Lab2 Image Interpolation

68/70

Hong-Jia Yang
Southern University of Science and Technology

Abstract—Digital images interpolation is of great significance in digital images processing. This report summaries main methods to achieve images interpolation with Python: nearest neighbor interpolation, bilinear interpolation and bicubic interpolation. Detailed principle and derivation will be shown in this report to explain these methods. The block diagrams of the algorithm, difference between the results from these methods, the efficiency of the methods and some code details will be discussed together in this report. Some detailed mathematical principle in discrete-time systems will also be explained. Also, some errors in online codes for images interpolation will be discussed.

Index Terms—Image interpolation, nearest neighbor interpolation, Python.

1 Introduction

In many cases, people need to further process the digital images. For example, image enlargement and detail processing are useful to make the remote sensing satellite images meaningful. It is necessary to use image interpolation technology to enlarge the original low resolution image or fuzzy image, and ensure the required clarity. Sometimes, the images may inevitably produce noise in the process of acquisition and transmission. These noises greatly damage the quality of images and affect the usability of them. Therefore, it is considered to denoise the images. The essence of denoising is to replace the gray value of the original noise point with a new gray estimation value under the denoising model. Therefore, the denoising problem can also be transformed into an interpolation problem.

partially is

Interpolation is divided into interpolation in images and interpolation between images. In this report, we mainly discuss interpolation in images. In digital image processing, interpolation is also called "resampling" and is used to fill the gap between pixels during image transformation.

methods used and brief results. -2

Fig. 1. Remote sensing satellite images

*Thanks

- *This work was supported in Department of Electrical and Electronic Engineering in Southern University of Science and Technology with Professor Yajun Yu, and the tutorial assistant and all classmates.*

Haha, thanks for the acknowledgement.

2 Lab Result

2.1 Preparation

How to understand interpolation:

Many people's initial understanding of image interpolation is based on the **position of each pixel block**(such as me), which is particularly obvious when applying the nearest neighbor algorithm. For example, we can easily enlarge a $3 * 3$ image to a $7 * 7$ or $9 * 9$ image. This number relationship of pixel blocks can ensure that we execute this interpolation algorithm. However, once the number of image pixel blocks changes, such as enlarging a $4 * 4$ image to a $7 * 7$ image, it is difficult for us to understand with the thought of pixel blocks.

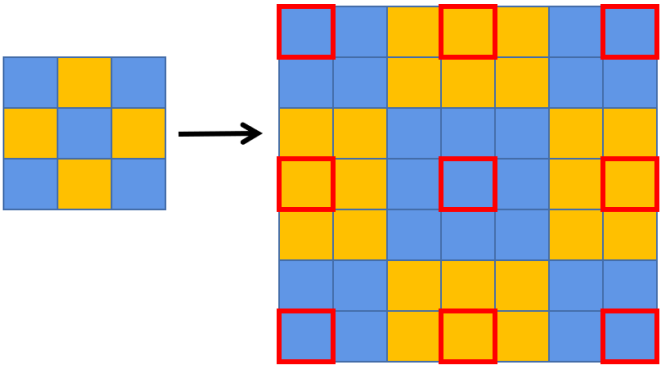

Fig. 2. A simple example for nearest neighbor interpolation

A better way to understand the image interpolation is mentioned in the textbook: "Suppose that an image of size pixels has to be enlarged 1.5 times to pixels. A simple way to visualize zooming is to create an imaginary grid with the same pixel spacing as the original, and then shrink it so that it fits exactly over the original image." **This is the real understanding of the image interpolation: retraction, matching and pixel spacing.**

We will use this method and mainly discuss the relationship between the pixels distance. It is very complicated to think about the pixel relationship of image interpolation. Many problems are caused by the lack of rigorous thinking and derivation. We should pay great attention to this.

2.2 Nearest Neighbor Interpolation

Nearest neighbor interpolation is the simplest method among these three methods. Its main idea is: Compress or enlarge the interpolated image to the original image's size, and set the gray value of each pixel in interpolated as the nearest neighbor pixel in the four neighboring pixels in original image.

My formula derivation is as follow:

Suppose we want to use nearest neighbor interpolation to interpolate a $a * a$ image to a $b * b$ image.

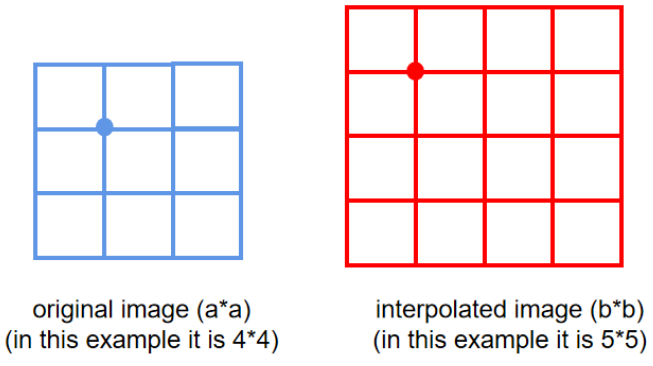

Fig. 3. Derivation for nearest neighbor interpolation: step1

The distance between the pixels in row and column is 1 in the original image and interpolated image. Using the method of retraction, we enlarge the scale of original image and change it to the size of interpolated image, that is, change the distance between pixels into $\frac{b-1}{a-1}$.

