

# C언어 기반 디지털 영상처리



만든이

류성수

제작 기간

2024.03.22 ~ 05.23

# 소개



전화번호 010-2470-3160

이메일 dbtjdtm325@gmail.com

## 간략한 소개

안녕하세요, 저는 전자공학을 전공한 류성수입니다. 학부 시절부터 현재까지 통신 및 신호처리 분야에 깊은 관심을 가지고 있습니다. 동시에, 컴퓨터 과학에도 큰 흥미를 가지고 있어 주로 C와 C++ 언어를 사용하여 임베디드, 이미지 처리 프로젝트 등을 진행해왔습니다.

저의 목표는 배워왔던 지식을 결합하여 디테일을 놓치지 않는 엔지니어가 되는 것입니다. 이를 통해 기술적 문제를 해결하고, 보다 효율적이고 발전된 시스템을 구현하는 데 기여하고자 합니다.

# 목차

- 프로젝트 오버뷰 [\(4.p\)](#)
- 솔루션 구성 [\(6.p\)](#)
- 대략적인 영상처리 과정 [\(7.p\)](#)
- 순서도 [\(8.p\)](#)
- 기능 설명 [\(10.p\)](#)
- 문제 해결과정 [\(44.p\)](#)
- 성취도 및 소감 [\(52.p\)](#)

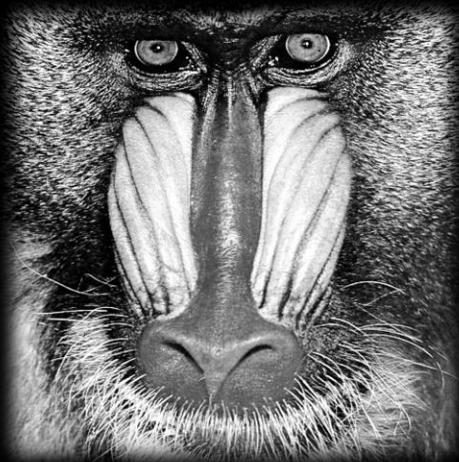


# 프로젝트 오버뷰

## 프로젝트 목표

Digital Image Processing (3rd) 의 주요기능을 직접  
C언어만을 사용하여 전부 구현해보는 것

## 구현 기능



### 1. 개별 화소 처리

- 반전처리
- 파라볼라처리
- 감마처리

### 2. 화소 영역 처리

- 블러링처리
- 샤프닝처리
- 엣지검출 처리
- 엠보싱 처리
- 중간값 청렬처리

### 3. 기하학적 처리

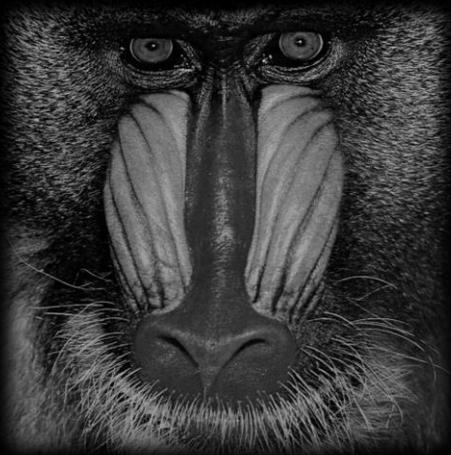
- 확대 처리
- 축소 처리
- 회전 처리

### 4. 통계학적 처리

- 히스토그램 평활화처리
- 히스토그램 스트레칭처리
- 이진화처리

// 총 16개의 기능

# 프로젝트 오버뷰



## 프로젝트의 고유특징

1. 커널 크기를 직접 제어할 수 있다
2. 이미지 처리를 위한 Value를 직접 제어할 수 있다
3. 시간복잡도를 줄이기 위해 선형대수적인 방법을 사용한다
4. 효율을 위해 BPP를 8로 설정하였다 -> 흑백

## GitHub 주소

[https://github.com/supergravityy/Image\\_Processing](https://github.com/supergravityy/Image_Processing)

## 사용언어

C

## 개발환경

Visual Studio 2022 (version : 17.9.6)  
Window 11 Home (version : 23H2)

# 솔루션 구성



## Sources



### Main.c

// 사용자의 파일 입력과 원하는 이미지 변환 모드를 입력 받는 소스



### Convert.c

// 해당 이미지를 불러오고 처리된 이미지 쓰는 소스



### BMP2TXT.c

// bmp 파일 자체를 TXT로 보기위한 소스



### Revise.c

// bmp의 헤더정보를 프로젝트 규칙으로 수정하는 소스



### Blurring.c , ...

// 원하는 이미지 처리를 하기 위해 각 기능들을 구현한 소스



## Headers



### Convert.h

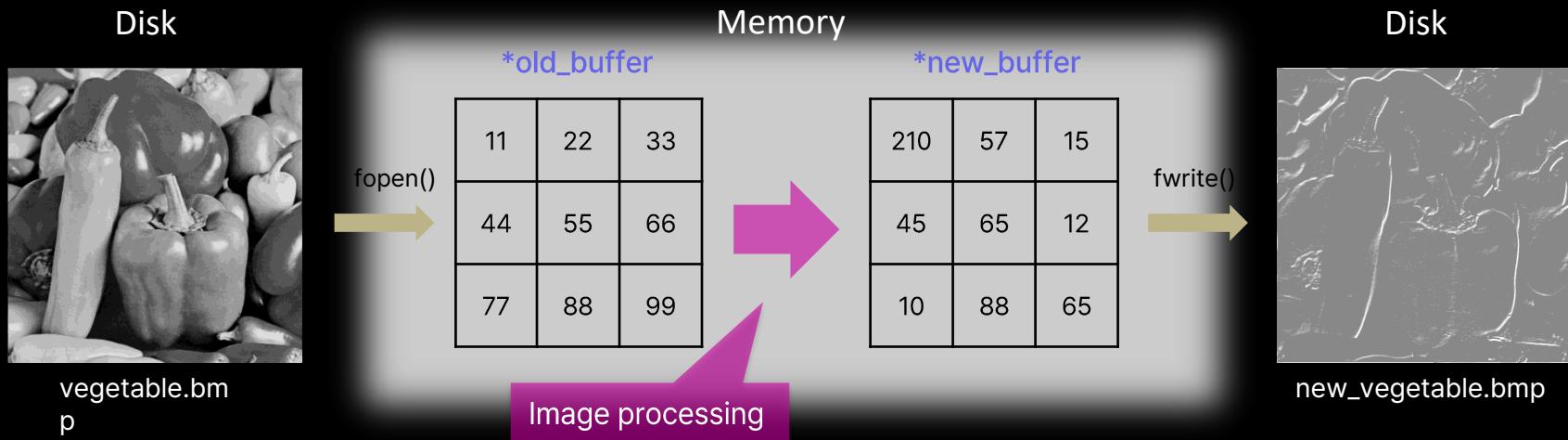
// 입력에 필요한 함수들과 bmp 서식, 구현에 필요한 라이브러리들



### Processing.h

// 이미지 변환에 필요한 기능 함수원형, define들

# 대량적인 영상처리 과정



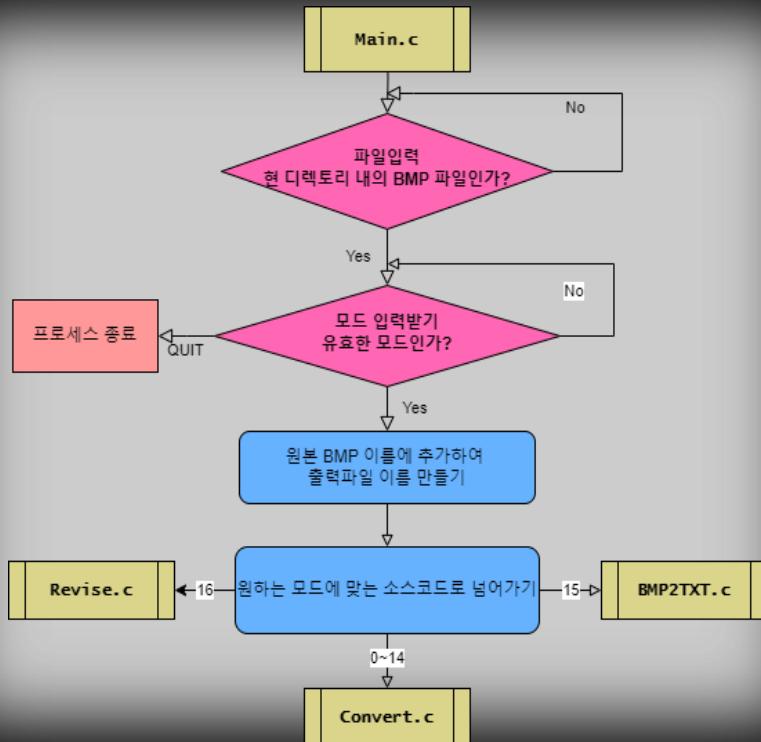
1. BMP파일의 RAW를 추출해서 가져오기

2. 원하는 알고리즘을 이용해 수정, 필요시 헤더 수정

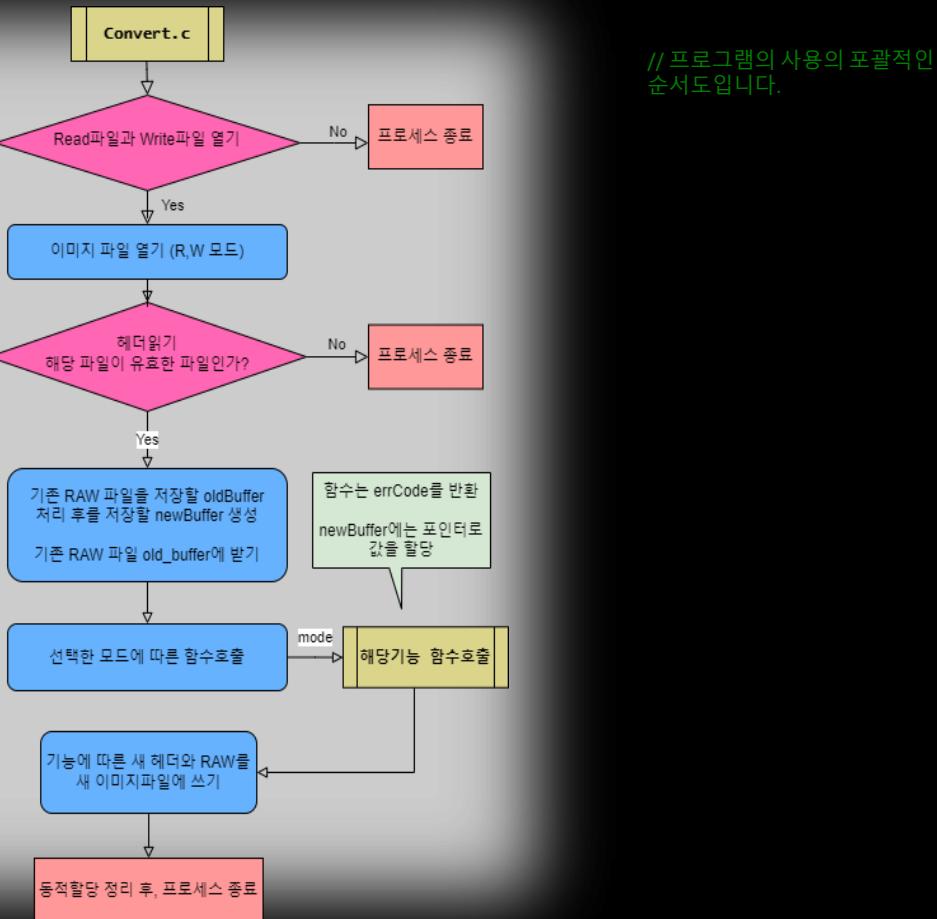
3. 수정된 RAW 파일과 헤더로 새 BMP 작성

# 순서도

// 프로그램의 사용의 포괄적인  
순서도입니다.



// 이전 슬라이드의 내용을 순서도로  
나타내었다



# 기능설명

## 1. 개별화소 처리 (11.p~12.p)

화소점의 원래 값이나 위치를 기준으로 화소값을 변경하는 알고리즘

- \* 반전처리
- \* 감마처리
- \* 포물선 처리

## 2. 화소영역 처리 (13.p~28.p)

기준 화소와 주변의 화소를 기반으로 기준 화소값을 변경하는 알고리즘

- \* 블러링 처리
- \* 샤프닝 처리
- \* 엣지 검출
- \* 중간값 검출
- \* 엠보싱 처리

## 3. 기하학 처리 (29.p~35.p)

화소점의 위치나 배열을 변경시키는 알고리즘

- \* 축소 처리
- \* 확대 처리
- \* 회전 처리

## 4. 통계학 처리 (36.p~43.p)

이미지의 전체 화소 히스토그램을 분석하여, 통계학과 연관시켜 처리하는 알고리즘

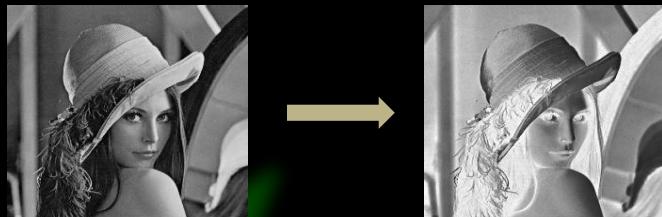
- \* 히스토그램 평활화
- \* 히스토그램 스트래칭
- \* 이진화 처리

# 1. 개별화소처리

## 반전처리 (Inverting.c)

화소의 Maximum 값 (=255)에서 해당 화소값을 빼면  
반전처리가 된다.  
여기서는 픽셀값에 not 비트연산을 취해서 완성하였다.

$$y = 255 - x$$



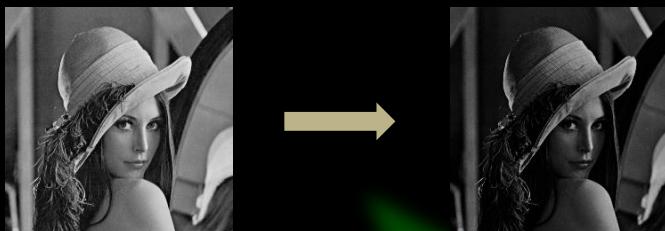
```
/*
// 이미지 반전은 그냥 기존이미지의 0xxxxxxxxx 에 전체적으로 not 연산을 하면 끝난다
*/
for (int h = 0; h < infoheader->height; h++)
    for (int w = 0; w < infoheader->width; w++)
        new_buffer[h * infoheader->width + w] = ~old_buffer[h * infoheader->width + w];
```

## 감마처리 (Gamma\_correcting.c)

디지털 이미지의 밝기와 대비를 조정하는 과정이다. 이는  
인간의 시각 특성 혹은, 디스플레이 장치의 특성을 맞추기  
위해 사용된다.

$$y = 255 \times \left(\frac{x}{255}\right)^r$$

// ← 해당 감마값은 사용자에게 입력받는다



```
/*
// 2. 감마값에 따른 계산수행
*/
double normalizedPXL;
double correctedPXL;

for (int idx = 0; idx < infoheader->imageSize; idx++)
{
    normalizedPXL = old_buffer[idx] / MAX_DOB_BRIT_VAL;
    correctedPXL = pow(normalizedPXL, gamma);
    new_buffer[idx] = (BYTE)round(correctedPXL * MAX_DOB_BRIT_VAL);
```

## 파라볼라 처리(Parabola\_processing.c)

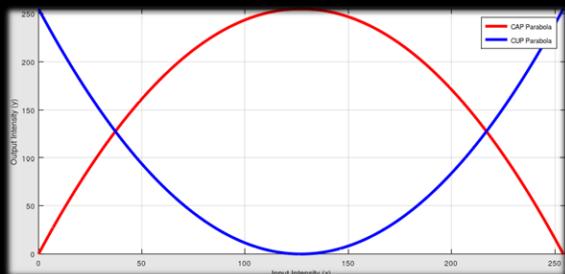
파라볼라 처리에서 다음 두 변환은 이미지의 밝은 영역과 어두운 영역을 강조하거나, 감소시키는데 사용된다.

