REPORT

Assignment #3

수강과목	전기전자심화설계및소프트웨어실습(2014)
담당교수	김원준
학 과	전기전자공학부
학 번	201810909
이 름	이재현
제출일자	2022. 10. 4(화)

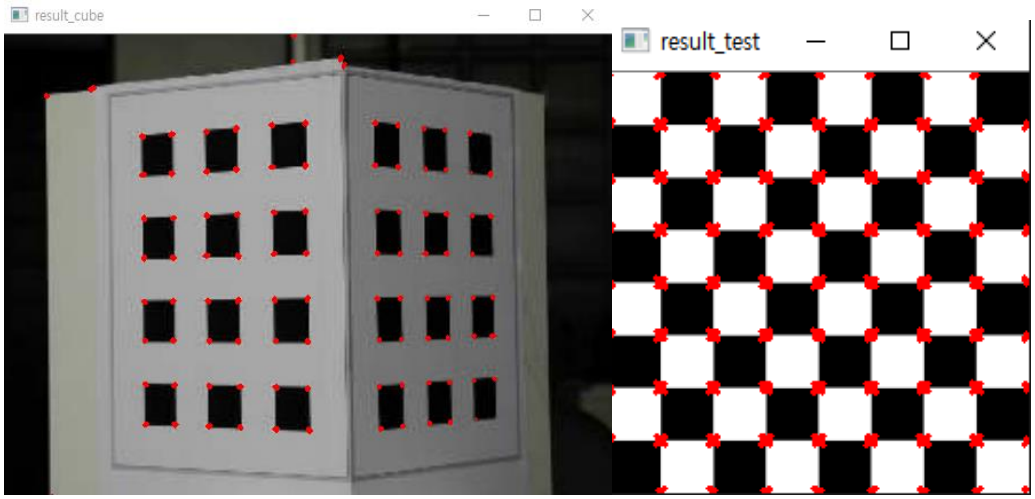
1. Harris Corner Detection

"A combined corner and edge detctior" 에서 Harris, Stephens 가 제안한 Corner 검출 방법론으로 하나의 픽셀을 중심으로 하는 window 를 각각의 축 방향으로 움직였을 때 픽셀 값의 차이가 얼마나 크게 발생하는 지를 계산하여 Corner 를 detection 하는 것이 기본 원리이다. 직관적으로 Corner pixel 을 중심으로 한 window 에서 픽셀 값의 차이가 가장 크게 나타날 것임을 알 수 있다. 이는 tayler series extension 에 의해 아래와 같이 approximation 할 수 있고 이를 matrix notation 으로 정리하면 아래와 같다.

$$E(u,v) = \sum_{(x_k,y_k)\in W}[I(x_k+u, y_k+v) - I(x_k,\ y_k)]^2 \cong \sum_{(x_k,y_k)\in W}\left(f_x^2 + f_y^2 + 2f_x f_y\right)$$

