Data Visualization - Homework 1 - Report

邓淇升 大数据学院 16307110232

1.1 Implement n-dimensional joint histogram algorithm.

- **Python script:**

  1_1_n-dimensional_joint_histogram.py

- **Input:**

  height_vs_weight.csv

- **Output:**

  height_vs_weight_two-dimensional_joint_histogram.png

- **Description:**

  Python 源代码中给出了n维联合直方图的实现算法。假设每一维度的取值个数为n_i，则一共有$\prod_i n_i$个直方图区域，包含了所有可能出现的行向量情况。为了计算n维联合直方图，需要统计每一个区域在数据中出现的次数，并将其记录于对应的直方图区域内。算法实现中需要用到多维数组的数据结构，储存空间在数据维度增大时会有很大的需求。

  实验采用二维数据集，数据集来源为 UCLA 的 SOCR Data，记录了 1993 年香港地区两万五千名18岁青年的身高体重数据。下载数据集后先把数据单位换算成为厘米和千克，再将数据取为整数以便统计，清洗后的数据集可见于文件 height_vs_weight.csv。数据集的二维联合直方图以热力图形式展示。

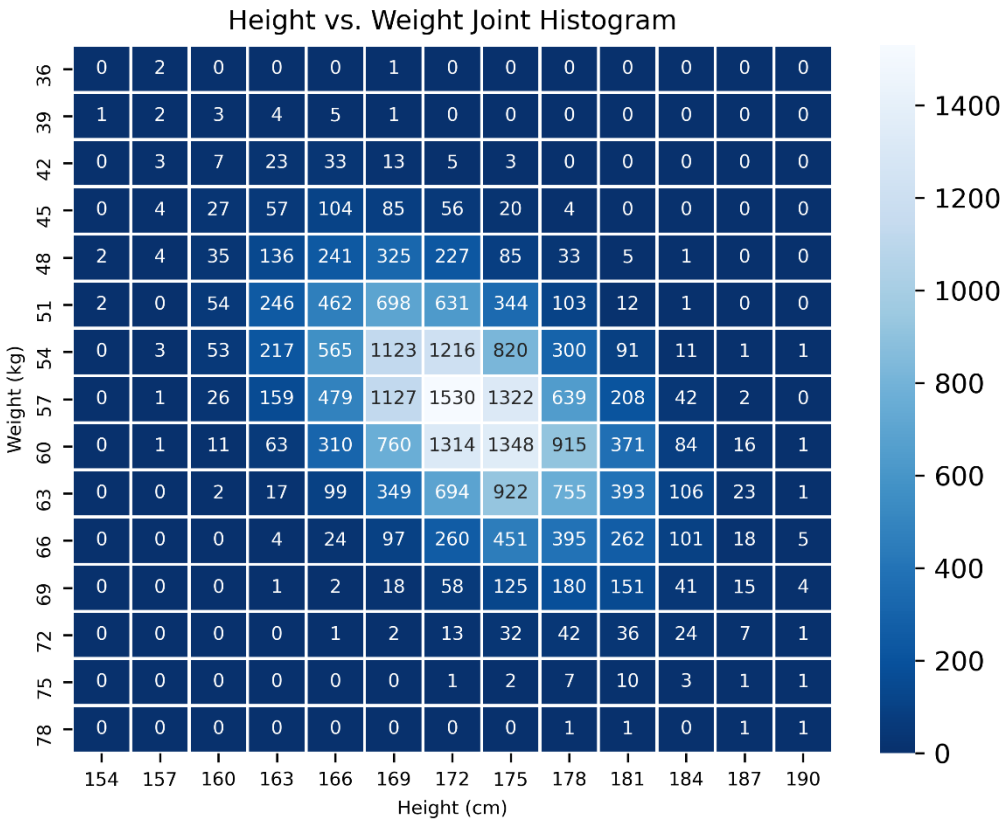
- **Source Code:**

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

import numpy as np    # 数组操作
import seaborn as sns    # 二维联合直方图（热力图）绘制
from matplotlib import pyplot as plt    # 绘图操作


class Data:

    def __init__(self, dataName):
        self.dataName = dataName.split('.')[0]    # 图像的名称
        self.dataPath = os.getcwd() + '\\' + dataName    # 数据集的文件路径
        self.dataSet = None    # 数据集的读入数据
        self.dataType = np.int    # 数据集的数据类型，测试集的数据类型为整型
        self.dimension = None    # 数据集的维度，即联合直方图的维度

    def inputData(self):    # 读入数据集
        self.dataSet = np.loadtxt(fname=self.dataPath, dtype=self.dataType, delimiter=',')    # 数据集的格式为csv，分隔符为逗号
        self.dimension = self.dataSet.shape[1]    # 数据形式为n维行向量，shape[1]表示列数，即数据的维度

    def evaluateHistogram(self):    # 计算直方图数据并可视化直方图

        # 计算直方图
        maxVal = np.amax(self.dataSet, axis=0)    # 数据各维度的最大值
        minVal = np.amin(self.dataSet, axis=0)    # 数据各维度的最小值
        binLen = 3 * np.ones(shape=self.dimension, dtype=np.int)    # 直方图组距，不能取过小数值，否则会导致数据聚集在边界上
        binNum = ((maxVal - minVal) / binLen + 1).astype(np.int)    # 直方图组数，向下取为整数
        histogram = np.zeros([var for var in binNum[::-1]], dtype=np.int)    # 储存直方图数据的n维数组，大小为各维度组数之积，初始值为0
        for val in self.dataSet:    # 遍历数据集的行向量
            binNo = []    # 该行向量在直方图中的n维坐标
            for index in range(self.dimension):    # 遍历行向量的各维度分量，以索引形式访问
                loc = int((val[index] - minVal[index]) / binLen[index])    # 计算变量的组号，区间左闭右开，组号从0到组数减1
                loc = max(loc, 0)    # 边界检查，组号不能小于0
                loc = min(loc, binNum[index] - 1)    # 边界检查，组号不能大于组数减1
                binNo.append(loc)    # 储存分量的组号，注意组号的顺序
            histogram[tuple(binNo[::-1])] += 1    # 在行向量的位置记录数据
```