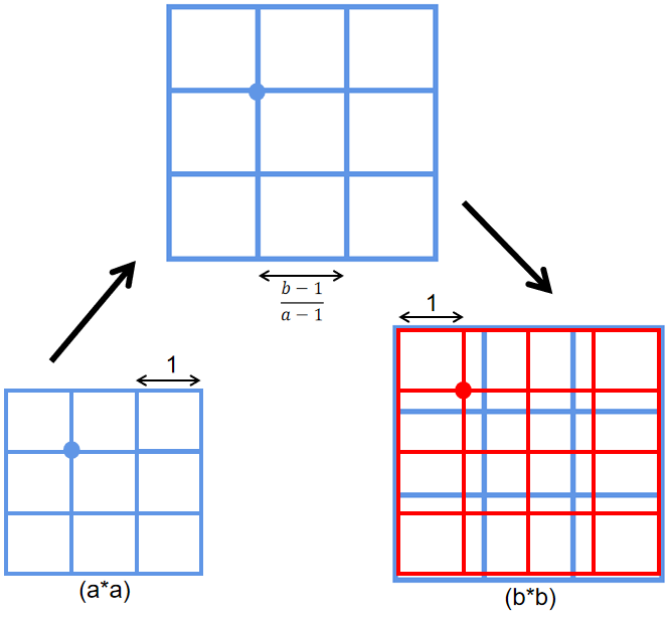

Fig. 4. Derivation for nearest neighbor interpolation: step2

In order to get a general conclusion, we pick out a square as shown in the figure and only consider the nearest neighbor interpolation horizontally.

We consider the distance from the original point in horizontal to the picked red pixel and the blue square. For the coordinate

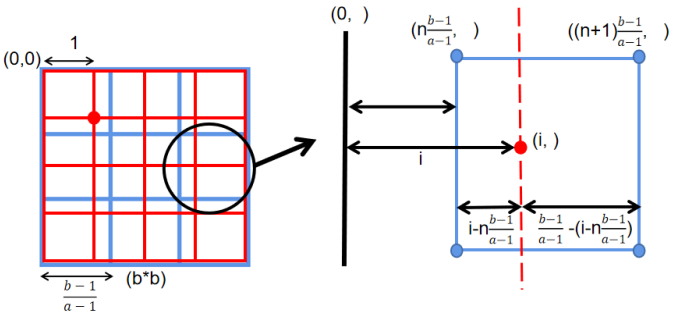

Fig. 5. Derivation for nearest neighbor interpolation: step3

of the red pixel is $(i,)$, its distance from the original point in horizontal is i. Suppose the red pixel can be contained in a blue square, with horizontal coordinate from $n\frac{b-1}{a-1}$ to $(n+1)\frac{b-1}{a-1}$, we should find the limit for the value n.

Obviously, n must satisfy

$i - n\frac{b-1}{a-1} < \frac{b-1}{a-1}$

so

$n = ceil(\frac{i(a-1)}{b-1} - 1)$

which is the minimum integer satisfy this equation.

Once we get the value of n, we can determine which pixel in original image to pick as the gray value for this red pixel.

If

$i - n\frac{b-1}{a-1} < \frac{b-1}{a-1} - (i - n\frac{b-1}{a-1})$

choose the left pixel's gray value as the red pixel's gray value. Otherwise, select the one on the right. This equation can be simplified as:

$i < (n + 0.5)\frac{b-1}{a-1}$

A controversial issue is that when the distance from the pixel to both sides is the same, which gray value should we use to replace it? Of course, the probability of this condition is relatively small and how to deal with it does not have a great impact on the final image. For this report, we choose the idea of rounding, set the right pixel's gray value as the red pixel's gray value.

