A Brief Introduction to Several Common Image Interpolation Algorithms

Wang Zhuoyang[1]

[1] *12112907, Department of Electrical and Electronic Engineering, SUSTech. Email: glverfer@outlook.com*

Abstract

This is the second lab report for the Digital Image Processing course, which provides an overview of several commonly used image interpolation algorithms, including nearest neighbor interpolation, linear interpolation, and bicubic interpolation. It explains the principles and analysis approaches of these algorithms, and also provides their corresponding Python implementations. Furthermore, this article highlights the significant optimization achieved in code performance during the experimental process.

Keywords: Image Interpolation; Optimization; Digital Image Processing

Contents

Digital Image Processing Laboratory Reports (2023)

Corresponding author
Wang Zhuoyang

1 Introduction

Background

Interpolation algorithms for digital images are a widely used class of classical image processing algorithms. When resizing an image, a common problem is how to determine the color value of each pixel in the new image. This is where interpolation algorithms come in to solve the problem. Different interpolation algorithms can produce different results, and our goal is to make the processed image look as natural and close to the original as possible.

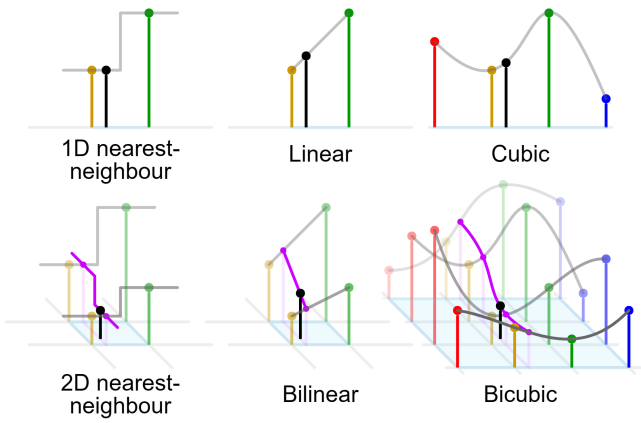


Figure 1. Visualization of three image interpolation algorithms (Commons 2020).

Our Work

In this experiment, we focused on interpolation algorithms for grayscale images, including *nearest neighbor interpolation*, *linear interpolation*, and *bicubic interpolation* (Figure 1). We compared and evaluated their effectiveness, efficiency, and practicality. During the process, we optimized the algorithms for performance several times, mainly by utilizing Numpy matrix operations and some techniques for type usage to avoid inefficient operations in native Python.

Disclaimer

Part of this article has been edited for language by ChatGPT, and the LaTeX template of this paper refers to a previous report (Fan 2021). **But these are in no way related to** the specific content of the text, such as the algorithm analysis, implementation, and optimization discussed in the article. This statement is hereby declared.

2 Nearest Neighbor Interpolation

2.1 Algorithm Principle

Before the Algorithm

The initial step in image interpolation is to resize the target image to the same size as the original image. The reason for doing so is that, for example, when enlarging an image, image interpolation does not fundamentally expand the image outward, but instead fills in reasonable values into gaps that did not originally exist within the image.

For convenience, we first abstract each pixel as a point located at its center, with its height representing the grayscale value of the pixel. We take the width of a pixel in the original image as a unit length of 1, and then scale the new image to fit the coordinate system of the original image.

For example, if the original image has a width of three pixels, their horizontal coordinates are 0, 1, and 2, respectively. If the new image is defined to have a width of four pixels, we map it to the positions approximately at 0, 0.67, 1.33, and 2 on the horizontal axis.

That is, we ensure that the centers of the pixels in the four corners of the new image coincide with those of the original image — We could also align the upper-left corner of the two images and the lower-right corner of the two images, which seems more intuitive. However, the difference in half-pixel scale can be almost negligible, and it is not as convenient to handle as the former, so we still adopt the aforementioned approach.

The Basic Principle of Nearest Neighbor Interpolation

The basic idea of the nearest neighbor algorithm is to directly use the grayscale of the nearest pixel in the original image as the grayscale of the pixel in the new image.

