Technical report-assignment07

컴퓨터공학 류다현(2376087)

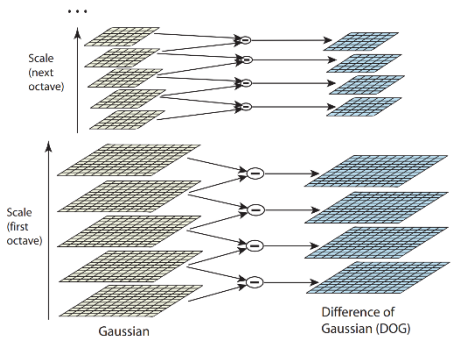
SIFT.cpp

SIFT는 스케일과 회전에 불변하는 특징을 추출하므로 이미지 크기가 달라지거나 이미지가 회전하더라도 이미지 간의 대응점을 찾을 수 있다. SIFT의 작동 과정은 다음과 같다.

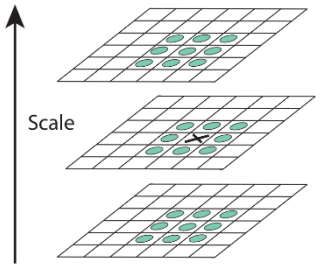
Scale-space extrema detection

스케일 공간에서 극점을 검출하는 단계로, cascade filtering approach를 사용해 모든 스케일에서 안정적인 특징을 추출한다.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$



위와 같이 원본 이미지 I(x, y)에 σ를 표준편차로 사용하는 가우시안 필터를 적용한 것을 한 레이어로 하고, 표준편차를 달리하는 여러 가우시안 필터들로 여러 레이어를 생성하여 이들의 묶음을 하나의 옥타브(octave)라 한다. 다음 옥타브는 1/2로 다운샘플링한 이미지로 구성되며, 마찬가지로 이미지에 여러 표준편차의 가우시안 필터를 적용한다.
이후 각 옥타브 내에서 인접한 두 레이어 $L(x, y, \sigma)$와 $L(x, y, k\sigma)$의 차이를 계산해 DoG인 $D(x, y, \sigma) = \big(G(x, y, k\sigma) - G(x, y, \sigma)\big) * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma)$를 얻는다.



이후 각 DoG 픽셀 $D(x, y, \sigma)$에 대해 D를 같은 레이어의 인접한 8개 픽셀과 상하 레이어의 각 9개 픽셀, 총 26개의 픽셀과 비교한다. D가 26개의 이웃 픽셀들 중 가장 크거나(local maxima) 가장 작은 경우에만(local minima) D를 선택하여 해당 픽셀을 키포인트 후보로 판단한다.

Key point localization

키포인트 후보를 찾은 뒤 위치와 스케일 등을 인근 데이터에 세밀하게 보정하고, 이 정보를 통해 대비가 낮거나 에지를 따라 잘 localize되지 않은 포인트를 제외할 수 있다.

$$D(\mathbb{x}) = D + \frac{\partial D^T}{\partial \mathbb{x}}\mathbb{x} + \frac{1}{2}\mathbb{x}^T\frac{\partial^2 D}{\partial \mathbb{x}^2}\mathbb{x}$$

위와 같이 2차항까지 테일러 전개하여 함수를 근사하면 극값의 위치 $\hat{\mathbb{x}} = \frac{\partial^2 D^{-1}}{\partial \mathbb{x}^2}\frac{\partial D}{\partial \mathbb{x}}$를 소수점 단위로 구할 수 있으며, 보정된 위치에서의 DoG 함수값이 threshold이하일 경우 대비가 낮은 unstable한 극점으로 간주하여 제거한다.
이후 에지 응답을 제거하기 위해 Hessian 행렬 $H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$를 사용해 에지, 코너, flat 영역을 판별한다. Harris 코너 검출기에서와 동일하게 H의 두 고유값이 모두 크면 코너, 하나만 크면 에지, 둘 다 작으

면 flat한 영역으로 간주하며, $r = \frac{Trace(H)^2}{Det(H)}$이 threshold보다 크면 해당 극값을 제외한다.

