

과제에 사용된 이미지



img1=(1,1) img2=(1,2)



img3=(2,1) img4=(2,1)



원본 이미지

공통된 코드

```
# 이미지를 불러옴
img1 = cv2.imread('img1.jpg',0) # query Img, 오른쪽 뷔 이미지
img2 = cv2.imread('img2.jpg',0) # train Img, 왼쪽 뷔 이미지

# SIFT 추출기 생성
sift = cv2.SIFT_create()

# SIFT 추출기를 이용하여 각 이미지의 특징점과 디스크립터를 찾는다.
# 특징점, 디스크립터 = sift.detectAndCompute(이미지, 마스크(optional))
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
```

이미지를 불러오고 SIFT 추출기를 생성한다. detectAndCompute함수는 sift를 이용하여 이미지의 특징점과 디스크립터를 찾아준다.

```
FLANN_INDEX_KDTREE = 1

index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees=5) #key=value
search_params = dict(checks= 50)
flann = cv2.FlannBasedMatcher(index_params, search_params)

matches = flann.knnMatch(des1, des2, k=2)

# knn으로 뽑은 1위(m)가 0.9*2위(n)보다 가까우면 [good]에 들어감
good=[]

for m,n in matches:
    if m.distance < 0.9*n.distance:
```

good.append(m)

FLANN은 Fast Library for Approximate Nearest Neighbors의 약자로 큰 이미지에서 특성을 매칭할 때 성능을 위해 최적화된 라이브러리 모음을 의미한다. FlannBasedMatcher는 자료형 인자 indexParams, searchParams를 필요로 한다. index_params algorithm=1이 되는데 이는 아래 이미지와 같이 사용할 알고리즘에 대한 선택자이다.

FLANN_INDEX_LINEAR=0: 선형 인덱싱, BFMatcher와 동일

FLANN_INDEX_KDTREE=1: KD-트리 인덱싱 (trees=4: 트리 개수(16을 권장))

FLANN_INDEX_KMEANS=2: K-평균 트리 인덱싱 (branching=32: 트리 분기 개수, iterations=11: 반복 횟수, centers_init=0: 초기 중심점 방식)

FLANN_INDEX_COMPOSITE=3: KD-트리, K-평균 혼합 인덱싱 (trees=4: 트리 개수, branching=32: 트리 분기 개수, iterations=11: 반복 횟수, centers_init=0: 초기 중심점 방식)

FLANN_INDEX_LSH=6: LSH 인덱싱 (table_number: 해시 테이블 수, key_size: 키 비트 크기, multi_probe_level: 인접 버킷 검색)

FLANN_INDEX_AUTOTUNED=255: 자동 인덱스 (target_precision=0.9: 검색 백분율, build_weight=0.01: 속도 우선순위, memory_weight=0.0: 메모리 우선순위, sample_fraction=0.1: 샘플 비율)

search_params는 특성 매칭을 위한 반복 횟수를 의미하고 높을 수록 정확도는 상승하지만 속도가 느려진다는 특징이 있다. flannBasedMatcher를 이용해 k개의 가까운 점들을 뽑아준다.

```
MIN_MATCH_COUNT = 4
if len(good)>MIN_MATCH_COUNT:
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)

    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    #M, mask = cv2.findHomography(src_pts, dst_pts)

    matchesMask = mask.ravel().tolist()

    h,w = img1.shape
    pts = np.float32([[0,0],[0,h-1],[w-1,h-1],[w-1,0]]).reshape(-1,1,2) # 차원, 행, 열

    dst = cv2.perspectiveTransform(pts,M)
    img2 = cv2.polylines(img2,[np.int32(dst)],True,255,3,cv2.LINE_AA)

else:
    print("Not enough matches are found - %d%%" %(len(good),MIN_MATCH_COUNT))
    matchesMask=(None)
```

if 문에서는 knn과 ratio로 뽑은 good 리스트에 포함된 요소가 5개 이상일 때만 실행된다. src_pts와 dst_pts는 good list의 요소로 반복되는 for문이다. m은 아래 사진과 같이 구성되어 있는데 data type이 DMatch인것을 확인할 수 있다.

```
cv::DMatch::DMatch ( int _queryIdx,
                     int _trainIdx,
                     int _imgIdx,
                     float _distance
)
```

m DMatch < cv2.DMatch 000001C3BB951D70>

name/data type/value

DMatch

DMatch는 매칭된 결과를 표현하는 객체로 queryIdx, trainIdx, imgIdx, distance로 구성되어 있다. 즉 순서대로 img1의 특징점 인덱스, img2의 특징점 인덱스, img가 많을 때 train설명자의 이미지 인덱스, 거리 값을 담고 있다. 따라서 m.queryIdx, m.trainIdx는 서로 매칭된 포인트 kp1, kp2의 인덱스값이 된다.

```

cv::KeyPoint::KeyPoint ( Point2f pt,
                        float    size,
                        float    angle = -1.,
                        float    response = 0.,
                        int     octave = 0,
                        int     class_id = -1
)

```

kp1	tuple	414	(< cv2.KeyPoint 00000	Parameters
kp2	tuple	425	(< cv2.KeyPoint 00000	<p>pt x & y coordinates of the keypoint</p> <p>size keypoint diameter</p> <p>angle keypoint orientation</p>

name/data type/size/value

KeyPoint

위 이미지와 같이 kp1, kp2는 value로 KeyPoint를 갖고 있다. kp1[Idx].pt를 통해 해당 포인트의 x,y 값을 받는 것이다. 그 후 findHomography함수로 행렬M을 구한다. source point, destination point만 사용했을 시 오류가 많기 때문에 ransac을 사용하고, 뒤의 숫자는 ransac 에러 허용치를 뜻한다. 사용된 이미지는 good리스트의 크기가 237이다. 237개 중 4개를 무작위로 뽑아서 M행렬을 생성한다. 하지만 237개에 M행렬 곱했을 때 각자의 destination point에 정확히 맞을 수 없게 되는데 이 오차를 여러 허용치와 비교하여 inlier, outlier를 구분하게 된다. 이 과정을 default값인 2000번 반복하여 inlier의 개수가 가장 많은 행렬M을 반환해주게 된다. mask는 0과 1로 이루어져 있는 array로 inlier인 매칭포인트는 1, outlier인 매칭포인트는 0으로 반환된다. ravel().tolist()은 현재 237행 1열로 이루어진 mask를 1행 237열로 바꾸고 datatype을 list로 변환한다는 의미이다.

0	0
1	0
2	0
3	0
4	0
5	0
6	1
7	0
8	1

mask, type=ndarray

pts, dst는 이미지에 그림을 그리기 위한 정보로 사각형의 끝점을 M행렬과 연산하여 polylines을 이용해 그림에 사각형을 선으로 나타내준다.

만약 good이 4개가 안된다면 ransac에서 사용할 수 있는 정보가 부족하기 때문에 else문이 실행된다. ratio가 과하게 적을 경우 good 매치의 수가 적어질 수 밖에 없기 때문에 else문에 들어오게 된다면 ratio를 재설정하거나 이미지 자체의 문제일 수 있다.

```

draw_params = dict(matchColor=(0,255,0), singlePointColor=None, matchesMask=matchesMask, flags=2)

img3 = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)

cv2.imshow('gray', img3)

# img1(오른쪽 뷰), img2(왼쪽 뷰)
width = img2.shape[1]+img1.shape[1]
height = img2.shape[0]+img1.shape[0]

dst=cv2.warpPerspective(img1, M, (width, height))
cv2.imshow("Warping right to left", dst)

dst[0:img2.shape[0], 0:img2.shape[1]] = img2
cv2.imshow("stiching", dst)

```

**'은 key와 값을 dictionary로 저장한다. img3=cv2.drawMatches(img1, kp1, img2, kp2, good, None, matchColor=(0,255,0), singlePointColor=None, matchesMask=matchesMask, flags=2)로 사용하여도 같은 결과가 나온다는 것을 알 수 있다. img1과 img2의 good에 들어간 점들을 초록색 선으로 나타내어준다. Warping 결과 이미지는 img1을 M행렬로 변화를 준 후 img1과 img2의 가로, 세로 크기를 각각 더한 크기만큼 출력해준다. stitching이미

지는 warping으로 오른쪽 이미지를 오른쪽으로 이동시킨 상태 위에 그대로 왼쪽 이미지를 입히는 것이다. 따라서 0부터 img2의 행크기, 0부터 img2의 열크기만큼의 영역을 img2로 넣어주면 stitching 결과가 나오는 것이다.