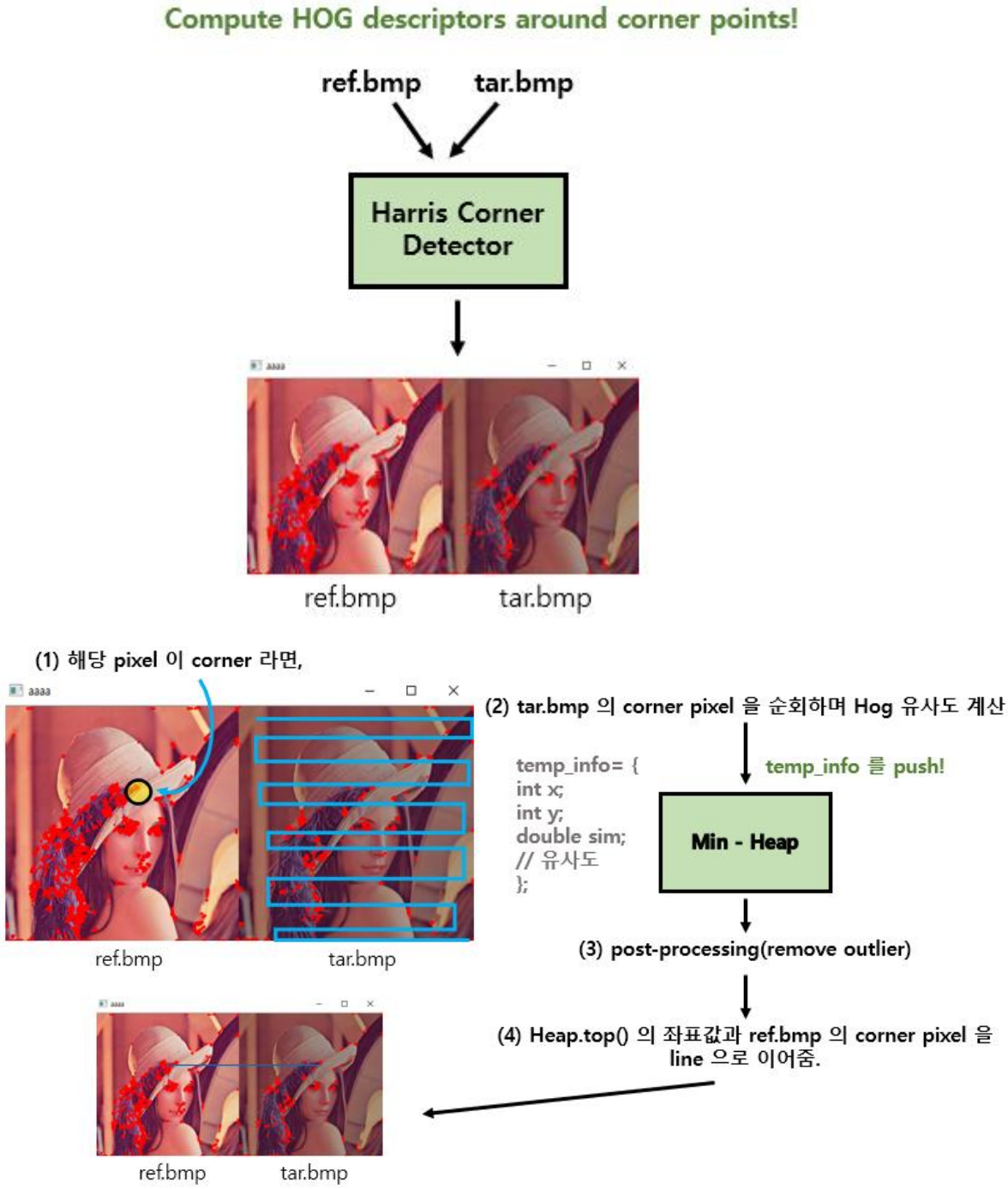
$$E(u,v) = [u\ v]\begin{bmatrix}\sum f_x^2 & \sum f_x f_y \\ \sum f_x f_y & \sum f_y^2\end{bmatrix}\begin{bmatrix}u \\ v\end{bmatrix}$$

E(u, v) 값이 크려면 가운데에 위치한 2 by 2 행렬이 커야 하는데, 이 행렬을 structure tensor 이라고 부른다. 양 방향으로 변화가 클 때, corner point 이므로 structure tensor 을 고유값 분해 하였을 때 나오는 고유값들이 모두 충분히 커야 한다. 다만 고유값 계산은 연산량이 많으므로 아래와 같이 대체하여 구해낼 수 있다. $R = \det(M) - k\left(trace(M)\right)^2$ structure tensor 의 고유값이 모두 크면 R > 0 이다. 이를 구현하였을 때, 아래와 같이 corner point 가 잘 검출되는 모습을 확인할 수 있었다. Harris corner dector 는 밝기 변화에 robust 하지만 scale 변화에는 취약하다는 특징이 있다.



