1. SIFT.cpp

코드 목적:

SIFT descriptor 를 이용해 특징점이 되는 keypoint 를 찾고, Affine Transform 을 수행한다.

함수 설명

euclidDistance(Mat& vec1, Mat& vec2)

vec1과 vec2 사이의 거리(둘이 얼마나 다른지)를 반환한다. 유사할수록 값이 작다.

nearestNeighbor(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors)

keypoints를 가지는 descriptors위의 점 중 vec와 가장 유사한, matching되는 점의 인덱스를 반환한다.

SecondNearestNeighbor(Mat& vec, vector<KeyPoint>& keypoints, Mat& descriptors, int first)

keypoints를 가지는 descriptors위의 점 중 vec와 두번째(first 다음으로)로 유사한 점의 인덱스를 반환한다.

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

descriptor1 의 keypoints1 에 matching 되는 descriptor2 의 keypoints2 를 찾아 srcPoints (descriptor 1 의 점)와 dstPoints(descriptor 2 의 점)에 저장한다.

crossCheck: cross check 여부, ratio_threshold: threshold ratio 사용 여부

cal_affine:

매개변수: int ptl_x[], int ptl_y[], int ptr_x[], int ptr_y[], int number_of_points

ptl_x[]: corresponding pixels 의 왼쪽 이미지에서의 x 좌표

ptl_y[]: corresponding pixels 의 왼쪽 이미지에서의 y 좌표

ptr_x[]: corresponding pixels 의 오른쪽 이미지에서의 x 좌표

ptr_y[]: corresponding pixels 의 오른쪽 이미지에서의 y 좌표

number_of_points: corresponding pixels 의 개수

함수 목적: ptl_x, ptl_y 와 계산해 ptr_x, ptr_y 를 구할 수 있는 Matrix(A_{12}, A_{21}) 반환

$$\begin{bmatrix} \text{ptr\_x} \\ \text{ptr\_y} \end{bmatrix} = A \begin{bmatrix} \text{ptl\_x} \\ \text{ptl\_y} \\ 1 \end{bmatrix}$$

AffineTransform(Mat input1, Mat input2, vector<Point2f>& srcPoints, vector<Point2f>& dstPoints)

input1 과 input2 의 matching 된 keypoints 의 좌표 담은 scrPoints 와 dstPoints 를 이용해 Affine Transform 을 수행한다. input2 를 input1 에 stitching 한 결과를 반환한다

SIFTfunc(Mat input1, Mat input2,vector<KeyPoint>& keypoints1, Mat& descriptors1, vector<KeyPoint>& keypoints2 , Mat& descriptors2)

input1 의 keypoint 정보를 담은 keypoints1 과 descriptors1 과 input2 의 keypoint 정보를 담은 keypoints2 과 descriptors2 를 받아 findPairs() 함수를 이용해 featurematching 을 진행하고, AffineTransform 를 이용해 AffineTransform 을 수행한다

설명-흐름대로(과제 01 의 affine transformatrion 과 과제 07 의 sift detection 을 결합한 것으로, 자세한 설명보다는 흐름에 집중해 설명하겠다)

1. main 함수) input1 과 input2 의 keypoints 를 찾는다.

```
vector<KeyPoint> keypoints1;
Mat descriptors1;
// Detect keypoints
detector->detect(input1_gray, keypoints1);
extractor->compute(input1_gray, keypoints1, descriptors1);
printf("input1 : %d keypoints are found.\n", (int)keypoints1.size());

// input2 이미지의 keypoint 찾기
vector<KeyPoint> keypoints2;
```

2. main 함수) SIFTfunc 함수 호출

I_f 는 input2 를 input1 에 붙이는것이고, I_f2 는 input1 를 input2 에 붙이는것이다

```
Mat I_f=SIFTfunc(input1, input2, keypoints1,descriptors1,keypoints2,descriptors2);
Mat I_f2=SIFTfunc(input2, input1, keypoints2,descriptors2,keypoints1,descriptors1);
```

3. SIFTfunc 함수) findPars()함수 호출해 feauture matching 수행한다

```
vector<Point2f> srcPoints;
vector<Point2f> dstPoints;
findPairs(keypoints2, descriptors2, keypoints1, descriptors1, srcPoints, dstPoints);
```

4. findPars 함수)

```
for (int i = 0; i < descriptors1.rows; i++) {
        KeyPoint pt1 = keypoints1[i];
        Mat desc1 = descriptors1.row(i);

        int nn = nearestNeighbor(desc1, keypoints2, descriptors2);

       int nn2=SecondNearestNeighbor(desc1, keypoints2, descriptors2,nn);

        Mat v1=descriptors2.row(nn);
        double dist1 = euclidDistance(desc1, v1);

        Mat v2=descriptors2.row(nn2);
        double dist2 = euclidDistance(desc1, v2);

        if ((dist1/dist2) > 0.65) continue;
```

