Laplacian of Gaussian

a) The purpose of the code

"LoG-skeleton.cpp"과 "LoG_RGB.cpp" 이 두개의 코드는 Laplacian of Gaussian을 구현한 코드이다. Grayscale 입력 이미지에 대해서는 "LoG-skeleton.cpp"을 사용하고, color 입력 이미지에 대해서는 "LoG_RGB.cpp"을 사용할 수 있다. Laplacian of Gaussian은 edge detection을 위한 알고리즘이다. Laplacian of Gaussian filtering은 gaussian filtering을 한 후에 laplacian filtering을 해준다.

b) The explanation of the code

Laplacain of Gaussian filtering을 하기 위해서는 첫번째로 gaussian filtering을 해주어서 불필요한 noise를 제거한다. Edge detection에서 noise가 있는 경우 edge가 어디에 있는지 파악하기 힘들다는 문제가 있다. 따라서 noise를 제거한 후에 laplacian filtering을 한다.

[LoG-skeleton.cpp]

이 코드는 grayscale 입력 이미지를 처리하기 위한 코드이다.

```
Mat Gaussianfilter(const Mat input, int n, double sigma_t, double sigma_s) {

    int row = input.rows;
    int col = input.cols;

    // generate gaussian kernel
    Mat kernel = get_Gaussian_Kernel(n, sigma_t, sigma_s, true);
    Mat output = Mat::zeros(row, col, input.type());

    //Intermediate data generation for mirroring
    Mat input_mirror = Mirroring(input, n);
    double kernelvalue;
    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++) {
                    kernelvalue = (double)kernel.at<double>(a + n, b + n);
                    sum += kernelvalue * (double)(input_mirror.at<double>(i + a, j + b));
                }
            }
            output.at<double>(i - n, j - n) = (double)sum;
        }
    }
    return output;
}
```

이 코드는 gaussian filtering을 해주기 위한 Gaussianfilter() 함수이다. Get_Gaussian_Kernel() 함수를 이용하여, 만들어진 Gaussian kernel 값을 가져올 수 있다.
입력 이미지의 모든 픽셀에 대하여, 한 픽셀을 중심으로 그 주변 픽셀과 커널 값을 각각 곱하여 더하는 것이 이미지 필터링을 하는 방법이다. 따라서 Kernelvalue 변수에 gaussian kernel 값을 넣어주고, boundary processing으로 mirroring을 한 이미지의 각 픽셀과 곱하면 gaussian filtering된 이미지를 얻을 수 있다.

```
Mat Laplacianfilter(const Mat input) {

    int row = input.rows;
    int col = input.cols;

    Mat kernel = get_Laplacian_Kernel();
    Mat output = Mat::zeros(row, col, input.type());

    int n = 1;
    Mat input_mirror = Mirroring(input, n);
    double kernelvalue;

    for (int i = n; i < row + n; i++) {
        for (int j = n; j < col + n; j++) {
            double sum = 0.0;
            double val = 0.0;
            for (int a = -n; a <= n; a++) {
                for (int b = -n; b <= n; b++) {
                    kernelvalue = (double)kernel.at<double>(a + n, b + n);
                    sum += kernelvalue * input_mirror.at<double>(i + a, j + b);
                }
            }
            val = abs(sum);
            if (val < 0)
                val = 0;
            else if (val > 255)
                val = 255;
            output.at<double>(i - n, j - n) = (double)(val * 10);
        }
    }

    return output;
```

이 코드는 Laplacian filtering을 위한 Laplacianfilter() 함수이다. get_Laplacain_Kernel() 함수를 이용하여서 laplacian kernel을 생성하고 kernel 값을 kernelvalue 변수에 저장해준다. Laplacian 필터를 입력 이미지와 필터링한 output은 | L * I |을 계산하여 얻을 수 있다. 따라서 Laplacian kernel 값과 boundary processing으로 mirroring한 input_mirror pixel 곱을 더한 값에 abs() 함수를 이용하여 절댓값을 얻었다. val 변수에 그 값을 넣어줌으로써 laplacian filtering한 결과 값을 얻을 수 있도록 하였다.