From this method, we traverse all pixels in the image and get the interpolated image. The flow-chart for this method is as follow:

From this method, we generate the python code:

```
def nearest_11911214_slow(input_file, dim):
  # create the interpolated image
  result_image=np.empty((dim[0],dim[1]),dtype=np.uint8)
  #input_height, input_width
  ih,iw = input_file.shape[0],input_file.shape[1]
  #result_height,result_width
  rh,rw = dim[0],dim[1]
  col_ratio=(rw-1)/(iw-1)
  row_ratio=(rh-1)/(ih-1)
  for i in range(rw):
      for j in range(rh):
          n_row=math.ceil(i/row_ratio-1)
          row=n_row+1 if
            i>=(n_row+0.5)*row_ratio else n_row
          n_col=math.ceil(j/col_ratio-1)
          col=n_col+1 if
            j>=(n_col+0.5)*col_ratio else n_col
          result_image[i,j] = input_file[row,col]
  return result_image
```

We can optimization this algorithm by putting the calculation for n_row and the judgement only in each i value change. This will shorten the time spend for the algorithm.

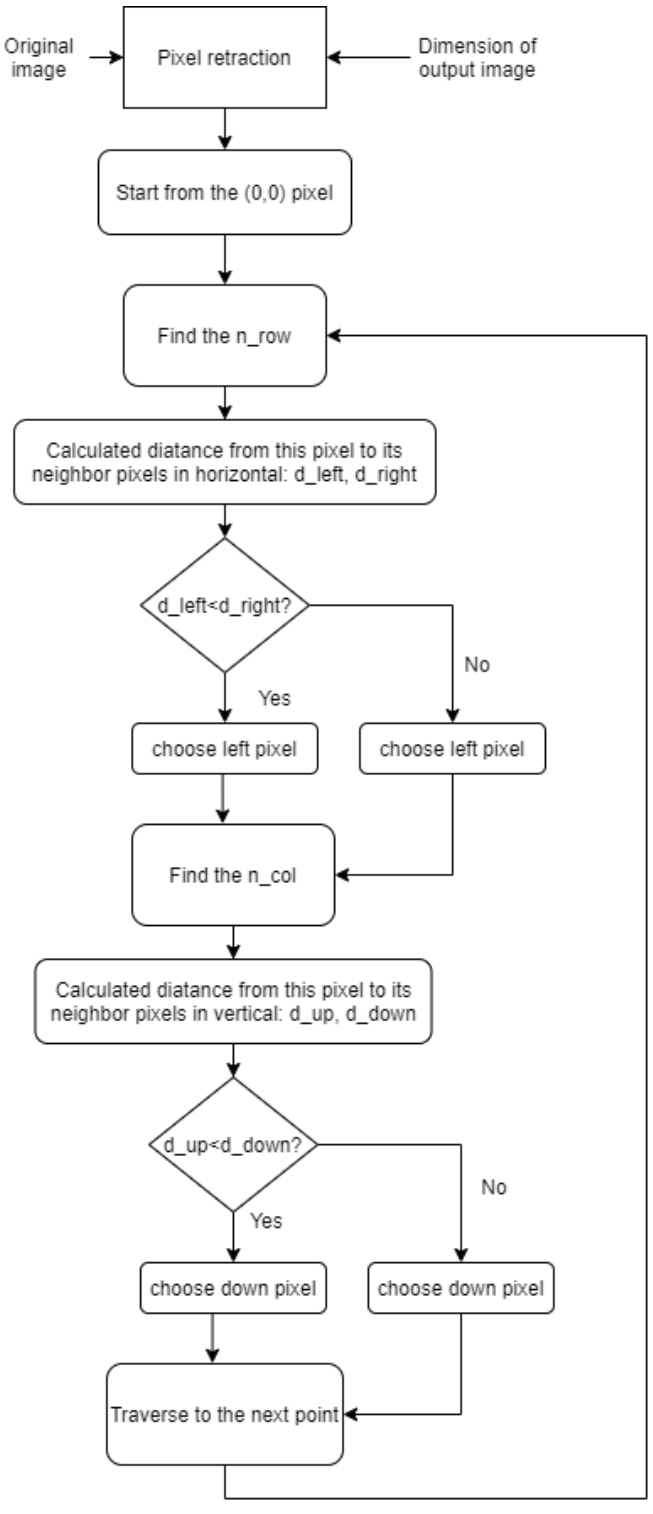

Fig. 6. Flow-chart for nearest neighbor interpolation

```python
def nearest_11911214(input_file, dim):
 # create the interpolated image
 result_image=np.empty((dim[0],dim[1]),dtype=np.uint8)
 #input_height, input_width
 ih,iw = input_file.shape[0],input_file.shape[1]
 #result_height,result_width
 rh,rw = dim[0],dim[1]
 col_ratio=(rw-1)/(iw-1)
 row_ratio=(rh-1)/(ih-1)
 for i in range(rw):
     n_row = math.ceil(i / row_ratio - 1)
     row = n_row + 1
         if i>=(n_row+0.5)*row_ratio else n_row
     for j in range(rh):
       n_col=math.ceil(j/col_ratio-1)
       col=n_col+1
         if j>=(n_col+0.5)*col_ratio else n_col
       result_image[i,j] = input_file[row,col]
 return result_image
```

We use these codes to calculate the time spend for the interpolation and calculate many times get the average value.

```python
time_start = time.time()
dst=nearest_11911214(original_image,enlarged_shape)
time_end = time.time()
print('time spend', time_end - time_start, 's')
```

From the result, we can shorten the nearest neighbor interpolation time from 0.224s to 0.116s.

