Homework 2

Anthony -22421378

1. Panorama Stitching

The classic panorama stitching pipeline normally encompasses the following operations: feature detection, usually extracted by detecting distinctive points, such as corners or edges in overlapping parts of the images, is followed by the description of such features by techniques such as SIFT but also pixel-based descriptors encoding the distinctive pattern around each of those features. Then, feature matching between images is usually done by comparing the descriptors with respect to feature matching. Once stitched, the relations between these images are given by a computed homography matrix, obtained by using techniques such as the Direct Linear Transform, followed by RANSAC to get rid of outliers. This is followed by warping one of these images to get it registered with the other. Finally, this is followed by warping and blending in which the overlapping areas of the images are feathered together to integrate seamlessly into one panorama. The final mosaic, in the end, is preserved as one single image representing the entire thing of the scene captured by the discreet frames.

2. Implementation

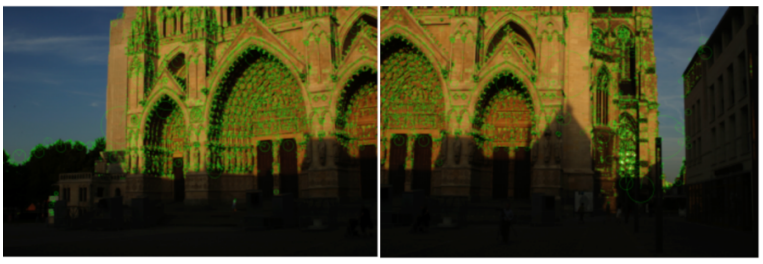
2.1 Feature detection, description and matching

For this implementation, I had used SIFT algorithm for first algorithm and concatenated pixel value for a local window. The code is described in below for detection, description and matching:

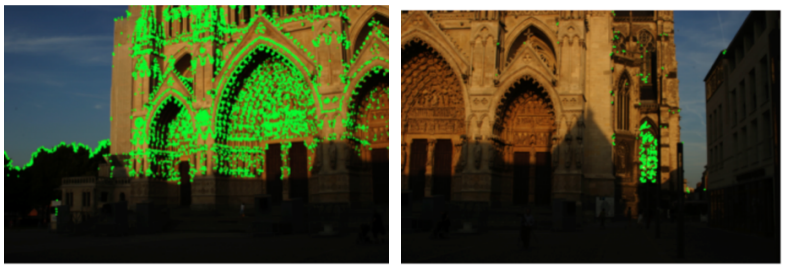
```
def knn_matching_algorithm(des1, des2, k=2):
    #NearestNeighbors model using euclidean distance because faster.
    model = NearestNeighbors(n_neighbors=k, algorithm='auto', metric='euclidean')
    model.fit(des2)
    distances, indices = model.kneighbors(des1)

    #Filter good matches based on the distances
    good_matches = []
    for i, (dist1, dist2) in enumerate(zip(distances[:, 0], distances[:, 1])):
        if dist1 < 0.75 * dist2:
            good_matches.append((i, indices[i, 0], dist1))

     return good_matches
```

```python
def local_window_descriptors(gray, window_size=9, k=0.04, threshold=0.01):
    gray_float = np.float32(gray)
    corner_response = cv2.cornerHarris(gray_float, 2, 3, k)
    corner_response = cv2.dilate(corner_response, None)
    corners = corner_response > threshold * corner_response.max()
    half_size = window_size // 2
    descriptors = []
    valid_keypoints = []
    for y in range(half_size, gray.shape[0] - half_size):
        for x in range(half_size, gray.shape[1] - half_size):
            if corners[y, x]: #Only consider the points where a corner is detected
                window = gray[y - half_size:y + half_size + 1, x - half_size:x + half_size + 1]
                descriptor = window.flatten()#Flatten the window to a 1D descriptor
                valid_keypoints.append(cv2.KeyPoint(x, y, window_size))
                descriptors.append(descriptor)
    return valid_keypoints, np.array(descriptors)
def Finding_Matches(image1,image2,method='sift'):
    image1_gray = cv2.cvtColor(image1,cv2.COLOR_BGR2GRAY)
    image2_gray = cv2.cvtColor(image2,cv2.COLOR_BGR2GRAY)
    if method =='sift': # SIFT algorithm
        print('SIFT Algorithm')
        sift =cv2.SIFT_create()
        image1_kp,image1_des = sift.detectAndCompute(image1_gray,None)
        image2_kp,image2_des = sift.detectAndCompute(image2_gray,None)
    elif method== None :#local_window_descriptors
        print('Local window')
        image1_kp,image1_des = local_window_descriptors(image1_gray)
        image2_kp,image2_des = local_window_descriptors(image2_gray)
    #i try to imlplement knn algorihtm for replace cv2.knn_matching that cannot use in this project
    good_matches = knn_matching_algorithm(image1_des, image2_des, k=2)
    return good_matches, image1_kp, image2_kp
```

