ICE4027 Digital Image Processing

Lab.3 – Linear Filtering

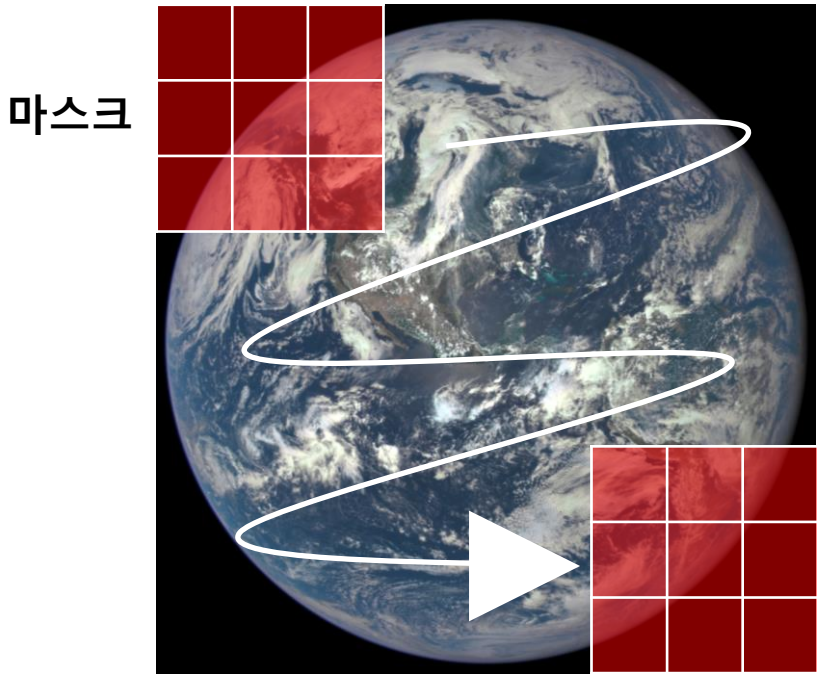
Prof. In Kyu Park

인하대학교 시각컴퓨팅 및 학습 연구실
Visual Computing and Learning Laboratory

Contents

Introduction

- **선형 필터링**
  - 마스크(또는 window, kernel)를 주어진 영상 위에서 이동하면서 처리하는 방식
  - 마스크와 함수를 결합하여 필터의 역할을 수행
  - 함수는 마스크 안의 화소들에 대한 곱셈, 덧셈으로 이루어진 선형 연산



Pixel Access

- **Mat 객체 data 기반 화소 접근법**
  - Mat 객체의 영상 데이터를 포인터로 받아와 직접 접근하는 방법
  - 데이터가 1차원 배열 형태이므로 이를 고려해 인덱싱을 수행해야함
  - 포인터를 사용해 신속하며, 동적 할당 문제를 해결
  - 포인터로 가리키고 있으므로 다시 Mat객체의 데이터에 따로 저장할 필요 없음

```
Mat myCopy(Mat srcImg) {
    int width = srcImg.cols;
    int height = srcImg.rows;
    Mat dstImg(srcImg.size(), CV_8UC1); // 입력영상과 동일한 크기의 Mat 생성
    uchar* srcData = srcImg.data; // Mat객체의 data를 가리키는 포인터
    uchar* dstData = dstImg.data;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            dstData[y * width + x] = srcData[y * width + x];
            // 화소 값을 일일이 읽어와 다른 배열에 저장
        }
    }

    return dstImg;
}
```

