2. Geometric Transformation(Process)

: Geometric Transformation은 주변 pixel value와 관련없이 pixel 하나 하나를 변형하는 기법인 'point transformation(process)'와 달리, **화소의 값을 바꾸지 않고 pixel의 위치를 바꾸는 기법**입니다. 영상을 상하좌우로 바꾸거나, 회전하거나, 이미지의 일부분을 뜯어내거나, 확대/축소하는 방법들이 Geometric transformation에 해당합니다.

Mapping(사상)

- Mapping이란 입력 영상과 출력 영상 사이의 관계를 뜻합니다. Mapping에는 입력 영상의 화소 위치를 이용해 출력 영상의 새로운 화소 위치를 계산하는 '**전방향 사상**'과, 출력 영상의 화소 위치를 이용해 역변환 방법으로 입력 영상의 화소 위치를 계산하는 '**역방향 사상**'이 있습니다.
- '전방향 사상' 방법으로 geometric transformation을 수행할 경우, 서로 다른 입력 화소들이 같은 출력 화소에 사상 되는 '**overlap**'과 입력 영상에서의 화소가 출력 화소에 사상 되지 않는 '**hole**'의 문제점이 발생할 수 있습니다. 따라서 geometric transformation을 수행할 때 '전방향 사상'보다 '역방향 사상'을 주로 이용합니다.
- Geometric transformation을 수행할 때 '화소의 위치가 다른 곳으로 옮겨진다'는 점에서 보간법(interpolation)이 필수적으로 수행되어야 합니다. 보간법이란 화소와 화소 사이에 mapping관계가 이루어졌을 때, 화소와 화소 사이에 있는 새로운 화소를 생성하는 방법입니다.

Interpolation(보간법)

- 보간법에는 크게 3가지 종류가 있습니다.
    - Nearest neighbor interpolation
    - Bilinear interpolation
    - Higher order interpolations

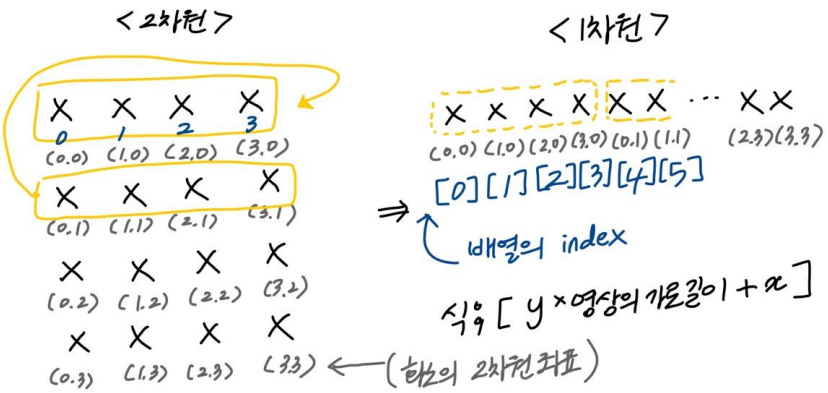
실습 1) 보간법(Interpolation)

(1) Nearest neighbor interpolation

- Nearest neighbor interpolation은 출력 화소에서 가장 가까이 있는 화소값을 그대로 이용하는 기법입니다. '반올림'을 통해서 가장 가까운 화소의 위치를 반환할 수 있습니다.

```
UChar NearesetNeighbor(UChar* Data, Double srcX, Double srcY, Int Stride) // Stride : WIDTH
{
    // 반올림을 통해서 가장 가까운 화소의 위치를 반환
    return Data[((int)(srcY + 0.5) * Stride + (int)(srcX + 0.5))];
}
```

➔ Double srcX와 Double srcY는 역방향 매핑된 화소값입니다. 실수형인 srcX와 srcY를 0.5를 더하고 정수형으로 바꿈으로써 반올림을 수행할 수 있습니다. 또한 2차원 좌표인 영상을 1차원 좌표로 바꾸기 위해 영상의 가로길이인 Stride를 y좌표에 곱하고 x좌표를 더합니다.



➔

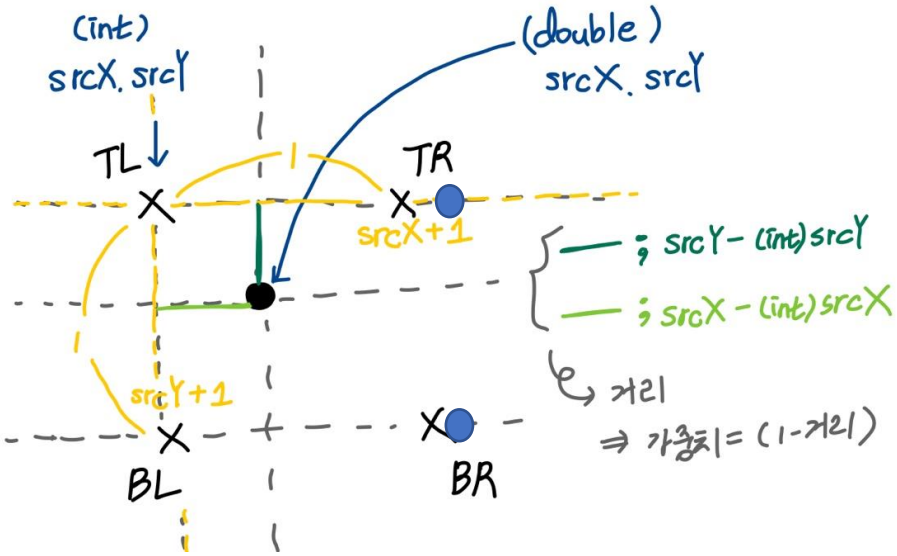
- Nearest neighbor interpolation은 간단하고 빠른 접근 방식이지만, 하나의 화소가 여러 번 사용되기 때문에 visual blockiness 현상이 발생하여 화질이 나쁘다는 단점이 있습니다.

