비주얼오도메트리와 증강현실

중간시험과제: automatic stitching of two images

120230191 이경희

1. Problem

이 프로젝트는 서강대학교의 두 개의 다른 뷰를 하나의 파노라마 이미지로 결합할 수 있는 알고리즘을 개발하는 것을 목표로 합니다. 이를 달성하기 위한 주요 도전 과제는 다음과 같습니다:

1. keypoint matching : 두 이미지 각각에서 keypoint를 뽑아 동일한 위치로 추정되는 keypoint간의 matching을 수행합니다. 이 정보는 두 이미지의 공간적 관계를 결정하는 데 사용됩니다.
2. Handling Outliers : 잘못된 matching을 stitching 과정에서 이상한 결과를 만들 수 있으므로 이러한 outlier를 식별하고 제거하는 것이 중요합니다. 이 프로젝트에서는 Lowe's ratio test와 ransac을 이용해 이상치를 제거합니다.
3. homography 계산: 올바른 match가 식별되면 이미지를 다른 이미지에 매핑할 수 있는 변환(이 경우 homography matrix)을 계산합니다.
4. Image warping: 계산된 homography matrix를 사용하여 이미지 중 하나를 다른 이미지와 정렬하도록 변환합니다.
5. stitching: 정렬 후 두 이미지를 부드럽게 블렌딩하여 일관된 파노라마를 만듭니다.

2. Algorithm Description:

ORB Keypoint Detection and Descriptor Computation:

ORB는 Oriented FAST and Rotated BRIEF의 약자로, 키 포인트 검출과 descriptor 계산을 위한 효율적인 알고리즘입니다. 이 프로젝트에서는 Opencv에 내장된 ORB를 사용하여 두 이미지에서 키 포인트와 descriptors를 계산합니다.

Bruteforce Matching with Hamming Distance:

두 이미지의 descriptors 간의 matching을 찾기 위해 Hamming distance를 사용한 Bruteforce matching 알고리즘을 적용합니다. Hamming distance는 두 descriptor 간의 비트 차이를 계산하여 matching의 정확성을 평가합니다. 이 프로젝트에서는 opencv를 통해 이 과정을 수행한 후 Lowe's ratio test를 사용하여 좋은 matching만을 선택합니다. 이를 통해 잘못된 matching을 필터링합니다.

RANSAC for Homography Matrix Computation:

RANSAC (Random Sample Consensus)은 잡음이 있는 데이터에서 robust한 모델을 추정하기 위한 반복적인 방법입니다. 이 프로젝트에서는 두 이미지 간의 homography matrix를 계산하기 위해 RANSAC을 사용합니다. 여러 번의 반복을 통해 데이터 샘플을 선택하고, 각 샘플에 대해 homography matrix를 계산합니다. 이후, 계산된 matrix를 사용하여 전체 데이터 세트에 대한 오차를 계산하고, 오차가 특정 threshold(outlier ratio)보다 작은 데이터 포인트를 inliers로 간주합니다. 최종적으로 가장 많은 inliers를 가진 모델이 선택됩니다. 이 프로젝트에서는 RANSAC의 반복 횟수(iteration)와 threshold(outlier ratio)을 실험적으로 결정하기 위해 여러 매개변수를 사용합니다. 이를 통해 최적의 결과를 얻을 수 있습니다.

Exception Handling:

알고리즘의 각 단계에서 예외 처리가 수행됩니다. 예를 들어, 이미지 파일이 존재하지 않거나, matching된 키 포인트의 수가 충분하지 않은 경우 해당 이미지는 처리되지 않습니다.

3. Source Code:

1. Choose two images

```
img1_path = f'data/{image_name}_1.jpg'
img2_path = f'data/{image_name}_2.jpg'

# 이미지 파일이 존재하는지 확인합니다.
if not os.path.exists(img1_path) or not os.path.exists(img2_path):
    print(f"Images {img1_path} or {img2_path} not found. Skipping...")
    return

# 이미지 파일이 존재하면 읽어옵니다.
img1 = cv2.imread(img1_path, cv2.IMREAD_COLOR)
img2 = cv2.imread(img2_path, cv2.IMREAD_COLOR)
```

한 쌍의 이미지를 불러옵니다. 만약 둘 중 하나라도 이미지가 없을 경우 에러 메시지를 출력한 후 종료합니다.

