Three Different Interpolation Realization

Student wu fengyang
SID 12210634

Introduction

The objectives of this experiment is using python to realize three typical interpolation algorithms including Bilinear, Nearest Neighbor, Bicubic interpolations. Ultimately when given an image and dimensions, it can be arbitrarily resized into that dimensions.

Interpolation is a fundamental technique that enhances the accuracy, quality, and usability of data across various domains. In image processing, it can determine how new pixel values are calculated based on the surrounding pixels. This is crucial for maintaining image quality. It can help improve the quality of low-resolution images when upscaling. Therefore, understanding and applying appropriate interpolation methods can significantly impact the effectiveness of analyses and visualizations.

Basic Principle

1. Nearest Neighbor Interpolation

This method assigns the value of the nearest pixel to the new pixel location. It is the simplest form of interpolation. For each pixel in the output image, find the nearest pixel in the input image. Assign the value of that nearest pixel to the output pixel.

$$I(x, y) = I(\text{round}(x'), \text{round}(y'))$$

2. Bilinear Interpolation

Bilinear interpolation considers the closest four pixels (a 2x2 square) surrounding the new pixel location and performs a linear interpolation first in one direction (x-axis) and then in the other (y-axis).

For each pixel in the output image, identify the four nearest pixels in the input image (top-left, top-right, bottom-left, bottom-right). Perform linear interpolation along the x-direction to find intermediate values between the top two pixels and the bottom two pixels. Then, perform linear interpolation along the y-direction using the results from the first step to get the final pixel value.

$$I(x', y') = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} I(x1, y1) & I(x1, y2) \\ I(x2, y1) & I(x2, y2) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}.$$

3. Bicubic Interpolation

Bicubic interpolation uses the closest 16 pixels (a 4x4 square) surrounding the new pixel location and applies cubic polynomials to interpolate values. This method is more sophisticated than bilinear interpolation.

For each pixel in the output image, identify the 16 nearest pixels in the input image. Use cubic polynomials to interpolate the pixel values in both the x and y directions. The interpolation takes into account the values of the surrounding pixels and their gradients, resulting in a smoother transition.

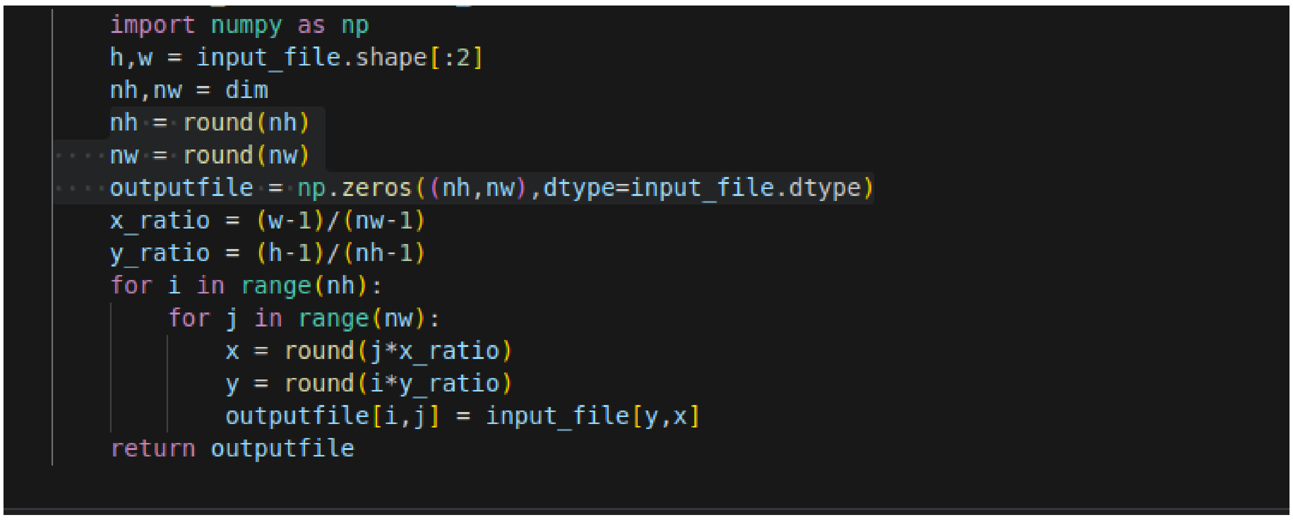
By some calculation, we can get the parameter aij for any 4 points block

