[중간시험과제] Automatic stitching of two images

120230238 인공지능학과 오지현

---

1. Take two views in Sogang University

Image_l

Image_r

```
# !pip install opencv-python
import numpy as np
import cv2
from tqdm import tqdm

image_r = cv2.imread('project1_image_1.jpg')  # 오른쪽 사진
image_l = cv2.imread('project1_image_2.jpg')  # 왼쪽 사진

gray_r = cv2.cvtColor(image_r, cv2.COLOR_BGR2GRAY)
gray_l = cv2.cvtColor(image_l, cv2.COLOR_BGR2GRAY)
```

2. Develop a ORB + Ransac + homography algorithm to create a panorama image from the two inputs.

2-1. Compute ORB keypoint and descriptors (opencv)

opencv의 ORB_create함수를 통해서 왼쪽과 오른쪽 이미지에 대해서 각각 keypoint와 descriptor를 계산합니다.

```
# create ORB
orb = cv2.ORB_create()

# calculate the keypoints, descriptors
keypoint_l, descriptor_l = orb.detectAndCompute(gray_l, None)
keypoint_r, descriptor_r = orb.detectAndCompute(gray_r, None)
```

2-2. Apply Bruteforce matching with Hamming distance (opencv)

Hamming distance를 이용하여 Bruteforce matching를 진행하는 코드로, 불필요한 특징점들을 제거하여 좋은 매칭점들만 남기기 위해 거리가 가까운 점들 중 상위 100개를 선정합니다.

```
# knnMatch using BF-Hamming
bfmatcher = cv2.BFMatcher(cv2.NORM_HAMMING)
matches = bfmatcher.match(descriptor_l, descriptor_r)

# sort the result of matching and save good matching
sorted_matches = sorted(matches, key=lambda x: x.distance)
good_matches = sorted_matches[:100]
```

2-3. Implement RANSAC algorithm to compute the homography matrix. (DIY)

주어진 포인트인 `src_pts`와 `dst_pts`에서 각각 무작위로 4개의 포인트를 선택하여 반환하는 함수입니다.

```
def choice_random(src_pts, dst_pts):
    random_indices = np.random.choice(len(src_pts), 4)
    return np.array([src_pts[i] for i in random_indices]),
           np.array([dst_pts[i] for i in random_indices])
```

다음은 RANSAC을 이용하여 가장 많은 inlier를 가지는 Homography matrix를 찾는 함수입니다. `src_pts`와 `dst_pts`는 전체 포인트들이고, `good_src_pts`와 `good_dst_pts`는 outlier제외한 좋은 포인트들입니다. 또한, `threshold`는 inlier로 취급할 에러의 임계값이고, `iter_limit`는 최대 반복 횟수를 의미합니다.

좀 더 구체적으로 설명하면,
반복횟수가 iter_limit보다 작거나 최대 inlier수가 threshold에 도달할 때까지 다음 과정들을 반복합니다. 우선 랜덤으로 포인트를 선택하고, 선택된 포인트로부터 Homography matrix를 계산합니다. 이후 모든 포인트들을 Homography matrix를 사용하여 입력 포인트를 변환한 후, `e = u - x_hat \ d = np.linalg.norm(e)` 를 통해 에러를 계산합니다. 에러가 일정 임계값 이내인 경우 해당 포인트를 `inlier`로 간주하고, 현재까지의 최대 `inlier(=best_inlier)`보다 많은 경우 업데이트합니다.

```
def find_homography_ransac(src_pts, dst_pts, good_src_pts, good_dst_pts,
                           threshold=100, iter_limit=2000):
    iter = 0
    best_inlier = 0
    best_H = 0

    while best_inlier < threshold and iter < iter_limit:
        iter += 1
        src, dst = choice_random(good_src_pts, good_dst_pts)
        H = calculate_homography(src, dst)

        inlier = 0
        for j in range(len(src_pts)):
            x = np.transpose(
                np.matrix([src_pts[j][0], src_pts[j][1], 1]))
            u = np.transpose(
                np.matrix([dst_pts[j][0], dst_pts[j][1], 1]))

            # x_hat is estimation result.
            x_hat = np.dot(H, x)
            x_hat = (1/x_hat.item(2))*x_hat

            e = u - x_hat
            d = np.linalg.norm(e)

            if d < 5:
                inlier += 1

            if best_inlier < inlier:
                best_inlier = inlier
                best_H = H

    return best_H
```

