Data Visualization - Homework 2 - Report

邓淇升 大数据学院 16307110232

1 Restate the basic global thresholding algorithm using the histogram of an image instead of the image itself.

- **Python script code:**

  1_basic_global_threshoding.py

- **Input:**

  noisy_fingerprint.tif

- **Output:**

  noisy_fingerprint_by_global_threshoding.png

- **Description:**

参考最佳全局阈值处理的 OTSU 算法，可以使用直方图数据代替图像数据以描述基本全局阈值处理算法。基本全局阈值处理算法步骤描述如下：

第一步：

计算输入图像的直方图，得到全局权值数组$hist[0:L]$。其中元素$hist[i]=n_i$，表示灰度值i在图像中的出现次数。i的取值范围为$0,1,\cdots,L-1$，灰度处理通常取$L=256$。

第二步：

设定全局阈值的初始估计值$T=T_0$和最小差值ΔT_0。

第三步：

使用全局阈值T分割全局权值数组$hist[0:L]$，得到局部权值数组$G_1[0:T]$和$G_2[T:L]$。

第四步：

对局部权值数组G_1和G_2分别计算灰度索引值的加权平均值，得到标量m_1和m_2。

第五步：

计算新全局阈值$T'=\frac{m_1+m_2}{2}$，计算迭代差值$\Delta T=|T'-T|$。

第六步：

比较迭代差值ΔT与最小差值ΔT_0。若$\Delta T<\Delta T_0$则取全局阈值为T'，结束算法，输出全局阈值T'；若$\Delta T\geq\Delta T_0$则使用迭代阈值$T=T'$，回到第三步。

- **Source Code:**

```python
import os  # 文件操作
import time  # 时间操作

import numpy as np  # 数组操作
from PIL import Image  # 图像操作


class ImageData:

    # 初始化实例
    def __init__(self, image_name):
        self.image_name = image_name.split('.')[0]  # 图像的名称
        self.image_path = os.getcwd() + '\\' + image_name  # 图像的路径

    # 图像输入
    def input_image(self):
        image = Image.open(self.image_path)  # 输入图像
        self.image_matrix = np.array(image.convert('L'))  # 将图像转换成灰度矩阵
        self.image_shape = self.image_matrix.shape  # 图像的形状
        self.image_height, self.image_width = self.image_shape  # 图像的高与宽

    # 基本全局阈值算法
    def global_thresholding(self):
        histogram = np.histogram(self.image_matrix, range(257))[0]  # 计算图像的全局直方图
        t_star = int(np.average(np.arange(256), weights=histogram))  # 设定初始全局阈值
        t, iteration = 0, 0  # 记录全局阈值与迭代次数
        while np.abs(t - t_star) > 1 and iteration < 10000:  # 最小阈值差值设为1，设定迭代上限防止死循环
            t = t_star  # 当前全局阈值
            m_1 = int(np.average(np.arange(t), weights=histogram[:t]))  # G_1像素组的灰度均值
            m_2 = int(np.average(np.arange(t, 256), weights=histogram[t:]))  # G_2像素组的灰度均值
            t_star = int(0.5 * (m_1 + m_2))  # 计算新的全局阈值
            iteration += 1  # 记录迭代次数
        self.global_matrix = np.where(self.image_matrix > t_star, 255, 0)  # 图像二值化处理

    # 图像输出
    def output_image(self):
        image_splice = Image.new(mode='L', size=(self.image_width * 2 + 48, self.image_height + 32), color=215)  # 生成空白拼接图像
        image_splice.paste(im=Image.fromarray(self.image_matrix), box=(16, 16))  # 生成原图灰度化图像
        image_splice.paste(im=Image.fromarray(self.global_matrix), box=(self.image_width + 32, 16))  # 生成全局阈值处理图像
        image_splice.save(self.image_name + '_by_global_thresholding.png')  # 保存图像
        # image_splice.show()  # 运行程序时显示图像

    # 图像分割处理接口
    def image_segmentation(self):

        # 输入
        self.input_image()  # 输入图像

        # 基本全局阈值处理
        self.global_thresholding()  # 调用基本全局阈值算法

        # 输出
        self.output_image()  # 输出图像


if __name__ == '__main__':

    print()  # 空行

    # 处理图像: noisy_fingerprint.tif
    image_name = 'noisy_fingerprint.tif'  # 图像的名称
    image_data = ImageData(image_name=image_name)  # 实例化ImageData对象
    print('Image: %s\n' % image_name)  # 实例的名称

    # 基本全局阈值算法
    start = time.time()  # 实例开始测试时刻
    image_data.image_segmentation()  # 调用基本全局阈值算法进行图像分割
    end = time.time()  # 实例结束测试时刻
    print('Program execute time: %.2f s\n' % (end - start))  # 实例测试时长
```

