

## SIFT

### 1. The purpose of the code

SIFT descriptor 구현을 목적으로 코드를 작성하였다. Keypoint와 descriptor를 찾는 과정은 opencv에서 제공하는 함수를 이용하여 구현되었다.

### 2. The explanation of the code

이 과제에서는 세가지 단계로 나누어서 SIFT descriptor를 구현한다.

Case1) Nearest neighbor 알고리즘을 이용하여 feature matching을 한다. 이 때 threshold는 사용하지 않았다.

```
int nearestNeighbor(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors) {  
    int neighbor = -1;  
    double minDist = 1e6;  
  
    for (int i = 0; i < descriptors.rows; i++) {  
        Mat v = descriptors.row(i); // each row of descriptor  
  
        //  
        // Fill the code  
        //  
        double dist = euclidDistance(vec, v);  
        if (dist < minDist) {  
            minDist = dist;  
            neighbor = i;  
        }  
    }  
  
    return neighbor;  
}
```

nearestNeighbor() 함수는 descriptors에서 vec와 가장 유사한 행을 찾고, 그 행의 index값을 반환하는 함수이다. 두 행의 유사도를 판단하기 위하여 euclidDistance()함수를 이용하였다. euclidDistance() 함수는 두개의 행이 얼마나 다른지, 그 정도를 값으로 반환해주는 함수이다. 따라서 가장 유사한 행을 찾기 위해서는 euclidDistance() 함수가 반환한 값인 dist가 가장 작은 행을 찾아야 한다. if문을 이용하여 dist가 작은 값을 가지도록 하는 descriptor의 행 i를 저장하였고 이 i값이 nearestNeighbor() 함수의 최종 결과값이 되는 것이다.

Case2) case1에서 현재 matching하고 싶은 feature 행과 가장 유사한 행의 index를 찾았다. 이 index를 바로 feature matching하지 않고 cross-checking을 하여 이 index가 적합한 값인지를 판단한다.

```
// Refine matching points using cross-checking  
if (crossCheck) {  
    //  
    // Fill the code  
    //  
    Mat desc2 = descriptors2.row(nn);  
    int l = nearestNeighbor(desc2, keypoints1, descriptors1);  
    if (l == nn){  
        // matched  
    }  
    else  
        continue;  
}
```

Cross-checking을 설명하기 위하여 f1~f5와 g1~g4가 있다고 설정하겠다. 만약 우리가 f1에 대하여 feature matching을 하고자 nearest neighbor 알고리즘을 사용하였다. 그 결과로 g4가 가장 유사하다고 나왔을 때, 과연 g4에 대하여 nearest neighbor 알고리즘을 사용한다면 f1이 결과로 나올 것인가에 대한 문제이다. 만약 f1은 g4를 가리키는데 g4는 f1이 아닌 다른 feature를 가리킨다면 이 둘을 matching하는 것은 적절하지 않다. Cross-checking은 서로가 서로를 유사하다고 이야기하는 지 확인해주는 작업이다.

따라서 descriptors1 i행에 대한 결과로 나온 nn행이 반대로 했을 때도 서로가 matching이 되는지를 확인하기 위하여 descriptors2 nn행을 기준으로 nearest neighbor 알고리즘을 이용하였다. 그 결과인 l값이 i와 같다면 이는 잘 matching된 것이고, 그렇지 않다면 잘못 matching된 것이므로 i행 feature는 최종 결과에서 제외된다.