아래는 주어진 포인트들로부터 Homography matrix를 계산하는 함수입니다.

```
def calculate_homography(src_points, dst_points):
    A = []
    for i in range(len(src_points)):
        x, y = src_points[i][0], src_points[i][1]
        u, v = dst_points[i][0], dst_points[i][1]
        A.append([x, y, 1, 0, 0, 0, -x*u, -u*y, -u])
        A.append([0, 0, 0, x, y, 1, -v*x, -v*y, -v])

    A = np.array(A)
    _, _, vt = np.linalg.svd(A)

    H = np.reshape(vt[-1], (3, 3))
    H = (1 / H.item(8)) * H
    return H
```

아래는 매칭된 특징점들의 좌표를 추출하고 배열의 형태를 조정하는 과정입니다.

`matches`는 특징점들 간의 매칭 결과이고 `m.queryIdx`와 `m.trainIdx`는 각각 쿼리 이미지와 학습 이미지의 특징점 인덱스를 나타냅니다. 이를 이용하여 `keypoint_l`과 `keypoint_r`에서 해당 특징점의 좌표를 추출하고, 이를 `np.float32` 타입으로 설정해줍니다. 최종적으로 `reshape((-1, 2))`를 통해 2차원 배열로 변환합니다.

```
dst_pts = np.float32(
    [keypoint_l[m.queryIdx].pt for m in matches]).reshape((-1, 2))
src_pts = np.float32(
    [keypoint_r[m.trainIdx].pt for m in matches]).reshape((-1, 2))

good_dst_pts = np.float32(
    [keypoint_l[m.queryIdx].pt for m in good_matches]).reshape((-1, 2))
good_src_pts = np.float32(
    [keypoint_r[m.trainIdx].pt for m in good_matches]).reshape((-1, 2))
```

마지막으로 RANSAC을 사용하여 두 이미지 간의 Homography matrix를 구한 결과입니다.

```
H = find_homography_ransac(src_pts, dst_pts,
                           good_src_pts, good_dst_pts)
print(H)
'''
[[ 3.68876671e-01 -1.55114181e-02  2.86938101e+03]
 [-2.49824500e-01  7.89024455e-01  5.38519684e+02]
 [-1.05721442e-04 -8.23995500e-06  1.00000000e+00]]
'''
```

3. Apply the algorithm to get a result.

최종적으로 파노라마 이미지를 생성하는 단계입니다.

```
def create_panorama(image_l, image_r, H):

    # warping
    src_locs = []
    for x in tqdm(range(image_r.shape[1])):
        for y in range(image_l.shape[0]):
            loc = [x, y, 1]
            src_locs.append(loc)
    src_locs = np.array(src_locs).transpose()

    dst_locs = np.matmul(H, src_locs)
    dst_locs = dst_locs / dst_locs[2, :]
    dst_locs = dst_locs[:2, :]
    src_locs = src_locs[:2, :]
    dst_locs = np.round(dst_locs, 0).astype(int)

    height, width, _ = image_l.shape

    # prepare a panorama image
    result = np.zeros((height, width * 2, 3), dtype=int)
    for src, dst in tqdm(zip(src_locs.transpose(), dst_locs.transpose())):
        if dst[0] < 0 or dst[1] < 0 or dst[0] >= width*2 or dst[1] >= height:
            continue
        result[dst[1], dst[0]] = image_r[src[1], src[0]]
    result[0: height, 0: width] = image_l

    return result
```

과정을 하나씩 설명하면,

1) Warping 과정:

   · 너비와 높이에 대해 반복하며 현재 좌표 `[x, y]`를 `[x, y, 1]`로 변환하여 `loc`에 저장하고 `src_locs` 리스트에 추가합니다.