NN을 이용해 가장 가까운 특징점을 찾고, ratio-based thresholding과 cross-checking모두 적용해준다. (hw07참고)

5. SIFTfunc함수) 구한 matching된 점들을 이용해 두 이미지를 붙이기 위해 AffineTransform 함수 호출

```
Mat I_f=AffineTransform(input1, input2,srcPoints,dstPoints);
```

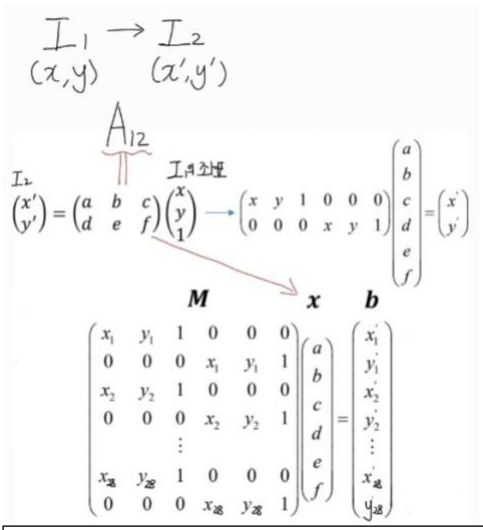
6. AffineTransform함수)
대응점들 새로운 배열에 저장

```
for (int i = 0; i < n; ++i) {
        ptl_x[i] = dstPoints[i].x;
        ptl_y[i] = dstPoints[i].y;
        ptr_x[i] = srcPoints[i].x;
```

cal_affine 함수 호출해 input1위의 점을 input2위의 점으로 변환하기 위한 행렬 A12와 input2위의 점을 input1위의 점으로 변환하기 위한 행렬 A21을 구한다.

```
Mat A12 = cal_affine<float>(ptl_x, ptl_y, ptr_x, ptr_y, n);
Mat A21 = cal_affine<float>(ptr_x, ptr_y, ptl_x, ptl_y, n);
```

7. cal_affine 함수)



```
Mat cal_affine(int ptl_x[], int ptl_y[], int ptr_x[], int ptr_y[], int number_of_points) {
        Mat M(2 * number_of_points, 6, CV_32F, Scalar(0));
        Mat b(2 * number_of_points, 1, CV_32F);

        Mat M_trans, temp, affineM;
for (int i = 0; i < number_of_points; i++) {
   M.at<T>(2 * i, 0) = ptl_x[i];      M.at<T>(2 * i, 1) = ptl_y[i];          M.at<T>(2 * i, 2) = 1;
   M.at<T>(2*i+1,3) = ptl_x[i];    M.at<T>(2 * i + 1, 4) = ptl_y[i];      M.at<T>(2 * i + 1, 5) = 1;
   b.at<T>(2 * i) = ptr_x[i];          b.at<T>(2 * i + 1) = ptr_y[i];
}
transpose(M, M_trans);
invert(M_trans * M, temp);
affineM = temp * M_trans * b;

return affineM;
```

주어진 대응점들을 이용해 M과 b를 만들고,

$\boldsymbol{Mx = b \rightarrow x = (M^{\mathrm{T}}M)^{-1}M^{\mathrm{T}}b}$ 를 계산한다

구한 x를 반환한다.

8. AffineTransform함수)

구한 A21을 이용해 input2의 네 꼭짓점을 결과 이미지로 옮긴다. input1의 꼭짓점과 input2를 이동시킨 꼭짓점을 이용해 결과 이미지의 크기를 구하고, 결과 이미지의 픽셀에 해당하는 input2의 점을 구해 그 값을 가져와 복사한다.
input1을 비율에 맞게 그리기 위해 blending_stitching함수를 호출한다.

