SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF MICROELECTRONIC

Digital Image Processing LAB Report #1

Student name & ID: *YUAN Tong 11810818*

Course: *LAB Session I* – Professor: *YU Yajun*
Date: *February 16, 2021*

1. Introduction

In this lab, we will use three kinds of different interpolation method for image resize. The method could be used to resize the image to any size. An image f(x,y) tells us the intensity values at the integral lattice locations, however, if we change the size of the image, some values will be missing, image interpolation refers to the "guess" of intensity values at missing locations.

2. Notation

In the explanation, we define the position of the interpolation point (xx, yy), and the for points around it as (x_1, y_1) to (x_4, y_4) as shown in figure 1.

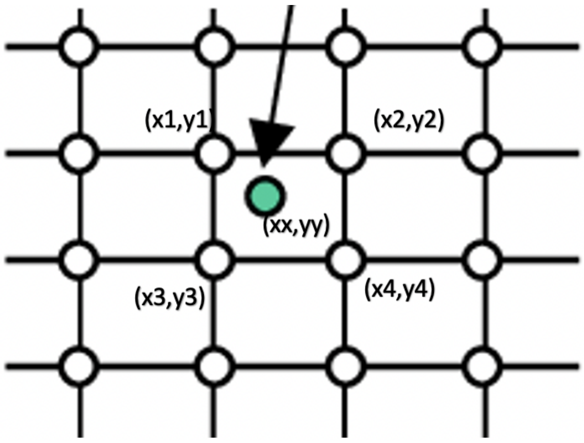


Figure 1: How each related pixels are noted

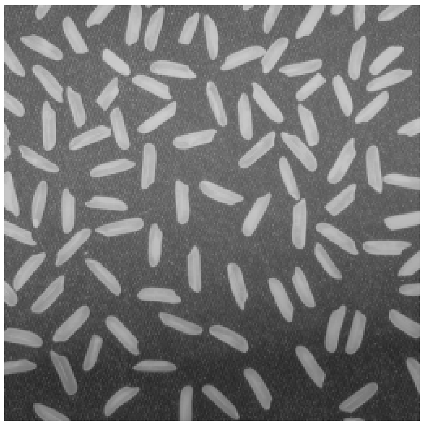
3. Pre-treat

To avoid in complex situation we may meet while handling the edge of the image, we will slightly change the position of the interpolation point so no point will be located in the edge of the image.

```python
def small_map2(x, range):
    ratio = (x-1)/(range)
    return 2.01 + ratio*(range-4.02)
```

Question 1, Part I

Use nearest neighbor interpolation to interpolate a grey scale image shown below.



Analysis. When we apply nearest neighbor interpolation, we will first map the position of the pixels in resized image to the origin image as shown in Figure 2, then we find the nearest pixel of the interpolation point be python function `round()`, which is used to find the nearest integer of a float, finally we use the value of the nearest pixel to replace the value in pixel of new image.

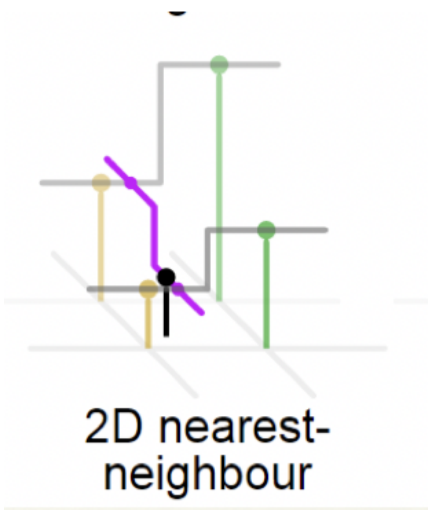


Figure 2: The position of the pixel form new image in origin image after position mapping

Algorithm 1: Nearest Neighbor Interpolation

```
input  : The origin image
output: The resized image
initialization;
special treatment for pixels at the edges;
foreach pixel x position in new image do
    foreach pixel y position in new image do
        xx ← $x \times (\frac{\text{width of source image}}{\text{width of new image}})$
        yy ← $y \times (\frac{\text{height of source image}}{\text{height of new image}})$
        output[x, y] = input[round(xx), round(yy)]
    end
end
```

```python
# Perform transform
for col in range(out_width):
    for row in range(out_height):
        x = col*((in_width-1)/(out_width-1))
        y = row*((in_height-1)/(out_height-1))
        out_image[col, row] = in_image[round(x), round(y)]
```

Question 1, Part II

Use bilinear interpolation to interpolate a grey scale image, the figure is shown in Question 1.

