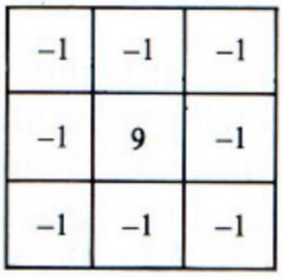
Programming Assignment No.1

Due Date: April 3, 2018

Q1. Implement the sharpening filter applying following sharpening kernel.



−1	−1	−1
−1	9	−1
−1	−1	−1

Figure. Sharpening filter kernel

Sharpening filter, (or high-pass spatial filter) detects edges in the image. When we perform convolution with Sharpening filter kernel above, pixels on edge (where intensity of the pixel shows distinct difference with the pixels surrounding it) have large absolute value, while pixels that have same intensity with the pixels surrounding it have small absolute value. Thus, we can detect edges by using the kernel above.

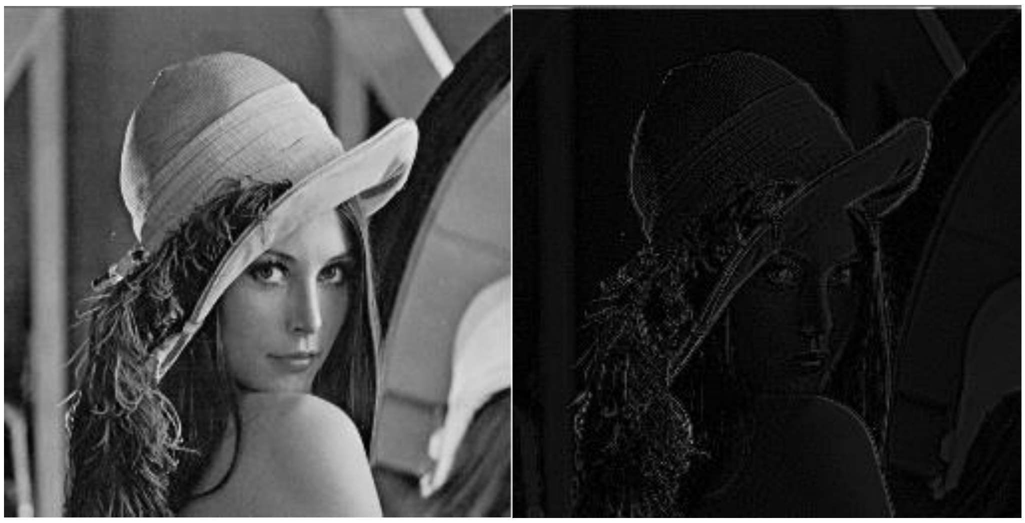
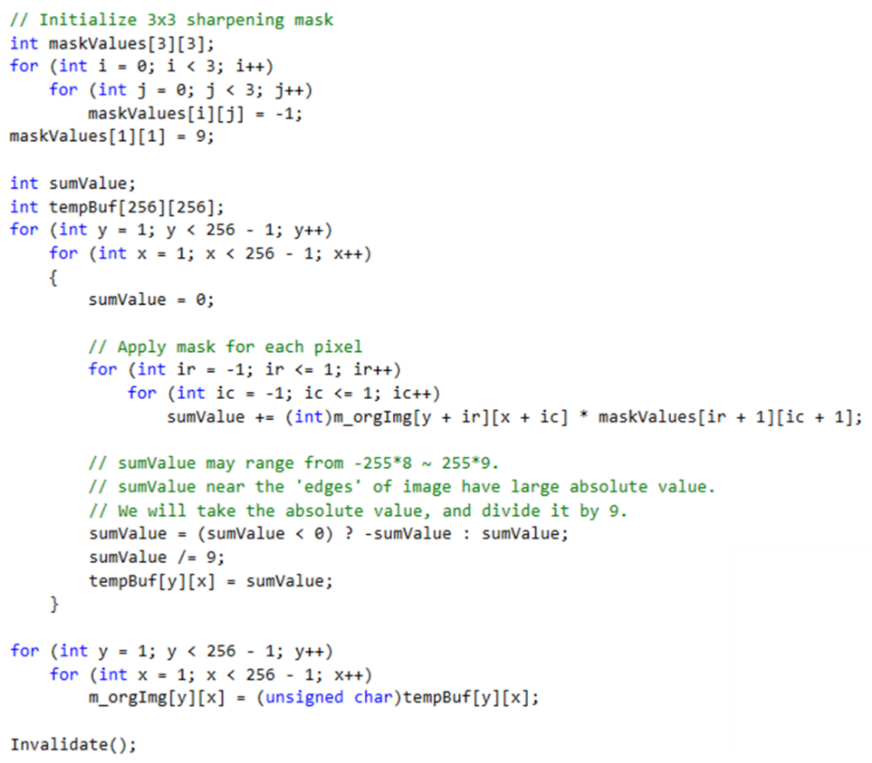


