Image Stitching Using Affine Transform

- The purpose of the code

"stitchingAffine1to2.cpp"와 "stitchingAffine2to1.cpp"이 코드에서는 affine transform을 이용한 image stitching을 구현한다. Affine transform에 사용할 feature matched points는 지난 과제 6에서 구현한 SIFT descriptor를 활용하여 구하였다. SIFT descriptor의 정확도 향상을 위하여 cross-checking과 ratio-based thresholding 알고리즘을 적용하였고 feature matched points를 구하였다. 이를 이용하여 affine transformation의 matrix를 구하고 image stitching을 구현하는 것이 이 코드의 목적이다. 두개 코드의 차이는 input1에서 2로 feature matching을 하는지, 그 반대로 하는지 이다.

"ransacStitchingAffine1to2.cpp"와 "ransacStitchingAffine2to1.cpp"에서는 "stitchingAffine.cpp"에 추가로 RANSAC 알고리즘을 적용하여outlier를 제거하는 과정을 구현하였다. 두개 코드의 차이는 input1에서 2로 feature matching을 하는지, 그 반대로 하는지 이다.

RANSAC을 이용하지 않은 코드와 이용한 코드를 순서대로 case1과 case2로 나누어 설명을 하도록 하겠다.

Case1) Affine transform estimation

- The explanation of the code

SIFT descriptor로 feature matched points를 구한 후 affine transformation을 이용한 image stitching을 하도록 코드를 작성하였다.

```
71      // Find nearest neighbor pairs
72      vector<Point2f> srcPoints;
73      vector<Point2f> dstPoints;
74      bool crossCheck = true;
75      bool ratio_threshold = true;
76      findPairs(keypoints2, descriptors2, keypoints1, descriptors1, srcPoints, dstPoints, crossCheck, ratio_threshold);
77      printf("%zd keypoints are matched.\n", srcPoints.size());
```

77번째 줄의 코드까지 디버그를 하면, 두개의 input이미지의 matched points가 srcPoints와 dstPoints에 값이 들어가게 된다. cross-checking과 ratio-based thresholding를 적용하였기 때문에 나름 정확한 matched points를 사용할 수 있게 된다.

77번째 이후 줄의 코드부터는 affine transformation을 한 후, image stitching을 하는 코드이다.

