哈爾濱工業大學

数字图像处理实验报告

实验 （二）

题　　目	Experiment 2
学　　院	计算机学院
专　　业	计算机科学与技术
学　　号	1160300909
学　　生	张志路
任课教师	姚鸿勋

哈尔滨工业大学计算机科学与技术学院

2018 年秋季

一、 实验内容

1. 实现对图像的亮度、对比度、饱和度、色度的调整。（5 Points）
2. 统计图像的直方图。（1 Point）
3. 实现图像的空域滤波：中值滤波和均值滤波。
   并选用合适的滤波方法增强如下图像。（5 Points）
4. 实现图像的边缘检测：Roberts 算子和 Sobel 算子。（5 Points）
5. 以下实验选做一个
   a) 实现中值滤波的快速算法。（本实验实现）（5 Points）
   b) 利用 CUDA 加速均值滤波。（5 Points）

注：仓库地址为 https://gitlab.com/1160300909/digital-image-processing-experiments

二、 实验目的

1. 了解图像的亮度、对比度、饱和度、色度的调整。
2. 了解图像直方图的概念。
3. 了解图像处理中常见的空域滤波算法。
4. 了解图像处理中常见的边缘检测算子。
5. 了解常见滤波的加速方式。

三、 实验环境

1. 编程语言：python 3.6.4
2. 编程工具：PyCharm 5.0.3、Anaconda3-5.1.0
3. 操作系统：Windows 10 中文版 64-bit

四、 实验设计、算法和流程

4.1 亮度调整

1. 亮度调整方法

亮度调节主要有两种实现方式：一种是非线性亮度调节，令一种是线性亮度调节。

(1) 非线性亮度调节

非线性亮度调节即对图像的 RGB 每个通道通过增加相同的值来实现。

这种方法的优点是代码简单，亮度调整速度快。缺点是图片信息损失比较大，调整

过的图像平淡,无层次感。

(2) 线性亮度调节

HSB（HSV 与 HSB 含义相同），代表色相(Hue)，饱和度(Saturation)，亮度(Brightness)三个通道的颜色，每个通道用 0~255 的数值表示。这种调节是通过对色相、饱和度和亮度三个颜色通道的变化以及相互之间的叠加来得到各种颜色。

线性亮度调节就是先将 RGB 表示的图像转换为 HSB 的颜色空间，然后对 B 通道进行调节，得到新的 B 值，再与 HS 通道合并为新的 HSB，最终转换为 RGB 得到新的图像。

这种方法的优点是调节过的图像层次感很强。缺点是代码复杂，调节速度慢，而且当亮度增减量较大时图像有较大失真。

2. 算法

本实验中利用非线性亮度调节的方法对亮度进行调整。步骤如下。

1) 利用 OpenCV 函数 cv2.cvtColor 将 RGB 转换为 HSB（HSV）的颜色空间。
2) 对每个像素点：
   a. B = B + weight，其中 weight 为权值。
   b. 对新的 B 值进行截断，保证其为 0~255 之间的整数。
   c. 新的 B 与原 H、S 通道合并为新的 HSB。
3) 将 HSB 转换为 RGB 颜色空间。
4) 生成图像。

其中，权值 weight 为正时，亮度增强；权值为负时，亮度减弱。在本实验报告中的结果展示部分取 weight = ±80。

3. 代码

```
# 调整亮度，权值weight:（-255.255）
def brightness(filename, weight):
    img = cv2.imread(filename, cv2.IMREAD_COLOR)
    img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)  # BGR to HSV(HSB)

    brightness = img_hsv[::, ::, 2]
    result = np.int16(img_hsv.copy())
    result[::, ::, 2] = np.int16(brightness) + weight  # B = B + weight

    result[result > 255] = 255  # 截断
    result[result < 0] = 0  # 截断
    result = np.uint8(result)
    result = cv2.cvtColor(result, cv2.COLOR_HSV2BGR)  # HSV(HSB) to BGR
    return result
```

4.2 对比度调整

1. 对比度调整方法

对比度（Contrast ratio）反应了图片上亮区域和暗区域的层次感。简单的来说就是使亮的地方更亮，暗的地方更暗。

反应到图像编辑上，调整对比度就是在保证平均亮度不变的情况下，扩大或缩小亮的点和暗的点的差异。既然是要保证平均亮度不变，所以对每个点的调整比例必须作用在该值和平均亮度的差值之上，这样才能够保证计算后的平均亮度不变。

最常用的调整对比度的算法是根据图像的灰度进行调整，对于每个像素点，调整公式为：$y = a + k * (x - a)$。

公式中 x 表示原始像素点亮度，a 表示整张图片的平均亮度，y 表示调整后的该像素点亮度，k 为系数。当 0<k<1 时，对比度减弱；当 k>1 时，对比度增强；当 k=1 时，图像不变。容易证明，经该公式作用后，整张图片的亮度不变。

2. 算法

1) 计算整幅图像的平均亮度 mean（可转灰度图后计算平均值）。
2) 对每一个像素点：
   a. 计算 B、G、R 所占比例 b、g、r。
   b. 新亮度值 newgray = mean + k * (gray - mean)，其中 gray 为该点原亮度值。
   c. 根据 B、G、R 比例和新亮度值计算新的 B、G、R。
3) 根据所有点的新 RGB 值，生成图像。

在本实验报告中的结果展示部分取 k = 2.3 和 k = 10/23。

3. 代码

```
# 调整对比度，系数k>0
def contrast(filename, k):
    img = cv2.imread(filename, cv2.IMREAD_COLOR)
    len1, len2, len3 = img.shape
    gray = np.dot(img[..., :3], [0.114, 0.587, 0.299])
    mean = np.sum(gray[::, ::]) / (len1 * len2)  # 平均亮度
    result = np.zeros(img.shape, np.int16)  # 记录结果

    for i in range(len1):
        for j in range(len2):
            sum_rgb = np.sum(img[i, j, ::])
            b = g = r = 1 / 3
            if sum_rgb != 0:
                b = img[i][j][0] / sum_rgb
                g = img[i][j][1] / sum_rgb
                r = img[i][j][2] / sum_rgb
            newgray = mean + k * (gray[i][j] - mean)
```

```
            newsum = newgray / (b * 0.114 + g * 0.587 + r * 0.299)
            result[i][j][0] = newsum * b
            result[i][j][1] = newsum * g
            result[i][j][2] = newsum * r

    result[result < 0] = 0
    result[result > 255] = 255
    result = np.uint8(result)
    return result
```