9. blending_stitching함수) input2가 그려진 부분은 비율 0.5에 맞게 input1을 그려주고, 아무것도 없는 부분엔 input1을 그려 AffineTransform함수로 반환

10. 완성된 이미지를 AffineTransform함수에서 SIFTfunc함수로, 다시 main함수로 반환해 출력

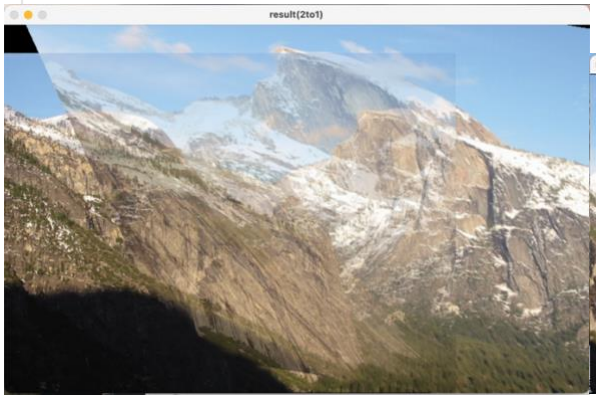
정리하자면, hw01에서 주어진 점을 이용해 Affine Transformation을 했었고, hw07에서 keypoints를 구하고 가장 비슷한 feature끼리 연결했었다. 이 코드에선 두 방법을 합쳐 keypoints를 구하고 featuring matching한 결과를 이용해 Affine Transformation을 수행하는 것이다.

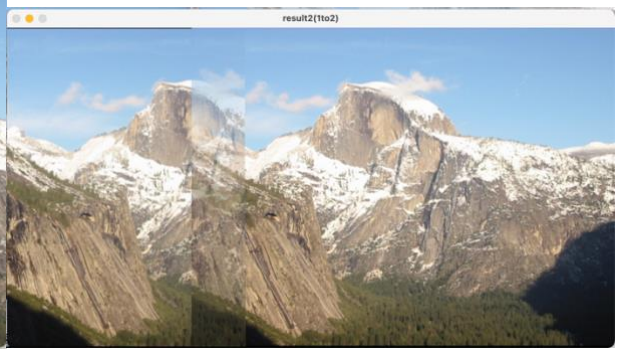
실행 결과:

ratio_threshold

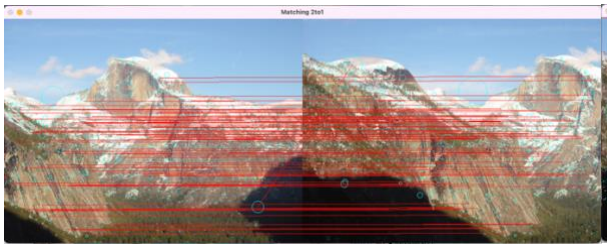
1) `if ((dist1/dist2) > 0.65) continue;`

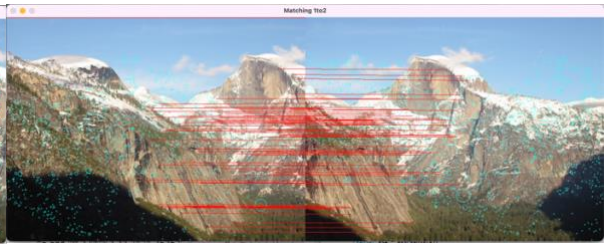
```
input1 : 2865 keypoints are found.
input2 : 2623 keypoints are found.
(2to1): 69 keypoints are matched.
(1to2): 68 keypoints are matched.
```





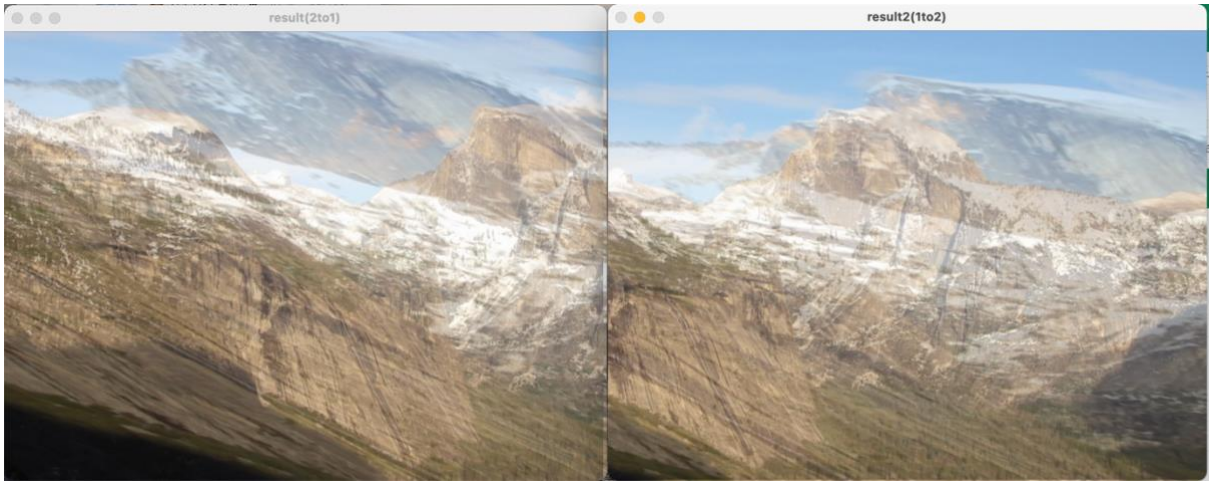
1to2에선 이미지가 잘 stitching되지만, 2to1에서는 그렇지 않은 것을 볼 수 있다.
아래 이미지는 hw07의 feature matching의 결과이다. 2to1의 가운데 보이는 outlier가 작용해 올바르지 않은 결과를 가져온 것으로 예상된다.



