**Project 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, noise):**

* Alternative line reduction

**Algorithm:**

Suppose it is reduced by n times

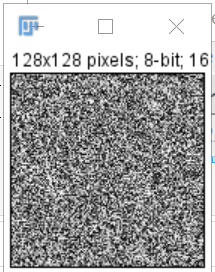
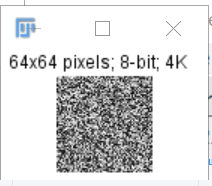
Replace the pixels of the new image in (u,v) with the pixel of the original image in (nu, nv)

Just like below:

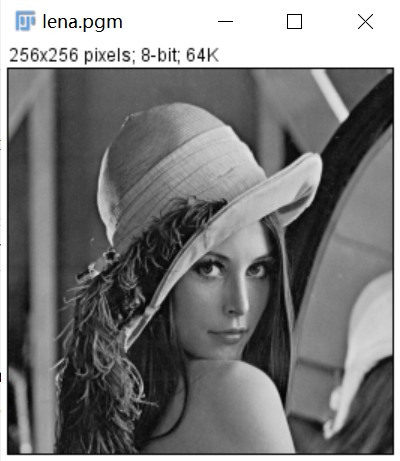
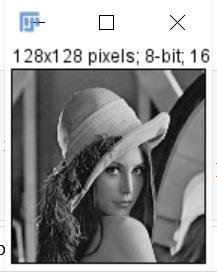
outData[i\*width + j] = inData[n \*image->Width\*i + n\*j];

**Results (including pictures):**

Source: after:

Source: after:

**Discussion:**

Because we only take the pixels with corresponding multiple intervals in the original image, it is obvious that the image is reduced according to the corresponding multiple

**Codes:**

Image \*AlternateReduction(Image \* image)

{

unsigned char \*tempin, \*temptout;

int i, j, size;

Image \*outimage;

int width = image->Width / n;//宽¨ª缩?小?n倍À?

int height = image->Height / n;//长¡è缩?小?倍À?

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

temptout = outimage->data;

if (image->Type == GRAY)

size = width\*height;

else if (image->Type == COLOR)

size = width\*height \* 3;

for (i = 0; i < height; i++)

{

for (j = 0; j < height; j++)

{

temptout[i\*width + j] = tempin[n \*image->Width\*i + n\*j];

}

}return(outimage);

}

* Fractional linear reduction to reduce images

**Algorithm:**

Assuming that the reduced array is a and the original image array is b, there are 7 rows and the reduction multiple is 1.2 times, then a row will contain 7 ÷ 1.2 ≈ 5.8, which can be taken as 5, a [0] = b [0], a [1] = b [1] and use subscript / multiple to calculate the reduced pixel value which corresponds to the original image.

0×1.2=0, we take b[0];

1×1.2=1.2, we take b[1];

2×1.2=2.4, we take b[2];

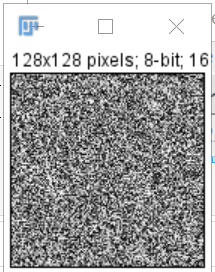
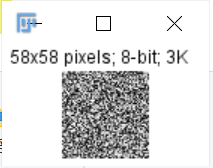
……

Just as above:

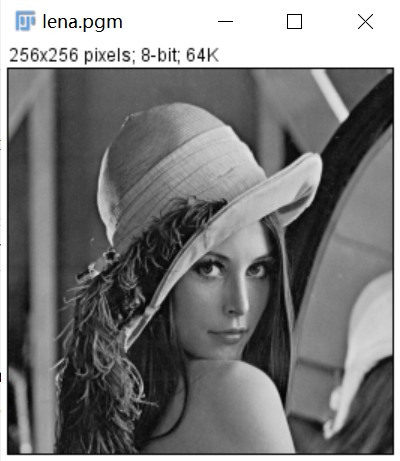
tempout[i\*width + j] = tempin[(int)(floor((float)(m2 \*i))\*image->Width + floor(m2\*j))];

**Results (including pictures):**

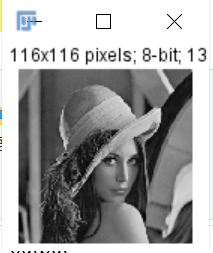
Source: after:

Source:



after:



**Discussion:**

The gray value of the image cannot be decimal. When we reduce it by non-integer times, we can only take the upper bound, lower bound or rounding. In the code, I take the lower bound. As we can see, the picture can be reduced to any multiple.

