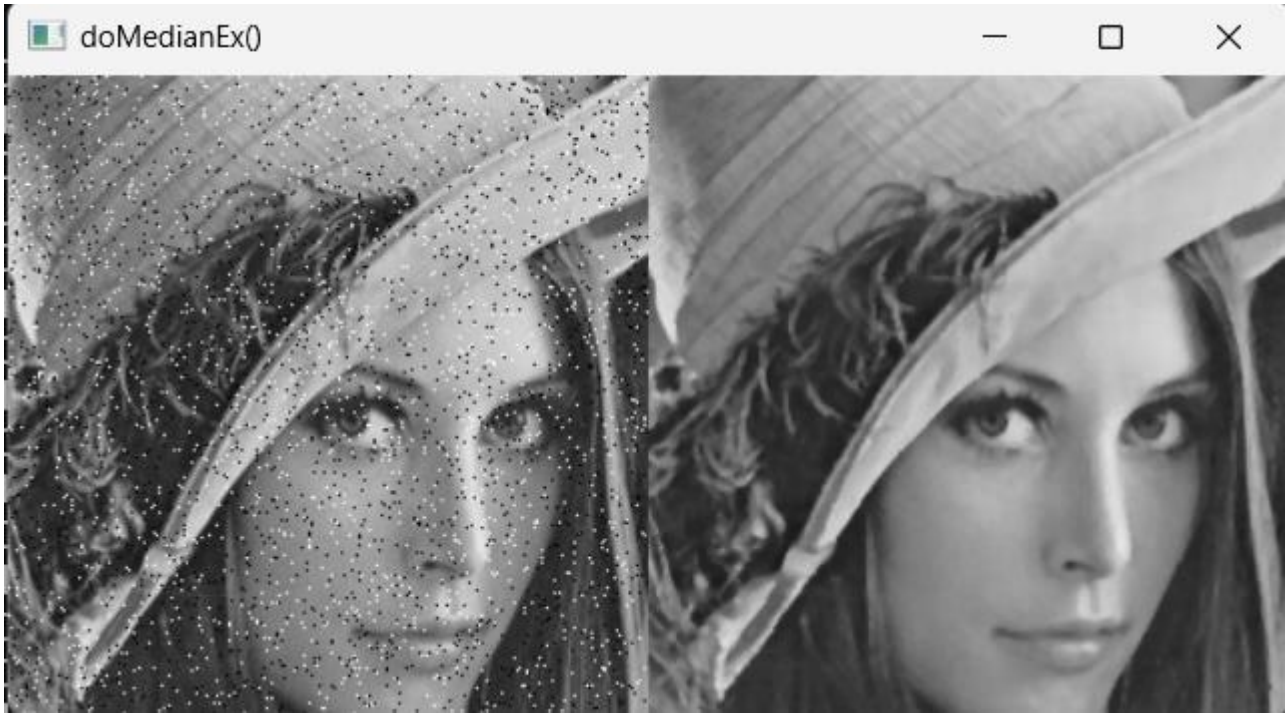
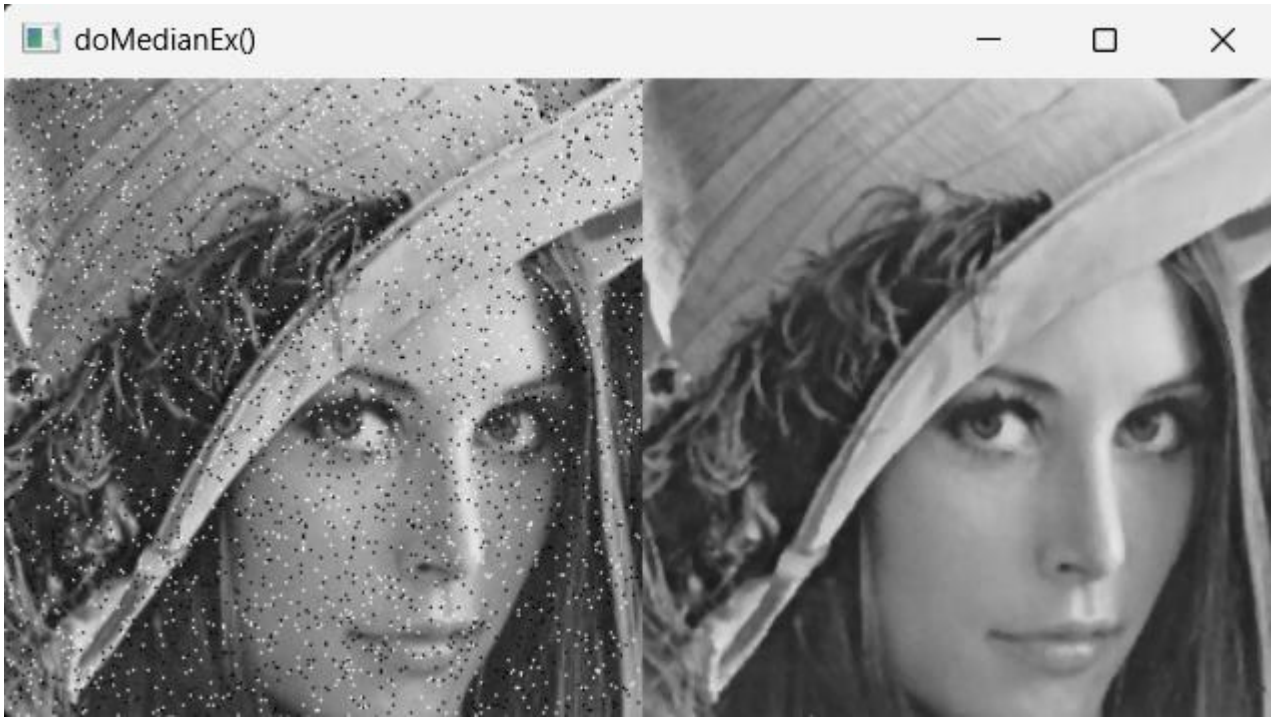
▪ salt_pepper2.png에 대해서 3x3, 5x5의 Median 필터를 적용해보고 결과를 분석할 것 (잘 나오지 않았다면 그 이유와 함께 결과를 개선해볼 것)



