

第三次作业

09021220 周天逸

一、Introduction

1. 基于二阶导数（建议用卷积操作完成）
2. 非锐化掩蔽(Unsharp Masking)（分析不同参数的选择对结果的影响）两种方法的图像增强。

二、Relative Work

1. 二阶导数的图像锐化

1. 锐化作用是突出灰度中的过渡。锐化能够通过空间微分来实现。导数算子的响应强度与应用该算子的点处的灰度不连续幅度成正比。因此，图像微分将增强边缘和其他不连续(如噪声)，并且不强调灰度缓慢变化的区域。
2. 二阶导数在图像处理中的原理
 1. 恒定灰度区域的二阶导数必须为零
 2. 灰度台阶或斜坡的开始处和结束处的二阶导数必须非零
 3. 灰度斜坡上的二阶导数必须为零
3. 二阶导数计算公式
 1. $\frac{\partial^2 f}{\partial^2 x} = f'(x) - f'(x-1) = f(x+1) + f(x-1) - 2f(x)$.在图像处理中一般是二维函数用偏导，其表达式也是一样的。
 2. 通常使用偏导模板表示图像的偏导计算，习惯上使用Laplacian算子来表示。拉普拉斯是导数算子，因此会突出图像中的急剧灰度过渡，并且不强调缓慢变化的灰度区域。这往往会产生具有灰色边缘线和其他不连续性的图像，它们都叠加在暗色无特征背景上。将拉普拉斯图像与原图像相加，就可“恢复”背景特征，同时保留拉普拉斯的锐化效果。
4. 计算图像的锐化
 1. 习惯上使用Laplacian算子，增强的表达式写为
$$\nabla^2 f = f(x+1, y) + f(x-1, y) + f(x, y-1) + f(x, y+1) - 4f(x)$$
 2. $g(x, y) = f(x, y) + \nabla^2 f$.
 3. 对于二阶偏导的模板中，还可以增加对角线方向的偏量。这种模板在突出图像边缘中效果更好，后续可以看第四部分实验结果中的展示

2. Unsharp Masking非锐化掩蔽

1. 原理

1. 将原图像通过反锐化掩模进行模糊预处理(相当于采用低通滤波)后与原图逐点做差值运算，然后乘上一个修正因子再与原图求和，以达到提高图像中高频成分、增强图像轮廓的目的。

2. 基本步骤

1. 平滑原图像：f→s。本实验使用的均值的滤波器，系数为1/9，通过卷积操作，模仿均值滤波器的效果获得平滑后的图像
2. 从原图像中减去模糊图像，产生的差值图像称为模板：m=f-s.由此可以求出图像的高频部分。通过原图像-通过均值滤波器后的低频部分图像。
3. 将模板乘以一个k 加到原图像中。两重for循环对应点相加即可。

3. 特点

1. 系数k越大对细节增强越明显.实验部分有比较不同的系数k的效果。
2. 平滑减弱的边缘，锐化后增强的更加明显。

三、Experiment

1. 二阶导数的图像锐化

1. Laplacian卷积模板

```
cv::Mat kernel = (Mat_<float>(3, 3) <<
    1, 1, 1,
    1, -8, 1,
    1, 1, 1);
kernel = (Mat_<float>(3, 3) <<
    0, -1, 0,
    -1, 4, -1,
    0, -1, 0);
```

本次实验对比了不同卷积模板对于图像边缘增强，锐化处理的差别。所以使用两种模板

2. 模板卷积操作

1. 通过调用函数 2. 获取函数卷积后的部分 3. 卷积函数

```
Mat MyConv(const Mat& inputImage, const Mat& kernel)
{
    Mat outputImage = Mat::zeros(inputImage.size(), inputImage.type());

    int kernelSize = kernel.rows;
    int border = kernelSize / 2;

    for (int y = border; y < inputImage.rows - border; y++) {
        for (int x = border; x < inputImage.cols - border; x++) {
            int sum = 0;
            for (int i = -border; i <= border; i++) {
                for (int j = -border; j <= border; j++) {
                    sum += inputImage.at<uchar>(y + i, x + j) *
kernel.at<float>(i + border, j + border); //卷积
                }
            }
            outputImage.at<uchar>(y, x) = saturate_cast<uchar>(sum);
        }
    }

    return outputImage;
}
```