1. Stitching - Level 1-a

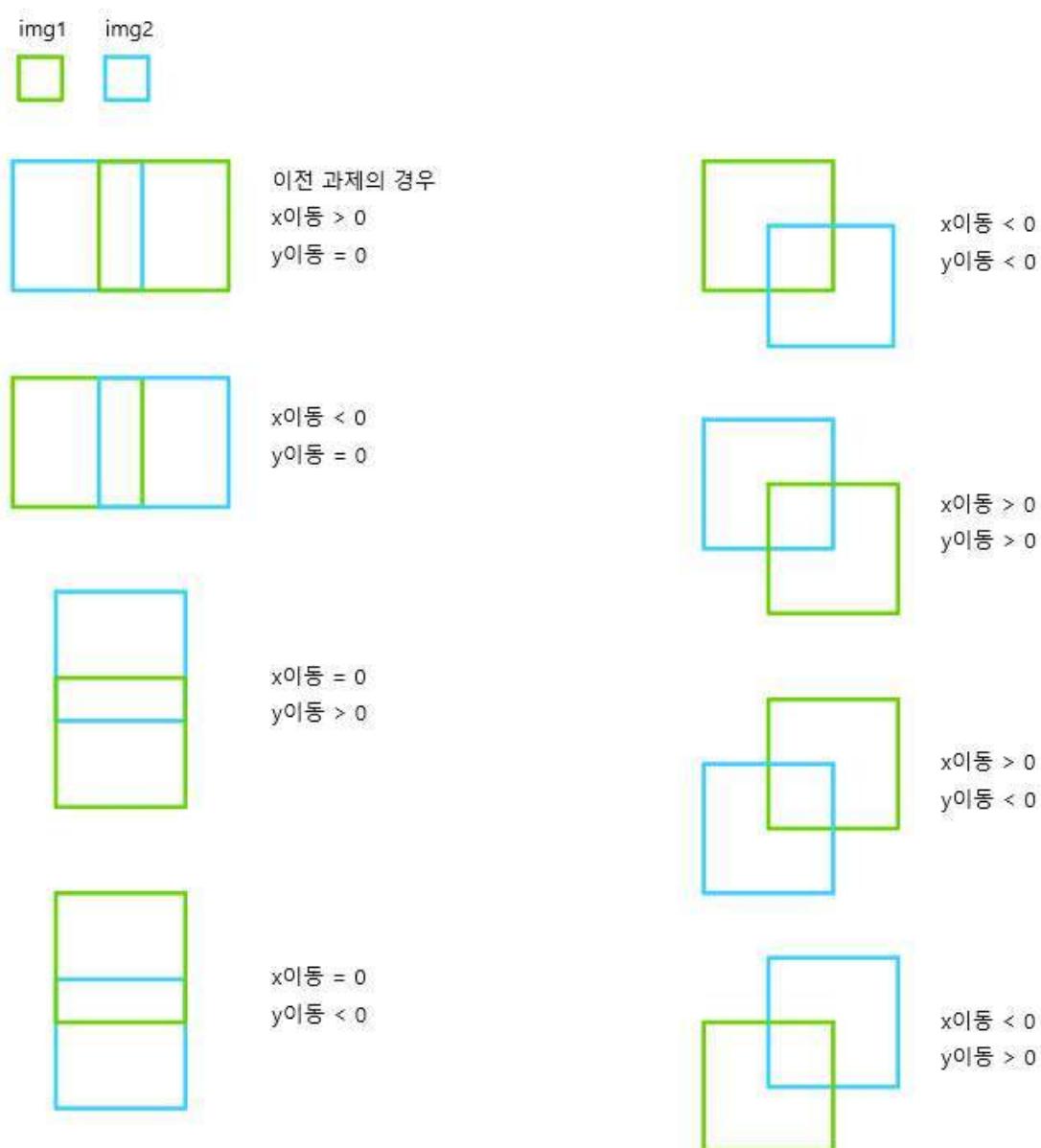
-2X2 영상 중 두 장의 영상 위치 관계 파악

1-1 idea 및 구현 방법

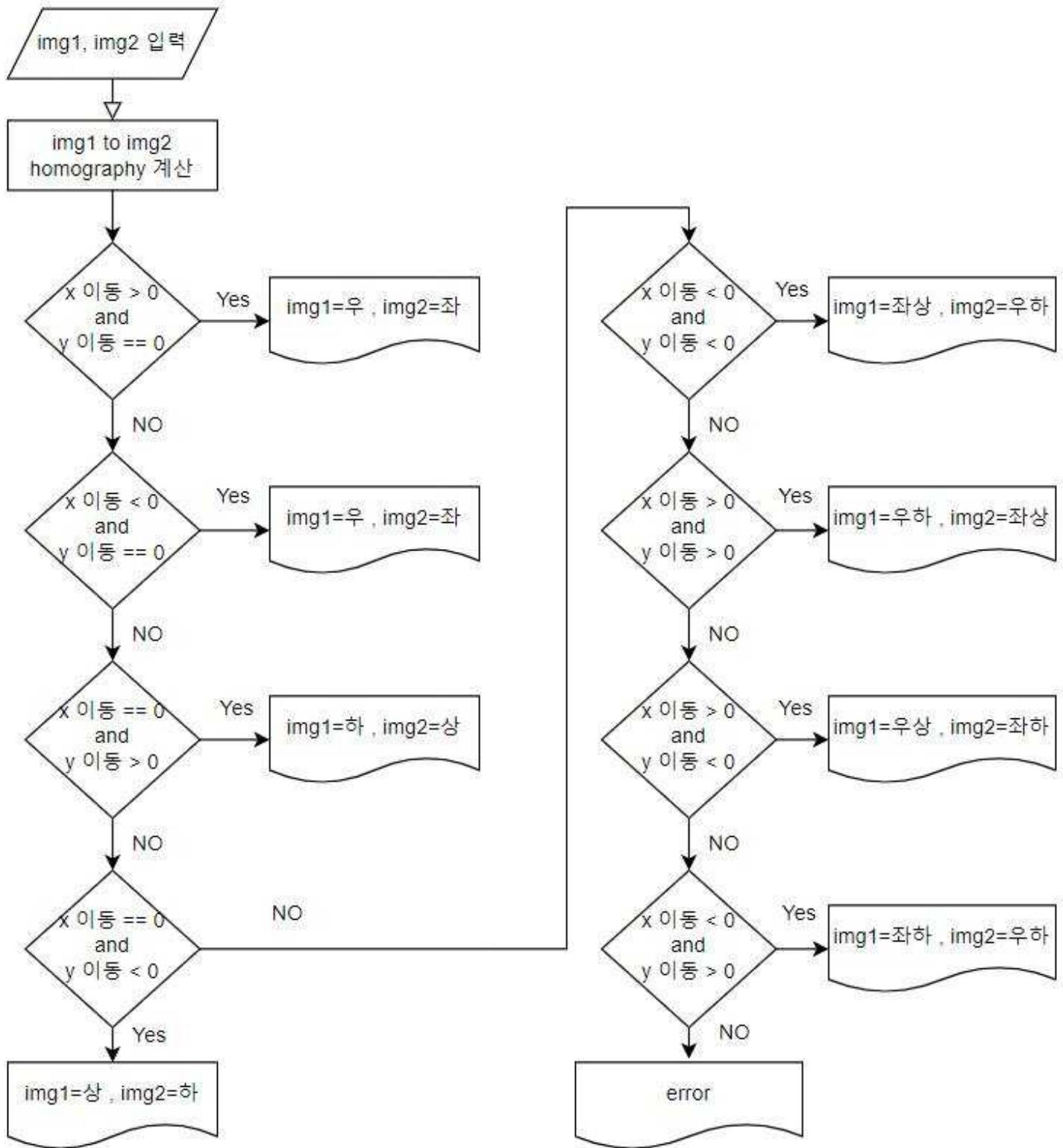
$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Homography

해당 과제에서는 homography 행렬 값 중 c, f를 이용하여 이미지의 위치를 파악할 수 있다. g,h가 0일 때, c는 x이동 f는 y이동을 나타낸다. 이전 과제 코드에서 input을 설정할 때 왼쪽 이미지를 기준으로 오른쪽 이미지가 움직이는 형태로 구현하였다. 그 이유는 cv2.findHomography의 결과로 나오는 M행렬의 (0,2)=c,(1,2)=f의 값이 음수가 나오지 않게 하기 위함이었다. 따라서 두 이미지의 위치를 고려하기 위해서는 M[0,2],M[1,2]의 값을 이용하여 상대적인 위치를 알 수 있을 것이다. 예를 들어 query이미지가 왼쪽, train 이미지가 오른쪽일 경우 M[0,2]는 query는 train에 맞추기 위해 기준보다 왼쪽으로 이동해야하기 때문에 음수값이 나오게 된다. 따라서 M[0,2],M[1,2] 음수값이 나온다는 의미는 query가 왼쪽, 위쪽임을 의미한다. 자세한 설명은 아래 이미지와 같다.