(2) Bilinear interpolation

- Bilinear interpolation은 역매핑된 화소의 근접한 4개의 pixel들에 가중치를 곱한 값들의 합을 사용하는 기법입니다.

```
UChar Bilinear(UChar* Data, Double srcX, Double srcY, Int Stride)
{
    int SrcX_Plus1, SrcY_Plus1; // SrcX_Plus1 : srcX의 오른쪽, SrcY_Plus1 : srcY의 아래
    double Hor_Wei, Ver_Wei; //Horizontal Weight, Vertical Weight
    int TL, TR, BL, BR; //각 화소 위치

    // 역매핑된 위치에서 왼쪽 위에 있는 정수 화소 위치가 기준
    SrcX_Plus1 = CLIPPIC_HOR(srcX+1); //
    SrcY_Plus1 = CLIPPIC_VER(srcY+1); //

    // 가중치 계산할 때 쓰이는 거리
    Hor_Wei = srcX - (int)srcX;
    Ver_Wei = srcY - (int)srcY;

    // 각 화소 위치를 정확하게 원본 영상 내의 어느위치인지 알기 위해서
    // 현재 SrcX, SrcY는 단순히 역매핑된 위치에서 왼쪽 위에, 왼쪽 아래, 오른쪽 위에, 오른쪽 아래에 있는 화소임
    TL = (int)srcY *Stride + (int)srcX; //
    TR = (int)srcY *Stride + SrcX_Plus1; //
    BL =  SrcY_Plus1 *Stride + (int)srcX; //
    BR =  SrcY_Plus1 *Stride + SrcX_Plus1; //

    // TMP를 구할 때 반올림을 해야함
    UChar TMP =
        (1 - Ver_Wei) * (((1-Hor_Wei) *Data[TL]) + (Hor_Wei *Data[TR])) + /*위쪽*/

        Ver_Wei * (((1-Hor_Wei) *Data[BL]) + (Hor_Wei *Data[BR])) + 0.5; /* 아래쪽*/

    return TMP;
}
```

➔ 실수 srcX와 srcY는 역방향으로 매핑된 화소의 위치입니다. srcX와 srcY를 정수형으로 형 변환하게 되면 역매핑된 위치의 가장 가까운 왼쪽 위에 있는 화소의 위치가 됩니다. Bilinear interpolation은 근접한 4개의 화소를 이용하기 때문에 정수 srcX와 srcY를 이용하여 왼쪽 위, 오른쪽 위, 왼쪽 아래, 오른쪽 아래에 있는 화소의 위치를 알 수 있습니다.



➔

➔ 따라서 TL의 화소의 위치는 ((int)srcX, (int)srcY)이며, 이를 1차원 좌표로 나타내면 위의 TL식과 같습니다. TR의 화소 위치는 ((int)srcX+1, (int)srcY)이고, BL의 화소 위치는 ((int)srcX, (int)srcY+1)이며, BR의 화소 위치는 ((int)srcX+1, (int)srcY+1)입니다. 이를 1차원 좌표로 나타내면 위 코드 식과 같습니다.

➔ 주변의 4개 화소의 위치를 구한 후, 가중치를 곱하고 더함으로써 interpolation 해야 합니다. 가까운 화소의 가중치가 더 커야 하기 때문에 각 화소의 가중치는 거리에 반비례합니다. 위의 코드에서 Hor_Wei는 화소의 수평축의 거리(연두색 선), Ver_Wei는 화소의 수직축 거리(초록색 선)입니다. 먼저 수평축부터 가중치를 곱하고 더한 후, 수직축에 가중치를 곱하고 더함으로써 interpolation되는 화소의 위치를 알 수 있습니다. 위쪽 화소에서는 TL이 TR보다 화소의 위치에 더 가깝기 때문에 가중치가 1-Hor_Wei이며, TR의 가중치는 Hor_Wei입니다. 아래쪽 화소에서도 BL이 BR보다 화소의 위치에 더 가깝기 때문에 가중치가 1-Hor_Wei이며, TR의 가중치는 Hor_Wei입니다. 이를 각각 곱해준 후 더하면 파란색 원 위치가 됩니다. 파란색 원에서 검정색 원으로 보간하기 위해서는 각 가중치를 곱한 후 더

해야 합니다. 이 때 수직축의 거리인 Ver_Wei를 사용하고, 더욱 가까운 위쪽의 파란색 원의 가중치는 1-Ver_Wei가 되며, 아래쪽의 파란색 원의 가중치는 Ver_Wei가 됩니다.

