비주얼오도메트리와증강현실

2023 Second Semester AIE6660-01
Project #01

1. 프로젝트 개요

1-1) 과제 설명

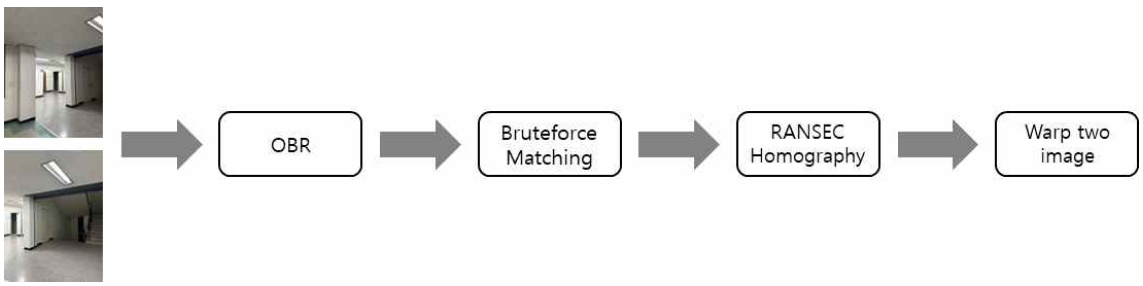
1. Choose two images
2. compute ORB keypoint and descriptors (opencv)
3. apply Bruteforce matching with Hamming distance (opencv)
4. implement RANSAC algorithm to compute the homography matrix. (DIY)
5. prepare a panorama image of larger size (DIY)
6. warp two images to the panorama image using the homography matrix (DIY)

Submit:

1. source code (python or C/C++) and the two source images and the result image
2. a report describing your work.

1-2) 세부 구현 계획

이번 프로젝트는 2장의 이미지를 입력받아 이를 파노라마 이미지로 변환해주는 것을 목표로 한다.



[1] 프로젝트 흐름

과제의 구현은 [1]과 같은 순서대로 진행된다. 이를 위해서 4개의 대응점이 필요하다. 대응점을 구하는 방법은 ORB(Oriented FAST and Rotated BRIEF)와 Bruteforce 알고리즘을 사용하여 구해준다. OpenCV 라이브러리를 활용하면 쉽게 ORB와 Bruteforce Matcher를 사용할 수 있다. orb를 적용하여 keypoint와 descriptor를 구하고, Bruteforce Matcher를 적용하여 대응점을 구해준다. 이 때 거리는 Hamming distance로 계산해준다. ORB와 Bruteforce 알고리즘으로 구해진 대응점들 중 적절한 4개를 선별하여 homography matrix를 생성하는 작업이 필요하다. 그 과정은 RANSAC을 활용하여 수행해준다.Homography matrix를 구했다면 이를 사용하여 forward 혹은 backward로 mapping시켜 두 이미지를 합친 파노라마를 생성하게 된다.

2. 프로젝트 구현

2-1) 입력 이미지

[2] 입력 이미지, 총 두 장의 이미지를 사용하였다. 크기는 300*400으로 동일하다.

파노라마를 생성할 때 사용 될 두 이미지는 위 사진을 적용하였다. 사진의 크기는 300 * 400으로 동일하다. 위 이미지 중 오른쪽 이미지를 homography matrix를 사용하여 mapping 시켜주어 파노라마 이미지를 제작하였다.

2-2) 코드 설명

구현을 위해 사용한 언어는 Python을 사용하였다.

```
import numpy as np
import cv2
import sys
```

사용한 주요 라이브러리는 OpenCV와 numpy이다.

1. Choose two images

```
img_left_path = sys.argv[1]
img_right_path = sys.argv[2]
img_left = cv2.imread(img_left_path)
img_right = cv2.imread(img_right_path)
img_left_gray = cv2.cvtColor(img_left,cv2.COLOR_BGR2GRAY)
img_right_gray = cv2.cvtColor(img_right,cv2.COLOR_BGR2GRAY)
```

두 개의 이미지를 argv로 path정보를 받아와 불러오는 코드이다. 이 때 RGB 이미지를 흑백으로 변경한 결과를 생성하는 이유는 ORB의 input을 생성하기 위함이다.

2. compute ORB keypoint and descriptors (opencv)

```
orb = cv2.ORB_create()
key_point_left, descriptors_left = orb.detectAndCompute(img_left_gray, None)
key_point_right, descriptors_right = orb.detectAndCompute(img_right_gray, None)
```

OpenCV 라이브러리에 이미 구현되어있는 ORB를 활용하여 keypoint와 descriptor를 생성하는 코드이다.

