

Computer Vision-HW2 Image stitching

109350008 張詠哲

1. Explain your implementation

1.1. image preprocessing

For the preprocessing, I only implement the cylindrical projection (according the 'other tips' in the PPT). I found that if I don't use the cylindrical projection, the image will get smaller than the previous image when the stitched image increase.

And I also calculate the intrinsic matrix, but I don't know the horizontal field of view in the camera. Therefore, I just tried a suitable value by myself.

```
def compute_camera_intrinsics_matrix(image_height, image_width, horizontal_fov):  
    ...  
    calculate intrinsic matrix  
    ...  
    vertical_fov = (image_height / image_width * horizontal_fov) * np.pi / 180  
    horizontal_fov *= np.pi / 180  
  
    f_x = (image_width / 2.0) / np.tan(horizontal_fov / 2.0)  
    f_y = (image_height / 2.0) / np.tan(vertical_fov / 2.0)  
  
    K = np.array([[f_x, 0.0, image_width / 2.0], [0.0, f_y, image_height / 2.0], [0.0, 0.0, 1.0]], dtype=np.float32)  
    return K
```

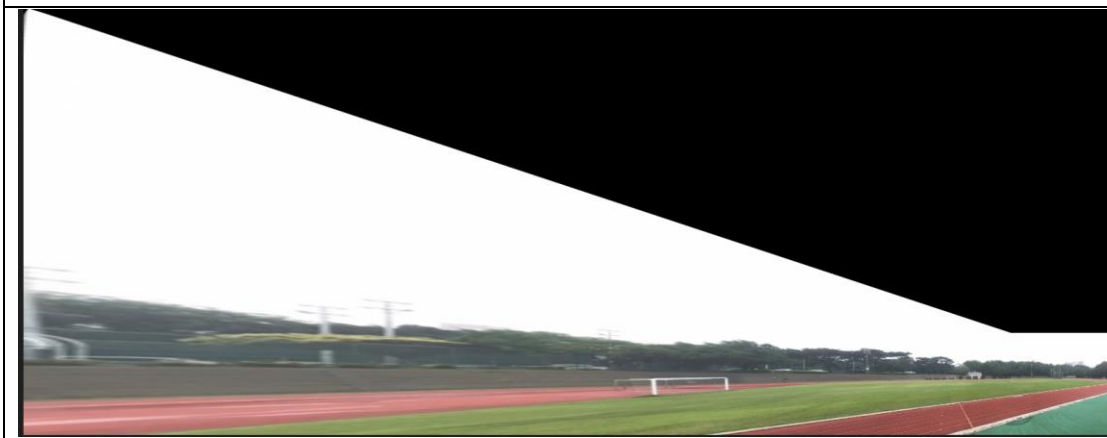
```
def cylindrical_projection(img):  
    height, width, _ = img.shape  
    K = compute_camera_intrinsics_matrix(height, width, 55) # intrinsic matrix  
    foc_len = (K[0][0] + K[1][1]) / 2  
  
    cylinder = np.zeros_like(img)  
    temp = np.mgrid[0:width, 0:height]  
    x, y = temp[0], temp[1]  
  
    # Compute theta and height  
    theta = (x - K[0][2]) / foc_len # angle theta  
    h = (y - K[1][2]) / foc_len # height  
  
    # Create points in cylindrical coordinate  
    p = np.array([np.sin(theta), h, np.cos(theta)])  
    p = p.T  
    p = p.reshape(-1, 3)  
  
    # Project points  
    image_points = K.dot(p.T).T  
  
    # Normalize  
    points = image_points[:, :-1] / image_points[:, [-1]]  
    points = points.reshape(height, width, -1)  
  
    cylinder = cv2.remap(img, points[:, :, 0].astype(np.float32), points[:, :, 1].astype(np.float32), cv2.INTER_LINEAR)  
    return cylinder
```

