Feature Extraction

I use ORB as our feature detector. (I also implement SIFT feature detector but not used. However, you can find the implementation as comment on my code.)

I use BruteForce Matcher to match detected features. Then to getting best features I used bf.match and store the points of matched features.

```
orb = cv2.ORB_create(nfeatures=nfeatures, scoreType=cv2.ORB_FAST_SCORE)
kp1, des1 = orb.detectAndCompute(src, None)
kp2, des2 = orb.detectAndCompute(tgt, None)


bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des1, des2)


matches = sorted(matches, key=lambda x: x.distance)
```

Feature Matching

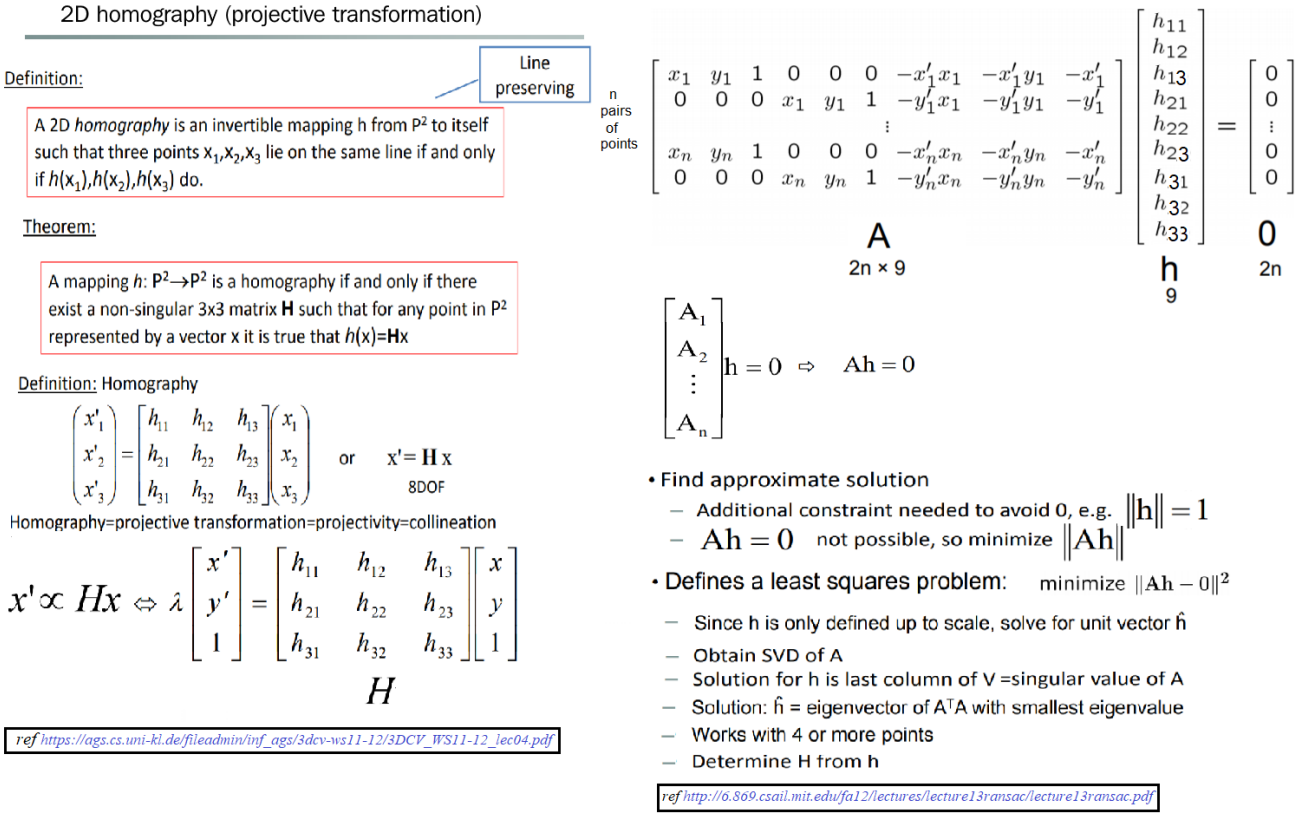
This function matches features in the src and tgt picture to return points of correspondonces of the same features between the two images

```
bestPoints = []
for i in range(len(matches)):
    x1y1 = np.float32(kp1[matches[i].queryIdx].pt)
    x2y2 = np.float32(kp2[matches[i].trainIdx].pt)
    feature = list(map(int, list(x1y1) + list(x2y2) + [matches[i].distance]))
    bestPoints.append(feature)

bestPoints = np.array(bestPoints)
```

Finding Homography

The most difficult part is the Homography part. I found Homography implementation while researching.

