


**[문제 1. Feature Descriptor and Feature Matching]**

1-1. “paired image” 폴더의 임의의 페어 이미지에 대하여 OpenCV를 이용해 3가지 이상의 알고리즘 (e.g. SIFT, SURF, ORB 등)을 통해 Keypoint와 특징을 추출하여 보세요.

원본 Image Name	SIFT	BRIEF	ORB
annapurna_right_01	keypoint: 1887 descriptor: (1887, 128)	keypoint: 304 descriptor: (304, 32)	keypoint: 500 descriptor: (500, 32)
annapurna_left_01	keypoint: 1906 descriptor: (1906, 128)	keypoint: 275 descriptor: (275, 32)	keypoint: 500 descriptor: (500, 32)
bryce_right_01	keypoint: 5892 descriptor: (5892, 128)	keypoint: 1086 descriptor: (1086, 32)	keypoint: 500 descriptor: (500, 32)
bryce_left_01	keypoint: 6140 descriptor: (6140, 128)	keypoint: 1291 descriptor: (1291, 32)	keypoint: 500 descriptor: (500, 32)
grand_canyon_right_01	keypoint: 7474 descriptor: (7474, 128)	keypoint: 1320 descriptor: (1320, 32)	keypoint: 500 descriptor: (500, 32)
grand_canyon_left_01	keypoint: 8041 descriptor: (8041, 128)	keypoint: 1283 descriptor: (1283, 32)	keypoint: 500 descriptor: (500, 32)
sedona_right_01	keypoint: 1764 descriptor: (1764, 128)	keypoint: 82 descriptor: (82, 32)	keypoint: 500 descriptor: (500, 32)
sedona_left_01	keypoint: 2752 descriptor: (2752, 128)	keypoint: 122 descriptor: (122, 32)	keypoint: 500 descriptor: (500, 32)

1-2. 추출된 Keypoint를 이미지 상에 원과 같은 형태(크기는 자유롭게)로 표시하여 보고, 왜 해당 부분이 Keypoint로 추출되었을 지 알고리즘과 연관 지어 자유롭게 분석하여 보세요.



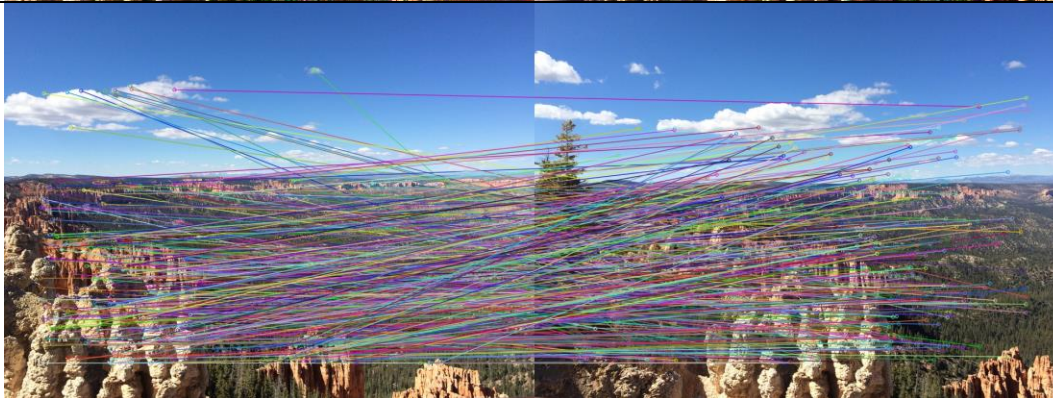
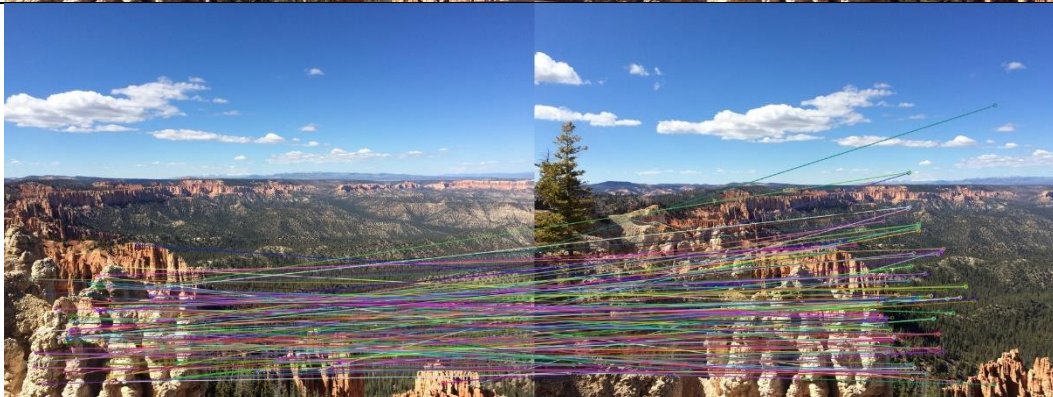
원본		bryce_left_01
----	--	---------------

