Image Interpolation Report

By Hewen Zhang

Southern University of Science and Technology

Email: 12010342@mail.sustech.edu.cn

1. Objective of the laboratory and the principle of the algorithm

The task of week is to design three algorithms in order to realize image interpolation (sometimes called resampling). Interpolation is practical in many aspects, like zooming in or out an image, image rotation and geometric correction. The algorithms mentioned above are based on three common methods of image interpolation, including nearest neighbor interpolation, bilinear interpolation and bicubic interpolation. They are mainly used for enlarging and shrinking image in this laboratory. The following figure shows the principle of the three ways of interpolation.

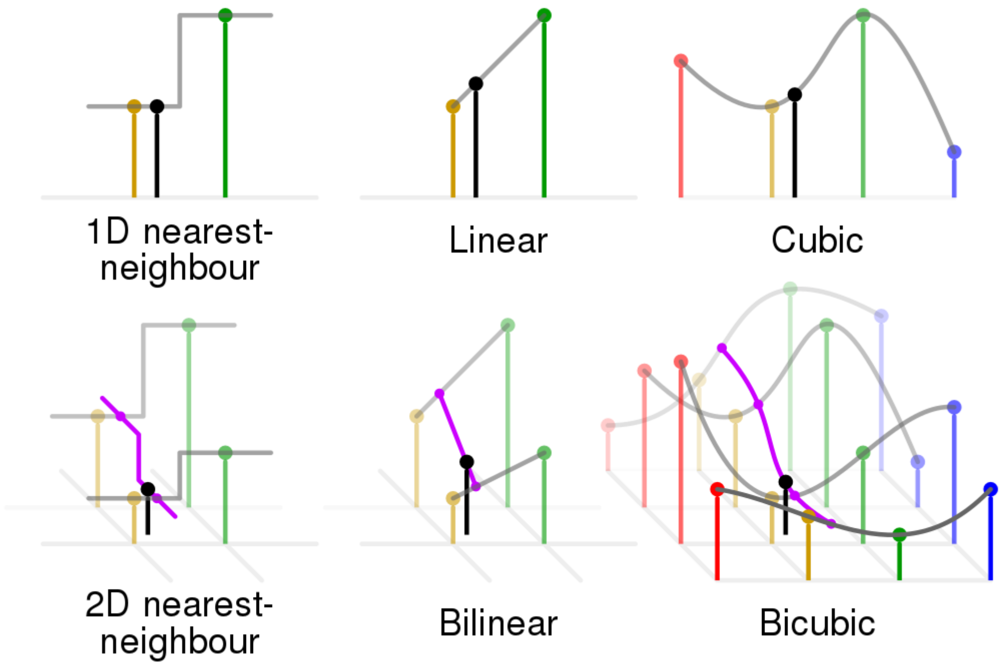

Fig. 1. the principles of nearest neighbor interpolation, linear interpolation, cubic interpolation and their forms in 2D

2. Pseudo code of the algorithms

1. pseudo code for 2D nearest neighbor interpolation

```
output ← np.zeros(dim)

hori_0 ← input.shape[0]-1
ver_0 ← input.shape[1]-1
hori_1 ← input.shape[0]-1
ver_1 ← input.shape[1]-1

for i ← 0 to dim[0]
    for j ← 0 to dim[1]
        output[i][j] ← input[round(i*hori_0/hori_1)][round(j*ver_0/ver_1)]
    end
end
```

This algorithm uses nested iterations to find the nearest pixel to each pixel of the target image in the coordinate system based on input image.

2. pseudo code for bilinear interpolation

derivation:

Let the pixels of the input image form the reference coordinate system.

Then the i row, j column point of the target image is at `(i*((input.shape[0]-1)/(target.shape[0]-1)),j*((input.shape[1]-1)/(target.shape[1]-1)))`.

The four adjacent points to the point at `(i,j)` have the following coordinates: `(floor(i*propi),floor(j*propj)),(floor(i*propi),ceil(j*propj)),(ceil(i*propi),floor(j*propj)),(ceil(i*propi),ceil(j*propj))`.