Combining with the previous definition, the centers of the pixels in the original image are located at integer points in the coordinate system. Therefore, finding the nearest neighbor of a pixel actually means finding the nearest integer values for both its horizontal and vertical coordinates. Someway we can use rounding to replace this process:

$$\rho_1(x) = \lfloor x + \frac{1}{2} \rfloor \tag{1}$$

$$\rho_2(x) = \lceil x - \frac{1}{2} \rceil \tag{2}$$

$$\rho_3(x) = round(x) \tag{3}$$

The three kernel functions ρ mentioned above (Equation 1, 2, 3) can all achieve the same effect, except for the behavior when the input decimal part is exactly 0.5. Using the round function in Python directly will round to the nearest even integer at 0.5, which we believe is a good approach, but in reality, we think there is no absolute right or wrong in these issues.

2.2 Train of Thoughts and Pseudos

We first implement this algorithm with the most basic approach, which is to project the new image onto the coordinate system of the original image, and then iterate each pixel and assign it the grayscale value of the nearest integer pixel (Algorithm 1).

The variables marked with $proj$ indicate the mapped coordinates of the new image pixels, while those marked with $nearest$ indicate the coordinates of their corresponding nearest neighbor points.

This is just the basic idea of the algorithm. During the actual implementation, a lot of details need to be taken into account, such as handling the boundaries and improving the code performance.

Algorithm 1: Nearest Neighbor Interpolation

Input: img, dim
Output: img_new

ratio ← (img.shape − 1) / (dim − 1)
img_new ← Zeros(dim)
foreach *(x, y) in img_new* **do**
    (x_proj, y_proj) ← (x, y) × ratio
    (x_nearest, y_nearest) ← (ρ(x_proj), ρ(y_proj))
    img_new[x, y] ← img[x_nearest, y_nearest]
end

The pseudocode only demonstrates the algorithm's functionality, while the specific implementation details can be found in the following section.

2.3 Python Implementation

A Brute-force Implementation

Here is the code directly implemented according to the algorithm. It uses two nested loops to iterate through each point in the new image, and then uses the function $round$ to obtain the nearest neighbor (Code 1).

```python
def nearest(img, dim):
    ratio = (np.array(img.shape) - 1) / (np.array(dim) - 1)

    res = np.zeros(dim)
    for i in range(dim[0]):
        for j in range(dim[1]):
            r, c = np.array([i, j]) * ratio
            res[i, j] = img[round(r), round(c)]
    return res
```

Listing 1. Nearest Neighbor Interpolation (Version 1)

Once we input a 256x256 grayscale image into the program and ask it to interpolate the image to a size of 1024x1024, the program took **14.105** seconds to output the interpolated image.

The main reason for the poor performance is the double loop that enumerates the pixel points. Native Python is an interpreted language with extremely low efficiency in executing statements, so it is very dangerous to process images with loops.

A Systematic Optimization

We optimize the algorithm by considering that the pixel points in each row of the image are aligned, which means that when finding the nearest neighbor of pixels in the same column, we will definitely get pixels that are either to the left or to the right, and the same is true for the row direction. Therefore, we can calculate the nearest neighbor for the row and column coordinates only once, and then directly use the calculation result, reducing the time complexity of the operation from $\mathcal{O}(mn)$ to $\mathcal{O}(m+n)$ (Code 2).

```python
def nearest(img, dim):
    img = img.astype(np.int32)
    ratio = (np.array(img.shape) - 1) / (np.array(dim) - 1)

    round_r_list = np.int32(np.round(np.arange(dim[0]) * ratio[0]))
    round_c_list = np.int32(np.round(np.arange(dim[1]) * ratio[1]))

    res = np.zeros(dim)
```

```
    for i in range(dim[0]):
        round_r = round_r_list[i]
        for j in range(dim[1]):
            round_c = round_c_list[j]
            res[i, j] = img[round_r, round_c]
    return res.astype(np.uint8)
```

Listing 2. Nearest Neighbor Interpolation (Version 2)

In addition to systematic optimizations such as reducing the number of round operations, some significant adjustments were made to improve performance in this optimization:

1. In line 2, we expand the data type of the image to prevent overflow during intermediate calculations. In line 14, we normalize the data type of the image, constraining it to integer values between 0 and 255.
2. In lines 5 and 6, we use $np.int32$ to convert the result of the round function to an integer for use as an array index. Using $int32$ or $int64$ instead of $int16$ results in significant performance gains, as explained in the following Extension section.
3. Experimental results show that converting the data type at the beginning and end of the code is more efficient than converting it every time it is used. Thus, to improve performance, it is necessary to minimize the execution time of native Python code.