The right part is the implementation part for us.



First of all, we create our own array that we will use. Then we open the pointin values that come to us as parameters with the for loop. Then we insert the values into certain matrix operations. After that, we convert the array to which we added the matrices into a numpy array. Then we find the SVD value and get the vh value and convert it back to the numpy array. After dividing the last element of the Array by the last element of the last element of the Array, we reshape it to be 3.3.

Here is the implementation of my Homography Function.

```
def homography(point):

    homographyArr = []
    for i in range(len(point)):
        x = point[i][0]
        y = point[i][1]
        a = point[i][2]
        b = point[i][3]

        homographyArr.append([x, y, 1,
                  0, 0, 0,
                  -a*x, -a*y, -a])

        homographyArr.append([0, 0, 0,
                  x, y, 1,
                  -b*x, -b*y, -b])

    homographyArr = np.array(homographyArr)

    svdValue = np.linalg.svd(homographyArr)

    vh = svdValue[2]

    vhArray = np.array(vh)

    vhLength = len(vhArray)

    L = vhArray[vhLength-1] / vhArray[vhLength-1, -1]

    homographyResult = L.reshape(3, 3)

    return homographyResult
```

Merging by Transformation

Defining dictionaries for various pyramids

```
g_pyramids = {}
l_pyramids = {}
W = images[0].shape[1]
```

Calculating Gaussian, Laplacian pyramids for various images before hand

```
for i in range(len(images)):
    # Gaussian Pyramids
    G = images[i].copy()
    g_pyramids[i] = [G]
    for k in range(5):
        G = cv2.pyrDown(G)
        g_pyramids[i].append(G)

    # Laplacian Pyramids
    l_pyramids[i] = [G]
    for j in range(len(g_pyramids[i])-2, -1, -1):
        G_up = cv2.pyrUp(G)
        G = g_pyramids[i][j]
        L = cv2.subtract(G, G_up)
        l_pyramids[i].append(L)
```

Blending Pyramids

```
common_mask = masks[0].copy()
common_pyramids = [l_pyramids[0][i].copy() for i in range(len(l_pyramids[0]))]
```