Analysis. When we apply bilinear interpolation, we will first find four point around the output pixels after mapping, as shown in Figure 3, the notated four points is the four points around, and we first connect each two points with same x value, then, we find the z value of the point with y position yy on two lines, finally, we connect two new points and find the z value at point (xx, yy).

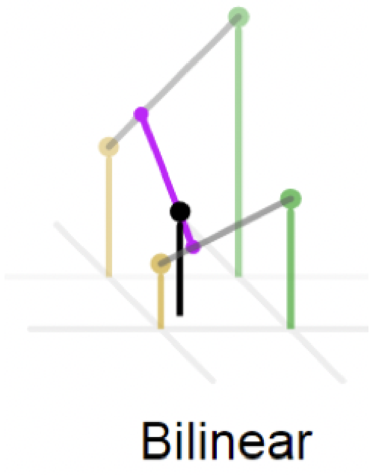


Figure 3: Bilinear Interpolation

Algorithm 2: Bilinear Interpolation

```
input  : The origin image
output: The resized image
initialization;
special treatment for pixels at the edges and on line;
foreach pixel x position in new image do
    foreach pixel y position in new image do
        xx ← $x \times (\frac{\text{width of source image}}{\text{width of new image}})$
        yy ← $y \times (\frac{\text{height of source image}}{\text{height of new image}})$
        left = input[x1,y1]+(input[x3,y3]-input[x1,y1])$\times(yy - y3)$
        right = input[x2,y2]+(input[x4,y4]-input[x2,y2])$\times(yy - y4)$
        output[x, y] = right + (left - right) $\times(xx - x1)$
    end
end
```

```python
def linear(x, y1, y2):
    if y2 > y1:
        return y1 + x * (y2-y1)
    else:
        return y2 + (1-x)*(y1-y2)

for col in range(out_width):
    for row in range(out_height):
        x = small_map(col*((in_width-1)/(out_width-1)), out_width)
        y = small_map(row*((in_height-1)/(out_height-1)), out_height)
        left=linear(y-math.floor(y), in_image[math.floor(x),
         ↪  math.floor(y)], in_image[math.floor(x), math.floor(y)+1])
        right=linear(y-math.floor(y), in_image[math.floor(x)+1,
         ↪  math.floor(y)], in_image[math.floor(x)+1, math.floor(y)+1])

        out_image[col, row] = round(linear(x-math.floor(x), left,
         ↪  right))

print(out_image)
```

Comment. After interpolated by Binearest we found that the output plot looks like many pixels are enlarged directly, that is because the interpolation pixels has the same color as one pixel near it. And for Bilinear interpolation we found that the plot in more smooth due to the color of the pixel is calculated by linear function.

Question 2

Use Python function "interp2d" from packet "scipy" or your own written algorithm to interpolate a grey scale image by using bicubic interpolation.

Analysis. When we applying bicubic interpolation to resize a image, we first choose a 4×4 block with the interpolation pixel at the middle, then we connect each line with a smooth line as shown in 4, next, we connect the points on the previous line with same x value with the interpolation point by another smooth line (cubic function), finally we pick the value on the line at the interpolation point.

Here we will use function `interp2d` in SciPy first, but there is another option to use function `interp1d`, we will use both function to complete the task. The submitted figure is interpolated by `interp2d`. And code and plots produced by `interp1d` are placed in folder `cubic_1d`

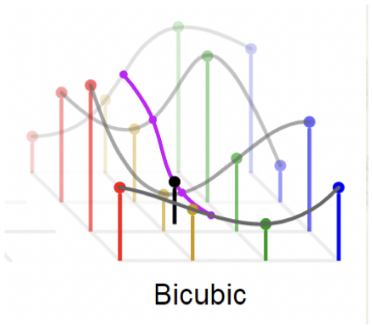


Figure 4: Bicubic Interpolation

Algorithm 3: Bilinear Interpolation

```
input  : The origin image
output: The resized image
initialization;
special treatment for pixels at the edges and on line;
foreach pixel x position in new image do
    foreach pixel y position in new image do
        pix_x ← $x \times (\frac{\text{width of source image}}{\text{width of new image}})$
        pix_y ← $y \times (\frac{\text{height of source image}}{\text{height of new image}})$
        line = [0, 0, 0, 0];
        for $i < 4$ do
            line[i]=interp1d(xs,ys,cubic)(pix_y);
        end
        out_image[x, y]=interp1d(xs, line)(pix_y)
    end
end
```

