1. LoG

코드 목적:

Laplacian of Gaussain 을 적용하여 이미지의 Edge 를 찾아낸다

코드 흐름:

1) lena.jpg를 불러와 흑백 이미지로 만든다.
2) Gaussianfilter()함수를 호출하여 가우시안 필터링을 한다.
3) Laplacianfilter()함수를 호출해 (2)의 이미지에 Laplacian filtering을 한다.
4) (1), (2), (3)의 각 결과 이미지를 새로운 창에 출력한다.

함수 설명: Gaussianfilter(const Mat input, int n, double sigma_t, double sigma_s);

매개변수:

input: input 이미지 행렬
n: kernel 의 크기 (2n+1)x(2n+1)
sigma_t: x 축에 대한 표준편차
sigma_s: y 축에 대한 표준편차

함수 목적: Gaussian filter를 이용해 Gaussian noise를 제거한 이미지를 반환

Laplacianfilter(const Mat input);

매개변수:

input: input 이미지 행렬

함수 목적:

input 이미지를 laplacianfiltering 하여 결과 이미지을 리턴한다.

get_Gaussian_Kernel(int n, double sigma_t, double sigma_s, bool normalize);

매개변수:

input: input 이미지 행렬
n: kernel 의 크기 (2n+1)x(2n+1)
sigma_t: x 축에 대한 표준편차
sigma_s: y 축에 대한 표준편차
normalize: normalize 여부

함수 목적: Gaussian Filter 에 사용되는 Kernel 을 계산하여 반환한다

get_Laplacian_Kernel (int n, double sigma_t, double sigma_s, bool normalize);

매개변수:

input: input 이미지 행렬
n: kernel 의 크기 (2n+1)x(2n+1)
sigma_t: x 축에 대한 표준편차
sigma_s: y 축에 대한 표준편차
normalize: normalize 여부

0	1	0
1	-4	1
0	1	0

함수 목적: Laplacian Filter 에 사용되는 Kernel 을 계산하여 반환한다

Mirroring(const Mat input, int n);

매개변수:

input: input 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

함수 목적: input 이미지에 (2n+1)x(2n+1) kernel 을 적용하기 위해 상하좌우로 2n 만큼의 pixel 에 mirroring 방식으로 값을 채워준다.

설명:

이미지에서, 미분을 이용해 Intensity 의 변화량을 알아내 Edge 를 찾아낼 수 있다. 하지만 이미지에 Noise 가 심할 경우, Noise 와 Edge 를 구분하기 어려울 수 있다는 문제점이 존재한다. 이를 해결하기 위해, Gaussian Filter 를 이용해 Noise 를 제거한 뒤, Laplacian Filter 를 적용해 Edge 를 찾아낸다.

1) LoG.cpp

코드 설명:

Gaussianfilter()

```
for (int i = n; i < row+n; i++) {
        for (int j = n; j < col+n; j++) {
            double sum=0.0;
            for (int a = -n; a <= n; a++) {
                for (int b = -n; b <= n; b++) {
                    sum += kernel.at<double>(a+n, b+n)*(float)(input_mirror.at<double>(i+a, j+b));
                }
            }
            output.at<double>(i-n, j-n) = (double)sum;
        }
    }
```

for문을 통해 (i,j) 픽셀의 filtering을 위한 kernel의 (a+n,b+n)픽셀에 접근한다. input_mirror에 접근해 가져온 데이터를 해당하는 kernel의 가중치에 곱해 sum에 더해준다. kernel의 모든 픽셀에 대한 가중치*input데이터를 더해준 값을 output 이미지의 해당 비트에 넣어준다.