	1	2	3	4	5	Average
Nearest neighbor	0.104s	0.11s	0.12s	0.13s	0.117s	0.116s
Nearest neighbor slow	0.262s	0.259s	0.194s	0.208s	0.198s	0.224s

Fig. 7. Test the improved algorithm time spend for nearest neighbor interpolation

Here I will have a further discussion about a popular code for nearest neighbor interpolation online, it is:

```python
def nearest_11911214_round(input_file, dim):
 result_image=np.empty((dim[0],dim[1]),dtype=np.uint8)
 input_height,input_width=
        input_file.shape[0],input_file.shape[1]
 result_height, result_width = dim[0],dim[1]
 for i in range(result_image_height):
  for j in range(result_image_width):
   col = round(j / result_width * input_width)
   row = round(i / result_height * input_height)
   result_image[i,j] = input_file[row,col]
 return result_image
```

It uses the Nearest neighbor formula,

$srcX = dstX * (srcWidth/dstWidth)$

and

$srcY = dstY * (srcHeight/dstHeight)$

We want to prove that only using this formula will cause the image center offset.

Also, **another formula can be found online, it correct the center offset, but it is still not perfect**:

$srcX = (dstX + 0.5) * (srcWidth/dstWidth) - 0.5$

$srcY = (dstY + 0.5) * (srcHeight/dstHeight) - 0.5$

We draw the result of retraction by the Nearest neighbor formula, correct formula and my method above:

From the result, **the Nearest neighbor formula will cause a offset to the center of the image, correct nearest neighbor formula can correct the offset, but it is not suitable also. My method can achieve the best retraction and get the suitable coordinate**. My method can do the retraction from interpolated image of the original image fully, considering the position of pixels' center.

We draw the result of the enlarged interpolation through my function, the round(Using Nearest neighbor formula) function and the result from the opencv itself function.

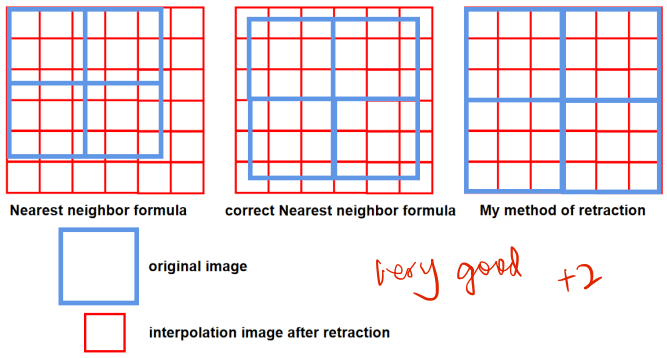

Fig. 8. Compare the three retraction methods

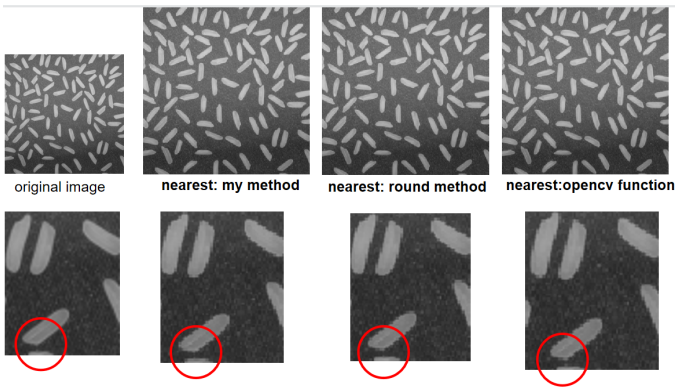

Fig. 9. Compare the nearest neighbor interpolation enlarged results

We can find that my method generate a good enlarged result, it has the minimum ambiguity, which prove that my method is more suitable. Other two methods' results has obvious ambiguity near the edge of the rice.

2.3 Bilinear Interpolation

Bilinear interpolation is based on the linear interpolation in each direction. Using linear interpolation in both horizontal and vertical direction and get the gray value for the undetermined pixel. The gray value of undetermined pixel is determined by its four neighbor pixels and the gray values are calculated by four weights according to the distance between pixels.