We take one image, blend it with our final image, and then repeat for n images

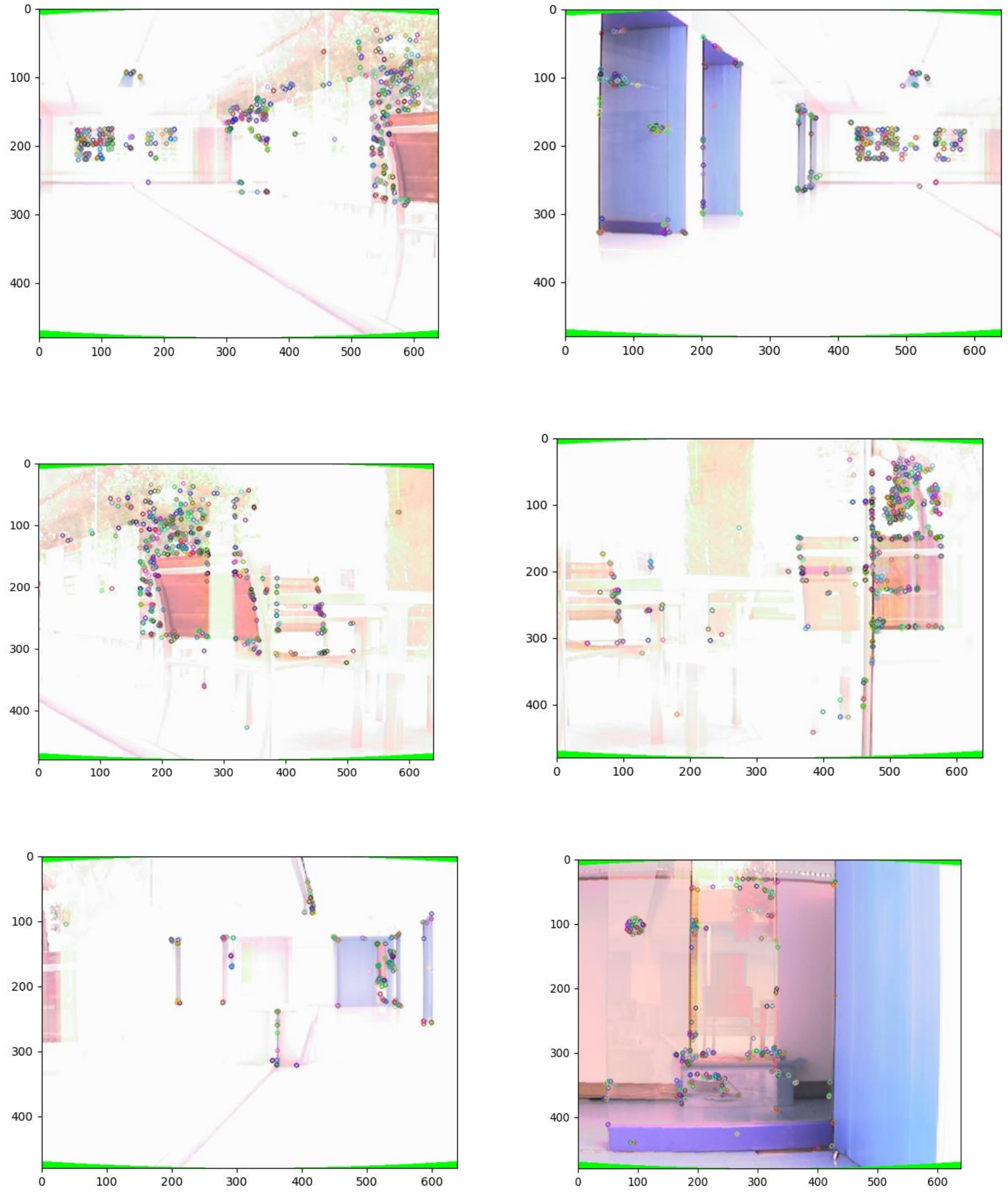
```
for i in range(1, len(images)):
    y1, x1 = np.where(common_mask == 1)
    y2, x2 = np.where(masks[i] == 1)

    if np.max(x1) > np.max(x2):
        left_py = l_pyramids[i]
        right_py = common_pyramids

    else:
        left_py = common_pyramids
        right_py = l_pyramids[i]
```

