Computer Vision and Deep Learning

Ch 03

강명묵

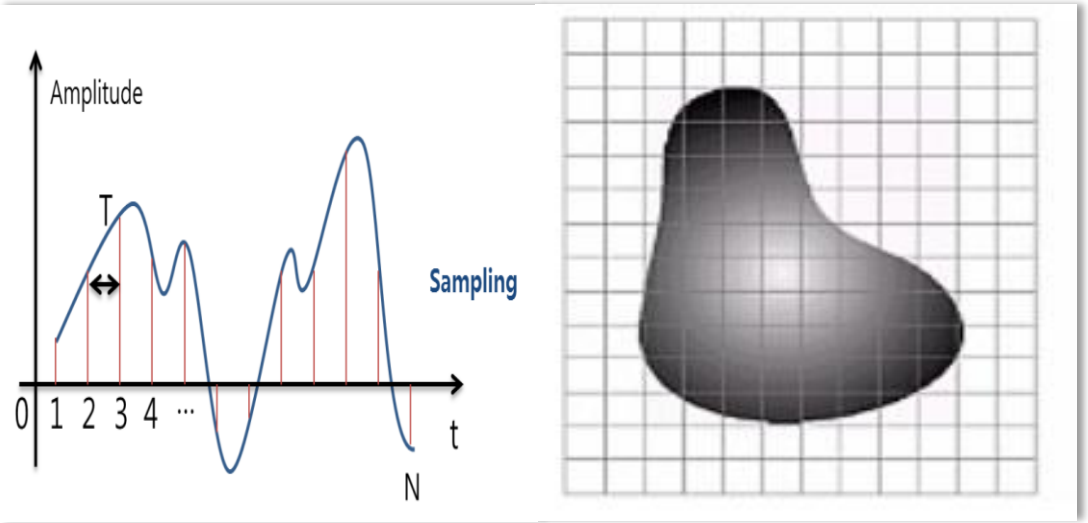
Contents

영상처리와 컴퓨터비전

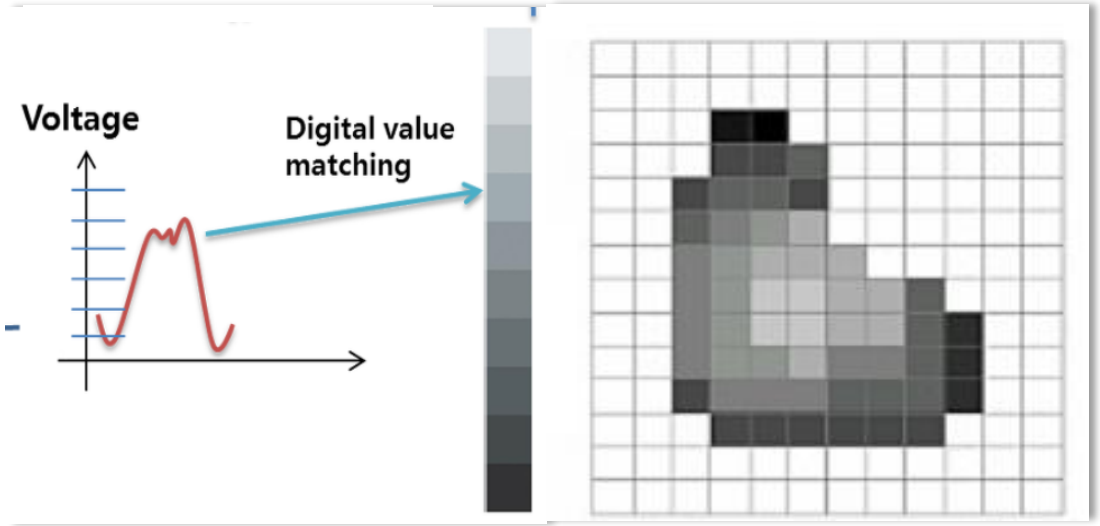
영상처리는 입력 영상을 처리하여 출력으로 처리된 영상을 얻는다.
컴퓨터비전은 기본적인 영상처리를 바탕으로 영상에서 특정한 정보를 추출하여 처리한다.

영상은 어떻게 획득하는가?

디지털 카메라는 실제 세상에 존재하는 피사체를 일정한간격으로 **Sampling**하고
명암을 일정한 간격으로 **Quantization**하는 과정을 통해 디지털 영상을 획득한다.



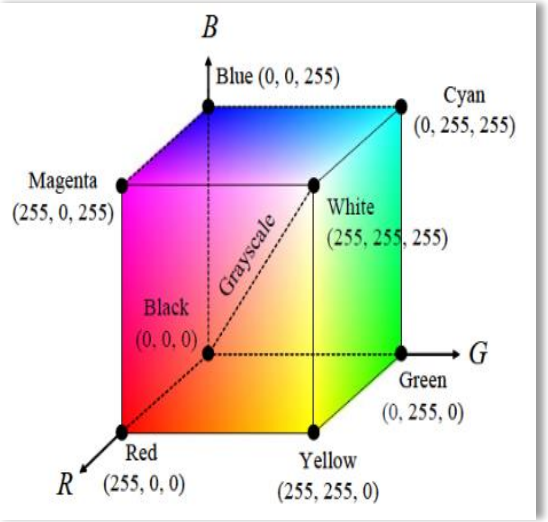
Sampling



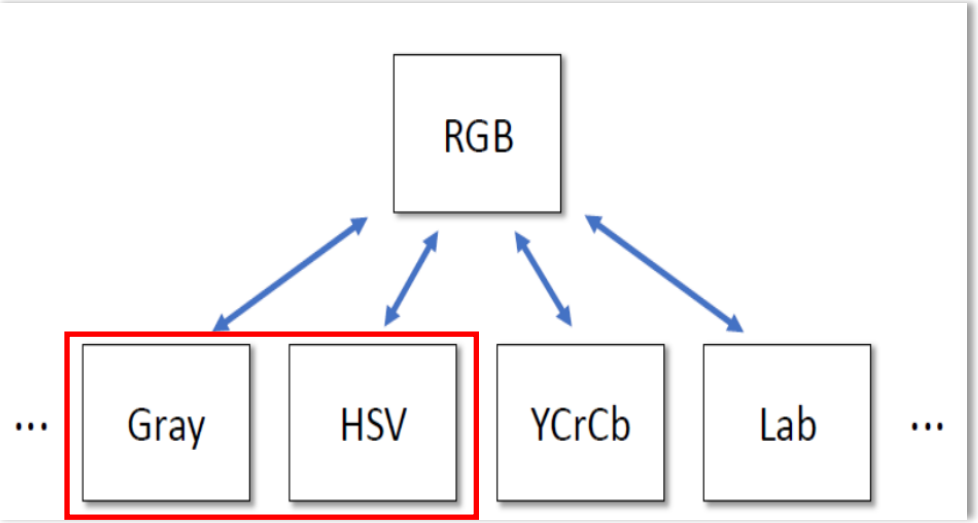
Quantization

RGB Model

- ✓ RGB 컬러 모델은 3원색을 서로 합하는 **가산 컬러 모델**이다.
- ✓ **3개의 채널**로 구성된 컬러영상으로써 3차원 구조의 배열로 표현한다
- ✓ **빛이 약해지면 R,G,B값이 모두 작아진다**는 단점이 있다.



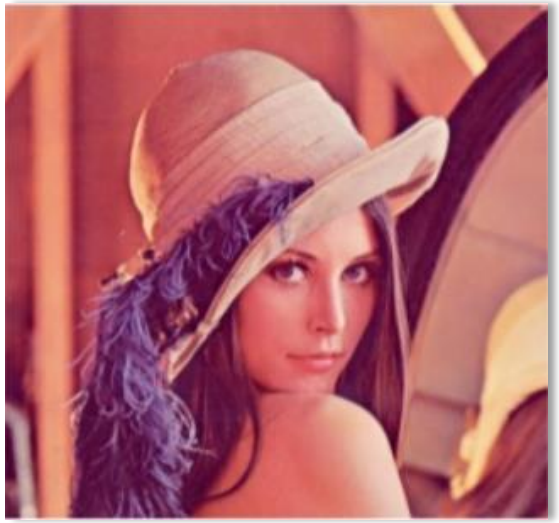
RGB



다양한 색 공간으로의 변환

Grayscale Model

- ✓ **1개의 채널**로 구성된 컬러영상으로써 2차원 구조의 배열로 표현한다.
- ✓ **Edge Detection에** 많이 사용한다. (**객체 인식, 영상 분할 및 특징 추출** 등)



