

# HW 04 – REPORT

소속 : 정보컴퓨터공학부

학번 : 201824469

이름 : 박동진

# 1. 서론

이번 과제에서는 강의에서 배운 내용을 바탕으로, RANSAC을 이용하여 여러 사진들을 이어붙인 파노라마를 제작하는 것을 목표로 한다. 파노라마를 만들기 위해서는 다음과 같은 과정을 거쳐야 한다.

1. 사진의 특징을 찾는다. 이 때, 찾은 특징들을 매칭할 수 있게 해당 특징을 설명할 수 있는 기준을 설정한다

-Feature Detection

2. 특징과 설명들을 이용하여 다른 사진들과 매칭시킨다.

-Match Features

3. homography를 RANSAC을 이용하여 찾아낸다.

4. Projection과 Homography를 이용하여 사진을 이어붙인다.

위의 과정을 코드를 작성해가면 하나씩 구현해본다. Feature Detection을 위해서는 SIFT를 사용할 것이다.

# 2. 본론

## Part1 : SIFT Key Point Matching

이번 과제에서는 Feature Description을 위해서 SIFT를 사용한다. 사진에서 요소들을 찾아낸 뒤 SIFT를 통해 각 요소를 설명하고, 주어진 설명을 통해서 다른 사진에 있는 요소들과 비교하여 같은 부분을 찾아내는 것이다.

SIFT는 과제에 첨부되어 코드로 제공되었다.

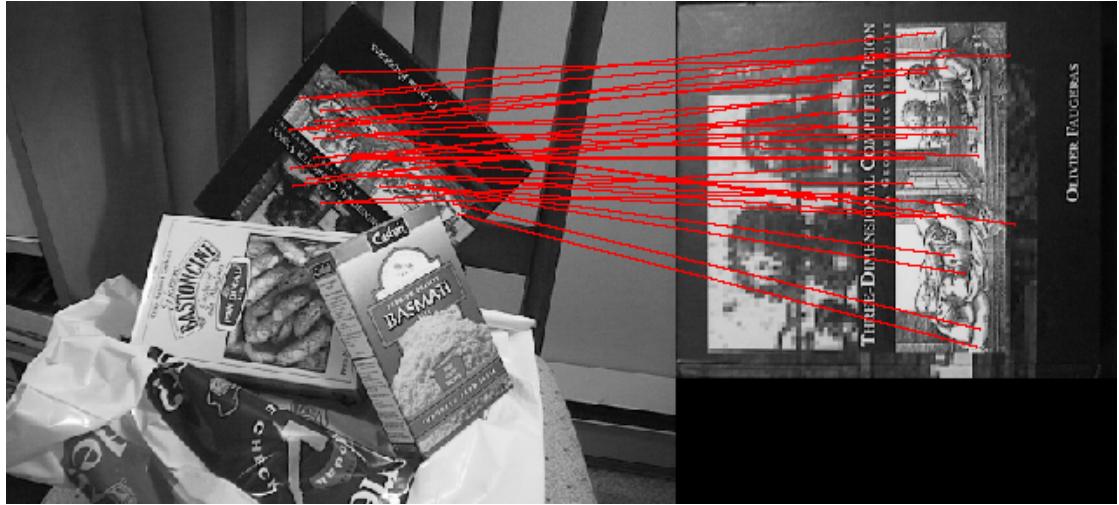
### FindBestMatch() :

`FindBestMatch()` 는 `descriptors` 두 개와 `threshold` 가 주어지면 두 개의 `descriptors` 에서 `threshold` 를 고려하여 가장 잘 매치된(가장 비슷한 descriptor쌍) 모든 요소들의 쌍을 튜플의 배열로써 반환한다.