3x3



5x5

주어진 salt_pepper2의 사진은 무작위로 분산 되어있는 밝은 점과 어두운 점이 사진에 있다.

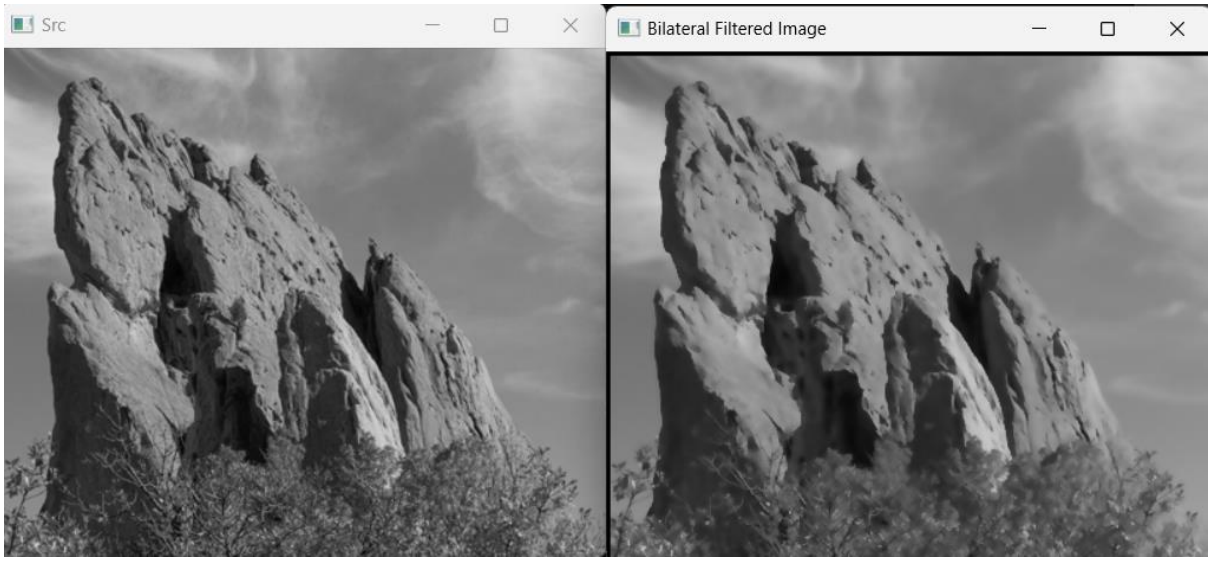
Median 필터를 통해 잡음을 제거하는데 이때 이 필터의 특징은 가장자리와 주요한 이미지의 특징은 보존한다는 것이다. 먼저 3x3 median 필터로 처리한 이미지를 살펴보면 이 필터를 토해 3x3 창 내에서 중간값이 계산되어 각 픽셀의 값이 이웃 픽셀의 중간값으로 대체된 것을 확인할 수 있다. 이 필터는 적용후에보면 그래도 흰색과 검은색 픽셀이 남아있는 것을 확인할 수 있지만 그래도 5x5보다는 최대한 원본이미지의 세부정보를 더 보존한 모습을 확인할 수 있다. 5x5 median 필터는 중간값이 더 큰 창 내에서 계산되어 더 많은 이웃픽셀을 고려한다. 5x5필터 중간값을 적용하면 대부분 모든 흰색과 검은색 픽셀이 사라지고 이미지가 더 깨끗해진 모습을 확인할 수 있다. 이 필터는 더 많은 이웃 픽셀을 고려하기 때문에 따라서 더 부드러운 이미지와 더 적은 잡음이 보이는 것을 확인할 수 있다.

```
void myMedian(const Mat& src_img, Mat& dst_img, const Size& kn_size) {
    dst_img = Mat::zeros(src_img.size(), CV_8UC1);

    int wd = src_img.cols; // 이미지의 너비
    int hg = src_img.rows; // 이미지의 높이
    int kwd = kn_size.width; // 커널 너비
    int khg = kn_size.height; // 커널 높이
    int rad_w = kwd / 2; // 반경 너비
    int rad_h = khg / 2; // 반경 높이

    uchar* src_data = (uchar*)src_img.data;// 입력 이미지 데이터
    uchar* dst_data = (uchar*)dst_img.data;// 출력결과 이미지 데이터

    float* table = new float[kwd * khg]; // 커널 데이터를 동적할당한다
    float tmp;

    // 커널 안에서 각각의 픽셀에 대해서 수행
    for (int c = rad_w; c < wd - rad_w; c++) {
        for (int r = rad_h; r < hg - rad_h; r++) {
            tmp = 0.0f;
            for (int kc = -rad_w; kc <= rad_w; kc++) {
                for (int kr = -rad_h; kr <= rad_h; kr++) {
                    tmp = (float)src_data[(r + kr) * wd + (c + kc)];
                    table[(kr + rad_h) * kwd + (kc + rad_w)] = tmp;
                }
            }
            sort(table, table + kwd * khg); // 커널 데이터를 정렬
            dst_data[r * wd + c] = (uchar)table[(kwd * khg) / 2]; // 중간값을 결과 이미지에 적용
        }
    }

    delete[] table; // 동적 할당된 메모리를 해제한다
}
```

이때 median 함수는 픽셀값을 정렬할 때 중간값을 이미지로 사용하기 위해 커널의 데이터를 정렬한다.

```cpp
//median함수를 만든것을 수행하게 하는 함수
void ex1() {
    cout << "--- doMedianEx() ---\n" << endl;

    // 이미지 불러오기
    Mat src_img = imread("salt_pepper.png", 0);
    if (!src_img.data) {
        printf("No image data \n");
    }

    // Median 필터링함수를 수행 수행
    Mat dst_img;
#ifdef USE_OPENCV
    medianBlur(src_img, dst_img, 5);
#else
    myMedian(src_img, dst_img, Size(5, 5));
#endif

    // 결과 출력
    Mat result_img;
    hconcat(src_img, dst_img, result_img); // 원본과 결과를 가로로 연결한다
    imshow("doMedianEx()", result_img);
    waitKey(0);
}



int main() {

    ex1();
```

