数字图像处理实验报告

频域的高通和低通滤波、噪声与图像的空域滤波重建

院系：电子信息与通信学院
班级：信卓 1901 班
姓名：段康晟
学号：U201913827
指导老师：程起敏
实验日期：2022.3.30

1 实验目的

1、对输入的原始图像分别做理想、巴特沃斯、高斯低通滤波及高通滤波处理，对比实验效果。
2、对输入的原始图像叠加不同类型的噪声，对比不同的空间滤波方法的图像复原效果。

2 实验方法

本次实验的内容是基于 python 实现的。开发环境为 miniconda3+vscode，使用到了可以进行图像处理的 opencv-python 库和提供丰富的矩阵处理方法的 nunpy 库。

2.1 频域滤波器

2.1.1 傅里叶变换和逆傅里叶变换

参考 opencv-python 官方提供的文档：https://opencv-python-tutorials.readthedocs.io/zh/latest/4.%20OpenCV 中的图像处理/4.11.1%20 傅里叶变换/。具体代码实现见附录。

大体思路是用 cv.dft 得到图片的频域矩阵，而后用 np.fft.fftshift 对矩阵元素进行循环平移，来把低频和直流原点移动到矩阵的中央部分，而高频被移动到边缘部分。

傅里叶反变换则先调用 np.fft.ifftshift 将频率的直流原点平移回矩阵的索引原点处，而后调用 cv.idft 将频谱图转换回时域图像。

需要注意的是，将时域图片转换到频域后，若用 numpy 的傅里叶变换函数会变成两个通道，而是用 opencv 的 dft 则会变成和单通道的复数矩阵。在转换回时域图片之后，由于精度的损失，转换回来的时域图仍为复数矩阵，需要调用一次 cv.magnitude 将其降维为单通道、和原图同深度的实矩阵。

2.1.2 低通滤波器

理想低通滤波器：调用 cv2.circle，在频域矩阵上生成一个实心圆的掩膜，圆内填充 1，圆外填充 0，即课生成频域的理想低通滤波器的频域矩阵；

巴特沃斯低通滤波器：利用 np.array 创建行和列的距离向量，然后用 L2 范数得到在两个维度上长成的距离矩阵（注：记录到矩阵中心点的距离）。然后根据巴特沃斯低通滤波器的频域响应函数 1/(1+np.power(Duv/D0,2*n))，得到其滤波器的频域矩阵；

高斯低通滤波器：类似巴特沃斯低通滤波器，将频域响应函数替换成高斯函数的频域响应 np.exp(-(Duv*Duv)/(2*D0*D0))即可。

2.1.3 高通滤波器

理想高通滤波器：类似理想低通滤波器，将圆内填充 0，圆外填充 1 即可；

巴特沃斯高通滤波器：类似巴特沃斯低通滤波器，将频域响应函数替换为高通的函数 1/(1+np.power(D0/Duv,2*n))即可；

高斯高通滤波器：类似高斯低通滤波器，将频域响应函数替换成高斯高通的频域函数 1-np.exp(-(Duv**2)/(2*D0**2))即可。

2.2 添加噪声

利用 numpy 提供的 np.random.normal、rayleigh、gamma、exponential 和 uniform 函数来生成高斯噪声、瑞利噪声、伽马噪声、指数噪声和均匀噪声的矩阵，然后叠加到图像矩阵上并作灰度归一化即可；

椒盐噪声通过 random.randint 随机选择坐标和撒上噪声的种类（盐或胡椒），然后给选中的坐标直接赋值 0 或 255 即可。

2.3 空域复原滤波器

2.3.1 均值滤波器组

算数均值滤波：利用 np.ones 生成对应参数的归一化滤波核，然后调用 cv2.filter2D 将图片经过滤波器，再将滤波后的图片做 8bit 的量化即可；

几何均值滤波：以核尺寸的一半对原图片的四周做边缘扩展，以核中心锚点的位置做二重遍历，对核锚框内的矩阵元素求几何均值（求积开根），而后量化到 8bit 即可；

谐波滤波：类似几何均值滤波，将核锚框内的运算替换成谐波滤波核的滤波核函数(kernel_size*kernel_size)/((1.0/(img_pad[i-pad:i+pad+1,j-pad:j+pad+1]+eps))+eps)即可；

反谐波滤波：类似谐波滤波，替换核函数为反谐波滤波的和函数即可。

2.3.2 统计排序滤波器组

中值滤波：以核尺寸的一半对原图片的四周做边缘扩展，以核中心锚点的位置做二重遍历，在核锚框内的矩阵上，调用 np.median 得到框内中值，而后量化到 8bit 即可；

最大（小）值滤波：类似中值滤波，替换 np.median 为 np.max(或 np.min)即可；

中点滤波：类似中值滤波，替换 np.median 为(np.min()+np.max())/2 即可；

修正阿尔法滤波：类似中值滤波，在二重遍历内，先用 np.flatten 将框内矩阵展平，并进行排序 np.sort，而后对排序后的向量进行头尾 d 个元素的截断，再求算数均值。

2.3.3 自适应滤波器

自适应局部均值滤波：首先调用 cv2.meanStdDev 对原始图片中噪声的均值和方差做估计，然后以核尺寸的一半对原图片的四周做边缘扩展，以核中心锚点的位置做二重遍历，在核锚框内的矩阵上，求出局部灰度均值和方差，而后根据加性噪声假设，写出自适应滤波器的核函数 g_xy-residual*(g_xy-z_Sxy)即可。