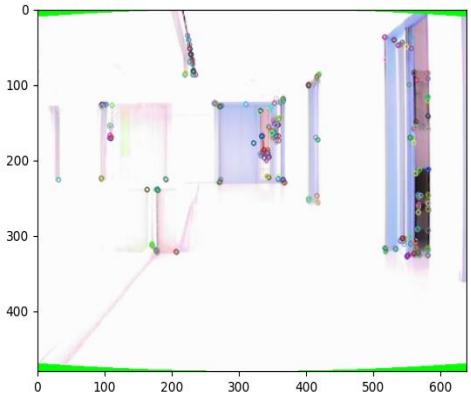
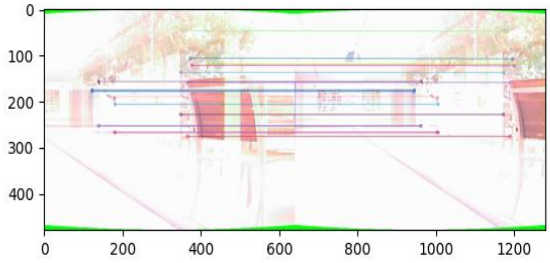
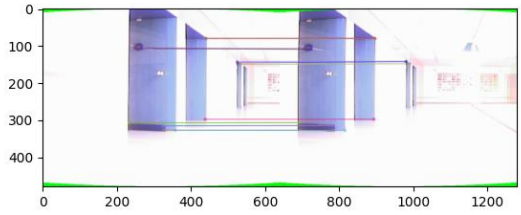
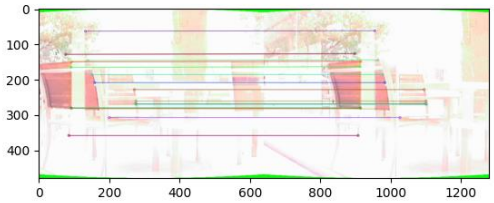
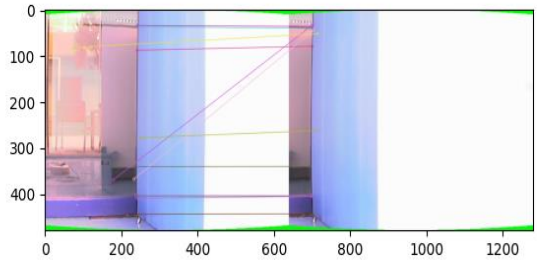
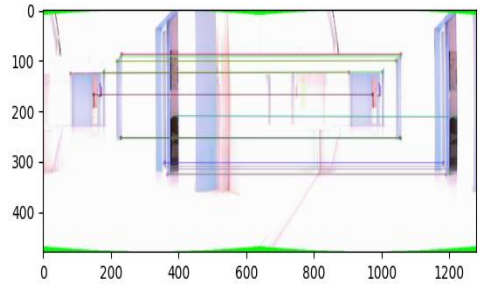
To check if the two pictures need to be blended are overlapping or not

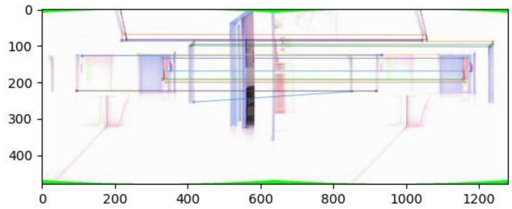
```
mask_intersection = np.bitwise_and(common_mask, masks[i])
```

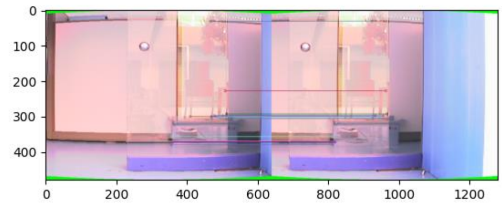
```
    if True in mask_intersection:
        # If images blend, we need to find the center of the overlap
        y, x = np.where(mask_intersection == 1)
        x_min, x_max = np.min(x), np.max(x)

        # We get the split point
        split = ((x_max-x_min)/2 + x_min)/W

        # Finally we add the pyramids
        LS = []
        for la, lb in zip(left_py, right_py):
            cols = la.shape[1]
            ls = np.hstack((la[:, 0:int(split*cols)], lb[:, int(split*cols):]))
            LS.append(ls)

    else:
        LS = []
        for la, lb in zip(left_py, right_py):
            ls = la + lb
            LS.append(ls)

    # Reconstructing the image
    ls_ = LS[0]
    for j in range(1, 6):
        ls_ = cv2.pyrUp(ls_)
        ls_ = cv2.add(ls_, LS[j])

    # Preparing the common image for next image to be added
    common_image = ls_
    common_mask = np.sum(common_image.astype(bool), axis=2).astype(bool)
    common_pyramids = LS

return ls_
```

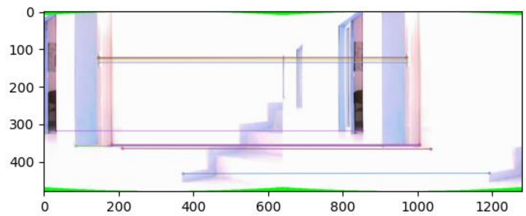
Plots showing feature points and draw matches for ORB detector