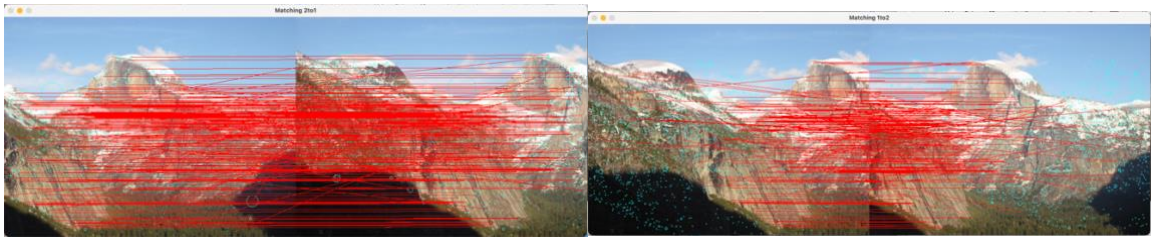


2) if ((dist1/dist2) > 0.85) continue;

```
input1 : 2865 keypoints are found.
input2 : 2623 keypoints are found.
(2to1): 241 keypoints are matched.
(1to2): 242 keypoints are matched.
```



아래는 feature matching 결과이다. 1 번 실행보다 outlier 가 많아, hw08 의두 결과 모두 잘못된 결과가 나온다



2. SIFT_RANSAC.cpp

코드 목적:

SIFT descriptor 를 이용해 특징점이 되는 keypoint 를 찾고, Affine Transform 을 수행한다. 이때, RANSAC 을 적용한다.

RANSAC 은 data 들 중 랜덤으로 몇 개를 골라 model parameters 들을 결정한 뒤, 이 model 과 threshold 이내에 존재하는 data 들의 개수 N 를 세고, S 번의 trial 중 가장 큰 N 을 가지는 model parameter 을 최종적으로 선택하는 과정이다. 이 과정을 통해 outlier 를 크게 줄일 수 있다.

1 번 코드와 같은 과정을 수행하지만, A21 과 A12 를 결정하는 과정이 다르다.

우선 #defined 으로 수를 정해주자 samplenumK 개를 뽑는 일을 trial 번 만큼 수행하고, 구한 Matrix 와의 차이가 thres 보다 차이나지 않는 data 를 셀 것이다.

```
#define trial 50
#define samplenumK 5
```

A21 과 A12 는 다음과 같은 과정으로 구한다.

```
Mat A12 = cal_affine<float>(ptl_x, ptl_y, ptr_x, ptr_y, n);
A12= cal_affine2<float>(ptl_x, ptl_y, ptr_x, ptr_y, n,A12);
Mat A21 = cal_affine<float>(ptr_x, ptr_y, ptl_x, ptl_y, n);
```

cal_affine 함수:

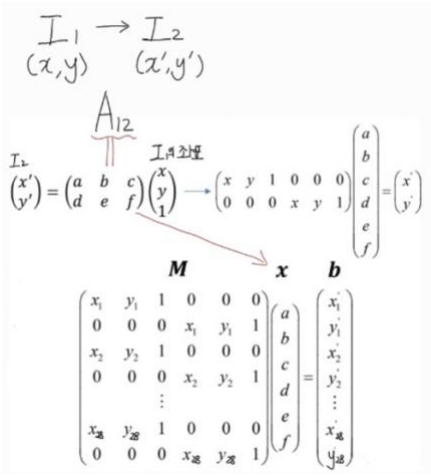
samplenumK 개수만큼 저장할 배열들을 선언한다.

rand()로 난수를 생성하고, number_of_points 로 나눠 0~number_of_points-1 사이의 난수 tmp 를 형성한다. tmp 에 해당하는 input1 의 x 좌표, y 좌표, input2 의 x 좌표, y 좌표를 저장한다.

```
int* ptl_x_sample = new int[samplenumK];      int* ptl_y_sample = new int[samplenumK];
int* ptr_x_sample = new int[samplenumK];      int* ptr_y_sample = new int[samplenumK];
Mat ans;

int MaxCnt=-1;
for(int i=0;i<trial;i++){
        Mat M(2 * samplenumK, 6, CV_32F, Scalar(0));
        Mat b(2 * samplenumK, 1, CV_32F);
        Mat M_trans, temp, affineM;
        int cnt=0;
        for(int i=0;i<samplenumK;i++){
```