以上所有实现的完整代码见附录部分。

3 实验结果

注：待处理的原图见附录。

3.1 实验一：频域滤波器

3.1.1 低通滤波器

图 1 理想低通滤波

如图 1 左侧为经过中间低通理想滤波器后得到的图像。图中中间为该理想低通滤波器的频谱图，右侧为经过该滤波器后得到的图像频谱图。

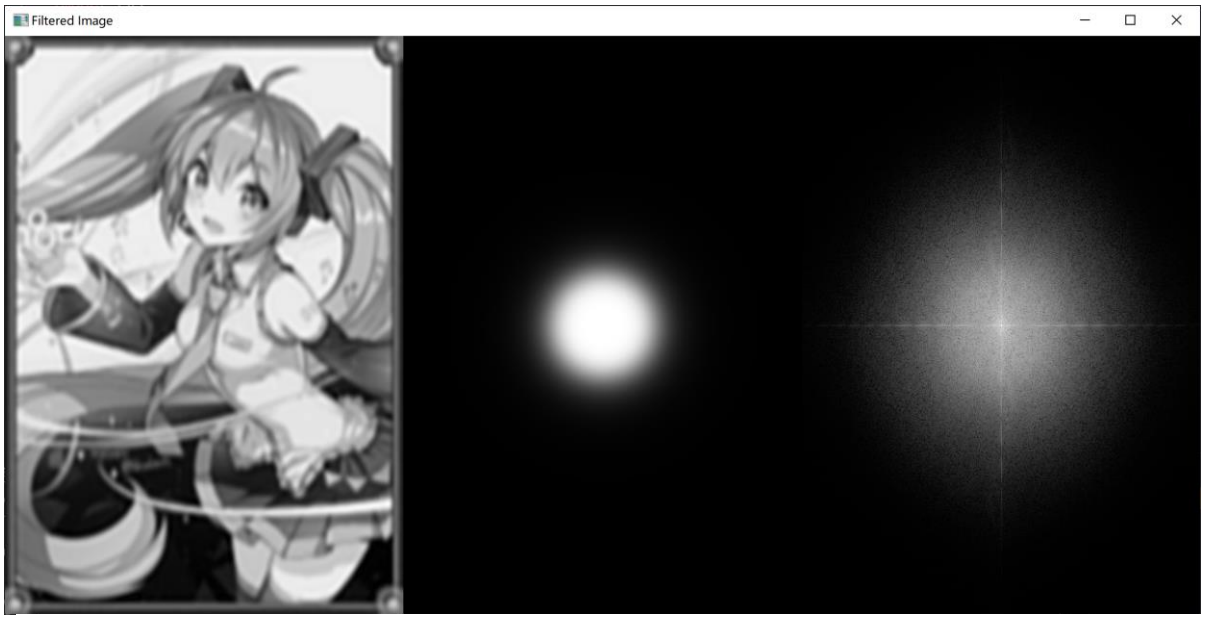

图 2 巴特沃斯低通滤波

如图 2 所示为经过巴特沃斯滤波器低通滤波器后得到的图像。对比图 1 中间的闾巷低通滤波核可以看到，巴特沃斯滤波核的边沿滚降部分存在滚降，而理想滤波器的边缘部分是陡降的。