1-2 알고리즘 흐름도



```

siftcv2.SIFT_create()
matrix = detect(img1, img2) #detect함수에서 findHomography의 결과 M행렬을 return해줌
round_M = matrix.astype('int32')
if round_M[0,2]>0 and round_M[1,2] in range(-10,10):
    print("img1 = 우, img2=좌")
elif round_M[0,2] in range(-10,10) and round_M[1,2]>0:
    print("img1 = 하, img2=상")
elif round_M[0,2]<0 and round_M[1,2] in range(-10,10):
    print("img1 = 좌, img2=우")
  
```

```

elif round_M[0,2] in range(-10,10) and round_M[1,2]<0:
    print("img1 = 상, img2=하")
elif round_M[0,2]>0 and round_M[1,2]>0:
    print("img1 = 우하, img2=좌상")
elif round_M[0,2]<0 and round_M[1,2]<0:
    print("img1 = 좌상, img2=우하")
elif round_M[0,2]>0 and round_M[1,2]<0:
    print("img1 = 우상, img2=좌하")
elif round_M[0,2]<0 and round_M[1,2]>0:
    print("img1 = 좌하, img2=우상")
else:

```

input으로 들어온 img1과 img2의 M을 구해준 후 형변환을 통해 round_M을 생성한다. round_M을 구하는 이유는 M[0,2],M[1,2]이 float형태로 값 비교를 할 시 0.00003과 같이 아주 작은 값이 0으로 계산되는 것이 아닌 양수로 계산되기 때문에 아래와 같은 결과가 출력될 수 있다. 아래 이미지는 y축 이동을 의미하는 M[1,2]이 아주 작은 값으로 인식되어 상하가 나뉜 것이다. 따라서 아주 작은 값들은 무시하기 위해 round_M행렬을 이용하여 구분해주었다.



그림1. 잘못된 예시

중간중간 보이는 `round_M[1,2] in range(-10,10)` 부분은 `round_M[1,2] == 0` 을 의미한다. 그림1처럼 round_M 값을 쓰더라도 미세한 오차로 0이 아닌 경우가 생겨 잘못 인식하는 경우가 생겼고, 이를 방지하기 위해 threshold 역할로 -10~10정도는 0으로 허용해주는 역할을 한다. 따라서 M[0,2]=210, M[1,2]=4 일 경우 육안으로 봤을 때는 y축 이동이 없어 상하 이동이 없다고 볼 수 있지만 조건을 M[1,2]==0으로 두면 False로 상하까지 나뉘어 그림1과 같이 출력되고, M[1,2] in range(-10,10)이면 True가 되어 좌우만 구분할 수 있는 것이다.

1-3 여러 예시에 적용 결과 (내 이미지)





교수님 이미지)





2. Stitching - Level 1-b

-네 장의 사분할 영상에 대한 Stitching (1,1)->(1,2)->(2,1)->(2,2)

2-1 HW#7에서 추가로 고려해야 할 점에 대해 분석 및 구현 방법

1. 출력 크기

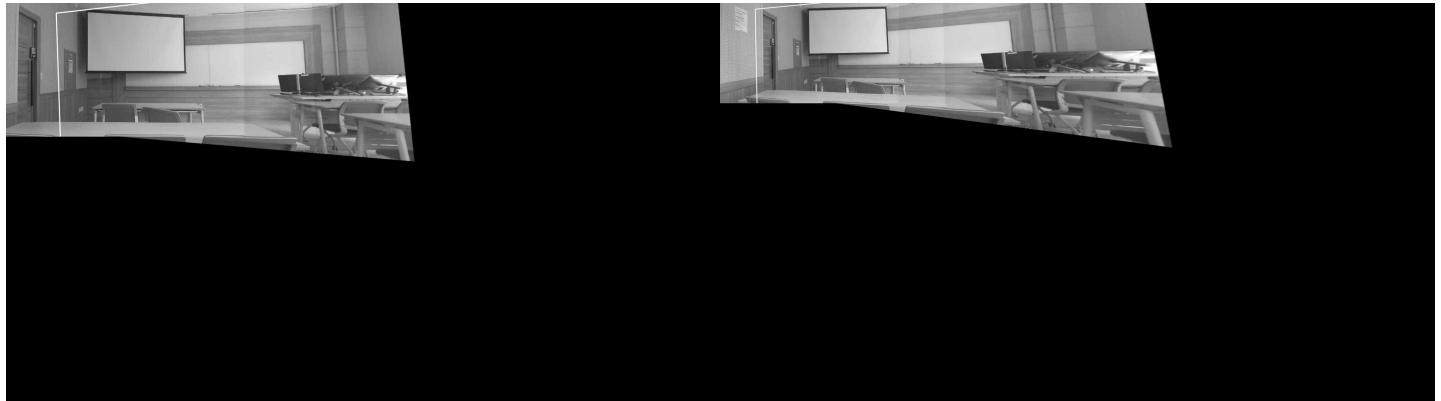


그림2 크기를 고려하지 않고 출력하였을 때

기존 코드에서는 `cv2.warpPerspective`의 파라미터인 dst 크기를 결정할 때 두 영상의 크기를 더해주는 형태로 연산을 하여 영상의 `borderValue`가 0으로 채워져 그림2처럼 불필요한 부분들이 많이 생겼다. 그림2는 이전 과제의 출력창으로 사진을 연결할수록 이미지의 크기가 불필요하게 커지는 것을 보여준다. 이렇게 불필요한 공간이 늘어나면 스티칭 해야하는 이미지가 많아질수록 연산량이 기하급수적으로 늘어날 것이다. 이런 현상을 수정하기 위해

```
if round_M[1,2]-mov_row-10>0:  
    total_row=total_row+round_M[1,2]-mov_row  
    mov_row=round_M[1,2]  
if round_M[0,2]-mov_col-10>0:  
    total_col=total_col+round_M[0,2]-mov_col  
    mov_col=round_M[0,2]  
  
dst=cv2.warpPerspective(right, matrix, (total_col, total_row), borderValue=0)
```

`mov_row`, `mov_col`의 초기값은 0이고 이미지들이 이동한 최대 거리를 의미한다. `total_row`와 `total_col`은 첫번째로 들어온 이미지의 크기로 설정된다. 즉 제일 처음 loop를 돌면 (1,1), (1,2)이 합쳐지며 `total_col`값이 업데이트 된다. 두 번째 loop를 돌면 `total_row`의 값이 업데이트되고, 마지막 루프때는 두 값 모두 변화가 없게 된다. 아래 이미지는 `round_M`행렬과 영상 크기 업데이트를 출력한 것이다. 원본은 같았음에도 불고하고 243, 242 / 360, 359처럼 오차가 존재한다. 따라서 level 1-a와 마찬가지로 10을 threshold 역할을 하기 위해 추가하였다.

```
[[ 1  0 243]  
 [ 0  1  0]  
 [ 0  0  1]]  
1 번째 loop에서 col 크기 변경  
  
[[ 0  0  0]  
 [ 0  0 360]  
 [ 0  0  1]]  
2 번째 loop에서 row 크기 변경  
  
[[ 1  0 242]  
 [ 0  1 359]  
 [ 0  0  1]]  
PS C:\Users\pdbst\Desktop\vis
```

2. 출력 방식

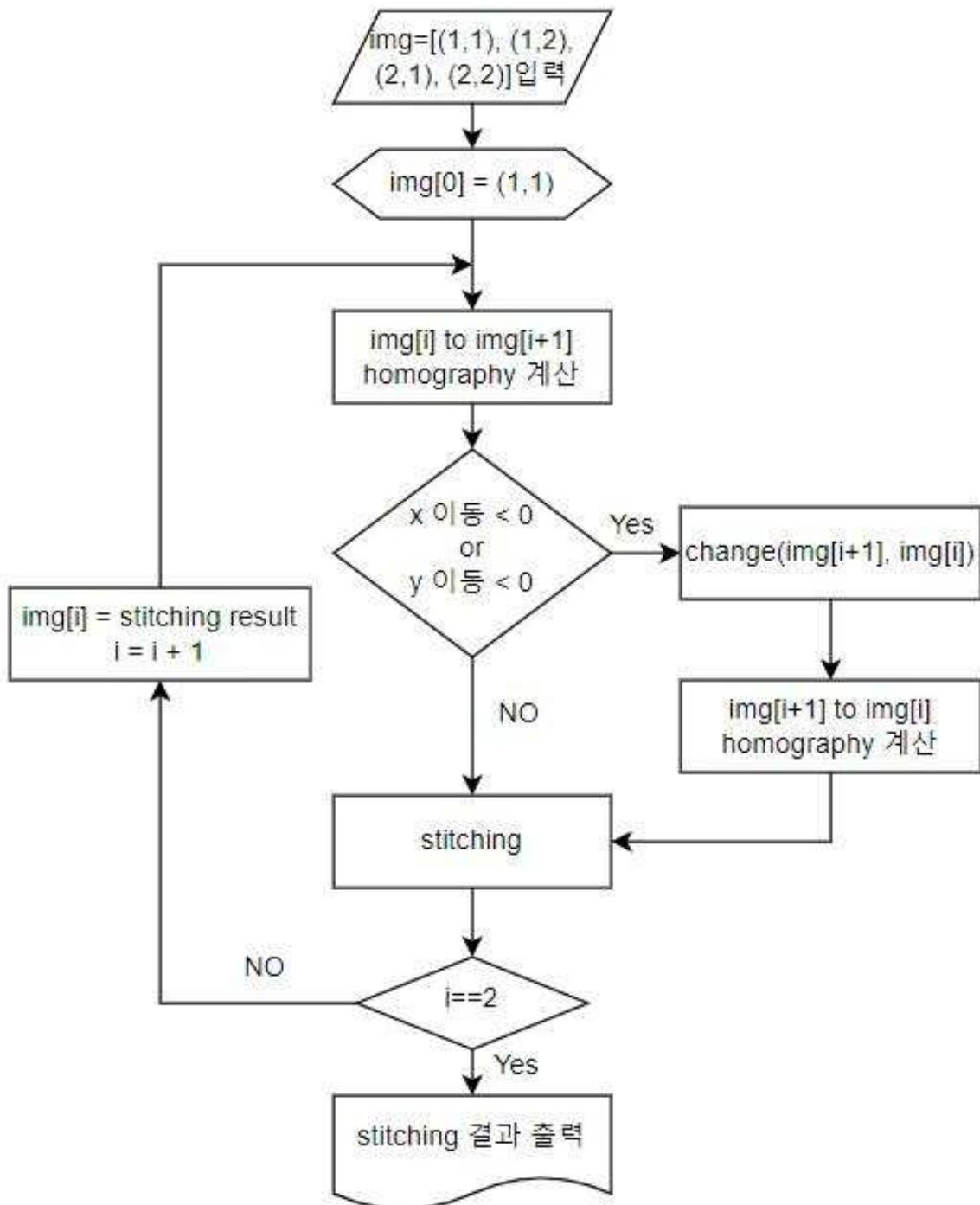
기존 코드는 `query`가 이동한 `dst`에 `train`을 그대로 붙이는 형태였다. 마지막 loop에 (1,1)+(1,2)+(2,1)을 `train`으로 두고 `query`가 (2,2)가 된 상태에서 `query`를 그대로 붙이면 (2,2)는 나타나지 않고 (1,1)+(1,2)+(2,1)을 출력한 것이 그대