2. Compute HOG descriptors around corner points.

본 과제에서 구현해야하는 프로그램은 아래와 같다. 먼저, ref.bmp 와 tar.bmp 에 대해서 Harris corner detection 을 수행하여 corner points 를 검출한다.



ref.bmp 에서 corner points 를 검출했다면, tar.bmp 의 corner point 들을 순회하며 HOG 히스토그램을 구해서 ref.bmp 에서의 corner point 와 가장 유사한 corner pixel 을 찾아서 line 으로 이어준다. 이를 위해서 해야 하는 task 는 먼저 HOG histogram 을 local 하게 동작하도록 modify 해야한다. tar.bmp 에서

corner point 를 검출하면, 해당 픽셀의 위치를 center 로 하는 17 by 17 block 에서 hog histogram 을 반환하는 함수를 구현해야 한다.

```
float* get_hog_histogram_around_corner(Mat input, int pos_x, int pos_y, float* mag, float* dir_arr) {
    int height = input.rows;
    int width = input.cols;

    int idx = 0;
    // block-based hog algorithm(modified)
    float* temp_histogram = (float*)calloc(9, sizeof(float));

    fill(temp_histogram, temp_histogram + 9, 0); // set all zero
    double dir_val = 0.0;
    for (int m = pos_y-BLK/2; m < pos_y + BLK/2; m+=1) { // calculate histogram at each block
        for (int n = pos_x-BLK/2; n < pos_x + BLK/2; n+=1) {
            if (m < 0 || m >= input.rows || n < 0 || n >= input.cols) continue; // out of bounds
            dir_val = dir_arr[m * width + n];
            temp_histogram[(int)dir_val / 20] += mag[m * width + n];
        }
    }

    // L-2 normalization
    float normalization_sum = 0.0;
    for (int i = 0; i < 9; i++) { normalization_sum += temp_histogram[i] * temp_histogram[i]; }
    normalization_sum = sqrt(normalization_sum);
    for (int i = 0; i < 9; i++) {
        if (normalization_sum == 0) break;
        temp_histogram[i] /= normalization_sum;
    }

    return temp_histogram; // return histogram
}
```

HOG histogram 을 계산하기 위해서 필요한 magnitude 값과 direction 값이 저장된 배열은 parameter 로 받도록 하였다. 또한 corner point 에 해당하는 pos_x, pos_y 또한 parameter 로 전달받도록 하여, tar.bmp 의 corner point 를 중심으로 하는 17 by 17 Block 에 대해 locally 하게 HOG Histogram 을 계산할 수 있도록 modify 하였다.

histogram 의 유사도는 Eculidean distance(L2-norm) 을 사용하였다.

```
float get_similarity(float* obj1, float* obj2, int size) {
    float score = 0.0;

    for (int i = 0; i < size; i++) {
        score += (abs(obj1[i] - obj2[i]))* (abs(obj1[i] - obj2[i]));
    }
    score = sqrt(score);
    //score = sqrt(score);
    // use Euclidean distance
    // 더 작을수록 유사도가 높은 것
    return score;
}
```

ref.bmp 에서 corner point 를 검출했다면 tar.bmp 의 corner point 들을 순회하며 가장 유사한 pixel 을 찾아야 한다. 이를 위해서 사용한 아이디어는 우선순위큐(priority queue)를 이용해서 최소 유사도를 가지는 points 를 검출해내는 것이다.

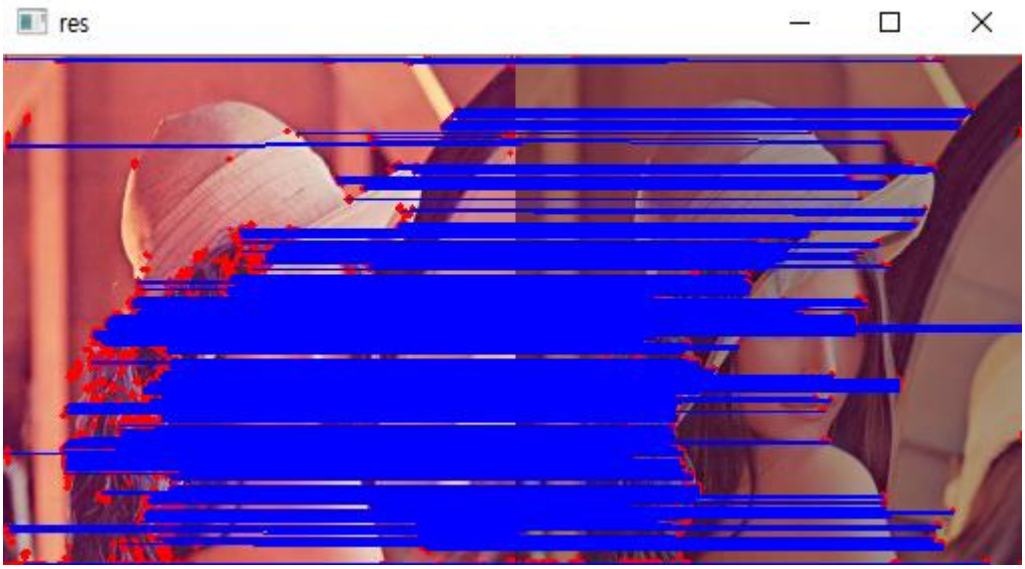
```
struct sim_x_y {
    int x; // 좌표
    int y;
    double sim; // 유사도
};

struct compare { // similiarity 가 작은 애가 더 유사한거임.
    bool operator()(const sim_x_y& m1, const sim_x_y& m2) {
        return m1.sim > m2.sim;
    }
};

priority_queue <sim_x_y, vector<sim_x_y>, compare> pq;
```