Case3) cross-checking과 ratio-based thresholding을 이용하여 보다 정확한 feature matching을 구현한다.

```
double nearestNeighborforRB(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors) {
    int neighbor = -1;
    double first = 1e6;
    double second = 1e6;

    for (int i = 0; i < descriptors.rows; i++) {
        Mat v = descriptors.row(i); // each row of descriptor

        //
        // Fill the code
        //
        double dist = euclidDistance(vec, v);
        if (dist < first) {
            second = first;
            first = dist;
        }
        else if((second > dist) && (dist > first)) {
            second = dist;
        }
    }

    double ratio = first / second;
    return ratio;
}
```

Ratio based thresholding 방법을 적용하기 위해서는 f1에 대한 feature matching을 한다고 가정했을 때 첫번째로 가장 유사한 feature vector와 f1의 차이 값, 두번째로 가장 유사한 feature vector와 f1의 차이 값이 필요하다. 첫번째로 유사한 값과 f1의 차를 두번째로 유사한 값과 f1의 차로 나누었을 때의 비율을 이용해야 하기 때문이다. 이 두개의 값을 구하기 위하여 if문을 사용하였고, 이 함수의 결과 값으로 두 값의 비율을 반환하도록 코드를 작성하였다.

```
// Refine matching points using ratio_based thresholding
if (ratio_threshold) {
    //
    // Fill the code
    //
    double ratio = nearestNeighborforRB(desc1, keypoints2, descriptors2);
    if (ratio < RATIO_THR) {}
    else
        continue;
}
```

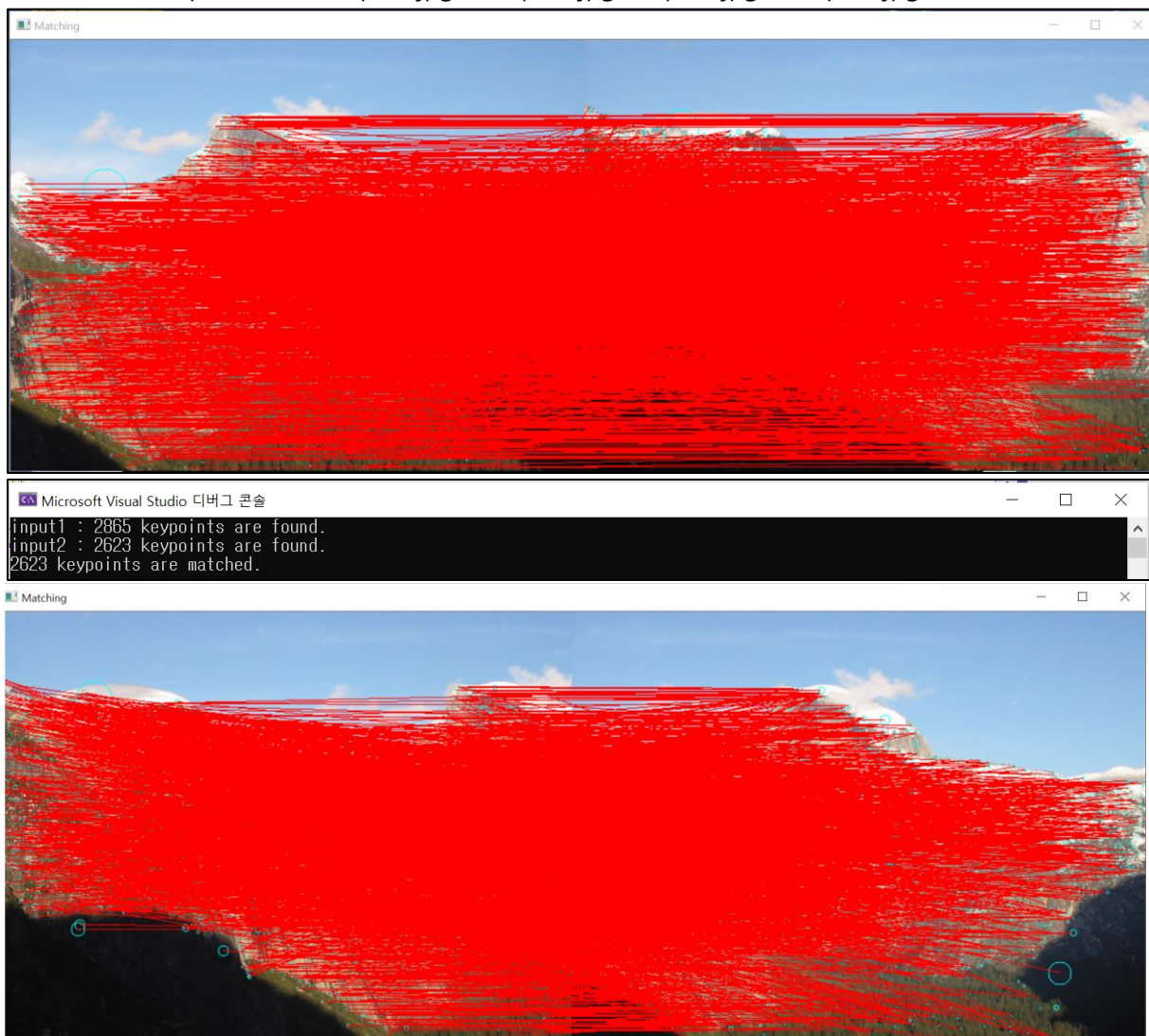
위에서 구한 비율이 특정 threshold보다 작으면 첫번째로 가장 유사한 feature vector와 두번째로 유사한 feature vector와의 차이가 크다는 것을 의미한다. 즉 작은 차이로 인하여 두번째로 유사한 feature vector가 아쉽게 선택되지 못한 것이 아니라는 이야기이다. 만약 이 조건을 만족한다면 feature matching이 되고, 그렇지 않으면 해당 feature는 최종 결과에서 제외된다.