Convolution on Mask

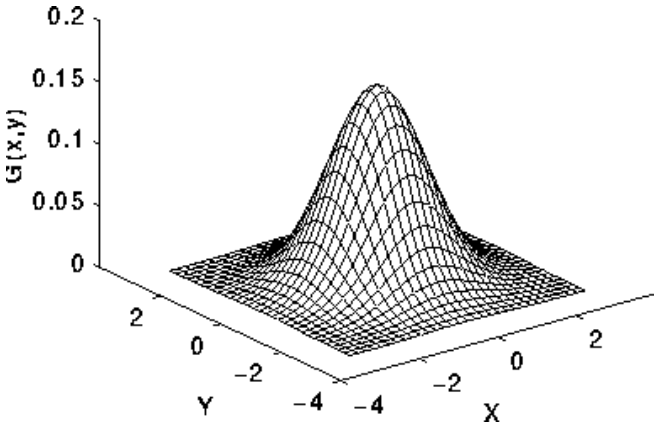
- **마스크에 대한 convolution**
  - 선형 필터링에서 마스크 기반 처리를 위해 필수적인 연산
  - 화소 접근법에서 이웃 화소의 값을 함께 읽어와 마스크와 연산하는 방식
  - 영상 가장자리에서는 영상 밖의 화소를 읽지 않도록 하는 조건이 추가

```
int myKernelConv3x3(uchar* arr, int kernel[][3], int x, int y, int width, int height) {
    int sum = 0;
    int sumKernel = 0;

    // 특정 화소의 모든 이웃화소에 대해 계산하도록 반복문 구성
    for (int j = -1; j <= 1; j++) {
        for (int i = -1; i <= 1; i++) {
            if ((y + j) >= 0 && (y + j) < height && (x + i) >= 0 && (x + i) < width) {
                // 영상 가장자리에서 영상 밖의 화소를 읽지 않도록 하는 조건문
                sum += arr[(y + j) * width + (x + i)] * kernel[i + 1][j + 1];
                sumKernel += kernel[i + 1][j + 1];
            }
        }
    }

    if (sumKernel != 0) { return sum / sumKernel; } // 합이 1로 정규화되도록 해 영상의 밝기변화 방지
    else { return sum; }
}
```

가장자리 문제

Linear Filter

- **Gaussian 필터**
  - Gaussian 분포 형태의 마스크로 현재 화소와 가까울수록 가중치를 부여
  - Blur(또는 smoothing)효과가 나타나므로 잡음 제거에 탁월
  - Blur의 정도는 표준편차 σ에 의해 좌우됨 (분포의 폭을 결정)



$\frac{1}{16} \times$

1	2	1
2	4	2
1	2	1

$\frac{1}{273} \times$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

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

Linear Filter

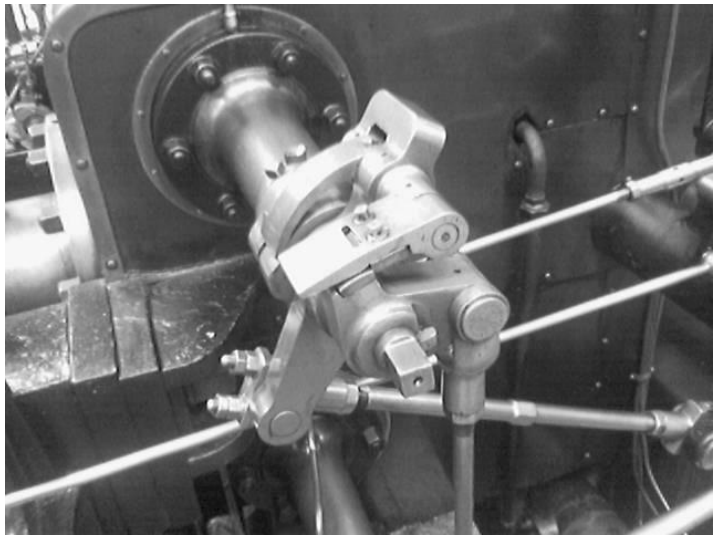
- **Gaussian 필터**

```
void cv::GaussianBlur ( InputArray  src,
                        OutputArray dst,
                        Size        ksize,
                        double      sigmaX,
                        double      sigmaY = 0 ,
                        int         borderType = BORDER_DEFAULT
                      )
```