2. compute ORB keypoint and descriptors (opencv)

```
orb = cv2.ORB_create()
kp1, des1 = orb.detectAndCompute(img1, None) # keypoint(위치)와 descripto
#print(kp1, des1)
kp2, des2 = orb.detectAndCompute(img2, None)
img_keypoints1 = cv2.drawKeypoints(img1, kp1, None, color=(0, 255, 0))
img_keypoints2 = cv2.drawKeypoints(img2, kp2, None, color=(0, 255, 0))
```

Opencv의 cv2.ORB_create()를 사용하여 ORB 객체를 생성하고, 이 객체를 사용하여 이미지에서 특징점(keypoints)과 해당 특징점의 특징 벡터(descriptors)를 추출하는 과정을 설명하고 있습니다. kp1/des1와 kp2/des2는 각각 두 이미지에서 추출된 특징점과 기술자를 저장하는 변수들입니다.

3. apply Bruteforce matching with Hamming distance (opencv)

```
bf = cv2.BFMatcher(cv2.NORM_HAMMING) # B
matches = bf.knnMatch(des1, des2, k=2)
```

이미지 간의 keypoint matching을 수행하기 위해 OpenCV의 Brute-Force matching 알고리즘을 사용합니다. cv2.BFMatcher 객체를 생성하여 bf 변수에 저장합니다. 이 때, cv2.NORM_HAMMING을 사용하여 ORB 특징 벡터 간의 hamming distance를 계산하는 방식으로 matching을 수행합니다.

bf.knnMatch 메서드를 사용하여 첫 번째 이미지의 각 특징 벡터(descriptor)와 두 번째 이미지의 특징 벡터 간의 거리를 계산하고, 가장 가까운 두 개의 matching 결과를 반환합니다. 이 결과는 matches 변수에 저장됩니다

```
good_matches = [m for m, n in matches if m.distance < 0.75 * n.distance]
img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None)
```

matching 결과 중에서 좋은 matching만을 선택하기 위해 Lowe's ratio test를 적용합니다. 이 테스트는 matching 간의 거리 비율을 사용하여 가장 가까운 matching과 두 번째로 가까운 matching 간의 거리가 특정 임계값보다 큰 경우 해당 matching을 좋은 matching으로 간주합니다. 이 방식을 통해 잘못된 matching을 걸러낼 수 있습니다.

```
if len(good_matches) < 4:
    print(f"Not enough matches for image {image_name}. Skipping...")
    return
```

RANSAC 알고리즘에서 homography를 추정하기 위해 무작위로 선택된 4개의 좌표를 사용합니다. matches가 4개 미만인 경우 두 이미지 간에 충분한 대응점이 없다는 것을 의미하며 4개의 좌표를 사용하여 Homography를 추정하는 과정에서 충분한 match가 없는 경우, 즉 좋은 match의 수가 4개 미만인 경우 homography 계산이 불가능하거나 매우 부정확합니다. 때문에 이전 과정에서 찾은 good_matches가 4개 미만인 경우 "Not enough matches for image"를 출력한 후 종료합니다.

4. implement RANSAC algorithm to compute the homography matrix. (DIY)

```
src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches])
dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches])
```

Src_pts : 첫 번째 이미지에서 선택된 keypoints 좌표 목록, dst_pts : 두 번째 이미지에서 선택된 keypoints 좌표 목록, 각 좌표들은 orb 알고리즘을 통해 검출된 특징점 중 서로 이미지의 특징점과 matching된 것들의 좌표이다. 이 두 좌표 목록은 이미지 간의 관계를 나타내는 homography 행렬을 계산하는데 사용됩니다.

```
outlier_ratio = np.arange(0.5, 10.0, 0.5) #outlier리스트

data = np.hstack((src_pts, dst_pts)) #매칭 리스트
max_iterations = calculate_ransac_iterations()
M, best_threshold = find_best_threshold(image_name, data, outlier_ratio, max_iterations)
```

```
inliers_idx = count_inliers(M, src_pts, dst_pts, best_threshold)
inlier_matches = [good_matches[i] for i in inliers_idx]
img_inlier_matches = cv2.drawMatches(img1, kp1, img2, kp2, inlier_matches, None)
```

