Homework 2

NYCU Computer Vision 2024 Spring

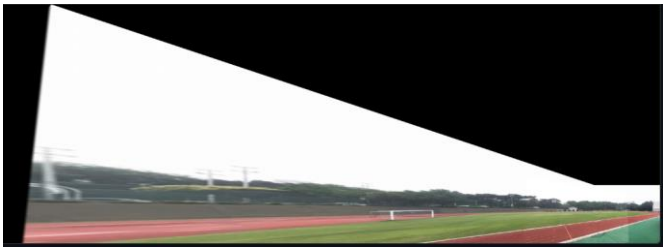
Student ID : 312605001 / Student Name : 歐庭維

A. Image Stitching

1. Base task

Stitch 2 image

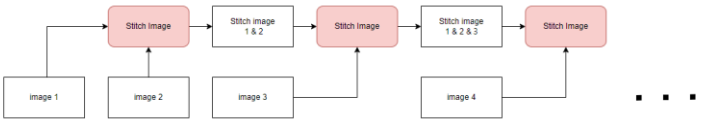
Stitch 3 image



2. Challenging task

B. My Complementation

1. Stitching Pipline



2. SIFT

I use OpenCV “SIFT_create” to get the key point position and its descriptor in this part.

```
def find_sift_kp_and_des( img_gray ):
    SIFT_Descriptor = cv.SIFT_create()
    kp, des = SIFT_Descriptor.detectAndCompute( img_gray, None )

    return kp, des
```

3. KNN

In my first trial, I use “cv.BFMatcher / knnMatch” to find the same kp between two image.

```
# Create BFMatcher object with cross check
bf = cv.BFMatcher()

# Match descriptors
matches = bf.knnMatch(self.descriptor1,self.descriptor2, k=2)

good = []
for m,n in matches:
    if m.distance < 0.75*n.distance:
        good.append( (m.trainIdx, m.queryIdx) )


print("There are ", len(good), 'Points with good match')
```

I also tried to implement the knn matcher by myself. The threshold remains the same, and the output stitched image look the same.

```
def knn_find_good_match(self, threshold=0.75):
    # KNN
    print("KNN matching ...")
    k_matches = []
    for i,d1 in enumerate( self.descriptor1 ):
        min_kp = [-1,np.inf]
        sec_min_kp = [-1,np.inf]
        for j,d2 in enumerate( self.descriptor2 ):
            dist = np.linalg.norm(d1 - d2)
            if min_kp[1] > dist:
                sec_min_kp = np.copy(min_kp)
                min_kp = [j,dist]
            elif sec_min_kp[1] > dist:
                sec_min_kp = [j,dist]
        k_matches.append((min_kp,sec_min_kp))
    # ratio test
    print("Ratio test ...")
    matches = []
    for i,(m1,m2) in enumerate(k_matches):
        # print("index : {}".format(i),m1,m2)
        # print(m1[1] , threshold*m2[1])
        if m1[1] < threshold*m2[1]:
            # unpacking the tuple to let one match stores 4 element (p1.x , p1.y , p2.x , p2.y)
            # It doesn't mean summoing up two position
            matches.append(list( self.keypoint1[i].pt + self.keypoint2[m1[0]].pt))
    return np.array(matches)
```

4. Homography matrix

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

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

After obtaining good matches, the next step is to calculate the homography matrix between two images. It's important to remember to standardize the homography matrix before returning it.

```python
def find_homography(self, pairs):
    A = np.zeros((8,9))
    array_a = 0
    array_b = 4
    for i in range(pairs.shape[0]):

        x1   = pairs[i][0]
        y1   = pairs[i][1]
        x1_b = pairs[i][2]
        y1_b = pairs[i][3]

        A[array_a][0] = x1
        A[array_a][1] = y1
        A[array_a][2] = 1
        A[array_a][6] = -x1*x1_b
        A[array_a][7] = -y1*x1_b
        A[array_a][8] = -x1_b

        A[array_b][3] = x1
        A[array_b][4] = y1
        A[array_b][5] = 1
        A[array_b][6] = -x1*y1_b
        A[array_b][7] = -y1*y1_b
        A[array_b][8] = -y1_b

        array_a += 1
        array_b += 1

    U, s, V = np.linalg.svd(A)   # h = the eigenvector of ATA associated with the smallest eigenvalue -> the last one
    H = V[-1].reshape(3, 3)
    H = H/H[2, 2]     # standardize

    return H
```

5. RANSAC

```python
def find_error(self, points, H):
    num_points = points.shape[0]
    all_p1 = np.column_stack((points[:, :2], np.ones(num_points)))
    all_p2 = points[:, 2:4]

    temp = np.dot(H, all_p1.T)
    estimate_p2 = (temp[:2] / temp[2]).T

    errors = np.linalg.norm(all_p2 - estimate_p2, axis=1) ** 2

    return errors

def ransac_find_best_H(self, good_matches, ransac_threshold=5, iteration=1000):
    num_best_inliers = 0
    for i in range( iteration ):
        # get random for pairs
        pairs = np.array( [ good_matches[idx] for idx in random.sample(range(len(good_matches)), 4) ] )
        H = self.find_homography(pairs)

        errors = self.find_error(good_matches, H)
        inliers = good_matches[ np.where(errors < ransac_threshold)[0] ]

        num_inliers = len(inliers)
        if num_inliers > num_best_inliers:
            best_inliers = inliers.copy()
            num_best_inliers = num_inliers
            best_H = H.copy()

    print("inliers/matches: {}/{}".format(num_best_inliers, len(good_matches)))

    return best_inliers, best_H
```

Due to numerous misaligned pairs, I resort to using RANSAC (Random Sample

Consensus) to find the best homography matrix. After conducting several trials, I found that setting the parameters "ransac_threshold=5" and "iteration=1000" consistently yields stable results in each case.