Example for SIFT algorithm:



Example of Local Window with Harris algorithm for feature capturing:

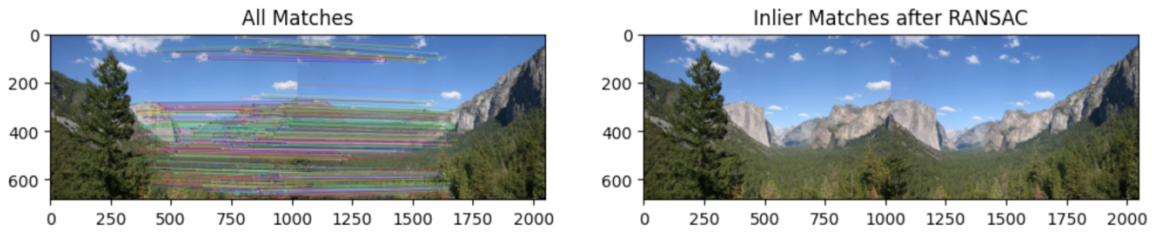


2.2 Estimation of homography

Code :

```
def ComputeRansacIterations(outlier_ratio, success_prob=0.99):
    if outlier_ratio == 1:
        return 0 #if rasnac algorithm cannot solve it return zero
    return math.ceil(math.log(1 - success_prob) / math.log(1 - (1 -
outlier_ratio)**4))
def FindHomography(Matches, BaseImage_kp, SecImage_kp):
    # If less than 4 matches found, exit the code
    if len(Matches) < 4:
        print("Not enough matches found between the images.")
        exit(0)
    Image1_pts = []
    Image2_pts = []
    for Match in Matches:
        i, j, dist = Match
        Image1_pts.append(BaseImage_kp[i].pt)
        Image2_pts.append(SecImage_kp[j].pt)
    Image1_pts = np.float32(Image1_pts)
    Image2_pts = np.float32(Image2_pts)
    #Finding the homography matrix using RANSAC
    HomographyMatrix, Status = cv2.findHomography(Image2_pts, Image1_pts,
cv2.RANSAC, 4.0)
    #Compute the number of inliers and outliers
    inliers = Status.ravel().tolist()
    inlier_count = sum(inliers)
    outlier_count = len(inliers) - inlier_count
    outlier_ratio = outlier_count / len(inliers) if len(inliers) > 0 else 0
    #Compute the outlier ratio
    iterations_needed = ComputeRansacIterations(outlier_ratio)
    print(f"Outlier Ratio: {outlier_ratio:.4f}")
    print(f"Minimum RANSAC Iterations: {iterations_needed}")
    return HomographyMatrix, Status
```

Example of this process :



2.3 Image Stitching

Code :

```
def StitchImages(image1, image2):
    SecImg_Cyl, mask_x, mask_y = ProjectOntoCylinder(image2)
    SecImg_Mask = np.zeros(SecImg_Cyl.shape, dtype=np.uint8)
    SecImg_Mask[mask_y, mask_x, :] = 255
    #Finding matches and keypoints between image1 and image2, by using sift or None
= Local window
    Matches, Base_kp, Sec_kp = Finding_Matches(image1, SecImg_Cyl,'sift')
    # Compute homography matrix
    HMatrix, Status = FindHomography(Matches, Base_kp, Sec_kp)
    NewSize, Corr, HMatrix = GetFrameSizeAndTransformMatrix(HMatrix,
SecImg_Cyl.shape[:2], image1.shape[:2])
    #Transforming image2 to new frame
    SecImg_Tf = cv2.warpPerspective(SecImg_Cyl, HMatrix, (NewSize[1], NewSize[0]))
    SecImg_Mask_Tf = cv2.warpPerspective(SecImg_Mask, HMatrix, (NewSize[1],
NewSize[0]))
    #Preparing image1 for blending
    BaseImg_Tf = np.zeros((NewSize[0], NewSize[1], 3), dtype=np.uint8)
    BaseImg_Tf[Corr[1]:Corr[1]+image1.shape[0], Corr[0]:Corr[0]+image1.shape[1]] =
image1
    StitchedImg = cv2.bitwise_or(SecImg_Tf, cv2.bitwise_and(BaseImg_Tf,
cv2.bitwise_not(SecImg_Mask_Tf)))
    return StitchedImg
```

