4nd Week Lab Assignment

12201928 이상혁

Task 1

▪ salt_pepper2.png에 대해서 3x3, 5x5의 Median 필터를 적용해보고 결과를 분석할 것 (잘 나오지 않았다면 그 이유와 함께 결과를 개선해볼 것)

```
Mat medianfilter(Mat Img, int filterSize) {
    Mat srcImg;
    cvtColor(Img, srcImg, COLOR_BGR2GRAY);
    imshow("Input Image", srcImg);

    int width = srcImg.cols;
    int height = srcImg.rows;
    Mat dstImg(srcImg.size(), CV_8UC1);
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;

    int halfSize = filterSize / 2;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
                int k = 0;
                int m_tempImg[25];
                for (int b = -halfSize; b <= halfSize; b++) {
                    for (int a = -halfSize; a <= halfSize; a++) {
                        m_tempImg[k] = srcData[(y + b) * width + (x + a)];
                        k++;
                    }
                }
                k = 0;
                sort(m_tempImg, m_tempImg + filterSize * filterSize);
                dstData[y * width + x] = m_tempImg[filterSize * filterSize / 2];
        }
    }
    return dstImg;
}
```

Figure 1 Median filter CODE

한가지 함수내에서 매개변수를 활용하여 필터의 크기를 조절할 수 있도록 계산하였다.

우선 입력한 이미지를 픽셀값 별로 처리할 수 있도록 grayscale 이미지로 cvtColor를 사용하여 변환하였다.
이미지의 모든 픽셀(height와 width)를 반복하여 진행하도록 반복문을 사용하였다.

Median 필터는 내가 지정한 크기내에 중간값을 계산할 수 있어야한다. 따라서 현재 픽셀을 중심으로 내가 지정한 크기만큼의 값들을 tempimg 배열을 생성하고 저장하였다.

이후 sort를 사용하여서 이 배열을 정렬을 한 이후에 중간값을 dstData에 저장할 수 있도록하였다.

Filtersize를 매개변수로 받고 이를 반으로 나누어준다. 이후 위에 제시한 내용을 반복하는 과정에서 중심픽셀을 기준으로 필터사이즈의 반만큼의 픽셀값을 가지고온다.

또한 filtersize를 사용하여 tempimg배열을 정렬하고 중간값을 찾을 수 있다.

```
void doMedian() {
    cout << "--- doMedianFillter() ---₩n" << endl;
    Mat color_img = imread("salt_pepper2.png", 1);
    if (!color_img.data) printf("No image data ₩n");
    Mat resultImg1 = medianfilter(color_img, 3);
    Mat resultImg2 = medianfilter(color_img,5);
    imshow("Median Test", resultImg1);
    imshow("Median Test", resultImg2);

    waitKey(0);
    destroyAllWindows();
}
```

Figure 2 Median filter Test CODE

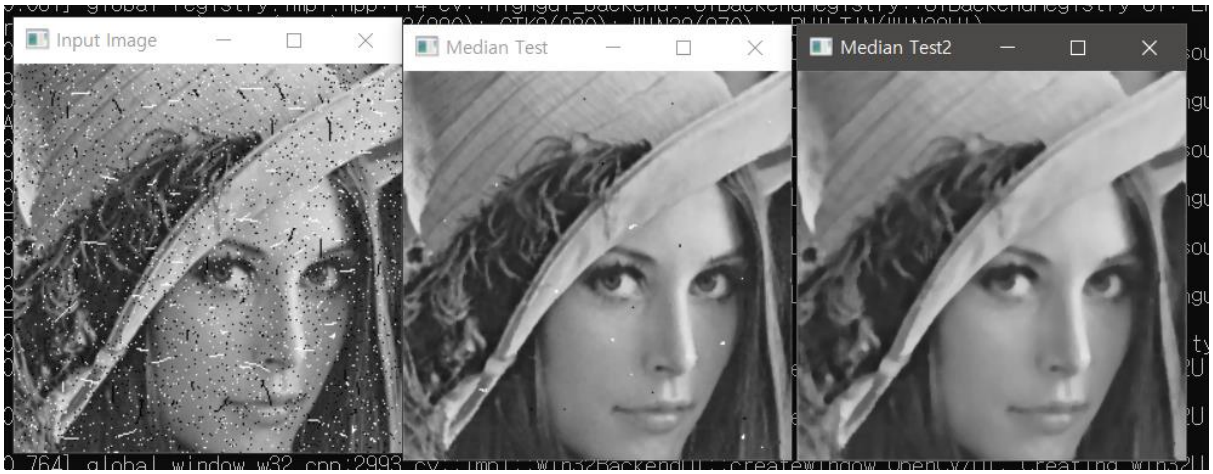

Figure 3 Median filter Result

실행하는 코드는 위의 사진이다.

