Technical report-Assignment08

컴퓨터공학 류다현(2376087)

Stitching.cpp

Assignment01에서 대응점을 하드코딩했던 것과 달리, SIFT 알고리즘을 사용해 이미지 간 대응점을 추출하여 stitching에 사용한다.

Cross checking, ratio-based thresholding 옵션을 포함한 SIFT 구현은 assignment07에서 구현한 것을 재사용하며, main()에서 SIFT 라이브러리로 두 이미지의 키포인트와 디스크립터를 추출한 뒤 findPairs() 함수에 전달해 crossCheck, ratio_threshold 옵션에 따라 매칭하는 구조는 동일하다. eucildDistance(), nearestNeighbor()의 유틸리티 함수 또한 이전과 동일하다.

findPairs() 이후 srcPoints, dstPoints 벡터를 사용해 두 이미지를 stitch하는 stitching() 함수를 호출한다.

1. **void stitching(const Mat& img1, const Mat& img2, vector<Point2f>& srcPoints, vector<Point2f>& dstPoints, String estOpt)**

   파라미터는 다음과 같다.

   - img1: 왼쪽 이미지
   - img2: 오른쪽 이미지. img1을 고정하고 img2를 img1에 stitch하도록 구현한다.
   - srcPoints: img2의 대응점 벡터
   - dstPoints: img1의 대응점 벡터
   - estOpt: affine 행렬 계산 시 case1, case2 지정 옵션

   stitching() 함수는 affine 행렬 M을 계산하는 부분을 제외하면 lab2 stitching.cpp 코드의 main() 함수와 유사하다.

```
Mat I1, I2;

img1.convertTo(I1, CV_32FC3, 1.0 / 255);
img2.convertTo(I2, CV_32FC3, 1.0 / 255);
```

   lab2에서는 I1, I2에 이미지를 곧바로 읽어 I1.convertTo(I1, CV_32FC3, 1.0 / 255); I2.convertTo(I2, CV_32FC3, 1.0 / 255);와 같이 변환했으나, 현재 stitching() 함수 외부에서 읽은 이미지 img1과 img2를 파라미터로 전달받았으므로 I1과 I2를 동일하게 선언하되 img1.convertTo(I1, CV_32FC3, 1.0 / 255); img2.convertTo(I2, CV_32FC3, 1.0 / 255);로 변환한다.

```
const float I1_row = I1.rows;
const float I1_col = I1.cols;
const float I2_row = I2.rows;
const float I2_col = I2.cols;

Mat M12, M21;
```

   추후 계산을 위해 두 이미지의 행과 열 크기를 별도로 저장하는 것은 동일하며, I1에서 I2로 변환하는

affine 행렬 M12와 I2에서 I1로 변환하는 M21을 선언한다.

1.1 Case1: srcPoints, dstPoints 벡터 전체를 이용하여 M 계산

```
if (estOpt == "case1") {
	int src_len = (int)srcPoints.size();
	int dst_len = (int)dstPoints.size();

	vector<int> ptl_x(dst_len), ptl_y(dst_len), ptr_x(src_len), ptr_y(src_len);
	for (int i = 0; i < dst_len; i++) {
		ptl_x[i] = (int)dstPoints[i].x;
		ptl_y[i] = (int)dstPoints[i].y;
	}
	for (int i = 0; i < src_len; i++) {
		ptr_x[i] = (int)srcPoints[i].x;
		ptr_y[i] = (int)srcPoints[i].y;
	}

	M12 = cal_affine<float>(ptl_x, ptl_y, ptr_x, ptr_y, src_len);    // I1 -> I2
	M21 = cal_affine<float>(ptr_x, ptr_y, ptl_x, ptl_y, dst_len);    // I2 -> I1
}
```

estOpt가 case1일 때 RANSAC 없이 srcPoints, dstPoints 벡터 전체를 사용하여 affine 행렬을 계산한다. 이는 Lab2, assignment01의 stitching과 동일한 방식이며 cal_affine() 함수 또한 재사용하지만, 이때 cal_affine()은 x 좌표와 y 좌표를 분리해 입력받으나 srcPoints, dstPoints는 (x, y)를 한 번에 가지는 Point2f형 벡터이므로 함수를 호출하기 전에 srcPoints와 dstPoints의 좌표를 분리해 ptl_x, ptl_y, ptr_x, ptr_y로 변환하는 전처리 과정을 거친다. cal_affine() 함수가 종료되면 M12와 M21에 완성된 affine 행렬이 저장된다.

```
template <typename T>
Mat cal_affine(int ptl_x[], int ptl_y[], int ptr_x[], int ptr_y[], int number_of_points){
	…
} // assignment01

template <typename T>
Mat cal_affine(vector<int> ptl_x, vector<int> ptl_y, vector<int> ptr_x, vector<int> ptr_y,
int number_of_points) {
	…
} // assignment08
```

이때 assignment01에서 대응점을 하드코딩함으로써 대응점 개수를 알고 크기가 고정된 대응점 배열을 사용할 수 있었던 것과 달리 현재 대응점 개수가 고정되어 있지 않으므로, cal_affine() 함수를 비롯한 코드 전반에서 대응점을 배열이 아닌 벡터에 저장하도록 수정한다.

1.2 Case2: RANSAC 이용하여 M 계산

```
else {
	double k = 4;
	double S = 20000;
	double delta = 3;

	M12 = cal_affine_RANSAC(dstPoints, srcPoints, k, S, delta);
	M21 = cal_affine_RANSAC(srcPoints, dstPoints, k, S, delta);
}
```

RANSAC 알고리즘으로 affine 행렬을 계산하는 cal_affine_RANSAC() 함수를 호출해 M12와 M21을 계산한다. 이때 k는 선택할 샘플 개수, S는 모델 계산을 반복할 횟수, delta는 inlier 판단 시 임계값이다. 위 코드에서는 k=4, S=20000, delta=3으로 하드코딩하여 호출하고 있다.

Affine 행렬 계산 이후는 lab2와 유사하다.

```
Point2f p1(M21.at<float>(0) * 0 + M21.at<float>(1) * 0 + M21.at<float>(2), M21.at<float>(3) * 0 +
M21.at<float>(4) * 0 + M21.at<float>(5)); // top left

Point2f p2(M21.at<float>(0) * 0 + M21.at<float>(1) * I2_row + M21.at<float>(2), M21.at<float>(3)
* 0 + M21.at<float>(4) * I2_row + M21.at<float>(5)); // bottom left

Point2f p3(M21.at<float>(0) * I2_col + M21.at<float>(1) * I2_row + M21.at<float>(2),
M21.at<float>(3) * I2_col + M21.at<float>(4) * I2_row + M21.at<float>(5)); // bottom right

Point2f p4(M21.at<float>(0) * I2_col + M21.at<float>(1) * 0 + M21.at<float>(2), M21.at<float>(3)
* I2_col + M21.at<float>(4) * 0 + M21.at<float>(5));   // top right
```

