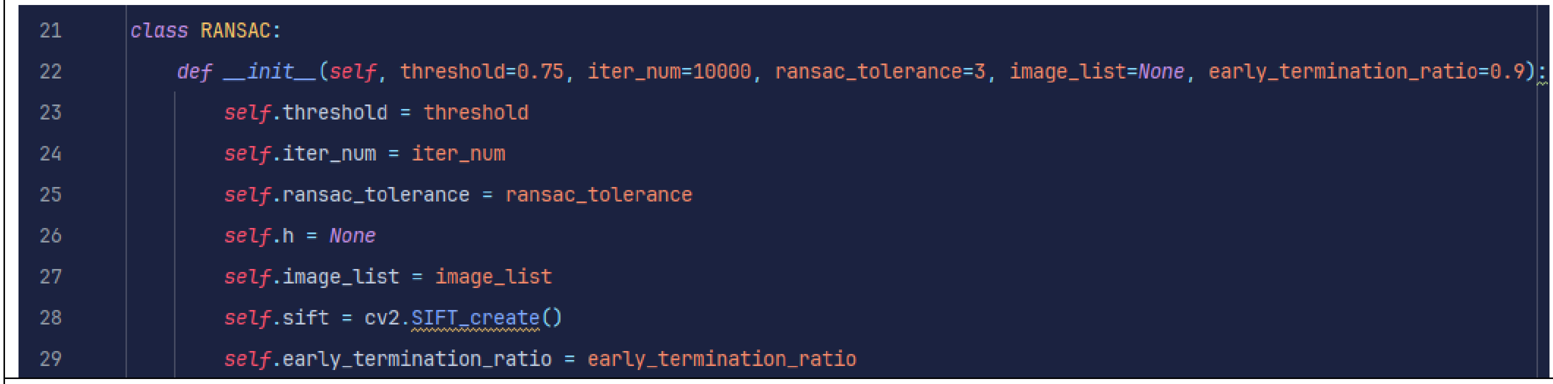
Computer Vision HW2 Report

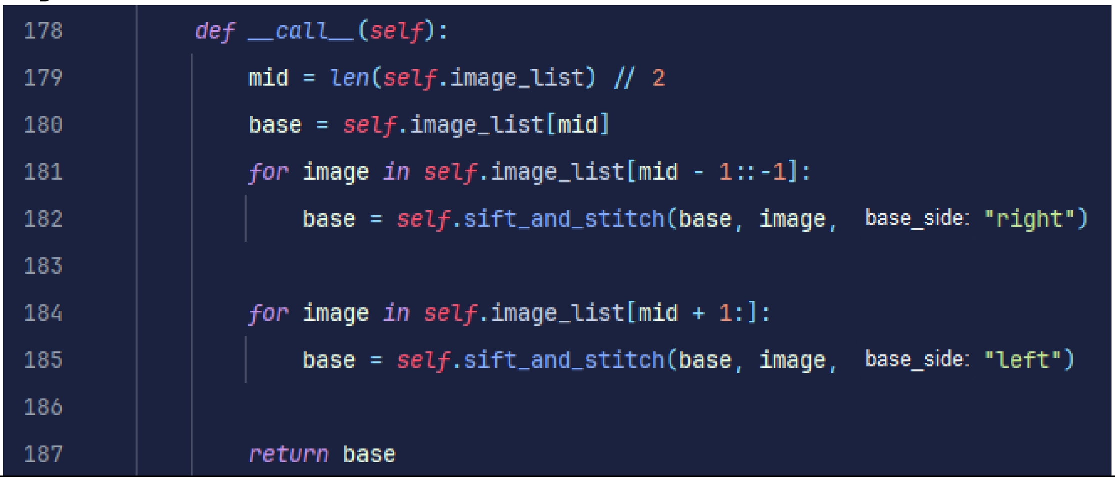
1. Implementation

I implemented a RANSAC class, these are the parameters, the usage of each would be described in later section.



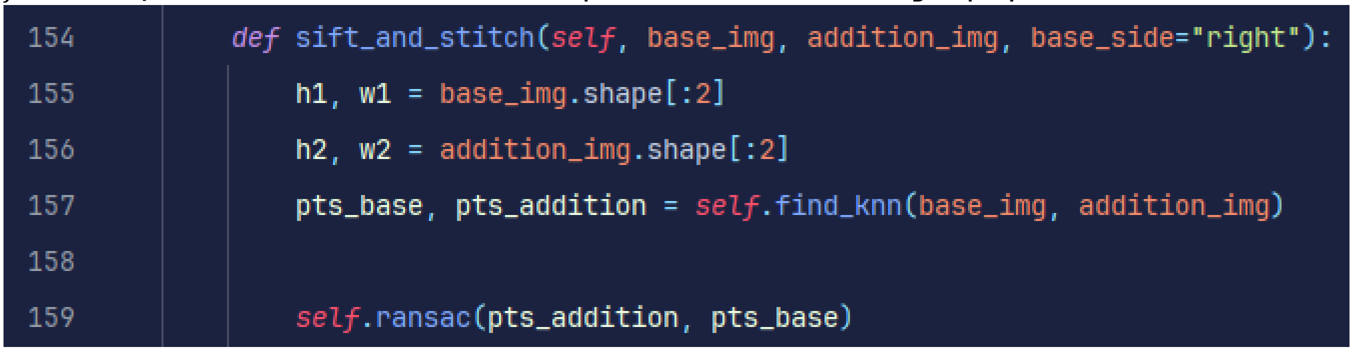
```python
class RANSAC:
    def __init__(self, threshold=0.75, iter_num=10000, ransac_tolerance=3, image_list=None, early_termination_ratio=0.9):
        self.threshold = threshold
        self.iter_num = iter_num
        self.ransac_tolerance = ransac_tolerance
        self.h = None
        self.image_list = image_list
        self.sift = cv2.SIFT_create()
        self.early_termination_ratio = early_termination_ratio
```

At the start of the process, it would set the middle image as the base (to get a better result), and stitch other images from middle to left and from middle to right.



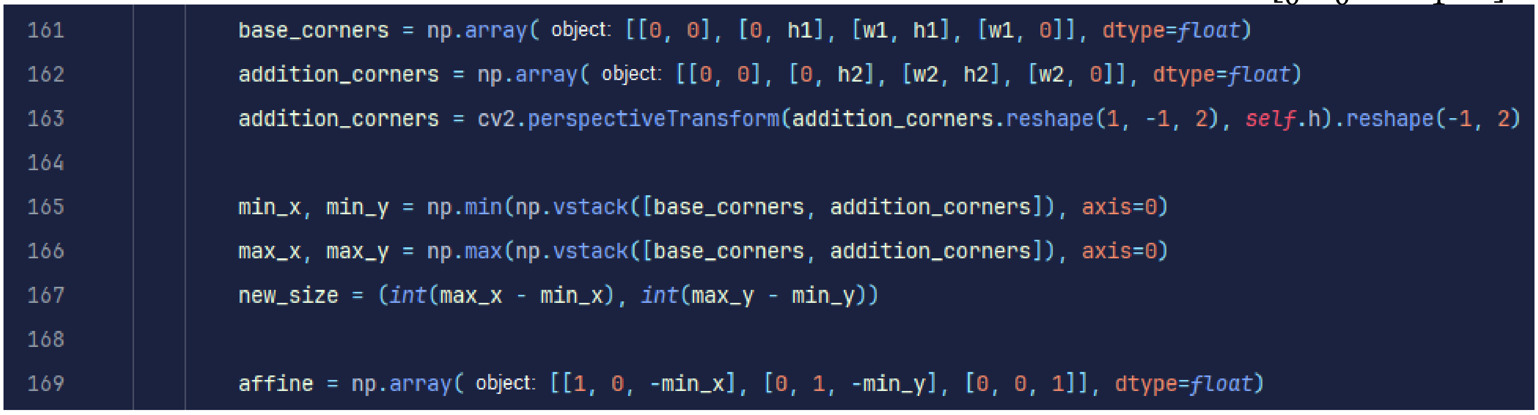
```python
    def __call__(self):
        mid = len(self.image_list) // 2
        base = self.image_list[mid]
        for image in self.image_list[mid - 1::-1]:
            base = self.sift_and_stitch(base, image, base_side: "right")

        for image in self.image_list[mid + 1:]:
            base = self.sift_and_stitch(base, image, base_side: "left")

        return base
```