랜덤으로 뽑힌 점들을 이용해 M 과 b 를 만들어주고, 이를 이용해 한 이미지 위의 점이 매핑되는 다른 이미지 위의 점의 좌표로 전환해주는 행렬을 계산한다.



$$Mx = b \rightarrow x = (M^{\mathrm{T}}M)^{-1}M^{\mathrm{T}}b$$

```
for (int j = 0; j < samplenumK; j++) {
        M.at<T>(2 * j, 0) = ptl_x_sample[j];                M.at<T>(2 * j, 1) = ptl_y_sample[j];
    M.at<T>(2 * j, 2) = 1;
        M.at<T>(2 * j + 1, 3) = ptl_x_sample[j];            M.at<T>(2 * j + 1, 4) = ptl_y_sample[j];
    M.at<T>(2 * j + 1, 5) = 1;
        b.at<T>(2 * j) = ptr_x_sample[j];           b.at<T>(2 * j + 1) = ptr_y_sample[j];
     }
```

구한 행렬을 이용해 점을 다른 이미지로 이동시켰을 때의 좌표와, feature matching 으로 구한 대응하는 좌표의 거리가 threshold 보다 적은 경우에 cnt 를 이용해 카운트를 해준다

```
        for(int j=0;j<number_of_points;j++){
            Point2f p(affineM.at<float>(0) * ptl_x[j] + affineM.at<float>(1) * ptl_y[j] +
        affineM.at<float>(2),affineM.at<float>(3) * ptl_x[j] + affineM.at<float>(4) *ptl_y[j]                    +
affineM.at<float>(5));

            float d=pow(p.x-ptr_x[j],2)+pow(p.y-ptr_y[j],2);
```

이 한번의 trial 에 대해, 구한 cnt 가 이전까지의 cnt 보다 크다면, MaxCnt 에 저장하고, ans 를 지금의 행렬로 업데이트해준다.

```
        if(cnt>MaxCnt){
            printf("%d\n",cnt);
            MaxCnt=cnt;
```

모든 trial 에 대한 for 문이 끝나면, ans 를 반환한다. 즉, threshold 보다 가까운 data 들의 개수가 가장 많았던 A12(또는 A21)을 반환한다. 이 과정을 통해, A21 와 A12 를 구할 때 outlier 가 무시된다.

```
    return ans;
```

cal_affine2 함수:

이제 구한 Matrix 에 대해, inlier 들로만 다시 A12, A21 을 계산해줄 것이다. cal_affine 함수에서는 랜덤으로 뽑힌 소수의 점만 사용되었으므로, 다시 inliers 들로만 계산해 정확한 Matrix 를 얻는다.

우선, inlier 인 점들의 개수 cnt 를 구한다

```
int cnt=0;
    for(int j=0;j<number_of_points;j++){
        Point2f p(affineM.at<float>(0) * ptl_x[j] + affineM.at<float>(1) * ptl_y[j] + affineM.at<float>(2),affineM.at<float>(3) * ptl_x[j] +
affineM.at<float>(4) *ptl_y[j] + affineM.at<float>(5));

        float d=pow(p.x-ptr_x[j],2)+pow(p.y-ptr_y[j],2);
        if(d<thres*thres){
```

구한 cnt 를 이용해 행렬들을 초기화해준다.

```
    Mat M(2 * cnt, 6, CV_32F, Scalar(0));
    Mat b(2 * cnt, 1, CV_32F);
```

모든 keypoints 를 돌며, inlier 인 경우에만 M 과 b 행렬을 채워준다

```
int i=0;
    for(int j=0;j<number_of_points;j++){
        Point2f p(affineM.at<float>(0) * ptl_x[j] + affineM.at<float>(1) * ptl_y[j] + affineM.at<float>(2),affineM.at<float>(3) * ptl_x[j] +
affineM.at<float>(4) *ptl_y[j] + affineM.at<float>(5));

        float d=pow(p.x-ptr_x[j],2)+pow(p.y-ptr_y[j],2);
        if(d<thres*thres){
            M.at<T>(2 * i, 0) = ptl_x[j];          M.at<T>(2 * i, 1) = ptl_y[j];          M.at<T>(2 * i, 2) = 1;
            M.at<T>(2 * i + 1, 3) = ptl_x[j];      M.at<T>(2 * i + 1, 4) = ptl_y[j];      M.at<T>(2 * i + 1, 5) = 1;
            b.at<T>(2 * i) = ptr_x[j];         b.at<T>(2 * i + 1) = ptr_y[j];
```