M21 행렬을 사용하여 네 꼭짓점 p1, p2, p3, p4를 구해 I2의 좌표계를 I1 좌표계로 변환하고,

```
int bound_u = (int)round(min(0.0f, min(p1.y, p4.y)));
int bound_b = (int)round(max(I1_row - 1, max(p2.y, p3.y)));
int bound_l = (int)round(min(0.0f, min(p1.x, p2.x)));
int bound_r = (int)round(max(I1_col - 1, max(p3.x, p4.x)));

// initialize merged image
Mat I_f(bound_b - bound_u + 1, bound_r - bound_l + 1, CV_32FC3, Scalar(0));
```

이를 바탕으로 I1에 I2를 stitching한 결과 이미지 크기를 계산해 해당 크기만큼의 최종 이미지 I_F를 선언한다.

```
for (int i = bound_u; i <= bound_b; i++) {
    for (int j = bound_l; j <= bound_r; j++) {
            float x = M12.at<float>(0) * j + M12.at<float>(1) * i + M12.at<float>(2) - bound_l;
            float y = M12.at<float>(3) * j + M12.at<float>(4) * i + M12.at<float>(5) - bound_u;

            float y1 = floor(y);
            float y2 = ceil(y);
            float x1 = floor(x);
            float x2 = ceil(x);

            float mu = y - y1;
            float lambda = x - x1;

            if (x1 >= 0 && x2 < I2_col && y1 >= 0 && y2 < I2_row)
                    I_f.at<Vec3f>(i - bound_u, j - bound_l) = lambda * (mu * I2.at<Vec3f>(y2,
                    x2) + (1 - mu) * I2.at<Vec3f>(y1, x2)) +
                    (1 - lambda) * (mu * I2.at<Vec3f>(y2, x1) + (1 - mu) * I2.at<Vec3f>(y1,
                    x1));
    }
}
```

이후 M12 행렬로 I1 좌표계를 I2 좌표계로 변환하여 I_F의 픽셀 (i, j)가 I2의 어떤 픽셀에 대응하는지 계산함으로써 lab2와 동일하게 inverse warping 및 bilinear interpolation한다.

```
blend_stitching(I1, I2, I_f, bound_l, bound_u, 0.5);
```

blend_stitching() 함수를 호출해 I_f에 I1과 I2를 blend한 것이 저장되도록 하며, 이때 blend_stitching()

함수는 lab2에서 사용했던 것과 동일하다.

2. Mat cal_affine_RANSAC(vector<Point2f>& srcPoints, vector<Point2f>& dstPoints, int k, int S, double delta)

RANSAC은 RANdom SAmple Consensus의 약자로, RANSAC을 사용해 데이터의 outlier를 배제하며 파라미터를 추정할 수 있다. 전체 데이터 중 모델을 정의하는 k개의 샘플을 랜덤하게 선정하고 그 샘플로 모델 파라미터를 추정한 후, 해당 모델에 근접하는 inlier를 카운트해 점수화한다. 위 과정을 S번 반복해 만들어진 모델 중 최고 점수를 달성한 것을 선택하는 것이 RANSAC의 기본 아이디어이다.

이를 바탕으로 RANSAC을 사용해 affine 행렬을 추정하며, 그 pseudocode는 다음과 같다.

반복 횟수 S만큼:

k개의 랜덤 대응점을 선택

선택한 대응점으로 affine 행렬 계산

계산한 affine 행렬로 거리 조건을 만족하는 inlier 카운트

inlier 개수가 가장 많은 affine 행렬 선택

해당 행렬의 inlier들로 affine 행렬을 다시 계산

여기서 이미지 I1의 점 p(x, y)와 이미지 I2의 점 p'(x', y')가 서로 대응되는 점이라 할 때, 이들이 inlier에 포함되려면 임계값 δ와 affine 행렬 T에 대해 $|Tp - p'|^2 < \delta^2$ 를 만족해야 한다.

이를 코드로 구현한 것이 cal_affine_RANSAC() 함수이며, 사용하는 파라미터는 다음과 같다.

- srcPoints, dstPoints: 대응점 벡터
- k: 선택할 샘플 개수
- S: 반복 횟수
- delta: inlier 카운트 시 사용할 임계값 δ

```
int srcSize = (int)srcPoints.size();
int maxInliers = 0;

vector<Point2f> bestSrcInliers, bestDstInliers

for (int i = 0; i < S; i++) {
    …
}
```

srcSize는 대응점의 총 개수, maxInliers는 이후 inlier 개수가 가장 많은 샘플의 inlier 개수를 기록하는 변수이며 bestScrInliers와 bestDstInliers는 각각 해당 샘플에서 추출된 inlier를 저장한다.

RANSAC은 대응점 선택-행렬 계산-inlier 카운트를 S번 반복하므로 0에서 (S-1)까지 for문으로 아래 과정을 S번 반복한다.

2.1 랜덤 대응점 k개 선택

```
while (sample_index.size() < k) {
	int randIndex = rand() % srcSize;

	if (find(sample_index.begin(), sample_index.end(), randIndex) ==
		sample_index.end()) {
		sample_index.push_back(randIndex);
	}
}
```

추출한 k개의 샘플이 srcPoints, dstPoints에서 가지는 인덱스를 저장할 sample_index 벡터를 선언하고, rand()로 전체 대응점 데이터 범위에서 난수를 생성해 이를 sample_index에 삽입한다. sample_index에 k개의 원소가 들어갈 때까지 while문으로 반복하며, 매 반복마다 난수가 중복으로 생성되었는지 검사하므로 중복 없는 k개의 샘플을 추출할 수 있다.

```
vector<Point2f> srcPoints_sample, dstPoints_sample;

for (int j = 0; j < k; j++) {
	srcPoints_sample.push_back(srcPoints[sample_index[j]]);
	dstPoints_sample.push_back(dstPoints[sample_index[j]]);
}
```

sample_index에는 추출한 샘플의 인덱스가 저장되어 있으므로, 실제 샘플 좌표를 저장할 srcPoints_sample, dstPoints_sample 벡터를 선언해 srcPoints와 dstPoints에서 sample_index의 인덱스를 가지는 점의 좌표를 해당 벡터에 push한다. for문을 벗어나면 srcPoints_sample과 dstPoints_sample에는 선택된 k개 대응점 샘플이 Points형 (x, y) 형태로 저장된다.

2.2 Affine 행렬 계산

```
vector<int> src_x(k), src_y(k), dst_x(k), dst_y(k);

for (int j = 0; j < k; j++) {
	src_x[j] = (int)srcPoints_sample[j].x;		// srcPoints 샘플
	src_y[j] = (int)srcPoints_sample[j].y;
	dst_x[j] = (int)dstPoints_sample[j].x;		// dstPoints 샘플
	dst_y[j] = (int)dstPoints_sample[j].y;
}

Mat Msd = cal_affine<float>(src_x, src_y, dst_x, dst_y, k);
```

cal_affine() 함수로 행렬을 계산하며, 위와 동일하게 x, y좌표를 분리하기 위해 src_x, src_y, dst_x, dst_y 벡터를 선언하고 for문으로 좌표를 분리해 각각 삽입한다. 이때 Msd는 src에서 dst로의 affine 행렬이다.