[LoG_RGB.cpp]

```
Mat Mirroring(const Mat input, int n)
{
    int row = input.rows;
    int col = input.cols;

    Mat input2 = Mat::zeros(row + 2 * n, col + 2 * n, input.type());
    int row2 = input2.rows;
    int col2 = input2.cols;

    for (int i = n; i < row + n; i++) {
        for (int j = n; j < col + n; j++) {
            for (int z = 0; z < 3; z++) {
                input2.at<Vec3d>(i, j)[0] = input.at<Vec3d>(i - n, j - n)[0];
                input2.at<Vec3d>(i, j)[1] = input.at<Vec3d>(i - n, j - n)[1];
                input2.at<Vec3d>(i, j)[2] = input.at<Vec3d>(i - n, j - n)[2];
            }
        }
    }
```

```
for (int i = n; i < row + n; i++) {
    for (int j = 0; j < n; j++) {
        for (int z = 0; z < 3; z++) {
            input2.at<Vec3d>(i, j)[0] = input2.at<Vec3d>(i, 2 * n - j)[0];
            input2.at<Vec3d>(i, j)[1] = input2.at<Vec3d>(i, 2 * n - j)[1];
            input2.at<Vec3d>(i, j)[2] = input2.at<Vec3d>(i, 2 * n - j)[2];
        }
    }
    for (int j = col + n; j < col2; j++) {
        for (int z = 0; z < 3; z++) {
            input2.at<Vec3d>(i, j)[0] = input2.at<Vec3d>(i, 2 * col - 2 + 2 * n - j)[0];
            input2.at<Vec3d>(i, j)[1] = input2.at<Vec3d>(i, 2 * col - 2 + 2 * n - j)[1];
            input2.at<Vec3d>(i, j)[2] = input2.at<Vec3d>(i, 2 * col - 2 + 2 * n - j)[2];
        }
    }
}
```

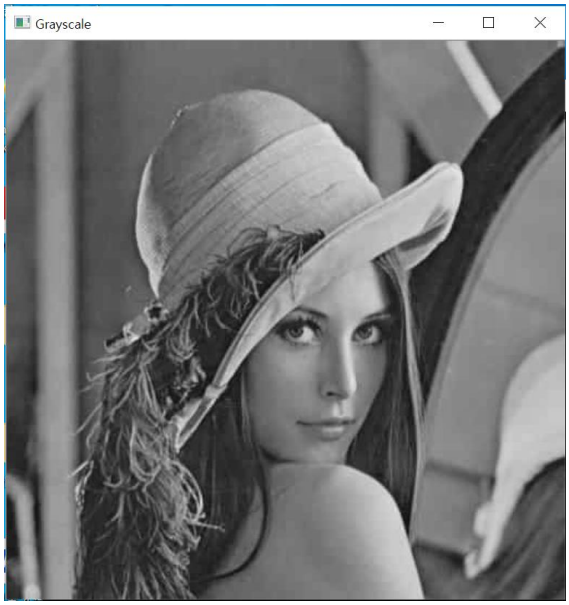
```
for (int j = 0; j < col2; j++) {
    for (int i = 0; i < n; i++) {
        for (int z = 0; z < 3; z++) {
            input2.at<Vec3d>(i, j)[0] = input2.at<Vec3d>(2 * n - i, j)[0];
            input2.at<Vec3d>(i, j)[1] = input2.at<Vec3d>(2 * n - i, j)[1];
            input2.at<Vec3d>(i, j)[2] = input2.at<Vec3d>(2 * n - i, j)[2];
        }
    }
    for (int i = row + n; i < row2; i++) {
        for (int z = 0; z < 3; z++) {
            input2.at<Vec3d>(i, j)[0] = input2.at<Vec3d>(2 * row - 2 + 2 * n - i, j)[0];
            input2.at<Vec3d>(i, j)[1] = input2.at<Vec3d>(2 * row - 2 + 2 * n - i, j)[1];
            input2.at<Vec3d>(i, j)[2] = input2.at<Vec3d>(2 * row - 2 + 2 * n - i, j)[2];
        }
    }
}

return input2;
```

