3

# 实验名称

图像增强(python实现)

# 实验目的与要求

本次实验主要实现并掌握图像增强相关算法；编程实现线性/非线性灰度变换、分段线性变换、对数变换、指数变换、幂律变换；实现直方图计算和直方图均衡化算法；编程实现邻域/中值平滑算法，并能分析邻域/中值平滑图像处理的效果，及其对于椒盐噪声和高斯噪声不同的处理结果。

# 实验内容

一、灰度变换  
a) 线性变换；(10分)  
b) 对数变换； (10分)  
c) 指数变换; (10分)  
d) 幂律变换； (10分)

注意：理解各个灰度变换算法实现的效果和可优化的场景

二、图像修正  
e) 利用直方图均衡算法实现图像修正

三、图像平滑  
f) 邻域平均滤波修正图像  
g) 中值滤波修正图像  
注意：理解两种算法的优缺点，考虑有没有提高程序运行效率的方法(针对中值滤波)

# 实验仪器与设备

联想笔记本电脑。  
13代i7处理器，32G内存。  
4070显卡。

# 实验原理

## 灰度变换

灰度变换是指根据某种目标条件按一定变换关系逐点改变源图像中每一个像素灰度值的方法。通常可以用公式表示为：

其中，f(x,y)是输入图像在点(x,y)处的灰度值，g(x,y)是输出图像在点(x,y)处的灰度值，T是变换函数。那灰度变换就是将输入图像f(x,y)的灰度值按照某种变换关系转换为输出图像g(x,y)的灰度值。

### 线性变换

线性变换主要作用是将原图像的灰度值线性拉伸到指定的范围，假定原图像的灰度值范围为[a,b]，线性变换后图像的灰度值范围为[c,d]，则线性变换公式为：

对原图像中的每个像素点做上述变换映射，即可将灰度范围线性拉伸到[c,d]。

有时为了突出感兴趣的灰度区间，相对抑制那些不感兴趣的灰度区间，可以采用分段线性变换。常用的3段线性变换数学表达式：

其中，a,b,c,d为常数，且a<b<255。

通过调整a,b,c,d的值，可以选择将不同区间的灰度值拉伸到不同的范围。

线性变换与分段线性变换的函数关系图：

### 对数变换

对数变换的作用是将输入图像的灰度值进行非线性拉伸，使得灰度值较低的区域得到增强，灰度值较高的区域得到压缩。对数变换的公式为：

其中，c为常数，f(x,y)为输入图像的灰度值，g(x,y)为输出图像的灰度值。

### 指数变换

指数变换的作用是将输入图像的灰度值进行非线性拉伸，使得灰度值较低的区域得到增强，灰度值较高的区域得到压缩。指数变换的公式为：

其中，c和a为常数，f(x,y)为输入图像的灰度值，g(x,y)为输出图像的灰度值。

### 幂律变换

幂律变换的作用是将输入图像的灰度值进行非线性拉伸，使得灰度值较低的区域得到增强，灰度值较高的区域得到压缩。幂律变换的公式为：

其中，c和a为常数，f(x,y)为输入图像的灰度值，g(x,y)为输出图像的灰度值。

非线性变换函数关系图：

## 图像修正

### 直方图均衡化

直方图均衡化是一种有效的图像增强技术,其目的是调整图像的灰度分布,使之更加均匀,从而提高图像的整体对比度。这种方法特别适用于背景和前景都较暗或较亮的图像。

#### 数学原理

设原始图像的灰度级为 L (通常为 256),像素总数为 n。令 表示灰度级为 的像素数,其中 。则图像的灰度直方图可以定义为:

这里 表示灰度级 出现的概率。

直方图均衡化的核心思想是找到一个变换函数 T(r),将输入图像的灰度分布转换为均匀分布。这个变换函数定义如下:

其中 是均衡化后的灰度值。

这个变换实际上是原始图像灰度的累积分布函数(CDF)的离散形式。通过这种变换可以得到一个新的灰度分布,理论上这个分布会更加均匀。

#### 数学证明

我们可以证明,这种变换确实能产生一个更均匀的灰度分布。设 是连续的单调递增函数,且 。令 和 分别表示 和 的概率密度函数。根据概率论,我们有:

对于直方图均衡化,我们有:

对 求导,得到:

将这个结果代入之前的等式:

这证明了变换后的灰度分布是均匀的。

## 图像平滑

### 邻域平均滤波

邻域平均滤波是一种简单的图像平滑技术,通过将每个像素的灰度值替换为其邻域内所有像素灰度值的平均值来实现。这种方法可以有效地减少图像中的噪声,但也会导致图像细节的模糊。邻域平均的实现通常使用一个卷积核，卷积核在图像上滑动，将每个像素的灰度值替换为其邻域内所有像素灰度值的平均值。

#### 数学原理

邻域平均滤波的数学原理可以用卷积操作来描述。设原始图像为 ，滤波后的图像为 ，滤波核为 ，则邻域平均滤波可以表示为:

其中，\* 表示卷积操作， 是滤波核的大小。

卷积操作示意图：

领域平均模板的卷积核系数加和为1。  
常用的两个邻域平均模版为：

**3x3均值Box模版：**

Box模板中加权系数相同，邻域中各个像素对平滑结果的影响相同。往往不能很好地保护图像细节，在图像去噪的同时，也破坏了图像的细节部分，从而使图像变得模糊。

**3x3高斯模版：**

领域高斯模板通过对二维高斯函数进行采样、量化并归一化得到，它考虑了像素之间的距离，距离中心越近的像素权重越大，距离越远的像素权重越小。因此，高斯模板在去噪的同时，能够更好地保护图像的细节部分，使图像更加清晰。

#### 性质分析

1. 噪声抑制：邻域平均滤波可以有效地抑制加性高斯噪声。对于每个像素，噪声的期望值为零，因此在计算平均值时，噪声会相互抵消。
2. 边缘模糊：由于邻域平均不考虑像素之间的差异，它会导致图像边缘和细节的模糊。边缘处的像素值会被周围较低或较高的像素值"拉平"。
3. 频域特性：在频域中，邻域平均滤波相当于一个低通滤波器，它会衰减高频分量（对应图像中的细节和边缘），而保留低频分量（对应图像中的整体结构）。

## 中值滤波

中值滤波是一种非线性滤波技术，它通过将每个像素的灰度值替换为其邻域内所有像素灰度值的中值来实现。这种方法可以有效地去除图像中的**椒盐噪声**，同时保留图像的细节。

### 数学原理

设原始图像为 f(x,y)，滤波后的图像为 g(x,y)，滤波窗口为 W，则中值滤波可以表示为:

其中，median 表示取中值操作，W 通常是一个正方形区域，如 3×3 或 5×5。

### 实现步骤

1. 定义滤波窗口大小（如 3×3）。
2. 对图像中的每个像素：  
   a. 提取以该像素为中心的邻域像素值。  
   b. 对这些像素值进行排序。  
   c. 选取排序后的中间值作为该像素的新值。
3. 重复步骤2，直到处理完所有像素。

### 性质分析

1. 噪声抑制：中值滤波对于去除椒盐噪声特别有效，因为这种噪声通常表现为极高或极低的灰度值，在排序过程中会被排到两端，不会影响中值。
2. 边缘保护：与均值滤波相比，中值滤波能更好地保护图像边缘。因为边缘通常表现为灰度值的阶跃变化，中值操作不会像均值那样"拉平"这种变化。
3. 细节保护：对于大面积的同质区域，中值滤波能很好地保留细节，因为这些区域的中值就是原始值。

### 优缺点

优点：

1. 对椒盐噪声有很强的抑制能力。
2. 能较好地保护图像边缘，不会像均值滤波那样导致边缘模糊。
3. 对于脉冲干扰和扫描噪声的处理效果较好。

缺点：

1. 计算复杂度较高，特别是当窗口尺寸较大时。
2. 对于高斯噪声的处理效果不如均值滤波。
3. 可能会导致图像中一些细小的线条或尖锐的角点被删除。

# 实验过程及代码

## 灰度变换

### 线性变换

#### 实现步骤

1. 获取原始图像的灰度范围
2. 通过用户交互获取目标灰度范围
3. 计算线性变换的斜率
4. 对每个像素应用线性变换公式:
5. 确保变换后的像素值在 范围内

代码中：

* k \* (img - a) + c:
  + 这行代码利用了NumPy的广播机制,一次性对整个图像数组进行操作。
  + img - a 将每个像素值减去最小灰度值a。
  + 乘以k后加c,完成线性变换。
* np.clip(new\_img, 0, 255):
  + 使用NumPy的clip函数将所有像素值限制在[0, 255]范围内。