1. **Orientation assignment & Descriptor Construction**

   이미지에 16*16 크기의 윈도우를 적용하고, L(x, y)=G(x, y,)*I(x, y)에 대해 magnitude M(x, y)와 angle (x, y)를 계산한다. 이후 M(x, y)가 threshold 이하인 노이즈에 민감한 약한 에지를 제거한다. 이후 각 샘플로 히스토그램을 생성해 윈도우 내에서 가장 많이 등장한 방향을 지배적인 방향으로 결정하고, 이 정보들로 디스크립터를 생성한다.

SIFT는 위 원리에 따라 키포인트와 디스크립터를 검출하며,

```
sift->detectAndCompute(input1_gray, noArray(), keypoints1, descriptors1);
sift->detectAndCompute(input2_gray, noArray(), keypoints2, descriptors2);
```

와 같이 opencv 제공 SIFT 라이브러리를 사용해 수행한 결과 keypoints1, descriptors1 배열에는 input1_gray의 키포인트와 descriptor가, keypoints2, descriptors2 배열에는 input2_gray의 키포인트와 descriptor가 저장된다. 이후 findPairs() 함수를 호출해 이들 배열에 대해 서로 대응되는 점을 찾는다(keypoint matching).

1. double euclidDistance(const Mat& vec1, const Mat& vec2)

```
double euclidDistance(const Mat& vec1, const Mat& vec2) {
	double sum = 0.0;
	int dim = vec1.cols;
	for (int i = 0; i < dim; i++) {
		sum += (vec1.at<float>(0, i) - vec2.at<float>(0, i)) * (vec1.at<float>(0, i) -
		vec2.at<float>(0, i));
	}

	return sqrt(sum);
}
```

euclidDistance() 함수는 원소 개수가 dim인 1*dim 크기의 Mat형 vec1, vec2를 전달받아 0부터 (dim-1)까지 for문을 순회하며 vec1(0, i)와 vec2(0, i) 간 차이의 제곱을 sum에 누적하고, for문이 종료된 후 sum의 제곱근을 구함으로써 유클리드 거리를 계산해 반환한다.

2. int nearestNeighbor(const Mat& vec, const vector<KeyPoint>& keypoints, const Mat& descriptors)

```
int nearestNeighbor(const Mat& vec, const vector<KeyPoint>& keypoints, const 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

		double dist = euclidDistance(vec, v);
		if (dist < minDist) {
			minDist = dist;
			neighbor = i;
		}
	}

	return neighbor;
}
```

nearestNeighbor() 함수는 벡터 vec과 키포인트 keypoints, keypoints에 대응하는 디스크립터 descriptors

를 입력으로 받아 디스크립터 중 vec과의 유클리드 거리가 가장 작은, 즉 vec과 가장 유사한 원소를 반환한다.

최근접 이웃을 저장할 neighbor 변수는 -1, 최소 거리 minDist 변수는 매우 큰 값으로 초기화한다. 이후 0부터 (descriptors.rows-1)까지 vec과 descriptors의 모든 원소를 탐색하며, 현재 원소 descriptors.row(i)와 vec의 유클리드 거리를 계산해 dist에 저장하고 dist가 minDist보다 작은 경우 minDist를 dist로 갱신한 뒤 neighbor 또한 현재 인덱스 i로 갱신한다.

3. **void findPairs(vector<KeyPoint>& keypoints1, Mat& descriptors1, vector<KeyPoint>& keypoints2, Mat& descriptors2, vector<Point2f>& srcPoints, vector<Point2f>& dstPoints, bool crossCheck, bool ratio_threshold)**

findPairs() 함수는 main에서 계산한 input_gray1, input_gray2의 키포인트와 디스크립터를 입력받고 이들에 대해 대응점을 계산한 것을 srcPoints, dstPoints에 저장한다. ratio_threshold 변수와 crossCheck 변수는 입력받은 키포인트들에 대해 ratio-based thresholding과 cross-checking을 수행할지 지정하는 변수이다.

```
KeyPoint pt1 = keypoints1[i];
Mat desc1 = descriptors1.row(i);
int nn = nearestNeighbor(desc1, keypoints2, descriptors2);
```

