Digital Image Processing

Lab2: Image Interpolation

Zhang Liyuan, 12012724

1. Introduction

Digital image is represented as a 2-d array of pixels. (i.e. the minimal unit of digital image which contains the intensity of the unit) The shape of the 2-d array, say m-by-n, is called the resolution rate of the image. The resolution rate determines the number of pixels in a digital image and higher resolution rate means containing more information and so gives better quality. Another concept of resolution rate is the spatial resolution, which is the number of pixels within a spatial unit. (ppi, pixels per inch, is the unit of spatial resolution rate.) The spatial resolution rate determines the overall quality of a digital image of a particular size. However, when we resize a digital image, the spatial resolution rate will change and hence is the quality of the image. While we wish to remain the image quality when resizing, we need to recover the spatial resolution rate. Hence, we need to either down-sample or interpolate the image. There are three basic interpolation methods in digital image processing: 1. Nearest neighbor interpolation, 2. Bilinear interpolation, 3. Bicubic interpolation. In this lab, we are going to implement these three interpolation algorithms with python, and apply the methods to a 256-by-256 digital image to evaluate the three methods.

2. Algorithms

2.1 Spatial Transformation

The whole image interpolation process contains two steps. The first one is spatial transformation, which maps every pixel of the target image to a 2-d coordinate in the source image. Spatial transformation includes affine transform and perspective transform, former of which transforms between two 2-d coordinate systems and latter of which projects a 3-d object to a 2-d coordinate system. For image interpolation, we only need the simplest spatial transformation: spatial scaling.

Suppose the size of the source image is m-by-n, and the size of the target image is p-by-q, and suppose a pixel on the target image with coordinate (x, y). Then the corresponding coordinate with respect to the source image will be (x*m/p, y*n/q). As the figure shown below, the red point is the mapped target pixel into source image. The interpolation process assigns the mapped point a value of intensity, by fitting it with several neighboring points on the source image.

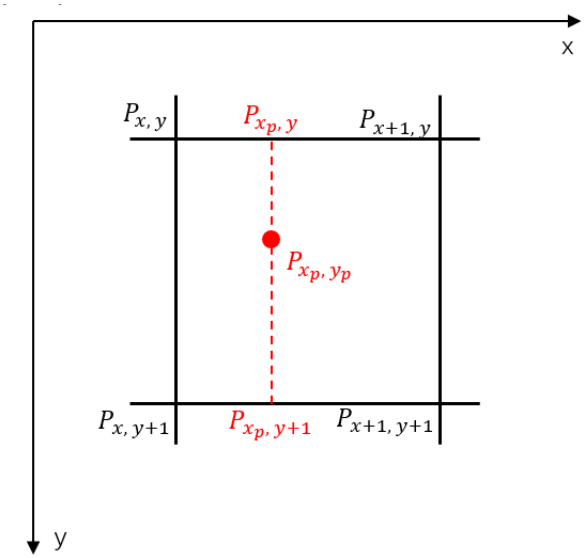

Figure1. spatial transformation

2.2 Nearest Neighbor Interpolation

The nearest neighbor interpolation is the simplest of the three interpolation methods. It assigns the mapped pixel with the value of the nearest pixel on the source image. The nearest neighbor interpolation is easiest to implement and requires least computations. It also preserves the sharpness features of the sources image. However, since the nearest neighbor method assigns the mapped pixel directly with the nearest value, it does not take the differentiation of intensity into accounts, and hence there will be block-shaped pseudo contour in the interpolated image.

Figure 2. source image

Figure 3. nearest-neighbor-interpolated image

2.3 Bilinear Interpolation

Bilinear interpolation is more complex than nearest neighbor. The mapped point is not determined only by the nearest one pixel, but by the nearest four pixels. Bilinear interpolation conducts three unilinear interpolations, two of which are on x-axis, and the other of which is on y-axis. The unilinear interpolation simply assigns the interpolating point with the linear combination of the two nearest points. As the figure 5 below shows, suppose we have an interpolating point with coordinate x. Then its value is determined by the two nearest points x0 and x1, and the value can be calculated as:

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

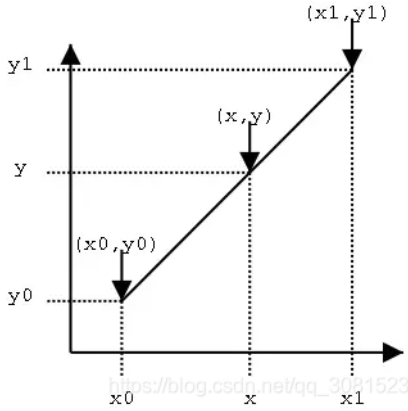

Figure 5. unilinear interpolation

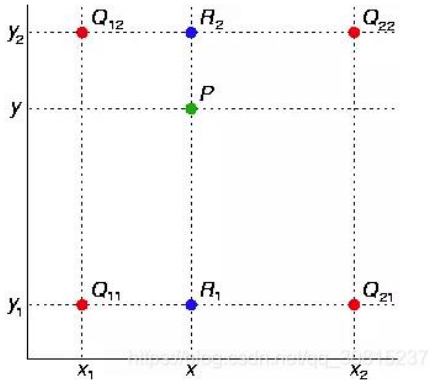

Figure 6. bilinear interpolation

As the figure 6 shown above, suppose now we have an interpolating point P, the value of P is the linear combination of R_1 and R_2, the projected points on two nearest horizontal lines. The value of R_1 and R_2 can be derived by two independent unilinear interpolations, where the value of R_1 is the linear combination of Q_{11} and Q_{21}, the value of R_2 is the linear combination of Q_{12} and Q_{22}. The overall formula for the value of P is:

$$f(x,y1) = f(x1,y1) + \frac{f(x2,y1) - f(x1,y1)}{x2 - x1}(x - x1)$$

$$f(x,y2) = f(x1,y2) + \frac{f(x2,y2) - f(x1,y2)}{x2 - x1}(x - x1)$$

$$f(x,y) = f(x,y1) + \frac{f(x,y2) - f(x,y1)}{y2 - y1}(y - y1)$$

The bilinear interpolation has better performance than nearest neighbor interpolation, and has less computational complexity than bicubic interpolation. It also does not blur the contour of the

source image. However, the image quality can still be improved, by more complex and flexible interpolation algorithms.

Figure 7. source image

Figure 8. bilinear-interpolated image

2.4 Bicubic Interpolation

Bicubic interpolation steps further than bilinear interpolation. The value of the mapped point is determined not by 4, but 16 nearest points, and the value is obtained by fitting a cubic function. As figure 8 shows, suppose now we have a mapped point P, the value of P is weighted sum of the value of the 16 nearest points. The weights of each point are obtained by fitting cubic function. Specifically, by mathematical derivation, we can derive a kernel function which receives the distance between the interpolating point and any nearest point and outputs the weight of that nearest point.

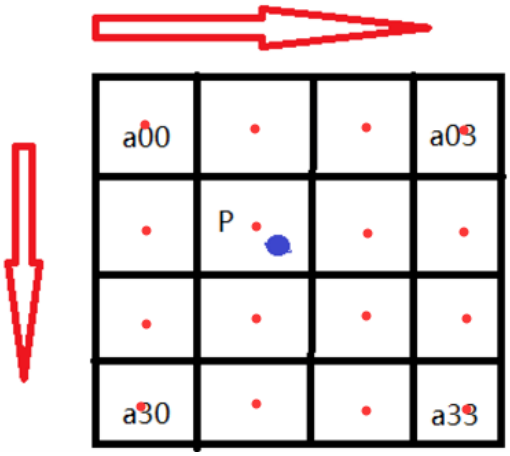

Figure 9. Bicubic interpolation

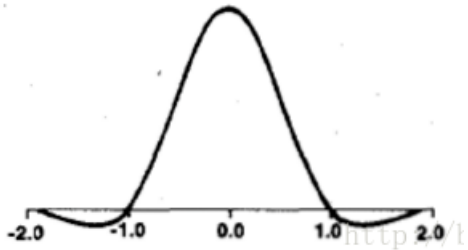

Figure 10. plot of kernel function

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

Figure 11. analytical expression of kernel function

After gaining the weights of all the nearest points, we can just calculate the weighted sum of the values of the 16 nearest points, and get the value of the interpolating point. This process can be accelerated through various optimizations, which will be introduced in the next section.

Bicubic interpolation gives the best image quality among the three interpolation methods. But it thereby has the biggest computational complexity and costs most runtime.