2.3 Inlier 카운트

```
int count = 0;
vector<Point2f> srcInliers, dstInliers;
```

count는 inlier 개수를, srcInliers와 dstInliers는 각각 srcPoints와 dstPoints에서 inlier로 판정된 점들을 저장한다.

```
for (int j = 0; j < srcSize; j++) {
     float x = Msd.at<float>(0) * srcPoints[j].x + Msd.at<float>(1) * srcPoints[j].y +
     Msd.at<float>(2);
     float y = Msd.at<float>(3) * srcPoints[j].x + Msd.at<float>(4) * srcPoints[j].y +
     Msd.at<float>(5);
     float dist = (x - dstPoints[j].x) * (x - dstPoints[j].x) + (y - dstPoints[j].y) *
     (y - dstPoints[j].y);

     if (dist < delta * delta) {
               count++;
               srcInliers.push_back(srcPoints[j]);
               dstInliers.push_back(dstPoints[j]);
     }
}
```

루프를 돌며 각 srcPoints[j]에 대해 affine 행렬 Msd로 예측된 대응점 (x, y)를 계산하고, 실제 대응점인 dstPoints[j]와 (x, y)의 유클리드 거리 제곱을 계산해 dist에 저장한다.. 이후 임계값 delta에 대해 dist가 delta*delta보다 작으면 inlier로 판단하여 count를 1 증가하고 srcInliers, dstInliers 벡터에 이들을 추가한다.

```
if (count > maxInliers) {
     maxInliers = count;

     bestSrcInliers = srcInliers;
     bestDstInliers = dstInliers;
}
```

현재 inlier 개수 count가 전체 inlier 개수인 maxInlier보다 크다면 maxInliers를 count로 변경하고, 가장 inlier가 많은 샘플의 inlier를 저장하는 bestSrcInliers와 bestDstInliers도 각각 srcInliers, dstInliers로 갱신한다. 이것으로 S만큼의 반복이 종료되며, 이 과정이 끝나면 bestSrcInliers와 bestDstInliers에는 S개의 추정 affine 행렬 중 가장 많은 inlier를 가지는, 즉 가장 최적인 행렬에 속한 inlier들이 저장되어 있다.

2.4 bestInliers에 대해 affine 행렬 재계산

```
int bestInlierSize = (int)bestSrcInliers.size();
```

bestInlierSize는 위에서 계산한 최적 행렬에 포함되는 inlier의 개수이다.

```
vector<int> best_src_x(bestInlierSize);
vector<int> best_src_y(bestInlierSize);

for (int j = 0; j < bestInlierSize; j++) {
     best_src_x[j] = (int)bestSrcInliers[j].x;
     best_src_y[j] = (int)bestSrcInliers[j].y;
}

return best_M;
```

bestSrcInliers와 bestDstInliers에 속한 대응점들로 최종 affine 행렬인 best_M을 계산하며, 위와 동일하게 cal_affine()을 호출하여 계산하므로 bestSrcInliers와 bestDstInliers의 x, y 좌표를 분리해 전달해야 한다. cal_affine()으로 best_M을 계산한 후 이를 반환하고 cal_affine_RANSAC() 함수를 종료한다.

3. 결과 분석

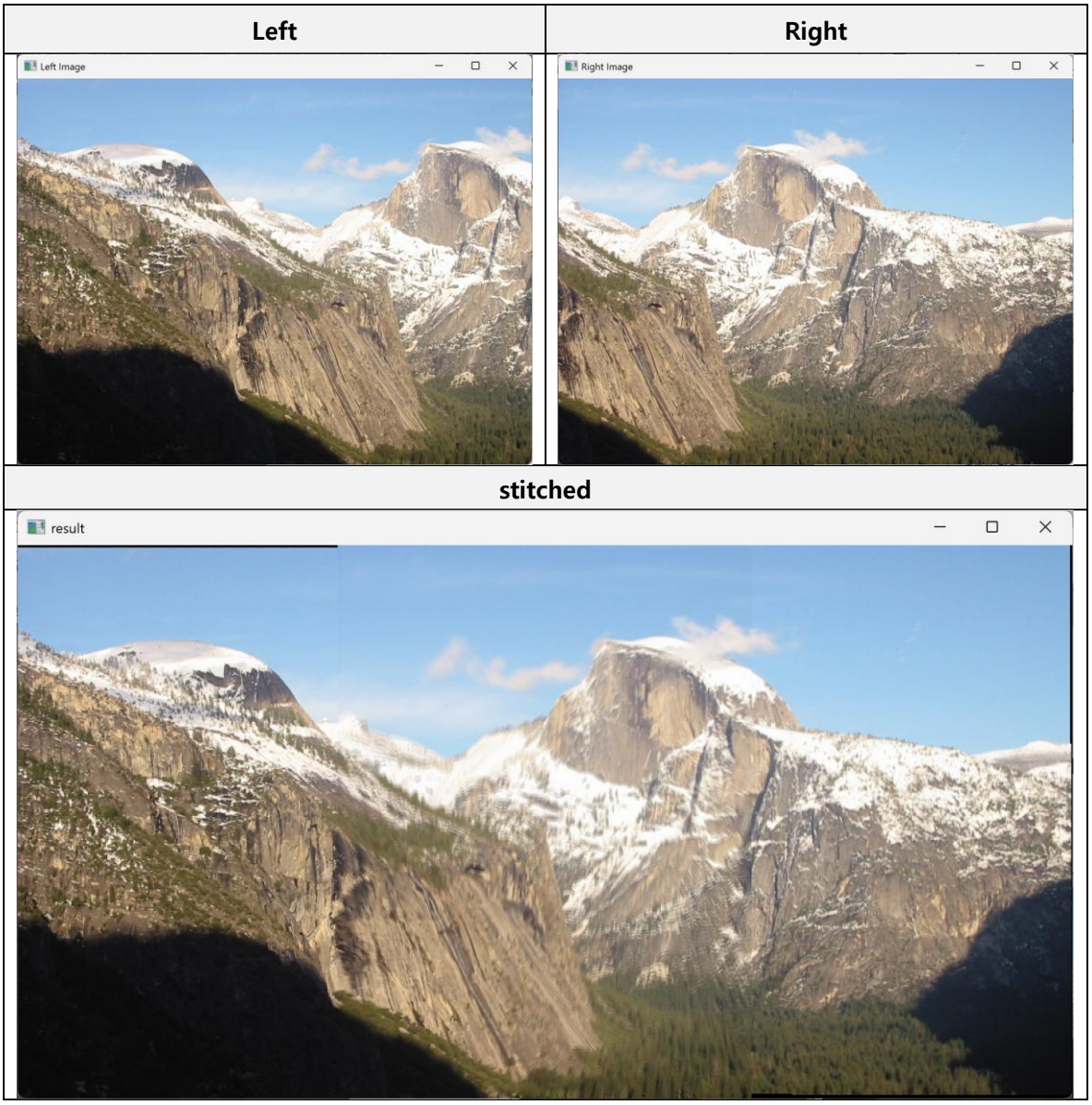
과제7의 input1.jpg, input2.jpg를 입력 이미지로 사용한다.