로 나타나게 된다. 따라서 결과를 출력하는 부분을 수정할 필요가 있었다.

```
left_pad = np.uint8(np.zeros((dst.shape[0], dst.shape[1])))
left_pad[:left.shape[0], :left.shape[1]] = left
result = np.maximum(dst, left_pad)
```

기존 코드가 덮어쓰기 형식이었다면 수정된 코드는 두 행렬 요소를 각각 비교하여 maximum을 채택하는 것이다. 이렇게 하면 (2,2)가 query의 0값보다 크기 때문에 이미지가 나타나게 된다.

2-2 알고리즘 흐름도



```

def change(i1, i2):
    temp = i1
    i1 = i2
    i2 = temp
    return i1, i2

-----
matrix = detect(I_1,I0)
round_M = matrix.astype('int32')
right = I_1
left = I0
if round_M[0,2]>0 and round_M[1,2] in range(-20,20):
    print("img1 = 우, img2=좌")
elif round_M[0,2] in range(-20,20) and round_M[1,2]>0:
    print("img1 = 좌, img2=상")
elif round_M[0,2]<0 and round_M[1,2] in range(-20,20):
    print("img1 = 좌, img2=우")
    right,left = change(I_1, I0)
    matrix=detect(right, left)
elif round_M[0,2] in range(-20,20) and round_M[1,2]<0:
    print("img1 = 상, img2=좌")
    right,left = change(I_1, I0)
    matrix=detect(right, left)
elif round_M[0,2]>0 and round_M[1,2]>0:
    print("img1 = 우상, img2=좌상")
elif round_M[0,2]<0 and round_M[1,2]<0:
    print("img1 = 좌상, img2=우하")
    right,left = change(I_1, I0)
    matrix=detect(right, left)

```

1a번과 마찬가지로 round_M행렬을 구하여 위치를 구분해준다. 만약 M[0,2], M[1,2] 중 음수인 값이 있으면 스티칭 할 때 이미지가 이동할 공간이 없게 된다. 따라서 음수값이 나오면 change함수를 사용하여 두 이미지를 바꾸어 다시 detect함수에 넣어준다. 그렇게 되면 이전 과제를 수행할 때 오른쪽 이미지를 img1으로 설정한 것과 같은 효과가 나타난다.

2-3 여러 예시에 대한 적용 결과 (내 이미지)



(1,1)+(1,2)



(1,1)+(1,2)+(2,1)+(2,2)

교수님 이미지)



(1,1)+(1,2)+(2,1)



(1,1)+(1,2)



(1,1)+(1,2)+(2,1)



(1,1)+(1,2)+(2,1)+(2,2)

3. Stitching - Level 2

-네 장의 사분할 영상에 대한 Stitching 영상 순서에 대한 정보 없음

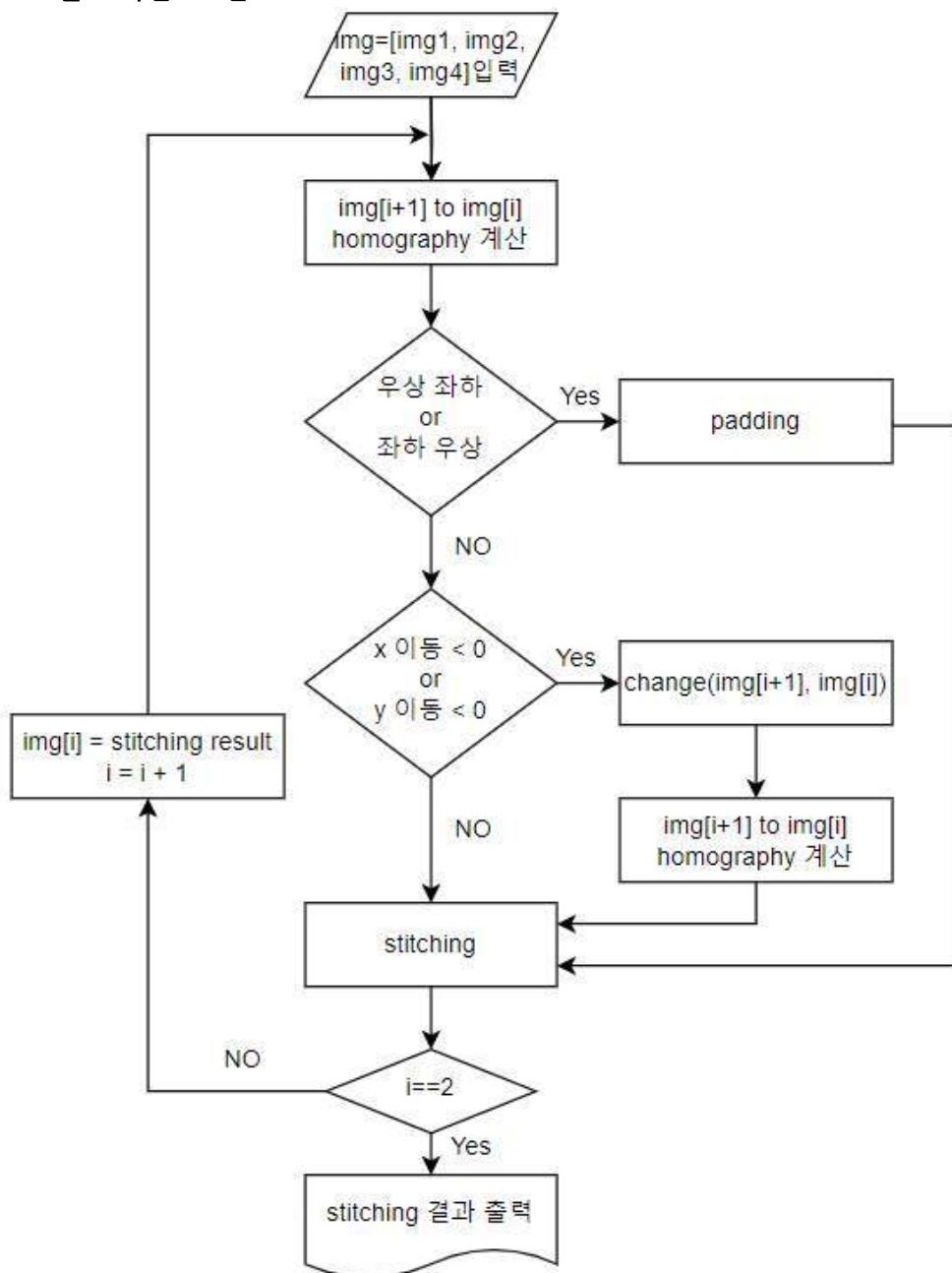
3-1 영상의 순서를 파악하기 위해 사용한 방법



그림3 (2->3->1->4) 순서

초기에는 A1a의 코드를 그대로 사용하여 영상 위치를 파악하였다. 그랬을 때, query이미지가 이전에 스티칭한 이미지로 업데이트되는 방식이 되었고, 그림3과 같은 결과가 나타났다(오른쪽 끝부분). 따라서 새로 들어올 이미지를 query이미지, 이전에 stitching결과를 train으로하여 영역이 큰이미지가 움직이는 것이 아닌 작은 이미지가 움직이는 방식으로 수정하였다.