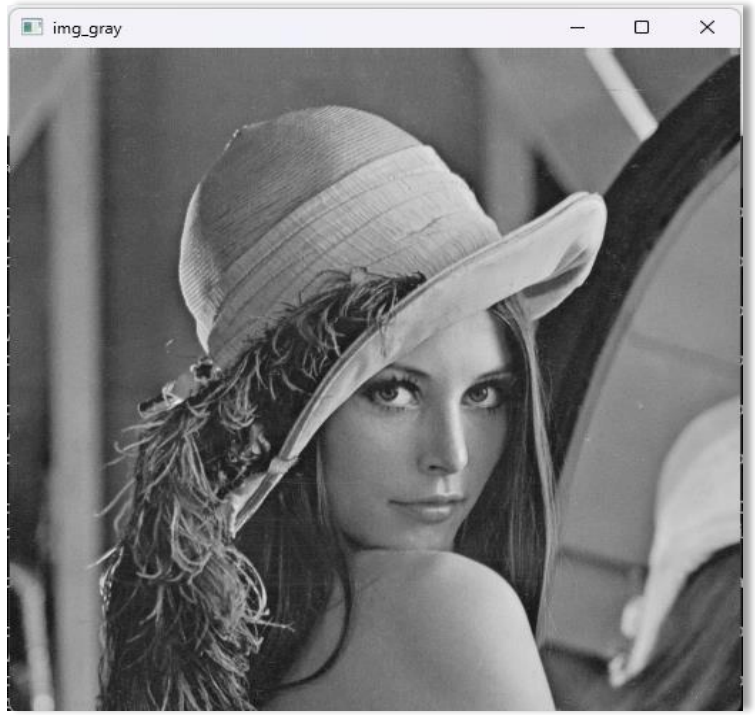
RGB

Grayscale

Color Model- RGB영상을 GRAY 영상으로 바꾸기

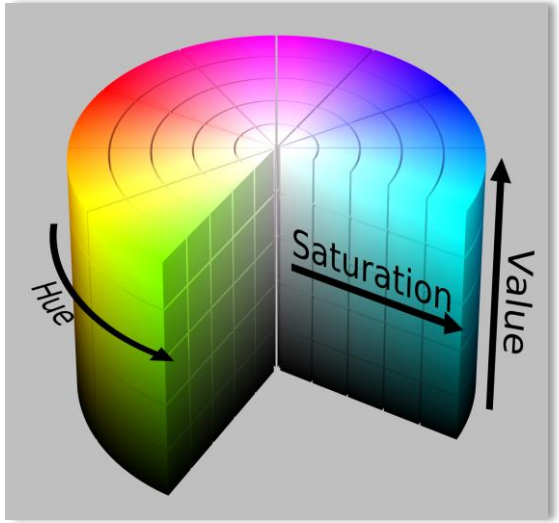
✓ **Grayscale모델**은 RGB **각 채널에 가중치**를 더하여 합한 것이다.

```
#include <opencv2\opencv.hpp>
#include <iostream>

using namespace std;
using namespace cv;

int main(int argc, char** argv) {
    Mat img_BGR = imread("lena.jpg", IMREAD_COLOR);
    if (img_BGR.empty()) { cout << "Image Open Error!" << endl; }
    Mat channels[3]; split(img_BGR, channels);
    Mat img_gray = 0.299 * channels[0] + 0.587 * channels[1] + 0.114 * channels[2];
    namedWindow("img_gray", WINDOW_AUTOSIZE);
    imshow("img_gray", img_gray);
    waitKey();
}
```



HSV Model

- ✓ **3개의 채널**로 구성된 컬러영상으로써 3차원 구조의 배열로 표현한다.
- ✓ **색상 표현이 직관적**이다.(인간 시각 모델에 기반)
- ✓ **색상이나 조명의 변화**에 상대적으로 **강인**하게 대응할 수 있다.(**객체 인식**에서 유용)



HSV

Hue(색상) [0~360°] : 가장 파장이 긴 빨간색이 0도

Saturation(채도) [0~100%] : 색의 진하기 정도
색이 가장 진한 상태 : 100

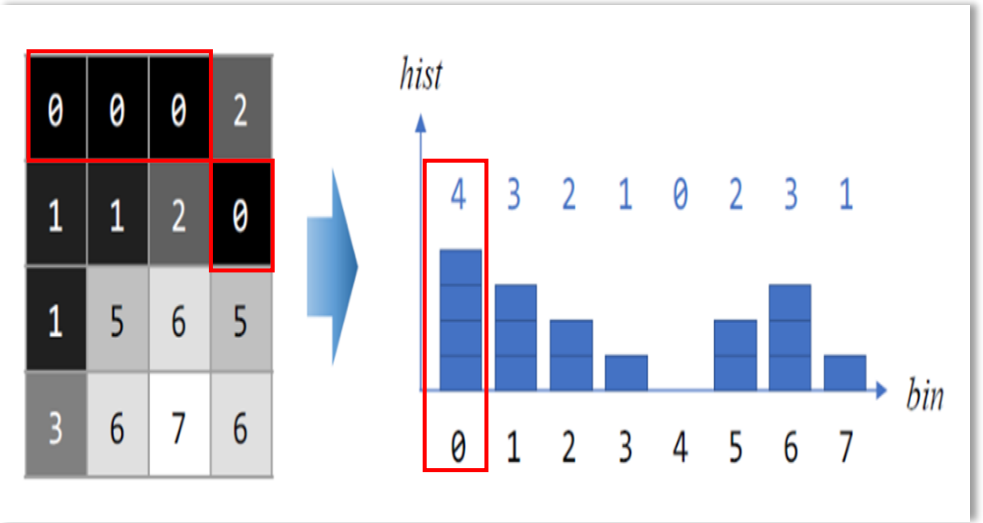
value(명도) [0~100%] : 밝은 정도
가장 밝은 상태: 100

이진화와 히스토그램

- ✓ **이진영상 :** 화소가 **흑(0)** 또는 **백(255)**인 영상이다.(**Edge 검출 후 영상표현**에 많이 사용)
- ✓ **히스토그램 :** 명암 단계 각각에 대해 화소의 발생 빈도를 나타내는 **1차원 배열**이다.

$$b(j,i) = \begin{cases} 1, f(j,i) \geq T \\ 0, f(j,i) < T \end{cases}$$

이진화



히스토그램

Binary Image

이진화

```
Mat binaryThreshold(const Mat& img_input, int threshold) {
    Mat img_binary = img_input.clone();
    int pixel_value = 0;

        for (int y = 0; y < img_binary.rows; y++)
            for (int x = 0; x < img_binary.cols; x++) {
        pixel_value = static_cast<int>(img_input.at<uchar>(y,x));
        img_binary.at<uchar>(y, x) = (pixel_value >= threshold) ? 255 : 0;
    }

    return img_binary;
}
```

Histogram

히스토그램 계산

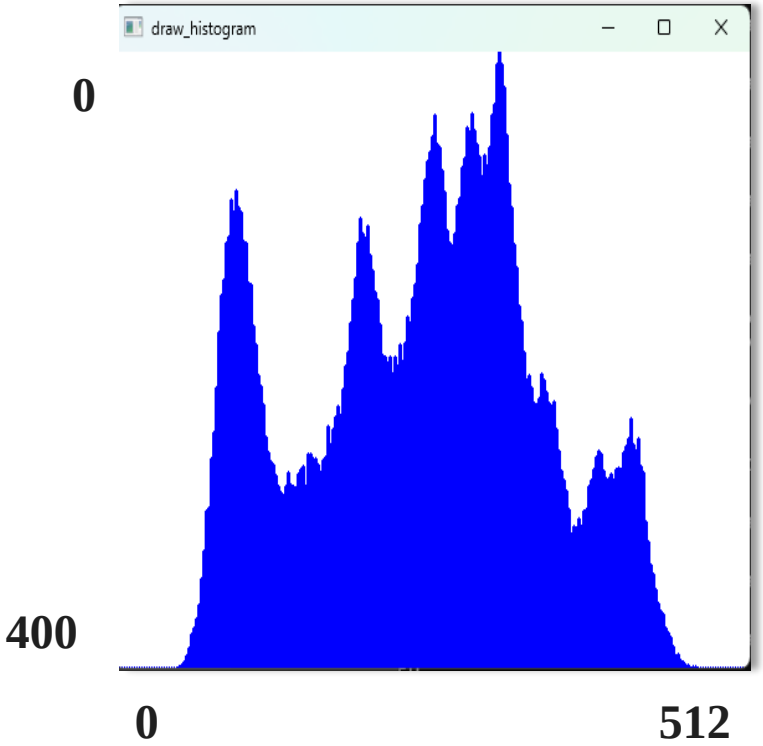
```
void calculateHistogram(const Mat& img, int histogram[256]) {
    for (int i = 0; i < 256; ++i)
        histogram[i] = 0;

    for (int y = 0; y < img.rows; y++) {
        for (int x = 0; x < img.cols; x++) {
            int pixel_value = static_cast<int>(img.at<uchar>(y, x));
            histogram[pixel_value]++;
        }
    }
}
```

```
Histogram[207] : 1027
Histogram[208] : 1114
Histogram[209] : 998
Histogram[210] : 971
Histogram[211] : 1024
Histogram[212] : 904
Histogram[213] : 874
Histogram[214] : 657
Histogram[215] : 566
Histogram[216] : 464
Histogram[217] : 424
Histogram[218] : 353
Histogram[219] : 293
Histogram[220] : 256
Histogram[221] : 246
Histogram[222] : 185
Histogram[223] : 159
Histogram[224] : 138
Histogram[225] : 99
Histogram[226] : 59
Histogram[227] : 66
Histogram[228] : 38
Histogram[229] : 30
Histogram[230] : 16
Histogram[231] : 16
Histogram[232] : 4
Histogram[233] : 7
Histogram[234] : 9
Histogram[235] : 0
```

Histogram

히스토그램 그리기

```
Mat drawHistogram(const int* histogram) {
  Mat histogram_img(400, 512, CV_8UC3, Scalar(255, 255, 255));
  int hist_w = histogram_img.cols ;    int hist_h = histogram_img.rows;
  int bin_w = cvRound((double)hist_w / 256);

  int max_value = 0;
  for (int i = 0; i < 256; i++) { if (histogram[i] > max_value) { max_value = histogram[i]; } }

  for (int i = 0; i < 256; i++) {
    int h = cvRound(histogram[i] * hist_h / max_value);

    line(histogram_img, Point(bin_w * i, hist_h), Point(bin_w * i, hist_h - h), Scalar(255, 0, 0), 2, 8, 0);
  }

  return histogram_img;
}
```



