Xilinx Zynq FPGA, TI DSP, MCU 기반의

프로그래밍 및 회로 설계 전문가 과정

강사 - Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 - 조윤정

yunreka@gmail.com

Feature Extraction - edge 편

(특징 추출)

특징 추출 이란, 영상에서 다른 부분들과 구별되는 성질을 추출하는 것

이렇게 추출된 특징들은 image processing 및 computer vision 의 다양한 주제에서 중요한 정보를 제공하는 역할을 한다. (단서를 제공한다 생각하면 될 것 같다)

특징 선택을 위해 고려해야 할 가장 중요한 요소로, **robustness** 가 있다.

이는 회전(rotation), 크기(size), 이동(translation)에 불변하는 특징을 선택해야 한다는 것이다. 즉, 환경에 따라 추출된 특징값들이 변화하는 것은 바람직한 feature 가 아니다.

==========

Edge Detection (엣지 검출)

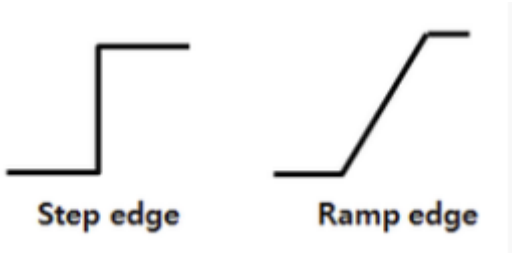
엣지는, 영상에서 밝기 변화가 날카로우며, 위치, 밝기, 기울기의 크기(magnitude), 최대 밝기 변화의 방향 등으로 표현이 되는 특징이다.

엣지는 일반적으로 영상의 밝기가 높은 곳에서 낮은 곳으로 변하거나, 또는 이와 반대로 낮은 곳에서 높은 곳으로 변하는 부분에 위치하는 화소들을 말한다. 즉, 엣지는 영상 내 객체의 경계(boundary) 또는 윤곽(contour)에 대한 정보를 가지고 있는데, 이러한 정보를 이용하여 우리는 영상 내 객체의 모양이나 방향을 탐지할 수 있는 정보를 얻을 수 있다.

엣지는 영상 내 객체의 넓이와 형태에 따라서, **ramp edge** 와 **step edge** 로 나뉜다.

ramp edge : 점진적으로 밝기값이 증가하거나 감소하는 엣지 (예: 5 5 4 3 2 1 0 0 ..)

step edge : 급격하게 밝기값이 증가하거나 감소하는 엣지 (예: 0 0 0 0 7 7 7 ..)



step edge는 밝기값의 변화가 급변하게 일어나기 때문에, 밝기값의 변화율이 크다.

(밝기값의 변화율 == 기울기 값 == gradient 라 한다)

반대로 ramp edge는 밝기값의 변화가 점진적으로 일어나기 때문에, 기울기 값이 작다. 여기서 이러한 **기울기 값을 구하는 것**이 바로 **1차 미분**이다(gradient).

그리고 이 **그레디언트의 크기**(magnitude)를 구하게 되면, 엣지를 검출할 수 있게 된다.

수학적인 의미의 1차 미분 연속함수의 개념이며, 이를 영상처리에 바로 적용하기 위해서는 영상의 데이터가 이산적인 관계로 바로 적용이 불가능하다. 따라서 **인접한 화소간의 차를 구하는 방법으로 미분을 근사화**시켜서 영상에서 1차 미분을 구하게 된다

(참고로 수학분야에서 1차 미분은 편미분 이나 차분 으로 불리운다)

1차 미분을 이용하여, 영상의 그레디언트 크기와 방향을 구하는 수식

그레디언트 크기:

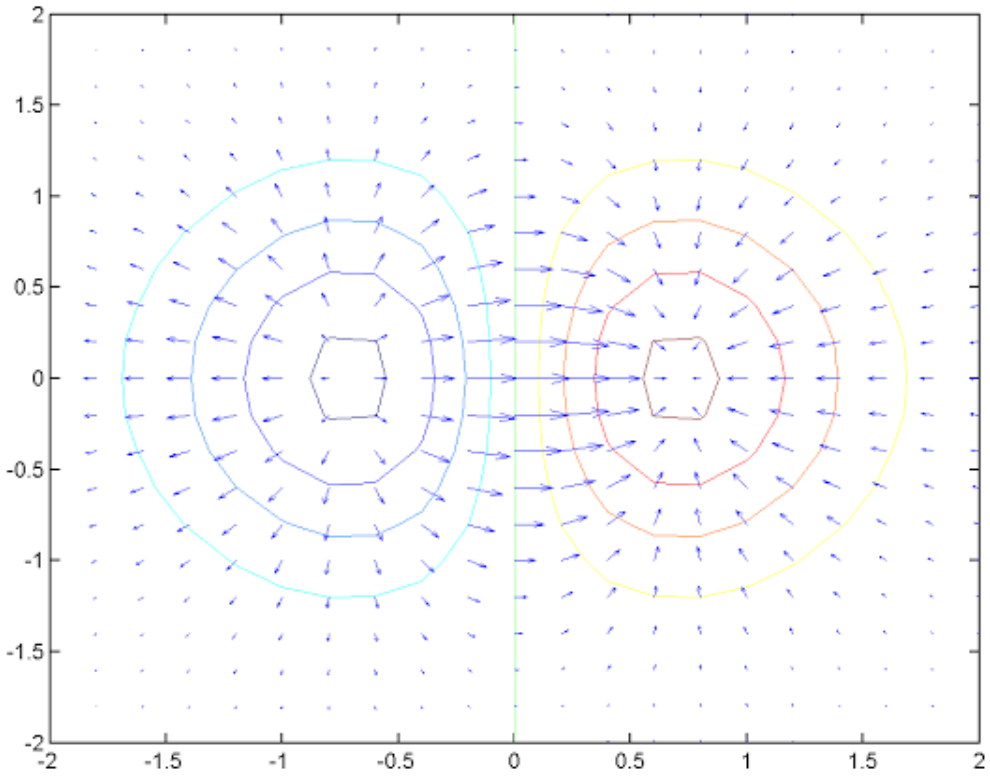
M = root(f(x+1, y) - f(x, y) * f(x+1, y) - f(x, y) + f(x, y+1) - f(x, y) * f(x, y+1) - f(x, y))

= | f(x+1, y) - f(x, y) | + | f(x, y+1) - f(x, y) |

그레디언트 방향: Q = tan의 -1 승 (f(x+1, y) - f(x, y) / f(x, y+1) - f(x, y))

위의 식에서 그레디언트의 크기는 엣지의 강도를 나타내며,

그레디언트의 방향은 엣지의 방향성을 나타낸다.