디스크립터의 모든 원소에 대해 for문을 사용해 순회하며, pt1은 input_gray1 이미지의 i번째 키포인트를, desc1은 input_gray1 이미지의 i번째 디스크립터를 저장한다. 이후 nearestNeighbor() 함수를 호출해 i번째 디스크립터와 가장 가까운 input_gray2 디스크립터 인덱스를 계산하고 nn에 저장한다.

```
if (ratio_threshold) {

    double closest = euclidDistance(desc1, descriptors2.row(nn));
    double secondClosest = 1e6;
    for (int j = 0; j < descriptors2.rows; j++) {
            if (j == nn) {
                    continue;
            }
            double temp = euclidDistance(desc1, descriptors2.row(j));
            if (temp < secondClosest) secondClosest = temp;
    }
    if (closest > RATIO_THR * secondClosest) {
            continue;
    }
}
```

ratio_threshold가 true일 경우 ratio based thresholding을 수행해 적절하지 않은 대응점을 제거한다. 단순히 가장 가까운 특징점을 사용한 global thresholding하는 것은 부적절할 수 있으므로, 최근접 이웃과 그 다음으로 가까운 이웃을 모두 고려해 대응점을 결정하는 방식을 사용한다.

이를 위해 가장 가까운 지점을 저장할 closest 변수를 선언해 desc1과 가장 가까운 이웃으로 초기화한다. 두 번째로 가까운 이웃을 저장할 secondClosest 변수는 매우 큰 값으로 초기화하며, descriptor2의 모든 원소를 순회하며 인덱스가 nn, 즉 최근접 이웃인 경우를 건너뛰고 나머지 중 가장 작은 거리를 두 번째 최솟값으로 갱신한다.

for문이 종료되면 closest가 secondClosest의 RATIO_THR배보다 크면 해당 매칭을 제외한다.

closest/secondClosest가 threshold보다 작은 경우, 즉 두 번째로 가까운 이웃까지의 거리보다 최근접 이웃까지의 거리가 유의미하게 작은 경우에만 대응점으로 간주하는데, 이는 올바른 대응점이 가장 가까운 틀린 대응점보다 확실히 가까워야만 올바르게 매칭된 것이기 때문이다.

```
if (crossCheck) {
     Mat desc2 = descriptors2.row(nn);
     int rev = nearestNeighbor(desc2, keypoints1, descriptors1);
     if (rev != i) {
               continue;
     }
}
```

crossCheck가 true일 경우 두 디스크립터에 대해 교차 검증하여 매칭의 정확도를 높인다. 최근접 이웃의 디스크립터를 desc2에 저장하고, 해당 위치에서 다시 원본 디스크립터를 탐색하여 desc2의 최근접 이웃 인덱스를 구해 rev에 저장한다. rev가 i와 다르다면 두 디스크립터가 서로 최단 거리의 이웃이 아님을 의미하므로 해당 매칭을 제외하고, 같다면 올바른 매칭으로 간주한다.

4. int main()

input1, input2에 매칭을 진행할 두 이미지를 읽어 온 뒤 cvtColor() 함수로 이들을 grayscale 변환한 것을 각각 input1_gray와 input2_gray에 저장한다.

```
Ptr<SIFT> sift = SIFT::create(
     0,                  // nFeatures
     4,                  // nOctaveLayers
     0.04,               // contrastThreshold
     10,                 // edgeThreshold
     1.6                 // sigma
);
```

이후 SIFT::create()를 호출하여 SIFT 객체 sift를 생성한다.

```
static Ptr< SIFT > cv::SIFT::create ( int          nfeatures,
                                      int          nOctaveLayers,
                                      double       contrastThreshold,
                                      double       edgeThreshold,
                                      double       sigma,
                                      int          descriptorType,
                                      bool         enable_precise_upscale = false )
```

이때 SIFT::create() 함수 정의는 위와 같으므로 검출할 특징점 개수 제한 없이 옥타브당 4개의 레이어를 가지고 각각 0.04, 10의 대비 threshold값과 에지 threshold값을 사용하며, 적용할 가우시안 필터의 표준편차는 1.6으로 설정한다.

```
Size size = input2.size();
Size sz = Size(size.width + input1_gray.size().width, max(size.height,
input1_gray.size().height));
Mat matchingImage = Mat::zeros(sz, CV_8UC3);

input1.copyTo(matchingImage(Rect(size.width, 0, input1_gray.size().width,
input1_gray.size().height)));
input2.copyTo(matchingImage(Rect(0, 0, size.width, size.height)));
```

