EE326 Lab Report 2: Image Interpolation

11812418 樊青远[1]

[1] *School of Microelectronics, SUSTech Email: fanqy2018@mail.sustech.edu.cn*

Abstract

This report provides the introduction, theory and the Python code implementation of several image interpolation methods that is widely used in the current industry.

Keywords: Image Interpolation, DIP

Contents

EE326 DIP (2021)

DOI: 10.1017/pan.xxxx.xx

Corresponding author
Qingyuan Fan

Advised by
Yajun Yu

1 Objective

The objective of this experiment is to let us understand the different interpolation algorithms and be able to implement several of them through Python code. In our experiment, we specifically discuss and implement the interpolation algorithm of Nearest Neighbor Interpolation Bilinear Interpolation and Bicubic Interpolation on a gray scale image. The graphical introduction of the algorithm can be seen in Figure 1.

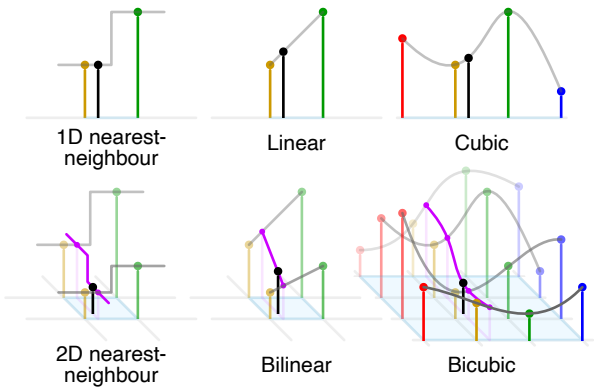


Figure 1. Comparison of 1D and 2D interpolation via multiple methods.(Cmglee, n.d.)

2 Nearest Neighbor Interpolation

2.1 Algorithm of Nearest Neighbor Interpolation

Nearest neighbor interpolation is a simple method of multivariate interpolation in one or more dimensions. The key algorithm of Nearest-neighbor interpolation is adapt the value of nearest point in the input image that is nearest to the output matrix without considering the values of neighboring points at all.

The algorithm is the most resource-saving one among the algorithms mentioned above.(Parker, Kenyon, and Troxel 1983)

2.2 Flow Chart

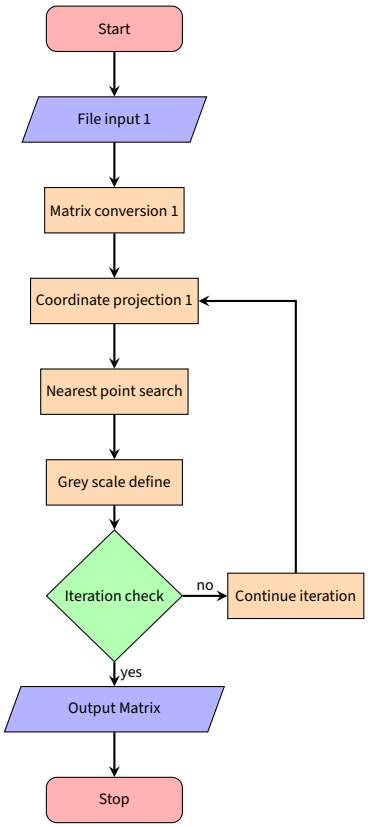


Figure 2. Flow chart of Nearest Neighbor Interpolation algorithm.

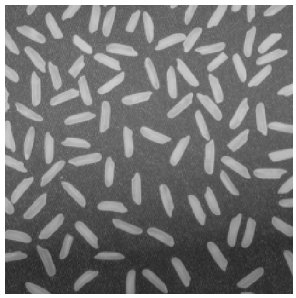
1. Image file and output image dimension

2.3 Python Code

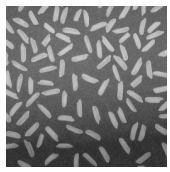
```
def Nearest_11812418(input_file, dim):
    image = Image.open(input_file)
    imarray = np.array(image)
    #find the dimenssion of input file
    input_height = imarray.shape[0]
    input_width = imarray.shape[1]

    #set the output
    #dim(0) is the height
    #dim(1) is the width
    #init an array
    output_arr = np.zeros(dim)

    #iterate in output array
    for i in range(dim[0]):
        for j in range(dim[1]):
            #i height (dim0)
            #j width (dim1)
```

```
            interpolation_h = round((i)*(input_height-1)/(dim[0]-1))
            interpolation_w = round((j)*(input_width-1)/(dim[1]-1))
            output_arr[i][j] = imarray[interpolation_h][interpolation_w]

    im = Image.fromarray(output_arr.astype(np.uint8))
    im.save("nearest.tif")
    return output_arr
```

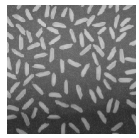
2.4 Result



(a) Enlarged image with Nearest-neighbor interpolation. (461*461)



(b) Original image. (256*256)



(c) Shrinked image with Nearest-neighbor interpolation. (205*205)

Figure 3. Interpolation result with Nearest neighbor interpolation.