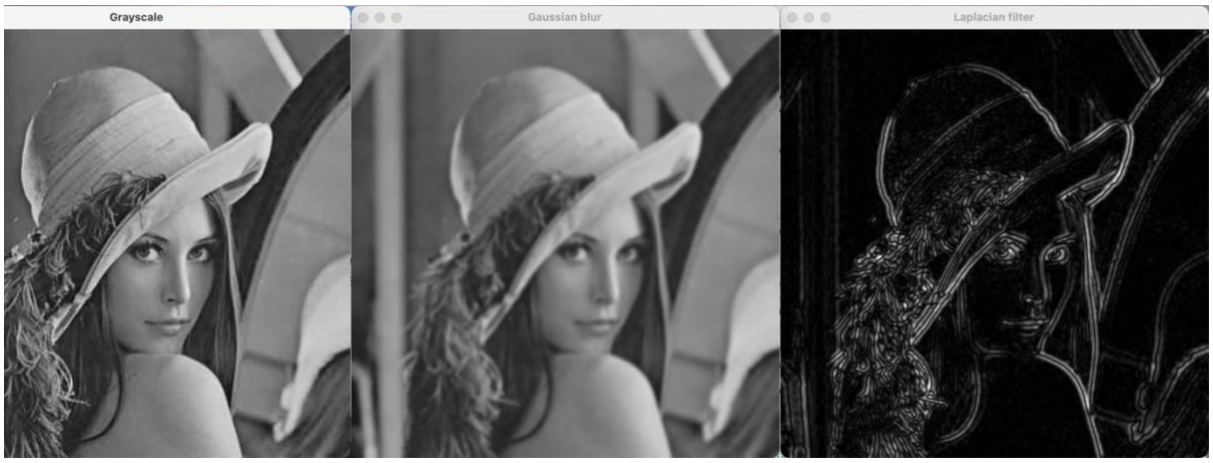
Laplacianfilter()

```
Mat kernel = get_Laplacian_Kernel();
for (int i = n; i < row + n; i++) {
        for (int j = n; j < col + n; j++) {
            double sum=0.0;
            for (int a = -n; a <= n; a++) {
                for (int b = -n; b <= n; b++) {
                    sum += kernel.at<double>(a+n, b+n)*(float)(input_mirror.at<double>(i+a, j+b));
                }
            }
            if(sum<0)sum*=-1;
            output.at<double>(i-n, j-n) = (double)sum;
        }
    }
```

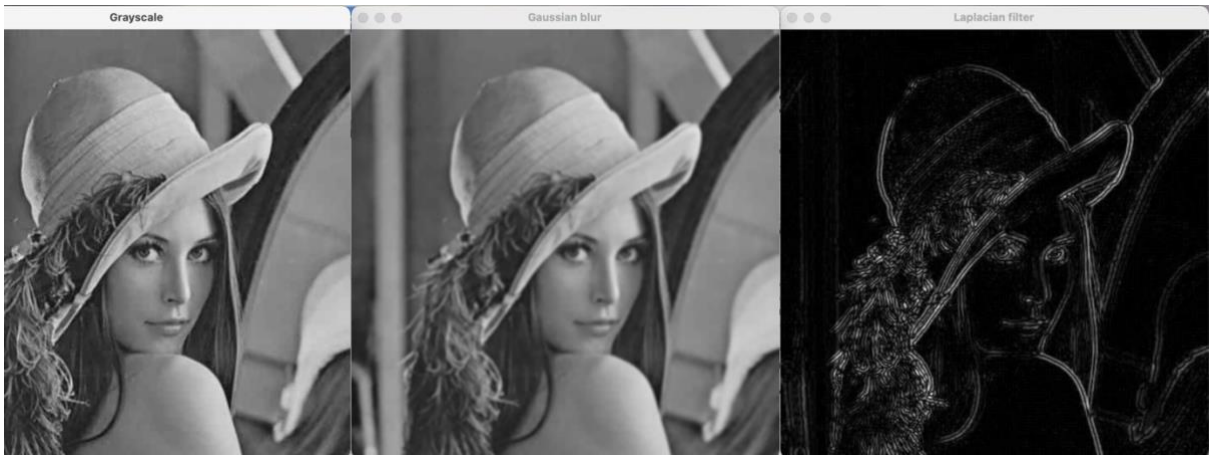
2 차 미분을 근사화 한 filter 를 get_Laplacian_Kernel()함수를 이용해 얻어낸다. (i,j)픽셀에 접근해 for 문으로 kernel 의 모든 pixel 에 접근한다. input_mirror 에 접근해 가져온 데이터를 해당하는 kernel 의 가중치에 곱해 sum 에 더해준다. Laplacian Filter 의 결과 이미지 O 는 $O = |L * I|$이므로, sum 이 음수인 경우 -1 을 곱해 양수로 만들어준다. 계산된 값을 output 이미지의 해당 픽셀에 넣어준다

실행 결과:

```
int window_radius = 20;
```



int window_radius = 2;



실행 결과, Gaussian Filtering 의 kernel 을 크게 하니 Noise 는 잘 제거되었지만, Edge 가 두껍게 검출됨을 알 수 있다.

2) LoG_RGB.cpp

(1)의 코드와 수행하는 일은 같지만, 컬러 이미지에 대해 LoG 를 적용시키는 코드이다.
모든 연산을 RGB 세 channel 에 대해 따로따로 계산해준다.