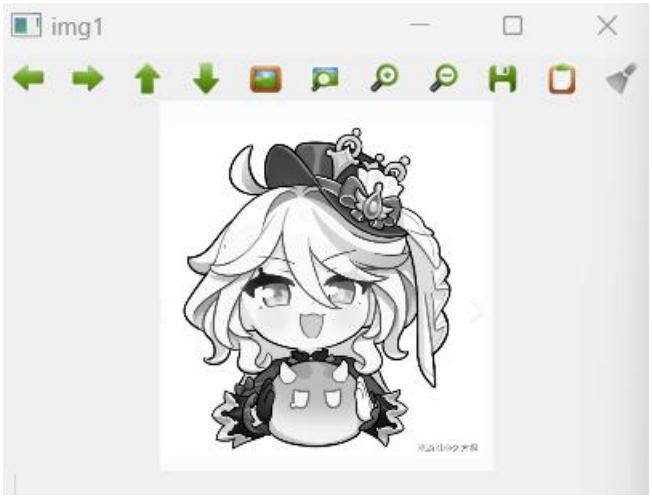

Figure 11. the source image

Figure 12. the bicubic-interpolated image

3. Implementation

3.1 Implementation by Myself

3.1.1 Nearest Neighbor Interpolation

The python implementation of a nearest neighbor interpolation is rather simple. First we make the spatial transform by scaling and map each pixel of target image onto the source image. Then we just use np.round() function to find the nearest neighbor.

```
def nearest_zly(img, dim):
    # numpy array expression of source image
    img1 = np.array(img)
    # numpy array expression initialization of target image
    img2 = np.zeros(dim, dtype=np.uint8)
    # spatial transformation
    x_scale = dim[0] / img.shape[0]
    y_scale = dim[1] / img.shape[1]
    for i in range(dim[0]):
        for j in range(dim[1]):
            # finding the nearest neighbor
            x = int(np.round(i / x_scale)-1)
            y = int(np.round(j / y_scale)-1)
            # assign the value of the nearest neighbor
            img2[i,j] = img1[x, y]
    return img2
```

3.1.2 Bilinear Interpolation

We need an extra boundary process in bilinear interpolation, for the value calculation and assignment of points mapped to the boundary are different from the points in the middle. For the boundary points, we make an unilinear interpolation rather than bilinear interpolation.

```
    def bilinear_zly(img,dim):
    # numpy array expression of source image
    img1 = np.array(img)
    # numpy array expression initialization of target image
    img2 = np.zeros(dim,dtype=np.uint8)
    # spatial transformation
    x_scale = dim[0] / img.shape[0]
    y_scale = dim[1] / img.shape[0]
    for i in range(dim[0]):
        for j in range(dim[1]):
            x = float(i) / x_scale
            y = float(j) / y_scale
            # finding the upper left neighbor
```

```
            x0 = int(np.floor(x))
            y0 = int(np.floor(y))

            # boundary process -- to unilinear interpolation
            if x0 >= img.shape[0] - 1 or y0 >= img.shape[1] - 1 :
                if x0 >= img.shape[0] - 1 :
                    img2[i, j] = int((int(img1[img.shape[0]-1,y0+1]) -
int(img1[img.shape[0]-1,y0])) * (y - y0) + int(img1[img.shape[0]-1,
y0]))
                    break
                elif y0 >= img.shape[1] - 1:
                    img2[i, j] = int(int(img1[x0+1,img.shape[1]-1]) -
int(img1[x0,img.shape[1]-1]) * (x - x0) + int(img1[x0, img.shape[1]-
1]))
                    break
            # calculating the slope of horizontal dimension
            kx_1 = int(img1[x0+1,y0]) - int(img[x0,y0])
            kx_2 = int(img1[x0+1,y0+1]) - int(img[x0,y0+1])
            # calculating the value of R1, R2
            value_x1 = int(img1[x0,y0]) + kx_1*(x-x0)
            value_x2 = int(img1[x0,y0+1]) + kx_2*(x-x0)
            # calculating the slope of vertical dimentsion
            k_y = value_x2 - value_x1
            # calculating the value of P
            value = int(value_x1 + (y-y0)*k_y)
            # assigning the value
            img2[i, j] = value
    return img2
```

3.1.3 Bicubic Interpolation

We first need to construct a function that calculate the value of kernel function which is used in the estimation of weights.

```
    def kernel(x,a):
    y = np.zeros(4,float)
    for i in range(4):
            abs = np.abs(x[i])
            t = abs
            if abs <= 1 and abs > 0:
                y[i] = (a+2)*(t**3) - (a+3)*(t**2) + 1
            elif abs <=2 and abs > 1:
                y[i] = a*(t**3) - (5*a)*(t**2) + (8*a)*t - 4*a
            else:
                y[i] = 0
    return y
```

We then apply the same procedure of spatial transformation and mapping. Worth mentioning, we need to add a small disturbance “epsilon=0.00001” to the scaling factors “x_scale” and “y_scale”. The reason to do this, is when the scaling factor is the multiples of 2, some of the outputs of kernel function will be 0, which means some pixels in the target image will be assigned as 0, visibly a black block. The below figures demonstrate this phenomenon.