outlier_ratio를 0.5에서 10.0까지 0.5 간격으로 설정하였습니다. 이 값은 실제로 이상치의 비율을 나타내는 것이 아니라, 두 이미지 간의 변환된 포인트와 실제 포인트 간의 거리를 계산할 때 사용되는 threshold 배열로 사용됩니다. 이전 RANSAC을 코딩 할 때 사용했던 변수명을 그대로 가져왔으나 outlier_ratio는 비율을 나타내므로 쓰임이 다릅니다. calculate_RANSAC_iterations 함수를 사용하여 RANSAC 알고리즘의 반복 횟수를 계산합니다. find_best_threshold 함수를 사용하여 주어진 임계값 배열 내에서 최적의 임계값을 찾습니다. 각 임계값에 대해 RANSAC 알고리즘을 적용하여 homography 행렬을 계산하고, 해당 임계값에서의 내부 데이터 수를 계산합니다. 가장 많은 내부 데이터를 가진 임계값이 최적의 임계값으로 선택됩니다. 최적의 threshold를 이용해 기존 matching 포인트 목록에서 불필요한 matching포인트를 제거합니다. 이 과정에서 사용된 함수들에 대해 설명하겠습니다. 모든 함수는 RANSAC_homography파일에 선언되어 있습니다.

```
def calculate_ransac_iterations(p=0.99, sample_size=2, outlier_ratio=0.5):
    """RANSAC iteration 계산"""
    # 내부 데이터 비율을 기반으로 필요한 반복 횟수 계산
    value = 1 - (1 - outlier_ratio)**sample_size
    if isinstance(value, complex) or value <= 0:
        return float('inf')  # 무한대 반환
    return math.ceil(math.log(1 - p) / math.log(value))
```

교재 (4.18), Table 4.3에 나온 RANSAC 실행 횟수 계산식(N = log(1 - p) / log(1 - (1 - ε)^s))을 이용해서 RANSAC 결과에 영향을 주는 매개변수인 iteration횟수를 계산합니다.

```python
def find_best_threshold(image_name, data, thresholds, max_iterations=100):
    best_threshold = None
    best_model = None
    max_inliers_count = 0

    # inlier 개수를 저장할 리스트 초기화
    inliers_counts = []

    for threshold in thresholds:
        model = ransac_homography(data, max_iterations, threshold)
        src_pts = data[:, :2]
        dst_pts = data[:, 2:]
        inliers_idx = count_inliers(model, src_pts, dst_pts, threshold)

        # 각 threshold에 대한 inlier 개수 저장
        inliers_counts.append(len(inliers_idx))

        if len(inliers_idx) > max_inliers_count:
            max_inliers_count = len(inliers_idx)
            best_model = model
            best_threshold = threshold

    return best_model, best_threshold
```

앞서 설명한 threshold값을 기반으로 가장 많은 inliers_count를 가지는 threshold값을 찾는 함수입니다. 각 threshold값을 이용해서 RANSAC 알고리즘을 실행해 inliers의 수를 계산하고 가장 많은 inliers를 가진 threshold값을 best_threshold로 선택합니다.

```python
def ransac_homography(data, max_iterations=100, threshold=1.0):
    # 최적의 모델 초기화
    best_model = None
    # 최대 내부 데이터 수 초기화
    max_inliers = 0
    # 전체 데이터 수
    total_data = len(data)

    # RANSAC 반복
    for i in range(max_iterations):
        # 데이터 집합에서 무작위로 4개의 데이터 샘플 선택
        subset = data[np.random.choice(total_data, 4, replace=False)]
        src_pts = subset[:, :2]
        dst_pts = subset[:, 2:]

        # 선택한 샘플로 homography 계산
        H = compute_homography(src_pts, dst_pts)

        # 모델 오차 계산
        inliers_idx = count_inliers(H, src_pts, dst_pts, threshold)

        # 내부 데이터 수가 이전의 최대값보다 큰 경우 모델 업데이트
        if len(inliers_idx) > max_inliers:
            best_model = H
            max_inliers = len(inliers_idx)

    return best_model
```