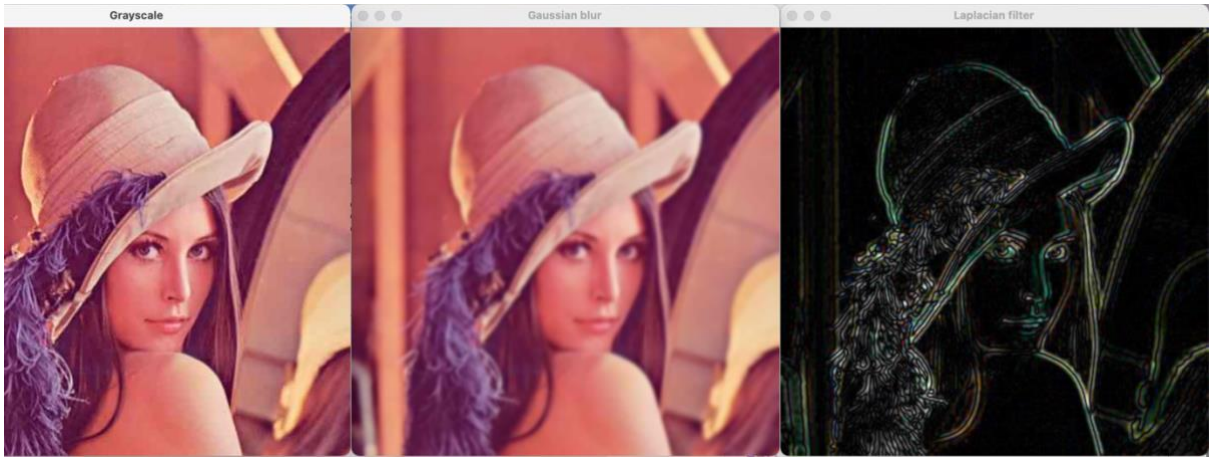
```
//Gaussianfilter
for (int i = n; i < row+n; i++) {
                for (int j = n; j < col+n; j++) {
                double sum1=0.0;
                double sum2=0.0;
                double sum3=0.0;
                for (int a = -n; a <= n; a++) {
                     for (int b = -n; b <= n; b++) {
                          sum1 += (double)kernel.at<double>(a + n, b +
n)*(double)(input_mirror.at<Vec3d>(i+a, j+b)[0]);
                          sum2 += (double)kernel.at<double>(a + n, b +
n)*(double)(input_mirror.at<Vec3d>(i+a, j+b)[1]);
                          sum3 += (double)kernel.at<double>(a + n, b +
n)*(double)(input_mirror.at<Vec3d>(i+a, j+b)[2]);


                     }
                }
                 output.at<Vec3d>(i-n, j-n)[0] = (double)sum1;
                output.at<Vec3d>(i-n, j-n)[1] = (double)sum2;
                output.at<Vec3d>(i-n, j-n)[2] = (double)sum3;
                }
          }
```

```
//Laplacianfilter
for (int i = n; i < row + n; i++) {
                for (int j = n; j < col + n; j++) {


                              //Fill the code
                double sum1=0.0;
                double sum2=0.0;
                double sum3=0.0;
                for (int a = -n; a <= n; a++) {
                     for (int b = -n; b <= n; b++) {


                          sum1 += kernel.at<double>(a+n,
b+n)*(double)(input_mirror.at<Vec3d>(i+a, j+b)[0]);
                          sum2 += kernel.at<double>(a+n,
b+n)*(double)(input_mirror.at<Vec3d>(i+a, j+b)[1]);
                          sum3 += kernel.at<double>(a+n,
b+n)*(double)(input_mirror.at<Vec3d>(i+a, j+b)[2]);
                     }
                }
                if (sum1 < 0) sum1 *= -1;
                if (sum2 < 0) sum2 *= -1;
                if (sum3 < 0) sum3 *= -1;

                output.at<Vec3d>(i-n, j-n)[0] = abs(sum1);
                output.at<Vec3d>(i-n, j-n)[1] = abs(sum2);
                output.at<Vec3d>(i-n, j-n)[2] = abs(sum3);
                }
          }
```

