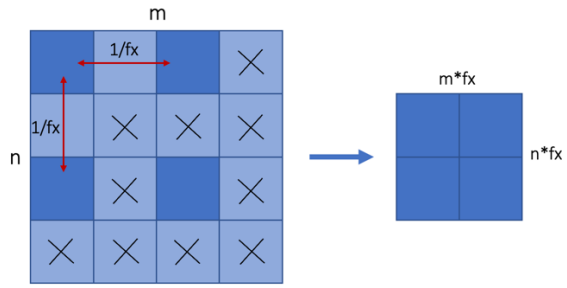
2. Image Reduction, Enlargement and Negative

1. Image reduction(lena, bridge, noise):

- Alternative line reduction

Algorithm:



Assume that the original image size is $m * n$, and the scale to reduce is $fx\ where\ (0 < fx < 1)$, and the output size will be $(m * fx) * (n * fx)$.

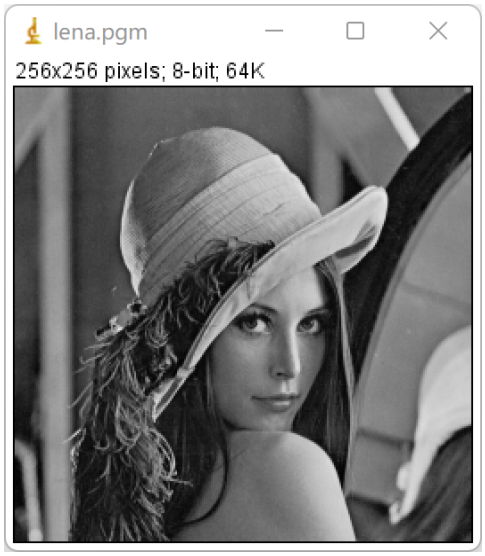
In the process, the output image will take the next pixel to be recorded after every $round(\frac{1}{fx})$ pixels distance (both row and column). So the pixels in the interval are ignored.

In the following code, it reduces the image to 1/2 the size fixedly to demonstrate the principle.

Results (including pictures)(Ratio=0.5):

Process result of "lena.pgm":

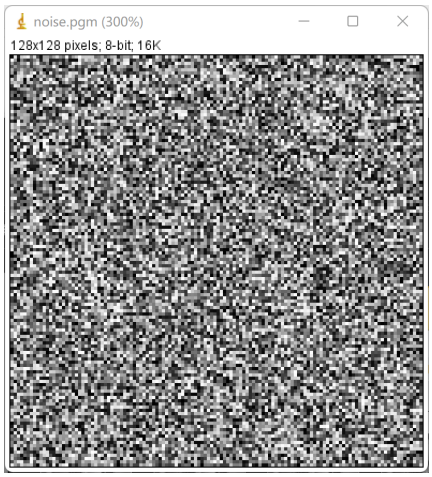
Source Image:



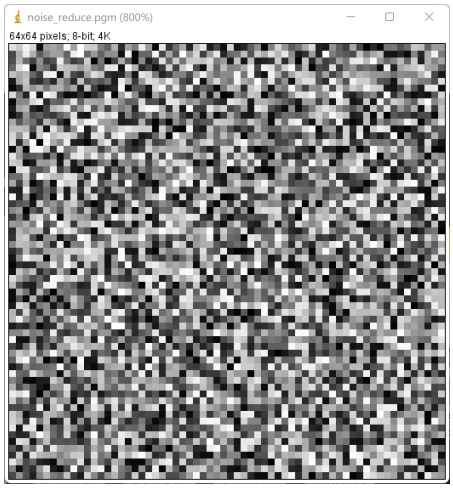
Result after alternative line reduction:

Process result of "noise.pgm":

Source Image:

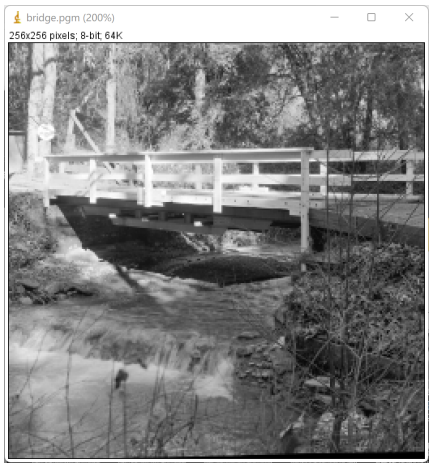


Result after alternative line reduction:



Process result of "bridge.pgm" :

Source Image:



Result after alternative line reduction:

Discussion:

This algorithm selects and discards pixels in each row and column according to the scale of the image to reduce its size. So all the pixels in the output are exactly from the original image.