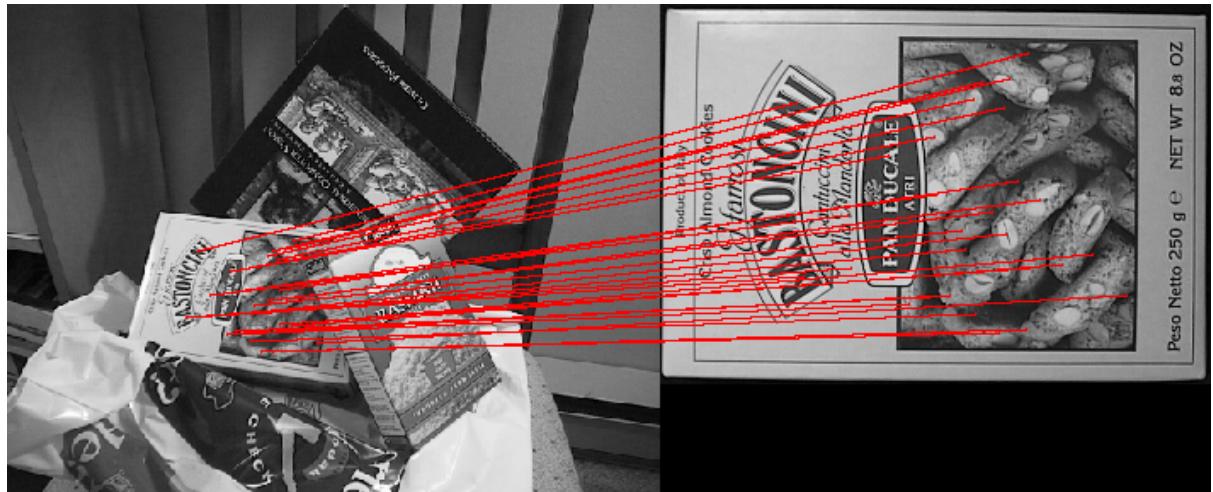
```
def FindBestMatches(descriptors1, descriptors2, threshold):
    assert isinstance(descriptors1, np.ndarray)
    assert isinstance(descriptors2, np.ndarray)
    assert isinstance(threshold, float)
    ## START
    ## the following is just a placeholder to show you the output format
    y1 = descriptors1.shape[0] # 해당 코드는 descriptor1 의 요소들을 순회하기 위해서 필요함
    y2 = descriptors2.shape[1] # 해당 코드는 descriptor2 의 요소들을 순회하기 위해서 필요함
    temp = np.zeros(y2)
    matched_pairs = []

    # 모든 요소를 비교함
    for i1 in range(y1):
        for i2 in range(y2):
            temp[i2] = math.acos(np.dot(descriptors1[i1], descriptors2[i2]))
    compare = sorted(range(len(temp)), key=lambda k: temp[k])
    if (temp[compare[0]] / temp[compare[1]]) < threshold:
        matched_pairs.append([i1, compare[0]])
```

```
## END
return matched_pairs
```



scene-book, ratio-threshold = 0.6



scene-book, ratio-threshold = 0.6

결과물은 위 사진과 같다.

### RANSACFilter():

`RANSACFilter()` 함수는 매칭된 짹들을 RANSAC을 이용하여 필터링하는 역할을 한다.