**Codes:**

Image \*fractionalReduction(Image \*image){

unsigned char \*tempin, \*tempout;

int size, i, j;

int height = floor((float)image->Height/m2);

int width = floor((float)image->Width/m2);

Image \*outimage;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

if (image->Type == GRAY)

size = width\*height;

else if (image->Type == COLOR)

size = width\*height \* 3;

for (i = 0; i < height; i++)

{

for (j = 0; j < height; j++)

{

tempout[i\*width + j] = tempin[(int)(floor((float)(m2 \*i))\*image->Width + floor(m2\*j))];

}

}return(outimage);

}

1. **image enlargement (lena, noise):**

* Pixel replication

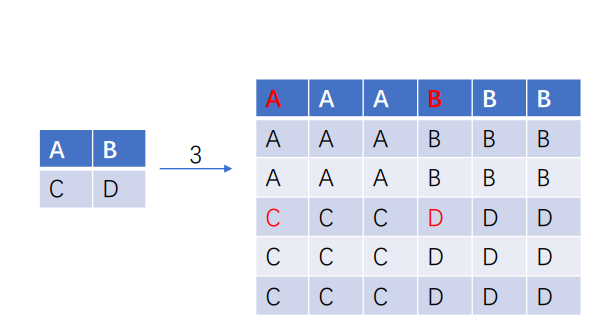
**Algorithm:**

Suppose it is enlarged by n times

Expand the area in (u,v) to a n × n window area and fill with the pixel of the original

image in (u, v).

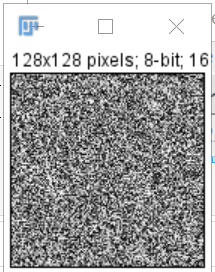
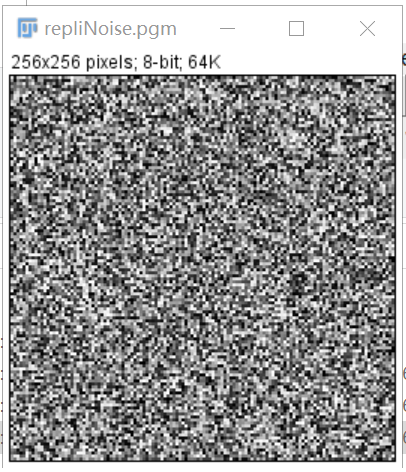
Just as below:



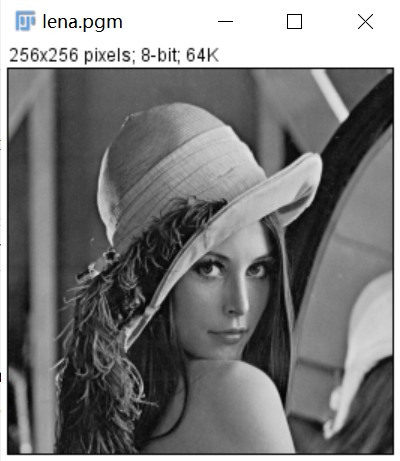
outData[(i\*m + p)\*width + j\*m + q] = inData[i \*image->Width + j];

**Results (including pictures):**

Source: after:

Source:



after:



**Discussion:**

Because we have expanded the pixel area of the original image, it is obvious that the image has become larger.