➔ 이렇게 보간한 화소값은 정수이기 때문에 반올림과 clipping과정을 거쳐야 합니다.

- Bilinear interpolation은 계산양은 Nearest neighbor interpolation보다 많지만, 화질은 훨씬 부드럽기 때문에 조금 더 정교한 접근이라고 할 수 있습니다.

(3) B-spline interpolation

- B-spline interpolation은 Higher order interpolation 중 하나의 기법입니다. B-spline interpolation은 역매핑된 화소의 근접한 16개 화소들을 이용하는 방법입니다. Bilinear interpolation과 비슷하게 16개의 화소 중 가로를 먼저 보간 하고, 보간 한 위치를 세로로 보간하여 interpolation 되어야 하는 화소값을 구합니다. (4)인 Cubic interpolation과는 보간 필터의 계수 유도 방정식만 다르고, 나머지는 동일합니다.
- B-spline 보간법의 보간 필터 계수 유도 방정식은 다음과 같습니다.

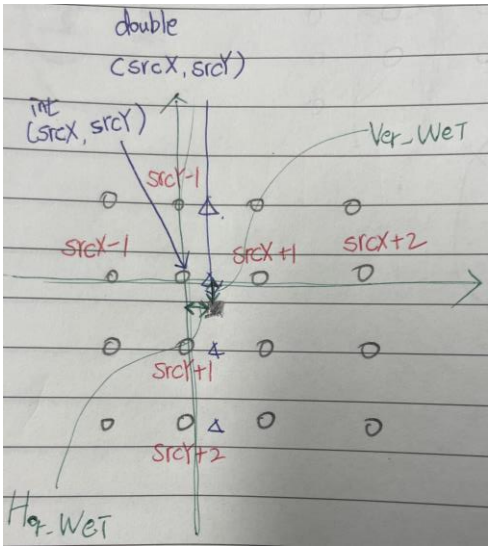
$$f(x) = \begin{cases} \frac{1}{2}|X|^3 - |X|^2 + \frac{2}{3}, & 0 \le |X| < 1 \\ -\frac{1}{6}|X|^3 + |X|^2 - 2|X| + \frac{4}{3}, & 1 \le |X| < 2 \\ 0, & 2 \le |X| \end{cases}$$

이를 코드로 나타내면 다음과 같습니다.

```
// B-Spline
double BSpline_function(Double x) {

    double result;

    if (fabs(x) >= 0 && fabs(x) < 1) {
        result = (1.0/2.0 * (pow(fabs(x), 3.0))) + (-1.0 * pow(fabs(x), 2.0)) + (2.0/3.0);
    }
    else if (fabs(x) >= 1 && fabs(x) < 2) {
        result = (-1.0 * 1.0/6.0 * pow(fabs(x), 3.0)) + pow(fabs(x), 2.0) + (-1.0*2.0*fabs(x)) + 4.0/3.0;
    }
    else
        result = 0;

    return result;
}
```

➔ pow(,)는 실수를 제곱하는 함수이며, fabs()는 실수의 절댓값을 구하는 함수입니다. 실수형 연산을 위해 모든 수를 실수형으로 작성하였습니다.

- 역매핑된 화소 주변의 16개 점을 나타내기 위한 그림은 다음과 같습니다.



- 주변 16개의 화소의 위치를 더욱 간결하게 구하기 위해 정수형 srcX와 srcY에 각각 -1, 1, 2를 더하여 변수에 저장합니다. 그 후 가중치를 구하기 위해 화소와의 거리를 구하여 변수에 저장합니다.

```
UChar B_Spline(UChar* Data, Double srcX, Double srcY, Int Stride)
{
    int Src_X_Minus_1, Src_X_Plus_1, Src_X_Plus_2;
    int Src_Y_Minus_1, Src_Y_Plus_1, Src_Y_Plus_2;
    double Hor_Wei, Ver_Wei; //Horizontal Weight, Vertical Weight
    double TMP_Hor[4] = { 0,0,0,0 };
    double TMP = 0;

    // 위치 화소 좌표
    // 어느 화소를 기준으로 주변 화소를 구할 것인지
    Src_X_Plus_1 = CLIPPIC_HOR((int)srcX+1);
    Src_X_Plus_2 = CLIPPIC_HOR((int)srcX+2);
    Src_Y_Plus_1 = CLIPPIC_VER((int)srcY+1);
    Src_Y_Plus_2 = CLIPPIC_VER((int)srcY+2);

    Src_X_Minus_1 = CLIPPIC_HOR((int)srcX-1);
    Src_Y_Minus_1 = CLIPPIC_VER((int)srcY-1);

    Hor_Wei = srcX - (int)srcX;
    Ver_Wei = srcY - (int)srcY;


    int X_Pix[] = { Src_X_Minus_1, (int)srcX, Src_X_Plus_1, Src_X_Plus_2 };
    int Y_Pix[] = { Src_Y_Minus_1, (int)srcY, Src_Y_Plus_1, Src_Y_Plus_2 };
    double Distance_Hor[] = { Hor_Wei + 1, Hor_Wei, 1 - Hor_Wei, (1 - Hor_Wei) + 1 };
    double Distance_Ver[] = { Ver_Wei + 1, Ver_Wei, 1 - Ver_Wei, (1 - Ver_Wei) + 1 };

    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            TMP_Hor[i] += BSpline_function(Distance_Hor[j]) * Data[Y_Pix[i] * Stride + X_Pix[j]];
        }
    }

    for (int i = 0; i < 4; i++)
    {
        TMP += BSpline_function(Distance_Ver[i]) * TMP_Hor[i];
    }

    // TMP는 화소값 - clipping & 반올림 과정
    TMP += 0.5;
    TMP = TMP > maxVal ? maxVal : TMP < minVal ? minVal : TMP;

    return (UChar)TMP;
}
```

➔ 위에서 구한 화소 위치 변수들을 X_pix배열과 Y_pix배열에 저장하고, 해당하는 거리 또한 Distance_Hor배열과 Distance_Ver 배열에 저장합니다.