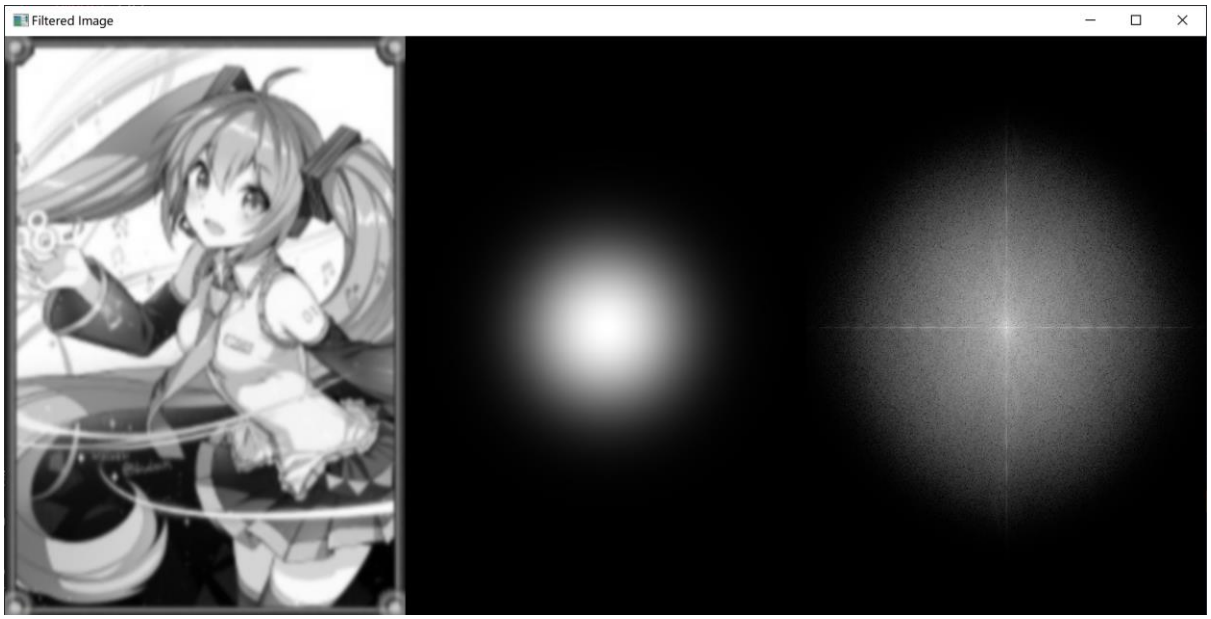

图 3 高斯低通滤波

如图 3 所示为经过高斯低通滤波器得到的图像。对比图 2 可以发现滤波器的边缘滚降更为平缓。

3.1.2 高通滤波器

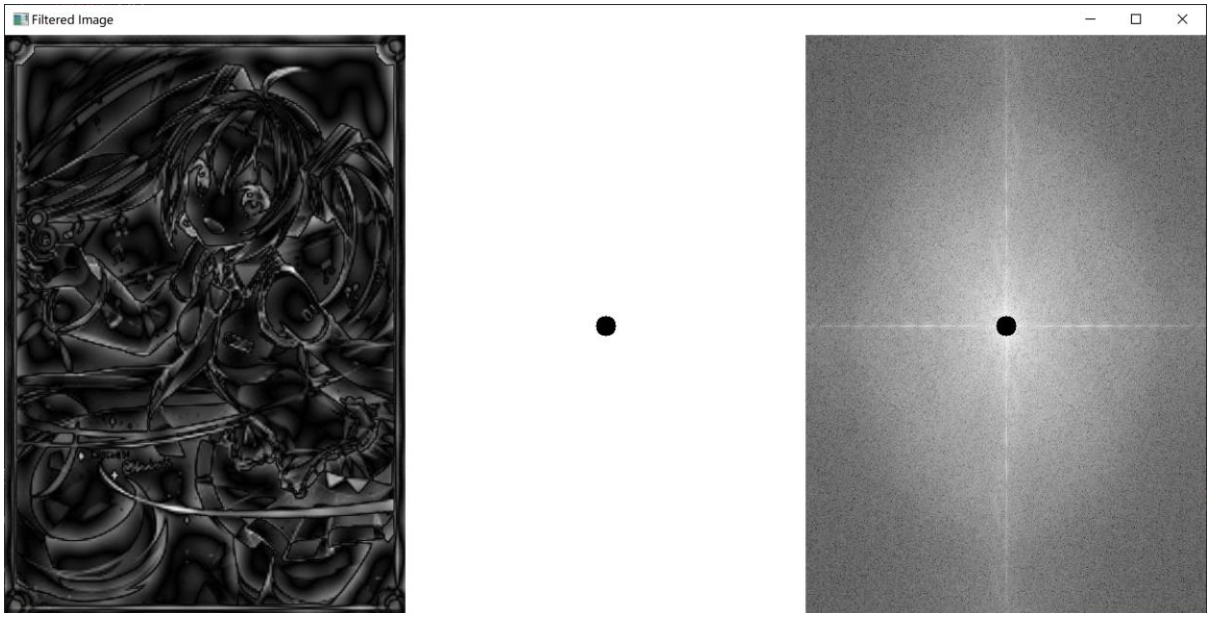

图 4 理想高通滤波

如图 4 所示为理想高通滤波。可以看到，仅仅只是滤去图像中的直流成分和极低频成分后，图像的观感上就发生了极大的变化，可故以认为原图中有很大一部分的灰度信息都在直流和极低频的频率成分里。