이 코드 설명은 input2.jpg에서 input1.jpg로 feature matching을 하는 것에 대한 내용이다. 만약 input1.jpg에서 input2.jpg로 feature matching을 하는 경우에는 다음과 같이 입력 이미지의 순서를 바꿔주면 된다.

```
Mat input1 = imread("input2.jpg", CV_LOAD_IMAGE_COLOR);  
Mat input2 = imread("input1.jpg", CV_LOAD_IMAGE_COLOR);
```

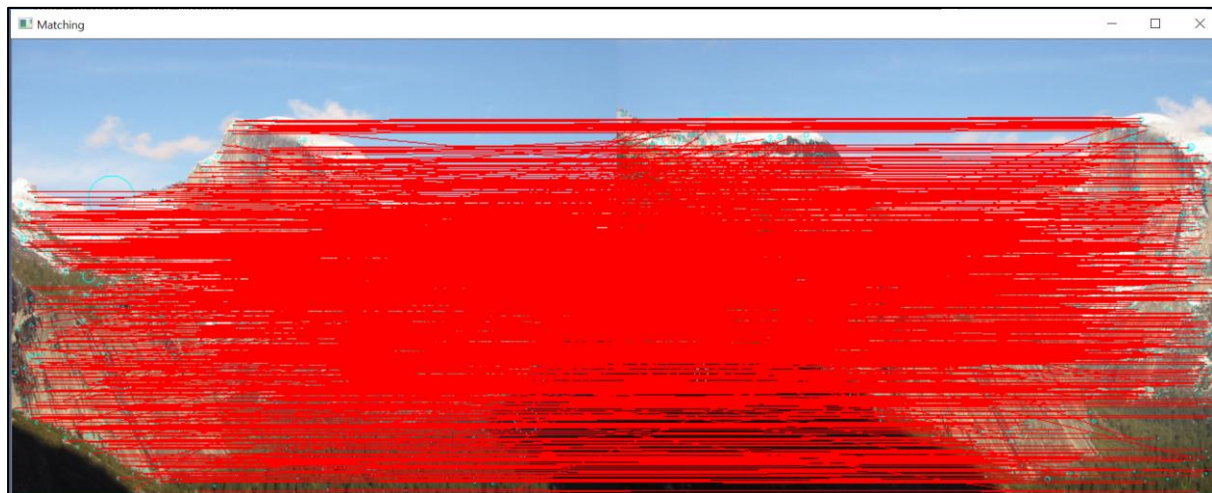
### 3. The output of the code

a) Output of case1(input2.jpg -> input1.jpg / input1.jpg -> input2.jpg)