3-2 알고리즘 흐름도



1. 우상,좌하 / 좌하,우상

```
elif round_M[0,2]>0 and round_M[1,2]<0:  
    #print("img1 = 우상, img2=좌하")  
    zero_mat=np.zeros((abs(round_M[1,2]),L_w))  
    plus_y=abs(round_M[1,2])  
    left=np.uint8(np.r_[zero_mat, left])  
    matrix = detect(right, left)  
  
elif round_M[0,2]<0 and round_M[1,2]>0:  
    #print("img1 = 좌하, img2=우상")  
    zero_mat=np.zeros((L_h,abs(round_M[0,2])))  
    plus_x=abs(round_M[0,2])  
    left=np.uint8(np.c_[zero_mat, left])  
    matrixdetect(right, left)
```



그림4 padding없이 b를 그대로 사용할 경우

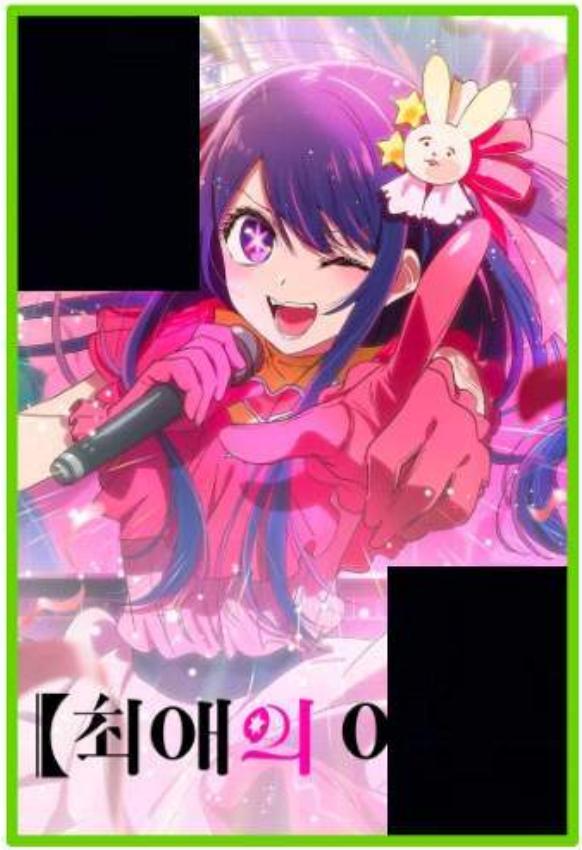
train



query



result



올바른 예시

그림5 padding을 추가하여 사용할 경우

우상 좌하의 경우 x이동이 +이면 y이동은 -가 되고, A1b번의 방식처럼 바꿔서 집어넣으면 x이동이 -, y이동이 +가 된다. 다른 case는 그대로 둘을 바꾸는 방식을 사용하였지만 우상 좌하의 경우 padding을 사용하였다. 만약 padding 없이 사용한다면 그림4처럼 이미지가 잘려나오고, 패딩을 한다면 그림5처럼 정상적으로 출력되는 것을 확인할 수 있다.

3-3 여러 예시에 대한 적용 결과

```

img1 = cv2.imread('1-1.jpg',0)
img2 = cv2.imread('1-2.jpg',0)
img3 = cv2.imread('2-1.jpg',0)
img4 = cv2.imread('2-2.jpg',0)
set1 = [[img1, img2, img3, img4],
         [img1, img2, img4, img3],
         [img1, img3, img2, img4],
         [img1, img3, img4, img2],
         [img1, img4, img2, img3],
         [img1, img4, img3, img2]]
set2 = [[img2, img1, img3, img4],
         [img2, img1, img4, img3],
         [img2, img3, img1, img4],
         [img2, img3, img4, img1],
         [img2, img4, img2, img1],
         [img2, img4, img1, img2]]
set3 = [[img3, img1, img4, img2], set4 = [[img4, img1, img2, img3],
         [img3, img1, img2, img4],
         [img3, img2, img4, img1],
         [img3, img2, img1, img4],
         [img3, img4, img1, img2],
         [img3, img4, img2, img1]]]
         [img4, img1, img3, img2],
         [img4, img2, img1, img3],
         [img4, img2, img3, img1],
         [img4, img3, img1, img2],
         [img4, img3, img2, img1]]
set1                                         set2
set3                                         set4
  