Comparison for using cylindrical projection and no using

Using cylindrical projection



No using



1.2 Feature Matching

I use the 2NN (nearest neighbors) to find the 2 most similar keypoints between 2 images. And use the Lowe's ratio test, to find the best matched keypoint. As for the threshold I set 0.75 (average the suggested threshold in the PPT)

```
def MatchKpts(self, kpts_1, descriptors_1, kpts_2, descriptor_2, threshold):
    good_matches = []
    for i, feat1 in enumerate(descriptors_1):
        min_idx = -1
        min_dist = np.inf
        sec_idx = -1
        sec_dist = np.inf

        # Lowe's Ratio test
        for j, feat2 in enumerate(descriptor_2):
            current_dist = np.linalg.norm(feat1 - feat2)

            if current_dist < min_dist:
                sec_idx = min_idx
                sec_dist = min_dist
                min_idx = j
                min_dist = current_dist
            elif current_dist < sec_dist and sec_idx != min_idx:
                sec_idx = j
                sec_dist = current_dist

        # find good match
        if min_dist <= sec_dist * threshold:
            good_matches.append([
                (int(kpts_1[i].pt[0]), int(kpts_1[i].pt[1])),
                (int(kpts_2[min_idx].pt[0]), int(kpts_2[min_idx].pt[1]))
            ])

    return good_matches
```

1.3 Find the Homography matrix with RANSAC

1.3.1 Homography

According to the method in PPT. First calculate A matrix, then use SVD and find the smallest number. Finally normalize homography matrix.

$$\mathbf{h} = (H_{11}, H_{12}, H_{13}, H_{21}, H_{22}, H_{23}, H_{31}, H_{32}, H_{33})^T$$
$$\mathbf{a}_x = (-x_1, -y_1, -1, 0, 0, 0, x'_2x_1, x'_2y_1, x'_2)^T$$
$$\mathbf{a}_y = (0, 0, 0, -x_1, -y_1, -1, y'_2x_1, y'_2y_1, y'_2)^T.$$

```
def get_H(kpts_1, kpts_2):
    A = []
    for i in range(len(kpts_1)):
        A.append([kpts_1[i, 0], kpts_1[i, 1], 1, 0, 0, 0, -kpts_1[i, 0] * kpts_2[i, 0], -kpts_1[i, 1] * kpts_2[i, 0], -kpts_2[i, 0]])
        A.append([0, 0, 0, kpts_1[i, 0], kpts_1[i, 1], 1, -kpts_1[i, 0] * kpts_2[i, 1], -kpts_1[i, 1] * kpts_2[i, 1], -kpts_2[i, 1]])
    u, s, v = np.linalg.svd(A)
    H = np.reshape(v[8], (3, 3))
    H = H/H.item(8)
    return H
```

1.3.2 RANSAC

To avoid outlier, we need to use RANSAC to find the best homography matrix. As for the hyperparameter, I set the iterator times 1000, and the distance threshold 5

```
max_inliner = 0
best_H = None
for _ in range(iter_num):
    ran_idx = random.sample(range(len(good_matches)), 4)
    H = get_H(img1_kpts[ran_idx], img2_kpts[ran_idx])

    current_inliner = 0
    for i in range(len(good_matches)):
        if i not in ran_idx:
            p1 = np.hstack((img1_kpts[i], [1]))
            p2 = img2_kpts[i]

            p2_hat = H @ p1.T
            if p2_hat[2] == 0: # avoid divide 0
                continue
            p2_hat = p2_hat / p2_hat[2]

            if (np.linalg.norm(p2_hat[:2] - p2) < threshold):
                current_inliner = current_inliner + 1

    if (current_inliner > max_inliner):
        max_inliner = current_inliner
        best_H = H
```

1.4 Warp and Blend image

1.4.1 Warp

First calculate the Affine translation matrix (according 'more detail' in the PPT). Then warp the image by calculated affine translation matrix A and given homography matrix H.

```

corners' = corners * H
x1' = min(min(corners'_x),0)
y1' = min(min(corners'_y),0)

```

Size we need = $(w2 + \text{abs}(x1'), h2 + \text{abs}(y1'))$

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

```

h1, w1, _ = img1.shape
h2, w2, _ = img2.shape

l_down = np.hstack(([0], [0], [1]))
l_up = np.hstack(([0], [h1-1], [1]))
r_down = np.hstack(([w1-1], [0], [1]))
r_up = np.hstack(([w1-1], [h1-1], [1]))
warped_ld = H @ l_down.T
warped_lu = H @ l_up.T
warped_rd = H @ r_down.T
warped_ru = H @ r_up.T
x = int(min(min(min(warped_ld[0],warped_lu[0]),
                    min(warped_rd[0], warped_ru[0])),
            0))

y = int(min(min(min(warped_ld[1],warped_lu[1]),
                    min(warped_rd[1], warped_ru[1])),
            0))
size = (w2 + abs(x), h2 + abs(y))

A = np.float32([[1, 0, -x], [0, 1, -y], [0, 0, 1]])
warped1 = cv2.warpPerspective(src=img1, M=A@H, dsize=size)
warped2 = cv2.warpPerspective(src=img2, M=A, dsize=size)