Specifically, we will firstly introduce linear interpolation:

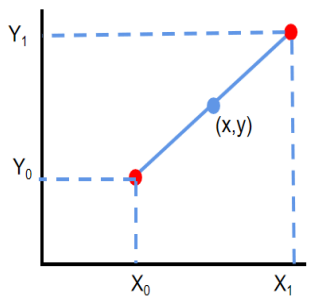

Fig. 10. Linear interpolation

The equation for the line is:

$$\frac{y-y_0}{x-x_0}=\frac{y_1-y_0}{x_1-x_0}.$$

That is,

$$y=\frac{x_1-x}{x_1-x_0}y_0+\frac{x-x_0}{x_1-x_0}y_1.$$

We will do this linear interpolation in horizontal twice and vertical once to achieve the bilinear interpolation.

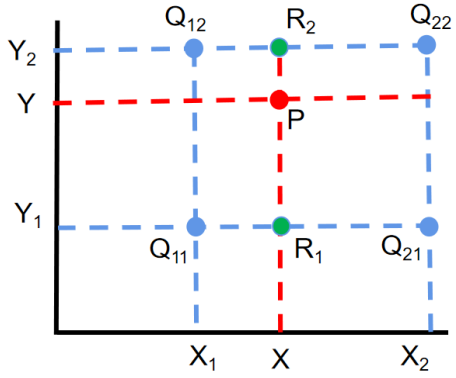

Fig. 11. Bilinear interpolation

We do retraction to put the interpolated image on the original image and want to get the gray value at $P=(x,y)$. We know the gray value at four pixels of original image Q_{11}, Q_{12}, Q_{21} and Q_{22}. We firstly use linear interpolation in horizontal twice to get R_1 and R_2's gray value, and use linear interpolation in vertical once to get gray value at P. Suppose the gray value at pixel x is f(x).

$$f(R_1)=\frac{x_2-x}{x_2-x_1}f(Q_{11})+\frac{x-x_1}{x_2-x_1}f(Q_{21}).$$

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

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

Thus, we get

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

The flow-chart for the bilinear interpolation is:

According to the folw-chart we generate the python code as follow:

```
5  def bilinear_11911214(input_file, dim):
6   # create the interpolated image
7   result_image=np.zeros(shape=(dim[0],dim[1]),dtype=np.
8   # input_height, input_width
9   ih,iw=input_file.shape[0],input_file.shape[1]
10  # result_height,result_width
11  rh, rw = dim[0], dim[1]
12  for i in range(rh):
```

① ② ③ 步-[illegible]