$$I(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j.$$

Pseudo code

Algorithm 1 Nearest Neighbor Interpolation

```
Input: An image intensity matrix $M_{input}$. A new dimensions which the image matrix should be resized to
Output: A new image matrix with that dimensions. $M_{output}$
1: for each $i \in [0, M_{output}.width - 1]$ do
2:     for each $j \in [0, M_{output}.height - 1]$ do
3:         transfer the point from output matrix to input matrix $x' \leftarrow i * \frac{M_{input}.width-1}{M_{output}.width-1}, y' \leftarrow j * \frac{M_{input}.height-1}{M_{output}.height-1}$
4:         $M_{output}[i,j] \leftarrow M_{input}[round(x'), round(y')]$
5:     end for
6: end for
       return $M_{output}$
```

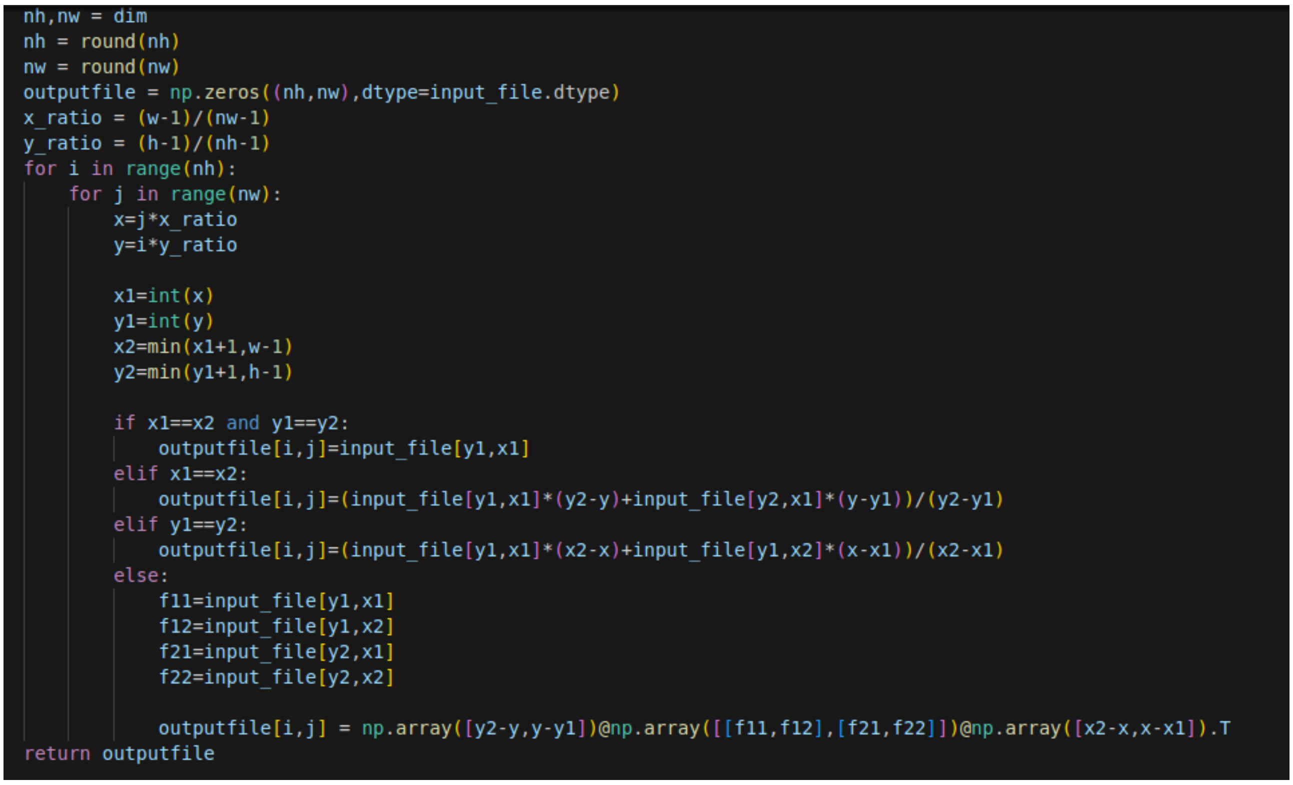


```
import numpy as np
h,w = input_file.shape[:2]
nh,nw = dim
nh = round(nh)
nw = round(nw)
outputfile = np.zeros((nh,nw),dtype=input_file.dtype)
x_ratio = (w-1)/(nw-1)
y_ratio = (h-1)/(nh-1)
for i in range(nh):
    for j in range(nw):
        x = round(j*x_ratio)
        y = round(i*y_ratio)
        outputfile[i,j] = input_file[y,x]
return outputfile
```