Figure. Original image (LENA.raw) and its edge detection result.

Since convolution result is in range -255*8~255*9, we have to put this in range 0~255 for visualization. We want to highlight edges (dark and bright edges), so we will put an absolute value on the convolution result and divide it by 9.

The C++ code for edge detection is as following.



```
// Initialize 3x3 sharpening mask
int maskValues[3][3];
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        maskValues[i][j] = -1;
maskValues[1][1] = 9;

int sumValue;
int tempBuf[256][256];
for (int y = 1; y < 256 - 1; y++)
    for (int x = 1; x < 256 - 1; x++)
    {
        sumValue = 0;

        // Apply mask for each pixel
        for (int ir = -1; ir <= 1; ir++)
            for (int ic = -1; ic <= 1; ic++)
                sumValue += (int)m_orgImg[y + ir][x + ic] * maskValues[ir + 1][ic + 1];

        // sumValue may range from -255*8 ~ 255*9.
        // sumValue near the 'edges' of image have large absolute value.
        // We will take the absolute value, and divide it by 9.
        sumValue = (sumValue < 0) ? -sumValue : sumValue;
        sumValue /= 9;
        tempBuf[y][x] = sumValue;
    }

for (int y = 1; y < 256 - 1; y++)
    for (int x = 1; x < 256 - 1; x++)
        m_orgImg[y][x] = (unsigned char)tempBuf[y][x];

Invalidate();
```

Figure. C++ code for edge detection

Image sharpening can be understood as edge emphasizing. Which means, we'd like to detect edges and highlight it on the original image. The simplest implementation would be just adding edge detection result to the original image.

But we need to remember that our edge detection ignores whether the edge is darker or brighter than the surroundings. (We delete the information by using absolute value.) The dark edges would have negative convolution value. Since we want to make bright edges brighter and dark edges darker, We will just divide the convolution result by 9 and add it to the original pixel intensity. It may have values outside 0~255, but we will just saturate those values to the limit for visualization.