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

모든 실험에서 SIFT 설정은 위와 같이 통일한다.

3.1 Case1

SIFT로 키포인트를 추출한 후, cross checking, ratio-based thresholding을 수행한다. ratio-based thresholding 시 임계값 RATIO_THR은 0.4로 설정한다.



Keypoints & matching

SIFT 결과 왼쪽 이미지 input1에서 2866개, 오른쪽 이미지 input2에서 2623개의 키포인트가 검출되었으며, cross-checking과 ratio-based thresholding 결과 이들 중 838개의 키포인트가 대응관계에 있는 것으로 판단되었다. 이들 대응점 벡터 전체를 사용해 affine 행렬을 계산한 결과 이미지는 위와 같고, input1을 기준으로 적절한 위치에 input2가 stitch된 것을 확인할 수 있다.

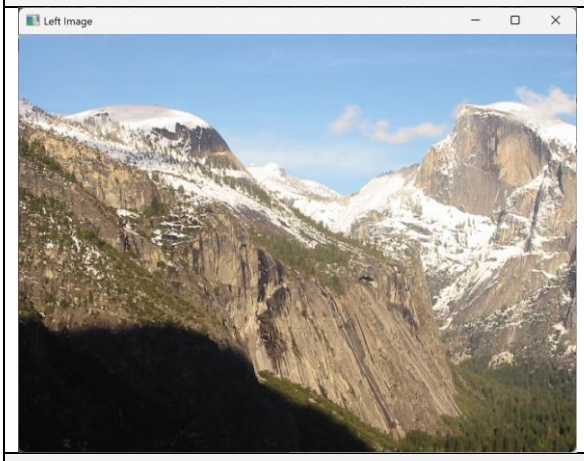
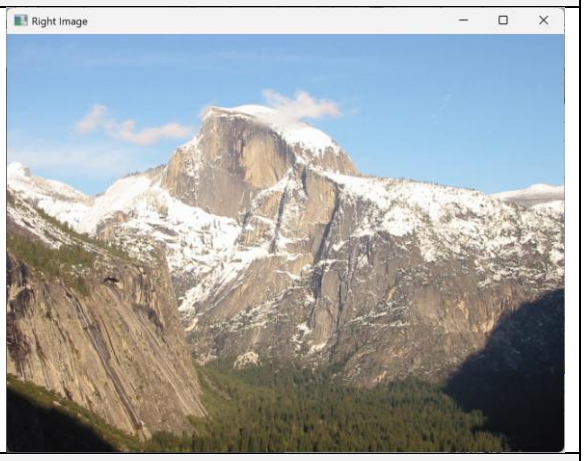
3.2 Case2

SIFT로 키포인트를 추출한 후, 추출된 키포인트로 cross checking, ratio-based thresholding을 수행한다. ratio-based thresholding 시 임계값 RATIO_THR은 0.4로 설정한다.

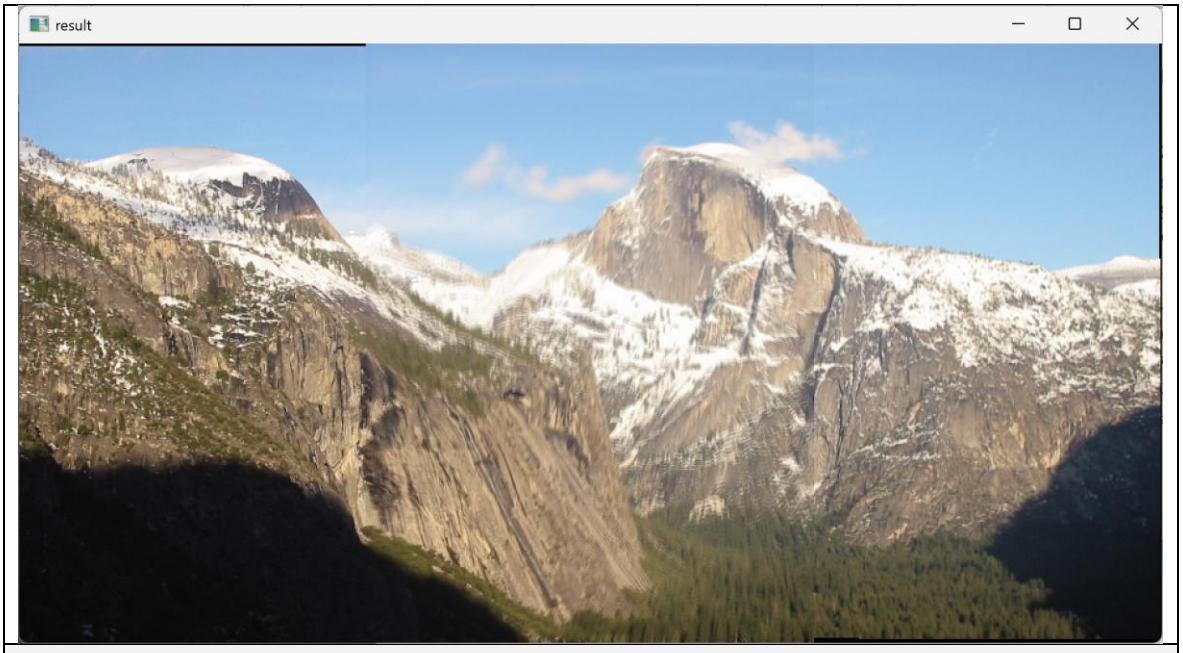
```
else {
	double k = 4;
	double S = 20000;
	double delta = 3;

	M12 = cal_affine_RANSAC(dstPoints, srcPoints, k, S, delta);
	M21 = cal_affine_RANSAC(srcPoints, dstPoints, k, S, delta);
}
```

행렬 계산 파라미터는 위와 같이 설정해 RANSAC 시 매번 k=4개의 샘플을 선택하고 대응점 거리 임계값 delta=3에 대해 inlier와 outlier를 판단하며, 이 과정을 S=20000번 반복해 총 20000개의 모델 중 가장 inlier가 많은 것을 선택하도록 구현하였다.

Left	Right
	

stitched



Keypoints & matching

3.1과 마찬가지로 SIFT 결과 왼쪽 이미지 input1에서 2866개, 오른쪽 이미지 input2에서 2623개의 키포인트가 검출되었으며, cross-checking과 ratio-based thresholding 결과 이들 중 838개의 키포인트가 대응 관계에 있는 것으로 판단되었다. RANSAC 알고리즘으로 affine 행렬을 계산한 결과 이미지는 위와 같고, input1을 기준으로 적절한 위치에 input2가 stitch된 것을 확인할 수 있다.