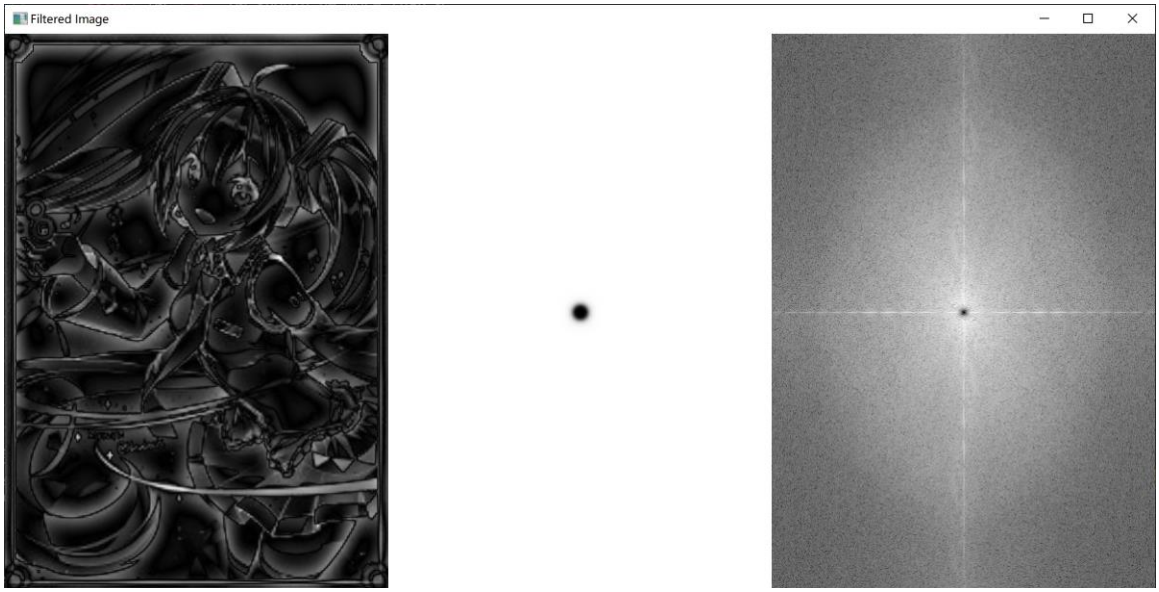

图 5 巴特沃斯高通滤波

如图 5 所示为经过巴特沃斯高通滤波器后得到的图像。对比图 4，二者在视觉观感上没有很大差别。

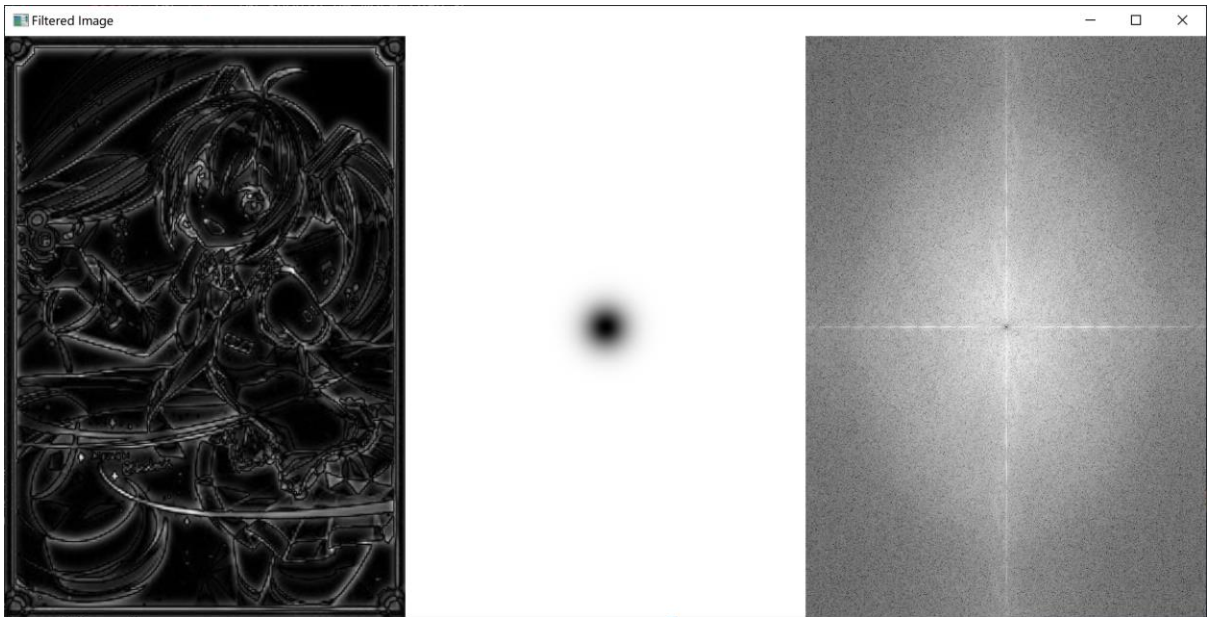

图 6 高斯高通滤波

如图 6 为经过高斯高通滤波处理后得到图像。比较中间的滤波器频谱可以看到理想滤波器的边缘是陡降的，而高斯和巴特沃斯的边缘都是有滚降现象。而随着巴特沃斯的阶数的增加，其边缘也越陡。

3.2 实验二：空域噪声与图像复原

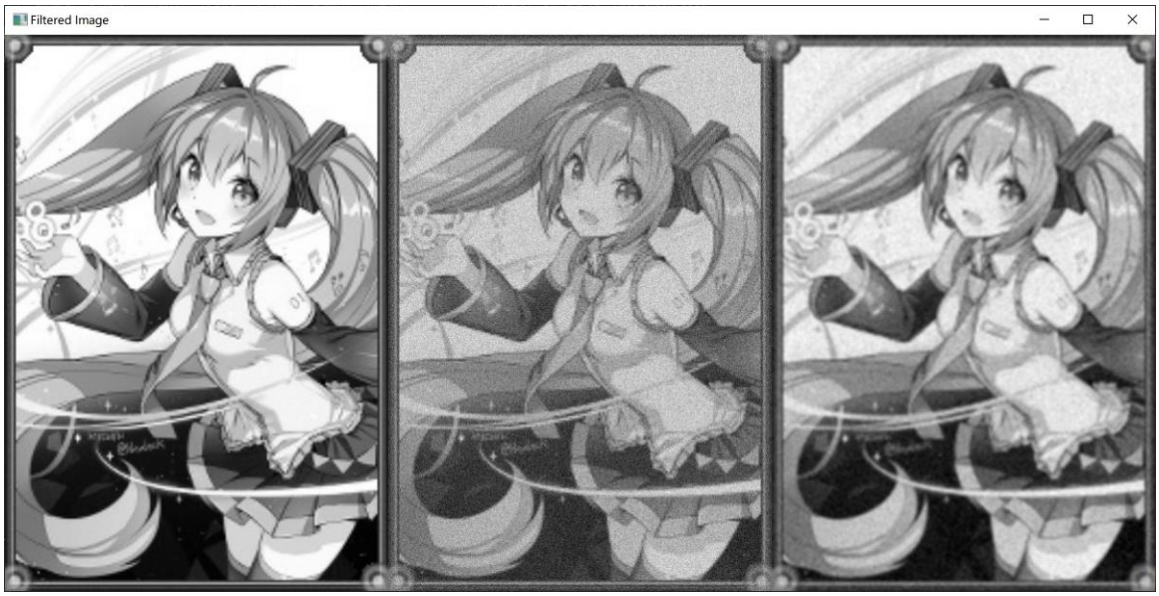

图 7 高斯噪声和算术均值滤波

如图 7 所示，左图为原始图像，中间为添加了均值 6、标准差 20 的高斯噪声后的退化图像，右图为经过算数均值滤波后得到的复原图像。可以看出算数均值滤波器一定程度上去除了噪声，复原了一部分原图的信息，但它也损失了一部分高频细节，从而使复原后图像相对原图而言变得较为模糊。