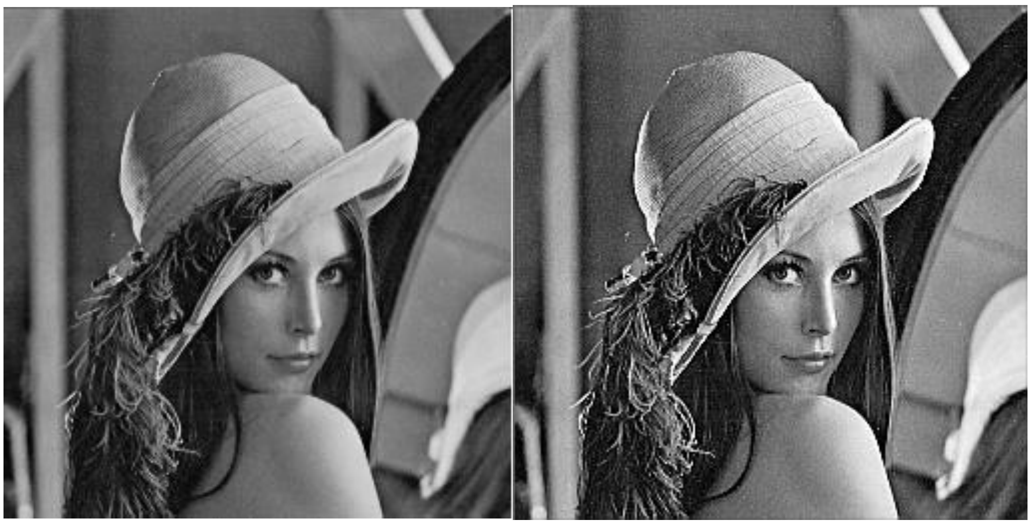
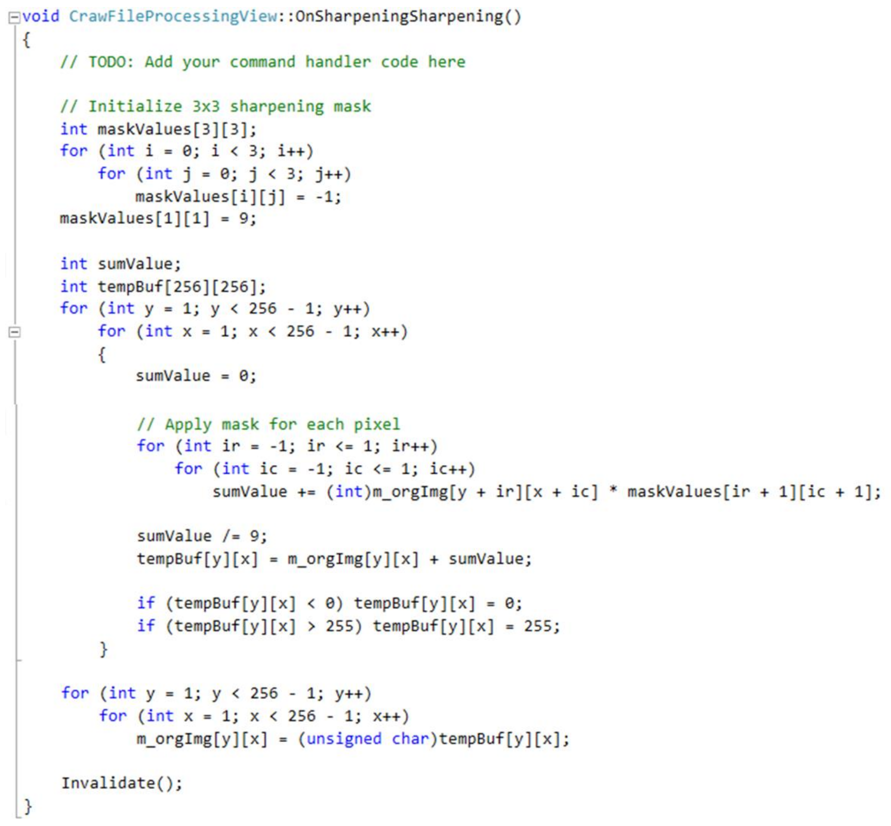


Figure. Original image (LENA.raw) and sharpened image

The C++ code for sharpening is as following.



```
void CrawFileProcessingView::OnSharpeningSharpening()
{
    // TODO: Add your command handler code here

    // Initialize 3x3 sharpening mask
    int maskValues[3][3];
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            maskValues[i][j] = -1;
    maskValues[1][1] = 9;

    int sumValue;
    int tempBuf[256][256];
    for (int y = 1; y < 256 - 1; y++)
        for (int x = 1; x < 256 - 1; x++)
        {
            sumValue = 0;

            // Apply mask for each pixel
            for (int ir = -1; ir <= 1; ir++)
                for (int ic = -1; ic <= 1; ic++)
                    sumValue += (int)m_orgImg[y + ir][x + ic] * maskValues[ir + 1][ic + 1];

            sumValue /= 9;
            tempBuf[y][x] = m_orgImg[y][x] + sumValue;

            if (tempBuf[y][x] < 0) tempBuf[y][x] = 0;
            if (tempBuf[y][x] > 255) tempBuf[y][x] = 255;
        }

    for (int y = 1; y < 256 - 1; y++)
        for (int x = 1; x < 256 - 1; x++)
            m_orgImg[y][x] = (unsigned char)tempBuf[y][x];

    Invalidate();
}
```

Figure. C++ code for Image sharpening

From attached project, you can click on this menu button (Sharpening – Sharpening) to conduct image sharpening.