아래는 이를 실행시킨 결과이다. 왼쪽부터 원본사진, 가운데가 3x3 필터 적용, 마지막이 5x5필터 적용이다. 필터의 크기가 커질수록 salt 효과들이 더 잘 없어지는 것을 확인할 수 있다
그 이유는 주변 픽셀들의 정보를 더 크게 고려하기 때문이다. 따라서 median 필터의 출력이 입력 이미지의 픽셀 값에 대해 더 부드럽게 변화할 수 있게된다. 노이즈가 있는 픽셀의 값이 다른 픽셀에 값에 미치는 영향이 줄어드는 이유도 있다.

Task 2

▪ rock.png에 대해서 Bilateral 필터를 적용해볼 것 (아래 table을 참고하여 기존 gaussian 필터와의 차이를 분석해볼 것)

```
void myBilateral(const Mat& src_img, Mat& dst_img, int diameter, double sig_r, double sig_s) {
    Mat guide_img = Mat::zeros(src_img.size(), CV_64F);
    dst_img = Mat::zeros(src_img.size(), CV_8UC1);
    int wh = src_img.cols; int hg = src_img.rows;
    int radius = diameter / 2;
    for (int c = radius + 1; c < hg - radius; c++) {
        for (int r = radius + 1; r < wh - radius; r++) {
            bilateral(src_img, guide_img, c, r, diameter, sig_r, sig_s);
        }
    }
    guide_img.convertTo(dst_img, CV_8UC1);
}
void bilateral(const Mat& src_img, Mat& dst_img, int c, int r, int diameter, double sig_r, double sig_s) {
    int radius = diameter / 2;
    double gr, gs, wei;
    double tmp = 0;
    double sum = 0;

    for (int kc = -radius; kc <= radius; kc++) {
        for (int kr = -radius; kr <= radius; kr++) {
            gr = gaussian((float)src_img.at<uchar>(c + kc, r + kr)
                - (float)src_img.at<uchar>(c, r), sig_r);
            gs = gaussian(distance(c, r, c + kc, r + kr), sig_s);
            wei = gr * gs;
            tmp += src_img.at<uchar>(c + kc, r + kr) * wei;
            sum += wei;
        }
    }
    dst_img.at<double>(c, r) = tmp / sum;
}

double gaussian(float x, double sigma) {
    return exp(-(pow(x, 2)) / (2 * pow(sigma, 2))) / (2 * CV_PI * pow(sigma, 2));
}
float distance(int x, int y, int i, int j) {
    return float(sqrt(pow(x - i, 2) + pow(y - j, 2)));
}
```

Figure 4 Bilateral 함수 Code

myBilateral 함수

입력 이미지에 양방향 필터를 적용하여 결과 이미지를 생성하는 함수.
먼저 입력 이미지의 크기와 같은 빈 guide_img와 dst_img를 생성하고 반복문을 통해 입력 이미지의 각 픽셀을 순회하며 bilateral 함수를 호출하여 양방향 필터를 적용한다.