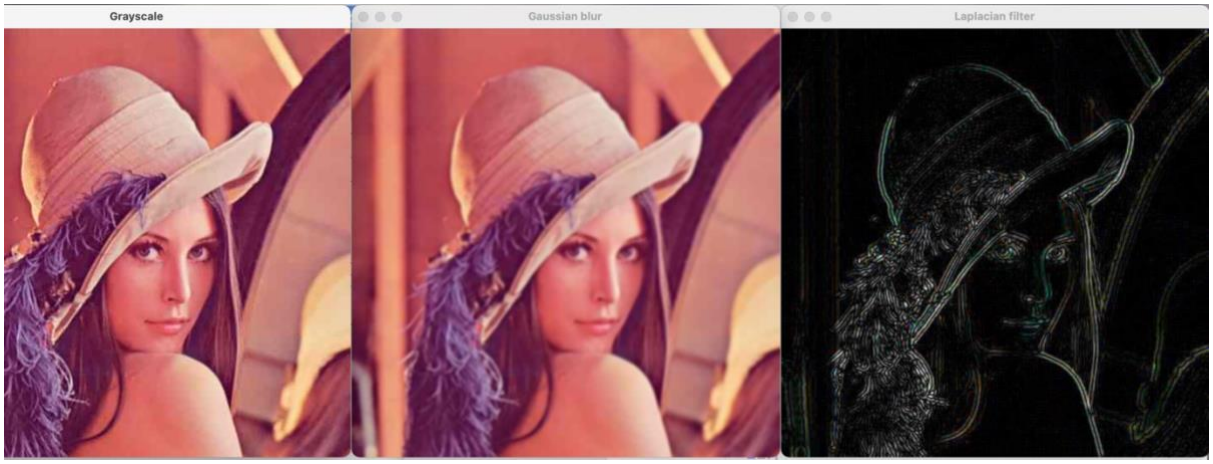
mirroring 함수 또한 각 채널에 대해 따로 계산해준다

실행결과:

int window_radius = 20;



int window_radius = 2;



2. Canny.cpp

코드 목적:

Opencv 가 제공하는 함수를 이용해 Canny Edge Detector 를 구현해본다.

코드 흐름:

1) lena.jpg를 불러와 흑백 이미지를 만든다.
2) Opencv가 제공하는 Canny()함수를 이용해 edge를 추출한다.
3) 결과 이미지를 새로운 창으로 출력

함수 설명: 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

설명:

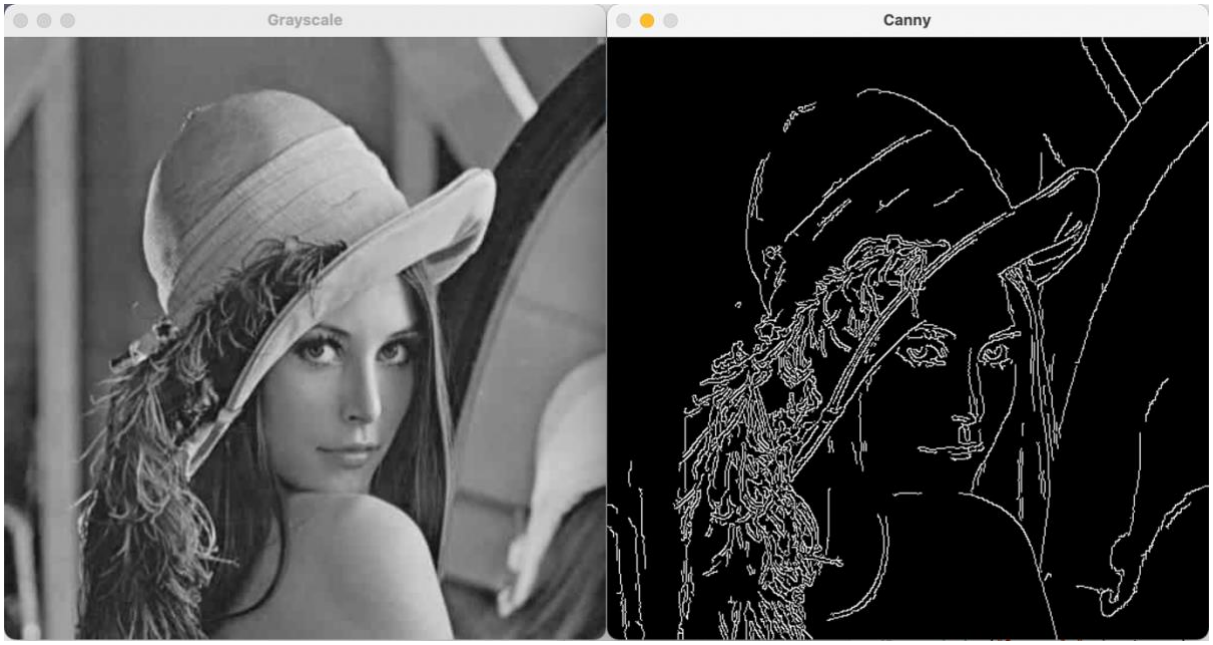
Canny Edge Detecter 는 Low-pass & High-pass filters 를 이용해 Edge 를 찾아낸 뒤, Edge 의 방향 $A(x,y) = tan^{-1}\left(\frac{G_x}{G_y}\right)$을 이용해, Non-maximum suppression 을 한다. Edge 방향에 해당하는 neighbor pixel 과 비교해, 자신이 더 클 경우만 Edge 로 판단한다. 또한 threshold1 보다 작으면 Edge 가 아니고, threshold2 보다 크면 Edge 가 맞으며 그 사이일때는 주변 pixel 들의 edge 여부를 따라가는 Doouble thresholding 를 사용해 값을 보정한다.