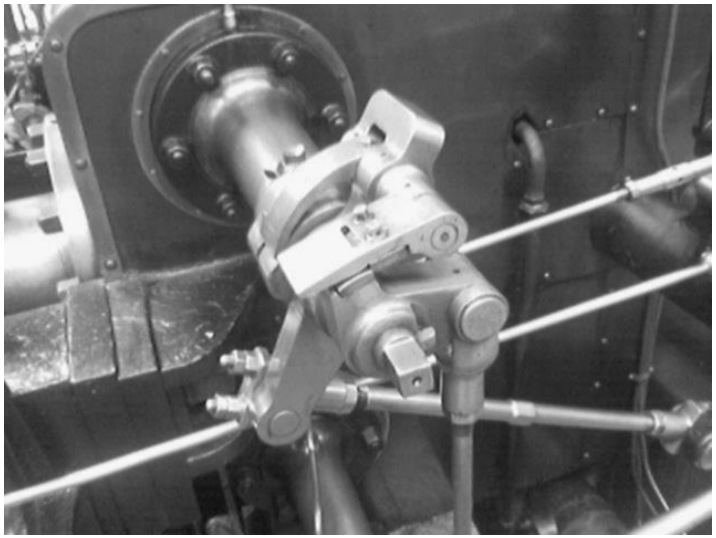
```
Mat myGaussianFilter(Mat srcImg) {
#if USE_OPENCV // OpenCV로 구현한 것
    Mat dstImg(srcImg.size(), CV_8UC1);

    cv::GaussianBlur(srcImg, dstImg, Size(3, 3), 0);
    // 마스크의 크기를 지정하면 자체적으로 마스크 생성 후 연산

    return dstImg;
#else // 직접 구현한 것 (매크로에 의해 컴파일시 선택됨)
    int width = srcImg.cols;
    int height = srcImg.rows;
    int kernel[3][3] = { 1, 2, 1,
                         2, 4, 2,
                         1, 2, 1 }; // 3x3 형태의 Gaussian 마스크 배열
    Mat dstImg(srcImg.size(), CV_8UC1);
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            dstData[y * width + x] = myKernelConv3x3(srcData, kernel, x, y, width, height);
            // 앞서 구현한 convolution에 마스크 배열을 입력해 사용
        }
    }

    return dstImg;
#endif
}
```

Linear Filter

- Gaussian 필터



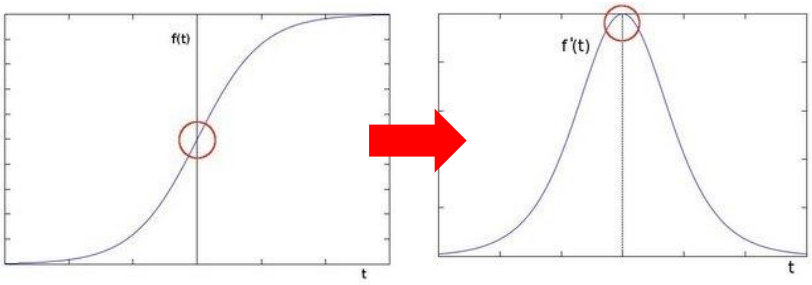
적용 전



적용 후

Linear Filter

- **Sobel 필터**
  - 대표적인 에지(edge) 검출 필터
  - 영상에서 특정 방향에 대한 미분 성격을 가지는 마스크를 사용
  - 가로방향과 세로방향에 대한 에지 검출을 별도로 수행하고 이를 절대값 합 형태로 합성해 최종적인 에지를 구할 수 있음



Z1	Z2	Z3
Z4	Z5	Z6
Z7	Z8	Z9

(a)

-1	-2	-1
0	0	0
1	2	1

(b)

-1	0	1
-2	0	2
-1	0	1

(c)