bilateral 함수

입력 이미지의 각 픽셀에 대해 양방향 필터를 적용하는 함수 필터링은 입력 이미지의 특정 위치 (c, r) 주변의 픽셀을 대상으로 이루어진다. 반복문을 통해 지정된 지름 내의 픽셀을 순회하며 가중치를 계산하여 필터를 적용한다.

```
void dobilateral() {
    cout << "--- doBilateralEx() ---\n" << endl;
    Mat src_img = imread("rock.png", 0);
    Mat dst_img, dst_img2, dst_img3, dst_img4, dst_img5, dst_img6, dst_img7, dst_img8, dst_img9;
    if (!src_img.data) printf("No image data \n");
    myBilateral(src_img, dst_img, 5, 0.1, 2);
    myBilateral(src_img, dst_img2, 5, 0.1, 6);
    myBilateral(src_img, dst_img3, 5, 0.1, 18);
    myBilateral(src_img, dst_img4, 5, 0.25, 2);
    myBilateral(src_img, dst_img5, 5, 0.25, 6);
    myBilateral(src_img, dst_img6, 5, 0.25, 18);
    myBilateral(src_img, dst_img7, 5, 100, 2);
    myBilateral(src_img, dst_img8, 5, 100, 6);
    myBilateral(src_img, dst_img9, 5, 100, 18);
    imshow("Ex: 1", dst_img);
    imshow("Ex: 2", dst_img2);
    imshow("Ex: 3", dst_img3);
    imshow("Ex: 4", dst_img4);
    imshow("Ex: 5", dst_img5);
    imshow("Ex: 6", dst_img6);
    imshow("Ex: 7", dst_img7);
    imshow("Ex: 8", dst_img8);
    imshow("Ex: 9", dst_img9);
    waitKey(0);
    destroyAllWindows();
}
```

Figure 5 dobilaterl code test1 & test2

Figure 6 Test1에 대한 Result

과제 테이블에 나온 것과 동일한 값으로 진행하였으나 시그마 s를 변형한 값의 차이가 크게 나타나지 않았다. 따라서 값을 조정해보았다.

```
myBilateral(src_img, dst_img, 5, 10, 10);
myBilateral(src_img, dst_img2, 5, 10, 100);
myBilateral(src_img, dst_img3, 5, 10, 10);
myBilateral(src_img, dst_img4, 5, 50, 10);
myBilateral(src_img, dst_img5, 5, 100, 10);
myBilateral(src_img, dst_img6, 5, 10, 100);
```