위와 같이 두 입력 이미지를 옆으로 연결해 매칭 결과를 시각화할 출력 이미지 matchingImage를 만들고, input1을 출력 이미지의 오른쪽 절반에, input2를 왼쪽 절반에 복사한다.

```
vector<KeyPoint> keypoints1;
Mat descriptors1;

sift->detectAndCompute(input1_gray, noArray(), keypoints1, descriptors1);
printf("input1 : %d keypoints are found.\n", (int)keypoints1.size());

vector<KeyPoint> keypoints2;
Mat descriptors2;

sift->detectAndCompute(input2_gray, noArray(), keypoints2, descriptors2);
printf("input2 : %zd keypoints are found.\n", keypoints2.size());
```

이후 input1_gray와 input2_gray의 키 포인트와 디스크립터를 저장할 keypoints1, descriptors1, keypoints2, descriptors2를 선언하고 detectAndCompute() 함수를 호출해 각 이미지의 키 포인트와 디스크립터를 추출해 저장한다. 이후 검출된 특징점의 개수를 각각 출력한다.

```
for (int i = 0; i < keypoints1.size(); i++) {
     KeyPoint kp = keypoints1[i];
     kp.pt.x += size.width;
     circle(matchingImage, kp.pt, cvRound(kp.size * 0.25), Scalar(255, 255, 0), 1, 8, 0);
}

for (int i = 0; i < keypoints2.size(); i++) {
     KeyPoint kp = keypoints2[i];
     circle(matchingImage, kp.pt, cvRound(kp.size * 0.25), Scalar(255, 255, 0), 1, 8, 0);
}
```

keypoints1과 keypoints2 전체를 각각 순회하며 검출된 키포인트를 matchingImage에 표시한다.

```
vector<Point2f> srcPoints;
vector<Point2f> dstPoints;

bool crossCheck = true;
bool ratio_threshold = true;

findPairs(keypoints2, descriptors2, keypoints1, descriptors1, srcPoints, dstPoints, crossCheck,
ratio_threshold);

printf("%zd keypoints are matched.\n", srcPoints.size());
```

이후 crossCheck와 ratio_threshold 옵션에 따라 추출된 키포인트에 대해 한번 더 검증을 진행한다.대응점을 저장할 srcPoints, dstPoints를 선언하고 findPairs() 함수를 호출하면 함수 내부에서 옵션에 따라 cross check와 ratio based thresholding을 적용해 유효한 매칭을 srcPoints와 dstPoints에 저장한다. 함수가 종료되면 대응점의 개수를 출력해 매칭 결과를 확인한다.