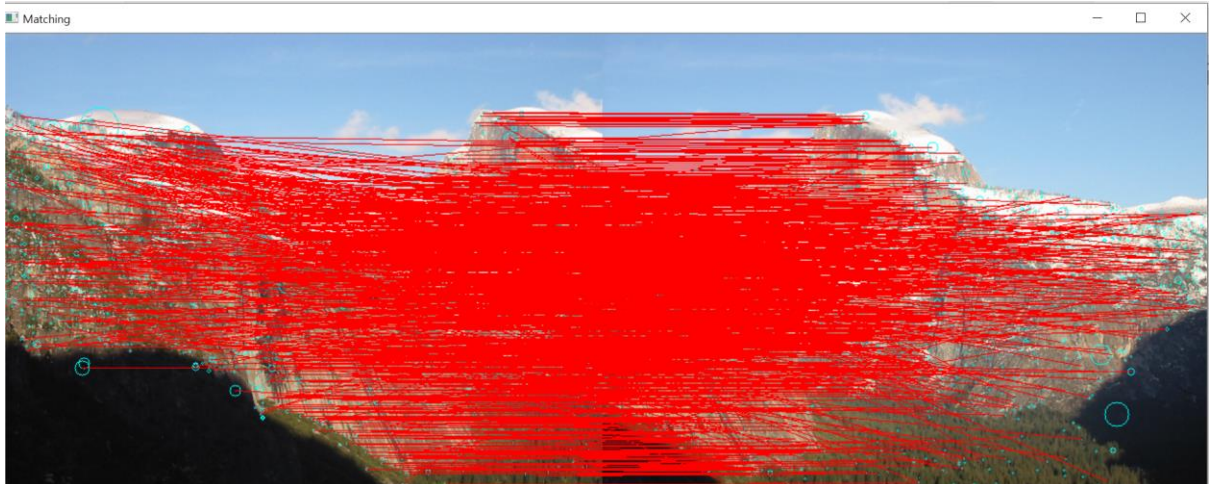


```
C:\Users\W100ji\source\repos\Project1\wx64\Debug\Project1.exe
input1 : 2623 keypoints are found.
input2 : 2865 keypoints are found.
2865 keypoints are matched.
```

b) Output of case2(input2.jpg -> input1.jpg / input1.jpg -> input2.jpg)