Interpolation by interp1d

```python
def small_map2(x, range):
    ratio = (x-1)/(range)
    return 2.01 + ratio*(range-4.02)

# The function is used to find the y value of point with in a line
↪ fitted by four points
def find_value(x, y, pix_x):
    f = interpolate.interp1d(x, y, kind='cubic')
    return f(pix_x)

def bicubic_11810818(input_file, dim):
    # Load image
    in_image = io.imread(input_file)
    out_width = dim[0]
    out_height = dim[1]
    in_width = in_image.shape[0]
    in_height = in_image.shape[1]
    out_image = np.zeros(dim, dtype=np.uint8)

    # Perform Exchange
    for col in range(out_width):
        for row in range(out_height):
            pix_x = small_map2(col*((in_width-1)/(out_width-1)),
             ↪ out_width)
            pix_y = small_map2(row*((in_height-1)/(out_height-1)),
             ↪ out_height)
            line = np.zeros(4)
            for i in range(4):
```

```python
            line[i] = find_value(
                [math.floor(pix_y) - 1, math.floor(pix_y), math.floor(pix_y) + 1, math.floor(pix_y) + 2],
                in_image[math.floor(pix_x)-1+i, math.floor(pix_y)-1:math.floor(pix_y)+3],
                pix_y
            )

        out_image[col, row] = find_value(
            [math.floor(pix_x)-1, math.floor(pix_x), math.floor(pix_x)+1, math.floor(pix_x)+2],
            line,
            pix_x
        )

# Save Image
io.imsave("shrank_bicubic_11810818.tif", out_image)
```

Interpolation by interp2d

```python
# Perform Exchange
for col in range(out_width):
    for row in range(out_height):
        pix_x = small_map2(col*((in_width-1)/(out_width-1)), out_width)
        pix_y = small_map2(row*((in_height-1)/(out_height-1)), out_height)
        space = np.zeros([4, 4])
        x_range = [math.floor(pix_x) - 1, math.floor(pix_x), math.floor(pix_x) + 1, math.floor(pix_x) + 2]
        y_range = [math.floor(pix_y) - 1, math.floor(pix_y), math.floor(pix_y) + 1, math.floor(pix_y) + 2]

        # print("x: "+str(x_range))
        # print("y: "+str(y_range))
        # print(str(pix_x) + " " + str(pix_y))

        for i in range(4):
            for j in range(4):
                space[i, j] = in_image[x_range[i], y_range[j]]

        out_image[col, row] = find_value(
            x_range,
            y_range,
            space,
            pix_x,
            pix_y
        )
```

Comment. Interestingly, we found that the image produced by two function are different, and the one produced by `interp1d` seems more smooth, so we analyse the different of two function, we found that, when we input a special array that, in the center 2x2 box of the array, one number is significantly differ from others, and the effect of this number is quite different in two functions.

```
def interp_test():

    x = [1, 2, 3, 4]
    y = x;
    z = [[106, 110, 154, 188],
         [103,  99, 12, 157],
         [100,  97, 101, 109],
         [102, 104,  96, 105]]


    f1_1 = interpolate.interp1d(x, z[0], kind='cubic')
    f1_2 = interpolate.interp1d(x, z[1], kind='cubic')
    z1 = [f1_1(2.5), f1_2(2.5), f1_1(2.5), f1_2(2.5)]
    f1_3 = interpolate.interp1d(x, z1, kind='cubic')
    print("interpolate.interp1d get: " + str(f1_3(2.5)))

    f2 = interpolate.interp2d(x, y, z, kind='cubic')
    print("interpolate.interp2d get: " + str(f2(2.5, 2.5)[0]))
```

We found that the number get by `interp1d` is 88 and 66 for `interp2d`, so in `interp2d` the number is closer to the special number 12, which is because in `interp2d` the interplation is down by fit 16 numbers with a plane and in `interp1d` we fit each 4 numbers in a line and than fit the line go though the interpolation point.