```
        # 可视化直方图
        if self.dimension == 2:    # 二维联合直方图的可视化
            sns.heatmap(data=histogram, cmap=sns.color_palette('Blues_r', 240),    # 借助颜色可视化频数，人眼难以分辨240种颜色，可以视为连续分布
                        annot=True, annot_kws={'size': 7}, fmt='d', linewidths=1,    # 标注频数
                        xticklabels=list(np.arange(minVal[0] + binLen[0] // 2, minVal[0] + binLen[0] // 2 + binLen[0] * binNum[0], binLen[0]).astype(np.int)),    # 标注x轴组中值
                        yticklabels=list(np.arange(minVal[1] + binLen[1] // 2, minVal[1] + binLen[1] // 2 + binLen[1] * binNum[1], binLen[1]).astype(np.int)))    # 标注y轴组中值
            plt.title('Height vs. Weight Joint Histogram', fontsize=10)    # 标注标题
            plt.xlabel('Height (cm)', fontsize=7)    # 标注x轴标签
            plt.ylabel('Weight (kg)', fontsize=7)    # 标注y轴标签
            plt.xticks(fontsize=7)    # 设定x轴字号
            plt.yticks(fontsize=7)    # 设定y轴字号
            plt.savefig(self.dataName + '_two-dimensional_joint_histogram.png', dpi=480)    # 导出直方图为png文件，分辨率为480dpi
            # plt.show()    # 运行程序时显示直方图
        else:    # histogram保存了直方图数据，可进行下一步计算
            pass    # 更高维联合直方图的可视化待补充

    def nDJointHistogram(self):    # 计算n维联合直方图的程序接口
        self.inputData()    # 读入数据
        self.evaluateHistogram()    # 计算直方图


if __name__ == '__main__':

    start = time.time()    # 程序开始运行时刻

    dataName = 'height_vs_weight.csv'    # 数据集的名称
    data = Data(dataName=dataName)    # 实例化Data对象
    data.nDJointHistogram()    # 调用函数计算n维联合直方图

    end = time.time()    # 程序结束运行时刻

    print('Program execute time: %.2f s' % (end - start))    # 程序运行时长
```

- **Result:**

二维联合直方图如下图所示。程序的运行时间为 1.02 秒。



1.2 Implement computation of local histograms using efficient update method.

- **Python script code:**

  1_2_local_histogram.py

- **Input:**

  lena_std.tif

- **Output:**

  lena_std_histogram_by_local_histogram_algorithm.png

- **Description:**

Python 源代码中给出了使用高效局部更新方法计算图像直方图的实现算法。局部直方图算法的时间复杂度取决于图像的边长n和感兴趣区域的边长d，普通局部直方图方法需要在每一步遍历感兴趣区域中的所有值，复杂度为$O(d^2)$，而高效局部更新方法只需要访问感兴趣区域中的其中两列，复杂度降为$O(d)$，能有效提高更新效率。

高效局部更新算法主要用于局部直方图均衡，能有效降低图像整体对细节的影响，使得图像能够进行自适应均衡化。

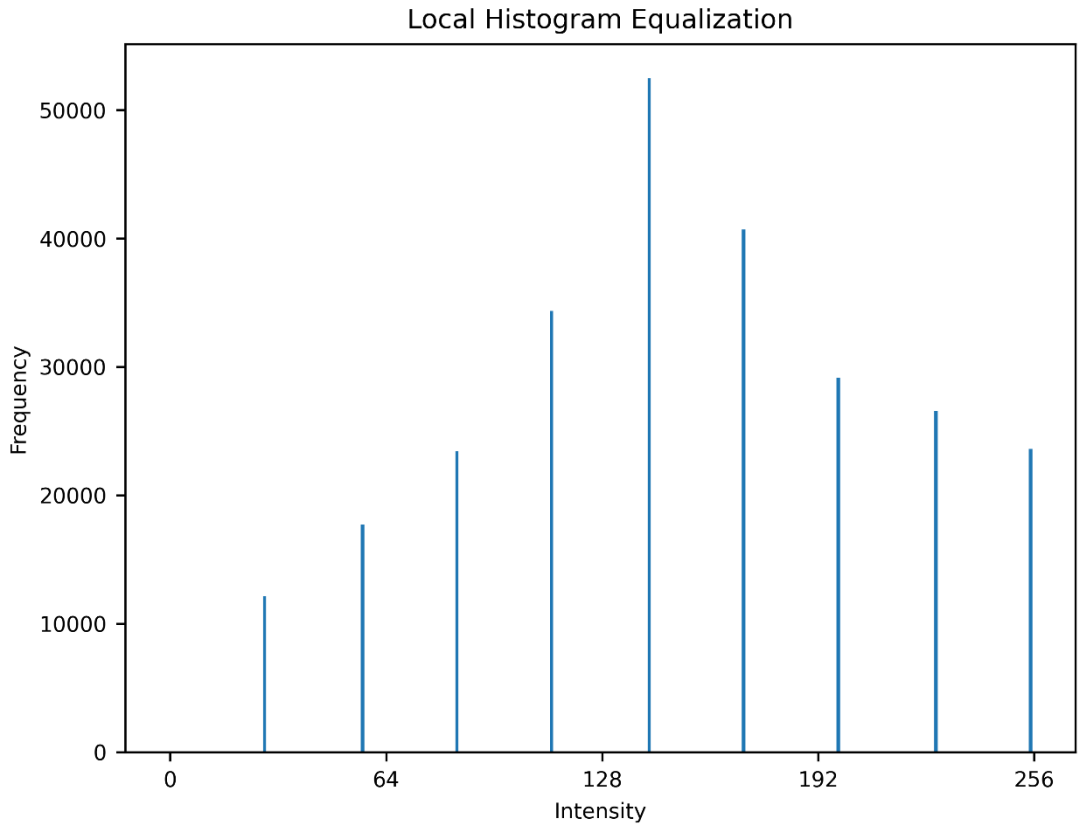
- **Source Code:**

