浙江大学实验报告

专业：计算机科学与技术
姓名：NoughtQ
学号：1145141919810
日期：2024 年 11 月 5 日

课程名称：图像信息处理　　指导老师：宋明黎　　成绩：_____

实验名称：图像对数增强与直方图均衡化

一、 实验目的和要求

1. 通过对图像的对数化操作来增强可视性
2. 实现图像的直方图均衡化

二、 实验内容和原理

2.1 对数化操作

为了增强图像的可视性，我们需要对图像中的像素进行基于对数的操作：

$$L_d = \frac{\log(L_w + 1)}{\log(L_{\max} + 1)}$$

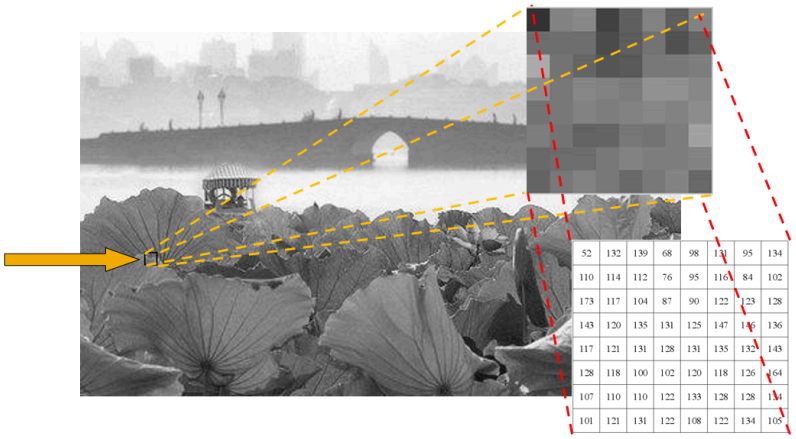
其中L_d表示显示亮度，L_w表示真实世界的亮度，$L_{\max}$表示场景中的最亮值。这个映射能够确保不管场景的动态范围是怎么样的，其最大值都能映射到 1（白），其他的值也能够比较平滑地变化。

2.2 图像直方图

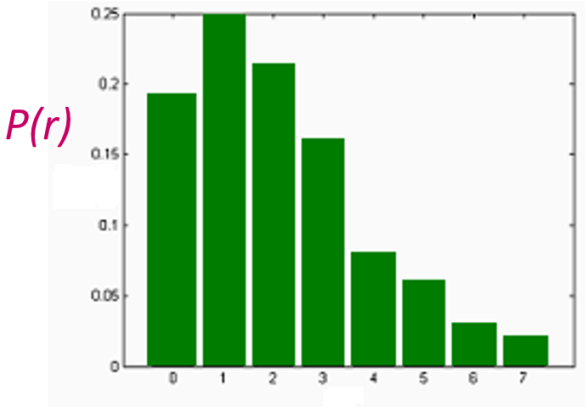
2.2.1 灰度直方图

先来了解灰度图的基本概念：

- 灰度图是一个由像素构成的，大小为$M \times N$的二维数组
- 每个像素用 8 位表示，灰度被分为$2^8 = 256$个等级，灰度强度$p \in [0, 255]$
- 灰度强度更小，像素越黑，反之亦然



灰度直方图(grayscale histogram): 一类统计图形, 它表示一幅图像中各个灰度等级的像素个数在像素总数中所占的比重。



- 纵坐标表示各个灰度等级的像素个数在像素总数中所占的比重
- 横坐标表示灰度等级

设灰度等级范围为$[0, L-1]$，灰度直方图用下列离散函数来表示:

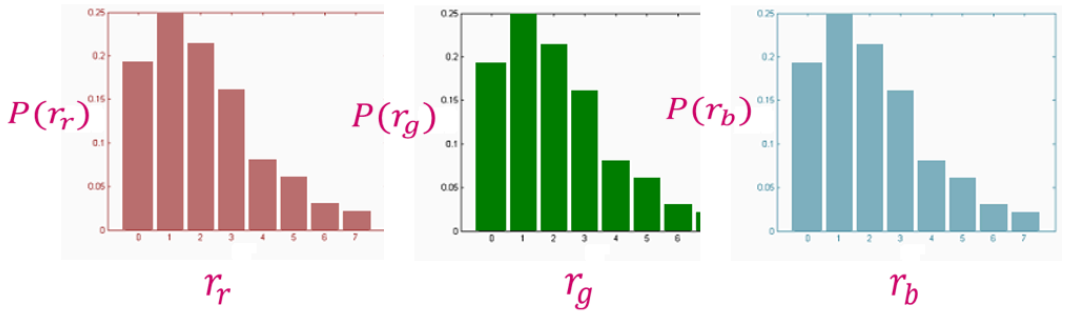
$$h(r_k) = n_k$$

其中, r_k为第k级灰度, n_k是图像中具有灰度级r_k的像素数目, n为图像总像素数, $0 \leq k \leq L-1, 0 \leq n_k \leq n-1$。

