CV HW2

Implementation

Preprocessing – linear projection and cylindrical projection

For linear projection, I cut the image into 4 parts, and transform it to desired points.

```python
def preprocess2(image, shift=10):
    flag = cv2.INTER_NEAREST
    # do affine transformation on 4 images
    # define the 4 points of the first image
    x, y = image.shape[1], image.shape[0]
    pts1 = np.float32([[0, 0], [x/2, 0], [0, y/2], [x/2, y/2]])
    dest1 = np.float32([[shift, shift], [x/2, 0], [shift, y/2], [x/2, y/2]])
    result1 = cv2.getPerspectiveTransform(pts1, dest1)
    img_temp = np.array(image)
    img1 = img_temp[0:y//2, 0:x//2]
    warped_image = cv2.warpPerspective(img1, result1, (x//2, y//2), flags=flag)

    pts2 = np.float32([[0, 0], [x-x/2, 0], [0, y/2], [x-x/2, y/2]])
    dest2 = np.float32([[0, 0], [x-x/2-shift, shift], [0, y/2], [x-x/2-shift, y/2]])
    result2 = cv2.getPerspectiveTransform(pts2, dest2)
    img2 = img_temp[0:y//2, x//2:x]
    warped_image2 = cv2.warpPerspective(img2, result2, (x - x//2, y//2), flags=flag)

    pts3 = np.float32([[0, 0], [x/2, 0], [0, y-y/2], [x/2, y-y/2]])
    dest3 = np.float32([[shift, 0], [x/2, 0], [shift, y-y/2-shift], [x/2, y-y/2]])
    result3 = cv2.getPerspectiveTransform(pts3, dest3)
    img3 = img_temp[y//2:y, 0:x//2]
    warped_image3 = cv2.warpPerspective(img3, result3, (x//2, y - y//2), flags=flag)

    pts4 = np.float32([[0, 0], [x-x/2, 0], [0, y-y/2], [x-x/2, y-y/2]])
    dest4 = np.float32([[0, 0], [x-x/2-shift, 0], [0, y-y/2], [x-x/2-shift, y-y/2-shift]])
    result4 = cv2.getPerspectiveTransform(pts4, dest4)
    img4 = img_temp[y//2:y, x//2:x]
    warped_image4 = cv2.warpPerspective(img4, result4, (x-x//2, y-y//2), flags=flag)

    # combine the 4 images
    if image.ndim == 2:
        result = np.zeros((y, x), np.uint8)
    else:
        result = np.zeros((y, x, 3), np.uint8)
    result[0:y//2, 0:x//2] = warped_image
    result[0:y//2, x//2:x] = warped_image2
    result[y//2:y, 0:x//2] = warped_image3
    result[y//2:y, x//2:x] = warped_image4
    return result
```

For cylindrical projection, I set the intrinsic matrix like below, and it turned out that this setting is good enough.

```python
def preprocess(img):
    h_,w_ = img.shape[:2]
    K = np.array([[w_,0,w_/2],[0,h_,h_/2],[0,0,1]])

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

    # calculate 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)

    # project back to image-pixels plane
    B = K.dot(A.T).T
    # 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)

    return cv2.remap(img, B[:,:,0].astype(np.float32), B[:,:,1].astype(np.float32), cv2.INTER_NEAREST)
```

The result is as follows: linear projection / cylindrical projection

For the challenge images, I select this two transformation for each image according to experiment, and I add histogram equalization on the grayscaled images to help knn to fit better.