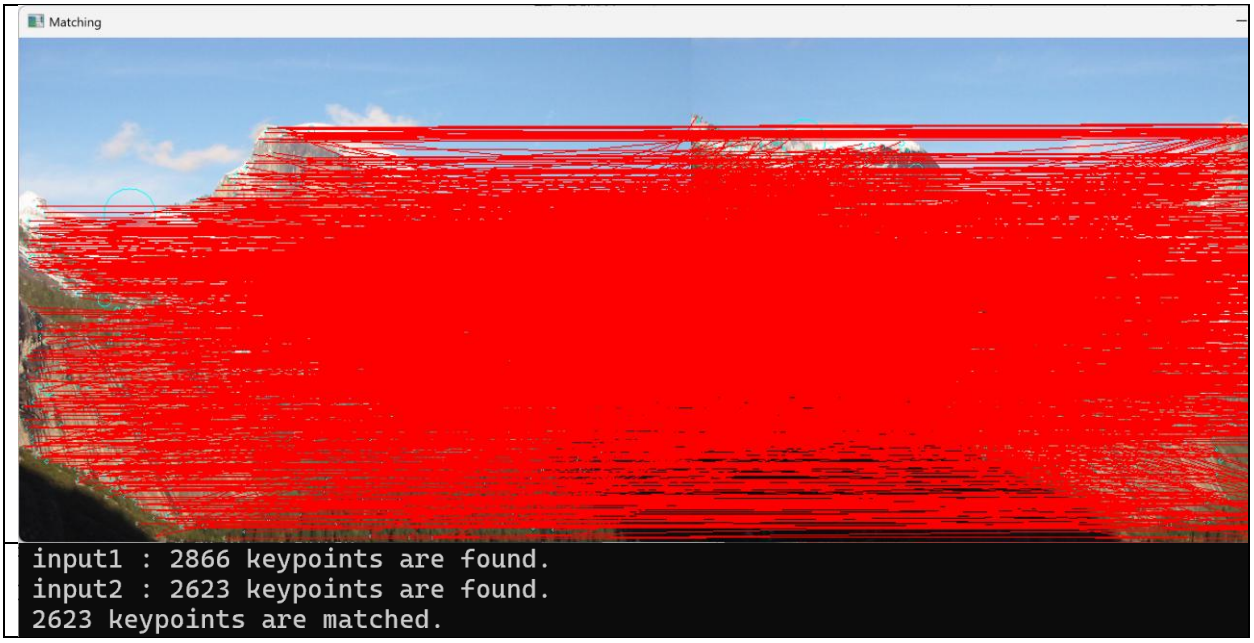
```
for (int i = 0; i < (int)srcPoints.size(); ++i) {
     Point2f pt1 = srcPoints[i];
     Point2f pt2 = dstPoints[i];
     Point2f from = pt1;
     Point2f to = Point(size.width + pt2.x, pt2.y);
     line(matchingImage, from, to, Scalar(0, 0, 255));
}

namedWindow("Matching");
imshow("Matching", matchingImage);
```

이후 각 이미지의 대응되는 점 사이에 선을 이어 표시하고, 결과 이미지 matchingImage를 출력하며 종

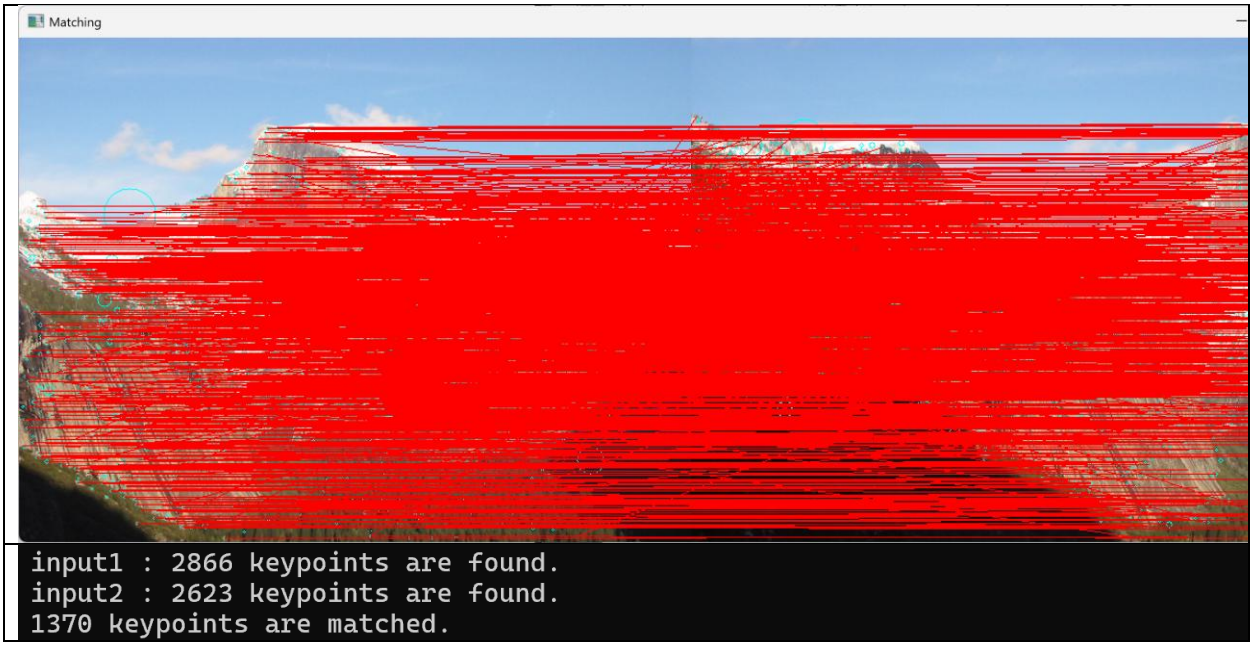
료한다.