Then we tried the example mentioned before, this time it only took **1.046** seconds to give the same result.

Pure Matrix Operations

But it is still not fast enough, and it is possible that we can abandon the loops and only employ matrix operations. So here comes the final version, which gave us the result in only **0.072** seconds (Code 3).

```
def nearest(img, dim):
    img = img.astype(np.int32)
    ratio = (np.array(img.shape) - 1) / (np.array(dim) - 1)

    c_mesh, r_mesh = np.meshgrid(np.arange(dim[1]), np.arange(dim[0]))
    round_r_mat = np.int32(np.round(r_mesh * ratio[0]))
    round_c_mat = np.int32(np.round(c_mesh * ratio[1]))

    res = img[round_r_mat, round_c_mat]
    return res.astype(np.uint8)
```

Listing 3. Nearest Neighbor Interpolation (Version 3)

We use $np.meshgrid$ to generate the coordinates of each point in the image, and then perform global scaling and nearest neighbor calculation. With the efficient matrix operations provided by NumPy, we achieve a win-win situation in terms of both time complexity and code complexity.

In theory, performance comparison should be conducted under various conditions and through multiple experiments. However, due to limited resources and the fact that there are orders of magnitude differences in time consumption between the different optimizations made, we omitted these comparisons and only presented the results from a single run of the experiment.

2.4 Results

The test image (256x256, grayscale) we mentioned before is as shown in Figure 2, and we will not show this image again (we think this image is not very appropriate, in fact).

According to the requirements of the task, we take the last digit of the student ID as 7, and use the algorithm to scale the image to 0.7x and 1.7x, as shown below (Figure 3).

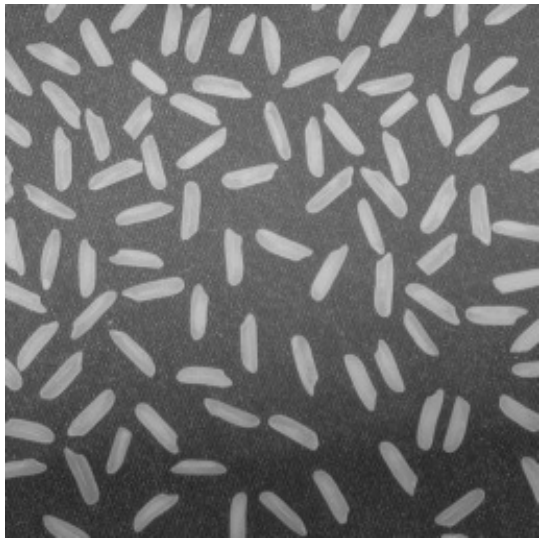
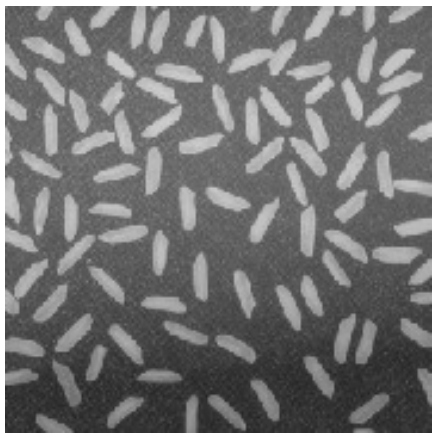
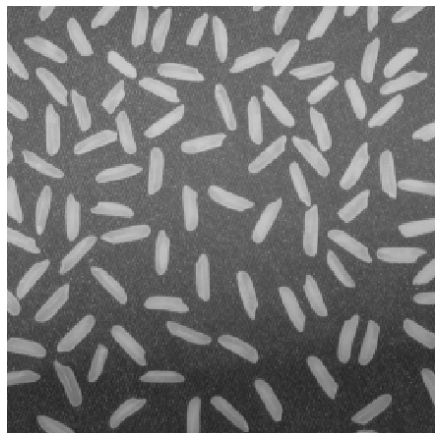


Figure 2. The raw image, 256x256, grayscale.



(a) The shrunk image, 179x179 (0.7x).



(b) The enlarged image, 435x435 (1.7x).

Figure 3. The images scaled with nearest neighbor interpolation.

We can see from the images that the result obtained by using nearest neighbor interpolation appears rough and lacks smooth transitions in some edges, especially in the shrunk one.