In the *sift_and_stitch* function, it firstly calculates the proper neighbors by *find_knn*, then use *ransac* to sample the best homography matrix.



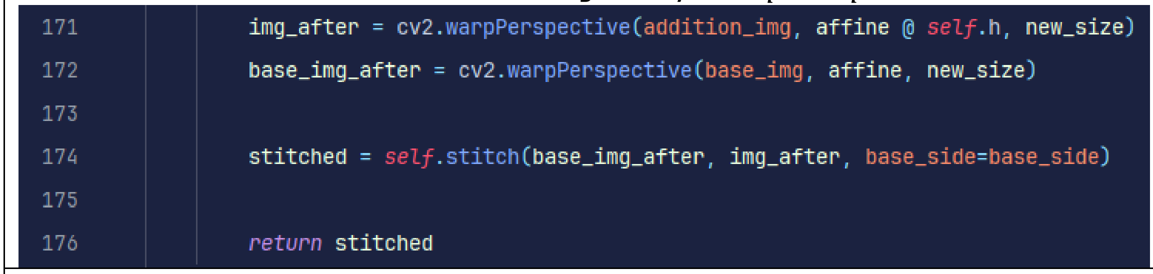
```python
    def sift_and_stitch(self, base_img, addition_img, base_side="right"):
        h1, w1 = base_img.shape[:2]
        h2, w2 = addition_img.shape[:2]
        pts_base, pts_addition = self.find_knn(base_img, addition_img)

        self.ransac(pts_addition, pts_base)
```

After calculating the homography matrix, it transforms the corners of addition image by the homography and get the transformed positions. Using the transformed corners and the corners of the base image, we can calculate the needed image size ($max\_x - min\_x, max\_y - min\_y$) and the affine translation matrix $\begin{bmatrix} 1 & 0 & -min\_x \\ 0 & 1 & -min\_y \\ 0 & 0 & 1 \end{bmatrix}$.



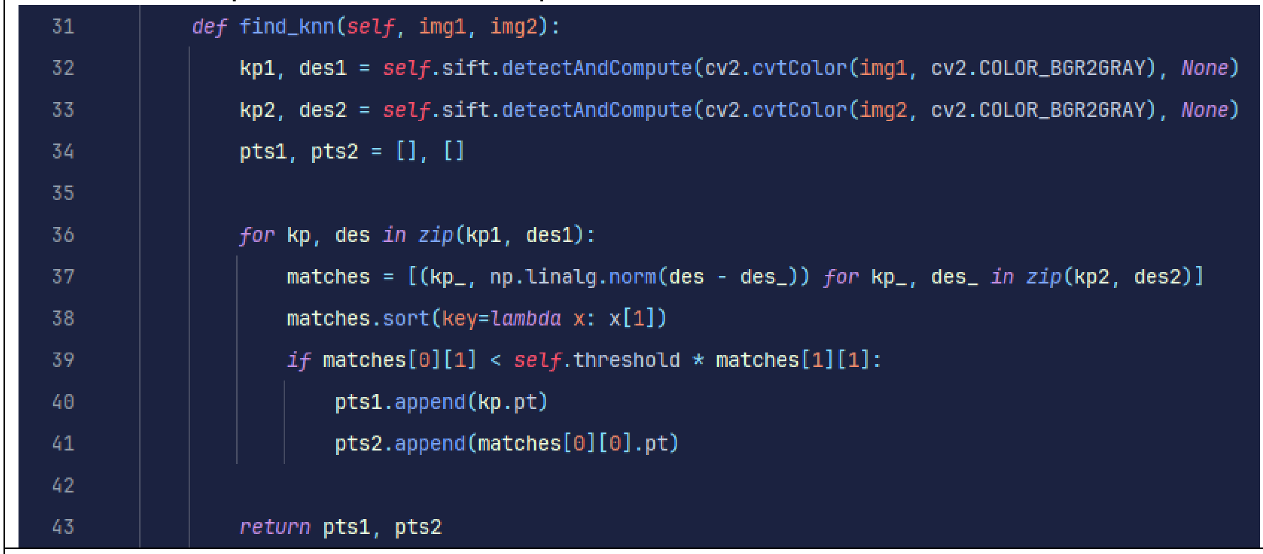
```python
        base_corners = np.array( object: [[0, 0], [0, h1], [w1, h1], [w1, 0]], dtype=float)
        addition_corners = np.array( object: [[0, 0], [0, h2], [w2, h2], [w2, 0]], dtype=float)
        addition_corners = cv2.perspectiveTransform(addition_corners.reshape(1, -1, 2), self.h).reshape(-1, 2)

        min_x, min_y = np.min(np.vstack([base_corners, addition_corners]), axis=0)
        max_x, max_y = np.max(np.vstack([base_corners, addition_corners]), axis=0)
        new_size = (int(max_x - min_x), int(max_y - min_y))

        affine = np.array( object: [[1, 0, -min_x], [0, 1, -min_y], [0, 0, 1]], dtype=float)
```

Then we can transform the images by *warpPerspective* and stitch them.