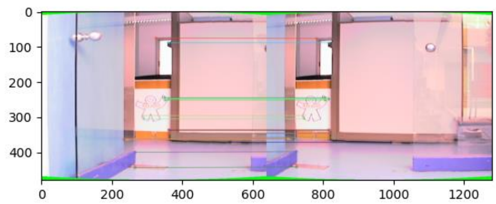


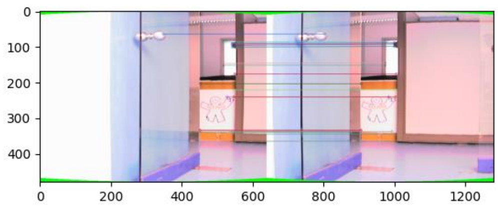







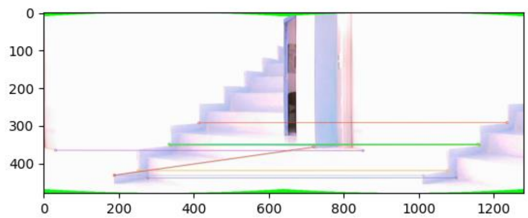

0
100
200
300
400
0 200 400 600 800 1000 1200


0
100
200
300
400
0 200 400 600 800 1000 1200

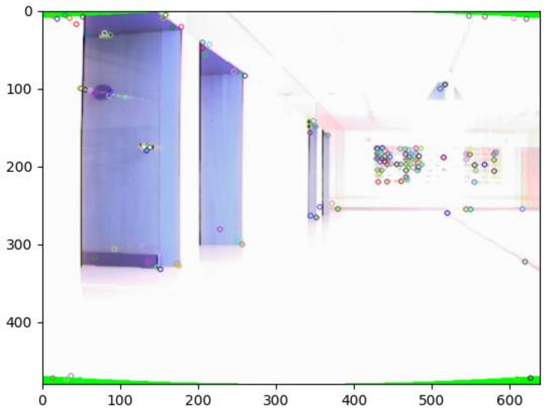

0
100
200
300
400
0 200 400 600 800 1000 1200

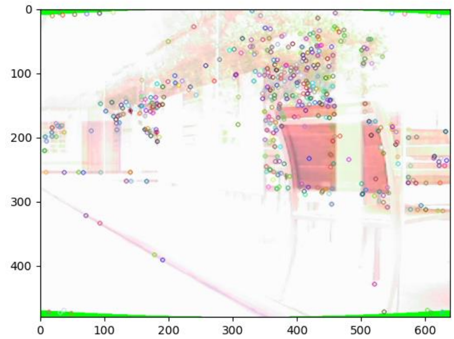

0
100
200
300
400
0 200 400 600 800 1000 1200

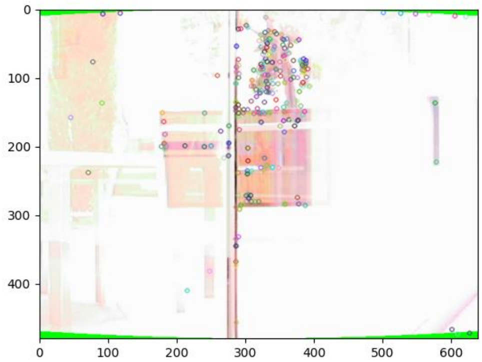

0
100
200
300
400
0 200 400 600 800 1000 1200

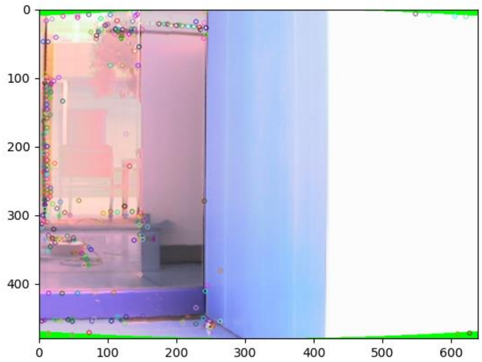

0
100
200
300
400
0 200 400 600 800 1000 1200

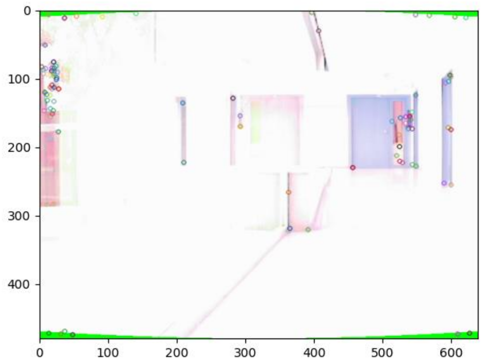
Plots showing feature points and draw matches for SIFT detector