Optimization

- ✓ 컴퓨터 비전은 주어진 문제를 **최적화 문제**로 공식화해 푸는 경우가 많다.
- ✓ Ex) 오츄 알고리즘, back-propagation

오츄 알고리즘

$$\hat{t} = \underset{t \in \{0,1,2,\cdots,L-1\}}{\boldsymbol{argmin}} J(t)$$

$$J(t) = n_0(t)v_0(t) + n_1(t)\ v_1(t)$$

1. 모든 명암값에 대해 목적함수 J를 계산

2. $\hat{t}$을 임곗값 T로 사용해 이진화한다.

n(t) : 화소의 개수

v(t) : 화소의 분산

오츄 알고리즘

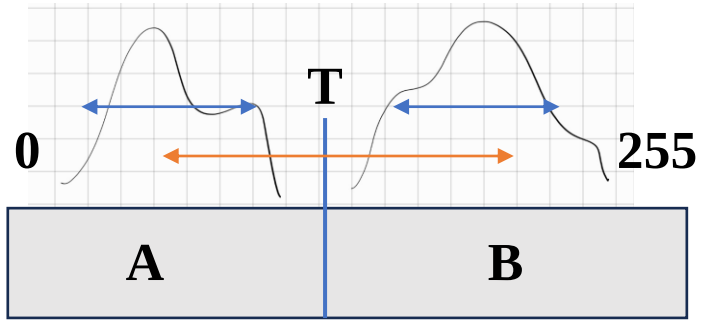
- ✓ **내부 분산**을 최소화 해서 이진화를 가장 잘하는 **임계값 T**를 찾아내는 것
- ✓ **내부 분산을 최소화**하는 것은 **외부분산을 최대화** 하는 것과 같다.

$$V_{within}(t) = W_0(t)V_0(t) + W_1(t)V_1(t)$$

$$W_0(t) = \sum_{i=0}^{t} \hat{h}(i) \quad , \quad W_1(t) = \sum_{i=t+1}^{L-1} \hat{h}(i)$$

$$V_0(t) = \frac{1}{W_0(t)} \sum_{i=1}^{t} \hat{h}(i)(i-\mu_0(t))^2, \quad V_1(t) = \frac{1}{W_1(t)} \sum_{i=t+1}^{L-1} \hat{h}(i)(i-\mu_1(t))^2$$

$$\arg\min V_{within}(t) = \arg\max V_{between}(t) = W_0 \times W_1 \times (\mu_0 - \mu_1)^2$$



Histogram

Histogram

Otsu Algorithm

```
int otsuAlgorithm(const int* histogram) {
    double total_pixel = 0;  double normalized_histogram[256] = { 0 };
    for (int i = 0; i < 256; i++) { total_pixel += histogram[i]; }
    for (int i = 0; i < 256; i++) { normalized_histogram[i] = histogram[i] / total_pixel; }
    int threshold = 0;    double sum1 = 0;  double sum2 = 0;
    double mean1 = 0; double mean2 = 0; double varMax = 0;

    for (int T = 0; T < 256; T++) {
        sum1 += normalized_histogram[T];
        sum2 = 1.0 - sum1;

        if (sum1 > 0 && sum2 > 0) {
            mean1 = 0;
            mean2 = 0;

            for (int i = 0; i <= T; i++) { mean1 += i * normalized_histogram[i]; }
            mean1 /= sum1;

            for (int i = T + 1; i < 256; i++) { mean2 += i * normalized_histogram[i]; }
            mean2 /= sum2;

            double varBetween = sum1 * sum2 * (mean1 - mean2) * (mean1 - mean2);

            if (varBetween > varMax) {
                varMax = varBetween;
                threshold = T;
            }
        }
    }
    return threshold;
}
```

Threshold = 117

히스토그램 평활화

✓ **Histogram Equalization :** 히스토그램이 평평하게 되도록 영상을 조작해서 영상의 **명암대비를 높이는** 기법이다.

Histogram Equalization

$$l' = \text{round}(\ddot{h}(l) \times (L-1))$$

$\dot{h}$ **: 정규화 히스토그램.**
(모든 칸을 전부 더하면 1이 되도록 하는 히스토그램)

$\ddot{h}$: **누적 정규화 히스토그램.**

Histogram

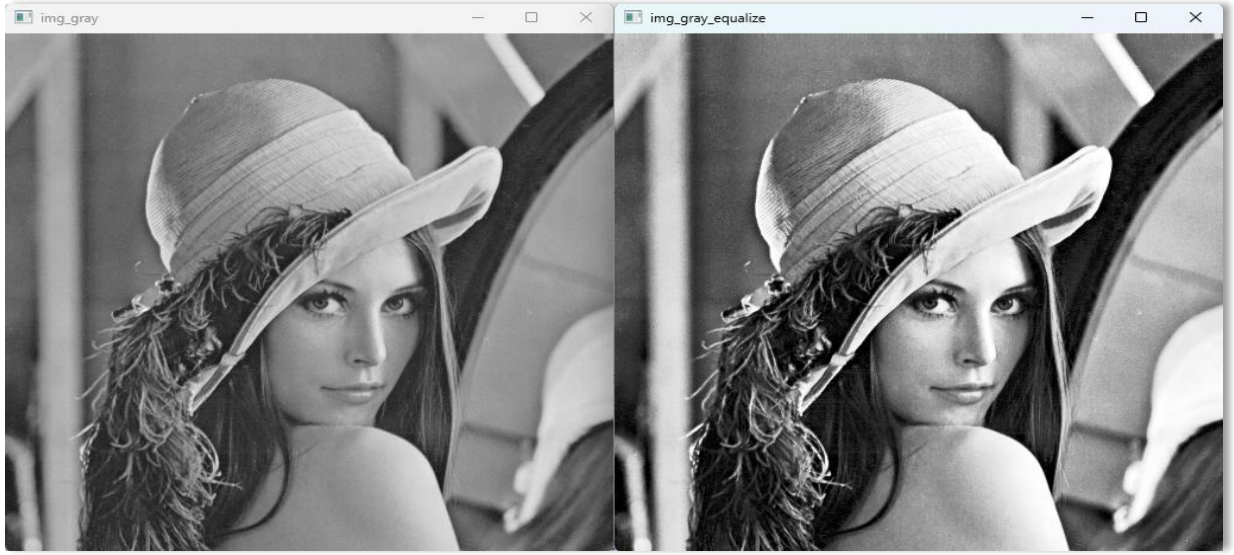
Histogram Equalization

```
Mat equalizeHistogram(const int* histogram, const Mat& img_input) {
  int cumulative_histogram[256] = { 0 };
  cumulative_histogram[0] = histogram[0];
  for (int i = 1; i < 256; i++) { cumulative_histogram[i] = cumulative_histogram[i - 1] + histogram[i]; }

  int total_pixel_value = 0;
  for (int i = 0; i < 256; i++) { total_pixel_value += histogram[i]; }
  double normalization_factor = 255.0 / total_pixel_value;

  int normalized_cumulative_histogram[256];
  for (int i = 0; i < 256; i++) { normalized_cumulative_histogram[i] = cvRound(cumulative_histogram[i] * normalization_factor); }

  Mat img_output = img_input.clone();
  for (int y = 0; y < img_output.rows; y++) {
    for (int x = 0; x < img_output.cols; x++) {
      int pixel_value = static_cast<int>(img_output.at<uchar>(y, x));
      img_output.at<uchar>(y, x) = static_cast<uchar>(normalized_cumulative_histogram[pixel_value]);
    }
  }

  return img_output;
}
```

1. 누적 histogram 구하기
2. 모든 histogram 누적합 구하기
3. 정규화 상수 구하기 ($\frac{255}{\text{누적합}}$)
4. 누적 histogram * 정규화 factor

↳ 히스토그램 평탄화

Gray_image **Equalized_gray_image**

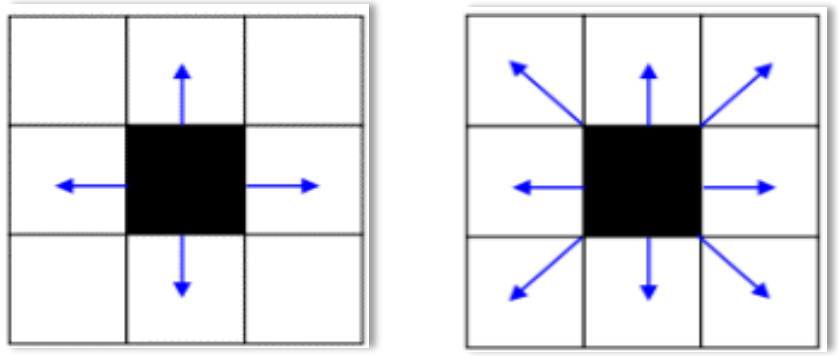


명암대비가 높아짐(인지도 증가)

