Technical report-Assignment06

컴퓨터공학 류다현(2376087)

LoG.cpp

이미지의 에지를 검출하려면 1차 미분을 계산해 기울기가 급격히 변화하는 지점을 에지로 간주할 수 있으며, 에지 부분에서 미분값의 부호가 바뀌는 특성을 이용해 2차 미분을 에지 검출에 사용할 수도 있다. 그러나 이러한 방식은 노이즈에 민감하므로, 도함수를 계산하기 전 가우시안 필터 등으로 이미지를 먼저 smooting한 뒤 라플라시안 필터를 적용하는 LoG 방식이 일반적으로 사용된다.

1. int main()

```
int window_radius = 2;
double sigma_t = 2.0;
double sigma_s = 2.0;
Mat h_f = Gaussianfilter(input_gray, window_radius, sigma_t, sigma_s)
Mat h_f_RGB = Gaussianfilter(input, window_radius, sigma_t, sigma_s);
```

커널 반지름 2, 수평/수직 방향 표준편차를 2로 Gaussianfilter 함수를 호출해 input_gray 이미지에 가우시안 필터를 적용한 h_f와 input에 가우시안 필터를 적용한 h_f_RGB를 생성한다.

```
Mat Laplacian = Laplacianfilter(h_f);
Mat Laplacian_RGB = Laplacianfilter(h_f_RGB);
```

h_f와 h_f_RGB에 라플라시안 필터를 적용한 것을 각각 Laplacian과 Laplcian_RGB에 저장한다.

```
normalize(Laplacian, Laplacian, 0, 1, NORM_MINMAX);
normalize(Laplacian_RGB, Laplacian_RGB, 0, 1, NORM_MINMAX);
```

이후 opencv에서 제공하는 normalize() 함수로 LoG를 적용한 이미지들을 정규화하는데, normalize()는 함수는 배열의 값을 정규화하여 특정 범위나 norm에 맞추는 함수로, 다음과 같은 구조이다.

```
void cv::normalize(     InputArray         src,
                        InputOutputArray   dst,
                        double             alpha = 1,
                        double             beta = 0,
                        int                norm_type = NORM_L2,
                        int                dtype = -1,
                        InputArray         mask = noArray() )
```

src와 dst는 각각 입력과 출력 이미지이며 dst의 픽셀 값이 [alpha, beta]범위가 되도록, 즉 $mindst(I) = alpha, maxdst(I) = beta$이도록 조정한다. 이후 생성된 이미지들을 별도의 창에 출력한다.

2. Mat get_Gaussian_Kernel(int n, double sigma_t, double sigma_s, bool normalize)

x축, y축 방향 표준편차 sigma_t, sigma_s로 (2n+1)*(2n+1) 크기 커널을 배열을 생성해 가우시안 커널 공식 $w(s,t) = \frac{1}{\sum_{m=-a}^{a}\sum_{n=-b}^{b}\exp\left(-\frac{m^2}{2\sigma_s^2}-\frac{n^2}{2\sigma_t^2}\right)}\exp\left(-\frac{s^2}{2\sigma_s^2}-\frac{t^2}{2\sigma_t^2}\right)$에 따라 커널 값을 저장하고 완성된 커널을 반환한다.

```
int kernel_size = (2 * n + 1);
Mat kernel = Mat::zeros(kernel_size, kernel_size, CV_64F);
double kernel_sum = 0.0;
```

kernel_size는 커널 한 변의 길이이고, kernel_size*kernel_size 크기 커널 배열을 선언한 후 커널 값 변수

kernel_sum을 0으로 초기화한다.

```
for (int i = -n; i <= n; i++) {
    for (int j = -n; j <= n; j++) {
            kernel.at<double>(i + n, j + n) = exp(-((i * i) / (2.0 * sigma_t * sigma_t) + (j *
            j) / (2.0 * sigma_s * sigma_s)));
            kernel_sum += kernel.at<double>(i + n, j + n);
    }
}
```

이를 kernel_sum에 더한다. 공식에 따라 $\exp\left(-\frac{s^2}{2\sigma_s^2}-\frac{t^2}{2\sigma_t^2}\right)$을 계산해 kernel[i+n][j+n] 위치에 저장하고 이를 kernel_sum에 더한다.

```
if (normalize) {
    for (int i = 0; i < kernel_size; i++)
            for (int j = 0; j < kernel_size; j++)
                    kernel.at<double>(i, j) /= kernel_sum;                    // normalize
}
```

전달받은 normalize 파라미터가 true라면 커널의 모든 원소를 kernel_sum으로 나눠 각 원소들의 합이 1이 되도록 정규화한다.

3. Mat get_Laplacian_Kernel()

3*3 라플라시안 커널을 생성해 반환한다.

```
Mat kernel = Mat::zeros(3, 3, CV_64F);

kernel.at<double>(0, 1) = 1.0;
kernel.at<double>(2, 1) = 1.0;
kernel.at<double>(1, 0) = 1.0;
kernel.at<double>(1, 2) = 1.0;
kernel.at<double>(1, 1) = -4.0;

return kernel;
```

[[0, 1, 0], [1, -4, 1], [0, 1, 0]] 구성이며, 별도 for문 없이 커널 내 0이 아닌 원소들에 값을 직접 할당한다.

4. Mat Mirroring(const Mat input, int n)

입력 이미지 input 둘레에 n 크기의 패딩을 적용한 (rows+2n)*(cols+2n) 크기의 input2 이미지를 생성하고 이 이미지에 mirroring으로 경계 처리하여 필터 적용 시 경계 값 손실을 방지한다.

```
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;
```

row, col은 입력 input의 높이와 너비이며, 이들 값에 2n을 더해 이미지 둘레 전체를 n만큼 확장한 input2 배열을 생성해 0으로 초기화한다.

