Class Lab 2

Use two images for each operation to do the following operations and write down their advantages and disadvantages and explain your results:

1. Image reduction(lena, bridge):

- Alternative line reduction

Algorithm:

A	B
C	D

$\xrightarrow{3}$

A	A	A	B	B	B
A	A	A	B	B	B
A	A	A	B	B	B
C	C	C	D	D	D
C	C	C	D	D	D
C	C	C	D	D	D

For each pixel of the new image $f(x_1, y_1)$, we can find the mapping point $g(x_2, y_2)$ in the old image.

$$\begin{cases} x_2 = \frac{x_1}{n} \\ y_2 = \frac{x_1}{n} \end{cases} \qquad n \textit{ is the enlargement factor}$$

In this case, let x_2 and y_2 are both integer by using rounding down.
Then we could get $f(x_1, y_1) = g(x_2, y_2)$.

Results (including pictures):

Result of processing "Lena.pgm":

Source Image:

Result after Nearest enlargement:

Result of processing “Bridge.pgm”:

Source Image:

Result after Nearest enlargement:

Discussion:

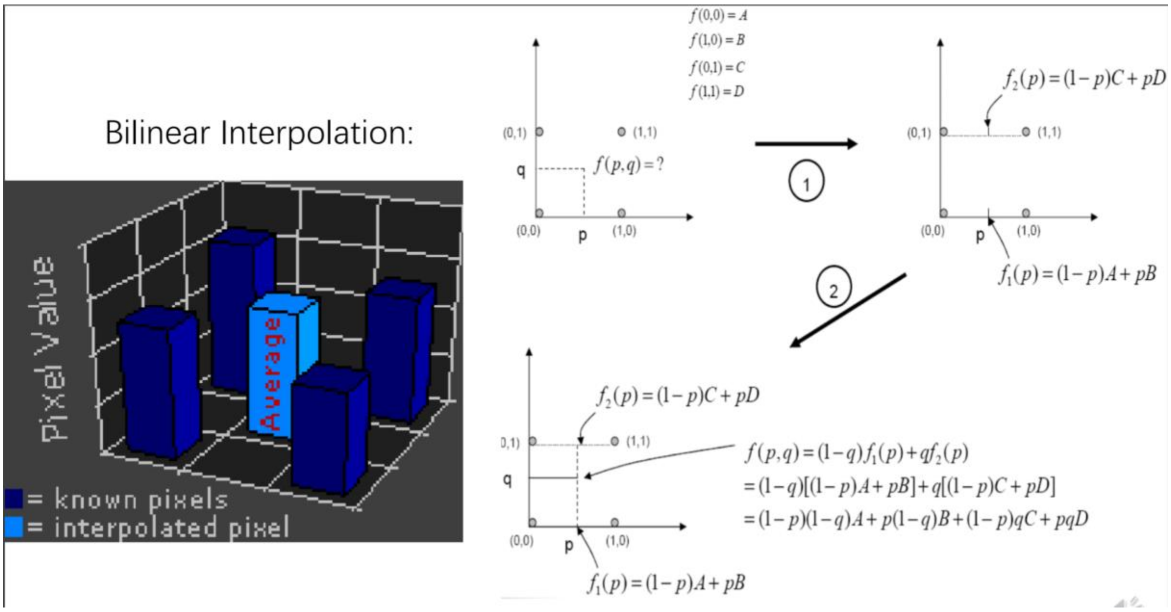
According to the images, we can conclude that image becomes blurred. The details are narrowed down.

Codes:

```
Image *PixelReplication(Image *image, float number) {        You, 4 hours ago • lab2
    unsigned char *tempin, *tempout;
    int size;
    Image *outimage;
    int newImg_Height = image->Height * number;
    int newImg_width = image->Width * number;
    outimage = CreateNewImage(image, newImg_Height, newImg_width, "#Pixel Replication");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < newImg_width; i++) {
        for (int j = 0; j < newImg_Height; j++, tempout++) {
            // round down
            int pre_x = i / number;
            int pre_y = j / number;
            if (pre_x > matrixWidth - 1) {
                pre_x = matrixWidth - 1;
            }
            if (pre_y > matrixWidth - 1) {
                pre_y = matrixWidth - 1;
            }
            *tempout = matrix[pre_x][pre_y];
        }
    }

    return (outimage);
}
```

- Fractional linear reduction to reduce images

Algorithm:



$$f(x,y) = (1-x)(1-y)A + x(1-y)B + (1-x)yC + xyD$$
$$A = f(0,0)$$
$$B = f(1,0)$$
$$C = f(0,1)$$
$$D = f(1,1)$$

Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

Result after Nearest enlargement:

Result of processing “Bridge.pgm”:

Source Image:

Result after Nearest enlargement:

Discussion:

The interpolation result is smoother than the nearest neighbor, but the smoothness of the details is equal to the appearance of blurry.