```python
import copy    # 复制操作
import os    # 文件操作
import time    # 时间操作

import numpy as np    # 数组操作
from matplotlib import pyplot as plt    # 绘图操作
from PIL import Image    # 图像操作


class ImageData:

    def __init__(self, imageName):
        self.imageName = imageName.split('.')[0]    # 图像的名称
        self.imagePath = os.getcwd() + '\\' + imageName    # 图像的路径
        self.imageHeight = None    # 图像的高
        self.imageWidth = None    # 图像的宽
        self.imageMatrix = None    # 图像的灰度矩阵

    def inputImage(self):
        image = Image.open(self.imagePath)    # 输入图像
        self.imageMatrix = np.array(image.convert('L'))    # 将图像转换成灰度矩阵
        self.imageHeight, self.imageWidth = self.imageMatrix.shape    # 图像大小

    def evaluateHistogram(self):    # 计算局部直方图数据并可视化直方图

        # 计算局部直方图
        localMatrix = copy.deepcopy(self.imageMatrix)    # 创建灰度矩阵的深拷贝
        roiHeight, roiWidth = 3, 3    # 感兴趣区域ROI的高和宽，规定为3*3邻域，能忽略边界造成的影响
        halfRoiHeight, halfRoiWidth = roiHeight // 2, roiWidth // 2    # 感兴趣区域ROI的半高和半宽
        roiHistogram = np.histogram(self.imageMatrix[:roiHeight, :roiWidth].reshape(1, -1), range(257))[0]    # 初始感兴趣区域ROI的直方图
        for h in range(self.imageHeight - roiHeight + 1):    # 遍历高方向，h代表感兴趣区域ROI的左上角纵坐标
            for w in range(self.imageWidth - roiWidth + 1):    # 遍历宽方向，w代表感兴趣区域ROI的左上角横坐标
                currentPixel = (halfRoiHeight + h, halfRoiWidth + w)    # 当前邻域中心像素位置
                currentValue = self.imageMatrix[currentPixel]    # 当前邻域中心像素值
                localMatrix[currentPixel] = np.round(255 * np.sum(roiHistogram[:currentValue + 1]) / (roiHeight * roiWidth))    # 映射均衡化后的像素值
                if w == self.imageWidth - roiWidth:    # 感兴趣区域ROI横向移动至末端，最后一次调用时处于越界状态，但不会对像素值赋值
                    roiHistogram = np.histogram(self.imageMatrix[h + 1:h + roiHeight + 1, :roiWidth].reshape(1, -1), range(257))[0]    # 计算次行首个感兴趣区域ROI的直方图
                else:    # 感兴趣区域ROI向右移动一个像素
                    roiHistogram -= np.histogram(self.imageMatrix[h:h + roiHeight, w], range(257))[0]    # 删除最左边一行
                    roiHistogram += np.histogram(self.imageMatrix[h:h + roiHeight, w + roiWidth], range(257))[0]    # 添加最右边一行
        histogram = np.histogram(localMatrix.reshape(1, -1), range(257))[0]    # 计算局部均衡后的图像直方图

        # 可视化直方图
        plt.figure()    # 创建空白图像
        plt.bar(range(256), histogram, width=1)    # 根据直方图数据绘制直方图
        plt.title('Local Histogram Equalization', fontsize=10)    # 标注标题
        plt.xlabel('Intensity', fontsize=8)    # 标注x轴标签
        plt.ylabel('Frequency', fontsize=8)    # 标注y轴标签
        plt.xticks(range(0, 257, 64), fontsize=8)    # 设定x轴字号
        plt.yticks(fontsize=8)    # 设定y轴字号
        plt.savefig(self.imageName + '_histogram_by_local_histogram_algorithm.png', dpi=480)    # 导出直方图为png文件，分辨率为480dpi
        # plt.show()    # 运行程序时显示直方图
```