Figure 1: nearest

Algorithm 2 Bilinear Interpolation

```
Input: An image matrix with every value meaning intensity of gray light $M_{input}$ A new dimensions which the image matrix should resize to
Output: A new image matrix with that dimensions. $M_{output}$
1: for each $i \in [0, M_{output}.width - 1]$ do
2:     for each $j \in [0, M_{output}.height - 1]$ do
3:         transfer the point from output point to original point $x' = i * \frac{M_{input}.width-1}{M_{output}.width-1}, y' = j * \frac{M_{input}.height-1}{M_{output}.height-1}$
4:         use this interpolated point to find its surrounding block
5:         $x1 = floor(x'), x2 = min(M_{input}.width - 1, x1 + 1)$
6:         $y1 = floor(y'), y2 = min(M_{input}.height - 1, y1 + 1)$
7:         doing boundary detection
8:         if $x2 = x1 \&\& y2 = y1$ then
9:             $M_{output}[i,j] \leftarrow M_{input}[x1, y1]$
10:        else if $x2 = x1$ then
11:            $M_{output}[i,j] \leftarrow (y2 - y')M_{input}[x1, y1] + (y' - y1)M_{input}[x1, y2]$
12:        else if $y2 = y1$ then
13:            $M_{output}[i,j] \leftarrow (x2 - x')M_{input}[x1, y1] + (x' - x1)M_{input}[x2, y1]$
14:        else
15:            $M_{output}[i,j] = \begin{bmatrix} x_2 - x' & x' - x_1 \end{bmatrix} \begin{bmatrix} M_{input}(x1, y1) & M_{input}(x1, y2) \\ M_{input}(x2, y1) & M_{input}(x2, y2) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}.$
16:        end if
17:    end for
18: end for
       return $M_{output}$
```



```
nh,nw = dim
nh = round(nh)
nw = round(nw)
outputfile = np.zeros((nh,nw),dtype=input_file.dtype)
x_ratio = (w-1)/(nw-1)
y_ratio = (h-1)/(nh-1)
for i in range(nh):
    for j in range(nw):
        x=j*x_ratio
        y=i*y_ratio

        x1=int(x)
        y1=int(y)
        x2=min(x1+1,w-1)
        y2=min(y1+1,h-1)

        if x1==x2 and y1==y2:
            outputfile[i,j]=input_file[y1,x1]
        elif x1==x2:
            outputfile[i,j]=(input_file[y1,x1]*(y2-y)+input_file[y2,x1]*(y-y1))/(y2-y1)
        elif y1==y2:
            outputfile[i,j]=(input_file[y1,x1]*(x2-x)+input_file[y1,x2]*(x-x1))/(x2-x1)
        else:
            f11=input_file[y1,x1]
            f12=input_file[y1,x2]
            f21=input_file[y2,x1]
            f22=input_file[y2,x2]

            outputfile[i,j] = np.array([y2-y,y-y1])@np.array([[f11,f12],[f21,f22]])@np.array([x2-x,x-x1]).T
return outputfile
```