Output images and comparison

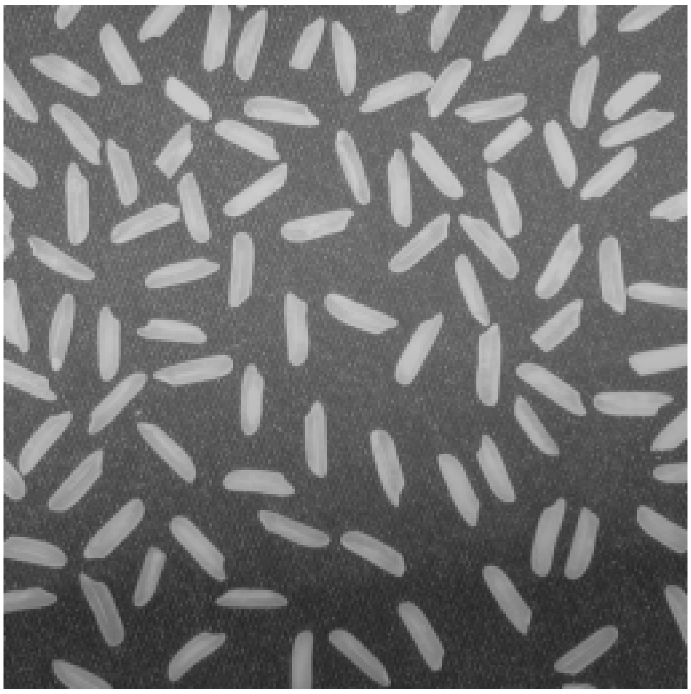


Figure 5: The original image

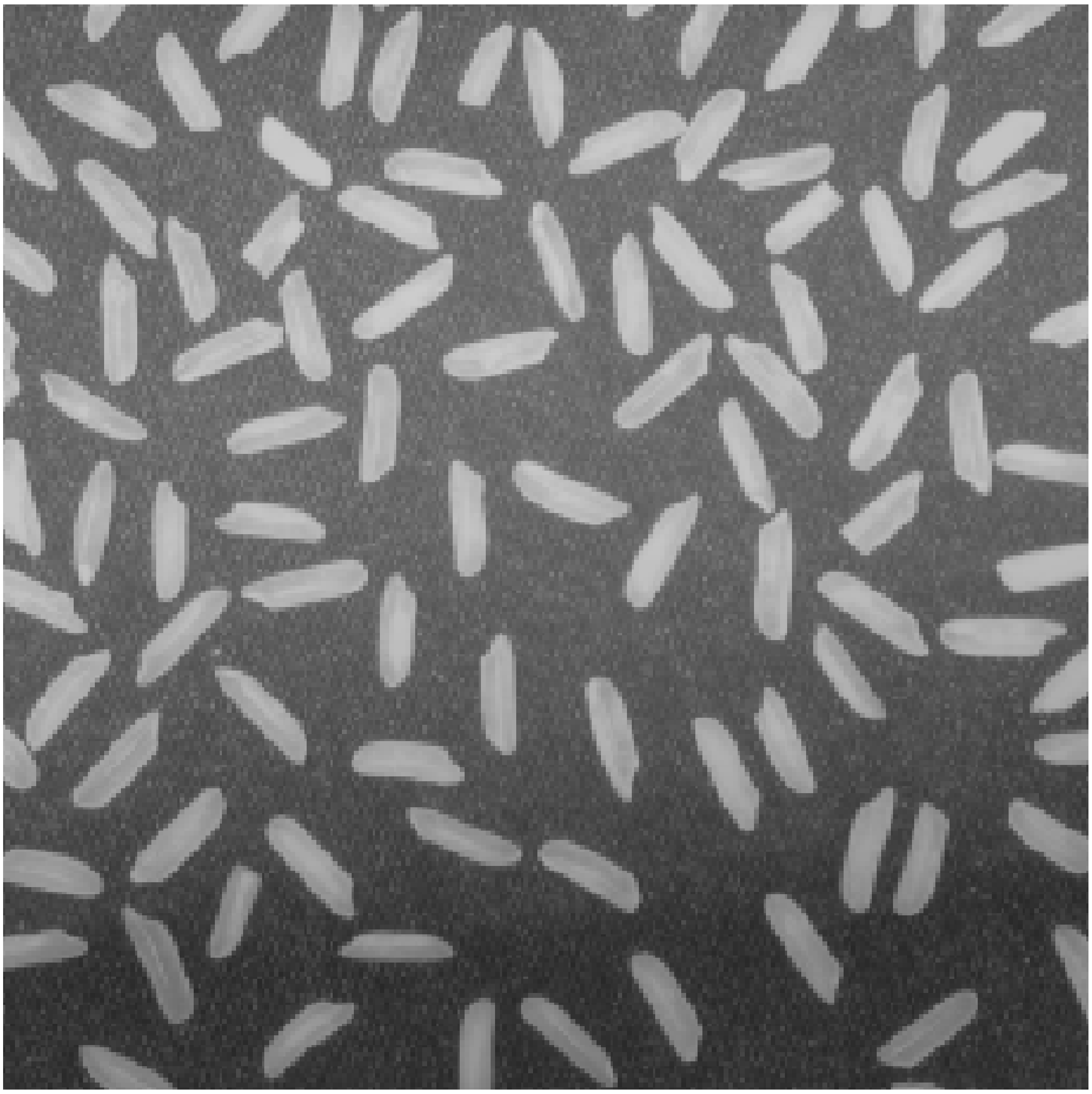