- **Result:**

使用直方图数据作为输入的基本全局阈值算法处理后的图像如下所示，原图像为noisy_fingerprint.tif。程序的运行时间为 0.09 秒。

2 Implement the locally adaptive thresholding algorithm based on moving average and local OSTU.

- **Python script code:**

  2_locally_adpative_thresholding.py

- **Input:**

  writing_round.tif

  writing_stripe.tif

- **Output:**

  writing_round_by_local_otsu.png

  writing_round_by_moving_average.png

  writing_stripe_by_local_otsu.png

  writing_stripe_by_moving_average.png

- **Description:**

Python 源代码中给出了分别基于移动平均方法和局部 OTSU 方法的局部可变阈值处理算法。调用 OTSU 算法可以对图像进行最佳全局阈值处理，但全局算法容易受到噪声和非均匀光照的影响，改进方案是使用局部可变阈值算法进行处理。

局部 OTSU 算法的实现重点是局部区域的选取，使用该算法前需要对图像的灰度分布情况进行人工观察，以确定局部区域的形状。例如，具有正弦干涉条纹影响的图像应采用较窄的水平移动窗口，而具有斑点影响的图像应采用较小的正方形移动窗口。

移动平均方法的实现重点是对图像矩阵进行合适的变换以适应 Z 字形遍历，代码中使用了 NumPy 软件包的数组形状变换方法以实现该需求。另外，移动平均方法中可以调整的参数为移动窗口的长度，实验证明不宜设置较大的移动窗口，理想的取值在 20 附近。

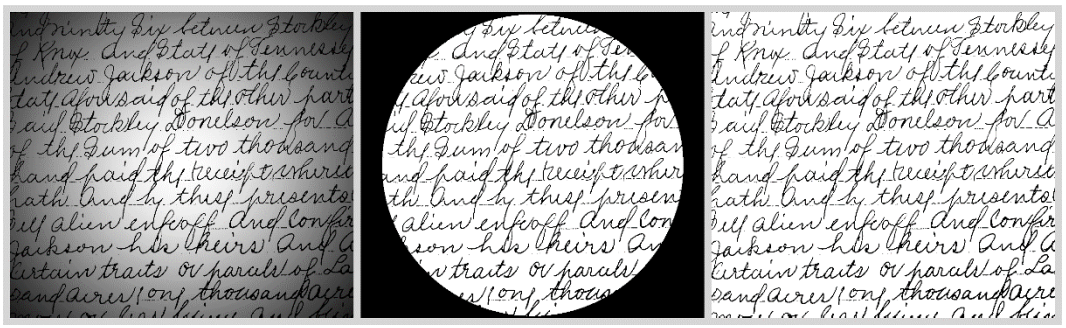
- **Source Code:**

```python
import itertools  # 迭代操作
import os  # 文件操作
import time  # 时间操作

import numpy as np  # 数组操作
from PIL import Image  # 图像操作


class ImageData:

    # 初始化实例
    def __init__(self, image_name):
        self.image_name = image_name.split('.')[0]  # 图像的名称
        self.image_path = os.getcwd() + '\\' + image_name  # 图像的路径

    # 图像输入
    def input_image(self):
        image = Image.open(self.image_path)  # 输入图像
        self.image_matrix = np.array(image.convert('L'))  # 将图像转换成灰度矩阵
        self.image_size = self.image_matrix.size  # 图像的面积
        self.image_shape = self.image_matrix.shape  # 图像的形状
        self.image_height, self.image_width = self.image_shape  # 图像的高与宽
```