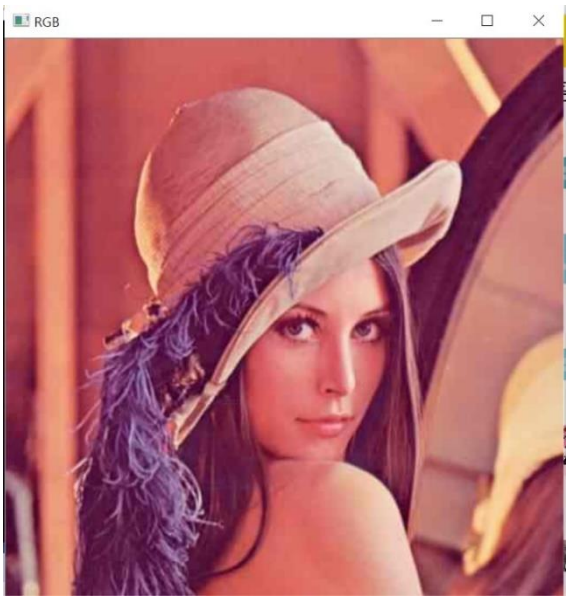
이 코드는 color 입력 이미지에 대하여 image boundary processing으로 mirroring을 하는 코드이다. Color 입력 이미지에 대해 LoG를 할 경우, color 이미지를 처리할 때의 특성을 고려하여 RGB 채널을 나눈 후에 각각 채널에 대하여 mirroring을 해주어야 한다.

Gaussian filter() 함수와 laplacian filter() 함수는 각 channel에 대하여 filtering을 해준다는 점만다를 뿐 나머지는 grayscale에 대한 코드와 동일하다.

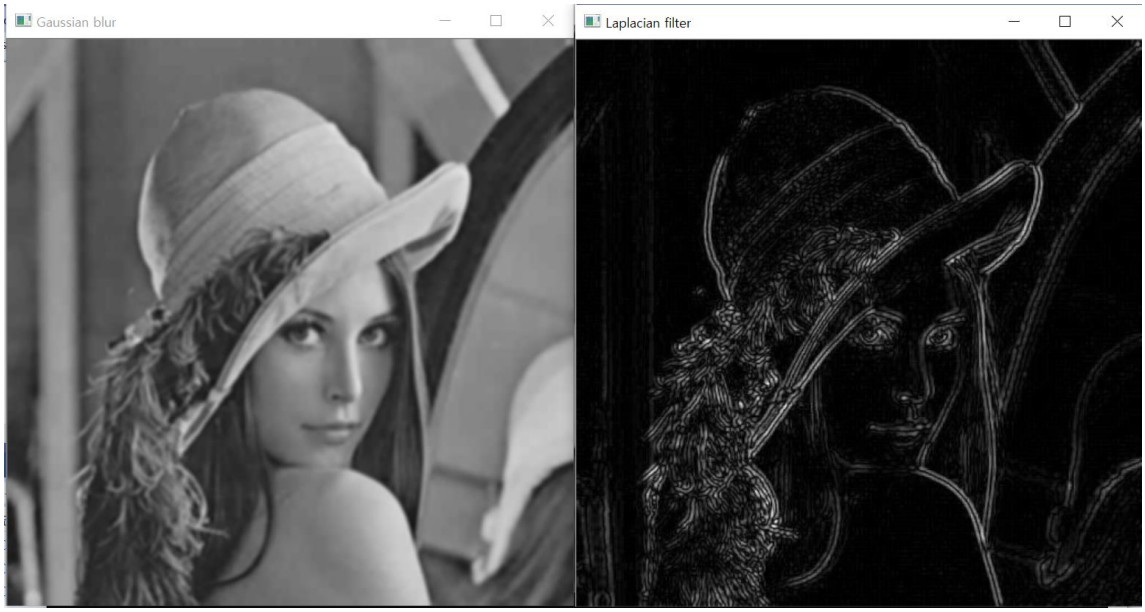
c) The output of the code

(1) Input (grayscale / RGB)