2) 좌표 변환 및 정규화, 좌표 정수형 변환::

   · `src_locs = np.array(src_locs).transpose()`: `src_locs` 리스트를 numpy 배열로 변환하고, 행과 열을 뒤집어서 저장합니다.

   · `dst_locs = np.matmul(H, src_locs)`: Homography matrix `H`를 사용하여 `src_locs`를 변환한 좌표를 계산합니다.

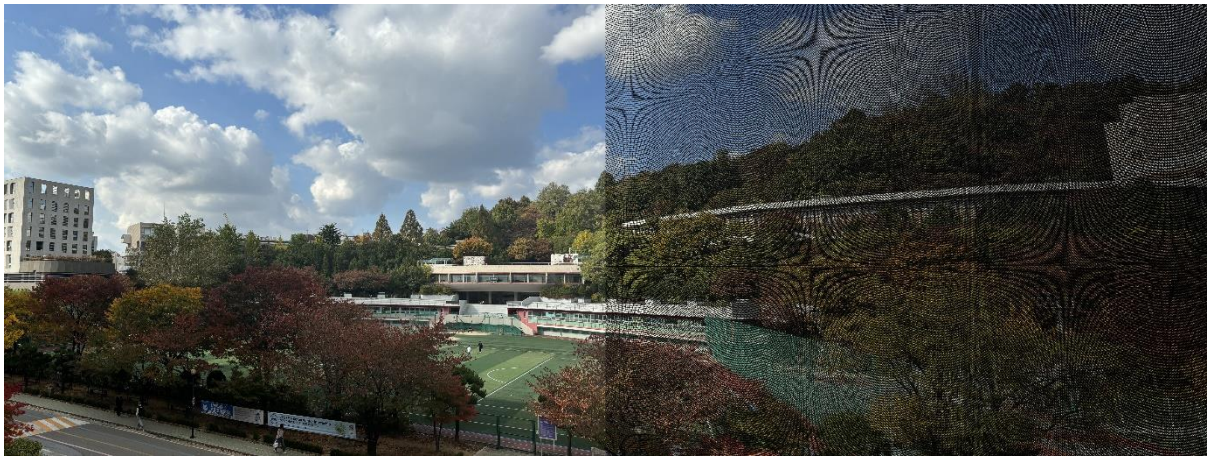
   · `dst_locs = dst_locs / dst_locs[2, :] \ dst_locs = dst_locs[:2, :]`: 변환된 좌표를 정규화하고 최종적으로 x, y좌표만 저장합니다. 추가적으로 반올림하여 정수 형태로 변환하는 과정도 진행합니다.

3) 파노라마 이미지 초기화:

`image_l`의 높이와 너비를 가져와서 파노라마 이미지를 초기화합니다. 이 때, 너비는 `image_l`의 2배로 설정되며, 높이는 `image_l`과 동일합니다.

4) 이미지 합성 및 파노라마 이미지 생성:
변환된 좌표 쌍을 하나씩 순회하면서 `if dst[0] < 0 or dst[1] < 0 or dst[0] >= width*2 or dst[1] >= height:` 를 통해 변환된 좌표가 파노라마 이미지의 범위를 벗어나면 건너뜁니다. `result[dst[1], dst[0]] = image_r[src[1], src[0]]`에서 변환된 좌표에 해당하는 `image_r`의 픽셀 값을 파노라마 이미지에 저장합니다. 마지막으로 `result[0: height, 0: width] = image_l`: 파노라마 이미지의 왼쪽 부분에 `image_l`을 덮어씌우고 파노라마 이미지를 반환합니다.

```
panorama_result = create_panorama(image_l, image_r, H) # forward maping
cv2.imwrite('panorama_forward.png', panorama_result)
```