이를 위해 위와 같이 좌표값과 유사도를 member 로 가지는 struct 인 sim_x_y 를 선언한 뒤, 이를 element 으로 가지는 priority_queue pq 를 선언한다. 중요한 것은 유사도인 sim 은 L2 - norm 으로 계산되므로 작을 수록 더 유사하다는 것이므로 pq 의 정렬 기준을 modify 해주어야 하므로 위와 같이 compare class 를 정의하여 사용하였다.

```
for (int i = 0; i < ref.rows; i++) {
    for (int j = 0; j < ref.cols; j++) {
        if (ref_cornerMap.at<uchar>(i, j) == 0) continue; // reference 에서 코너가 아니면 pass
        histo1 = get_hog_histogram_around_corner(ref, j, i, ref_mag, ref_dir_arr); // 코너이면 histogram 추출

        while(!pq.empty())pq.pop();
        for (int m = 0; m < tar.rows; m++) {
            for (int n = 0; n < tar.cols; n++) {
                if (tar_cornerMap.at<uchar>(m, n) == 0) continue; // target 에서 코너가 아니면 pass
                sim_x_y temp_info = {0,0,0.0}; float sim = 0.0;
                // target 에서 코너 point 이면 histogram 추출
                histo2 = get_hog_histogram_around_corner(tar, n, m, tar_mag, tar_dir_arr);
                sim = get_similarity(histo1, histo2, 9); // 유사도 추출

                temp_info.sim = sim;
                temp_info.x = n;
                temp_info.y = m;

                pq.push(temp_info); // min_heap 에 넣기

            }
        }
        if (!pq.empty() && pq.top().sim < 0.05)
        {

            cout << "sim : " << pq.top().sim << "\n";
            tar_cornerMap.at<uchar>(pq.top().y, pq.top().x) = 0;
            ref_cornerMap.at<uchar>(pq.top().y, pq.top().x) = 0;
            line(result, Point(j, i), Point(pq.top().x + ref.cols, pq.top().y), Scalar(255, 0, 0), 1, 8, 0);
            imshow("result", result);
            imshow("ref_c", ref_cornerMap);
            imshow("tar_c", tar_cornerMap);
            cv::waitKey(10);

            while (!pq.empty()) pq.pop();
        }
    }
}
```

이후에는 위와 같이 ref.bmp 에서 corner point 이라면 tar.bmp 의 corner point 를 순회하며 HOG histogram 을 계산하고, 유사도를 얻는다. 유사도와 좌표를 담은 temp_info 를 pq 에 push 하는 것을 반복한다. 이렇게 하면 tar.bmp 를 모두 순회하게 되면 pq 의 top 에는 가장 최소 L2-norm 값을 가진

temp_info 가 남는다. 이제 이에 대해서 post-processing 을 통해 outlier 을 제거해주고, pq.top() 에 저장된 좌표값과 ref.bmp 의 corner point 를 line 으로 이어주는 것을 반복한다. 그러면 아래와 같이 matching 된 point 들이 line 으로 연결되는 모습을 확인할 수 있다.