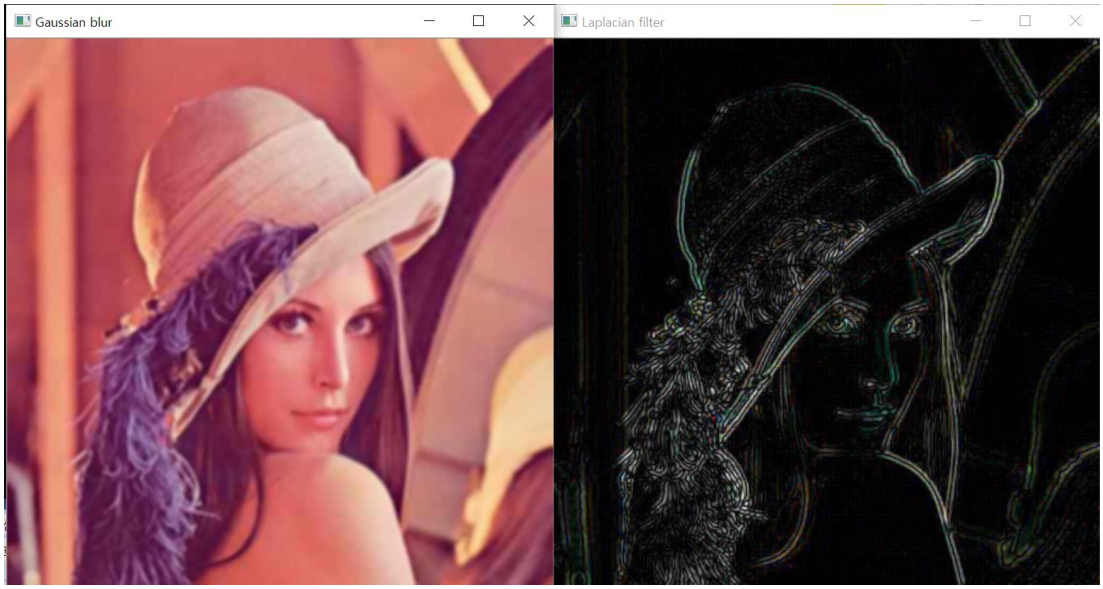




(2) Grayscale output(gaussian filtering / gaussian + Laplacian filtering)



(3) RGB output(gaussian filtering / gaussian + Laplacian filtering)



d) Analysis

Gaussian filtering 특성상 low pass filter이기 때문에 noise는 제거되지만 blur해진다. 그것을 gaussian blur output 이미지를 통해 확인할 수 있다. 하지만 noise를 제거한 후에 Laplacian filtering을 하여 edge detection을 잘 되었음을 Laplacian filter 이미지를 통해 확인할 수 있다.

Canny Edge Detector

a) The purpose of the code

Canny-skeleton.cpp 이 코드는 opencv에서 제공하는 함수인 Canny()를 이용하여서 canny edge detector를 구현한다. Canny edge detection은 가장 많이 사용하는 edge detection 방법이다. 첫번째로 gaussian filter와 soble filter로 filtering을 한다. opencv에서 제공하는 함수 Canny()도 이와 같은 알고리즘을 사용한다.

b) The explanation of the code

```
//Fill the code using 'Canny' in OpenCV.
Canny(input_gray, output, 30, 125, 3, false);
```