코드:

```
Canny(input_gray, output, 80, 170);
```

실행결과:

Canny(input_gray, output, 80, 170);



intensity 만 고려한 이미지와 다르게, position+intensity clustering 은 가까이 있는 비슷한 색들이 같은 그룹에 속함을 확인 가능하다

3. Harris_corner.cpp

코드 목적: 입력 이미지에 대해 Harris Corner Detector 를 이용해 corner 을 찾아낸다

코드 흐름:

1) 이미지를 불러와 흑백이미지를 만든다.
2) cornerHarris함수에 흑백 이미지를 넣어 코너를 찾은 output이미지를 반환받고, 출력
3) output이미지의 최댓값*0.01을 threshold로 하여 corner에 빨간 동그라미 그려 출력
4) NonMaxSupp이 true라면 NonMaximum_Suppression()함수를 이용해 Corner들 중 주변값보다 가장 큰 값을 가지는 pixel만 Corner로 설정
5) Subpixel이 true라면 openCV가 제공하는 cornerSubPix을 이용해 정확한 Corner의 위치 추출

함수 설명:

NonMaximum_Suppression(const Mat input, Mat corner_mat, int radius)

input: 입력 이미지 행렬

corner_mat: corner이면 1, 아니면 0이 저장된 행렬

radius: Non-maximum suppression에서 고려할 이웃들의 범위 반지름

함수 목적: corner pixel들에 접근해 주변 pixel 값 중 자신이 가장 클 때만 corner로 판단

Mirroring(const Mat input, int n);

input: input 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

함수 목적: input 이미지에 (2n+1)x(2n+1) kernel 을 적용하기 위해 상하좌우로 2n 만큼의 pixel 에 mirroring 방식으로 값을 채워준다.

void type2str(int type)

함수 목적: Matrix의 type를 모를 때 사용

vector<Point2f> MatToVec(const Mat input)

input: 입력 행렬

함수 목적: 행렬을 벡터로 전환해 반환

opencv 제공 함수:

cornerHarris(img, output, blockSize, ksize, k, borderType)

img: 입력 이미지

output: 반환할 결과 이미지

blockSize: corner detection에서 고려할 이웃 픽셀 크기

ksize: (미분을 위한) 소벨 연산자를 위한 커널 크기
k: 해리스 코너 검출 상수
borderType: 가장자리 픽셀 확장 방식
함수목적: output에 해리스 코너 값 반환. 흑백이미지에서, 하얀 점이 코너이다.

minMaxLoc(Img, &minVal, &maxVal, &minLoc, &maxLoc);
Img: 입력 이미지
&minVal: 최솟값
&maxVal: 최댓값
&minLoc: 최솟값의 좌표
&maxLoc: 최댓값의 좌표
함수목적: Img 행렬의 모든 pixel중 최댓값, 최솟값, 그 좌표들을 찾는다

cornerSubPix(image, corners, winSize, zeroZone, criteria)
image: input 이미지
corners: 코너점
iwnSize: Search Window 의 절반 사이즈
zeroZone: Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done
criteria: 종료 시점
함수목적: image 의 컴출된 코너점의 조금 더 정확한 위치를 추출

circle(img, Point(x, y), radius, Scalar(b,g,r), thickness, lineType, shift)
img: 이미지 파일
Point(x, y): 원의 중심 좌표
radius: 원의 반지름
Scalar(b,g,r): 색상 0~255
thickness: 선 두께
lineType: 선 종류
shift: fractional bit (default 0)
함수목적: img 의 Point 에 매개변수의 옵션을 가지는 원을 그려 반환한다.