3. Discussion

본 과제에서는 Harris corner detector 으로 검출한 corner point 에 대해서 HOG descriptor 으로 ref.bmp 와 tar.bmp 의 유사도가 가장 좋은 corner point 를 matching 하는 프로그램을 구현하였다. ref.bmp 의 corner point 와 가장 유사한 tar.bmp 의 corner point 를 찾아 matching 하기 위해 좌표값과 유사도를 member 로 가지는 구조체인 temp_info 를 element 로 가지는 priority_queue 를 선언하고, 유사도 값이 가장 작을 수록 top 에 위치하도록 compare 함수를 modify 하여 사용하였고, 가장 유사한 point 를 찾아낸 다음에 line 으로 연결시켜서 matching 하는 프로그램을 구현할 수 있었다. HOG Algorithm 으로 얻어낸 gradient based feature 을 토대로, corner point 를 matching 할 수 있었다.

4. Full source code

```
#define _CRT_SECURE_NO_WARNINGS

// 2022. 10. 04. 전기전자공학부 이재현

#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>

#include <iostream>
#include <stdio.h>

#include <queue>
#include <math.h>


#define PI 3.14159265358979323846264338327950288419716939
#define BLK 17 // Block size

using namespace cv;
using namespace std;

Mat cvtBGR2Gray(Mat Image) {
	int height, width;
	height = Image.rows, width = Image.cols;
	Mat result(height, width, CV_8UC1);
	for (int i = 0; i < height; i++) {
		for (int j = 0; j < width; j++) {
			result.at<uchar>(i, j) = (Image.at<Vec3b>(i, j)[0] + Image.at<Vec3b>(i, j)[1]
+ Image.at<Vec3b>(i, j)[2]) / 3;
		}
	}
	return result;
}
void getGradientMap(float** X, float** Y, Mat input) {
	int x, y, xx, yy;
	int height = input.rows;
	int width = input.cols;
	float conv x, conv y;

	int mask_x[9] = { -1, -2, -1, 0, 0, 0, 1, 2, 1 };
	int mask_y[9] = { -1,0,1,-2,0,2,-1,0,1 };

	float min = 1000000, max = -1;

	for (y = 0; y < height; y++) {
		for (x = 0; x < width; x++) {
			// cur
			conv_x = 0;
			conv_y = 0;

			for (yy = y - 1; yy <= y + 1; yy++) {
				for (xx = x - 1; xx <= x + 1; xx++) { // calc conv_x, conv_y
					if (yy >= 0 && yy < height && xx >= 0 && xx < width) {
						// indexing 에 주의!
						conv_x += input.at<uchar>(yy, xx) * mask_x[(yy -
(y - 1)) * 3 + (xx - (x - 1))];
						conv_y += input.at<uchar>(yy, xx) * mask_y[(yy -
(y - 1)) * 3 + (xx - (x - 1))];
					}
				}
			}
			conv_x /= 9.0;
			conv_y /= 9.0; // scaling
			X[y][x] = conv_x;
			Y[y][x] = conv_y;
			// mag[y * width + x] = sqrt(conv_x * conv_x + conv_y * conv_y); // calc
magninute
			// if (max < mag[y * width + x]) max = mag[y * width + x];
			// if (min > mag[y * width + x]) min = mag[y * width + x];
		}
	}
}
```

```
// 조명 변화에 robust 하지만 scale 변화에 취약
Mat Harris_CornerDetect(Mat img, float k, Mat cornerMap, int thresholding) {
	// img - Input image. It should be grayscale and float32 type.
	// k - Harris detector free parameter in the equation.
	Mat gray_img(img.rows, img.cols, CV_8UC1);
	Mat result = img.clone();
	if (img.channels() == 3) // gray
		gray_img = cvtBGR2Gray(img);
	else
		gray_img = img;
	// gaussian filtering to improve performance.
	int gaussian_mask[9] = { 1,2,1,2,4,2,1,2,1 };
	for (int y = 0; y < gray_img.rows; y++) {
		for (int x = 0; x < gray_img.cols; x++) {
			// cur
			float conv = 0.0;

			for (int yy = y - 1; yy <= y + 1; yy++) {
				for (int xx = x - 1; xx <= x + 1; xx++) { // calc conv_x, conv_y
					if (yy >= 0 && yy < gray_img.rows && xx >= 0 && xx < 
gray_img.cols) {
						// indexing 에 주의!
						conv += gray_img.at<uchar>(yy, xx) * 
gaussian_mask[(yy - (y - 1)) * 3 + (xx - (x - 1))];
					}
				}
			}
			gray_img.at<uchar>(y, x) = conv / 16.0f;
		}
	}
	// calculate gradient
	float** grad_x = (float**)calloc(gray_img.rows, sizeof(float*));
	float** grad_y = (float**)calloc(gray_img.rows, sizeof(float*));
	float** r_map = (float**)calloc(gray_img.rows, sizeof(float*));