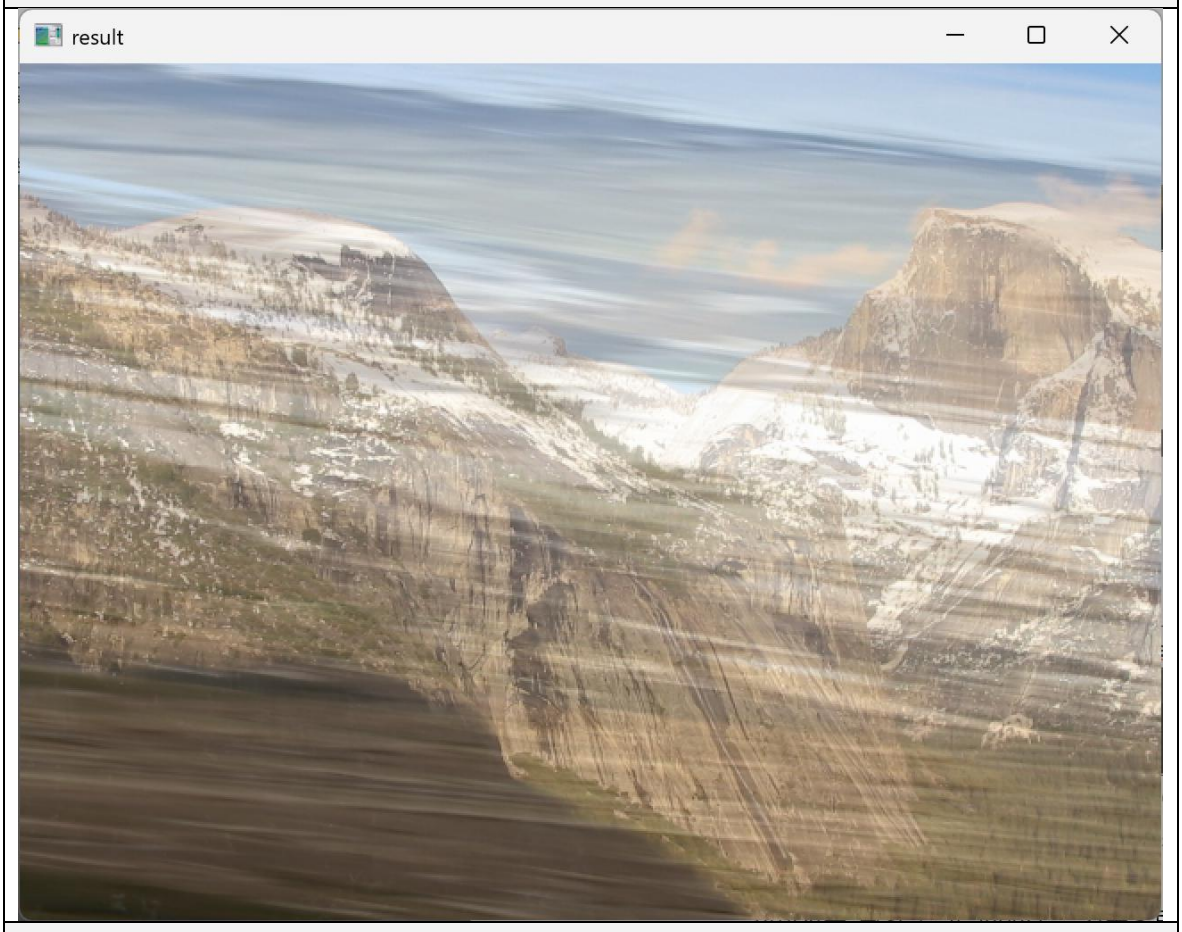
이때 3.1과 3.2 실험에서 case1, case2의 성능 차이가 뚜렷하게 나타나지 않는 것을 관찰할 수 있는데, 이는 RANSAC 임계값 delta를 임의의 값으로 하드코딩했기 때문으로 보인다. 대응점들의 실제 거리 분포를 고려하지 않고 임계값을 잘못 선택했을 경우 임계값이 지나치게 낮으면 유효한 매칭까지 outlier로 잘못 판단되어 제거되고, 너무 높으면 outlier를 배제하지 못하므로 결과 이미지가 왜곡될 수 있다. 이와 더불어 SIFT 결과 키포인트에 cross checking, ratio-based thresholding을 모두 적용해 입력 벡터의 대부분이 이미 유효한 대응점들이었으므로, case1에서 srcPoints, dstPoints 전체를 사용해 affine 행렬을 계산하였으나 실제로는 inlier들만으로 모델을 추정한 것과 유사한 효과가 발생하였음을 추론할 수 있다.

3.3 Outlier 추가한 경우

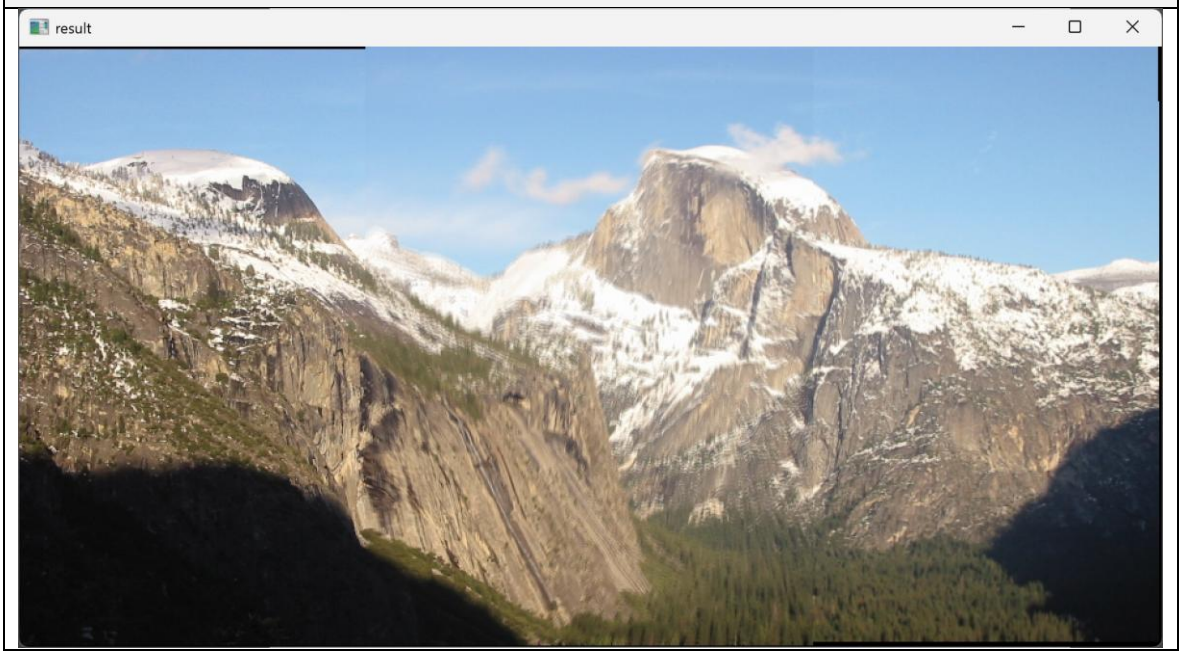
3.2의 추론을 바탕으로 RANSAC을 적용했을 때의 성능 차이를 확실히 검증하기 위해 ratio-based thresholding하지 않고 실험함으로써 대응점 품질을 의도적으로 손상시켰으며, 이때 ratio_threshold를 제외한 다른 부분은 이전과 똑같이 두어 비교가 타당하도록 하였다. 이전 경우들과 달리 srcPoints, dstPoints에 유효하지 않은 매칭, 즉 outlier가 포함되어 있으므로 case1

보다 case2에서 stitching 성능이 우수할 것으로 예상되며 그 결과는 아래와 같다.