通常用概率密度函数来归一化直方图: $P(r_k) = \frac{n_k}{n}$ $P(r_k)$为灰度级r_k所发生的概率(概率密度函数)，此时满足下列条件: $\sum_{k=0}^{L-1} P(r_k) = 1$

2.2.2 彩色直方图

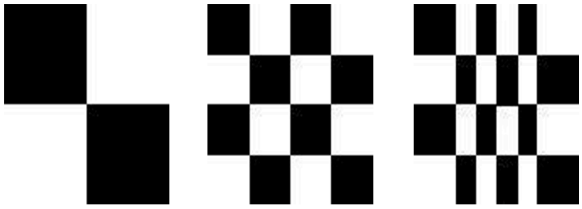
彩色直方图(color histogram): 一类统计图形，它表示一幅图像中 r, g, b 通道上各个灰度等级的像素个数在像素总数中所占的比重。



2.2.3 直方图特征

直方图的优点:

- 是空间域处理技术的基础
- 反映图像灰度的分布规律，但不能体现图像中的细节变化情况
- 对于一幅给定的图像，其直方图是唯一的
- 不同的图像可以对应相同的直方图



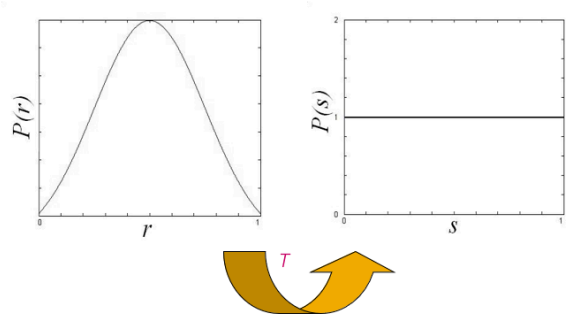
对直方图进行操作能有效地用于图像增强、压缩和分割, 是图像处理的一个实用手段。

直方图的缺点:

- 带来噪声
- 丢失结构信息：只知颜色分布，不知结构

2.3 直方图均衡化

直方图均衡化(historam equalization)：将原图像的非均匀分布的直方图通过变换函数T修正为均匀分布的直方图, 然后按均衡直方图修正原图像。图像均衡化处理后, 图像的直方图是平直的，即各灰度级具有相同的出现频数。



直方图均衡化的关键在于找到变换函数T。在计算前，我们做出以下假设和规定:

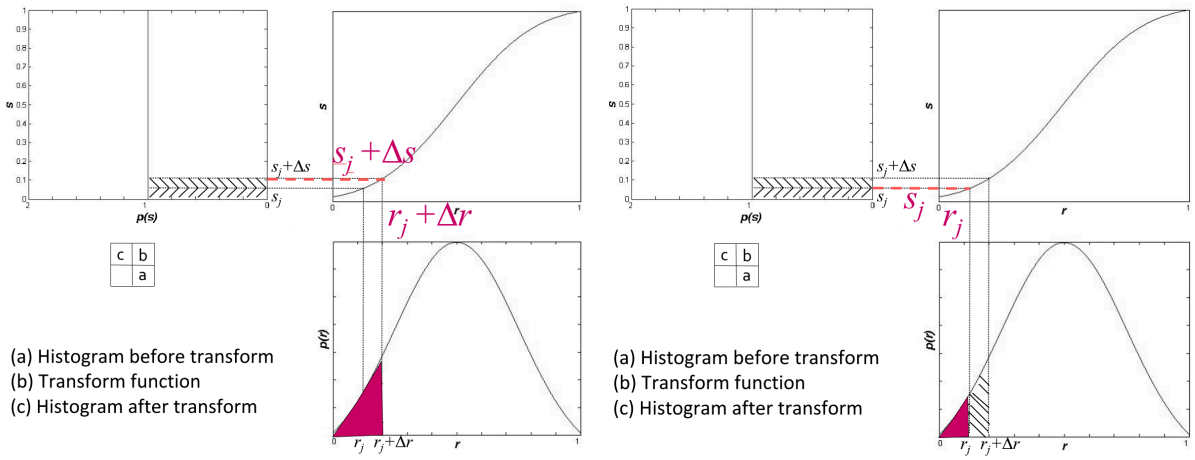
- 假设：
  - 令r和s分别代表变化前后图像的灰度级，并且$0 \leq r, s \leq 1$
  - $P(r)$和$P(s)$分别为变化前后各级灰度的概率密度函数（r和s值已归一化，最大灰度值为 1）
- 规定：
  - $T(r)$是单调递增函数，并且$0 \leq r \leq 1$，$0 \leq T(r) \leq 1$
  - 反变换$r = T^{-1}(s)$也为单调递增函数