Figure 6: enlarged image with nearest neibor interpolation

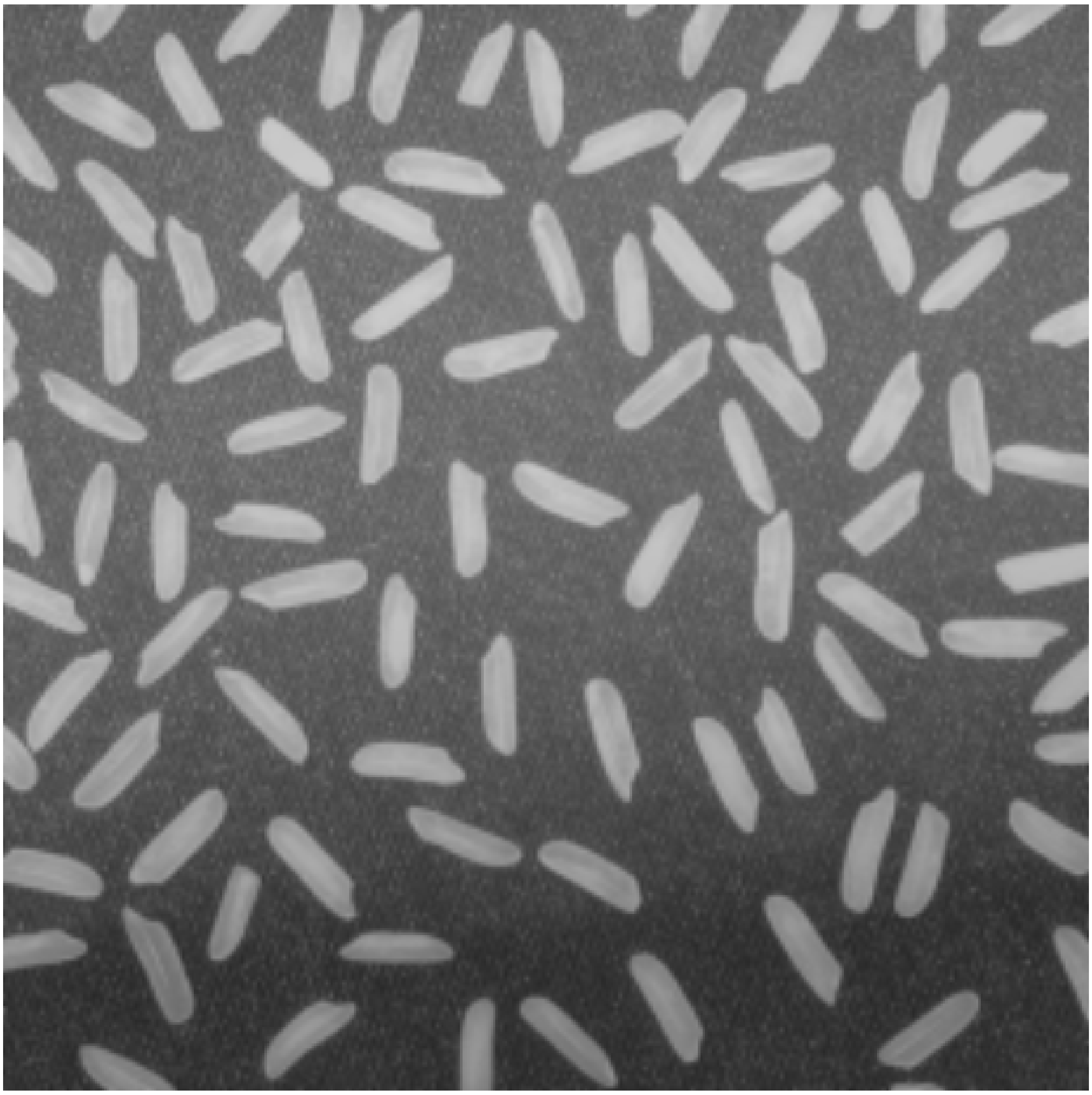


Figure 7: enlarged