3 Bilinear Interpolation

3.1 Algorithm Principle

Basic Process

Regarding the mapping process between the coordinates of the new and old images, we will not go into details here, please refer to the nearest neighbor interpolation section.

Linear interpolation in one dimension refers to replacing the value of an intermediate point with a straight line connecting the two closest points. Given two points at a distance of 1, whose function values are a and b respectively, we can define a function $inter$ to solve for the function value at a position c ($0 \leq c \leq 1$) distance away from the first point (Equation 4):

$$inter(a, b, c) = a + (b - a) * c \tag{4}$$

For bilinear interpolation, we simply do the one-dimension interpolation on two directions. We

can perform row interpolation first and then column interpolation, or perform column interpolation first and then row interpolation. It can be proved that the results obtained from both approaches are consistent.

Edge Processing

When an interpolation algorithm needs a group of pixels around a point, the problem of edge processing arises: that is, how to handle the value of pixels that fall outside the image boundary when calculating pixels close to the edge.

In practice, nearest neighbor interpolation and bilinear interpolation do not need to be particularly concerned about this issue, because these two algorithms involve at most the four pixels around the target pixel. Just make sure to use floor instead of ceiling when dealing with pixels on the image boundary - in fact, this situation will never occur.

In concrete code implementation, we need to pay a little attention, for example, using $np.clip$ to constrain the coordinates to the valid range of the original image while ensuring the equivalence of the algorithm.

By the way, in the subsequent bicubic interpolation algorithm, we need to consider the edge issue seriously because we need to use the sixteen points around each point. Our approach is to use $np.pad$ to extend the edges, and we will explain the specifics in the next section.

3.2 Train of Thoughts and Pseudos

Bilinear interpolation only requires us to perform linear interpolation on rows and columns separately.

We define the function $inter$ as the linear interpolation function in the previous equation (Equation 4), and provide the following pseudocode (Algorithm 2):

Algorithm 2: Bilinear Neighbor Interpolation

Input: img, dim
Output: img_new

ratio $\leftarrow$ (img.shape $-$ 1) / (dim $-$ 1)
img_new $\leftarrow$ Zeros(dim)
foreach *(x, y) in img_new* **do**
    (x_proj, y_proj) $\leftarrow$ (x, y) $\times$ ratio
    (x_topleft, y_topleft) $\leftarrow$ (floor(x_proj), floor(y_proj))
    x_dot, y_dot $\leftarrow$ x_proj $-$ x_topleft, y_proj $-$ y_topleft
    inter_t $\leftarrow$ inter(img[x_topleft, y_topleft], img[x_topleft $+$ 1, y_topleft], x_dot)
    inter_b $\leftarrow$ inter(img[x_topleft, y_topleft $+$ 1], img[x_topleft $+$ 1, y_topleft $+$ 1], x_dot)
    img_new[x, y] $\leftarrow$ inter(inter_t, inter_b, y_dot)
end

In the pseudocodes, the variables marked with $topleft$ represents the coordinate of the top-left of the four nearest points around the desired pixel; the variables marked with dot refers to the values obtained by subtracting this top-left coordinate from the current coordinate, which is actually the decimal part of the mapped coordinate; the variables marked with $inter$ is the interpolation result of the two rows — finally, we interpolate the columns to get the result.

3.3 Python Implementation

The implementation of bilinear interpolation also goes through an iterative process similar to the nearest-neighbor interpolation algorithm described earlier. Here we only provide the final version using matrix operations (Code 4). The code is basically the same as the pseudocode except for the

lack of loops, so no additional comments are added (i.e. do not know how to add comments).

```
def bilinear(img, dim):
    def inter(a, b, c):
        return a + (b - a) * c

    img = img.astype(np.int32)
    ratio = (np.array(img.shape) - 1) / (np.array(dim) - 1)

    c_mesh, r_mesh = np.meshgrid(np.arange(dim[1]), np.arange(dim[0]))
    r_mat = r_mesh * ratio[0]
    c_mat = c_mesh * ratio[1]
    tl_r_mat = np.minimum(np.int32(np.floor(r_mat)), img.shape[0] - 2)
    tl_c_mat = np.minimum(np.int32(np.floor(c_mat)), img.shape[1] - 2)
    tl_r_mat_n = tl_r_mat + 1
    tl_c_mat_n = tl_c_mat + 1
    rdots_mat = r_mat - tl_r_mat
    cdots_mat = c_mat - tl_c_mat

    res = inter(
        inter(img[tl_r_mat, tl_c_mat], img[tl_r_mat, tl_c_mat_n], cdots_mat)
    ,
        inter(img[tl_r_mat_n, tl_c_mat], img[tl_r_mat_n, tl_c_mat_n],
    cdots_mat),
        rdots_mat
    )
    return res.astype(np.uint8)
```

Listing 4. Bilinear Interpolation (Version 3)

The three versions of bilinear interpolation code took **22.450** seconds, **2.691** seconds, and **0.100** seconds, respectively, for the case of zooming in from 256x256 to 1024x1024. It can be seen that there is a qualitative leap.