Using the formula: $y = y0 + (y1 - y0) * (x - x0)/(x1 - x0)$ to get the relative intensity in the line between the two parallel lines consist of the above four points. Next, using the two intensity values to calculate the point at `(i,j)`.

```
output ← np.zeros(dim)

propi ← (input.shape[0]-1)/(output.shape[0]-1)
propj ← (input.shape[1]-1)/(output.shape[1]-1)

for i ← 0 to dim[0]
    for j ← 0 to dim[1]
        i1,j1 ← floor(i*propi,j*propj)
        //get the largest integer smaller than or equal to the input number
        i2,j2 ← ceil(i*propi,j*propj)
        //get the smallest integer greater than or equal to the input number

        h1 ← calculate_height(j1,j2,input[i1][j1],input[i1][j2],j*propj)
        //get the intensity in the line consisting of two adjacent points near the target point
        h2 ← calculate_height(j1,j2,input[i2][j1],input[i2][j2],j*propj)
        //the two lines are parallel

        h ← calculate_height(i1,i2,h1,h2,i*propi)
        //calculate_height() is a function calculating the height at the point between two known points

        output[i][j] ← h
```

This algorithm first calculates two intensity values using four nearest pixels. Then, the two values are used to get the pixel intensity of the target image.

3. pseudo code for bicubic interpolation

```
output ← np.zeros(dim)

x ← np.arange(0,dim[1],(output.shape[1])/(input.shape[1]))
y ← np.arange(0,dim[0],(output.shape[0])/(input.shape[0]))
interp ← interp2d(x,y,input,'cubic')

x1 ← np.arange(0,dim[1],1)
y1 ← np.arange(0,dim[0],1)
output ← interp(x1,y1)
```

This algorithm utilizes well-integrated `scipy.interpolate.interp2d()` to easily get the image after bicubic interpolation.

3. Python codes

1. important part of the python code for bilinear interpolation

```
def calculate_height(x0,x1,y0,y1,x):
    if x1-x0==0 :
        return y0
    else:
        return y0+(y1-y0)*(np.round(x,3)-x0)/(x1-x0)

//the definition of `calculate_height()` mentioned in the pseudo code part
```

4. Results

My input image is a grayscale image of size 256*256.

My enlarged output images is of size 307*307.

My shrunk output image is of size 205*205.

All of the following descriptions incluing "enlarging" and "shrinking" are based on these two sizes.

The original image is below:

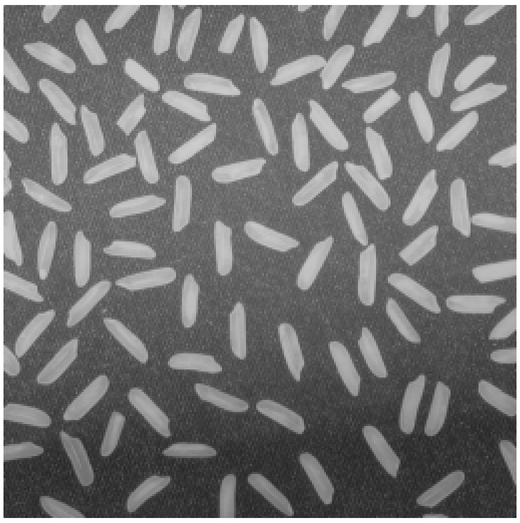

Fig. 2. the initial image of size 256*256

1. results of nearest neighbor interpolation

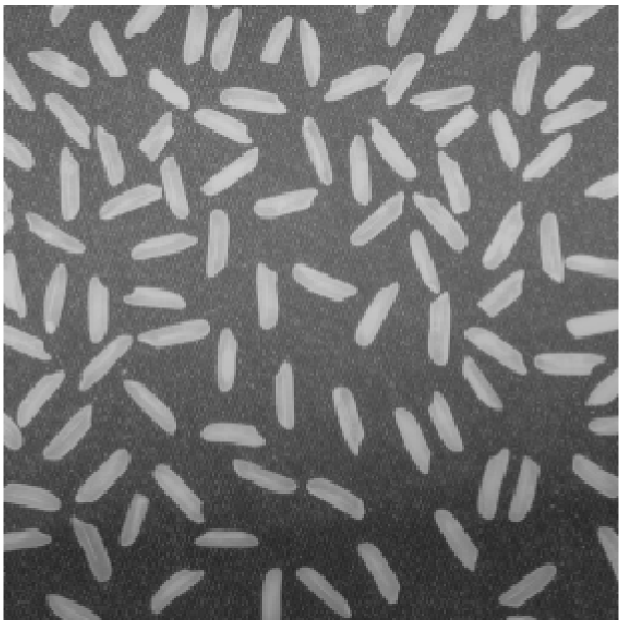

Fig. 3. the enlarged image of size 307*307 after nearest interpolation

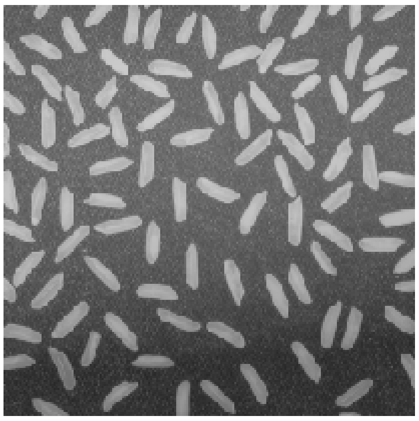

Fig. 4. the shrunk image of size 205*205 after nearest interpolation

2. results of bilinear interpolation

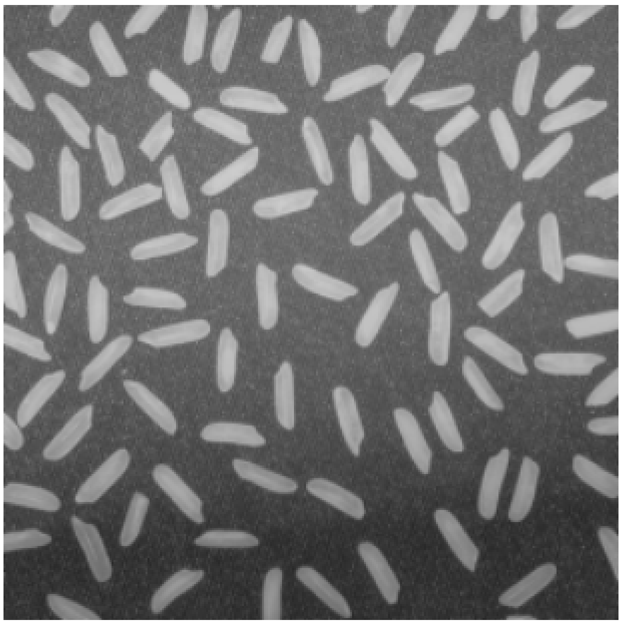

Fig. 5 the enlarged image of size 307*307 after bilinear interpolation