(추가) Backward mapping and Interpolation

위의 결과를 보면, 검정색 빗살이 쳐지는 현상이 발생하는데 이는 이미지가 변환되면서 이전 point들이 모든 pixel영역을 채워주지 못하기 때문입니다. 이에 Backward mapping과 Bilinear interpolation을 적용하여 pixel 영역을 채워 더 나은 결과를 도출하고자 합니다. Backward mapping은 출력 이미지의 각 픽셀을 입력 이미지로 역 변환해서 입력 이미지에서 어떤 위치에 해당하는지 계산해준 다음, 해당 위치의 값을 가져와서 출력 이미지의 픽셀에 할당하는 방식입니다. 따라서 빈 공간을 채워주는 역할을 합니다.
(Forward mapping에 Interpolation만 구현할 경우, 여전히 자연스럽게 채워주지 못했기 때문에 Backward Mapping을 사용하였습니다.)

(추가-1) Backward mapping without Bilinear Interpolation

`backward_without_interpolation`는 역방향 변환을 수행하여 이미지를 반환하는 함수로, 보간 없이 픽셀 값을 복사합니다. `Image`는 변환할 입력 이미지, `H`는 Homography `matrix` `output_shape`는 결과 이미지의 크기를 나타내는 튜플입니다. 핵심은 `H_inv = np.linalg.inv(H)`로, 변환 행렬 H의 역행렬을 계산하는 부분입니다.

```
def backward_without_interpolation(image, H, output_shape):
    height, width = image.shape[:2]
    warped_image = np.zeros(
        (output_shape[1], output_shape[0], image.shape[2]), dtype=image.dtype)

    H_inv = np.linalg.inv(H)

    for y_out in range(output_shape[1]):
        for x_out in range(output_shape[0]):
            point = np.dot(H_inv, np.array([x_out, y_out, 1]))
            point = point / point[2]

            x_in, y_in = int(point[0]), int(point[1])

            if 0 <= x_in < width and 0 <= y_in < height:
                warped_image[y_out, x_out] = image[y_in, x_in]

    return warped_image
```

```
result_bw_wo_ip = backward_without_interpolation(image_r, H,
                                                 (image_r.shape[1] + image_l.shape[1],
                                                  image_r.shape[0]))
result_bw_wo_ip[0: image_r.shape[0], 0: image_l.shape[1]] = image_l
cv2.imwrite('result_backward_wo_interpolation.png', result_bw_wo_ip)
```

(추가-2) Backward mapping with Bilinear Interpolation

본 과정은 Backward mapping에 Bilinear Interpolation까지 적용하면 더 나은 결과가 나오는지 확인하기 위해 추가 수행하였으나, 출력 결과를 보면 큰 차이가 없었습니다. 따라서 코드에 대한 구체적인 설명은 생략하였으며, 코드는 아래와 같습니다. 또한 OpenCV로 간단하고 빠르게 구현하는 방법도 하단에 주석으로 추가하였습니다.

```
def warp_perspective_with_interpolation(image, H, output_shape):

    height, width = image.shape[:2]
    warped_image = np.zeros((output_shape[1], output_shape[0], image.shape[2]),
                            dtype=image.dtype)

    H_inv = np.linalg.inv(H)

    for y_out in range(output_shape[1]):
        for x_out in range(output_shape[0]):
            # transform the output coordinates to input coordinates
            # using an inverse transformation
            point = np.dot(H_inv, np.array([x_out, y_out, 1]))
            point = point / point[2]

            x_in, y_in = int(point[0]), int(point[1])

            # checks if it is within the input image boundaries
            if 0 <= x_in < width-1 and 0 <= y_in < height-1:
                # bilinear interpolation
                dx, dy = point[0] - x_in, point[1] - y_in
                for channel in range(image.shape[2]):
                    if 0 <= x_in+1 < width and 0 <= y_in+1 < height:
                        warped_image[y_out, x_out, channel] = (
                            (1 - dx) * (1 - dy) * image[y_in, x_in, channel] +
                            dx * (1 - dy) * image[y_in, x_in + 1, channel] +
                            (1 - dx) * dy * image[y_in + 1, x_in, channel] +
                            dx * dy * image[y_in + 1, x_in + 1, channel]
                        )

    return warped_image

result_usingdiy = warp_perspective_with_interpolation(image_r, H,
                                                      (image_r.shape[1] + image_l.shape[1],
                                                       image_r.shape[0]))
result_usingdiy[0 : image_r.shape[0], 0 : image_l.shape[1]] = image_l
cv2.imwrite('result_usingdiy.png', result_usingdiy)

# (참고) 위의 코드를 opencv 라이브러리를 이용해서 더 빠르게 구현하는 방법
'''result_usingcv = cv2.warpPerspective(image_r, H, (image_r.shape[1] + image_l.shape[1],
                                      panorama_result = create_panorama(image_l, image_r,
cv2.imwrite('panorama_forward.png', panorama_result)image_r.shape[0]))
result_usingcv[0 : image_r.shape[0], 0 : image_l.shape[1]] = image_l
cv2.imwrite('result_usingcv.png', result_usingcv)'''
```