```
53
54      def localHistogram(self):
55          self.inputImage()    # 输入图像
56          self.evaluateHistogram()    # 计算局部直方图
57
58
59  if __name__ == '__main__':
60
61      start = time.time()    # 程序开始运行时刻
62
63      imageName = 'lena_std.tif'    # 图像的名称
64      imageData = ImageData(imageName=imageName)    # 实例化Image对象
65      imageData.localHistogram()    # 调用函数生成局部直方图
66
67      end = time.time()    # 程序结束运行时刻
68
69      print('Program execute time: %.2f s' % (end - start))    # 程序运行时长
```

- **Result:**

使用高效局部更新方法进行局部直方图均衡后的图像直方图如下图所示，原图像为 lena_std.tif。程序的运行时间为 48.87 秒。



2 Implement piecewise linear transformation function for image contrast stretching.

- **Python script code:**

  2_piecewise_linear_transformation.py

- **Input:**

  lena_std.tif

- **Output:**

  lena_std_by_image_contrast_stretching.png

- **Description:**

  Python 源代码中给出了利用分段线性函数变换进行图像对比度拉伸的实现算法。分段线性函数变换的主要目的是使图像的暗端变得更暗而亮端变得更亮，由于线性函数能够任意定义，相比于其他的函数变换更为自由，故能通过调整线性函数的参数来局部优化图像的灰度。

  实验采用三段式的分段线性函数进行处理，其中前后两段线段的斜率相等，两个拐点取在三等分的位置，具体参数在代码中给出。

- **Source Code:**

```python
import copy    # 复制操作
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:

    def __init__(self, imageName):
        self.imageName = imageName.split('.')[0]    # 图像的名称
        self.imagePath = os.getcwd() + '\\' + imageName    # 图像的路径
        self.imageHeight = None    # 图像的高
        self.imageWidth = None    # 图像的宽
        self.imageSplice = None    # 空白拼接图像
        self.imageMatrix = None    # 图像的灰度矩阵
        self.transMatrix = None    # 分段线性变换的灰度矩阵

    def inputImage(self):
        image = Image.open(self.imagePath)    # 输入图像
        self.imageMatrix = np.array(image.convert('L'))    # 将图像转换成灰度矩阵
        self.imageHeight, self.imageWidth = self.imageMatrix.shape    # 图像大小

    def piecewiseLinearTransformation(self, r1=85, r2=170, k=0.4):    # 应用分段线性变换处理图像
        assert 0 <= r1 <= r2 <= 255 and k >= 0, 'wrong parameters'    # 给定参数范围，为了完成对比度拉伸的效果，k需要小于1
        self.transMatrix = copy.deepcopy(self.imageMatrix)    # 创建灰度矩阵的深拷贝
        for h in range(self.imageHeight):    # 遍历高方向
            for w in range(self.imageWidth):    # 遍历宽方向
                val = self.imageMatrix[h][w]    # 某点的灰度值
                if val < r1:
                    val = val * k    # 将暗端调暗
                elif r1 <= val < r2:
                    val = val * k + 255 * (1 - k) * (val - r1) / (r2 - r1)    # 提高对比度
                else:
                    val = val * k + 255 * (1 - k)    # 将亮端调亮
                self.transMatrix[h][w] = val    # 处理后的灰度值

    def outputImage(self):
        self.imageSplice = Image.new(mode='L', size=(self.imageWidth * 2, self.imageHeight))    # 生成空白拼接图像
        self.imageSplice.paste(im=Image.fromarray(self.imageMatrix))    # 生成灰度化图像
        self.imageSplice.paste(im=Image.fromarray(self.transMatrix), box=(self.imageWidth, 0))    # 生成分段线性变换图像
        self.imageSplice.save(self.imageName + '_by_image_contrast_stretching.png')    # 保存图像
        # self.imageSplice.show()    # 运行程序时显示图像

    def imageContrastStretching(self):    # 图像对比度拉伸的程序接口
        self.inputImage()    # 输入图像
        self.piecewiseLinearTransformation()    # 分段线性变换对比度拉伸
        self.outputImage()    # 输出图像


if __name__ == '__main__':

    start = time.time()    # 程序开始运行时刻

    imageName = 'lena_std.tif'    # 图像的名称
    imageData = ImageData(imageName=imageName)    # 实例化Image对象
    imageData.imageContrastStretching()    # 调用函数处理对比度拉伸