image with bilinear interpolation

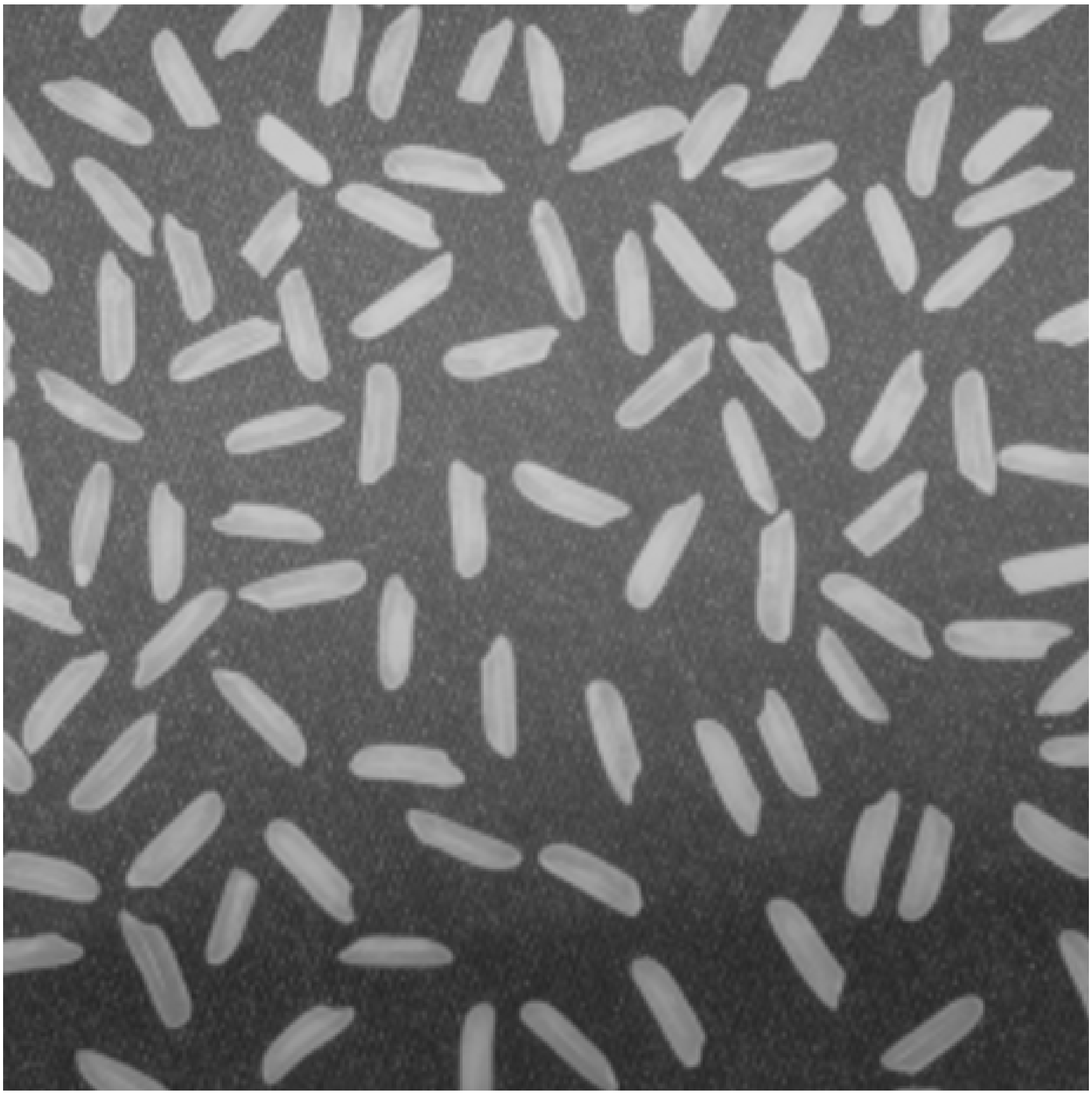


Figure 8: enlarged image with bicubic interpolation (1d)