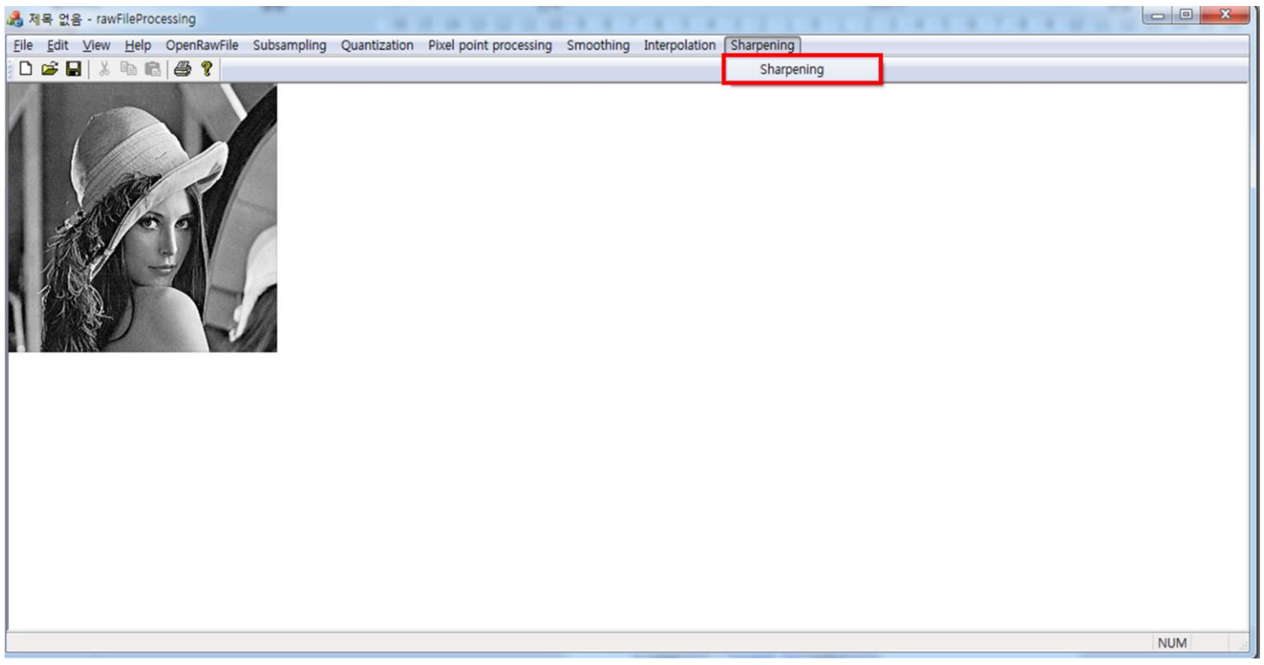
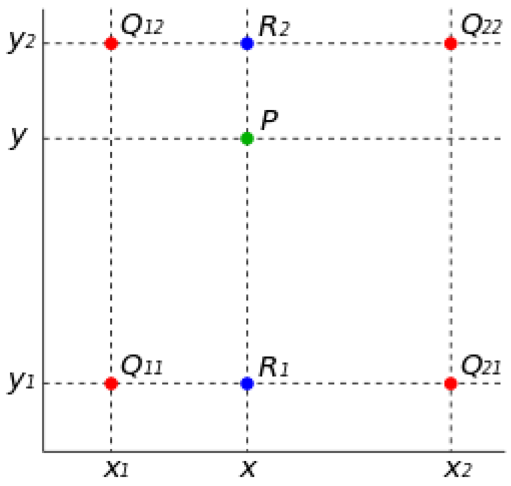


Figure. How to access Sharpening

Q2. Implement 1:2 bilinear interpolation

Bilinear interpolation is an extension of linear interpolation on a rectilinear 2D grid. The key idea is to perform linear interpolation first in one direction, and then again in the other direction.



Look above Figure. In order to calculate intensity of point P, we should observe the known grind Q_{11}, Q_{12}, Q_{21}, Q_{21}. First, use linear interpolation on x axis to find intensity of point R_1 and R_2. Since they have same x values with point P, now we can use linear interpolation on y axis to find intensity of point P.

For nearest neighbor interpolation, the procedure is similar, but with different value.