    end = time.time()    # 程序结束运行时刻

    print('Program execute time: %.2f s' % (end - start))    # 程序运行时长
```

- **Result:**

使用分段线性函数变换进行图像对比度拉伸的图像前后效果如下图所示，其中左图为变换前，右图为变换后。程序的运行时间为 1.94 秒。

3.1 Implement histogram equalization algorithm.

- **Python script code:**

  3_1_global_histogram_equalization.py

- **Input:**

  lena_std.tif

- **Output:**

  lena_std_by_global_histogram_equalization.png

  lena_std_histogram_by_GHE.png

- **Description:**

Python 源代码中给出了图像全局直方图均衡的实现算法。通过计算图像灰度的累积分布函数，将灰度的分布转换成灰度累积分布的分布，而灰度累计分布的分布近似为均匀分布，可以通过全局均衡化提高图像整体的对比度。由于灰度值为整数，具体转换时需要对计算得到的累积分布值进行四舍五入。

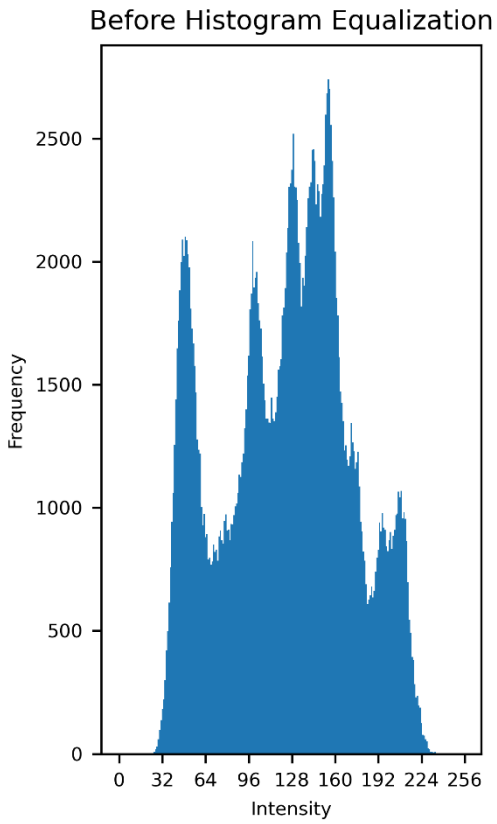
在全局直方图均衡过程中，图像的每部分都会受到来自图像整体分布的影响。

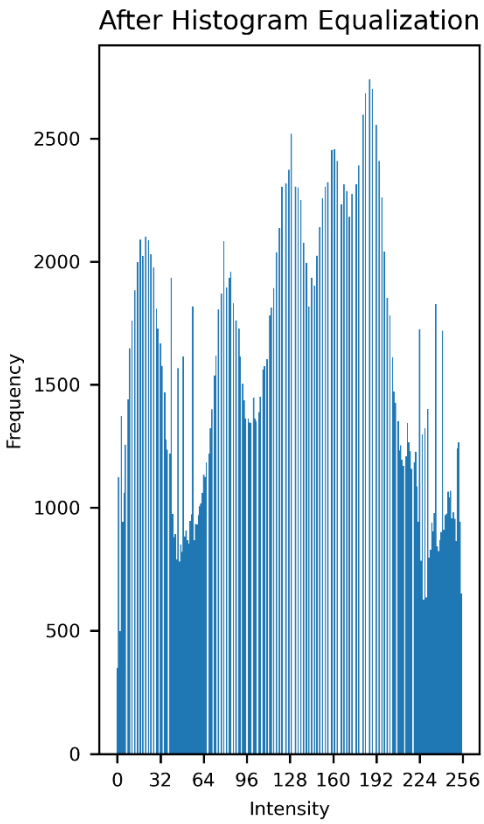
- **Source Code:**

```python
import copy    # 复制操作
import os    # 文件操作
import time    # 时间操作

import numpy as np    # 数组操作
from matplotlib import pyplot as plt    # 绘图操作
from PIL import Image    # 图像操作


class ImageData:

    def __init__(self, imageName):
        self.imageName = imageName.split('.')[0]    # 图像的名称
        self.imagePath = os.getcwd() + '\\' + imageName    # 图像的路径
        self.imageHeight = None    # 图像的高
        self.imageWidth = None    # 图像的宽
        self.imageSplice = None    # 空白拼接图像

        self.imageMatrix = None    # 图像的灰度矩阵
        self.globalMatrix = None    # 全局直方图均衡的灰度矩阵

        self.histogram = None    # 均衡前的图像直方图
        self.globalHistogram = None    # 全局直方图均衡后的图像直方图

    def inputImage(self):
        image = Image.open(self.imagePath)    # 输入图像
        self.imageMatrix = np.array(image.convert('L'))    # 将图像转换成灰度矩阵
        self.imageHeight, self.imageWidth = self.imageMatrix.shape    # 图像大小
        self.histogram = np.histogram(self.imageMatrix.reshape(1, -1), range(257))[0]    # 计算均衡前的图像直方图

    def histogramEqualization(self):    # 计算全局直方图均衡
        self.globalMatrix = copy.deepcopy(self.imageMatrix)    # 创建灰度矩阵的深拷贝
        mapping = {val: np.round(255 * np.sum(self.histogram[:val + 1]) / (self.imageHeight * self.imageWidth)) for val in range(256)}    # 全局直方图均衡
        for h in range(self.imageHeight):    # 遍历高方向
            for w in range(self.imageWidth):    # 遍历宽方向
                self.globalMatrix[h][w] = mapping[self.imageMatrix[h][w]]    # 映射均衡化后的像素值
        self.globalHistogram = np.histogram(self.globalMatrix.reshape(1, -1), range(257))[0]    # 计算均衡后的图像直方图

    def outputHistogram(self):    # 可视化直方图
        plt.figure()    # 创建空白图像
        plt.subplot(1, 2, 1)    # 左子图
        plt.bar(range(256), self.histogram, width=1)    # 根据均衡前的直方图数据绘制直方图
        plt.title('Before Histogram Equalization', fontsize=10)    # 标注标题
        plt.xlabel('Intensity', fontsize=7)    # 标注x轴标签
        plt.ylabel('Frequency', fontsize=7)    # 标注y轴标签
        plt.xticks(range(0, 257, 32), fontsize=7)    # 设定x轴字号
        plt.yticks(fontsize=7)    # 设定y轴字号
        plt.subplot(1, 2, 2)    # 右子图
        plt.bar(range(256), self.globalHistogram, width=1)    # 根据均衡后的直方图数据绘制直方图
        plt.title('After Histogram Equalization', fontsize=10)
        plt.xlabel('Intensity', fontsize=7)    # 标注x轴标签
        plt.ylabel('Frequency', fontsize=7)    # 标注y轴标签
        plt.xticks(range(0, 257, 32), fontsize=7)    # 设定x轴字号
        plt.yticks(fontsize=7)    # 设定y轴字号
        plt.subplots_adjust(wspace=0.5)    # 调整两子图间的距离
        plt.savefig(self.imageName + '_histogram_by_GHE.png', dpi=480)    # 导出两幅直方图为png文件，分辨率为480dpi
        # plt.show()    # 运行程序时显示直方图

