ECE 561: Primitive Panorama Stitching

Soo Min Kwon

September 10, 2021

1 Wide-angle scene

The following is two pictures that captures the "wide-angle scene":

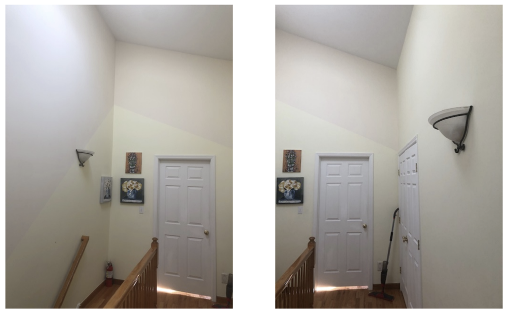


Figure 1: Two images to be used for stitching via primitive panorama stitching.

In Figure 1, as we can see, there is an overlap between the two pictures. Both pictures include the door and parts of the paintings. Our objective is to stitch the images together like a panorama, making it seem like it was taken as one wide-scene image. The steps we are going to be taking in this assignment are the following: (1) perform feature extraction via Scale Invariant Feature Transform (SIFT), (2) match descriptors and correspondence points, (3) run RANSAC to find the homography matrix for image stitching. Each step is organized into its own section, followed by a brief introduction and explanation.

2 SIFT for key-points and descriptors

Firstly, we compute the SIFT key points and descriptors by using the Python package, OpenCV. Briefly, SIFT uses a difference of Gaussians for scale-space feature detection. The process can simply be described as the following:

1. Take 16×16 square window around detected features and compute gradient orientation for each pixel
2. Discard "weak" edges through a thresholding mechanism
3. Create a histogram for kept edges and divide the windows into 4×4 windows
4. Compute an orientation histogram for each cell (16 cells $\times$ 8 orientations $= 128$ dimensional descriptor)

By performing SIFT and plotting the key points onto the images, we obtain the following figure:

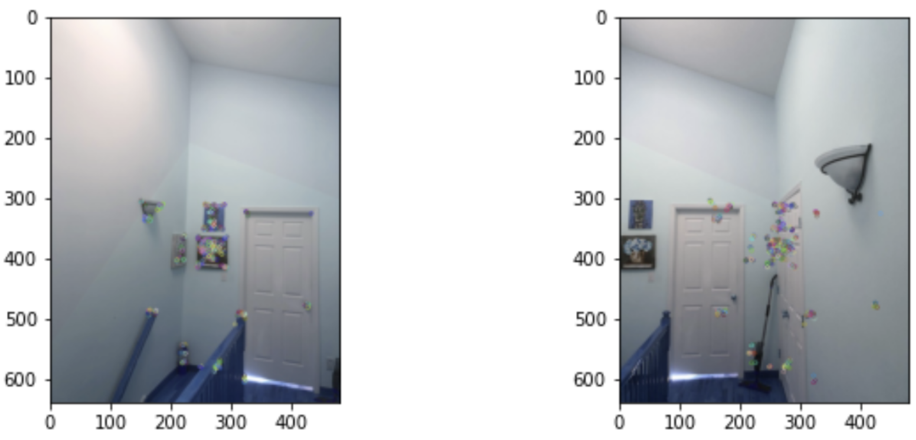


Figure 2: Key points detected by SIFT onto wide-angle scene images

We can obtain these figures by running the following code:

```
def sift_and_plot(first_img, second_img):
    # initiate SIFT detector
    orb = cv2.ORB_create()

    first_img = cv2.imread(first_img)
    second_img = cv2.imread(second_img)

    # find the keypoints and descriptors with SIFT
    kp1, des1 = orb.detectAndCompute(first_img, None)
    kp2, des2 = orb.detectAndCompute(second_img, None

    img1 = cv2.drawKeypoints(first_img, kp1, first_img)
    img2 = cv2.drawKeypoints(second_img, kp2, second_img)

    # plotting images
    plt.imshow(img1)
    plt.show()

    plt.imshow(img2)
    plt.show()
```

Running the lines of code in Python with the correct dependencies will reproduce the images in Figure 2.