\* CAP(center around point) : 입력 밝기 값이 중간에서 최대 값을 가지며, 양쪽 끝으로 갈수록 감소하게 변환한다.

$$y = 255 \left( \frac{x}{127} - 1 \right)^2$$

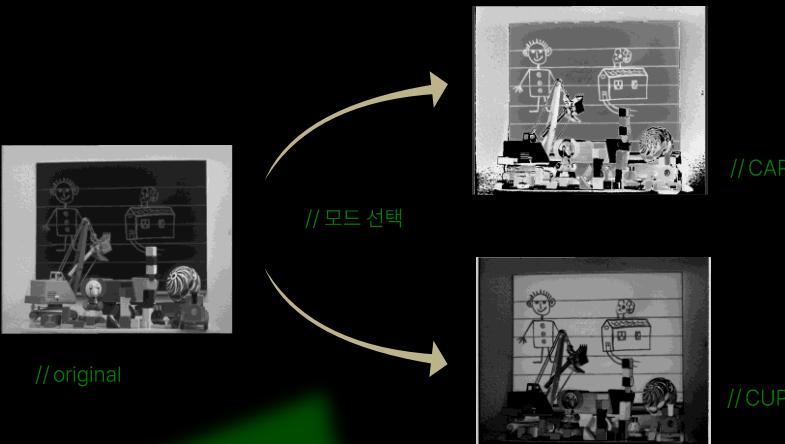
\* CUP(center up point) : 입력 밝기 값이 중간에서 최소 값을 가지며, 양쪽 끝으로 갈수록 증가하게 변환한다.

$$y = -255 \left( \frac{x}{127} - 1 \right)^2 + 255$$



// prarbola는  
포물선이라는 의미를  
갖는다

// 두 수식을 그래프로  
시각화하였다(matlab)



```

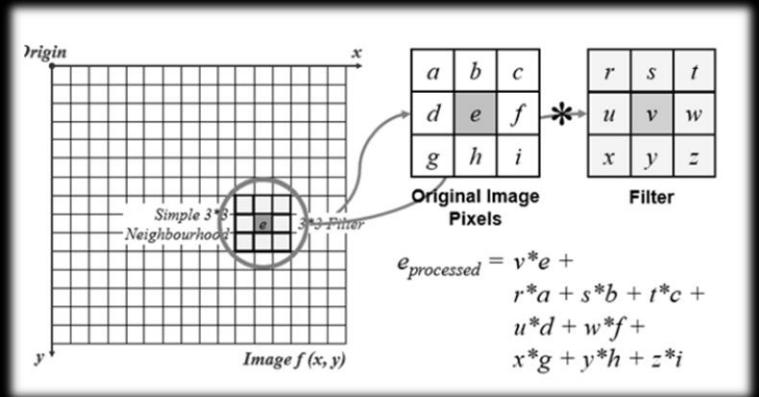
double temp;
int denominator = MAX_BRIT_VAL / 2 + 1; // 식을 한줄로 쓰기위해서 이렇게 함

switch (option)
{
case 1:
printf("WnYou select CAP parabola!Wn");
for (int idx = 0; idx < infoheader->imageSize; idx++)
{
    temp = MAX_DOB_BRIT_VAL - MAX_DOB_BRIT_VAL * pow(old_buffer[idx] / ((double)denominator) - 1., 2);
    new_buffer[idx] = (BYTE)clipping((int)round(temp));
}
break;
// CAP 파라볼라 처리는 밝은 부분을 더 강조한다
// 밝은 부분을 기준으로 임체적으로 보임

case 2:
printf("WnYou select CUP parabola!Wn");
for (int idx = 0; idx < infoheader->imageSize; idx++)
{
    temp = MAX_DOB_BRIT_VAL * pow(old_buffer[idx] / ((double)denominator) - 1., 2);
    new_buffer[idx] = (BYTE)clipping((int)round(temp));
}
break;
// CUP 파라볼라 처리는 어두운 부분을 더 강조한다
// 어두운 부분을 기준으로 임체적으로 보임

```

## 2. 화소 영역처리



### 기본개념 설명

컨볼루션 마스크를 이용한 화소영역처리는 이미지의 특정영역을 처리하기 위해 **컨볼루션 연산**을 사용하는 기법이다.

먼저, 원하는 기능에 맞는 커널을 정의하고, 대상 픽셀의 주변값과 컨볼루션을 취하여, 새로운 픽셀값을 계산한다.

대상픽셀의 주변값이 없을 때를 대비하여, **다양한 방법**을 사용한다.



// 컨볼루션  
계산과정을  
코드로 구현

```
BYTE regular_cal(BYTE* old_buffer, double* kernel, int curX, int curY, BITMAPINFOHEADER* infohead)
{
    double sum = 0.0;

    int wrapped_j = 0; int wrapped_i = 0;

    int Height = infoheader->height;
    int Width = infoheader->width;
    int Radius = Sizeside / 2;
    int kernel_idx;

    for (int i = curY - Radius; i <= curY + Radius; i++)
    {
        wrapped_i = circular_wrapping(i, Height);

        for (int j = curX - Radius; j <= curX + Radius; j++)
        {
            wrapped_j = circular_wrapping(j, Width);
            kernel_idx = (i - curY + Radius) * Sizeside + (j - curX + Radius);
            sum += (double)old_buffer[wrapped_i * Width + wrapped_j] * kernel[kernel_idx];
        }
    }

    sum = clipping((int)round(sum));
    return (BYTE)sum;
}
```

// 클리핑 함수는 픽셀당 1바이트인 이미지에서  
비트 오버플로와 언더플로를 방지한다

### 왜 하필 컨볼루션인가?

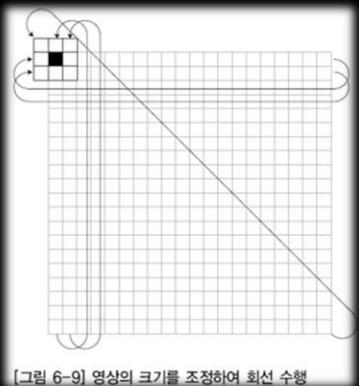
이미지 처리에서 컨볼루션과 코릴레이션의 연산과정은 똑같지만, 연산전에 커널을 반전하느냐에 따른 차이가 있다.

컨볼루션을 코릴레이션 대신 사용하는 이유중 하나는 밑의 그림과 같이 단위 임펄스와 연산하면, 결과가 반전되어서 출력되는 것을 막기 위함이다.



// 코릴레이션은 결과가  
반전되지만, 컨볼루션은  
미리 뒤집었기 때문에,  
결과가 반전되지 않는다.

## 경계값 처리 방법



대상픽셀의 주변값이 없을 때에는 할 수 있는 방법이 크게 두가지로 나뉜다

- \* 없는 부분을 0으로 처리하기
- \* 영상의 시작과 끝부분이 연결된 것으로 처리하는 방법

현재 프로젝트에서는 2번째 방법을 채택하였다.



// 나머지 연산을 통하여 구현하였다.

```
inline int circular_wrapping(int idx, int max)
{
    if (idx < 0)
        return (max + idx) % max;
    return idx % max;
}
```

## 분리 가능한 커널

만약, 커널이 특정조건들을 만족하면, 커널은 행벡터와 열벡터의 외적(outer product)과 같게 된다.

이는 특히, clk가 낮은 임베디드 환경에서 매우 중요한데, 기존의 방법보다 33%의 시간복잡도를 줄일 수 있다.

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times [b_1 \ b_2 \ b_3] = \begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \end{bmatrix} = u \times v = K$$

계산 시, 커널의 원소 하나는 아래와 같기 때문이다.

$K_{ij} = u_i \times v_j$  다음과 같기 때문에, 아래로써도 표현이 가능하다  
 $Image * K = Image * (u \otimes v) = (Image * u) * v$

그렇다면 특정조건이란 무엇인가?

- 커널이 중심 대칭 이어야 한다.
- 각 행, 열이 일정한 스칼라 배수관계를 가져야 한다.

마지막 조건을 선형대수에선 rank가 1이라고 표현한다.

## 분리가능한 커널 코드 구현

일단, 분리가능한 커널이 가능한지의 조건을 구현하였다.

```

int Symmetric = 1;
// 맥, y축 대칭인지를 파악하기 위해선 그냥, 맨 처음과 맨 끝부터 1씩 증가시켜 똑같으면 된다.

for (int i = 0; i < side; i++)
{
    for (int j = 0; j < side; j++)
    {
        if (compareDouble(kernel[i * side + j], kernel[(side-1 - i) * side + (side-1 - j)]))
        // 부동소수점이기에 오차허용범위를 두고 연산을한다.

        {
            Symmetric = 0;
            break;
        }
    }
    if (!Symmetric)
        break;
}

return Symmetric;
}

// is_seperatable 함수의 일부분 rank = 1인지 검사한다.

// 첫 번째 행을 기준으로 행렬의 모든 행이 배수 관계인지 확인
for (int Y = 1; Y < Sizeside; Y++)
{
    double row_ratio = kernel[Y * Sizeside] / kernel[0]; // 첫행과 배수관계에 있다고 가정

    for (int X = 0; X < Sizeside; X++)
    {
        // kernel[X]는 첫번째 행의 원소를 의미
        // 대상 행의 원소들과 첫 행의 원소를 하나씩 빼가며, 차이를 확인

        if (fabs(kernel[Y * Sizeside + X] - row_ratio * kernel[X]) > threshold) // 둘의 차이가 크
            return 0;
    }

    // 첫 번째 열을 기준으로 행렬의 모든 열이 배수 관계인지 확인
    for (int X = 1; X < Sizeside; X++)
    {
        double col_ratio = kernel[X] / kernel[0]; // 첫열과 배수관계에 있다고 가정

        for (int Y = 0; Y < Sizeside; Y++)
        {
            // kernel[Y * Sizeside]는 첫번째 열의 원소를 의미

            if (fabs(kernel[Y * Sizeside + X] - col_ratio * kernel[Y * Sizeside]) > threshold)
                return 0;
        }
    }
}

return 1; // 모두 통과시, 분리가능한 커널
}

```

조건이 둘다 성립하면, 첫번째 행과 열을 분리하여 순차적으로 컨볼루션을 한다.  
만약, 이 둘의 조건을 만족하지 못한다면, 기존의 방식대로 계산한다.

```

if (check_symmetry(kernel, kernel_size) && is_seperable(kernel, Sizeside))
// K = Kx * Ky ^ (T) 의 형태로 풀개지면 분리가능한 커널이다.
// 대체而言 rank = 1 이 해결되어야 한다. 이를 검사하기 위해 조건

```

1. 먼저 첫 행(행벡터)과 이미지를 컨볼루션 해준다.

```

printf("Row Vector of kernel!\n");

for (idx = 0; idx < Sizeside; idx++) // 커널의 행벡터(x축) 나열
{
    vector_buffer[idx] = kernel[idx * Sizeside];
    printf("%lf ", vector_buffer[idx]);
}

printf("\n\nNow row vec calculation is start!\n\n"); // 가로부터 행렬계산

for (int Y = 0; Y < infoheader->height; Y++)
{
    for (int X = 0; X < infoheader->width; X++)
    {
        result = row_cal(old_buffer, vector_buffer, X, Y, infoheader, Sizeside);
        temp_buffer[Y * infoheader->width + X] = (BYTE)result;
        //printf("%d ", (int)result);
    }
}

```

2. rank = 1이어서, 첫 열이 스칼라배일 수도 있기에 정규화를 한다음,  
열벡터 컨볼루션은 위의 단계와 똑같은 과정으로 계산한다.

```

double N = kernel[0];
for (idx = 0; idx < Sizeside; idx++) // 커널의 행벡터(x축)
{
    vector_buffer[idx] = kernel[idx * Sizeside] / N;
    // 위의 분리가능한 커널을 풀개서 다시 병이면, N값이 제곱이 되어서 나오기에 정규화
    printf("%lf ", vector_buffer[idx]);
}

```

// 계산과정은 자리가 없기에 생략되었다.

## 블러링 처리(Blurring.c)

이미지의 detail을 제거하여 영상을 흐리게 하거나 부드럽게하는 기술. 여기서, 적절한 커널을 사용하면, 이미지의 디테일을 적절히 감소시킬 수 있다.

이미지에서 대상이 나뉘어지는 기준은 화소값이 변화하는 구간이다.

이러한 detail은 주파수의 관점으로 생각해보면, 고주파 성분으로 볼 수 있는데, 블러링은 [low-pass 필터](#)의 역할을 하게 된다.

또한, 블러링 처리는 픽셀영역을 건들지 않아야 하기에 [총합이 1](#) 이어야한다. (조건)

이와 같은 조건 블러링 처리는 크게 두가지의 커널(필터)로 나뉜다.

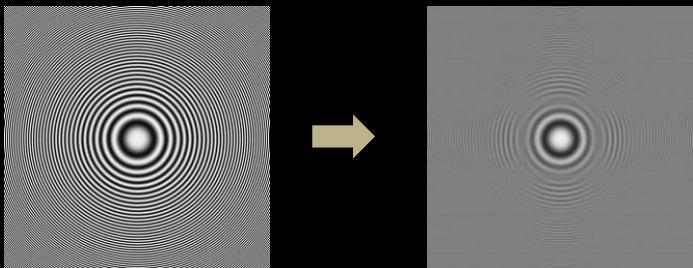
- \* 평균필터 : 주위의 화소들과 평균을 내서 계산되게 하는 필터
- \* 가우시안 필터 : 픽셀로부터 떨어진 거리에 가중치를 주게하는 필터

일단, 블러링을 위해서는 다음과 같은 순서를 거쳐야 한다.

- 원하는 필터 생성
- 해당필터가 분리가능한 커널인지 확인
- 결과에 맞는 계산법 이용
- 새 RAW파일에 작성

### 평균 필터

이와 같은 필터를 사용하여 블러링 처리를 하게 되면 다음과 같은 결과를 얻게된다.



// 원본과 비교하여 디테일이 전체적으로 줄어든 것을 확인할 수 있다. (커널크기 7\*7)

```
double* gen_AVG_kernel(int size)
{
    // 여기서 . 커널을 동적 할당한다
    double* AVG_kernel = (double*)malloc(sizeof(double) * size);
    double sum = 0;
    int idx;

    for (idx = 0; idx < size; idx++)
    {
        AVG_kernel[idx] = 1;
        sum += AVG_kernel[idx];
    }

    for (idx = 0; idx < size; idx++)
        AVG_kernel[idx] = 1 / sum;

    // 평균필터 완성!
    return AVG_kernel;
}
```

// 평균필터의 각 요소는  
필터의 전체 크기의 역수와  
같다!

## 가우시안 필터

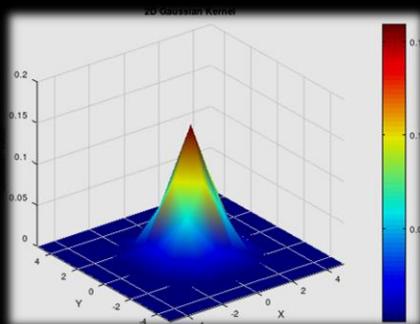
기존의 평균 필터는 영상의 디테일을 완전히 뭉개버리는 식으로 작업이 되었다.

이 커널을 왜 쓰는가?

그러나, 위와같은 효과대신, 고주파 성분을 부드럽게 제거하고, 어느 정도의 디테일을 살리고 싶을때 사용한다  
이러한 필요성을 위해서, 2차원 가우시안 분포도를 사용할 수 있다.

왜냐하면, 가우시안 커널과의 컨볼루션은 중심으로부터 대상까지의 거리에 따라 가중치를 부여하기 때문이다.

또한, 식의 표준편차값( $\sigma$ )을 제어하여, 요소들의 분포도를 제어하고, 기준( $\sigma=1$ )과는 다른 블러링 효과를 얻을 수도 있다.

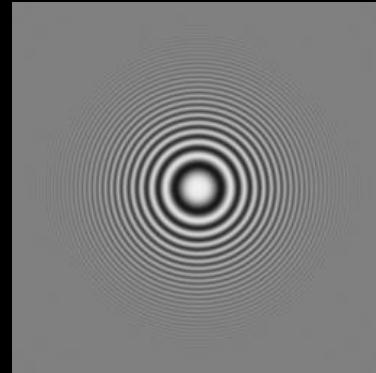


$$f(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

// 위의 식을 매트랩을 사용하여, 3D 그래픽으로 구현하였다.

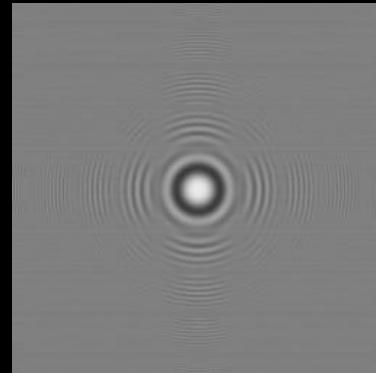
이는 기우시안 커널이 중심으로부터의 거리에 따라 가중치를 부여하게 된다는 것을 잘 보여준다. ( $\sigma=1$ )

가우시안 커널을 만드는 과정의 코드는 여기에 넣기엔 너무 길기 때문에, 생략하였다.



// 가우시안 블러 ( $\sigma = 1$ , 커널크기 81)

오른쪽에서 보이는 두 이미지에서 비교할 수 있듯이, 똑같은 조건에서 블러 처리를 하였지만, 디테일에서 굉장한 차이를 불러온다.



// 평균 블러 (커널크기 81)

## 샤프닝 처리(Sharpening.c)

이전까지 블러링은 영상의 디테일들을 감소시키는 역할을 하였지만, 이번에는 영상의 디테일을 증폭시키는 샤프닝 처리이다.

영상의 디테일은 무엇인가?

영상의 디테일은 픽셀값이 좌표적인 환경에서 변화하는 구간이다. 이는 이미지 상의 물체의 경계, texture을 나타내고, 이러한 디테일은 임펄스 형태로 존재한다.

왜 하필 2차 미분이 필요한가? // ex 광원체에 따른 밝기 그레이션

그러나, 만약, 어떤 물체의 단면이 그레이션 효과처럼 일정하게 색상이 변화된다면, 단지 1차 미분으로써, 이러한 변화마저 감지하기 되기에, 이러한 오차를 줄여서 정확도를 높이기 위해, 변화량의 변화량을 체크한다.

결국, 2차 도함수를 테일러 근사화 시키면, x축과 y축에 대한 2차 차분식(~미분식)은 아래와 같이 나타난다.