4.3 饱和度调整

1. 饱和度调整方法

饱和度是指色彩的鲜艳程度，也称色彩的纯度。饱和度取决于该色中含色成分和消色成分（灰色）的比例。含色成分越大，饱和度越大；消色成分越大，饱和度越小。

HSL 中 S 代表饱和度(Saturation)，用 0~255 的数值表示，因此可通过调节 HSL 中的 S 来调节饱和度。

2. 算法

1) 利用 OpenCV 函数 cv2.cvtColor 将 RGB 转换为 HSL 的颜色空间。
2) 对每个像素点：
   a. S = S + weight，其中 weight 为权值。
   b. 对新的 S 值进行截断，保证其为 0~255 之间的整数。
   c. 新的 S 与原 H、L 通道合并为新的 HSL。
3) 将 HSL 转换为 RGB 颜色空间。
4) 生成图像。

其中，权值 weight 为正时，饱和度增强；权值为负时，饱和度减弱。在本实验报告中的结果展示部分取 weight = ±80。

3. 代码

```
# 调整饱和度，权值weight:（-255.255）
def saturation(filename, weight):
    img = cv2.imread(filename, cv2.IMREAD_COLOR)
    img_hls = cv2.cvtColor(img, cv2.COLOR_BGR2HLS)  # BGR to HLS

    saturation = img_hls[:, :, 2]
    result = np.int16(img_hls.copy())
    result[:, :, 2] = np.int16(saturation) + weight  # S = S + weight

    result[result > 255] = 255  # 截断
    result[result < 0] = 0  # 截断
```

```
    result = np.uint8(result)
    result = cv2.cvtColor(result, cv2.COLOR_HLS2BGR)  # HLS to BGR
    return result
```

4.4 色度调整

1. 色度调整方法

色相（Hue）指的是色彩的外相，是在不同波长的光照射下，人眼所感觉不同的颜色，如红色、黄色、蓝色等。

在 HSL 和 HSV 色彩空间中，H 指的就是色相，是以红色为 0 度（360 度）；黄色为 60 度；绿色为 120 度；青色为 180 度；蓝色为 240 度；品红色为 300 度。因此可通过调节 HSL 中的 H 来调节色相。

2. 算法

1) 利用 OpenCV 函数 cv2.cvtColor 将 RGB 转换为 HSL 的颜色空间。
2) 对每个像素点：
   a. H = H + weight，其中 weight 为权值。
   b. 对新的 H 值进行截断，保证其为 0~255 之间的整数。
   c. 新的 H 与原 L、S 通道合并为新的 HSL。
3) 将 HSL 转换为 RGB 颜色空间。
5) 生成图像。

在本实验报告中的结果展示部分取 weight = 80,160,240。

3. 代码

```
# 调整色度，权值weight:（-255.255）
def hue(filename, weight):
    img = cv2.imread(filename, cv2.IMREAD_COLOR)
    img_hls = cv2.cvtColor(img, cv2.COLOR_BGR2HLS)  # BGR to HLS

    hue = img_hls[::, ::, 0]
    result = np.int16(img_hls.copy())
    result[::, ::, 0] = np.int16(hue) + weight  # H = H + weight

    result[result > 255] = 255  # 截断
    result[result < 0] = 0  # 截断
    result = np.uint8(result)
    result = cv2.cvtColor(result, cv2.COLOR_HLS2BGR)  # HLS to BGR
    return result
```

4.5 直方图

1. 直方图简介

直方图给出了一幅图像或一组图像中拥有给定数值的像素数量。

灰度图图像（单通道）的直方图有 256 个条目。0 号条目给出值为 0 的像素个数，1 号条目给出值为 1 的像素个数，……，以此类推。

当然直方图可以归一化，归一化后的所有条目之和等于 1，每一个条目是该特定值像素在图像中所占的比例。大多数情况下，直方图是一个单通道或者三通道的图像。

图像是由像素构成的，反映像素分布的直方图往往可以作为图像一个很重要的特征。在实际工程中，图像直方图在特征提取、图像匹配等方面都有很好的应用。

2. 算法

以下为单通道归一化后的直方图算法。

1) 读取图像，根据公式 Y = 0.299*R + 0.587*G + 0.114*B 转灰度图。
2) 初始化大小为 256 的数组 hist 为零。
3) 遍历灰度图，统计 0~255 各值的数目，记录到数组 hist 中。
4) 将数组 hist 归一化，即将数组内各值除以像素点数目。
5) 画图。

3. 代码

```
# 直方图
def hist(filename):
    img = cv2.imread(filename, cv2.IMREAD_COLOR)
    gray = np.dot(img[..., :3], [0.114, 0.587, 0.299])
    #gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    gray = np.reshape(np.uint8(gray), (-1))  # 二维转一维

    hist = np.zeros(256, int)
    for i in gray:
        hist[i] += 1
    histfre = (hist / len(gray)).tolist()

    plt.subplot(311)
    plt.bar(range(0, 256), histfre)  # 频率柱状图
    pl.ylabel("Frequency bar")
    plt.subplot(312)
    plt.hist(gray, bins=255, normed=len(gray))  # 频率直方图
    pl.ylabel("Frequency hist")
    plt.subplot(313)
    plt.hist(gray, bins=255, normed=len(gray), cumulative=True)  # 累计直方图
    pl.ylabel("Cumulative hist")
    pl.xlabel("n (0~255)")
```

```
    pl.show()
    return histfre  # 频率
```