이 때, 요소의 비교에는 orient와 scale의 변화를 고려한다. random하게 뽑은 것들과 큰 차이가 없으면 inlier로 구별하고 큰 차이가 있으면 outlier로 구별한다. 이 과정을 10번 반복하여 가장 inlier가 많은 것을 최종인 solution으로 설정하고 해당 solution에 대한 inlier들을 최종적인 matching으로 결정한다.

```
def RANSACFilter(
    matched_pairs, keypoints1, keypoints2,
    orient_agreement, scale_agreement):
    """
    This function takes in `matched_pairs`, a list of matches in indices
    and return a subset of the pairs using RANSAC.
```

```

Inputs:
    matched_pairs: a list of tuples [(i, j)],
        indicating keypoints1[i] is matched
        with keypoints2[j]
    keypoints1, 2: keypoints from image 1 and image 2
        stored in np.array with shape (num_pts, 4)
        each row: row, col, scale, orientation
    *_agreement: thresholds for defining inliers, floats
Output:
    largest_set: the largest consensus set in [(i, j)] format

HINTS: the "*_agreement" definitions are well-explained
       in the assignment instructions.

"""
assert isinstance(matched_pairs, list)
assert isinstance(keypoints1, np.ndarray)
assert isinstance(keypoints2, np.ndarray)
assert isinstance(orient_agreement, float)
assert isinstance(scale_agreement, float)
## START
largest_set = []
for i in range(10): # repeat ten times
    rand = random.randrange(0, len(matched_pairs)) # generate random number
    choice = matched_pairs[rand]
    orientation = (keypoints1[choice[0]][3] - keypoints2[choice[1]][3]) % (
        2 * math.pi) # calculation first-orientation
    scale = keypoints2[choice[1]][2] / keypoints1[choice[0]][2] # calculation first-scale ratio
    temp = []
    for j in range(len(matched_pairs)): # calculate the number of all cases
        if j is not rand:
            # calculation second-orientation
            orientation_temp = (keypoints1[matched_pairs[j][0]][3] -
                keypoints2[matched_pairs[j][1]][3]) % (2 * math.pi)
            # calculation second-scale-ratio
            scale_temp = keypoints2[matched_pairs[j][1]][2] /\
                keypoints1[matched_pairs[j][0]][2]
            # check degree error +-30degree
            if (orientation - orient_agreement / 6) < orientation_temp \
                < (orientation + orient_agreement / 6):
                # check scale error +- 50%
                if scale - scale * scale_agreement < scale_temp\
                    < scale + scale * scale_agreement:
                    temp.append([i, j])
    if len(temp) > len(largest_set):
        largest_set = temp
for i in range(len(largest_set)):
    largest_set[i] = (matched_pairs[largest_set[i][1]][0],
                      matched_pairs[largest_set[i][1]][1])

## END
assert isinstance(largest_set, list)
return largest_set

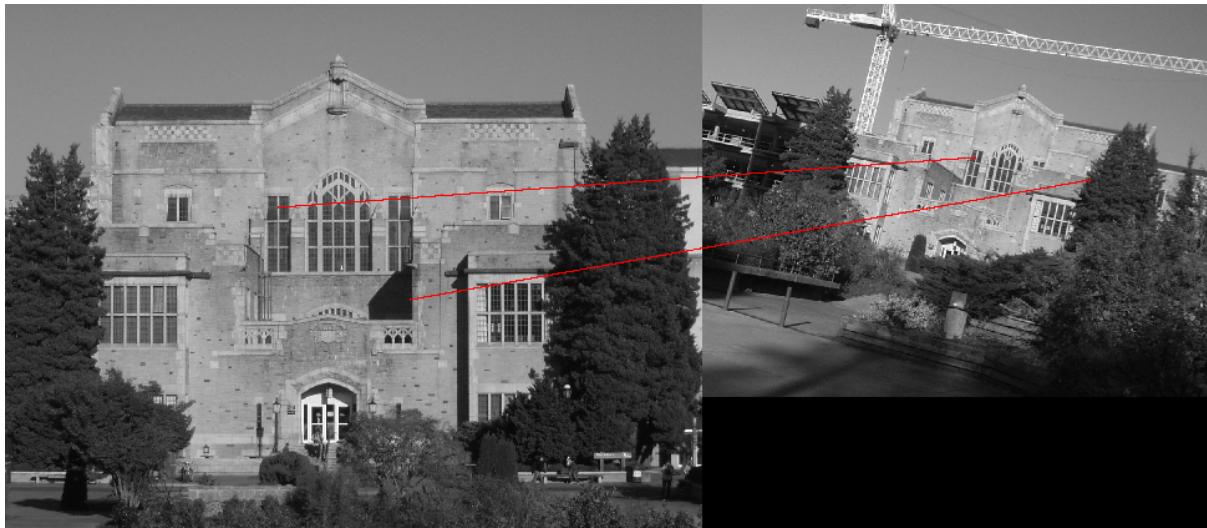
```