Fig. 1. (a) 3x3 region of an image. (b) 0^0 Sobel kernel. (c) 90^0 Sobel kernel

$$G_x = (Z_7 + 2 * Z_8 + Z_9) - (Z_1 + 2 * Z_2 + Z_3)$$

$$G_y = (Z_3 + 2 * Z_6 + Z_9) - (Z_1 + 2 * Z_4 + Z_7)$$

Magnitude of vector ∇f can be defined as

$$mag(\nabla f) = \sqrt{{G_x}^2 + {G_y}^2}$$

For faster computation Equation (4) is approximated as

$$mag(\nabla f) \approx |G_x| + |G_y|$$

Linear Filter

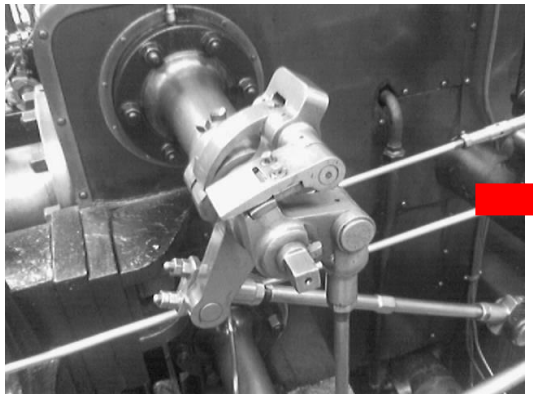
- **Sobel 필터**

```
Mat mySobelFilter(Mat srcImg) {
#if USE_OPENCV
    Mat dstImg(srcImg.size(), CV_8UC1);
    Mat sobelX, sobelY;

    Sobel(srcImg, sobelX, CV_8UC1, 1, 0); // 가로방향 Sobel
    Sobel(srcImg, sobelY, CV_8UC1, 0, 1); // 세로방향 Sobel
    dstImg = (abs(sobelX) + abs(sobelY))/2;
    // 두 에지 결과의 절대값 합 형태로 최종결과 도출

    return dstImg;
#else
    int kernelX[3][3] = { -1, 0, 1,
                          -2, 0, 2,
                          -1, 0, 1 }; // 가로방향 Sobel 마스크
    int kernelY[3][3] = { -1, -2, -1,
                           0,  0,  0,
                           1,  2,  1 }; // 세로방향 Sobel 마스크
    // 마스크 합이 0이 되므로 1로 정규화하는 과정은 필요 없음
    Mat dstImg(srcImg.size(), CV_8UC1);
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;
    int width = srcImg.cols;
    int height = srcImg.rows;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            dstData[y * width + x] = (abs(myKernelConv3x3(srcData, kernelX, x, y, width, height)) +
                                      abs(myKernelConv3x3(srcData, kernelY, x, y, width, height)))/2;
            // 두 에지 결과의 절대값 합 형태로 최종결과 도출
        }
    }

    return dstImg;
#endif
}
```

```
void cv::Sobel ( InputArray  src,
                 OutputArray dst,
                 int         ddepth,
                 int         dx,
                 int         dy,
                 int         ksize = 3 ,
                 double      scale = 1 ,
                 double      delta = 0 ,
                 int         borderType = BORDER_DEFAULT
               )
```

Linear Filter

- Sobel 필터



적용 전

Sobel X

Sobel Y

Sobel X + Y

Image Pyramid

- **Gaussian pyramid**
  - 영상에 대한 down sampling(또는 sub sampling)과 Gaussian 필터 기반 smoothing을 조합해 점차 작은 해상도의 영상을 반복적으로 생성
  - 위 과정으로 생성된 모든 영상을 모은 것은 Gaussian pyramid라고 지칭 (즉, 여러 scale의 영상을 가지고 있는 것)

* down sampling은 화소를 홀수 또는 짝수 칸만 남겨서 작은 영상을 생성하는 것

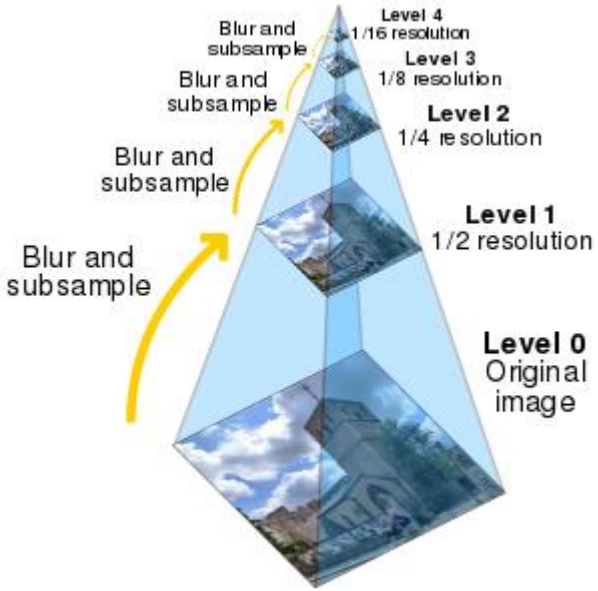


Image Pyramid

- **Gaussian pyramid**

```
Mat mySampling(Mat srcImg) {
    int width = srcImg.cols / 2;
    int height = srcImg.rows / 2;
    Mat dstImg(height, width, CV_8UC1);
    // 가로 세로가 입력 영상의 절반인 영상을 먼저 생성
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            dstData[y * width + x] = srcData[(y * 2) * (width * 2) + (x * 2)];
            // 2배 간격으로 인덱싱 해 큰 영상을 작은 영상에 대입할 수 있음
        }
    }

    return dstImg;
}
```

Down sampling의 구현

Image Pyramid

- **Gaussian pyramid**

```
vector<Mat> myGaussainPyramid(Mat srcImg) {
    vector<Mat> Vec; // 여러 영상을 모아서 저장하기 위해 STL의 vector컨테이너 사용

    Vec.push_back(srcImg);
    for (int i = 0; i < 4; i++) {
#if USE_OPENCV
        pyrDown(srcImg, srcImg, Size(srcImg.cols / 2, srcImg.rows / 2));
        // Down sampliong과 Gaussian filter가 포함된 OpenCV 함수
        // 영상의 크기가 가로, 세로 절반으로 줄어들도록 출력사이즈 지정
#else
        srcImg = mySampling(srcImg); // 앞서 구현한 down sampling
        srcImg = myGaussianFilter(srcImg); // 앞서 구현한 Gaussian filtering
#endif
        Vec.push_back(srcImg); // vector 컨테이너에 하나씩 처리결과를 삽입
    }

    return Vec;
}
```