    def outputImage(self):
        self.imageSplice = Image.new(mode='L', size=(self.imageWidth * 2, self.imageHeight))    # 生成空白拼接图像
        self.imageSplice.paste(im=Image.fromarray(self.imageMatrix))    # 生成灰度化图像
        self.imageSplice.paste(im=Image.fromarray(self.globalMatrix), box=(self.imageWidth, 0))    # 生成直方图均衡图像
        self.imageSplice.save(self.imageName + '_by_global_histogram_equalization.png')    # 保存图像
        # self.imageSplice.show()    # 运行程序时显示图像

    def histogramProcessing(self):    # 计算直方图均衡的程序接口
        self.inputImage()    # 输入图像
        self.histogramEqualization()    # 全局直方图均衡
        self.outputHistogram()    # 输出直方图
        self.outputImage()    # 输出图像


if __name__ == '__main__':

    start = time.time()    # 程序开始运行时刻

    imageName = 'lena_std.tif'    # 图像的名称
    imageData = ImageData(imageName=imageName)    # 实例化Image对象
    imageData.histogramProcessing()    # 调用函数计算直方图均衡

    end = time.time()    # 程序结束运行时刻

    print('Program execute time: %.2f s' % (end - start))    # 程序运行时长
```

- **Result:**

进行全局直方图均衡的图像及其直方图前后效果如下图所示，其中左图为均衡前，右图为均衡后。程序的运行时间为 1.27 秒。


Before Histogram Equalization
Frequency
2500
2000
1500
1000
500
0
0 32 64 96 128 160 192 224 256
Intensity


After Histogram Equalization
Frequency
2500
2000
1500
1000
500
0
0 32 64 96 128 160 192 224 256
Intensity

3.2 Implement local histogram equalization algorithm using efficient update method.

- **Python script code:**

  3_2_local_histogram_equalization.py

- **Input:**

  lena_std.tif

  square_noisy.tif

- **Output:**

  lena_std_by_local_histogram_equalization.png

  lena_std_histogram_by_LHE.png

  square_noisy_by_local_histogram_equalization.png

  square_noisy_histogram_by_LHE.png

- **Description:**

  Python 源代码中给出了使用高效局部更新方法进行图像局部直方图均衡的实现算法。局部直方图均衡能有效降低图像整体对细节的影响，使得图像能够进行自适应均衡化。

  实验采用两幅图像进行局部直方图均衡处理测试，其中对课本中的例图进行了结果重现。不同于全局直方图均衡，局部直方图均衡对图像细节的调整更为有效。

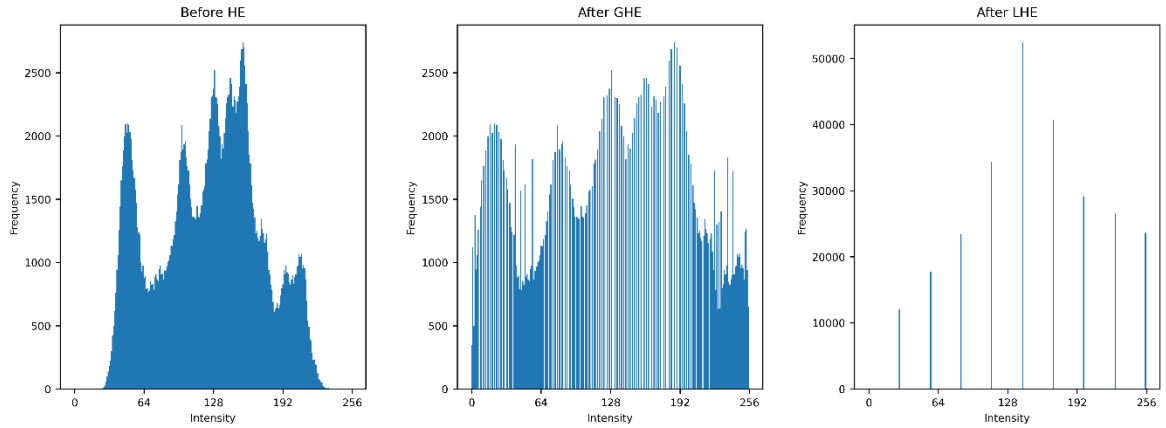
- **Source Code:**

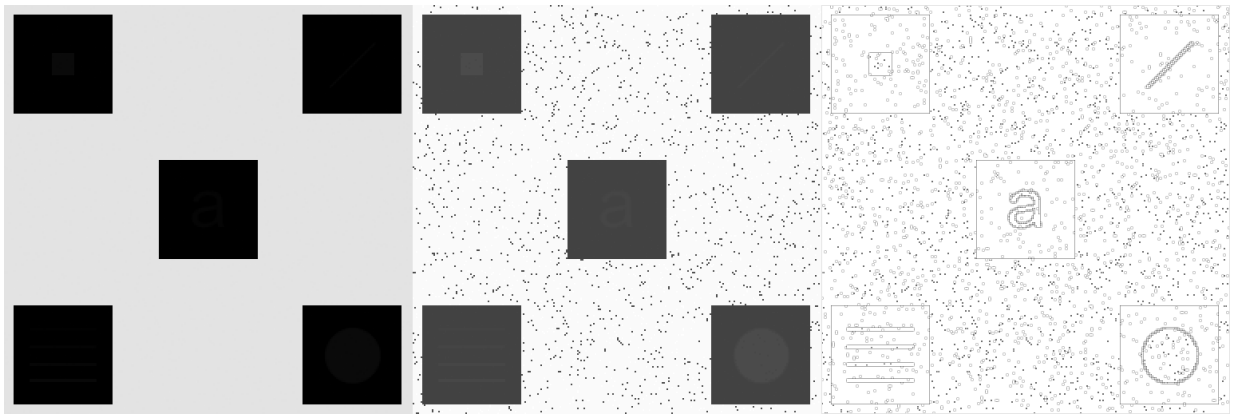
```python
import copy    # 复制操作
import os    # 文件操作
import time    # 时间操作