Stitched-case1



Stitched-case2



Keypoints & matching

input1에서 2866개, input2에서 키포인트 2623개가 검출되었으나, ratio-based thresholding으로는 키포인트를 제거하지 않고 cross checking된 것만 사용하므로 3.1, 3.2의 832개보다 많은 1370개가 대응점으로 간주된다.

입력 벡터에 유효하지 않은 대응점이 대량 존재하므로(3.1 대비 1370-838=532개 증가) 이들을 모두 사용해 affine 행렬을 계산했을 때 완전히 잘못된 행렬이 계산되며, 그 결과 stitching한 이미지가 크게 왜곡된 것을 확인할 수 있다. 반면 case2에서는 입력 벡터에 outlier가 존재하더라도 RANSAC으로 이를 배제하므로 inlier들로만 affine 행렬을 계산할 수 있으며, 따라서 3.2의 결과 이미지와 3.3의 결과 이미지를 비교했을 때 큰 차이가 관찰되지 않았다.

Hough.cpp

RANSAC은 outlier에 강인(robust)하고 파라미터 차원이 큰 경우에도 적용할 수 있으며 최적화 파라미터 선택이 쉽지만, outlier의 비율과 파라미터 수가 증가함에 따라 계산량이 증가하고 여러 모델을 추정하기 어렵다는 단점이 존재한다.

이러한 문제를 보완하기 위해 Hough 변환을 사용할 수 있다. Hough 변환은 모델이 가질 수 있는 파라미터 공간에 격자를 설정하고, 원본 이미지의 각 특징점이 자신이 속할 수 있는 여러 파라미터에 투표할 수 있게 해 그 투표를 누적함으로써 여러 모델의 추정을 가능하게 한다. Hough 변환으로 이미지에서 직선을 검출하는 방식은 다음과 같다.

직선 모델은 $xcos\theta + ysin\theta = r$로 표현할 수 있으므로 파라미터 공간의 좌표는 (r, θ)이며, 원본 이미지에서 에지에 포함되는 점 (x, y)가 어떤 어떤 (r, θ)들에 대응하는지 계산해 해당 격자에 투표한다. 이때 현재 탐색하는 점 (x, y)를 고정하면 $xcos\theta + ysin\theta = r$를 만족하는 (r, θ)가 무수히 많고 이산화된 좌표 영역에서 계산해야 하므로 θ를 일정 간격으로 quantize한 θ_i들을 사용한다. 즉, (x, y)와 (r, θ)를 quantize한 각 (r, θi)에 대해 $xcos\theta_i + ysin\theta_i = r$를 계산하며, 따라서 하나의 에지 점이 자신이 속할 가능성이 있는 여러 모델에 투표할 수 있게 된다. 모든 에지 점이 이 과정을 반복하며 격자에 투표를 누적하는데 원본 이미지에서 실제로 같은 직선에 포함되는 에지 점들은 동일한 (r, θ)을 계산하므로 해당 (r, θ)의 accumulator에 득표 수가 많아지게 된다. 따라서 accumulator의 local maxima (r_i, θ_i)를 여러 개 선택하면 그 각각이 이미지의 직선 하나를 나타내므로 여러 모델을 한 번에 검출할 수 있다.

아래 구현에서는 에지 이미지에 Hough 변환을 수행하고 검출된 직선을 표시한다.

```
Mat src = imread("building.jpg", IMREAD_COLOR);
Mat dst, color_dst;

// check for validation
if (!src.data) {
        printf("Could not open\n");
        return -1;
}

Canny(src, dst, 50, 200, 3);
cvtColor(dst, color_dst, COLOR_GRAY2BGR);
```

사용할 이미지를 src에 읽어 오고 유효성을 검사한다. Hough 변환 시 에지 이미지와 직선 파라미터 공간 사이 변환이 필요하므로 Hough 수행 전에 Canny 에지 검출기로 에지를 우선 검출한다. 검출된 에지 이미지는 dst에 저장된다.

1. **HoughLines() 사용**

   HoughLines() 함수를 사용해 직선을 검출한다. 이때 HoughLines() 함수는 표준 Hough 변환을 사용해 직선을 검출하는 함수로, 그 구조는 아래와 같다.

```
void cv::HoughLines ( InputArray          image,
                      OutputArray         lines,
                      double              rho,
                      double              theta,
                      int                 threshold,
```

```
                    double              srn = 0,
                    double              stn = 0,
                    double              min_theta = 0,
                    double              max_theta = CV_PI
)
```

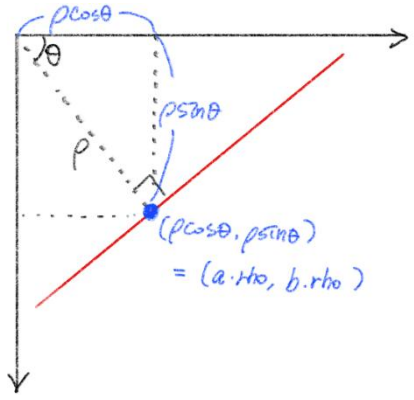
입력 이미지 image에 대해 Hough 변환을 적용해 직선을 검출한 것을 lines에 저장하며, 이때 각 직선은 2개 또는 3개의 원소를 가지는 벡터 (ρ, θ) 또는 (ρ, θ, votes)로 표현된다. ρ는 이미지의 좌측 상단 점 (0, 0)으로부터 거리. θ는 직선의 각도이며 votes는 accumulator 값, 즉 득표 수이다. rho와 theta는 각각 accumulator에서 거리와 각도 해상도(거리와 각도를 어떤 간격으로 나눌지 설정), threshold는 직선을 판단하기 위해 필요한 득표 수 임계값이다.

```
vector<Vec2f> lines;
HoughLines(dst, lines, 1, CV_PI / 180, 200);

for (size_t i = 0; i < lines.size(); i++)
{
     float rho = lines[i][0];
     float theta = lines[i][1];
     double a = cos(theta), b = sin(theta);
     double x0 = a*rho, y0 = b*rho;
     Point pt1(cvRound(x0 + 1000 * (-b)), cvRound(y0 + 1000 * (a)));
     Point pt2(cvRound(x0 - 1000 * (-b)), cvRound(y0 - 1000 * (a)));
     line(color_dst, pt1, pt2, Scalar(0, 0, 255), 3, 8);
}
```

따라서 HoughLines() 함수를 사용하는 경우 위와 같이 구현하며, HoughLines(dst, lines, 1, CV_PI / 180, 200);으로 호출해 ρ를 1픽셀 단위로 누적하고 각도 해상도를 π/180 간격으로 설정한다. threshold=150이므로 accumulator에서 득표 수가 200 이상인 (ρ, θ)쌍만을 직선으로 간주해 lines에 저장한다.