3번 예외처리에서도 설명했듯 homography 계산을 위해 4개의 좌표를 사용합니다. Calculate_RANSAC_iteration함수에서 계산된 값을 기반으로 RANSAC을 반복합니다. 한 번의 RANSAC과정에서 무작위 4개의 데이터 샘플 선택, 선택한 샘플로 homography 계산, 계산된

Homography 행렬을 사용하여 원본 포인트를 변환하고, 변환된 포인트와 목표 포인트 간의 거리를 계산하여 오차 계산, inliers_idx가 큰 모델로 업데이트 과정이 반복 됩니다. 최종적으로 최적의 Homography 행렬을 반환합니다

```
def compute_homography(src_pts, dst_pts):
    # DLT를 사용하여 homography 행렬 계산
    A = []
    for i in range(0, src_pts.shape[0]):
        x, y = src_pts[i]
        u, v = dst_pts[i]
        A.append([-x, -y, -1, 0, 0, 0, u*x, u*y, u])
        A.append([0, 0, 0, -x, -y, -1, v*x, v*y, v])
    A = np.asarray(A)
    U, S, Vh = np.linalg.svd(A)
    L = Vh[-1,:] / Vh[-1,-1]
    H = L.reshape(3, 3)
    return H
```

compute_homography 함수는 두 이미지 간의 관계를 나타내는 homography 행렬을 계산하는 함수입니다. 이 함수는 Direct Linear Transformation (DLT) 방법을 사용하여 homography 행렬을 추정합니다. 각 특징점 쌍에 대해 선형 방정식을 구성하는 행(A)을 생성합니다. 이 방정식은 원본 이미지의 특징점과 대상 이미지의 특징점 간의 관계를 나타냅니다. 행렬 A에 대한 Singular Value Decomposition을 계산하여 U(직교 행렬), S(대각 행렬), Vh(직교행렬)의 서로 다른 행렬로 분해합니다. vh의 마지막 행을 재구성하여 3x3 homography행렬 H를 얻습니다.

```
def count_inliers(H, src_pts, dst_pts, threshold):
    transformed_pts = apply_homography(H, src_pts)
    distances = np.linalg.norm(transformed_pts - dst_pts, axis=1)
    return np.where(distances < threshold)[0]
```

Count_inliers는. apply_homography 함수를 사용하여 주어진 Homography 행렬로 원본 포인트를 변환합니다. 변환된 포인트와 목표 포인트 간의 유클리드 거리를 계산합니다. 내부 데이터 결정: 계산된 거리가 주어진 임계값보다 작은 경우 해당 포인트는 내부 데이터로 간주됩니다. 이러한 내부 데이터의 수를 반환합니다.

```
def apply_homography(H, src_pts):
    # Homography 행렬을 사용하여 포인트 변환
    src_pts_homogeneous = np.column_stack([src_pts, np.ones(src_pts.shape[0])])
    transformed_pts = np.dot(H, src_pts_homogeneous.T).T
    transformed_pts = transformed_pts[:, :2] / transformed_pts[:, 2:]
    return transformed_pts
```

apply_homography는 주어진 homography 행렬을 사용하여 포인트들을 변환합니다. 입력으로 주어진 포인트들을 Homogeneous Coordinates로 변환합니다. 이는 z 좌표에 1을 추가하여 3xN 형태의 행렬로 만드는 과정입니다. Homogeneous coordinate와 homography 행렬을 곱하여 변환된 point를 얻습니다. 변환된 포인트의 z 좌표로 나누어 x, y 좌표를 정규화합니다.