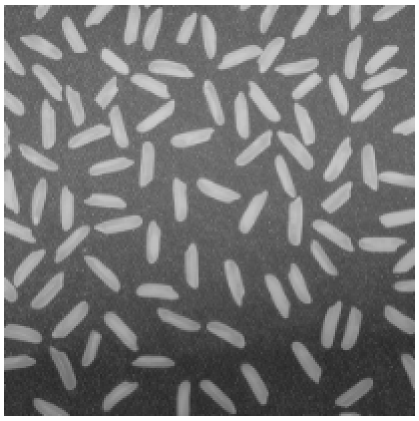

Fig. 6. the enlarged image of size 205*205 after bilinear interpolation

3. results of bicubic interpolation

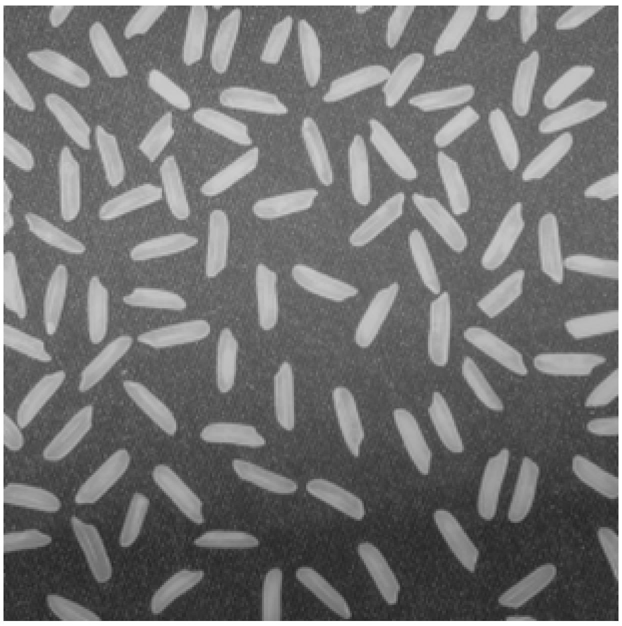

Fig. 7 the enlarged image of size 307*307 after bicubic interpolation

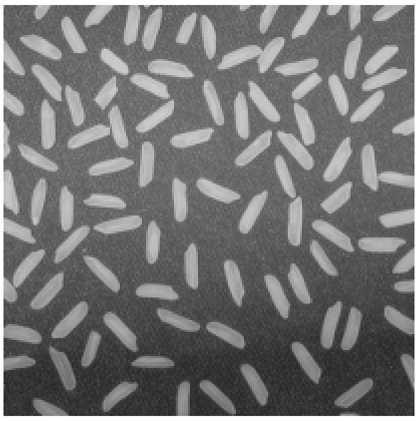

Fig. 8. the enlarged image of size 205*205 after bicubic interpolation

4. other results

These three algorithms can also be applied to create non-square images.