4.1 if (input.type() == CV_64F)

입력 이미지가 grayscale일 때 다음과 같이 계산한다.

```
for (int i = n; i < row + n; i++) {
```

```
        for (int j = n; j < col + n; j++) {
                input2.at<double>(i, j) = input.at<double>(i - n, j - n);
        }
}
for (int i = n; i < row + n; i++) {
        for (int j = 0; j < n; j++) {
                input2.at<double>(i, j) = input2.at<double>(i, 2 * n - j);
        }
        for (int j = col + n; j < col2; j++) {
                input2.at<double>(i, j) = input2.at<double>(i, 2 * col - 2 + 2 * n - j);
        }
}
for (int j = 0; j < col2; j++) {
        for (int i = 0; i < n; i++) {
                input2.at<double>(i, j) = input2.at<double>(2 * n - i, j);
        }
        for (int i = row + n; i < row2; i++) {
                input2.at<double>(i, j) = input2.at<double>(2 * row - 2 + 2 * n - i, j);
        }
}
```

경계 픽셀이 아닌 경우 출력 이미지의 (i+n, j+n)에 원본 이미지 (i, j) 픽셀을 그대로 복사하고, 왼쪽 경계 열의 픽셀들에 출력 이미지의 (2n-j)열 픽셀들을, 오른쪽 경계의 픽셀들에는 2*(col-2+2n-j)열의 픽셀들을 복사한다. 위쪽 경계와 아래쪽 경계 행에도 마찬가지로 (2n-i)행의 픽셀과 2*(row-2+2n-i)행의 픽셀들을 복사해 미러링한다.

4.2 if (input.type() == CV_64FC3)

입력 이미지가 컬러일 때 다음과 같이 미러링한다.

```
for (int i = n; i < row + n; i++) {
      for (int j = n; j < col + n; j++) {
              input2.at<Vec3d>(i, j) = input.at<Vec3d>(i - n, j - n);
      }
}
for (int i = n; i < row + n; i++) {
      for (int j = 0; j < n; j++) {
              input2.at<Vec3d>(i, j) = input2.at<Vec3d>(i, 2 * n - j);
              for (int j = col + n; j < col2; j++) {
                      input2.at<Vec3d>(i, j) = input2.at<Vec3d>(i, 2 * col - 2 + 2 * n - j);
              }
      }
      for (int j = 0; j < col2; j++) {
              for (int i = 0; i < n; i++) {
                      input2.at<Vec3d>(i, j) = input2.at<Vec3d>(2 * n - i, j);
              }
              for (int i = row + n; i < row2; i++) {
                      input2.at<Vec3d>(i, j) = input2.at<Vec3d>(2 * row - 2 + 2 * n - i, j);
              }
      }
}
```

grayscale의 경우와 동일하지만, input과 input2에 접근할 때 3채널 double 벡터용 Vec3d 타입을 사용한다.

```
return input2;
```

이후 확장 및 미러링된 input2 이미지를 반환한다.

5. Mat Gaussianfilter(const Mat input, int n, double sigma_t, double sigma_s)

입력 이미지 input에 가우시안 필터를 적용한 출력 output 이미지를 반환한다.

```
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);
```

전달받은 n, sigma_t, sigma_s 파라미터와 정규화=true 옵션으로 get_Gaussian_kernel() 함수를 호출해 가우시안 커널을 생성하고 이를 kernel에 저장한다. 입력 input과 크기와 타입이 동일한 출력 이미지 배열 output을 선언해 0으로 초기화하고, Mirroring() 함수를 사용해 input을 미러링한 input_mirror 배열을 생성한다.

5.1 if (input.type() == CV_64F)

grayscale 이미지일 때 아래와 같이 처리한다.

```
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) *
				input_mirror.at<double>(i + a, j + b);
			}
		}
		output.at<double>(i - n, j - n) = (double)sum;
	}
}
```

n부터 row+n까지, n부터 col+n까지 전체 픽셀에 커널을 적용하며, 내부의 2중 for문으로 커널의 모든 픽셀들에 대해 커널 값, 즉 가중치와 입력 픽셀 값을 곱해 sum에 누적한다. 이후 결과 이미지에 누적합 sum을 저장하는데, 이때 output은 input과 동일한 크기이므로 인덱스를 i-n, j-n으로 하여 저장한다.

5.2 if (input.type() == CV_64F)

컬러 이미지일 때 아래와 같이 처리한다.

```
for (int i = n; i < row + n; i++) {
	for (int j = n; j < col + n; j++) {
		Vec3d sum = Vec3d(0, 0, 0);

		for (int a = -n; a <= n; a++) {
			for (int b = -n; b <= n; b++) {
				sum += kernel.at<double>(a + n, b + n) * input_mirror.at<Vec3d>(i +
				a, j + b);
			}
		}
		output.at<Vec3d>(i - n, j - n) = sum;
	}
```

```
}
```

grayscale의 경우와 동일하지만 input_mirror 이미지와 output 이미지에 접근할 때 Vec3d 타입을 사용한다.

6. Mat Laplacianfilter(const Mat input)

입력 이미지 input에 라플라시안 필터를 적용한 output 이미지를 반환한다.

```
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);
```

row, col, output은 위와 동일하다. get_Laplacian_Kernel() 함수를 호출해 kernel에 라플라시안 커널을 저장하고, Laplacianfilter() 함수는 커널 크기 파라미터를 전달받지 않으므로 별도의 패딩 크기 변수 n을 선언해 1로 설정하고 Mirroring 함수를 사용해 input을 미러링한 것을 input_mirror에 저장한다.

가우시안 필터 함수와 동일하게 grayscale 이미지와 컬러 이미지를 모두 처리할 수 있게 구현한다. 입력이 grayscale일 경우 input_mirror 이미지에 double 타입을, 컬러일 경우 Vec3d를 사용한다.