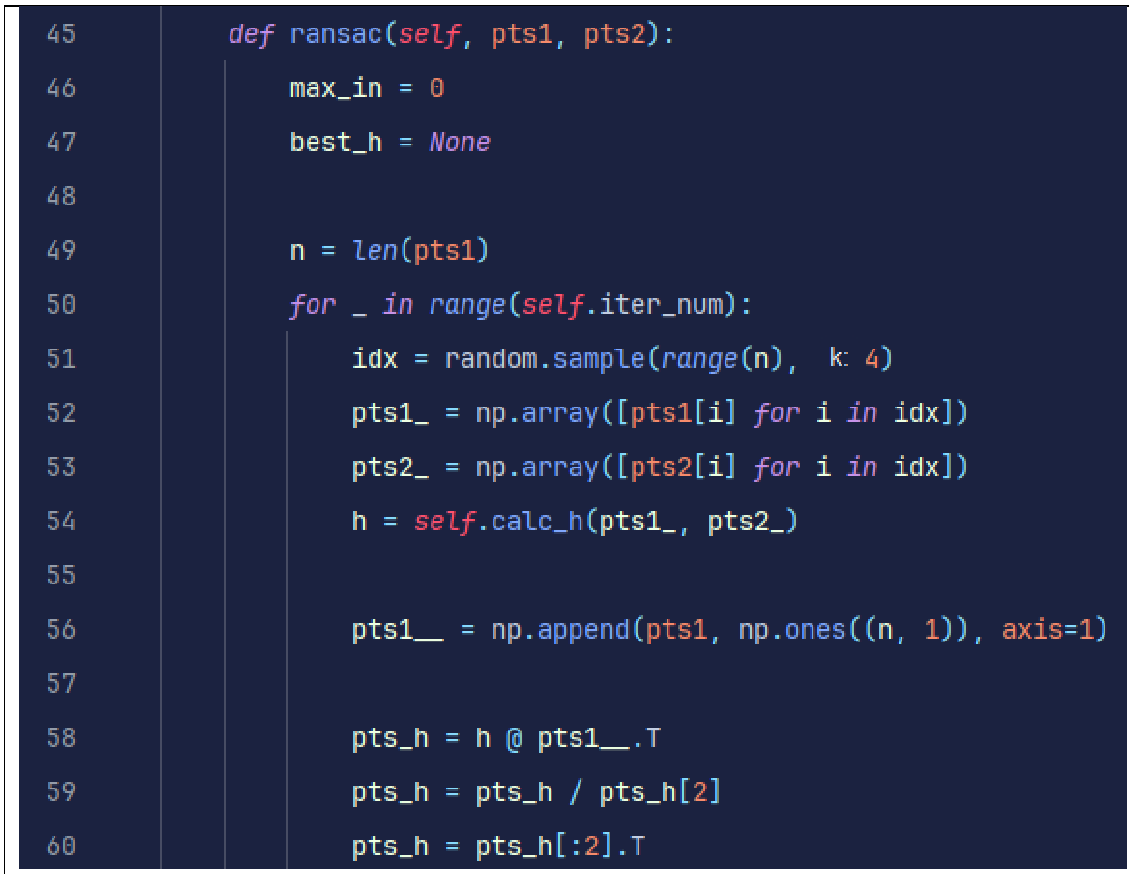
```python
        img_after = cv2.warpPerspective(addition_img, affine @ self.h, new_size)
        base_img_after = cv2.warpPerspective(base_img, affine, new_size)

        stitched = self.stitch(base_img_after, img_after, base_side=base_side)

        return stitched
```

In the *find_knn* function, it calculates the keypoints and for each description in image 1, calculate all the distance of it and the descriptors in image 2, get the keypoints with the two lowest distances. If the distance of the first closest is smaller than *threshold* * the distance of the second closest, then record the positions to be possible matches.



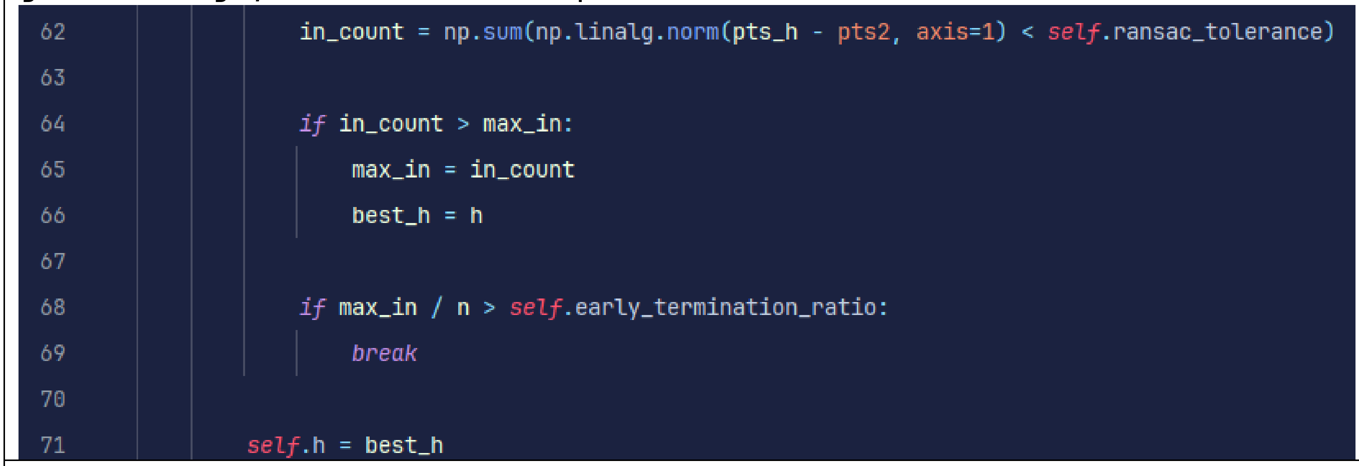
```python
    def find_knn(self, img1, img2):
        kp1, des1 = self.sift.detectAndCompute(cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY), None)
        kp2, des2 = self.sift.detectAndCompute(cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY), None)
        pts1, pts2 = [], []

        for kp, des in zip(kp1, des1):
            matches = [(kp_, np.linalg.norm(des - des_)) for kp_, des_ in zip(kp2, des2)]
            matches.sort(key=lambda x: x[1])
            if matches[0][1] < self.threshold * matches[1][1]:
                pts1.append(kp.pt)
                pts2.append(matches[0][0].pt)

        return pts1, pts2
```

In the *ransac* function, it runs for *iter_num* iterations, and sample four random indices as the position pairs to fit homography, and calculate the homography by the *calc_h* function. Then transform all the points on addition image to its position on base image.



```python
    def ransac(self, pts1, pts2):
        max_in = 0
        best_h = None

        n = len(pts1)
        for _ in range(self.iter_num):
            idx = random.sample(range(n),  k: 4)
            pts1_ = np.array([pts1[i] for i in idx])
            pts2_ = np.array([pts2[i] for i in idx])
            h = self.calc_h(pts1_, pts2_)

            pts1__ = np.append(pts1, np.ones((n, 1)), axis=1)

            pts_h = h @ pts1__.T
            pts_h = pts_h / pts_h[2]
            pts_h = pts_h[:2].T
```