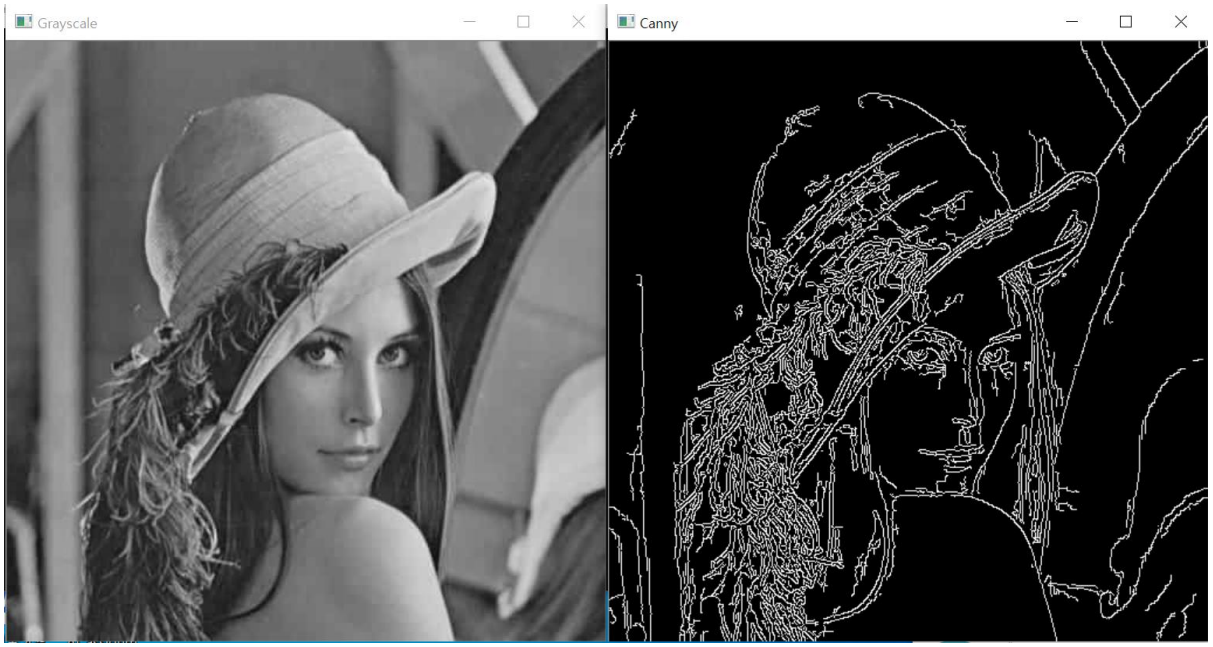
Opencv의 Canny() 함수를 이용한 코드이다. Canny() 함수의 경우, 다음과 같이 인자를 넣어주도록 되어있다.

```
void cv::Canny          (InputArray          image,
                         OutputArray         edges,
                         double              threshold1,
                         double              threshold2,
                         int                 apertureSize = 3,
                         bool                L2gradient = false
                        )
```

이를 따라서 input image로 grayscale로 변환한 input_gray를 넣어주었고, output을 받아올 array로 output 변수를 넣어주었으며, threshold로 임의의 적당한 수를 넣어주었다.

c) The output of the code

(1) Input / output



d) Analysis

Canny 함수를 사용하여 edge detection이 된 것을 확인할 수 있다.

Harris corner detector

a) The purpose of the code

"Harris_corner-skeleton.cpp"은 opencv에서 제공하는 cornerHarris() 함수를 이용하여 harris corner detection을 구현한 코드이다. 또한 cornerSubPix() 함수를 이용하여 보다 정확한 corner detection이 되는지 확인해보고, non-maximum suppression 알고리즘을 이용하여 corner가 중복되어 잡히는 문제를 해결해 보았다.

b) The explanation of the code

```
//Fill the code////////////////////////////////////
cornerHarris(input_gray, output, 2, 3, 0.04, BORDER_DEFAULT);
///////////////////////////////////////////////////
```

이 코드는 opencv에서 제공하는 함수인 cornerHarris() 함수로 harris corner detection을 구현할 수 있다. cornerHarris() 함수는 다음과 같이 인자를 넣어주어야 한다.

```
void cv::cornerHarris (InputArray           src,
                       OutputArray          dst,
                       int       blockSize,
                       int       ksize,
                       double    k,
                       int       borderType = BORDER_DEFAULT
                      )
```