Figure 7 추가 Test

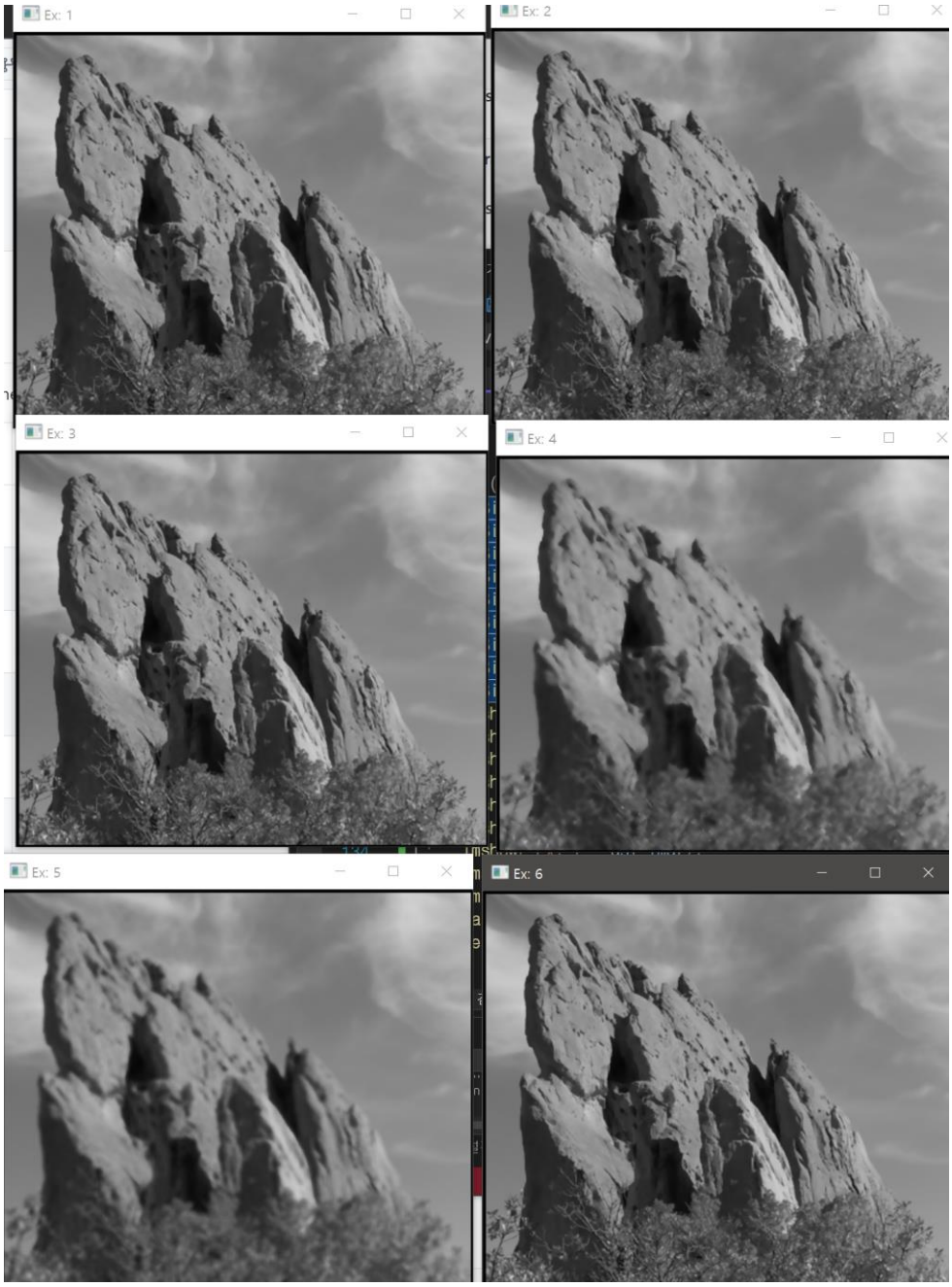

Figure 7 추가 Test Result

Sigma s의 값은 경계가 유지되면서 영역내 필터링을 진행하는 것이고 거리공간 표준편차이다. sigma r의 값은 경계가 보존되지 않은 상태에서 블러링을 진행하며 색공간 표준편차이므로 앞과

같은 결과가 나오게 된다.

강의노트에 나온사진처럼 유사한 결과가 나오게 하기 위해 시그마 파라미터의 값을 조절해보았으나 생각만큼 잘 진행되지 않았다.

sigmaSpace를 엄청 크게 줘도 sigmarange 값이 작다면 큰 변화가 없는 것으로 분석하였다.

따라서 가우시안 필터와의 비교를 명확하게 두줄로 정리해보겠다.

가우시안 필터의 경우 edge 부분까지 블러링을 진행하여 이미지의 형태를 뭉게거나 알아보기 힘든경우도 존재한다.
그러나 양방향 필터의 경우 edge 부분을 제외하고 블러링을 진행하여서 물체의 윤곽 및 경계를 유지한 상태로 블러링을 진행할 수 있다.

Task 3

▪ OpenCV의 Canny edge detection 함수의 파라미터를 조절해 여러 결과를 도출하고 파라미터에 따라서 처리시간이 달라지는 이유를 정확히 서술할 것

Canny 함수를 사용하면 경계선 즉 edge가 나오게 된다.
Canny(src, dst(결과영상),30(낮은경계값),127(높은경계값), 3(sobel 커널크기), false(정교한 작동 여부) 와 같은 파라미터를 받아 설정할 수 있다.

```
void canny() {
    cout << "--- doCanny edge detection() ---₩n" << endl;
    Mat srcimg = imread("gear.jpg", 1);
    if (!srcimg.data) printf("No image data ₩n");
    Mat dst_0, dst_4, dst_1, dst_2, dst_3;
    Canny(srcimg, dst_0, 0, 100, 3, false);
    Canny(srcimg, dst_1, 100, 100, 3, false);
    Canny(srcimg, dst_2, 100, 200, 3, false);
    Canny(srcimg, dst_3, 100, 127, 3, false);
    Canny(srcimg, dst_4, 100, 127, 5, true);
    imshow("Original", srcimg);
    //imshow("Ex: 0", dst_0);
    imshow("Ex: 1", dst_1);
    //imshow("Ex: 2", dst_2);
    //imshow("Ex: 3", dst_3);
    //imshow("Ex: 4", dst_4);
    waitKey(0);
    destroyAllWindows();
}
```