代码：

# 线性变换 只显示核心部分  
def linear\_strench(img):  
 time1 = time.time() # 程序计时开始  
 # 核心代码：线性拉伸  
 k = (d - c) / (b - a)   
 new\_img = k \* (img - a) + c # img - a 利用广播机制，对每个像素进行操作  
 new\_img = np.clip(new\_img, 0, 255).astype(np.uint8) # 确保像素值在0-255之间 且像素值为整数  
 time2 = time.time() # 程序计时结束  
 print("灰度增强程序处理时间：%.3f毫秒" % ((time2 - time1) \* 1000))  
 return new\_img

### 对数变换

#### 实现步骤

1. 确定用户输入的参数a,b,c
2. 对每个像素应用对数变换公式:
3. 确保变换后的像素值在 范围内

代码中：

* np.log1p(img):
  + 使用NumPy的log1p函数计算 的对数。
  + 利用广播机制，对每个像素进行操作。

代码：

# 对数变换 只显示核心部分  
def log\_strench(img):  
 time1 = time.time() # 程序计时开始  
 # 核心代码：对数变换  
 new\_img = a + np.log1p(img) / (c \* np.log(b)) # np.log1p(x) = np.log(1+x) 还是利用广播机制  
 new\_img = np.clip(new\_img, 0, 255).astype(np.uint8) # 确保像素值在0-255之间 且像素值为整数  
  
 time2 = time.time() # 程序计时结束  
 print("对数变换程序处理时间：%.3f毫秒" % ((time2 - time1) \* 1000))  
 return new\_img

### 指数变换

#### 实现步骤

1. 确定用户输入的的参数a，b，c
2. 对每个像素应用指数变换公式:
3. 确保变换后的像素值在 范围内

代码中：

* np.exp(img):
  + 使用NumPy的exp函数计算 。
  + 利用广播机制，对每个像素进行操作。

代码：

# 指数变换 只显示核心部分  
def exp\_strench(img):  
 time1 = time.time() # 程序计时开始  
 # 核心代码：指数变换  
 new\_img = c \* np.exp(a \* img)  
 new\_img = np.clip(new\_img, 0, 255).astype(np.uint8) # 确保像素值在0-255之间 且像素值为整数  
 time2 = time.time() # 程序计时结束  
 print("指数变换程序处理时间：%.3f毫秒" % ((time2 - time1) \* 1000))  
 return new\_img

### 幂律变换

#### 实现步骤

1. 确定用户输入的的参数c，r
2. 对每个像素应用幂律变换公式:
3. 确保变换后的像素值在 范围内

代码中：

* img \*\* r:
  + 使用NumPy的幂运算符计算 。
  + 利用广播机制，对每个像素进行操作。

代码：

# 幂律变换 只显示核心部分  
def power\_strench(img):  
 time1 = time.time() # 程序计时开始  
 # 核心代码：幂律变换  
 new\_img = c \* img \*\* r  
 new\_img = np.clip(new\_img, 0, 255).astype(np.uint8) # 确保像素值在0-255之间 且像素值为整数  
 time2 = time.time() # 程序计时结束  
 print("幂律变换程序处理时间：%.3f毫秒" % ((time2 - time1) \* 1000))  
 return new\_img

好的,我来帮您完善直方图均衡化的实验过程及代码说明。

## 图像修正

### 直方图均衡化

#### 实现步骤

1. 计算图像的直方图 （这里使用的是numpy的**bincount函数**）
2. 计算累积分布函数(CDF)
3. 归一化CDF到新的灰度范围(通常是0-255)
4. 使用归一化后的CDF作为查找表,将原始图像的每个像素映射到新的灰度值
5. 对于彩色图像,对**每个颜色通道**分别进行上述步骤

#### 具体计算过程

假设我们有一个4x4的灰度图像,灰度值范围为0-7:

1. 计算直方图:

| 灰度值 | 像素数 | 概率 |
| --- | --- | --- |
| 0 | 1 | 1/16 |
| 1 | 2 | 2/16 |
| 2 | 3 | 3/16 |
| 3 | 3 | 3/16 |
| 4 | 2 | 2/16 |
| 5 | 2 | 2/16 |
| 6 | 2 | 2/16 |
| 7 | 1 | 1/16 |