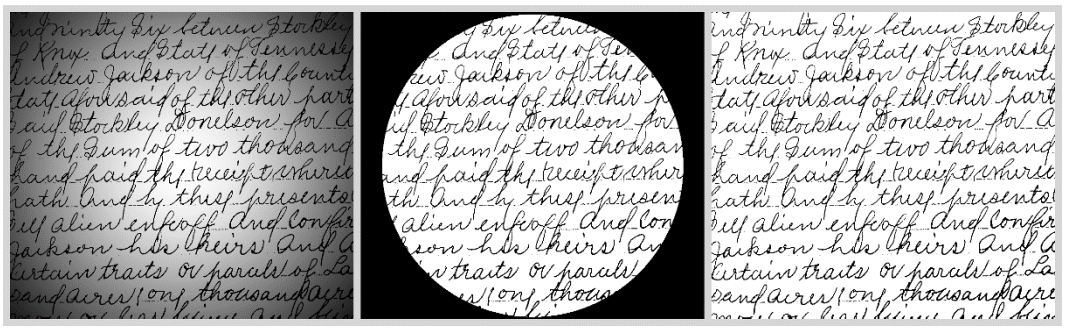
```
    # 任意区域OTSU算法
    def region_otsu(self, matrix):
        region_histogram = np.histogram(matrix, range(257))[0] / matrix.size  # 计算区域的归一化直方图
        prob_K = np.cumsum(region_histogram)  # 计算区域的累积和P1(k)，256维行向量
        mean_K = np.cumsum(region_histogram * np.array(range(256)))  # 计算区域的累积均值m(k)，256维行向量
        mean_G = mean_K[-1]  # 计算区域的灰度均值mG，标量
        epsilon = 1e-7  # 防止方差出现nan值，标量
        var_B_K = (mean_G * prob_K - mean_K)**2 / ((prob_K + epsilon) * (1 - prob_K - epsilon))  # 计算区域的类间方差sigmaB^2(k)，256维行向量
        k_star = int(np.mean(np.where(var_B_K == np.nanmax(var_B_K[1:-1]))))  # 找到最大化类间方差的k值平均值
        return np.where(matrix > k_star, 255, 0)  # 图像二值化处理

    # 全局OTSU算法
    def global_otsu(self):
        self.global_otsu_matrix = self.region_otsu(self.image_matrix)  # 应用OTSU算法的最佳全局阈值处理

    # 局部OTSU算法
    def local_otsu(self, height, width):
        self.local_adaptive_matrix = np.zeros(self.image_shape)  # 新建图像矩阵
        horizontal = zip(range(0, self.image_width, width), range(width, self.image_width + width, width))  # 将图像横向切割为等宽窗口，边界除外
        vertical = zip(range(0, self.image_height, height), range(height, self.image_height + height, height))  # 将图像纵向切割为等高窗口，边界除外
        for (hl, hr), (vt, vb) in itertools.product(horizontal, vertical):  # 将图像划分为等大小窗口，边界除外
            self.local_adaptive_matrix[vt:vb, hl:hr] = self.region_otsu(self.image_matrix[vt:vb, hl:hr])  # 应用OTSU算法的局部阈值处理

    # 移动平均算法
    def moving_average(self, length, const):

        # 输入阶段
        mean = np.zeros(self.image_shape)  # 新建均值矩阵
        mean[::2, :] = self.image_matrix[::2, :]  # 偶数行不变
        mean[1::2, :] = self.image_matrix[1::2, ::-1]  # 奇数行翻转
        mean = mean.reshape(-1)  # 将Z字形矩阵转换成均值数组

        # 输出阶段
        threshold = np.concatenate((mean[:length], (mean[length:self.image_size] - mean[:self.image_size - length]) / length))  # 新建阈值数组
        threshold[length - 1:] = np.cumsum(threshold[length - 1:])  # 计算移动平均
        threshold = threshold.reshape(self.image_shape)  # 将阈值数组转换成阈值矩阵
        threshold[1::2, :] = threshold[1::2, ::-1]  # 奇数行翻转

        self.local_adaptive_matrix = np.where(self.image_matrix > const * threshold, 255, 0)  # 图像二值化处理

    # 图像输出
    def output_image(self):
        image_splice = Image.new(mode='L', size=(self.image_width * 3 + 64, self.image_height + 32), color=215)  # 生成空白拼接图像
        image_splice.paste(im=Image.fromarray(self.image_matrix), box=(16, 16))  # 生成原图灰度化图像
        image_splice.paste(im=Image.fromarray(self.global_otsu_matrix), box=(self.image_width + 32, 16))  # 生成全局OTSU处理图像