library를 RANSAC을 이용해 Feature Matching을 수행하면 다음과 같은 결과가 나온다. 많은 변수를 시도해 봤지만, 정확한 하나의 Matching을 도출하기 위해서는 아래와 같은 변수를 지정해 줘야 했다. 이보다 조건을 더 관대하게 설정할 경우 inlier는 발견되지 않고, outlier만이 증가하는 결과를 얻었다.

```

im = utils.MatchRANSAC(
    './data/library', './data/library2',
    ratio_thres=0.60, orient_agreement=30, scale_agreement=0.85)

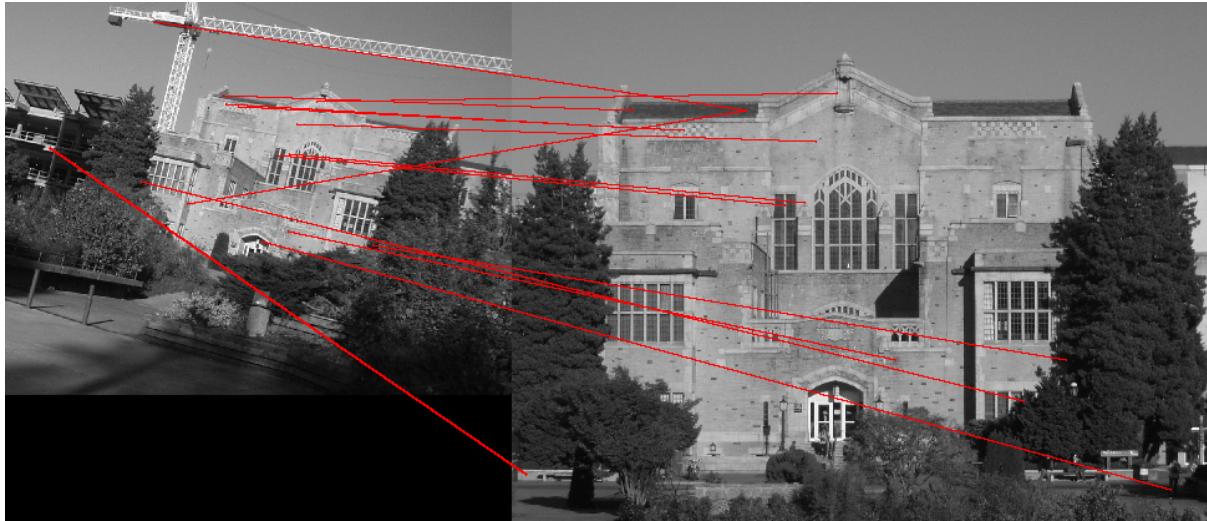
```



위의 코드의 결과물에 해당하는 이미지

흥미로운 점은 사진의 순서를 바꾸자, 더 향상된 결과가 나왔다는 점이다. 이 때, 시각적으로 확인하여 가장 적절한 각 변수는 아래의 코드와 같았다.

```
im = utils.MatchRANSAC(  
    './data/library2', './data/library',  
    ratio_thres=0.75, orient_agreement=45, scale_agreement=0.80)
```



## Part2: Panorama

이번 파트에서는 위에서 구현한 RANSAC을 바탕으로 panorama를 제작한다.