➔ i를 이용한 반복문은 y좌표를 뜻하고, j를 이용한 반복문은 x좌표를 뜻합니다. 이중 for문에서는 4개의 수평축에 대한 화소 위치를 보간하고(사진에서의 파란색 삼각형), 아래의 단일 for문에서는 보간한 4점을 수직축으로 보간합니다. B-spline 보간법에서는 정수 화소의 위치 간의 거리에 따라 보간 필터 계수의 유도방정식이 달라지므로, 보간할 때 거리에 보간필터계수의 유도방정식 함수를 적용해야 합니다.

➔ 보간한 화소값 또한 정수이기 때문에 clipping하고 반올림하는 과정이 필요합니다. 반올림을 하기 위해 0.5를 더하고, 화소의 최대값보다 크면 최대값으로 화소의 최소값보다 작으면 최소값으로 대응해주는 clipping과정을 더했습니다. TMP > maxVal ? maxVal : TMP 코드는 TMP > maxVal이 참이면 maxVal로, 거짓이면 TMP라는 의미를 가지고, 이어서 등장하는 TMP < minVal ? minVal : TMP 또한 TMP < minval이 참이면 minval을 거짓이면 TMP를 TMP에 대응시킨다는 의미입니다.

(4) Cubic interpolation

- Cubic interpolation은 Higher order interpolation 중 하나의 기법입니다. Cubic interpolation은 B-spline interpolation와 동일하게 역매핑된 화소의 근접한 16개 화소들을 이용하는 방법입니다. 보간하는 방식은 보간 필터의 계수 유도 방정식을 제외하고, B-spline interpolation과 동일합니다.
- Cubic interpolation의 보간 필터의 계수 유도 방정식은 다음과 같습니다.

$$f(x) = \begin{cases} (a+2)|X|^3 - (a+3)|X|^2 + 1, & 0 \le |X| < 1 \\ a|X|^3 - 5a|X|^2 + 8a|X| - 4a, & 1 \le |X| < 2 \\ 0, & 2 \le |X| \end{cases}$$

-> a(alpha)값은 0.5로 고정됩니다.

이를 코드로 나타내면 다음과 같습니다.

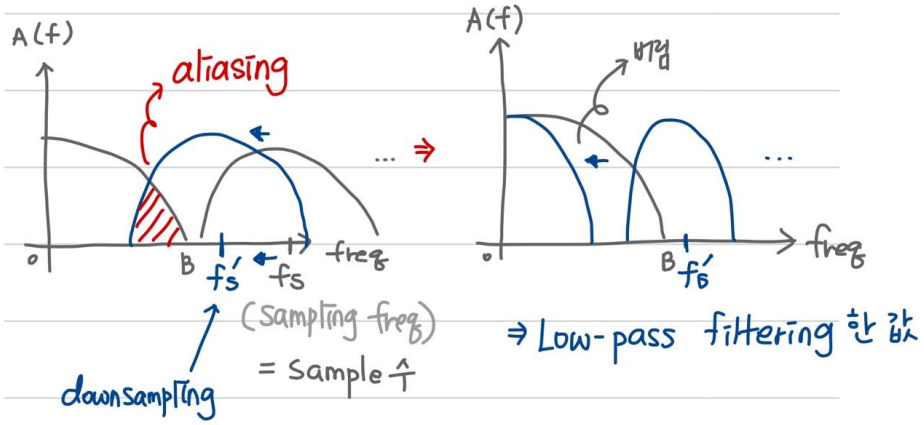
```
double Cubic_function(Double x) {

    double result;
    double a = 0.5;

    if (fabs(x) >= 0 && fabs(x) < 1) {
        result = ((a+2)*(pow(fabs(x), 3.0))) +(-1.0*(a+3)*pow(fabs(x), 2.0)) + 1.0;
    }
    else if (fabs(x) >= 1 && fabs(x) < 2) {
        result = (a*pow(fabs(x), 3.0)) + (-1.0*5.0*a*pow(fabs(x), 2.0)) + 8.0*a*fabs(x) - 4.0*a;
    }
    else
        result = 0;

    return result;
}
```

```
UChar Cubic(UChar* Data, Double srcX, Double srcY, Int Stride)
{
    int Src_X_Minus_1, Src_X_Plus_1, Src_X_Plus_2;
    int Src_Y_Minus_1, Src_Y_Plus_1, Src_Y_Plus_2;
    double Hor_Wei, Ver_Wei; //Horizontal Weight, Vertical Weight
    double TMP_Hor[4] = { 0,0,0,0 };
    double TMP = 0;

    // 위치 화소 좌표
    // 어느 화소를 기준으로 주변 화소를 구할 것인지
    Src_X_Plus_1 = CLIPPIC_HOR((int)srcX+1);
    Src_X_Plus_2 = CLIPPIC_HOR((int)srcX+2);
    Src_Y_Plus_1 = CLIPPIC_VER((int)srcY+1);
    Src_Y_Plus_2 = CLIPPIC_VER((int)srcY+2);

    Src_X_Minus_1 = CLIPPIC_HOR((int)srcX-1);
    Src_Y_Minus_1 = CLIPPIC_VER((int)srcY-1);

    Hor_Wei = srcX - (int)srcX;
    Ver_Wei = srcY - (int)srcY;


    int X_Pix[] = { Src_X_Minus_1, (int)srcX, Src_X_Plus_1, Src_X_Plus_2 };
    int Y_Pix[] = { Src_Y_Minus_1, (int)srcY, Src_Y_Plus_1, Src_Y_Plus_2 };
    double Distance_Hor[] = { Hor_Wei + 1, Hor_Wei, 1 - Hor_Wei, (1 - Hor_Wei) + 1 };
    double Distance_Ver[] = { Ver_Wei + 1, Ver_Wei, 1 - Ver_Wei, (1 - Ver_Wei) + 1 };

    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            TMP_Hor[i] += Cubic_function(Distance_Hor[j]) * Data[Y_Pix[i] * Stride + X_Pix[j]];
        }
    }

    for (int i = 0; i < 4; i++)
    {
        TMP += Cubic_function(Distance_Ver[i]) * TMP_Hor[i];
    }

    // TMP는 화소값 - clipping & 반올림 과정
    TMP += 0.5;
    TMP = TMP > maxVal ? maxVal : TMP < minVal ? minVal : TMP;

    return (UChar)TMP;
}
```