입력 이미지로 grayscale로 변환한 input_gray를 넣어주고 output을 받아올 output 변수를 넣어주었으며 나머지 인자들은 opencv harris detector tutorial을 참고하여 적당한 수로 넣어주어서 Harris corner detection한 결과를 얻을 수 있도록 하였다.

```
//Fill the code////////////////////////////////////////////////////
output.convertTo(output, CV_8UC1, 1.0 / 255);
cornerSubPix(output, points, subPixWinSize, Size(-1, -1), termcrit);
///////////////////////////////////////////////////////////////////
```

이 코드는 opencv의 cornerSubPix()를 사용한 코드이다. cornerSubPix() 함수는 다음과 같이 인자를 넣어줄 수 있다. 인자를 넣어줄 때 주의할 점은 input으로 들어가는 array가 8bit여야 한다는 점이다. 따라서 convertTo() 함수를 이용하여 output 변수를 8bit로 변환시켰다.

```
void cv::cornerSubPix(cv::InputArray image, cv::InputOutputArray corners, cv::Size winSize, cv::Size zeroZone, cv::
    TermCriteria criteria)
```

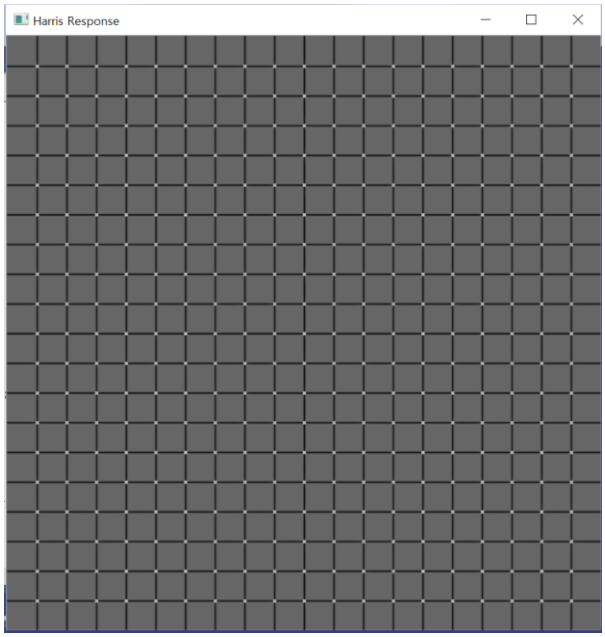
더 정확도 높은 corner detection을 위하여 cornerSubPix() 함수를 사용하였다.

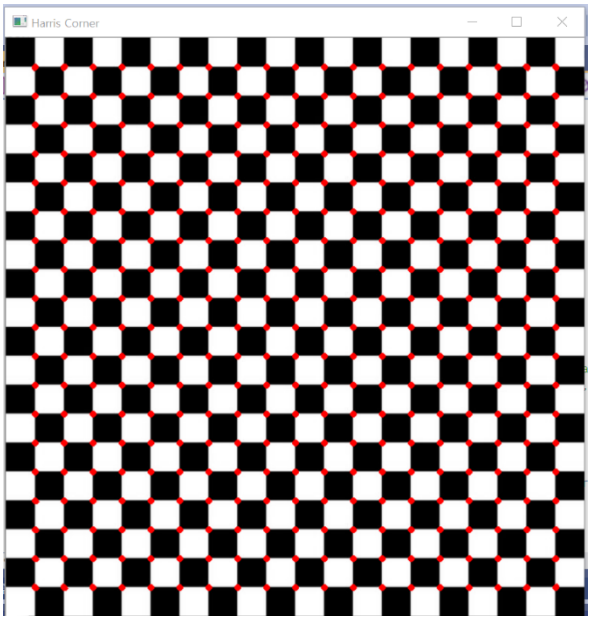
```
Mat temp=Mat::zeros(2 * radius + 1, 2 * radius + 1, input_mirror.type());
//Fill the code////////////////////////////////////////////////////////
for (int a = -radius; a < radius; a++) {
    for (int b = -radius; b < radius; b++) {
        temp.at<float>(a + radius, b + radius) = input_mirror.at<float>(i + a, j + b);
    }
}

double minVal, maxVal;      Point minLoc, maxLoc;
minMaxLoc(temp, &minVal, &maxVal, &minLoc, &maxLoc);
if (Point(radius, radius) == maxLoc) {
    corner_mat.at<uchar>(i - radius, j - radius) = 1;
    input_mirror.at<float>(i, j) = 0;
}
else {
    corner_mat.at<uchar>(i - radius, j - radius) = 0;
}
```