Connectivity and Morphology

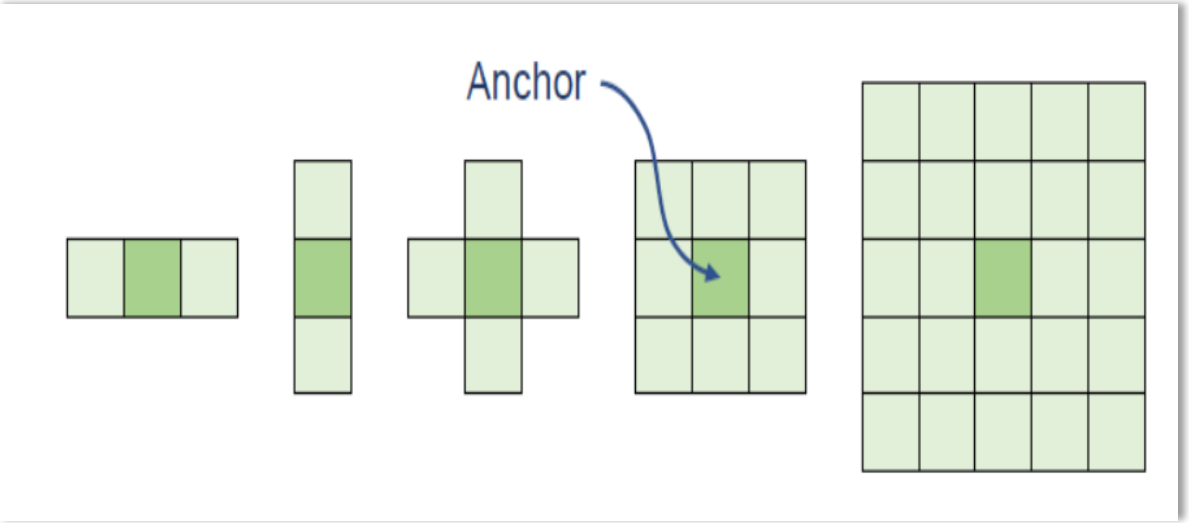
- ✓ **연결성**은 픽셀의 연결 여부를 결정한다.
- ✓ **모폴로지**는 영상을 형태학적으로 다루는 기법이다.

연결성



4-connectivity **8-connectivity**

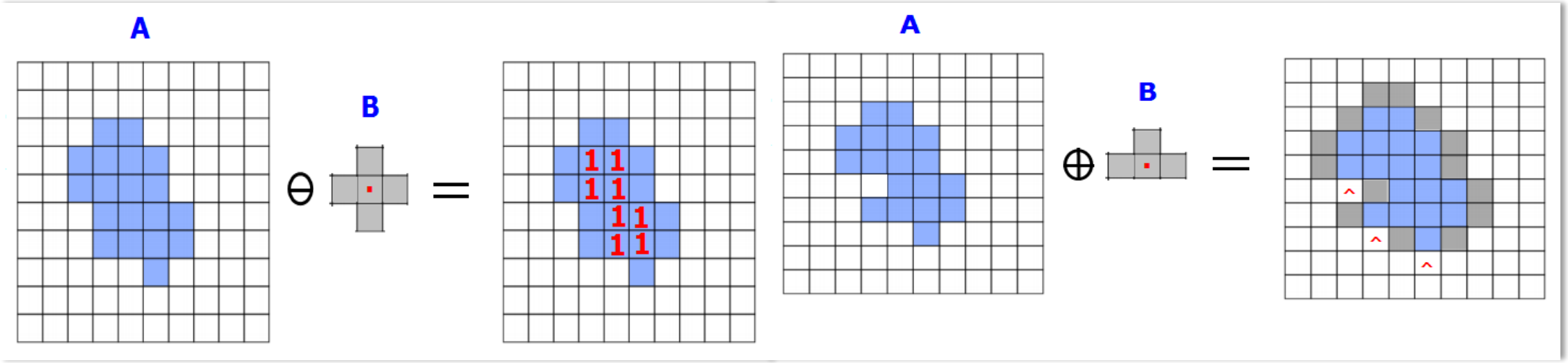
모폴로지의 구조요소



Morphology

열림(opening) : 침식 -> 팽창
닫힘(closing) : 팽창 -> 침식

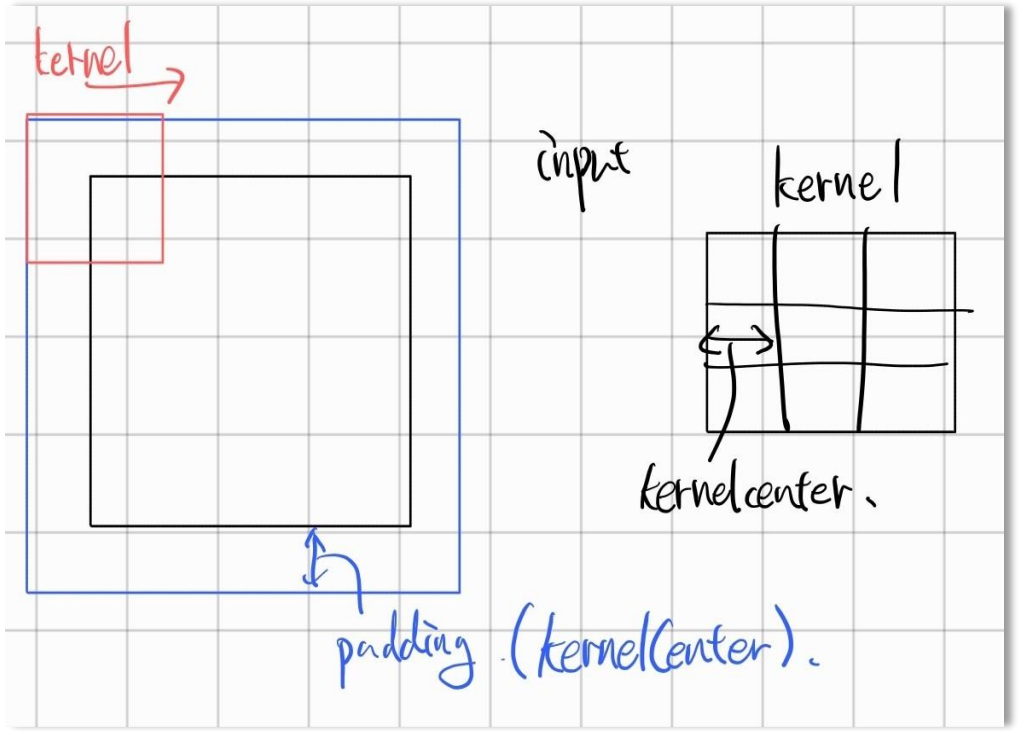
- ✓ **이진 영상**에서 많이 사용하지만, **그레이 영상**에도 적용 가능하다.
- ✓ 객체의 형태를 **분석**하거나 **전처리**하는데 활용된다.
- ✓ 기본 연산은 **팽창(dilation)**과 **침식(erosion)**이다.



Dilation **Erosion**

Morphology

Algorithm



erode.

padding : 255.

& 연산.

kernel 범위 내에서

하나라도 255보다 작으면

그 값으로 변경.

(여기서는 0)

dilate.

padding : 0

& 연산.