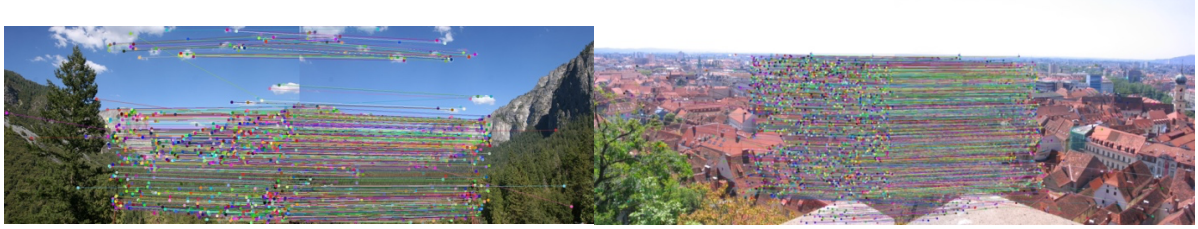
2.4 Result of Stitching Image

Using SIFT algorithm (input using all the images in folder):

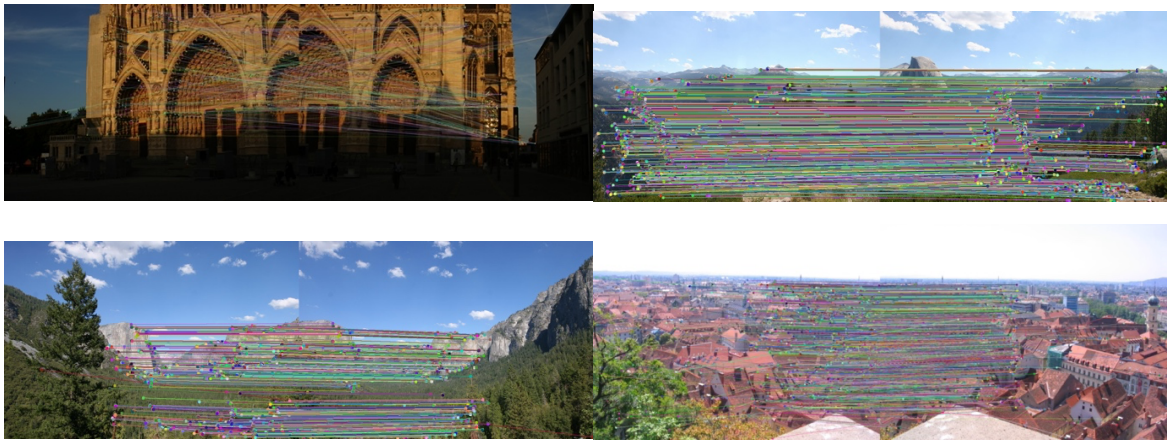
Using Local Window with Harris algorithm for feature capturing (input using all the images in folder):

2.5 Comparison of Different Feature Descriptors mat

Matching Result using SIFT Algorithm



Matching Result using Local Window with Harris algorithm for feature capturing



2.6 Comparison of Different Feature Descriptors

Comparison of Feature Descriptor

Type	Dataset	Outlier Ratio	Minimum Rasnac Iteration
SIFT	Data 1	0.7753	1805
		0.4643	54
		0.5447	105
		0.7527	1230
		0.4594	52
	Data 2	0.0417	3
		0.1944	9
		0.1919	9
	Data 3	0.0716	4
		0.0244	2
	Data 4	0.2414	12
		0.1955	9
		0.2787	15
Local Window	Data 1	0.6062	190
		0.6798	436
		0.2356	12
		0.4851	64
		0.4679	56
	Data 2	0.4015	34
		0.0694	4
		0.1591	7
	Data 3	0.1423	6
		0.1049	5
	Data 4	0.2752	15
		0.4655	55
		0.831	5638

Comparison of Feature Descriptor

Total Testing: 20

Type of Feature	Data	Average Outlier	Minimum Iteration
SIFT	Data 1	0.72	11
	Data 2	0.23	1
	Data 3	0.12	1
	Data 4	0.62	4
Concatenated pixel values	Data 1	0.65	2
	Data 2	0.21	1
	Data 3	0.07	1
	Data 4	0.55	2

The first table shows the results of the entire experiment presenting 1 iteration with an acceptance probability of 0.99 in the RANSAC algorithm and providing the overall experimental results in the form of a mosaic image. Then, the second table shows a comparison for an acceptance ratio of 0.5 in the RANSAC algorithm with 20 iterations (with selection of random images). SIFT produces more outliers and requires more iterations, while concatenated pixel values show fewer outliers and require fewer iterations, making it more efficient in most cases.

3. Discussion

In this experiment, the challenges in homography adjustment include high outlier ratio, feature distribution and perspective distortion with large changes in viewing angle to produce an image or mosaic image that is appropriate and can be viewed well. With the various viewing angles of multiple images, this experiment is difficult to produce photos that can be viewed/produce good photos. In my opinion, photos that have multiple photos that have features that have many matches in one photo with another photo, can produce better results, while photos with few matching relationships will produce photos that are less clear or have bad results.