```

1.4.2 Blend

After we get the 2 warped images. We need to deal with color difference between the 2 images by blending. I implement 2 blending methods (linear blending and linear blending with constant width), but the linear blending with constant width gets the better result (I will discuss in the part 3). Therefore, I only show it's result in this section.

linear blending with constant width is like the compromise method between the direct warp image and linear blending. First, identify the centerline of the overlap part between 2 images. Then, take a fixed width on either side of this centerline and perform linear blending within this range.

```

def lBwC(self, imgs, cons_w = 5):
    img_left, img_right = imgs
    h1, w1, _ = img_left.shape
    hr, wr, _ = img_right.shape
    left_mask = np.zeros((hr, wr), dtype="int")
    right_mask = np.zeros((hr, wr), dtype="int")

    for i in range(h1):
        for j in range(w1):
            if np.count_nonzero(img_left[i, j]) > 0:
                left_mask[i, j] = 1
    for i in range(hr):
        for j in range(wr):
            if np.count_nonzero(img_right[i, j]) > 0:
                right_mask[i, j] = 1

    # find overlap
    overlap_mask = np.zeros((hr, wr), dtype="int")
    for i in range(hr):
        for j in range(wr):
            if (np.count_nonzero(left_mask[i, j]) > 0 and np.count_nonzero(right_mask[i, j]) > 0):
                overlap_mask[i, j] = 1

    # compute the alpha mask
    alpha_mask = np.zeros((hr, wr))
    for i in range(hr):
        min_idx = -1
        max_idx = -1
        for j in range(wr):
            if (overlap_mask[i, j] == 1 and min_idx == -1):
                min_idx = j
            if (overlap_mask[i, j] == 1):
                max_idx = j
        if (min_idx == max_idx):
            continue
        d_step = 1 / (max_idx - min_idx)
        mid_idx = int((max_idx + min_idx) / 2)

        # left
        for j in range(min_idx, mid_idx + 1):
            if (j >= mid_idx - cons_w):
                alpha_mask[i, j] = 1 - (d_step * (j - min_idx))
            else:
                alpha_mask[i, j] = 1
        # right
        for j in range(mid_idx + 1, max_idx + 1):
            if (j <= mid_idx + cons_w):
                alpha_mask[i, j] = 1 - (d_step * (j - min_idx))
            else:
                alpha_mask[i, j] = 0

    blend_img = np.copy(img_right)
    blend_img[:h1, :w1] = np.copy(img_left)
    for i in range(hr):
        for j in range(wr):
            if (np.count_nonzero(overlap_mask[i, j]) > 0):
                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(left_mask[i, j]) > 0):
                blend_img[i, j] = img_left[i, j]
            else:
                blend_img[i, j] = img_right[i, j]

    return blend_img

```

1.5 Challenge

For challenge task, we need to minimize intensity difference of overlapping pixels by task-gain compensation. First calculate the number of pixels in the overlapping area (N_{ij}), and the average of image 1 and image 2 in the overlapping area (\bar{I}_{ij} , \bar{I}_{ji}). Finally calculate the gain between warped image of previous result and the warped image that need to be stitched. I set the $\sigma_N = 100$, $\sigma_g = 0.9$. Result is quite different between using gain compensation and no use.