SIFT		Keypoint (특징점)로 나온 값이 가장 많으며, 이미지에서 식별 가능한 물체 위주에 위치하여 점들이 산출되었습니다.
BRIEF		이미지 내에서 밝기가 변하는 위치 위주로 Keypoint (특징점)이 산출 되었으며, 주로 물체의 경계 위주로 위치해 있습니다.
ORB		코너 위주로 Keypoint (특징)이 발생 하였으며, Keypoint의 크기가 다른 알고리즘 방법에 비해 크게 나왔습니다.

1-3. 앞서 추출한 3가지 이상의 알고리즘을 통한 Keypoint와 특징을 이용하여 Feature Matching을 수행해보세요. 이 때, OpenCV 라이브러리를 활용해 Matching하고, 매칭 결과를 시각화해 나 타내어 보세요. 또한, 알고리즘 별 매칭 결과의 차이를 분석해보세요.



> Keypoint가 가장 많이 나온 SIFT 방법이 Matching 또한 가장 많이 이루어졌으며, 모든 공통된 특징이 거의 다 Matching된 것처럼 보입니다. 반면에 BRIEF의 경우는 SIFT보다 공통된 특징에 대해 덜 Matching이 이루어진 것처럼 보이며, ORB의 경우는 특정 영역이 전부 Matching 되지 않는 것을 볼 수 있었습니다.



원본	
SIFT	
BRIEF	
ORB	

## [문제 2. Homography and Stitching]

2-1. "paired image" 폴더의 임의의 페어 이미지에 대하여 Homography 행렬  $H$ 를 구해보세요. 이 때, OpenCV 함수를 사용하고, 알고리즘은 RANSAC이 아닌 SVD / Least-Square을 사용하며, 모든 매칭 페어들을 사용하여  $H$ 를 구해보세요.  $H$ 를 구하기 위한 매칭 페어들은 1번 문제에서 의 결과를 자유롭게 재사용하셔도 됩니다.

코드	<pre># homography 행렬 산출 src = np.float32([kp1[m.queryIdx].pt for m in matches]).reshape((-1, 1, 2)) dst = np.float32([kp2[m.trainIdx].pt for m in matches]).reshape((-1, 1, 2))  # RANSAC일 경우와 그 외의 경우이다. # METHOD방법을 주지 않으면 최소자승법(Least-Square)적용된다. if sti_gubun == 'RANSAC':     H1, status1 = cv2.findHomography(src, dst, cv2.RANSAC, 5.0) else:     H1, status1 = cv2.findHomography(src, dst, None)</pre>
homography행렬 (3*3 matrix)	<pre>[[-3.79089105e-01 -2.06750230e-01 4.32564480e+02]  [-1.69984367e-01 -6.45304236e-02 1.78581058e+02]  [-8.65573823e-04 -5.02447014e-04 1.00000000e+00]]</pre>

2-2. 앞서 구해진 Homography  $H$  행렬을 이용해 Image Stitching을 수행해 보세요. 이 때, cv2.warpPerspective(...) 함수를 사용하세요.