3. apply Bruteforce matching with Hamming distance (opencv)

```
matcher = cv2.BFMatcher_create(cv2.NORM_HAMMING)
matches = matcher.match(descriptors_left, descriptors_right)
matches = sorted(matches, key =lambda x: x.distance)
good_matches = matches[:100]
dst_key_points = np.float32([key_point_left[match.queryIdx].pt for match in
matches]).reshape((-1, 2))
src_key_points = np.float32([key_point_right[match.trainIdx].pt for match in
matches]).reshape((-1, 2))
good_dst_key_points = np.float32([key_point_left[match.queryIdx].pt for match in
good_matches]).reshape((-1, 2))
good_src_key_points = np.float32([key_point_right[match.trainIdx].pt for match in
good_matches]).reshape((-1, 2))
```

OpenCV 라이브러리에 있는 BFMatcher를 활용하여 BruteForece로 각 점을 매칭시켜주는 코드이다. 이 때 사용한 Distance는 Hamming distance를 사용하였다. 매칭된 결과 중 가장 유사도가 높은 100개를 따로 선별하였다. 그 이유는 추후에 RANSAC으로 random하게 4개씩 점을 sampling하여 mapping을 할 때 유사한 100가지에 대해서만 sampling을 하면 좀 더 좋은 결과를 얻을 수 있기 때문이다.(경험적인 판단, 전체에 대하여 sampling한 경우보다 유사도가 높은 100개에 대하여 sampling한 후 시도했을 때 좀 더 좋은 파노라마 이미지를 얻을 수 있었음)

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

```
def ransac_for_homography(src_key_points, dst_key_points, good_src_key_points,
good_dst_key_points, threshold =150, iter_limit =4000):
    max_inlier_cnt =0
    max_H =0
    iter_cnt =0

    while max_inlier_cnt < threshold and iter_cnt < iter_limit:
        iter_cnt +=1
        src, dest = generate_random_set(good_src_key_points, good_dst_key_points)
        H = cal_H(src, dest)

        inlier_cnt =0
        for j in range(len(src_key_points)):
            d = dist(src_key_points[j], dst_key_points[j], H)
            if d <5:
                inlier_cnt +=1

            if max_inlier_cnt < inlier_cnt :
                max_inlier_cnt = inlier_cnt
                max_H = H
    return max_H
```

RANSAC 알고리즘을 활용하여 Homography matrix를 찾는 함수이다.
4개의 match된 점을 random하게 sampling하여 Homography matrix를 생성하고 그 matrix를 활용하여 다른 점들을 mapping하였을 때 mapping된 점과 실제 목적된 위치의 distance가 5가 넘지 않는 데이터를 inlier로 판단하도록 하여 최적의 matrix를 찾도록 하였다. Random한 Sampling은 최대 4000번 수행하도록 하고, 150개 이상의 inlier가 존재하면 해당 H를 사용할 수 있도록 default로 정의하였다.

```
def generate_random_set(src_key_points, dst_key_points):
    r = np.random.choice(len(src_key_points), 4)
    return np.array([src_key_points[i] for i in r]), np.array([dst_key_points[i] for i in r])
```

RANSAC 알고리즘을 수행할 때 랜덤하게 4개의 점을 뽑아 return하는 function이다.

```
def cal_H(src, dest):
    A = []
    for i in range(len(src)):
        x, y = src[i][0], src[i][1]
        x_prime, y_prime = dest[i][0], dest[i][1]
        A.append([x, y, 1, 0, 0, 0, -x * x_prime, -x_prime * y, -x_prime])
        A.append([0, 0, 0, x, y, 1, -y_prime * x, -y_prime * y, -y_prime])
    A = np.array(A)
    _, _, V = np.linalg.svd(A)
    H = np.reshape(V[-1], (3, 3))
    H = (1 / H.item(8)) * H
    return H
```

랜덤하게 선택한 4개의 점을 활용하여 Homography matrix를 계산하는 함수이다. 4개의 점이 주어진다면, $H\chi_i = \chi'_i$, $(x,y) \rightarrow (x', y')$의 식을 만족하는 Homography matrix(H)를 구해주어야 한다.

대응점에 대하여 관계식을 수식으로 정리하면

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11}x + h_{12}y + h_{13} \\ h_{21}x + h_{22}y + h_{23} \\ h_{31}x + h_{32}y + h_{33} \end{bmatrix}$$ 이고,

이를 x'과 y'으로 정리하면

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}, \quad y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$ 이 된다.

