비주얼오도메트리와증강현실 프로젝트 보고서

인공지능학과 120230472 박다빈

주제 : automatic stitching of two images

연구 목표 :

ORB와 RANSAC을 사용하여 2D homography computation을 진행시켜보는 것이 목표이다. 마지막으로 두 이미지를 올바르게 stitching 하는 것이 최종 목표이다.

알고리즘 :

1. ORB

우선 이미지 2개를 비슷한 부분끼리 겹치게 만들기 위해서는 이미지에서 특징이 되는 부분을 선정하여 비교를 해야한다. 이때 특징이 되는 지점을 key point라고 하고, 특징이 되는 부분에 대한 정보가 담긴 것을 discriptor라고 한다.

우리는 실험에서 ORB알고리즘을 사용할 것인데, 이것은 key point detector인 **FAST**와 descriptor를 계산하는 부분인 **BRIEF**를 사용하는 알고리즘 이며, 추가적으로 어떤 물체를 rotate하더라도 식별에 어려움을 겪지 않는 rotation invariant 기능과 이미지 데이터의 노이즈에 강한 resistant to noise 기능이 탑재되어있는 알고리즘이다.

1-1) FAST

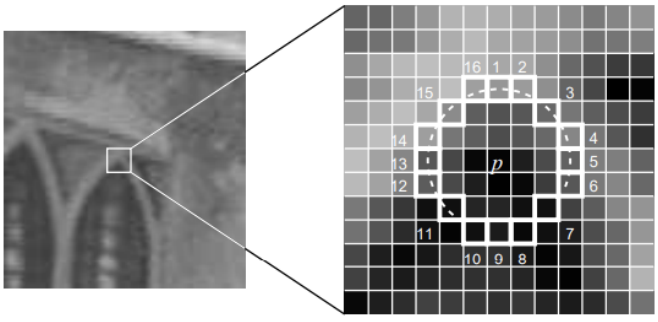


사진 출처 : Machine learning for high-speed corner detection (Edward Rosten and Tom Drummond)

p라는 픽셀에서 반지름이 약 3 정도되는 거리에 있는 픽셀들과의 밝기를 비교하여 밝기가 threshold 이상인 지점이 N개 이상일 경우 해당 부분을 key point 라고 선택한다.

find p′ such that $(I_p - I_{p'}) > \mathrm{T}$

if p′ are more than N, p is key point.

1-2) BRIEF

앞서 말한 바와 같이 key point 들의 binary 한 descriptor를 계산하는 부분이다. key point 주변의 여러 픽셀들의 묶음을 만들어서 0,1로 이루어진 descriptor를 만드는 방식이다. BRIEF를 사용하여 만든 descriptor는 binary로 이루어져 있기 때문에 Hamming distance를 사용하여 비교를 할 수 있다.

2. BFMatcher 로 point 이어주기 (with Hamming distance)

BFMatcher(Brute-Force Matcher)는 image_1 과 image_2의 key point 들의 discriptor들을 하나하나 확인해 매칭되는지 판단하는 알고리즘이다. 이때 비교 방식은 Hamming distance를 사용한다.

2-1) Hamming distance : 두개의 벡터가 서로 같아지기 위해 몇번 element 들을 바꿔야하는지 개수를 출력한다.