This method is simple and effective to determine the new pixels, but the missing of some lines of pixels makes the image jagged, especially the edges of objects become very uneven.

Codes:

```
81  Image *ImageReduce_AlternativeLine(Image *image, float ratio) {
82      // Image Reduction: Alternative line method.
83      // This algorithm is to fixedly reduce the image to 1/2 size.
84      unsigned char *tempin, *tempout;
85      Image *outimage;
86      int row = 0, column = 0;
87
88      outimage = CreateNewImage(image, (char*)"#testing Reduction", 1, 0.5);
89      tempin = image->data;
90      tempout = outimage->data;
91
92      for(int i = 0; i < image->Height/2 - 1; i++) {
93          for(int j = 0; j < image->Width/2 - 1; j++) {
94              tempout[(outimage->Width)*i + j] = tempin[(image->Width)*row + column];
95              column += 2;
96          }
97          row += 2;
98          column = 0;
99      }
100     return (outimage);
101 }
```

- Fractional linear reduction to reduce images

Algorithm:

We define fx as the scaling of the output image (when $0 < fx < 1$ is to reduce).

And the corresponding relationship between the output image and the original image will be:

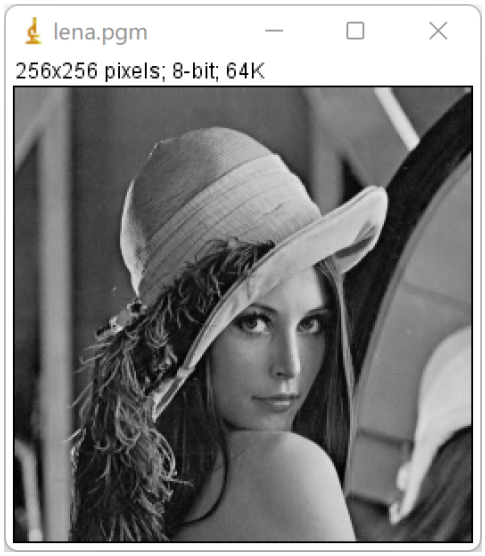
$$dst(x, y) = src(round\left(\frac{x}{fx}\right), round\left(\frac{y}{fx}\right)).$$

The following code can reduce the image to any smaller size according to the input ratio.

Results (including pictures) (Ratio=0.2):

Process result of "lena.pgm":

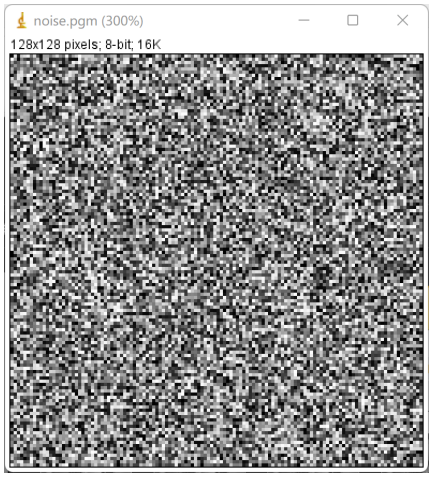
Source Image:



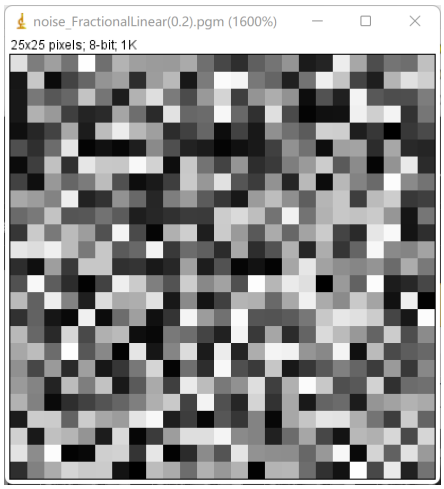
Result after fractional linear reduction:

Process result of "noise.pgm":

Source Image:



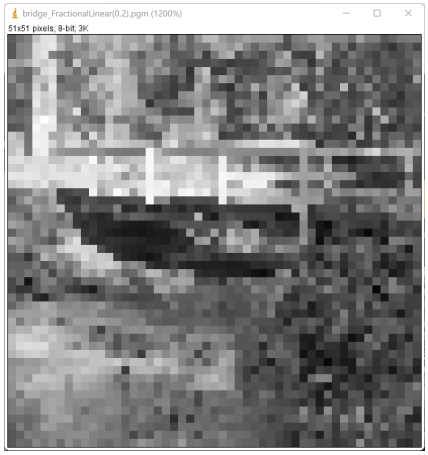
Result after fractional linear reduction:



Process result of "bridge.pgm"

Source Image:

Result after fractional linear reduction:



Discussion:

This algorithm selects pixels at the corresponding position of the original image according to the reduction ratio of the image, so all the pixels in the image also come from the original image.
It is simple to take the pixels and it can reduce the image to any size we want, but we can clearly see that the image becomes blurry and jagged.