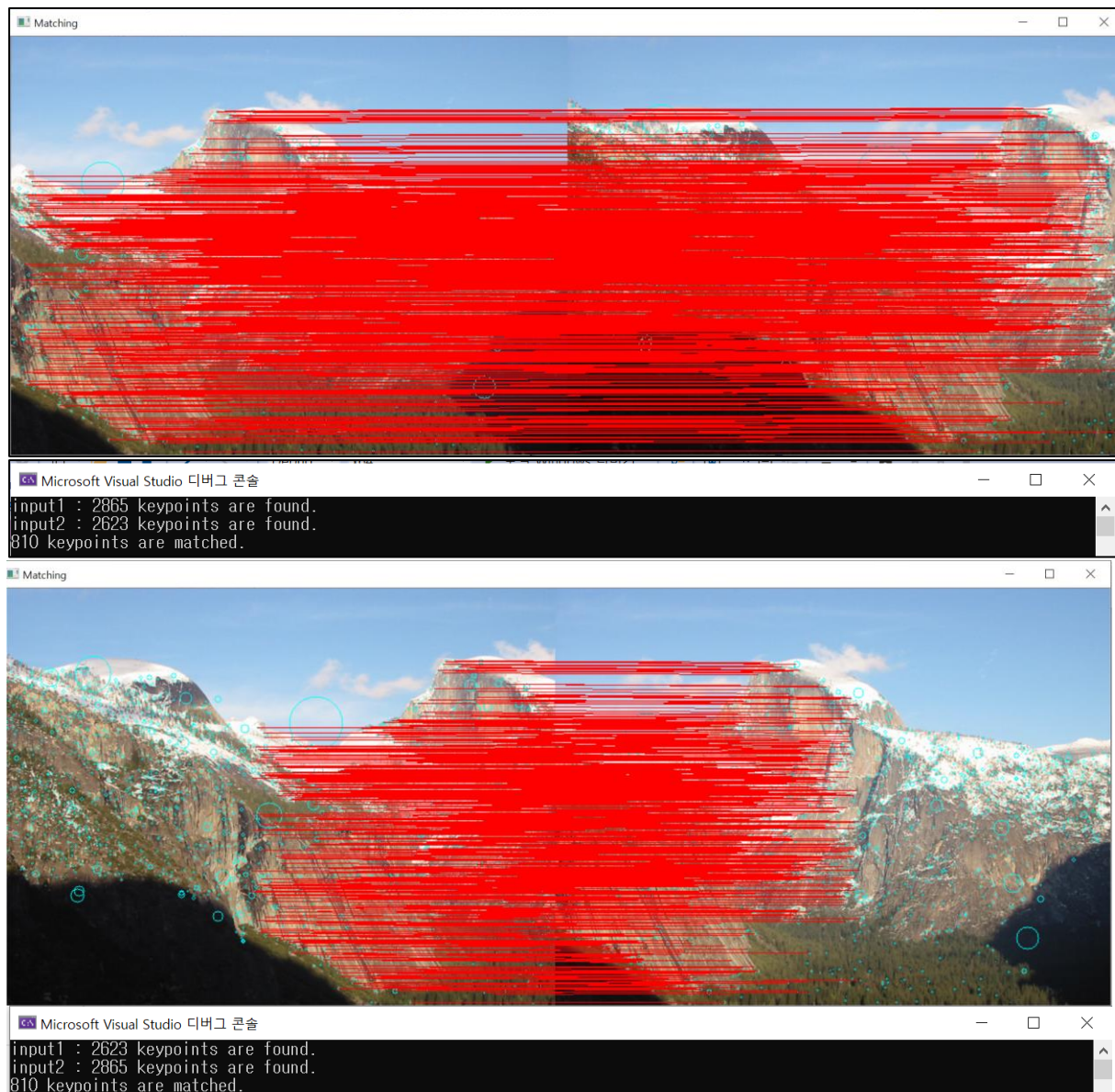
```
C:\Users\W100ji\source\repos\osp_assign2\wx64\Debug\osp_assign2.exe
input1 : 2865 keypoints are found.
input2 : 2623 keypoints are found.
1354 keypoints are matched.
```



```
C:\Users\W100ji\source\repos\Project1\wx64\Debug\Project1.exe
input1 : 2623 keypoints are found.
input2 : 2865 keypoints are found.
1354 keypoints are matched.
```

c) Output of case3 (input2.jpg -> input1.jpg / input1.jpg -> input2.jpg)





#### 4. Analysis

Case1에서 case2, case2에서 case3로 갈수록 부정확한 feature matching이 사라지면서 keypoints의 개수가 줄어든 것을 확인할 수 있다. 또한 결과 이미지에서도 feature matching을 나타내기 위하여 그려진 선의 수가 줄어든 것을 파악할 수 있다.