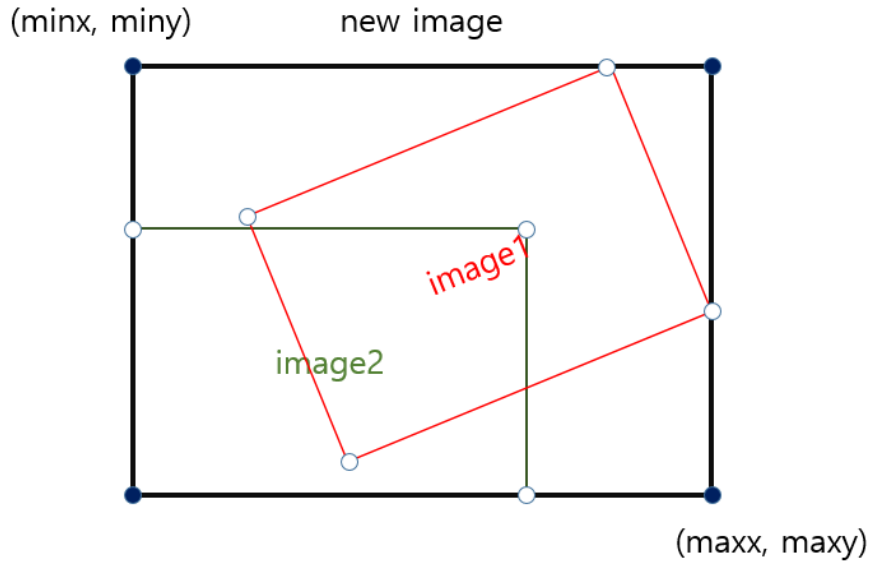
3. find Homography

2번에서 나온 image_1과 image_2의 key point matching 결과 쌍을 사용하여 Homography를 구한다. 이때 올바르지 않은 match가 있을 수 있기 때문에 RANSAC 알고리즘을 사용하여 오류를 무시하고 최적의 결과가 나오도록 homography를 계산할 수 있도록 한다. 나는 해당 코드의 RANSAC 알고리즘의 threshold 를 5 정도 주었다.

cv2 에서 제공하는 findHomography 함수를 활용하여 Homography matrix와 mask 를 구할 수 있다. 이때 mask의 역할은 matching 결과가 정확하지 않은 것들을 배제하는 역할을 해준다. 따라서 이전 2번에서 나온 matching 중 올바른 matching 만 남길 수 있도록 한다.

4. stitching two images (image1 and image2)

이후 3번에서 구한 homography matrix를 사용하여 imaeg 1를 이동시켜서 image 2에 이어붙일 것이다. homography에 image 1의 사진 제일 가장자리에 대한 좌표값을 준 후 위치 변형을 한다. (warpPerspective 함수 사용) 이후 new image의 왼쪽 아래 부분에 image2를 붙이면 image stitching을 완료할 수 있다. 이해를 위해 아래 그림 자료를 첨부한다.



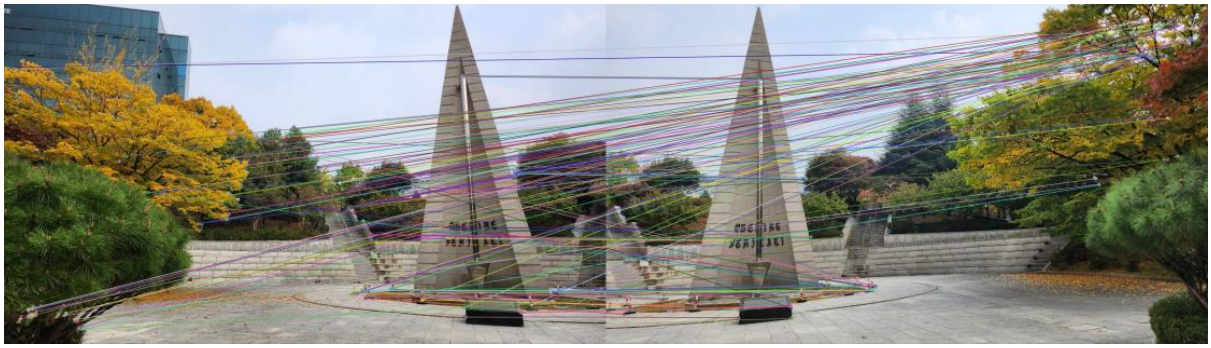
new image의 크기 = maxx-minx , maxy-miny

결과 :