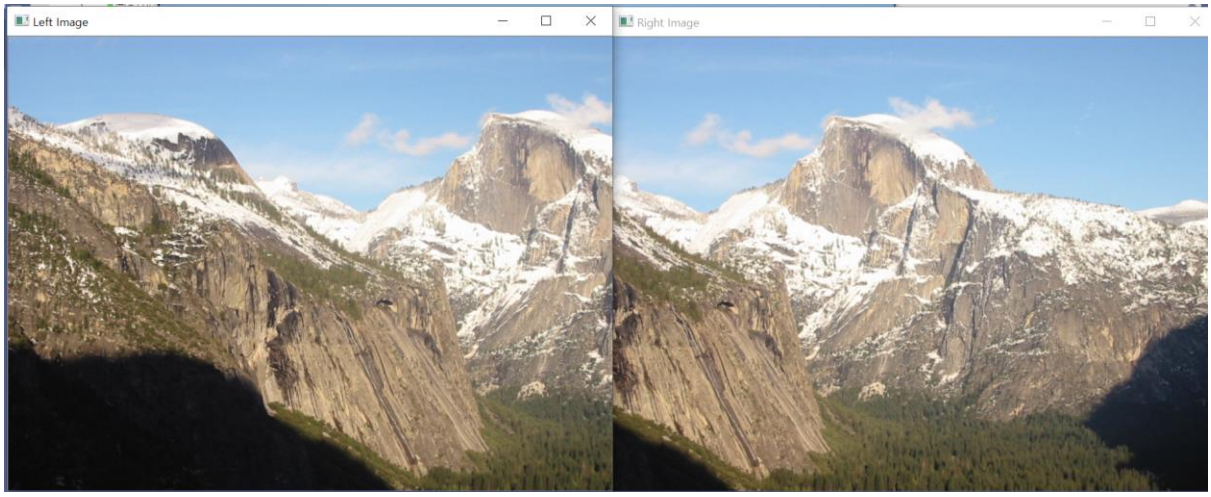
```
int* ptl_x = new int[n];
int* ptl_y = new int[n];
int* ptr_x = new int[n];
int* ptr_y = new int[n];

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

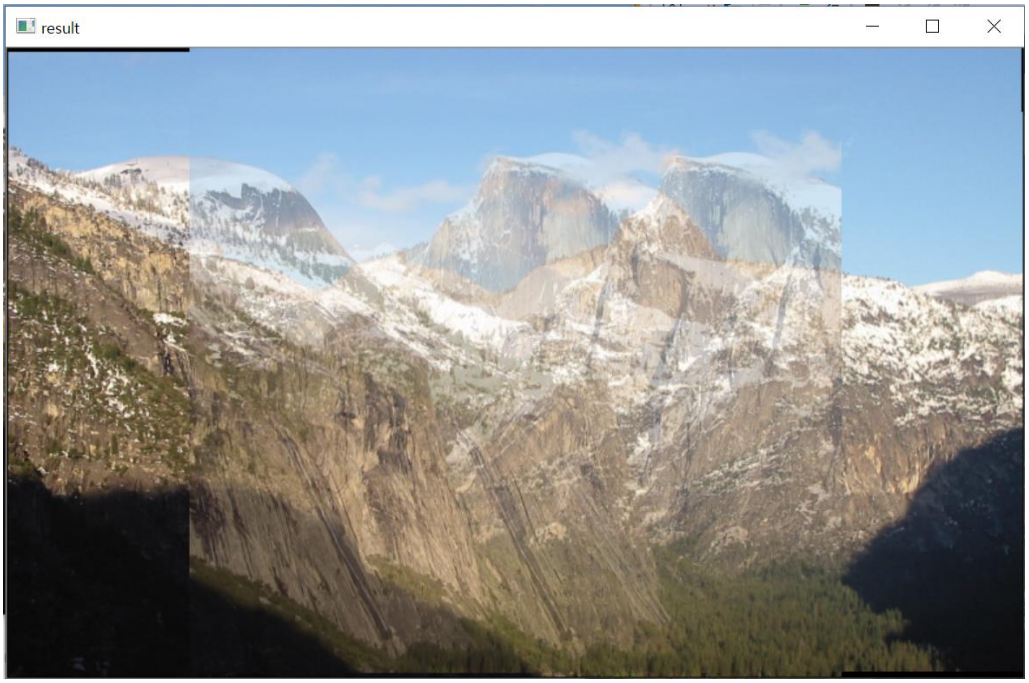
앞에서 구한 srcPoints와 dstPoints를 affine transformation을 위한 matrix를 구하기 위해서는 각 이미지 matched points의 x와 y값을 다른 리스트에 넣어서 분리해주어야 한다. 만약 ptl_x에 dstPoints.x 또는 srcPoints.x를 넣게 되면 image stitching이 잘못되어 결과 이미지를 출력하기 때문에 주의해야 한다.