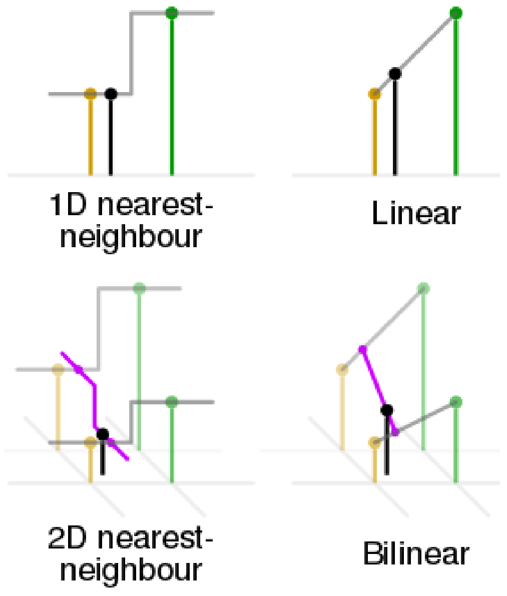


Figure. Concepts of 2D nearest-neighbor and Bilinear interpolation

I came with another example to enhance understanding of nearest-neighbor and bilinear interpolation.

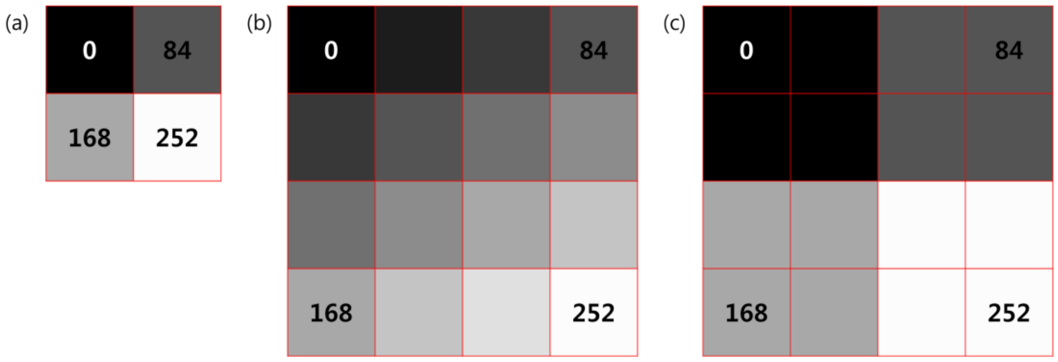
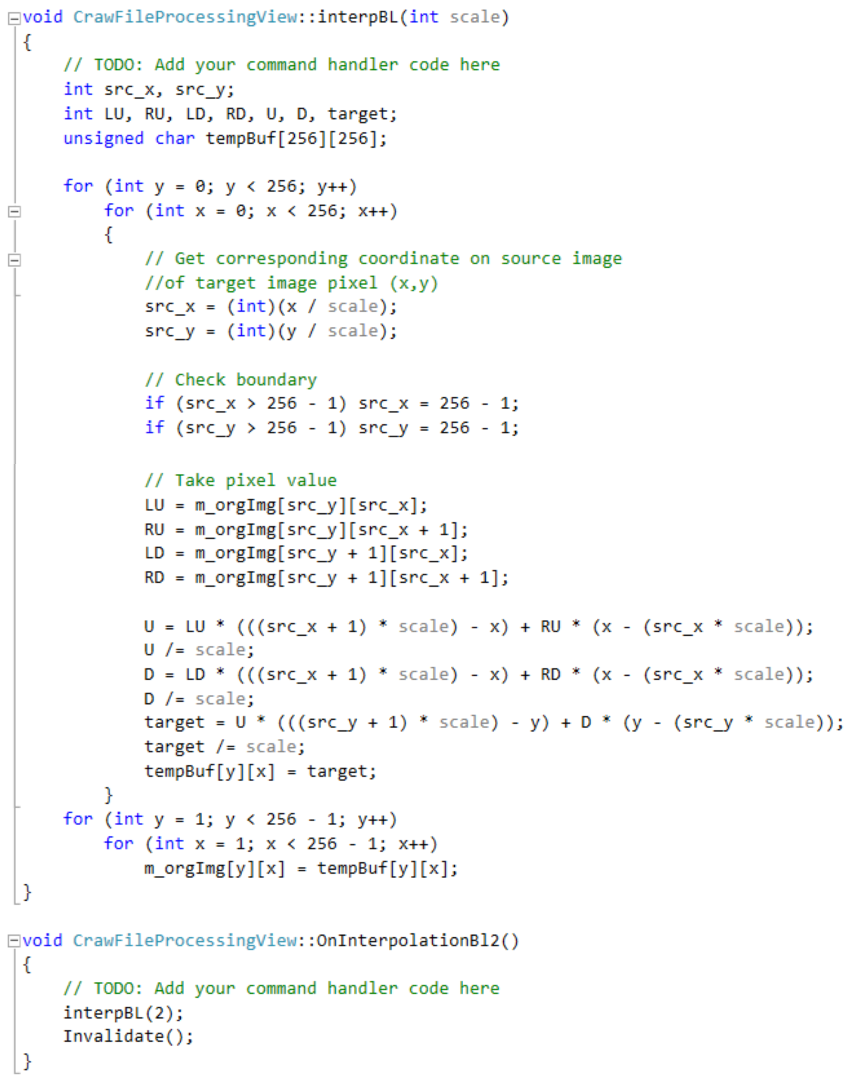