6. Cylinder projection

After many trial, I have determined that setting the focal length to 4000 produces the best results in this particular context.

```python
def cylinder_projection( img, focal=4000 ):
    h, w = img.shape[:2]
    cylinder_projection_img = np.zeros_like( img )
    center_x, center_y = w//2, h//2

    for y in range( h ):
        for x in range( w ):
            theta = ( x - center_x ) / focal
            h_dash = ( y - center_y ) / np.cos( theta ) + center_y
            x_proj = int( np.sin(theta) * focal + center_x )
            y_proj = int( h_dash )
            if 0 <= x_proj < w and 0 <= y_proj < h:
                cylinder_projection_img[y, x] = img[y_proj, x_proj]

    return cylinder_projection_img
```

Focal = 1200	Focal = 2200
	

7. Crop image

I stitch each image from left to right, so the left image will be distorted. So I crop the distorted part after stitching each images.

```python
def crop_img( img, crop_ratio=1):
    height, width = img.shape[:2]

    crop_width = int(width * crop_ratio)
    crop_height = int(height * crop_ratio)

    start_x = width - crop_width
    start_y = height - crop_height

    cropped_img = img[start_y:height, start_x:width]

    return cropped_img
```

8. Remove Black Border

To address the sharp edge at the jointed place when stitching two images together, I employ the "medianBlur" technique to smooth out the transition and create a more seamless blend.

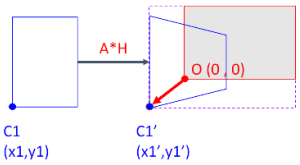
```
def removeBlackBorder( img ):
    return cv.medianBlur(img, 5)
```

9. Stitching IMG

In this part, I use “cv.warpPerspective” to transfer the image by the best homography matrix and stitching the left and right image together.

Affine translation matrix (A) = $\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$

In this case $\Delta x = -x1^{'}, \Delta y = -y1^{'}$
Because origin moves to down left,
image2 needs move to top right relatively




```
def stitch_img(left, right, H):

    # Convert to double and normalize. Avoid noise.
    left = cv.normalize(left.astype('float'), None, 0.0, 1.0, cv.NORM_MINMAX)
    # Convert to double and normalize.
    right = cv.normalize(right.astype('float'), None, 0.0, 1.0, cv.NORM_MINMAX)

    # left image
    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       # [wx', wy', w]
    x_news = corners_new[0] / corners_new[2]    # x' = wx'/w
    y_news = corners_new[1] / corners_new[2]    # y' = wy'/w
    y_min = min(y_news)
    x_min = min(x_news)

    translation_mat = np.array([[1, 0, -x_min], [0, 1, -y_min], [0, 0, 1]])
    H = np.dot(translation_mat, H)

    # Get height, width
    height_new = int(round(abs(y_min) + height_l))
    width_new = int(round(abs(x_min) + width_l))
    size = (width_new, height_new)

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

    # 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 = cv.warpPerspective(src=right, M=translation_mat, dsize=size)

    black = np.zeros(3)  # Black pixel.

    # Stitching procedure, store results in warped_l.
    for i in tqdm(range(warped_r.shape[0])):
        for j in range(warped_r.shape[1]):
            pixel_l = warped_l[i, j, :]
            pixel_r = warped_r[i, j, :]

            if np.sum(pixel_l)>=np.sum(pixel_r):
                warped_l[i, j, :] = pixel_l
            else:
                warped_l[i, j, :] = pixel_r

    stitch_image = warped_l[:warped_r.shape[0], :warped_r.shape[1], :]

    return stitch_image
```

C. Discuss different blending method result.

1. Considering the Brightness

I noticed significant differences in the ISO values in the challenge task, which could lead to abnormal stitching results. Therefore, before stitching each image, I calculate the brightness of each one and find the average brightness. Then, I calibrate each image to the same brightness to address this issue.

Considering Brightness	Without Considering Brightness
	

```python
def calculate_average_brightness(img_list):
    total_brightness = 0

    for img_path in img_list:
        img = cv.imread(img_path)
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
        current_brightness = cv.mean(gray)[0]
        total_brightness += current_brightness

    avg_brightness = total_brightness / len(img_list)

    return avg_brightness

def auto_adjust_brightness(image, target_brightness):
    gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    current_brightness = cv.mean(gray)[0]
    ratio = target_brightness / current_brightness
    adjusted_image = cv.convertScaleAbs(image, alpha=ratio, beta=0)

    return adjusted_image
```

2. Distorted issue in the base case

Since I stitched the first two photos together before overlaying them with the others, it causes the leftmost image (the first one) to be continuously affected by the homography matrix in subsequent processing, resulting in exaggerated deformations. To address this issue, I crop out the excessively deformed areas after each stitching process.