$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y)$$

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) - 2f(x, y)$$

식에 의한 커널의 구성

그리고, 이를 이미지의 좌표공간에 대입하려 더하게 되면, 이렇게 된다.

$$\begin{aligned} \nabla^2 f(x, y) &= f(x+1, y) + f(x-1, y) + f(x, y+1) \\ &\quad + f(x, y-1) - 4f(x, y) \end{aligned}$$

그러나, 이는 영상의 디테일을 찾아내는 식이기에, 샤프닝 처리처럼 단지 임펄스만을 증폭시키기 위해선, 입력영상을 더해준다.

$$g(x, y) = f(x, y) + c[\nabla^2 f(x, y)]$$

여기서  $f(x, y)$ 와  $g(x, y)$ 는 각각 입력, 샤프닝된 영상이다.

위 식에서  $c$ 값은  $f(x, y)$ 의 부호와 정 반대여야 한다. ( $c = -1$ )

위 공식은 90도씩 증가하는 회전에 대해 등방성 결과를 제공하는 행렬로 구현될 수 있다.

이로 인해 커널값은 이러한 형태로 표현이 되게된다. (3\*3 기준)

결국, 커널의 총합은 1이어야 한다.

<b>0</b>	-1	<b>0</b>
-1	5	-1
0	-1	0

-1	-1	-1
-1	9	-1
-1	-1	-1

// 오른쪽의 커널 역시 각 대각선 방향마다 2개의 항을 추가하였지만, 디지털 리플리시안의 정의에 편입이 가능하다.

## 샤프닝 효과

샤프닝 커널을 사용하기 위해선, 블러링과 똑같이 원하는 커널을 고르면, 해당 커널을 이용해 컨볼루션을 하게 된다.

현재 프로젝트에선 널리쓰이는 3가지의 커널을 만들어 놓았다.

```
double* gen_HF_kernel()
{
    double* HF_kernel = (double*)malloc(sizeof(double) * KERNEL33);

    double temp[KERNEL33] = { 0., -1., 0., -1., 5., -1., 0., -1., 0. };

    for (int i = 0; i < KERNEL33; i++)
        HF_kernel[i] = temp[i];

    return HF_kernel; // 1번
```

```
double* gen_fHF_kernel()
{
    double* fHF_kernel = (double*)malloc(sizeof(double) * KERNEL33);

    double temp[KERNEL33] = { -1., -1., -1., -1., 9., -1., -1., -1., -1. };

    for (int i = 0; i < KERNEL33; i++)
        fHF_kernel[i] = temp[i];

    // 가장 안정적인 커널이다.
    return fHF_kernel; // 2번
```

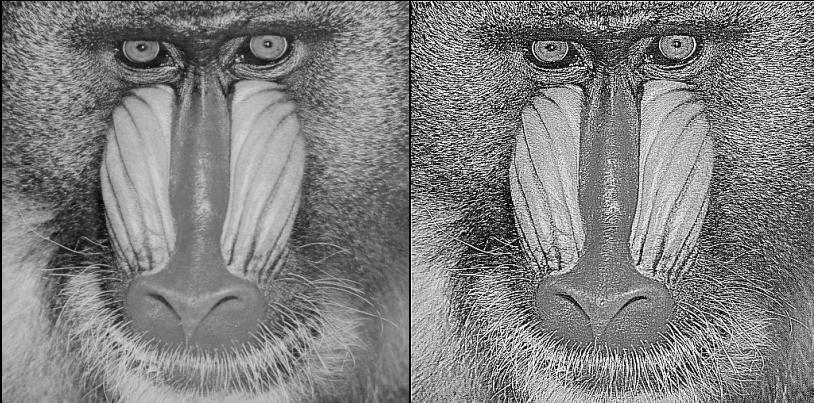
```
//double* gen_fHFs_kernel()
//{
//    double* fHFs_kernel = (double*)malloc(sizeof(double) * KERNEL33);

//    double temp[KERNEL33] = { -1., / 3., 1., / 3., -1., / 3., -1., / 3., 7., / 3., -1., / 3., -1., / 3., 1., / 3., -1., / 3., 1. };

//    for (int i = 0; i < KERNEL33; i++)
//        fHFs_kernel[i] = temp[i];

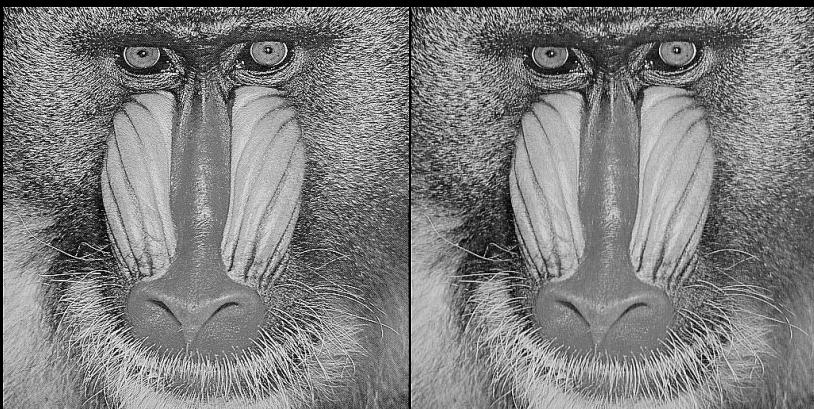
//    return fHFs_kernel; // 3번
```

대체적으로 털 사이의 디테일이 강조된것을 확인할 수 있다



// original

// 1번



// 2번

// 3번

## 중간값 정렬 처리(Median\_filtering.c)

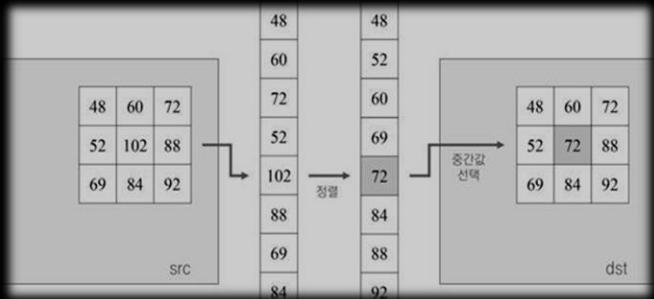
중간값 정렬 처리는 대상 픽셀을 중심으로 미리 정해놓은 커널 크기만큼의 주변 픽셀을 담아서 그 커널속에서 중간 밝기값을 찾아서 새로운 RAW 파일에 넣는 처리 기법이다.

이걸 왜 쓰는가?

**Salt & Pepper 노이즈**: 디지털 이미지에서 흔히 발생하는 일종의 잡음이다. 이 잡음은 이미지에 무작위로 밝은 점과 어두운 점으로 나타난다.

이로 인해 이미지의 특정 픽셀 값이 극단적인 값으로 바뀌게 된다. 이러한 노이즈는 블러링으로도 해결하기 힘든 노이즈이기 때문에, 중간값 정렬 처리를 이용하면, 노이즈를 제거할 수 있다.

소금 후추 노이즈는 극단적인 값을 갖기에, 중간값 정렬처리에 영향을 주지 않는다.



```

for (int i = h - 2; i < h + 3; i++)
{
    wrapped_h = circular_wrapping(i, height);

    for (int j = w - 2; j < w + 3; j++)
    {
        wrapped_w = circular_wrapping(j, width);
        temp[idx] = old_buffer[wrapped_h * width + wrapped_w];
        idx++;
    }
}

for (int i = 0; i < 24; i++)
    for (int j = 0; j < 25 - i; j++)
        if (temp[i] + 1 < temp[i])
            swap(BYTE, temp[i], temp[j+1]);

// 경계 수가 적기 때, 버블정렬사용

return temp[13];

```

동작 순서

- 모든 픽셀들을 순회하면서, 커널크기만큼의 주변픽셀을 커널로 옮긴다.
- 해당 커널을 정렬하여, 중간값을 반환한다.

// 중간값 정렬 처리를 하면, 수채화로 그린 것 같은 효과를 보인다.



## 엠보싱 처리(Embossing.c)

엠보싱 처리는 이미지를 양각(3D) 처럼 보이게 만드는 처리방법이다. 이 방법은 샤프닝 처리와 마찬가지로 2차미분 필터의 바리에이션이지만, 이미지의 모든 밝기값을 회색으로 맞춰, 이미지의 디테일 부분(경계)에서 차이를 계산하여 입체감을 부여한다.

일반적으로 대각선 방향으로 픽셀값을 비교하고, 그 차이를 계산하는 방식이다.

### 필터 구현모습

```
if (option == 1)
{
    double temp[9] = { -1,-1,0,0,0,0,0,1,1 };
    for (int i = 0; i < 9; i++)
        kernel[i] = temp[i];
}
```

-1	-1	0
0	0	0
0	1	1

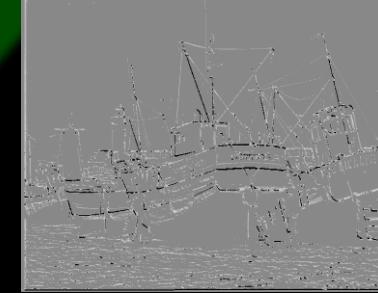
```
else
{
    double temp[9] = { 0,-1,-1,0,0,0,1,1,0 };
    for (int i = 0; i < 9; i++)
        kernel[i] = temp[i];
}
```

0	-1	-1
0	0	0
1	1	0

일단, 화소영역처리와 똑같이 원하는 기능의 커널을 생성하고, 해당 커널과 대상픽셀부터 주변픽셀까지를 컨볼루션한다.

하지만, 현재 필터의 총합이 0이기에, 0~255까지의 값을 가질수 있는 픽셀 밝기값의 절반인 128을 더해준 후, 혹시모를 비트 오버런을 방지하기 위해 클리핑을 거친다.

```
for (int h = 0; h < infoheader->height; h++)
{
    for (int w = 0; w < infoheader->width; w++)
    {
        temp = 128 + regular_cal(old_buffer, kernel, w, h, infoheader, 3);
        new_buffer[h * infoheader->width + w] = clipping(temp);
    }
}
```



// 엣지가 분명한 곳에서는 엣지검출처럼 엣지를 검출해낸다.  
엠보싱 처리가 엣지검출의 바리에이션임을 나타내는 좋은 예이다.

## 엣지 검출 처리(Edge\_detecting.c)

일단 엣지검출에 필요한 라플라시안 필터는 샤프닝 처리 과정에서 언급하였다.

### 샤프닝처리와 엣지검출처리의 차이?

다만, 엣지검출처리와 샤프닝 처리와의 다른점은 샤프닝은 기준 영상에서 디테일(임펄스 값)만을 증폭시키는 방법이었고,

엣지검출처리는 정말 디테일만을 남기고 전부 0으로 만드는 방법이다.

그렇기 때문에, 커널의 총합은 0이 되어야 한다.

결론적으로, 엣지검출 커널은 블러링 커널과 정반대로 high-pass 필터의 역할을 한다.

엣지검출처리에선 정말 다양한 필터를 구현하였다.

#### 구현된 필터종류

- \* Roberts 필터
- \* Prewitt 필터
- \* Sobel 필터
- \* 2차 라플라시안 필터
- \* LoG (Laplacian of Gaussian) 필터
- \* DoG (Difference of Gaussian) 필터

//특히 LoG와 DoG의 특징은 가우시안 블러링과정처럼 표준편차값을 제어할 수 있다.

### 1차 미분필터

Roberts 필터 & Prewitt 필터 & Sobel 필터들이 속한다.

이 두 필터는 1차 미분필터로 작동하는 필터이다.

1차 도함수와 2차 도함수를 근사화 시킨 식과 비교해 본다면

$$\begin{aligned}\nabla f(x) &= f(x+h) - f(x) \\ \nabla^2 f(x) &= f(x+h) - 2f(x) + f(x-h) \quad (h = 1)\end{aligned}$$

이러한 차이를 보이게 된다. // $f'x$ 를 미분하고,  $x+1$ 을 중점으로 잡는다.

1차 미분필터 : 

1	-1
---	----

2차 미분필터 : 

1	-2	1
---	----	---

위는 각 도함수들의 식을 필터화 시킨 것이다.

현재 프로젝트에서는 1차 미분필터는 x축과 y축의 차를 한번씩 계산하지만, 2차 미분필터는 위와 차이를 두기위해 이 둘을 더했다.

### 엣지 검출의 순서

- 원하는 필터 생성 (1차 미분필터 / 2차 미분필터)
- 필터와 이미지를 컨볼루션  
(1차 미분은 x축 차, y축 차를 차례로 컨볼루션)  
(2차 미분은 한번 컨볼루션)

## Roberts 필터

로버츠 미분필터는 대각선 방향의 차를 계산하는 1차 미분필터이다.

```
if (tryNum == 0)
{
    printf("it is a first Roberts kernel!"); // 첫번째

    double temp[KERNEL33] = { -1,0,0,0,1,0,0,0,0 };

    for (int idx = 0; idx < KERNEL33; idx++)
        roberts[idx] = temp[idx];

    tryNum++;

    return roberts;
}
```

// 몇번째로 호출되나에따라서 생성의차를 두기위해 static 변수를 이용했다.(=tryNum)

```
else
{
    printf("it is a second Roberts kernel!"); // 두번째

    double temp[KERNEL33] = { 0,0,-1,0,1,0,0,0,0 };

    for (int idx = 0; idx < KERNEL33; idx++)
        roberts[idx] = temp[idx];

    return roberts;
}
```

<b>-1</b>	<b>0</b>	<b>0</b>
0	<b>1</b>	0
0	0	0

<b>0</b>	<b>0</b>	<b>-1</b>
0	<b>1</b>	0
0	0	0



## Prewitt 필터

단순히 한축으로의 차이만 계산하는 보편적인 1차미분필터이다.

```
double temp[KERNEL33] = { -1,0,1,-1,0,1,-1,0,1 };

for (int idx = 0; idx < KERNEL33; idx++)
    prewitt[idx] = temp[idx];

tryNum++;

return prewitt;
```

<b>-1</b>	<b>0</b>	<b>1</b>
-1	0	1
-1	0	1

// x축(열) 끼리의 차를 계산

```
double temp[KERNEL33] = { 1,1,1,0,0,0,-1,-1,-1 };

for (int idx = 0; idx < KERNEL33; idx++)
    prewitt[idx] = temp[idx];

return prewitt;
```

<b>-1</b>	<b>-1</b>	<b>-1</b>
0	0	0
1	1	1

// y축(행) 끼리의 차를 계산



지금 보는것과 같이 1차 미분의 한계를 보여준다. 야채의 디테일들이 검출되긴 했지만, 텍스처의 질감과 그라데이션마저 그대로 노출되었다.

또한, low-pass 필터를 거치지 않아, 노이즈도 노출 되었다.

## Sobel 필터

소벨 필터는 이전의 두 필터와 다르게 블러링의 개념이 섞인 1차 미분필터이다.

```
double temp[KERNEL33] = { -1,0,1,-2,0,2,-1,0,1 };

for (int idx = 0; idx < KERNEL33; idx++)
    sobel[idx] = temp[idx];

tryNum++;

return sobel;
```

```
double temp[KERNEL33] = { 1,2,1,0,0,0,-1,-2,-1 };

for (int idx = 0; idx < KERNEL33; idx++)
    sobel[idx] = temp[idx];

return sobel;
```



프리윗 필터와는 다르게 텍스쳐의 노이즈 부분이 확연히 줄어든 모습을 보인다.

이는 소벨필터가 블러링의 개념이 섞였기 때문이다.

<b>-1</b>	<b>0</b>	<b>1</b>
-2	0	2
-1	0	1

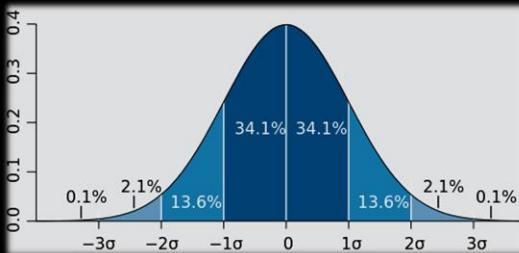
// x축(열) 끼리의 차를 계산

<b>-1</b>	<b>-2</b>	<b>-1</b>
0	0	0
1	2	1

// y축(행) 끼리의 차를 계산

그렇다면, 소벨필터가 왜 블러링 개념이 섞인 것인가?

일단, 현재 소벨필터는 각 축으로의 차를 계산한다.  
1차원 정규분포를 생각해보면,



평균으로부터의 얼만큼 떨어져있는지를 표준편차를 기준으로  
확률밀도함수에 가중치를 두었다.

이와 마찬가지로, 소벨필터는 대상픽셀과 가장 가까운 픽셀에  
가중치를 부여하여, outer product로 계산하여 완성되었다.

$$\begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} \times [-1 \ 0 \ 1] = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

// 정규분포를 인위적으로 균사화 한 수직 필터를  
1차분 필터와 행렬곱하여 완성된 모습

이러한 원리로, 소벨 필터는 제한적으로 블러링 효과도 겸비한다.

## 엣지검출에서 필요한 전처리 과정

엣지검출은 말 그대로, 엣지만을 검출하여야 한다.  
그러나, 모든 영상이 정직하게 엣지만을 포함하지 않는다.

카메라는 필연적으로 실제 빛을 이미지 센서로 순간적으로  
캡쳐하여  
센서가 기록한 수치를 데이터(=이미지)로 남기기 때문에, 만약  
센서에 문제가 생겨서, 아까 언급했던 Salt & Pepper 같은  
돌발적인 임펄스값 (=Noise)도 포함할 수도 있다.