For comparison purposes, we also used the function $interpolate.interp2d$ from the package $scipy$ to implement bilinear interpolation (Code 5), and under the same conditions it took only **0.016** seconds — It remains stable in terms of computation time when the image scale increases, and is slightly slower than our program in small scales.

We attempted to look into the internal implementation of $interp2d$, but couldn't figure it out. We'll try again next time, definitely, perhaps.

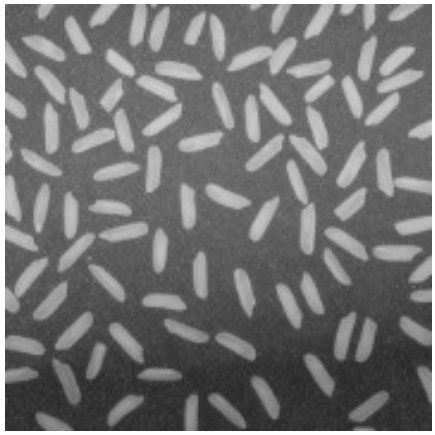
```
def bilinear_scipy(img, dim):
    f = interpolate.interp2d(np.arange(img.shape[0]), np.arange(img.shape
    [1]), img, kind='linear')
    res = f(np.linspace(0, img.shape[1] - 1, dim[1]), np.linspace(0, img.
    shape[0] - 1, dim[0]))
    return res.astype(np.uint8)
```

Listing 5. Bilinear Interpolation (Scipy)

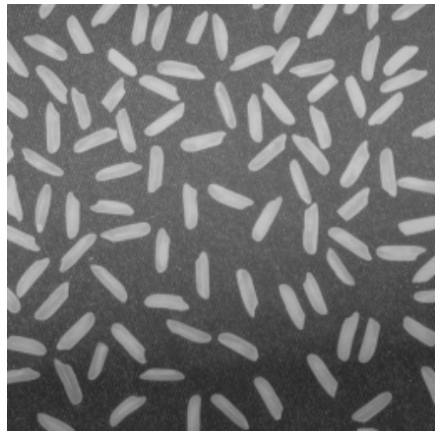
3.4 Results

According to the requirements of the task, we take the last digit of the student ID as 7, and use the algorithm to scale the image to 0.7x and 1.7x, as shown below (Figure 5).

We can see from the images that the result obtained by using bilinear interpolation looks better than the previous one. The image appears smoother and has a better transition.



(a) The shrunk image, 179x179 (0.7x).



(b) The enlarged image, 435x435 (1.7x).

Figure 4. The images scaled with bilinear interpolation.

4 Bicubic Interpolation

4.1 Algorithm Principle

The Bicubic Function

Bicubic interpolation uses cubic splines instead of straight lines in linear interpolation to achieve a smoother interpolation effect. Since the determination of a cubic curve requires four points, when calculating the grayscale of each pixel in bicubic interpolation, the grayscale of the sixteen pixels around it are needed to help solve the curve.

If the grayscale of these points are known, the bicubic kernel function $W(d)$ can be used (Equation 5):

$$W(d) = \begin{cases} (a+2)|d|^3 - (a+3)|d|^2 + 1, & |d| \leq 1 \\ a|d|^3 - 5a|d|^2 + 8a|d| - 4a, & 1 < |d| < 2 \\ 0, & otherwise \end{cases} \tag{5}$$

where a is a variable that we usually choose -0.5 as its value. And then the grayscale value of the target pixel can be represented as a weighted sum (Equation 6):

$$val(x,y) = \sum_{(i,j)} img[i,j] \times W(x-i) \times W(y-j) \tag{6}$$

where i, j is the absolute coordinates of one of the 16 pixels.