KNN

Here I compute the distance between all pairs of points in the two set and select the 2 smallest pairs. If the pair pass the lowe's ratio test, add it to the set of found points.

```python
def get_matches(keypoints1, descriptor1, keypoints2, descriptor2, lowes_ratio=0.8, debugging=False):
    good_matches = []
    points1 = []
    points2 = []

    if debugging:
        bf = cv2.BFMatcher()
        matches = bf.knnMatch(descriptor1, descriptor2, k=2)
        for m,n in matches:
            if m.distance < lowes_ratio*n.distance:
                good_matches.append(m)
        points1 = np.float32([keypoints1[m.queryIdx].pt for m in good_matches])
        points2 = np.float32([keypoints2[m.trainIdx].pt for m in good_matches])

    else:
        for i in range(len(keypoints1)):
            min_dist = np.inf
            second_min_dist = np.inf
            min_dist_index = -1

            # compute the dinstance between two descriptors and update
            for j in range(len(keypoints2)):
                dist = np.linalg.norm(descriptor1[i] - descriptor2[j]).item()
                if dist < min_dist:
                    second_min_dist = min_dist
                    min_dist = dist
                    min_dist_index = j
                elif dist < second_min_dist:
                    second_min_dist = dist
            # lowe's ratio test
            if min_dist  < lowes_ratio * second_min_dist:
                points1.append(keypoints1[i].pt)
                points2.append(keypoints2[min_dist_index].pt)
    return np.float32(points1), np.float32(points2)
```

RANSAC & Homography

After we have a set of matched points, we need to find the best homography matrix for the points using RANSAC. We randomly select 4 pairs of points and find the homography matrix of these 4 pairs. We then check how many other points also fit this homography matrix with a threshold. We then choose the best

H that has as many points fit as possible.

RANSAC:

```
def RANSAC_for_H(points1, points2, RANSAC_n_iter=1000, RANSAC_threshold=5, debugging=False):
    if debugging:
        best_H, _ = cv2.findHomography(points1, points2, cv2.RANSAC)
    else:
        maximum_inliers = 0
        best_H = None

        for i in range(RANSAC_n_iter):
            random_index = random.sample(range(len(points1)), 4)
            random_points1 = points1[random_index]
            random_points2 = points2[random_index]
            H = get_Homography(random_points1, random_points2)
            inliers = 0
            for j in range(len(points1)):
                p1 = np.array([points1[j, 0], points1[j, 1], 1])
                p2 = np.array([points2[j, 0], points2[j, 1], 1])
                p2_hat = np.dot(H, p1)
                p2_hat = p2_hat / p2_hat[2]
                if np.linalg.norm(p2 - p2_hat) < RANSAC_threshold:
                    inliers += 1
            if inliers > maximum_inliers:
                maximum_inliers = inliers
                best_H = H
    return best_H
```

Homography:

I use another function to compute the homography matrix given three points by filling in the matrix described in [homography_estimation.pdf (ucsd.edu)](), and solve with svd decomposition. We chose the smallest one (last element) and normalize the laset element to 1.

$$\mathbf{a}_x^T \mathbf{h} = \mathbf{0}$$
$$\mathbf{a}_y^T \mathbf{h} = \mathbf{0}$$

, where

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

```
def get_Homography(points1, points2):
    # construct A
    A = []
    for i in range(len(points1)):
        A.append([points1[i, 0], points1[i, 1], 1, 0, 0, 0, -points1[i, 0] * points2[i, 0], -points1[i, 1] * points2[i, 0], -points2[i, 0]])
    for i in range(len(points1)):
        A.append([0, 0, 0, points1[i, 0], points1[i, 1], 1, -points1[i, 0] * points2[i, 1], -points1[i, 1] * points2[i, 1], -points2[i, 1]])

    # solve using SVD
    _, _, vt = np.linalg.svd(A)

    # pick smallest number & normalization
    H = np.reshape(vt[-1], (3, 3))
    H = H / (H[2, 2]+1e-8)
    return H
```

