + (1 - alpha_mask[i, j]) * img_right[i, j]
        elif (np.count_nonzero(img_left_mask[i, j])>0):
            linear_blend_img[i, j] = img_left[i,j]
        elif (np.count_nonzero(img_right_mask[i, j])>0):
            linear_blend_img[i, j] = img_right[i,j]

return linear_blend_img
```

G. Image Stitching

In this part, I follow the slide's instruction as well.

```
# Step 4 Stitch Image
print("Step 4: Stiching Image ...")
img_stitched = stitch_img(img_left, img_right, H)

# Change img back to [0, 255] for storing
img_stitched*=255.0
img_stitched = img_stitched.astype(np.uint8)
```

In the beginning, I convert left & right image into double then normalize them for latter calculation. Later, I try to get new corners, sizes, and affine matrix, then get new height and width and do matrix multiplication for those two images by cv2.warpPerspective( ).

For the left image, I want it to fit in with the right one, so I use the height and width of right image for calculating the left one's new shape.

```
# Step 4: Stitich image
def stitch_img(left, right, H):

    # both convert to double & normalize, for avoiding noise.
    left = cv2.normalize(left.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)
    right = cv2.normalize(right.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)

    # left img
    height_l, width_l, channel_l = left.shape
    corners = [[0, 0, 1], [width_l, 0, 1], [width_l, height_l, 1], [0, height_l, 1]]
    corners_new = [np.dot(H, corner) for corner in corners]
    corners_new = np.array(corners_new).T
    x_news = corners_new[0] / corners_new[2]
    y_news = corners_new[1] / corners_new[2]
    y_min = min(y_news)
    x_min = min(x_news)
    y_min = min(y_min, 0)#
    x_min = min(x_min, 0)#
    translation_mat = np.array([[1, 0, -x_min], [0, 1, -y_min], [0, 0, 1]])
    H = np.dot(translation_mat, H)

    # Get h&w of right img for "left" img
    h2, w2 ,c2= right.shape
    y_min = round(y_min)
    x_min = round(x_min)
    height_new = int(round(abs(y_min)) + h2)
    width_new = int(round(abs(x_min)) + w2)
    size = (width_new, height_new)

    warped_l = cv2.warpPerspective(src=left, M=H, dsize=size)
```

For the right image, I basically apply the same procedure but not use height & width of left image for calculation. In the end, apply blender method to the new-stitched image.

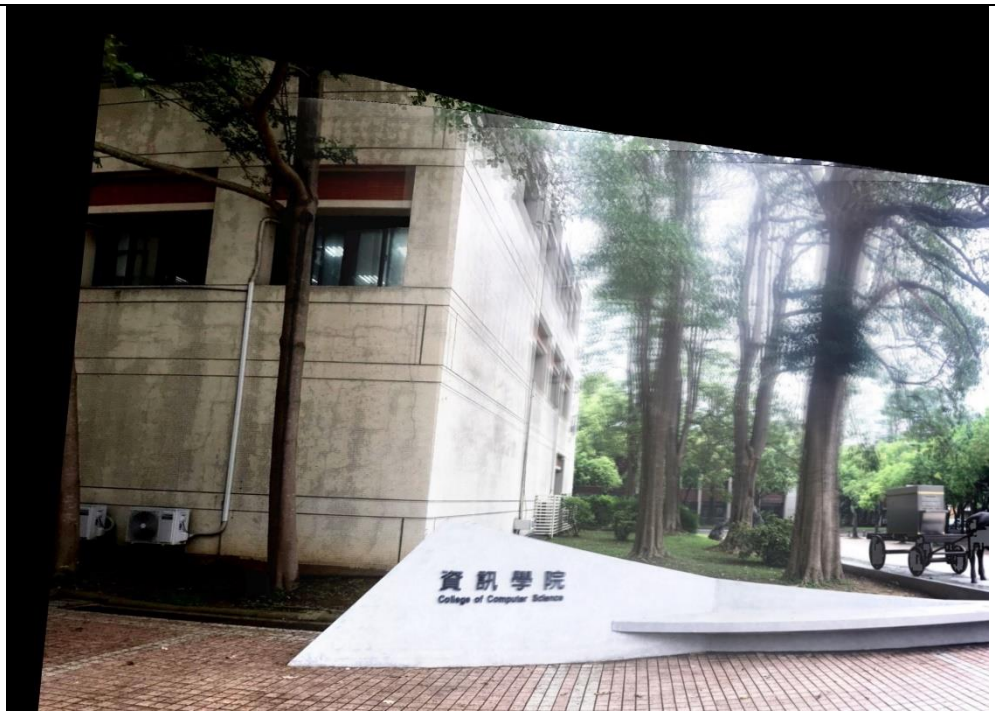
```
# right image
height_r, width_r, channel_r = right.shape
height_new = int(round(abs(y_min) + height_r))
width_new = int(round(abs(x_min) + width_r))
size = (width_new, height_new)