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

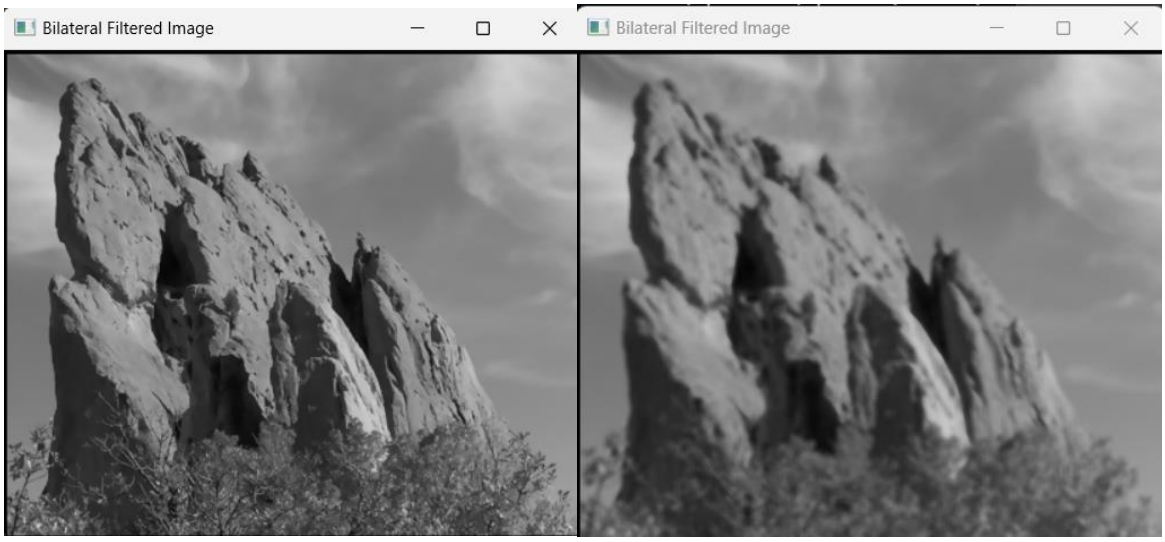


Bilateral 필터는 이미지를 부드럽게 만드는데 사용되는 노이즈를 제거하는 필터인데, Guassian 필터와는 달리 bilateral 필터는 주변 픽셀과의 유사성을 고려하여 필터링을 수행하여 경계를 보존하면서도 노이즈를 줄일 수 있다. 가우시안 필터는 경계를 뭉개면서 노이즈를 제거하고 경계를 부드럽게 만들지만 bilateral 필터는 경계를 보존하면서 노이즈를 제거한다. 이 때문에 계산량이 좀 더 많고 파라미터를 좀더 세밀하게 조정한다.

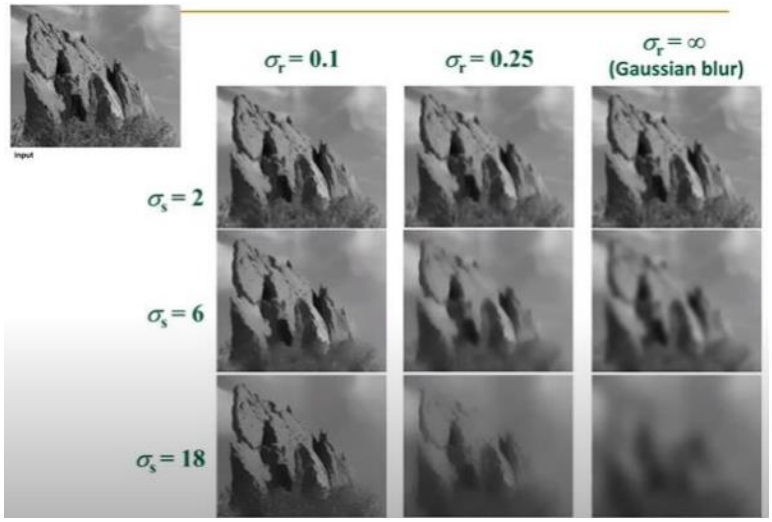
```cpp
// 가우시안 함수
double gaussian(float x, double sigma) {
    return exp(-(pow(x, 2)) / (2 * pow(sigma, 2))) / (2 * CV_PI * pow(sigma, 2));
}

// 두 점 사이의 거리 계산 함수
float distance(int x, int y, int i, int j) {
    return float(sqrt(pow(x - i, 2) + pow(y - j, 2)));
}

//bilateral 필터
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 = 0.0;
    double sum = 0.0;

    //가우시안 연산
    for (int kc = -radius; kc <= radius; kc++) {
        for (int kr = -radius; kr <= radius; kr++) {
            //range 계산
            gr = gaussian((float)src_img.at<uchar>(c + kc, r + kr)
                - (float)src_img.at<uchar>(c, r), sig_r);
            //spatial 계산
            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; //정규화
}
```

```cpp
void myBilateral(const Mat& src_img, Mat& dst_img,
    int diameter, double sig_r, double sig_s) {
    dst_img = Mat::zeros(src_img.size(), CV_8UC1);

    Mat guide_img = Mat::zeros(src_img.size(), CV_64F);
    int wh = src_img.cols; int hg = src_img.rows;
    int radius = diameter / 2;

    // < 필셀 연산시 (가장자리 제외) >
    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_
            // 화소별 Bilateral 계산 수행
        }
    }

    guide_img.convertTo(dst_img, CV_8UC1); // Mat type 변환
}



void ex2() {
    cout << "--- doBilateralEx() ---\n" << endl;

    // 이미지
    Mat src_img = imread("rock.png", 0);
    Mat dst_img;
    if (!src_img.data) printf("No image data \n");

    // < Bilateral 필터링 수행 >
#if USE_OPENCV
    bilateralFilter(src_img, dst_img, 5, 25.0, 50.0);
#else
    int diameter = 5;
    double sig_r = 25.0;
    double sig_s = 50.0;
    myBilateral(src_img, dst_img, diameter, sig_r, sig_s);
#endif
```