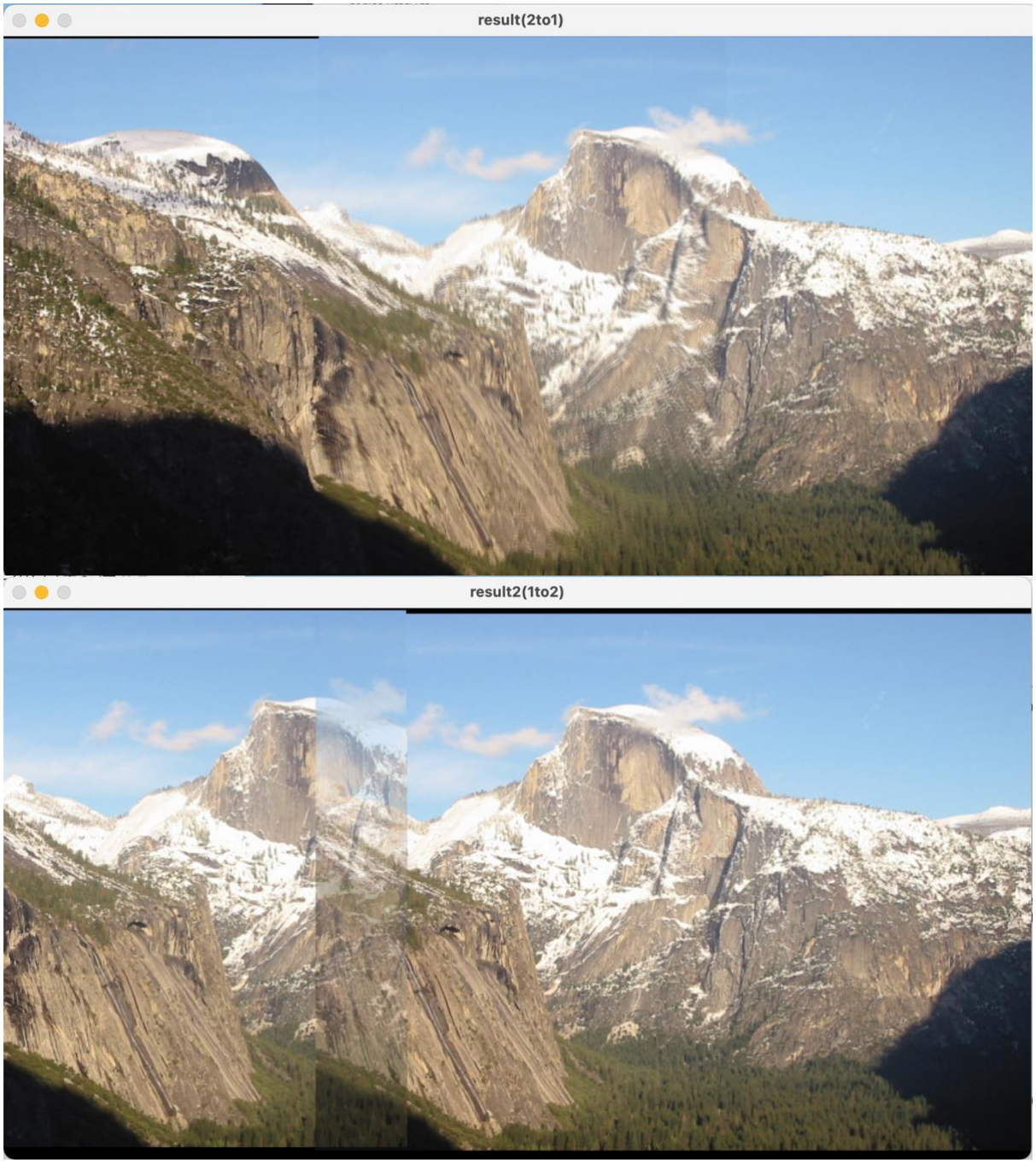
구한 M 과 b 에 대해 x 를 계산해 반환한다.

```
transpose(M, M_trans);                  //M_trans = M^T
    invert(M_trans * M, temp);              //temp = (M^T * M)^(-1)
    affineM = temp * M_trans * b;           //affineM = (M^T * M)^(-1) * M^T * b
```

즉, 이 코드는 1 번 코드와 같은 작업을 하지만, A12, A21 을 구할 때, RANSAC 을 이용해 outlier 를 제거했고, 그렇게 구한 행렬에 대해 다시한번 inlier 들로만 계산해 정확한 행렬을 계산해 낸다.