连续版本的直方图均衡化

考虑到灰度变换不影响像素的位置分布，也不会增减像素数目。所以有：

$$s = T(r) = \int_0^r P(r)dr$$

几何意义：转换函数T在变量r处的函数值s，是原直方图中灰度等级为$[0, r]$以内的直方图曲线所覆盖的面积。



其中右下为原来的直方图，右上为变换函数，左上为均衡化后的直方图。

离散版本的直方图均衡化

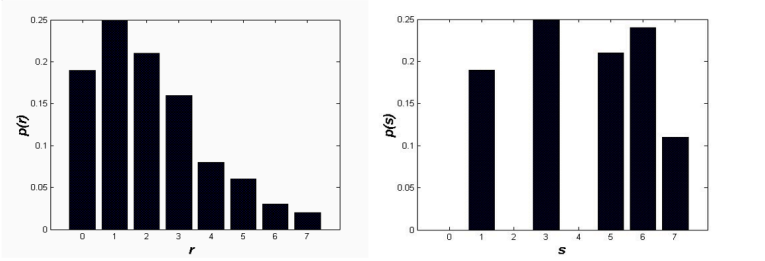
设一幅图像的像素总数为n，分L个灰度级，n_k为第k个灰度级出现的像素数，则第k个灰度级出现的概率为：

$$P(r_k) = \frac{n_k}{n}(0 \leq r_k \leq 1, k = 0, 1, 2, ..., L-1)$$

离散灰度直方图均衡化的转换公式为：

$$s_k = T(r_k) = \frac{1}{n}\sum_{i=0}^{k} n_i$$

最终效果如下：



(a)Before histogram equalization (b) After histogram equalization

总结

直方图均衡化实质上是减少图像的灰度级以换取对比度的加大。在均衡过程中，原来的直方图上出现概率较小的灰度级被归入很少几个甚至一个灰度级中，故得不到增强。若这些灰度级所构成的图象细节比较重要，则需采用局部区域直方图均衡化处理。

三、 实验步骤与分析

3.1 对数化操作

具体实现类似Lab1完成的改变亮度的函数`ChangeLuminance()`，唯一的区别在于改变亮度的方法从原来简单地乘上一个倍率，改为现在的对数化方法（公式见上面的“实验内容和原理”）。

```
// 对数化操作
BMPFILE LogarithmOp(BMPFILE bf) {
    BMPFILE lImg;
    RGB rgb;
    YUV yuv;
    BYTE val;
    int x, y;
    int pos;
    double maxY = -1;

    // 分配空间
    rgb = (RGB)malloc(sizeof(struct rgb));
    yuv = (YUV)malloc(sizeof(struct yuv));
    lImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));

    // 将旧图的数据拷贝到新图上
    memcpy(&(lImg->bmfh), &(bf->bmfh), sizeof(BITMAPFILEHEADER));
    memcpy(&(lImg->bmih), &(bf->bmih), sizeof(BITMAPINFOHEADER));
```

```
        lImg->aBitmapBits = (BYTE *)malloc(sizeof(BYTE) * lImg->bmih.biSizeImage);

    // 找到最大的亮度
    for (y = 0; y < bf->bmih.biHeight; y++)
        for (x = 0; x < bf->bmih.biWidth; x++) {
            pos = y * ColorBitWidth + x * 3;
            rgb = (RGB)malloc(sizeof(struct rgb));
            yuv = (YUV)malloc(sizeof(struct yuv));
            rgb->B = bf->aBitmapBits[pos];
            rgb->G = bf->aBitmapBits[pos + 1];
            rgb->R = bf->aBitmapBits[pos + 2];
            yuv = rgb2yuv(rgb);
            // 防止越界
            yuv->Y = rearrangeComp(yuv->Y);
            maxY = yuv->Y > maxY ? yuv->Y : maxY;
        }

    // RGB -> YUV, 通过改变 Y 来改变亮度, 然后再变回 RGB, 注意越界处理
    for (y = 0; y < bf->bmih.biHeight; y++)
        for (x = 0; x < bf->bmih.biWidth; x++) {
            pos = y * ColorBitWidth + x * 3;
            rgb = (RGB)malloc(sizeof(struct rgb));
            yuv = (YUV)malloc(sizeof(struct yuv));
            rgb->B = bf->aBitmapBits[pos];
            rgb->G = bf->aBitmapBits[pos + 1];
            rgb->R = bf->aBitmapBits[pos + 2];
            yuv = rgb2yuv(rgb);
            // 对数化操作
            yuv->Y = log(yuv->Y + 1) / log(maxY + 1) * 255;
            // 防止越界
            yuv->Y = rearrangeComp(yuv->Y);
            rgb = yuv2rgb(yuv);
            lImg->aBitmapBits[pos] = rearrangeComp(rgb->B);
            lImg->aBitmapBits[pos + 1] = rearrangeComp(rgb->G);
            lImg->aBitmapBits[pos + 2] = rearrangeComp(rgb->R);
        }

    return lImg;
}
```