1. 计算累积分布函数(CDF):

| 灰度值 | 像素数 | 概率 | CDF |
| --- | --- | --- | --- |
| 0 | 1 | 1/16 | 1/16 |
| 1 | 2 | 2/16 | 3/16 |
| 2 | 3 | 3/16 | 6/16 |
| 3 | 3 | 3/16 | 9/16 |
| 4 | 2 | 2/16 | 11/16 |
| 5 | 2 | 2/16 | 13/16 |
| 6 | 2 | 2/16 | 15/16 |
| 7 | 1 | 1/16 | 16/16 |

1. 归一化CDF到新的灰度范围(0-7):  
     
   其中，7为灰度值范围的最大值

| 原灰度值 | CDF | 归一化CDF | 新灰度值 |
| --- | --- | --- | --- |
| 0 | 1/16 | 0 | 0 |
| 1 | 3/16 | 0.875 | 1 |
| 2 | 6/16 | 2.1875 | 2 |
| 3 | 9/16 | 3.5 | 4 |
| 4 | 11/16 | 4.375 | 4 |
| 5 | 13/16 | 5.25 | 5 |
| 6 | 15/16 | 6.125 | 6 |
| 7 | 16/16 | 7 | 7 |

1. 使用新的灰度值映射原始像素:

原图像:

均衡化后的图像:

1. 结果分析:
   * 原图像的灰度值分布在0-7之间,但不均匀。
   * 均衡化后,灰度值仍然在0-7范围内,**但分布更加均匀**。
   * 一些相邻的原始灰度值(如3和4)被映射到了相同的新灰度值,是因为它们的累积概率相近。
   * 中间的灰度值(如2,3,4)被拉伸到更宽的范围(2,4),**增加了图像的对比度**。

#### 代码实现及说明

def hist\_equalization(img, jug):  
 """直方图均衡化算法"""  
 new\_img = np.zeros(img.shape, dtype=np.uint8)  
 time1 = time.time()  
 hist = creat\_histogram(img)  
  
 # 核心代码：直方图均衡化  
 if img.ndim == 2: # 灰度图像  
 cdf = np.cumsum(hist)  
 cdf\_normalized = (cdf - cdf.min()) \* 255 / (cdf.max() - cdf.min())  
 new\_img = cdf\_normalized[img].astype(np.uint8)  
 else: # 彩色图像  
 for i in range(3):  
 cdf = np.cumsum(hist[i])  
 cdf\_normalized = (cdf - cdf.min()) \* 255 / (cdf.max() - cdf.min())  
 new\_img[:,:,i] = cdf\_normalized[img[:,:,i]].astype(np.uint8)  
  
 time2 = time.time()  
 print(f"图像均衡算法程序处理时间：{(time2 - time1) \* 1000:.3f}毫秒")  
  
 if jug: # 绘制直方图  
 plot\_histogram(img, new\_img)  
 return new\_img  
  
def creat\_histogram(img):  
 """计算图像直方图"""  
 if img.ndim == 2: # 灰度图像  
 hist = np.bincount(img.ravel(), minlength=256)  
 elif img.ndim == 3: # 彩色图像  
 hist = [np.bincount(img[..., i].ravel(), minlength=256) for i in range(3)]  
 return hist

代码中：

* np.cumsum(hist):
  + 计算累积分布函数(CDF)。
  + 如果hist=[2,3,1],则cumsum结果为[2,5,6]。
* (cdf - cdf.min()) \* 255 / (cdf.max() - cdf.min()):
  + 归一化CDF到0-255范围。
  + 减去最小值并除以范围确保结果从0开始并充分利用整个灰度范围。
* cdf\_normalized[img]:
  + 使用归一化后的CDF作为查找表。
  + 原图像的每个像素值被用作索引,直接查找对应的新灰度值。
* 对于彩色图像,分别处理每个颜色通道。

上述方法利用了NumPy的向量化操作,处理速度还是很快的。直方图均衡化可以有效地增强图像对比度,特别是对于那些灰度值集中在某个狭窄范围内的图像。但是也可能导致图像细节丢失。

好的,我来帮您完善邻域平均滤波的实验过程及代码说明。

## 图像平滑

### 邻域平均滤波

#### 实现步骤

1. 获取用户定义的卷积核(滤波模板)
2. 对原始图像进行**边缘填充** （这里使用的是边缘复制填充）
3. 用**滑动窗口方法**获取每个像素的邻域
4. 对每个邻域应用卷积核(计算**加权平均**)
5. 将结果限制在0-255范围内并转换为整数

#### 具体计算过程

假设有一个5x5的灰度图像,使用3x3的均值滤波器:

原始图像:

均值滤波器(卷积核):

1. 边缘填充(这里使用边缘复制填充):
2. 对中心像素进行滤波计算，加权平均
3. 对所有像素重复此过程,得到结果:

#### 代码实现及说明

def neighbor\_average(img):  
 """邻域平均滤波"""  
 rows, cols = img.shape[:2]  
 new\_img = np.zeros(img.shape, dtype=np.uint8)  
   
 # 获取用户定义的卷积核  
 np\_kernel = get\_user\_kernel()  
 np\_kernel = np\_kernel / np\_kernel.sum() # 正则化  
  
 time1 = time.time()  
  
 # 核心代码：邻域平均平滑  
 padded\_img = np.pad(img, ((1, 1), (1, 1)), mode='edge')  
 windows = np.lib.stride\_tricks.sliding\_window\_view(padded\_img, (3, 3))  
 new\_img = np.sum(windows \* np\_kernel, axis=(2, 3))  
 new\_img = np.clip(new\_img, 0, 255).astype(np.uint8)  
  
 time2 = time.time()  
 print(f"邻域平均平滑程序处理时间：{(time2 - time1) \* 1000:.3f}毫秒")  
 return new\_img

代码中：

* np.pad(img, ((1, 1), (1, 1)), mode='edge'):
  + 对图像进行边缘填充,使用**边缘像素值**填充。
  + ((1, 1), (1, 1)) 表示在上下左右各填充1个像素。
* np.lib.stride\_tricks.sliding\_window\_view(padded\_img, (3, 3)):
  + 创建3x3的滑动窗口视图。
  + 这个操作不会创建新的数组,而是创建原数组的视图,因此实际上并没有创建新的内存，性能很高。
* np.sum(windows \* np\_kernel, axis=(2, 3)):
  + 对每个3x3窗口应用卷积核。
  + windows \* np\_kernel 进行元素级乘法。
  + np.sum(..., axis=(2, 3)) 对最后两个维度(3x3窗口)求和。

实现方法中利用了NumPy的高效数组操作，使用sliding\_window\_view创建窗口视图，就是使用一个窗口在图像上滑动，对每个窗口内的像素进行卷积操作。可以很大程度上减少计算量，提高效率。

邻域平均滤波可以有效地减少图像中的噪声,但也会导致图像细节的模糊。对于不同的应用场景,可能需要调整卷积核的大小和权重来平衡噪声消除和细节保留。邻域平均比较适用于平滑高斯噪声。

好的,我来帮您完善中值滤波的实验过程及代码说明。

### 中值滤波

#### 实现步骤

1. 定义**滤波窗口**大小(这里使用3×3)
2. 对原始图像进行边缘填充
3. 使用**滑动窗口方法**获取每个像素的邻域
4. 对每个邻域内的像素值进行**排序**,选取**中间值**
5. 将中值作为对应位置的新像素值

#### 具体计算过程

比如对中心像素(80)进行中值滤波计算:

排序后:

中值为:  
80

依次对所有像素进行上述操作，即对窗口内的所有像素进行排序，然后取中值。直到遍历完所有像素。

#### 代码实现及说明

def median\_filter(img):  
 """中值滤波"""  
 rows, cols = img.shape[:2] # 获取宽和高  
 new\_img = np.zeros(img.shape, dtype=np.uint8) # 新建同原图大小一致的空图像  
 len = 3 # 定义中值滤波模板3×3  
  
 time1 = time.time() # 程序计时开始  
  
 # 核心代码：中值滤波 numpy实现  
 padded\_img = np.pad(img, ((1, 1), (1, 1)), mode='edge')  
 windows = np.lib.stride\_tricks.sliding\_window\_view(padded\_img, (3, 3))  
 new\_img = np.median(windows, axis=(2, 3)).astype(np.uint8)  
  
 time2 = time.time() # 程序计时结束  
 print(f"中值滤波程序处理时间：{(time2 - time1) \* 1000:.3f}毫秒")  
 return new\_img

代码中：

* np.pad(img, ((1, 1), (1, 1)), mode='edge'):
  + 对图像进行边缘填充,使用边缘像素值填充。
  + ((1, 1), (1, 1)) 表示在上下左右各填充1个像素。
* np.lib.stride\_tricks.sliding\_window\_view(padded\_img, (3, 3)):
  + 创建3x3的滑动窗口视图。
  + 这个操作不会创建新的数组,而是创建原数组的视图,非常高效。
* np.median(windows, axis=(2, 3)):
  + 对每个3x3窗口计算中值。
  + axis=(2, 3) 表示在最后两个维度(3x3窗口)上计算中值。