3 Bilinear Interpolation

3.1 Algorithm of Bilinear Interpolation

The Bilinear interpolation uses the nearest four points in the meshgrid and the relative position between the edge and the output point to interpolate the data.

To simplify the model, supposing we want to find the value of the unknown function f at the point (x, y). It is assumed that we know the value of f at the four points $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$, and $Q_{22} = (x_2, y_2)$.

First, we implement the linear interpolation on the X axis, which is the height axis of the image.

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}), \tag{1}$$

$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}). \tag{2}$$

We secondly implement the interpolation on the Y axis, which is the width axis of the image.

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \tag{3}$$

$$= \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right) \tag{4}$$

$$= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}. \tag{5}$$

In the every iteration of the image interpolation process, we find the four points mentioned

above and their relative position. Then we apply the equation 3 above to the bitmap image could reach the correct result.

3.2 Flow Chart

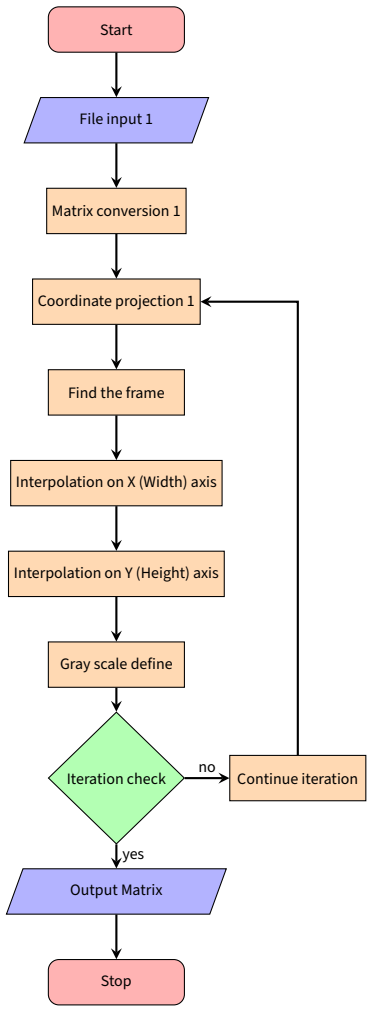


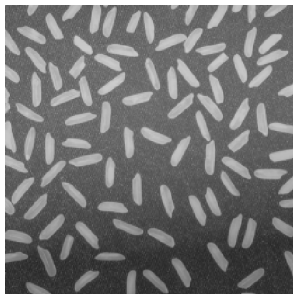
Figure 4. Flow chart of Bilinear Interpolation algorithm.

3.3 Python Code

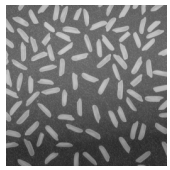
```
def Bilinear_11812418(input_file, dim):
    image = Image.open(input_file)
    imarray = np.array(image)
    #find the dimenssion of input file
    input_height = imarray.shape[0]
    input_width = imarray.shape[1]

    #set the output
    #dim(0) is the height
    #dim(1) is the width
    #init an array
    output_arr = np.zeros(dim)
```

```
13
14      delta_height = 1
15      delta_width = 1
16      #base: output
17      #iterate in output array
18      for i in range(dim[0]):
19          for j in range(dim[1]):
20              #i height (dim0)
21              #j width (dim1)
22
23              #start from 1
24              #transform to input coordinate
25              projected_height = (i)*(input_height-1)/(dim[0]-1)
26              projected_width = (j)*(input_width-1)/(dim[1]-1)
27              #print(str(projected_height) + ' ' + str(projected_width))
28
29              #find the border
30
31              interpolation_h_up = int(np.ceil(projected_height))
32              interpolation_h_down = int(np.floor(projected_height))
33              interpolation_w_up = int(np.ceil(projected_width))
34              interpolation_w_down = int(np.floor(projected_width))
35
36              k_w_down = (abs(imarray[interpolation_h_up][interpolation_w_down]-
37              imarray[interpolation_h_down][interpolation_w_down]))/delta_height
38
39              w_down_val = imarray[interpolation_h_down][interpolation_w_down] +
40              k_w_down*(projected_height-interpolation_h_down)
41
42              k_w_up = (abs(imarray[interpolation_h_up][interpolation_w_up]-
43              imarray[interpolation_h_down][interpolation_w_up]))/delta_height
44              w_up_val = imarray[interpolation_h_down][interpolation_w_up] +
45              k_w_up*(projected_height-interpolation_h_down)
46
47
48              #special case
49              if(projected_height % 1):
50                  w_down_val = imarray[int(projected_height)][int(projected_width)]
51                  w_up_val = imarray[int(projected_height)][int(projected_width)]
52
53              if(projected_width % 1):
54                  interpolation_w_down = projected_width
55
56              k_h = (abs(w_up_val-w_down_val)/delta_width)
57              h_val = w_down_val + k_h*(projected_width-interpolation_w_down)
58
59              output_arr[i][j] = h_val
60      print(output_arr)
61      return output_arr
```