Figure. (a) Original grid (b) 1:4 Bilinear interpolation (c) 1:4 Nearest-neighbor interpolation

The C++ code for (1:2) bilnear interpolation is as following.



```
void CrawFileProcessingView::interpBL(int scale)
{
    // TODO: Add your command handler code here
    int src_x, src_y;
    int LU, RU, LD, RD, U, D, target;
    unsigned char tempBuf[256][256];

    for (int y = 0; y < 256; y++)
        for (int x = 0; x < 256; x++)
        {
            // Get corresponding coordinate on source image
            //of target image pixel (x,y)
            src_x = (int)(x / scale);
            src_y = (int)(y / scale);

            // Check boundary
            if (src_x > 256 - 1) src_x = 256 - 1;
            if (src_y > 256 - 1) src_y = 256 - 1;

            // Take pixel value
            LU = m_orgImg[src_y][src_x];
            RU = m_orgImg[src_y][src_x + 1];
            LD = m_orgImg[src_y + 1][src_x];
            RD = m_orgImg[src_y + 1][src_x + 1];

            U = LU * (((src_x + 1) * scale) - x) + RU * (x - (src_x * scale));
            U /= scale;
            D = LD * (((src_x + 1) * scale) - x) + RD * (x - (src_x * scale));
            D /= scale;
            target = U * (((src_y + 1) * scale) - y) + D * (y - (src_y * scale));
            target /= scale;
            tempBuf[y][x] = target;
        }
    for (int y = 1; y < 256 - 1; y++)
        for (int x = 1; x < 256 - 1; x++)
            m_orgImg[y][x] = tempBuf[y][x];
}

void CrawFileProcessingView::OnInterpolationBl2()
{
    // TODO: Add your command handler code here
    interpBL(2);
    Invalidate();
}
```

Figure. C++ for 1:2 Bilinear interpolation

And this is the result of interpolations.