kernel 범위 내에서

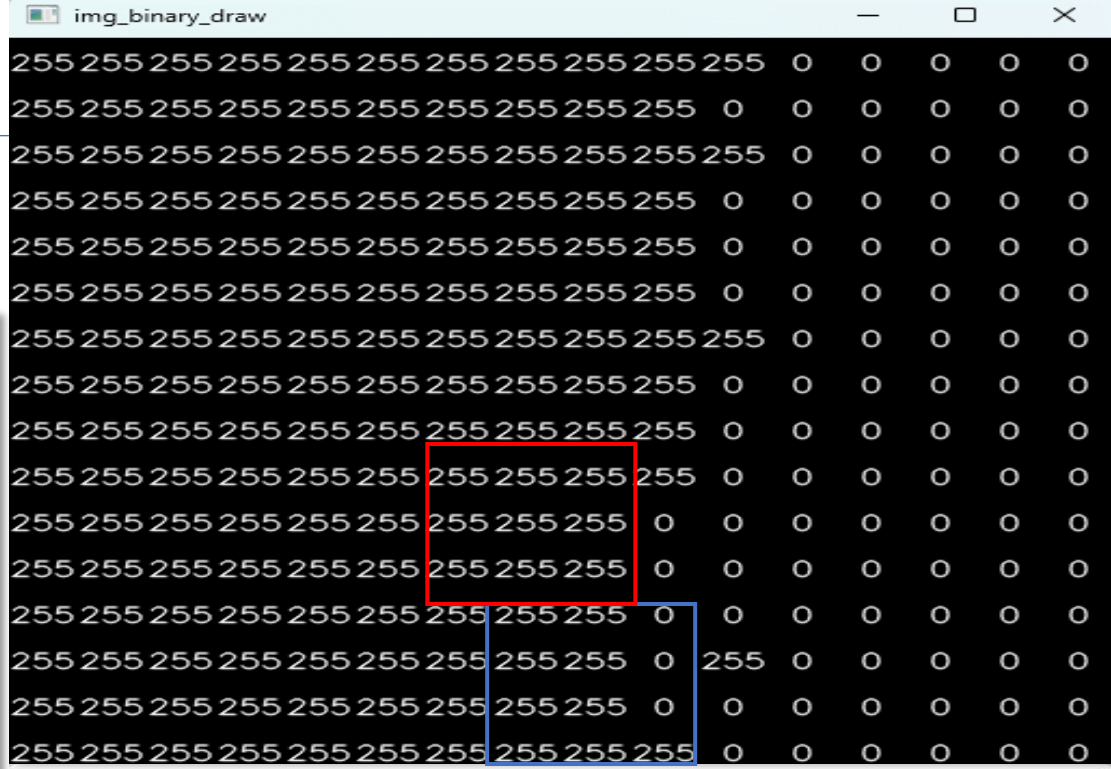
하나라도 0보다 크면

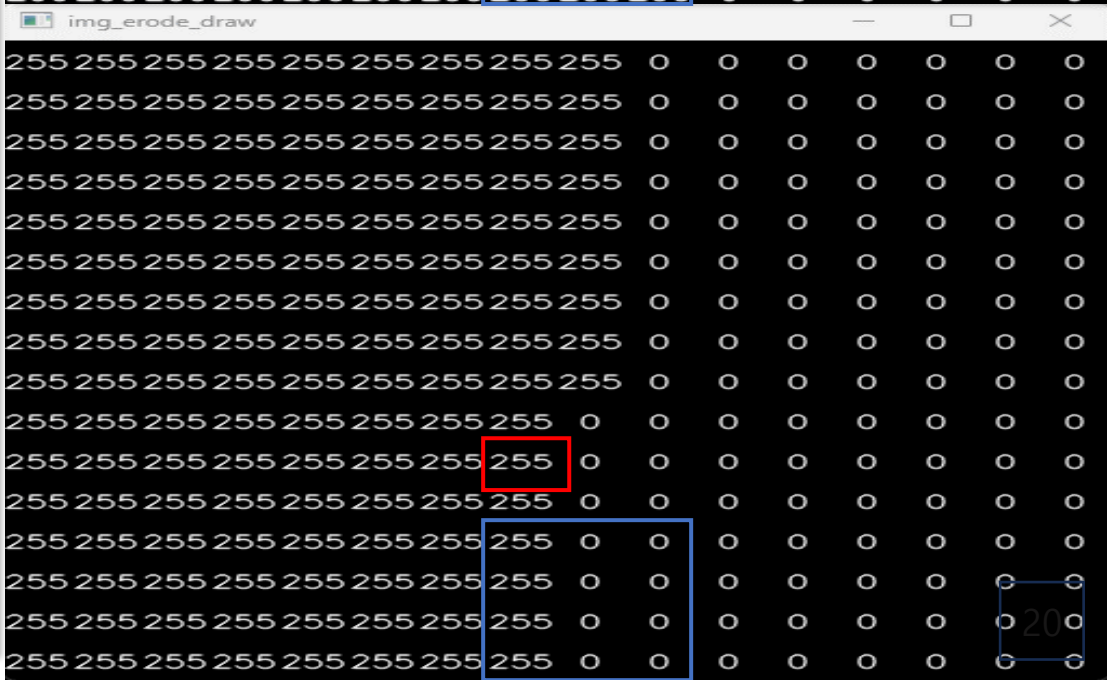
그 값으로 변경

Morphology

Erode

```
Mat Morphology_erode(Mat& img, Mat& kernel) {
  Mat img_erode = Mat::zeros(img.size(), img.type());
  int kernelCenter = (kernel.rows - 1) / 2;

  Mat img_padded;  copyMakeBorder(img, img_padded, kernelCenter, kernelCenter, kernelCenter, kernelCenter, BORDER_CONSTANT, 255);

  for (int i = kernelCenter; i < img_padded.rows - kernelCenter; i++) {
    for (int j = kernelCenter; j < img_padded.cols - kernelCenter; j++) {
      uchar minVal = 255;

      for (int m = 0; m < kernel.rows; m++) {
        for (int n = 0; n < kernel.cols; n++) {
          uchar pixelVal = img_padded.at<uchar>(i + m - kernelCenter, j + n - kernelCenter);
          uchar kernelVal = kernel.at<uchar>(m, n);
          uchar result = pixelVal & kernelVal;

          if (result < minVal) minVal = result;
        }
      }
      img_erode.at<uchar>(i - kernelCenter, j - kernelCenter) = minVal;
    }
  }

  return img_erode;
}
```

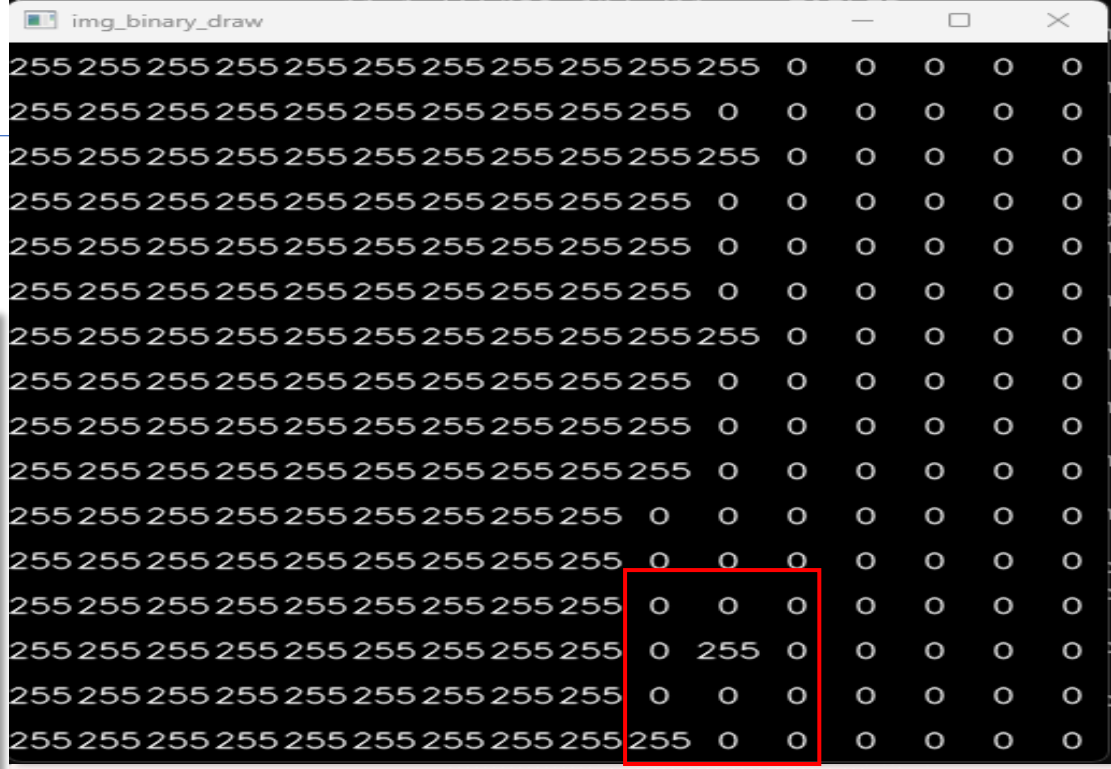


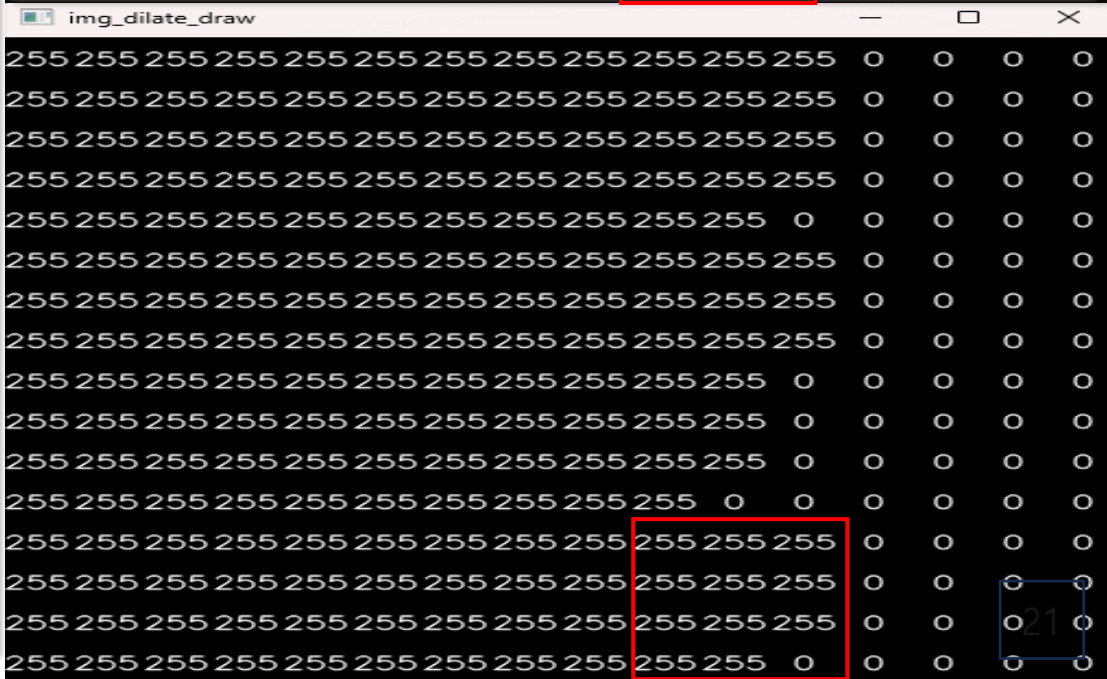


Morphology

Dilate

```
Mat Morphology_dilate(Mat& img, Mat& kernel) {
  Mat img_dilate = Mat::zeros(img.size(), img.type());
  int kernelCenter = (kernel.rows - 1) / 2;

  Mat img_padded;  copyMakeBorder(img, img_padded, kernelCenter, kernelCenter, kernelCenter,  kernelCenter, BORDER_CONSTANT, 0);

  for (int i = kernelCenter; i < img_padded.rows - kernelCenter; i++) {
    for (int j = kernelCenter; j < img_padded.cols - kernelCenter; j++) {
      uchar maxVal = 0;

      for (int m = 0; m < kernel.rows; m++) {
        for (int n = 0; n < kernel.cols; n++) {
          uchar pixelVal = img_padded.at<uchar>(i + m - kernelCenter, j + n - kernelCenter);
          uchar kernelVal = kernel.at<uchar>(m, n);
          uchar result = pixelVal & kernelVal;

          if (result > maxVal) maxVal = result;
        }
      }
      img_dilate.at<uchar>(i-kernelCenter, j-kernelCenter) = maxVal;
    }
  }
  return img_dilate;
}
```