5. prepare a panorama image of larger size (DIY)

```
panorama = warpImages(img2, img1, M)
```

Warpimage 함수에서 두 이미지 간의 homography 관계를 사용해 두 이미지를 하나의 이미지로 결합해 결과 이미지를 생성합니다.

아래는 warpImages.py에 있는 내용입니다.

```
output_shape = (y_max - y_min, x_max - x_min, 3)
output_img = np.zeros(output_shape, dtype=img2.dtype)
```

output_shape는 최종 파노라마 이미지의 크기를 나타내는 튜플입니다. 이 변수는 두 이미지를 결합할 때 필요한 전체 출력 이미지의 크기를 정의합니다. 코드에서는 두 이미지의 모서리 좌표를 사용하여 변환된 이미지의 최소 및 최대 x, y 좌표 값을 계산하고, 이를 기반으로 output_shape의 크기를 결정합니다.

output_img는 최종 파노라마 이미지를 저장하는 배열입니다. 초기에는 모든 픽셀 값이 0인 검은색 이미지로 설정됩니다. 이는 np.zeros 함수를 사용하여 output_shape 크기의 검은색 이미지로 초기화됩니다. output_img의 크기와 형태는 output_shape에 정의된 대로입니다.

6. warp two images to the panorama image using the homography matrix (DIY)

```
def warpImages(img1, img2, H):
    # 이미지의 행과 열 크기를 가져옵니다.
    rows1, cols1 = img1.shape[:2]
    rows2, cols2 = img2.shape[:2]

    # 각 이미지의 모서리 좌표를 정의합니다.
    list_of_points_1 = np.float32([[0, 0], [0, rows1], [cols1, rows1], [cols1, 0]])
    temp_points = np.float32([[0, 0], [0, rows2], [cols2, rows2], [cols2, 0]])

    # Homography를 사용하여 img2의 모서리 좌표를 변환합니다.
    list_of_points_2 = apply_homography(H, temp_points)

    # 두 이미지의 모서리 좌표를 결합합니다.
    list_of_points = np.concatenate((list_of_points_1, list_of_points_2), axis=0)

    # 변환된 좌표의 최소 및 최대 값을 계산합니다.
    [x_min, y_min] = np.int32(list_of_points.min(axis=0).ravel() - 0.5)
    [x_max, y_max] = np.int32(list_of_points.max(axis=0).ravel() + 0.5)

    # 출력 이미지 내에서 원본 이미지의 시작 위치를 계산합니다.
    translation_dist = [-x_min, -y_min]

    # 이동 행렬을 생성합니다.
    H_translation = np.array([[1, 0, translation_dist[0]], [0, 1, translation_dist[1]], [0, 0, 1]])
    H = np.dot(H_translation, H)
```

두 입력 이미지 img1과 img2의 행과 열의 크기를 각각 rows1, cols1 및 rows2, cols2로 추출합니

다. 이를 기반으로 각 이미지의 모서리 좌표를 정의하며, img1의 모서리 좌표는 list_of_points_1에, img2의 모서리 좌표는 temp_points에 저장됩니다. 다음으로, apply_homography 함수를 사용하여 img2의 모서리 좌표를 변환하고, 이 변환된 좌표는 list_of_points_2에 저장됩니다. 두 이미지의 모서리 좌표를 결합하여 list_of_points를 생성하고, 이 좌표들의 최소 및 최대 x, y 값을 계산합니다. 이를 통해 출력 이미지 내에서 원본 img1의 시작 위치를 계산하고, 해당 위치를 기반으로 이동 행렬 H_translation을 생성합니다. 마지막으로, 이 이동 행렬을 기존의 Homography 행렬 H와 결합하여 업데이트합니다. 이 정보는 두 이미지를 stitching하는 데 사용됩니다.