warped_r = cv2.warpPerspective(src=right, M=translation_mat, dsize=size)

# stitching process, results stored in warped_img
blender = Blender()
warped_img= blender.linear_blend([warped_l, warped_r])
```

II. Results :

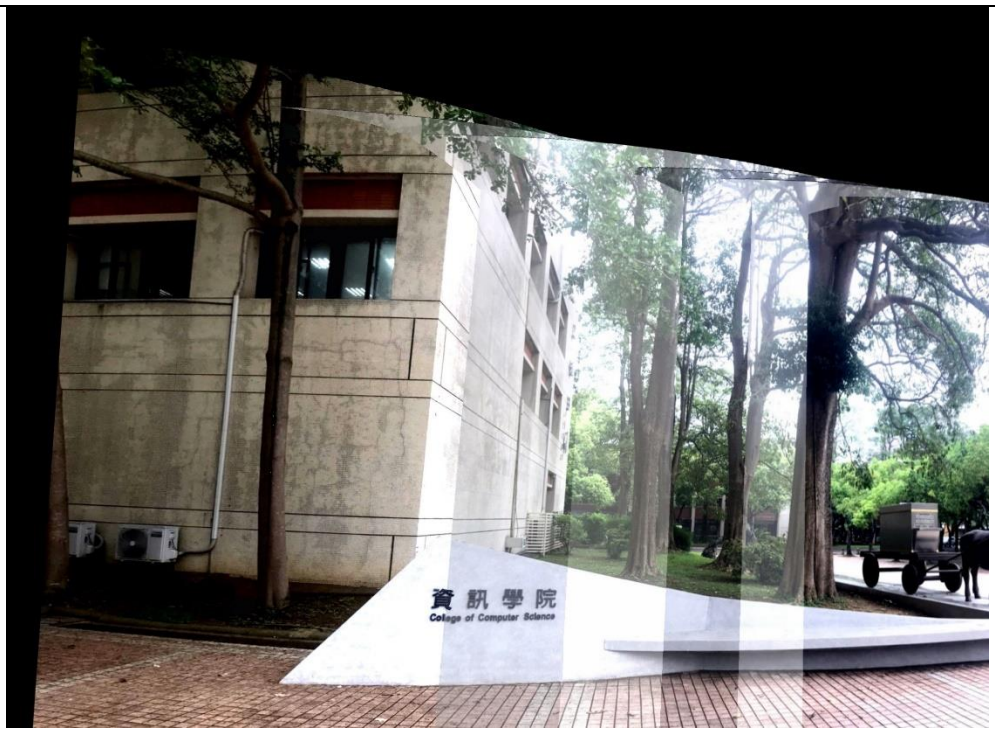
Base

Base (blend with constant width)

Challenge



Challenge (blend with constant width)



III. Discussion(different blending method) :

Boundary Problem

This is the first problem I encountered. At the beginning, I thought of the aliasing problem and tried to solve it with inverse mapping. But after trying blending, not only the problem seems to be solved, I also don't have to do inverse mapping.

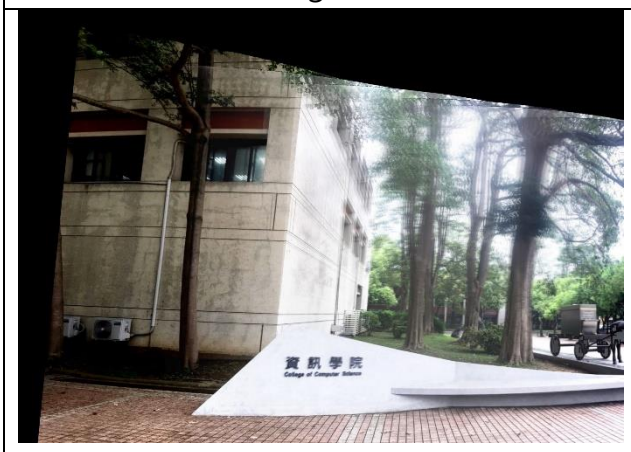
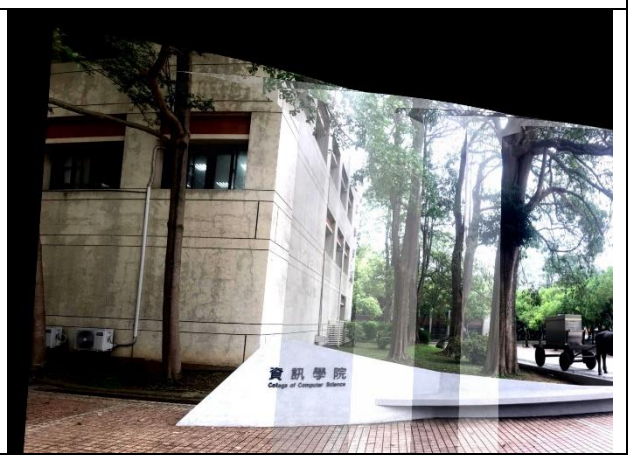
Ghost Problem

Ghost problem means that some scenes are doubled, due to the blending process, which is apparent in "Challenge" task. Here I try "linear blending with constant width" instead, which only blend constant width instead of all the overlapped parts. To implement this, I find the middle line of overlapping regions first, then do linear blending only to regions close to middle line(distance <= constant width).