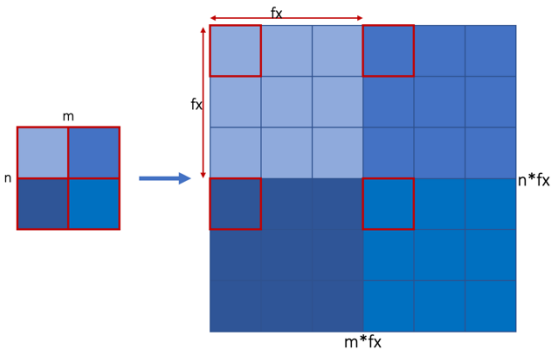
Codes:

```
59  // Algorithms Code:
60  Image *Image_FractionalLinear(Image *image, float ratio) {
61      // Fractional linear method.
62      // This algorithm can resize the image to any size(Both enlarge and reduce).
63      unsigned char *tempin, *tempout;
64      Image *outimage;
65      int x_in, y_in;// relative position coordinates of the input image
66
67      outimage = CreateNewImage(image, (char*)"#testing Enlargement", 3, ratio);
68      tempin = image->data;
69      tempout = outimage->data;
70
71      for(int i = 0; i < outimage->Height; i++) {
72          for(int j = 0; j < outimage->Width; j++) {
73              x_in = round((float)j / ratio);
74              y_in = round((float)i / ratio);
75              tempout[outimage->Width * i + j] = tempin[image->Width * y_in + x_in];
76          }
77      }
78      return (outimage);
79  }
```

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

- Pixel replication

Algorithm:



Assume that the original image size is $m * n$, and the scale to enlarge is $fx\ where\ (fx > 1)$, and the output size will be $(m * fx) * (n * fx)$.

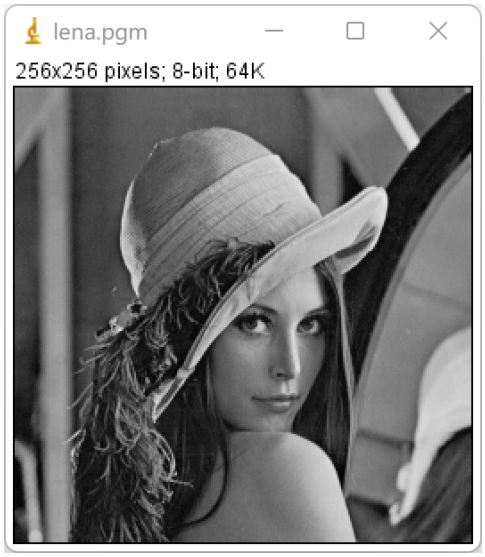
In the process, each pixel in the original image will occupy $\left(round(fx) * round(fx)\right)$ pixels after being enlarged, and the color will be exactly same as original.

In the following code, it enlarges the image 3 times fixedly to demonstrate the principle.

Results (including pictures) (Ratio=3.0):

Process result of "lena.pgm":

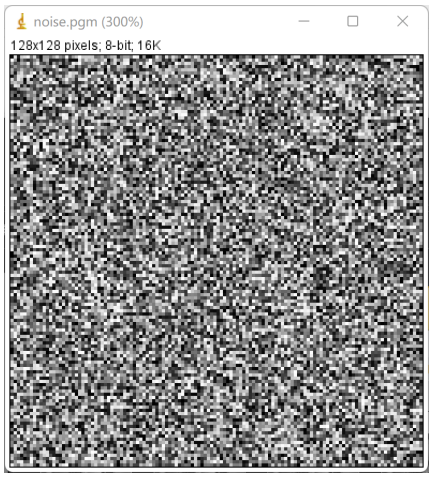
Source Image:



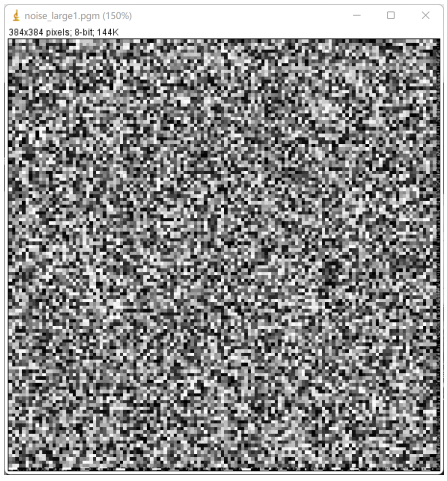
Result after pixel replication enlargement:

Process result of "noise.pgm":

Source Image:



Result after pixel replication enlargement:



Process result of “bridge.pgm”

Source Image:

Result after pixel replication enlargement:

Discussion:

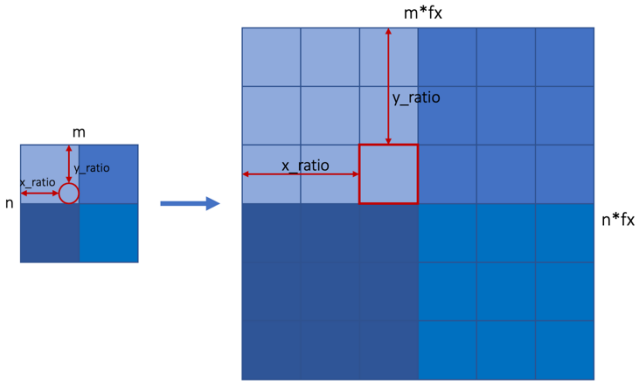
This algorithm scales up each pixel in the original image by a proportional amount. It does not add information that doesn't exist in the original image. So we can find that it just enlarges the image size without making it any clearer.

Codes:

```
103 Image *ImageEnlarge_PixelRep(Image *image, float ratio) {
104     // Image Enlargement: Pixel replication method.
105     // This algorithm is to fixedly enlarge the image to twice the size.
106     unsigned char *tempin, *tempout;
107     Image *outimage;
108     int row = 0, column = 0;
109 
110     outimage = CreateNewImage(image, (char*)"#testing Enlargement", 2, 3);
111     tempin = image->data;
112     tempout = outimage->data;
113 
114     for(int i = 0; i < image->Height - 1; i++) {
115         for(int j = 0; j < image->Width - 1; j++) {
116             unsigned char currPixel = tempin[(image->Width)*i + j];
117             for(int x = row; x <= row+2; x++) {
118                 for(int y = column; y <= column+2; y++) {
119                     tempout[(outimage->Width)*x + y] = currPixel;
120                 }
121             }
122             column += 3;
123         }
124         row += 3;
125         column = 0;
126     }
127     return (outimage);
128 }
```

- Nearest enlargement

Algorithm:



Assume that the original image size is $m * n$, and the scale to enlarge is $fx\ where\ (fx > 1)$, and the output size will be $(m * fx) * (n * fx)$.

We record the relative positions (horizontal and vertical) of the output pixels as x_ratio and y_ratio, and use them to find the corresponding relative positions of the original image pixels.

The algorithm is:

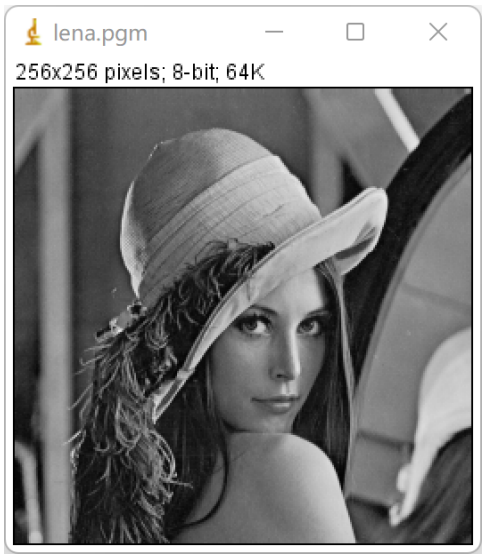
$$x_{ratio} = \frac{dst(y)}{m * fx},\ y_{ratio} = \frac{dst(x)}{n * fx}.$$

So $dst(x, y) = src(round(y_{ratio} * n * fx),\ round(x_{ratio} * m * fx))$.