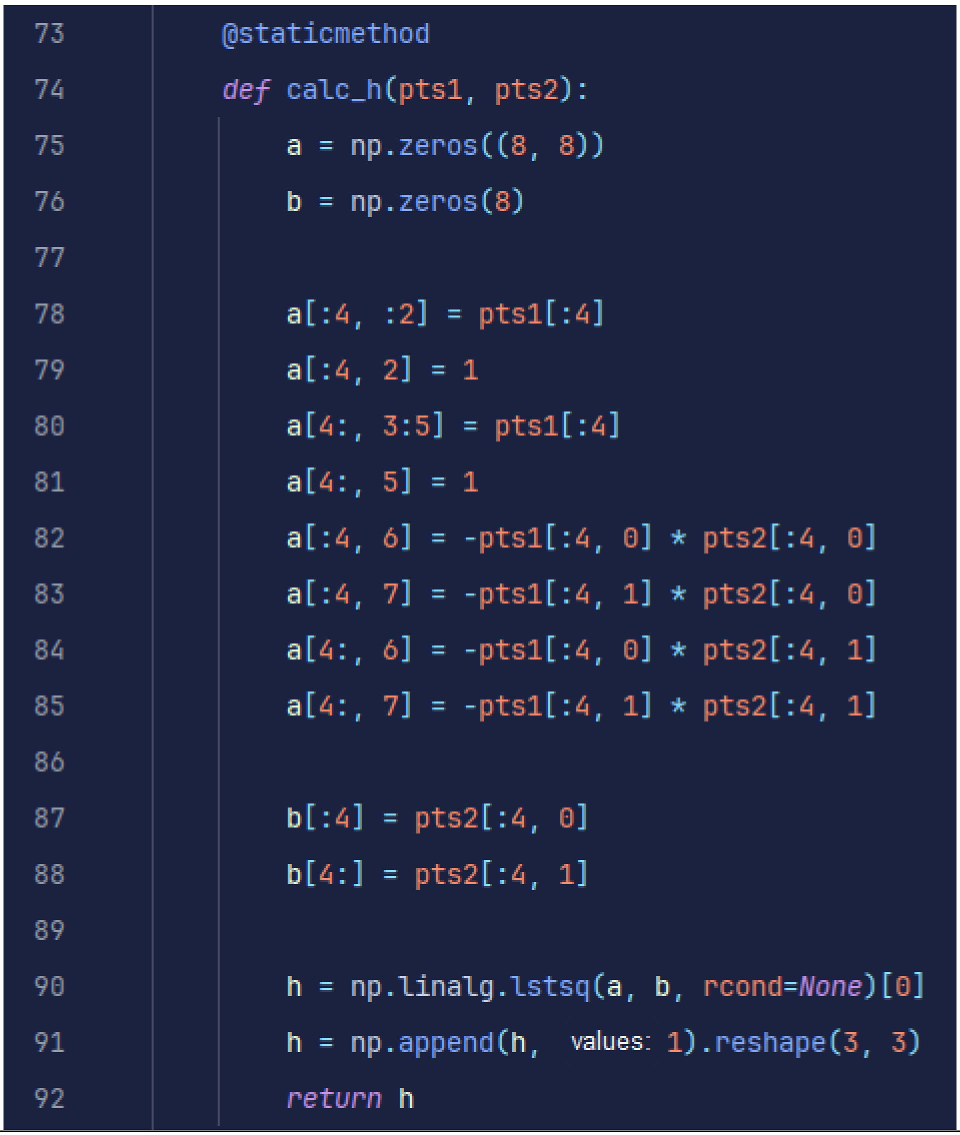
After transforming the points, compare the transformed position and its corresponding point on base image, calculate the numbers of the close enough point pairs. If this sample is better, then record it. If the homography is great enough, then break the process.



```python
            in_count = np.sum(np.linalg.norm(pts_h - pts2, axis=1) < self.ransac_tolerance)

            if in_count > max_in:
                max_in = in_count
                best_h = h

            if max_in / n > self.early_termination_ratio:
                break

        self.h = best_h
```

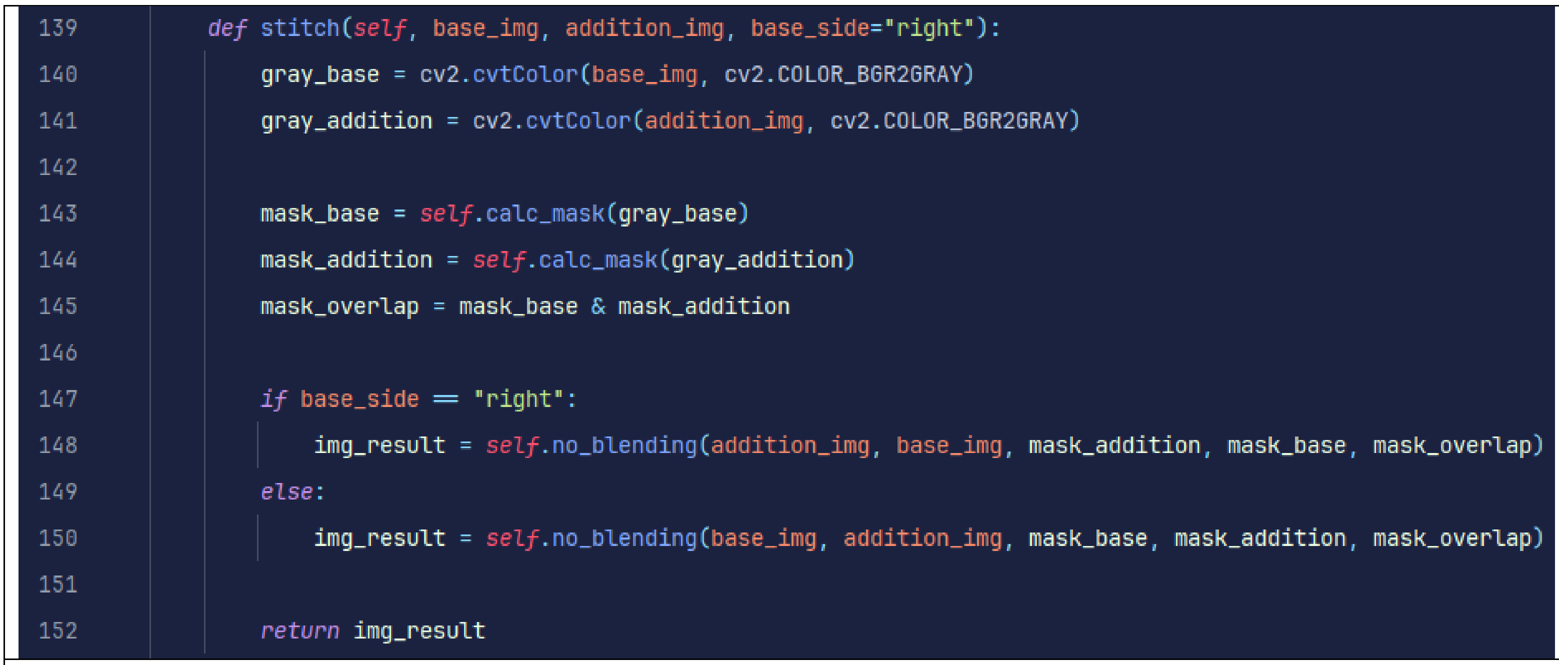
To calculate homography, I fill the A matrix and b vector, then solve the homography parameters by least square.

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



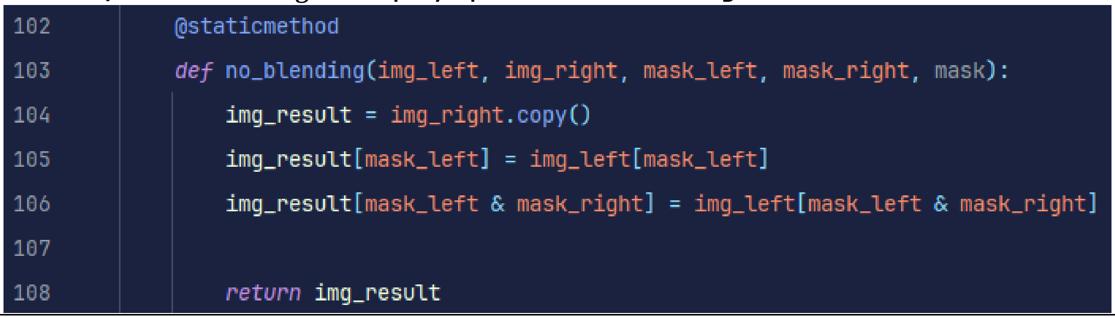
```python
    @staticmethod
    def calc_h(pts1, pts2):
        a = np.zeros((8, 8))
        b = np.zeros(8)

        a[:4, :2] = pts1[:4]
        a[:4, 2] = 1
        a[4:, 3:5] = pts1[:4]
        a[4:, 5] = 1
        a[:4, 6] = -pts1[:4, 0] * pts2[:4, 0]
        a[:4, 7] = -pts1[:4, 1] * pts2[:4, 0]
        a[4:, 6] = -pts1[:4, 0] * pts2[:4, 1]
        a[4:, 7] = -pts1[:4, 1] * pts2[:4, 1]

        b[:4] = pts2[:4, 0]
        b[4:] = pts2[:4, 1]

        h = np.linalg.lstsq(a, b, rcond=None)[0]
        h = np.append(h,  values: 1).reshape(3, 3)
        return h
```

In the *stitch* function, it firstly converts the images to gray and calculate masks from the gray images. For the overlapped origin, calculate another overlap mask to be blended in following process.