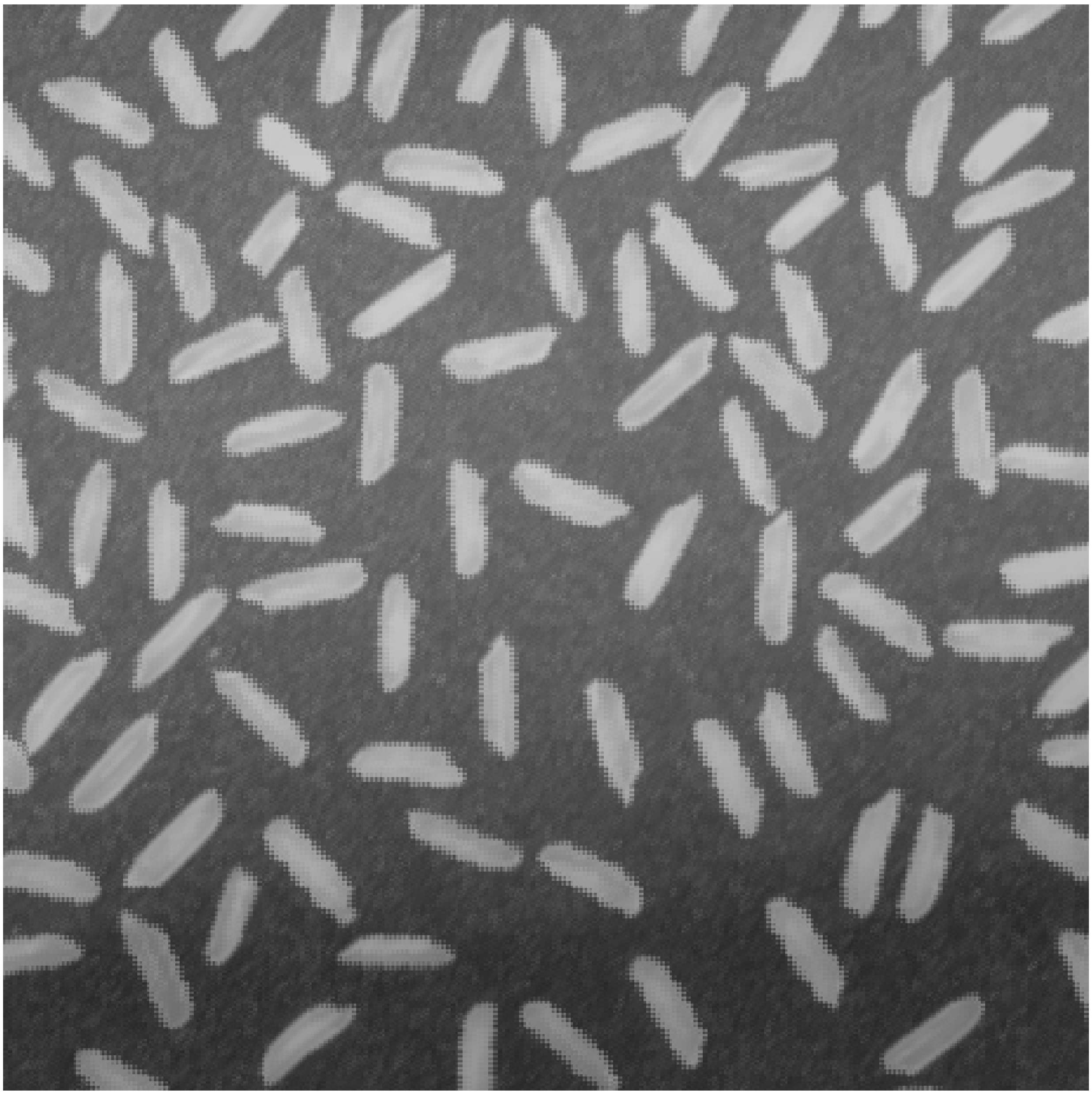


Figure 9: enlarged image with bicubic interpolation (2d)