4.6 中值滤波

1. 中值滤波简介

空域滤波按照对像素的操作方法，可以分为两类：线性滤波和非线性滤波。按照图像处理的效果，可以分为平滑滤波和锐化滤波。

中值滤波法是一种非线性平滑技术，它将每一像素点的灰度值设置为该点某邻域窗口内的所有像素点灰度值的中值，以便让周围的像素值接近真实值，从而消除孤立的噪声点。方法是用某种结构的二维滑动模板，将板内像素按照像素值的大小进行排序，生成单调上升（或下降）的为二维数据序列，然后取中值。

由于中值滤波不依赖于邻域内与要处理像素真实值相差很大的值(噪声像素)，中值滤波对处理斑点噪声和椒盐噪声非常有效。它能够很好地滤除脉冲噪声，同时又能够保护目标图像边缘。而缺点是对高斯噪声的抑制效果不是很好。

2. 算法

1) 根据窗口大小对图像进行边缘处理，复制最近的像素来对图像边缘进行填充。
2) 对每一个像素点：
   a. 确定其所在的窗口范围。
   b. 分别对窗口内的 R、G、B 元素值进行排序。
   c. 记录 R、G、B 排序后元素的中值，形成新的像素值。
3) 根据新的像素值生成新图像。

3. 代码

注：如下的 readimg 函数和 pretreatment_filter 函数在 4.7 节和 4.10 节都要用到，但不再另行给出。

```
# 读文件
def readimg(filename):
    img = cv2.imread(filename, cv2.IMREAD_COLOR)
    return img


# 预处理数据，填充最近像素值，以供滤波
def pretreatment_filter(img):
    len1, len2, len3 = img.shape
    image_out = np.zeros((len1 + 2, len2 + 2, len3), np.uint8)

    image_out[1:len1 + 1, 1:len2 + 1] = img  # 内部复制
    image_out[0:1, 1:len2 + 1] = img[0]  # 第一行
```

```
    image_out[len1 + 1:len1 + 2, 1:len2 + 1] = img[len1 - 1]  # 最后一行
    image_out[1:len1 + 1, 0:1] = np.reshape(img[:, 0], (len1, 1, 3))  # 第一列
    image_out[1:len1 + 1, len2 + 1:len2 + 2] = np.reshape(
        img[:, len2 - 1], (len1, 1, 3))  # 最后一列

    image_out[0, 0] = img[0, 0]  # 矩阵左上
    image_out[0, len2 + 1] = img[0, len2 - 1]  # 矩阵右上
    image_out[len1 + 1, 0] = img[len1 - 1, 0]  # 矩阵左下
    image_out[len1 + 1, len2 + 1] = img[len1 - 1, len2 - 1]  # 矩阵右下

    return image_out


# 中值滤波 n*n 窗口
def medianfilter(filename, n):
    k = n // 2
    preinit = pretreatment_filter(readimg(filename))  # 初始化，外部填充一层
    for i in range(k - 1):  # 初始化，外部填充多层
        pre = pretreatment_filter(preinit)
        preinit = pre.copy()

    len1, len2, len3 = preinit.shape
    image_out = preinit.copy()

    for i in range(k, len1 - k):
        for j in range(k, len2 - k):
            image_out[i][j][0] = np.median(
                preinit[i - k:i + k + 1, j - k:j + k + 1, 0])
            image_out[i][j][1] = np.median(
                preinit[i - k:i + k + 1, j - k:j + k + 1, 1])
            image_out[i][j][2] = np.median(
                preinit[i - k:i + k + 1, j - k:j + k + 1, 2])

    image_out = image_out[k:len1 - k, k:len2 - k]  # 取内层
    return image_out
```

4.7 均值滤波

1. 均值滤波简介

均值滤波是一种线性滤波，其采用的主要方法为邻域平均法。

相比于中值滤波用中值代替原图像中的各个像素值，均值滤波的基本原理是用均值代替原图像中的各个像素值。

均值滤波的优点是算法简单，计算速度快。缺点是降低噪声的同时使图像产生模糊，特别是景物的边缘和细节部分。

2. 算法

1) 根据窗口大小对图像进行边缘处理，复制最近的像素来对图像边缘进行填充。

2) 对每一个像素点：

a. 确定其所在的窗口范围。

b. 分别对窗口内的 R、G、B 元素值取均值。

c. 记录 R、G、B 元素的均值，形成新的像素值。

3) 根据新的像素值生成新图像。

3. 代码

```
# 均值滤波 n*n 窗口
def meanfilter(filename, n):
    k = n // 2
    preinit = pretreatment_filter(readimg(filename))  # 初始化，外部填充一层
    for i in range(k - 1):  # 初始化，外部填充多层
        pre = pretreatment_filter(preinit)
        preinit = pre.copy()

    len1, len2, len3 = preinit.shape
    image_out = preinit.copy()

    for i in range(k, len1 - k):
        for j in range(k, len2 - k):
            image_out[i][j][0] = np.average(
                preinit[i - k:i + k + 1, j - k:j + k + 1, 0])
            image_out[i][j][1] = np.average(
                preinit[i - k:i + k + 1, j - k:j + k + 1, 1])
            image_out[i][j][2] = np.average(
                preinit[i - k:i + k + 1, j - k:j + k + 1, 2])

    image_out = image_out[k:len1 - k, k:len2 - k]  # 取内层
    return image_out
```

4.8 Roberts 算子

1. Roberts 算子

Roberts 算子中在图像处理和计算机视觉用于边缘检测。它是最早的边缘探测器之一，最初由 Lawrence G. Roberts 于 1963 年提出。

作为局部差分算子，Roberts 算子的原理是通过计算离散微分逼近图像的梯度，而

对离散微分的计算是通过计算对角相邻像素之间差异的平方和实现的。

Roberts 算子对噪声敏感，无法抑制噪声的影响，也因此影响到计算定位时容易丢失一部分的边缘。但是 Roberts 算子的定位精度较高，对具有陡峭的低噪声图像响应较好。

2. 算法

1) 彩色图转灰度图 gray。
2) 初始化全零二维数组 roberts，大小与灰度图大小相同。
3) 对第 i 行 j 列的像素点 gray[i][j]：
   a. roberts[i][j]=

$$\sqrt{(\text{gray}[i][j]-\text{gray}[i+1][j+1])^2-(\text{gray}[i+1][j]-\text{gray}[i][j+1])^2}$$