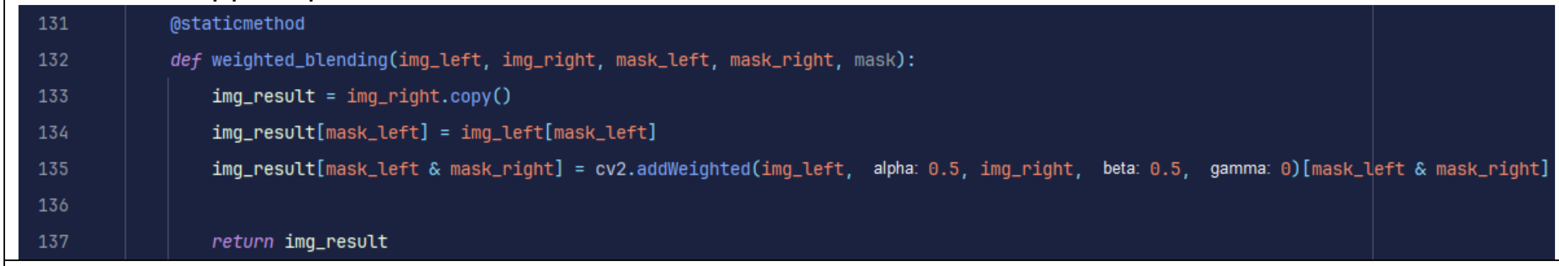
```python
    def stitch(self, base_img, addition_img, base_side="right"):
        gray_base = cv2.cvtColor(base_img, cv2.COLOR_BGR2GRAY)
        gray_addition = cv2.cvtColor(addition_img, cv2.COLOR_BGR2GRAY)

        mask_base = self.calc_mask(gray_base)
        mask_addition = self.calc_mask(gray_addition)
        mask_overlap = mask_base & mask_addition

        if base_side == "right":
            img_result = self.no_blending(addition_img, base_img, mask_addition, mask_base, mask_overlap)
        else:
            img_result = self.no_blending(base_img, addition_img, mask_base, mask_addition, mask_overlap)

        return img_result
```

For blending, I've implemented three functions to blend the result.
First, *no_blending* simply paste the image to the left on the image to the right.



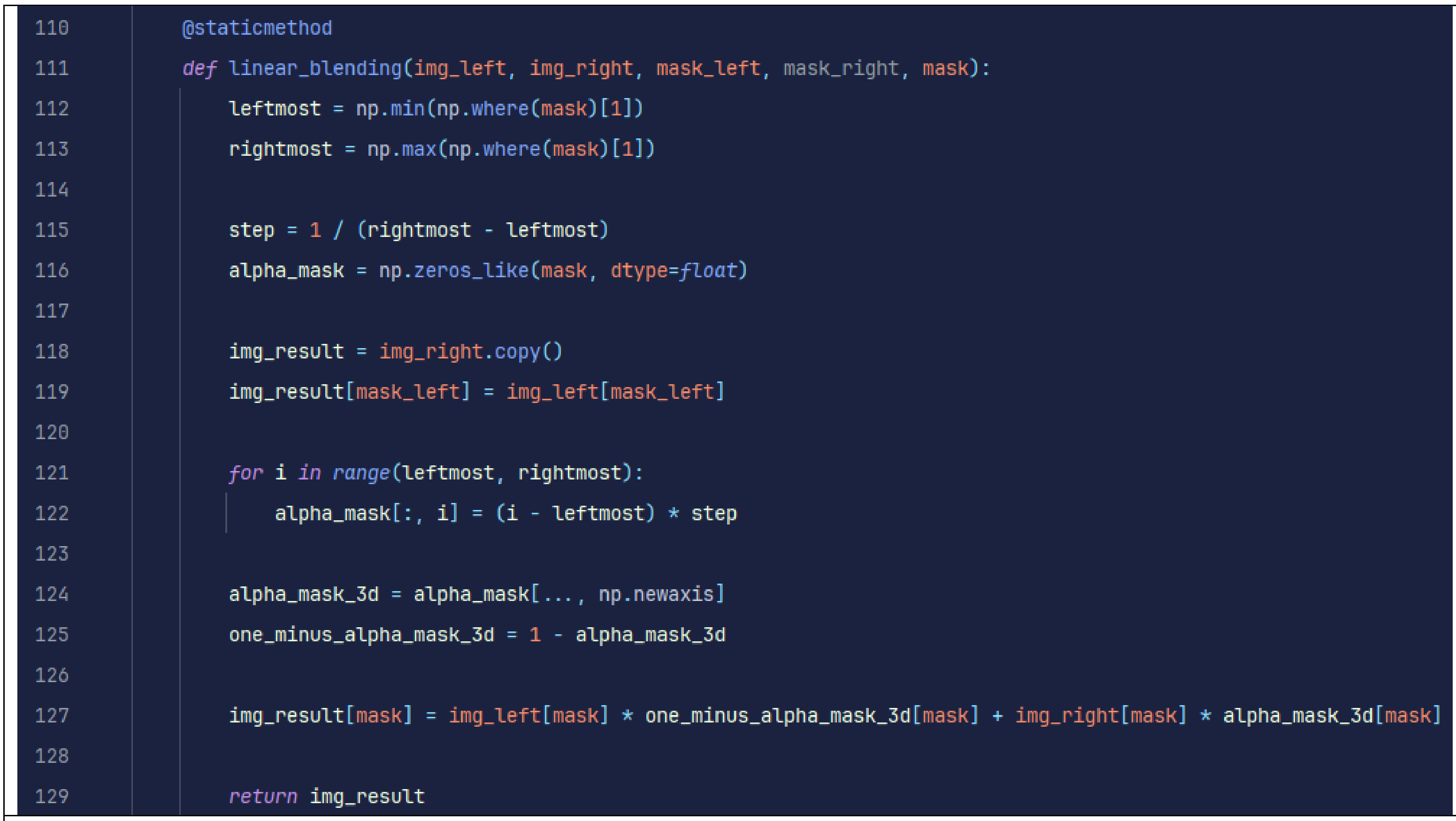
```python
    @staticmethod
    def no_blending(img_left, img_right, mask_left, mask_right, mask):
        img_result = img_right.copy()
        img_result[mask_left] = img_left[mask_left]
        img_result[mask_left & mask_right] = img_left[mask_left & mask_right]

        return img_result
```

Second, *weighted_blending* use *cv2.addWeighted* to average the two images and replace the overlapped part with the blended result.