➔ Cubic interpolation 또한 정수 화소 위치의 거리에 따라 보간 필터의 유도 방정식이 달라지기 때문에 가중치를 곱할 때 Cubic function을 사용한다는 것 외에는 B-spline interpolation과 설명이 동일합니다.

실습 2) 영상의 확대 및 축소

- Geometric transformation 중 영상을 축소할 때 주의할 점이 있습니다. 축소(downsampling)를 할 때 화소값이 소실됩니다. 예를 들어 영상의 크기를 두배를 줄인다면 2개 중 1개를 버립니다. 따라서 영상의 일부가 사라지는 aliasing이 발생하게 되는 문제점이 발생합니다. 이를 피하기 위해 low-pass filtering(=blurring) 과정이 필요합니다.
- 상세한 설명은 다음 그림과 같습니다.



➔ downsampling에서 sampling frequency가 작아져 겹쳐지게 되면, 겹쳐지는 부분에서 aliasing이 발생합니다. 이 때 Low-pass filtering으로 겹쳐지는 부분이 생기지 않게 함으로써 aliasing을 막을 수 있습니다. 따라서 축소할 때는 blurring 과정이 필수적으로 필요합니다.

- 우선 실습을 진행하기 위한 geometric transformation의 코드는 다음과 같습니다.

```
void Geometric_Transformation(UChar* Data, Img_Buf* img)
{
    Int wid = WIDTH; Int hei = HEIGHT;
    Int min = minVal; Int max = maxVal;

    Double scaleVal = scaleConstant;

    // scaling을 위한 변수
    Int dstWid; // 스케일링 적용된 영상의 가로 길이
    Int dstHei; // 스케일링 적용된 영상의 세로 길이

    dstWid = wid * scaleVal + 0.5; // 반올림
    dstHei = hei * scaleVal + 0.5; // 반올림

    Rotation(Data); //회전

    if (scaleVal < 1) // 축소 시 원본 영상 블러링 적용 - 예습을 하고 싶은 사람만 봐도 됨
    {
        Image_Filtering(Data, img);
        memcpy(Data, img->Result_Blurring, sizeof(UChar) * wid * hei);
        free(img->Result_Blurring);
    }

    Scaling(Data, dstWid, dstHei, scaleVal); // 원본, 결과영상의 가로, 결과영상의 세로, scaling상수
}
```

➔ dstWid와 dstHei는 스케일링이 적용된 영상의 가로, 세로 길이를 뜻하며, scaleVal 변수에 scaling상수를 저장합니다. scaling상수는 1보다 크면 확대, 1보다 작으면 축소입니다. 그리고 회전함수와 scaling함수를 호출합니다.

```
// 역추적 mapping을 했을 때 화소가 영상의 범위를 벗어나는 경우에 가장 가까운 화소로 mapping 해 줌
#define CLIPPIC_HOR(x) (x < 0) ? 0 : x >= WIDTH  ? WIDTH  - 1 : x
#define CLIPPIC_VER(x) (x < 0) ? 0 : x >= HEIGHT ? HEIGHT - 1 : x
```

➔ 역방향 매핑을 했을 때 화소가 영상의 범위가 벗어나는 경우가 있을 수 있으므로 가까운 화소로 mapping해주는 함수는 GEO.h에 선언합니다. 이 함수는 x가 영상의 넓이보다 크면 WIDTH-1로, 작으면 x로 매핑해주고, 이때의 결과 x가 0보다 작으면 0으로, 크거나 같으면 x로 매핑하는 함수입니다.