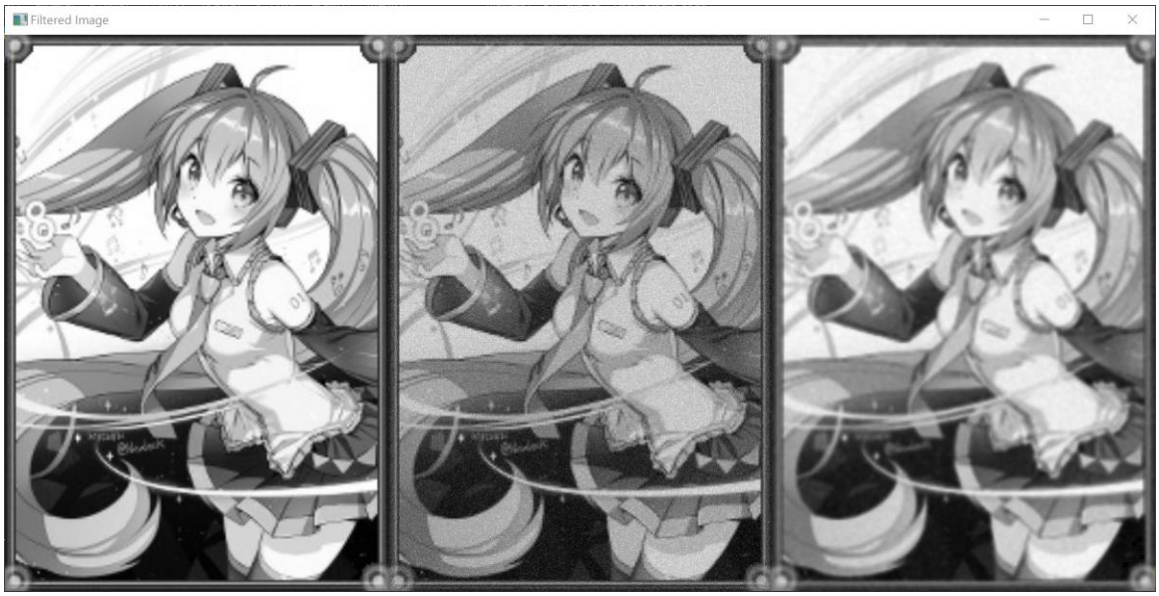

图 8 瑞利噪声和几何均值滤波

如图 8 所示为经过 a=30 的瑞利噪声退化后的图像和经过几何均值滤波后复原的图像。几何均值滤波器的滤波核大小是 3*3，但它已经能够很明显地产生模糊了。另一方面，这个小的几何均值滤波核在我的电脑上需要运算的时间比算数均值要长不少，但其对瑞利噪声复原的效果，我认为不是很能接受。

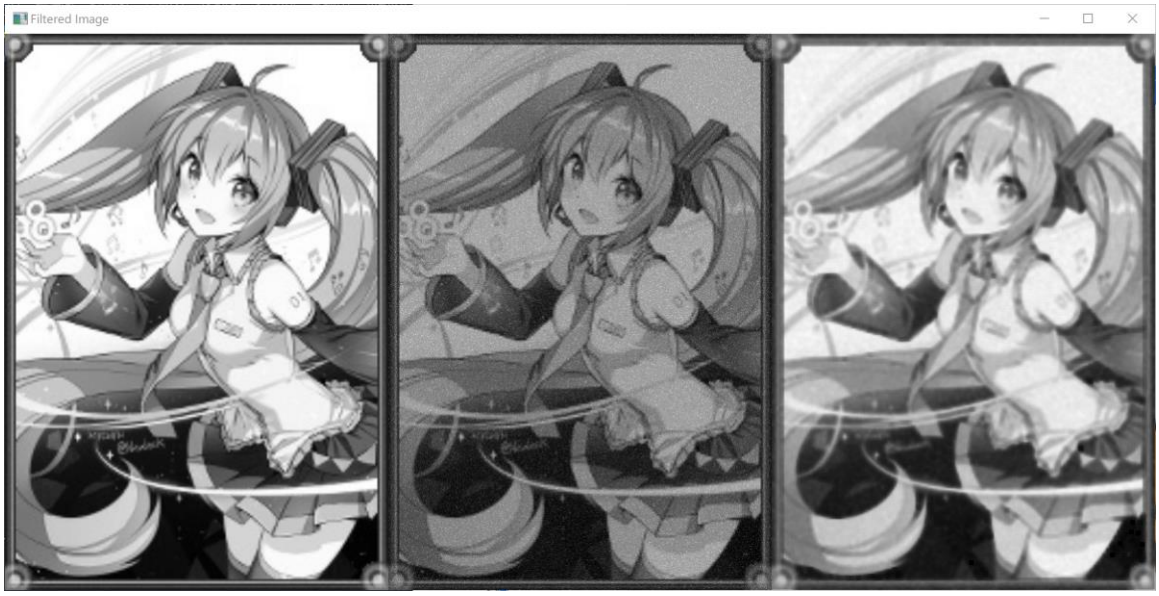

图 9 Gamma 噪声和谐波均值滤波

如图 9 为经过 a=10,b=2.5 的 Gamma 退化后的图像，和经过滤波核大小为 3*3 的谐波均值滤波器复原的图像。大体上看，谐波均值滤波还是达到了去噪的效果的，但它也会产生模糊的失真，而且它计算起来也很慢。