Results (including pictures) (Ratio=3.0):

Process result of "lena.pgm" :

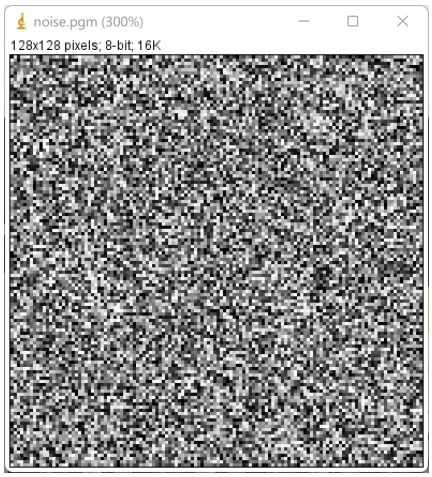
Source Image:



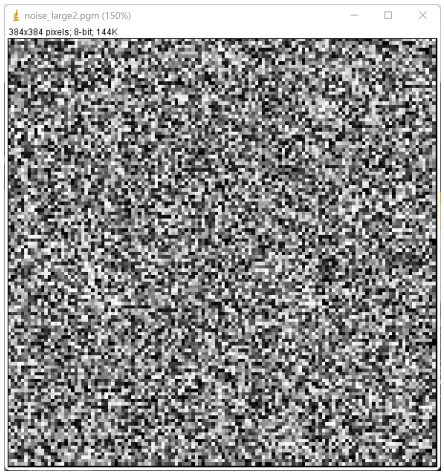
Result after nearest enlargement:

Process result of "noise.pgm" :

Source Image:



Result after nearest enlargement:



Process result of “bridge.pgm”

Source Image:

Result after nearest enlargement:

Discussion:

This algorithm finds the closest pixel in the original image based on the relative position of the new pixel in the image. So it does not add information that doesn't exist in the original image. This method takes up little memory and has no color distortion. However, the enlarged image has obvious aliasing, which is to simply copy the pixel information in the original image to the new image. It also doesn't make the image any clearer.

Codes:

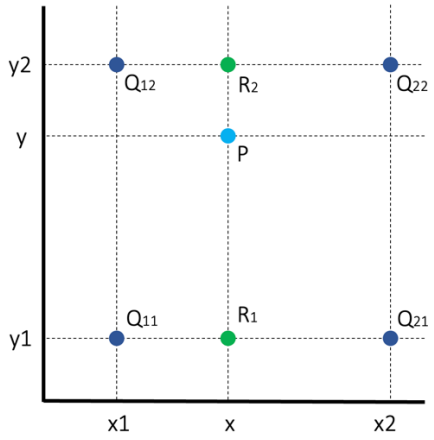
```
Image *ImageEnlarge_Nearest(Image *image, float ratio) {
    // Image Enlargement: Nearest enlargement method.
    // This algorithm can enlarge the image to any size.
    unsigned char *tempin, *tempout;
    Image *outimage;
    float ratio_row, ratio_column;

    // If the input ratio < 1, then automatically enlarge to 3 times:
    if(ratio <= 1) outimage = CreateNewImage(image, (char*)"#testing Enlargement", 2, 3);
    else outimage = CreateNewImage(image, (char*)"#testing Enlargement", 3, ratio);
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < outimage->Height - 1; i++) {
        ratio_column = (float)(i) / (float)outimage->Height;
        for(int j = 0; j < outimage->Width - 1; j++) {
            ratio_row = (float)(j) / (float)outimage->Width;
            int temp = image->Width * round(image->Height * ratio_column) + round(image->Width *
                ratio_row);
            tempout[outimage->Width * i + j] = tempin[temp];
        }
    }
    return (outimage);
}
```

- Bilinear interpolation

Algorithm:



As shown above, Q11, Q12, Q21, Q22 are known pixels and P is the point to be interpolated.
First, interpolate the two points R1, R2 on the x-axis:

$$f(R_1) \approx \frac{x_2-x}{x_2-x_1} f(Q_{11}) + \frac{x-x_1}{x_2-x_1} f(Q_{21}), where\ R_1 = (x, y_1),$$

$$f(R_2) \approx \frac{x_2-x}{x_2-x_1} f(Q_{12}) + \frac{x-x_1}{x_2-x_1} f(Q_{22}), where\ R_2 = (x, y_2).$$

Then interpolate point P based on R1 and R2:

$$f(P) \approx \frac{y_2-y}{y_2-y_1} f(R_1) + \frac{y-y_1}{y_2-y_1} f(R_2).$$

After simplification:

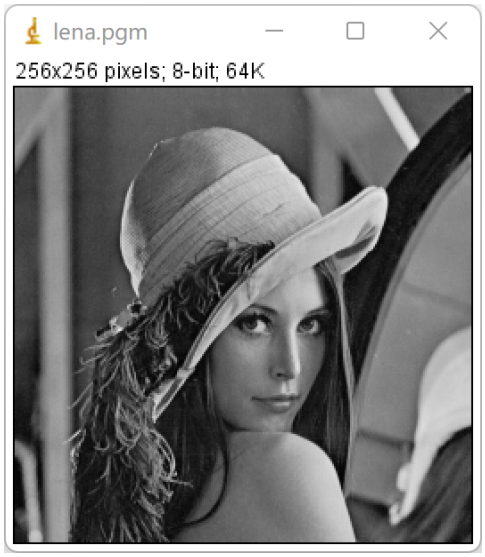
$$f(x,y) \approx \frac{(x_2-x)(y_2-y)}{(x_2-x_1)(y_2-y_1)} f(Q_{11}) + \frac{(x-x_1)(y_2-y)}{(x_2-x_1)(y_2-y_1)} f(Q_{21}) + \frac{(x_2-x)(y-y_1)}{(x_2-x_1)(y_2-y_1)} f(Q_{12}) + \frac{(x-x_1)(y-y_1)}{(x_2-x_1)(y_2-y_1)} f(Q_{22})$$

The following code can enlarge the image to any bigger size according to the input ratio.

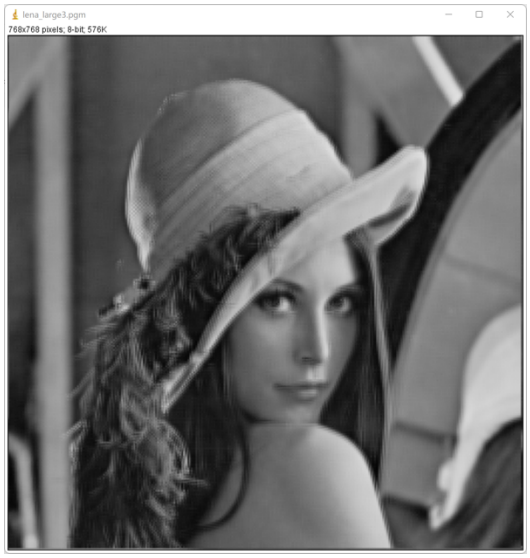
Results (including pictures) (Ratio=3.0):

Process result of “lena.pgm”:

Source Image:

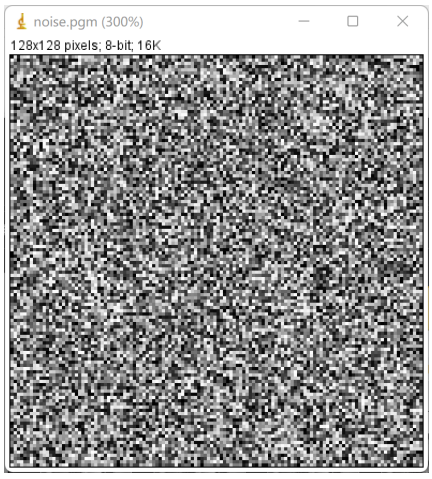


Result after bilinear interpolation enlargement:

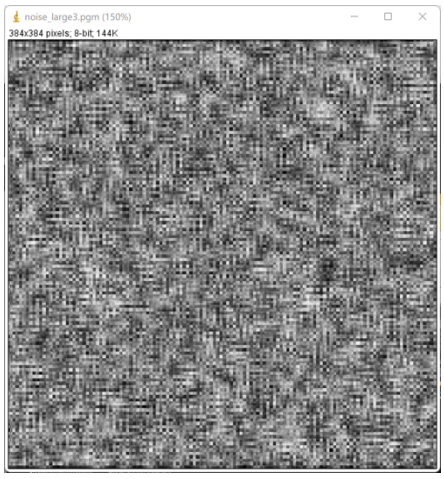


Process result of “noise.pgm”:

Source Image:



Result after bilinear interpolation enlargement:



Process result of “bridge.pgm”

Source Image:

Result after bilinear interpolation enlargement:

Discussion:

The bilinear interpolation algorithm is to multiply the value of the four pixels near the original sampling point by the weight to obtain the pixel information of the new image. So the resulting image contains pixels (weighted value) that don't exist in the original image, making the whole image look smoother.

This method can effectively anti-alias and enlarge the image to any size. However, it does not consider edge gradient changes and blur the image.