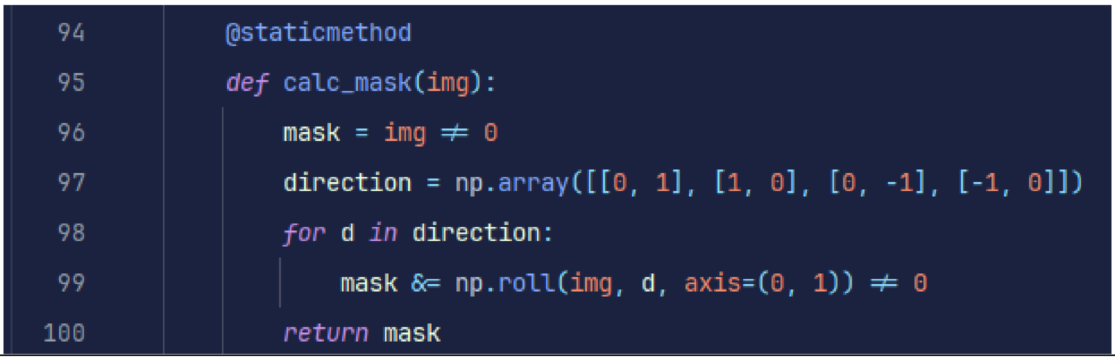
```python
    @staticmethod
    def weighted_blending(img_left, img_right, mask_left, mask_right, mask):
        img_result = img_right.copy()
        img_result[mask_left] = img_left[mask_left]
        img_result[mask_left & mask_right] = cv2.addWeighted(img_left,  alpha: 0.5, img_right,  beta: 0.5,  gamma: 0)[mask_left & mask_right]

        return img_result
```

Third, *linear_blending* take the leftmost and rightmost pixel of overlapped region, and from the left to the right, the result takes weighted from the left image and (1-weighted) from the right image. For the weights, it is gradually decreased from the leftmost overlapped pixel to the rightmost overlapped pixel.



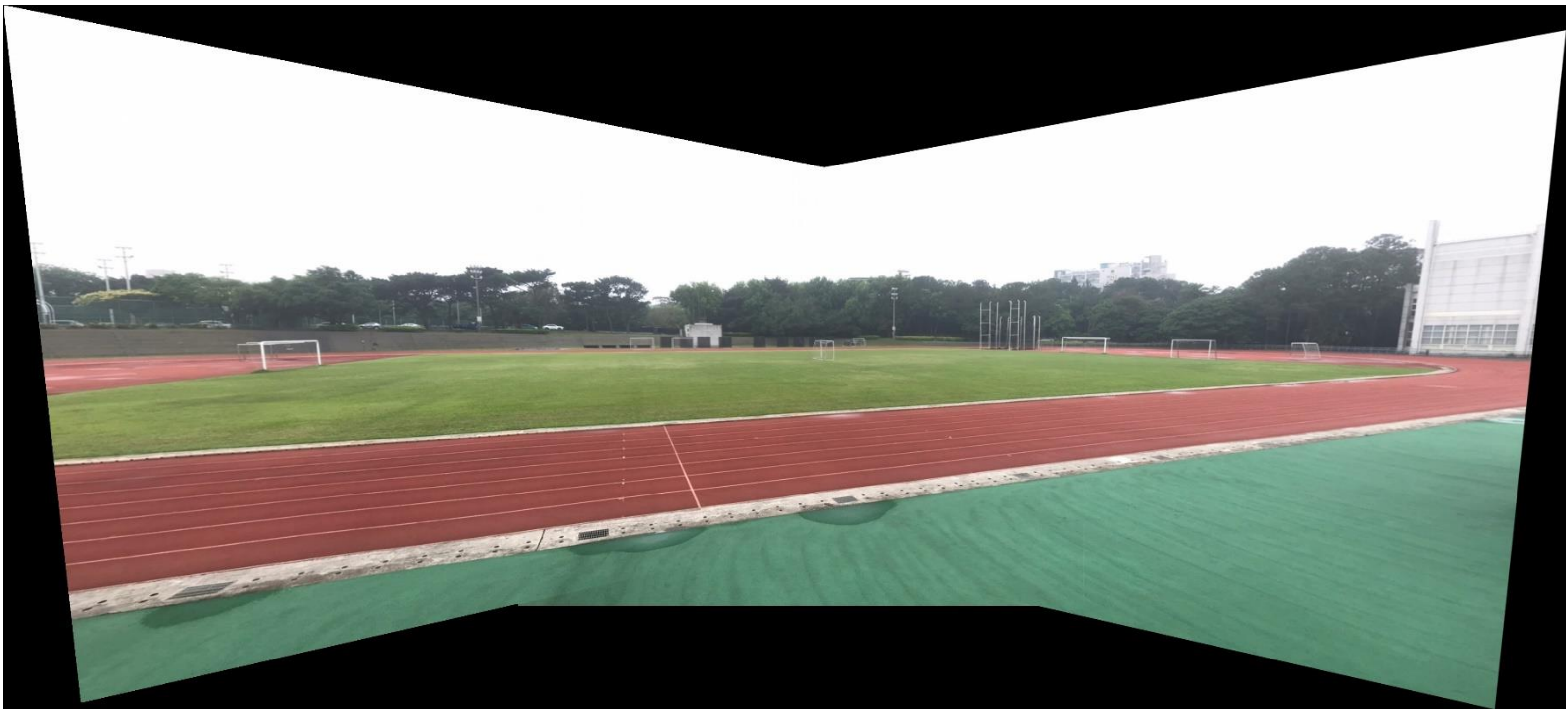
```python
    @staticmethod
    def linear_blending(img_left, img_right, mask_left, mask_right, mask):
        leftmost = np.min(np.where(mask)[1])
        rightmost = np.max(np.where(mask)[1])

        step = 1 / (rightmost - leftmost)
        alpha_mask = np.zeros_like(mask, dtype=float)

        img_result = img_right.copy()
        img_result[mask_left] = img_left[mask_left]

        for i in range(leftmost, rightmost):
            alpha_mask[:, i] = (i - leftmost) * step

        alpha_mask_3d = alpha_mask[..., np.newaxis]
        one_minus_alpha_mask_3d = 1 - alpha_mask_3d

        img_result[mask] = img_left[mask] * one_minus_alpha_mask_3d[mask] + img_right[mask] * alpha_mask_3d[mask]

        return img_result
```

For *calc_mask*, I noticed that border of transformed image could be some dark pixel but can't simply capture that by $gray\_image \neq 0$, so I implement another function to extract the pixels with its neighbors all greater than zero as the mask.
(the difference is showed below)