이는 엣지검출의 목적에 부합하지 않기에, 아주 미세한 low-pass 필터를 거친 후, 엣지검출을 하여야 한다.

그래서, 약소한 블러링 처리를 거친 후, 엣지검출을 하면 더욱  
효과적으로 목적에 더욱 가까운 결과물을 얻어낼 수 있다.

주의 : 코드에서는 소벨, LoG, DoG 필터를 제외한 나머지  
필터들은 이러한 전처리 과정을 포함하지 않는다.  
전처리 과정의 유무 차이를 두고 결과를 관찰하기 위한 목적으로  
작성되었다.

## 2차 라플라시안 필터

2차 라플라시안 필터는 샤프닝 필터와 비교하여, 합이 0이  
되게하는 2차 미분 필터이다.

```
double* Lp_kernel = (double*)malloc(sizeof(double) * KERNEL33);
double temp[KERNEL33] = { -1,-1,-1,-1,8,-1,-1,-1,-1 };
for (int idx = 0; idx < KERNEL33; idx++)
    Lp_kernel[idx] = temp[idx];
return Lp_kernel;
```

-1	-1	-1
-1	8	-1
-1	-1	-1

// 라플라시안 필터



// 프리윗 필터



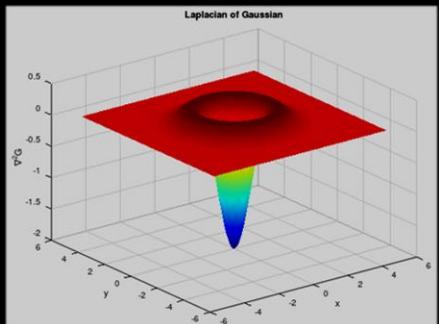
1차 미분 필터와 비교 하여 그레이디션 같은 디테일이 없어진 것을 확인 할  
수 있다.

## LoG 필터

LoG 필터는 엣지 디텍팅에서 전처리 과정으로 요구되는 low-pass 필터 통과 과정과 high-pass 필터 통과 과정을 결합하였다.

가우시안 필터를 2차 미분하면, 이 두 가지를 충족시킬 수 있다.  
가장 먼저, 가우시안 식을 2차 미분하면 아래와 같은 식을 얻을 수 있다.

$$\nabla^2 G(x, y) = \left( \frac{x^2 + y^2 - 2\sigma^2}{\sigma^4} \right) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

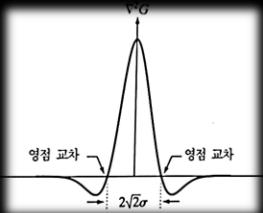


위의 수식을 matlab으로 구현하였다. 현재 3D로 나타내어서, 2차 미분의 특징인 영점교차점(=변곡점)이 잘 눈에 띄지가 않는다.

그 아래 그래프는 뒤집은 후, x축과 크기로만 나타낸 수식이다.

2차 미분의 특징인 영점교차점이 잘 보인다.

이러한 특징으로 LoG 필터는 엣지 검출 필터로써 사용될 수 있다.



```
printf("This is LoG kernel elements\n");
for (Y = -Radius; Y <= Radius; Y++)
{
    for (X = -Radius; X <= Radius; X++)
    {
        idx = (Y + Radius) * Sizeside + (X + Radius);

        Front = (X * X + Y * Y - 2 * sigma_pow) / (sigma_pow * sigma_pow);
        Back = exp(-(X * X + Y * Y) / (2 * sigma_pow));
        LoG_kernel[idx] = Constant * Front * Back;
        sum += LoG_kernel[idx];
    }
}
```

// LoG 커널을 만드는 함수의 일부분

표준편차값은 사용자에게 입력받는다.

주의 : 표준편차값을 너무 높게 설정하면, 과도한 블러링으로, 엣지가 점점 희미해진다.  
엣지를 선명히 하기 위해, 추가적인 처리가 필요하게 될 수도 있다



이전 슬라이드의 2차 라플라시안 필터의 결과 이미지와 비교해 보면, 야채안의 노이즈가 급격히 줄어든 모습을 확인 할 수 있다.

// σ = 0.8, 커널크기 7\*7

## DoG 필터

DoG 필터는 LoG 필터의 계산 복잡성을 줄이기 위해 고안되었다.

어떻게 만드는 것인가?

DoG 필터는 두 개의 다른 표준 편차를 가진 가우시안 필터끼리 빼서 근사적인 LoG 필터를 생성한다.  
같다고 볼 수는 없지만, 이 과정으로 DoG와 LoG가 같은 영점교차점을 갖게 만들 수는 있다.

$$DoG(x, y) = \frac{1}{2\pi\sigma_1^2} e^{-\frac{x^2+y^2}{2\sigma_1^2}} - \frac{1}{2\pi\sigma_2^2} e^{-\frac{x^2+y^2}{2\sigma_2^2}}$$

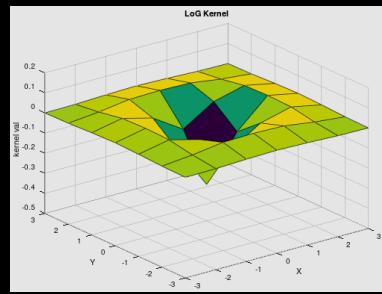
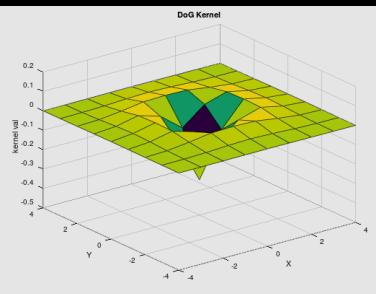
원하는  $\sigma$ 를 갖는 LoG 필터를 근사하기 위해서, 뺄셈에 이용되는 두 가우시안 필터는  $\sigma_1, \sigma_2$ 를 얻어야 한다.

그렇다면, 원하는  $\sigma$ 의 LoG를 근사하기 위해  $\sigma_1, \sigma_2$ 는 어떻게 얻는가?

$$\sigma^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 - \sigma_2^2} \ln \left[ \frac{\sigma_1^2}{\sigma_2^2} \right] \quad (\sigma_1 > \sigma_2)$$

그러나, 연구결과를 통하여, 두 표준편차값을 따로 구할 필요 없이 1.6:1의 비율로서, LoG와 DoG 함수들이 최고로 근접하게 근사화가 가능하다.

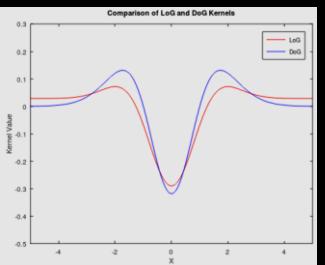
이를 풀어쓰면,  $\sigma_2 = \frac{\sigma}{\sqrt{1.641 \times \ln(2.56)}}$  으로 나타낼 수 있다.



// 두 그래프 모두 동일한 조건을 갖는다 ( $\sigma = 1$ , 커널크기 81)

위의 두 그래프는 DoG와 LoG를 matlab으로 구현한 그래프이다.  
살짝, 모양이 다르긴 하지만 매우 유사한 모습을 하고 있다.

그렇다면, 1차원 그래프를 통해 비교해보자. 두 그래프를 비교해보자.



```
Two sigma values in DoG: sigma1 (1.288240), sigma2 (0.805150)
LoG zero crossings at: -1.266   1.265
DoG zero crossings at: -1      0.999
>> |
```

완벽히 똑같다고는 볼 수 없지만, 그래프의 모양도 비슷해졌고, 두 변곡점의 값도 어느정도 비슷하다고 볼 수 있을 것 같다.

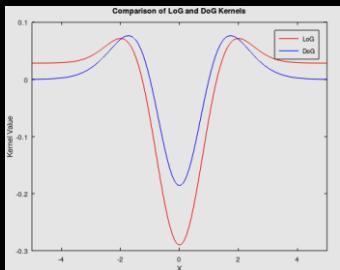
가장 중요한 건, 수식 자체로 계산복잡도를 크게 줄였다는 것이다.

// x축의 범위는 (-7:0.01:7) 이다.  
LoGo = 1로 만들었고, 그에 따른  
DoGo 두개를 구하였다.

## DoG의 세부 프로세싱

여기서 중요한 사실이 하나 있는데, 원하는 특정 LoG 커널을 근사하기 위해, DoG 커널을 만들 때에는 여러 가지의 추가적인 과정들을 거쳐야 한다.

1. 커널 배열의 중간값을 기준으로 절대값이 가장 큰 배열요소 (입력 축의 중심)를 기준으로 스케일링을 해줘야 한다



입력축의 중심을 기준으로 스케일링을 빼먹었을 시의 그라프이다.

이렇게 되면, 보는 것과 같이 근사율은 더욱 낮아질 수 밖에 없다.

2. 또한 이러한 과정을 거쳤기에, 총합이 0이 되지 않을 수 있는데, 총합이 0이 되게끔 또 다시 스케일링을 해줘야 한다

이는 엣지 검출 처리의 기본이기에 반드시 빼먹으면 안된다.

```
double sigma_pow = sigma * sigma;
double Constant = 1.0/(2 * M_PI * sigma_pow);
X = 0; Y = 0;
double Front = (X * X + Y * Y - 2 * sigma_pow) / (sigma_pow * sigma_pow);
double Back = exp(-(X * X + Y * Y) / (2 * sigma_pow));

double center_LoG = Constant * Front * Back;

double scale_factor = center_LoG / DoG_kernel[Radius * Sizeside + Radius];
```

// 반드시 LoG의 중점을 기준으로 스케일링이 계산되어야 한다.

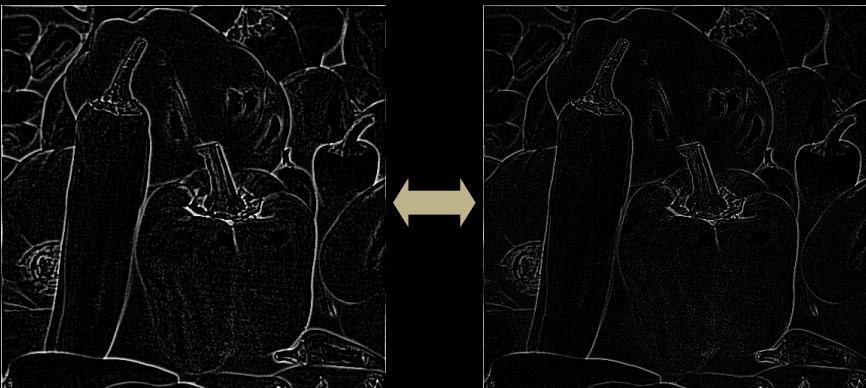
```
double GAU1, GAU2;
int Y, X;
int idx;
double sum = 0;
double element;
printf("This is DoG kernel elements\n");
printf("But it can NOT work as a LoG substitute perfectly\n");

for (Y = -Radius; Y <= Radius; Y++)
{
    for (X = -Radius; X <= Radius; X++)
    {
        idx = (Y + Radius) * Sizeside + (X + Radius);
        GAU1 = exp(-(X * X + Y * Y) / (2 * sigma1)) / Constant1;
        GAU2 = exp(-(X * X + Y * Y) / (2 * sigma2)) / Constant2;
        element = GAU1 - GAU2;
        DoG_kernel[idx] = element;
    }
}
```

이 코드는 DoG 커널 자체를 계산하는 코드이다.

왼쪽에서 언급했던 스케일링 단계와 커널 총합 0 처리는 공간이 부족해서 담지 못하였다.

아래는 DoG 커널과 LoG 커널을 비교한 모습이다.



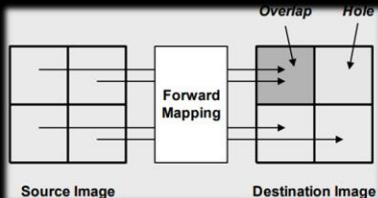
// 왼쪽이 DoG 커널의 결과, 오른쪽이 LoG 커널의 결과  
LoGo = 0.8이고, DoG의 두 σ는 LoG로 근사하기 위해 추가적으로 계산되었다.

### 3. 기하학 처리

이 처리는 픽셀들의 배열을 기하학적인 변환으로 변경하여 새로운 BMP파일을 만드는 것을 목적으로 한다.

#### 순방향 매핑

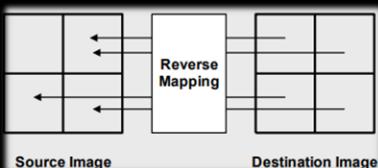
원본 영상의 각 픽셀을 변환된 이미지의 새로운 위치로 직접 매핑하는 방법.



원본영상을 변환하여 직접 목적영상에 매핑(변환)하니,  
오버랩으로 인한 훈이 다수 발생할 수 있다.

#### 역방향 매핑

변환된 이미지의 각 픽셀이 원본이미지의 어떤 위치에서 왔는지를 찾아내는 방법.



역방향 매핑에서는 오버랩이 잘 발생하지 않는다. 왜냐하면 변환된 이미지의 각 픽셀이 하나의 원본 이미지 픽셀 위치로 매핑되기 때문이다.

그러나, 변환된 후 정수좌표로 매핑될 때, 정수형 변환처리같은 연산으로 변환영상이 제대로 매핑되지 않아 훈이 발생할 수 있다!

#### 오버랩 현상

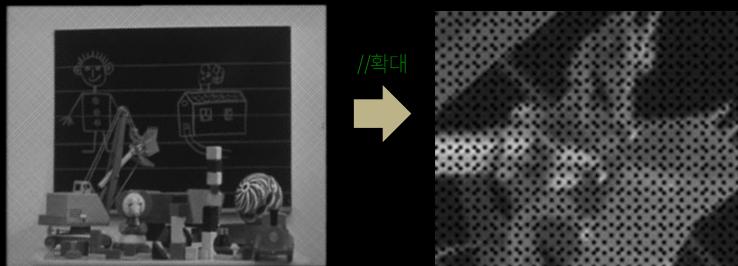
여러 원본 픽셀이 동일한 변환된 위치에 매핑되는 경우, 겹치는 현상을 의미한다. 이는 변환된 좌표가 소수점차이로 가깝게 매치될 때, 반올림 같은 처리로 발생할 수 있다.

오버랩 현상이 발생하면, 겹쳐진 픽셀값이 순실되어 훈이 발생할 수 있다.

#### 훈 현상

변환된 이미지에서 일부 픽셀 위치가 비어있는 현상을 훈이라고 한다. 역시나 반올림, 버림 같은 정수형 변환처리로 올바르게 매핑받지 못해서 빈공간이 발생할 수 있다.

훈 현상이 발생하면, 이미지에서 검은색 점이 보인다.



이러한 훈 현상을 메꾸기 위해서, 보간이란 방법을 사용하여 훈을 수학적인 알고리즘을 통하여 메꾼다.  
총 두개의 보간법을 구현하였다.

- 쌍선형 보간법
- 최근접위치 보간법

## 축소 처리(Minimizing.c)

원본이미지를 원하는 만큼의 크기로 축소시키는 과정이다.

축소처리는 해당 순서로 이루어진다

- 얼마의 크기로 축소할것인지를 입력받는다(검사필요)
- 해당 크기에 맞춰서 RAW파일의 가로세로 비율을 계산한다
- 크기가 달라졌기에 파일헤더를 올바르게 수정한다
- 해당 계산값에 맞춰서 RAW를 저장할 버퍼 크기를 재할당해준다
- 새 RAW의 좌표값을 기준으로 기존 RAW 와 역매핑하여, 채워준다.

### 크기스펙 검사

BMP파일의 영상의 너비는 4의 배수이어야 한다. 이는 BMP파일을 읽어서 화면에 출력할 때, 속도를 최적화하기 위해서이다.

그렇기에, 4의 배수가 아닐경우, 0으로 패딩해서, 4의 배수를 맞춰준다.

```
if (*newWidth < infoheader->width && *newWidth > 0)
{
    remain = (*newWidth) % 4;

    if (remain != 0) // 이미지 읽기 최적화를 위해서 너비는 언제나 4의 배수일것!!!
    {
        if (*newWidth < (infoheader->width) / 2) // 0에 가까울 확률 높음
        {
            remain = 4 - remain;
            *newWidth += remain;
        }
        else // 이전 너비 크기에 가까울 확률이 높다
            *newWidth -= remain;
    }

    width_check = 0;
```

// 입력받은  
크기스펙이  
올바른지를  
검사하는  
check\_size\_4m  
함수의 일부분

새 크기스펙과 원본 크기스펙을 각각 너비, 높이 비율로 계산한다(생략)  
해당 계산값에 맞춰서 new\_buffer(새 BMP의 RAW부분)의 크기를  
재할당 한다.

```
BYTE* new_addr = realloc(*new_buffer, sizeof(char) * newSize);

if (new_addr == NULL)
{
    printf("realloc Error Occured! \n");
    *errCode = 2;
    return 0;
}

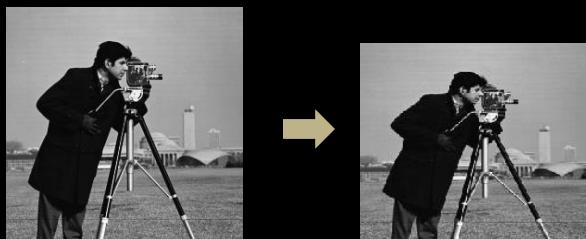
*new_buffer = new_addr; // 힙 영역이어서 로컬함수 스평프에서 처리해도 상관이 없다.
```

계산되었던 비율을 이용해, 역매핑과정을 수행한다.

```
int tempX; int tempY;

for (int y = 0; y < newHeight; y++)
{
    tempY = (int)floor(y * height_Ratio);
    // ratio = old/new 이기때, 여기에 new를 곱해주면, old의 인덱스가 나온다.

    for (int x = 0; x < newWidth; x++)
    {
        tempX = (int)floor(x * width_Ratio);
        (*new_buffer)[y * newWidth + x] = oldBuffer[tempY * oldWidth + tempX];
        // 제일 먼저 포인터 먼저 들어가 준다음, 배열 문법을 사용하자!!!
    }
}
```



// 원본은 256\*256  
변환을 200\*180으로  
맞췄더니  
영상이 약간 찌그리진것을  
확인할 수가 있다.