        image_splice.paste(im=Image.fromarray(self.local_adaptive_matrix), box=(self.image_width * 2 + 48, 16))  # 生成局部可变阈值处理图像
        image_splice.save(self.image_name + '_by_' + self.mode + '.png')  # 保存图像
        # image_splice.show()  # 运行程序时显示图像

    # 局部阈值处理接口
    def adaptive_thresholding(self, mode='moving_average', height=512, width=512, length=20, const=0.5):

        # 输入
        self.input_image()  # 输入图像

        # 全局阈值处理
        self.global_otsu()  # 调用全局OTSU算法

        # 局部可变阈值处理
        self.mode = mode  # 局部阈值处理的模式
        if mode == 'local_otsu':
            self.local_otsu(height, width)  # 调用局部OTSU算法
        elif mode == 'moving_average':
            self.moving_average(length, const)  # 调用移动平均算法
        else:
            raise BaseException('wrong mode')  # 抛出模式错误异常

        # 输出
        self.output_image()  # 输出图像


if __name__ == '__main__':

    print()  # 空行

    # 处理图像: writing_round.tif
    image_name = 'writing_round.tif'  # 图像的名称
    image_data = ImageData(image_name=image_name)  # 实例化ImageData对象
    print('Image: %s\n' % image_name)  # 实例的名称

    # 局部OTSU算法
    start = time.time()  # 实例开始测试时刻
    image_data.adaptive_thresholding(mode='local_otsu', height=50, width=50)  # 调用局部OTSU算法进行局部可变阈值处理
    end = time.time()  # 实例结束测试时刻
    print('Processing mode: %s' % 'local OTSU')  # 实例测试的算法
    print('Program execute time: %.2f s\n' % (end - start))  # 实例测试时长

    # 移动平均算法
    start = time.time()  # 实例开始测试时刻
    image_data.adaptive_thresholding(mode='moving_average', length=20, const=0.5)  # 调用移动平均算法进行局部可变阈值处理
    end = time.time()  # 实例结束测试时刻
    print('Processing mode: %s' % 'moving average')  # 实例测试的算法
    print('Program execute time: %.2f s\n' % (end - start))  # 实例测试时长

    print()  # 空行

    # 处理图像: writing_stripe.tif
    image_name = 'writing_stripe.tif'  # 图像的名称
    image_data = ImageData(image_name=image_name)  # 实例化ImageData对象
    print('Image: %s\n' % image_name)  # 实例的名称

    # 局部OTSU算法
    start = time.time()  # 实例开始测试时刻
    image_data.adaptive_thresholding(mode='local_otsu', height=350, width=16)  # 调用局部OTSU算法进行局部可变阈值处理
    end = time.time()  # 实例结束测试时刻
    print('Processing mode: %s' % 'local OTSU')  # 实例测试的算法
    print('Program execute time: %.2f s\n' % (end - start))  # 实例测试时长

    # 移动平均算法
    start = time.time()  # 实例开始测试时刻
    image_data.adaptive_thresholding(mode='moving_average', length=20, const=0.5)  # 调用移动平均算法进行局部可变阈值处理
    end = time.time()  # 实例结束测试时刻
    print('Processing mode: %s' % 'moving average')  # 实例测试的算法
    print('Program execute time: %.2f s\n' % (end - start))  # 实例测试时长
```

