Report

```python
# img_1 = base image
img_1, img_gray_1 = read_img('./test/m1.jpg')
img_1 = np.array(img_1) # 756 x 1008 x 3
img_gray_1 = np.array(img_gray_1) # 756 x 1008

img_2, img_gray_2 = read_img('./test/m2.jpg')
img_2 = np.array(img_2) # 756 x 1008 x 3
img_gray_2 = np.array(img_gray_2) # 756 x 1008

SIFT_Detector = cv2.SIFT_create()

kp_1, des_1 = SIFT_Detector.detectAndCompute(img_gray_1, None)
kp_2, des_2 = SIFT_Detector.detectAndCompute(img_gray_2, None)

# kp : number of keypoints
# des : number of keypoints x 128
```

Firstly, I obtain the two images from the file names. img_1 would be the first (left) image to be combined with img_2 (right image). Then I use cv2.SIFT_create() to obtain the key points and descriptors.

```python
matching_kp = [] # each entry is a matching kp [image_1_index, image_2_index] between both iamges

for j in range(len(kp_1)):

    min_diff = [inf, inf] # min norm diff
    min_index = [-1, -1] # index of min norm diff

    for k in range(len(kp_2)):
        for l in range(2):
            diff = np.abs(np.linalg.norm(des_1[j] - des_2[k]))
            if diff < min_diff[l]:
                min_diff[l] = diff
                min_index[l] = k
                break

    ### Lowe's Ratio Test
    if min_diff[0] < 0.7 * min_diff[1]:
        matching_kp.append([j, min_index[0]])
```

Next step is feature matching, I need to compare every single key point on the first image with the every key point on the second image. j corresponds to the current key point index on img_1 and k corresponds to the current key point index on img_2. I use 2-NN to find the 2

smallest distance. I calculate the distance between the current key point on both images, if they're smaller than an entry on min_diff, I replace that entry with the current one. I also use min_index to track the index of the key point on the img_2 that might potentially match keypoint of index j in image one.

After comparing a key point in image_1 with every key point on image_2, I use Lowe's Ratio Test to see if the key points with the smallest distance is a good match by comparing it with the key point with the 2nd smallest distance, here I use threshold = 0.7 . If it fulfills the test, then I add the matching key point index of both img_1 and img_2.

```
p1 = []
p2 = []

# Each point (x,y) in p1 of image_1 corresponds to point p2 of image_2

for i in range(len(matching_kp)):
    p1.append([kp_1[matching_kp[i][0]].pt[0], kp_1[matching_kp[i][0]].pt[1]])
    p2.append([kp_2[matching_kp[i][1]].pt[0], kp_2[matching_kp[i][1]].pt[1]])

np.save('p1.npy', p1)
np.save('p2.npy', p2)

p1 = np.load('p1.npy')
p2 = np.load('p2.npy')
```

After that, I convert those key point indexes to actual coordinates of the key points. Every coordinate of index i in p1 corresponds to a key point in img_1 that matches a coordinate of index I in p2 that corresponds to a key point in img_2.

```
inlier_index = random.sample(range(len(p1)), 4)
```

Next is RANSAC, I randomly select 4 random indexes of a pair of matching key point to pick as inlier points.

```
A = []
b = []

for i in range(4):
    x1 = p1[inlier_index[i]][0]
    y1 = p1[inlier_index[i]][1]
    x2 = p2[inlier_index[i]][0]
    A.append([x1, y1, 1, 0, 0, 0, -x1*x2, -y1*x2])
    b.append([x2])

for i in range(4):
    x1 = p1[inlier_index[i]][0]
    y1 = p1[inlier_index[i]][1]
    y2 = p2[inlier_index[i]][1]
    A.append([0, 0, 0, x1, y1, 1, -x1*y2, -y1*y2])
    b.append([y2])
A = np.array(A)
b = np.array(b)
```

to 1

A

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

Then I create the matrix A and b from those randomly selected keypoints. b here would refer to the matrix on the right hand side of the operation.

```
H = np.matmul( np.matmul( np.linalg.inv( np.matmul(A.T, A) ), A.T), b)
# u, s, vh = np.linalg.svd(A)
# H = vh[7]

H = np.append(H, 1)
H.resize(3,3)
```

Since h33 = 1, we need to solve a matrix equation Ax = b. Here I obtain the Moore Penrose pseudo inverse of A, which then I multiply with b.

$$A^{+} = (A^{T}A)^{-1}A^{T}$$

That way, I obtain x with refers to the matrix with entry h11 until h32. I construct the homography matrix by appending h33 = 1 then changing the dimension to a 3x3 matrix.

```
num_support = 0

for i in range(len(p1)):
    x_p1 = p1[i][0]
    y_p1 = p1[i][1]
    x_p2 = p2[i][0]
    y_p2 = p2[i][1]

    p_temp = np.matmul(H, [[x_p1],[y_p1],[1]]) # projection result
    x_temp = p_temp[0] / p_temp[2]
    y_temp = p_temp[1] / p_temp[2]

    if i not in inlier_index:
        dist = math.dist([x_p2,y_p2],[x_temp,y_temp])
        if dist < 1:
            num_support += 1
```