호출 결과 lines에 검출된 직선의 정보가 저장되므로 이후 for문에서 lines의 전체 원소를 순회하며 (ρ, θ)가 의미하는 직선을 계산하고 이미지에 표시한다. lines는 (ρ, θ) 형태의 Vec2f형 벡터이고 lines[i][0]과 lines[i][1]는 각각 현재 표시할 직선의 거리와 각도이므로 이들을 rho와 theta에 저장하며, $xcos\theta + ysin\theta = r$ 공식을 사용하기 위해 cos(theta), sin(theta)를 계산해 a와 b에 저장한다. 최종적으로 xcosθ와 ysinθ인 a*rho, b*rho를 계산해 x0과 y0에 저장하는데, 이때 (ρcosθ, ρsinθ)=(a*rho, b*rho)=(x0-y0) 지점은 이미지의 원점(upper left 지점)에서 현재 탐색 중인 직선으로의 수선의 발이다.



이 과정을 도식화하면 우측 그림과 같으며, 이미지 전체를 가로지르는 직선을 그리기 위해 (x0, y0)에서 직선의 진행 방향 (-sinθ, cosθ)으로 충분히 큰 수 1000만큼 연장한 양 끝 지점 pt1에서 pt2까지 line() 함수로 선을 표시한다.

2. HoughLines()에서 직선 시작점 및 종료점 계산

HoughLines()로 직선을 검출했을 때 직선의 시작점과 종료점을 구분하지 않고 창 전체를 가로지르는 직

선이 표시된다. 이들 직선을 segmentation하여 실제 에지 범위에 맞는 선분을 검출하기 위해 별도의 알고리즘이 필요하며, 그 pseudocode는 다음과 같다.

(ρ,θ) 직선 내의 모든 픽셀에 대해:

현재 픽셀이 실제로 에지이고 선분의 첫 픽셀이라면:

현재 픽셀을 선분의 시작 픽셀로 고정

공백 카운트 1 증가

선분 길이 카운트 1 증가

현재 픽셀이 에지 픽셀이 아니라면:

현재 픽셀이 선분의 첫 픽셀이 아니라면:

공백 카운트 1 증가

공백 길이가 최대 공백 길이에 도달했으면:

지금까지 선분 길이가 최소 선분 길이 이상일 경우:

현재 픽셀을 선분 종료 픽셀로 설정

시작 픽셀-종료 픽셀 직선 그리기

공백 카운트, 선분 길이 카운트 초기화

위 루프의 마지막 선분이 유효하다면:

현재 픽셀을 선분 종료 픽셀로 설정

시작 픽셀-종료 픽셀 직선 그리기

이처럼 실제 에지 픽셀 정보와 HoughLines()로 검출한 직선 정보 lines 배열을 비교해 선분의 시작점과 끝점을 결정할 수 있다. lines의 직선 내 픽셀 중 에지가 아닌 픽셀(공백)이 공백 임계값보다 크면 지금까지 모인 연속 에지 픽셀의 길이가 선분 길이 임계값 이상인지 검사하고, 충분히 길다면 해당 픽셀들을 실제 선분 요소로 확정한 뒤 새로운 선분 탐색을 시작한다.

2.1 void getSegment(const Mat image, const Vec2f& lines, int minLineLength, int maxLineGap, Mat& colorDst)

위 pseudocode를 실제로 구현하면 다음과 같다. 자세한 설명은 주석으로 대체한다.

```
Mat output = image.clone(); // 결과 이미지

float rho = lines[0];
float theta = lines[1];
double a = cos(theta), b = sin(theta);
```

```
double x0 = a * rho, y0 = b * rho;
Point pt1(cvRound(x0 + 1000 * (-b)), cvRound(y0 + 1000 * (a)));
Point pt2(cvRound(x0 - 1000 * (-b)), cvRound(y0 - 1000 * (a)));

LineIterator it(image, pt1, pt2, 8); // LineIterator: 직선 내 픽셀 순회
int gap = 0;     // gap count
int count = 0;   // line segment count
Point start;

for (int i = 0; i < it.count; i++, ++it) {
      Point p = it.pos();  // 현재 픽셀 위치
      if (p.x < 0 || p.x >= image.cols || p.y < 0 || p.y >= image.rows) {
            continue;    // 이미지 범위 밖이면 skip
      }
      bool isEdge = image.at<uchar>(p) == 255;   // 에지 픽셀인지 여부 저장

      if (isEdge) {  // 에지 픽셀이라면
            if (count == 0) {    // line segment 첫 픽셀이라면
                  start = p;
            }
            gap++;
            count++;
      }
      // 에지 픽셀이 아닌데
      else if (count > 0) {   // line segment가 존재한다면
            gap++;  // gap count 증가
            if (gap == maxLineGap) { // gap이 maxLineGap과 같으면 line segment가 아님
                  if (count >= minLineLength) {
                  // line segment가 존재하고 길이가 minLineLength보다 크면
                  // -> line segment 끝내야 함
                        Point end = Point(
                              it.pos().x - cvRound(gap * (-b)),
                              it.pos().y - cvRound(gap * (a))
                        );
                        line(colorDst, start, end, Scalar(0, 0, 255), 3, 8);
                  }
                  count = 0;     // line segment count 초기화
                  gap = 0;       // gap count 초기화
            }
      }
}
// 마지막 segment
if (count >= minLineLength) {
       Point end = it.pos();
       line(colorDst, start, end, Scalar(0, 0, 255), 3, 8);
}
```

3. HoughLinesP() 사용

2.2의 아이디어를 사용하는 opencv 라이브러리 함수가 HoughLinesP()이며, 확률적 Hough 변환으로 직선이 아닌 선분을 검출한다. 함수의 정의는 아래와 같다.

```
void cv::HoughLinesP ( InputArray         image,
```

```
                        OutputArray        lines,
                        double             rho,
                        double             theta,
                        int                threshold,
                        double             minLineLength = 0,
                        double             maxLineGap = 0
)
```

image, rho, theta, threshold 파라미터는 HoughLines()와 동일하지만 HoughLinesP()에서는 lines에 검출된 선분의 시작점 (x1, y1)과 끝점(x2, y2)를 포함하는 4원소 벡터 (x1, y1, x2, y2)를 저장한다.

```
vector<Vec4i> lines;
HoughLinesP(dst, lines, 1, CV_PI / 180, 50, 50, 10);

for (size_t i = 0; i < lines.size(); i++)
{
     line(color_dst, Point(lines[i][0], lines[i][1]),
              Point(lines[i][2], lines[i][3]), Scalar(0, 0, 255), 3, 8);
}
```

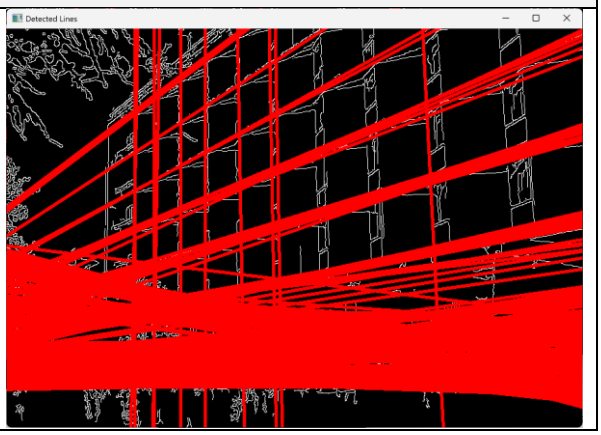
이후 동일하게 for문으로 lines 배열 전체를 탐색하며 검출된 시작점과 끝점을 잇는 선분을 표시한다. (lines[i][0], lines[i][1])은 (x1, y1)로 시작점 좌표이며, (lines[i][2], lines[i][3])은 (x2, y2)로 끝점 좌표이므로 line() 함수를 이용해 해당 지점을 연결한다. HoughLines() 함수를 사용한 경우와 달리 시작점, 끝점 좌표가 반환되므로 좌표를 계산할 필요 없이 직선을 표시하는 것으로 충분하다.