- **Result:**

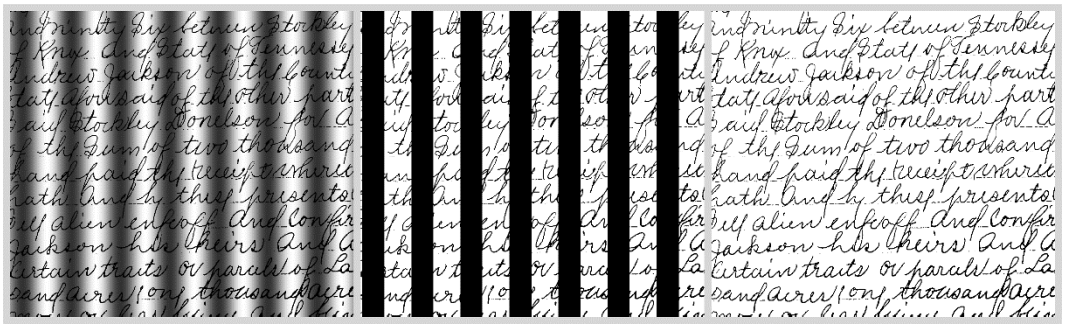
使用分别基于 OTSU 方法和移动平均方法的局部可变阈值算法处理后的图像如下所示，原图像为 writing_round/stripe.tif。四个子程序的平均运行时间为 0.1 秒。



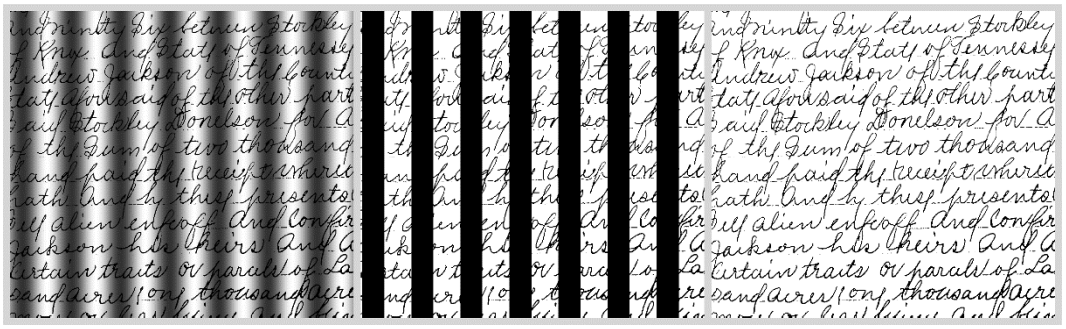
局部 OTSU 方法



移动平均算法



局部 OTSU 方法



移动平均算法

3 Implement linear interpolation algorithm to modify spatial resolution.

- **Python script code:**

  3_linear_interpolation.py

- **Input:**

  lena_std.tif

- **Output:**

  lena_std_zoom_0.5_by_linear_interpolation.png

  lena_std_zoom_2_by_linear_interpolation.png

- **Description:**

Python 源代码中给出了基于双线性插值的空间分辨率修改算法。为了提高处理效率，当放大倍数为整数时，可以设计一个使用 NumPy 矩阵化处理的快速方法计算双线性插值，以避免对图像进行大范围遍历产生的巨大开销。

适用于整数倍放大的双线性插值算法的实现重点是数据结构的设计。根据双线性插值的理论公式，插值公式可以写为以下矩阵形式：

$$I[r+x,c+y]=\begin{bmatrix}1-x & x\end{bmatrix}\begin{bmatrix}I[r,c] & I[r,c+1]\\ I[r+1,c] & I[r+1,c+1]\end{bmatrix}\begin{bmatrix}1-y\\ y\end{bmatrix}$$

推广至矩形区域则有：

$$I[r:r+1,c:c+1]=\begin{bmatrix}1 & 0\\ \vdots & \vdots\\ 1-x & x\\ \vdots & \vdots\\ 0 & 1\end{bmatrix}\begin{bmatrix}I[r,c] & I[r,c+1]\\ I[r+1,c] & I[r+1,c+1]\end{bmatrix}\begin{bmatrix}1 & \cdots & 1-y & \cdots & 0\\ 0 & \cdots & y & \cdots & 1\end{bmatrix}$$