실행 결과:

```
if ((dist1/dist2) > 0.85) continue;
```



outlier 때문에 잘못된 결과가 나왔던 코드 1 번과 다르게, input1 을 input2 에 붙이는 작업과 input2 를 uinput1 에 붙이는 작업 모두 성공적인 것을 볼 수 있다.

3. Hough.cpp

목적: canny 함수를 이용해 egde 를 구하고, HoughLines 함수와 HoughLinesP 함수를 이용해 edge 를 선으로 그려낸다.

Hough Transform 은 point 들을 y=ax+b 의 x,y 에 대입한 후 a 와 b 의 값을 구하여 식을 그래프로 그린다. 이를 voting 한다고 한다. 이때 그래프들의 선이 겹치는 점들이 생기는데, 가장 많은 선이 겹치는 점을 구한다. 그런데 이 식은 y 의 계수가 1 로 고정되어 0 이 될 수 없으므로, r 과 각도 θ를 이용한 식을 이용한다. 극좌표는 연속적인 값이 아니도록 quantize 하는 과정을 통해 line fitting 을 할 수 있다.

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

canny 함수:

Canny(InputArray, OutputArray, threshold1, threshold2)

매개변수:

InputArray: input 이미지 행렬

OutputArray: 결과 반환할 행렬

threshold1, threshold2: 기준점

함수 목적: InputArray 를 Image Filtering(Low-pass & High-pass filters)한 뒤, Non-maximum suppression & Doouble thresholding 보정을 하여 OutputArray 에 반환한다

```
        //Standard Hough transform (using 'HoughLines')
#if 1
        vector<Vec2f> lines;
        //Fill this line
    HoughLines(dst, lines, 1, CV_PI / 180, 150, 0, 0);


        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);
```

HoughLines 함수:

매개변수: 입력이미지, 결과 반환 변수, r 의 정밀도, θ의 정밀도

반환받은 lines 에 대해 선을 긋는 과정이다. HoughLines 함수는 x,y 가 아닌 r 과 θ를 사용했기 때문에, 변환하는 과정이 필요하다

```
        //Probabilistic Hough transform (using 'HoughLinesP')
#else
        vector<Vec4i> lines;
        //Fill this line
    HoughLinesP(dst, lines, 1, CV_PI / 180, 50, 50, 10);
        for (size_t i = 0; i < lines.size(); i++)
        {
```

HoughLinesP 함수

HoughLines 함수와 다르게 뒤에 두 수가 추가되어있다.

50 은 이 인자값보다 작으면 선이 아니라는 조건값이고, 10 은 2 개의 선 사이의 거리가 이 값보다 작다면 하나의 선으로 인식하라는 조건값이다.

HoughLinesP 함수가 line 의 길이까지 생각하기 때문에 더 유용할 것이다 이를 수두 코드로 작성해 보자면 아래와 같다.

```
if( detectLines[i].length <50) break.
else
        if(distance( detectLines[i], detectLines[j]) <10)
                ans.push_back(detectLines[i].start, detectLines[j].end)
        else
                ans.push_back(detectLines[i].start, detectLines[i].end)
```