Edge Processing

As mentioned earlier, we will use the 16 points surrounding each pixel. However, experimental results show that due to some properties of the cubic spline (observe the function graph to learn more), if we use the 16 points that are as far outside as possible but not out of bounds when computing the edge points, the resulting image will be biased towards black, causing the image to have a black border. Therefore, we have to consider some edge processing methods. We use the $np.pad$ function to expand the original image outwards by two pixels in a $reflect$ manner, where the $reflect$ method fills the extended part with mirrored values from the edge. In fact, the filling method has similar effects in image interpolation and can be chosen freely.

4.2 Train of Thoughts and Pseudos

According to the above approach, the pseudocode is as follows (Algorithm 1).

Algorithm 3: Bicubic Neighbor Interpolation

Input: img, dim
Output: img_new

ratio ← (img.shape − 1) / (dim − 1)
use np.pad to extend img by two pixels outward
img_new ← Zeros(dim)
foreach *(x, y) in img_new* **do**
    (x_proj, y_proj) ← (x, y) × ratio
    (x_topleft, y_topleft) ← (floor(x_proj) − 1, floor(y_proj) − 1)
    val ← 0
    foreach *(i, j) in the 4x4 matrix with the upper left corner at (x_topleft, y_topleft))* **do**
        (x_it, y_it) ← (x_topleft + i, y_topleft + j)
        val ← val + img[x_it, y_it] × W(x_proj − x_it) × W(y_proj − y_it)
    end
    img_new[x, y] ← val.clip(0, 255)
end

Please note that there are some differences from the previous algorithm: first, we need to traverse the 16 points and accumulate them; second, the cubic spline interpolation may produce values higher than the original data range, so the result needs to be constrained within 255; and finally, pay attention to image edge processing.

4.3 Python Implementation

For the bicubic interpolation algorithm, we implemented only the matrix operation version (Code 6).

```python
def bicubic(img, dim, a=-0.5):
    def W(x):
        x_fabs = np.fabs(x)
        res = x_fabs
        flag = x_fabs <= 1
        res[flag] = (a + 2) * x_fabs[flag] ** 3 - (a + 3) * x_fabs[flag] ** 2 + 1
        flag = x_fabs > 1
        res[flag] = a * x_fabs[flag] ** 3 - 5 * a * x_fabs[flag] ** 2 + 8 * a * x_fabs[flag] - 4 * a
        return res

    ratio = (np.array(img.shape) - 1) / (np.array(dim) - 1)
    img = np.pad(img.astype(np.int32), ((2, 2), (2, 2)), 'reflect')

    c_mesh, r_mesh = np.meshgrid(np.arange(dim[1]), np.arange(dim[0]))
    r_mat = r_mesh * ratio[0] + 2
    c_mat = c_mesh * ratio[1] + 2
    tl_r_mat = np.clip(np.int32(np.floor(r_mat)) - 1, 0, img.shape[0] - 4)
    tl_c_mat = np.clip(np.int32(np.floor(c_mat)) - 1, 0, img.shape[1] - 4)

    res = np.zeros(dim)
    for i in range(4):
        it_r_mat = tl_r_mat + i
        for j in range(4):
            it_c_mat = tl_c_mat + j
            res += img[it_r_mat, it_c_mat] * W(r_mat - it_r_mat) * W(c_mat - it_c_mat)
    return np.clip(res, 0, 255).astype(np.uint8)
```

Listing 6. Bicubic Interpolation

It took **2.566** seconds to calculate the same image, which is much slower than the algorithms before — That is exactly how it should be.

And we also write a version that calls the function $interp2d$ from $scipy$ (Code 7):

```
def bicubic_scipy(img, dim):
    f = interpolate.interp2d(np.arange(img.shape[0]), np.arange(img.shape
    [1]), img, kind='cubic')
    res = f(np.linspace(0, img.shape[1] - 1, dim[1]), np.linspace(0, img.
    shape[0] - 1, dim[0]))
    return res.astype(np.uint8)
```