Warping images

After finding the homography matrix, we need to know the size of warped image and how many pixels we should shift left. The following function is to return the two warped images given the two images on H. Note that the H should be used to transform the first image. (I use this for the base cases, and loop through all images with all H to find max/min x&y.

```python
def patch_images(img1, img2, H):
    # get an affine transformation matrix
    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 = np.dot(H,left_down.T) / np.dot(H, left_down.T)[2]
    warped_left_up = np.dot(H, left_up.T) / np.dot(H, left_up.T)[2]
    warped_right_down =  np.dot(H , right_down.T) / np.dot(H, right_down.T)[2]
    warped_right_up = np.dot(H, right_up.T) / np.dot(H, right_up.T)[2]

    x1 = min(warped_left_up[0], warped_left_down[0], warped_right_down[0], warped_right_up[0], 0)
    x2 = max(warped_left_up[0], warped_left_down[0], warped_right_down[0], warped_right_up[0], img2.shape[1])
    y1 = min(warped_left_up[1], warped_left_down[1], warped_right_down[1], warped_right_up[1], 0)
    y2 = max(warped_left_up[1], warped_left_down[1], warped_right_down[1], warped_right_up[1], img2.shape[0])
    width = int(x2 - x1)
    height = int(y2 - y1)
    size = (width, height)

    A = np.float32([[1, 0, -x1], [0, 1, -y1], [0, 0, 1]])
    warped1 = cv2.warpPerspective(src=img1, M=A@H, dsize=size, flags=cv2.INTER_NEAREST)
    warped2 = cv2.warpPerspective(src=img2, M=A, dsize=size, flags=cv2.INTER_NEAREST)

    cv2.imwrite('warped1.jpg', warped1)
    cv2.imwrite('warped2.jpg', warped2)

    return warped1, warped2
```

Blending

I use linear blending for challenge and base images, and the alpha is multiplied on the first image (left image). I set the alpha to decrease from 1 to 0 uniformly by setting a linspace and choose the value according to the index. (Here I set a mask to mask out the unwanted 0 regions. To make the padded areas not conflict to the black areas in the image, I clip the value of the original image from 3 to 255, which makes it distinguishable form padded area while having little visual effect.)

```python
def blend(img1, img2, detect_threshold=6):
    height, width, _ = img1.shape
    img1_mask = np.zeros((height, width), dtype=np.int16)
    img2_mask = np.zeros((height, width), dtype=np.int16)

    # find locations of the non-black pixels in both images
    for i in range(height):
        for j in range(width):
            if np.sum(img1[i, j]) > detect_threshold:
                img1_mask[i, j] = 1
            if np.sum(img2[i, j]) > detect_threshold:
                img2_mask[i, j] = 1

    overlap_mask = img1_mask * img2_mask
    blended_image = np.zeros_like(img1, dtype=np.float32)

    for i in range(height):
        # create a scalar mask for blending if needed
        if np.count_nonzero(overlap_mask[i]) > 0:
            left_most = width
            right_most = 0
            for j in range(width):
                if overlap_mask[i, j] == 1:
                    left_most = min(left_most, j)
                    right_most = max(right_most, j)
            blend_width = right_most - left_most + 1
            blend_mask = np.linspace(1, 0, blend_width)

        # fill in the values
        for j in range(width):
            if overlap_mask[i, j] == 1:
                # perform blending
                alpha = blend_mask[j - left_most]
                blended_image[i, j] = alpha * img1[i, j] + (1 - alpha) * img2[i, j]
            elif img1_mask[i, j] == 1:
                blended_image[i, j] = img1[i, j]
```

```
            else:
                blended_image[i, j] = img2[i, j]

    blended_image = np.clip(blended_image, 0, 255).astype(np.uint8)
    return blended_image
```

Aditional component on Challenge part

Histogram equalization

I convert it to YUV color space and equalize only on the lighting [:, :, 0].

Gamma correction

Directly apply histogram transformations on images 3 and 5 will produce many noises because it is overexposed (many values are the same). So, I do gamma correction on the two images to make them look normal. If you do not apply gamma correction, the image can not align well (maybe caused by inaccuracies of SIFT).

```
def adjust_gamma(image, gamma=1.0):
    # build a lookup table mapping the pixel values [0, 255] to
    # their adjusted gamma values
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255 for i in np.arange(0, 256)]).astype("uint8")
    # apply gamma correction using the lookup table
    for i in range(3):
        for j in range(image.shape[0]):
            for k in range(image.shape[1]):
                image[j][k][i] = table[image[j][k][i]]
    return image
```

The effect: before / after

Task gain componsation

I find the optimal gs by setting their partial direvatives to 0. I write the partial direvatives of each in the form of gs and solve the matrix using psuedo-inverse.