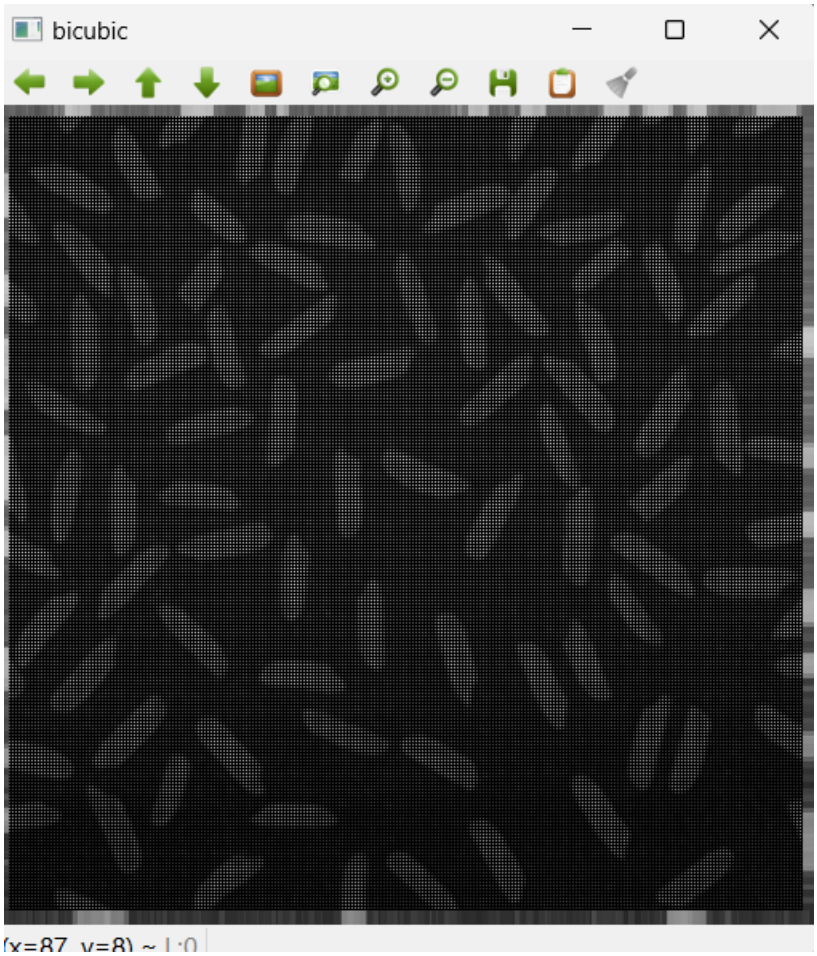

Figure 13. with no disturbance

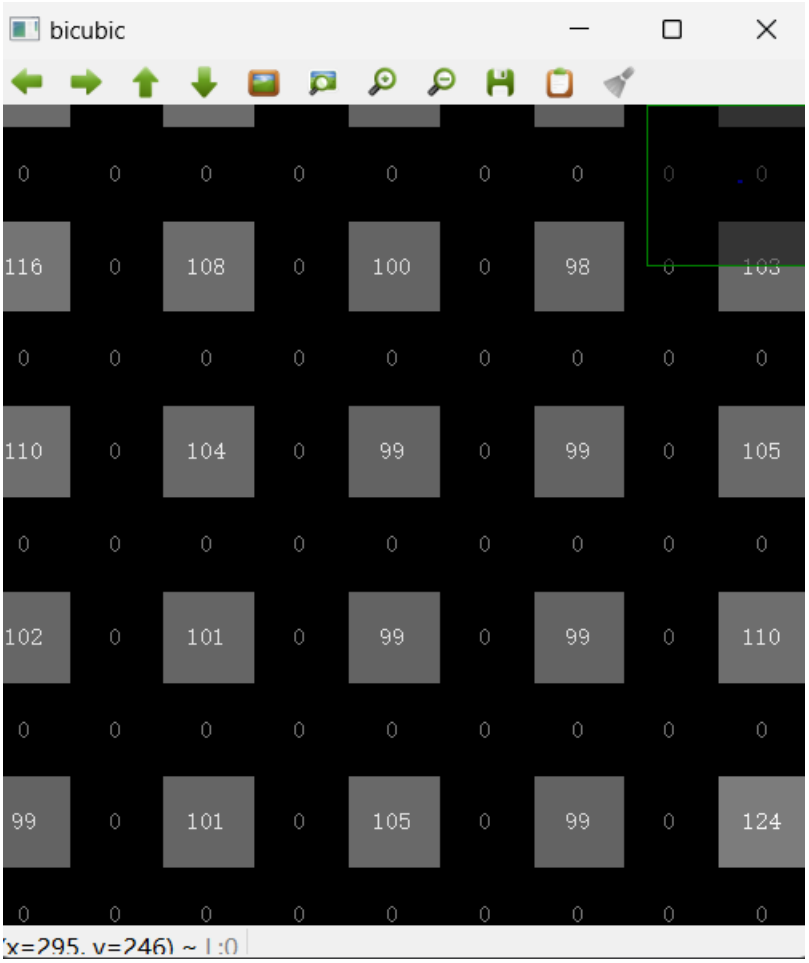

Figure 14. zoomed-out

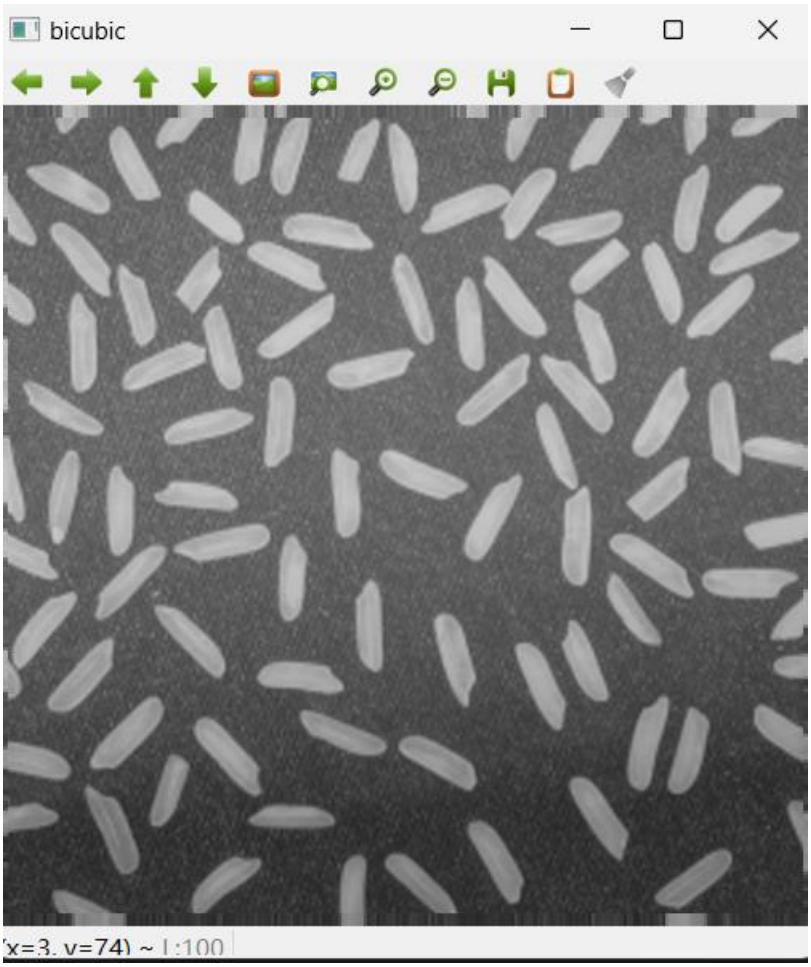

Figure 15. with disturbance

We also need to conduct a boundary process, same as the bilinear interpolation.

```
def bicubic_zly(img,dim):
    # setting the value of a in kernel function
    a = -0.75
    # numpy array expression of source image
    img1 = np.array(img)
    # numpy array expression initialization of target image
    img2 = np.zeros(dim,dtype=np.uint8)
    # a tiny disturbance added to the scales, to avoid mapping to
zeros of kernel funciton.
    epsilon = 0.00001
    # scaling
    x_scale = dim[0] / img.shape[0] + epsilon
    y_scale = dim[1] / img.shape[1] + epsilon
    for i in range(dim[0]):
        for j in range(dim[1]):
            # spatial transformation
```

```
            x = float(i) / x_scale
            y = float(j) / y_scale
            # finding the nearest neighbor
            tmp_x = round(x)
            tmp_y = round(y)
            x0 = int(tmp_x)
            y0 = int(tmp_y)

            # the decimal part of x, y
            u = x - x0
            v = y - y0

            # determining the pattern of distribution of 16 nearest
points.
            if u>=0:
                i_arr = np.array([-1, 0, 1, 2])
            elif u<0:
                i_arr = np.array([-2, -1, 0, 1])

            if v>=0:
                j_arr = np.array([-1, 0 ,1 ,2])
            elif v<0:
                j_arr = np.array([-2, -1, 0, 1])

            # vectorization of u,v
            u_arr = np.zeros(4,float)