- 확대, 축소 과정을 위한 scaling함수는 다음과 같습니다.

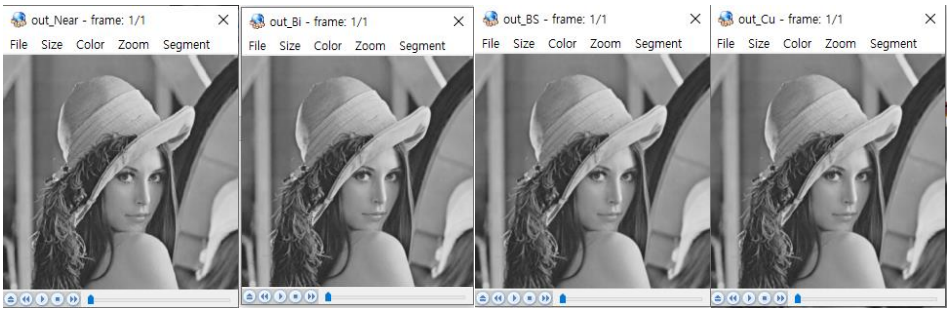
```
void Scaling(UChar* Data, Int dstWid, Int dstHei, Double scaleVal)
{
    SCALE scale;
    double srcX, srcY; //역 추적한 원본 화소 위치 (정수 위치가 아닐 수 있음)

    Char String[4][10] = { "Near", "Bi" , "BS" ,"Cu" };

    scale.Near = (UChar*)calloc(dstWid * dstHei, sizeof(UChar));
    scale.Bi = (UChar*)calloc(dstWid * dstHei, sizeof(UChar));
    scale.BS = (UChar*)calloc(dstWid * dstHei, sizeof(UChar));
    scale.Cu = (UChar*)calloc(dstWid * dstHei, sizeof(UChar));

    // 결과 영상의 화소 하나하나를 역추적해야 함
    for (int i = 0; i < dstHei; i++)
    {
        for (int j = 0; j < dstWid; j++)
        {
            // 실질적으로 역추적 하는 부분 - 원본 영상에서 결과 영상만들 때는 곱하기였으니, 결과영상에서 원본 영상을 만들 때는 나눗셈
            srcX = (double)j / scaleVal;
            srcY = (double)i / scaleVal;

            // 역추적한 화소 위치가 정수 위치일수도 있지만, 실수 위치일 경우가 더욱 많기 때문에 보간을 합니다.
            // DATA : 원본 영상의 정보 - 역추적했을 때의 화소 위치가 원본 영상 내에 있기 때문,
            // srcX, srcY : 역추적한 화소 위치
            // WIDTH : 화소가 2차원 좌표계지만, 1차원으로 쓰기 위해 WIDTH
            scale.Near[i * dstWid + j] = NearesetNeighbor(Data, srcX, srcY, WIDTH);
            scale.Bi[i * dstWid + j] = Bilinear(Data, srcX, srcY, WIDTH);
            scale.BS[i * dstWid + j] = B_Spline(Data, srcX, srcY, WIDTH);
            scale.Cu[i * dstWid + j] = Cubic(Data, srcX, srcY, WIDTH);
        }
    }

    ImageOutput(scale.Near, dstWid, dstHei, String[0]);
    ImageOutput(scale.Bi, dstWid, dstHei, String[1]);
    ImageOutput(scale.BS, dstWid, dstHei, String[2]);
    ImageOutput(scale.Cu, dstWid, dstHei, String[3]);

    free(scale.Near);
    free(scale.Bi);
    free(scale.BS);
    free(scale.Cu);
}
```

➔ Scaling을 하기 위해서는 화소의 하나하나 역추적 해야 합니다. 역방향 매핑을 사용하기 때문에 위에서 선언한 각 보간 함수들이 필요합니다. 각 보간 함수를 적용한 영상들을 저장하기 위해 공간을 동적할당으로 할당해줍니다. 전방향 매핑에서 확대 및 축소를 할 때 scaling 상수를 곱해줬기 때문에 역방향 매핑에서는 scaling 상수를 나누어야 합니다. 그리고, 2차원 좌표계인 화소의 위치를 1차원으로 쓰기 위해 y좌표*가로의길이+x좌표로 화소의 값을 저장합니다.

```
11
12  #define scaleConstant  0.47          // 스케일링 상수 - 축소/확대
13
```

➔ main.h에 선언되어 있는 scaleConstant를 바꿔줌으로써 확대 및 축소를 할 수 있습니다.

- 결과 영상은 다음과 같습니다.



➔ 위의 영상은 원래 영상을 축소한 영상이며, 각각 nearest neighbor, Bilinear, B-spline, Cubic interpolation을 사용하여 보간한 영상입니다. Scaling 상수가 0.47이기 때문에 영상의 크기를 241x241로 출력하였습니다.