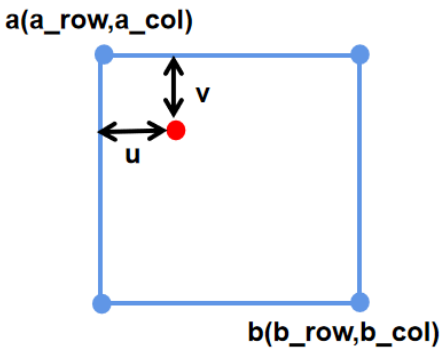

Fig. 12. Bilinear interpolation

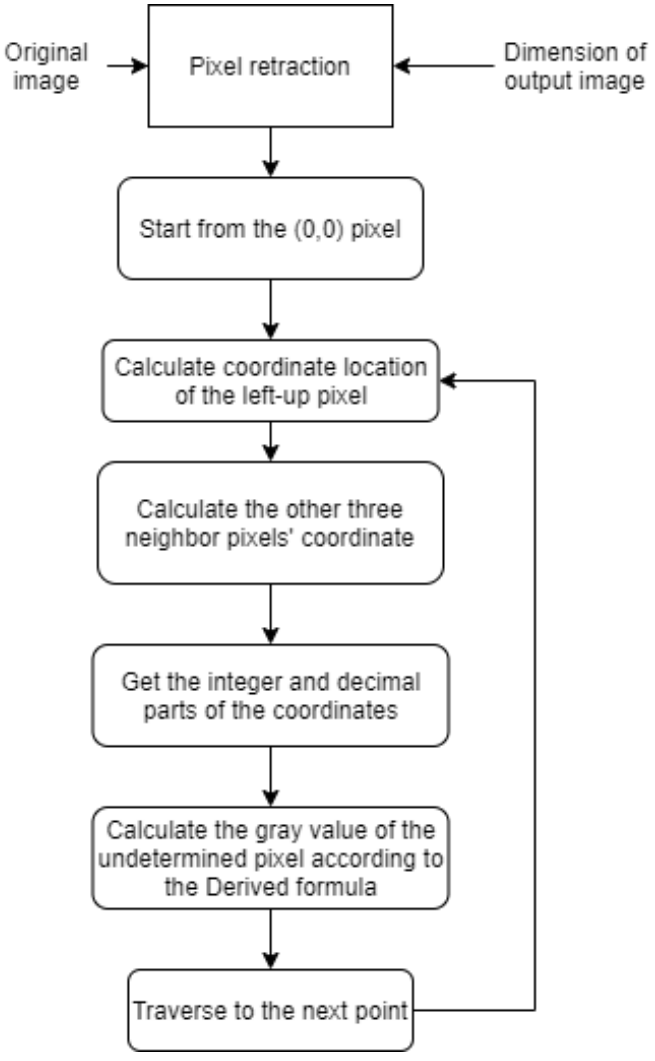

Fig. 13. Flow-chart for billinear interpolation

```
13    for j in range(rw):
14     row = i/(rh-1)*(ih-1)
15     col = j/(rw-1)*(iw-1)
16     a_row = int(np.floor(row))
17     a_col = int(np.floor(col))
18     b_row = int(np.ceil(row))
19     b_col = int(np.ceil(col))
20     u=row-a_row
21     v=col-a_col
```

```
22      result_image[i][j]=int(
23              input_file[a_row][a_col]*(1-u)*(1-v)
24              +input_file[b_row][a_col]*u*(1-v)
25              +input_file[a_row][b_col]*(1-u)*v
26              +input_file[b_row][b_col]*u*v)
27    return result_image
```

Note that in lines 18 and 19 we do not just plus 1 to the a_row and a_col, **this is to avoid that when we traverse the pixels on the edge, there are not all four neighbor pixels around it, resulting in our index crossing the boundary.**

2.4 Bicubic Interpolation

In the bilinear interpolation, we select the nearest four points of the undetermined pixel. In the bicubic interpolation, we select the nearest 16 pixels as the parameters to calculate the pixel's gray value, so the formula is more complex.

The influence of the 16 pixels on the undetermined pixel are determined by the selected sampling function. Different sampling function will make different results. We denoted the function as F(x) where x is the distance from the undetermined pixel to the each pixel of the 16 pixels in horizontal and vertical distance.

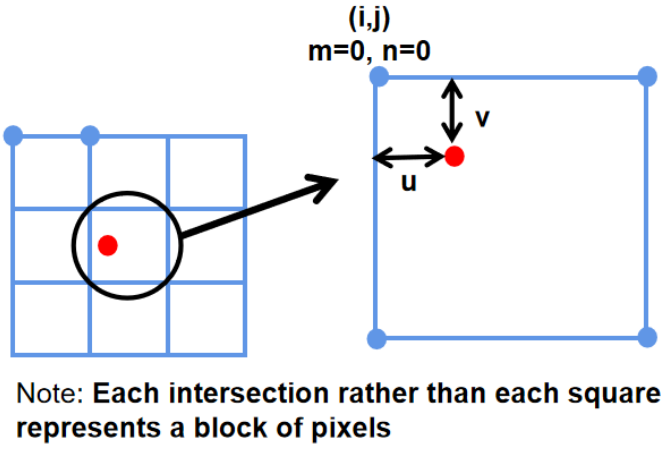

Fig. 14. Bicubic interpolation

The formula for the bicubic interpolation is:

$$gray(i+u, j+v) = \sum_{m=-1}^{2} \sum_{n=-1}^{2} gray(i+m, j+n) * F(m-u) * F(n-v)\cdot$$

The flow-chart for the bicubic interpolation is:

We choose three common bicubic interpolation sampling function online, their formula are:

And the python code for the bicubic interpolation according to the flow-chart.

Using interp2d package method's code is as follow:

```
6   def bicubic_11911214_interp2d(input_file, dim):
7    row = np.arange(input_file.shape[0])
8    col = np.arange(input_file.shape[1])
9    row_new = np.linspace(row[0],row[-1] , dim[0])
10   col_new = np.linspace(col[0],col[-1],dim[1])
11   f=interp2d(row, col,input_file, kind='cubic')
12   f_new=np.array(f(row_new,col_new),dtype='uint8')
13   return f_new
```