            v_arr = np.zeros(4,float)
            for cnt in range(4):
                u_arr[cnt] = u
            for cnt in range(4):
               v_arr[cnt] = v
            # vectorization of x0,y0
            x0_arr = np.zeros(4,int)
            y0_arr = np.zeros(4,int)
            for cnt in range(4):
                x0_arr[cnt] = x0
            for cnt in range(4):
                y0_arr[cnt] = y0

            # boundary processing (a very preliminary one)
            flag = False

            if tmp_x >= img.shape[1] - 4 or tmp_y >= img.shape[0] -4:
                if tmp_x >= img.shape[1] - 4:
```

```
                tmp_x = img.shape[1] -1
                if tmp_y > img.shape[0] - 1:
                    tmp_y = img.shape[0] - 1
                img2[i, j] = img1[img.shape[1]-1, tmp_y]
                flag = True
            elif tmp_y >= img.shape[0] - 4:
                tmp_y = img.shape[0]-1
                if tmp_x > img.shape[1] - 1:
                    tmp_x = img.shape[0] -1
                img2[i, j] = img1[tmp_x,img.shape[0] - 1]
                flag = True

        if tmp_x <= 3 or tmp_y <= 3:
            if tmp_x <= 3:
                img2[i, j] = img1[0, tmp_y]
                flag = True
            elif tmp_y <= 3:
                img2[i, j] = img1[tmp_x,0]
                flag = True

        # non-boundary points calculation
        if flag == False :
            # construct the computational matrix
            temp = [

[img1[x0+i_arr[0],y0+j_arr[0]],img1[x0+i_arr[0],y0+j_arr[1]],img1[x0+
i_arr[0],y0+j_arr[2]],img[x0+i_arr[0],y0+j_arr[3]]],

[img1[x0+i_arr[1],y0+j_arr[0]],img1[x0+i_arr[1],y0+j_arr[1]],img1[x0+