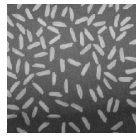
3.4 Result



(a) Enlarged image with Bilinear interpolation. (461*461)



(b) Original image. (256*256)



(c) Shrinked image with Bilinear interpolation. (205*205)

Figure 5. Interpolation result with Bilinear interpolation.

4 Bicubic Interpolation

4.1 Algorithm of Bicubic Interpolation

The Bicubic interpolation is an extension of cubic interpolation for interpolating data points on a two-dimensional regular grid. Comparing to the Nearest interpolation and the Bilinear interpolation, the Bicubic could provide a smoother output but with higher resource usage.

The Bicubic interpolation uses 16 points around the output point for sampling and interpolation. The sample spline is listed in formula 6

$$u(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & \text{for } |x| \le 1, \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{for } 1 < |x| < 2, \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

In scipy(“scipy/scipy,” n.d.)(“scipy.interpolate.interp2d —SciPy v1.6.0 Reference Guide,” n.d.), the interpolation is implemented by Cubic Hermite spline, so $a = -0.5$. After sampling, we multiple the sampling result with the interpolation kernel to get the corresponding grey scale value of the specific coordinate, which is described in formula 7.

$$value(x,y) = \begin{pmatrix} u(x_1) & u(x_2) & u(x_3) & u(x_4) \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} & f_{14} \\ f_{21} & f_{22} & f_{23} & f_{24} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{41} & f_{42} & f_{43} & f_{44} \end{pmatrix} \begin{pmatrix} u(y_1) \\ u(y_2) \\ u(y_3) \\ u(y_4) \end{pmatrix} \tag{7}$$

4.2 Flow Chart

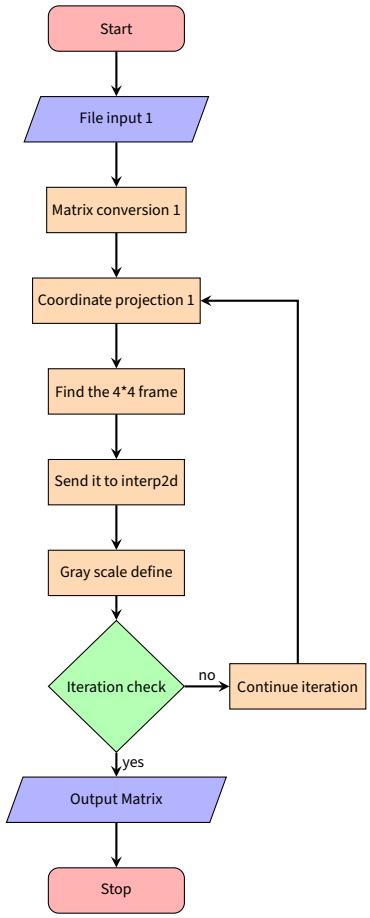


Figure 6. Flow chart of Bicubic Interpolation algorithm.