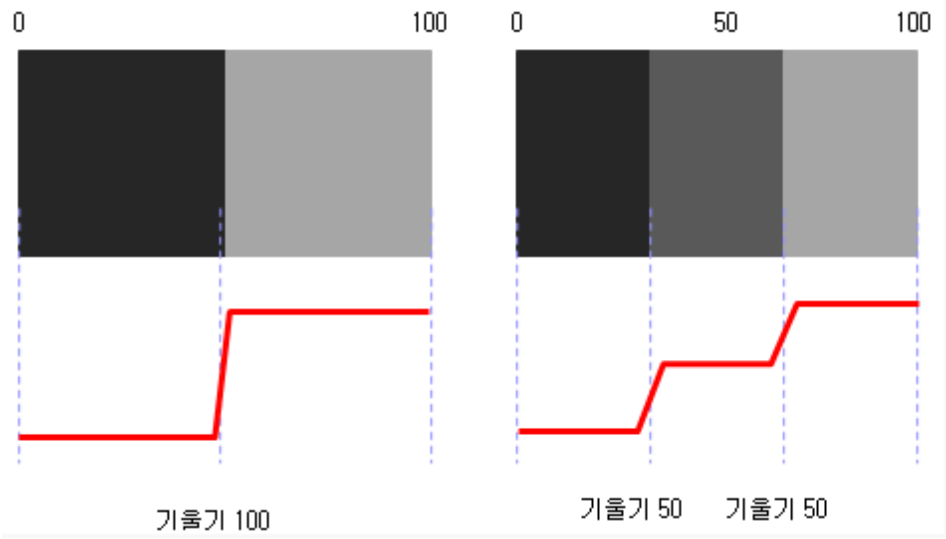


[그림 3] 2차원 공간에 투영한 기울기 특성

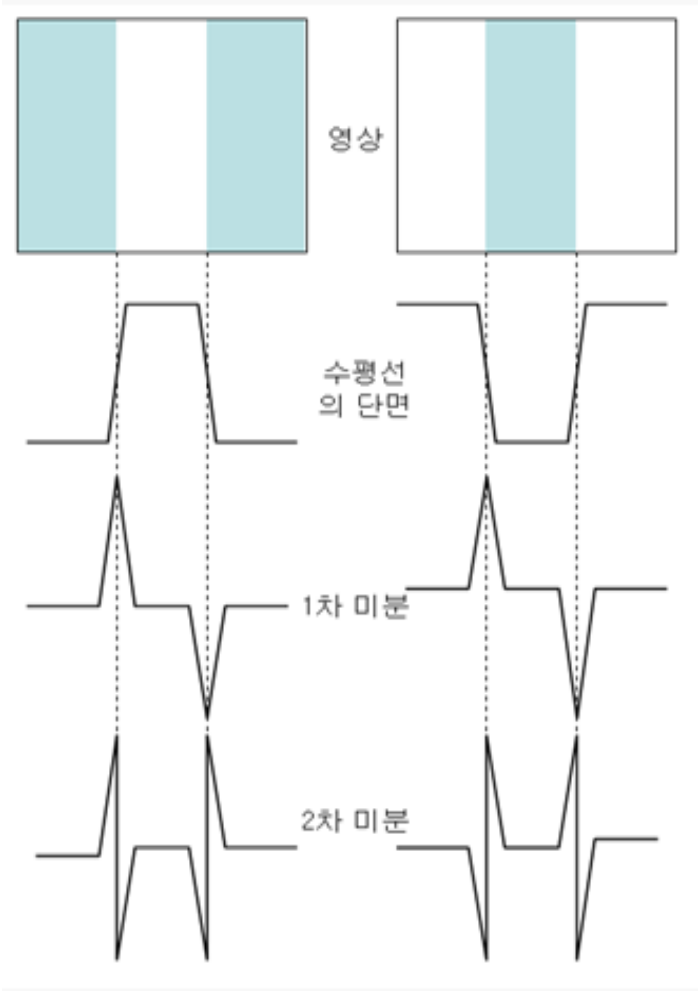
여기서 참고로 그레디언트의 크기를 구할 때 계산 시간을 줄이기 위해서 절대값을 이용하여 위의 간소화된 표현식으로도 구현할 수 있다는 것을 알아두자.

| f(x+1, y) - f(x, y) | 는 x 방향, 즉 수평방향으로의 1차 미분을 의미하며,

| f(x, y+1) - f(x, y) | 는 y 방향, 즉 수직방향으로의 1차 미분을 의미한다.



1차 미분을 이용하여 x 방향 또는 y 방향으로 엣지를 검출할 때는, 바로 옆의 인접한 화소와의 차를 구하는 방식으로 구현하는 것을 확인할 수 있다.



2차 미분을 이용하여, 영상의 그레디언트 크기와 방향을 구하는 수식

x 방향 2차 미분: f(x+1, y) + f(x-1, y) - 2*f(x, y)

y 방향 2차 미분: f(x, y+1) + f(x, y-1) - 2*f(x, y)

x 방향 + y 방향 2차 미분:

f(x+1, y) + f(x-1, y) - 2*f(x, y) + f(x, y+1) + f(x, y-1) - 2*f(x, y)

= f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4*f(x, y)

2차 미분을 이용하여 x 방향 또는 y 방향으로 엣지를 검출시

중심 화소를 기준으로 이전 화소와 다음 화소를 더한 값을 구한 후
중심 화소에 2를 곱한 값을 빼주는 방식으로 구현하는 것을 확인할 수 있다.