Sig_r, sig_s 값 변경



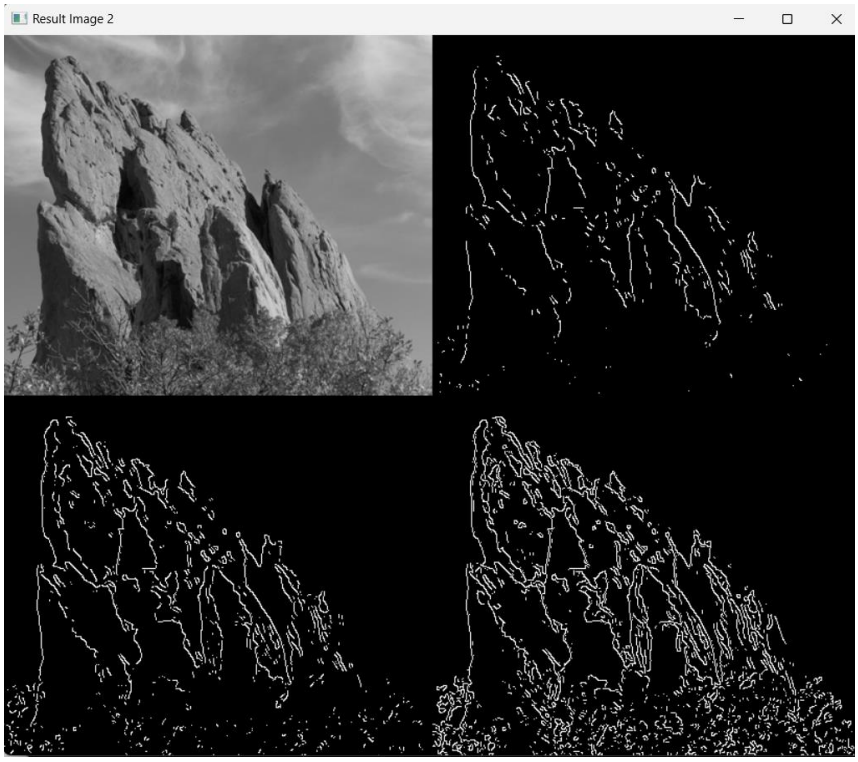
sig_r:10 sig_s:100 sig_r:100 sig_s: 100

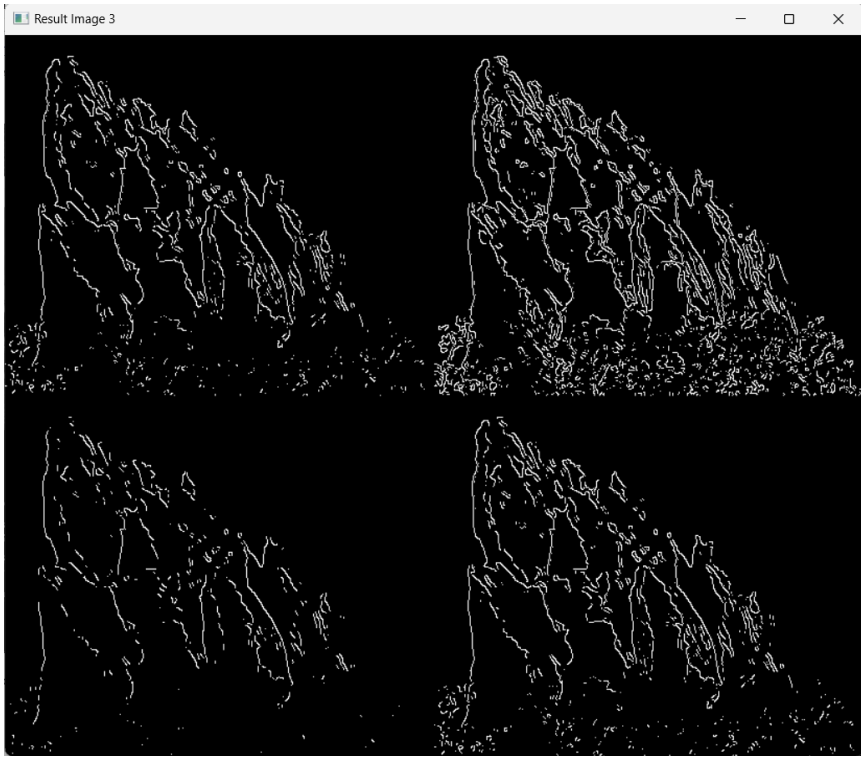


이 표에서 range rate의 시그마인 sig_r은 무한대로 갈수록 가우시안 필터의 역할을 한다. Bilateral 필터를 잘 보여주는 예시는 sig_r =0.25, sig_s =6인 부분이라고 할 수 있다. 경계는 살아있고 비슷한 속성은 유지되는 것이 적절한 예시이다. 주어진 코드에서도 이와 비슷한 값으로 필터를 진행하였고 시그마r의 값이 무한대로 가면서 범위를 넓혀 다 블러처리를 하는 가우시안 필터와의 비교를 위해 sig_r:100 sig_s: 100처럼 점점 시그마의 값을 코드에서 변경했더니 블러의 범위가 넓어지고 edge를 살리지 않는다는 것을 확인할 수 있었다.