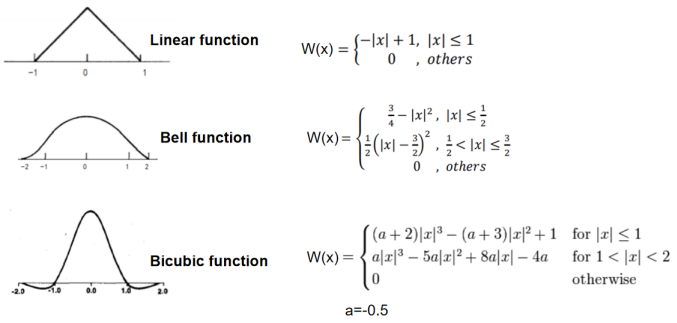

Fig. 15. Three common function for bicubic interpolation

```
def bicubic(x):
    x = abs(x)
    a = -0.5
    if x <= 1:
        return 1-(a+3)*(x**2)+(a+2)*(x**3)
    elif x < 2:
        return -4*a+8*a*x-5*a*(x**2)+a*(x**3)
    else:
        return 0
def triangle(x):
    x = abs(x)
    if x <= 1:
        return -x + 1
    else:
        return 0
def bell(x):
    x = abs(x)
    if x <= 1/2:
        return 3/4-x ** 2
    elif x <= 3/2:
        return 1/2*(x - 3/2)**2
    else:
        return 0

def bicubic_11911214(input_file, dim):
  # create the interpolated image
  result_image=np.zeros(shape=(dim[0],dim[1]),dtype=r
  F= bicubic
  # input_height, input_width
  ih,iw=input_file.shape[0],input_file.shape[1]
  # result_height,result_width
  rh, rw = dim[0], dim[1]
  for i in range(rh):
    for j in range(rw):
      row = i / (rh-1) * (ih - 1)
      col = j / (rw-1) * (iw- 1)
      row0 = int(row)
      col0 = int(col)
      u = row - row0
      v = col - col0
      temp = 0
      for m in range(-1, 3):
        for n in range(-1, 3):
          if (row0+m)<0 or (col0+n)< 0
              or (row0+m)>=ih or (col0+n)>=iw:
            continue
          temp+=input_file[row0 + m][col0 + n]
                *F(m-u)*F(n-v)
          result_image[i][j] = temp
  return result_image
```

The result using different sampling function is as follow:

From the results, although we use different sampling functions for bicubic interpolation, the difference is not obvious. This may means that different functions will not influence the result as obvious as different interpolation methods.

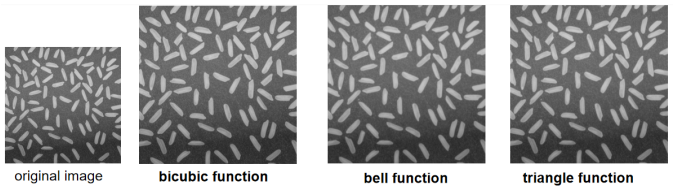

Fig. 16. Three sampling function for bicubic interpolation results

2.5 Results Analysis

We will compare the results obtained from nearest, bilinear, and bicubic interpolation. All the enlarged and shrunk results are shown below.

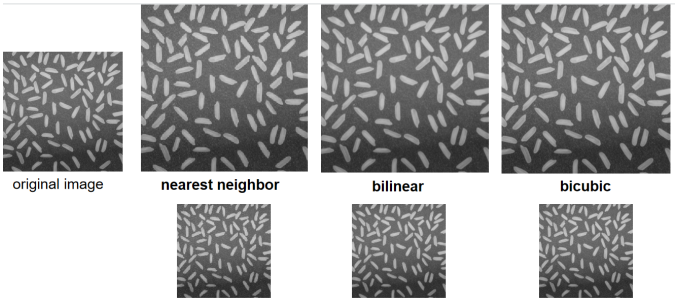

Fig. 17. Results from three interpolation methods

We zoom the enlarged results to see more details.

better to be their respective original size

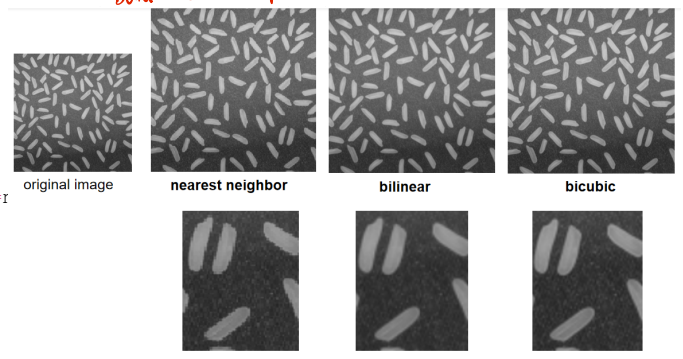

Fig. 18. Results from three interpolation methods

Nearest neighbor interpolation is the simplest to understand and its algorithm complexity is the lowest. But the enlarged image has serration. This is because we directly assign the undetermined pixel the gray value of its neighbor pixel. Thus, the result is not smooth enough, especially at the region of the edge of rice.

Bilinear interpolation can reduce the serration for it assign the gray value with the weight of four neighbor pixels. The edge is smoother but more blurred, which is caused by we only consider four neighbor, it is too ~~small~~. Also, the change rate between the neighbor pixel has not been considered.

Bicubic interpolation avoids the serration and blurring. The cost is that its algorithm is very complex. It consider both the influence of neighbor pixels and the influence of their gray value change rate. The result images of this algorithm is full and more natural.

Also, we can see the complex of different interpolation methods from time: the average time spend for nearest neigh-

bor interpolation is 0.116s, 3.26s for billinear interpolation and 22.96s for bicubic interpolation, they have a large gap.