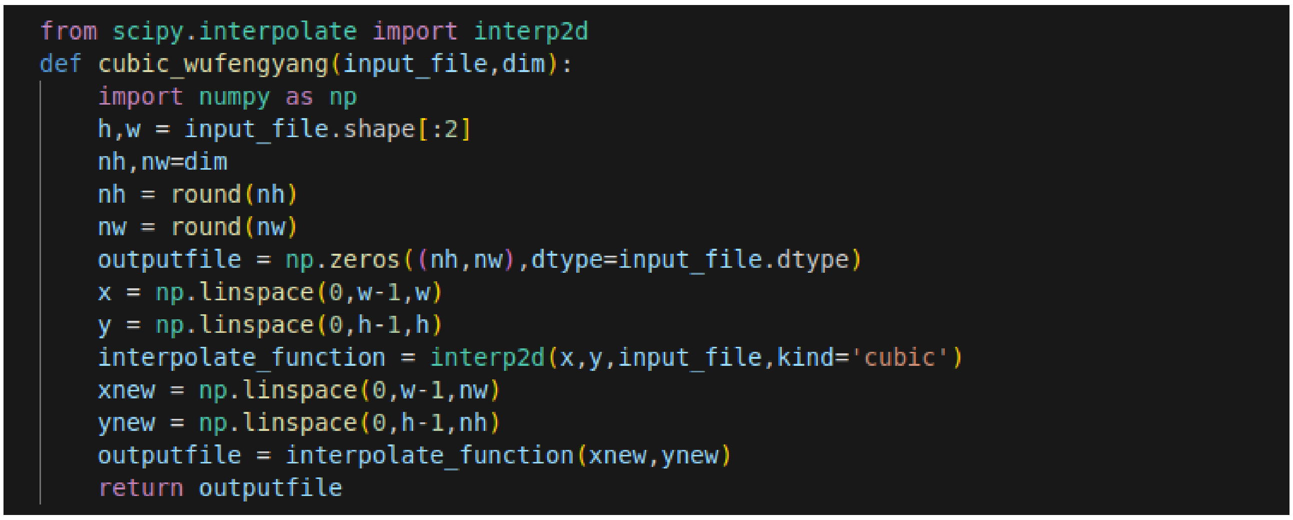
Figure 2: Here y and x switch Bilinear

Algorithm 3 Bicubic Interpolation

Input: An image matrix with every value meaning intensity of gray light M_{input} A new dimensions which the image matrix should resize to

Output: A new image matrix with that dimensions. M_{output}

```
1: for each $i \in [0, M_{output}.width - 1]$ do
2:     for each $j \in [0, M_{output}.height - 1]$ do
3:         getting interpolated function p: $p(x, y) = interp(x, y, M_{input}, 'cubic')$
4:         where x,y $=[0, 1, ..., M_{input}.width], [0, 1, ..., M_{input}.height]$
5:         $x_{new} = linspace(0, M_{input}.width, M_{output}.width)$,
6:         $y_{new} = linspace(0, M_{input}.height, M_{output}.height)$
7:         $M_{output}[i, j] \leftarrow p(x_{new}, y_{new})$
8:     end for
9: end for
       return $M_{output}$
```



```
from scipy.interpolate import interp2d
def cubic_wufengyang(input_file,dim):
    import numpy as np
    h,w = input_file.shape[:2]
    nh,nw=dim
    nh = round(nh)
    nw = round(nw)
    outputfile = np.zeros((nh,nw),dtype=input_file.dtype)
    x = np.linspace(0,w-1,w)
    y = np.linspace(0,h-1,h)
    interpolate_function = interp2d(x,y,input_file,kind='cubic')
    xnew = np.linspace(0,w-1,nw)
    ynew = np.linspace(0,h-1,nh)
    outputfile = interpolate_function(xnew,ynew)
    return outputfile
```

Figure 3: cubic

Result&Analysis

From the result we can easily see that bicubic interpolation has the best effectiveness. Then the bilinear is middle. The Nearest interpolation is the worst since we can see some blur on the image. In general, nearest should cost the least time, then is bilinear. The cubic should cost the longest time. But in my code, I find that cubic algorithm cost the least time. The reason may be it doesn't use python 'loop'. So it can calculate very fast and I don't find a good other algorithm to reduce the time of running nearest and bilinear interpolation.