Figure 9 Tesk3에 대한 C ode

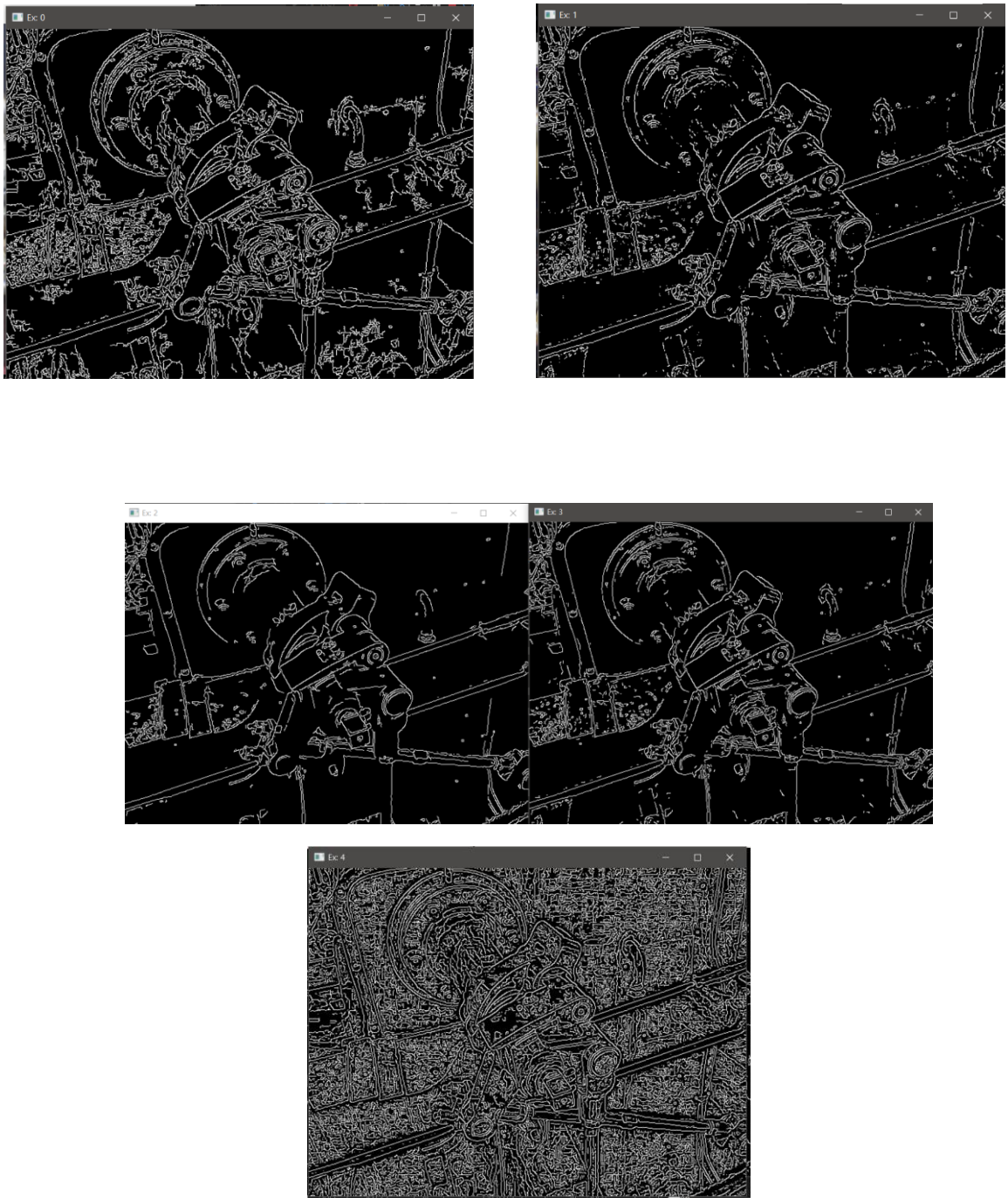

Figure 10 테스트한 사진들

내가 설정한 파라미터는 다음과 같으며 총 9장 정도의 테스트를 진행해보았다.