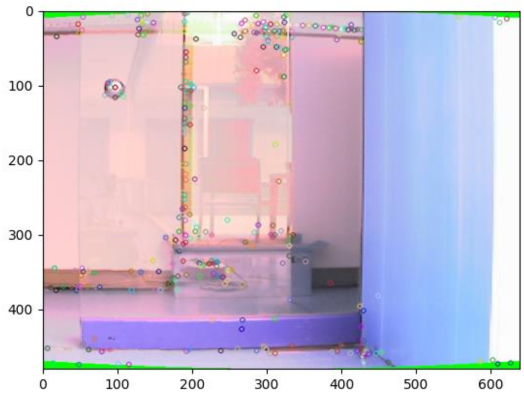


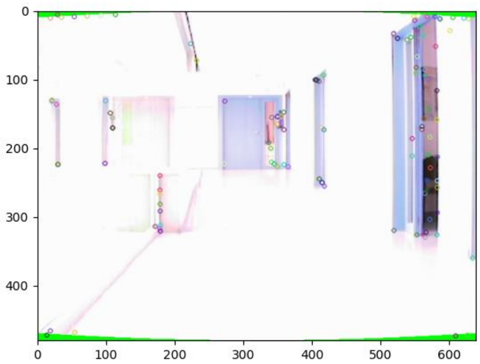


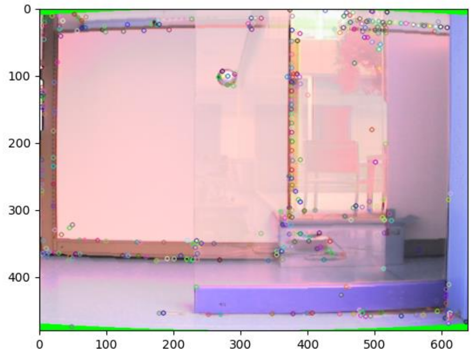


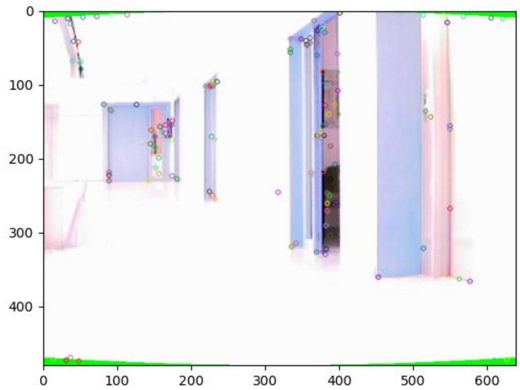


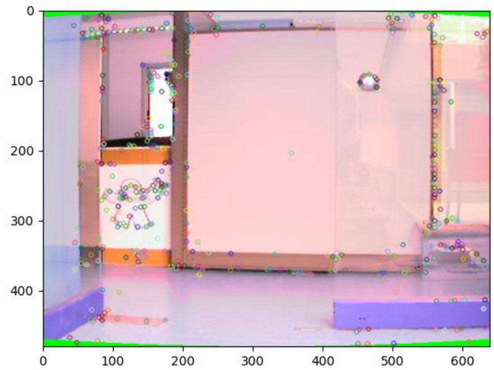


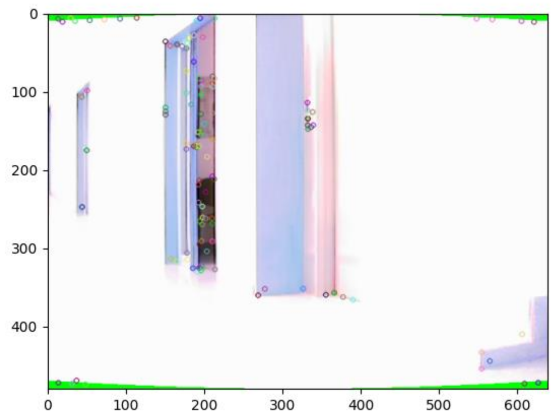


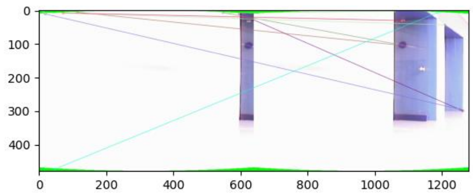

0 100 200 300 400
0 100 200 300 400 500 600

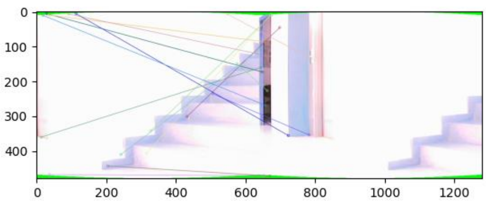

0 100 200 300 400
0 100 200 300 400 500 600


0 100 200 300 400
0 100 200 300 400 500 600


0 100 200 300 400
0 100 200 300 400 500 600


0 100 200 300 400
0 100 200 300 400 500 600

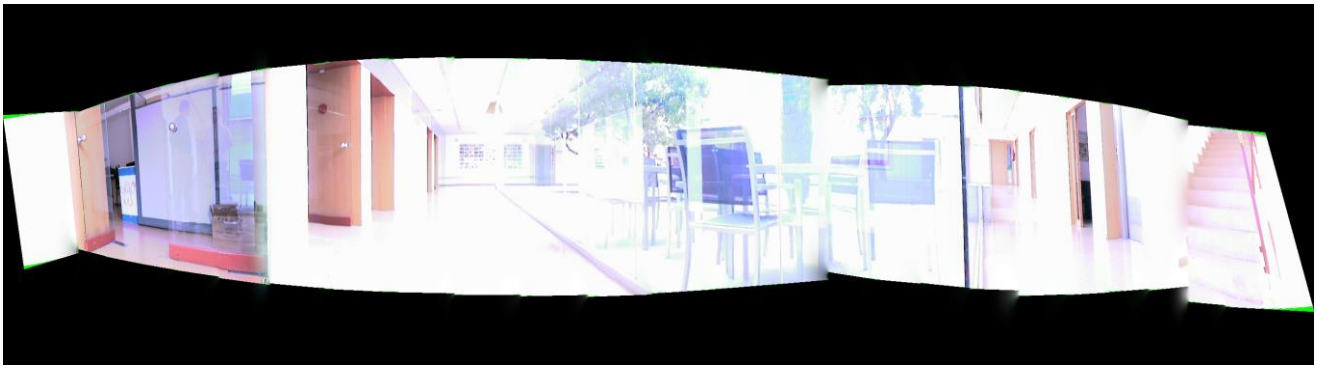

0 100 200 300 400
0 200 400 600 800 1000 1200


0 100 200 300 400
0 200 400 600 800 1000 1200

Example Output Image from My Program

Dataset Stitched Panorama 1

Dataset Stitched Panorama 3



Dataset Stitched Panorama 4

Note: I dont put all keypoint plots and drawMatches plots (it would too much plots). However, you can find it on my code as comment. As you can see below.

(line 157)

```
"""
# displaying the image with drawing matches
img3 = cv2.drawMatches(src, kp1, tgt, kp2, matches[:20], tgt, flags=2)
plt.imshow(img3), plt.show()
# displaying the image with keypoints as the output on the screen
img_1 = cv2.drawKeypoints(src, kp1, src)
plt.imshow(img_1), plt.show()
"""
```

As a result, I make a program which can image stitched with keypoint descriptors. Its run time depends on image counts. I use 3 building images in the pdf and my program give the output correctly in 1 minutes. However, when we try to stitch in the dataset images which count is 32, its run time takes 5-10 min.

RESOURCES

1- https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html

2- https://www.analyticsvidhya.com/blog/2019/10/detailed-guide-powerful-sift-technique-image-matching-python/

3- https://theailearner.com/tag/laplacian-pyramid-opencv/

4- https://www.geeksforgeeks.org/feature-matching-using-orb-algorithm-in-python-opencv/