4) 根据数组 roberts 画出图像。

3. 代码

```
# Roberts 算子
def roberts(filename):
    img_gray = cv2.imread(filename, 0)
    len1, len2 = img_gray.shape
    image_out = np.zeros((len1, len2), np.uint8)

    for i in range(0, len1 - 1):
        for j in range(0, len2 - 1):
            a = int(img_gray[i][j]) - int(img_gray[i + 1][j + 1])
            b = int(img_gray[i + 1][j]) - int(img_gray[i][j + 1])
            image_out[i][j] = np.sqrt(np.power(a, 2) + np.power(b, 2))

    plt.subplot(131), plt.imshow(img_gray, cmap='gray')
    plt.title('Original'), plt.xticks([]), plt.yticks([])
    plt.subplot(132), plt.imshow(image_out, cmap='gray')
    plt.title('Roberts'), plt.xticks([]), plt.yticks([])
    plt.subplot(133), plt.imshow(255 - image_out, cmap='gray')
    plt.title('Roberts'), plt.xticks([]), plt.yticks([])
    plt.show()
    return image_out
```

4.9 Sobel 算子

1. **Sobel 算子**

Sobel 算子是一种一阶微分算子，它能根据像素点周围的梯度值来计算该像素点的梯度，之后会根据事先设好的阈值比较，来进行取舍。

与 Roberts 算子不同的是，Sobel 算子是 3×3 算子模板。该算子包含两组 3×3 的矩阵，分别为横向及纵向，将之与图像作平面卷积，即可分别得出横向及纵向的亮度差分近似值。

如果以 A 代表原始图像，G_x 及 G_y 分别代表经横向及纵向边缘检测的图像，则结果 G 可表示如下。

可令 $G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A$ ，$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A$

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

2. 算法

1) 彩色图转灰度图 gray。
2) 初始化全零二维数组 Sobel，大小与灰度图大小相同。
3) 对第 i 行 j 列的像素点 gray[i][j]：
   a. Sobel [i][j] = G（如上所述 G）。
4) Sobel[Sobel<=threshold] = 0。
5) 根据数组 Sobel 画出图像。

在本实验报告中的结果展示部分阈值取 threshold = 50。

3. 代码

```
# Sobel 算子
def sobel(filename):
    img_gray = cv2.imread(filename, 0)
    len1, len2 = img_gray.shape
    image_out = np.zeros((len1, len2), np.uint8)

    for i in range(1, len1 - 1):
        for j in range(1, len2 - 1):
            a = int(img_gray[i - 1][j - 1] + 2 * img_gray[i][j - 1] +
                    img_gray[i + 1][j - 1])
            b = int(img_gray[i - 1][j + 1] + 2 * img_gray[i][j + 1] +
                    img_gray[i + 1][j + 1])
            c = int(img_gray[i + 1][j - 1] + 2 * img_gray[i + 1][j] +
                    img_gray[i + 1][j + 1])
            d = int(img_gray[i - 1][j - 1] + 2 * img_gray[i - 1][j] +
                    img_gray[i - 1][j + 1])
            image_out[i][j] = np.sqrt(np.power(a - b, 2) + np.power(c - d, 2))

    threshold = 50  # 阈值
    image_out[image_out <= threshold] = 0

    plt.subplot(131), plt.imshow(img_gray, cmap='gray')
```

```
    plt.title('Original'), plt.xticks([]), plt.yticks([])
    plt.subplot(132), plt.imshow(image_out, cmap='gray')
    plt.title('Sobel'), plt.xticks([]), plt.yticks([])
    plt.subplot(133), plt.imshow(255 - image_out, cmap='gray')
    plt.title('Sobel'), plt.xticks([]), plt.yticks([])
    plt.show()
    return image_out
```

4.10 中值滤波的快速算法

1. 中值滤波快速方法

对于朴素的中值滤波算法而言，其需要在每个像素位置上都要对一个矩形内部的所有像素进行排序，然后进行排序，这样的开销很大。在一般的情况下，该算法的复杂度是 $O(r^2 \log r)$，其中 r 为核的半径。当像素的可能取值是个常数时，比如对于 8 位图像，可以使用桶排序(Bucker sort)，该排序使得算法的复杂度降低为 $O(r^2)$。

在《A Fast Two-Dimensional Median Filtering Algorithm》一文中，提出了基于直方图统计的快速中值滤波，其时间复杂度为 O(r)。其基本想法是：当窗口沿着行移一列时，窗口内容的变化只是舍去了最左边的列，而取代为一个新的右侧列，因此对于 x 行 y 列的中值滤波窗口，有 xy - 2m 个像素并没有变化，并不需要重新排序。

此外，在《A Coarse-to-Fine Algo-rithm for Fast Median Filtering of Image Data With a Huge Number of Levels》一文中，提出了多尺度直方图的思想。其基本想法是维持一个平行的较小的直方图，直方图记录了图像的高位数据。

在《Median Filtering in Constant Time》一文中，通过并行化、缓存优化和条件更新核等操作，甚至将算法复杂度降至线性。

本实验中仅实现了基于直方图统计的快速中值滤波算法，具体描述如下。

2. 算法

1) 置 th = int(xy / 2)，x 和 y 为窗口的长和宽。
2) 将窗口移至一个新行的开始，对其内容排序，确定其中值 m，记下亮度≤m 的像素数目 N_m，并建立窗口像素的直方图 H。
3) 对于左列亮度是 P_g 的每一个像素 p，做：$H[P_g] = H[P_g] - 1$。
   如果 $P_g \le m$，置 $N_m = N_m - 1$。
4) 将窗口右移一列，对于右列亮度是 P_g 的每一个像素 p，做：$H[P_g] = H[P_g] + 1$。
   如果 $P_g \le m$，置 $N_m = N_m + 1$。
5) 如果 $N_m > th$：重复 $m = m - 1$，$N_m = N_m - H[m+1]$；直到 $N_m \le th$。
6) 如果 $N_m < th$：重复 $m = m + 1$，$N_m = N_m + H[m]$；直到 $N_m \ge th$。
7) 记录 m，当前 m 即为中值。