由此可以计算原图像中的一个2×2的插值矩阵，在放大倍数较大时可节约大量时间。

而适用于任意倍放缩的双线性插值算法采用的是对新图像遍历填值的形式。使用简单的求商求余函数计算出新图像的像素对应于原图像的像素区域，便可以使用该像素区域进行双线性插值，由于每个新像素必定可以映射到一个旧像素区域，所以可以实现任意倍放缩的效果。但该方法的缺点是处理大图像时程序的运行时间较长，所以该方法更适用于对图像进行缩小。

另外需要注意的是，整数倍放大方法产生新图像的子区域大小完全相等，故考虑边界条件相对简单，但缺点是两个相邻区域的边界会被计算两次，从而产生额外的时间开销。任意倍放缩方法则不存在重复计算的问题，仅仅需要遍历新图像区域即可进行插值处理。

- **Source Code:**

```python
import os  # 文件操作
import time  # 时间操作

import numpy as np  # 数组操作
from PIL import Image  # 图像操作


class ImageData:

    # 初始化实例
    def __init__(self, image_name):
        self.image_name = image_name.split('.')[0]  # 图像的名称
        self.image_path = os.getcwd() + '\\' + image_name  # 图像的路径

    # 图像输入
    def input_image(self):
        image = Image.open(self.image_path)  # 输入图像
        self.image_matrix = np.array(image.convert('L'))  # 将图像转换成灰度矩阵
        self.image_size = self.image_matrix.size  # 图像的面积
        self.image_shape = self.image_matrix.shape  # 图像的形状
        self.image_height, self.image_width = self.image_shape  # 图像的高与宽

    # 整数倍像素矩阵化处理方法
    def integral_pixel_method(self, row, col):
        pixel = np.array(self.image_matrix[row:row + 2, col:col + 2])  # 原图对应2*2区域
        weight = np.array([np.arange(self.zoom, -1, -1), np.arange(self.zoom + 1)])  # 梯度比例矩阵
        return (np.linalg.multi_dot([weight.T, pixel, weight]) / (self.zoom)**2).astype(int)  # 双线性插值的矩阵形式

    # 整数倍放大双线性插值算法
    def integral_linear_interpolation(self, zoom):
        self.zoom = zoom  # 整数倍放大倍数
        self.zoom_height = (self.image_height - 1) * zoom + 1  # 整数倍放大后图像高度
        self.zoom_width = (self.image_width - 1) * zoom + 1  # 整数倍放大后图像宽度
        self.zoom_matrix = np.zeros((self.zoom_height, self.zoom_width), dtype=int)  # 新建图像矩阵
        for x in range(self.image_height - 1):  # 遍历原图像水平方向像素
            for y in range(self.image_width - 1):  # 遍历新图像垂直方向像素
                self.zoom_matrix[x * zoom:(x + 1) * zoom + 1, y * zoom:(y + 1) * zoom + 1] = self.integral_pixel_method(x, y)  # 对每个等容新正方形区域调用像素矩阵化方法

    # 任意倍像素处理方法
    def pixel_method(self, row, col):
        y, p_row = np.modf(row / self.zoom)  # y表示新正方形区域中的水平方向比例，p_row表示原图对应2*2区域的左上角横坐标
        x, p_col = np.modf(col / self.zoom)  # x表示新正方形区域中的垂直方向比例，p_col表示原图对应2*2区域的左上角纵坐标
        p_row, p_col = int(p_row), int(p_col)  # modf函数得到的整数部分为float类型，转换为int类型
        if p_col == self.image_width - 1:  # 右侧边界
            x, p_col = 1, p_col - 1  # 使用左方的新正方形区域计算
        if p_row == self.image_height - 1:  # 下侧边界
            y, p_row = 1, p_row - 1  # 使用上方的新正方形区域计算
        return int((1 - x) * (1 - y) * self.image_matrix[p_row, p_col] + x * (1 - y) * self.image_matrix[p_row, p_col + 1] +
                   (1 - x) * y * self.image_matrix[p_row + 1, p_col] + x * y * self.image_matrix[p_row + 1, p_col + 1])  # 双线性插值

    # 任意倍放缩双线性插值算法
    def linear_interpolation(self, zoom):
        self.zoom = zoom  # 放缩倍数