### SURF

#### 1. 코드 실행

SURF 코드를 실행하고자 [https://docs.opencv.org/3.4.1/d7/dff/tutorial\\_feature\\_homography.html](https://docs.opencv.org/3.4.1/d7/dff/tutorial_feature_homography.html)에서 코드를 복사하였다. 다른 헤더파일에 대해서는 문제가 없었으나, "opencv2/xfeatures2d.hpp"를 include하는 것에 문제가 있었다. Opencv2는 기본적으로 xfeatures2d가 제공된다고 알고 있으

나, "No such file or directory" 에러가 발생하였고 이를 해결하기 위하여 opencv github에서 xfeatures2d.hpp 파일이 포함되어 있는 opencv\_contrib를 다운 받았다. Cmake를 이용하여 Opencv\_contrib를 사용할 수 있도록 설정을 하였고 visual studio에서 CMakeTargets 폴더의 INSTALL 파일 또한 빌드를 마쳤다. 그러나 계속해서 xfeatures2d.hpp 파일을 읽을 수 없다는 에러가 발생하여서 코드 결과 이미지로 opencv에서 제공하는 이미지를 다운받아 사용하였다.

python으로 해당 코드를 돌려보았다. 코드 분석에 있는 코드는 visual studio c++코드이며, 결과 이미지는 anaconda python으로 돌린 결과이다.

python에서 돌리기 위하여 opencv-python과 opencv-contrib-python을 3.4.2.16 버전으로 다운받아 사용하였다.

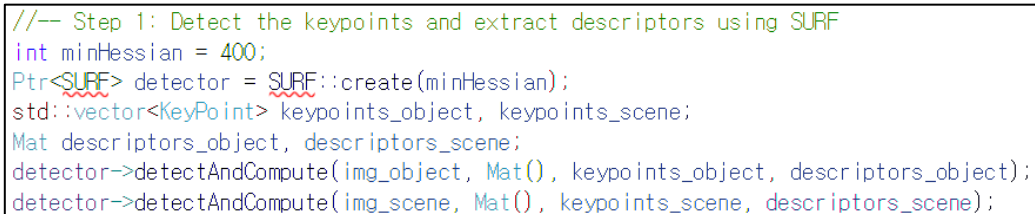


```
C:\> 명령 프롬프트
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\#100ji>pip list | findstr opencv
opencv-contrib-python          3.4.2.16
opencv-python                  3.4.2.16
```

## 2. 코드 분석

SURF는 Speeded up robust features의 약자로 sift보다 빠른 속도로 feature matching을 한다.



```
//-- Step 1: Detect the keypoints and extract descriptors using SURF
int minHessian = 400;
Ptr<SURF> detector = SURF::create(minHessian);
std::vector<KeyPoint> keypoints_object, keypoints_scene;
Mat descriptors_object, descriptors_scene;
detector->detectAndCompute(img_object, Mat(), keypoints_object, descriptors_object);
detector->detectAndCompute(img_scene, Mat(), keypoints_scene, descriptors_scene);
```

SURF는 Hessian matrix를 이용하여 keypoint를 찾는다. 이를 위해 hessian matrix를 생성하는 코드가 step1에 작성되어 있다. 생성된 hessian matrix로 scene과 object 각각의 keypoint를 찾고 descriptor를 만든다.

```

//-- Step 2: Matching descriptor vectors using FLANN matcher
FlannBasedMatcher matcher;
std::vector< DMatch > matches;
matcher.match(descriptors_object, descriptors_scene, matches);
double max_dist = 0; double min_dist = 100;
//-- Quick calculation of max and min distances between keypoints
for (int i = 0; i < descriptors_object.rows; i++)
{
    double dist = matches[i].distance;
    if (dist < min_dist) min_dist = dist;
    if (dist > max_dist) max_dist = dist;
}
printf("-- Max dist : %f \n", max_dist);
printf("-- Min dist : %f \n", min_dist);
//-- Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
std::vector< DMatch > good_matches;
for (int i = 0; i < descriptors_object.rows; i++)
{
    if (matches[i].distance <= 3 * min_dist)
    {
        good_matches.push_back(matches[i]);
    }
}