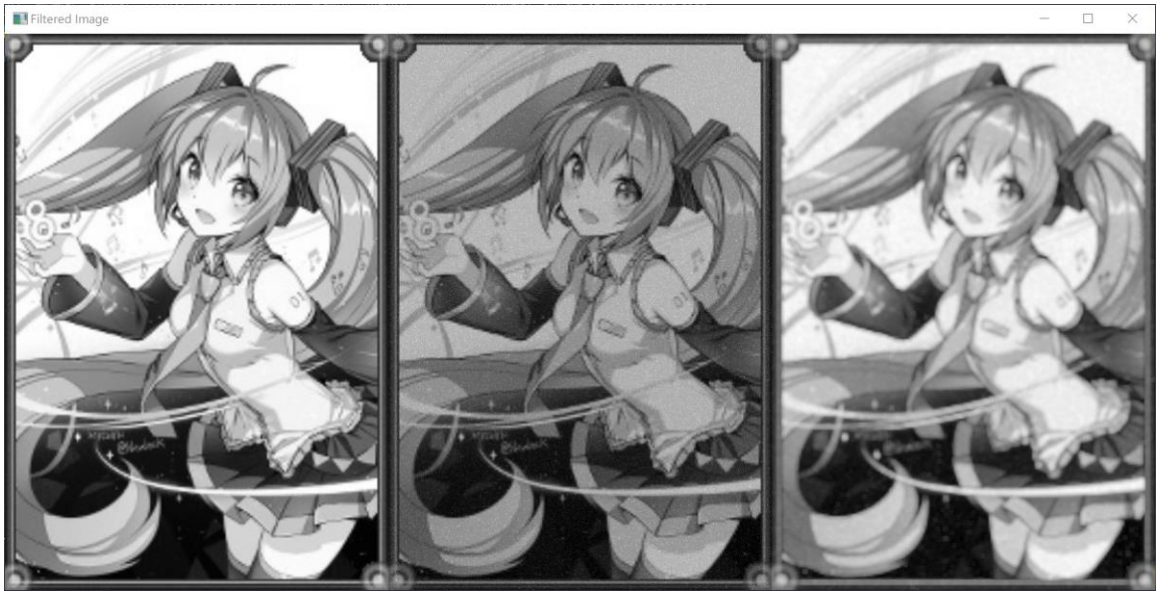

图 10 指数噪声和反谐波均值滤波

如图 10 为经过 a=10 的指数噪声退化后的图像，和经过 Q=1.5 的反谐波均值滤波器复原后得到的图像，其滤波核大小为 3*3。个人观察来说和图 9 基本看不出有什么区别（但确实是两种不同的噪声和两种不同的复原）。不同地，反谐波滤波器算起来比谐波均值滤波器更慢。

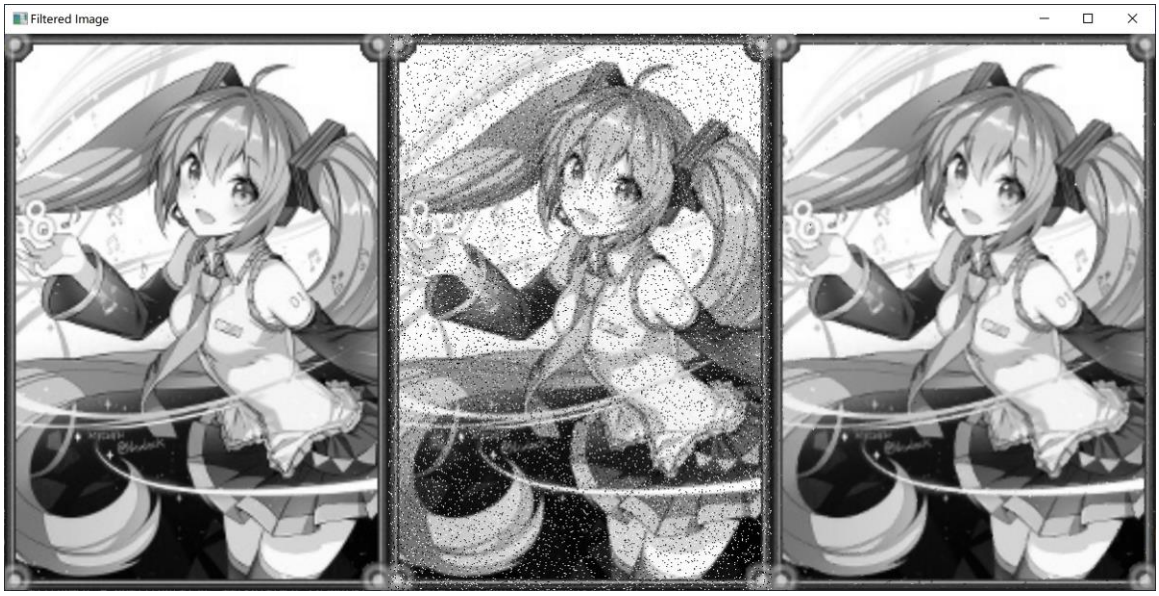

图 11 椒盐噪声和中值滤波

如图 11 为经过 proportion=10%椒盐噪声退化后的图像，和经过 3*3 的中值滤波后得到的复原图像。可以直观地看出，椒盐噪声用中值滤波器处理是非常合适的，中值滤波核对胡椒和盐粒的极端值的成分不敏感，正适合去除。

图 12 均匀噪声和自适应局部降噪滤波

如图 12 所示为经过 a=10,b=60 的均匀噪声退化后的图像，和经过 3*3 的局部自适应降噪滤波器复原后的图像。可以看出自适应滤波器的复原效果还是可接受的，其模糊失真程度也没有几个均值滤波那么严重。但是自适应局部降噪滤波器的计算复杂度也有点大，处理起来的时间较长。