이를 $AH^T = O,\ H = [h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}, h_{33}]$ 의 형태로 변형을 하게 된다면

$$\begin{aligned} x'(h_{31}x + h_{32}y + h_{33}) &= h_{11}x + h_{12}y + h_{13} \\ y'(h_{31}x + h_{32}y + h_{33}) &= h_{21}x + h_{22}y + h_{23} \\ h_{11}x + h_{12}y + h_{13} - x'(h_{31}x + h_{32}y + h_{33}) &= 0 \\ h_{21}x + h_{22}y + h_{23} - y'(h_{31}x + h_{32}y + h_{33}) &= 0 \end{aligned}$$

다음과 같은 수식이 된다. 이제 아래의 정리된 두 줄의 수식을 $AH^T = O$와 같은 행렬과 Vector의 연산 형태로 표현하면,

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -xx' & -yx' & -x' \\ 0 & 0 & 0 & x & y & 1 & -xy' & -yy' & -y' \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = O$$

다음과 같이 정의될 수 있다. 결론적으로 4개의 점에 대하여 x, y, x', y'을 사용하여 A 행렬을 만들어주면 Homography matrix를 구해줄 수 있다.

Homography matrix는 구해진 A행렬에 대한 SVD를 구하여 찾아낼 수 있다.

$A = UDV^T$에서 V의 마지막 열 벡터가 Homography matrix가 된다.

```
def dist(src, dest, H):
    x = np.transpose(np.matrix([src[0], src[1], 1]))
    x_prime = np.transpose(np.matrix([dest[0], dest[1], 1]))

    x_hat = np.dot(H, x)
    x_hat = x_hat /x_hat.item(2)

    e = x_prime - x_hat
    return np.linalg.norm(e)
```

RANSAC을 수행할 때 생성된 Homography matrix가 적절한지 판단하기 위해 mapping된 점과 실제 점의 거리를 측정하기 위한 함수이다.

```
H = ransac_for_homography(src_key_points, dst_key_points, good_src_key_points,
good_dst_key_points)
```

정의한 함수인 ransac_for_homography()를 활용하여 최종적으로 Homography matrix를 생성하는 코드이다.

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

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

파노라마 이미지로 변환하기 위해서 Homography matrix로 생성된 새로운 좌표로 Mapping 하는 과정이 필요하다. 이를 위해 forward mapping 방법을 적용시켰으나, mapping이 이루어지지 않은 부분 때문에 결과에 문제가 있어 이를 해결하기 위하여 중앙값을 활용하는 interpolation과 backward mapping 방법을 적용시켜보았다.

```
def forward_mapping(img_left, img_right, H):
    src_locs = []
    for x in range(img_right.shape[1]):
        for y in range(img_right.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, _ = img_left.shape
    result = np.zeros((height, width *2, 3), dtype =int)
    for src, dst in 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]] = img_right[src[1], src[0]]
    result[0: height, 0 : width] = img_left
    return result
```

Forward mapping은 $Loc_{new} = HLoc_{old}$ 연산을 수행하여 mapping시켜주는 방법이다. 두 개의 이미지 중 하나의 이미지에 대하여만 Homography 연산을 수행한 뒤 붙여주는 작업을 진행하여 파노라마 이미지를 생성하였다.(두 이미지 중 하나의 이미지는 그대로 사용하게 된다.)

이러한 경우 Homography를 적용한 새로운 공간에 모든 픽셀들이 mapping되지 않는 경우가 발생한다. 따라서 모든 점이 채워지지 않기 때문에 이미지에 검은색으로 줄무늬가 발생하는 문제점이 생긴다. 이를 해결하기 위해서 픽셀의 값이 비어있는 공간을 Homography를 적용하지 않았을 때 근사한 픽셀의 주변(3*3) RGB 값들에 대한 median 값으로 Interpolation 기법을 적용하려고 시도하였다. 그리고 Backward mapping도 추가적으로 구현하여 이를 비교해보았다.

```
def forward_mapping_interpolation_median(img_left, img_right, H):
    ...
    interpolation_locs = []
    for x in range(width *2):
        for y in range(height):
            if np.sum(result[y, x] == [0,0,0]) ==3:
                loc = [x, y, 1]
                interpolation_locs.append(loc)

    interpolation_locs = np.array(interpolation_locs).transpose()

    H_inverse = np.linalg.inv(H)
    ori_locs = np.matmul(H_inverse, interpolation_locs) # Original pixels index
    ori_locs = ori_locs / ori_locs[2, :]
    ori_locs = ori_locs[:2, :]
    interpolation_locs = interpolation_locs[:2, :]
    ori_locs = np.round(ori_locs, 0).astype(int)

    for ori, res in zip(ori_locs.transpose(), interpolation_locs.transpose()):
        if ori[1] >= height or ori[0] >= width or ori[0] <0 or ori[1] <0:
            continue

        near_rgb = [img_right[ori[1], ori[0]]]
        up_plag, down_plag, left_plag, right_flag =False, False, False, False
        if ori[1] >0:
            near_rgb.append(img_right[ori[1]-1, ori[0]])
            up_plag =True
        if ori[1] < height -1:
            near_rgb.append(img_right[ori[1]+1, ori[0]])
            down_plag =True
        if ori[0] >0:
            near_rgb.append(img_right[ori[1], ori[0]-1])
            left_plag =True
        if ori[0] < width -1:
            near_rgb.append(img_right[ori[1], ori[0]+1])
            right_flag =True
        if up_plag and left_plag:
            near_rgb.append(img_right[ori[1]-1, ori[0]-1])
        if up_plag and right_flag:
            near_rgb.append(img_right[ori[1]-1, ori[0]+1])
        if down_plag and left_plag:
            near_rgb.append(img_right[ori[1]+1, ori[0]-1])
        if down_plag and right_flag:
            near_rgb.append(img_right[ori[1]+1, ori[0]+1])
```

```
            near_rgb = np.array(near_rgb)
            median_b = np.median(near_rgb.transpose()[0])
            median_g = np.median(near_rgb.transpose()[1])
            median_r = np.median(near_rgb.transpose()[2])
            result[res[1], res[0]] = [median_b, median_g, median_r]

    return result
```