3.2 直方图均衡化

我在本实验设计了三种直方图均衡化的实现方式，分别为：

- 仅保留亮度，直方图均衡化为灰度图
- 单独处理 RGB 的每个通道，直方图均衡化为彩色图
- 亮度修改后转为 RGB，即共同处理 RGB 通道，直方图均衡化为彩色图

为了代码编写上的方便，先处理第 1 和第 3 类实现方式

```
BMPFILE HstEqualization(BMPFILE bf, int * choice) {
    // 去掉了繁琐的初始化步骤

    // 向用户询问进行何种类型的直方图均衡化
    *choice = AskforChoicev3();

    // 仅改变亮度（灰度）通道
    if (*choice != 2) {
        // 初始化
        for (i = 0; i < GRAYLEVELNUM; i++) {
            grayLevel[i] = 0;
        }

        // 统计每个灰度级下的像素个数（分为 256 级）
        for (y = 0; y < bf->bmih.biHeight; y++)
            for (x = 0; x < bf->bmih.biWidth; x++) {
                pos = y * ColorBitWidth + x * 3;
                rgb = (RGB)malloc(sizeof(struct rgb));
                yuv = (YUV)malloc(sizeof(struct yuv));
                rgb->B = bf->aBitmapBits[pos];
                rgb->G = bf->aBitmapBits[pos + 1];
                rgb->R = bf->aBitmapBits[pos + 2];
                yuv = rgb2yuv(rgb);
                grayLevel[rearrangeComp(yuv->Y)]++;
            }

        // 计算直方图数据
        grayLevel[0] /= pixelNum;
        for (i = 1; i < GRAYLEVELNUM; i++) {
            grayLevel[i] = grayLevel[i - 1]
                           + grayLevel[i] / pixelNum;
        }

        // 直方图均衡化
```

```
        for (y = 0; y < bf->bmih.biHeight; y++)
            for (x = 0; x < bf->bmih.biWidth; x++) {
                pos = y * ColorBitWidth + x * 3;
                rgb = (RGB)malloc(sizeof(struct rgb));
                yuv = (YUV)malloc(sizeof(struct yuv));
                rgb->B = bf->aBitmapBits[pos];
                rgb->G = bf->aBitmapBits[pos + 1];
                rgb->R = bf->aBitmapBits[pos + 2];
                yuv = rgb2yuv(rgb);
                // 根据直方图数据修改亮度（灰度）
                yuv->Y = grayLevel[rearrangeComp(yuv->Y)]
                                                * (GRAYLEVELNUM - 1);

                // 彩色图处理
                if (*choice == 3) {
                    rgb = yuv2rgb(yuv);
                    colorImg->aBitmapBits[pos] =
                                                rearrangeComp(rgb->B);
                    colorImg->aBitmapBits[pos + 1] =
                                                rearrangeComp(rgb->G);
                    colorImg->aBitmapBits[pos + 2] =
                                                rearrangeComp(rgb->R);
                } else {    // 灰度图处理
                    grayImg->aBitmapBits[y * GrayBitWidth + x] =
                                                rearrangeComp(yuv->Y);
                }
            }

        if (*choice == 1)
            return grayImg;
        else
            return colorImg;
    } else {
      // 放在后面
    }
}
```