왜 2차 미분을 하는 것일까?

- 1차 미분은 수평, 수직, 대각 방향의 엣지에 반응성이 강한 반면에,
  2차 미분은 모든 방향의 엣지를 검출한다.
- 1차 미분의 결과는 일반적으로 두꺼운 엣지를 검출하는 반면에,
  2차 미분은 엣지 보다는 영상내에 있는 고립점(isolated point)이나
  가는 선(thin line)에 강한 반응성을 보인다.
- 2차 미분은 점진적으로 밝기값이 변화하는 부분에 대한 반응성을 약하게 하는 동시에, 엣지를 더욱 더 강조시킨다.
- 2차 미분은 폐곡선(곡선위의 한점이 한 방향으로 움직여 다시 출발점으로 돌아오는 형태의 곡선을 뜻한다) 형태로 엣지의 경계를 찾기 때문에, 폐곡선의 선명한 엣지를 검출하며, 날카로운 엣지를 검출한다.
  따라서 검출된 엣지는 영상분할(image segmentation)을 위한 알고리즘에 전처리 조건으로 적용할 수 있다.
- 2차 미분은 1차 미분에 비하여 잡음에 약하므로, 이를 보완하기 위해서 LoG(Laplacian of Gaussian)이나 DoG(Difference of Gaussian) 와 같은 리플라시안 응용 기법을 사용하기도 한다.

잡음에 약하다고 하면, 잡음을 제거하는 알고리즘을 적용시킨 후 엣지를 검출하는 알고리즘을 적용하면, 잡음이 제거되면서 엣지가 검출되지 않을까?
이러한 원리에 착안하여 나온 알고리즘이 바로 위에서 소개한 **LoG**(Laplacian of Gaussian)이나 **DoG**(Difference of Gaussian)이다. 또한 잡음 제거뿐 아니라, 보다 정확한 엣지 화소를 찾기 위해서 수학적인 방법에 근거하여 접근하는 **캐니(Canny) 엣지** 검출 알고리즘이 있다.

그럼 각각의 방법에 대해 하나하나 살펴보자.

1) LoG(Laplacian of Gaussian) : 이름대로.. 잡음 제거를 위해 가우시안 필터링을 적용한 후, 엣지 검출은 라플라시안 계열의 2차 미분 기법을 이용하는 방법이다. 중요한 것은 LoG는 엣지 검출 이전에 가우시안 필터링을 적용하기 때문에,
표준편차 값에 따라서 결과가 달라질 수 있다. (가우시안필터링의 단점임)
즉, 표준편차의 값에 따라 **마스크의 크기**도 달라지며, **추출된 엣지**의 **종류**와 **모양** 역시 다양하게 된다.
잡음을 많이 섞인 영상에 대하여 **표준편차 값을 크게 설정**하면, **스무딩 현상**이 증가하게 되므로,
뚜렷한 엣지만을 검출하는 반면에
표준편차값을 작게 설정하면, **상대적으로 적은 스무딩 현상**이 발생하므로 **세밀**한 부분의 엣지까지도 검출하게 된다.

* 단점: 가우시안 필터링 -> LoG(라플라시안 엣지검출)을 이용할 경우 표준편차 값에 따른 마스크의 크기 설정 문제로 가우시안 필터링을 적용하고 라플라시안 필터링을 하는데 있어서 시간이 많이 걸리며, LoG 공식을 직접 적용하더라도 LoG 공식에 의해 마스크를 생성하기 때문에 역시 많은 시간이 소요된다. 이러한 단점을 보완하기 위해서 나온 알고리즘이 바로 DoG 이다.