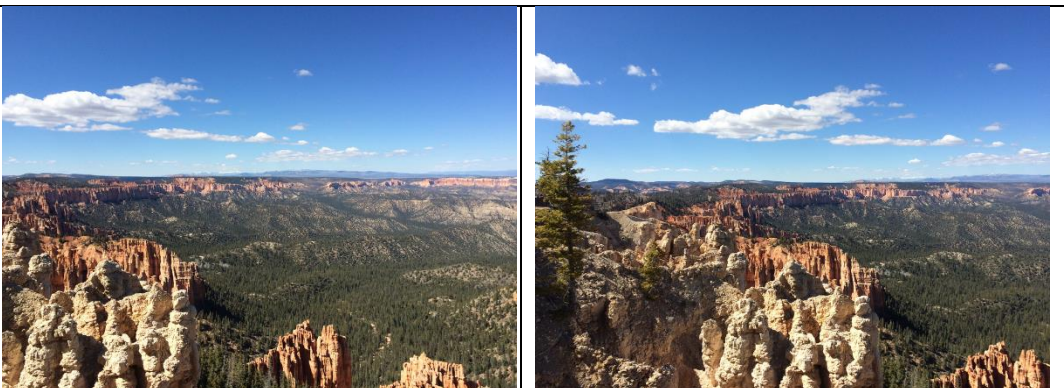
원본	
SIFT	


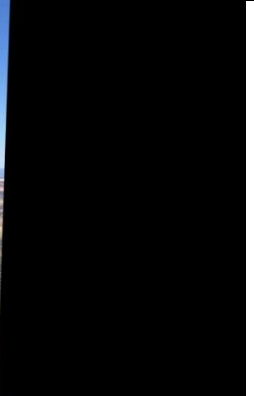

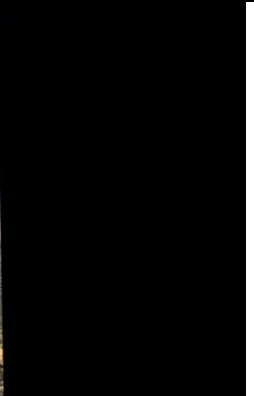

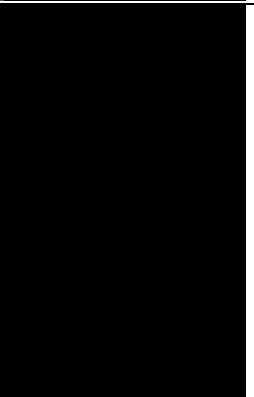


BRIEF	
ORB	

2-3. Homography 행렬  $H$ 를 모든 매칭 페어가 아닌 좋은 매칭 페어만 남도록 필터링 기준을 만들어 좋은 매칭 페어만을 사용해  $H$ 를 2-1과 동일하게 구해보세요. 또한, 마찬가지로 Image Stitching을 수행해보고, 2-1 결과와 어떤 점이 개선되었는지 분석해보세요.

> 모든 매칭 페어를 사용했을 경우 Image Stitching이 정상적으로 이루어지지 않았는데, 좋은 매칭 페어만 남도록 필터링 기준을 만들어 사용한 결과 Image Stitching이 잘 이루어진 것을 확인할 수 있었습니다. 또한 필터링 기준을 거리에 따라 상위 10~100개까지 진행해 보았는데, 너무 적은 수를 사용할 경우에는 Image Stitching이 제대로 이루어지지 않는 것을 확인할 수 있었습니다.






원본	
----	--

SIFT		
BRIEF		
ORB		





2-4. Homography 행렬  $H$ 를 RANSAC을 사용해 구해보세요. 이 때, 마찬가지로 OpenCV 함수를 사용하세요. 앞서와 마찬가지로, Image Stitching을 수행하고 해당 결과를 2-1, 2-3의 결과들과 비교 분석하여 보세요.

> RANSAC을 사용하지 않았을 경우에는 모든 매칭 페어를 사용할 때와 좋은 매칭 페어를 사용했을 때의 Image Stitching 결과에 매우 큰 차이가 발생하였으나, RANSAC을 이용할 경우 두 경우의 차이를 육안으로는 식별하기 어려웠습니다. 또한 Keypoint를 추출하는 방법에 따라 Image Stitching 결과가 조금씩 다르게 나오는 것을 확인할 수 있었습니다. ORB를 사용하였을 때 SIFT와 BRIEF 방법을 사용하였을 때보다 Image Stitching 시 두 이미지가 이어지는 부분에 있어 좀 더 부자연스럽게 이어진 것을 확인할 수 있었습니다.