**Codes:**

Image \*Pixel\_replication(Image \*image)

{

unsigned char \*tempin, \*tempout;

int size, i, j, p, q;

int height = image->Height\*m;

int width = image->Width\*m;

Image \*outimage;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

for (i = 0; i < image->Height; i++)

{

for (j = 0; j < image->Width; j++)

{

for (p = 0; p < m; p++)

{

for (q = 0; q < m; q++)

{

tempout[(i\*m + p)\*width + j\*m + q] = tempin[i \*image->Width + j];

}

}

}

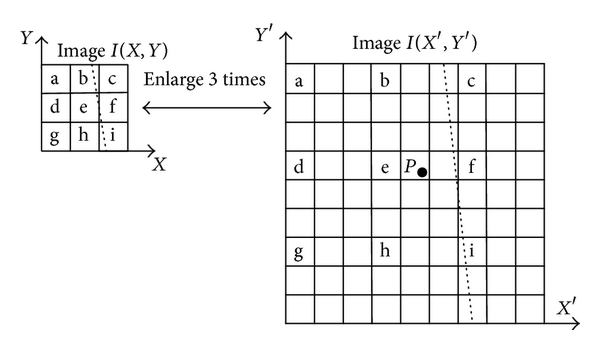
}

return(outimage);

}

* Nearest enlargement

**Algorithm:**

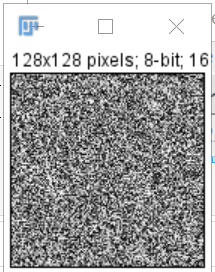
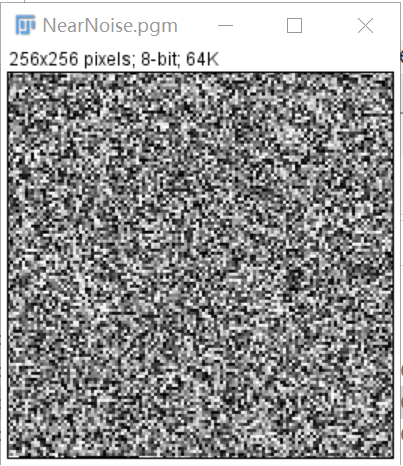


Nearest-neighbor interpolation algorithm is to calculate the point in the image and its surrounding pixels e, f, h, i and the distance and then choose the shortest distance between the gray values of the pixels, as their gray values.

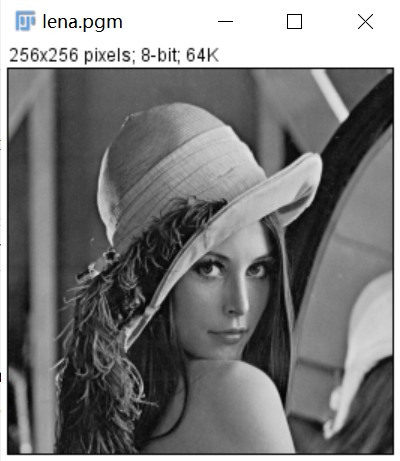
outData[i\*width + j] = inData[p\*image->Width + q];

**Results (including pictures):**

Source: after:

Source:



after:



**Discussion:**

The enlargement process using the nearest-neighbor interpolation algorithm will be encounter with the problem of blocking the effect.

The function is the simplest and most efficient interpolation algorithm. However, since it is easier to calculate its results in lower image quality, enlarged images usually display jagged and blocked features.

**Codes:**

Image \*Nearest\_enlargement(Image \*image)

{

unsigned char \*tempin, \*tempout;

int size, i, j, p, q;

int height = image->Height\*m;

int width = image->Width\*m;

Image \*outimage;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

float M = (float)m;

for (i = 0; i < height; i++)

{

for (j = 0; j < width; j++)

{

int p = ceil(i / M);

int q = ceil(j / M);

tempout[i\*width + j] = tempin[p\*image->Width + q];