Gaussian pyramid의 구현

Image Pyramid

- Gaussian pyramid

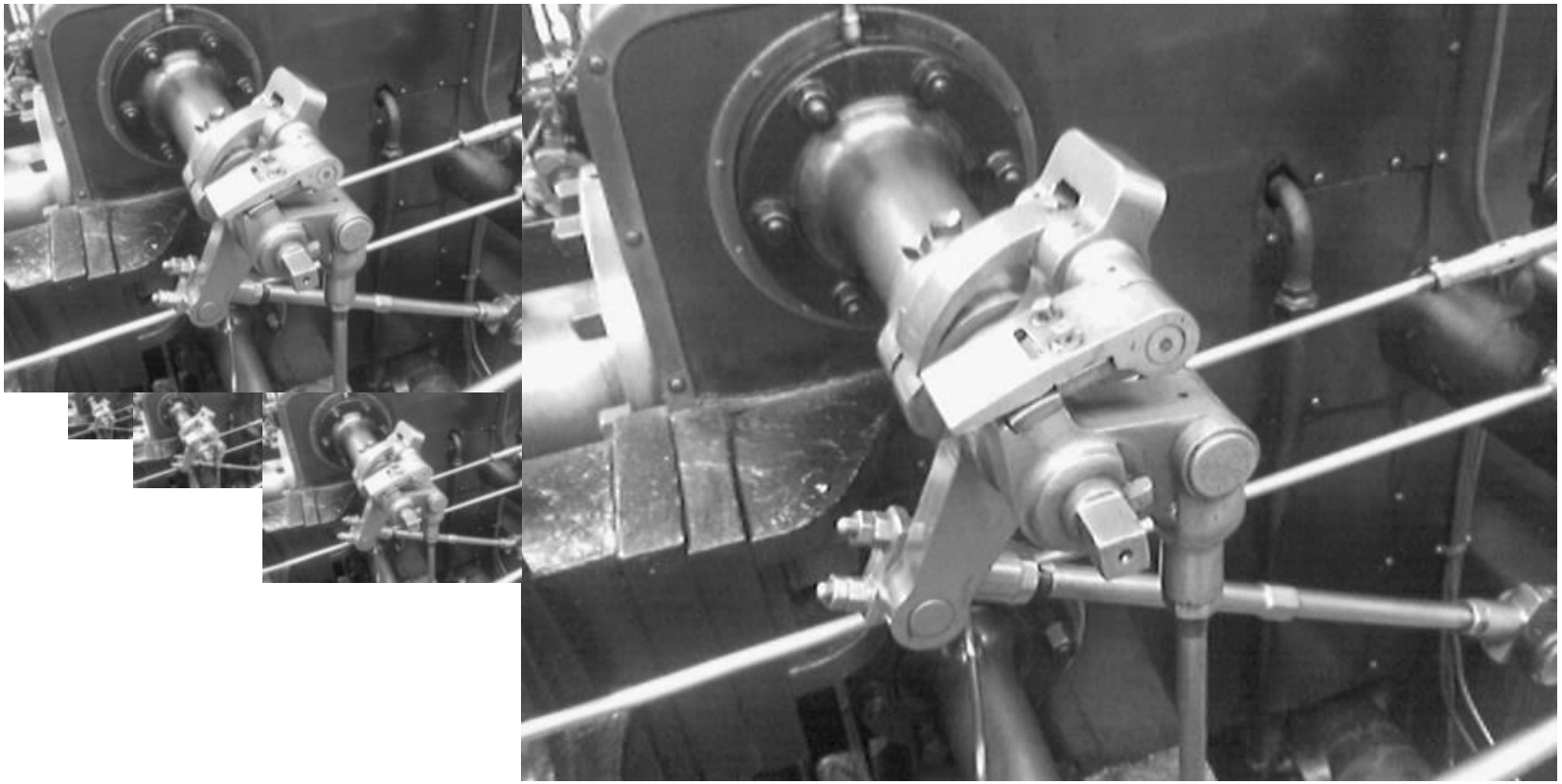


Image Pyramid

- **Laplacian pyramid**
  - 기본적으로 Gaussian pyramid와 유사하나 높은 해상도의 영상과 작이진 영상 간의 차 영상을 저장하는 방식
  - 가장 작은 크기의 영상은 차 영상이 아닌 작은 해상도의 영상이며 이를 up sampling하고 차 영상을 더하여 높은 해상도의 영상을 복원할 수 있음