명암 조절

- ✓ 인간의 눈은 빛의 밝기 변화에 **비선형적**으로 반응한다.
- ✓ **감마보정** : 이러한 **비선형적인 시각반응을 보정**하는데 활용

감마보정

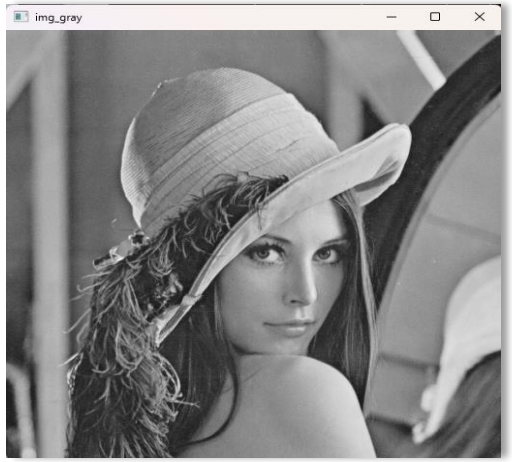
$$f'(j,i) = (L-1) \times \dot{f}(j,i)^{\gamma}$$

$\dot{f}$은 [0,L-1] 범위를 [0,1] 범위로 **정규화** 한 영상이다.

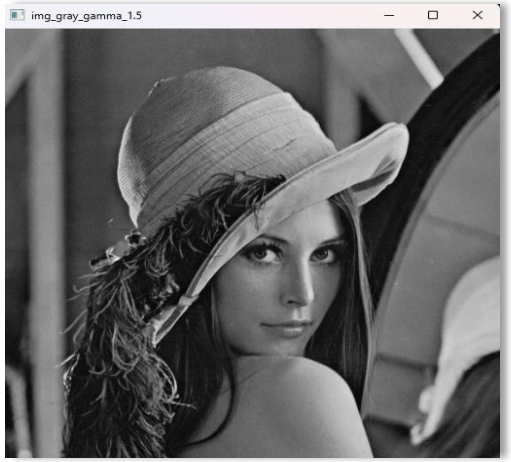
Histogram

Gamma correction

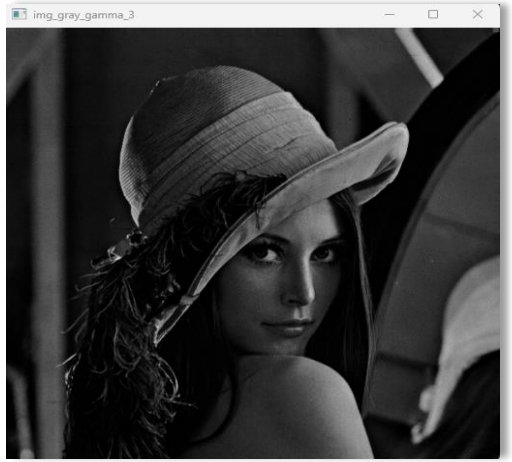
```
Mat gamma_correction(Mat& img, double gamma_value) {
  Mat img_normalized(img.rows, img.cols, CV_64F);
  Mat result(img.rows, img.cols, CV_8UC1);

  for (int y = 0; y < img.rows; y++)
    for (int x = 0; x < img.cols; x++) {
      img_normalized.at<double>(y, x) = static_cast<double>(img.at<uchar>(y, x)) / 255.0;
    }


  for (int y = 0; y < img.rows; y++)
    for (int x = 0; x < img.cols; x++) {
      img_normalized.at<double>(y, x) = pow(img_normalized.at<double>(y, x), gamma_value);
    }


  for (int y = 0; y < img.rows; y++)
    for (int x = 0; x < img.cols; x++) {
      img_normalized.at<double>(y, x) *= 255.0;
    }


  for (int y = 0; y < img.rows; y++)
    for (int x = 0; x < img.cols; x++) {
      result.at<uchar>(y, x) = static_cast<uchar>(img_normalized.at<double>(y, x));
    }


  return result;
}
```



Gamma = 0



Gamma = 1.5

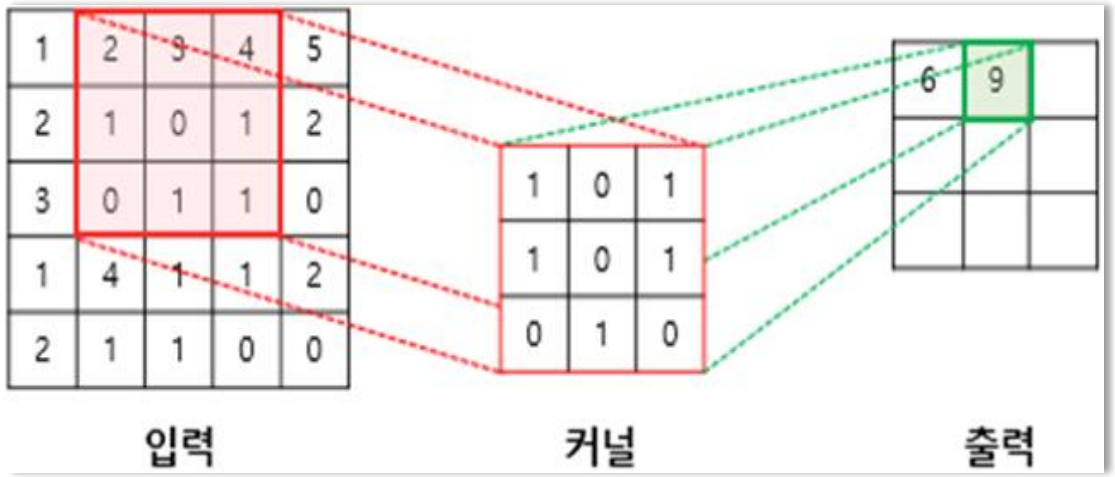


Gamma = 3

컨볼루션

- ✓ 입력 영상 f의 **각 화소에 필터를 적용해 곱의 합**을 구하는 연산이다.
- ✓ **Stride** : 커널이 입력 이미지를 얼마나 건너뛰면서 이동할지를 나타낸다.
- ✓ **Padding** : 입력 이미지 주변에 추가되는 가상의 픽셀을 나타낸다.

$$\mathbf{f'(y,x)} = \sum_{j=-(h-1)/2}^{(h-1)/2} \sum_{i=-(w-1)/2}^{(w-1)/2} u(j,i) f(y+j, x+j)$$



컨볼루션

- ✓ **Convolution** 자체는 특정한 목적이 없는 일반 연산이다.
- ✓ **목적에 따른 필터선택**을 하면 된다.
- ✓ Ex) **스무딩 필터, 샤프닝 필터**