2) DoG(Difference of Gaussian) : DoG 는 LoG 와 마찬가지로 가우시안 함수식에 기반을 두고 있기 때문에 LoG와 유사한 알고리즘의 성격을 가진다. DoG는 표준편차의 크기에 따른 **가우시안 필터링 결과의 차**로서 계산된다.
DoG의 가장 큰 장점은 계산 시간을 줄이면서, 각 표준편차의 값을 조절함으로써 LoG에 비해서 다른 여러가지 결과를 얻을 수 있다는 점이다.
DoG 를 가장 간단하게 구하는 방법은, 원 영상에 대해서 가우시안 필터링 1을 적용한 영상과 원 영상에 대해서 가우시안 필터링 2를 적용한 영상에 대해서
서로 뺄셈 연산을 해주면 간단하게 구현할 수 있다. 여기서 가우시안 필터링 1과 2를 각각 적용하는 이유는 **서로 다른 표준편차값을 설정**함으로 인해 **각 표준편차의 값을 조절**할 수 있도록 하기 위해서다
.

3) Canny : 위의 두 알고리즘은 일반적으로 잡음에 약하기 때문에, 작은 잡음이라도 이를 엣지로 간주하여 검출하는 단점이 생긴다. Canny edge 검출 알고리즘은, 이러한 단점들을 보완하기 위해 고안되었다.
Canny edge 검출 알고리즘에서 사용되는 엣지 모델은 가우시안 잡음에 의해서 훼손된 **스텝 엣지**

모델(step edge)을 사용한다.

(영상처리에 있어서 가장 많이 사용되는 잡음 모델은 가우시안 잡음 모델이다)

즉, 엣지는 스텝 엣지(step edge)를 가정하며,

잡음은 가우시안 잡음을 가정하며,

Canny edge 검출 알고리즘의 목적은 잡음을 최대한 줄이면서, 보다 정확한 엣지를 검출하기 위함이다.

이러한 엣지를 검출하기 위해서는 다음과 같은 3 가지의 조건을 만족해야 한다.

- 탐지성(Detection) : 엣지 검출 알고리즘은 모든 **실제 엣지(true edge) 화소**만을 찾아야 하며, 어떠한 거짓 엣지(false edge) 화소라도 찾아서는 안된다.
- 국부성(Localization) : 엣지는 올바른 공간상에서만 발견되어야 한다.
  즉, 실제 엣지와 발견된 엣지와의 차이는 **최소화** 되어야 한다.
- 단일 응답성(Single Response) : 단일한 엣지에 대해서 발견된 엣지는 여러 엣지를 가져서는 안된다. 즉, 각 엣지에 대해서 **단일한 응답**만을 가져야 한다.

이러한 조건을 충족하기 위한 Canny edge 검출 알고리즘은 일반적으로
다음의 4 가지로 구성되는 set of algorithms 의 형태를 가지게 된다.

- 잡음 제거를 위해서 **가우시안 필터를 적용**한다.
- **소벨**이나 **프리윗 엣지 검출**기 등과 같이 이와 유사한 1 차 미분 기반의 공식을 이용하여 **그레디언트의 크기와 방향**을 찾는다.
- 가는(Thinned) 엣지를 얻기 위해서
  NMS(Non Maximum Suppression, 비최대치 억제)을 적용한다.
- 최종적으로 엣지를 검출하기 위해서 **두개의 임계치**를 적용한다.
  (이러한 방법은 hysteresis threshold, 이력 임계치 방법이라고 불림)

이러한, 이중 임계값을 이용하여 거짓 엣지를 제거해야 하는데,
임계치를 낮게 설정하면, 엣지 화소가 아닌 화소가 남아있을 가능성이 있으며,
반대로 임계치를 높게 설정하면, 실제 엣지에 해당하는 화소의 일부를 검출하지 못할 가능성이 있다.

