Lab 2

Use two images for each operation to do the following operations and explain your results(if any):

1. Image reduction
   Alternative line reduction
   Linear reduction to reduce images to any smaller size
2. Perform image enlargement
   Pixel replication
   Nearest enlargement
   Bilinear interpolation
   Linear expansion to expand images to any larger size
3. Perform negative image operation
   Negative images

Algorithm:

Use the pseudo code or figure to display the algorithm.

1. Alternative line reduction

```
Input: an unsigned char array that stores the source image data
Output: an unsigned char array that stores the output image data
For i < inimage.height
    For j < inimage.width
        outimage[i/2][j/2]=inimage[i][j]
        i=i+2
        j=j+2
return outimage
```

2. Any Size Adjustment

```
Input: an unsigned char array that stores the source image data, a float ratio of adjustment
Output: an unsigned char array that stores the output image data
For i < outimage.height; i++
    For j < outimage.width; j++
        outimage[i][j]=inimage[i/ratio][j/ratio]
return outimage
```

3. Pixel Replication

```
Input: an unsigned char array that stores the source image data, an integer of magnification times
Output: an unsigned char array that stores the output image data
For i < inimage.height; i++
    For j < inimage.width; j++
        For p < multiple
            For q < multiple
                outimage[i * multiple + p][ j * multiple + q]=inimage[i][j]
return outimage
```

4. Nearest enlargement

   Input: an unsigned char array that stores the source image data, an integer of magnification times

Output: an unsigned char array that stores the output image data

```
 For i < outimage.height; i++
        For j < outimage.width; j++
               x = round((float)(i) / multiple);
               y = round((float)(j ) / multiple);
               outimage[i][j]=inimage[x][y]
 return outimage
```

5. Bilinear interpolation

   Input: an unsigned char array that stores the source image data, an integer of magnification times

   Output: an unsigned char array that stores the output image data

```
scaleX = (float)(image->Width - 1) / (outImage->Width - 1); // The ratio of enlargement
scaleY = (float)(image->Height - 1) / (outImage->Height - 1);
 For i < outimage.height; i++
        For j < outimage.width; j++
               scrx = i * scaleY; // The coordinate of inimage after projection
               scry = j * scaleX;
               x = floor(scrx);// Get the decimal part
               y = floor(scry);
               u = scrx - x; // Get the weight of 2 neighbours
               v = scry - y;
               outimage[i ][ j] = (1 - u) * (1 - v) * inimage[x][y] // Sum up the value of 4 neighbours
                                                                    //after mulitipling the weight
                      +u * (1 - v) * inimage [x + 1][y]
                      + (1 - u) * v * inimage [x][y + 1]
                      + u * v * inimage [x + 1] [y + 1];
 return outimage
```

6. Negative images

   For every pixel in inimage

```
        Outimage[i][j]=255-inimage[i][j]
```

Results (compare the results with the original image):

Paste the result images and the original ones.

1. Alternative line reduction

Source

Result

Source

Result

2. Any Size Adjustment

Source

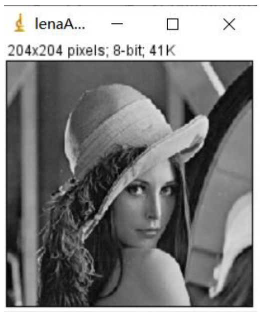
Expansion(1.2 times)

Reduction(0.8 times)

Source

Expansion(1.2 times)

Reduction(0.8 times)



3. Pixel Replication

Source

Result(3 times)

Source

Result(3 times)

4. Nearest enlargement

Source

Result(3 times)

Source

Results(3 times)

5. Bilinear interpolation

Source

Results(3 times)

Source

Results(3 times)

6. Negative images

Source

Result

Source

Result

Discussion:

White down your discovery about the test.

1. Alternative line reduction

   The conduction of this method is removing the middle line and column between 2 lines or columns. The result is to reduce the size to half of the input image.

2. Any size adjustment

   Here I use a projection that calculates the source coordinates by multiplying the ratio. Using the round method when having a float number. Because it is just a simple projection, there is a noticeable jagged shape in the image

3. Pixel Replication & Nearest Interpolation Enlargement

   The pixel replication method enlarges the image by replicating every pixel in the input image and the nearest interpolation enlargement method requires assigning the nearest value to the new pixels. The results of these two methods both have a noticeable jagged shape in the images.

4. Bilinear interpolation

   Bilinear interpolation method calculates the value of a new pixel from its 4 neighbors. The result of bilinear interpolation is smooth, but also a little fuzzy. The effectiveness seems better than the methods above.

5. Negative images

   Using 255 minus the source pixel can get the negative images.

Codes:

You don't need to paste all the codes. Just show the pieces of code that present the algorithm displayed above.