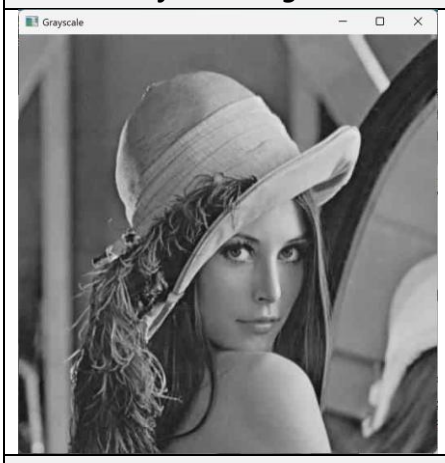
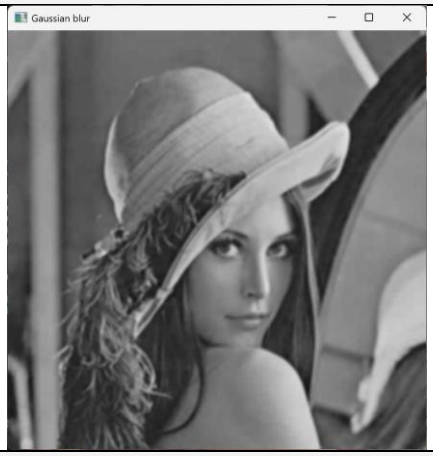
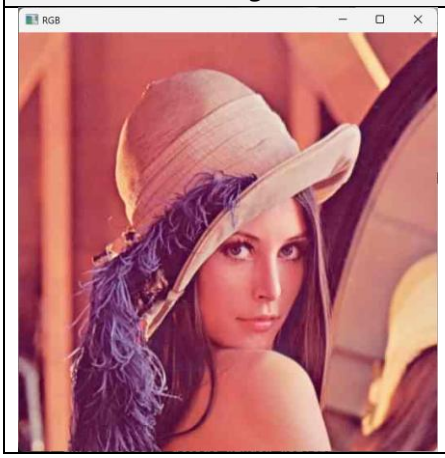
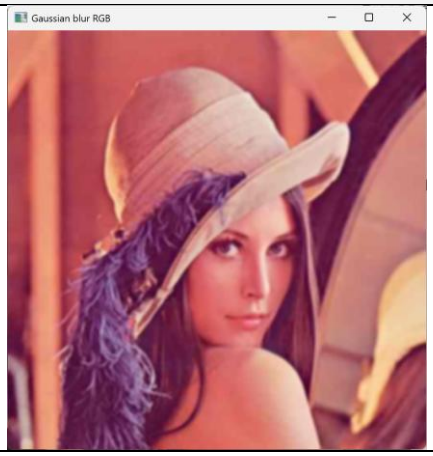
```
if (input.type() == CV_64F) {    // grayscale
    for (int i = n; i < row + n; i++) {
        for (int j = n; j < col + n; j++) {
            double sum = 0.0;

            for (int a = -n; a <= n; a++) {
                for (int b = -n; b <= n; b++) {
                    sum += kernel.at<double>(a + n, b + n) * input_mirror.at<double>(i +
                    a, j + b);
                }
            }
            output.at<double>(i - n, j - n) = (double)sum;
        }
    }
}
else {        // color
    for (int i = n; i < row + n; i++) {
        for (int j = n; j < col + n; j++) {
            Vec3d sum = Vec3d(0, 0, 0);

            for (int a = -n; a <= n; a++) {
                for (int b = -n; b <= n; b++) {
                sum += kernel.at<double>(a + n, b + n) * input_mirror.at<Vec3d>(i + a, j +
                b);
                }
            }
            output.at<Vec3d>(i - n, j - n) = sum;
        }
    }
}
```

커널 내 모든 픽셀들에 대해 가중치*입력 픽셀 연산한 값을 sum에 누적하고, 반복문이 종료되면 output

이미지에 누적합 sum을 저장한다. 마찬가지로 output에 접근할 때 i-n, j-n 인덱스를 사용한다.

7. 결과 분석

lena.jpg를 입력 이미지로 하고 가우시안 필터 커널 반지름과 수직/수평 표준편차를 모두 2로 설정하여 실행하였으며 결과는 아래와 같다.

Grayscale Original	Grayscale Gaussian	Grayscale Laplacian
		
RGB Original	**RGB Gaussian**	**RGB Laplacian**
		

입력 이미지에 가우시안 필터가 적용되어 이미지가 흐려졌으며, 가우시안 필터링한 이미지에 라플라시안 필터를 적용하자 에지가 검출된 것을 확인할 수 있다.

Canny.cpp

1. Void cv::Canny()

Canny.cpp에서는 opencv 제공 Canny() 함수를 사용해 에지를 검출하며, Canny() 함수는 아래와 같다.

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

InputArray와 OutputArray는 각각 8비트 단일 채널의 입력 이미지와 출력 이미지 배열, threshold1과 threshold2는 double thresholding에 사용되는 작은 threshold와 큰 threshold 값이다. apertureSize 파라미터는 (1)에서 소벨 필터를 적용할 때 필터 커널 사이즈이며, L2gradient는 이미지 gradient를 계산할 때 1차 미분만을 사용할지 2차 미분을 사용할지 지정하는 파라미터로 L2gradient가 true일 때 2차 미분한다.

canny edge detector Canny()는 아래와 같이 동작한다.

1) **low-pass 필터와 high-pass 필터를 순차적으로 적용해 gradient 계산**
   노이즈를 제거하기 위해 5*5가우시안 필터로 smoothing을 먼저 수행하고, 이후 apertureSize 크기의 소벨 필터를 적용해 이미지의 수직, 수평 방향 미분 값 $\nabla G = (G_x, G_y) = (\frac{\partial G}{\partial x}, \frac{\partial G}{\partial y})$를 구한다. 이후 G_x, G_y를 이용해 gradient 크기 $M = \sqrt{G_x^2 + G_y^2}$와 방향 $A(x,y) = tan^{-1}(\frac{G_y}{G_x})$을 계산한다.
2) **non-maximum suppression**
   각 픽셀의 $A(x,y)$에 따라 위/아래 또는 대각선의 인접 픽셀 두 개와 M값을 비교하고, 비교 결과 현재 픽셀의 M이 비교 픽셀의 M보다 크면 에지 후보이므로 값을 유지하고 그렇지 않다면 0으로 매핑한다(non-maximum suppression). 이때 A값은 연속적이지만 이미지는 이산적인 값을 가지므로 연속적인 A값을 45° 간격으로 quantization해야 한다.
3) **double thresholding**
   (2)에서 에지 후보로 결정되었던 픽셀들을 threshold와 비교해 한번 더 걸러낸다. 2개의 threshold T_H, T_L을 사용하며, 현재 픽셀의 $M(x,y)$가 threshold2보다 클 경우 에지로 판단해 값을 유지하고 threshold1보다 작을 경우 에지가 아닌 것으로 간주해 0으로 매핑하며, 그 사이 값이라면 에지인 인접 픽셀이 있을 경우에만 에지로 판단한다.