결과 이미지는 아래와 같습니다.

[Figure 1. result_backward_w/o_interpolation.png]

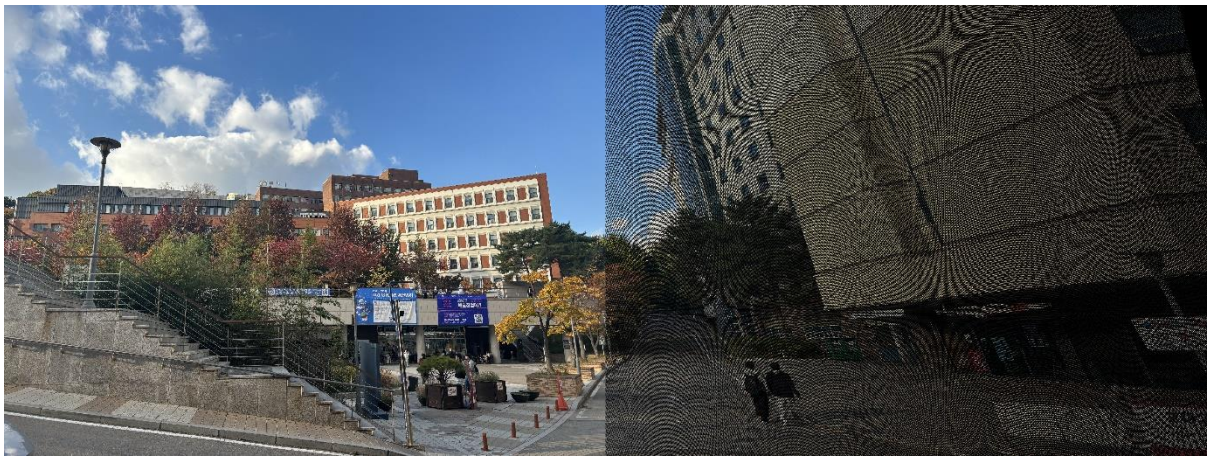
[Figure 2. result_backward_w/_interpolation.png]

4. Take another set of two views in Sogang University

Image_l

Image_r

5. Produce output



[Figure 3. result_forward.png]

[Figure 4. result_backward_w/o_interpolation.png]

[Figure 5. result_backward_w/_interpolation.png]

최종 소스 코드는 아래와 같습니다.