4.3 Python Code

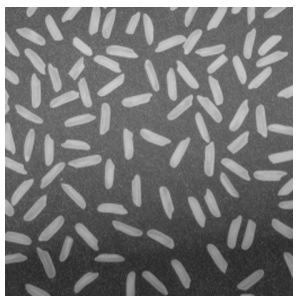
In order to reduce the vignetting caused by padding, the program will judge whether interpolation frame (kernel) is on the corner of the image, then automatically shift the interpolation kernel to prevent the issues of ”out of index” during the interpolation kernel extraction.

```
1  def Bicubic_11812418(input_file, dim):
2      image = Image.open(input_file)
3      imarray = np.array(image)
4
5      #find the dimenssion of input file
6      input_height = imarray.shape[0]
7      input_width = imarray.shape[1]
8
9      #set the output
10     #dim(0) is the height
11     #dim(1) is the width
12     #init an array
13     output_arr = np.zeros(dim)
14
```

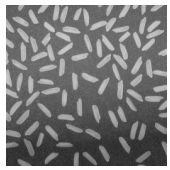
```
    delta_height = 1
    delta_width = 1
    #base: output

    # construct the bicubic function
    def interp2d_bicubic_scipy(array_data,relative_y,relative_x):
        x = [0,1,2,3]
        y = [0,1,2,3]
        f = interpolate.interp2d(y, x, array_data, kind='cubic')
        interp_result = f(relative_y,relative_x)
        return interp_result

    #iterate in output array
    for i in range(dim[0]):
        for j in range(dim[1]):
            #i height (dim0)
            #j width (dim1)

            #start from 1
            #transform to input coordinate
            projected_height = (i)*(input_height-1)/(dim[0]-1)
            projected_width = (j)*(input_width-1)/(dim[1]-1)
            #print(str(projected_height) + ' ' + str(projected_width))

            #find the border
            interpolation_x_floor = int(np.floor(projected_width))
            interpolation_y_floor = int(np.floor(projected_height))

            array16 = []
            frame_position_x = interpolation_x_floor
            frame_position_y = interpolation_y_floor
            #special case
            #XY MIN
            if((interpolation_x_floor-1<0) or (interpolation_y_floor-1<0) or
            (interpolation_x_floor + 2>input_width-1) or
            (interpolation_y_floor + 2>input_height-1)):

                border_X_Min = max(0,interpolation_x_floor-1)
                border_Y_Min = max(0,interpolation_y_floor-1)
                border_X_Max = min(input_width-1,interpolation_x_floor+2)
                border_Y_Max = min(input_height-1,interpolation_y_floor+2)

                if ((interpolation_x_floor-border_X_Min)<
                (border_X_Max-interpolation_x_floor-1)):
                    #left border
                    frame_position_x = border_X_Min+1
                elif ((interpolation_x_floor-border_X_Min)>
                (border_X_Max-interpolation_x_floor-1)):
                    #right border
```

```
                    frame_position_x = border_X_Max-2
                if ((interpolation_y_floor-border_Y_Min)<
                (border_Y_Max-interpolation_y_floor-1)):
                    #lower border
                    frame_position_y = border_Y_Min+1
                elif ((interpolation_y_floor-border_Y_Min)>
                (border_Y_Max-interpolation_y_floor-1)):
                    #upper border
                    frame_position_y = border_Y_Max-2


            for k in range(4):
                array16 =
                np.concatenate((array16, imarray[frame_position_y -1 +
                k][frame_position_x -1 :frame_position_x + 3]), axis=None)



            output_arr[i][j] = interp2d_bicubic_scipy(array16,
            (projected_height-frame_position_y+1),(projected_width-frame_position_x+1)

    print(output_arr)
    return output_arr
```

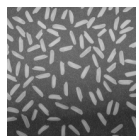
4.4 Result



(a) Enlarged image with Bicubic interpolation. (461*461)



(b) Original image. (256*256)



(c) Shrinked image with Bicubic interpolation. (205*205)

Figure 7. Interpolation result with Bicubic interpolation.

5 Other Algorithms: Biquintic Interpolation

5.1 Algorithm of Biquintic Interpolation

According to the documentation of scipy.interpolate.interp2d, it also provide the interpolation method with Biquintic interpolation.("scipy.interpolate.interp2d —SciPy v1.6.0 Reference Guide," n.d.) The algorithm of Biquintic interpolation is quite similar to the Bicubic interpolation mentioned in the previous paragraph(Mund, Hallet, and Hennart 1975). The only difference between them is that the Biquintic interpolation use quintic function as its spline during the interpolation, the Bicubic function only use cubic function as its spline. We could also write the formula of Biquintic interpolation in the form of matrix operation as formula 8. The $u(X_i)$ is the spline function of Biquintic interpolation. In order to minimize the loss of the interpolation quality caused by the

complemented edge, we use a $6 \cdot 6$ frame for spline interpolation.