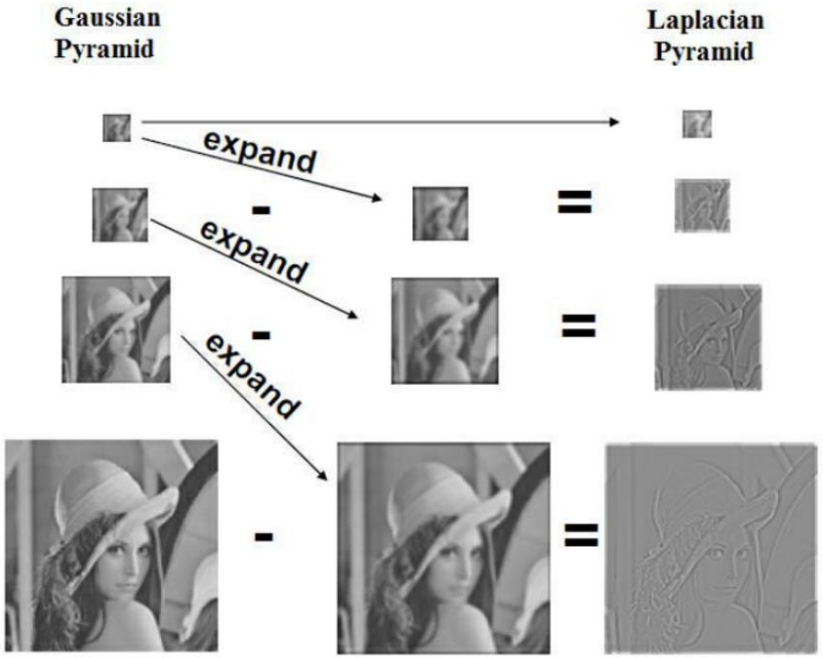


Image Pyramid

- **Laplacian pyramid**

```
vector<Mat> myLaplacianPyramid(Mat srcImg) {
    vector<Mat> Vec;

    for (int i = 0; i < 4; i++) {
        if (i != 3) {
            Mat highImg = srcImg; // 수행하기 이전 영상을 백업
#if USE_OPENCV
            pyrDown(srcImg, srcImg, Size(srcImg.cols / 2, srcImg.rows / 2));
#else
            srcImg = mySampling(srcImg);
            srcImg = myGaussianFilter(srcImg);
#endif
            Mat lowImg = srcImg;
            resize(lowImg, lowImg, highImg.size());
            // 작아진 영상을 백업한 영상의 크기로 확대
            Vec.push_back(highImg - lowImg + 128);
            // 차 영상을 컨테이너에 삽입
            // 128을 더해준 것은 차 영상에서 오버플로우를 방지하기 위함
        }
        else {
            Vec.push_back(srcImg);
        }
    }

    return Vec;
}
```