실행 예시:

```
python midterm_automatic_stitching_of_two_images.py
--input_left_dir=image_left.jpg –input_right_dir= image_right.jpg
--output_dir= output_images/
```

```
# midterm_automatic stitching of two images.py

import numpy as np
import cv2
from tqdm import tqdm
import argparse


def choice_random(src_pts, dst_pts):
    random_indices = np.random.choice(len(src_pts), 4)
    return np.array([src_pts[i] for i in random_indices]), np.array([dst_pts[i]
for i in random_indices])


def find_homography_ransac(src_pts, dst_pts, good_src_pts, good_dst_pts,
                           threshold=100, iter_limit=2000):

    iter = 0
    best_inlier = 0
    best_H = 0

    while best_inlier < threshold and iter < iter_limit:
        iter += 1
        src, dst = choice_random(good_src_pts, good_dst_pts)
        H = calculate_homography(src, dst)

        inlier = 0
        for j in range(len(src_pts)):
            x = np.transpose(
                np.matrix([src_pts[j][0], src_pts[j][1], 1]))
            u = np.transpose(
                np.matrix([dst_pts[j][0], dst_pts[j][1], 1]))

            # x_hat is estimation result.
            x_hat = np.dot(H, x)
            x_hat = (1/x_hat.item(2))*x_hat

            e = u - x_hat
            d = np.linalg.norm(e)

            if d < 5:
```

```
                inlier += 1

            if best_inlier < inlier:
                best_inlier = inlier
                best_H = H

    return best_H


def calculate_homography(src_points, dst_points):
    A = []
    for i in range(len(src_points)):
        x, y = src_points[i][0], src_points[i][1]
        u, v = dst_points[i][0], dst_points[i][1]
        A.append([x, y, 1, 0, 0, 0, -x*u, -u*y, -u])
        A.append([0, 0, 0, x, y, 1, -v*x, -v*y, -v])

    A = np.array(A)
    _, _, vt = np.linalg.svd(A)

    H = np.reshape(vt[-1], (3, 3))
    H = (1 / H.item(8)) * H
    return H


def create_panorama(image_l, image_r, H):

    # warping
    src_locs = []
    for x in tqdm(range(image_r.shape[1])):
        for y in range(image_l.shape[0]):
            loc = [x, y, 1]
            src_locs.append(loc)
    src_locs = np.array(src_locs).transpose()

    dst_locs = np.matmul(H, src_locs)
    dst_locs = dst_locs / dst_locs[2, :]
    dst_locs = dst_locs[:2, :]
    src_locs = src_locs[:2, :]
    dst_locs = np.round(dst_locs, 0).astype(int)

    height, width, _ = image_l.shape

    # prepare a panorama image
    result = np.zeros((height, width * 2, 3), dtype=int)
    for src, dst in tqdm(zip(src_locs.transpose(), dst_locs.transpose())):
        if dst[0] < 0 or dst[1] < 0 or dst[0] >= width*2 or dst[1] >= height:
            continue
```

```
        result[dst[1], dst[0]] = image_r[src[1], src[0]]
    result[0: height, 0: width] = image_l

    print("============finished============")

    return result


def backward_without_interpolation(image, H, output_shape):
    height, width = image.shape[:2]
    warped_image = np.zeros(
        (output_shape[1], output_shape[0], image.shape[2]), dtype=image.dtype)

    H_inv = np.linalg.inv(H)

    for y_out in tqdm(range(output_shape[1])):
        for x_out in range(output_shape[0]):
            point = np.dot(H_inv, np.array([x_out, y_out, 1]))
            point = point / point[2]

            x_in, y_in = int(point[0]), int(point[1])

            if 0 <= x_in < width and 0 <= y_in < height:
                warped_image[y_out, x_out] = image[y_in, x_in]

    print("============finished============")

    return warped_image


def backward_with_interpolation(image, H, output_shape):

    height, width = image.shape[:2]
    warped_image = np.zeros(
        (output_shape[1], output_shape[0], image.shape[2]), dtype=image.dtype)

    H_inv = np.linalg.inv(H)

    for y_out in tqdm(range(output_shape[1])):
        for x_out in range(output_shape[0]):
            # transform the output coordinates to input coordinates using an
inverse transformation
            point = np.dot(H_inv, np.array([x_out, y_out, 1]))
            point = point / point[2]