▪ OpenCV의 Canny edge detection 함수의 파라미터를 조절해 여러 결과를 도출하고 파라미터에

따라서 처리시간이 달라지는 이유를 정확히 서술할 것






```cpp
void followEdges(int x, int y, Mat& magnitude, int tUpper, int tLower, Mat& edges) {
    edges.at<float>(y, x) = 255;

    // 이웃 픽셀 에지 따는 과정
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
            if ((i == 0) && (j == 0) && (x + i >= 0) && (x + j >= 0) &&
                (x + i < magnitude.cols) && (y + j < magnitude.rows)) {
                if ((magnitude.at<float>(y + j, x + i) > tLower) &&
                    (edges.at<float>(y + j, x + i) != 255)) {
                    followEdges(x + i, y + j, magnitude, tUpper, tLower, edges);
                    // 재귀적 방법으로 이웃 픽셀에서 확실한 edge를 찾아 edge를 구성
                }
            }
        }
    }
}

// 에지 감지 함수
void edgeDetect(Mat& magnitude, int tUpper, int tLower, Mat& edges) {
    int rows = magnitude.rows;
    int cols = magnitude.cols;

    edges = Mat(magnitude.size(), CV_32F, 0.0);

    // 픽셀 에지따기
    for (int x = 0; x < cols; x++) {
        for (int y = 0; y < rows; y++) {
            if (magnitude.at<float>(y, x) > tUpper) {
                followEdges(x, y, magnitude, tUpper, tLower, edges);
                // edge가 확실하면 이웃 픽셀을 따라확실한 edge를 탐색
            }
        }
    }
}
```

```cpp
// 최대가 아닌 부분은 억제하는 함수
// 최대가 아니면 없애서 thinning과정됨
void nonMaximumSuppression(Mat& magnitudeImage, Mat& directionImage) {
    Mat checkImage = Mat(magnitudeImage.rows, magnitudeImage.cols, CV_8U);
    MatIterator_<float >itMag = magnitudeImage.begin<float>();
    MatIterator_<float >itDirection = directionImage.begin<float>();
    MatIterator_<unsigned char>itRet = checkImage.begin<unsigned char >();
    MatIterator_<float > itEnd = magnitudeImage.end<float>();

    for (; itMag != itEnd; ++itDirection, ++itRet, ++itMag) {
        const Point pos = itRet.pos();
        float currentDirection = atan(*itDirection) * (180 / 3.142);
        while (currentDirection < 0) currentDirection += 180;
        *itDirection = currentDirection;
        if (currentDirection > 22.5 && currentDirection <= 67.5) {
            if (pos.y > 0 && pos.x > 0 && *itMag <= magnitudeImage.at<float>(pos.y - 1, pos.x - 1)) {
                magnitudeImage.at<float >(pos.y, pos.x) = 0;
            }
            if (pos.y < magnitudeImage.rows - 1 && pos.x < magnitudeImage.cols - 1 && *itMag <= magnitudeImage.at<float >(pos.y + 1, pos.x + 1)) {
                magnitudeImage.at<float>(pos.y, pos.x) = 0;
            }
        }
        else if (currentDirection > 67.5 && currentDirection <= 112.5) {
            if (pos.y > 0 && *itMag <= magnitudeImage.at<float >(pos.y - 1, pos.x)) {
                magnitudeImage.at<float>(pos.y, pos.x) = 0;
            }
            if (pos.y < magnitudeImage.rows - 1 && *itMag <= magnitudeImage.at<float >(pos.y + 1, pos.x)) {
                magnitudeImage.at<float>(pos.y, pos.x) = 0;
            }
        }
        else if (currentDirection > 112.5 && currentDirection <= 157.5) {
            if (pos.y > 0 && pos.x < magnitudeImage.cols - 1 && *itMag <= magnitudeImage.at<float >(pos.y - 1, pos.x + 1)) {
                magnitudeImage.at<float >(pos.y, pos.x) = 0;
            }
            if (pos.y < magnitudeImage.rows - 1 && pos.x>0 && *itMag <= magnitudeImage.at<float >(pos.y + 1, pos.x - 1)) {
                magnitudeImage.at<float >(pos.y, pos.x) = 0;
            }
        }
        else {
            if (pos.x > 0 && *itMag <= magnitudeImage.at<float >(pos.y, pos.x - 1)) {
                magnitudeImage.at<float >(pos.y, pos.x) = 0;
            }
            if (pos.x < magnitudeImage.cols - 1 && *itMag <= magnitudeImage.at<float >(pos.y, pos.x + 1)) {
                magnitudeImage.at<float >(pos.y, pos.x) = 0;
            }
        }
    }

}
```

```cpp
void ex3() {
    cout << "--- doCannyEx() ---\n" << endl;

    // 이미지 로드
    Mat src_img = imread("rock.png", 0);