概括一下上述代码的实现步骤：

1. 询问进行何种类型的直方图均衡化
2. 统计每个灰度级下的像素个数以及总像素数，然后通过计算得到直方图数据
3. 根据直方图数据改变图像像素的亮度
4. 如果是灰度图，仅保留亮度，否则需要转化为 RGB

接下来再完成第 2 种实现方法, 它与前面的最大差别在于要同时得到 RGB 的直方图数据并对每个通道单独处理，但本质上是差不多的。

```
// 上接前面的 if-else 语句
else {       // RGB 三个通道分别改变
    // 初始化
    for (i = 0; i < GRAYLEVELNUM; i++) {
        RLevel[i] = GLevel[i] = BLevel[i] = 0;
    }

    // 统计每个通道的灰度级下的像素个数（分为 256 级）
    for (y = 0; y < bf->bmih.biHeight; y++)
        for (x = 0; x < bf->bmih.biWidth; x++) {
            pos = y * ColorBitWidth + x * 3;
            BLevel[bf->aBitmapBits[pos]]++;
            GLevel[bf->aBitmapBits[pos + 1]]++;
            RLevel[bf->aBitmapBits[pos + 2]]++;
        }

    // 计算直方图数据
    RLevel[0] /= pixelNum;
    GLevel[0] /= pixelNum;
    BLevel[0] /= pixelNum;
    for (i = 1; i < GRAYLEVELNUM; i++) {
        RLevel[i] = RLevel[i - 1] + RLevel[i] / pixelNum;
        GLevel[i] = GLevel[i - 1] + GLevel[i] / pixelNum;
        BLevel[i] = BLevel[i - 1] + BLevel[i] / pixelNum;
    }

    // 直方图均衡化
    for (y = 0; y < bf->bmih.biHeight; y++)
        for (x = 0; x < bf->bmih.biWidth; x++) {
            pos = y * ColorBitWidth + x * 3;
            colorImg->aBitmapBits[pos] =
                  rearrangeComp(BLevel[bf->aBitmapBits[pos]]
                                * (GRAYLEVELNUM - 1));
            colorImg->aBitmapBits[pos + 1] =
                  rearrangeComp(GLevel[bf->aBitmapBits[pos + 1]]
                                * (GRAYLEVELNUM - 1));
            colorImg->aBitmapBits[pos + 2] =
                  rearrangeComp(RLevel[bf->aBitmapBits[pos + 2]]
                                * (GRAYLEVELNUM - 1));
        }
```

```
    return colorImg;
}
```

四、 实验环境及运行方法

4.1 实验环境

我的开发环境如下:

- Windows 11 24H2
- gcc version 14.2.0
- GNU Make 4.4.1

4.2 运行方法

若要运行主程序，需要遵循以下步骤:

> **注意**
>
> 在执行以下命令前，请检查一下路径下 `./代码/test` 是否存在`.bmp` 图片，并且检查一下`./代码/scripts/bmp.h` 文件的宏定义 `BMPFILEPATH` 是否指的是该图片。若有问题请自行修改，否则程序无法正常运行。

1. 将目录切换至 `./代码/build`
2. 执行以下命令:

```
# 编译代码
$ make

# 运行可执行文件
$ ./lab3
Successfully open the file!
Size: 1548022(bit)
ColorBitWidth: 2144
Width: 714
Height: 722
Image Size: 1547968
===== Logarithmic Operation =====
Finish the conversion successfully!
===== Histogram Equalization =====
Which type of the histogram equalization do you like?
1) Grayscale style
```

```
2) Color image with RGB changed separately
3) Color image with RGB changed overall
Other numbers can cancel the operation.
Please input your choice(1, 2, 3):
```

当终端显示上述内容时，表明程序已经完成了对数化操作，并且现在程序询问使用何种方式进行直方图均衡化。可以选择的方式有：生成灰度图（输入 1），分别处理 RGB 三个通道后得到彩色图（输入 2），以及同时处理 RGB（即改变亮度）得到彩色图（输入 3）。下面以输入 3 为例：

```
Please input your choice(1, 2, 3): 3

Valid choice!
Please wait a minute for the generation of the new image...
Finish the conversion successfully!
```

如果终端显示上述内容，表明程序成功地生成了直方图均衡化后的图像。

3. 来到 `./代码/tests` 目录，此时可以看到得到的新图（对数化操作后的图像、直方图均衡化后的图像）。

五、 实验结果展示

> **注意**
>
> 本报告采用 typst 书写，但是 typst 不支持插入 BMP 图像文件，因此这里展示的是它们的 PNG 形式，对应的 BMP 形式见目录 `./代码/tests`。