import numpy as np    # 数组操作
from matplotlib import pyplot as plt    # 绘图操作
from PIL import Image    # 图像操作


class ImageData:

    def __init__(self, imageName):
        self.imageName = imageName.split('.')[0]    # 图像的名称
        self.imagePath = os.getcwd() + '\\' + imageName    # 图像的路径
        self.imageHeight = None    # 图像的高
        self.imageWidth = None    # 图像的宽
        self.imageSplice = None    # 空白拼接图像

        self.imageMatrix = None    # 图像的灰度矩阵
        self.globalMatrix = None    # 全局直方图均衡的灰度矩阵
        self.localMatrix = None    # 局部直方图均衡的灰度矩阵

        self.histogram = None    # 均衡前的图像直方图
        self.globalHistogram = None    # 全局直方图均衡后的图像直方图
        self.localHistogram = None    # 局部直方图均衡后的图像直方图

    def inputImage(self):
        image = Image.open(self.imagePath)    # 输入图像
        self.imageMatrix = np.array(image.convert('L'))    # 将图像转换成灰度矩阵
        self.imageHeight, self.imageWidth = self.imageMatrix.shape    # 图像大小
        self.histogram = np.histogram(self.imageMatrix.reshape(1, -1), range(257))[0]    # 计算均衡前的图像直方图

    def globalHistogramEqualization(self):    # 计算全局直方图均衡
        self.globalMatrix = copy.deepcopy(self.imageMatrix)    # 创建灰度矩阵的深拷贝
        mapping = {val: np.round(255 * np.sum(self.histogram[:val + 1]) / (self.imageHeight * self.imageWidth)) for val in range(256)}    # 全局直方图均衡
        for h in range(self.imageHeight):    # 遍历高方向
            for w in range(self.imageWidth):    # 遍历宽方向
                self.globalMatrix[h][w] = mapping[self.imageMatrix[h][w]]    # 映射均衡化后的像素值
        self.globalHistogram = np.histogram(self.globalMatrix.reshape(1, -1), range(257))[0]    # 计算全局均衡后的图像直方图

```

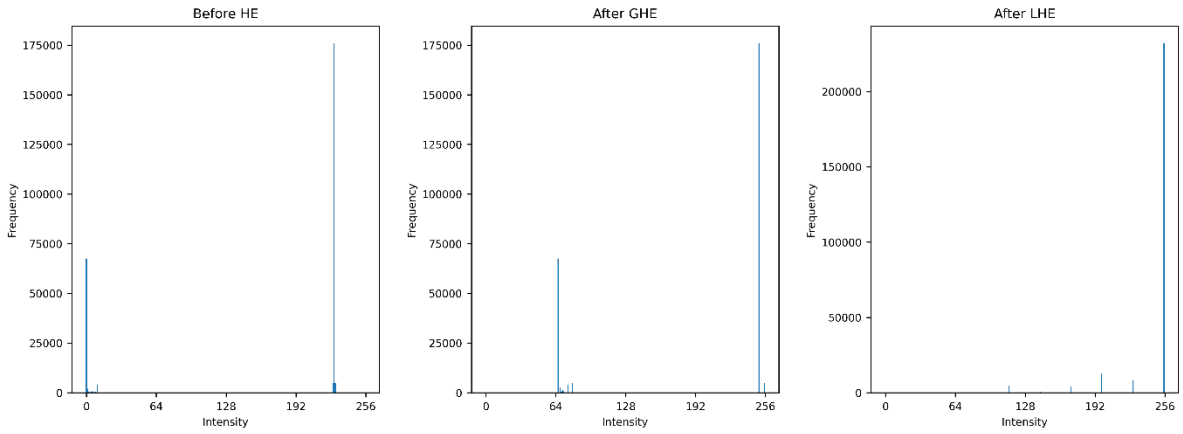
```
    def localHistogramEqualization(self):    # 计算局部直方图均衡
        self.localMatrix = copy.deepcopy(self.imageMatrix)    # 创建灰度矩阵的深拷贝
        roiHeight, roiWidth = 3, 3    # 感兴趣区域ROI的高和宽，规定为3*3邻域，能忽略边界造成的影响
        halfRoiHeight, halfRoiWidth = roiHeight // 2, roiWidth // 2    # 感兴趣区域ROI的半高和半宽
        roiHistogram = np.histogram(self.imageMatrix[:roiHeight, :roiWidth].reshape(1, -1), range(257))[0]    # 初始感兴趣区域ROI的直方图
        for h in range(self.imageHeight - roiHeight + 1):    # 遍历高方向，h代表感兴趣区域ROI的左上角纵坐标
            for w in range(self.imageWidth - roiWidth + 1):    # 遍历宽方向，w代表感兴趣区域ROI的左上角横坐标
                currentPixel = (halfRoiHeight + h, halfRoiWidth + w)    # 当前邻域中心像素位置
                currentValue = self.imageMatrix[currentPixel]    # 当前邻域中心像素值
                self.localMatrix[currentPixel] = np.round(255 * np.sum(roiHistogram[:currentValue + 1]) / (roiHeight * roiWidth))    # 映射均衡化后的像素值
                if w == self.imageWidth - roiWidth:    # 感兴趣区域ROI横向移动至末端，最后一次调用时处于越界状态，但不会对像素值赋值
                    roiHistogram = np.histogram(self.imageMatrix[h + 1:h + roiHeight + 1, :roiWidth].reshape(1, -1), range(257))[0]    # 计算次行首个感兴趣区域ROI的直方图
                else:    # 感兴趣区域ROI向右移动一个像素
                    roiHistogram -= np.histogram(self.imageMatrix[h:h + roiHeight, w], range(257))[0]    # 删除最左边一行
                    roiHistogram += np.histogram(self.imageMatrix[h:h + roiHeight, w + roiWidth], range(257))[0]    # 添加最右边一行
        self.localHistogram = np.histogram(self.localMatrix.reshape(1, -1), range(257))[0]    # 计算局部均衡后的图像直方图