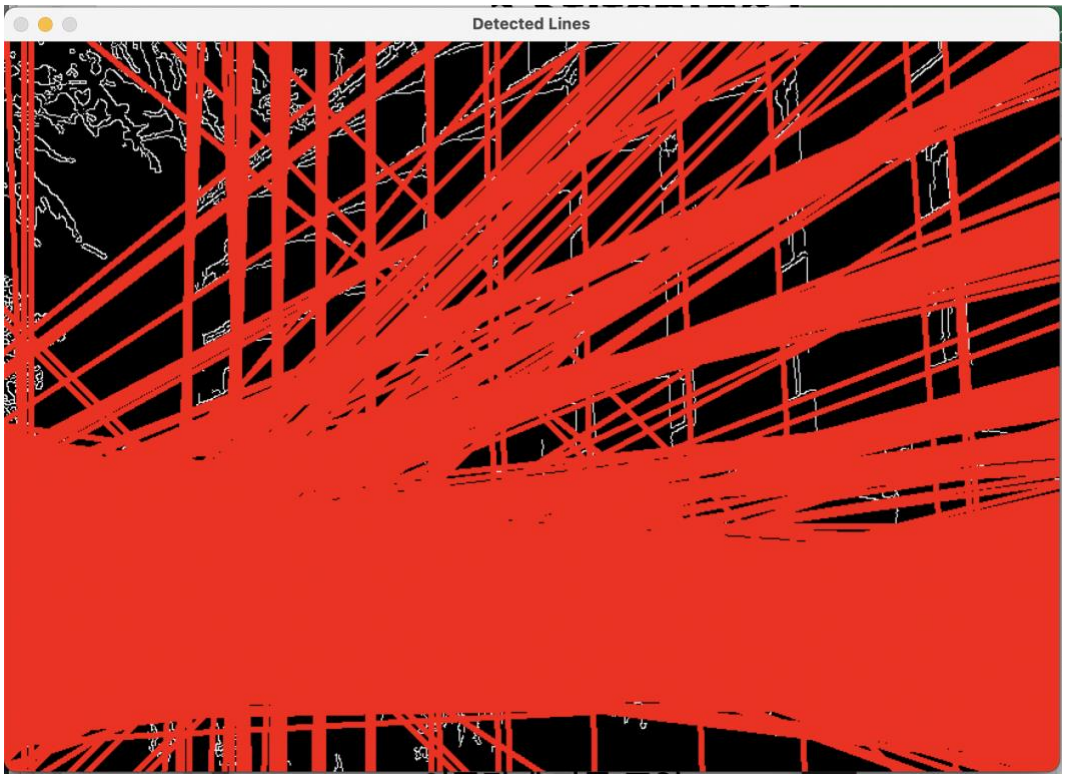
```
#if 0
        A
#else
        B
#endif
```

는 쉽게 주석처리하기 위한 도구로,

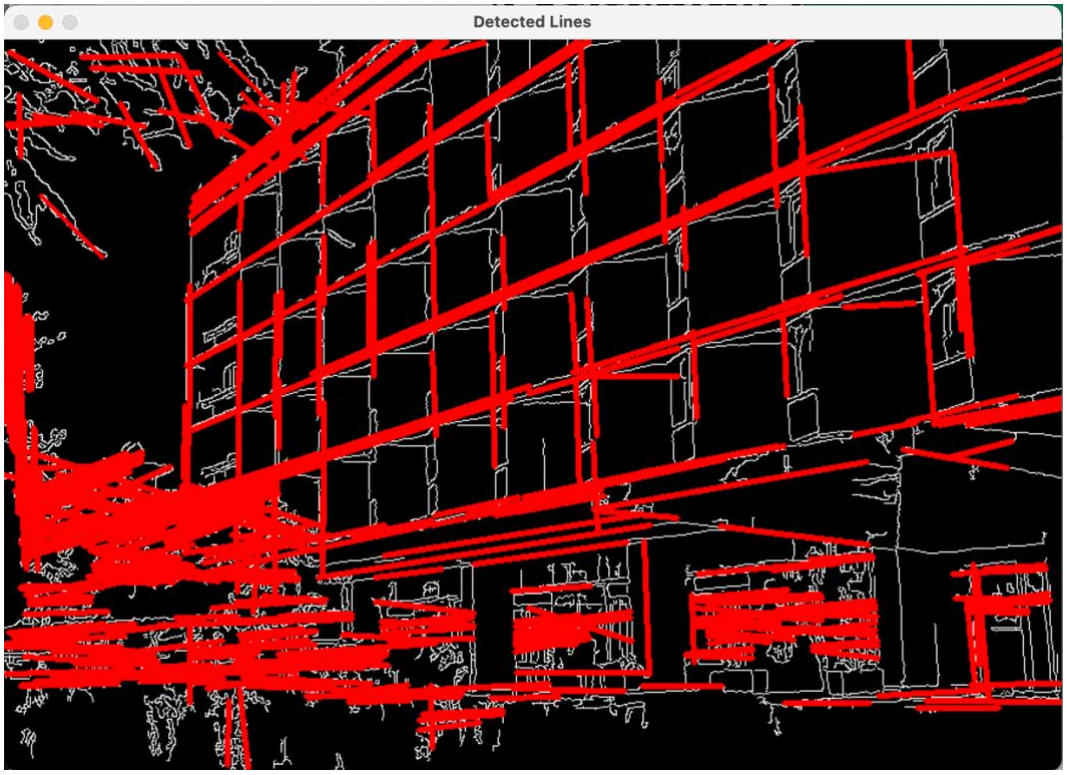
if 뒤의 수를 1 로 하면 A 가 실행, B 가 주석처리되며, 0 으로 하면 A 가 주석처리되어 B 가 실행된다.

실행 결과

1) #if 1



2) #if 0



참고자료:
오픈 SW 프로젝트 Lec09 수업자료