➔ 위의 영상은 원래 영상을 확대한 영상이며, 각각 nearest neighbor, Bilinear, B-spline, Cubic interpolation을 사용하여 보간한 영상입니다. Scaling 상수가 2.32이기 때문에 영상의 크기를 1188x1188로 출력하였습니다. 영상이 너무 커서 전체 화면을 캡쳐할 수 없었습니다. 영상을 확대 시켰을 때 nearest neighbor interpolation 보간법을 사용한 영상의 화질이 다른 interpolation을 사용한 영상보다 더 낮음을 더욱 잘 확인할 수 있었습니다. 또한 원본 LENA영상 보다 확대했을 때의 화질이 약간 나빠짐을 확인할 수 있었습니다.

실습 3) 영상의 회전

- Geometric transformation의 하나인 영상을 회전시킬 때 역방향 매핑을 하기 위해서는 다음과 같은 식이 필요합니다.

$$\begin{pmatrix} X_{src} \\ Y_{src} \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} X_{dst} - C_x \\ Y_{dst} - C_y \end{pmatrix} + \begin{pmatrix} C_x \\ C_y \end{pmatrix}$$

-> Cx와 Cy는 회전시킬 때의 중심점입니다.

➔ 이 행렬 계산식은 회전할 때 화소 하나하나에 대해 계산해야 합니다.

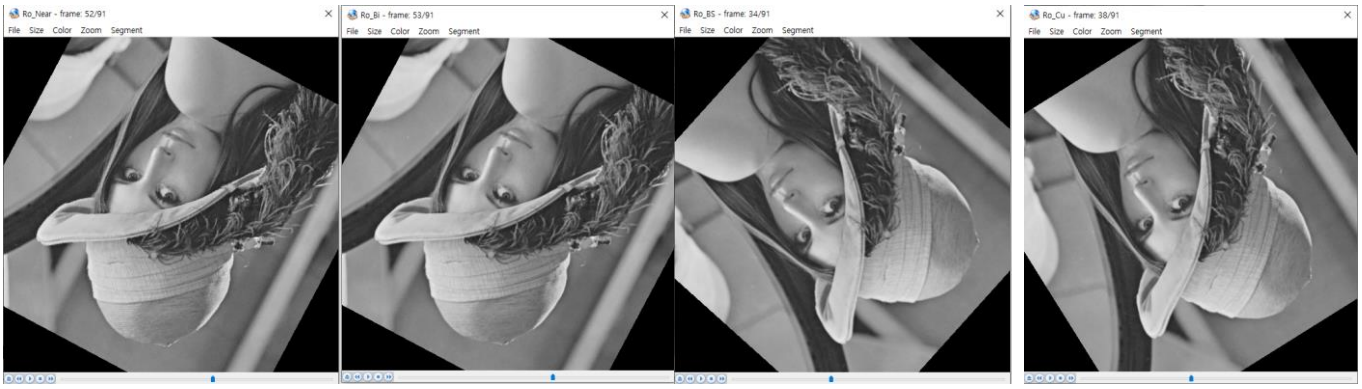
- 영상 회전을 위한 코드는 다음과 같습니다.

```
void Rotation(UChar* Data)
{
    ROTATION rot;
    FILE* up1, * up2, * up3, * up4;

    double Angle;
    double srcX, srcY;  // Source 위치

    int New_X, New_Y;
    int Center_X = WIDTH / 2, Center_Y = HEIGHT / 2; // 영상의 가운데
    // int Center_X = 0, Center_Y = 0; // (0,0)을 중심으로 할 때

    fopen_s(&up1, "Ro_Near.raw", "wb");
    fopen_s(&up2, "Ro_Bi.raw", "wb");
    fopen_s(&up3, "Ro_BS.raw", "wb");
    fopen_s(&up4, "Ro_Cu.raw", "wb");

    rot.Near = (UChar*)calloc(WIDTH * HEIGHT, sizeof(UChar));
    rot.Bi = (UChar*)calloc(WIDTH * HEIGHT, sizeof(UChar));
    rot.BS = (UChar*)calloc(WIDTH * HEIGHT, sizeof(UChar));
    rot.Cu = (UChar*)calloc(WIDTH * HEIGHT, sizeof(UChar));

    for (Angle = 0; Angle <= 360; Angle += 4)
    {
        double Seta = PI / 180.0 * Angle;

        for (int i = 0; i < HEIGHT; i++)
        {
            for (int j = 0; j < WIDTH; j++)
            {
                // 행렬 계산
                srcX = cos(Seta)*((double)j - Center_X) + sin(Seta)*((double)i - Center_Y) + Center_X;
                srcY = -sin(Seta)*((double)j - Center_X) + cos(Seta)*((double)i - Center_Y) + Center_Y;

                New_X = (int)srcX;
                New_Y = (int)srcY;

                // 역매핑한 원본 화소의 위치가 원본 영상 내에 있는지 확인
                if (!(New_X < 0 || New_X >= WIDTH - 1 || New_Y < 0 || New_Y >= HEIGHT - 1)) // 원시 화소가 영상 경계 밖에 위치
                {
                    // interpolation함수
                    rot.Near[i * WIDTH + j] = NearesetNeighbor(Data, srcX, srcY, WIDTH);
                    rot.Bi[i * WIDTH + j] = Bilinear(Data, srcX, srcY, WIDTH);
                    rot.BS[i * WIDTH + j] = B_Spline(Data, srcX, srcY, WIDTH);
                    rot.Cu[i * WIDTH + j] = Cubic(Data, srcX, srcY, WIDTH);
                }
                else
                {
                    rot.Near[i * WIDTH + j] = 0;
                    rot.Bi[i * WIDTH + j] = 0;
                    rot.BS[i * WIDTH + j] = 0;
                    rot.Cu[i * WIDTH + j] = 0;
                }
            }
        }

        fwrite(rot.Near, sizeof(UChar), (WIDTH * HEIGHT), up1);
        fwrite(rot.Bi, sizeof(UChar), (WIDTH * HEIGHT), up2);
        fwrite(rot.BS, sizeof(UChar), (WIDTH * HEIGHT), up3);
        fwrite(rot.Cu, sizeof(UChar), (WIDTH * HEIGHT), up4);
    }

    free(rot.Near);
    free(rot.Bi);
    free(rot.BS);
    free(rot.Cu);

    fclose(up1);
    fclose(up2);
    fclose(up3);
    fclose(up4);
}
```