5. 결과 분석

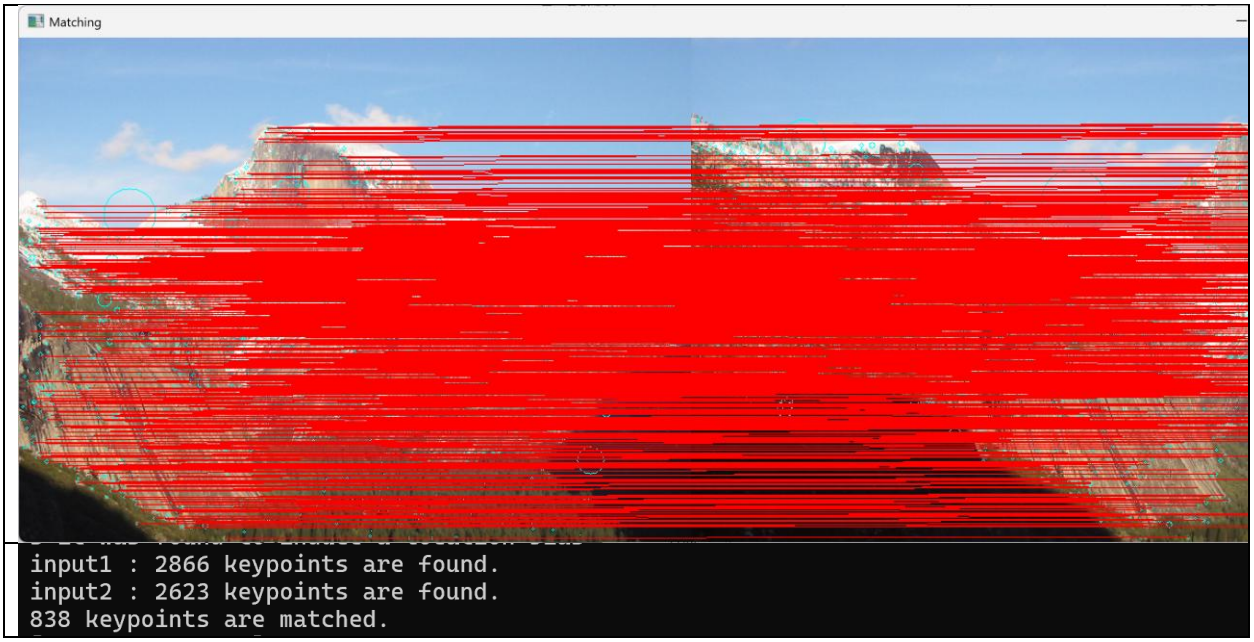
5.1. nearest neighbor만 고려하는 경우



5.2. cross checking만 수행하는 경우



5.3. cross checking과 ratio based thresholding을 모두 수행하는 경우



5.4. 결론

모든 경우 detectAndCompute() 함수로 키포인트와 디스크립터를 추출하므로 키포인트 개수는 동일하다. 하지만 cross checking과 ratio based thresholding을 모두 적용하지 않은 경우 단순 거리만을 고려해 키포인트를 결정하므로, 즉 거리가 모든 최근접 이웃을 키포인트로 간주하므로 매칭된 키포인트 개수가 가장 많을 뿐 아니라 모든 키포인트가 매칭에 참여한 것을 확인할 수 있다. cross checking과 ratio based thresholding을 모두 적용한 경우 최근접 이웃과 그 다음 근접 이웃의 비율을 고려하며 교차 검증까지 수행하므로 더 많은 키포인트가 매칭에서 제외되어 매칭된 키포인트 개수가 가장 적다. cross checking만 적용한다면 매칭된 키포인트 개수가 모두 적용하지 않은 경우보다는 많고 모두 적용한 경우보다는 적다.