### KeypointProjection()

`KeypointProjection()` 함수는 주어진 xy좌표를 homography matrix를 이용해 다른 위치로 projection하는 기능을 수행한다.

```

def KeypointProjection(xy_points, h):
    """
    This function projects a list of points in the source image to the
    reference image using a homography matrix `h`.

    Inputs:
        xy_points: numpy array, (num_points, 2)
        h: numpy array, (3, 3), the homography matrix
    Output:
        xy_points_out: numpy array, (num_points, 2), input points in
        the reference frame.
    """
    assert isinstance(xy_points, np.ndarray)
    assert isinstance(h, np.ndarray)
    assert xy_points.shape[1] == 2
    assert h.shape == (3, 3)

    # START
    # 결과를 위한 배열을 생성
    xy_points_out = np.ones((np.shape(xy_points)[0], 3))
    # print(np.shape(xy_points_out))
    # print(np.shape(xy_points))
    # 배열의 모든 pointer에 대해서
    for i in range(len(xy_points)):
        # 각 pointer를 h와 dot연산하여 가져온다.
        xy_points_out[i] = np.dot(h, np.append(xy_points[i], 1))
        # 마지막 차원의 값에 따라서 x, y값 변경하기. 마지막 값이 0일 경우에는 대체값을 적용
        if xy_points_out[i][2] != 0:
            xy_points_out[i] = xy_points_out[i] / xy_points_out[i][2]
        else:
            xy_points_out[i] = xy_points_out[i] / 1e10
    # 마지막 축은 제외한 채 x,y point 반환
    xy_points_out = xy_points_out[:, :2]
    # END
    return xy_points_out

```

위의 구현을 완료한 후 `main_proj.py` 를 실행한 결과는 아래와 같다.



## RANSACHomography()

다음으로 구현할 `RANSACHomography()`는 RANSAC을 이용해 두 Feature들 사이의 Homography Matrix를 찾는 것이다. RANSAC을 통해 네 개의 Feature쌍을 찾은 다음, 해당 쌍을 서로 투영하는 Homography Matrix를 계산한다. 모든 주어진 Feature들에 대해서 Homography Matrix를 적용했을 때, 얼마나 많은 inlier가 발생하는지를 카운트하여 가장 많은 inlier를 발생시키는 Homography Matrix라고 판단하여 이미지를 투영시키는 것이다. 랜덤으로 추출한 4개의 Features쌍을 이용하여 Matrix를 구하는 식은 아래와 같다.

## Solving for homographies

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1 x_1 & -x'_1 y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1 x_1 & -y'_1 y_1 & -y'_1 \\ & & & & & \vdots & & & \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_n x_n & -x'_n y_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_n x_n & -y'_n y_n & -y'_n \end{bmatrix} = \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Defines a least squares problem:  $\text{minimize } \|Ah - 0\|^2$

- Since  $\mathbf{h}$  is only defined up to scale, solve for unit vector  $\hat{\mathbf{h}}$
- Solution:  $\hat{\mathbf{h}} = \text{eigenvector of } \mathbf{A}^T \mathbf{A} \text{ with smallest eigenvalue}$
- Works with 4 or more points

A-Matrix를 구한 뒤, Transpose와 곱연산한 9X9배열의 eigen value들 중 가장 작은 것에 대응하는 eigen vector가 h-matrix의 요소들이 된다.