	1	2	3	4	5	Average
Nearest neighbor	0.104s	0.11s	0.12s	0.13s	0.117s	0.116s
Bilinear	2.67s	4.26s	2.73s	3.01s	3.62s	3.26s
Bicubic	17.69s	20.41s	29.21s	26.02s	21.45s	22.96s

Fig. 19. Time spend comparison for three methods

3 Conclusion

In this lab, we learned how to use python achieve the image interpolation with three main methods: nearest neighbor interpolation, bilinear interpolation and bicubic interpolation. We analyze the properties of each method and have a deeper understanding based on their principle, derivation process, flow-charts, codes and result.

For nearest neighbor interpolation, we mainly discuss the problem of retraction. We point out the error in the Nearest neighbor formula in retraction and give a new method to do this through detailed mathematical derivation. For bilinear interpolation, we first introduce the linear interpolation and use this to achieve the bilinear interpolation. For bicubic interpolation, we introduce three sampling function and compare the results to discuss the influence of functions.

Finally, we compare the three interpolation methods from their results and algorithm complexity. We use detailed images and time spend data to explain our conclusion.

As we can see, **there is a balance between the time efficiency of different algorithm and its results' performance. We should select appropriate interpolation methods according to the actual situation, and constantly propose new interpolation methods or algorithm optimization to improve the effect of image processing.**

My main harvest is the detailed analysis in Part 2. We learn the main method to achieve the interpolation using python, and the method to solve many important problems due to the coding of python. Now we have a deeper understanding towards image interpolation and digital image processing. This is a course that requires careful thinking

An analytic analysis of the complexity would be nice -2

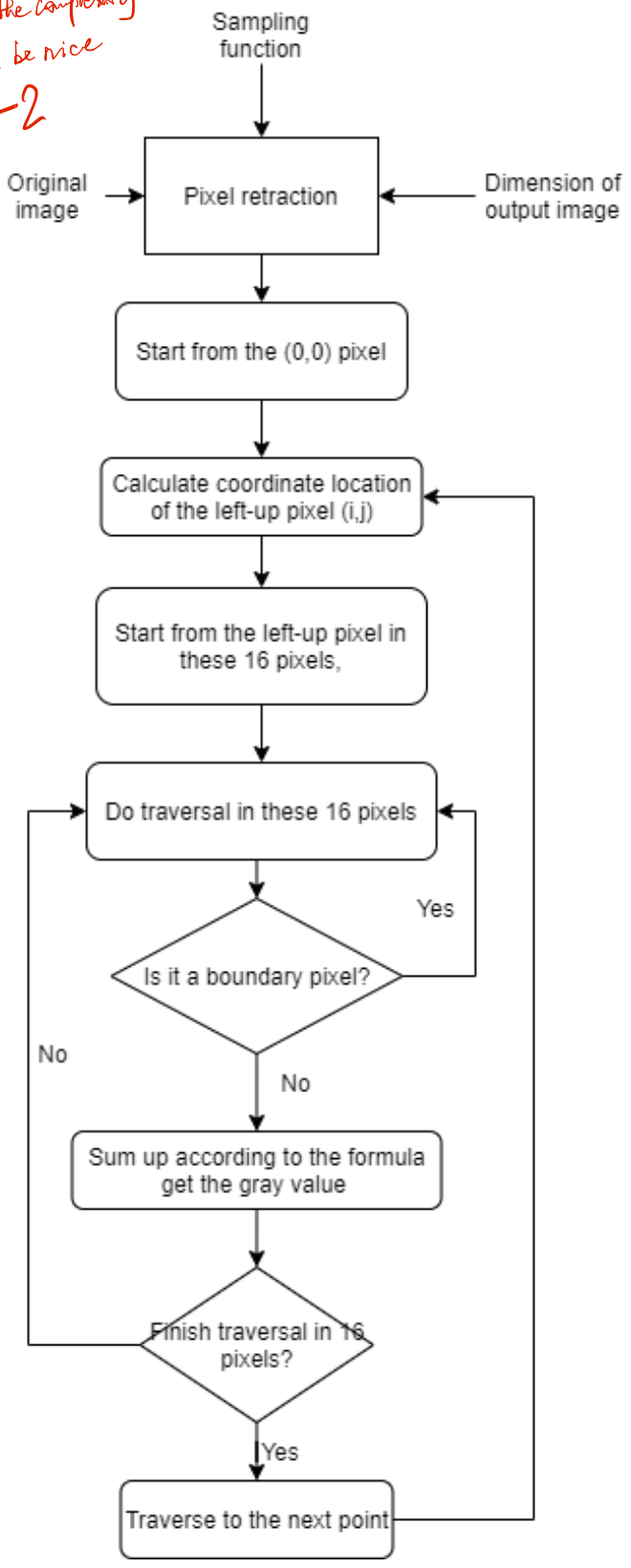

Fig. 20. Flow-chart for bicubic interpolation