## 확대 처리(Magnifying.c)

원본 이미지를 원하는 만큼의 크기로 확대시키는 과정이다.

확대처리는 해당 순서로 이루어진다

- 얼마의 크기로 확대할 것인지를 입력받는다 (검사필요)
- 해당 크기에 맞춰서 RAW파일의 가로세로 비율을 계산한다
- 크기가 달라졌기에 파일헤더를 올바르게 수정한다
- 해당 계산값에 맞춰서 RAW를 저장할 버퍼 크기를 재할당해준다
- 새 RAW의 좌표값을 기준으로 기존 RAW와 역매핑하여, 채워준다.
- 훌을 찾고, 쌍선형 보간법으로 메꿔준다.

### 쌍선형 보간법

주어진 좌표가 원래 영상의 정수좌표 사이에 위치할 때, 해당 밝기값을 매핑할 수 없으니, 근처 좌표 정보들을 이용해 밝기값을 계산한다. 좌표계산을 두번 (x축끼리 한번, y축끼리 한번) 진행한다.

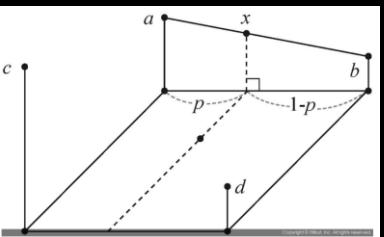
이때, 실수형 픽셀좌표가 네개의 픽셀 중 어디쪽에 위치한지에 따라서 비례식을 이용한 가중치를 주기 때문에, 더욱 부드러운 보간을 수행할 수 있다.

선형보간을 두번 실행하지만, 완전히 하나의 식으로 묶어줄 수 있다.  
다음은 쌍선형 보간법의 원리를 설명한다.

만약, 계산한 실수형 좌표가  $Z(\alpha, \beta)$ 라고 가정하면, 주변의 픽셀좌표는 다음과 같이 나타낼 수 있을 것이다.

$$\begin{array}{ll} A(\text{floor}(\alpha), \text{floor}(\beta)), & B(\text{floor}(\alpha+1), \text{floor}(\beta)), \\ C(\text{floor}(\alpha), \text{floor}(\beta+1)), & D(\text{floor}(\alpha+1), \text{floor}(\beta+1)) \end{array}$$

$x$ 의 계산은  $[p, p-1]$ 과  $[x, a, b]$  계산식]이 똑같다는 가정하에 A, B의 X축 좌표와 Z의 X좌표와의 비례식으로 진행된다. [그림의  $x(\alpha, \text{floor}(\beta))$ ]

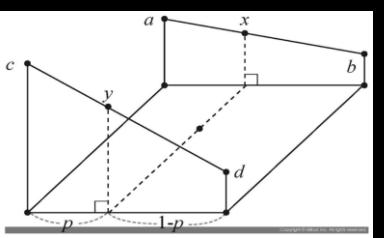


$$\frac{p}{1-p} = \frac{|x - A|}{|x - B|}$$

위 식은 다음과으로 전개가 가능하다.

$$x = pA + (1-p)B$$

A, B 뿐만이 아니라 C, D에서 이 식을 작성하고, 둘의 X좌표끼리의 계산을 진행하면 다음과 같이 진행된다. [그림의  $y(\alpha, \text{floor}(\beta+1))$ ]



$$\frac{p}{1-p} = \frac{|y - C|}{|y - D|}$$

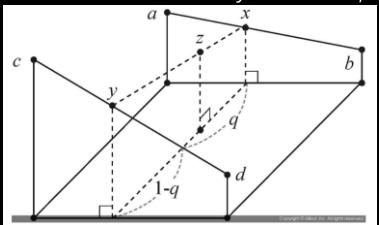
위 식은 다음과으로 전개가 가능하다.

$$y = pC + (1-p)D$$

//  $x, y, A, B, C, D$ 는 밝기값을 의미한다.

이제  $x$ 와  $y$ 를 구현했으니, 본격적인 실수형 좌표의 밝기값을 구할 차례이다.

$z$ 도 마찬가지로  $x$ 와  $y$ 를 이용해, 비례식을 이용하여 계산한다.



$$z = qx + (1 - q)y$$

이렇게 따로따로 계산해도 되지만,  $x$ 와  $y$ 를 대입해서 한꺼번에 하나의 식으로 구할 수도 있다.  
코드에서는 하나의 식으로 구현하였다.

다음은  $x$ 와  $y$ 를 대입한 식이다.

$$z = q [ pA + (1 - p)B ] + (1 - q)[ pC + (1 - p)D ] \quad (3.1)$$

이로써, 쌍선형보간법의 식이 완성되었다.

이 보간법의 핵심은 구할수 없는 좌표의 밝기값을 거리의 비례식을 이용해, 밝기값을 계산하는 것이다.

다음은 (3.1)의 식을 코드로 구현한 모습이다.

```
line BYTE bilinear_interpolation(double upLeft, double upRight, double downLeft, double downRight, double xDiff, double yDiff)
{
    return (BYTE)floor((1 - yDiff) * (downLeft * xDiff + (1 - xDiff) * downRight) + yDiff * (upLeft * xDiff + (1 - xDiff) * upRight));
}
```

// 파라미터 설명

upLeft = A, upRight = B, downLeft = C, downRight = D를 의미한다.  
또한,  $xDiff = 1-p$ 를 의미하고  $yDiff = 1-q$ 를 의미한다.

// 아무리 양호한 보간법이어도 너무 키우면 디테일이 굉장히 떨어진다 →

```
double xRatio = (double)oldWidth / (double)newWidth;
double yRatio = (double)oldHeight / (double)newHeight; // 새 이미지 좌표를 원본에 투영시키기 위한 확대비

int intX, intY;
double realX, realY;
double upLeft, upRight, downLeft, downRight; // 원본좌표에 적용될 좌표 매핑용 변수들
BYTE result;
int newX, newY;
double xDiff, yDiff;
int nextX, nextY;

// 양선형 보간법 시작

for (newY = 0; newY < newHeight; newY++)
{
    realY = ((double)newY) * yRatio;
    intY = (int)floor(realY);
    yDiff = realY - intY;

    nextY = (intY + 1 > oldHeight - 1) ? intY : intY + 1; // 원본이미지의 높이범위를 벗어나지 않게 처리

    for (newX = 0; newX < newWidth; newX++)
    {
        realX = ((double)newX) * xRatio;
        intX = (int)floor(realX);
        xDiff = realX - intX;

        nextX = (intX + 1 > oldWidth - 1) ? intX : intX + 1; // 원본이미지의 너비범위를 벗어나지 않게 처리

        upLeft = (double)oldBuffer[intY * oldWidth + intX];
        upRight = (double)oldBuffer[intY * oldWidth + nextX];
        downLeft = (double)oldBuffer[nextY * oldWidth + intX];
        downRight = (double)oldBuffer[nextY * oldWidth + nextX];
        // 사용 확대된 이미지의 좌표를 원본 이미지 좌표에 매핑시킨것이 intY*yDiff, intX*xDiff 이다.

        result = bilinear_interpolation(upLeft, upRight, downLeft, downRight, xDiff, yDiff);
        // 기울치라는 개발보다는 원본이 투영된 걸 좌표와 근접 좌표를끼리의 비율계산 (양선형 보간법!)
        (new_buffer)[newY * newWidth + newX] = result;
    }
}
```



// 쌍선형  
보간법을  
구현한  
코드이다.

중간에  
(3.1)식을  
구현한 함수를  
호출한다

// 10배로  
키운 모습이다.

100%크기 그  
자체이다.  
네모를  
주목하라

## 회전 처리(Rotating.c)

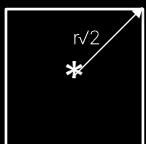
원본 이미지를 원하는 만큼의 각도로 회전시키는 과정이다.

회전처리는 해당 순서로 이루어진다

- 이미지의 전체 크기의  $\sqrt{2}^2$  배 만큼의 NULL 버퍼를 생성한다
- 기존 이미지를 NULL 버퍼에 복사해 가운데로 평행이동 시킨다
- 이전 처리들처럼 헤더와 버퍼 크기를 수정한다.
- 얼마의 각도로 회전할 것인지를 입력 받는다 (라디안 변환 필요)
- 회전 선형변환식을 이용해 정매핑하여 채워준다 (오버랩 발생!)
- 이미지의 홀을 찾아, 최근접 보간법으로 메꿔준다

왜 이미지의 2배 만큼 키워주는 것인가?

이미지가 회전하면, 필연적으로 기존의 크기로는 회전한 이미지 전체를 담아낼 수 없다.



그렇기에, 중심으로서부터의 가장 먼 거리인 45도 각도를 그 크기로 하여 어느 각도여도 이미지가 전부 잘리지 않게 크기를 키웠다.

라디안 변환식

프로그래밍 언어의 삼각함수에선 각도가 아닌 라디안 값만 받는다.

$$rad = \frac{ang}{180}\pi$$

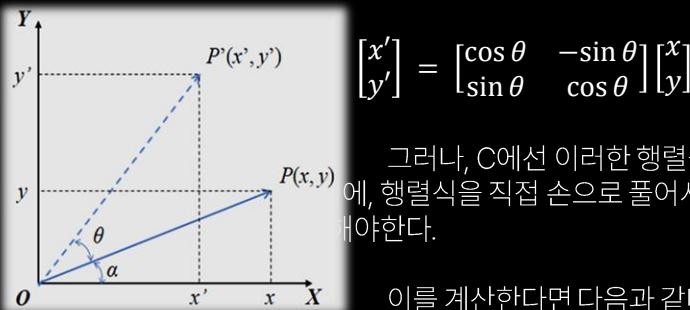


```
#define ANG2RAD(angle) ((M_PI * (double)(angle)) / 180.0)
```

## 회전 선형변환식

회전변환은 선형대수학에서 행렬계산으로 정의할 수 있는 계산법이다.

아래와 같은 그림으로 설명할 수 있고, 아래와 같은 식으로 정의가 가능하다



그러나, C에선 이러한 행렬을 계산할 수  
에 행렬식을 직접 손으로 풀어서  
해야 한다.

이를 계산한다면 다음과 같다.

$$\begin{aligned} x' &= x \cos \theta - y \sin \theta \\ y' &= x \sin \theta + y \cos \theta \end{aligned}$$

### 주의

그러나, 그냥 사용하기엔 심각한 문제가 있다. RAW 버퍼를 좌표 공간으로 생각한다면, 원점은 좌상단에 위치하기 때문에, 그냥 식을 사용한다면, 좌상단을 기점으로 이미지가 돌아가게 된다.

이를 방지하기 위해, 이미지의 중점을 구해, 회전변환식을 계산하기 전에 미리 빼주고, 계산을 한 후 더해주어야 한다.

그럼 식은 다음과 같이 나온다.

$$\begin{aligned} x' &= [(x - mid_x) \cos \theta - (y - mid_y) \sin \theta] + mid_x \\ y' &= [(x - mid_x) \sin \theta + (y - mid_y) \cos \theta] + mid_y \end{aligned}$$

다음은 위 회전변환을 구현한 함수이다.

```
int rotate_RAWdata(BYTE* temp_buffer, BYTE* new_buffer, BITMAPINFOHEADER* infoheader, double radian)
{
    int Width = infoheader->width;
    int Height = infoheader->height;
    int tempX, tempY;

    double realX, realY;
    int amidWidth = Width / 2; int amidHeight = amidWidth; // 정사각형이기에

    double cosVal = cos(radian); double sinVal = sin(radian);
    // 이미지 좌표공간은 기존 수학의 좌표공간과 y축 대칭이지만, RAW 파일형식이 거꾸로 되어있고
    // 그 RAW 이미지를 화면에 불러올때의 규칙때문에 기존 선형변환 공식을 써도 상관이 없다

    int newX, newY; // 이미지의 중심을 기점으로 회전한다는 것을 잊지말자

    for (tempY = 0; tempY < Height; tempY++)
    {
        for (tempX = 0; tempX < Width; tempX++)
        {
            realX = (tempX - amidWidth) * cosVal - (tempY - amidHeight) * sinVal;
            realY = (tempX - amidWidth) * sinVal + (tempY - amidHeight) * cosVal;

            newX = (int)round(realX) + amidWidth;
            newY = (int)round(realY) + amidHeight;

            if (newX >= 0 && newX < Width && newY >= 0 && newY < Height) // 유효한 좌표인가?
                new_buffer[newY * Width + newX] = temp_buffer[tempY * Width + tempX];
        }
    }

    return 0;
}
```

현재, 위의 일부분을 보면 의아해 할 수 있다.

```
int amidWidth = Width / 2; int amidHeight = amidWidth; // 정사각형이기에
```

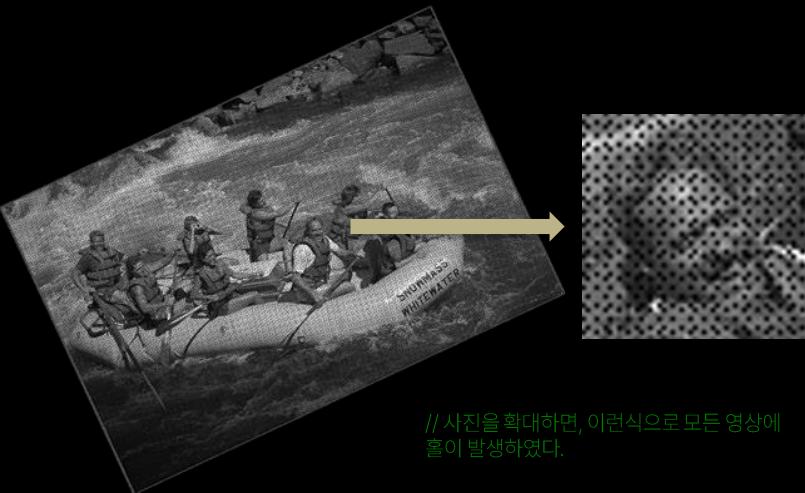
모든 파일은 정사각형의 스펙을 가지지 않기 때문이다. 그렇기 때문에, 직사각형 스펙의 사진들은 특정각도에서 필연적으로 잘려나간다.

이를 방지하기 위해, 애초에 변들을  $\sqrt{2}$ 배 해주고, 그중에서 가장 큰 변만으로 NULL 버퍼를 동적할당 하였다.

## 최근접 보간법

이러한 회전변환은 정방향 매핑으로 수행되었기에, 필연적으로 오버랩이 발생하고, 이는 홀을 발생시킨다.

다음은 보간을 수행하지 않은 반시계 방향으로 25도만큼 회전된 영상이다. 자세히 보면, 오버랩으로 인한 홀이 영상에 전방위적으로 발생하였다.



이러한 홀을 메우기 위해서 최근접 보간법을 필요에 맞게 개조해서 사용하였다.

최근접 보간법은 홀이 발생한 좌표에서 근처의 가장 가까운 좌표에 있는 픽셀의 밝기값을 복사해온다.

현재 코드에선 회전변환된 이미지에서 훌(=NULL값)을 찾으면 보간법을 사용하게끔 만들었다.

```
// Hole이 발생하는 이유는 행렬계산을 한 후, 반올림을 하여 경수처리를 하기 때문이다.  
// 그렇기 때문에 논리적으로 x방향 y방향 연속적으로 hole이 발생 할 수 없다고 판단했다.
```

```
for (Y = 0; Y < infoheader->height; Y++)  
    for (X = 0; X < infoheader->width; X++)  
        if (buffer[Y * infoheader->width + X] == 0)  
            buffer[Y * infoheader->width + X] = Mynearest_interpolate(buffer, X, Y, infoheader->width, infoheader->height);
```

`// RAW 파일에서 NULL을 찾는  
finding_holes 함수의 일부분`

그러나, 최근접보간법으로 하면, 계산이 복잡해지고, 또다시 매핑할 버퍼를 만들어야 하기에, 이는 너무 복잡하다고 판단하여, 반복문이 +방향으로 가는것을 이용하여, 오른쪽과 아래쪽의 픽셀들을 참고하여, 두 부분의 밝기값으로 훌을 보간하는 방법을 사용하였다.

```
int referX = curX + 1; int referY = curY + 1;  
  
// 1차원 배열이기에, +방향으로 읽게된다. 그런데 여기서, -방향의 픽셀을 참조하면, 이미지가 끊어진다 => 아직 건드리지 않은 +방향  
  
if (referX < Width && (buffer[curY * Width + referX] != 0))  
    return buffer[curY * Width + referX];  
  
// 해당픽셀 (y,x+1) 이 최대 너비를 벗어나면, 현재 픽셀을 오른쪽 경계로 가정  
  
if (referY < Height && (buffer[referY * Width + curX] != 0))  
    return buffer[referY * Width + curX];  
  
// 참조픽셀값이 훌일때 || 참조 픽셀이 경계범위 밖일때, 공백으로 판정  
return 0;
```

`// Mynearest_Interpolate 함수  
내용`

그런데 아까, NULL버퍼를 선언했었다. 훌도 NULL값일텐데, 이 공백과 훌을 구분해야 한다.