4. 결과 분석

4.1 HoughLines() 사용

```
HoughLines(dst, lines, 1, CV_PI / 180, 180);
```

위와 같이 호출하여 실행해 거리는 1픽셀 단위, 각도는 π/180 간격으로 설정하고 득표 수가 180 이상인 (ρ, θ)쌍을 직선으로 간주한다.

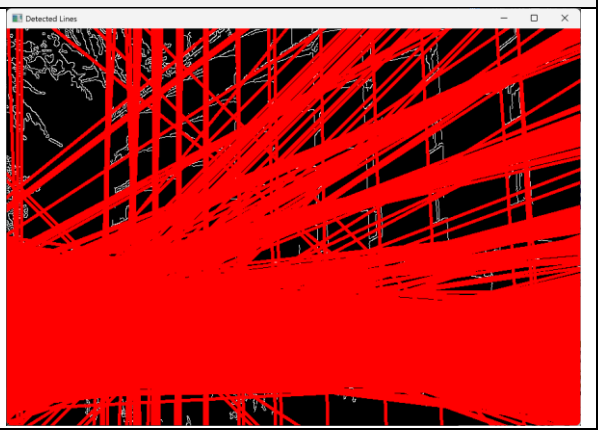
Source	Detected lines
	

실행 결과 에지 이미지에 직선이 검출되어 표시된 이미지가 출력되었으며, 이때 검출된 직선은 끝나지 않고 이미지 전체로 이어진다.

```
HoughLines(dst, lines, 1, CV_PI / 180, 180);
```

threshold=150으로 변경해 다시 실행할 경우, 득표 수가 더 적은 (ρ, θ)쌍도 직선으로 판정되므로

아래와 같이 더 많은 직선이 검출되는 것을 확인할 수 있다.

Source	Detected lines(threshold=150)
	

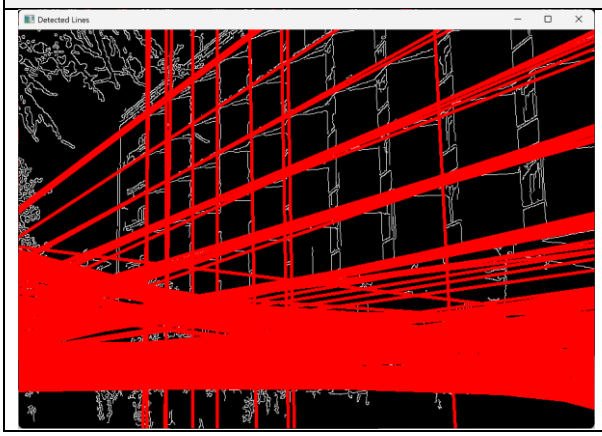
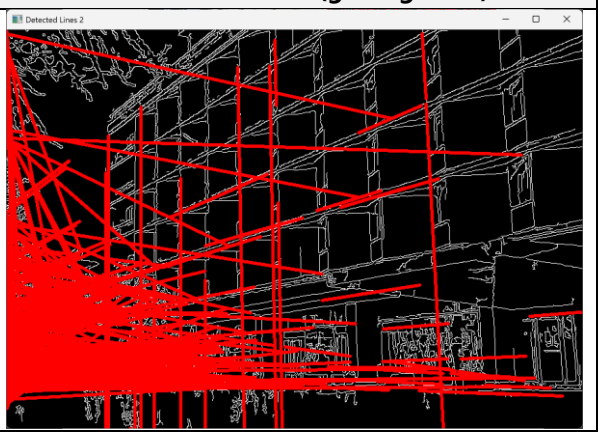
4.2 Line segment 추정

2와 2.1의 방식이 타당한지 검증하기 위해 구현한 getSegment() 함수를 사용해 4.1의 검출된 직선에서 선분을 추출한다.

```
Mat color_dst_test = color_dst.clone();
HoughLines(dst, lines, 1, CV_PI / 180, 180);

for (size_t i = 0; i < lines.size(); i++) {
        getSegment(dst, lines[i], 10, 50, color_dst_test);
}
```

위와 같이 getSegment(dst, lines[i], 10, 50, color_dst_test);로 호출해 길이가 최소 10인 선분을 검출하도록 하였으며, 이때 50픽셀 이하의 공백을 허용한다.

Detected lines	Detected lines 2(getSegment)
	

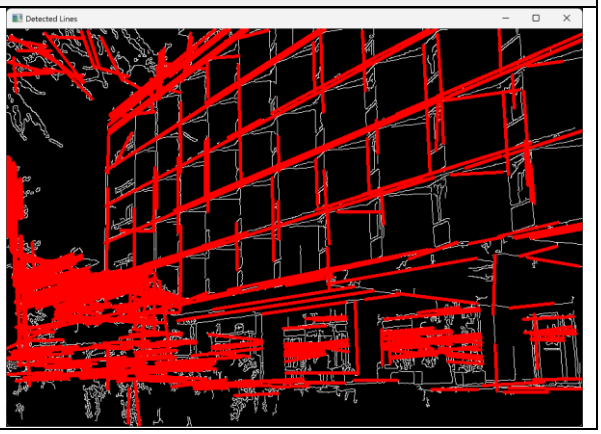
HoughLines()를 단독으로 적용했을 때는 창 전체를 가로지르는 직선이 출력되었으나, HoughLines 한 lines 배열에 getSegment() 함수를 적용한 경우(Detected lines 2) 직선이 실제 에지 픽셀의 시작과 끝에서 단절된 형태로 출력된 것을 확인할 수 있다.

4.3 HoughLinesP() 사용

HoughLinesP() 함수를 사용해 직선이 아닌 선분을 검출해 표시한다.

```
HoughLinesP(dst, lines, 1, CV_PI / 180, 50, 50, 10);
```

위와 같이 호출하며, 거리는 1픽셀 단위, 각도는 π/180 간격으로 설정하고 길이가 50 이상인 픽셀을 선분으로 간주하며, 이때 공백 임계값은 10으로 두어 공백이 10 미만인 선분은 연속된 것으로 판단한다.

Source	Detected lines
	

실행 결과 에지 이미지에 선분이 검출되어 표시된 이미지가 출력되며, 4.1과 달리 직선이 segmentation되어 시작점과 끝점을 확인할 수 있다.