i_arr[1],y0+j_arr[2]],img[x0+i_arr[1],y0+j_arr[3]]],

[img1[x0+i_arr[2],y0+j_arr[0]],img1[x0+i_arr[2],y0+j_arr[1]],img1[x0+
i_arr[2],y0+j_arr[2]],img[x0+i_arr[2],y0+j_arr[3]]],

[img1[x0+i_arr[3],y0+j_arr[0]],img1[x0+i_arr[3],y0+j_arr[1]],img1[x0+
i_arr[3],y0+j_arr[2]],img[x0+i_arr[3],y0+j_arr[3]]]
                    ]
            # calculation of matrix multiplication, and assignment
of the value.
            img2[i, j] =np.dot(np.dot(kernel(u - i_arr , a) ,
temp) , kernel(v - j_arr , a).T)

    return img2
```

3.2 Implementation by OpenCV API

We can use the existing OpenCV function resize(), which is based on interpolation, to resize the image with interpolation algorithm. We can designate the interpolation algorithm via the parameter of the resize function.

```
    def nearest_cv(img,dim):
    return cv2.resize(img, dsize = (dim[1],dim[0]),fx = 1, fy = 1 ,
interpolation=cv2.INTER_NEAREST)
def bilinear_cv(img,dim):
    return cv2.resize(img, dsize = (dim[1],dim[0]),fx = 1, fy = 1 ,
interpolation=cv2.INTER_LINEAR)
def bicubic_cv(img,dim):
    return cv2.resize(img, dsize = (dim[1],dim[0]),fx =1, fy = 1 ,
interpolation=cv2.INTER_CUBIC)
```

4. Evaluation

Figure 16: source image

Implementation by myself:

Figure 17. results

Implementation by OpenCV API:

Figure 18. results

The result from the implementation by myself is consistent with the one we implement with OpenCV API, which verifies the basic correctness of our implementation. We can also see the effects of three different interpolation algorithms: the nearest neighbor with sawtooth contour, the bicubic has slightly better performance than bilinear.