    if (!src_img.data) printf("No image data \n");

    //Mat dst_img;
    Mat dst_img1, dst_img2, dst_img3, dst_img4, dst_img5;
#if USE_OPENCV
    // < Canny 에지 탐색 수행 >
    Canny(src_img, dst_img, 180, 240);
#else
    // Gaussian 필터 기반 노이즈 제거
    Mat blur_img;
    GaussianBlur(src_img, blur_img, Size(3, 3), 1.5);

    // Sobel필터를 통한 에지 검출
    Mat magX = Mat(src_img.rows, src_img.cols, CV_32F);
    Mat magY = Mat(src_img.rows, src_img.cols, CV_32F);
    Sobel(blur_img, magX, CV_32F, 1, 0, 3);
    Sobel(blur_img, magY, CV_32F, 0, 1, 3);

    // sobel 결과로 에지 방향 계산
    Mat sum = Mat(src_img.rows, src_img.cols, CV_64F);
    Mat prodX = Mat(src_img.rows, src_img.cols, CV_64F);
    Mat prodY = Mat(src_img.rows, src_img.cols, CV_64F);
    multiply(magX, magX, prodX);
    multiply(magY, magY, prodY);
    sum = prodX + prodY;
    sqrt(sum, sum);

    // < 에지 크기 계산 >
    Mat magnitude = sum.clone();
```

```
    //magnitude.convertTo(magnitude, CV_32F);  // 데이터 타입 변환

    // Non-maximum suppression을 적용하여 엣지 변환
    Mat slopes = Mat(src_img.rows, src_img.cols, CV_32F);
    divide(magY, magX, slopes);
    // gradient의 방향 계산
    nonMaximumSuppression(magnitude, slopes);

    // < Edge tracking by hysteresis >
    /*edgeDetect(magnitude, 200, 100, dst_img);*/
    /*dst_img.convertTo(dst_img, CV_8UC1);*/
#endif

    //edge tracking by hysteresis
    start = clock();
    edgeDetect(magnitude, 200, 100, dst_img1);
    finish = clock();
    cout << (float)(finish - start) << "ms" << endl;
    start = clock();
    edgeDetect(magnitude, 150, 100, dst_img2);
    finish = clock();
    cout << (float)(finish - start) << "ms" << endl;
    start = clock();
    edgeDetect(magnitude, 100, 50, dst_img3);
    finish = clock();
    cout << (float)(finish - start) << "ms" << endl;
    start = clock();
    edgeDetect(magnitude, 200, 30, dst_img4);
    finish = clock();
    cout << (float)(finish - start) << "ms" << endl;
    start = clock();
    edgeDetect(magnitude, 150, 50, dst_img5);
    finish = clock();
    cout << (float)(finish - start) << "ms" << endl;

    dst_img1.convertTo(dst_img1, CV_8UC1);
    dst_img2.convertTo(dst_img2, CV_8UC1);
    dst_img3.convertTo(dst_img3, CV_8UC1);
```

```
    dst_img1.convertTo(dst_img1, CV_8UC1);
    dst_img2.convertTo(dst_img2, CV_8UC1);
    dst_img3.convertTo(dst_img3, CV_8UC1);
    dst_img4.convertTo(dst_img4, CV_8UC1);
    dst_img5.convertTo(dst_img5, CV_8UC1);



    // 결과 이미지 생성 세로로 합침
    Mat result_img, result_img2, result_img2_2, result_img3;
    hconcat(src_img, dst_img1, result_img);
    hconcat(dst_img2, dst_img3, result_img2);
    hconcat(dst_img4, dst_img5, result_img3);
    // 가로로 합침
    vconcat(result_img, result_img2, result_img2_2);
    vconcat(result_img2, result_img3, result_img3);


    // 각 결과 이미지를 따로 표시
    imshow("Result Image 1", result_img);
    waitKey(0); // 사용자가 키를 누를 때까지 기다림

    imshow("Result Image 2", result_img2_2);
    waitKey(0); // 사용자가 키를 누를 때까지 기다림

    imshow("Result Image 3", result_img3);
    waitKey(0); // 사용자가 키를 누를 때까지 기다림

}


int main() {



    ex3();
```

```
//edge tracking by hysteresis
        start = clock();
        edgeDetect(magnitude, 200, 100, dst_img1);
        finish = clock();
        cout << (float)(finish - start) << "ms" << endl;
        start = clock();
        edgeDetect(magnitude, 150, 100, dst_img2);
        finish = clock();
        cout << (float)(finish - start) << "ms" << endl;
        start = clock();
        edgeDetect(magnitude, 100, 50, dst_img3);
        finish = clock();
        cout << (float)(finish - start) << "ms" << endl;
        start = clock();
        edgeDetect(magnitude, 200, 30, dst_img4);
        finish = clock();
        cout << (float)(finish - start) << "ms" << endl;
        start = clock();
        edgeDetect(magnitude, 150, 50, dst_img5);
```

```
finish = clock();
cout << (float)(finish - start) << "ms" << endl;
```