이를 구분하기 위해, 일단, RAW데이터에서 밝기값이 0인 픽셀을 찾으면, 근처의 픽셀들을 확인하고, 두 부분의 픽셀또한 훌일 경우에 공백으로 판정하게 구현하였다.

이러한 일련의 과정을 거쳐서 이미지의 회전을 구현하였다.

아래는 구현의 결과이다.



```
oldWidth(768) * root(2) = newWidth(1088)  
oldHeight(512) * root(2) = newHeight(726)  
Consider non-square cases and grow the buffer into larger sides  
newSide : 1088, newSize : 1183744
```

`// 이미지의 스펙에  $\sqrt{2}$ 를 곱해서 가장 큰  
변으로 정사각형 버퍼를 만들었다는 부분 (1번)`

```
This is temp_buffer specification  
marginWidth : 320, marginHeight : 576, moveWidth : 160, moveHeight : 288
```

`// 해당 버퍼의 정 가운데로 평행이동을  
했다는 부분 (2번)`

```
The center location of Image : (X : 544, Y : 544)  
The cosVal : 0.707107, sinVal : 0.707107
```

`// 회전변환을 하는 부분(5번)`



해당부분을 확대처리하였을때,  
쌍선형 보간법에 비해 화질이  
많이 떨어지는 모습을 확인할 수  
있다.

## 4. 통계학 처리

이 처리는 이미지를 데이터로 분석하고 변환하여 더 나은 결과를 도출하기 위한 처리이다.  
특히 명암 대비(콘트라스트)를 다루는 것에 중점을 둔다.

왜 이미지의 명암 콘트라스트에 초점을 맞추는가?

명암 대비는 이미지의 선명도와 디테일 인식을 크게 향상시킨다. 높은 명암 대비는 밝은 부분과 어두운 부분의 차이를 극대화하여 시각적으로 더 명확하고 선명한 이미지를 제공한다.

예를 들어서, 어두운 곳에 올블랙으로 차려입은 사람이 있다면 정말 알아차리기 힘들 것이다. 그 이유는 우리가 물체를 구별할 때, 비슷한 색끼리 뭉쳐져 있는 것을 무의식적으로 하나의 물체로 인식하기 때문이다.

만약, 이러한 명암대비를 표로 나타낸다면, 인식의 정도를 다른 관점으로도 볼 수 있을 뿐만 아니라, 수학적인 도구로써, 인식의 정도를 높일 수도 있다.

### 밝기값 히스토그램

밝기 히스토그램은 이미지의 밝기 분포를 시각적으로 표현한 그래프이다.

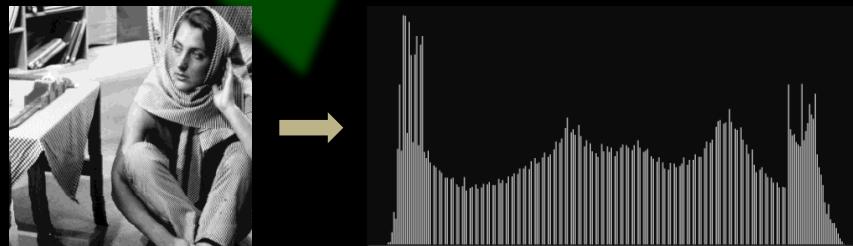
x축은 밝기 수준을, y축은 해당 밝기 수준의 픽셀 수를 나타낸다. 이를 통해 이미지의 노출, 대비, 밝기 등을 분석할 수 있으며, 이를 기반으로 이미지 보정 및 최적화를 수행할 수 있다

// 이미지에서 히스토그램을 추출하는 함수  
배열의 인덱스 번호 자체가 밝기값을 나타내고, 배열의 인덱스가 그 밝기값의 빈도수를 나타낸다.

```
void init_ARR(BYTE* buffer, BITMAPINFOHEADER* infoheader, double* temp_arr, STATISTICS* data)
{
    // 밝기값에 따라 빈도수 설정하기
    data->mean = 0; data->pow_sum = 0; data->sum = 0; data->variance = 0;
    // 통계를 위한 구조체 전부 초기화

    for (unsigned int idx = 0; idx < infoheader->ImageSize; idx++)
    {
        for (int brit = 0; brit <= MAX_BRIT_VAL; brit++)
        {
            if (buffer[idx] == brit)
            {
                temp_arr[brit]++;
                data->sum += brit; // 모든 픽셀들의 밝기값의 합
                data->pow_sum += pow(brit, 2);
                break; // 퍽셀의 밝기값을 찾았으니, 뛰어넘어버리자
            }
        }
    }

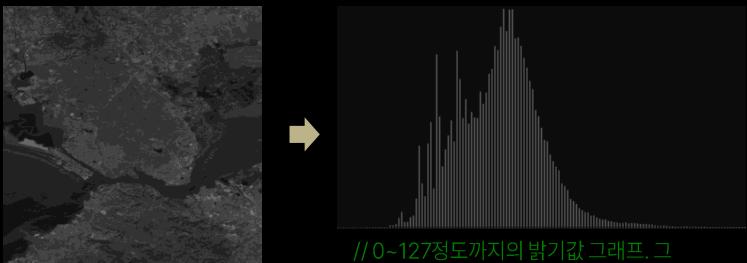
    data->mean = data->sum / infoheader->ImageSize;
    data->variance = (data->pow_sum / infoheader->ImageSize) - pow(data->mean, 2);
    // 히스토그램의 평균과 분산을 구해서 보기 위함
}
```



// 위의 함수로 이미지를 처리하였을 때, 나오는 히스토그램.  
영상의 BitPerPixel = 8이고, 팔레트정보를 흑백으로 맞췄기에,  
총 0~255까지의 밝기값 단계를 가진다.

## 히스토그램 평활화 처리(Equalizing.c)

아래 영상에서 안의 물체를 구별하기 힘들다. 옆은 아래 영상을 히스토그램화 시킨 표이다.



영상의 분포도가 한쪽으로 치우쳐져 있는 것을 확인 할 수 있다.

만약, 이러한 영상의 빈도수를 자동으로 재 분배해서, 영상의 명암 콘트라스트를 향상시킬 수 있다면, 매우 유용할 것이다.



히스토그램을 이용해, 수학적인 기법으로 아래 영상처럼 영상의 대비를 고르게 분포시키는 작업을 히스토그램 평활화라고 한다.

대략적인 순서를 나누자면

- 이미지를 히스토그램화 하기
- 빈도수를 전체이미지로 정규화하여 확률질량함수 생성
- 이들을 누적분포함수로 만든 다음, 최대밝기값으로 스케일링
- 그 값들을 반올림하여, 역매핑하여 각 픽셀에 재분포하기

이러한 순차적인 방법으로 이루어진다.

이론 설명에 앞서...

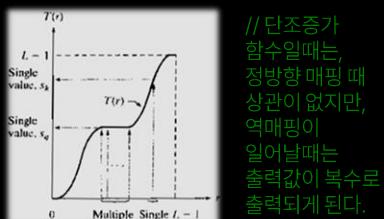
- 히스토그램화 + 정규화가 완료되었다고 가정한다
- 이론적인 배경을 설명할 때에는 히스토그램처럼 이산적인 환경이 아닌 연속적인 환경(히스토그램)으로 가정한다.
- 또한, 연속적인 환경이기에, 이론에서는 확률밀도함수를 가정한다
- 변환전의 밝기값은  $r$ , 변환함수는  $T(r)$ , 변환후의 밝기값은  $s$ , 이미지의 최대밝기값은  $L$ , 랜덤변수가  $r$  주변에서 가질 확률밀도함수는  $Pr(r)$ 로 둔다

## 히스토그램 평활화의 원리

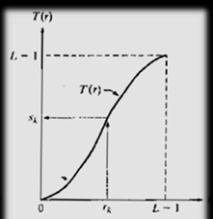
$r$ 의 범위는  $[0 \sim L-1]$  까지이고,  $s = T(r)$ 의 범위 역시  $[0 \sim L-1]$ 이다.

아직 변환함수  $T$ 를 알지 못하지만, 역매핑이 이뤄진다는 점에서  $T$ 는 **엄밀 단조 증가 함수여야 한다.** (기울기가 항상 양수일것)

이는 각  $r$ 이 고유하기 때문이고,  $r$ 이 갖는 빈도수가 흐트러지면 안되기 때문이다.



// 단조증가  
함수일때는,  
정방향 매핑  
상관이 없지만,  
역매핑이  
일어날때는  
출력값이 복수로  
출력되게 된다.



// 엄밀 단조증가  
함수일때,  
1대1대응이  
되기에,  
역매핑에서도  
1대1대응이 된다.

그리고, 정규화가 끝났으니 밝기값  $r$ 과  $s$ 를 랜덤변수로써 바라볼 수 있다.  
 $Pr(r)$ 과 변환함수  $T(r)$ 이 알려져 있고,  $T(r)$ 이 연속적이고 미분 가능하다면,  $s$ 의 PDF는 간단한 공식을 사용하여 얻을 수 있다.

$$P_s(s) = P_r(r) \left| \frac{dr}{ds} \right| \quad (4.1)$$

이는,  $T$ 가  $r \rightarrow s$ 로의 랜덤변수의 변환을 의미하기에, 둘의 확률은 같아야 하기 때문이다.

## $|dr/ds|$ 는 무엇인가?

이는  $s = T(r)$ 을 편미분하여 얻어진다.  $\rightarrow ds = T'(r)dr$

그렇기에,  $s$ 의 PDF를 얻는 공식을 다시 바꿔보면, 아래를 확인할 수 있다.

$$T'(r)^{-1} = \left| \frac{dr}{ds} \right|$$

이를 기억하고, 변환함수를 이러한 형태로 알고 있다고 가정해보자

$$T(r) = (L-1) \int_0^r Pr(w)dw \quad (4.2)$$

이 형태는  $0 \sim r$ 까지의 구간에서  $r$ 이 나올 확률(누적분포함수화)을 최대밝기값으로 스케일링한 개념이다.

(4.2) 식을 바탕으로  $Ps(s)$ 를 구하기 위해, 이걸 다시 미분한다.

$$\frac{ds}{dr} = \frac{dT(r)}{dr} = (L-1) P_r(r) \quad (4.3)$$

이 식을 (4.1)의 식에 대입하면, 결과가 나온다.

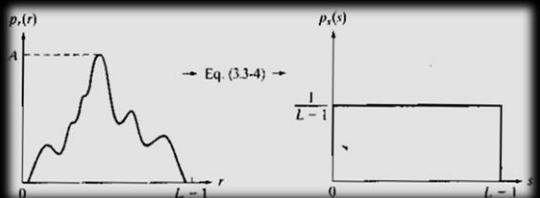
$$P_s(s) = P_r(r) \left| \frac{1}{(L-1)P_r(r)} \right| = \frac{1}{L-1}$$

이 결과로  $Pr(r)$ 의 분포와 상관 없이  $Ps(s)$ 는 균일 PDF로 나오게 된다는 것을 알 수 있다.

## 히스토그램 평활화의 원리2

앞의 슬라이드에서  $Pr(r)$ 의 분포와 상관 없이  $Ps(s)$ 는 균일 PDF로 나오게 된다는 사실을 알았다.

아래는 위의 과정을 그래프로 표시한 것이다.



역매핑 과정

변환함수(CDF+스케일링)를 거쳤지만, 원본 히스토그램과 비교하면, 그 인덱스 번호 순서는 유지된다.(변환함수가 1대1 대응이었기 때문)

즉, 이미지의 각 밝기값(=인덱스 번호)이 변환된 배열에서 동일한 순서를 지닌 밝기값  $s$ (=인덱스)로 매핑되었다.

기존의 이미지에서 픽셀들의 밝기 값  $r$ 이 갖던 빈도수와 순서에 맞게, 새로운 배열 인덱스가 가진 밝기 값  $s$ 로 매핑하는 과정을 거치면 히스토그램 평활화 처리 과정이 종료된다.

그렇다면, 실제 이산적인 환경 (= 히스토그램)상에서는?

위의 설명이 사실은 확률밀도함수를 가정하였기에, 현실적으로는 사용하는 히스토그램과 거리가 있다.

히스토그램 평활화에선 cdf로 만들고 스케일링의 과정을 거친후, 반올림 연산을 거친다.

이는 밝기값  $s$  자체가 정수값을 갖기 때문이다.

bin	0	1	2	3	4	5	6	7
$h(g)$	4	3	2	1	0	2	3	1
$p(g)$	4/16	3/16	2/16	1/16	0	2/16	3/16	1/16
$cdf(g)$	4/16	7/16	9/16	10/16	10/16	12/16	15/16	1

2	2	2	4
3	3	4	2
3	5	7	5
4	7	7	7

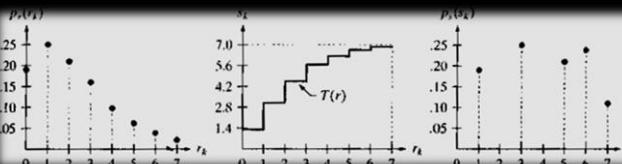
  

$cdf(g) * L_{max}$	1.75	3.06	3.94	4.38	4.38	5.25	6.56	7
2	3	4	4	4	4	5	7	7

// 전체적인 히스토그램 평활화의 순서를 그림으로 나타내었다.

// 출처:  
<https://velog.io/@redorangeyellow/ch03-%EA%B8%80%EB%B3%BB%EC%A0%81%EC%9D%8B-%EC%98%81%EC%83%81-%EC%B2%98%EB%A6%AC-%EA%B8%80%EB%B2%95-%ED%9E%88%EC%8A%A4%ED%86%AO%EA%B7%BB%EB%9E%A8-%ED%8F%89%ED%99%9C%ED%99>

위의 사진에서처럼 반올림처리를 거침으로써,  $s$ 값이 같은 밝기값을 나타낼 수도 있다. 그렇기에, 일반적으로 연속적인 환경에서처럼의 이상적인 직사각형모양의 분포도가 나오지 않을 수도 있다.



// (왼쪽) 정규화된 히스토그램, (중간) CDF된 히스토그램, (오른쪽) 평활화된 히스토그램

## 코드 분석 & 결과

- 히스토그램 평활화처리의 순서 중
- 변환함수 (CDF화 + 스케일링)
  - 반올림처리
  - 역매핑과정

위의 세단계를 구현한 코드이다.

```
normalize_CDF(old_buffer, infoheader, TEMP_ARR, &bmp_Data);
for (unsigned int idx = 0; idx < infoheader->ImageSize; idx++)
    new_buffer[idx] = (BYTE)TEMP_ARR[old_buffer[idx]];

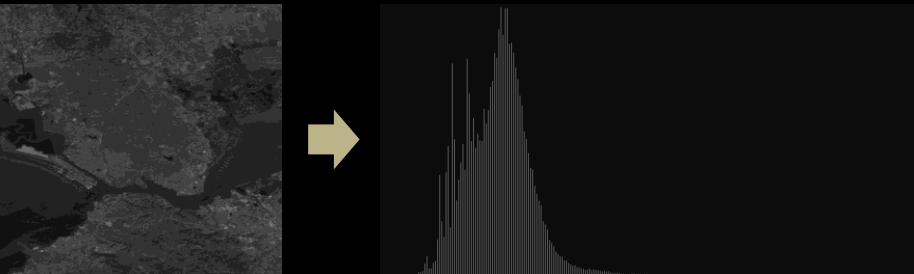
// 반복문은 역매핑 과정을 구현부분
// 기존 이미지의 밝기값을 변환된 배열의 인덱스에 넣어서 새로운
// 밝기값으로 매핑하고, 이를 새로 작성할 버퍼에 써주는 부분

double scale_factor = (double)MAX_BRIT_VAL / (double)infoheader->ImageSize;
int temp;
double sum = 0;

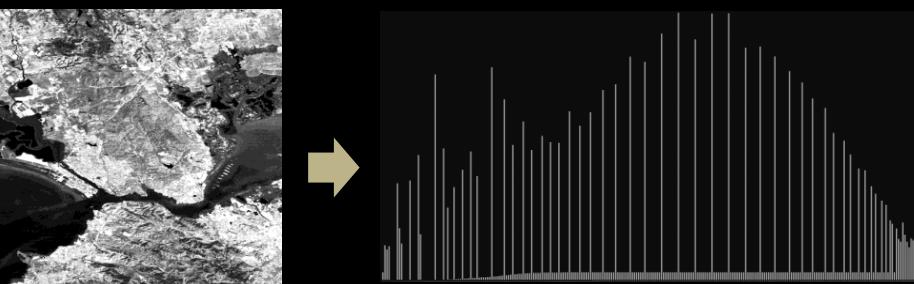
for (int brit = 0; brit <= MAX_BRIT_VAL; brit++)
{
    sum += temp_arr[brit];
    temp = (int)(round(sum * scale_factor)); // 정규화 후, 자동반올림
    temp_arr[brit] = temp; // 다시 기존의 히스토그램 빈도수로 넣어놓기
}

// 변환함수와 반올림 처리를 하는 normalize_CDF 함수의 일부분
```

이러한 처리를 거치면, 원본 이미지와 비교해 처리된 이미지의 히스토그램의 분포와 그에따른 분산이 눈에 띄게 올라감을 확인할 수 있다.