1. Alternative line reduction

```
Image* AlternativeLineReduction(Image* image) {
    Image* outImage;
    int i, j, size;
    unsigned char* tempin, * tempout;

    Image* outImage;
    outImage = (Image*)malloc(sizeof(Image));
    outImage->Width = image->Width * 0.5;
    outImage->Height = image->Height * 0.5;
    outImage->Type = image->Type;
    if (outImage->Type == GRAY) size = outImage->Width * 0.5 * outImage->Height * 0.5;
    if (outImage->Type == COLOR) size = outImage->Width * 0.5 * outImage->Height * 0.5 * 3;
    outImage->data = (unsigned char*)malloc(size);

    tempout = outImage->data;
    tempin = image->data;
    for (i = 0; i < image->Height; i += 2) {
        for (j = 0; j < image->Width; j += 2) {
            tempout[(image->Width / 2) * (i / 2) + j / 2] = tempin[image->Width * i + j];
        }
    }
    return outImage;
}
```

2. Any size Adjustment

```
Image* AnySizeAdjustment(Image* image, float ratio) {
    int i, j;
    int size;
    unsigned char* tempin, * tempout;

    Image* outImage;
    outImage = (Image*)malloc(sizeof(Image));

    outImage->Width = image->Width * ratio;
    outImage->Height = image->Height * ratio;
    outImage->Type = image->Type;
    if (outImage->Type == GRAY) size = outImage->Width * outImage->Height;
    if (outImage->Type == COLOR) size = outImage->Width * outImage->Height * 3;
    outImage->data = (unsigned char*)malloc(size);

    tempout = outImage->data;
    tempin = image->data;

    for (i = 0; i < outImage->Height; i++) {
        for (j = 0; j < outImage->Width; j++) {
            tempout[i * outImage->Width + j] = tempin[(int)(i/ratio)*image->Width+ (int)(j/ratio)];
        }
    }
    return outImage;
}
```

3. Pixel replication

```
Image* PixelReplication(Image* image, int multiple) {
    int i, j, p, q;
    int size;
    unsigned char* tempin, * tempout;

    Image* outImage;
    outImage = CreateNewImage(image, "#NearesNeighbour", multiple);

    tempout = outImage->data;
    tempin = image->data;

    int count = 0;
    for (i = 0; i < image->Height; i++) {
        for (j = 0; j < image->Width; j++) {
            for (p = 0; p < multiple; p++) {
                for (q = 0; q < multiple; q++) {
                    tempout[i * multiple * outImage->Width + j * multiple + p * outImage->Width + q] = tempin[i * image->Width + j];
                }
            }
        }
    }
    return outImage;
}
```

4. Nearest enlargement

```
Image* NearestNeighbour(Image* image,int multiple) {
    int i, j;
    float x, y;
    unsigned char* tempin, * tempout;

    Image* outImage;
    outImage = CreateNewImage(image, "#NearesNeighbour", multiple);
    tempout = outImage->data;
    tempin = image->data;

    for (i = 0; i < outImage->Height; i++) {
        for (j = 0; j < outImage->Width; j++) {
            x = round((float)(i) / multiple);
            y = round((float)(j ) / multiple);
            tempout[outImage->Width * i + j] = tempin[(int)(image->Width * (int)x + (int)y)];
        }
    }
    return outImage;
}
```

5. Bilinear enlargement

```
Image* BilinearInterpolation(Image* image, int multiple) {
    int i, j, x, y;
    float scrx, scry, u, v;
    float scaleX,scaleY;
    unsigned char* tempin, * tempout;

    Image* outImage;
    outImage = CreateNewImage(image, "#BilinearInterpolation", multiple);

    tempout = outImage->data;
    tempin = image->data;

    scaleX = (float)(image->Width - 1) / (outImage->Width - 1);
    scaleY = (float)(image->Height - 1) / (outImage->Height - 1);
    printf("%f", scaleX);

    for (i = 0; i < outImage->Height; i++) {
        for (j = 0; j < outImage->Width; j++) {
            scrx = i * scaleY;
            scry = j * scaleX;
            x = floor(scrx);
            y = floor(scry);
            u = scrx - x;
            v = scry - y;
            tempout[i * outImage->Width + j] = (float)(1 - u) * (float)(1 - v) * (float)tempin[x * image->Width + y]
                + (float)u * (float)(1 - v) * (float)tempin[(x + 1) * image->Width + y]
                + (float)(1 - u) * (float)v * (float)tempin[x * image->Width + y + 1]
                + (float)u * (float)v * (float)tempin[(x + 1) * image->Width + y + 1];
        }
    }
    return outImage;
}
```

6. Only gray image

```
Image* NegativeImage(Image* image) {
    int i, j;
    unsigned char* tempin, * tempout;

    Image* outImage;
    outImage = CreateNewImage(image, "#NegativeImage",1);

    tempout = outImage->data;
    tempin = image->data;

    for (i = 0; i < outImage->Height; i++) {
        for (j = 0; j < outImage->Width; j++) {
            tempout[outImage->Width * i + j]=255- tempin[image->Width * i + j];
        }
    }
    return outImage;
}
```