```

Descriptors\_object와 descriptors\_scene 이 둘의 feature matching으로 FLANN을 사용한다. FLANN이란 Fast library for approximate nearest neighbors의 약자로 빠른 속도로 feature matching을 해주는 알고리즘이다. Matching이 완료된 결과는 matches에 담긴다. Matches에 있는 feature matching이 적절하게 되었는지 확인해서 좋은 feature matching만을 good\_matches에 넘겨서 보다 정확하게 matching 되도록 한다.

```

Mat img_matches;
drawMatches(img_object, keypoints_object, img_scene, keypoints_scene,
    good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
    std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
//-- Localize the object
std::vector<Point2f> obj;
std::vector<Point2f> scene;
for (size_t i = 0; i < good_matches.size(); i++)
{
    //-- Get the keypoints from the good matches
    obj.push_back(keypoints_object[good_matches[i].queryIdx].pt);
    scene.push_back(keypoints_scene[good_matches[i].trainIdx].pt);
}
Mat H = findHomography(obj, scene, RANSAC);
//-- Get the corners from the image_1 ( the object to be "detected" )
std::vector<Point2f> obj_corners(4);
obj_corners[0] = cvPoint(0, 0); obj_corners[1] = cvPoint(img_object.cols, 0);
obj_corners[2] = cvPoint(img_object.cols, img_object.rows); obj_corners[3] = cvPoint(0, img_object.rows);
std::vector<Point2f> scene_corners(4);
perspectiveTransform(obj_corners, scene_corners, H);

```

Good\_matches에 있는 matching값들을 obj와 scene에 넣어서 matching을 시켜준다.

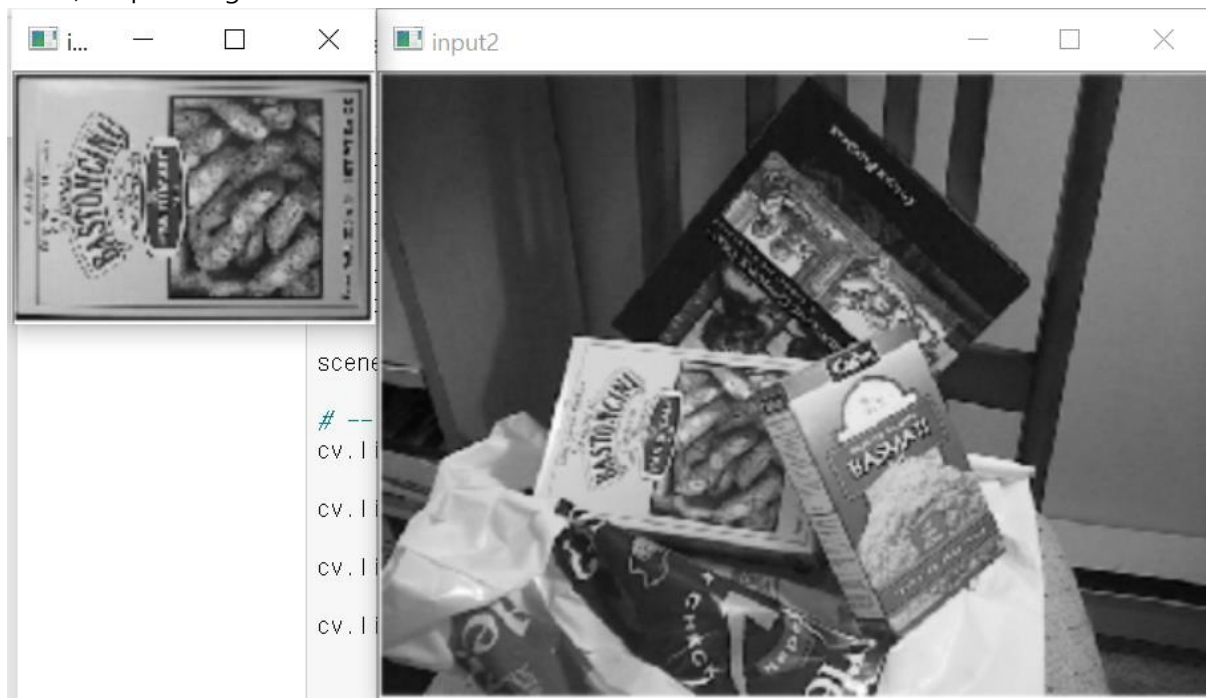
Scene에 obj가 어떻게 어떤 형태로 위치하고 있는지 그 모양을 찾기 위하여 findHomography()함수를 이용한다. 이렇게 구한 H 행렬과 obj\_corners를 perspectiveTransform()함수에 인자로 입력을 하면 scene의 어느 위치에 object가 있는지를 찾아서 각 corner값을 scene\_corners에 결과를 담아준다.

```
//-- Draw lines between the corners (the mapped object in the scene - image_2)
line(img_matches, scene_corners[0] + Point2f(img_object.cols, 0), scene_corners[1] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
line(img_matches, scene_corners[1] + Point2f(img_object.cols, 0), scene_corners[2] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
line(img_matches, scene_corners[2] + Point2f(img_object.cols, 0), scene_corners[3] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
line(img_matches, scene_corners[3] + Point2f(img_object.cols, 0), scene_corners[0] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
//-- Show detected matches
imshow("Good Matches & Object detection", img_matches);
```