3 Matching correspondence points

Once we are able to obtain feature key points and descriptors from SIFT, we can perform feature matching between the descriptors in both images by defining a distance function that compares the descriptors. The approach is straightforward:

1. Define a ratio threshold, ϵ
2. Compute
$$\frac{||f_1 - f_2||}{||f_1 - f_2'||} \leq \epsilon, \tag{1}$$
where f_1 is description vector from the first image and f_2 and f_2' are two different descriptors from the second image
3. Gather k of the closest vectors from the threshold

Since the code for this is rather messy, we first give the **pseudocode** for computing correspondence points and give code using OpenCV that we use to verify our results. For the OpenCV code, we use `cv2.BFMatcher` and `cv2.knnMatch` to choose the k closest descriptors.

The **pseudocode**:

```
def correspondence_points(ratio, image_1, image_2):
    key_points1, descriptor_1 = SIFT(image_1)
    key_points2, descriptor_2 = SIFT(image_2)

    descriptor_matches = []

    for i in descriptor_1:
        for j in descriptor_2:
            a = distance(i, j)
            b = distance(i, j+1)
            if a < b * ratio:
                descriptor_matches.append(i)

    return descriptor_matches
```

In essence, the code above compares descriptors from the first and second image, and if they are within a certain threshold, they become added to a list which we use for further analysis.

Below is a **working implementation** using OpenCV in Python to compute correspondence points:

```
def sift_and_match(ratio, first_img, second_img):
    # initiate SIFT detector
    orb = cv2.ORB_create()

    first_img = cv2.imread(first_img)
    second_img = cv2.imread(second_img)

    # find the keypoints and descriptors with SIFT
    kp1, des1 = orb.detectAndCompute(first_img, None)
    kp2, des2 = orb.detectAndCompute(second_img, None)

    matches = cv2.BFMatcher()
    match_des = matches.knnMatch(des1, des2, k=2)

    total_points = []
    total_matches = []

    for i, j in match_des:
        if i.distance < ratio * j.distance:
            total_points.append((i.trainIdx, i.queryIdx))
            total_matches.append([i])

    return kp1, kp2, des1, des2, total_points, total_matches
```

Upon using `cv2.drawMatches` on the `total matches`, we can obtain the following plot:

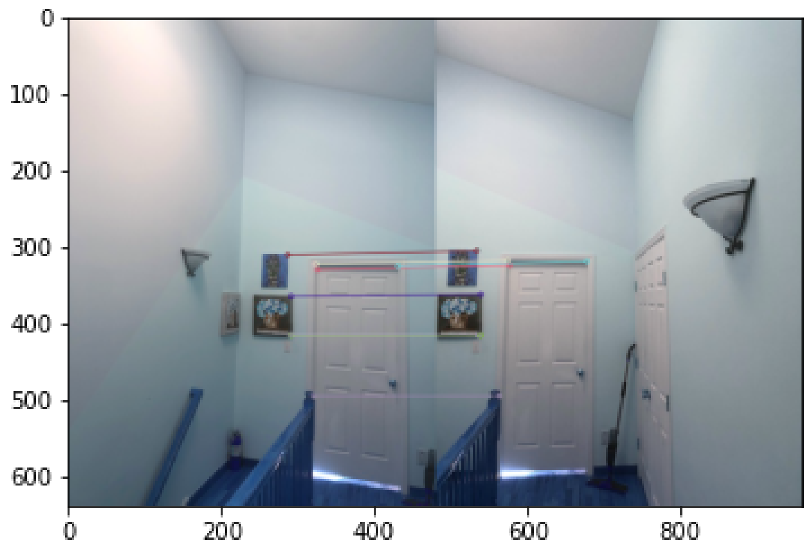
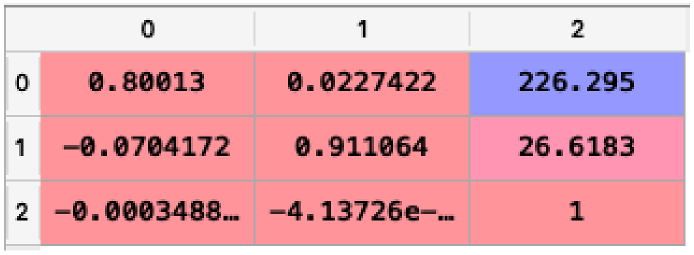