이를 구현한 코드는 아래와 같다.

```
def RANSACHomography(xy_src, xy_ref, num_iter, tol):
    """
    Given matches of keypoint xy coordinates, perform RANSAC to obtain
    the homography matrix. At each iteration, this function randomly
    choose 4 matches from xy_src and xy_ref. Compute the homography matrix
    using the 4 matches. Project all source "xy_src" keypoints to the
    reference image. Check how many projected keypoints are within a `tol`
    radius to the corresponding xy_ref points (a.k.a. inliers). During the
    iterations, you should keep track of the iteration that yields the largest
    inlier set. After the iterations, you should use the biggest inlier set to
    compute the final homography matrix.

    Inputs:
        xy_src: a numpy array of xy coordinates, (num_matches, 2)
        xy_ref: a numpy array of xy coordinates, (num_matches, 2)
        num_iter: number of RANSAC iterations. 몇 번 랜덤하게 뽑는가?
        tol: float inlier의 기준

    Outputs:
    """

    # ... (Implementation details omitted)
```

```

    h: The final homography matrix.
"""

assert isinstance(xy_src, np.ndarray)
assert isinstance(xy_ref, np.ndarray)
assert xy_src.shape == xy_ref.shape
assert xy_src.shape[1] == 2
assert isinstance(num_iter, int)
assert isinstance(tol, (int, float))
tol = tol * 1.0

# START
# 네 개 쁙아, homogeneous 만들고, 다 투영시켜서 inlier 숫자 선별, inlier가 가장 많은 것을 선택하여 반환
h = np.identity(3)
num_inliers = 0
h_temp = np.array([])
for stage in range(num_iter):
    selected_index = random.sample(range(len(xy_src)), k=4)
    # A 배열 만들기
    A_array = []
    for i in selected_index:
        A_array.append([xy_src[i][0], xy_src[i][1], 1, 0, 0, 0,
                       -xy_src[i][0] * xy_ref[i][0], -xy_ref[i][0] * xy_src[i][1], xy_ref[i][0]])
        A_array.append([0, 0, 0, xy_src[i][0], xy_src[i][1], 1,
                       -xy_ref[i][1] * xy_src[i][0], -xy_ref[i][1] * xy_src[i][1], xy_ref[i][1]])
    A = np.array(A_array)