I found that normally calling the resize function is bilinear interpolation in most library like cv2 and scipy. It has a relatively high speed to transfer and has good effectiveness. Every pixel only depends on 4 neighbor pixels. While the bicubic depends on 16 neighbor pixels to calculate. Therefore in spacial domain, it needs more spaces to store parameters. And the nearest interpolation need only one neighbor pixel to get the value. Thus, spacial complexity and time complexity rank are both bicubic $> bilinear >$nearest

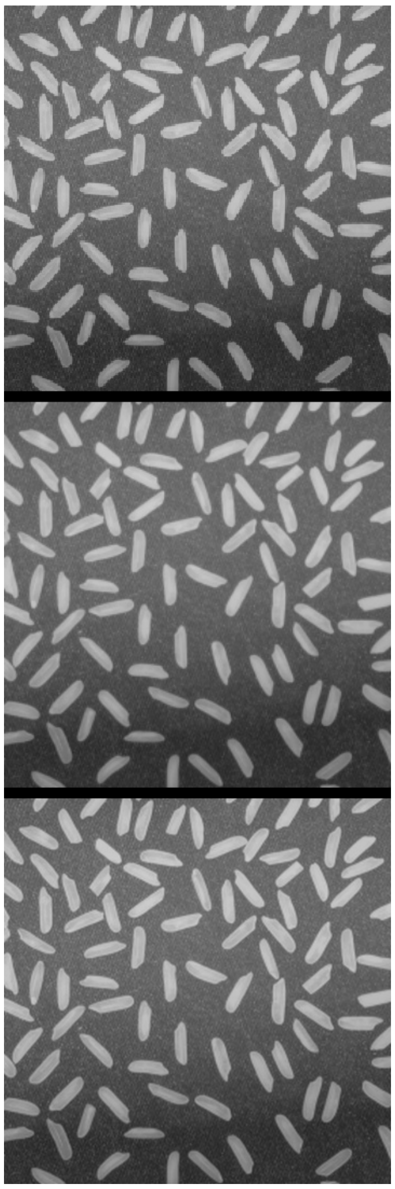


Figure 4: big 1.Nearest 2.bilinear 3.bicubic

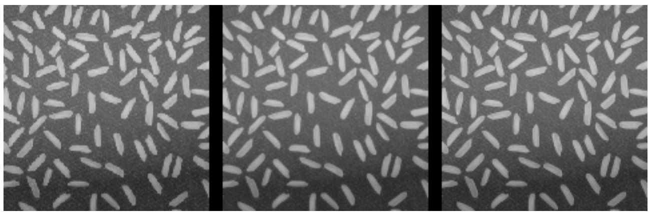
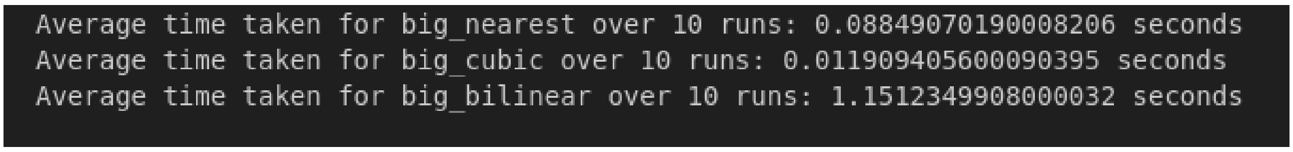


Figure 5: small 1.Nearest 2.bilinear 3.bicubic



```
Average time taken for big_nearest over 10 runs: 0.08849070190008206 seconds
Average time taken for big_cubic over 10 runs: 0.011909405600090395 seconds
Average time taken for big_bilinear over 10 runs: 1.1512349908000032 seconds
```

Figure 6: Enter Caption

Conclusion

In this experiment, I have learnt that how an image is reshaped and manually run some code to test those interpolation algorithm. Interpolation is a effective tool to process an image to what we want it to be like. However, this process often loss some information so that after many times resize, it may become unreadable. Therefore, interpolation for resizing should only do a limited time. In a word, Interpolation design should consider time cost and spacial cost. Some algorithm may run fast but losing more information. Different environment can apply different algorithm to meet their requirement.