```

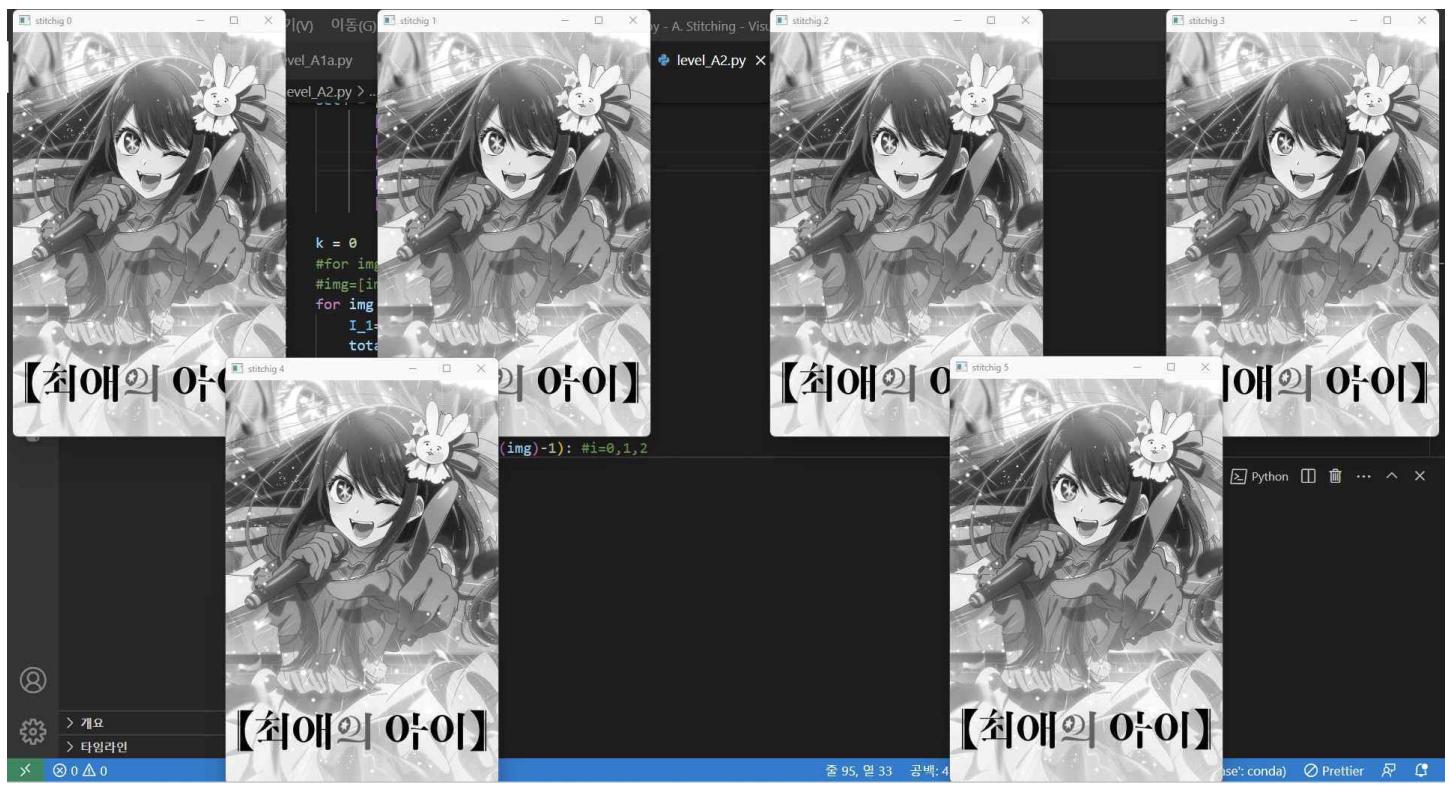
set3

set4

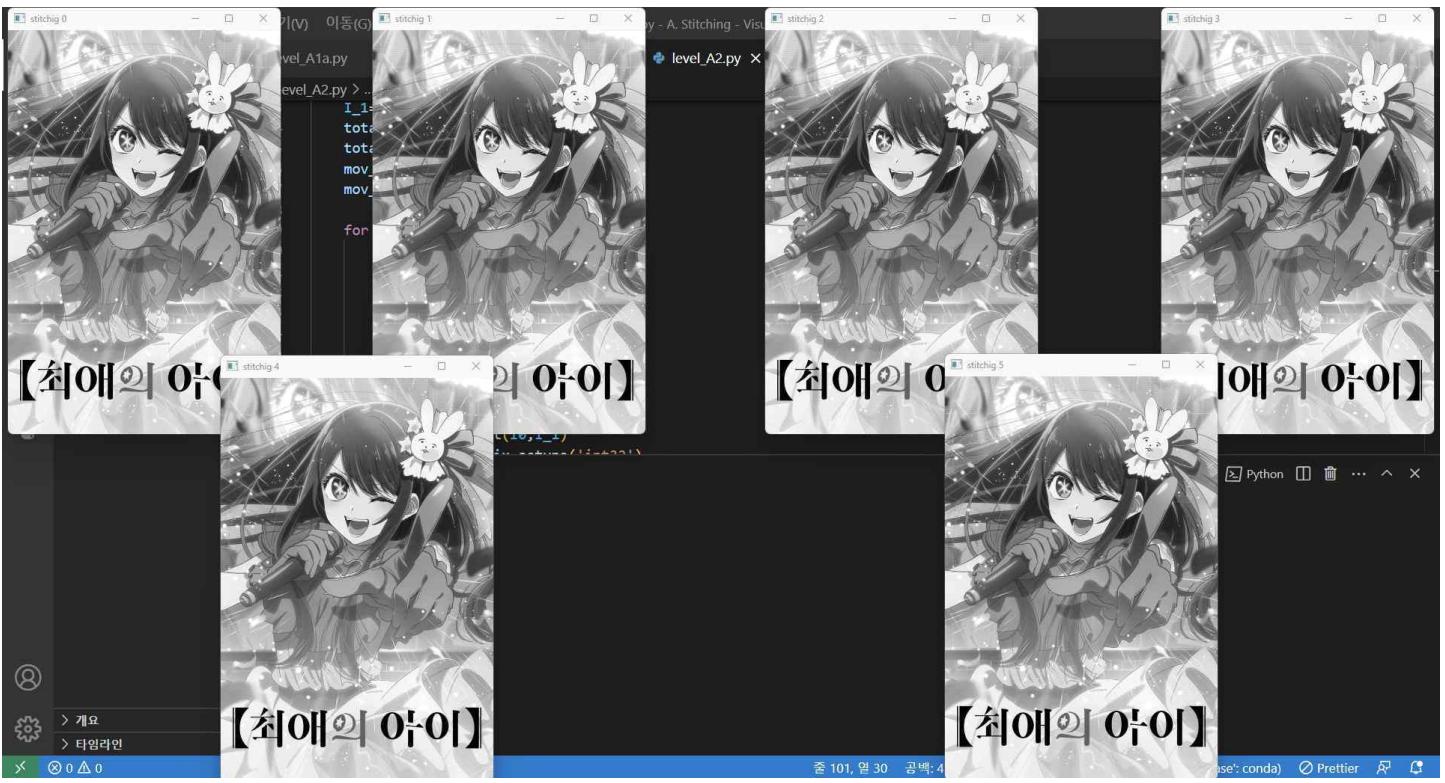
내 이미지)



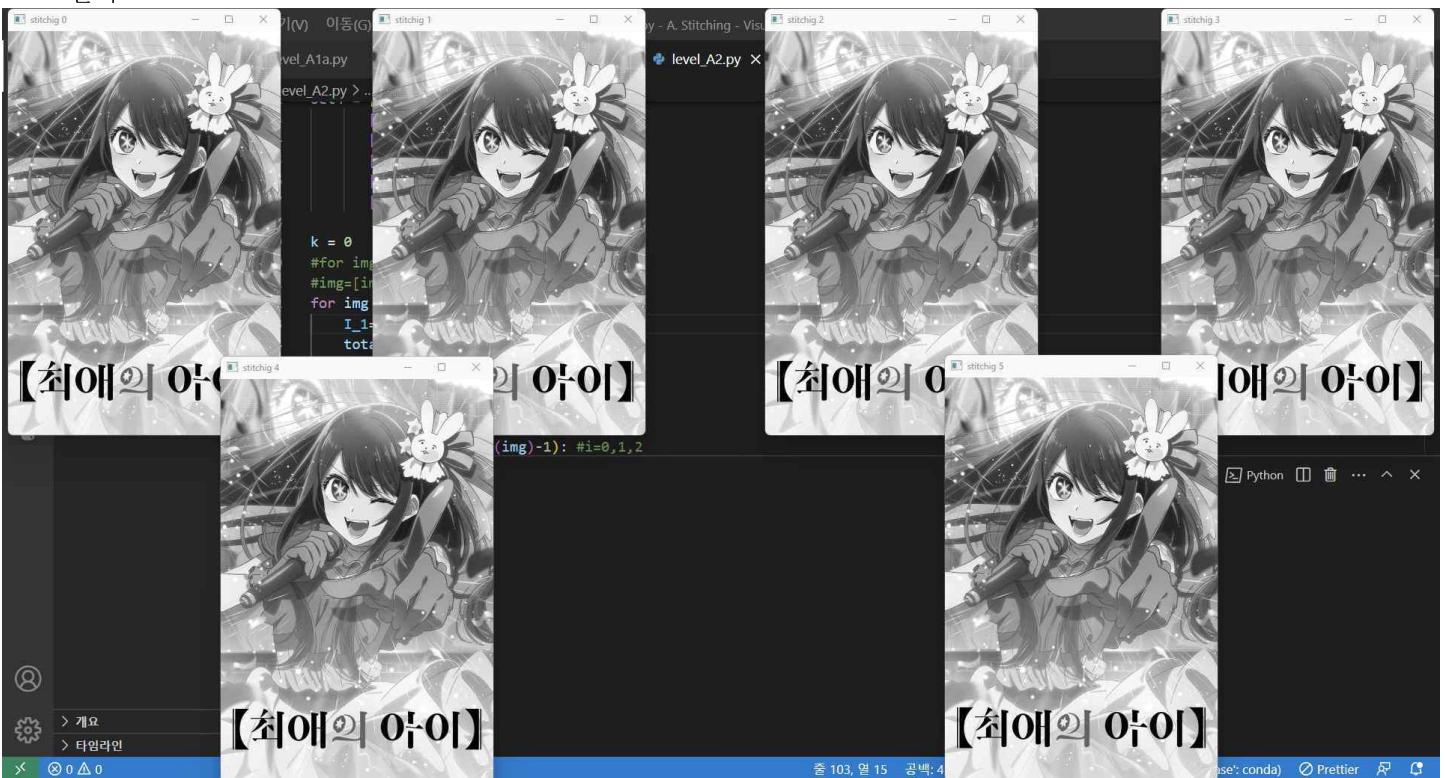
set1 결과



set2 결과

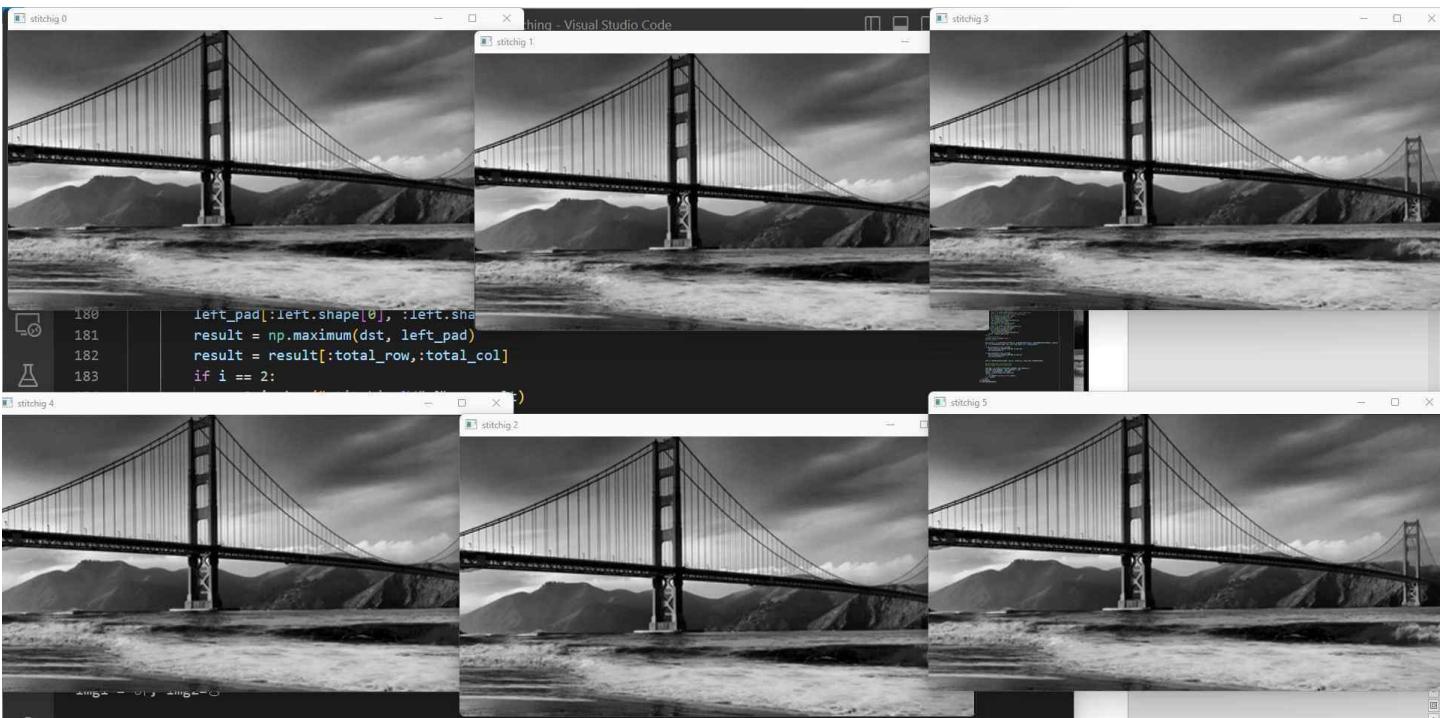


set3 결과



set4 결과

교수님 이미지)