卷积函数就是通过移动卷积中心不断移动，并且在周围3*3范围内于卷积模板相乘。3. 获取结果 1. 由前面的理论。 $g(x, y) = f(x, y) + \nabla^2 f$ 按照不同的卷积模板的系数决定最后的结果是加还是减。 4. 注意事项： 1. 每个核的系数之和为零。基于卷积的滤波实现乘积之和，因此当导数核通过图像中的一个恒定区域时，这个位置的卷积结果必为零。使用系数之和为零的核可以实现这一目的。 5. 结果展示 进行对角处理后，可以明显看到月球的突起，坑洼部分被加强了，这就是锐化的效果。最右边的图片就是进行Laplacian模板处理后的图像的高频部分，可以看到主要突出的是图像的边缘部分。

2. Unsharp Masking

1. 首先还是定义卷积模板，不同于Laplacian，这次的模板是均值滤波器，要获取到图像的低频部分。

```
cv::Mat kernel = (Mat_<float>(3, 3) <<
    1, 1, 1,
    1, 1, 1,
    1, 1, 1)/9;
Mat filter = MyConv(input,kernel);
```

使用系数全相同的模板同时也要保证系数之和为1. 2. 通过滤波器获取低频部分的图像 1. 使用卷积函数+均值模板过滤一遍图像，获取低频部分

```
Mat MyConv(const Mat& inputImage, const Mat& kernel)
{
    Mat outputImage = Mat::zeros(inputImage.size(), inputImage.type());

    int kernelSize = kernel.rows;
    int border = kernelSize / 2;

    for (int y = border; y < inputImage.rows - border; y++) {
        for (int x = border; x < inputImage.cols - border; x++) {
            int sum = 0;
            for (int i = -border; i <= border; i++) {
                for (int j = -border; j <= border; j++) {
                    sum += inputImage.at<uchar>(y + i, x + j) *
kernel.at<float>(i + border, j + border); //卷积
                }
            }
            outputImage.at<uchar>(y, x) = saturate_cast<uchar>(sum);
        }
    }

    return outputImage;
}
```

3. 获取高频部分

1. 通过用原图-低频部分的图片就可以获取图片的高频部分。
4. 将模板乘以一个k 加到原图像中以突出图片的高频边缘特性。

```
Mat Unsharp(Mat &input,Mat &highpass,float alpha)
{
    cv::Mat outputImage = cv::Mat::zeros(input.size(), input.type());
    for (int y = 0; y < input.rows; y++)
    {
        for (int x = 0; x < input.cols; x++)
```

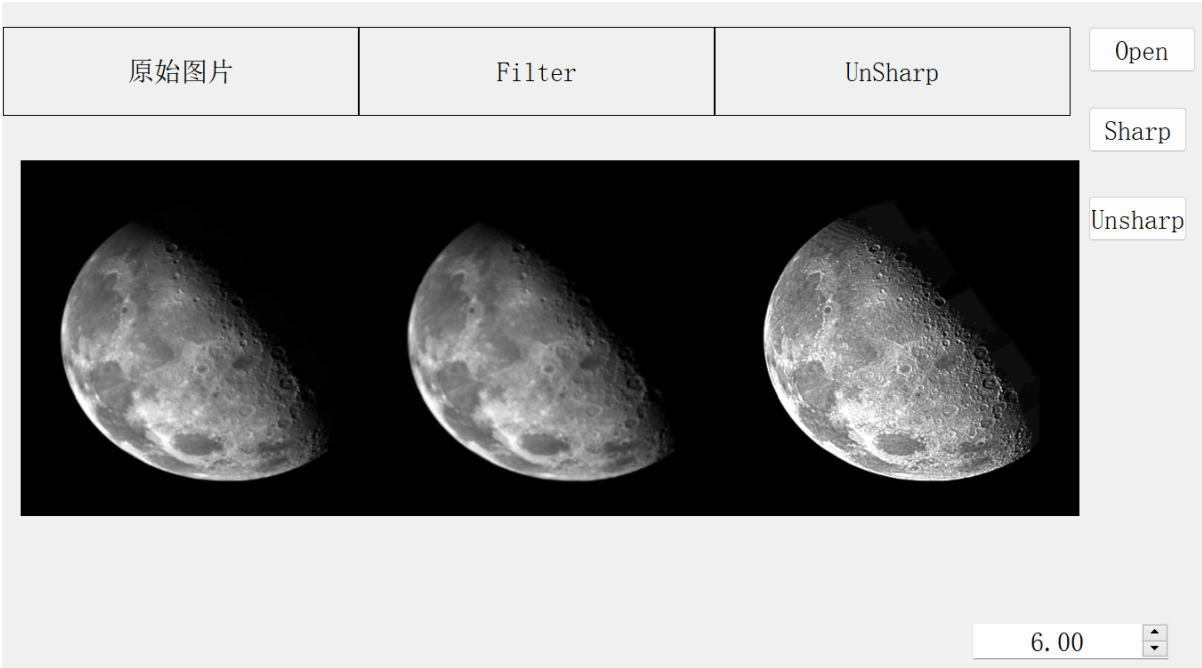
```
        {
            int newValue = input.at<uchar>(y, x) + cv::saturate_cast<uchar>
(alpha * highpass.at<uchar>(y, x)) ;//原图+α*高频部分
            outputImage.at<uchar>(y, x) = cv::saturate_cast<uchar>
(newValue);
        }

    return outputImage;
}
```