Ex 0,1을 비교해보면 낮은 경계값이 100정도 차이가 난다. 낮은 경계값이 작은 사진이 조금 더 경계선들이 많이 보이며 커질수록 사라진다. 그러나 중요한 굵은 경계선들은 사라지진 않는다.

Ex 2,3을 비교해보면 높은 경계값이 3이 더 크다. 이때 경계선이 2보다 좀 줄어든 모습을 확인할 수 있다.

Ex4를 보면 sobel 커널 필터수를 3 -> 5로 지정해주었다. 그러니 거의 모든 경계선들이 다 나왔으며 지저분한 모습임을 확인할 수 있다.

True 와 false를 비교하기 위해 한줄의 코드를 더 작성해보았다. 그 결과는 아래와 같다.

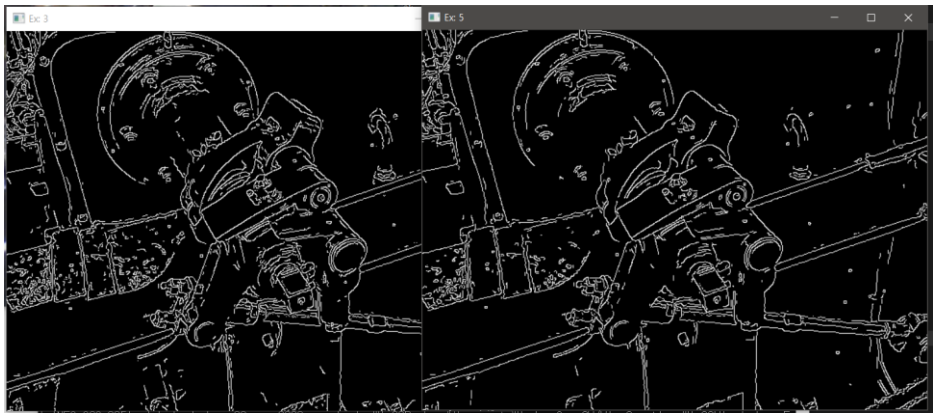

Figure 11 추가 Test 결과

True는 정교하게 작동하는 여부를 설정할 수 있다. 이 결과 왼쪽이 false인데 이때 발견된 경계선의 일부가 사라진 모습을 확인할 수 있었다.

코드를 실행시킨 후 터미널이 작동되고 난 이후에 시간을 측정하였고 값에 따라 처리시간이 어떻게 소요되는지 확인해보고자 하였다.

기본적으로 엣지가 많아지는 경우에 처리시간이 증가하였다. 많은 엣지를 찾아내야 하기 때문이다. 따라서 낮은 경계값과 높은 경계값 사이의 값의 차이가 커질수록 처리시간이 증가함을 확인할 수 있었다. 이는 sobel 커널 크기를 조정하는 부분에서도 적용이 되었다. 3에서 5로 증가 될 때 엣지가 증가하므로 처리시간이 늘었다. 그 이유로는 3에서 5로 더 넓은 영역을 고려해서 계산하여야 하기 때문이다.

마지막으로 true, false는 L2 gradient를 사용여부이다. 이때 사용을 하게 되면 사진에서는 조금더 선명한 엣지를 찾는걸 확인하였다. 이처럼 이러한 연산을 거쳐 불필요한 엣지를 제거하므로 이 연산또한 false 보다 true가 미미하게 조금 더 걸리는 걸로 확인되었다.

이미지가 엄청 복잡하진 않아 대부분 0.~초 사이의 증가가 존재하였다. 이미지의 크기가 커지고 복잡해질수록 연산속도는 더 늘어날 것으로 생각되며 파라미터 조정에 따른 처리속도차이는 존재하였다.

고찰 : openCV 에서 여러가지 필터를 사용해보고 어떠한 역할을 하는지, 어떻게 사용하는지, 직접 구현해보고 원리를 파악 할 수 있었던 기회였다. Task 2 를 진행하는 과정에서 강의노트에 나와있던 모습과 조금 다르게 나와서 당황했지만 자료조사와 여러가지 검색을 통하여 이유를 분석할 수 있었다. 앞으로 이미지를 처리할 때 내가 필요한 이미지, 또한 입력 이미지에 맞추어 적절한 필터를 사용할 수 있을 것 같다.