SURF.cpp

SURF는 SIFT보다 노이즈에 민감하지만, SIFT에서 LoG를 DoG로 근사하는 반면 SURF는 box filter를 사용해 LoG를 근사함으로써 속도를 3배 가량 개선한다. Box filter를 계산하기 위해 픽셀 값의 누적 합 $\sum\sum I(x,y)$를 미리 계산하면 임의의 박스 안 픽셀의 합을 4번의 덧셈 및 뺄셈만으로 빠르게 구할 수 있다.

1. 코드 분석

```
Mat img_object = imread("input1.jpg", IMREAD_GRAYSCALE);
Mat img_scene = imread("input2.jpg", IMREAD_GRAYSCALE);
if (!img_object.data || !img_scene.data)
{
    std::cout << " --(!) Error reading images " << std::endl; return -1;
}
```

객체 이미지와 배경 이미지를 읽어 와 각각 img_object와 img_scene에 저장한다. 두 이미지 중 하나라도 오류가 있다면 에러 메시지를 출력하고 프로그램을 종료한다.

```
int minHessian = 400;
Ptr<SURF> detector = SURF::create(minHessian);
```

이후 create() 함수로 SURF 검출기 객체 detector를 생성한다. create() 함수의 구조는 아래와 같으며,

```
static Ptr<SURF> cv::xfeatures2d::SURF::create ( double     hessianThreshold = 100,
                                                 int        nOctaves = 4,
                                                 int        nOctaveLayers = 3,
                                                 bool       extended = false,
                                                 bool       upright = false
)
```

이때 minHessian은 hessian 키포인트 검출기의 threshold 값인 hessianThreshold 파라미터로 사용되므로 minHessian이 클수록 키포인트가 적게 검출된다.

```
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);
```

객체 이미지와 배경 이미지의 키포인트를 저장할 벡터 keypoints_obj와 keypoints_scn, 디스크립터를 저장할 descriptors_obj와 descriptors_scn을 선언하고 이들을 파라미터로 입력해 detectAndCompute() 함수로 두 이미지의 키포인트와 디스크립터를 추출한다.

```
FlannBasedMatcher matcher;
std::vector< DMatch > matches;
matcher.match(descriptors_object, descriptors_scene, matches);
```

이후 FLANN 매처로 디스크립터를 매칭한다. FLANN은 Fast Library for Approximate Nearest Neighbors의 약자로 효율적인 매칭점 검색을 가능하게 하며, FLANN 매처 객체 matcher를 생성하고 knnMatch() 함수를 호출하여 객체 이미지 디스크립터와 배경 이미지 간의 디스크립터의 매칭 지점을 추출한다.

```
double max_dist = 0; double min_dist = 100;
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;
}
```

최대 거리와 최소 거리를 저장할 변수를 생성하고 각각 0과 100으로 초기화하여 갱신이 용이하게 하며, for문으로 객체 이미지 디스크립터의 모든 원소를 순회하며 최대 거리와 최소 거리를 저장한다.

```
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]);
    }
}
```

이후 ratio based thresholding을 적용해 옳은 매칭을 추출한다. ratio based thresholding의 원리는 SIFT에서와 동일하며, 이때 RATIO_THR을 3으로 지정하여 matches[i]의 거리가 min_dist의 3배 이하인 경우에만 유효한 매칭으로 간주한다.

```
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);
```

이후 검출한 good_matches의 대응점들 간 선을 표시하며, 수행 결과 img_matches의 왼쪽에는 객체 이미지, 오른쪽에는 배경 이미지가 그려지며 그 위에 이들의 대응점을 연결하는 선이 표시된다.

```
std::vector<Point2f> obj;
std::vector<Point2f> scene;
for (size_t i = 0; i < good_matches.size(); i++)
{
  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);
```