2. int main()

Canny() 함수를 사용한 에지 검출의 전체 과정은 다음과 같다.

```
cvtColor(input, input_gray, COLOR_RGB2GRAY);
input_gray.convertTo(input_gray, CV_8U, 255.0);
```

컬러 이미지 input을 grayscale로 변환해 input_gray에 저장하고, input_gray를 8비트 단일 채널로 변환해 Canny() 함수 파라미터로 전달할 수 있게 한다.

```
double threshold1 = 50.0;
```

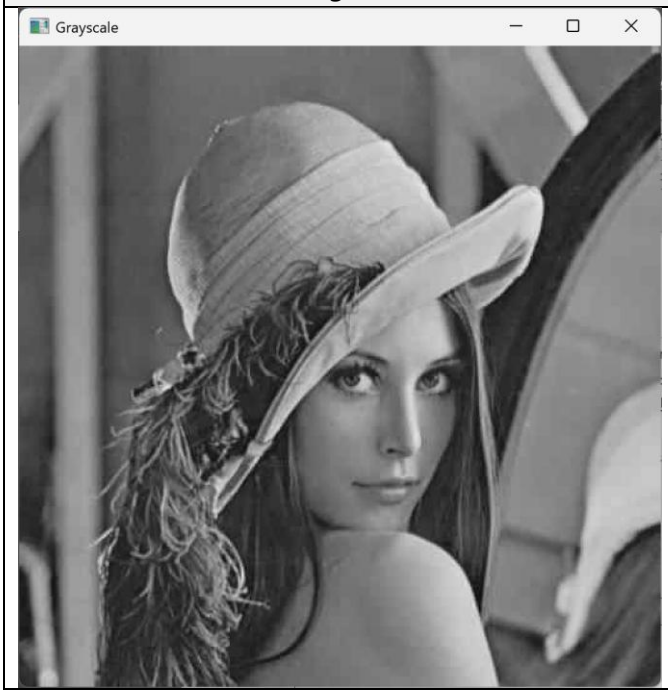
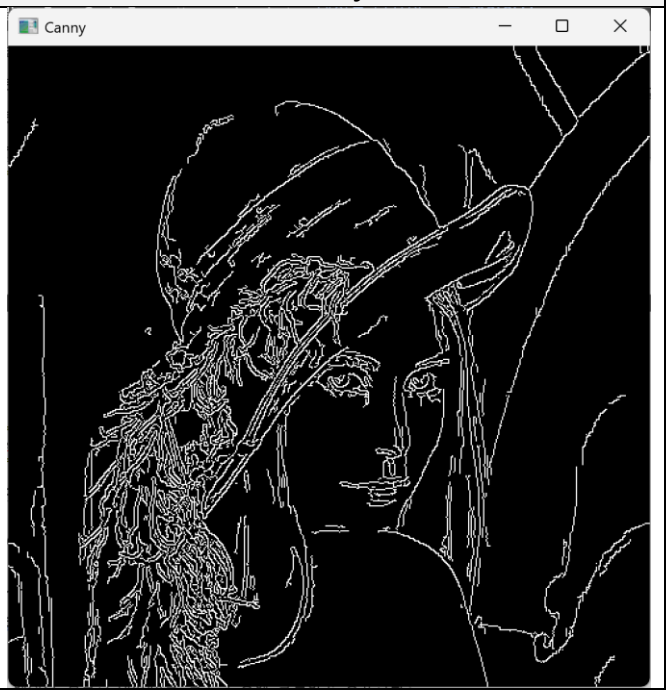
```
double threshold2 = 150.0;
int    apertureSize = 3;
bool   L2gradient = false;

Canny(input_gray, output, threshold1, threshold2, apertureSize, L2gradient);
```

이후 함수 호출에 사용할 파라미터를 초기화하고 Canny() 함수를 호출해 input_gray에 대해 에지 검출한 결과를 output 배열에 저장한다. 이때 낮은 threshold는 50, 높은 threshold는 150으로 설정해 double thresholding 과정에서 gradient M(x, y)가 150 이상인 픽셀은 에지로, 50 이하인 픽셀은 에지가 아닌 것으로 간주하며 (50, 150) 범위의 픽셀은 인접 픽셀이 에지인 경우에만 에지로 판단한다. apertureSize는 3이므로 소벨 필터 는 3*3 크기이며 L2gradient를 false로 설정하여 gradient 계산 시 1차 미분 값을 사용한다.

이후 원본 이미지와 에지가 검출된 output을 별도의 창에 출력하고 종료한다.

3. 결과 분석

lena.jpg를 입력으로 하여 2. 에서 초기화한 설정으로 실행한다.

Original	Canny
	

원본 이미지와 canny edge detection으로 에지가 검출된 이미지가 출력되었다.

Harris_corner.cpp

Harris 코너 검출은 이미지에 윈도우를 설정하고 윈도우를 움직였을 때 변화를 관찰해 해당 픽셀이 flat한 영역인지, 에지에 포함되는지, 코너에 포함되는지 판단하는 기법이다.

이미지 I의 한 픽셀 I(x, y)를 오프셋 (u, v)만큼 이동했을 때의 변화량 E(u, v)는 $E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2$로 계산하며, 이때 w(x, y)는 가우시안 필터 등 픽셀에 가중치를 부여하는 커널이다. E는 $[u\ v]\, M \begin{bmatrix} u \\ v \end{bmatrix}$로 근사할 수 있고 행렬 M은 $M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$, I_x와 I_y는 각각 I를 x와 y에 대해 편미분한 값이다. 이때 픽셀의 코너 여부를 판단하기 위해 행렬의 eigenvalue와 eigenvector를 계산하여 $R = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$를 계산하며, eigenvalue를 구하지 않고 $R = det(M) - \alpha[trace(M)]^2$로 계산할 수도 있다. R>0이면 해당 픽셀을 코너, R<0이면 에지로 판단하고 |R|이 작으면 flat한 영역으로 간주한다.

opencv가 제공하는 Harris 코너 검출 함수는 위 원리를 이용해 픽셀이 코너에 포함되는지 여부를 판단한다.

1. int main()

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

if (!input.data) {
     printf("Could not open\n");
     return -1;
}

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

Mat input_gray, input_visual;
Mat output, output_norm, corner_mat;
vector<Point2f> points;

corner_mat = Mat::zeros(row, col, CV_8U);

bool NonMaxSupp = true;
bool Subpixel = true;

cvtColor(input, input_gray, COLOR_RGB2GRAY);        // convert RGB to Grayscale
```

input은 입력 이미지, input_gray는 input을 grayscale CV_32F 타입으로 변환한 이미지, input_visual은 코너 위치를 표시하기 위해 input을 복제한 이미지 배열이다. output은 cornerHarris() 함수 호출 파라미터로 전달되어 코너 탐색 결과를 저장하며, normalize() 함수로 output을 0에서 1 범위로 정규화한 것을 output_norm에 저장한다. corner_mat 배열은 코너 후보 픽셀 정보를 저장하는 배열이고, points는 corner_mat에서 1인 픽셀 좌표를 모아 만든 배열이다.
NonMaxSupp과 Subpixel은 각각 non-maximum suppression 실행 여부와 cornerSubPix() 함수 호출 여부를 설정하는 변수이다.