	for (int i = 0; i < gray_img.rows; i++) {
		grad_x[i] = (float*)calloc(gray_img.cols, sizeof(float));
		grad_y[i] = (float*)calloc(gray_img.cols, sizeof(float));
		r_map[i] = (float*)calloc(gray_img.cols, sizeof(float));
	}
	getGradientMap(grad_x, grad_y, gray_img);

	//compute R at every pixel position
	float txx, txy, tyy;
	float det = 0.0; float tr = 0.0;
	for (int i = 1; i < gray img.rows - 1; i++) {
		for (int j = 1; j < gray_img.cols - 1; j++) {
			txx = 0.0, txy = 0.0, tyy = 0.0;

			for (int y = 0; y < 3; y++) { // use 3x3 window
				for (int x = 0; x < 3; x++) {
					txx += grad_x[i + y - 1][j + x - 1] * grad_x[i + y - 1][j + 
x - 1];
					txy += grad_x[i + y - 1][j + x - 1] * grad_y[i + y - 1][j + 
x - 1];
					tyy += grad_y[i + y - 1][j + x - 1] * grad_y[i + y - 1][j + 
x - 1];
				}
				det = txx * tyy - txy * txy;
				tr = txx + tyy;
				r_map[i][j] = det - k * tr * tr;
			}
		}
	}
	// non - maxima suppression
	for (int i = 1; i < gray_img.rows - 1; i++) {
		for (int j = 1; j < gray_img.cols - 1; j++) {
			// thresholding
			float max = 0.0;
			if (r_map[i][j] < 0) continue;
			for (int x = 0; x < 3; x++) {
				for (int y = 0; y < 3; y++) {
					if (max < r_map[x][y])
						max = r_map[x][y];
				}
			}
```

```
                                for (int x = 0; x < 3; x++) {
                                        for (int y = 0; y < 3; y++) {
                                                if (r_map[x][y] < max)
                                                        r_map[x][y] = 0;
                                        }
                                }
                        }
                }

                Scalar c;
                Point pCenter;
                float radius = 0.8;
                c.val[0] = 0, c.val[1] = 0, c.val[2] = 255;
                for (int i = 1; i < gray_img.rows - 1; i++) {
                        for (int j = 1; j < gray_img.cols - 1; j++) {
                                // thresholding
                                if (r_map[i][j] > thresholding) {
                                        pCenter.x = j;
                                        pCenter.y = i;
                                        circle(result, pCenter, radius, c, 2, 8, 0);
                                        cornerMap.at<uchar>(i, j) = 255;
                                }
                        }
                }
                free(grad_y);
                free(grad_x);
                free(r_map);
                return result;
}
float* get_hog_histogram_around_corner(Mat input, int pos_x, int pos_y, float* mag, float* dir_arr) {
                int height = input.rows;
                int width = input.cols;

                int idx = 0;
                // block-based hog algorithm(modified)
                float* temp_histogram = (float*)calloc(9, sizeof(float));

                float dir_val = 0.0;
                for (int m = pos_y-BLK/2; m <= pos_y + BLK/2 ; m++) { // calculate histogram around corner points
                        for (int n = pos_x-BLK/2; n <= pos_x + BLK/2 ; n++) {
                                if (m < 0 || m >= input.rows || n < 0 || n >= input.cols) continue; // out of
bounds
                                dir_val = dir_arr[m * width + n];
                                temp_histogram[(int)(dir_val / 20)] += (float)mag[m * width + n];
                        }
                }

                // L-2 normalization
                float normalization_sum = 0.0;