```
def linear_blend(self, imgs):
    img_left, img_right = imgs
    (hl, wl) = img_left.shape[:2]
    (hr, wr) = img_right.shape[:2]
    img_left_mask = np.zeros((hr, wr), dtype="int")
    img_right_mask = np.zeros((hr, wr), dtype="int")
    constant_w = 0.01 # constant width # 0.01
```

```
# with constant width, for challenge
# for finding middle line of overlapping regions
# only do linear blending to regions very close to the middle line.
middleIdx = int((maxIdx + minIdx) / 2)
# left
for j in range(minIdx, middleIdx + 1):
    if (j >= middleIdx - constant_w):
        alpha_mask[i, j] = 1 - (decrease_step * (i - minIdx))
    else:
        alpha_mask[i, j] = 1
# right
for j in range(middleIdx + 1, maxIdx + 1):
    if (j <= middleIdx + constant_w):
        alpha_mask[i, j] = 1 - (decrease_step * (i - minIdx))
    else:
        alpha_mask[i, j] = 0
```

This method does solve the ghost problem, but it makes the boundary more apparent, since it only blend certain width around the boundary.

Original	With Constant Width
	

Light Problem

For “Challenge”, the brightness of each image differs a lot, which could make the stitching process harder. So I try to make their brightness close to each other.

When reading the images, I also record their brightness then calculate the average brightness of them. In the stitching part, I assign the average brightness to each image part, which somehow makes them stitched better.

```
for i in range(1,img_number+1):
    path = base +f"/Base{str(i)}.jpg"#Base,Challenge{str(i)}
    img, img_gray= read_img(path)
    imgs.append(img)
    img_grays.append(img_gray)
    brightness = cv2.mean(img_gray)[0]
    bright_sum += brightness
```

```
# Step 5 Adjusting brightness
def adjust_brightness(img, target_bright):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    curr_bright = cv2.mean(gray)[0]
    ratio = target_bright / curr_bright
    adjusted_image = cv2.convertScaleAbs(img, alpha=ratio, beta=0)

    return adjusted_image
```