스무딩 필터

$\frac{1}{9} \times$

1	1	1
1	1	1
1	1	1

박스 필터

$\frac{1}{4.8976} \times$

0.3679	0.6065	0.3679
0.6065	1.0000	0.6065
0.3679	0.6065	0.3679

가중 평균 필터

잡음제거, Blurring

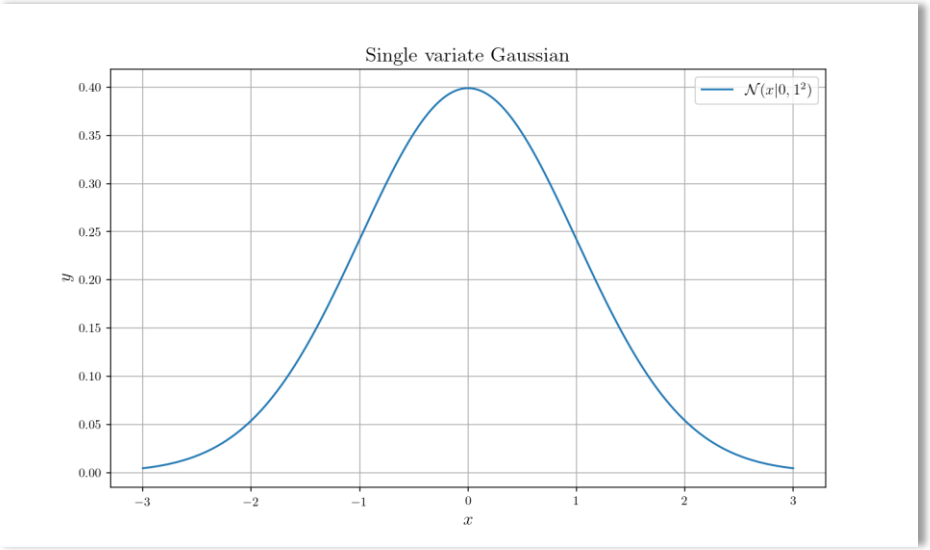
샤프닝 필터

−1	−1	−1
−1	8	−1
−1	−1	−1

에지를 선명하게

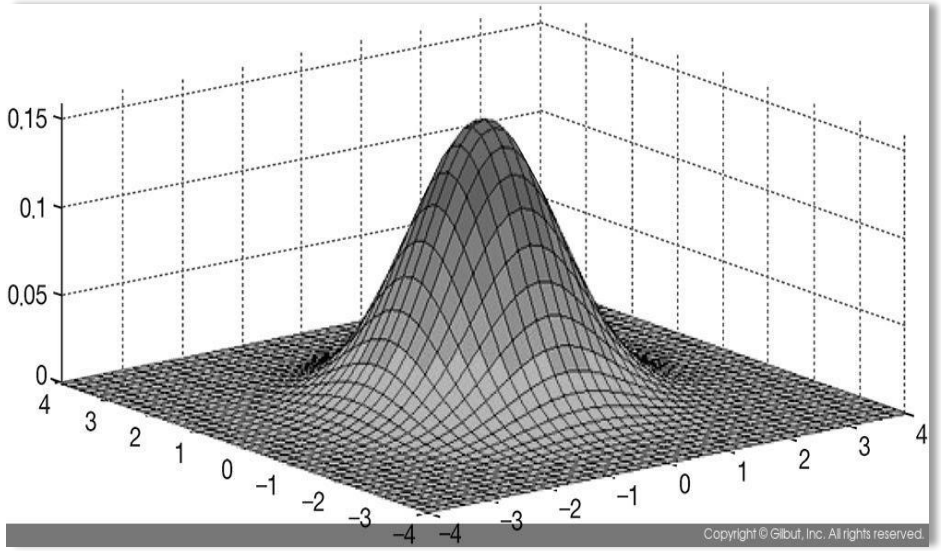
Gaussian

1차원 가우시안 분포



$$g(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

2차원 가우시안 분포



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

Gaussian

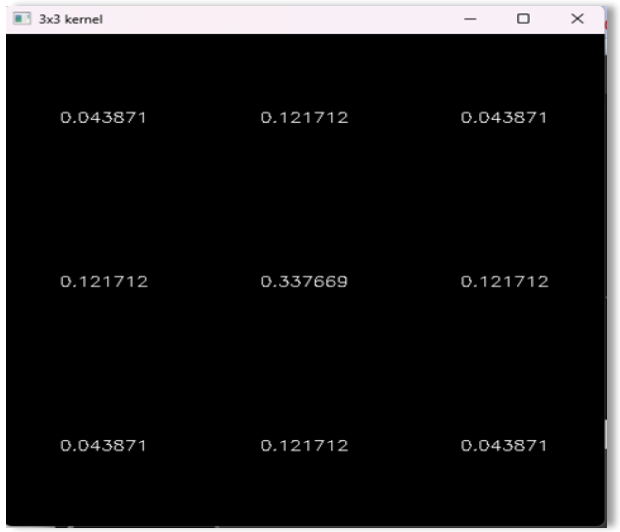
가우시안 잡음 생성

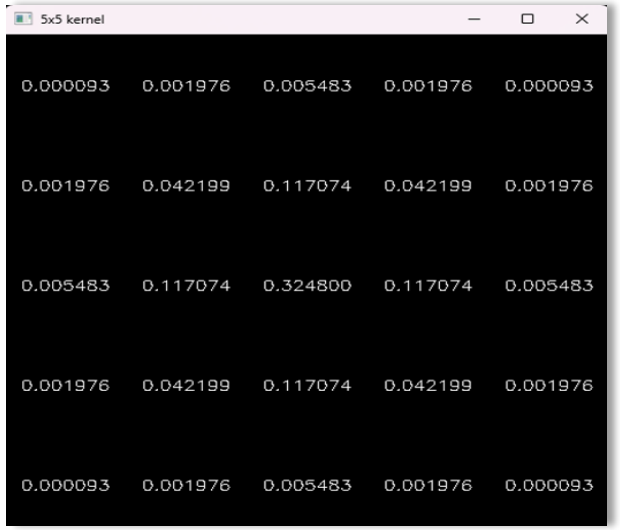
```
Mat add_Gaussian_Noise(Mat& img, double mean, double std) {
    Mat noise(img.size(), CV_8UC1);
    Mat result(img.size(), img.type());

    unsigned seed = static_cast<unsigned>(chrono::system_clock::now().time_since_epoch().count());
    default_random_engine generator(seed);
    normal_distribution<double> distribution(mean, std);

    for (int y = 0; y < img.rows; ++y) {
        for (int x = 0; x < img.cols; ++x) {
            double noise_val = distribution(generator);
            result.at<uchar>(y, x) = saturate_cast<uchar>(img.at<uchar>(y, x) + noise_val);
        }
    }

    return result;
}
```

Gaussian

가우시안 커널 생성

```
Mat Gaussian_Kernel(int size, double sigma) {
    Mat kernel(size, size, CV_64F);

    int kernelCenter = (size - 1) / 2;
    double sum = 0.0;

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int x = kernelCenter - i;
            int y = kernelCenter - j;
            kernel.at<double>(i, j) = exp(-(static_cast<double>(x * x) +
                static_cast<double>(y * y)) / (2.0 * sigma * sigma)) / sqrt(2.0 * PI * sigma * sigma);
            sum += kernel.at<double>(i, j);
        }
    }
    kernel /= sum;
    return kernel;
}
```

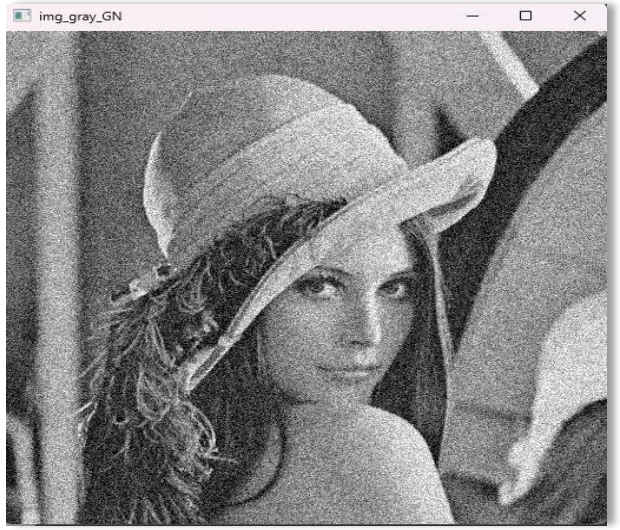


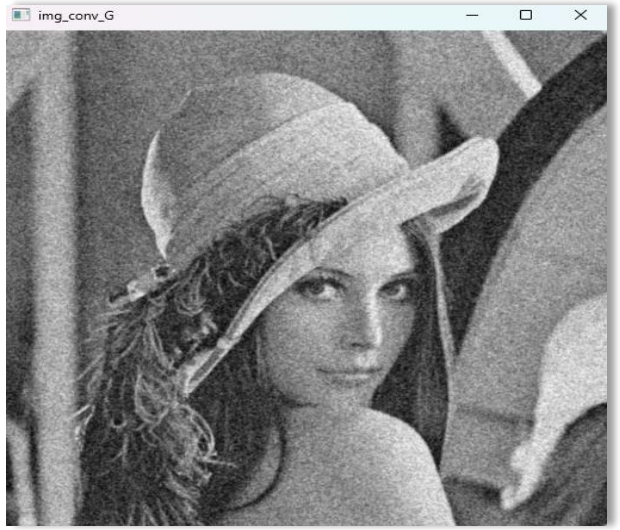


Gaussian

가우시안 컨볼루션 수행

```
Mat convolution_Gaussian(Mat& img, Mat& kernel) {
  int kernelCenter = (kernel.rows - 1) / 2;
  Mat img_conv(img.cols - 2 * kernelCenter, img.rows - 2 * kernelCenter, CV_8UC1);

  for (int i = kernelCenter; i < img.rows - kernelCenter; i++) {
    for (int j = kernelCenter; j < img.cols - kernelCenter; j++) {
      double value = 0;
      for (int m = 0; m < kernel.rows; m++) {
        for (int n = 0; n < kernel.cols; n++) {
          value += static_cast<double>(img.at<uchar>(i - kernelCenter + m,
            j - kernelCenter + n)) * kernel.at<double>(m, n);
        }
      }

      img_conv.at<uchar>(i - kernelCenter, j - kernelCenter) = static_cast<uchar>(value);
    }
  }

  return img_conv;
}
```





Affine Transform

- ✓ **Affine Transform : 직선성, 평행 관계가 유지**되는 변환
- ✓ **평행 이동, 회전, 크기변환 대칭** 등

homogeneous coordinate : $\bar{p} = (x,y,1)$

평행 이동

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

회전

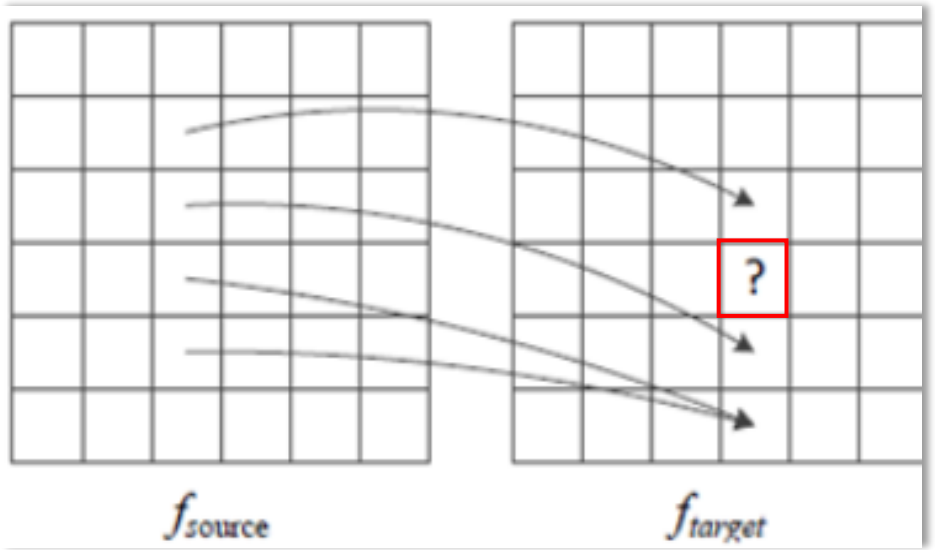
$$T(t_x, t_y) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