8) 如果窗口的右侧列不是图像的右边界，跳转至步骤 3。

9) 如果窗口的低行不是图像的下边界，跳转至步骤 2。

3. 代码

```
# 中值滤波快速算法，n*n 窗口
def fast_medianfilter(filename, n):
    k = n // 2
    th = n * n // 2
    preinit = pretreatment_filter(readimg(filename))  # 预处理，外部填充一层
    for i in range(k - 1):  # 外部填充多层
        pre = pretreatment_filter(preinit)
        preinit = pre.copy()
    len1, len2, len3 = preinit.shape
    image_out = preinit.copy()  # 记录结果

    #  每行维护直方图
    def winhist(win):
        for i in np.reshape(win[::, ::, 0], (-1)):
            hist[0][i] += 1
            if i <= medpoint[0]:
                num[0] += 1
        for j in np.reshape(win[::, ::, 1], (-1)):
            hist[1][j] += 1
            if j <= medpoint[1]:
                num[1] += 1
        for k in np.reshape(win[::, ::, 2], (-1)):
            hist[2][k] += 1
            if k <= medpoint[2]:
                num[2] += 1

    medpoint = np.zeros(3, np.int16)  # 记录中值，RGB 三通道
    for i in range(k, len1 - k):
        image_out[i][k][0] = medpoint[0] = np.median(
            preinit[i - k:i + k + 1, 0:2 * k + 1, 0])
        image_out[i][k][1] = medpoint[1] = np.median(
            preinit[i - k:i + k + 1, 0:2 * k + 1, 1])
        image_out[i][k][2] = medpoint[2] = np.median(
            preinit[i - k:i + k + 1, 0:2 * k + 1, 2])

        win = preinit[i - k:i + k + 1, 0:2 * k + 1, ::]  # 每行的第一个窗口
        hist = np.zeros((3, 256), int)  # 直方图，RGB 三通道
        num = np.zeros(3, int)  # 记录<=中值的数目，RGB 三通道
        winhist(win)
```

```
        for j in range(k + 1, len2 - k):
            # 处理左列
            def leftdeal(left):
                for x in range(2 * k + 1):
                    for y in range(3):
                        hist[y][left[x][y]] -= 1
                        if left[x][y] <= medpoint[y]:
                            num[y] -= 1

            left = preinit[i - k:i + k + 1, j - k - 1, ::]
            leftdeal(left)

            # 处理右列
            def rightdeal(right):
                for x in range(2 * k + 1):
                    for y in range(3):
                        hist[y][right[x][y]] += 1
                        if right[x][y] <= medpoint[y]:
                            num[y] += 1

            right = preinit[i - k:i + k + 1, j + k, ::]
            rightdeal(right)

            for ii in range(3):
                while num[ii] > th:
                    medpoint[ii] -= 1
                    num[ii] -= hist[ii][medpoint[ii] + 1]
                while num[ii] < th:
                    medpoint[ii] += 1
                    num[ii] += hist[ii][medpoint[ii]]
                image_out[i][j][ii] = medpoint[ii]

    image_out = image_out[k:len1 - k, k:len2 - k]  # 截取内层数据
    return image_out
```