5. Conclusion

In this lab, we learn three most fundamental image interpolation algorithms: nearest neighbor, bilinear and bicubic, from principle to implementation. Resizing a picture of rice by the three different algorithms, we compared the effects of the three algorithms, learning their features. During the coding, a lot of problems are faced, but then most of them are fixed with some solutions.

6. Appendix

Full code:

```
import cv2
import numpy as np
import scipy as sci

def nearest_zly(img, dim):
    # numpy array expression of source image
    img1 = np.array(img)
    # numpy array expression initialization of target image
    img2 = np.zeros(dim, dtype=np.uint8)
    # spatial transformation
    x_scale = dim[0] / img.shape[0]
    y_scale = dim[1] / img.shape[1]
    for i in range(dim[0]):
        for j in range(dim[1]):
            # finding the nearest neighbor
            x = int(np.round(i / x_scale)-1)
            y = int(np.round(j / y_scale)-1)
            # assign the value of the nearest neighbor
            img2[i,j] = img1[x, y]
    return img2


def bilinear_zly(img,dim):
    # numpy array expression of source image
    img1 = np.array(img)

    img2 = np.zeros(dim,dtype=np.uint8)
    # spatial transformation
    x_scale = dim[0] / img.shape[0]# numpy array expression
initialization of target image
    y_scale = dim[1] / img.shape[0]
    for i in range(dim[0]):
        for j in range(dim[1]):
            x = float(i) / x_scale
            y = float(j) / y_scale
            # finding the upper left neighbor
            x0 = int(np.floor(x))
            y0 = int(np.floor(y))

            # boundary process -- to unilinear interpolation
            if x0 >= img.shape[0] - 1 or y0 >= img.shape[1] - 1 :
                if x0 >= img.shape[0] - 1 :
                    img2[i, j] = int((int(img1[img.shape[0]-1,y0+1]) -
```

```
int(img1[img.shape[0]-1,y0])) * (y - y0) + int(img1[img.shape[0]-1,
y0]))
                break
            elif y0 >= img.shape[1] - 1:
                img2[i, j] = int(int(img1[x0+1,img.shape[1]-1]) -
int(img1[x0,img.shape[1]-1]) * (x - x0) + int(img1[x0, img.shape[1]-
1]))
                break
          # calculating the slope of horizontal dimension
          kx_1 = int(img1[x0+1,y0]) - int(img[x0,y0])
          kx_2 = int(img1[x0+1,y0+1]) - int(img[x0,y0+1])
          # calculating the value of R1, R2
          value_x1 = int(img1[x0,y0]) + kx_1*(x-x0)
          value_x2 = int(img1[x0,y0+1]) + kx_2*(x-x0)
          # calculating the slope of vertical dimentsion
          k_y = value_x2 - value_x1
          # calculating the value of P
          value = int(value_x1 + (y-y0)*k_y)
          # assigning the value
          img2[i, j] = value
    return img2


# kernel function definition
def kernel(x,a):
    y = np.zeros(4,float)
    for i in range(4):
          abs = np.abs(x[i])
          t = abs
          if abs <= 1 and abs > 0:
              y[i] = (a+2)*(t**3) - (a+3)*(t**2) + 1
          elif abs <=2 and abs > 1:
              y[i] = a*(t**3) - (5*a)*(t**2) + (8*a)*t - 4*a
          else:
              y[i] = 0
    return y


def bicubic_zly(img,dim):
    # setting the value of a in kernel function
    a = -0.75
    # numpy array expression of source image
    img1 = np.array(img)
    # numpy array expression initialization of target image
```

```
    img2 = np.zeros(dim,dtype=np.uint8)
    # a tiny disturbance added to the scales, to avoid mapping to
zeros of kernel funciton.
    epsilon = 0.00001
    # scaling
    x_scale = dim[0] / img.shape[0] + epsilon
    y_scale = dim[1] / img.shape[1] + epsilon
    for i in range(dim[0]):
        for j in range(dim[1]):
            # spatial transformation
            x = float(i) / x_scale
            y = float(j) / y_scale
            # finding the nearest neighbor
            tmp_x = round(x)
            tmp_y = round(y)
            x0 = int(tmp_x)
            y0 = int(tmp_y)