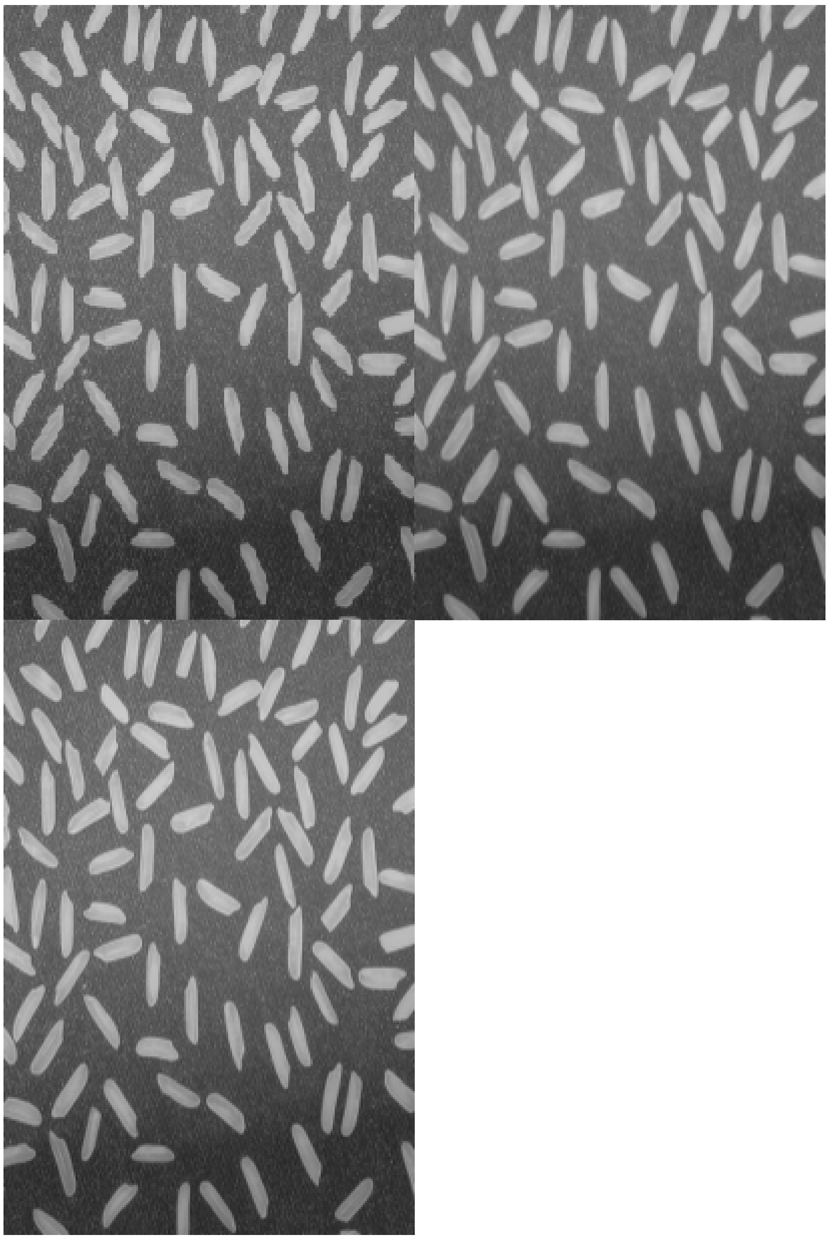

Fig. 8. the interpolated image of size 307*205 respectively after nearest, bilinear and bicubic interpolation

5. Analysis

1. Effects

After comparing the output images from the three algorithms, I find that the interpolation effect is: nearest neighbor interpolation>bilinear interpolation>bicubic interpolation. The output of nearest interpolation has many irreagular sawteeth which makes the image look unnatural. The output of bilinear interpolation or bicubic interpolation has smooth edges and looks natural.

2. Time

The time cost by the three algorithms is in the following table.

	nearest interpolation	**bilinear interpolation**	**bicubic interpolation**
enlarging image	0.10s	2.47s	0.02s
shrinking image	0.05s	0.98s	0.01s

The time cost of the three algorithms has the relation that: bilinear interpolation>>nearest interpolation>bicubic interpolation.

It is obvious that enlarging image using bilinear interpolation costs most time. The reason may be that the nested iteration has great complexity and generates a large amount of caculation. Additionally, the complexity of the three algorithms apparently has the order that bilinear interpolation>nearest interpolation>bicubic. Therefore, the result is reasonable.

I also use a well-defined method `cv2.resize()` to compare the time cost. It takes around 0.01s for the method to enlarge and shrink image by using nearest interpolation(code: `cv2.resize(input,dim,interpolation=cv2.INTER_NEAREST)`) and bilinear interpolation(code: `cv2.resize(input,dim,interpolation=cv2.INTER_LINEAR)`). It takes 0.02s to enlarge and shrink image by using bicubic interpolation(code: `cv2.resize(input,dim,interpolation=cv2.INTER_CUBIC)`). When nearest interpolation or bilinear interpolation is calculated, `cv2.resize()` is faster than mine. When bilinear interpolation is calculated, my algorithm and `cv2.resize()` has almost the same speed. It is deduced that there must be some optimized algorithms to realize different image interpolation. This reflects the beauty of programming, including always having more possibilities to make code lighter.

Actually, when I run my program in another computer, all these time data decrease, but their relationship does not change. To some degree, this proves the comparison does make sense.

6.Conclusion

I write three algorithms to realize nearest interpolation, nearest interpolation and bilinear interpolation. It is discovered that the effect of their results is: nearest neighbor interpolation>bilinear interpolation>bicubic interpolation, and the time cost is: bilinear interpolation>>nearest interpolation>bicubic interpolation. The results are reasonable basically due to their complexity.