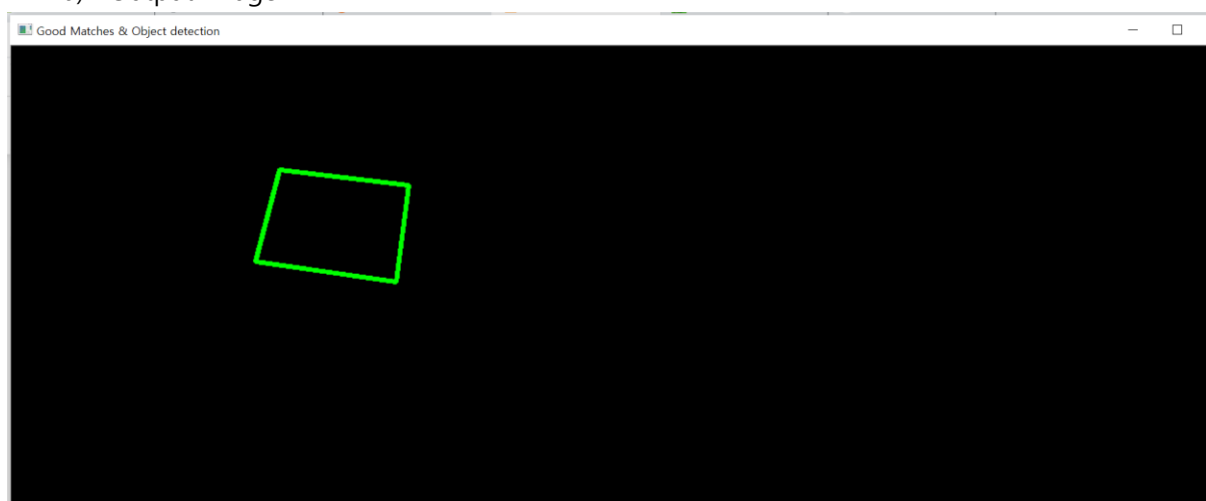
Scene\_corners에는 scene에서 object가 어느 위치에 있는지 정보를 가지고 있으므로 이를 이용하여 scene 속의 object의 corner를 찾을 수 있게 되고, edge에 line을 그려주는 코드이다.

### 3. 코드 결과

#### a) Input image



#### b) Output image



line함수를 이용한 결과로 scene안의 object edge에 초록색 선이 그려진 것을 확인할 수 있다.