Laplacian pyramid의 구현

Image Pyramid

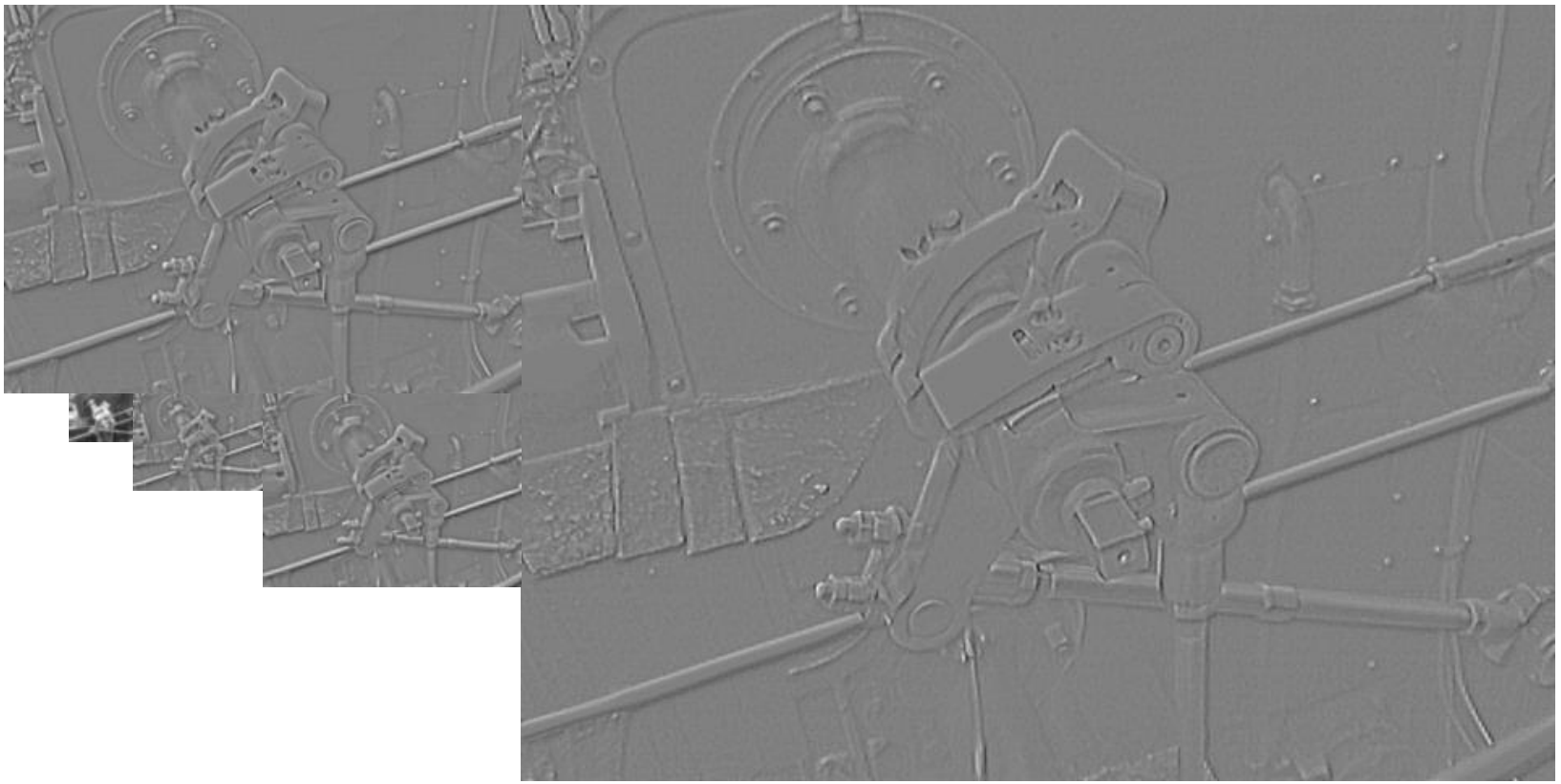
- **Laplacian pyramid**

```
// < EX5 >
src_img = imread("gear.jpg", 0);

vector<Mat> VecLap = myLaplacianPyramid(src_img);
// Laplacian pyramid 확보
reverse(VecLap.begin(), VecLap.end());
// 작은 영상부터 처리하기 위해 vector의 순서를 반대로

for (int i = 0; i < VecLap.size(); i++) {
    // Vector의 크기만큼 반복
    if (i == 0) {
        dst_img = VecLap[i];
        // 가장 작은 영상은 차 영상이 아니기 때문에 바로 불러옴
    }
    else {
        resize(dst_img, dst_img, VecLap[i].size());
        // 작은 영상을 확대
        dst_img = dst_img + VecLap[i] - 128;
        // 차 영상을 다시 더해 큰 영상을 복원
        // 오버플로우 방지용으로 더했던 128을 다시 빼줌
    }

    string fname = "ex5_lap_pyr" + to_string(i) + ".png";
    imwrite(fname, dst_img);
    imshow("EX5", dst_img);
    waitKey(0);
    destroyWindow("EX5");
}
```