Interpolation이 필요한 픽셀 정보들이 없는 부분을 찾아준 뒤 각 부분들을 H^{-1}로 계산하여 Homography matrix로 옮겨지기 전 정보에서 mapping 될 수 있는 픽셀의 위치를 계산한다. 그리고 그 픽셀을 중심으로 8가지 방향에 대한 pixel 정보를 가져온다. 최대 총 9개의 pixel 정보를 사용할 수 있다. 해당 pixel 정보로 RGB 각각의 median값을 계산하여준다. 그리고 해당 값으로 Interpolation을 수행하게 된다.

```
def backward_mapping(img_left, img_right, H):
    dst_locs = []
    for x in range(img_right.shape[1], img_right.shape[1] *2):
        for y in range(img_right.shape[0]):
            loc = [x, y, 1]
            dst_locs.append(loc)
    dst_locs = np.array(dst_locs).transpose()

    H_inverse = np.linalg.inv(H)

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

    dst_locs = dst_locs[:2, :]
    height, width, _ = img_right.shape
    result = np.zeros((height, width *2, 3), dtype =int)
    for src, dst in zip(src_locs.transpose(), dst_locs.transpose()):
        if src[1] >= height or src[0] >= width or src[0] <0 or src[1] <0:
            continue

        result[dst[1], dst[0]] = img_right[src[1], src[0]]
    result[0: height, 0 : width] = img_left
    return result
```

Backward mapping은 $Loc_{old} = H^{-1}Loc_{new}$ 연산을 통해 수행할 수 있다. 파노라마 이미지 결과에 mapping되는 원본이미지의 정보를 가져오게 된다. Backward mapping 과정에서 단순히 반올림으로 좌표를 mapping하였는데 이 부분을 조금 다르게 사용한다면 좀 더 좋은 화질의 파노라마 이미지를 구할 수 있을 것 같다.

3. 프로젝트 결과

3.1) Normal Result

Forward Mapping | Forward Mapping With Interpolation | Backward Mapping

3.2) Best Result

Forward Mapping | Forward Mapping With Interpolation | Backward Mapping

3.3) Worst Result

3.4) Backward mapping vs. interpolation

위 사진은 forward mapping과 interpolation을 수행한 이미지와(위) backward mapping(아래)을 수행한 결과이다. Backward 기법을 적용한 결과가 좀 더 선명하게 파노라마 이미지가 생성된 것을 볼 수 있다. 그 이유를 추측해보면, Interpolation 기법을 3*3 RGB의 median 값을 사용하는 것을 적용하였기에 흐릿한 결과가 나온

것이라 생각한다. 다른 Interpolation 기법(e.g. bicubic interpolation)을 적용하였다면 좀 더 선명한 파노라마 이미지를 생성할 수 있었을 것 같다.

4. 결론

RANSAC 알고리즘을 활용하다보니, Random하게 선택되는 4개의 대응점에 따라 다른 결과가 나타나는 것을 볼 수 있다. Mapping을 수행한 이미지는 화질이 좋지 못하다. Mapping을 수행할 때 Homograpy matrix로 연산하여 찾은 좌표의 소수점을 단순히 반올림으로 처리하였는데, 이 부분에 대하여 다른 아이디어를 생각하여 적용하면 좀 더 좋은 화질의 파노라마 이미지를 생성할 수 있을 것 같다고 생각하였다.