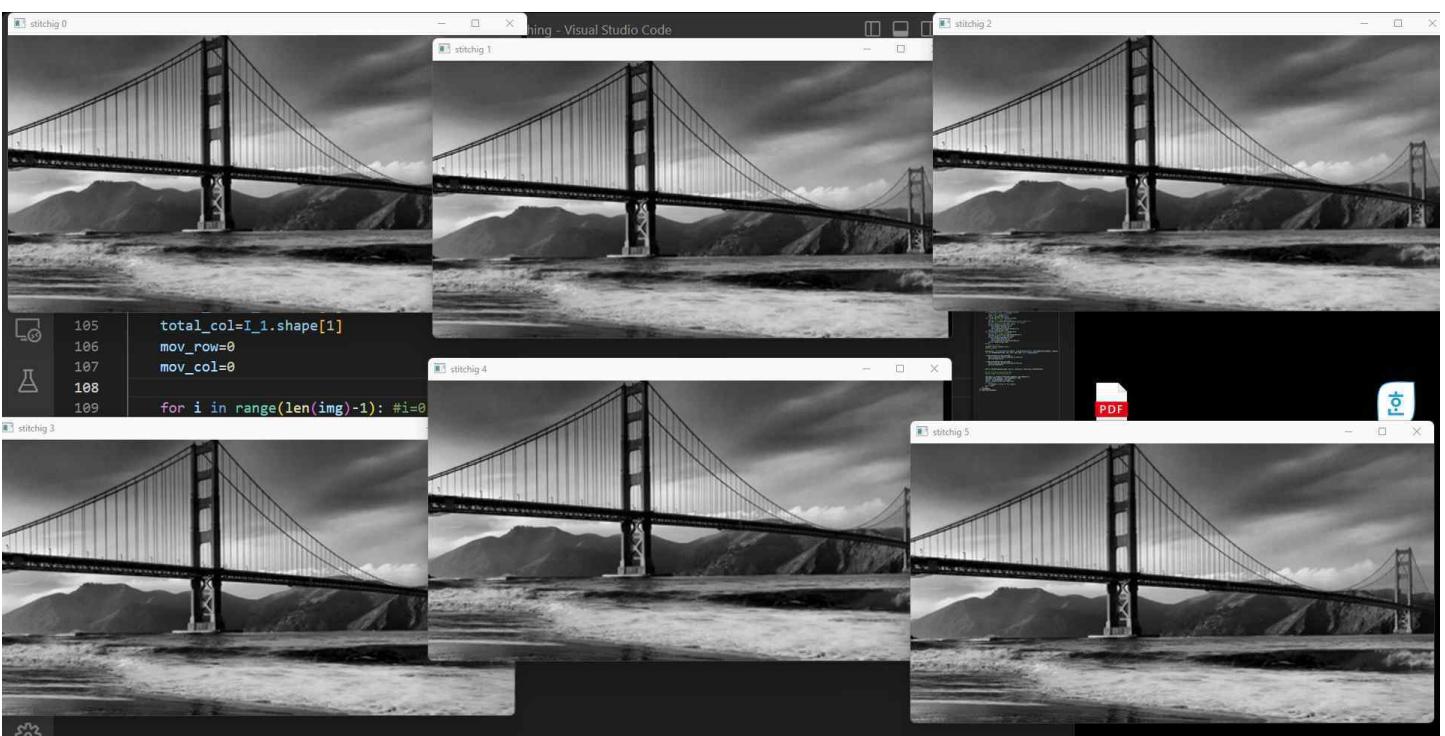
set1 결과



set2 결과



set3 결과



set4 결과

4. What Time? - Level 1

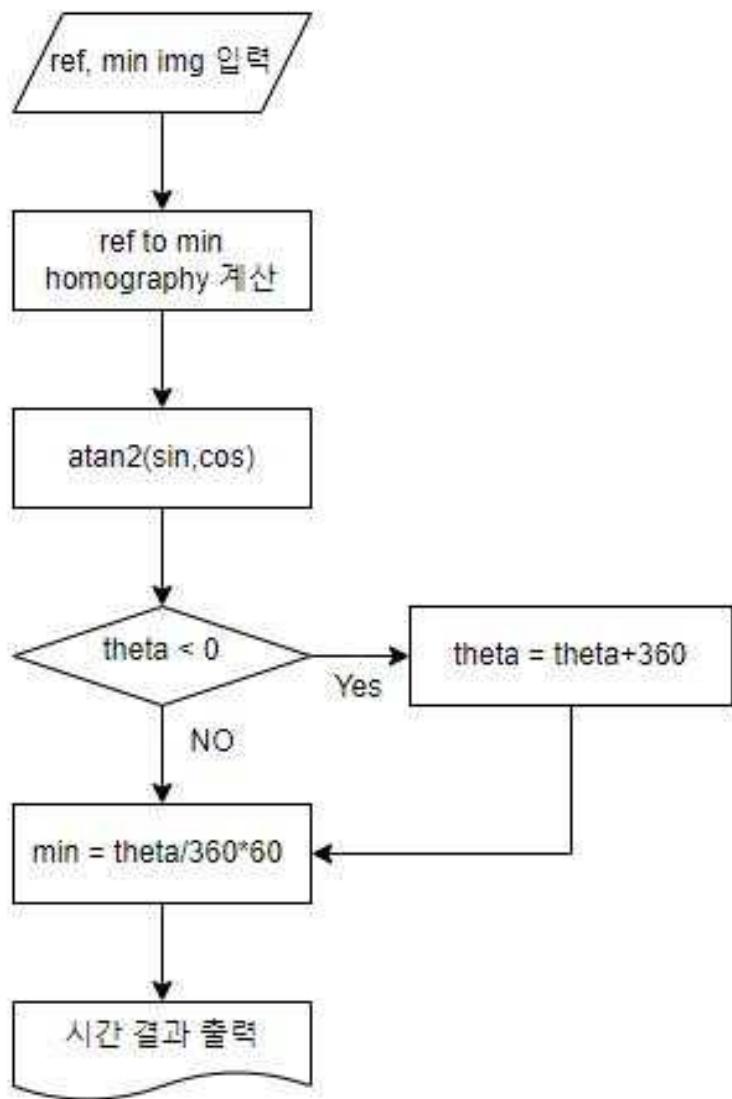
-분침만 존재하는 아날로그 시계 사진에서 현재 몇 분인지 확인하기

4-1 구현을 위해 생각한 IDEA 및 참고사진 활용방안

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

rotation 정보를 포함한 부분은 코드의 $M[0,0]$, $M[0,1]$, $M[1,0]$, $M[1,1]$ 이다. 즉 query이미지가 train이미지로 가기 위해 얼마나 회전해야하는지에 대한 정보가 담겨있다. 따라서 $\text{arctan2}(y,x)$ 함수를 이용하여 시계의 회전각을 구해줄 수 있을 것이다. 이때 00시에서 특정한 시각을 향하는 것이기 때문에 query를 00시, train을 다른 초침으로 둔다.

4-2 알고리즘 흐름도



```

img=[img1, img2, img3, img4]
I_1=img[0] #ref image
for i in range(len(img)-1):
    I0=img[i+1]

    matrix=detect(I_1, I0)
    cos=matrix[0,0]
    sin=matrix[1,0]
    theta=math.degrees(math.atan2(sin,cos))

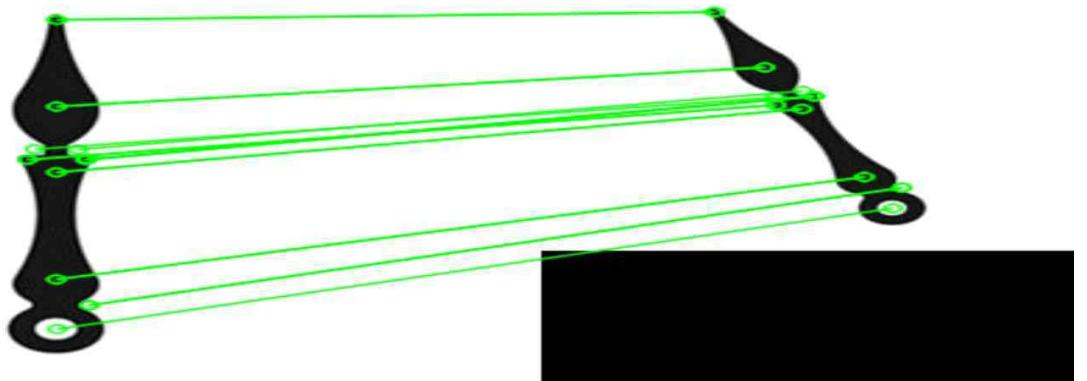
    if theta<0:
        theta=360+theta
    else:
        theta=theta

    t_minute=theta/360*60
    print("시간=",np.int8(t_minute))

```

반복문에 train이미지를 바꾸어가며 시간을 구하는 알고리즘이다. detect함수에서 변환행렬을 구해준 후 cos 요소와 sin 요소를 이용하여 theta를 구한 후 degree로 바꾸어 준다. 만약 theta가 음수값이 뜬다면 360을 더해도 같은 각도 이기 때문에 360을 더해주고, 최종적인 값이 나오게 된다.