Laplacian pyramid의 복원

Image Pyramid

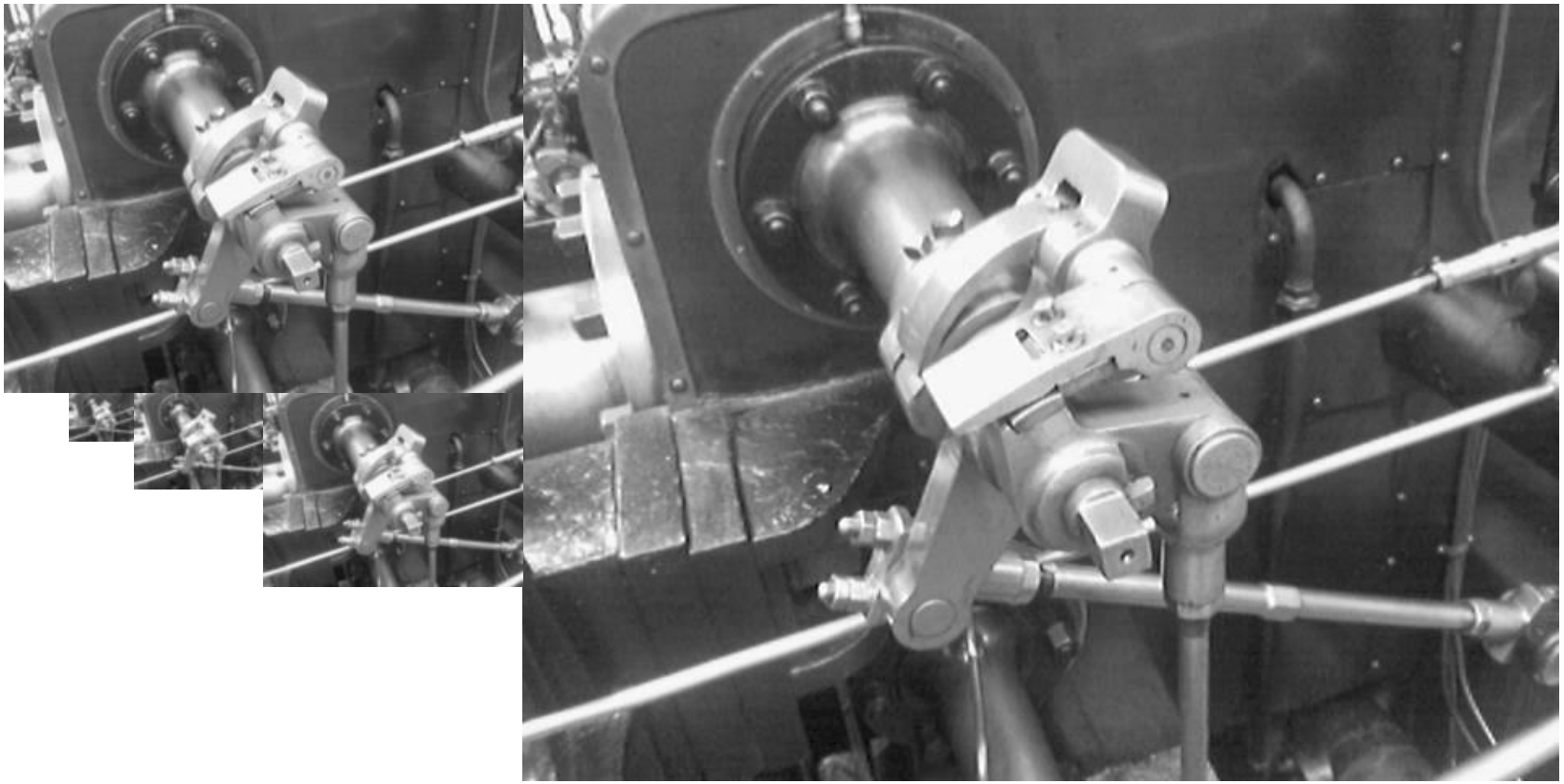
- Laplacian pyramid



Laplacian pyramid

Image Pyramid

- Laplacian pyramid



Laplacian pyramid의 복원

Homework

실습 및 과제

- 9x9 Gaussian filter를 구현하고 결과를 확인할 것
- 9x9 Gaussian filter를 적용했을 때 히스토그램이 어떻게 변하는지 확인할 것
- 영상에 Salt and pepper noise를 주고, 구현한 9x9 Gaussian filter를 적용해볼 것
- 45도와 135도의 대각 에지를 검출하는 Sobel filter를 구현하고 결과를 확인할 것
- 컬러영상에 대한 Gaussian pyramid를 구축하고 결과를 확인할 것
- 컬러영상에 대한 Laplacian pyramid를 구축하고 복원을 수행한 결과를 확인할 것

1. OpenCV를 사용하지 말 것
2. 주어진 영상(gear.jpg) 만을 사용할 것