```
output = Mat::zeros(row, col, CV_32F);
int blockSize = 2;
int ksize = 3;
double k = 0.04;
cornerHarris(input_gray, output, blockSize, ksize, k);
```

코너 검출 결과를 저장할 output 배열을 선언하고 cornerHarris() 함수를 호출해 Harris 코너 검출을 수행한다. 이때 탐색 윈도우 크기 blockSize를 2로, 소벨 필터 크기 ksize는 3으로, 행렬 M 계산 공식 $R = det(M) - \alpha[trace(M)]^2$에서 α로 사용될 k를 0.04로 설정해 파라미터로 전달한다.

```
normalize(output, output_norm, 0, 1.0, NORM_MINMAX);
namedWindow("Harris Response", WINDOW_AUTOSIZE);
imshow("Harris Response", output_norm);
```

이후 normalize() 함수를 호출해 output 배열의 값들을 0에서 1 범위로 정규화하고 이를 output_norm에 저장하며, output_norm을 "Harris Response" 창에 출력한다.

```
input_visual = input.clone();
double minVal, maxVal;
Point minLoc, maxLoc;

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

다음으로 계산한 output을 이용해 코너 후보 픽셀들을 추출하고 input_visual 이미지에 해당 픽셀들을 표시해 출력한다. input 이미지에 코너 후보 픽셀들을 표시하므로 input_visual은 input을 복제한 것으로 설정한 뒤, global minimum과 maximum 값을 가지는 픽셀 값을 저장할 minVal, maxVal 변수와 해당 픽셀의 위치를 저장할 minLoc, maxLoc 포인터를 선언하고 minMaxLoc() 함수를 호출한다. 함수 호출 결과 output 배열에는 Harris 코너 계산 공식의 R 값이 저장된다.

```
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;
    }
}
```

이후 2중 for문으로 row, col 범위 내 input의 전체 픽셀을 탐색하며 output 값이 0.01*maxVal보다 큰 경우, 즉 R(i, j) 값이 큰 경우 코너 후보로 간주하여 corner_mat(i, j)에 1을 저장하고 circle() 함수를 사용해 input_visual에 빨간 원으로 표시한다. corner_num 변수는 코너 픽셀 개수를 저장하며, for문 내부에서 코너 후보 픽셀을 발견할 때마다 corner_num을 1 증가한다.

```
printf("After cornerHarris, corner number = %d\n\n", corner_num);
namedWindow("Harris Corner", WINDOW_AUTOSIZE);
imshow("Harris Corner", input_visual);
```

cornerHarris 호출 후 코너 개수 corner_num과 input_visual을 출력해 검출된 코너가 표시된 이미지를 "Harris Corner" 윈도우에 출력한다.

```
if (NonMaxSupp) {
    …
```

```
}
```

NonMaxSupp=true로 설정되었을 경우 non-maximum suppression을 수행한다.

```
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++;
          }
     }
}
```

2중 for문을 순회하며 input_visual 배열에 코너 픽셀을 표시하는 것은 위와 동일하지만, 그 전에 NonMaximum_Suppression() 함수를 호출해 후보 픽셀 클러스터 중 local maximum만을 남기고 나머지 픽셀들을 0으로 변경하고 코너로 판단된 픽셀들을 corner_mat 배열에 저장하는 과정을 거친다. 이후 output이 아닌 local maximum 픽셀 배열인 corner_mat을 사용해 픽셀을 표시하고 corner_num을 1 증가하는 작업을 수행한다.

```
printf("After non-maximum suppression, corner number = %d\n\n", corner_num);
namedWindow("Harris Corner (Non-max)", WINDOW_AUTOSIZE);
imshow("Harris Corner (Non-max)", input_visual);
```

마찬가지로 코너 픽셀 수 corner_num과 non-maximum suppression해 검출한 결과 이미지를 출력한다.

```
if (Subpixel) {
     …
}
```

Subpixel=true로 설정된 경우 cornerSubPix() 함수를 호출해 코너 픽셀들을 subpixel 단위로 세밀하게 조정한다.

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

points = MatToVec(corner_mat);

Size zeroZone = Size( -1, -1 );
TermCriteria criteria = TermCriteria( TermCriteria::EPS + TermCriteria::COUNT, 40, 0.001 );
cornerSubPix(input_gray, points, subPixWinSize, zeroZone, criteria);
```

subPixWinSize는 탐색 윈도우 반지름, termcrit은 종료 조건, zeroZone은 무시할 zero zone 크기를 저장한다. MatToVec() 함수로 corner_mat 배열에서 값이 1인 픽셀들을 벡터로 변환하여 points에 저장하고, 이들을 파라미터로 cornerSubPix() 함수를 호출해 input_gray에서 points 배열의 코너 픽셀을 subpixel 단위로 조정한 것을 다시 points에 저장한다.

```
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);
}
```

이후 동일하게 input_visual에 코너 픽셀을 표시한다. 이때 points 배열 내에 이미 코너 픽셀들이 저장되어 있으므로 위 코드에서 조건문을 사용해 output 또는 corner_mat 배열의 픽셀이 일정 값 이상인지 별도로 판단했던 과정 없이 for문에서 points 배열의 값을 바로 표시하는 것으로 구현한다.

```
printf("After subpixel-refinement, corner number = %d\n\n", points.size());
    namedWindow("Harris Corner (subpixel)", WINDOW_AUTOSIZE);
    imshow("Harris Corner (subpixel)", input_visual);
```

같은 이유로 corner_num 변수를 설정하지 않고 points 배열의 크기를 코너 픽셀 수로 출력하며, subpixel refine된 결과 이미지를 "Harris Corner(subpixel)" 창에 출력한다.
사용된 함수들의 자세한 설명은 아래와 같다.

2. **cv::cornerHarris()**

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

src는 단일 채널 8비트 또는 floating point 입력 이미지이고, src에 대해 R을 계산해 출력 dst 배열에 저장한다. 이때 dst는 CV_32FC1 타입이며 src와 동일한 크기이다. blockSize 파라미터는 행렬 M을 계산할 때 사용할 윈도우 크기 변수로, 인접한 blockSize*blockSize 크기의 윈도우 영역 내에서 I_x, I_y 를 계산하여 해당 픽셀의 행렬 M 요소를 구하는 데 사용한다. ksize는 소벨 필터 크기 변수, k는 harris detector free parameter로 위 공식에서 α이다.

3. **cv::cornerSubPix()**

cornerSubPix() 함수는 코너를 보다 정확하게 검출할 수 있도록 한다. 검출된 코너 픽셀을 subpixel 수준으로 정확하게 조정할 수 있게끔 하며, 다음과 같이 정의된다.

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

image는 8비트 또는 float형의 단일 채널 입력 이미지 배열이며, corners는 입력 이미지에 함수를 적용한 후 정제된 코너 정보를 담고 있다. winSize는 코너 조정 시 사용하는 탐색 범위 윈도우의 반지름, zeroZone은 검색 중 summation에 포함하지 않을 dead zone의 반지름 파라미터이다. criteria는 함수의 종료 조건으로, 아래 코드에서는 오차(벡터 크기 변화)가 일정 수준 이하로 줄어들거나 최대 반복 횟수에 도달하면 함수를 종료하는 것으로 구현되어 있다.

4. **cv::minMaxLoc()**

minMaxLoc() 함수는 주어진 배열에서 전역(global) 최소 및 최댓값과 그 위치를 찾는 함수이다.