Codes:

```
Image *BilinearInterpolationImage(Image *image, float number) {
    unsigned char *tempin, *tempout;
    int size;
    Image *outimage;
    int newImg_Height = image->Height * number;
    int newImg_width = image->Width * number;
    outimage = CreateNewImage(image, newImg_Height, newImg_width, "#Bilinear Interpolation");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < newImg_width; i++) {
        for (int j = 0; j < newImg_Height; j++, tempout++) {
            float p = i / number;
            float q = j / number;
            int new_pre_x = (int)p;
            int new_pre_y = (int)q;
            p = 1 / p;
            q = 1 / q;
            if (new_pre_x > matrixWidth - 2) {
                new_pre_x = matrixWidth - 2;
            }
            if (new_pre_y > matrixWidth - 2) {
                new_pre_y = matrixWidth - 2;
            }
            int piont1 = matrix[new_pre_x][new_pre_y];
            int piont2 = matrix[new_pre_x][new_pre_y + 1];
            int piont3 = matrix[new_pre_x + 1][new_pre_y];
            int piont4 = matrix[new_pre_x + 1][new_pre_y + 1];

            *tempout = (1 - p) * (1 - q) * piont1 + (1 - p) * q * piont2 + p * (1 - q) * piont3 + p * q * piont4;
        }
    }

    return (outimage);
}
```

2. Image Enlargement (lena, bridge, noise):

- Pixel replication

Algorithm:

A	B
C	D

$\xrightarrow{3}$

A	A	A	B	B	B
A	A	A	B	B	B
A	A	A	B	B	B
C	C	C	D	D	D
C	C	C	D	D	D
C	C	C	D	D	D

For each pixel of the new image $f(x_1, y_1)$, we can find the mapping point $g(x_2, y_2)$ in the old image.

$$\begin{cases} x_2 = \frac{x_1}{n} \\ y_2 = \frac{x_1}{n} \end{cases} \qquad n \text{ is the enlargement factor}$$

In this case, let x_2 and y_2 are both integer by using rounding down.
Then we could get $f(x_1, y_1) = g(x_2, y_2)$.

Results (including pictures):

Result of processing "Lena.pgm":

Source Image:

Result after pixel replication:

Result of processing “Bridge.pgm”:

Source Image:

Result after pixel replication::

Discussion:

According to the images, we can conclude that image becomes blurred by pixel replication. The details are narrowed down.

Codes:

```
Image *PixelReplication(Image *image, float number) {        You, 4 hours ago • lab2
    unsigned char *tempin, *tempout;
    int size;
    Image *outimage;
    int newImg_Height = image->Height * number;
    int newImg_width = image->Width * number;
    outimage = CreateNewImage(image, newImg_Height, newImg_width, "#Pixel Replication");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < newImg_width; i++) {
        for (int j = 0; j < newImg_Height; j++, tempout++) {
            // round down
            int pre_x = i / number;
            int pre_y = j / number;
            if (pre_x > matrixWidth - 1) {
                pre_x = matrixWidth - 1;
            }
            if (pre_y > matrixWidth - 1) {
                pre_y = matrixWidth - 1;
            }
            *tempout = matrix[pre_x][pre_y];
        }
    }

    return (outimage);
}
```

- Nearest enlargement

Algorithm:

A	B
C	D

$\xrightarrow{3}$

A	A	B	B	B	
A	A	B	B	B	
C	C	D	D	D	
C	C	D	D	D	
C	C	D	D	D	

For each pixel of the new image $f(x_1, y_1)$, we can find the mapping point $g(x_2, y_2)$ in the old image.

$$\begin{cases} x_2 = \dfrac{x_1}{n} \\ y_2 = \dfrac{x_1}{n} \end{cases} \qquad n \text{ is the enlargement factor}$$

In this case, let x_2 and y_2 are both integer by using round function.
Then we could get $f(x_1, y_1) = g(x_2, y_2)$.

Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

Result after Nearest enlargement:

Result of processing "Bridge.pgm":

Source Image:

Result after Nearest enlargement:

Discussion:

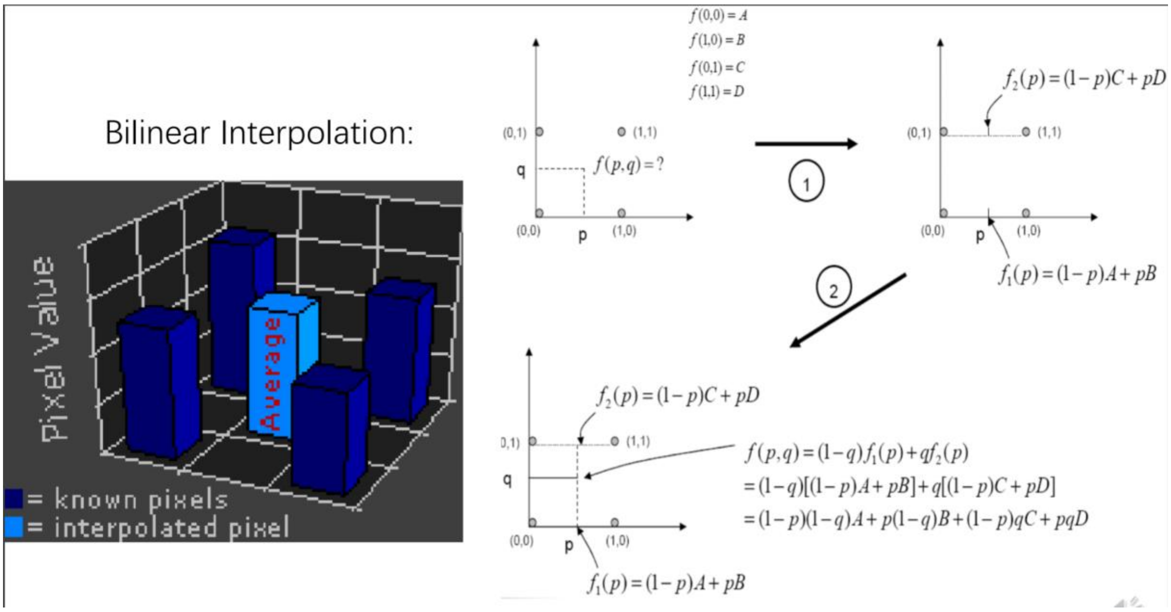
The algorithm is simple and fast, but the edge is badly serrated.

Codes:

```
Image *NearestNeighborImage(Image *image, float number) {        You, 5 hours ago • lab2
    unsigned char *tempin, *tempout;
    int size;
    Image *outimage;
    int newImg_Height = image->Height * number;
    int newImg_width = image->Width * number;
    outimage = CreateNewImage(image, newImg_Height, newImg_width, "#Nearest Neighbor Interpolation");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < newImg_width; i++) {
        for (int j = 0; j < newImg_Height; j++, tempout++) {
            // round
            int pre_x = round(i / number);
            int pre_y = round(j / number);
            if (pre_x > matrixWidth - 1) {
                pre_x = matrixWidth - 1;
            }
            if (pre_y > matrixWidth - 1) {
                pre_y = matrixWidth - 1;
            }
            *tempout = matrix[pre_x][pre_y];
        }
    }
```

- Bilinear interpolation

Algorithm:



$$f(x,y) = (1-x)(1-y)A + x(1-y)B + (1-x)yC + xyD$$
$$A = f(0,0)$$
$$B = f(1,0)$$
$$C = f(0,1)$$
$$D = f(1,1)$$

Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

Result after Bilinear interpolation:

Result of processing "Bridge.pgm":

Source Image:

Result after Bilinear interpolation:

Discussion:

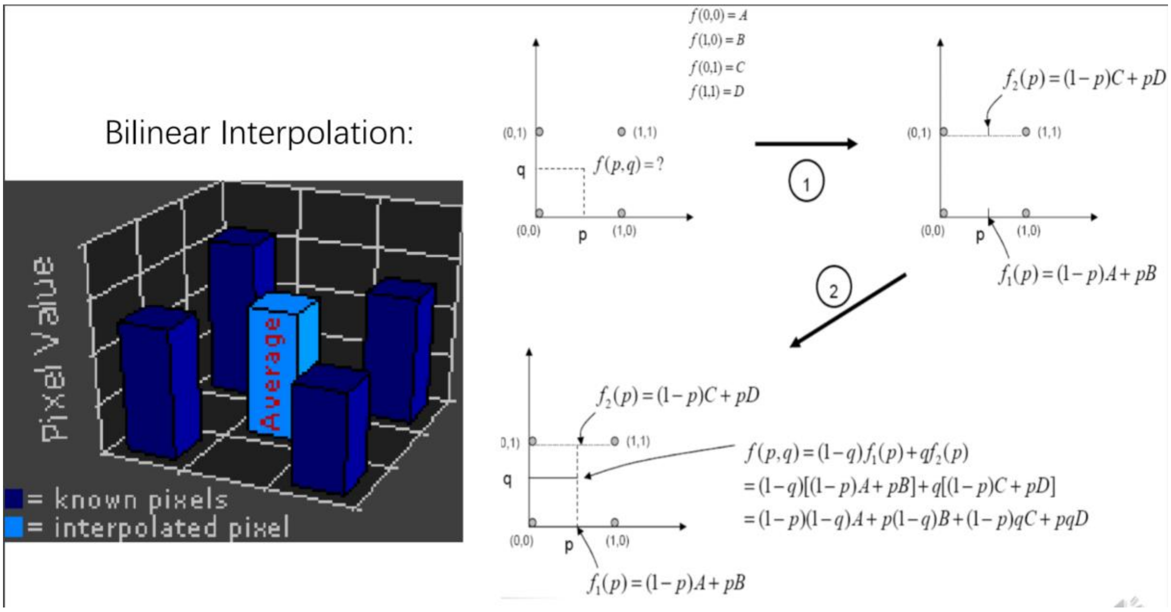
The interpolation result is smoother than the nearest neighbor, but the smoothness of the details is equal to the appearance of blurry.