처리되기 전의 평균은 어두운쪽으로 쓸려있고, 그에 맞춰 분산도 작아 분포도가 서로 밀집되어 있는것을 확인이 가능하다.



처리되기 후의 평균은 가운데로 보정되었고 그에 맞춰 분산도 커져서, 이미지의 세부적인 디테일을 쉽게 확인할 수 있다.

## 히스토그램 스트레칭 처리(Stretching.c)

히스토그램 스트레칭 처리는 원래 이미지의 밝기 값 분포가 좁은 범위에 집중되어 있을 때, 이를 전체 밝기 값 범위에 걸쳐 확장하여 이미지의 대비를 향상시킬 수 있다.

### 단계 설명

- 이미지의 최소 밝기값 ( $m$ )과 최대 밝기값 ( $M$ )을 찾아 밝기값 스코프를 계산
- 찾은 스코프를  $[0 \sim L-1]$ 까지 확장시키는 변환함수 통과

아래 두 코드는 첫번째 단계를 구현한 코드이다.

```
// 최소와 최대를 구한 후, 스코프까지 계산하는 코드이다.

BYTE denominator;
BYTE factor[2] = {UCHAR_MAX, 0};

STATICS bmp_Data;
double temp_arr[MAX_BRIT_VAL+1] = {0};
find_min_max(old_buffer, infoheader, factor);
// 0번째 인덱스는 최소값, 1번째 인덱스는 최소값

denominator = factor[1] - factor[0]; // 분모 생성

if (denominator==0) // 0으로 나눌 수도 있기에
{
    printf("This is not picture!\n All pixel has same brit_level!!\n");
}

void find_min_max(BYTE* old_buffer, BITMAPINFOHEADER* infoheader, BYTE* factor)
{
    // 0번은 최소, 1번은 최대

    for (unsigned int idx = 0; idx < infoheader->ImageSize; idx++)
    {
        if (factor[0] > old_buffer[idx])
            factor[0] = old_buffer[idx];

        if (factor[1] < old_buffer[idx])
            factor[1] = old_buffer[idx];
    }
}
```

### 변환함수

예를 들어서, 이미지가 갖는 최소 밝기값이 20 최대 밝기값이 150라면, 130의 스코프를 다시 L로 넓혀줘야 한다.

$$T(r) = \frac{(r - r_{min})}{(r_{min} - r_{max})}(L - 1)$$

이러한 식으로써, 이미지 분포를  $[0 \sim L-1]$ 로 확장할 수 있다  
아래의 코드는 그러한 과정을 구현하였다.

```
double scale_factor = (double)MAX_BRIT_VAL / denominator;
for (unsigned int idx = 0; idx < infoheader->ImageSize; idx++)
{
    int new_val = (int)round((old_buffer[idx] - factor[0]) * scale_factor);
    new_buffer[idx] = clipping(new_val);
}
```

아래는 변환된 모습이다

mean : 54.87712, variance : 214.94128



// 영상의 디테일과 전체적인 밝기가 약간 증가하였다.

mean : 56.52169, variance : 480.47843



## 이진화 처리(Binarizing.c)

이진화 처리는 이미지를 흑백만 가지는 이미지로 변환하는 과정이다. 이는 주로 이미지에서 물체와 배경을 분리하거나, 텍스트 인식, 객체 검출 등 다양한 분야에서 사용된다.

이진화는 밝기 값의 임계값을 기준으로 픽셀 값을 흰색 또는 검은색으로 변환하는 작업이다.

여기서 더 나아가 자동으로 임계값을 설정하는 Otsu 알고리즘을 구현하였다.

### Otsu 알고리즘의 오버뷰

이 알고리즘은 이미지의 히스토그램을 분석하여 두 클래스 간의 분산이 최소가 되는 임계값을 찾게 한다.

전체 이미지의 밝기 값을 두 클래스로 나누어 분산이 최소화되도록 임계값을 결정하고, 그 임계값에 따라서 이진화 처리를 실시한다.

순서를 나누자면 다음과 같이 나눌 수 있다.

- 히스토그램 생성
- 최적화를 위한 최소 밝기값과 최대 밝기값을 찾아서 스코프 계산
- 해당 스코프에서 임계값을 대조해가며, 두 클래스의 분산의 최소값 탐색
- 해당 임계값으로 영상을 이진화

### 클래스 내 분산 ( $V_c$ )

스코프가 [min Max]로 나누어져 있다면, 이진화이기에, 해당 스코프에서 두 그룹으로 나누어야만 할 것이다.

그럼, 첫번째 대조에서 Threshold = th를 기준으로, class1 = [min~th-1], class2 = [th ~ Max]로 나누게 될것이다.

이때 각 클래스 내의 분산이 작을수록, 해당 클래스는 클래스 평균 근처에 빽빽하게 뭉쳐있을 것이다. 그럼 전체적인 분산으로 보았을 때, 그 분산은 커질것이고, 이는 이미지에서 객체를 더욱 쉽게 구분할 수 있게 한다.

이때, 두 클래스의 분산의 크기를 각각 짤 수 없으니 그냥 합쳐서 그 값만 생각하자고 제안하는게 Otsu 씨가 생각해 낸 알고리즘이다.

수식을 보자면 다음과 같이 표시된다.

$$\sigma_B^2(th)_{min} = \sigma_{G1}^2 + \sigma_{G2}^2$$

혹은, 다음처럼도 정의 될 수 있지만, 코드구현은 위의 수식대로 구현했다.

$$\sigma_B^2(th)_{min} = P_1(m_1 - m_G)^2 + P_2(m_2 - m_G)^2$$

// 개념적으로 표현한것이 아닌, 계산량을 줄이기 위해 만들었다.

## 코드 구현

일단, 히스토그램화는 끝내고, 유효 스코프 범위도 찾았다고 가정한다.  
그리고, 해당 스코프만큼의 double 형으로, 두 클래스 분산값을 저장할 배열을 선언한다.

```
int min = find_valid_Val('m', histogram);
int max = find_valid_Val('M', histogram);
printf("Min valid Max Value : %d, max, min)

int threshold=0; // 오즈 알고리즘에서 가장 핵심적인 역할
int Diff = max - min; // 둘의 차이가 얼마나인가?

double* Variances = (double*)malloc(sizeof(double) * Diff);
// 분위값에 의한 두 클래스내의 분산을 저장할 용도
```

그리고, 문턱값을 유효 스코프 범위내에서 움직여 가며, 두 클래스 내의 분산값을 찾아서 선언한 배열에 저장해준다.

```
double var_G1,var_G2;
int temp_Th;
int Width = infoheader->width;
int Height = infoheader->height;

for (temp_Th = 0; temp_Th < Diff; temp_Th++)
{
    var_G1 = within_class_Var(histogram, 0, min + temp_Th - 1, Width, Height, &data_G1);
    var_G2 = within_class_Var(histogram, min + temp_Th, max, Width, Height, &data_G2);

    Variances[temp_Th] = var_G1 + var_G2;

double within_class_Var(double* histogram, int min, int max, int Width, int Height, STATICS* classData)
{
    int brit;

    for (brit = min; brit <= max; brit++)
    {
        classData->sum += histogram[brit]; // 통계값의 sum은 클래스 전체를 의미한다 -> 나누기 요소
        classData->mean += brit * histogram[brit];
        classData->pow_sum += pow(brit, 2) * histogram[brit];
    }

    if (classData->sum == 0)
        return 0; // 0나누기 피하기

    classData->mean /= classData->sum;
    classData->variance = (classData->pow_sum / classData->sum) - pow(classData->mean, 2);

    return classData->variance;
```

// 임계값으로 나누어진 클래스의  
분산을 찾는 함수

```
for ( ; idx < Diff; idx++)
    if (Var_min > Variances[idx])
        Var_min = Variances[idx];

// 이것과 매칭되는 threshold값 찾기

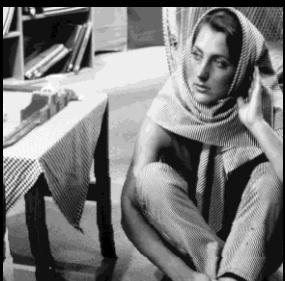
for (idx = 0; idx < Diff; idx++)
{
    if (Var_min == Variances[idx])
    {
        threshold = idx; // 인덱스에 Min을 더하여 실제 임계값을 찾음
        break;
    }
}

threshold += min;
```

이제, 해당 배열에서 최소값을 가진 임계값을 찾아준다.

마지막으로 최소값을 갖는 클래스 내의 분산으로 이진화를 적용하면 다음과 같은 결과를 얻게 된다.

```
int Otsu_final(BYTE* old_buffer, BYTE* new_buffer, BYTE threshold, BITMAPINFOHEADER* infoheader)
{
    int X, Y;
    int Width = infoheader->width; int Height = infoheader->height;
    for (Y = 0; Y < Height; Y++)
        for (X = 0; X < Width; X++)
            (old_buffer[Y * Width + X] > threshold) ? (new_buffer[Y * Width + X] = (BYTE)MAX_BRIT_VAL) :
            (new_buffer[Y * Width + X] = (BYTE)0);
```



// Vc와 그 임계값 -> The Minimum VAR : 2537.090387, The threshold : 124



# 문제해결 과정

해당 챕터를 소개하기에 앞서....

이 과정은 해당 프로젝트를 진행하면서 가장 어려웠던 문제들과 그 원인, 해결과정 등을 정리하였습니다.

총 세개의 디버깅 과정을 담았습니다.

- 커널의 분리 이슈
- 밝기값 감소 이슈
- 포인터 추적 이슈

각 과정마다 세가지의 주제를 설명합니다

- 증상
- 원인
- 해결

이전의 스크린샷이 없는 부분도 있습니다

## 1. 커널의 분리 이슈

### \* 증상

블러링 처리에서, 커스텀 가우시안 커널을 만들 때, 특정 커널 크기를 초과하면, 발생했던 문제입니다.

가우시안 커널은 그 자체로 모든 축에 대칭이고 분리 가능한 커널이기에, 커스텀으로 만들어 두어도, 분리가 가능하다고 생각하였습니다.

기존까지는 테스트용으로  $3 \times 3$  크기의 커널만을 만들어서 이슈가 발생하지 않아, 완성되었다고 판단하였지만,  $5 \times 5$  크기 이상의 커널부터, 어떠한 표준편차 값에도 이러한 비슷한 사진만을 출력하였습니다.



게다가, 커널의 크기가 더욱 커질수록, 이진화 영상처럼 거의 육안으로 구별했을 때, 거의 흑백으로만 보이는 처리 결과도 있었습니다

그러나 최근접 보간법을 이용(MS그림판)해, 확대를 해보면 완벽한 이진화는 아니었습니다

## \* 원인

첫번째로 시도했던 것은, 분리가능한 커널로 처리되는 각 구문에서 열벡터와 행벡터 값들을 확인하는 것이었습니다.

```
0.000028 0.000239 0.001073 0.001769 0.001073 0.000239 0.000028
0.000239 0.002917 0.013071 0.021551 0.013071 0.002917 0.000239
0.001073 0.013071 0.058582 0.096585 0.058582 0.013071 0.001073
0.001769 0.021551 0.096585 0.159241 0.096585 0.021551 0.001769
0.001073 0.013071 0.058582 0.096585 0.058582 0.013071 0.001073
0.000239 0.002917 0.013071 0.021551 0.013071 0.002917 0.000239
0.000028 0.000239 0.001073 0.001769 0.001073 0.000239 0.000028
```

```
The sum of the kernels is NOT 1!
The kernel is symmetric about both x and y axes
As a result, your kernel is a detachable kernel!
When calculating kernels, the time cost is reduced by 33! Compared to:
```

```
Row Vector of kernel!
0.000028 0.000239 0.001073 0.001769 0.001073 0.000239 0.000028
Now row vec calculation is start!
```

```
Column Vector of kernel!
1.000000 12.182494 54.598150 90.017131 54.598150 12.182494 1.000000
Now col vec calculation is start!
```

현재  $7 \times 7$  열벡터의 요소들의 값이 굉장히 높은 것을 확인할 수 있습니다.

단지 부동소수점의 문제라고 판단하여, 실제값과 얼마나 다른지를 확인하기 위해 실제 파이썬의 Decimal모듈을 이용해, 코드를 다시 짜보았습니다.

그러나, C로 했을 때와 비교해, 치명적으로 문제를 일으킬 만큼 큰 차이를 보이지 않았습니다.

그러나, 문제는 이 부분에 있었습니다.

```
for (Y = -Radius; Y <= Radius; Y++)
{
    for (X = -Radius; X <= Radius; X++)
    {
        int idx = (Y + Radius) * side + (X + Radius);
        GAU_kernel[idx] = exp(-(X * X + Y * Y) / (2 * sigma * sigma)) / (2 * M_PI * sigma * sigma);
        sum += GAU_kernel[idx];
    }
}

for (idx = 0; idx < size; idx++)
    GAU_kernel[idx] /= sum;
```

// gen\_GAU\_kernel 함수의 일부부

본래의 취지는 각 커널의 요소를 무작정 sum으로 나눠서 총합이 1이 되게끔 만드는 것이었지만, 이러한 정규화 과정을 거쳐도 합이 1이 되지 않는다는 것이었습니다.

어림잡아 커널크기에 따라서 총합이 약 0.97~1.2 정도의 총합을 가겠습니다. 이를 해결하기 위해 정규화의 과정을 몇번 더 수행하게 하니, 그 오차는 점점 더 불어나서 1.45까지 커지는 것을 확인하였습니다.

잘못된 총합을 가지는 잘못된 커널을 사용하여, 분리가능한 커널로 연산을 하니, 이진화 현상에 점점 가까워졌었던 것이고, 밝기의 표현능력이 점점 줄어드는 것입니다.

또한, 이러한 커널의 요소로 아래와 같은 정규화를 진행하고, 열벡터로의 컨볼루션을 진행하였기에, 커널을 키울수록, 정규화를 진행할 수록 더욱 이상해졌던 거였습니다.

```
double N = kernel[0];
for (idx = 0; idx < size * side; idx++) // 커널의 행벡터(x축)
{
    vector_buffer[idx] = kernel[idx * size * side] / N;
    // 위의 분리가능한 커널을 조개서 다시 붙이면, N값이 제곱이 되어서 나오기 때문에 정규화
    printf("%f ", vector_buffer[idx]);
}
```

## \* 해결

기존의 블러링 처리 과정을 전면 개편하기로 하였습니다.

첫번째 방법은 다른방법으로 1로 조정하는 방법을 찾는 것이었습니다.

기존과는 다르게, 총합으로만 나누지 않고, 현재 총합과 1과의 차이를 계산하여, 크기로 나눠주고 이를 각각에 더하는 것이었습니다.

```
double Diff = 1.0 - sum;
sum = 0.0;

if (!compareDouble(Diff, 0.))
{
    printf("kernel sum is NOT 1\n");
    for (idx = 0; idx < size; idx++)
    {
        GAU_Kernel[idx] += Diff / size;
        sum += GAU_Kernel[idx];
    }
}
```

눌론 이마저도 완벽하게 1을 맞출수는 없었습니다.

그러나, 이러한 과정을 함으로써, 기존과는 다르게 정말 1에 더욱 가깝게 총합을 구성할 수 있었습니다.

오른쪽의 사진은 코드를 수정한 후의 조정된 커널의 총합입니다.

그러나, 위와같은 방법으로 더하게되면, 더이상 분리가능한 커널이 아니라고 생각하여, 선형대수에서의 분리가능한 행렬을 조사하였습니다.

rank=1인 행렬로써, 분리가능한 커널을 확인할 수 있다는 것을 확인하였기에 두번째 방법으로, rank=1인지를 확인하는 코드를 구현하였습니다.

```
Not normalized sum of kernel : 0.999459
kernel sum is NOT 1
kernel sum = 1.000000
// 콘솔창 출력결과:
똑같은 7*7 크기, σ=1
```

// 이를 구현한 코드는 이전에 화소영역처리 기능설명에서 설명하였기에 적지 않았습니다.

또한, 기존의 문제점은 분리가능한 커널을 단지, 커널이 y축과 x축으로 대칭이면, 분리가능한 커널로 인지하게끔하였기에, 다른 한편으로, 새로운 검사함수를 기존 조건문에 추가로 적용하였습니다.

```
if (check_symmetry(kernel, kernel_size) && is_seperatable(kernel, size))
// K = Kx × Ky ^ (T) 의 형태로 조개지면 분리가능한 커널이다.
// 대칭이며, rank = 1 인 행렬이어야 한다. 이를 검사하기 위한 조건
{
```

기존의 check\_symmetry함수는 대칭인지를 검사하고,

is\_seperatable함수는 rank=1인지, 열벡터에서 정규화를시킬 요소(kernel[0])가 0인지를 검사합니다

이러한 세가지의 처리를 함으로써, 치명적인 이슈를 해결해 낼 수 있었습니다.