코드 설명:
① 코너검출

```
cornerHarris(input_gray, output, 2, 3, 0.04, BORDER_DEFAULT);
```

코너 검출하여 output 에 반환

output 행렬의 최대, 최소값 추출

```
double minVal, maxVal;          Point minLoc, maxLoc;
minMaxLoc(output, &minVal, &maxVal, &minLoc, &maxLoc);
```

output이미지의 최댓값*0.01을 threshold로 하여 이보다 크면 corner로 판단하여, 원본 이미지에 circle()함수로 빨간 원을 그려주고, corner_mat의 값을 1로하고, corner의 개수를 corner_num변수를 이용해 세준다. 결과적으로 input_visual 이미지의 corner에 빨간 원이 그려지고, corner들에만 1이 저장된 corner_mat이 완성된다.

```
int corner_num = 0;
for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (output.at<float>(i, j) > 0.01*maxVal) {
                circle(input_visual, Point(j, i), 2, Scalar(0, 0, 255), 1, 8, 0);
                corner_mat.at<uchar>(i, j) = 1;
                corner_num++;
            }
            else
                output.at<float>(i, j) = 0.0;
        }
}
```

②Non-maximum suppression

위의 NonMaxSupp 변수가 true 일때만 수행한다.

NonMaximum_Suppression 을 호출한다.

```
if (NonMaxSupp) {
        NonMaximum_Suppression(output, corner_mat, 2);

        corner_num = 0;
        input_visual = input.clone();
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (corner_mat.at<uchar>(i, j) == 1) {
                    circle(input_visual, Point(j, i), 2, Scalar(0, 0, 255), 1, 8, 0);
                    corner_num++;
                }
            }
        }
```

NonMaximum_Suppression():
각 pixel 에 접근한다. 해당 인덱스의 corner_matrix 의 값이 1 이라면 corner 이므로, Non-maximum suppression 를 수행한다. 주변 이웃한 pixel 들 중 자신이 가장 큰 값일 때만 corner 로 판단하는 것이다. 즉, corner 이 두껍게, 정확하지 않게 구해졌으므로 가장 뚜렷한 corner 만을 구하는 것이다.
매개변수로 받은 radius 에 대해, 2*radius+1 크기의 kernel 을 사용, 상하좌우 radius 만큼 떨어져 있는 pixel 들을 neighbor 로 여긴다. 이웃한 픽셀의 값이 center 보다 클 경우 해당 인덱스의 corner_matrix 를 0 으로 바꿔주고, 더이상 연산이 필요없으므로 break 한다.

```
for (int i = radius; i < row+radius; i++) {
        for (int j = radius; j < col+radius; j++) {
            Mat kerner= Mat::zeros(row, col, CV_8U);

            if(corner_mat.at<uchar>(i-radius,j-radius)==1){
                    float center=input_mirror.at<float>(i-radius,j-radius);
                    for(int a=-radius;a<radius;a++){
                     for(int b=-radius;b<radius;b++){
                         if(input_mirror.at<float>(i+a,j+b)>center){
                             corner_mat.at<uchar>(i-radius,j-radius)=0;
                             break;
                         }
                     }
                }
         }
        }
}
```

NonMaximum_Suppression()함수 연산을 통해 main 함수 안의 corner_matrix 값이 업데이트 되었다. 다시 main 함수로 돌아와 수행한다. 바뀐 corner_matrix 에 따라, corner 에 빨간 원을 다시 그려준다.

```
if (NonMaxSupp) {
        NonMaximum_Suppression(output, corner_mat, 2);

        corner_num = 0;
        input_visual = input.clone();
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (corner_mat.at<uchar>(i, j) == 1) {
                    circle(input_visual, Point(j, i), 2, Scalar(0, 0, 255), 1, 8, 0);
                    corner_num++;
                }
            }
        }
```

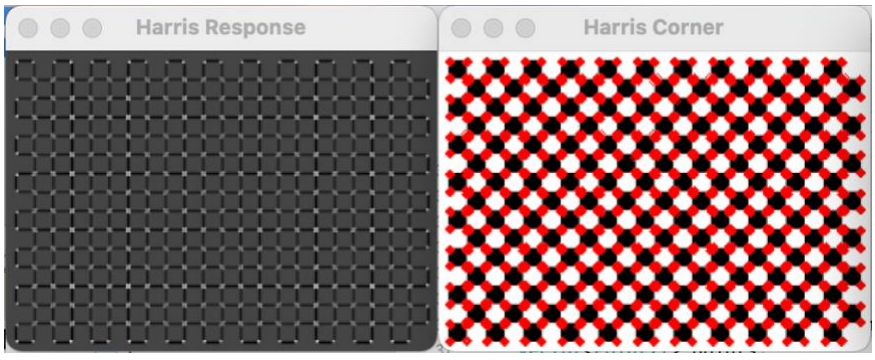
③ Subpixel

Subpixel 변수가 true 일때만 수행한다. OpenCV 에서 지원해주는 cornerSubPix 함수를 이용하면 조금 더 정확한 Corner 의 위치를 추출할 수 있다. 결과값을 받은 points 에 대해 다시 원을 그려준다. 행렬이 아닌 벡터를 사용하므로 값의 처리가 필요하다.