Codes:

```
Image *BilinearInterpolationImage(Image *image, float number) {
    unsigned char *tempin, *tempout;
    int size;
    Image *outimage;
    int newImg_Height = image->Height * number;
    int newImg_width = image->Width * number;
    outimage = CreateNewImage(image, newImg_Height, newImg_width, "#Bilinear Interpolation");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < newImg_width; i++) {
        for (int j = 0; j < newImg_Height; j++, tempout++) {
            float p = i / number;
            float q = j / number;
            int new_pre_x = (int)p;
            int new_pre_y = (int)q;
            p = 1 / p;
            q = 1 / q;
            if (new_pre_x > matrixWidth - 2) {
                new_pre_x = matrixWidth - 2;
            }
            if (new_pre_y > matrixWidth - 2) {
                new_pre_y = matrixWidth - 2;
            }
            int piont1 = matrix[new_pre_x][new_pre_y];
            int piont2 = matrix[new_pre_x][new_pre_y + 1];
            int piont3 = matrix[new_pre_x + 1][new_pre_y];
            int piont4 = matrix[new_pre_x + 1][new_pre_y + 1];

            *tempout = (1 - p) * (1 - q) * piont1 + (1 - p) * q * piont2 + p * (1 - q) * piont3 + p * q * piont4;
        }
    }

    return (outimage);
}
```

- Fractional linear expansion to expand images to any larger size

Algorithm:



$$f(x,y) = (1-x)(1-y)A + x(1-y)B + (1-x)yC + xyD$$
$$A = f(0,0)$$
$$B = f(1,0)$$
$$C = f(0,1)$$
$$D = f(1,1)$$

Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

Result after Bilinear interpolation:

Result of processing “Bridge.pgm”:

Source Image:

Result after Bilinear interpolation:

Discussion:

The interpolation result is smoother than the nearest neighbor, but the smoothness of the details is equal to the appearance of blurry.

Codes:

```
Image *BilinearInterpolationImage(Image *image, float number) {
    unsigned char *tempin, *tempout;
    int size;
    Image *outimage;
    int newImg_Height = image->Height * number;
    int newImg_width = image->Width * number;
    outimage = CreateNewImage(image, newImg_Height, newImg_width, "#Bilinear Interpolation");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < newImg_width; i++) {
        for (int j = 0; j < newImg_Height; j++, tempout++) {
            float p = i / number;
            float q = j / number;
            int new_pre_x = (int)p;
            int new_pre_y = (int)q;
            p = 1 / p;
            q = 1 / q;
            if (new_pre_x > matrixWidth - 2) {
                new_pre_x = matrixWidth - 2;
            }
            if (new_pre_y > matrixWidth - 2) {
                new_pre_y = matrixWidth - 2;
            }
            int piont1 = matrix[new_pre_x][new_pre_y];
            int piont2 = matrix[new_pre_x][new_pre_y + 1];
            int piont3 = matrix[new_pre_x + 1][new_pre_y];
            int piont4 = matrix[new_pre_x + 1][new_pre_y + 1];

            *tempout = (1 - p) * (1 - q) * piont1 + (1 - p) * q * piont2 + p * (1 - q) * piont3 + p * q * piont4;
        }
    }

    return (outimage);
}
```

3. Perform negative image operation:

- On gray images

Algorithm:

We suppose the processed image is $g(x_2, y_2)$, and the original image is $f(x_1, y_1)$.

So $g(x_2, y_2) = 255 - f(x_1, y_1) \quad where\ x_1 = x_2,\ y_1 = y_2$

Results (including pictures):

Result of processing "Lena.pgm":

Source Image:

Result after negative:

Result of processing "Bridge.pgm":

Source Image:

Result after negative:

Discussion:

Enhance white or gray detail embedded in dark regions of an image, especially when the black areas are dominant in size.

Codes:

```
Image *NegativeImage(Image *image) {
    unsigned char *tempin, *tempout;
    int size;
    Image *outimage;
    outimage = CreateNewImage(image, image->Width, image->Height, "#Negative");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixWidth; j++, tempout++) {
            *tempout = 255 - matrix[i][j];
        }
    }

    return (outimage);
}
```