4-3 여러 예시에 대한 적용 결과



distance ratio가 0.9이지만 주변 배경이 깔끔하여 특징점이 잘 잡히는 것으로 확인되었다.

시간 = 17 분

시간 = 44 분

시간 = 56 분

출력창

1분씩 오차를 보이지만 큰 오차없이 출력되는 것을 확인할 수 있다.

5. What Time? - Level 2

-분침만 존재하는 아날로그 시계 사진에서 현재 몇 분인지 확인하기

-참고 입력 모두 테두리 존재

-참고 입력 시계 크기 다를 수 있음

5-1 구현을 위해 생각한 IDEA(시계 테두리 또는 크기의 다양성에 대한 문제)

테두리 문제)

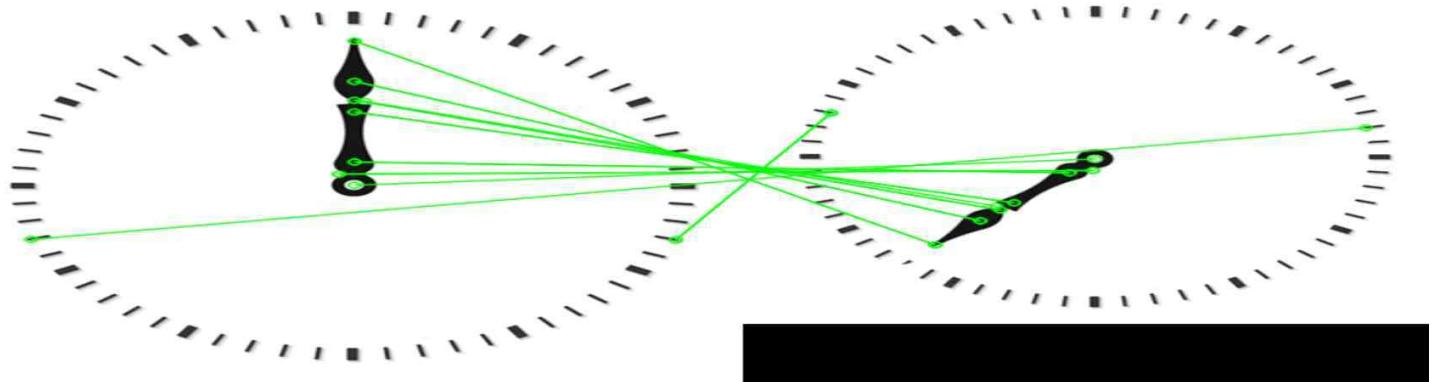
knn으로 뽑은 1위(m)가 ratio*2위(n)보다 가까우면 [good]에 들어가게 된다. 따라서 테두리 패턴은 일정하기 때문에 패턴이 유사할수록 되려 인식이 안될 확률이 높아지기 때문에 distance ratio를 적당하게 잡아주면 해결할 수 있을 것이다.

크기 문제)

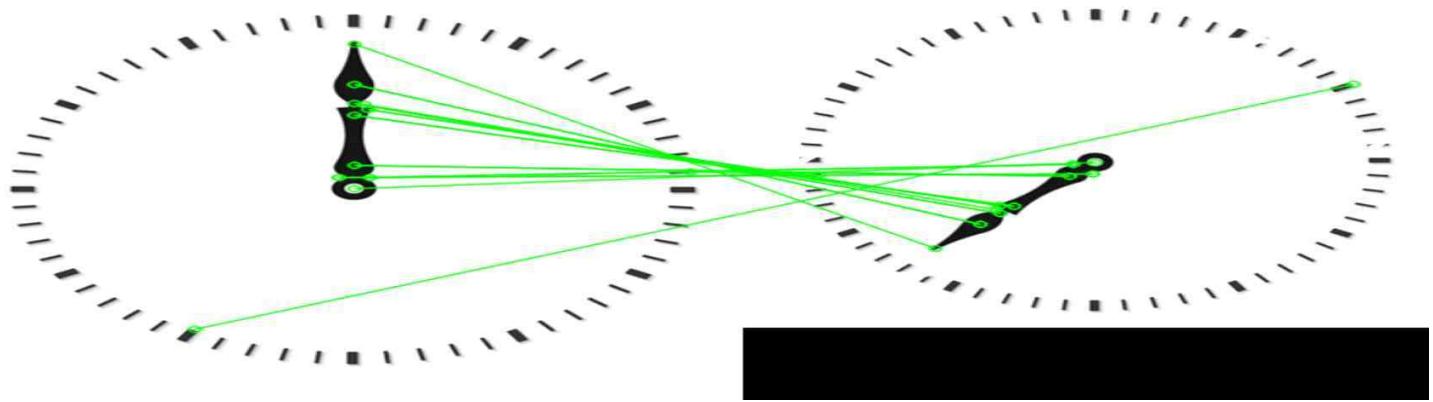
scale factor는 R행렬의 모든 요소에 똑같이 곱하는 값이기 때문에 두 영상의 scale 차이가 있더라도 $(\text{scale} \cdot \sin) / (\text{scale} \cdot \cos) = \sin / \cos$ 으로 B1상황과 같아져 scale문제 또한 해결이 가능해진다.

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

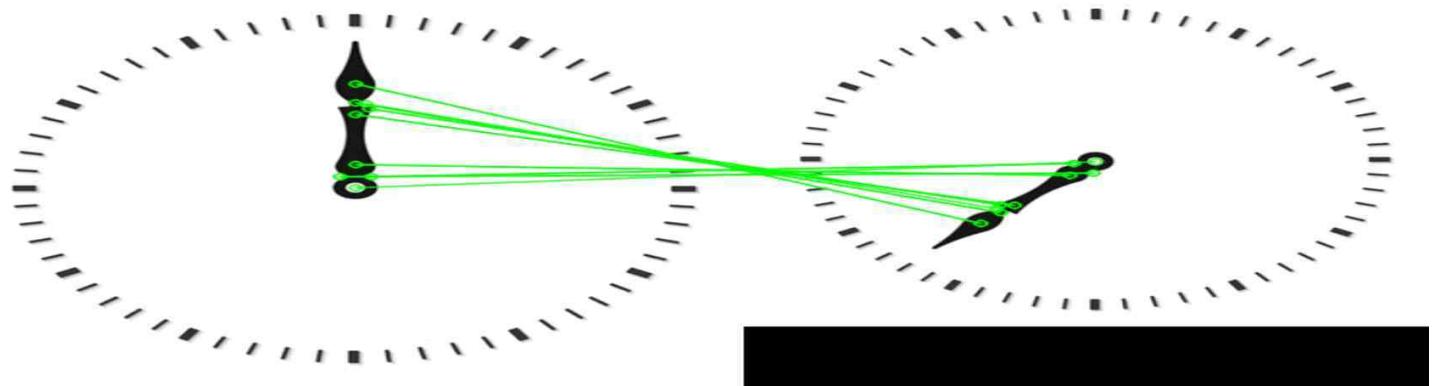
5-2 알고리즘 흐름도



distance ration=0.9



distance ration=0.7



distance ration=0.5

```

good=[]
for m,n in matches:
    if m.distance < 0.4*n.distance:
        good.append(m)

```

detect함수의 good match에 들어가는 조건문에서 0.4는 위에서 설명한 distance ratio를 나타낸다. 현재 knn의 k 값이 2개이기 때문에 matches의 요소는 m(가장 가까움), n(두번째로 가까움)으로 n*ratio값보다 가까워야 해당값을 좋은 특징점으로 선정하는 것이다. 이때 B1과제와 같은 조건인 0.9로 설정하였을 때는 일부분 테두리들이 인식되었고, 값을 줄일 수록 초침만 인식할 수 있었다.

```

img=[img1, img2, img3, img4]
I_1=img[0] #ref image
for i in range(len(img)-1):
    I0=img[i+1]

    matrix=detect(I_1, I0)
    cos=matrix[0,0]
    sin=matrix[1,0]
    theta=math.degrees(math.atan2(sin,cos))

    if theta<0:
        theta=360+theta
    else:
        theta=theta

    t_minute=theta/360*60
    print("시간=",np.int8(t_minute))

```

반복문에 train이미지를 바꾸어가며 시간을 구하는 알고리즘이다. detect함수에서 변환행렬을 구해준 후 cos 요소와 sin 요소를 이용하여 theta를 구한 후 degree로 바꾸어 준다. 만약 theta가 음수값이 뜬다면 360을 더해도 같은 각도 이기 때문에 360을 더해주고, 최종적인 분이 나오게 된다.

5-3 여러 예시에 대한 적용 결과

```

시간 = 37 분
시간 = 45 분
시간 = 52 분

```

순서대로 정답은 37분, 45분, 51분으로 큰 오차없이 잘 출력되는 것을 확인할 수 있다.