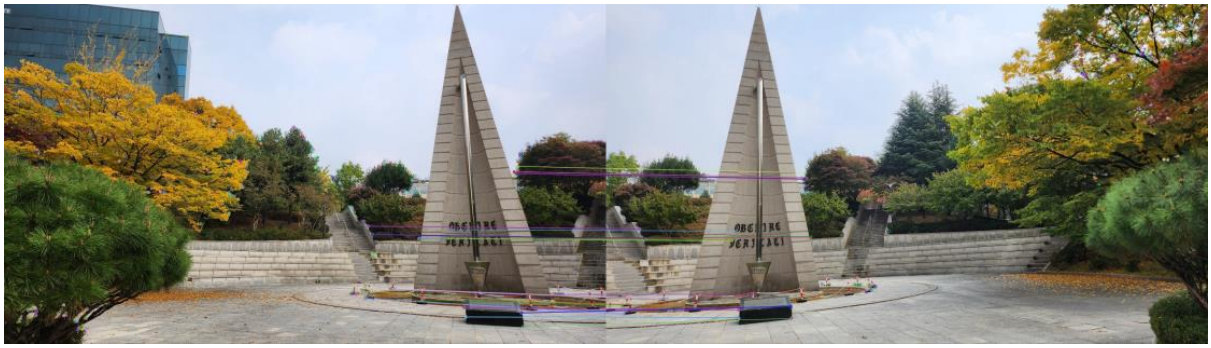
input 사진은 갤럭시Z 플립4로 촬영하였습니다.

1

image1 ,image2



matching result (no mask)



better matching result (mask o)

stitching result.

2.

image1 ,image2

matching result (no mask)

better matching result (mask o)

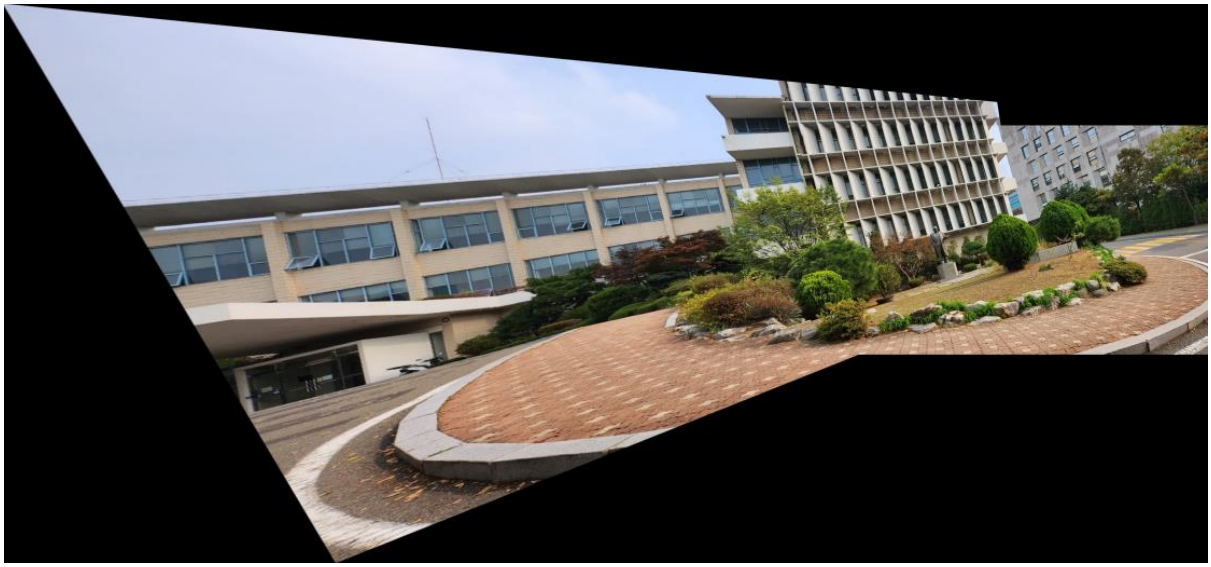
stitching result.