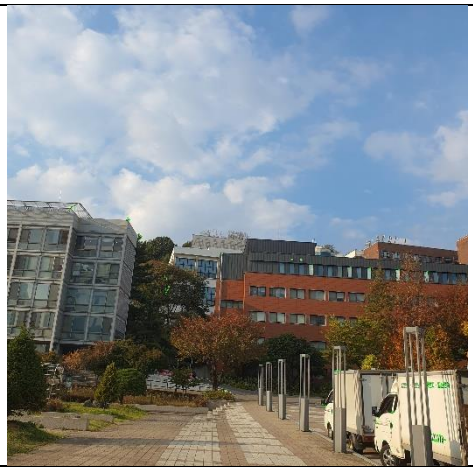
```
# 출력 이미지의 모든 좌표에 대한 그리드를 생성합니다.
y, x = np.indices(output_shape[:2])
homogeneous_coords = np.stack((x.ravel(), y.ravel(), np.ones(y.size)))

# 역 Homography 행렬을 적용하여 원본 좌표를 계산합니다.
transformed_coords = np.dot(np.linalg.inv(H), homogeneous_coords)
transformed_coords /= transformed_coords[2]

x_src, y_src = transformed_coords[0], transformed_coords[1]

# 변환된 좌표가 img2 내에 있는지 확인하는 마스크를 생성합니다.
mask = (0 <= x_src) & (x_src < cols2) & (0 <= y_src) & (y_src < rows2)

# 추가적으로, 변환된 좌표가 유효한지 (NaN 또는 무한대 값이 아닌지) 확인하는 마스크를 생성합니다.
valid_mask = ~np.isnan(x_src) & ~np.isnan(y_src) & ~np.isinf(x_src) & ~np.isinf(y_src)

# 두 마스크를 결합합니다.
final_mask = mask & valid_mask

# 마스크를 사용하여 유효한 좌표만 정수로 변환합니다.
x_src = x_src[final_mask].astype(int)
y_src = y_src[final_mask].astype(int)

# 마스크를 사용하여 유효한 좌표만 선택합니다.
valid_x = x.ravel()[final_mask]
valid_y = y.ravel()[final_mask]

# img2에서 유효한 좌표를 출력 이미지로 매핑합니다.
output_img[valid_y, valid_x] = img2[y_src, x_src]


# img1을 출력 이미지에 오버레이합니다.
output_img[translation_dist[1]:rows1 + translation_dist[1], translation_dist[0]:cols1 + translation_dist[0]] = img1

return output_img
```

이 코드는 두 이미지를 stitching하기 위한 과정을 나타냅니다. 먼저, 주어진 좌표에 역 Homography 행렬을 적용하여 원본 이미지의 좌표로 변환합니다. 이를 통해 두 번째 이미지의 픽셀이 첫 번째 이미지에 어떻게 매핑되는지 확인할 수 있습니다. 다음으로, 변환된 좌표가 두 번째 이미지 내에 존재하는지, 그리고 유효한 값인지 확인하기 위한 마스크를 생성합니다. 이 마스크를 사용하여 유효한 좌표만 선택하고, 이 좌표에 해당하는 픽셀 값을 출력 이미지에 매핑합니다. 마지막으로, 첫 번째 이미지를 출력 이미지 위에 오버레이하여 두 이미지가 서로 겹치는 영역에서 stitching을 완료합니다. 이렇게 하여 두 이미지는 하나의 연속된 이미지로 합쳐집니다.

4. Experimental Results:

두 이미지는 ORB를 사용하여 키 포인트와 descriptors를 계산한 후, Bruteforce matching을 사용하여 matching되는 키 포인트를 찾습니다. 이후, RANSAC 알고리즘을 사용하여 outliers를 제거하고 inliers만을 사용하여 homography matrix를 계산합니다. 결과적으로, 두 이미지는 계산된 homography matrix를 사용하여 왜곡되고 합쳐져 하나의 파노라마 이미지를 형성합니다.