```
void cv::minMaxLoc     ( InputArray       src,
                         double *         minVal,
                         double *         maxVal = 0,
                         Point *          minLoc = 0,
                         Point *          maxLoc = 0,
                         InputArray       mask = noArray() )
```

단일 채널 배열인 입력 이미지 src에서 이미지 전체에 대해 픽셀 정보를 추출하거나, mask 파라미터로 탐색할 영역이 지정되어 있다면 해당 영역 내에서 최소 및 최대 픽셀을 탐색한다. 이때 minVal과 minLoc은 탐색한 최소 픽셀 값과 해당 값이 나타나는 위치를 가리키는 포인터이며, maxVal과 maxLoc 또한 최대 픽셀 값과 그 위치를 가리키는 포인터이다.

5. **vector<Point2f> MatToVec(const Mat input)**

MatToVec() 함수는 입력 input 중 값이 1인 코너 픽셀들을 추출해 픽셀 좌표를 벡터 형태로 반환한다.

6. **Mat NonMaximum_Suppression(const Mat input, Mat &corner_mat, int radius)**

NonMaximum_Suppression() 함수는 radius 파라미터 반경 이내에서 local maximum인 픽셀 하나만을 남기고 나머지 픽셀들을 0으로 매핑한다. input은 전체 입력 이미지, corner_mat은 이전에 계산한 코너 후보 픽셀들을 저장하고 있는 배열이다. 이때 전달된 input은 코너 후보가 아닌 픽셀이 0으로 처리된 이미지이다.

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

Mat input_mirror = Mirroring(input, radius);
Mat corner_mat_clone = corner_mat.clone();
```

탐색 윈도우 반지름이 radius이므로 Mirroring() 함수를 호출해 입력 이미지 input의 둘레를 radius만큼 확장하며, 이때 Mirroring() 함수는 위에서 서술한 것과 동일하다.

```
for (int i = radius; i < row+radius; i++) {
     for (int j = radius; j < col+radius; j++) {
          if (input_mirror.at<float>(i, j) > 0) {
               float current = input_mirror.at<float>(i, j);
                    bool isMax = true;

                    for (int a = -radius; a <= radius; a++) {
                         for (int b = -radius; b <= radius; b++) {
                              if (input_mirror.at<float>(i + a, j + b) > current) {
                                   isMax = false;
                                   break;
                              }
                         }
                    }
                    if (!isMax) {
                         corner_mat_clone.at<uchar>(i - radius, j - radius) = 0;
                    }
                    else {
                         corner_mat_clone.at<uchar>(i - radius, j - radius) = 1;
                    }
               }

     }
}
```

radius부터 row+radius까지, radius부터 col+radius까지 2중 for문으로 전체 픽셀을 탐색하고, 이때 input을 복제한 input_mirror에는 input과 마찬가지로 코너 후보가 아닌 픽셀에 0이 저장되어 있으므로 픽셀 값이 0이 아닌 경우에만 연산해도 무방하다.

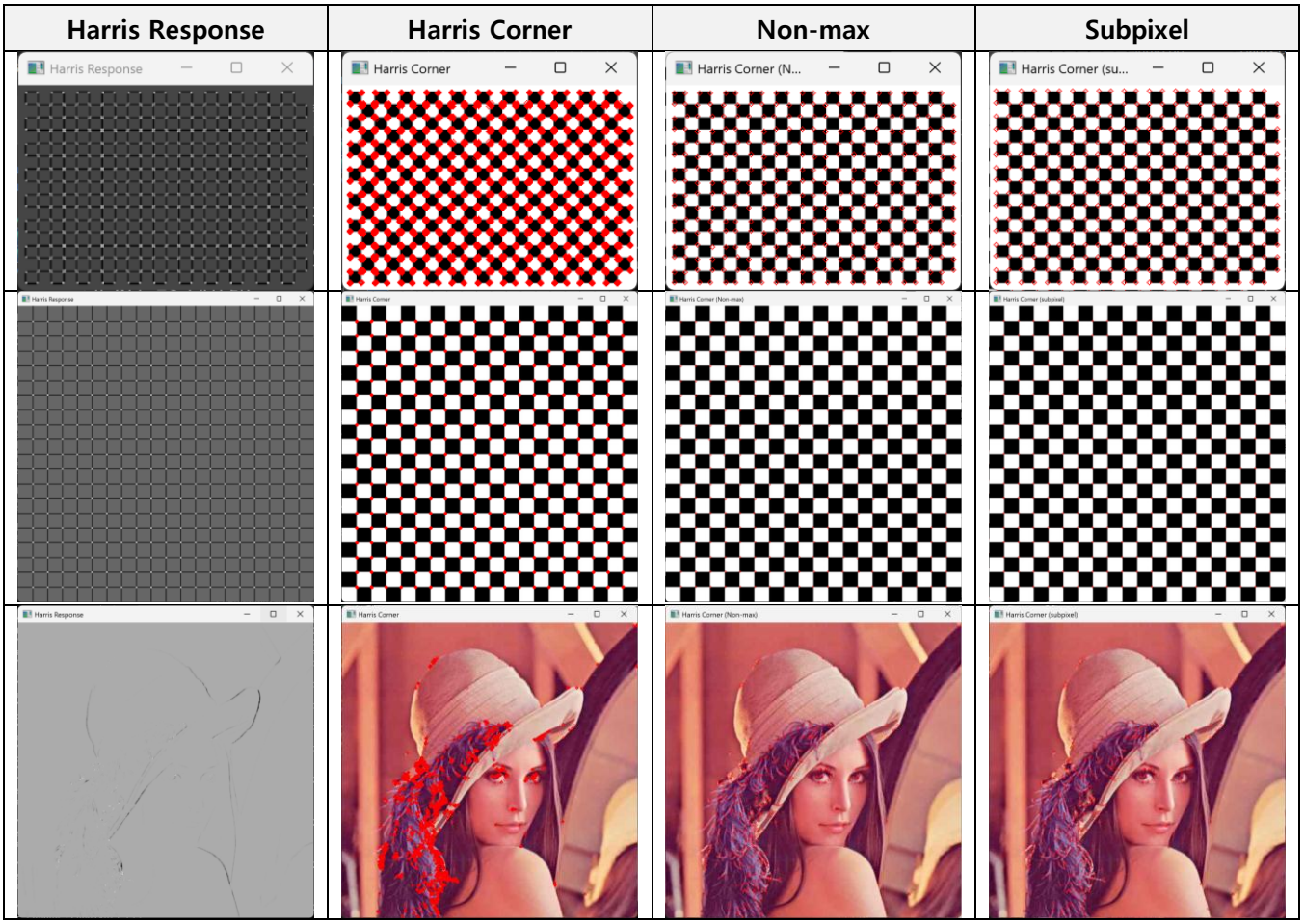
current 변수에 현재 탐색할 픽셀 input_mirrori(i, j)를 저장하고, current가 local maximum인지 판단할 boolean형 isMax 변수를 선언해 true로 초기화한다. 탐색이 진행되며 current가 local maximum이 아니라면 isMax를 false로 변경하고 반복문을 종료하며, 반복문을 빠져나온 후 조건문을 통해 isMax=false라면, 즉 current가 local maximum이라면 출력 코너 배열 corner_mat_clone의 (i, j) 픽셀을 0으로, current가 local maximum이라면 corner_mat_clone(i, j)를 1로 변경한다.

```
corner_mat = corner_mat_clone;
```