                for (int i = 0; i < 9; i++) { normalization_sum += (temp_histogram[i] * temp_histogram[i]); }
                normalization_sum = sqrt(normalization_sum);
                for (int i = 0; i < 9; i++) {
                        if (normalization_sum == 0) continue;
                        temp_histogram[i] /= normalization_sum;
                }

                return temp_histogram; // return histogram
}
float get similarity(float* obj1, float* obj2, int size) {
                float score = 0.0;

                for (int i = 0; i < size; i++) {
                        score += (abs(obj1[i] - obj2[i]))* (abs(obj1[i] - obj2[i]));
                }
                score = sqrt(score);
                //score = sqrt(score);
                // use Euclidean distance
                // 더 작을수록 유사도가 높은 것
                return score;
}
Mat HOG_arround_conner(Mat ref, Mat tar, float k) {
                int x, y, xx, yy;
                int height = ref.rows;
                int width = ref.cols;
                float conv_x, conv_y, dir;
```

```
		float* ref_mag = (float*)calloc(height * width, sizeof(float));
		float* ref_dir_arr = (float*)calloc(height * width, sizeof(float));
		float* tar_mag = (float*)calloc(height * width, sizeof(float));
		float* tar_dir_arr = (float*)calloc(height * width, sizeof(float));

		int mask_x[9] = { -1, -2, -1, 0, 0, 0, 1, 2, 1 }; // sobel mask
		int mask_y[9] = { -1,0,1,-2,0,2,-1,0,1 };


		Mat result(ref.rows, ref.cols*2, CV_8UC3);
		Mat ref_tmp(ref.rows, ref.cols*2, CV_8UC3);
		Mat tar_tmp(ref.rows, ref.cols*2, CV_8UC3);

		Mat res_cornerMap(ref.rows, ref.cols*2, CV_8UC1);
		Mat ref_cornerMap(ref.rows, ref.cols, CV_8UC1);
		Mat tar_cornerMap(ref.rows, ref.cols, CV_8UC1);

		ref_tmp = Harris_CornerDetect(ref, k, ref_cornerMap, 30000);
		tar_tmp = Harris_CornerDetect(tar, k, tar_cornerMap, 30000);

		for (int i = 0; i < ref.rows; i++) { // appending image
			for (int j = 0; j < ref.cols * 2; j++) {
				if (j < ref.cols) {
					result.at<Vec3b>(i, j)[0] = ref_tmp.at<Vec3b>(i, j)[0];
					result.at<Vec3b>(i, j)[1] = ref_tmp.at<Vec3b>(i, j)[1];
					result.at<Vec3b>(i, j)[2] = ref_tmp.at<Vec3b>(i, j)[2];
				}
				else {
					result.at<Vec3b>(i, j)[0] = tar_tmp.at<Vec3b>(i, j)[0];
					result.at<Vec3b>(i, j)[1] = tar_tmp.at<Vec3b>(i, j)[1];
					result.at<Vec3b>(i, j)[2] = tar_tmp.at<Vec3b>(i, j)[2];
				}
			}
		}

		ref = cvtBGR2Gray(ref);
		tar = cvtBGR2Gray(tar);

		// calculate mag_map, dir_map
		float min = 1000000, max = -1;
		for (y = 0; y < height; y++) { // calculate magnitude and direction
			for (x = 0; x < width; x++) {
				// cur
				conv_x = 0;
				conv_y = 0;

				for (yy = y - 1; yy <= y + 1; yy++) {
					for (xx = x - 1; xx <= x + 1; xx++) { // calc conv_x, conv_y
						if (yy >= 0 && yy < height && xx >= 0 && xx < width) {
							// indexing 에 주의!
							conv_x += ref.at<uchar>(yy, xx) * mask_x[(yy - (y
- 1)) * 3 + (xx - (x - 1))];

							conv_y += ref.at<uchar>(yy, xx) * mask_y[(yy - (y
- 1)) * 3 + (xx - (x - 1))];
						}
					}
				}
				conv_x /= 9.0;
				conv_y /= 9.0; // scaling

				ref_mag[y * width + x] = sqrt(conv_x * conv_x + conv_y * conv_y); // calc
magninute

				dir = atan2(conv_y, conv_x) * 180.0 / PI; // calc direction ( radian to
degree )
				if (dir < 0) dir += 180.0;
				ref_dir_arr[y * width + x] = dir;

				//histogram[(int)(dir / 20)] += mag[y * width + x];
				if (max < ref_mag[y * width + x]) max = ref_mag[y * width + x];
				if (min > ref_mag[y * width + x]) min = ref_mag[y * width + x];
			}
		}