실험에는 9가지 종류의 이미지가 사용되었으나, 가장 결과가 잘 나온 2, 3번 이미지만 보고서에 첨부하겠습니다. 다른 결과들은 result 폴더에서 확인하실 수 있습니다. [https://github.com/eolgaeng-i/Visual-Odometry]

<table>
<tr><td colspan="2">1. Choose two images</td></tr>
<tr><td> 
image1</td><td> 
image2</td></tr>
<tr><td colspan="2">2. compute ORB keypoint and descriptors (opencv)</td></tr>
<tr><td>  </td><td>  </td></tr>
<tr><td colspan="2">3. apply Bruteforce matching with Hamming distance (opencv)</td></tr>
</table>

4. implement RANSAC algorithm to compute the homography matrix. (DIY)

5. prepare a panorama image of larger size (DIY)

6. warp two images to the panorama image using the homography matrix (DIY)

4. take another set of two views in Sogang University

1. Choose two images

image1

image2

2. compute ORB keypoint and descriptors (opencv)

3. apply Bruteforce matching with Hamming distance (opencv)

4. implement RANSAC algorithm to compute the homography matrix. (DIY)

5. prepare a panorama image of larger size (DIY)

6. warp two images to the panorama image using the homography matrix (DIY)

Keypoint 값

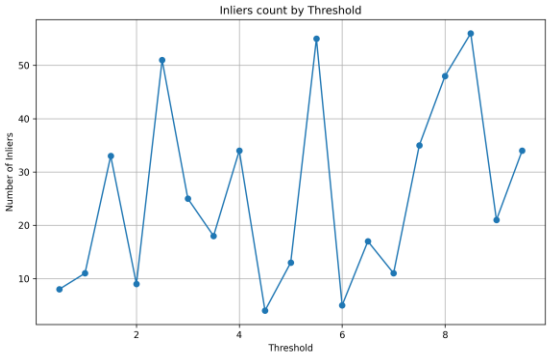


Figure 1. threshold에 따른 keypoint matching개수

Figure 2. Keypoint matching이 잘못된 결과

Threshold값에 따라 keypoint의 개수가 많이 차이나는 것을 확인할 수 있습니다. Lowe's ratio test 혹은 RANSAC 등 matching된 keypoint결과를 필터링 하지 않는 경우 잘못 matching된 keypoint 값으로 인해 이미지가 warp과정에서 아예 접혀버리는 현상을 관찰할 수 있었습니다.
이를 통해 정확한 파노라마 이미지를 output으로 얻기 위해 keypoint matching 필터링이 중요하다는 것을 알 수 있었습니다.

대칭 이미지

알바탑 이미지는 거의 대칭을 이루고 있습니다. 그렇기 때문에 알바탑 이미지를 겹치는 부분이 있도록 반으로 나눠 2D homography computation with ORB and RANSAC을 진행했을 때 실제 이미지에선 왼쪽과 오른쪽인 이미지가 같은 point로 matching되는 것을 확인할 수 있었습니다.

ORB의 경우 descriptor를 사용해 keypoint 주변의 정보를 기반으로 데이터를 저장하므로 회전에 불변인 특징을 가집니다. 그렇기 때문에 완벽하게 대칭인 사물에 대해 좌우가 반전된 사물로 인식이 된 것 입니다.

이러한 과정에 따라 img2를 img1의 좌우반전 인 것처럼 인식했기 때문에 output도 그냥 img2가 그대로 나왔습니다. 이러한 실험 결과를 통해 다음과 같은 결론을 얻을 수 있었습니다. ORB는 회전과 크기에 불변인 특징을 가지므로, 대칭적인 이미지에서는 좌우 반전된 이미지와 원본 이미지의 특징점이 유사하게 인식될 수 있습니다. 따라서, 완벽하게 대칭인 이미지에서 ORB를 사용하여 특징점을 추출하면, 좌우 반전된 이미지와 원본 이미지의 특징점이 matching될 가능성이 높습니다. 이러한 특성 때문에, 대칭적인 이미지를 두 부분으로 나누어 2D homography computation을 진행할 경우, 각 부분의 이미지가 서로 같은 point로 matching될 수 있습니다. 결과적으로, img2가 img1의 좌우 반전된 것처럼 인식되어 output 이미지에서 img2가 그대로 나타나게 됩니다. 이러한 현상은 ORB의 특징과 대칭적인 이미지의 특성 때문에 발생하는 것으로, 이를 고려하여 이미지 stitching이나 다른 작업을 진행할 때 주의가 필요합니다.