}

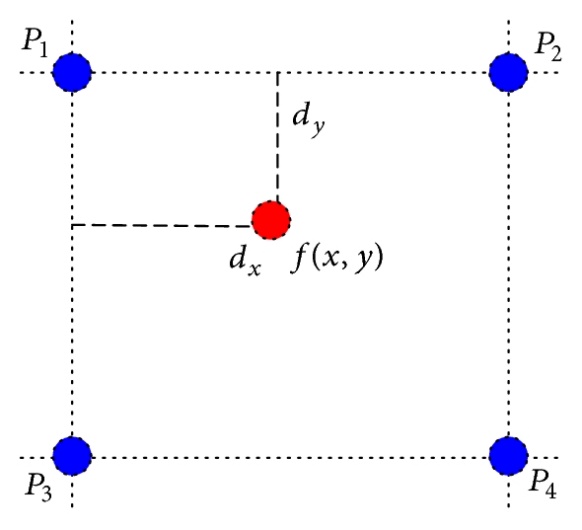
}return(outimage);

}

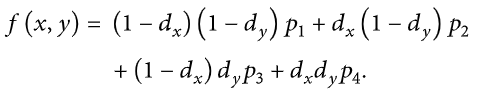
* Bilinear interpolation

**Algorithm:**

Bilinear interpolation considers the closest 2 × 2 neighborhood of known pixel values surrounding the unknown pixel’s computed location. It then takes a weighted average of these 4 pixels to arrive at its final, interpolated value. The weight on each of the 4 pixel values is based on the computed pixel’s distance from each of the known points.

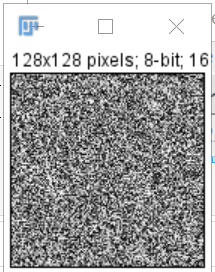
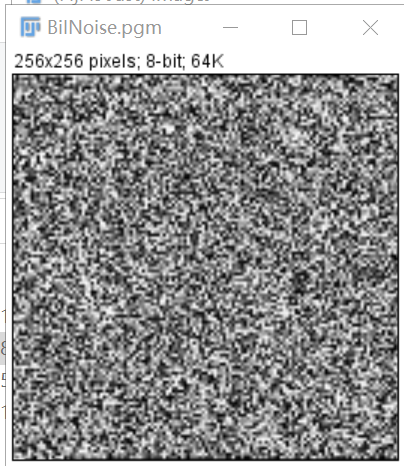


Interpolation is calculated as follows:

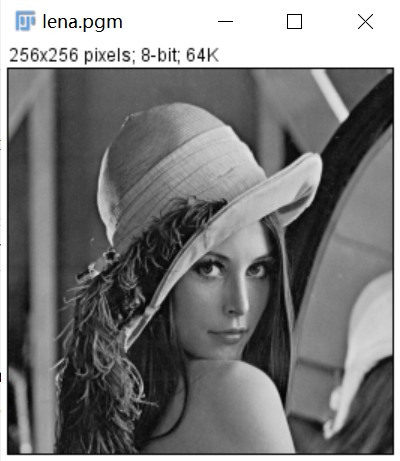


**Results (including pictures):**

Source: after:

Source:



after:



**Discussion:**

Bilinear interpolation can be used where perfect image transformation with pixel matching is impossible, so that one can calculate and assign appropriate intensity values to pixels. Unlike other interpolation techniques such as nearest neighbor interpolation, bilinear interpolation uses only the 4 nearest pixel values which are located in diagonal directions from a given pixel in order to find the appropriate color intensity values of that pixel.