4 附录

4.1 部分代码实现

完整代码见：https://github.com/AnduinD/DIP-Experiment

4.1.1 实验一：频率域滤波中的几个滤波器的实现

```python
def lpfilter(flag, rows, cols, D0, n):
    '''低通滤波器
    @param flag: 滤波器类型
    @param rows: 被滤波的矩阵高度
    @param cols: 被滤波的矩阵宽度
    @param D0: 滤波器大小 D0
    @param n: 巴特沃斯阶数
    @return 滤波器矩阵
    '''
    assert D0 > 0, 'D0 should be more than 0.' #断言滤波核规模
    filterMat = None # 生成滤波核对象矩阵
    if flag == IDEAL_FILTER:  # 理想低通滤波
        filterMat = np.zeros((rows, cols), np.uint8)
        cv2.circle(filterMat,(int(cols/2), int(rows/2)) ,D0, (1, 1, 1), thickness=-1)
    elif flag == BUTTERWORTH_FILTER: # 巴特沃斯低通
        Duv = fft_distances(rows,cols)
        filterMat = 1/(1+np.power(Duv/D0, 2*n))
    elif flag == GAUSSIAN_FILTER: # 高斯低通
        Duv = fft_distances(rows,cols)
        filterMat = np.exp(-(Duv*Duv)/(2*D0*D0))
    filterMat = cv2.merge((filterMat, filterMat))# （频域有实部和虚部）
    return filterMat

def hpfilter(flag, rows, cols, D0, n):
    '''高通滤波器
    @param flag: 滤波器类型
    @param rows: 被滤波的矩阵高度
    @param cols: 被滤波的矩阵宽度
    @param D0: 滤波器大小 D0
    @param n: 巴特沃斯阶数
    @return 滤波器矩阵
    '''
    assert D0 > 0, 'D0 should be more than 0.'
    filterMat = None
    if flag == IDEAL_FILTER:
        filterMat = np.ones((rows, cols), np.uint8)
        cv2.circle(filterMat, (int(cols/2), int(rows/2)),D0, (0, 0, 0), thickness=-1)
    elif flag == BUTTERWORTH_FILTER:
        Duv = fft_distances(rows, cols)
        Duv[int(rows/2),int(cols/2)] = 0.000001
# Duv 有 0 值(中心距离中心为0)， 为避免 0 到分母上了，设中心为 0.000001
        filterMat = 1/(1 + np.power(D0/Duv, 2*n))
    elif flag == GAUSSIAN_FILTER:
        Duv = fft_distances(*fftShiftMat.shape[:2])
        filterMat = 1-np.exp(-(Duv**2)/(2*D0**2))
    filterMat = cv2.merge((filterMat, filterMat))
    return filterMat
```

4.1.2 实验二：几种噪声的生成和空域复原滤波器的实现

```python
## 添加各种噪声的函数
def gaussian_noise(img, mean=10, sigma=30):
    '''
    添加高斯噪声
    @param img: 原图
    @param mean: 均值
    @param sigma: 标准差
    @return gaussian_out: 噪声处理后的图片
    '''
    noise = np.random.normal(mean, sigma, img.shape)# 产生高斯 noise
    gaussian_out = img + noise# 将噪声和图片叠加
    return  depth_quantize(gaussian_out)# 将图片灰度范围的恢复为 0-255

def rayleigh_noise(img,a=20):
    '''添加瑞利噪声'''
    noise = np.random.rayleigh(a, size=img.shape)
    rayleigh_out = img + noise
    return depth_quantize(rayleigh_out)

def gamma_noise(img,a=10,b=2.5):
    '''添加 Gamma 噪声'''
    assert a>b,"a should be larger than b"
    noise = np.random.gamma(shape=b, scale=a, size=img.shape)
    gm_out = img + noise
    return depth_quantize(gm_out)

def exponent_noise(img,a=10):
    '''添加指数噪声'''
    noise = np.random.exponential(scale=a, size=img.shape)
    exp_out = img + noise
    return depth_quantize(exp_out)

def uniform_noise(img,a=10,b=150):
    '''
    添加均匀噪声
    @param image: 需要加噪的图片
    @param a: 均匀噪声直方图上的取值下限
    @param b: 均匀噪声直方图上的取值上限
    @return uf_out
    '''
    assert a<b,"a should be smaller than b"
    noise = np.random.uniform(low=a,high=b,size=img.shape) #生成均匀噪声矩阵
    uf_out = img+noise #叠加噪声
    return depth_quantize(uf_out) #灰度量化

def salt_pepper_noise(img, proportion=0.1):
    '''
    添加椒盐噪声
    @param proportion: 加入噪声的百分比
    @return img with salt and pepper
    '''
    sp_out = img
    rows, cols = sp_out.shape#获取高度宽度像素值
    num = int(rows*cols*proportion) #一个准备加入多少噪声小点
    for i in range(num):
        c = random.randint(0, cols - 1)
        r = random.randint(0, rows - 1)
        if random.randint(0, 1) == 0:  sp_out[r, c] = 0  #撒胡椒
        else:                          sp_out[r, c] = 255 # 撒盐
    return sp_out