나머지 코드는 앞의 과제에서 제출한 코드와 동일하다. 또한 Feature matching을 만약 image1에서 2가 아닌 그 반대로 코드를 실행하고 싶은 경우, 아래의 코드에서 input1과 input2의 위치를 바꿔주면 된다. 이외의 다른 코드는 동일하며 이에 대한 코드 설명은 생략한다.

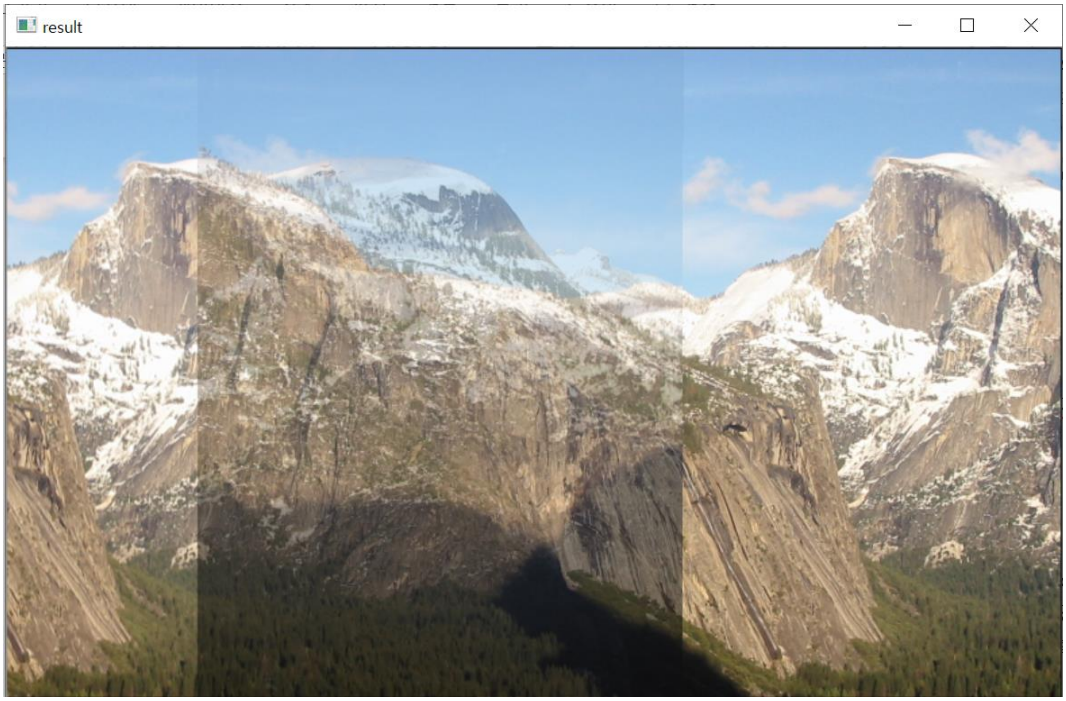
- Input image and Results of the code



1) Input image를 Resize하지 않은 경우 (1 -> 2 / 2 -> 1)

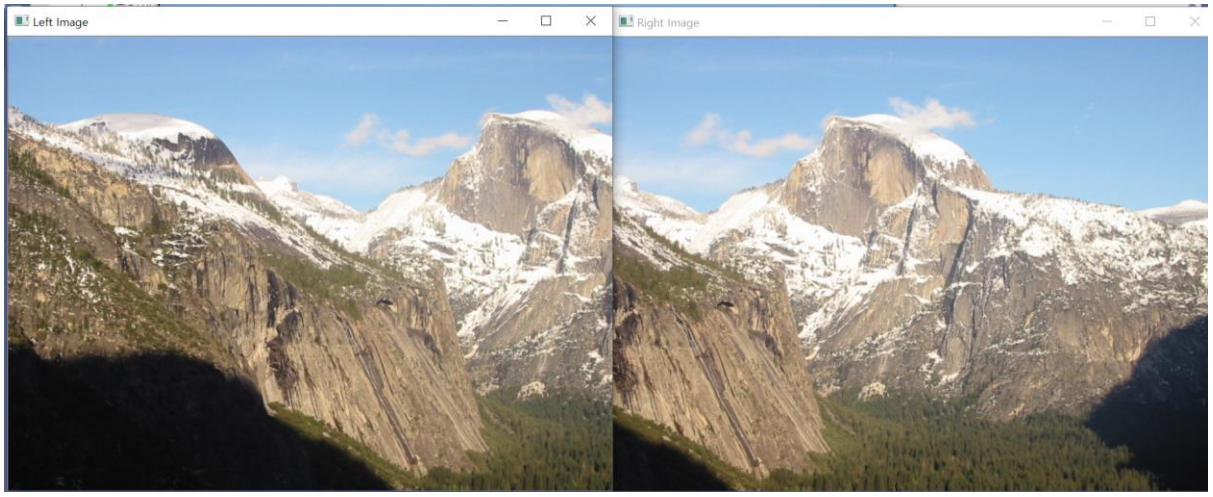


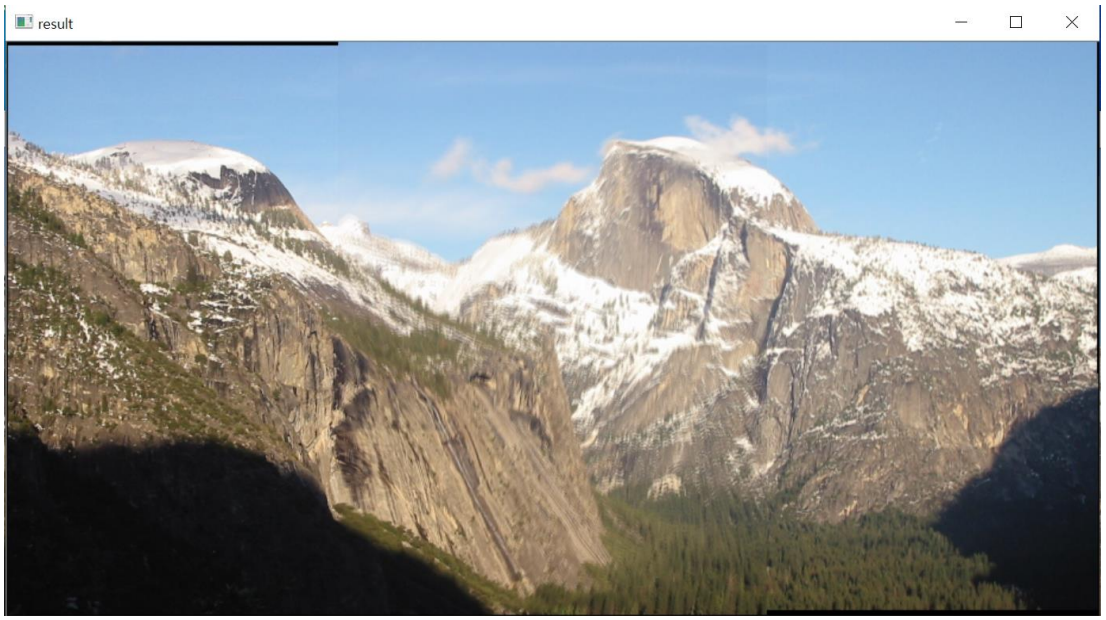