$$val(x,y) = \begin{pmatrix} u(x_1) & u(x_2) & u(x_3) & u(x_4) & u(x_5) & u(x_6) \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} & f_{14} & f_{15} & f_{16} \\ f_{21} & f_{22} & f_{23} & f_{24} & f_{25} & f_{26} \\ f_{31} & f_{32} & f_{33} & f_{34} & f_{35} & f_{36} \\ f_{41} & f_{42} & f_{43} & f_{44} & f_{45} & f_{46} \\ f_{51} & f_{52} & f_{53} & f_{54} & f_{55} & f_{56} \\ f_{61} & f_{62} & f_{63} & f_{64} & f_{65} & f_{66} \end{pmatrix} \begin{pmatrix} u(y_1) \\ u(y_2) \\ u(y_3) \\ u(y_4) \\ u(y_5) \\ u(y_6) \end{pmatrix} \tag{8}$$

5.2 Flow Chart

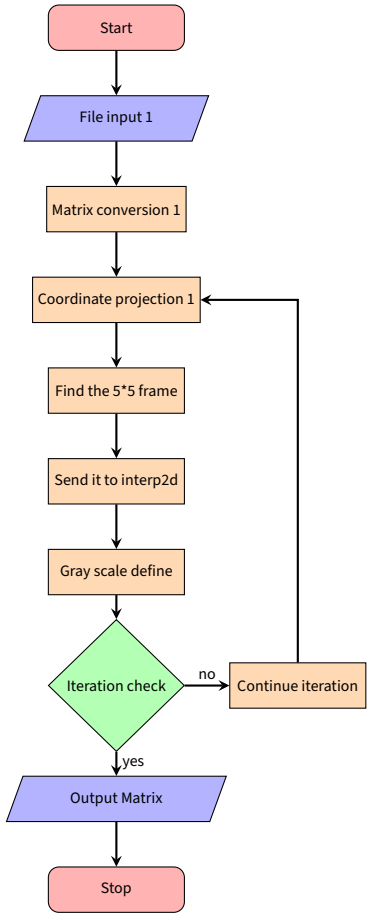


Figure 8. Flow chart of Biquintic Interpolation algorithm.

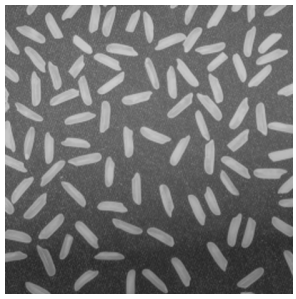
5.3 Python Code

```
from scipy import interpolate
port numpy as np
port matplotlib.pyplot as plt
om PIL import Image
port statistics

```

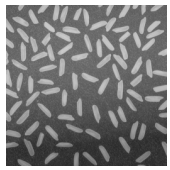
```
f Biquintic_11812418(input_file, dim):
  image = Image.open(input_file)
  imarray = np.array(image)

  #find the dimenssion of input file
  input_height = imarray.shape[0]
  input_width = imarray.shape[1]

  #set the output
  #dim(0) is the height
  #dim(1) is the width
  #init an array
  output_arr = np.zeros(dim)

  delta_height = 1
  delta_width = 1
  #base: output

  # construct the biquintic function
  def interp2d_biquintic_scipy(array_data,relative_y,relative_x):
      x = [0,1,2,3,4,5]
      y = [0,1,2,3,4,5]
      f = interpolate.interp2d(y, x, array_data, kind='quintic')
      interp_result = f(relative_y,relative_x)

      return interp_result

  #iterate in output array
  for i in range(dim[0]):
      for j in range(dim[1]):
          #i height (dim0)
          #j width (dim1)

          #start from 1
          #transform to input coordinate
          projected_height = (i)*(input_height-1)/(dim[0]-1)
          projected_width = (j)*(input_width-1)/(dim[1]-1)

          #find the border
          interpolation_x_floor = int(np.floor(projected_width))
          interpolation_y_floor = int(np.floor(projected_height))

          array36 = []
          frame_position_x = interpolation_x_floor
          frame_position_y = interpolation_y_floor
          #special case
          #XY MIN
          if((interpolation_x_floor-2<0) or (interpolation_y_floor-2<0) or
          (interpolation_x_floor + 2>input_width-2) or
```

```
        (interpolation_y_floor + 2>input_height-2)):

            border_X_Min = max(0,interpolation_x_floor-2)
            border_Y_Min = max(0,interpolation_y_floor-2)
            border_X_Max = min(input_width-1,interpolation_x_floor+3)
            border_Y_Max = min(input_height-1,interpolation_y_floor+3)

            if ((interpolation_x_floor-border_X_Min)<
            (border_X_Max-interpolation_x_floor-1)):
                #left border
                frame_position_x = border_X_Min+2
            elif ((interpolation_x_floor-border_X_Min)>
            (border_X_Max-interpolation_x_floor-1)):
                #right border
                frame_position_x = border_X_Max-3
            if ((interpolation_y_floor-border_Y_Min)<
            (border_Y_Max-interpolation_y_floor-1)):
                #lower border
                frame_position_y = border_Y_Min+2
            elif ((interpolation_y_floor-border_Y_Min)>
            (border_Y_Max-interpolation_y_floor-1)):
                #upper border
                frame_position_y = border_Y_Max-3


        for k in range(6):
            array36 = np.concatenate((array36,
            imarray[frame_position_y -2 + k][frame_position_x -2 :frame_position_x + 4


        output_arr[i][j] = interp2d_biquintic_scipy(array36,
        (projected_height-frame_position_y+2),
        (projected_width-frame_position_x+2))

print(output_arr)
return output_arr
```