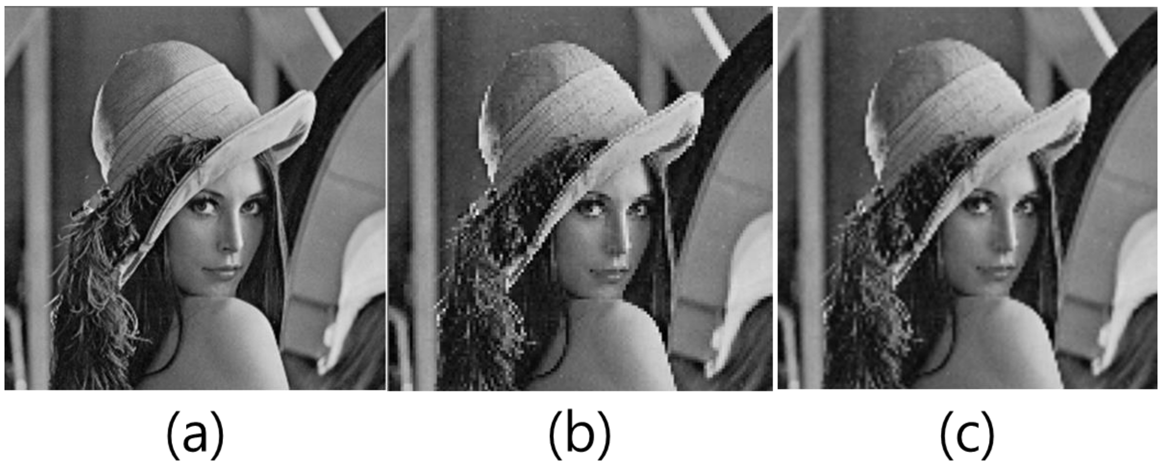
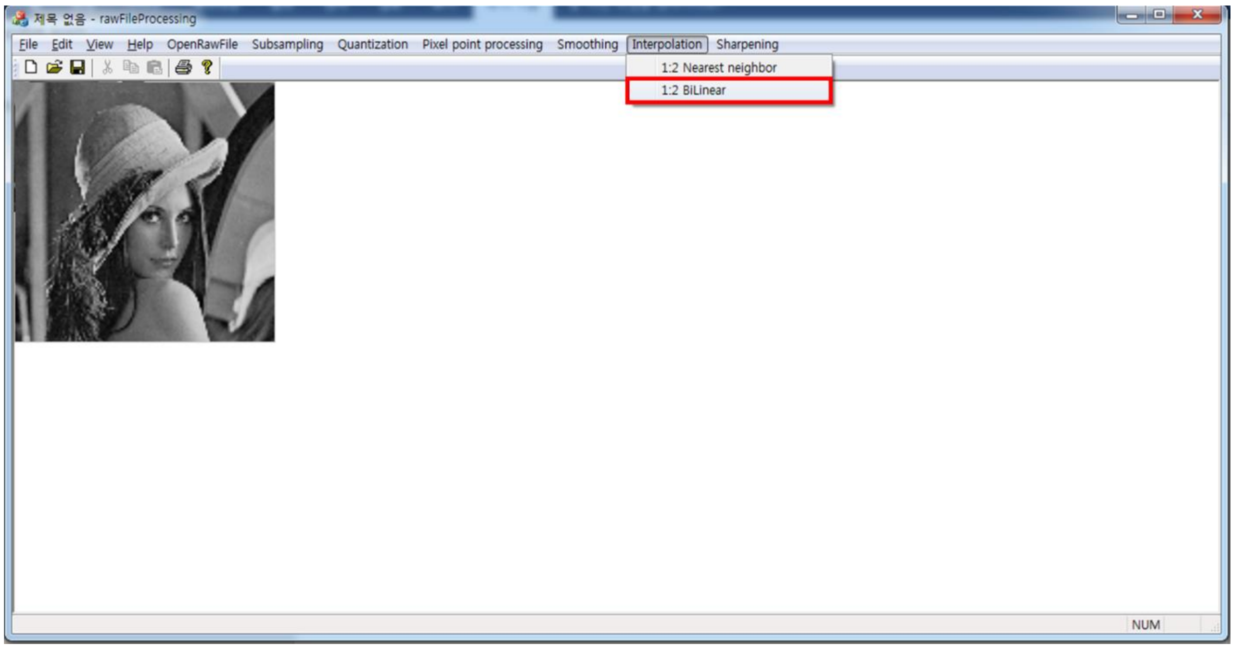


Figure. (a) Original image (LENA.raw), (b) 1:2 Nearest-neighbor interpolation, (c) 1:2 Bilinear interpolation

We need to subsample original image 2:1 to create same size interpolation images. We can see that image recreated by interpolation does have smaller resolution than the original image, but bilinear interpolation has smoother detail than the nearest-neighbor interpolation, because the intensity gap between adjacent pixels are much bigger on the nearest-neighbor interpolation.

From attached project, you can click on this menu button (Interpolation – 1:2 Bilinear) to conduct image sharpening.


제목 없음 - rawFileProcessing
File Edit View Help OpenRawFile Subsampling Quantization Pixel point processing Smoothing Interpolation Sharpening
1:2 Nearest neighbor
1:2 BiLinear
NUM