HW2 Image stitching

1. Explain your implementation

In this homework, I follow the architecture provided by TA.

- **Detecting key point(feature) on the images**
  I use the SIFT algorithm to find keypoints and descriptors of images. The method I use to implement is to call function provided by OpenCV. Since SIFT algorithm only can handle gray image, I preprocess them into grayscale before using it.

```python
img1,img_gray1 = read_img(img_path_lists[0])
img2,img_gray2 = read_img(img_path_lists[1])
kts1, dcpt1 = SIFT_detector.detectAndCompute(img_gray1, None)
kts2, dcpt2 = SIFT_detector.detectAndCompute(img_gray2, None)
```

- **Finding features correspondences (feature matching)**
  To find the corresponding key points in a paired image, we utilize the KNN algorithm. We implement it using brute force. Besides, we use Lowe's Ratio test algorithm to filter out some bad matching.(Threshold = 0.75)

```python
def feature_matching_implement(kts1, dcpt1,kts2, dcpt2):

    # Match descriptors
    good_matches = []
    for ii,des1 in enumerate(tqdm(dcpt1)):
        min_value_1_distance = 100000000000001
        min_value_1_idx = -1
        min_value_2_distance = 100000000000000
        min_value_2_idx = -1
        for jj,des2 in enumerate(dcpt2):
            distance = (des1 - des2) * (des1 - des2)
            distance  = sum(distance)
            distance = math.sqrt(distance)
            if(distance < min_value_1_distance):
                min_value_2_distance = min_value_1_distance
                min_value_2_idx = min_value_1_idx
                min_value_1_distance = distance
                min_value_1_idx = jj
            elif(distance < min_value_2_distance and distance >= min_value_1_distance):
                min_value_2_distance = distance
                min_value_2_idx = jj
        if(min_value_1_distance < min_value_2_distance*0.75):
            m = cv2.DMatch()
            m.queryIdx = ii
            m.trainIdx = min_value_1_idx
            m.distance = min_value_1_distance
            m.imgIdx = 0
            good_matches.append(m)

    return good_matches
```

- **Computing homography matrix.**
  Because of the existence of outliers, we utilize the RANSAC algorithm to find an appropriate homography that can transform pixels on image A to imageB. In the RANSAC algorithm, we randomly select four paired points and use them to estimate the homography matrix.

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

After getting an estimated homography matrix, we apply this matrix to all key points to get the size of the consensus set. If this consensus set is larger than ever, we mark this homography matrix as “best homography matrix”. We repeat this process MAX_ITERATION times.

```
best_H = {}
src_pts = np.float32([kts1[m.queryIdx].pt for m in matches]).reshape(-1, 2)
dst_pts = np.float32([kts2[m.trainIdx].pt for m in matches]).reshape(-1, 2)

best_H = None
s_pts = src_pts.copy()
s_pts = np.concatenate((s_pts,np.ones((s_pts.shape[0],1))),axis=1)
d_pts = dst_pts.copy()
d_pts = np.concatenate((d_pts,np.ones((d_pts.shape[0],1))),axis=1)
# Find homography matrix
# H, _ = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC)
max_iteration = 2000
S = -1
final_H = None
for ii in range(max_iteration):
    # Random select 4 points
    points_idx = np.random.choice(src_pts.shape[0], 4, replace=False)
    src_points = src_pts[points_idx]
    dst_points = dst_pts[points_idx]
    # # Start to construct A

    A = []
    for idx in range(len(src_points)):
        A.append([src_points[idx, 0], src_points[idx, 1], 1, 0, 0, 0, -src_points[idx, 0] * dst_points[idx, 0], -src_points[idx, 1] * dst_points[idx, 0], -dst_points[idx, 0]])
        A.append([0, 0, 0, src_points[idx, 0], src_points[idx, 1], 1, -src_points[idx, 0] * dst_points[idx, 1], -src_points[idx, 1] * dst_points[idx, 1], -dst_points[idx, 1]])
    # Solve system of linear equations Ah = 0 using SVD
    u, sigma, vt = np.linalg.svd(A)
    # pick H from last line of vt
    H = np.reshape(vt[8], (3, 3))
    # normalization, let H[2,2] equals to 1
    H = (1/H.item(8)) * H
    inliner_num = 0
    x = s_pts.copy()
    pred = H @ x.T
    pred = pred.T
    for i in range(pred.shape[0]):
        if pred[i,2] <= 1e-8:
            continue
        pred[i] /= pred[i,2]
    criteria = np.linalg.norm(d_pts - pred,axis=1)
    value = np.sum(criteria<5)
    if(value>S):
        S = value
        best_H = H
return best_H
```