五、 实验结果

1. 亮度调整结果

如下图所示，图 1.1 为原图，图 1.2 为亮度增强后的结果，图 1.3 为亮度减弱后的结果。

可以看出，亮度增强后画面整体偏白，而亮度减弱后画面整体偏暗。

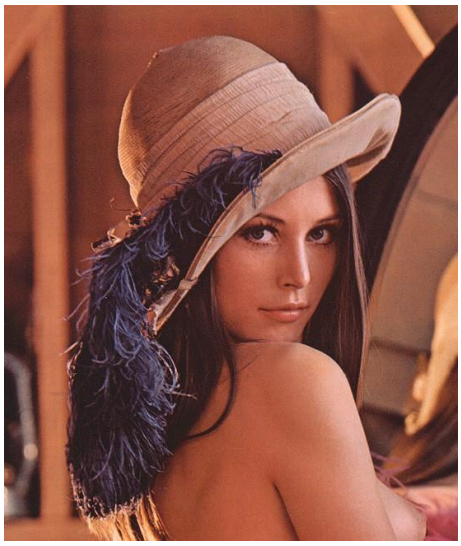

图 1.1

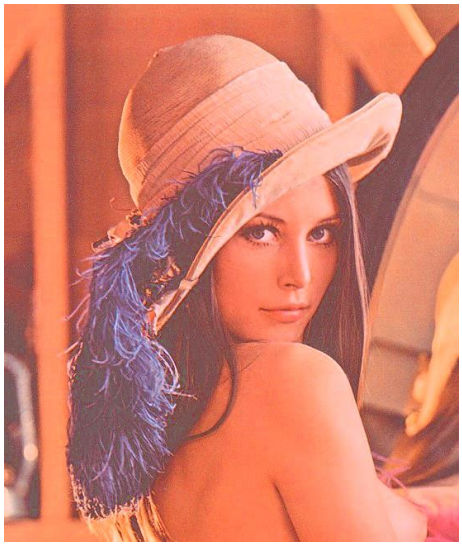

图 1.2

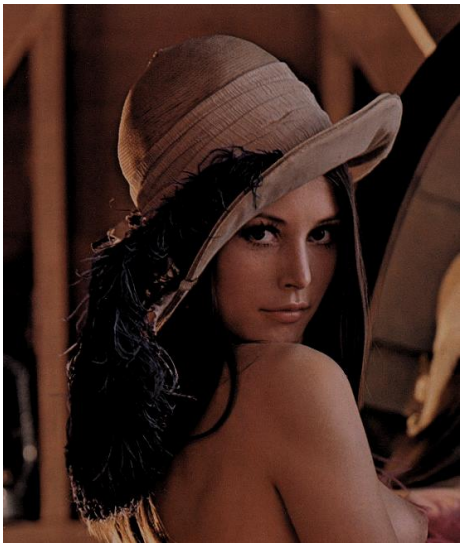

图 1.3

2. 对比度调整结果

如下图所示，图 2.1 为原图，图 2.2 为对比度增强后的结果，图 2.3 为对比度减弱后的结果。

可以看出，对比度增强后亮的点和暗的点的差异扩大，而对比度减弱后亮的点和暗的点的差异缩小。

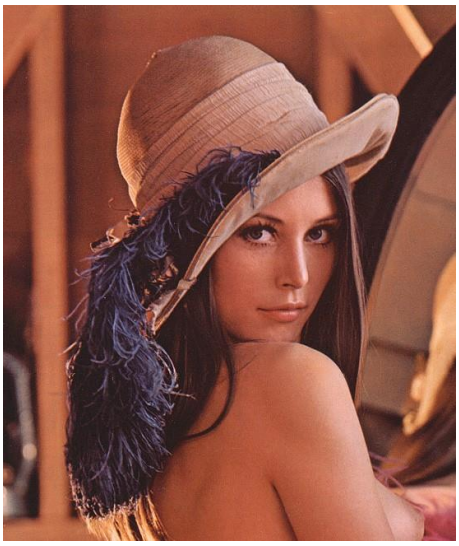

图 2.1

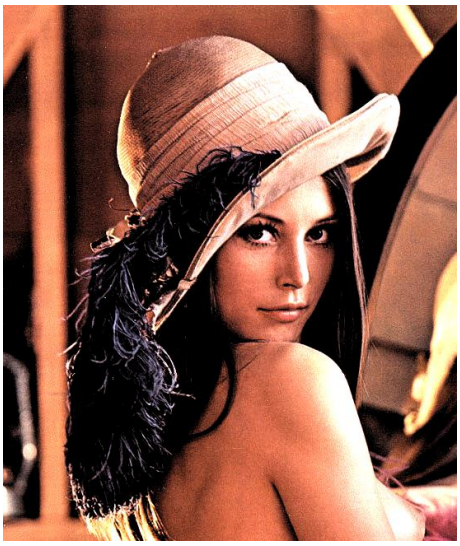

图 2.2

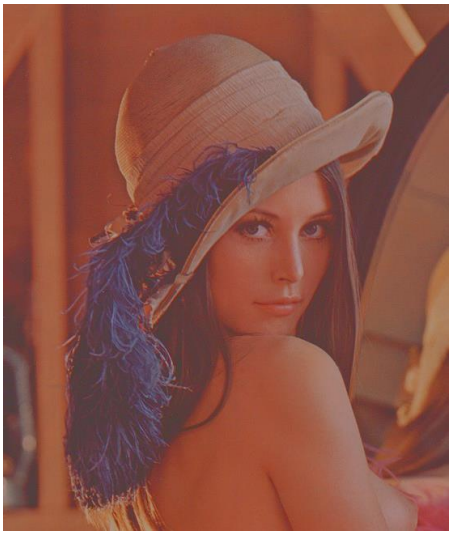

图 2.3

3. 饱和度调整结果

如下图所示，图 3.1 为原图，图 3.2 为饱和度增强后的结果，图 3.3 为饱和度减弱后的结果。

可以看出，饱和度增强后颜色更加鲜艳，色彩的纯度更高，而饱和度减弱后色彩的纯度明显减弱。

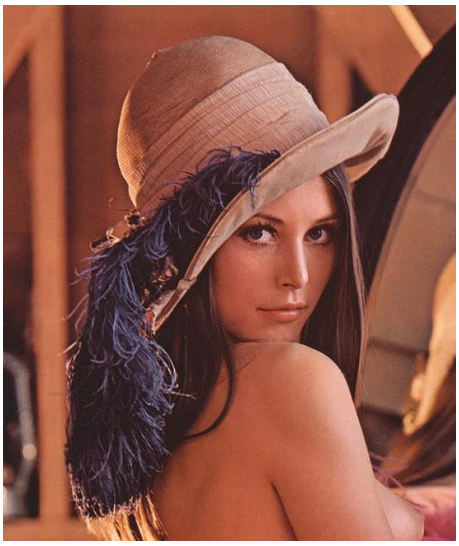

图 3.1

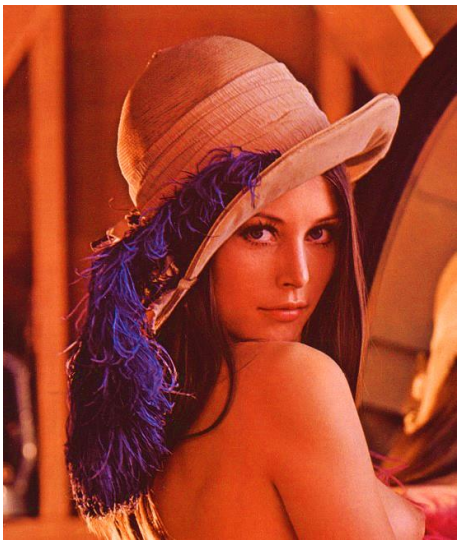

图 3.2

图 3.3

4. 色度调整结果

如下图所示，图 4.1 为原图，图 4.2 为色度增强 80（HSL 中的 H，下同）后的结果，图 4.3 为色度增强 160 后的结果，图 4.4 为色度增强 240 后的结果。