Figure 3: Figure of matching correspondence points via feature matching

4 RANSAC, homography, image stitching

Lastly, we run RANSAC to estimate the homography matrix and use `cv2.warpPerspective` to stitch the image together. The following is the solved homography matrix, followed by the stitched image:



	0	1	2
0	0.80013	0.0227422	226.295
1	-0.0704172	0.911064	26.6183
2	-0.0003488…	-4.13726e-…	1

Figure 4: Computed homography matrix through RANSAC

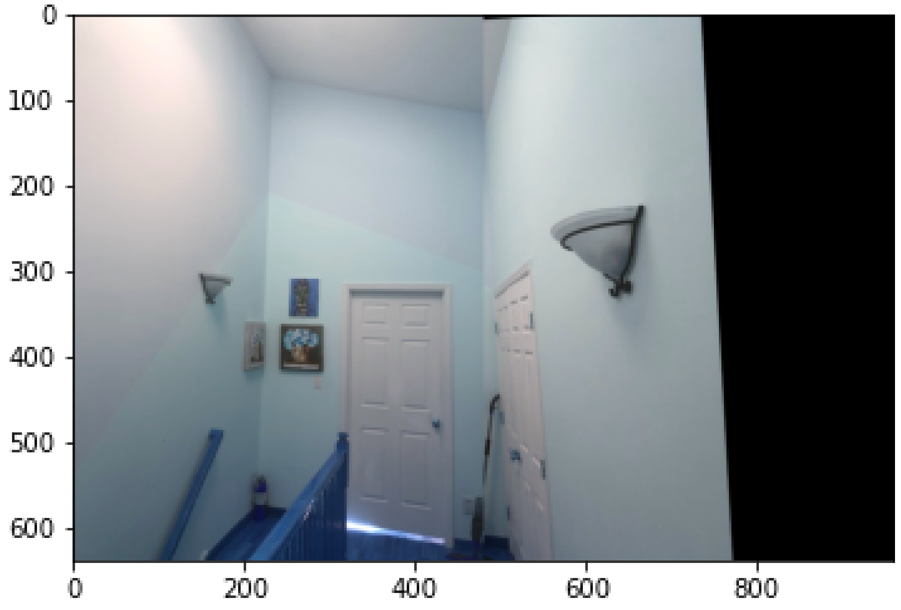


Figure 5: Two images from Figure 1 stitched together as a panorama

The following is the true image (taken later in the day with laundry):

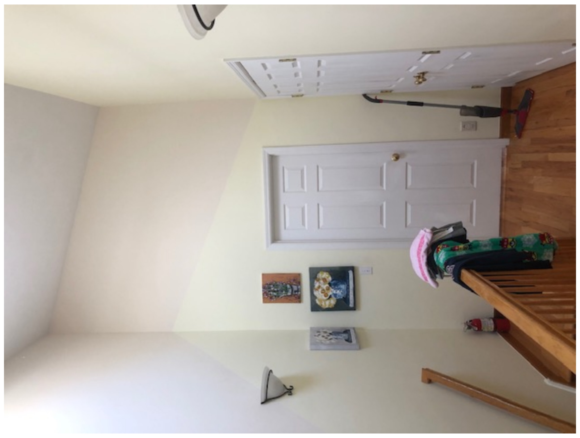


Figure 6: Original picture of hallway

Some comments:

- Clearly, comparing Figures 5 and 6, we have some flaws in the stitched image in between columns 400 to 600 and 800+. There are actually ways to get rid of this, and the idea is to use a ”mask” or weighted matrix in stitching the image together.
- Overall, I thought this was an exciting application of using SIFT, RANSAC, and homographic matrices.