- **Stitching image (warp images into same coordinate system)**
  In this section we wrap images into the same coordinate system. We can use “best homography matrix” to transform pixels from image A to image B. However, some pixels from image A may translate to values outside the boundary of image B, either less than 0 or exceeding its own boundary. Thus, we construct another matrix A to ensure that all translated pixel positions will be greater than 0 and determine the size of transformed image by the translated corners of image A.

```python
def stitch_image_imp(final_H,img1, img2,bl,initial):
    H = final_H
    left_down = np.hstack(([0], [0], [1]))
    left_up = np.hstack(([0], [img1.shape[0]-1], [1]))
    right_down = np.hstack(([img1.shape[1]-1], [0], [1]))
    right_up = np.hstack(([img1.shape[1]-1], [img1.shape[0]-1], [1]))

    warped_left_down = H @ left_down.T
    warped_left_up = H @ left_up.T
    warped_right_down =  H @ right_down.T
    warped_right_up = H @ right_up.T

    warped_left_down /= warped_left_down[-1]
    warped_left_up /= warped_left_up[-1]
    warped_right_down /= warped_right_down[-1]
    warped_right_up /= warped_right_up[-1]

    x1 = int(min(min(min(warped_left_down[0],warped_left_up[0]),min(warped_right_down[0], warped_right_up[0])), 0))
    y1 = int(min(min(min(warped_left_down[1],warped_left_up[1]),min(warped_right_down[1], warped_right_up[1])), 0))

    x1_max = int(max(max(max(warped_left_down[0],warped_left_up[0]),max(warped_right_down[0], warped_right_up[0])), img2.shape[1]))
    y1_max = int(max(max(max(warped_left_down[1],warped_left_up[1]),max(warped_right_down[1], warped_right_up[1])), img2.shape[0]))
    size = (abs(x1) + x1_max , abs(y1) + y1_max)
    A = np.float32([[1, 0, -(x1)], [0, 1, -(y1)], [0, 0, 1]])



    warped1 = cv2.warpPerspective(src=img1, M=A@H, dsize=size)
    warped2 = cv2.warpPerspective(src=img2, M=A, dsize=size)

    gain1,gain2 = get_gain_compensations(warped1,warped2)

    for i in range(3):
        warped1[:,:,i] = warped1[:,:,i] * gain1[i]
        warped2[:,:,i] = warped2[:,:,i] * gain2[i]

    if(initial == False):
        result = bl.linearBlendingWithConstantWidth([warped2, warped1])
    else:
        result = bl.linearBlendingWithConstantWidth([warped1, warped2])
    cv2.imwrite("intermediate_result.png",result)
    return result
```

There still remain two questions to solve, one for blendering and the other for gain compensations.

- Blendering:

  Since our homography matrix is not perfect, it is weird if we put two images together directly.

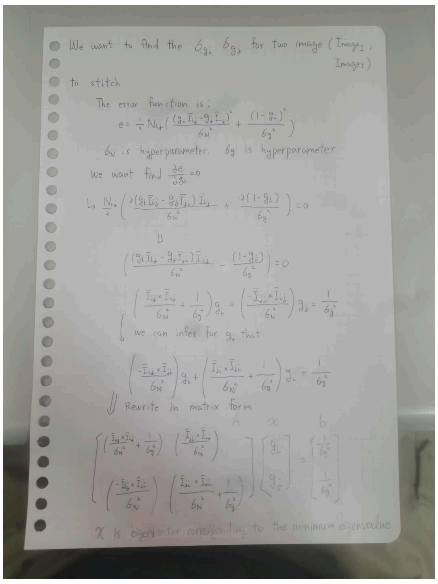
We could use some blendering algorithm to help. The algorithm I use is “linearBlendingWithConstantWidth”. In my implementation, I use the following equation:

$$Image_{result} = \alpha * image_A + (1 - \alpha) * image_B$$

This equation is only applied to a constant width in the overlap area.

```python
def linearBlendingWithConstantWidth(self, imgs):
    '''
    linear Blending with Constat Width, avoiding ghost region
    # you need to determine the size of constant with
    '''
    img_left, img_right = imgs
    (hl, wl) = img_left.shape[:2]
    (hr, wr) = img_right.shape[:2]
    img_left_value = np.sum(img_left,axis=2)
    img_right_value = np.sum(img_right,axis=2)
    img_left_mask = np.zeros((hl, wl), dtype="int")
    img_right_mask = np.zeros((hr, wr), dtype="int")
    constant_width = 3 # constant width

    # find the left image and right image mask region(Those not zero pixels)
    img_left_mask[img_left_value != 0] = 1
    img_right_mask[img_right_value != 0] = 1

    # find the overlap mask(overlap region of two image)
    overlap_mask = np.zeros((hr, wr), dtype="int")
    mask = np.logical_and(img_left_mask!=0,img_right_mask!=0)
    overlap_mask[mask] = 1

    # compute the alpha mask to linear blending the overlap region
    alpha_mask = np.zeros((hr, wr)) # alpha value depend on left image
    for i in tqdm(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): # represent this row's pixels are all zero, or only one pixel not zero
            continue

        decrease_step = 1 / (maxIdx - minIdx)

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



    # linearBlendingWithConstantWidth_img = np.copy(img_right)
    linearBlendingWithConstantWidth_img = np.copy(img_left)
    com = (1 - alpha_mask)
    # linear blending with constant width
    linearBlendingWithConstantWidth_img[overlap_mask>0] = alpha_mask[overlap_mask>0][:,None] * img_left[overlap_mask>0] + \
                                                          com[overlap_mask>0][:,None] * img_right[overlap_mask>0]
    mask_left = np.logical_and(overlap_mask==0,img_left_mask>0)
    mask_right = np.logical_and(overlap_mask==0,img_right_mask>0)
    linearBlendingWithConstantWidth_img[mask_left>0] = img_left[mask_left>0]
    linearBlendingWithConstantWidth_img[mask_right>0] = img_right[mask_right>0]
```

❖ Gain compensations:
The Gain compensations want to solve the problem for the difference of Intensity between two different paired images. I derive the equation that helps me to construct a matrix that optimizes the result.



We want to find the σ_{g_i} σ_{g_j} for two image ($Image_I$, $Image_J$) to stitch

The error function is:

$$e = \frac{1}{2} N_{ij} \left(\frac{(g_i \bar{I}_{ij} - g_j \bar{I}_{ji})^2}{\sigma_N^2} + \frac{(1-g_i)^2}{\sigma_g^2} \right)$$

σ_N is hyperparameter. σ_g is hyperparameter

We want find $\frac{\partial e}{\partial g_i} = 0$

$$\hookrightarrow \frac{N_{ij}}{2} \left(\frac{2(g_i \bar{I}_{ij} - g_j \bar{I}_{ji}) \bar{I}_{ij}}{\sigma_N^2} + \frac{-2(1-g_i)}{\sigma_g^2} \right) = 0$$

$$\Downarrow$$

$$\left(\frac{(g_i \bar{I}_{ij} - g_j \bar{I}_{ji}) \bar{I}_{ij}}{\sigma_N^2} - \frac{(1-g_i)}{\sigma_g^2} \right) = 0$$

$$\left(\frac{\bar{I}_{ij} \times \bar{I}_{ij}}{\sigma_N^2} + \frac{1}{\sigma_g^2} \right) g_i + \left(\frac{-\bar{I}_{ji} \times \bar{I}_{ij}}{\sigma_N^2} \right) g_j = \frac{1}{\sigma_g^2}$$

we can infer for g_j that

$$\left(\frac{-\bar{I}_{ij} \times \bar{I}_{ji}}{\sigma_N^2} \right) g_i + \left(\frac{\bar{I}_{ji} \times \bar{I}_{ji}}{\sigma_N^2} + \frac{1}{\sigma_g^2} \right) g_j = \frac{1}{\sigma_g^2}$$

$\Downarrow$ Rewrite in matrix form

$$\underbrace{\begin{bmatrix} \left(\frac{\bar{I}_{ij} \times \bar{I}_{ij}}{\sigma_N^2} + \frac{1}{\sigma_g^2} \right) & \left(\frac{-\bar{I}_{ji} \times \bar{I}_{ij}}{\sigma_N^2} \right) \\ \left(\frac{-\bar{I}_{ij} \times \bar{I}_{ji}}{\sigma_N^2} \right) & \left(\frac{\bar{I}_{ji} \times \bar{I}_{ji}}{\sigma_N^2} + \frac{1}{\sigma_g^2} \right) \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} g_i \\ g_j \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} \frac{1}{\sigma_g^2} \\ \frac{1}{\sigma_g^2} \end{bmatrix}}_{b}$$