5.1 对数化操作

下面展示几组对数化操作前后的对比图像：

Figure 7: 对数化操作前

Figure 8: 对数化操作后

Figure 9: 对数化操作前

Figure 10: 对数化操作后

Figure 11: 对数化操作前

Figure 12: 对数化操作后

可以看到，对于一张可视度在接受范围内的图像，经过对数化操作后，图像就会变得更加朦胧，仿佛覆盖了一层雾气似的；但是对于原来可视度较差（亮度太暗）的图像，对数化操作后能够使图像恢复到较为正常的亮度，当然也有可能会带来一层“雾气”。

5.2 直方图均衡化

下面展示几组直方图均衡化操作前后的对比图像：

Figure 13: 原图像

Figure 14: 对数化操作

Figure 15: 直方图均衡化（灰度图）

Figure 16: 直方图均衡化（彩色图，单独设置 RGB 通道）

Figure 17: 直方图均衡化（彩色图，共同设置 RGB 通道）

Figure 18: 原图像

Figure 19: 直方图均衡化（彩色图，单独设置 RGB 通道）

Figure 20: 直方图均衡化（彩色图，共同设置 RGB 通道）

Figure 21: 原图像

Figure 22: 直方图均衡化（彩色图，单独设置 RGB 通道）

Figure 23: 直方图均衡化（彩色图，共同设置 RGB 通道）

下面根据结果简要分析直方图均衡化的三种实现方法的特点：

- **灰度图**：个人感觉从效果上和直接转化的灰度图，故不多作评论。
- **彩色图，单独设置 RGB 通道**：如果对亮度正常、且偏暖色调的原图像（比如这里的莱娜图）进行此类操作，则会使图片变得偏冷色调；但如果对亮度偏暗的图像进行此类操作，则能够很好地增强原图像的亮度，还原效果较好。

- **彩色图，共同设置 RGB 通道**：如果对亮度正常的原图像进行此类操作，则会使图片变得偏暖色调；但如果对亮度偏暗的图像进行此类操作，则能够增强原图像的亮度，还原效果较好。

简单总结一下：如果原图像偏暗且偏冷色调，推荐使用第二种实现方式；如果原图像偏暗且偏暖色调，则推荐使用第三种实现方式。

六、 心得体会

这次实验较为简单，因为一些操作的框架已在前面几个 Lab 中设计过了，只需根据理论知识修改其中的核心操作即可。在做 Lab 的过程中，我遇到以下几个比较有意思的错误：

- 对数化操作中，我忘记在亮度的对数公式后面再乘上 255（因为原公式的计算结果范围在 $[0, 1]$ 之间），导致得到的图像呈暗红色，如下所示：

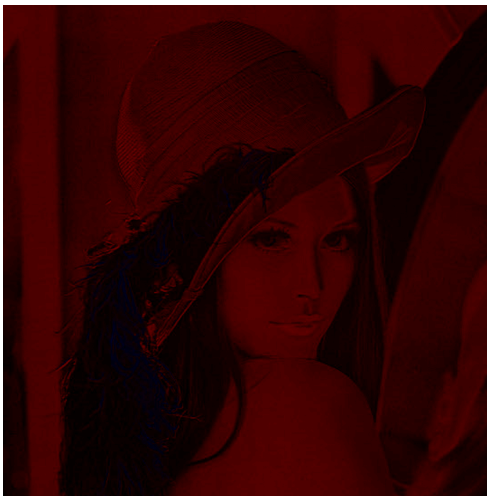


Figure 24: 失败作品 1

- 直方图均衡化的第二个实现方式中，因为我的粗心，导致在根据直方图数据修正 RGB 值的时候，R、G、B 三个值只根据其中一个通道的直方图数据修改。直到后来给代码添加注释的时候我才发现这个错误，因为得到的图像看上去很正常。出于好奇，我又故意生成另外两种错误的情况，想看看什么效果，结果如下：

Figure 25: 仅参照 B 通道直方图

Figure 26: 仅参照 G 通道直方图

Figure 27: 仅参照 R 通道直方图

虽然没有尝试过，但我猜想这三张图片重叠放在一起的效果可能就是原图的样子。

总而言之，我从本次实验中进一步学习和巩固了图像对数化操作和直方图均衡化的理论知识和实践方法，使我受益颇多。