I then would check every point in p1 and project it using the homography matrix to obtain p_temp (p2'). From p_temp I can obtain the projected point (x_temp, y_temp). If the current index is not the inlier points, I obtain the distance between p_temp and corresponding point in p2. I keep track of distances that are smaller than a threshold (here I use 1) by using a counter "num_support".

```
if num_support > 250:
    break
```

If num_support is larger than a threshold (here I use 250), then the loop ends and we use this current homography matrix. Otherwise the process repeats until we get a good homography matrix.

```
x_corner = [0, 0, np.shape(img_1)[0], np.shape(img_1)[0]]
y_corner = [0, np.shape(img_1)[1], 0, np.shape(img_1)[1]]

# Transform the corners

for i in range(2):
    x_p1 = x_corner[i]
    y_p1 = y_corner[i]
    p_temp = np.matmul(H, [[x_p1],[y_p1],[1]]) # projection result
    x_corner[i] = (p_temp[0] / p_temp[2])[0]
    y_corner[i] = (p_temp[1] / p_temp[2])[0]
```

Next I obtain the coordinate of the 4 corners of img_1 and project those 4 points using the homography matrix.

```
x1_prime = min(int(min(x_corner)),0)
y1_prime = min(int(min(y_corner)),0)

size = [np.shape(img_2)[1] + np.abs(x1_prime), np.shape(img_2)[0] + np.abs(y1_prime)]
```

- Size we need = (w2 + abs(x1'), h2 + abs(y1'))

And use them to determine the size of the new combined image.

```
A = np.array([[1.0, 0, -x1_prime],[0, 1.0, -y1_prime],[0, 0, 1.0]],dtype=np.float64)
H = np.matmul(A, H)
warped1 = cv2.warpPerspective(src=img_1,M=H,dsize=size)
warped2 = cv2.warpPerspective(src=img_2,M=A,dsize=size)
```

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

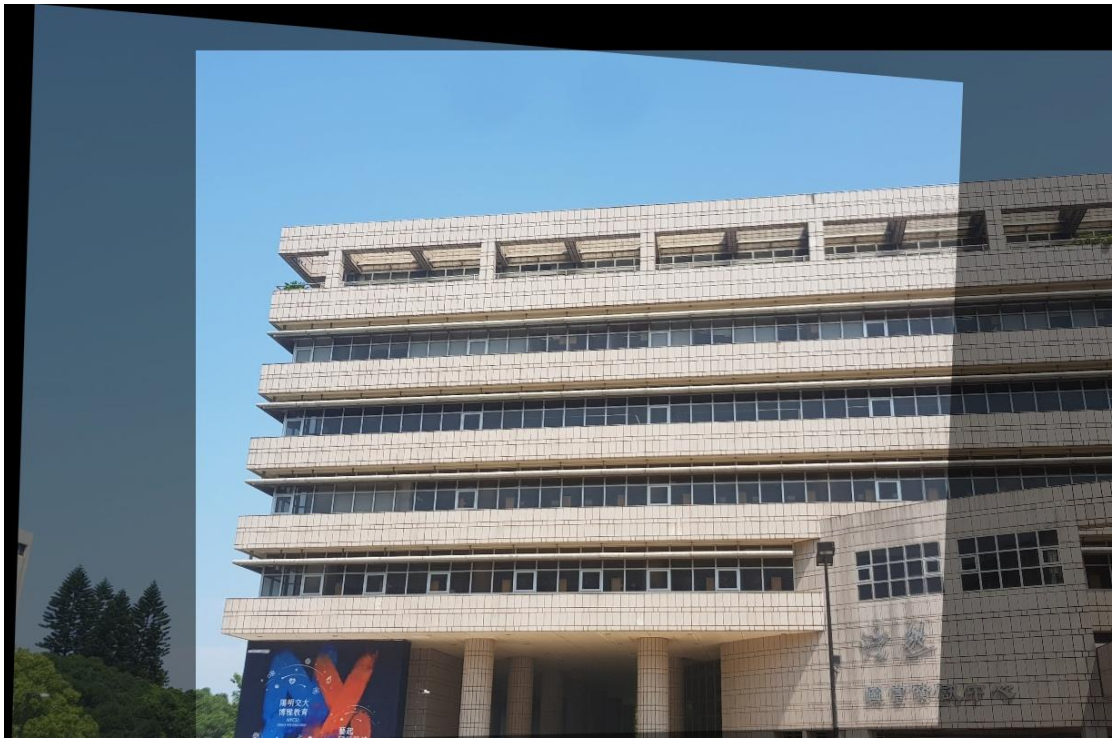
In this case $\Delta x = -x1^{'}, \Delta y = -y1^{'}$

Then I create the Affine translation matrix A and multiply H with it. Then I project img_1 with H and img_2 with A to obtained the projected images warped1 and warped2.

```
alpha = 0.5
beta = 0.5
dst = cv2.addWeighted(warped1, alpha, warped2, beta, 0.0)
creat_im_window("Image",dst)
cv2.imwrite("Blend_2.jpg",dst)
im_show()
```

I combine the two images using addWeighted with alpha = beta = 0.5. Then I show the combined image and save it as a file.

The result of combining m1 and m2



This process can be repeated by changing the file name of img_1 and img_2 in the beginning and the file destination of the final image.