x is eigenvector corresponding to the minimum eigenvalue

Follow the the result of derivation, we can find the best σ_A and σ_B.

```python
def calculateI(image,overlap):
    I = (
        np.sum(
            image * np.repeat(overlap[:, :, np.newaxis], 3, axis=2),
            axis=(0, 1),
        )
        / overlap.sum()
    )
    return I


def get_gain_compensations(
    warp1,warp2,sigma_n: float = 10.0, sigma_g: float = 0.1
):
    """
    Compute the gain compensation for each image, and save it into the images objects.

    Args:
        images: Images of the panorama
        pair_matches: Pair matches between the images
        sigma_n: Standard deviation of the normalized intensity error
        sigma_g: Standard deviation of the gain
    """
    coefficients = []
    results = []
    (hl, wl) = warp1.shape[:2]
    (hr, wr) = warp2.shape[:2]
    img_left_value = np.sum(warp1,axis=2)
    img_right_value = np.sum(warp2,axis=2)
    img_left_mask = np.zeros((hl, wl), dtype="int")
    img_right_mask = np.zeros((hr, wr), dtype="int")
    img_left_mask[img_left_value != 0] = 1
    img_right_mask[img_right_value != 0] = 1
    overlap = np.zeros((hr, wr), dtype="int")
    mask = np.logical_and(img_left_mask!=0,img_right_mask!=0)
    overlap[mask] = 1

    pair_match_I_0 = calculateI(warp1,overlap)
    pair_match_I_1 = calculateI(warp2,overlap)

    coefs = [ np.zeros(3) for _ in range(2) ]
    result = np.zeros(3)
    coefs[0] = (
        (pair_match_I_0 ** 2 / sigma_n ** 2) + (1 / sigma_g ** 2)
    )
    coefs[1] -= (
        (1 / sigma_n ** 2) * pair_match_I_0 * pair_match_I_1
    )
    result += 1 / sigma_g ** 2
    coefficients.append(coefs)
    results.append(result)


    coefs = [ np.zeros(3) for _ in range(2) ]
    result = np.zeros(3)
    coefs[0] -= (
        (1 / sigma_n ** 2) * pair_match_I_0 * pair_match_I_1
    )
    coefs[1] = (
        (pair_match_I_1 ** 2 / sigma_n ** 2) + (1 / sigma_g ** 2)
    )
    result += 1 / sigma_g ** 2
    coefficients.append(coefs)
    results.append(result)



    coefficients = np.array(coefficients)
    results = np.array(results)

    gains = np.zeros_like(results)

    for channel in range(coefficients.shape[2]):
        coefs = coefficients[:, :, channel]
        res = results[:, channel]

        gains[:, channel] = np.linalg.solve(coefs, res)
    images = [warp1,warp2]
    max_pixel_value = np.max([image for image in images])

    if gains.max() * max_pixel_value > 255:
        gains = gains / (gains.max() * max_pixel_value) * 255

    return gains[0],gains[1]
```

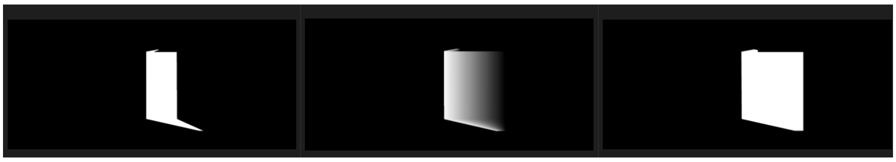
2. Show the result of stitching “Base” images (and “Challenge image” if you did that part).
    - Base images

- Challenge images

3. Discuss different blending method results.

I have tried the following method to blend images.

- Linear blending vs Linear blending with constant width
  The result of these two methods is not significant. However, Linear blending with constant width can save lots of computation resources compared to Linear blending. Because Linear blending with constant width only constructs the alpha part in constant width instead constructing in the whole overlap region.

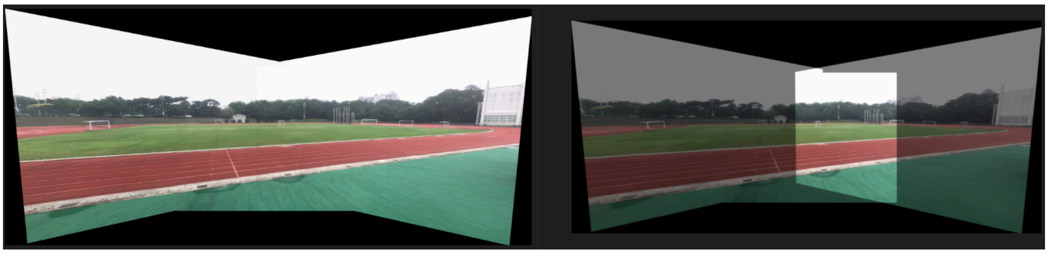


The left is the alpha map in “Linear blending with constant width”. The middle part is the alpha map in “Linear blending”. The right part is the overlap region of two images.

The result of Linear blending and Linear blending with constant width. The left one is Linear blending. The right one is Linear blending with constant width.

- Simple blending vs Multiband blending
  Compared to simple blending, multi band blending needs more computational time and resources. Besides, the result of multi band blending is more blur. However, the result of multi band blending is more seemless.

This is the result of simple blending. The left is the whole result. The right part focuses on the overlap region.



This is the result of multiband blending. The left is the whole result. The right part focuses on the overlap region.