可以看出，调整后的色调发生了明显变化。

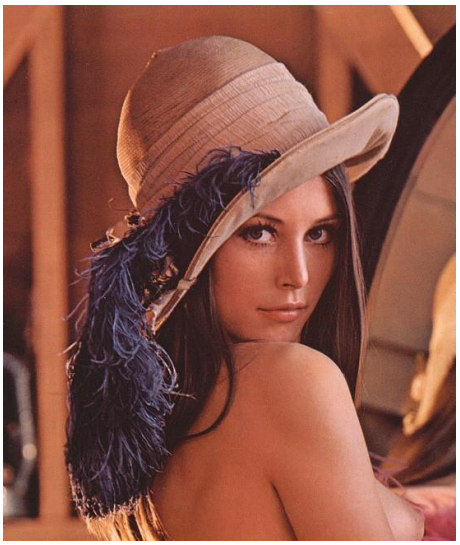

图 4.1

图 4.2

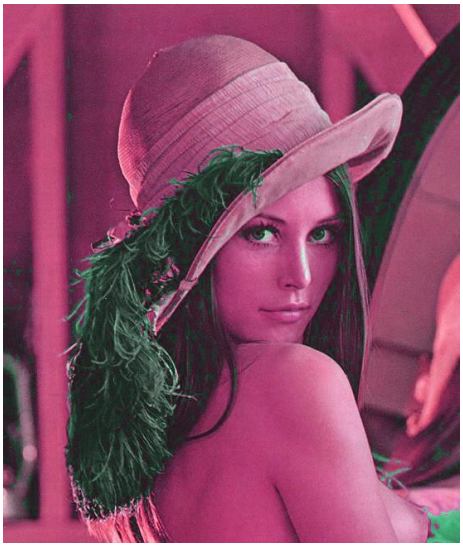

图 4.3

图 4.4

5. 直方图结果

如下图所示，图 5.1 为原图；图 5.2 为其灰度图；图 5.3 为利用 matplotlib 库根据灰度图所做的直方图，其中第一行是计算出频率数组后利用 plt.bar 所做频率柱状图，第二行是直接利用 plt.hist 函数生成的频率直方图，第三行是直接利用 plt.hist 函数生成的累计频率直方图。

可以看出，程序计算出频率后所画的直方图与直接利用 plt.hist 函数所画直方图基本一致。

图 5.1

图 5.2

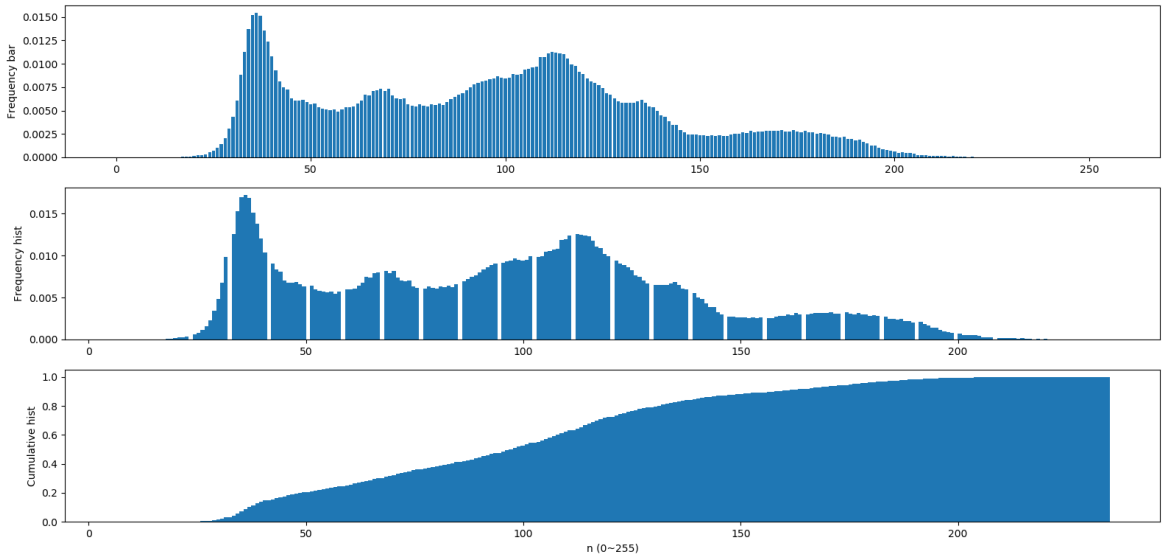

图 5.3

6. 中值滤波结果

如下图所示，图 6.1.1 为原图，图 6.1.2 为经 3*3 窗口中值滤波后的结果，图 6.1.3 为经 5*5 窗口中值滤波后的结果。图 6.2.1 为原图，图 6.2.2 为经 3*3 窗口中值滤波后的结果，图 6.2.3 为经 5*5 窗口中值滤波后的结果。

滤波的目的主要是为了去除噪声。从统计学的观点来看，凡是统计特征不随时间变化的噪声称为平稳噪声，而统计特征随时间变化的噪声称为非平稳噪声。幅值基本相同，但是噪声出现的位置是随机的，称为椒盐噪声；如果噪声的幅值是随机的，根据幅值大小的分布，有高斯型和瑞利型两种，分别称为高斯噪声和瑞利噪声。

两幅图经处理后的共同特点是：窗口模板越大，图像越模糊。去除噪音的同时，也平滑了图像的细节。

对于图 6.1.1，其经处理后，去除了部分噪声，但图像也更加模糊，相对来讲有点得不偿失。

对于图 6.2.1，其经 3*3 窗口处理后噪声变得已经不明显，经 5*5 窗口处理后噪声基本消失，而图像的清晰度也在一定程度上得以保留。

图 6.1.1

图 6.1.2（3*3 窗口）

图 6.1.3（5*5 窗口）

图 6.2.1

图 6.2.2（3*3 窗口）

图 6.2.3（5*5 窗口）

7. 均值滤波结果

如下图所示，图 7.1.1 为原图，图 7.1.2 为经 3*3 窗口均值滤波后的结果，图 7.1.3 为经 5*5 窗口均值滤波后的结果。图 7.2.1 为原图，图 7.2.2 为经 3*3 窗口均值滤波后的结果，图 7.2.3 为经 5*5 窗口均值滤波后的结果。

与中值滤波相同的是，两幅图经处理后的共同特点是：窗口模板越大，图像越模糊，特别是景物的边缘和细节部分。去除噪音的同时，也平滑了图像的细节。

对于图 7.1.1，其经处理后的图像与中值滤波差异并不明显，去除了部分噪声的同时，图像也更加模糊。

而对于图 7.2.1，其经处理后噪声依然比较明显，本来的点状噪声变成了块状噪声。而且与相同模板的中值滤波相比，图像也更加模糊。

综合均值滤波和中值滤波的结果，得出结论，对于图 7.1.1（或 6.1.1），使用中值滤波与均值滤波的效果并无明显差异；而对于图 7.2.1（或 6.2.1），使用中值滤波效果更好。