Canny edge 검출을 위해 조정이 필요한, 3가지 파라미터

1. 가우시안 필터링의 스무딩 정도를 결정하는 **시그마값**(표준편차)
2. NMS 및 hysterisis 를 위한 **tlow**(하한 임계치) 의 적절한 설정
3. NMS 및 hysterisis 를 위한 **thigh**(상한 임계치) 의 적절한 설정

소스코드

라인 엣지 검출> 원하는 방향의 엣지 검출 구현

```
void DibEdgeLineDegree0(CDib &dib)
{
  register int i, j;

  // method 1: 0도방향, method 2: 45도방향, method3: 90도방향, method4: 135도방향
  int method = 1;

  int temp;
  int w = dib.GetWidth();
  int h = dib.GetHeight();
  CDib cpy = dib;
  idb.CreateGrayImage(w, h, 0);
  BYTE** ptr1 = dib.GetPtr();
  BYTE** ptr2 = cpy.GetPtr();
  for(j=1; j<h-1; j++){
    for(i=1; i<w-1; i++){
    //0도 방향 검출
      if(method == 1)
      {
        temp = -1*ptr2[j-1][i-1]  +  -1*ptr2[j-1][i]  +  -1*ptr2[j-1][i+1] +
                  2*ptr2[j][i-1]     +   2*ptr2[j][i]      +   2*ptr[j][i+1]      +
                 -1*ptr2[j+1][i-1]  + -1*ptr2[j+1][i]  +  -1*ptr2[j+1][i+1] ;
      }
    //45도 방향 검출
      else if(method == 2)
      {
        temp = -1*ptr2[j-1][i-1]  +  -1*ptr2[j-1][i]  +   2*ptr2[j-1][i+1]  +
```

```
                -1*ptr2[j][i-1]    +   2*ptr2[j][i]    +   -1*ptr[j][i+1]     +
                 2*ptr2[j+1][i-1] +  -1*ptr2[j+1][i] +  -1*ptr2[j+1][i+1] ;
    }
   //90도 방향 검출
    else if(method == 3)
    {
      temp = -1*ptr2[j-1][i-1]  +   2*ptr2[j-1][i]  +   -1*ptr2[j-1][i+1]  +
                -1*ptr2[j][i-1]    +   2*ptr2[j][i]    +   -1*ptr[j][i+1]     +
                -1*ptr2[j+1][i-1] +   2*ptr2[j+1][i] +  -1*ptr2[j+1][i+1] ;
    }
   //90도 방향 검출
    else if(method == 4)
    {
      temp =  2*ptr2[j-1][i-1]  +  -1*ptr2[j-1][i]  +   -1*ptr2[j-1][i+1]  +
                -1*ptr2[j][i-1]    +   2*ptr2[j][i]    +   -1*ptr[j][i+1]     +
                -1*ptr2[j+1][i-1] +  -1*ptr2[j+1][i] +   2*ptr2[j+1][i+1] ;
    }
    ptr1[j][i] = (BYTE)limit(temp);
   }
  }
}
```

1차 미분> 영역 필터 함수 구현

```
void quick_sorting(BYTE *data, int left, int right);
void DibEdgeRangeFilter(CDib &dib)
{
  register int i, j, m, n, cnt;
  int mask_h = 3;
  int mask_w = 3;
  BYTE max_val, min_val, dif_val;
  int h = dib. GetHeight();
```

```
  int w = dib. GetWdith();
  CDib cpy = dib;
  dib.CreateGrayImage(w, h, 0);
  BYTE** ptr1 = dib.GetPtr();
  BYTE** ptr2 = cpy.GetPtr();
  BYTE* mask = (BYTE *)calloc(mask_h * mask_w, sizeof(BYTE));
  for(j=0; j<h-mask_h; j++){
    for(i=0; i<w-mask_w; i++){
      cnt = -a;
      for(m=0; m<mask_h; m++){
        for(n=0; n<mask_w; n++){
          mask[++cnt] = ptr2[m+j][n+i];
        }
      }
      quick_sorting(mask, 0, cnt);

      max_val = mask[cnt];
      min_val = mask[0];

      //최대값과 최소값의 차이를 구함
      dif_val = max_val - min_val;
      ptr1[j][i] = (BYTE)limit(dif_val);
    }
  }
}

// 퀵 정렬
void quick_sorting(BYTE *data, int left, int right)
{
  int i, j;
  BYTE x, y;
  i = left;
  j = right;
```

```
  x = data[ (left+right)/2 ];

  do
  {
    while(data[i]<x && i<right) i++;
    while(x<data[j] && j>left) j--;
    if(i<=j)
    {
      y = data[i];
      data[i] = data[j];
      data[j] = y;
      i++;
      j++;
    }
  }while(i<=j);
  if(left<j)  quick_sorting(data, left, j);
  if(i<right)  quick_sorting(data, i, right);
}
```