크기

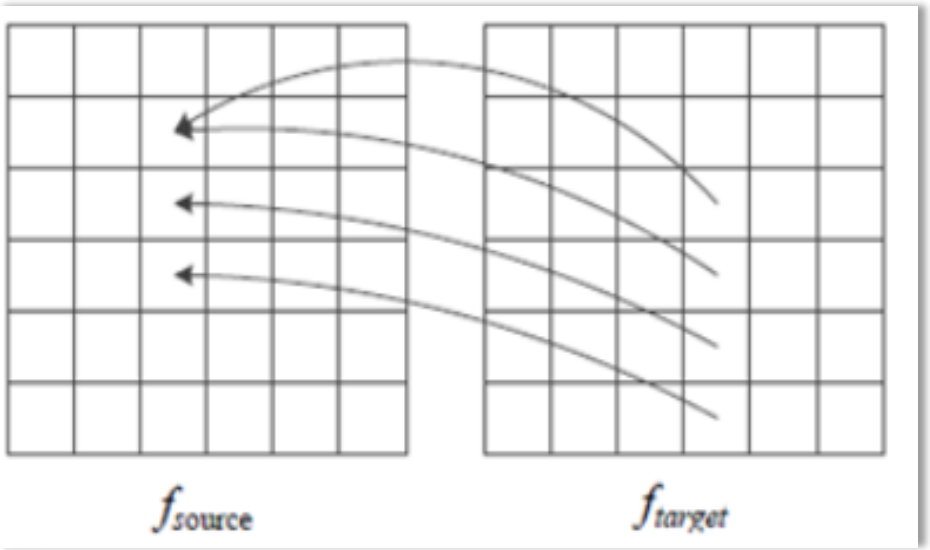
$$T(t_x, t_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

영상의 기하변환

- ✓ **전방 변환**은 **Aliasing**현상이 발생한다.
- ✓ **후방 변환**을 이용한 **Anti-Aliasing**을 통해 문제 해결



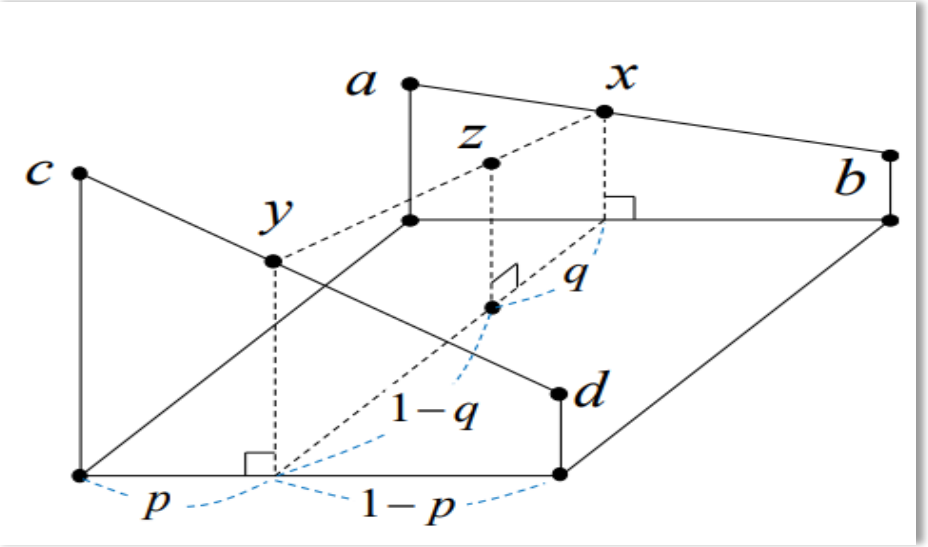
전방 변환(변환행렬 : A)



후방 변환(변환행렬 : A^{-1})

Interpolation

- ✓ Nearest Neighbor Interpolation :가장 가까운 픽셀의 픽셀 값으로 할당하는 방법
- ✓ Bilinear interpolation method : 4개 화소와 걸친 비율에 따라서 가중 평균하여 화소 값을 결정
- ✓ Cubic Convolution Interpolation : 주위에 있는 16개의 픽셀 값으로 보간하는 방법



Bilinear interpolation method

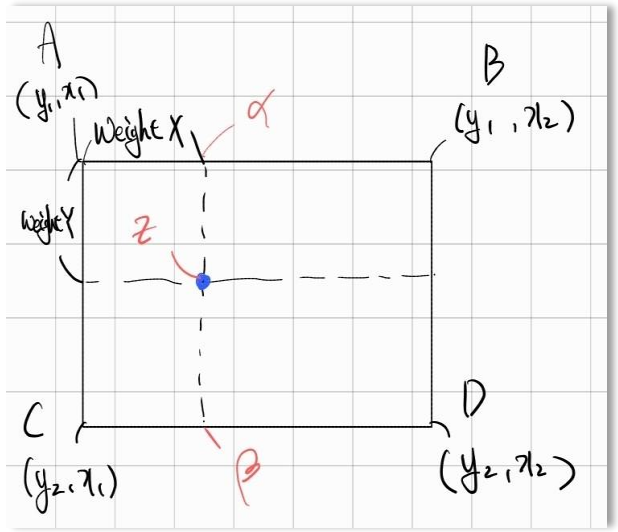
$$x = (1-p)a + pb$$
$$y = (1-p)c + pd$$
$$z = (1-q)x + qy$$

$$z = (1-p)(1-q)a + p(1-q)b + (1-p)qc + pqd$$

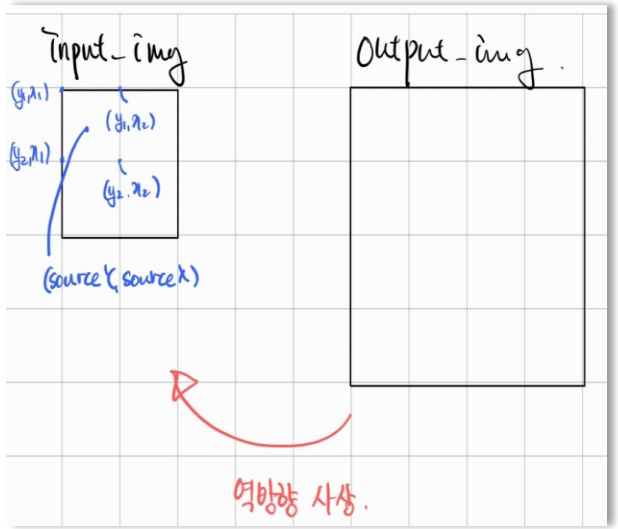
Interpolation

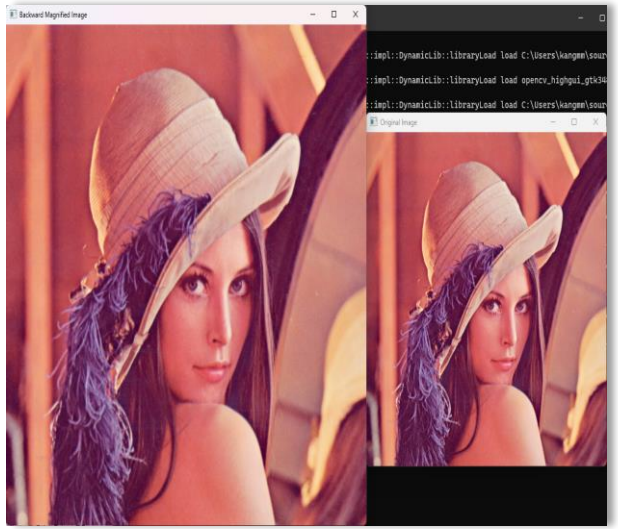
Bilinear interpolation method

```
Mat bilinearInterpolation(const Mat& input, float scaleX, float scaleY) {
    int inputWidth = input.cols;    int inputHeight = input.rows;
    int outputWidth = static_cast<int>(inputWidth * scaleX);
    int outputHeight = static_cast<int>(inputHeight * scaleY);

    Mat output = Mat::zeros(outputHeight, outputWidth, input.type());

    for (int y = 0; y < outputHeight; ++y) {
        for (int x = 0; x < outputWidth; ++x) {
            float sourceX = x / scaleX;
            float sourceY = y / scaleY;

            int x1 = static_cast<int>(sourceX);
            int y1 = static_cast<int>(sourceY);
            int x2 = x1 + 1;
            int y2 = y1 + 1;

            float weightX = sourceX - x1;
            float weightY = sourceY - y1;

            x1 = min(max(x1, 0), inputWidth - 1);
            x2 = min(max(x2, 0), inputWidth - 1);
            y1 = min(max(y1, 0), inputHeight - 1);
            y2 =min(max(y2, 0), inputHeight - 1);

            output.at<Vec3b>(y, x) =
                (1 - weightX) * (1 - weightY) * input.at<Vec3b>(y1, x1) +
                weightX * (1 - weightY) * input.at<Vec3b>(y1, x2) +
                (1 - weightX) * weightY * input.at<Vec3b>(y2, x1) +
                weightX * weightY * input.at<Vec3b>(y2, x2);
        }
    }
    return output;
}
```



$$A(1-weightX)+B(weightX)=\alpha$$
$$C(1-weightX)+D(weightX)=\beta$$
$$\alpha(1-weightY)+\beta(weightY)=z$$





감사합니다.