$$e = \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n N_{ij} ((g_i \bar{I}_{ij} - g_j \bar{I}_{ji})^2 / \sigma_N^2 + (1 - g_i)^2 / \sigma_g^2)$$

```

def get_gain(img1, img2, sig_n = 100, sig_g = 0.9):
    """
    Compute the task-gain compensation
    """
    coef = np.zeros((2, 2, 3))
    results = np.zeros((2, 3))
    I_ij, I_ji, N_ij = cal_pair(img1, img2, detect_threshold=6)
    if N_ij == 0:
        return
    coef[0][0] += N_ij * ((2 * I_ij ** 2 / sig_n ** 2) + (1 / sig_g ** 2))
    coef[0][1] -= (2 / sig_n ** 2) * N_ij * I_ij * I_ji
    coef[1][0] -= (2 / sig_n ** 2) * N_ij * I_ji * I_ij
    coef[1][1] += N_ij * ((2 * I_ji ** 2 / sig_n ** 2) + (1 / sig_g ** 2))

    results[0] += N_ij / sig_g ** 2
    results[1] += N_ij / sig_g ** 2

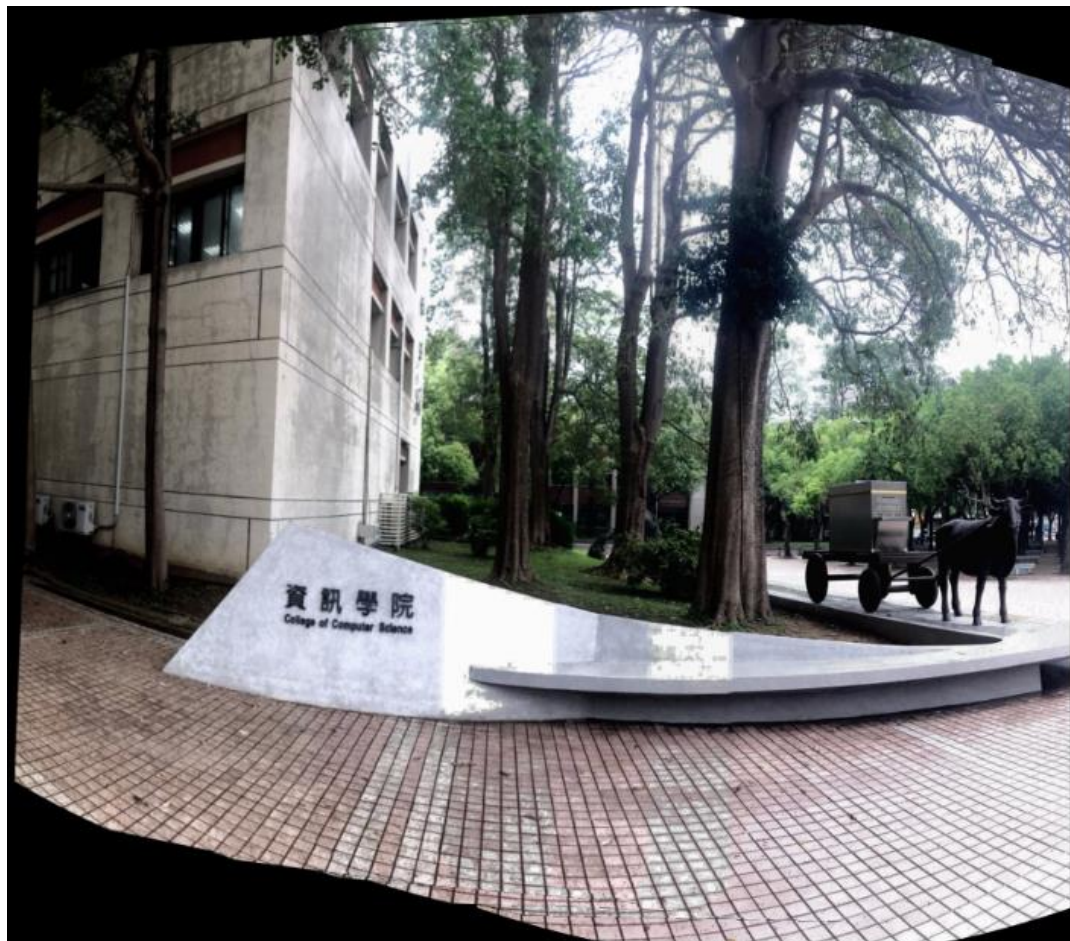
    gains = np.zeros_like(results)
    coef = coef / 1e8
    results = results / 1e8
    for i in range(coef.shape[2]):
        coefs = coef[:, :, i]
        res = results[:, i]
        gains[:, i] = np.linalg.pinv(coefs) @ res

    return gains

```

Comparison result.

Using gain compensation



No using



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

2.1 Base

2.1.1 Result

1 + 2.