Codes:

```
Image *ImageEnlarge_Bilinear(Image *image, float ratio) {
    // Image Enlargement: Bilinear interpolation method.
    // This algorithm can enlarge the image to any size.
    unsigned char *tempin, *tempout;
    Image *outimage;
    int x1, x2, y1, y2;// coordinate of known pixels
    unsigned char color1, color2, color3, color4;// color of known pixels
    float x_in, y_in;

    // If the input ratio < 1, then automatically enlarge to 3 times:
    if(ratio <= 1) outimage = CreateNewImage(image, (char*)"#testing Enlargement", 2, 3);
    else outimage = CreateNewImage(image, (char*)"#testing Enlargement", 3, ratio);
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < outimage->Height; i++) {
        for(int j = 0; j < outimage->Width; j++) {
            if(ratio <= 1) {
                // In case the input ratio <1.
                x_in = (float)j / 3;
                y_in = (float)i / 3;
            }
            else {
                x_in = (float)j / ratio;
                y_in = (float)i / ratio;
            }
            x1 = (int)(x_in + 1);
            x2 = (int)(x_in - 1);
            y1 = (int)(y_in + 1);
            y2 = (int)(y_in - 1);
            color1 = tempin[image->Width * y1 + x1];
            color2 = tempin[image->Width * y2 + x1];
            color3 = tempin[image->Width * y1 + x2];
            color4 = tempin[image->Width * y2 + x2];
            tempout[outimage->Width * i + j] = (unsigned char)(((color1*(x2-x_in)*(y2-y_in)) +
                (color2*(x2-x_in)*(y_in-y1)) + (color3*(x_in-x1)*(y2-y_in)) +
                (color4*(x_in-x1)*(y_in-y1))) / ((x2-x1) * (y2-y1)));

        }
    }
    return (outimage);
}
```

- Fractional linear expansion to expand images to any larger size

Algorithm:

We define fx as the scaling of the output image (when $fx > 1$ is to enlarge).

And the corresponding relationship between the output image and the original image will be:

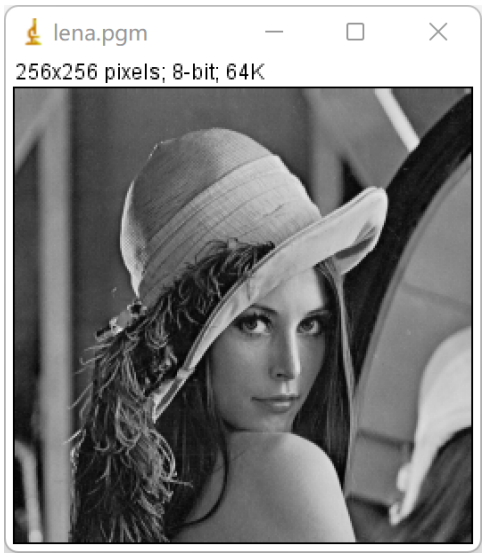
$$dst(x,y) = src(round\left(\frac{x}{fx}\right), round\left(\frac{y}{fx}\right)).$$

The following code can enlarge the image to any bigger size according to the input ratio.

Results (including pictures) (Ratio=10):

Process result of "lena.pgm" :

Source Image:

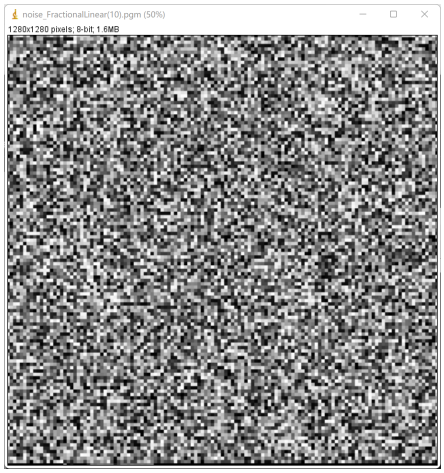


Result after fractional linear expansion:

Process result of “noise.pgm”:

Source Image:

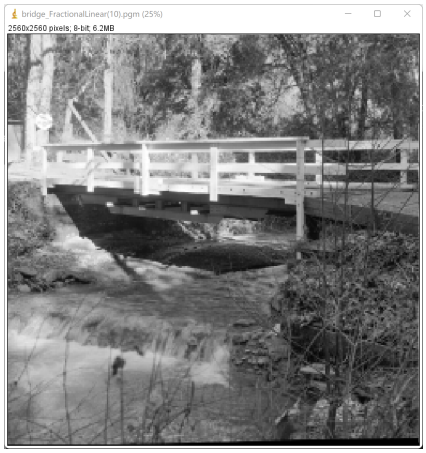
Result after fractional linear expansion:



Process result of “bridge.pgm”

Source Image:

Result after fractional linear expansion:



Discussion:

This algorithm selects pixels at the corresponding position of the original image according to the enlargement ratio of the image, so all the pixels in the image also come from the original image. It is simple to take the pixels and it can enlarge the image to any size we want, but it does not add any new detail in it. So it just enlarges the size without making the image clearer.