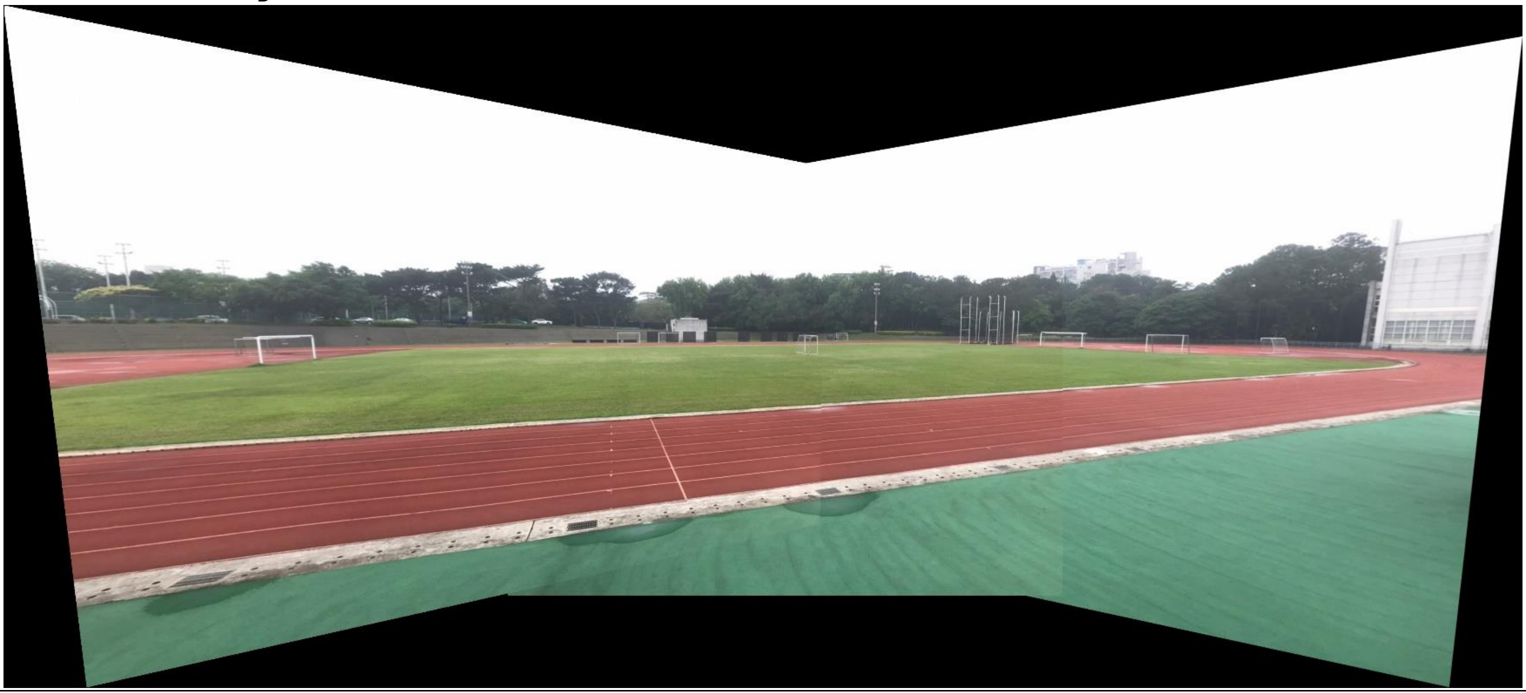
```python
    @staticmethod
    def calc_mask(img):
        mask = img != 0
        direction = np.array([[0, 1], [1, 0], [0, -1], [-1, 0]])
        for d in direction:
            mask &= np.roll(img, d, axis=(0, 1)) != 0
        return mask
```

2. Stitching result (linear blending)



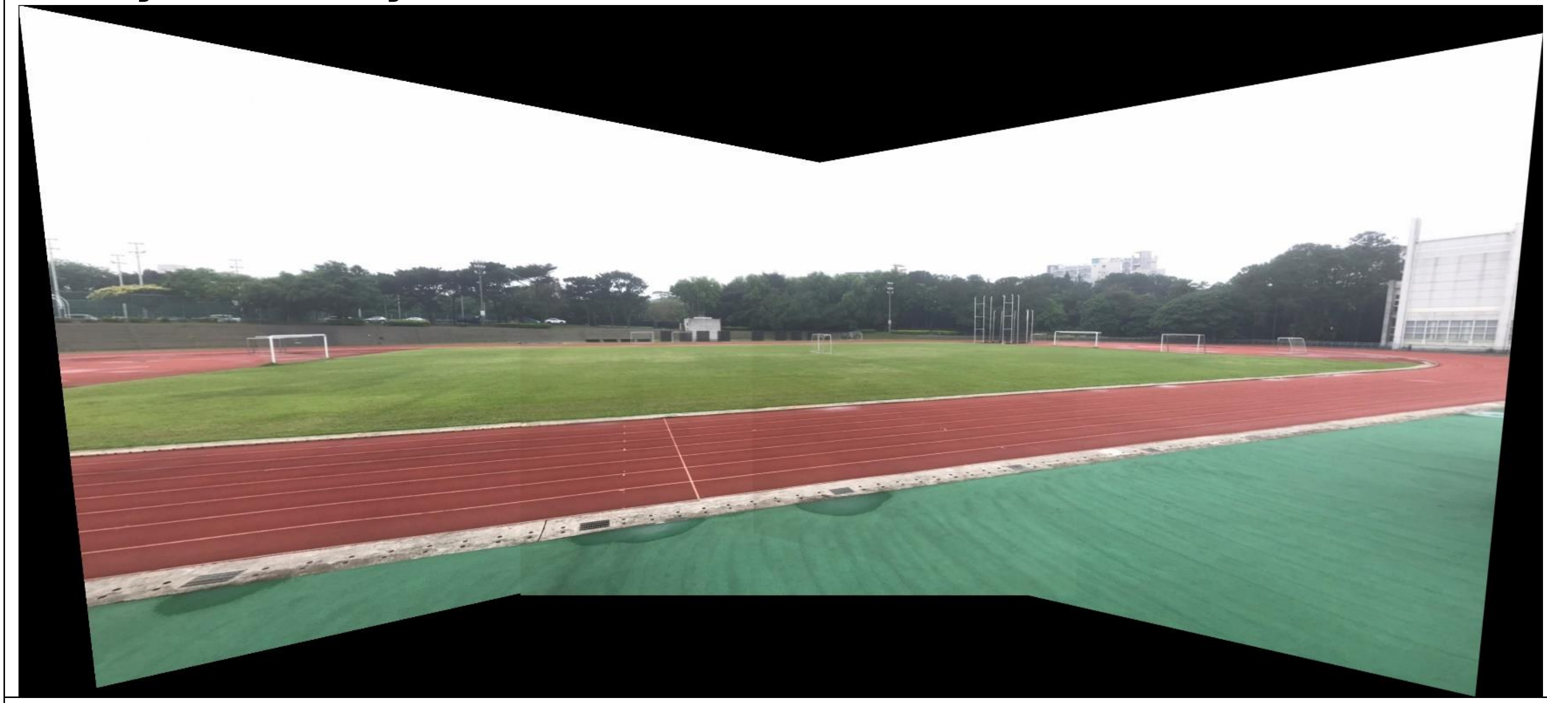
3. Comparison

1. No blending (custom defined mask):

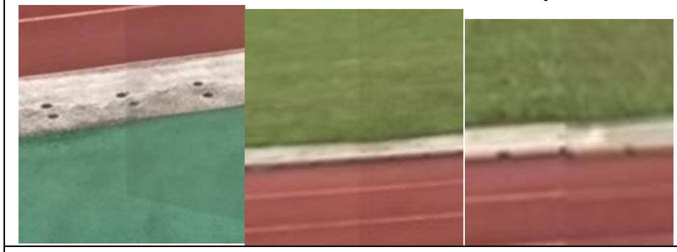


We can find some discontinuous part in the result and also the color is not blended.