```
Microsoft Visual Studio 디버그 콘솔
input1 : 703 keypoints are found.
input2 : 619 keypoints are found.
209 keypoints are matched.
```




```
C:₩Users₩100ji₩source₩repos₩Project1₩x64₩Debug₩Project1.exe
input1 : 619 keypoints are found.
input2 : 703 keypoints are found.
205 keypoints are matched.
```

2) Input image를 Resize한 경우 (1 -> 2)





- Analysis of the result

Resize한 경우와 하지 않은 경우 matched points의 개수가 달라짐에 따라 결과 이미지의 정확도 차이가 나는 것을 확인할 수 있었다.

Case2) affine transform estimation with RANSAC

- The explanation of the code

RANSAC을 하는 방법은 다음과 같다. 3이상의 k개를 matching된 point에서 랜덤하게 뽑는다.

```
// ######################## RANSAC ###############################

srand((unsigned int)time(NULL));
Mat I1, I2;
int k = 4;
int s = 5;
int threshold = 9;
```

여기서는 k값을 4, s값을 5, threshold를 9로 설정하였다. (s와 threshold에 대해서는 뒤에서 설명.)

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

```
for (int j = 0; j < k; j++) {
    int index = rand() % n;
    templ_x[j] = ptl_x[index];
    templ_y[j] = ptl_y[index];
    tempr_x[j] = ptr_x[index];
    tempr_y[j] = ptr_y[index];
}
```

Affine transformation 함수에 넣어서 matrix를 구해야 하기 때문에, random하게 뽑은 point들은 왼쪽 이미지, 오른쪽 이미지, x, y를 구분하여서 각각 temp list에 넣어준다.

```
for (int t = 0; t < k; t++) {
    float x = tempA12.at<float>(0) * templ_x[t] + tempA12.at<float>(1) * templ_y[t] + tempA12.at<float>(2);
    float y = tempA12.at<float>(3) * templ_x[t] + tempA12.at<float>(4) * templ_y[t] + tempA12.at<float>(5);
    float x_ = tempr_x[t];
    float y_ = tempr_y[t];
    float comp_x = pow(abs(x - x_), 2);
    float comp_y = pow(abs(y - y_), 2);
    if (comp_x < threshold && comp_y < threshold) {
        count++;
        inliner_x[count] = templ_x[t];
        inliner_y[count] = templ_y[t];
        bestPR_x[count] = tempr_x[t];
        bestPR_y[count] = tempr_y[t];
```

tempA12에는 cal_affine 함수를 이용하여 affine transformation을 가능하게 하는 matrix가 들어있다. 이때 | x - x_|^2을 계산한 값이 threshold값보다 작으면 해당 point는 inliner인 것임을 알 수 있게 된다. 이렇게 inliner인 point값들을 저장하고, 이 과정을 s번 반복한다. Inliner가 제일 많은 transformation matrix를 선택한다. 이것이 RANSAC 알고리즘을 구현하는 방식이다.