    # 가장 작은 eigenvalue와 그에 해당하는 eigenvector를 구한다.
    eigenvalues, eigenvectors = np.linalg.eig(np.dot(np.transpose(A), A))

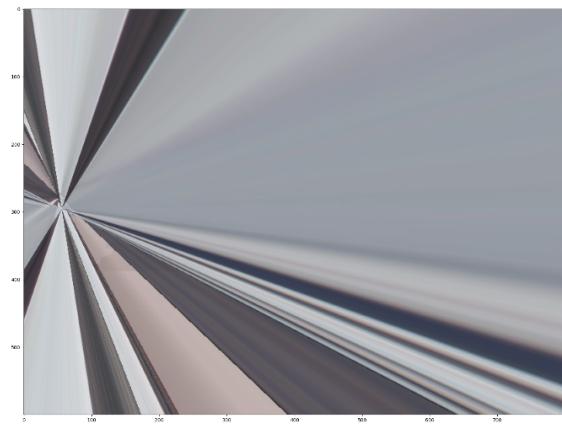
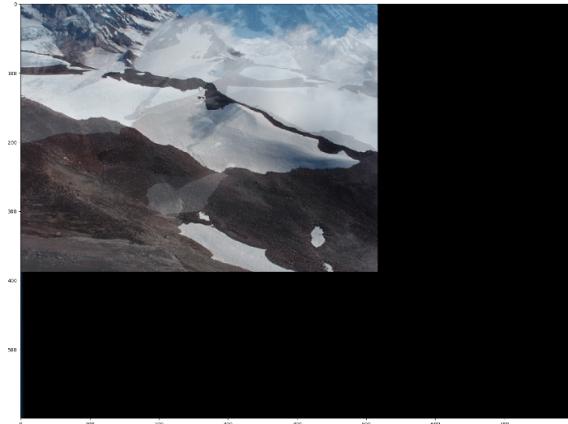
    min_value = float('inf')
    min_index = -1
    for i in range(len(eigenvalues)):
        if eigenvalues[i] < min_value:
            min_value = eigenvalues[i]
            min_index = i
    # 찾아낸 eigenvalue로 h 생성
    # print(min_index)
    h_temp = eigenvectors[min_index]
    h_temp = h_temp.reshape(3, 3)
    # print(h_temp)
    # 생성된 h를 이용해서 xy_src의 좌표들을 투영
    xy_proj = KeypointProjection(xy_src, h)

    # inlier들을 골라냄
    temp_num_inlier = 0
    for i in range(len(xy_proj)):
        # print(np.linalg.norm(xy_proj[i] - xy_ref[i]))
        if np.linalg.norm(xy_proj[i] - xy_ref[i]) < tol:
            temp_num_inlier += 1
    # 가장 적은 inlier일 경우 h를 변경 아니라면 폐기
    # print(temp_num_inlier)
    if temp_num_inlier > num_inliers:
        num_inliers = temp_num_inlier
        h = h_temp

# END
# homography의 마지막 요소는 1이어야 하므로 scaling
h = h/h[2][2]
assert isinstance(h, np.ndarray)
assert h.shape == (3, 3)

# h, _ = cv2.findHomography(xy_src, xy_ref)
return h

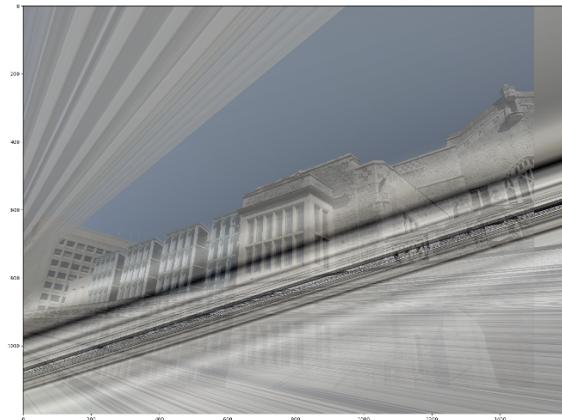
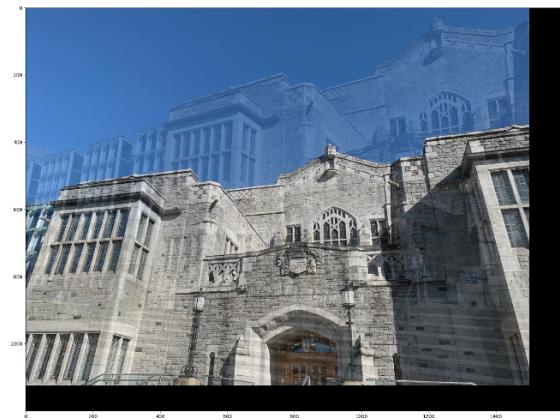
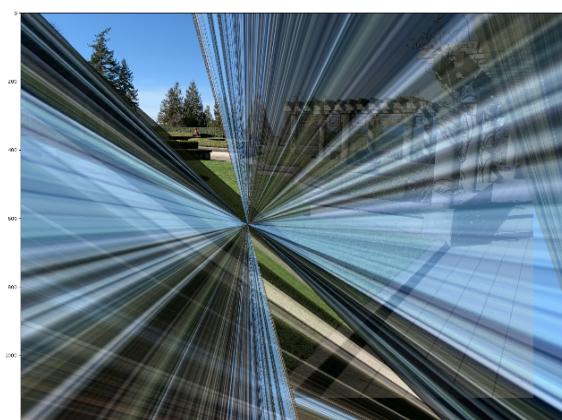
```



결과는 왼 쪽과 같았다. RANSAC으로 찾아낸 Homography를 출력해 보면 값이 나옴은 확인되었지만, 투영이 제대로 되지 않은 것이다. 이유는 확인할 수 없었다. cv의 함수

```
h, _ = cv2.findHomography(xy_src, xy_ref)
```

를 이용했을 때, 결과물은 더욱 난해하게 나왔다. 이유는 확인할 수 없었다. 다른 이미지에 대해서도 비슷한 경향을 띠었다.



왼쪽은 직접 작성한 homography를 적용한 모습 오른쪽은 cv의 함수를 이용하여 구한 homography를 적용한 모습이다.

### 3. 결론

파노라마 기능을 구현하기 위해서는 아래의 절차를 따라야 한다.

1. Feature Detection
2. Match Features
3. homography를 RANSAC을 이용하여 찾아낸다.
4. Projection과 Homography를 이용하여 사진을 이어붙인다.

1번과 2번의 기능과 projection의 기능은 구현되었지만 homography와 관련된 기능은 구현에 실패했다. RANSAC을 통해 만들어진 homography를 출력하여 identity matrix가 아닌 값이 존재하는 matrix가 계산되었음을 확인했지만, 적절한 투영은 이루어지지 않았다.