이 부분을 보면 각각 그림에 따라 임계값을 다르게 설정하여 각각의 canny edge detection결과가 다르게 나타남을 알 수 있다. 처음에는 high threshold를 200 low threshold를 100으로 설정한다. 이는 강력한 200을 넘는 에지들을 먼저 탐지한 수 100을 넘는 자잘자잘한 에지로 이를 이어주면서 canny edge detection을 완료하겠다는 것이다. 높은 임계값의 설정으로 그림에서 모서리 부분이 매우 있을것만 있고 자잘한게 없는 것을 알 수 있다. 그 다음 그림은 high threshold만 150으로 조절해서 가한 에지를 탐색할때의 기준치를 좀 낮게 잡아 이전 그림보다는 좀더 자잘하다. 아예 100, 50으로 잡은건 높은 임계치와 낮은 임계치 모두 낮기에 굵고 확실한 에지보다는 자잘한게 더 많이 잡히고 200 30으로하면 큰 에지가 잡히긴하지만 많이 끊겨있어 30이상의 임계치로 이를 이어주면 자잘한 부분이 나중에 많이 잡힌다. 이런식으로 high threshold 와 low threshold의 값이 출력되는 이미지의 에지를 달라지게한다. Non-maximum supression은 굵은 에지가 검출될 때 bilinear interpolation을 통해 나온 값중 최대가 아닌 것은 다 없애고 최대인 값만 에지로 남겨서 thinning 과정을 일으킨다.

Canny edge detection에서 파라미터를 조절하면 처리시간이 달라지는 이유는 파라미터에 따라 edge가 달라지게 되므로 이때 픽셀수에 따른 연산량이 달라져서 처리 시간에 차이가 나타난다. 예를 들어 높은 임계값을 사용하면 감지되는 가장자리의 양이 줄어들어 연산량이 감소한다. 큰 에지를 쉽게 탐지하고 빈 부분으로 low threshold를 통해 이으므로 일단 높은 임계치로 에지를 탐지하는 것이 연산 시간을 줄인다.