이후 corner_mat에 non-maximum suppression된 코너 배열 corner_mat_clone을 저장하고 함수를 종료한다.

7. 결과 분석

변수들은 위에서 설명한 것과 동일하게 설정하였으며, non-maximum suppression과 subpixel refine을 수행하도록 하여 "checkerboard.png", "checkerboard2.jpg", "lena.jpg" 3개의 이미지로 각각 실행시킨 결과는 다음과 같다.

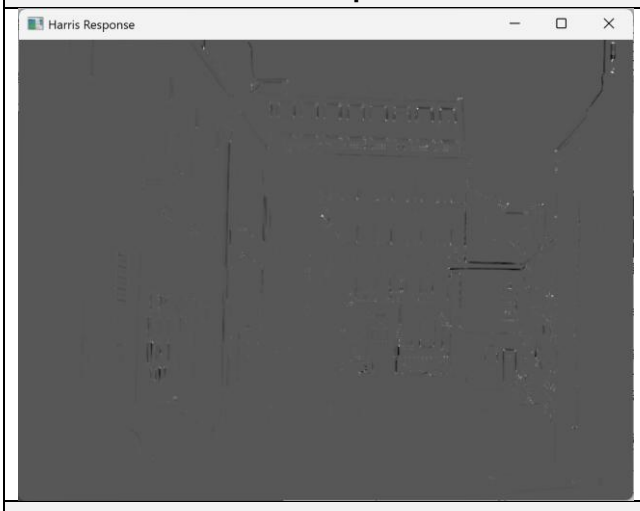
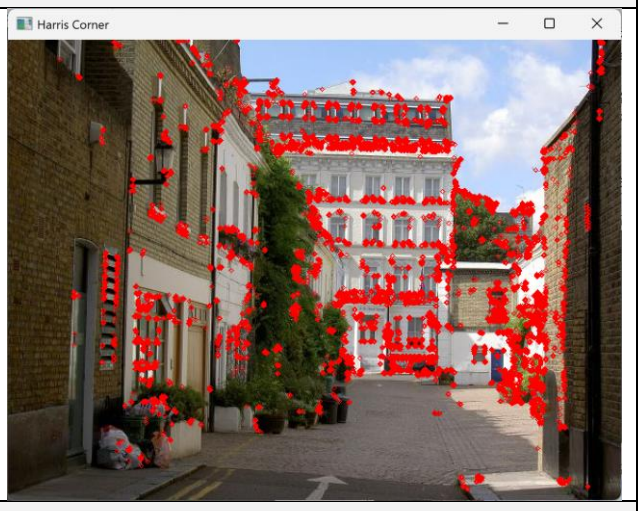
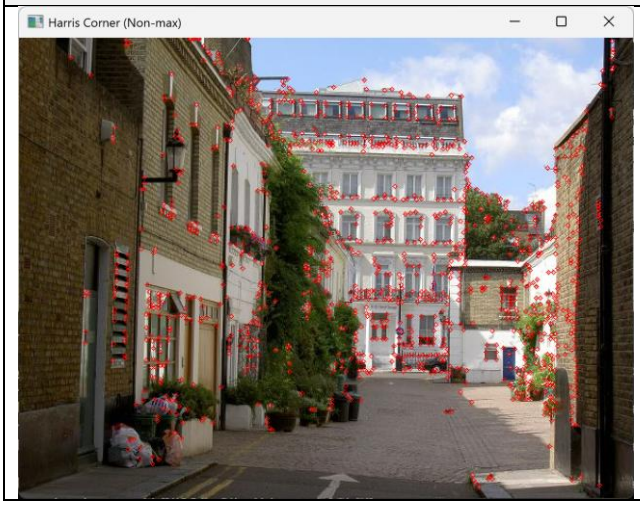
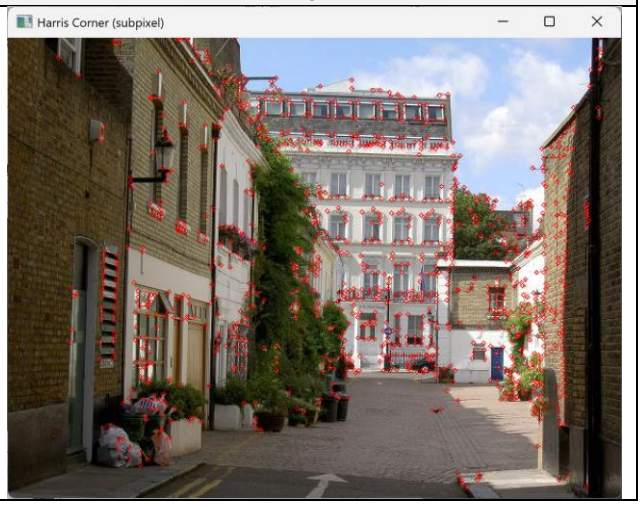


```
[ INFO:0@0.320] global window_w32.cpp:2996 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Harris Response (1)
After cornerHarris, corner number = 3543

[ INFO:0@0.634] global window_w32.cpp:2996 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Harris Corner (1)
After non-maximum suppression, corner number = 368

[ INFO:0@0.708] global window_w32.cpp:2996 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Harris Corner (Non-max) (1)
After subpixel-refinement, corner number = 368
```

세 경우 모두 cornerHarris() 함수만으로 코너를 검출한 것보다 non-maximum suppression과 subpixel refine했을 때 검출된 코너 픽셀 개수가 더 적었는데, 이는 non-maximum suppression으로 클러스터 내의 local maximum을 제외한 픽셀이 코너 후보에서 제거되고 subpixel refine으로 약한 코너 후보가 제외되었기 때문이다.