The derivation is as follows:

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

one image pair (ij), $N_{ij} = N_{ji}$, total e for this pair e_{ij}:

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

$$\frac{\partial e_{ij}}{\partial g_i} = (N_{ij}/\sigma_N^2)\,4\,(g_i \bar{I}_{ij} - g_j \bar{I}_{ji})\,\bar{I}_{ij} + \frac{2(g_i - 1)}{\sigma_g^2}$$

we want $\sum \frac{\partial e_{ij}}{\partial g_i} = 0 \rightarrow$ we can do division directly

$$\frac{\partial e_{ij}}{\partial g_i} = (2N_{ij}\bar{I}_{ij}^2/\sigma_N^2 + 1/\sigma_g^2)\,g_i + (-2N_{ij}\bar{I}_{ij}\bar{I}_{ji}/\sigma_N^2)\,g_j - N_{ij}/\sigma_g^2$$

$$\frac{\partial e_{ij}}{\partial g_j} = (-2N_{ij}\bar{I}_{ij}\bar{I}_{ji}/\sigma_N^2)\,g_i + (2N_{ij}\bar{I}_{ji}^2/\sigma_N^2 + 1/\sigma_g^2)\,g_j - N_{ij}/\sigma_g^2$$

$$\Rightarrow \underset{A}{\begin{bmatrix} \cdots \end{bmatrix}} \underset{x}{\begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_n \end{bmatrix}} = \underset{b}{\begin{bmatrix} \cdots \end{bmatrix}}$$

I thought the the σ_N and σ_g is the standard deviation of N and G, but it turns out that σ_N should be set properly, or it will be really large number, making the g become all 1. Since it will be troublesome if you want to put σ_g when calculating the error, so I also make a tunable parameter.

```python
def task_gain_compensation(images, sigma_n=10, sigma_g=0.9):
    # extract overlap pairs
    n_images = len(images)

    coefficients = np.zeros((n_images, n_images, 3))
    results = np.zeros((n_images, 3))

    # fill in the matrix
    for i in range(n_images-1):
        for j in range(i+1, n_images):
            I_ij, I_ji, N_ij = calculate_pair(images[i], images[j])
            if N_ij == 0:
                continue
            # /1e6 for numerical stability
            coefficients[i][i] += N_ij * ( (2 * I_ij ** 2 / sigma_n ** 2) + (1 / sigma_g ** 2) ) / 1e6
            coefficients[i][j] -= (2 / sigma_n ** 2) * N_ij * I_ij * I_ji / 1e6
            coefficients[j][i] -= (2 / sigma_n ** 2) * N_ij * I_ji * I_ij / 1e6
            coefficients[j][j] += N_ij * ( (2 * I_ji ** 2 / sigma_n ** 2) + (1 / sigma_g ** 2) )  / 1e6
            results[i] += N_ij / sigma_g ** 2  / 1e6
            results[j] += N_ij / sigma_g ** 2  / 1e6

    gains = np.zeros_like(results)
    for channel in range(coefficients.shape[2]):
        coefs = coefficients[:, :, channel]
        res = results[:, channel]
        # solve with psuedo-inverse
        gains[:, channel] = np.linalg.pinv(coefs) @ res

    max_pixel_value = np.max([image for image in images])
    print(gains)
    # normalize
    if gains.max() * max_pixel_value > 255:
        gains = gains / (gains.max() * max_pixel_value) * 255

    return gains
```

I compute the gain for 3 channels separately. Here is the function to fine N_ij, I_ij, I_ji given two images.