이후 for문을 순회하며 good_matches의 대응점을 obj와 scene 벡터에 저장한다. 이때 obj 벡터는 객체 이미지의 점, scene 벡터는 배경 이미지의 벡터이며 이후 findHomography() 함수를 호출해 RANSAC 기반의 homography 행렬 H를 계산한다.

```
std::vector<Point2f> obj_corners(4);
obj_corners[0] = Point(0, 0); obj_corners[1] = Point(img_object.cols, 0);
obj_corners[2] = Point(img_object.cols, img_object.rows); obj_corners[3] = Point(0,
img_object.rows);
std::vector<Point2f> scene_corners(4);
perspectiveTransform(obj_corners, scene_corners, H);
```

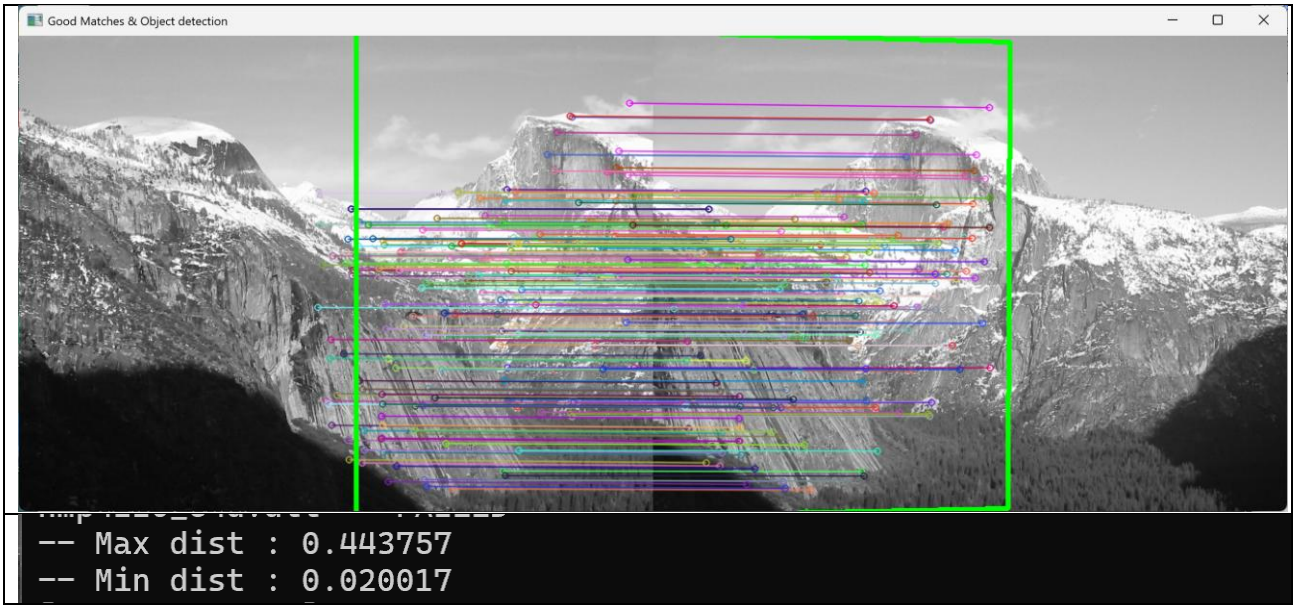
객체 이미지의 코너 네 개(좌측 상단, 우측 상단, 우측 하단, 좌측 하단)를 obj_corners 벡터에 저장하고, 위에서 계산한 homography 행렬 H를 이용해 perspective matrix transformation을 수행하고 변환된 결과를 scene_corners 벡터에 저장한다.

```
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);
 imshow("Good Matches & Object detection", img_matches);
```

이후 계산된 scene_corners의 네 점을 선으로 연결해 객체 이미지와 배경 이미지의 관계를 시각화한다. 이때 객체 이미지는 img_matches의 왼쪽, 배경 이미지는 오른쪽에 위치해 있으므로 x축 방향으로 객체 이미지의 너비만큼 오프셋을 적용하면 올바른 위치에 선을 표시할 수 있다. 이후 완성된 img_matches 이미지를 출력하고 프로그램을 종료한다.

2. 결과 분석



실행 결과 객체 이미지, 배경 이미지, 각 이미지의 매칭 지점과 이들을 연결하는 선이 표시된 결과 이미지가 출력되며 최대 거리와 최소 거리 또한 출력된다.