Codes:

```
59  // Algorithms Code:
60  Image *Image_FractionalLinear(Image *image, float ratio) {
61      // Fractional linear method.
62      // This algorithm can resize the image to any size(Both enlarge and reduce).
63      unsigned char *tempin, *tempout;
64      Image *outimage;
65      int x_in, y_in;// relative position coordinates of the input image
66
67      outimage = CreateNewImage(image, (char*)"#testing Enlargement", 3, ratio);
68      tempin = image->data;
69      tempout = outimage->data;
70
71      for(int i = 0; i < outimage->Height; i++) {
72          for(int j = 0; j < outimage->Width; j++) {
73              x_in = round((float)j / ratio);
74              y_in = round((float)i / ratio);
75              tempout[outimage->Width * i + j] = tempin[image->Width * y_in + x_in];
76          }
77      }
78      return (outimage);
79  }
```

3. Perform negative image operation:

- On gray images

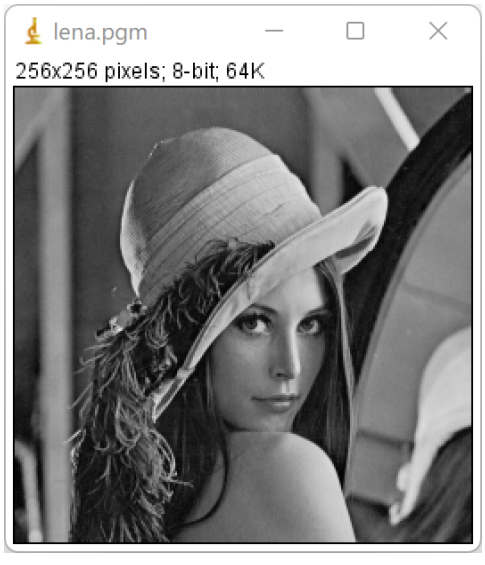
Algorithm:

$$dst(x, y) = 255 - src(x, y).$$

Results (including pictures):

Process result of "lena.pgm" :

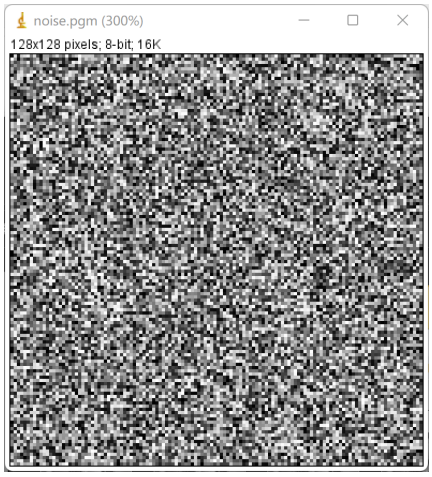
Source Image:



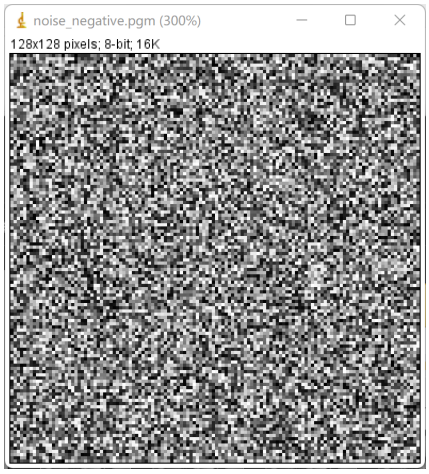
Result after negative operation:

Process result of “noise.pgm” :

Source Image:



Result after negative operation:



Process result of “bridge.pgm”

Source Image:

Result after negative operation:

Discussion:

The quantization level of each gray color is 256, then the value of each pixel in the original image is subtracted from 255 as the color of the new image.

After processing, the layers of the picture are more distinct, and the color contrast is more intense. So we can see more details in the new images.

Codes:

```
194 Image *Negative(Image *image) {
195     // Perform negative image operation.
196     unsigned char *tempin, *tempout;
197     Image *outimage;
198     int size;
199 
200     outimage = CreateNewImage(image, (char*)"#testing Nagetive", 0, 1);
201     tempin = image->data;
202     tempout = outimage->data;
203 
204     if(image->Type == GRAY) size = image->Width * image->Height;
205     else if(image->Type == COLOR) size = image->Width * image->Height * 3;
206 
207     for(int i = 0; i < size; i++) {
208         *tempout = 255 - *tempin;
209         tempin++;
210         tempout++;
211     }
212     return (outimage);
213 }
214 // Algorithms End.
```