而一般来讲，对于高斯噪声一般采用均值滤波，对于椒盐噪声一般采用中值滤波。

图 7.1.1

图 7.1.2（3*3 窗口）

图 7.1.3（5*5 窗口）

图 7.2.1

图 7.2.2（3*3 窗口）

图 7.2.3（5*5 窗口）

8. Roberts 算子结果

如下图所示，图 8.1 为灰度图，图 8.2 为经 Roberts 算子作用后得到的结果，图 8.3 为图 8.2 求反后的结果。

可以看出，Robert 算子定位比较精确，且检测垂直边缘的效果好于斜向边缘，但由于不包括平滑，所以对于噪声比较敏感，无法抑制噪声的影响。

图 8.1

图 8.2

图 8.3

9. Sobel 算子结果

如下图所示，图 9.1 为灰度图，图 9.2 为经 Sobel 算子（阈值为 50）作用后得到的结果，图 9.3 为图 9.2 求反后的结果。

可以看出，Soble 算子对噪声有抑制作用，因此不会出现很多孤立的边缘像素点。然而 Sobel 算子对边缘的定位不是很准确，图像的边界宽度往往不止一个像素，计算方向单一，对复杂纹理的情况显得乏力。

直接用阈值的方法来判断边缘点也欠合理解释，会造成较多的噪声点误判。但综合来看，Sobel 算子仍比 Roberts 算子的效果要好一些。

图 9.1

图 9.2

图 9.3

10. 快速中值滤波

本实验实现的快速滤波中值算法理论上的时间复杂度是 O(r)（r 是窗口半径），相比于朴素算法的 $O(r^2)$级有了明显提高。

但是由于本实验中编程语言是 python，当实现朴素算法时，利用的是 numpy 库中

的 np.median 函数，而该函数大部分过程都是使用 C 实现的。因此使得朴素算法与快速算法的实际运行用时并不具有可比性。实际测量结果也表明，快速算法用时甚至达到朴素算法的 2 倍。

11. 主界面

运行时主界面如图 11 所示。

```
D:\学习-课程\大三\视听觉\实验\视觉\实验2\code\code v2>ipython 1160300909-Exp-2.py path/to/Test-Image-2.bmp
 1.  调整亮度
 2.  调整对比度
 3.  调整饱和度
 4.  调整色度
 5.  显示直方图
 6.  中值滤波
 7.  均值滤波
 8.  Roberts算子边缘检测
 9.  Sobel算子边缘检测
 10. 快速中值滤波

请输入所选序号（1-9）：1
请输入亮度调整权重（-255,255）：50
已经存入 Test-Image-2--brightness(weight=50).bmp 文件

是否继续?
请输入1(是)或者0(否)：1
 1.  调整亮度
 2.  调整对比度
 3.  调整饱和度
 4.  调整色度
 5.  显示直方图
 6.  中值滤波
 7.  均值滤波
 8.  Roberts算子边缘检测
 9.  Sobel算子边缘检测
 10. 快速中值滤波

请输入所选序号（1-9）：2
请输入对比度调整倍数（>0）：2.3
已经存入 Test-Image-2--contrast(k=2.3).bmp 文件
```

图 11

六、 结论

实验主要结论和结果已经在第五部分一并叙述，这部分谈谈体会与收获。

1. 本次实验可以说是达到了实验目的，完成了实验内容。
2. 在做亮度、对比度、饱和度以及色度调整时，面临着多种调整方法，在不断地查找资料，尝试实现不同方法中，逐渐确定较为准确而且简单的算法。其实当理论基础足够扎实，很好地掌握这些基本概念以及色彩空间时，完全可以一步到位，确定出最佳的算法。这也反映出自己的理论知识不够扎实，需要不断地进行学习。
3. 做实验时，一边自己编程进行实现，一边利用 OpenCV 的函数进行检测，这让我对 OpenCV 也加深了了解。
4. 实现中值滤波快速算法时，我查阅了许多论文，这似乎让我感到实验真正的意义，即不断探索，不断尝试，不断追求完美的结果。但遗憾的是，限于时间与精力，我只实现了 O(r)级别的算法，并没有实现 O(1)的算法，过后一定要再进行尝试。
5. 图像处理是一件有趣的事，把有趣的事做好是一件美好的事。

七、 参考文献

[1] 冈萨雷斯, 伍兹. 数字图像信号处理[M]. 北京：电子工业出版社，2011：249-267.

[2] List of color spaces and their uses[EB/OL]. [2018-12-12]. https://en.wikipedia.org-/wiki/List_of_color_spaces_and_their_uses.

[3] HSL and HSV[EB/OL]. [2018-12-12]. https://en.wikipedia.org/wiki/ HSL_and_HSV.

[4] Contrast ratio[EB/OL]. [2018-12-12]. https://en.wikipedia.org/wiki/Contrast_ratio.

[5] Hue[EB/OL]. [2018-12-05]. https://en.wikipedia.org/wiki/Hue.

[6] Median filter[EB/OL]. [2018-12-05].https://en.wikipedia.org/wiki/Median_filter.

[7] Roberts cross[EB/OL]. [2018-12-05].https://en.wikipedia.org/wiki/Roberts_cross.

[8] Sobel operator[EB/OL]. [2018-12-05].https://en.wikipedia.org/wiki/Sobel_operator.

[9] S. Perreault, P. Hebert. Median Filtering in Constant Time[J]. IEEE Transactions on Image Processing, 2007.

[10] T.S. Huang, G.J. Yang, G.Y. Tang. A fast two-dimensional median filtering algorithm[J]. IEEE Transactions on Acoustics, Speech, and Signal Processing, 1979, 27(1):13-18.

[11] L. Alparone, V. Cappellini, A. Garzelli. A Coarse-to-Fine Algorithm for Fast Median Filtering of Image Data With a Huge Number of Levels[J]. Signal Processing, 1994, 39(1/2):33–41.