        self.zoom_height = int(self.image_height * zoom)  # 放缩后图像高度
        self.zoom_width = int(self.image_width * zoom)  # 放缩后图像宽度
        self.zoom_matrix = np.zeros((self.zoom_height, self.zoom_width), dtype=int)  # 新建图像矩阵
        for x in range(self.zoom_height):  # 遍历新图像水平方向像素
            for y in range(self.zoom_width):  # 遍历新图像垂直方向像素
                self.zoom_matrix[x, y] = self.pixel_method(x, y)  # 对每个像素调用像素方法

    # 图像输出
    def output_image(self):
        small_width = abs((self.zoom_width - self.image_width) // 2) + 16  # 较小图像的水平位置
        small_height = abs((self.zoom_height - self.image_height) // 2) + 16  # 较小图像的垂直位置
        if self.zoom > 1:
            image_splice = Image.new(mode='L', size=(self.zoom_width * 2 + 48, self.zoom_height + 32), color=215)  # 生成空白拼接图像
            image_splice.paste(im=Image.fromarray(self.image_matrix), box=(small_width, small_height))  # 生成原图灰度化图像
            image_splice.paste(im=Image.fromarray(self.zoom_matrix), box=(self.zoom_width + 32, 16))  # 生成双线性插值图像
        else:
            image_splice = Image.new(mode='L', size=(self.image_width * 2 + 48, self.image_height + 32), color=215)  # 生成空白拼接图像
            image_splice.paste(im=Image.fromarray(self.image_matrix), box=(16, 16))  # 生成原图灰度化图像
            image_splice.paste(im=Image.fromarray(self.zoom_matrix), box=(self.image_width + 16 + small_width, small_height))  # 生成双线性插值图像
        image_splice.save(self.image_name + '_zoom_' + str(self.zoom) + '_by_linear_interpolation.png')  # 保存图像
        # image_splice.show()  # 运行程序时显示图像

    # 放缩空间分辨率接口
    def modify_spatial_resolution(self, zoom=2):

        # 输入
        self.input_image()  # 输入图像

        # 双线性插值
        if isinstance(zoom, int):  # 整数倍放大
            self.integral_linear_interpolation(zoom)  # 调用整数倍放大双线性插值算法
        else:  # 任意倍放缩，数据量大时慢于整数倍放大
            self.linear_interpolation(zoom)  # 调用任意倍放缩双线性插值算法

        # 输出
        self.output_image()  # 输出图像


if __name__ == '__main__':

    print()  # 空行

    # 处理图像：lena_std.tif
    image_name = 'lena_std.tif'  # 图像的名称
    image_data = ImageData(image_name=image_name)  # 实例化ImageData对象
    print('Image: %s\n' % image_name)  # 实例的名称

    # 放大双线性插值算法
    start = time.time()  # 实例开始测试时刻
    image_data.modify_spatial_resolution(zoom=2)  # 调用函数放大空间分辨率
    end = time.time()  # 实例结束测试时刻
    print('Zoom: %d' % image_data.zoom)  # 放大倍数
    print('Program execute time: %.2f s\n' % (end - start))  # 实例测试时长

    # 缩小双线性插值算法
    start = time.time()  # 实例开始测试时刻
    image_data.modify_spatial_resolution(zoom=0.5)  # 调用函数缩小空间分辨率
    end = time.time()  # 实例结束测试时刻
    print('Zoom: %.1f' % image_data.zoom)  # 缩小倍数
    print('Program execute time: %.2f s\n' % (end - start))  # 实例测试时长
```

- **Result:**

使用双线性插值进行空间分辨率放缩的前后图像如下图所示，其中左图为变换前，右图为变换后，原图像为 lena_std.tif。放大 2 倍的运行时间为 3.99 秒，缩小至 0.5 倍的运行时间为 0.41 秒。

左图为原图，右图为放大 2 倍的图像

左图为原图，右图为缩小至 0.5 倍的图像