仍然是使用sliding\_window\_view创建窗口视图,使得整个中值滤波操作可以一次性完成,非常高效。

在实际应用中,中值滤波特别适合用于去除**椒盐噪声**,同时保留图像边缘细节。

# 实验结果与分析

## 灰度变换

灰度变换部分都是采用的numpy特性根据公式进行计算，处理时间都是比较短的。  
各个变换处理时间：

| 变换类型 | 处理时间/ms |
| --- | --- |
| 线性变换 | 3-5 |
| 对数变换 | 11-14 |
| 指数变换 | 11-13 |
| 幂律变换 | 10-12 |

处理效果图示例：

## 图像修正

### 用直方图均衡化修正图像

处理图像分别是图像1和图像2，处理效果图示例：

处理时间：

| 图像 | 直方图创建方法 | 处理时间/ms |
| --- | --- | --- |
| 图像1 | 遍历 | 18.9-20.1 |
|  | numpy | 0.0-1.1 |
|  | opencv | 0.0-1.0 |
| 图像2 | 遍历 | 102.0-106.0 |
|  | numpy | 2.8-4.2 |
|  | opencv | 1.9-3 |

## 图像平滑

### 邻域平均滤波

处理图像为如下高斯噪声图像：

| 处理方式 | 处理时间/ms |
| --- | --- |
| numpy | 3.8-4.0 |
| opencv | 0.0-0.05 |

处理效果：

卷积核为 均值卷积核：

卷积核为 高斯卷积核：

### 中值滤波

处理图像为如下椒盐噪声图像：

| 处理方式 | 处理时间/ms |
| --- | --- |
| numpy | 1.0-5.5 |
| opencv | 0.0-1.0 |

处理效果：

好的,我来帮您总结一下实验心得及体会,主要围绕NumPy的高效实现和图像处理原理的结合。

# 实验总结及心得体会

一、更加地了解numpy对于实现图像处理中一些高效算法的强大。

* **向量化操作**  
  NumPy的向量化操作允许对整个数组进行操作,比如在线性拉伸的时候,只需一行代码 k \* (img - a) + c 就能完成整个图像的变换,不仅使代码简洁,而且大大提高了运行效率。
* **广播机制** （基本每次实验都会用到numpy的广播，因为这在提升效率上真的很有用）  
  NumPy的广播机制可以使不同形状的数组进行运算,我体会到这在邻域平均滤波中非常有用,我可以直接用卷积核与图像窗口相乘,而不需要手动对齐尺寸。
* **数组视图**:  
  使用 **np.lib.stride\_tricks.sliding\_window\_view** 创建滑动窗口视图。这个函数可以在不创建新数组的情况下获取所有的图像窗口,大大减少了内存使用并提高了处理速度。发现了新大陆，因为这个函数在卷积操作中可以极大提高效率。

二、深入理解图像处理的原理和应用

* **灰度变换的本质**:  
  通过实现线性、对数、指数和幂律变换,更深入地理解了灰度变换的本质是像素值的重新映射。每种变换都有其特定的用途。  
  线性变换可以调整图像的对比度和亮度。比如将一张暗淡的图片的灰度范围从[50, 180]拉伸到[0, 255]时,图像的对比度得到提升。对数变换对于增强图像中的暗部细节比较有效。也就是增强低灰度值，压缩高灰度值。指数变换则强调亮部细节。幂律变换(伽马校正)可以根据不同的伽马值调整图像的整体亮度。
* **直方图均衡化**:  
  直方图的计算采用numpy的**bincount**可以达到很快的速度，本质上都是对数组的处理。  
  另外均衡化中，求解累积分布函数，采用numpy的**cumsum**可以达到很快的速度。cumsum是利用了前缀和的性质，大大提高了效率。
* **滤波器的选择**:  
  不同滤波器有不同的特点和适用场景:
  1. 邻域平均滤波适合处理高斯噪声，但可能导致图像边缘的模糊。
  2. 中值滤波适合去除椒盐噪声。并且可以很好的保留边缘细节。
  3. 另外我**尝试了不同大小的滤波窗口**。发现较大的窗口(5x5)可以更好地去除噪声,但也会导致更多的细节丢失。