```
// calculate affine Matrix A12, A21
Mat A12 = cal_affine<float>(bestPL_x, bestPL_y, bestPR_x, bestPR_y, n);
Mat A21 = cal_affine<float>(bestPR_x, bestPR_y, bestPL_x, bestPL_y, n);
```

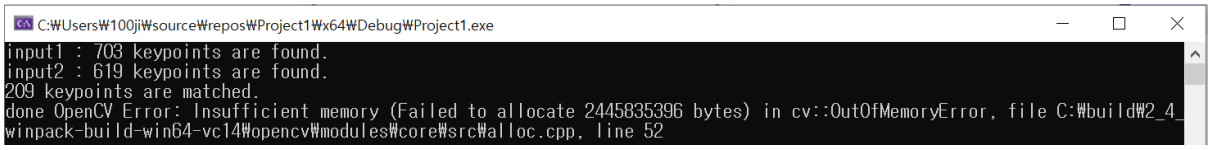
여기에 추가로 더 좋은 성능의 결과 이미지를 얻기 위하여 inliner로만 구성된 list로 affine transformation을 다시 계산하였다. 그런 후에 image stitching을 하여 코드를 완성하였다.

Feature matching을 만약 image1에서 2가 아닌 그 반대로 코드를 실행하고 싶은 경우, 아래의 코

드에서 input1과 input2의 위치를 바꿔주면 된다. 이외의 다른 코드는 동일하며 이에 대한 코드 설명은 생략한다.

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

- Results of the code



- Analysis of the result

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

위와 같이 오류가 난 이유를 파악해보니 위의 코드를 실행하는 시점에서 오류가 발생하는 것을 파악하였다. 메모리가 부족하다는 에러 메시지 확인 후, 아래와 같이 메모리를 사용하고 있는 변수들의 메모리를 release 시켜주었으나 에러가 발생하였고 결과 이미지를 확인하지 못했다.

```
delete inliner_x;
delete inliner_y;
```

```
input1.release();
input2.release();
```

```
srcPoints.shrink_to_fit();
dstPoints.shrink_to_fit();
```

Line Fitting Using Hough Transform

- Explanation of how to estimate the line segments in the Hough transform

Hough transform을 이용한 line fitting을 하는 방법은 다음과 같다. image의 point들을 y=ax+b라는 식의 x와 y값에 대입한 후에 a와 b에 대한 식을 그래프로 그린다. 이때 그래프의 선들이 겹치는 점이 존재하는데, 가장 많이 겹치는 점을 구한다는 것이 핵심 개념이다. 수직선을 표현할 수 있도록 극좌표를 사용하고 연속적인 값이 아니도록 quantize하는 과정을 통해 line fitting을 할 수 있다.

```
def houghTransform:
    for point in inputImage:
        for x, y in point:
     for i in range(num_of_point):
        for j in range(4):            # angle: -45, 0, 45, 90 => 4 value
                        r [ i ] [ j ] = x*cosθ + y*sinθ
    sort(r)
    max=find_max(r)
    min=find_min(r)
    for min to max:    # min ~ max of r value
```

```
                if r_value_at_matrix_r==row && anlgle_of_point==column:
                    table[row][column]++                    # row: r value, column: angle
            result=find_all_max(table)
            return result
```

이와 같이 Hough transform의 pseudo code를 작성해보았다.

- Run the Hough transform using OpenCV functions

```
#if 0
    vector<Vec2f> lines;
    //Fill this line
    HoughLines(dst, lines, 1, CV_PI / 180, 100, 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);
        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);
    }

    //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++)
    {
        line(color_dst, Point(lines[i][0], lines[i][1]),
            Point(lines[i][2], lines[i][3]), Scalar(0, 0, 255), 3, 8);
    }
```

HoughLines함수와 HoghLinesP함수를 사용하여 line fitting을 해본 결과 HoughLines함수로 나온 결과의 경우 만족할 만한 line fitting이 이루어지지 않아, HoughLinesP를 사용하도록 if 문 조건을 1에서 0으로 변경하였다. 그 결과 아래와 같은 이미지를 얻을 수 있었다.