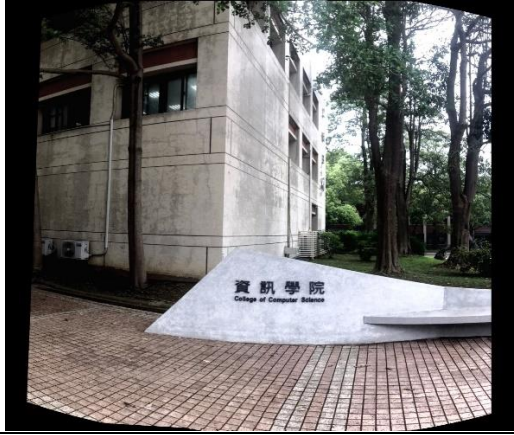
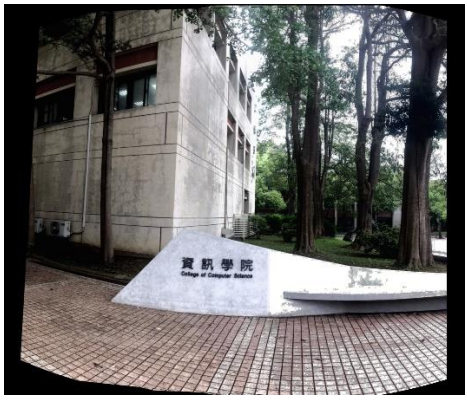

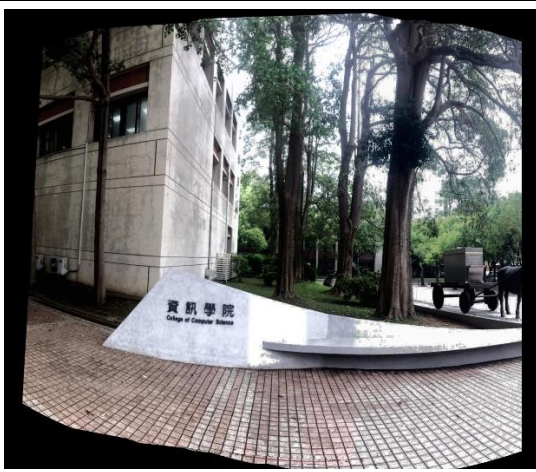


2.1.2 Final result

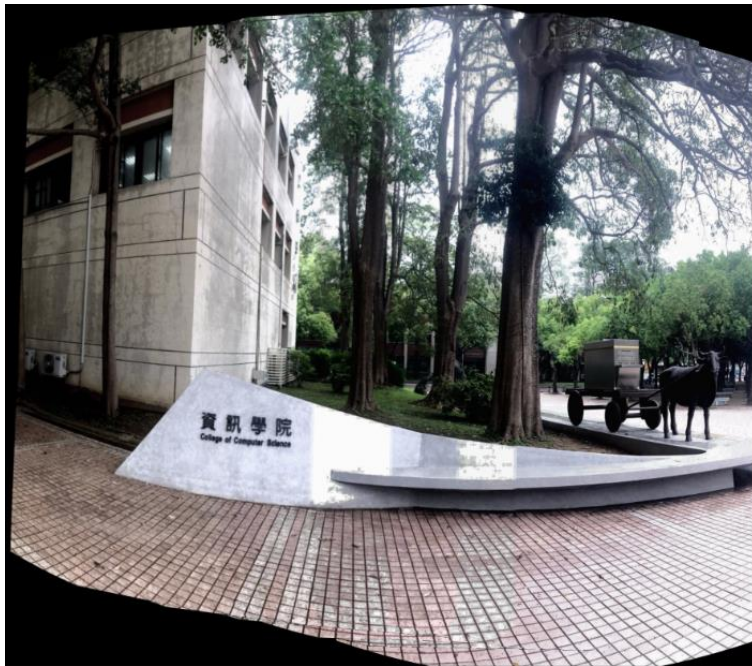


2.2 Challenge

2.2.1 Result

1 + 2	1 + 2 + 3
	
1 + 2 + 3 + 4	1 + 2 + 3 + 4 + 5
	

2.2.2 Final result



3. Discuss different blending method result.

In this assignment, I implement 2 blending methods (linear blending and linear blending with constant width). And I will take the Base task as demonstration.

Linear Blending



Linear Blending with Constant Width



As the comparison table shows, if use linear blending will cause the blurry place (ghost) in the connection place, while linear blending with constant width don't. I think the reason is the blending weights vary across the overlapping region, leading to inconsistent intensity levels and visible seams in linear blending. But linear blending with a constant width ensures that the blending weights remain consistent within the constant width, reducing ghosting artifacts by maintaining a smooth transition between images without abrupt intensity changes.