2-1 모든 매칭점 사용		
원본	 	
SIFT		
BRIEF		
ORB		



2-3 좋은 매칭점 사용		
원본	 	
SIFT		
BRIEF		
ORB		



	모든 매칭점 사용	좋은 매칭점 사용	비고
SIFT			SIFT, BRIEF, ORB 알고리즘 사용에 따라 Image Stitching 결과가 조금씩 다르지만, 3가지 알고리즘 모두 모든 매칭점을 사용할 때 보다 좋은 매칭점을 사용했을 때 이미지간 연결이 좀더 자연스러워진것을 확인할 수 있습니다.
BRIEF			
ORB			

### [문제 3. Homography with RANSAC 직접 구현]

3-1. RANSAC 알고리즘을 통한 Homography 행렬  $H$ 를 구하는 함수를 직접 구현해 구해보세요. 이 때, cv2.findHomography의 SVD / Least-Square 메소드 옵션이나 Numpy와 같이 중간 결과를 편하게 계산해주는 Library 사용은 모두 가능합니다. 보고서에서는 반드시 중요 코드/알고리즘에 대하여 설명을 포함해주세요. 또한, 2번 문제와 같이 "paired images"의 임의의 페어 이미지에 대하여 Image Stitching을 수행하고 해당 결과를 2-4번 결과(OpenCV 사용)와 비교 해 직접 구현한 알고리즘이 올바르게 작동하는지 비교해보세요.

> Homography 행렬  $H$ 를 구하기 위해서는 이미지의 매칭되는 4개의 특징점을 random 하게 선택하여 행렬을 생성합니다. 4개의 특징점을 이용하여 Homography 행렬을 구한 후 RANSAC 알고리즘을 적용합니다. Matching 점으로 이루어진 행렬과 Homography 행렬 사이의 거리를 구해 가장 많은 inlier를 가진 Homography 행렬을 구하는 방식으로 구현하였습니다. matching point의 경우 기존에 사용하였던 함수를 사용하였으며 SIFT, BRIEF, ORB 알고리즘을 적용하였습니다. 해당 알고리즘 구현 후에 모든 매칭점과 좋은 매칭점을 사용하여 Image Stitching을 진행한 결과 SIFT와 BRIEF 알고리즘의 경우 OpenCV 사용과 비슷한 결과를 얻었지만 ORB의 경우는 Image Stitching이 조금 부자연스럽게 연결되었습니다.

> 중요 코드 및 알고리즘

중요 코드 / 알고리즘

```
def ransac(corr, thresh):
    maxInliers = []
    finalH = None
    for i in range(1000):
        #4개의 random point 추출
        corr1 = corr[random.randrange(0, len(corr))]
        corr2 = corr[random.randrange(0, len(corr))]
        # 행렬 생성
        randomFour = np.vstack((corr1, corr2))

        corr3 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr3))

        corr4 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr4))

        #Homography 행렬 생성
        h = calculateHomography(randomFour)
        inliers = []

        for i in range(len(corr)):
            d = geometricDistance(corr[i], h)
            # 차이값으로 정하기 나름이다, 여기서는 5를 사용
            if d < 5:
                inliers.append(corr[i])

        # inlier의 길이 > maxInliers 길이 이면 inlier의 길이를 max로 옮기고
        # 그때의 Homography 행렬을 넣어준다.
        if len(inliers) > len(maxInliers):
            print("len(maxInliers)", len(maxInliers))
            maxInliers = inliers
            finalH = h
        # threshold 값을 곱한값이 maxInliers의 길이보다 작으면 멈춤
        if len(maxInliers) > (len(corr)*thresh):
            break

    return finalH, maxInliers

# Homography matrix와 거리 구하기
def geometricDistance(correspondence, h):

    # h : 3*3 matrix , p1:3*1 matrix
    p1 = np.transpose(np.matrix([correspondence[0].item(0), correspondence[0].item(1), 1]))









    # estimatep2 : 3*1 matrix / h: Homography matrix
    estimatep2 = np.dot(h, p1)
    estimatep2 = (1/estimatep2.item(2))*estimatep2





    p2 = np.transpose(np.matrix([correspondence[0].item(2), correspondence[0].item(3), 1]))
    error = p2 - estimatep2

    # 벡터의 norm을 반환한다.
    return np.linalg.norm(error)
```



> 결과 이미지

my_ransac 2-1 모든 매칭점 사용		
원본		
SIFT		
BRIEF		
ORB		

my_ransac 2-3 좋은 매칭점 사용		
원본	 	
SIFT		
BRIEF		
ORB	