		min = 1000000, max = -1;
```

```
	for (y = 0; y < height; y++) { // calculate magnitude and direction
		for (x = 0; x < width; x++) {
			// cur
			conv_x = 0;
			conv_y = 0;

			for (yy = y - 1; yy <= y + 1; yy++) {
				for (xx = x - 1; xx <= x + 1; xx++) { // calc conv_x, conv_y
					if (yy >= 0 && yy < height && xx >= 0 && xx < width) {
						// indexing 에 주의!
						conv_x += tar.at<uchar>(yy, xx) * mask_x[(yy - (y - 1)) * 3 + (xx - (x - 1))];
						conv_y += tar.at<uchar>(yy, xx) * mask_y[(yy - (y - 1)) * 3 + (xx - (x - 1))];
					}
				}
			}
			conv_x /= 9.0;
			conv_y /= 9.0; // scaling

			tar mag[y * width + x] = sqrt(conv x * conv x + conv y * conv y); // calc magninute

			dir = atan2(conv_y, conv_x) * 180.0 / PI; // calc direction ( radian to degree )
			if (dir < 0) dir += 180.0;
			tar dir arr[y * width + x] = dir;

			if (max < tar_mag[y * width + x]) max = tar_mag[y * width + x];
			if (min > tar_mag[y * width + x]) min = tar_mag[y * width + x];
		}
	}

	float* histo1 = nullptr;
	float* histo2 = nullptr;

	struct sim_x_y {
		int x; // 좌표
		int y;
		float sim; // 유사도
	};

	struct compare { // similiarity 가 작은 애가 더 유사한거임.
		bool operator()(const sim_x_y& m1, const sim_x_y& m2) {
			return m1.sim > m2.sim;
		}
	};

	priority_queue <sim_x_y, vector<sim_x_y>, compare> pq;

	for (int i = 0; i < ref.rows; i++) {
		for (int j = 0; j < ref.cols; j++) {
			if (ref_cornerMap.at<uchar>(i, j) == 0) continue; // reference 에서 코너가 아니면 pass
			histo1 = get_hog_histogram_around_corner(ref, j, i, ref_mag, ref_dir_arr); // 코너이면 histogram 추출

			while(!pq.empty())pq.pop();
			for (int m = 0; m < tar.rows; m++) {
				for (int n = 0; n < tar.cols; n++) {
					if (tar_cornerMap.at<uchar>(m, n) == 0) continue; // target 에서 코너가 아니면 pass

					sim_x_y temp_info = {0,0,0.0}; float sim = 0.0;
					// target 에서 코너 point 이면 histogram 추출
					histo2 = get_hog_histogram_around_corner(tar, n, m, tar_mag, tar_dir_arr);
					sim = get_similarity(histo1, histo2, 9); // 유사도 추출

					temp_info.sim = sim;
					temp_info.x = n;
					temp info.y = m;

					pq.push(temp_info); // min_heap 에 넣기
```

```
                                        }
                                }
                                if (!pq.empty() && pq.top().sim < 0.05)
                                {

                                        cout << "sim : " << pq.top().sim << '\n';
                                        tar_cornerMap.at<uchar>(pq.top().y, pq.top().x) = 0;
                                        ref_cornerMap.at<uchar>(pq.top().y, pq.top().x) = 0;
                                        line(result, Point(j, i), Point(pq.top().x + ref.cols, pq.top().y),
Scalar(255, 0, 0), 1, 8, 0);
                                        imshow("result", result);
                                        imshow("ref_c", ref_cornerMap);
                                        imshow("tar_c", tar_cornerMap);
                                        cv::waitKey(10);

                                        while (!pq.empty()) pq.pop();
                                }
                        }
                }

        return result;
}

int main(int ac, char** av) {

        Mat ref = imread("images/Lecture4/ref.bmp", CV_LOAD_IMAGE_COLOR);
        Mat tar = imread("images/Lecture4/tar.bmp", CV_LOAD_IMAGE_COLOR);

        float k = 0.04f;
        Mat res = HOG_arround_conner(ref, tar, k);
        imshow("res", res);
        imwrite("assignment03_result.bmp", res);
        waitKey(0);

        return 0;
}
```