3.


image1 ,image2

matching result (no mask)

better matching result (mask o)



stitching result.

source code :

```
import cv2

import numpy as np


img_1 = cv2.imread('test6_1.jpg')

img_2 = cv2.imread('test6_2.jpg')

#이미지를 흑백으로 변환

gray_1 = cv2.cvtColor( img_1 , cv2.COLOR_BGR2GRAY )

gray_2 = cv2.cvtColor( img_2 , cv2.COLOR_BGR2GRAY )


#ORB 추출기 생성

orb = cv2.ORB_create()

# keypoint 와 descriptor 생성

keyp_1 , descrip_1 = orb.detectAndCompute(gray_1,None)

keyp_2 , descrip_2 = orb.detectAndCompute(gray_2,None)
```

```
#BFMatcher로 point 이어주기. with Hamming distance
bfmatcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
#knn match로 확실한 포인트들만 추출하도록 설계 :
# knn match 대신 homography의 mask 사용해보자 어차피 Ransac이기 때문에..!?
matching = bfmatcher.match(descrip_1,descrip_2)

# 두 matching distance 비율이 0.75이하인 것만 추출한다.
'''
ratio_test = 0.75

best_matches = []

for m1, m2 in matching:
    if m1.distance < 0.75 * m2.distance:
        best_matches.append([m1])

#for debugging (나중에 delete 할 것)
print('matches: %d %d'%(len(best_matches),len(matching)))
print(type(best_matches))
'''
start_points = np.float32([keyp_1[x.queryIdx].pt for x in matching])
end_points = np.float32([keyp_2[x.trainIdx].pt for x in matching])

#find homography using RANSAC with threshold : 5.0
# 5.0 은 RANSAC에서 사용될 threshold
```

```
matrixH, maskH = cv2. findHomography(start_points, end_points, cv2.RANSAC, 5.0)

H, W = img_1.shape[:2]

img_1_points = np.float32([ [[0,0]],[[0, H-1]],[[W-1, H-1]],[[W-1, 0]] ])


print('matrixH.shape: ', matrixH.shape) #3x3

print('start_points.shape: ', start_points.shape) #171x2

print('img_1_points.shape: ', img_1_points.shape)


dst_points = cv2.perspectiveTransform(img_1_points,matrixH)

#new_img2 = cv2.polylines(img_2, [np.int32(dst_points)], True, 255, 3, cv2.LINE_AA)


#maskH에서 남은 괜찮은 matching만 남긴다.

best_matching = maskH.ravel().tolist()




#draw result with best_matches

img_r= cv2.drawMatches(img_1, keyp_1, img_2, keyp_2, matching, None)

img_r2 = cv2.drawMatches(img_1, keyp_1, img_2, keyp_2, matching, None,
matchesMask=best_matching)


# warp two images to the panorama image using the homography matrix

#image 2를 이동시키고 image 1을 상대적으로 이동시키지 말아보자


H_2, W_2 = img_2.shape[:2]

H_1, W_1 = img_1.shape[:2]
```

```
edge_pts1 = np.float32([[0,0], [0,H_2],[W_2,H_2],[W_2,0]]).reshape(-1,1,2)

edge_pts2 = np.float32([[0,0], [0,H_1],[W_1,H_1],[W_1,0]]).reshape(-1,1,2)

edge_pts2_ = cv2.perspectiveTransform(edge_pts2, matrixH)

edge_pts = np.concatenate((edge_pts1, edge_pts2_), axis = 0)


#파노라마 된 그림의 크기를 찾기 위해

[min_x, min_y] = np.int32(edge_pts.min(axis=0).ravel()) #- 0.5

[max_x, max_y] = np.int32(edge_pts.max(axis=0).ravel()) # + 0.5

t_p = [-min_x, -min_y]


matrixT = np.array([[1,0,t_p[0]], [0,1,t_p[1]], [0,0,1]])

result3 = cv2.warpPerspective(img_1,matrixT.dot(matrixH), (max_x-min_x, max_y-min_y))

result3[t_p[1]:t_p[1]+H_2, t_p[0]:t_p[0]+W_2] = img_2


cv2.imwrite('result3.jpg', result3)


cv2.imwrite('result.jpg', img_r)

cv2.imwrite('result2.jpg', img_r2)

cv2.waitKey(0)

cv2.destroyAllWindows()
```