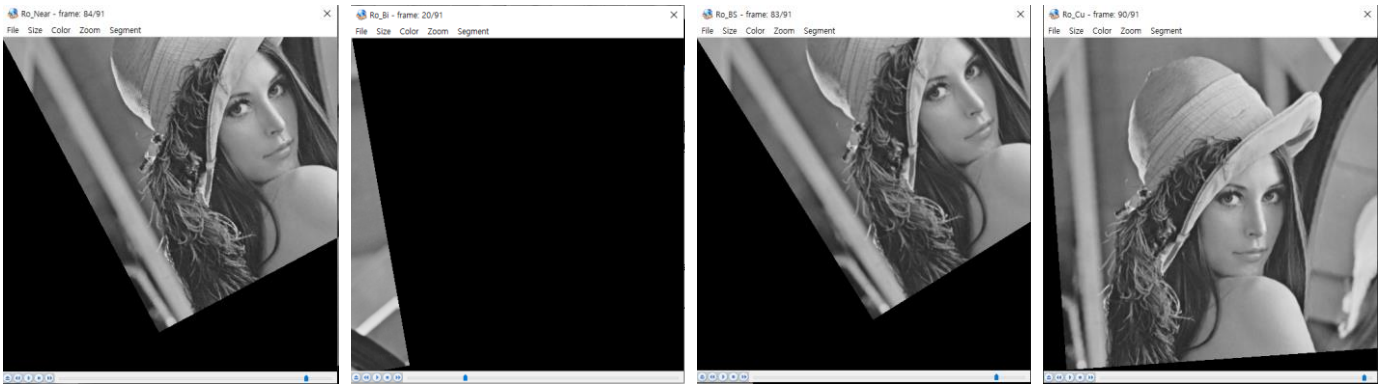
➔ 회전을 하기 위해서는 영상을 임의의 방향으로 특정한 seta만큼 회전시키는 과정에서의 역방향 매핑 식을 적용해야 합니다. 또한 seta는 pi를 각도 angle로 나눈 것이기 때문에 seta를 각도마다 갱신하고, source 화소값을 구하기 위해 행렬 계산을 하여 각각 srcX와 srcY에 저장합니다. 역방향 매핑한 화소의 위치가 원본 영상 내에 있는지 확인하기 위한 과정을 거쳐야 합니다. 원본 화소 경계 내에 위치할 경우 4가지의 보간법으로 보간하고, 영상 경계의 밖에 위치할 경우 0으로 지정합니다. 또한 2차원 좌표값을 1차원으로 바꾸기 위해 'y좌표*가로의 길이+x좌표'에 해당하는 화소값을 넣어줍니다.

➔ Center_X와 Center_Y는 중심점을 뜻합니다. 영상의 가운데를 회전의 중심으로 두고 영상을 회전시킬 때는 Center_X와 Center_Y를 각각 넓이와 높이의 1/2한 값으로 지정하고, (0,0)을 중심으로 두고 영상을 회전시킬 때는 0으로 지정합니다.

- 결과 영상은 다음과 같습니다.



➔ 위의 영상은 영상의 가운데를 회전의 중심으로 두고 원래 영상을 회전한 영상이며, 각각 nearest neighbor, Bilinear, B-spline, Cubic interpolation을 사용하여 보간한 영상입니다. 회전하는 도중에 영상을 캡쳐하였습니다. 이 때 nearest neighbor interpolation으로 보간한 영상이 회전할 때 다른 영상에 비해 visual blockiness 현상이 나타남을 확인할 수 있었습니다. 이 현상은 가장 가까운 화소값으로 보간하는 nearest neighbor interpolation의 특성상 나타나는 현상입니다.



➔ 위의 영상은 (0,0)을 회전의 중심으로 두고 원래 영상을 회전한 영상이며, 각각 nearest neighbor, Bilinear, B-spline, Cubic interpolation을 사용하여 보간한 영상입니다. 회전하는 도중에 영상을 캡쳐하였습니다. 영상의 가운데를 회전의 중심으로 두고 실습을 진행한 것과 같이 nearest neighbor interpolation으로 보간한 영상이 회전할 때 다른 영상에 비해 visual blockiness 현상이 나타남을 확인할 수 있었습니다. 이 현상은 가장 가까운 화소값으로 보간하는 nearest neighbor interpolation의 특성상 나타나는 현상입니다