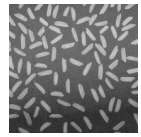
5.4 Result



(a) Enlarged image with Biquintic interpolation. (461*461)



(b) Original image. (256*256)



(c) Shrinked image with Biquintic interpolation. (205*205)

Figure 9. Interpolation result with Biquintic interpolation.

6 Comparison

Interpolation method	Execution time
Nearest interpolation	0.25s
Bilinear interpolation	4.98s
Bicubic interpolation	19.84s
Biquintic interpolation	23.60s

Table 1. Execution time comparison between different interpolation methods.

Although it is clear from the result that with the increase of polynomial coefficient of spline, the quality of the image has been significantly improved, the increase of polynomial coefficient of spline also leads to the increase of program execution time, which is listed in table above. In terms of the degree of graininess and smoothness of the image, the image generated by nearest interpolation and bilinear interpolation still have visible mosaic at 100% view, when it comes to bicubic and biquintic interpolation, the image is much smoother than the lower polynomial coefficient interpolation method. At the same time, there are very few visible mosaics in the result of bicubic and biquintic interpolation.

But it is worth noting that interpolation with too high order spline will also make the image lose high-frequency details, which has great importance in the field of document processing, although we can alleviate it by adding a layer of sharpening filter after interpolation. However, based on the extra calculations brought by high-order spline interpolation and padding with a bigger interpolation frame (kernel), we think this is not worth the gain from increase the order of interpolation splines endlessly. As a reference, the famous picture processing software developed by Adobe use Bicubic interpolation as its default interpolation method.(“How to resize images in Photoshop,” n.d.)

7 Conclusion

In this experimental report, we implemented a variety of interpolation methods including Nearest Neighbor interpolation, Bilinear interpolation, Bicubic interpolation and Biquintic interpolation through python, and applied it to real images. We also compare the result and the program execution efficiency between these interpolation methods.

In my personal view, considering the issues of padding and resources required for operation,

the Bicubic interpolation in the above interpolation method is the relatively optimal solution when adapting them into the real world usage.

8 Supplementary Information

8.1 Code Repository of EE326 Lab

The source code of this lab can be retrived from https://github.com/sparkcyf/SUSTech_EE326_Digital_image_Processing/.

References

Cmglee. n.d. *English: Comparison of nearest-neighbour, linear, cubic, bilinear and bicubic interpolation methods by CMG Lee. The black dots correspond to the point being interpolated, and the red, yellow, green and blue dots correspond to the neighbouring samples. Their heights above the ground correspond to their values.* Accessed February 9, 2021. https://commons.wikimedia.org/wiki/File:Comparison_of_1D_and_2D_interpolation.svg.

"How to resize images in Photoshop." n.d. Accessed February 22, 2021. https://helpx.adobe.com/photoshop/using/resizing-image.html.

Mund, E., P. Hallet, and J. P. Hennart. 1975. "An algorithm for the interpolation of functions using quintic splines." *Journal of Computational and Applied Mathematics* 1 (4): 279–288.

Parker, J. A., R. V. Kenyon, and D. E. Troxel. 1983. "Comparison of interpolating methods for image resampling." *IEEE Transactions on medical imaging* 2 (1): 31–39.

"scipy.interpolate.interp2d —SciPy v1.6.0 Reference Guide." n.d. Accessed February 9, 2021. https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.interp2d.html.

"scipy/scipy." GitHub. n.d. Accessed February 9, 2021. https://github.com/scipy/scipy.