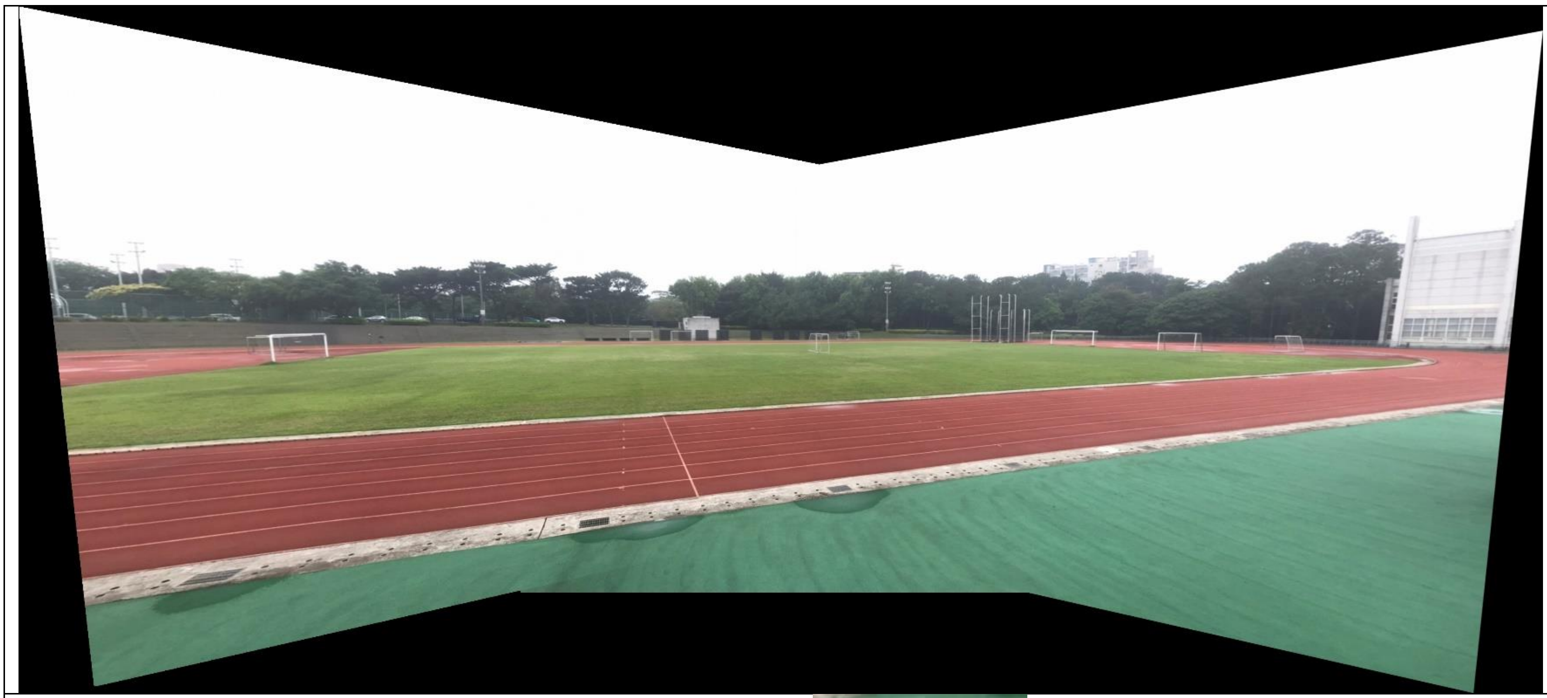
2. Weighted blending (custom defined mask):



The color is blended better, but still have some obvious color gap.

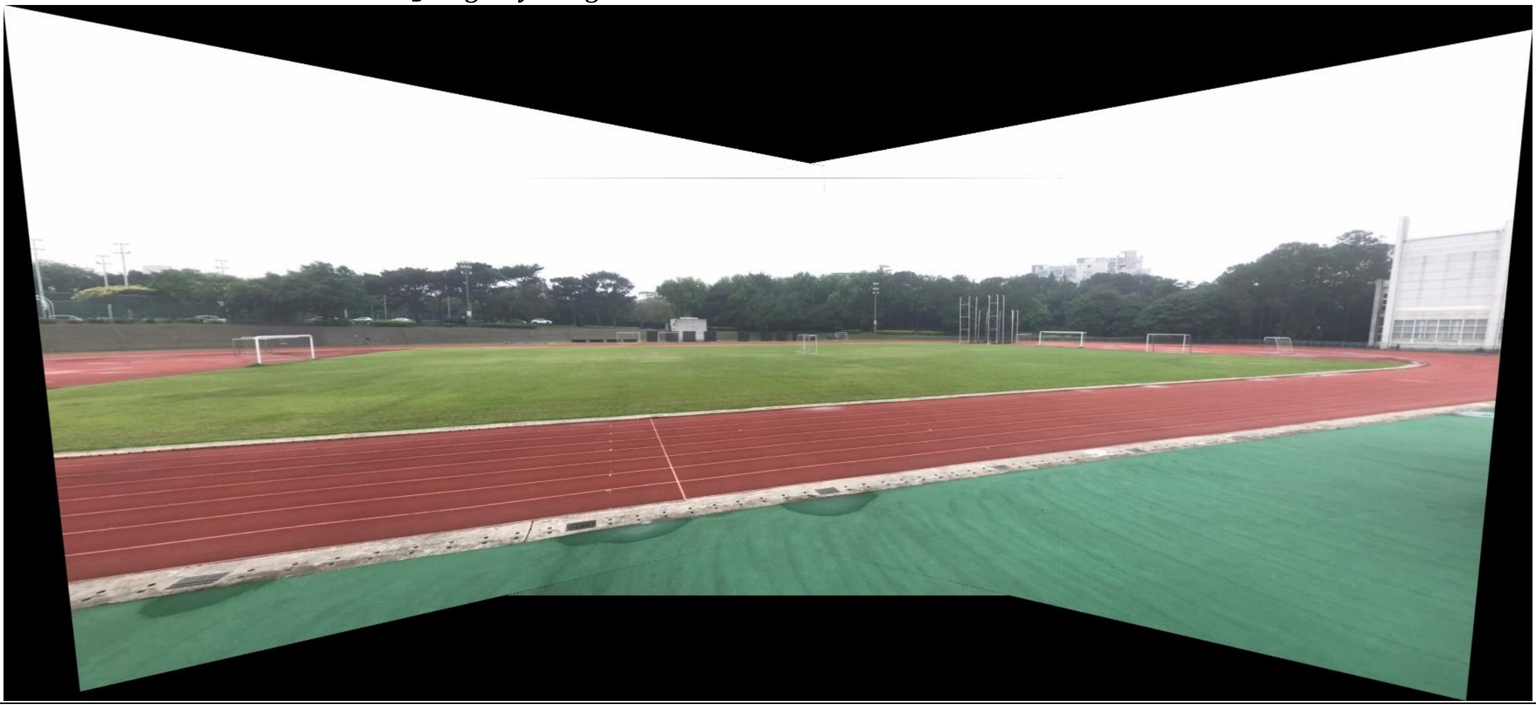


3. Linear blending (custom defined mask):



I think the only flaw is the bottom part.

Extra: Linear blending ($gray\_img \neq 0$ mask):



It is obvious that there are some black lines because of the dark border of the transformed images.