2차미분> 라플라시안 오퍼레이터를 활용한 에지 검출 소스코드

```
private void myEdge(Bitmap src, int amp)
{
        //Bitmap은 회색조(Gray)로 영상을 바꾼 소스를 얻어 내며,
        //amp는 사용자에게 입력받은 값으로 출력 레벨을 결정하는데 사용되는 수이다.
        int i, j, iColorValue;

        // 라프라시안 필터
        int[] iFilter = new int[] { -1, -1, -1, -1, 8, -1, -1, -1, -1 };
        int[,] iArrayValue = new int[src.Width, src.Height];
```

```
        Color[] cArrayColor = new Color[9]; // 색정보의 배열 중간점을 기준으로
//라프라시안 필터링 적용할 픽셀들

        Color color;
 // 화상에 대한 필터 처리
        // 각각 너비와 길이에 대하여 -1을 하는 이유는 맨 마지막 pixel을
// 기준으로 잡을 수 없기 때문
        for (i = 1; i < src.Width - 1; i++)
            for (j = 1; j < src.Height - 1; j++)
            {
                cArrayColor[0] = src.GetPixel(i - 1, j - 1);
                cArrayColor[1] = src.GetPixel(i - 1, j);
                cArrayColor[2] = src.GetPixel(i - 1, j + 1);
                cArrayColor[3] = src.GetPixel(i, j - 1);
                cArrayColor[4] = src.GetPixel(i, j);
                cArrayColor[5] = src.GetPixel(i, j + 1);
                cArrayColor[6] = src.GetPixel(i + 1, j - 1);
                cArrayColor[7] = src.GetPixel(i + 1, j);
                cArrayColor[8] = src.GetPixel(i + 1, j + 1);

                // 필터 처리
                iColorValue = iFilter[0] * cArrayColor[0].R + iFilter[1] * cArrayColor[1].R
+
iFilter[2] * cArrayColor[2].R + iFilter[3] *
cArrayColor[3].R + iFilter[4] * cArrayColor[4].R +
iFilter[5] * cArrayColor[5].R + iFilter[6] *
cArrayColor[6].R + iFilter[7] * cArrayColor[7].R +
iFilter[8] * cArrayColor[8].R;

//출력 레벨에 따라서 각기 다른 결과물이 나올 수 있다.
                iColorValue = amp * iColorValue;   // 출력 레벨의 설정
                // iColorValue가 0보다 작은 경우
```

```
                if (iColorValue < 0)
                    iColorValue = -iColorValue;     // 정의값에 변환
                // iColorValue가255보다 클 경우
                if (iColorValue > 255)
                    iColorValue = 255;     // iColorValue를 255으로 설정
                iArrayValue[i, j] = iColorValue;      // iColorValue의 설정
            }
        // 필터 처리 결과 출력

        for (i = 1; i < src.Width - 1; i++)
            for (j = 1; j < src.Height - 1; j++)
            {
                color = Color.FromArgb(iArrayValue[i, j], iArrayValue[i, j],
                        iArrayValue[i, j]);
                          // iArrayValue 값에 의한 색 설정
                src.SetPixel(i, j, color);   // 픽셀의 색 설정
            }
        //pictureBox1.Image = bBitmap;   // 변경 결과 출력
        edgePic.Image = src;
}
```

reference

- https://m.blog.naver.com/neverabandon/100054546233
- https://ghebook.blogspot.kr/2010/07/gradient.html
- https://m.blog.naver.com/PostView.nhn?blogId=nersion&logNo=140142324117&categoryNo=23&proxyReferer=https%3A%2F%2Fwww.google.co.kr%2F
- http://blog.naver.com/PostView.nhn?blogId=neverabandon&logNo=100054485092&redirect=Dlog&widgetTypeCall=true