            # the decimal part of x, y
            u = x - x0
            v = y - y0

            # determining the pattern of distribution of 16 nearest
points.
            if u>=0:
                i_arr = np.array([-1, 0, 1, 2])
            elif u<0:
                i_arr = np.array([-2, -1, 0, 1])

            if v>=0:
                j_arr = np.array([-1, 0 ,1 ,2])
            elif v<0:
                j_arr = np.array([-2, -1, 0, 1])

            # vectorization of u,v
            u_arr = np.zeros(4,float)
            v_arr = np.zeros(4,float)
            for cnt in range(4):
                u_arr[cnt] = u
            for cnt in range(4):
               v_arr[cnt] = v
            # vectorization of x0,y0
            x0_arr = np.zeros(4,int)
            y0_arr = np.zeros(4,int)
```

```
            for cnt in range(4):
                x0_arr[cnt] = x0
            for cnt in range(4):
                y0_arr[cnt] = y0

            # boundary processing (a very preliminary one)
            flag = False

            if tmp_x >= img.shape[1] - 4 or tmp_y >= img.shape[0] -4:
                if tmp_x >= img.shape[1] - 4:
                    tmp_x = img.shape[1] -1
                    if tmp_y > img.shape[0] - 1:
                        tmp_y = img.shape[0] - 1
                    img2[i, j] = img1[img.shape[1]-1, tmp_y]
                    flag = True
                elif tmp_y >= img.shape[0] - 4:
                    tmp_y = img.shape[0]-1
                    if tmp_x > img.shape[1] - 1:
                        tmp_x = img.shape[0] -1
                    img2[i, j] = img1[tmp_x,img.shape[0] - 1]
                    flag = True

            if tmp_x <= 3 or tmp_y <= 3:
                if tmp_x <= 3:
                    img2[i, j] = img1[0, tmp_y]
                    flag = True
                elif tmp_y <= 3:
                    img2[i, j] = img1[tmp_x,0]
                    flag = True

            # non-boundary points calculation
            if flag == False :
                # construct the computational matrix
                temp = [

[img1[x0+i_arr[0],y0+j_arr[0]],img1[x0+i_arr[0],y0+j_arr[1]],img1[x0+
i_arr[0],y0+j_arr[2]],img[x0+i_arr[0],y0+j_arr[3]]],

[img1[x0+i_arr[1],y0+j_arr[0]],img1[x0+i_arr[1],y0+j_arr[1]],img1[x0+
i_arr[1],y0+j_arr[2]],img[x0+i_arr[1],y0+j_arr[3]]],

[img1[x0+i_arr[2],y0+j_arr[0]],img1[x0+i_arr[2],y0+j_arr[1]],img1[x0+
i_arr[2],y0+j_arr[2]],img[x0+i_arr[2],y0+j_arr[3]]],
```

```
[img1[x0+i_arr[3],y0+j_arr[0]],img1[x0+i_arr[3],y0+j_arr[1]],img1[x0+
i_arr[3],y0+j_arr[2]],img[x0+i_arr[3],y0+j_arr[3]]]
                    ]
            # calculation of matrix multiplication, and assignment
of the value.
            img2[i, j] =np.dot(np.dot(kernel(u - i_arr , a) ,
temp) , kernel(v - j_arr , a).T)

    return img2


def nearest_cv(img,dim):
    return cv2.resize(img, dsize = (dim[1],dim[0]),fx = 1, fy = 1 ,
interpolation=cv2.INTER_NEAREST)
def bilinear_cv(img,dim):
    return cv2.resize(img, dsize = (dim[1],dim[0]),fx = 1, fy = 1 ,
interpolation=cv2.INTER_LINEAR)
def bicubic_cv(img,dim):
    return cv2.resize(img, dsize = (dim[1],dim[0]),fx =1, fy = 1 ,
interpolation=cv2.INTER_CUBIC)



img = cv2.imread('./rice.tif', cv2.IMREAD_GRAYSCALE)
source_size = img.shape;
target_size = [source_size[0]*1.4,source_size[1]*1.4];
print(source_size);
print(target_size);
img2 = nearest_zly(img,[int(target_size[0]),int(target_size[1])])
img3 = bilinear_zly(img,[int(target_size[0]),int(target_size[1])])
#img4 = bicubic_zly(img,[int(target_size[0]),int(target_size[1])])
img5 = nearest_cv(img,[int(target_size[0]),int(target_size[1])])
img6 = bilinear_cv(img,[int(target_size[0]),int(target_size[1])])
img7 = bicubic_cv(img,[int(target_size[0]),int(target_size[1])])
cv2.imshow('img1', img)
cv2.imshow('nearest', img2)
cv2.imshow('bilinear', img3)
#cv2.imshow('bicubic',img4)
cv2.imshow('cv_nearest',img5)
cv2.imshow('bilinear',img6)
cv2.imshow('cv_bicubic',img7)
cv2.waitKey(0)
```