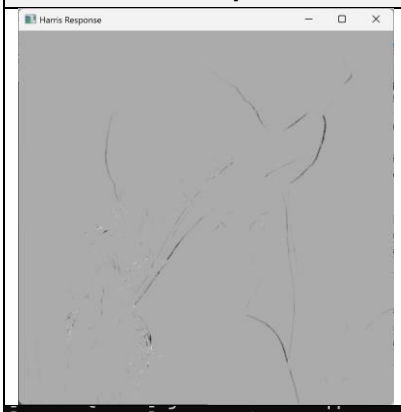
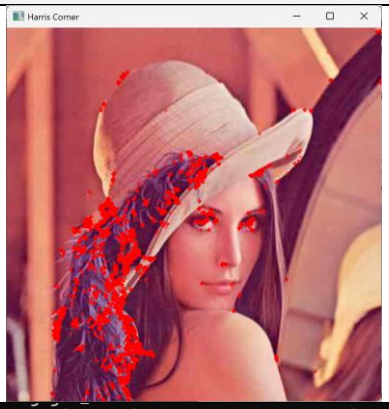
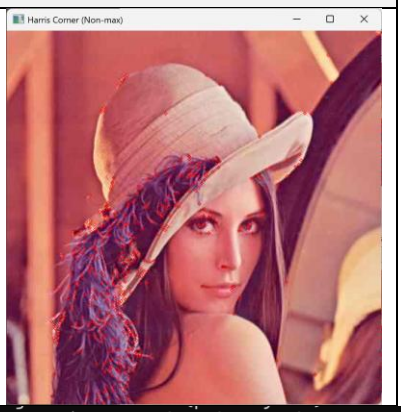
또, NonMaxSupp와 Subpixel을 모두 true로 설정한 경우 subpixel refine에서 사용하는 corner_mat 배열은 이미 non-maximum suppression 과정을 거친 것으로, cornerSubPix() 함수는 전달된 코너 픽셀들을 subpixel 수준에서 조정할 뿐 픽셀 개수를 변경하지 않으므로 두 경우에서 동일한 코너 픽셀 수가 출력되었다.

코너가 강조되는 다른 이미지로 실행하면 아래와 같이 코너가 검출된 것과 옵션별 차이를 더욱 뚜렷하게 확인할 수 있다.

Harris Response	Harris Corner
	
Non-max	**Subpixel**
	

7.1 NonMaxSupp만 true인 경우

NonMaxSupp=true, Subpixel=false로 설정하고 "lena.jpg"를 사용해 실행하면 결과는 다음과 같다.

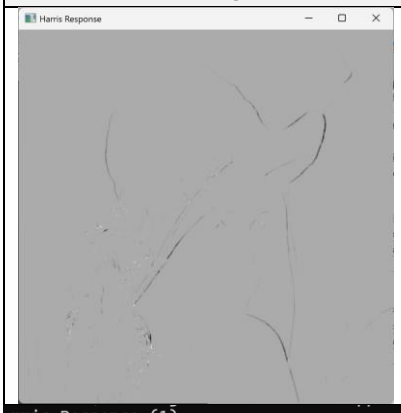
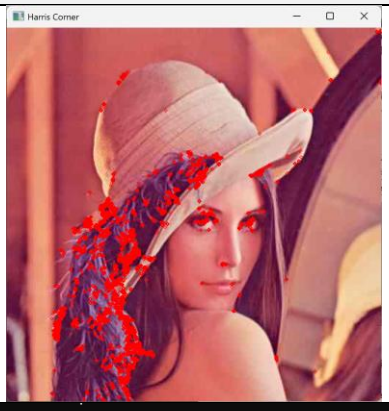
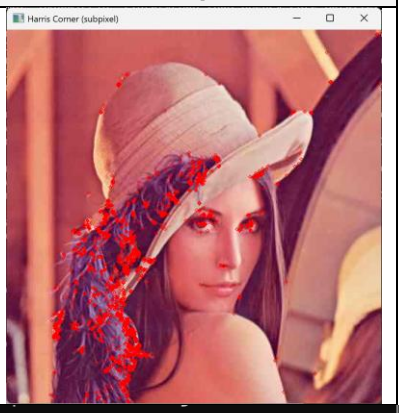
Harris Response	Harris Corner	Non-max
		

```
[ INFO:0@0.320] global window_w32.cpp:2996 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Ha
rris Response (1)
After cornerHarris, corner number = 3154

[ INFO:0@0.557] global window_w32.cpp:2996 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Ha
rris Corner (1)
After non-maximum suppression, corner number = 524
```

cornerHarris() 함수만 적용하면 3154개의 코너 픽셀이 검출되고, non-maximum suppression을 적용했을 때 코너 픽셀 수가 524개로 줄어들었다. 출력 이미지에서도 전자의 경우 후보 픽셀 개수가 더욱 많고 후자는 클러스터별 하나의 local maximum 픽셀만 보존되어 픽셀 개수가 적은 것을 확인할 수 있다.

7.2 Subpixel만 true인 경우

NonMaxSupp=false, Subpixel=true로 설정하고 "lena.jpg"를 사용해 실행하면 결과는 다음과 같다.

Harris Response	Harris Corner	Subpix
		

```
rris Response (1)
After cornerHarris, corner number = 3154

[ INFO:0@0.554] global window_w32.cpp:2996 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Ha
rris Corner (1)
After subpixel-refinement, corner number = 3154
```

subpixel refine만 수행한 경우 non-maximum suppression만 적용했을 때와 달리 cornerHarris() 적용 결과와 검출된 코너 픽셀 수가 동일한 것을 확인할 수 있는데, 이는 cornerSubPix() 함수가 전달된 픽셀 배열의 픽셀 수를 변경하는 것이 아니라 픽셀의 위치를 세부 조정하기 때문이다.