**Codes:**

Image \*Bilinear\_interpolation(Image \*image)

{

unsigned char \*tempin, \*tempout;

int i, j;

float p, q;

Image \*outimage;

int width = image->Width \* m;

int height = image->Height \* m;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

float M = (float)m;

for (i = 0; i < height; i++)

{

for (j = 0; j < width; j++)

{

//defeine 4 points

int x21 = floor(i / M);

int y21 = floor(j / M);

int x12 = ceil(i / M);

int y12 = ceil(j / M);

int x11 = x12 - 1;

int y11 = y12;

int x22 = x21 + 1;

int y22 = y21;

p = i / M - x21;

q = j / M - y21;

tempout[i\*width + j] = (1 - q)\*(1 - p)\*tempin[x21\*image->Width + y21] + p\*(1 - q)\*tempin[x22\*image->Width + y22] + (1 - p)\*q\*tempin[x12\*image->Width + y12] + p\*q\*tempin[x11\*image->Width + y11];

}

}

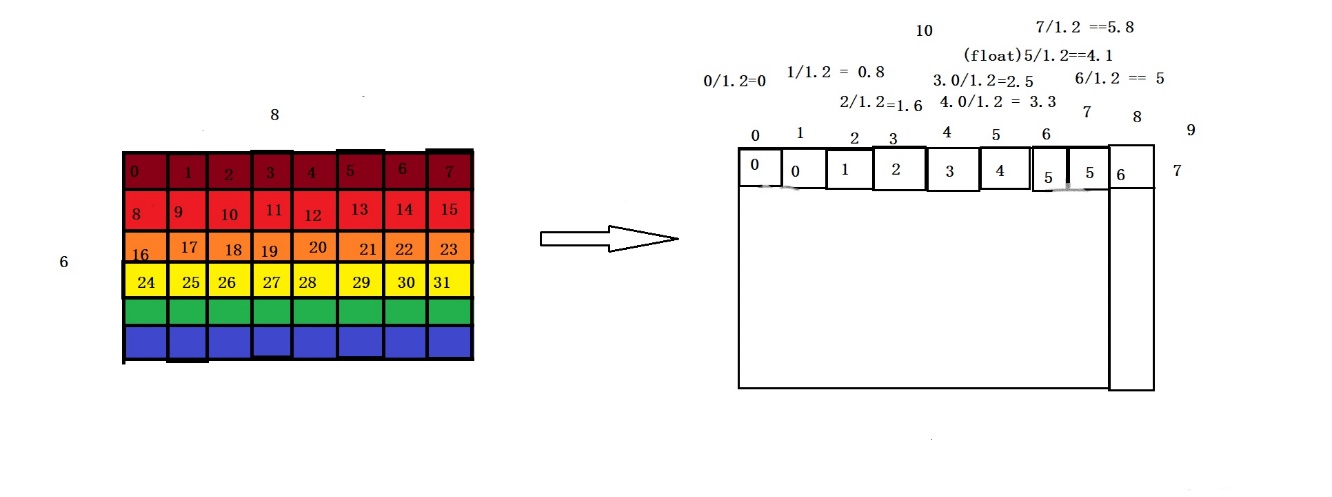
return(outimage);

}

* Fractional linear expansion to expand images to any larger size

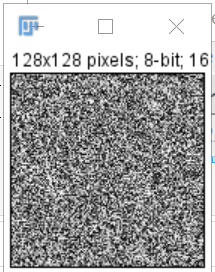
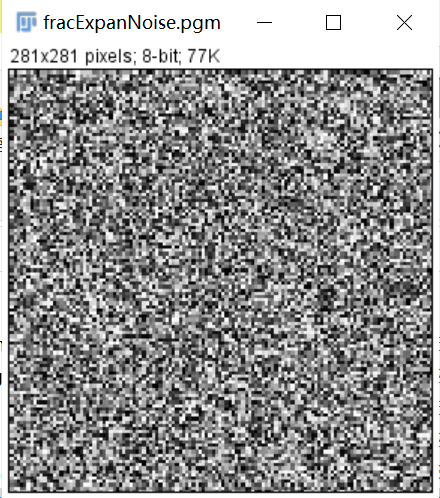
**Algorithm:**

Assuming that the enlarged array is a and the original image array is b, there are 7 rows and the magnification is 1.2 times, then a row will contain 7 \* 1.2 = 8.4, take 8, a [0] = b [0], a [1] = b [0], and use subscript / multiple to calculate the enlarged pixel value which corresponds to the original image.