            x_in, y_in = int(point[0]), int(point[1])

            # checks if it is within the input image boundaries
```

```
            if 0 <= x_in < width-1 and 0 <= y_in < height-1:
                # bilinear interpolation
                dx, dy = point[0] - x_in, point[1] - y_in
                for channel in range(image.shape[2]):
                    if 0 <= x_in+1 < width and 0 <= y_in+1 < height:
                        warped_image[y_out, x_out, channel] = (
                            (1 - dx) * (1 - dy) * image[y_in, x_in, channel] +
                            dx * (1 - dy) * image[y_in, x_in + 1, channel] +
                            (1 - dx) * dy * image[y_in + 1, x_in, channel] +
                            dx * dy * image[y_in + 1, x_in + 1, channel]
                        )
    print("============finished============")

    return warped_image


def parse_args():
    parser = argparse.ArgumentParser(description="Panorama Creation Script")
    parser.add_argument("--input_left_dir", type=str,
                        required=True, help="Path to the left input image")
    parser.add_argument("--input_right_dir", type=str,
                        required=True, help="Path to the right input image")
    parser.add_argument("--output_dir", type=str, default="./",
                        help="Output directory for saving images")
    return parser.parse_args()


if __name__ == "__main__":

    args = parse_args()

    input_right_dir = args.input_right_dir
    input_left_dir = args.input_left_dir
    output_dir = args.output_dir

    # images load
    image_r = cv2.imread(input_right_dir)  # 오른쪽 사진
    image_l = cv2.imread(input_left_dir)  # 왼쪽 사진

    gray_r = cv2.cvtColor(image_r, cv2.COLOR_BGR2GRAY)
    gray_l = cv2.cvtColor(image_l, cv2.COLOR_BGR2GRAY)

    # create ORB
    orb = cv2.ORB_create()

    # calculate the keypoints, descriptors
    keypoint_l, descriptor_l = orb.detectAndCompute(gray_l, None)
    keypoint_r, descriptor_r = orb.detectAndCompute(gray_r, None)
```

```
    # knnMatch using BF-Hamming
    bfmatcher = cv2.BFMatcher(cv2.NORM_HAMMING)
    matches = bfmatcher.match(descriptor_l, descriptor_r)

    # sort the result of matching and save good matching
    sorted_matches = sorted(matches, key=lambda x: x.distance)
    good_matches = sorted_matches[:100]

    dst_pts = np.float32(
        [keypoint_l[m.queryIdx].pt for m in matches]).reshape((-1, 2))
    src_pts = np.float32(
        [keypoint_r[m.trainIdx].pt for m in matches]).reshape((-1, 2))

    good_dst_pts = np.float32(
        [keypoint_l[m.queryIdx].pt for m in good_matches]).reshape((-1, 2))
    good_src_pts = np.float32(
        [keypoint_r[m.trainIdx].pt for m in good_matches]).reshape((-1, 2))

    # compute homography matrix
    H = find_homography_ransac(src_pts, dst_pts,
                               good_src_pts, good_dst_pts)

    # result1: forward mapping
    panorama_result = create_panorama(image_l, image_r, H)
    cv2.imwrite(f'{output_dir}result_forward.png', panorama_result)

    # result2: backward mapping (without bilinear interpolation)
    result_bw_wo_ip = backward_without_interpolation(
        image_r, H, (image_r.shape[1] + image_l.shape[1], image_r.shape[0]))
    result_bw_wo_ip[0: image_r.shape[0], 0: image_l.shape[1]] = image_l
    cv2.imwrite(
        f'{output_dir}result_backward_wo_interpolation.png', result_bw_wo_ip)

    # result3: backward mapping with bilinear interpolation
    result_bw_w_ip = backward_with_interpolation(
        image_r, H, (image_r.shape[1] + image_l.shape[1], image_r.shape[0]))
    result_bw_w_ip[0: image_r.shape[0], 0: image_l.shape[1]] = image_l
    cv2.imwrite(
        f'{output_dir}result_backward_w_interpolation.png', result_bw_w_ip)

    # (참고) 위의 코드를 opencv 라이브러리를 이용해서 구현하는 방법
    '''result_usingcv = cv2.warpPerspective(image_r, H, (image_r.shape[1] +
image_l.shape[1], image_r.shape[0]))
    result_usingcv[0 : image_r.shape[0], 0 : image_l.shape[1]] = image_l
    cv2.imwrite('result_usingcv.png', result_usingcv)'''
```