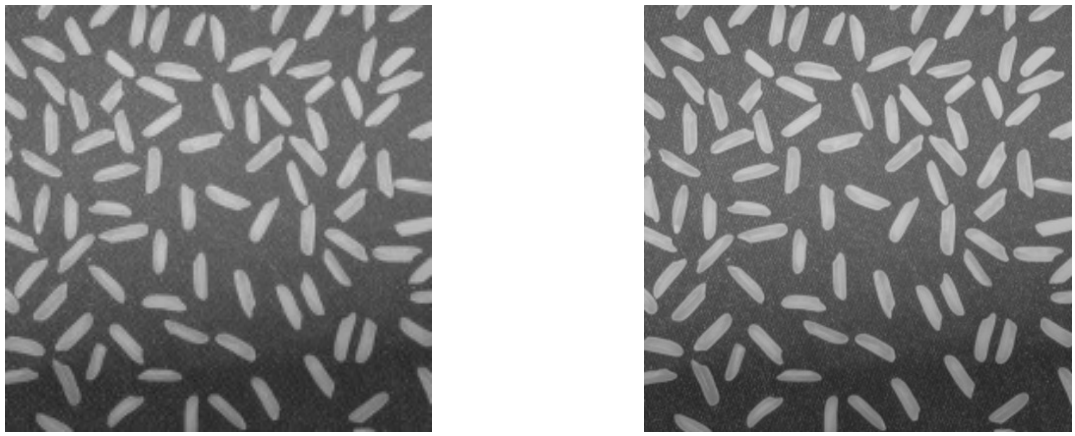
Listing 7. Bicubic Interpolation (Scipy)

It took only **0.042** seconds.

Of course, we think our code could be faster. Because there are still may repeated values during the calculation, which means there must be a better way to make full use of them.

4.4 Results

According to the requirements of the task, we take the last digit of the student ID as 7, and use the algorithm to scale the image to 0.7x and 1.7x, as shown below (Figure 5).



(a) The shrunk image, 179x179 (0.7x).

(b) The enlarged image, 435x435 (1.7x).

Figure 5. The images scaled with bicubic interpolation.

We can observe from the images that the outcome obtained through bicubic interpolation does not differ significantly from the previous bilinear interpolation. Certainly, the effects might be more significant in a larger scale image with more details.

5 Extension

5.1 Other Algorithms Which We Should Use

Of course, there are other image interpolation methods, such as bicubic interpolation, which uses cubic splines for interpolation. However, we believe that this is unnecessary in normal application scenarios.

In fact, linear interpolation is sufficient for most cases, unless the image is special, such as having few details or exhibiting regularity (such as a two-dimensional function waveform). Moreover, high-order interpolation methods consume much more computational resources than linear interpolation, so linear interpolation is the optimal choice for general application scenarios.

5.2 Performance differences due to data types

The reason for this is that NumPy's matrix operations are based on libraries such as BLAS, which are written in other lower-level languages. Common CPUs usually have a width of 32 or 64 bits, including internal registers. Using longer data types reduces the overhead of dividing registers or memory into smaller bits in the case of larger data sizes.

However, this is just a speculation and does not necessarily represent the actual situation. Perhaps some information here can make help: A question from Edureka Community (*Numpy multiplying large arrays with dtype int8 is slow* 2021) and another from Stack Overflow (*Why is it faster to perform float by float matrix multiplication compared to int by int matrix multiplication?* 2017).

6 Conclusion

The summary of the major points found from this experiment report is as follows:

1. The basic idea of image interpolation algorithm is to obtain the value of the middle point by using the values of surrounding points.
2. Edge processing of the image often needs to be considered based on the above situation.
3. Using matrix operation can solve most performance problems.
4. For the balance between quality and performance, linear interpolation algorithm is the most suitable choice.

If there are any other suggestions, they are welcome to be added. Thanks for reading.

References

Commons, W. 2020. *File:Comparison of 1D and 2D interpolation.svg — Wikimedia Commons, the free media repository.* [Online; accessed 25-February-2023]. https://commons.wikimedia.org/w/index.php?title=File:Comparison_of_1D_and_2D_interpolation.svg&oldid=487944857.

Numpy multiplying large arrays with dtype int8 is slow. 2021, March. https://www.edureka.co/community/3249/numpy-multiplying-large-arrays-with-dtype-int8-is-slow.

Fan, Q. 2021. *EE326 Lab Report 2: Image Interpolation.* https://github.com/sparkcyf/SUSTech_EE326_Digital_image_Processing/blob/main/lab2/submission/interpolation_11812418.pdf.

Why is it faster to perform float by float matrix multiplication compared to int by int matrix multiplication? 2017. https://stackoverflow.com/questions/45373679/why-is-it-faster-to-perform-float-by-float-matrix-multiplication-compared-to-int.