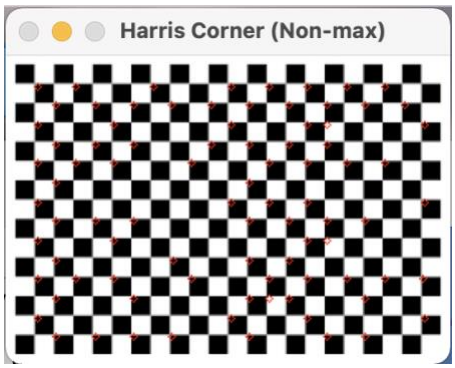
```
if (Subpixel) {
        Size subPixWinSize(3, 3);
        TermCriteria termcrit(TermCriteria::COUNT | TermCriteria::EPS, 20, 0.03);

        points = MatToVec(corner_mat);

        cornerSubPix(input_gray, points, subPixWinSize, Size(-1, -1), termcrit);

        input_visual = input.clone();
        for (int k = 0; k < points.size(); k++) {
            int x = points[k].x;
            int y = points[k].y;
            if (x<0 || x>col - 1 || y<0 || y>row - 1) {
                        points.pop_back();
                        continue;
            }
            circle(input_visual, Point(x, y), 2, Scalar(0, 0, 255), 1, 8, 0);
        `
```

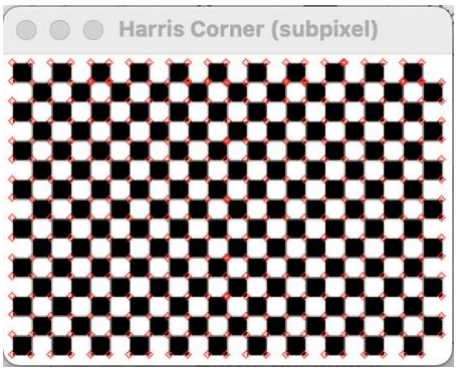
실행 결과:

Mat input = imread("checkerboard.png", CV_LOAD_IMAGE_COLOR);

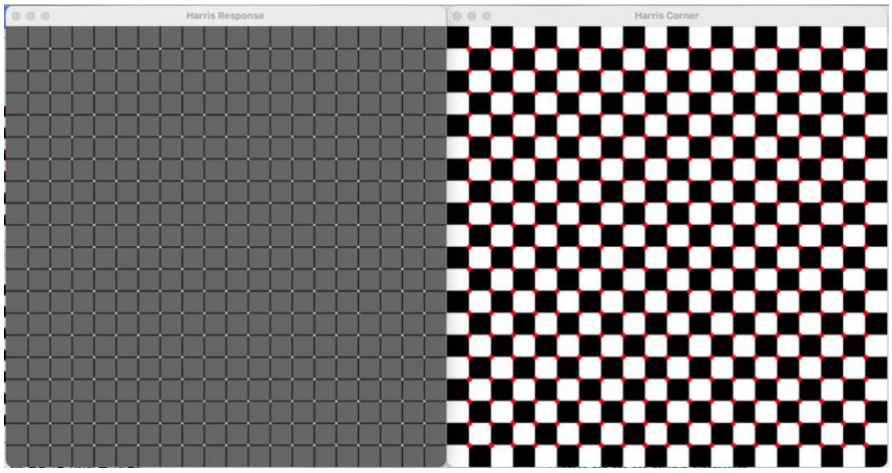


i)bool NonMaxSupp = true; bool Subpixel = false; ii)bool NonMaxSupp = false; bool Subpixel = true;