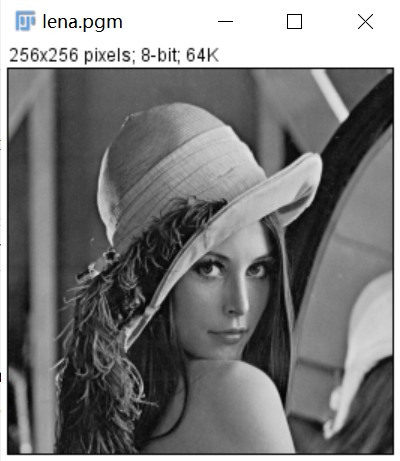


**Results (including pictures):**

Source: after:

Source:



after:



**Discussion:**

The gray value of the image cannot be decimal. When we enlarge it by non-integer times, we can only take the upper bound, lower bound or rounding. In the code, I take the lower bound. As we can see, the picture can be enlarged to any multiple.

**Codes:**

Image \*fractionalExpansion(Image \*image){

unsigned char \*tempin, \*tempout;

int size, i, j;

int height = floor((float)image->Height\*m2);

int width = floor((float)image->Width\*m2);

Image \*outimage;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

if (image->Type == GRAY)

size = width\*height;

else if (image->Type == COLOR)

size = width\*height \* 3;

for (i = 0; i < height; i++)

{

for (j = 0; j < height; j++)

{

tempout[i\*width + j] = tempin[(int)(floor((float)(i/m2))\*image->Width + floor(j/m2))];

}

}return(outimage);

}

1. **Perform negative image operation(lena,noise)**

* On gray images

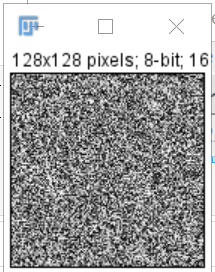
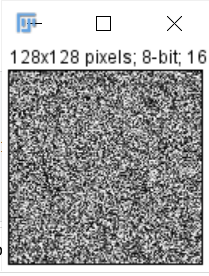
**Algorithm:**

g(x,y)=255-f(x,y)

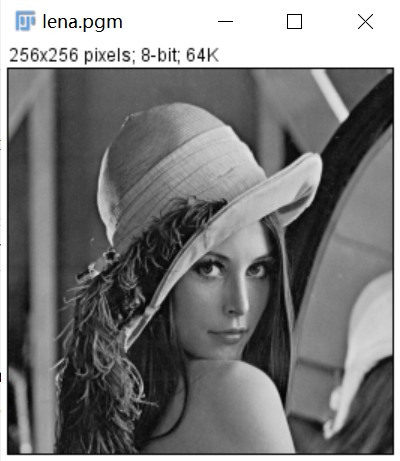
tempout[i\*width + j] = 255 - tempin[i\*width + j];

**Results (including pictures):**

Source: after:

Source: after:

**Discussion:**

It is obvious that black and white are reversed in lena.pgm. However, it's not so obvious in appearance in noise.pgm, because black and white itself is messy.

**Codes:**

Image \*negative\_image(Image \*image)

{

unsigned char \*tempin, \*tempout;

int i, j, size;

Image \*outimage;

int width = image->Width;

int height = image->Height;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

if (image->Type == GRAY)

{

size = width\*height;

for (i = 0; i < height; i++)

{

for(j = 0; j < width; j++)

{

tempout[i\*width + j] = 255 - tempin[i\*width + j];

}

}

}

else if (image->Type == COLOR)

{

size = width\*height \* 3;

for (i = 0; i < height; i++)

{

for (j = 0; j < width; j++)

{

for (int m = 0; m < 3; m++)

{

tempout[3 \* i\*width + 3 \* j + m] = 255 - tempin[3 \* i\*image->Width + 3 \* j + m];

}

}

}

}

return(outimage);

}