```

```python
# 均值滤波器系
def arithmetic_mean_filter(img, kernel_size):
    '''算术均值滤波器'''
    img_out = np.zeros(img.shape)#初始化对象
    kernel_mask = np.ones((kernel_size,kernel_size),np.float32)
                          /(kernel_size*kernel_size)  # 生成归一化盒式核
    img_out = cv2.filter2D(img, -1, kernel_mask) # 均值滤波
    return depth_quantize(img_out) #归一化和量化

def geometric_mean_filter(img, kernel_size):
    '''几何均值滤波器'''
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size/2)
    img_pad = np.pad(img.copy(),
                     ((pad,kernel_size-pad-1),(pad,kernel_size-pad-1)),
                     mode="edge").astype(np.float64) # 原图的边缘填充
    eps = 1e-8
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            prod = np.prod(img_pad[i-pad:i+pad+1, j-pad:j+pad+1]+eps)  #求核内乘积
            img_out[i-pad][j-pad] = np.power(prod,1/(kernel_size*kernel_size)) #
    return depth_quantize(img_out)

def harmonic_mean_filter(img, kernel_size):
    '''谐波均值滤波器'''
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size/2)
    img_pad = np.pad(img.copy(),
                     ((pad,kernel_size-pad-1),(pad,kernel_size-pad-1)),
                     mode="edge").astype(np.float64)
    eps = 1e-8
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            kernel_temp = (1.0/(img_pad[i-pad:i+pad+1, j-pad:j+pad+1]+eps))
            kernel_sum = kernel_temp.sum()
            img_out[i-pad][j-pad] = (kernel_size*kernel_size)/(kernel_sum+eps)
    return depth_quantize(img_out)

def inv_harmonic_mean_filter(img, kernel_size, Q = 1.5):
    '''逆谐波均值滤波器'''
    Q = cv2.getTrackbarPos('Q(inv harmonic)', filterWin)
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size / 2)
    img_pad = np.pad(img.copy(),
                     ((pad, kernel_size-pad-1), (pad, kernel_size-pad-1)),
                     mode="edge").astype(np.float64)
    eps = 1e-8 # 一个很小的值，防止0在分母上
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            kernel_temp = img_pad[i-pad:i+pad+1, j-pad:j+pad+1]+eps # 中间量
            numerator = np.sum(np.power(kernel_temp,Q+1)) #分子
            denominator = np.sum(np.power(kernel_temp,Q)) #分母
            img_out[i-pad][j-pad] = numerator/(denominator+eps)
    return depth_quantize(img_out)

# 统计排序滤波器系
def median_filter(img, kernel_size):
    '''中值滤波器'''
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size / 2)
```

```
    img_pad = np.pad(img.copy(),
                     ((pad, kernel_size-pad-1), (pad, kernel_size-pad-1)), mode="edge")
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            img_out[i-pad][j-pad] = np.median(img_pad[i-pad:i+pad+1, j-pad:j+pad+1])
    return depth_quantize(img_out)

def max_filter(img, kernel_size):
    '''最大值滤波器'''
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size / 2)
    img_pad = np.pad(img.copy(),
                     ((pad, kernel_size-pad-1), (pad, kernel_size-pad-1)), mode="edge")
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            img_out[i-pad][j-pad] = np.max(img_pad[i-pad:i+pad+1, j-pad:j+pad+1])
    return depth_quantize(img_out)

def min_filter(img, kernel_size):
    '''最小值滤波器'''
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size / 2)
    img_pad = np.pad(img.copy(),
                     ((pad, kernel_size-pad-1), (pad, kernel_size-pad-1)), mode="edge")
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            img_out[i-pad][j-pad] = np.min(img_pad[i-pad:i+pad+1, j-pad:j+pad+1])
    return depth_quantize(img_out)

def mid_filter(img, kernel_size):
    '''中点值滤波器'''
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size/2)
    img_pad = np.pad(img.copy(),
                     ((pad, kernel_size-pad-1), (pad, kernel_size-pad-1)),
                     mode="edge").astype(np.float64)
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            kernel_temp = img_pad[i-pad:i+pad+1, j-pad:j+pad+1]
            img_out[i-pad][j-pad] = (kernel_temp.min()+kernel_temp.max())/2
    return depth_quantize(img_out)

def cor_alpha_filter(img,kernel_size,d=2):
    '''修正阿尔法均值滤波器''' # 统计排序和算数均值的结合
    d = cv2.getTrackbarPos('d(cor alpha)', filterWin)
    kernel_size = max(int(np.sqrt(2*d+1)),kernel_size)
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size/2)
    img_pad = np.pad(img.copy(),
                     ((pad, kernel_size-pad-1), (pad, kernel_size-pad-1)), mode="edge")
    eps = 1e-8
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            kernel_temp = img_pad[i-pad:i+pad+1, j-pad:j+pad+1]
            kernel_temp = np.sort(kernel_temp.flatten())  # 对邻域像素按灰度值排序
            kernel_sum = np.sum(kernel_temp[d:-d-1]) # 删除 d 个最大灰度值,d 个最小灰度值
            img_out[i-pad][j-pad] =
                   kernel_sum/(kernel_size*kernel_size-2*d+eps)  # 对剩余像素进行算术平均
    return depth_quantize(img_out)


```

```python
# 自适应滤波器系
def adaptive_local_filter(img,kernel_size):
    '''自适应局部降噪滤波器'''
    img_out = np.zeros(img.shape)
    rows,cols = img.shape
    pad = int(kernel_size/2)
    img_pad = np.pad(img.copy(),
                     ((pad, kernel_size-pad-1), (pad, kernel_size-pad-1)), mode="edge")
    # 估计原始图像的噪声方差
    noise_mean, noise_stddev = cv2.meanStdDev(img)
    noise_var = noise_stddev ** 2 # 标准差转成方差
    # 自适应局部降噪
    eps = 1e-8
    for i in range(pad,rows+pad):
        for j in range(pad,cols+pad):
            kernel_temp = img_pad[i-pad:i+pad+1, j-pad:j+pad+1]
            g_xy = img_pad[i,j]  # 含噪声图像的原本像素点
            z_Sxy = np.mean(kernel_temp)  # 局部平均灰度
            kernel_var = np.var(kernel_temp)  # 灰度的局部方差
            residual =
                min(noise_var/(kernel_var+eps),1.0) # 加性噪声假设：sigma_eta<=sigma_Sxy
            img_out[i-pad][j-pad] = g_xy - residual * (g_xy - z_Sxy)
    return depth_quantize(img_out)
```

4.2 待处理的原始图像

图 13 miku 原图和她的 8bit 灰度图