이 코드는 non-maximum suppression을 구현한 코드이다. Non-maximum suppression을 하는 방법이 일정한 size의 window를 기준으로 하여 (i, j)가 가장 큰 값을 가지면 corner이고, 그렇지 않으면 corner가 아니기 때문에 버리는 것이다. Corner_mat을 이용하여 corner임을 1값을 주어 나타내고, corner가 아님을 0 값을 주어서 나타냈다. 또한, corner로 이미 선택된 경우에는 무시하고 계산해야 하기 때문에 모든 intensity 값 중 가장 작은 값인 0을 주었다.

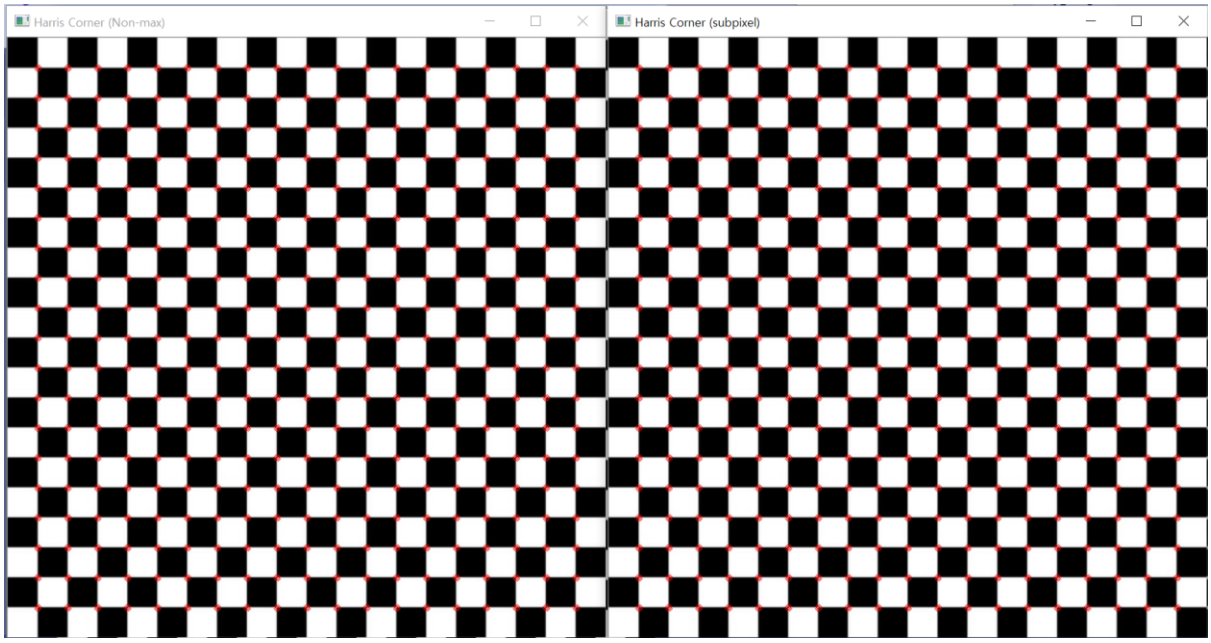
c) The output of the code

(1) Harris Response / Harris Corner





(2) Harris Corner(non-max) / Harris Corner(subpixel)



d) Analysis

cornerHarris() 함수를 사용하여 corner detection을 한 것 보다 non-maximum suppression을 적용한 결과 이미지에, corner를 체크하기 위해 그린 원이 더 적은 것을 확인할 수 있다. 즉, 중복되어 체크된 corner들이 줄어든 것이다.