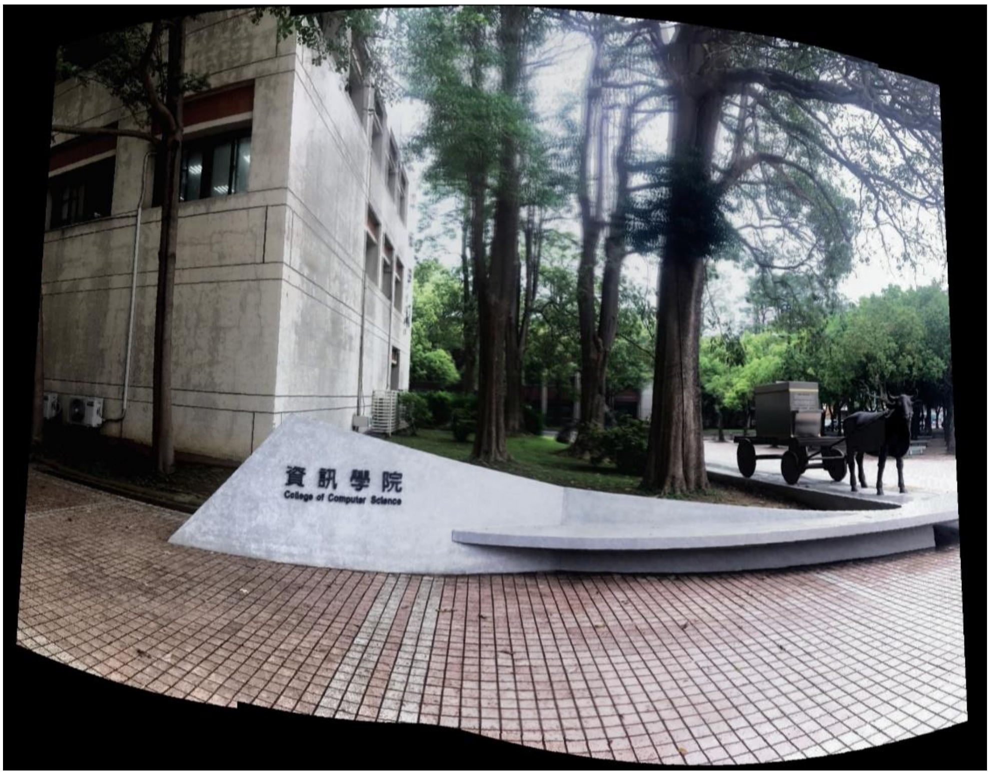
```
def calculate_pair(img1, img2):
    width, height = img1.shape[1], img1.shape[0]
    img1_mask = np.zeros((height, width), dtype=np.int16)
    img2_mask = np.zeros((height, width), dtype=np.int16)
    detect_threshold = 6
    for i in range(height):
        for j in range(width):
            if np.sum(img1[i, j]) > detect_threshold:
                img1_mask[i, j] = 1
            if np.sum(img2[i, j]) > detect_threshold:
                img2_mask[i, j] = 1

    overlap_mask = img1_mask * img2_mask
    N_ij = np.count_nonzero(overlap_mask)
    if N_ij == 0:
        return None, None, 0
    img1 = img1.astype(np.float32)
    img2 = img2.astype(np.float32)
    overlap_mask = overlap_mask.astype(np.float32)
    I_ij = np.sum(img1 * np.stack([overlap_mask, overlap_mask, overlap_mask], axis=2), axis=(0, 1)) / N_ij
    I_ji = np.sum(img2 * np.stack([overlap_mask, overlap_mask, overlap_mask], axis=2), axis=(0, 1)) / N_ij
    return I_ij, I_ji, N_ij
```

Results

Base: linear projection / cylindrical projection

Challenge



Blending method anaylysis

1. Linear blending method

   As described in the previous context.

2. Constant blending method

   Setting the alpha to 0.5 directly.

As you can see, constant blending will have obvious borders of the overlapping region, but the linear blending will look smooth. This is because when the point is near the border from the left, the value of left image will be cosidered more (having a huger alpha), and when it is near the border, the value will almost indentical to the value of left image.