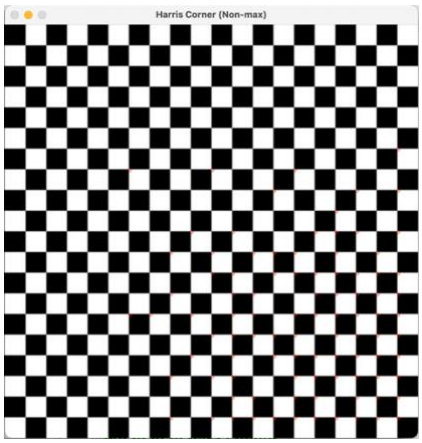




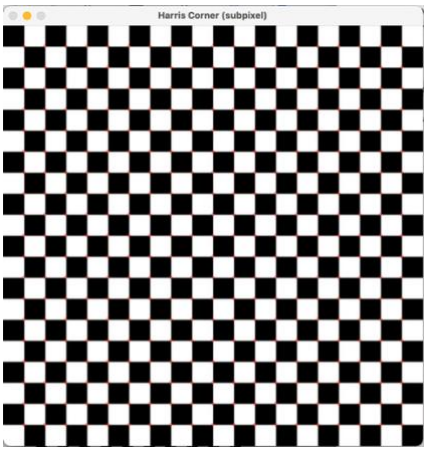
```
Mat input = imread("checkerboard2.jpg", CV_LOAD_IMAGE_COLOR);
```



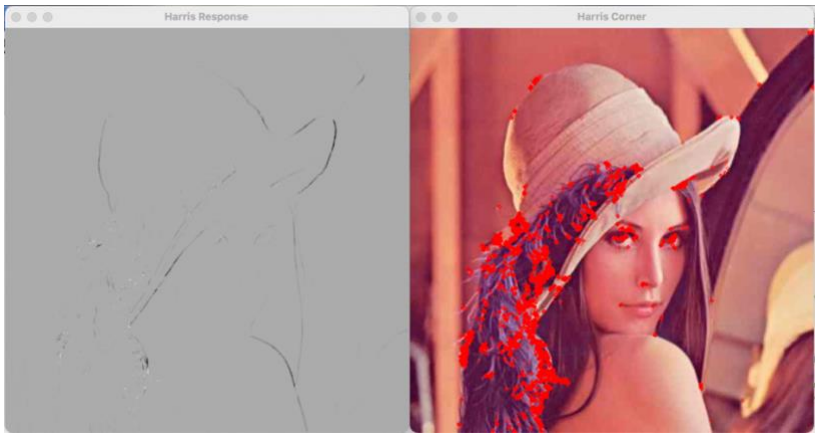
i)`bool NonMaxSupp = true; bool Subpixel = false;`



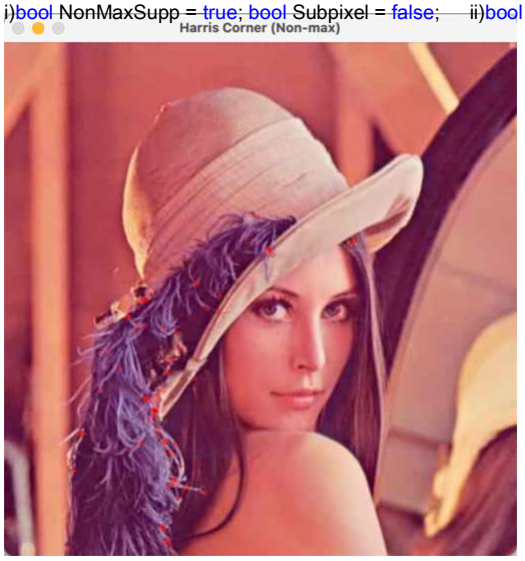
ii)`bool NonMaxSupp = false; bool Subpixel = true;`

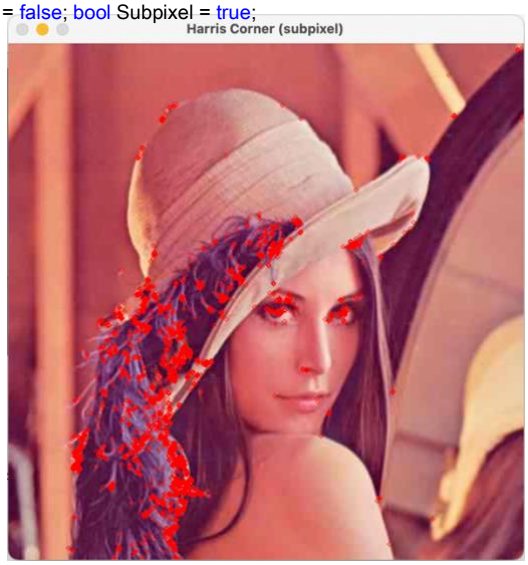



```
Mat input = imread("lena.jpg", CV_LOAD_IMAGE_COLOR);
```



i)bool NonMaxSupp = true; bool Subpixel = false; ii)bool NonMaxSupp = false; bool Subpixel = true;





참고자료:

오픈 SW 프로젝트 Lec07 수업자료