그러나, 커스텀 가우시안 커널로 블러링 처리를 할 시, 기존에 3×3 크기의 커널에서 작동하였던 분리 컨볼루션이 완전히 작동하지 않는 부작용을 가지게 되었습니다.

## 2. 밝기값 감소 이슈

### \* 증상

이전의 커널의 분리 이슈를 처리하고, 정석적인 컨볼루션을 진행하던 중 다음과 같은 사실을 발견하였습니다.

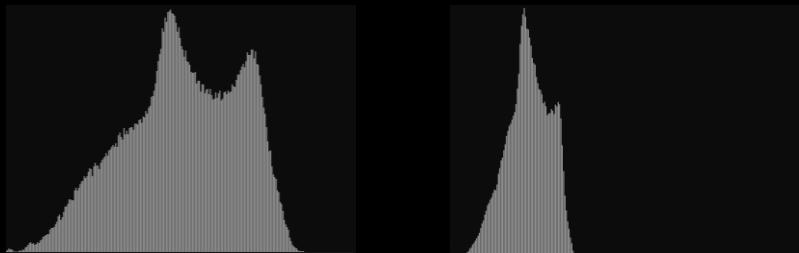


가우시안 블러링이 처리되었던 것을 확인 할 수 있었지만, 영상의 전체 밝기가 낮아진 것을 확인 할 수가 있었습니다.

그리고, 커널의 크기와 시그마 값에 따라서 영상의 전체적인 밝기가 달라지기도 하였습니다.

또한, 위 영상을 히스토그램화 시키고 통계값을 분석하여, 다음과 같은 사실을 확인할 수 있었습니다.

// 왼쪽: 원본 ↔ 오른쪽: 블러링 처리후



왼쪽이 원본이고 오른쪽은 블러링 처리 후의 통계값입니다.

mean : 129.116341, variance : 1790.304832

mean : 56.049770, variance : 252.390463

현재, 원본과 비교해, 전체평균이 73정도 깎이고, 전체분산이 굉장히 줄어든 것을 확인할 수 있습니다.

또한, 처리 후의 히스토그램이 원본의 히스토그램처럼 두개의 큰 뿔모양의 분포도를 유지하고는 있지만, 해당 디테일이 감소한 것을 확인 할 수 있습니다.

일단 처리된 비트맵 파일이 그래도 baboon이라는 모습은 유지하였기에, 코드에서 어딘가에 문법적인 작은 오류가 있겠다고 예상은 하고있었습니다

연속적인 디버깅으로 인한 피로와 블러링 처리를 위한 코드를 구현했을 때, 논리적으로 완벽하다는 자만심으로 인해, 정말 사소한 문제임에도 불구하고 이러한 문제를 해결하는데 꽤 많은 시간을 할애했습니다.

예상했던대로 정말 사소한 문제로 인해 이러한 오류는 일어났었습니다.

## \* 원인

해당 문제를 해결하기 위해 첫번째로 시도했던 커널값을 분석하는 것이었습니다.

일단, 총합 자체가 엄밀히 1이 아니라는 것은 인지하고 있었기 때문에, 총합을 1로 만드는 커널요소 조정하는 부분의 코드를 여러번 시도하게 만들어 보았지만, 결과는 달라지지 않았습니다.

두번째로 시도했던 방법은 기존의 커스텀 가우시안 커널을 만드는 루트 대신에, 혹시모를 평균 커널을 사용하여 진행하였습니다.

결과는 역시 이전과 똑같았습니다.

그래서, 커널의 문제가 아님을 이때 확인하고, 커널생성 이후를 기점으로 중단점을 찍고 한줄씩 실행해 보았습니다.

문제는 아래 부분에 존재하였습니다.

```
void flipping(double* kernel, int Sizeside)
{
    int Size = (int)pow(Sizeside, 2);
    int Mid = Size / 2 + 1;
    int idx;

    for (idx = 0; idx <= Mid; idx++) // y축과 x축에 대해 flip 연산
        swap(BYTE, kernel[idx], kernel[(Size - 1) - idx]);
}
```

만약 그렇지 않다면, 커널을 뒤집고, 정석적인 컨볼루션을 진행하는 부분에서, 커널을 뒤집는 부분의 함수인 flipping 함수에 작은 문법 오류였습니다.

그러나, 여기서 추가적인 의문점이 들었습니다.

이를 double 형이 아닌 unsigned char로 형변환을 하면, 전부 0이 나와서 아무것도 출력이 되지 않아야 하는데 일단 나오기는 하였기에, flipping이 된 직후의 커널값을 확인하였습니다.

0.000020	0.000239	0.001073	0.001769	0.001873	0.000239	0.000020
0.000239	0.002917	0.013071	0.021551	0.013071	0.002917	0.000239
0.001073	0.013071	0.058582	0.096585	0.058582	0.013071	0.001073
0.001769	0.021551	0.090000	0.090000	0.095685	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

커널의 중간이상의 인덱스 번호를 갖는 커널값들은 모두 예상대로 0이 나왔지만, 다른 부분은 값이 그대로 유지되었습니다.

```
#define swap(type,x,y) do{ type t=(x); (x)=(y); (y)=t }while(0)
#define ANG2RAD(angle) ((M_PI * (double)(angle)) / 180.0)
```

해답은 swap 매크로 함수에 있었습니다. 해당, type은 옮길값을 임시로 저장할 변수를 만드는데 사용되는 자료형이었는데, 그 부분만 BYTE로 처리하고 받았기에 버림연산이 되어 중간이상의 인덱스들이 0으로 처리가 되었던 것었습니다.

## \* 해결

```
void flipping(double* kernel, int Sizeside)
{
    int Size = (int)pow(Sizeside, 2);
    int Mid = Size / 2 + 1;
    int idx;

    for (idx = 0; idx <= Mid; idx++) // y축과 x축에 대해 flip 연산
        swap(double, kernel[idx], kernel[(Size - 1) - idx]);
```

별다른 처리 없이, 문제되는 부분을 double 형으로 바꿔주었더니, 정상적으로 동작하는 것을 확인하였습니다.

### 3. 포인터 주적 이슈

#### \* 증상

기존의 이미지 확대처리를 함에 있어서, 발생하였던 문제입니다.

```
clean_up:
    free(old_buffer);
    free(new_buffer);
close:
if (oldMP != NULL) fclose(oldMP);
if (newMP != NULL) fclose(newMP);

return errorCode; // 경상작동은 false를 반환
}

// 여기서 중단점 명령이 실행되었다고 자주 뜸.

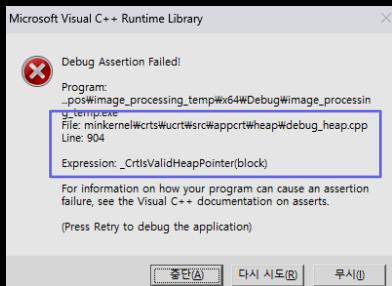
// 중단점 명령 _debugbreak() 문 또는 유사한 호출이
// Image_Processing.exe에서 실행되었습니다.

void print_info(BITMAPFILEHEADER* fileheader, B
{
    printf("file Header\n");
}
```

이 오류는 확대처리를 하고, 새 RAW 버퍼를 새로운 BMP파일에 작성하고, 동적할당을 해제까지 한 후에, 기존 caller로 리턴하는 중에 발생했던 오류입니다.

어떠한 내용도 적하지 않고, 중단점 명령이 실행되었다고만 뜹니다.  
이러한 경고문으로는 아무런 단서를 얻을 수 없었습니다.

그래서, 만들어진 .obj 파일을 실행해본 결과 다음과 같은 경고창이 떴습니다.

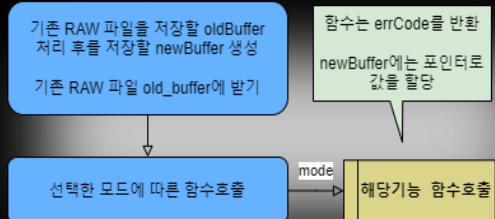


강조해 놓은 부분을 잘 보면, heap영역에서 오류가 발생하였다고 떠있는 것을 확인 할 수 있습니다.

heap 영역은 동적할당 시킨 메모리 공간이 위치하는 곳입니다  
이를 확인함으로써, 동적할당에 문제가 있음을 인지하였습니다

#### \* 원인

현재 대략적인 확대처리의 단계는 다음과 같습니다.



newBuffer(새 RAW 버퍼)는 원본 이미지가 가지던 크기만큼 할당됩니다. 그리고, 선택했던 모드에 따른 처리함수를 호출하는 함수를 호출합니다.

Magnifying 함수에서 원하는 크기를 입력받고 계산하여, realloc으로 newBuffer의 크기를 재 할당합니다.

마지막으로, 확대처리를 한 후, fwrite 함수로 newBuffer의 내용을 작성해주고, 동적할당받았던 모든 버퍼들을 할당해제 합니다.

일단은 heap 영역에 문제가 있다는 것을 확인하였기에, realloc 부분에서 문제가 있다고 생각하였습니다.

주변이들에게 물어보니 realloc 함수의 자체적인 문제라고도 들어서, realloc 함수의 자체적인 문제가 있는것인지 생각을 하였습니다

그러나, 굉장히 이상했던게, 그동안 이러한 부분의 코드스타일을 자주 써왔었다는 점이었습니다.

그래서, 첫번째로 시도해 봤던 것은 메인함수와 확대함수만으로 구성된 코드를 새롭게 만들어보았었습니다.

중간에 어떤 분들은 RAW버퍼 자체가 너무 커서 heap영역에서 문제가 생긴것이라고도 말하였기 때문이었습니다.

그러나, 새로 작성한 코드에서는 원하는 모습으로 확대된 BMP파일을 확인할 수 있었습니다. 애초에 이상한게, 그동안 새 RAW버퍼를 1MB 언저리에서 할당했었었습니다.

두번째로 시도했던 방법은 혹시나, `realloc`함수에서 OS가 기존의 `newBuffer`를 할당해제하고 새로운 영역에 재 할당하는 방법을 써서 그 부분에서 문제가 발생하지 않을까란 생각에 의심되는 주소값을 전부 출력해보았습니다.

```
BYTE* new_addr = realloc(*new_buffer, newSize);
if (new_addr == NULL)
{
    printf("realloc Error occurred!\n");
    return 0;
}
*new_buffer = new_addr;
```

1. newBuffer가 할당되자마자 주소값을 print (convertBMP)
2. realloc을 하고 반환된 new\_addr의 주소값을 print (magnifying)
3. \*newBuffer의 주소값을 print (magnifying)
4. 할당해제 하기전 newBuffer의 주소값을 print (convertBMP)

두번째 시도에서 실마리를 찾을 수 있었습니다.

원래라면, 2번, 3번과 4번의 주소가 똑같아야 하는데, 4번의 주소는 1번과 같았습니다.

이를 통해, OS가 기존의 `newBuffer`를 할당해제하고, 재할당 하였지만, `ConvertBMP`함수의 `newBuffer` 포인터변수까지 반영되지 않았다라는 사실을 알수 있었습니다.

또한, 새로작성한 코드와 문제가 되었던 코드의 차이점을 생각해보면, 문제가 되는 코드는 `Convert->mode_select->Magnifying` 식으로, 중간에 또 한번의 징검다리를 건너다는 차이점이 있었습니다.

다음은 이 문제점이 어디에서 잘못되었는지를 설명하는 코드부분입니다.

```
/*-----*/
// 할당된 메모리(newBuffer)의 시작위치를 넘겨줌
int result = mode_select(old_buffer, new_buffer, &infoheader, &fileheader.mode, &errCode);

int mode_select(char* old_buffer, char* new_buffer, BITMAPINFOHEADER* infoheader,
{
    switch (mode)
    {
        case 6:
            minimizing(old_buffer, &new_buffer, infoheader, fileheader, errCode);
            break;
        case 7:
            magnifying(old_buffer, &new_buffer, infoheader, fileheader, errCode);
            break;
    }
}

int magnifying(BYTE* old_buffer, BYTE** new_buffer, BITMAPINFOHEADER* infoheader,
{
    // 양선형 보간법
    BYTE* new_addr = realloc(*new_buffer, newSize);
    // mode_select의 지역변수에만 주소값이 반영됨 이를 호출했던 ConvertBMP의
    // 지역변수에는 반영 X
```

이전의 사진에서처럼, 애초에 mode\_select를 호출할때, 넘겨주었던 newBuffer의 파라미터부터가 잘못되었었습니다.

realloc으로 메모리를 재할당해도 mode\_select의 파라미터까지만 변경되지, convertBMP함수에서 newBuffer 포인터변수는 값을 그대로 갖고있었고,

이미 할당해제된 메모리 영역의 주소값을 갖고 fwrite를 하고, 그 주소값을 free함수로 할당해제를 하려고 하니,  
해당 과정에서 OS가 할당되지도 않은 부분에 마음대로 접근하고 할당해제를 하는 과정에서 문제가 있다고 판단,  
해당 프로세스를 kill해버리고, heap 영역의 버그가 있다고 출력하였던  
것이었습니다.

## \* 해결

realloc함수로 재할당된 변화가 convertBMP의 newBuffer 포인터 변수에 반영이 되면 이와 같은 문제는 해결이 되는 것이기에, 다음의 수정들이 필요했습니다

- mode\_select에 newBuffer 포인터 변수의 주소값을 넘기기
- mode\_select는 이를 이중포인터로 받기
- 나머지 처리함수의 파라미터에 동적할당된 버퍼의 주소값을 넘기기
- realloc 되는 부분은 newBuffer 포인터 변수의 주소값을 넘기기

다음과 같이 수정을 하니 예상대로 코드는 정상적으로 동작하게 되었습니다.

```
/*-----*/ //Caller의 포인터변수 주소를 넘겨줄것
int result = mode_select &old_buffer, &new_buffer, &infoheader, &fileheader, mode, &

int mode_select BYTE** old_buffer, BYTE** new_buffer BITMAPINFOHEADER* infoheader, E
{
    switch (mode) //Caller의 포인터변수 주소도 받아야 하기 때문에 이중포인터로 선언
    {
        case 1:
            duplicate(*old_buffer, *new_buffer, infoheader, errCode);
            blurring(*old_buffer, *new_buffer, infoheader, errCode);
            break; //동적할당된 메모리영역의 시작주소 넘기기
        case 2:
            sharpening(*old_buffer, *new_buffer, infoheader, errCode);
            break;
        case 3:
            mid_filtering(*old_buffer, *new_buffer, infoheader, errCode);
            break;
        case 4:
            edge_detecting(*old_buffer, *new_buffer, infoheader, errCode);
            break;
        case 5:
            rotating(*old_buffer, new_buffer, infoheader, fileheader, errCode);
            break;
        case 6:
            minimizing(*old_buffer, new_buffer, infoheader, fileheader, errCode);
            break;
        case 7:
            magnifying(*old_buffer, new_buffer, infoheader, fileheader, errCode); //newBuffer 포인터주소의 주소값을 넘기기
            break;
    }
}
```

```
int magnifying(BYTE* old_buffer, BYTE** new_buffer, BITMAPINFOHEADER* infoheader,
{ // 약선택 부가법
    BYTE* new_addr = realloc(*new_buffer, newSize);
    if (new_addr == NULL)
    { printf("realloc Error occurred!\n");
        return 0;
    }
    *new_buffer = new_addr;
}
```

//처리함수부분은 기존의 방식을 유지하여도 된다.

# 성취도 및 소감

이번 프로젝트는 멀티미디어 시스템 과목의 과제들을 통합하여 하나의 어플리케이션을 만드는 도전에서 시작되었습니다.

기존의 신호및시스템과 DSP는 이론으로도 굉장히 어려웠고, 그걸 실습하면서도 핵심이 무엇인지, 왜 그런 결과가 나오는지가 쉽게 감이 잡히질 않았는데, 영상신호처리에서는 컴퓨터로 바로바로 영상을 확인하면서 그 원리로 인한 결과를 직관적으로 확인하고 이해할 수 있었기 때문에 더더욱 이전 과목들보다 매력적이지 않나 싶습니다.

과제 이상의 기능들을 구현하며, 이론으로만 배우던 영역까지 도전했습니다. 특히, 디버깅과 함수 간의 통신 설계는 새로운 경험되었고, 여태껏 경험했던 적이 없었을만큼, 프로젝트의 규모가 커서 많은 어려움을 겪었습니다.

이 프로젝트를 해나가는 과정에서 가장 크게 배운 점은 "현재 내가 쓴 코드가 아니면 무조건 의심하라"는 것입니다. 예전에 썼던 코드나 남이 작성한 코드, ChatGPT가 추천해준 코드를 검증 없이 사용하면 디버깅에 정말 많은 시간이 걸렸습니다.

매일 조금씩 프로젝트를 완성해가면서 C 언어에 대한 이해도와 사고의 깊이가 달라졌고, 조금이나마 어려웠던 이론으로 자랑할만한 라이브러리를 만듬으로써 엔지니어의 꿈을 간접적으로나마 체험해 볼 수 있었습니다.

이 프로젝트는 전반적으로 저에게 성장의 기회를 제공한 소중한 경험이었습니다.  
이 과목을 강의하셨던 교수님, 그리고 도움을 주셨던 모든분들께 너무나도 감사합니다.