    def outputHistogram(self):    # 可视化直方图
        plt.figure(figsize=(15, 5))    # 创建空白图像
        plt.subplot(1, 3, 1)    # 左子图
        plt.bar(range(256), self.histogram, width=1)    # 根据均衡前的直方图数据绘制直方图
        plt.title('Before HE', fontsize=10)    # 标注标题
        plt.xlabel('Intensity', fontsize=8)    # 标注x轴标签
        plt.ylabel('Frequency', fontsize=8)    # 标注y轴标签
        plt.xticks(range(0, 257, 64), fontsize=8)    # 设定x轴字号
        plt.yticks(fontsize=8)    # 设定y轴字号
        plt.subplot(1, 3, 2)    # 中子图
        plt.bar(range(256), self.globalHistogram, width=1)    # 根据全局均衡后的直方图数据绘制直方图
        plt.title('After GHE', fontsize=10)
        plt.xlabel('Intensity', fontsize=8)    # 标注x轴标签
        plt.ylabel('Frequency', fontsize=8)    # 标注y轴标签
        plt.xticks(range(0, 257, 64), fontsize=8)    # 设定x轴字号
        plt.yticks(fontsize=8)    # 设定y轴字号
        plt.subplots_adjust(wspace=0.5)    # 调整两子图间的距离
        plt.subplot(1, 3, 3)    # 右子图
        plt.bar(range(256), self.localHistogram, width=1)    # 根据全局均衡后的直方图数据绘制直方图
        plt.title('After LHE', fontsize=10)
        plt.xlabel('Intensity', fontsize=8)    # 标注x轴标签
        plt.ylabel('Frequency', fontsize=8)    # 标注y轴标签
        plt.xticks(range(0, 257, 64), fontsize=8)    # 设定x轴字号
        plt.yticks(fontsize=8)    # 设定y轴字号
        plt.subplots_adjust(wspace=0.3)    # 调整两子图间的距离
        plt.savefig(self.imageName + '_histogram_by_LHE.png', dpi=480)    # 导出两幅直方图为png文件，分辨率为480dpi
        # plt.show()    # 运行程序时显示直方图

    def outputImage(self):
        self.imageSplice = Image.new(mode='L', size=(self.imageWidth * 3, self.imageHeight))    # 生成空白拼接图像
        self.imageSplice.paste(im=Image.fromarray(self.imageMatrix))    # 生成灰度化图像
        self.imageSplice.paste(im=Image.fromarray(self.globalMatrix), box=(self.imageWidth, 0))    # 生成全局直方图均衡图像
        self.imageSplice.paste(im=Image.fromarray(self.localMatrix), box=(self.imageWidth * 2, 0))    # 生成局部直方图均衡图像
        self.imageSplice.save(self.imageName + '_by_local_histogram_equalization.png')    # 保存图像
        # self.imageSplice.show()    # 运行程序时显示图像

    def histogramProcessing(self):    # 计算直方图均衡的程序接口
        self.inputImage()    # 输入图像
        self.globalHistogramEqualization()    # 全局直方图均衡
        self.localHistogramEqualization()    # 局部直方图均衡
        self.outputHistogram()    # 输出直方图
        self.outputImage()    # 输出图像


if __name__ == '__main__':

    start = time.time()    # 程序开始运行时刻

    # 图像一
    imageName1 = 'square_noisy.tif'    # 图像的名称
    imageData1 = ImageData(imageName=imageName1)    # 实例化Image对象
    imageData1.histogramProcessing()    # 调用函数计算直方图均衡

    # 图像二
    imageName2 = 'lena_std.tif'    # 图像的名称
    imageData2 = ImageData(imageName=imageName2)    # 实例化Image对象
    imageData2.histogramProcessing()    # 调用函数计算直方图均衡

    end = time.time()    # 程序结束运行时刻

    print('Program execute time: %.2f s' % (end - start))    # 程序运行时长
```

- **Result:**

使用高效局部更新方法进行局部直方图均衡的图像及其直方图前后效果如下图所示，其中左图为均衡前，中图为全局均衡后，右图为局部均衡后。程序的运行时间为 99.88 秒。


Before HE
After GHE
After LHE
Frequency
Intensity




Before HE
After GHE
After LHE
Frequency
Intensity