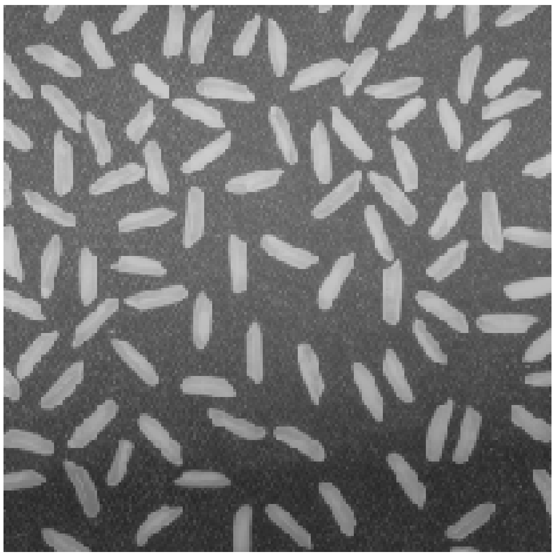


Figure 10: shrank image with nearest neibor interpolation

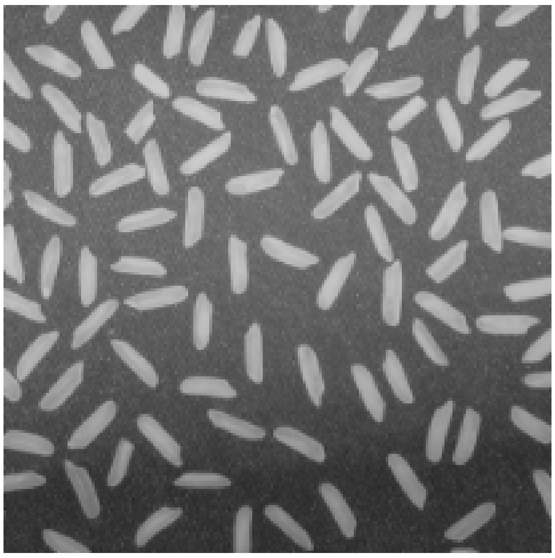


Figure 11: shrank image with bilinear interpolation

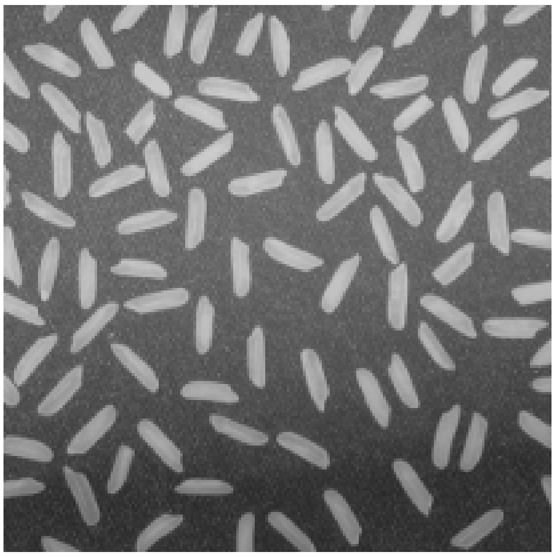


Figure 12: shrank image with bicubic interpolation

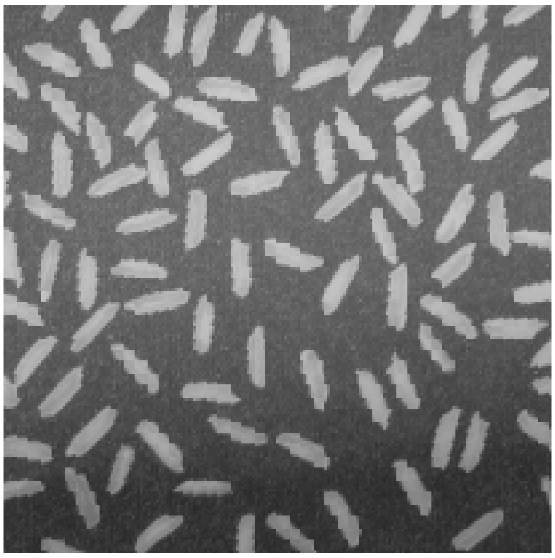


Figure 13: shrank image with bicubic interpolation(2)

Analysis. By cooperation, we find that either for enlarged or for shrank image the bilinear performs better than nearest neighbor and bicubic performs better than bilinear.