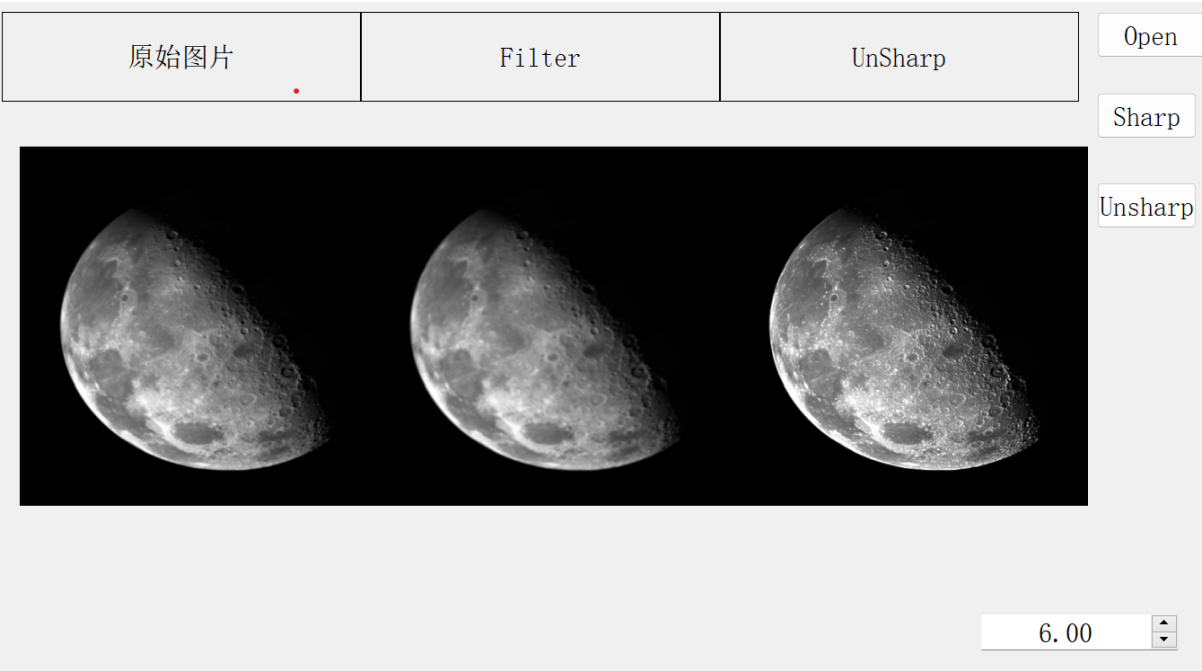
本实验中的参数 α 是通过UI界面用户输出而确定，体现了人机交互性，也方便后续讨论不同的参数对于图像的增强效果。具体思路就是，类似前文的卷积部分的过程，只不过在统计时是在原图相同位置的地方加上 α 高频部分。5. 结果展示 使用低通滤波器后的图片展示在Filter窗口处，可以看到图像比原图更加模糊不清了。后文会对比使用opencv的库函数滤波器对于实验的影响。Unsharp窗口展示的就是使用原图+ α 高频部分得到的图片发现效果和使用Laplacian算子的结果一样

四、Discussion

- 1. 对比不同的低通滤波器对于相同参数 α 对于结果的图片的影响。
 - 1. 自己写的滤波器



2. 均值滤波器



- 3. 可以发现使用库函数的均值滤波器的效果比自己写的效果好，这样处理出来的图像的相关更好
- 2. 分析不同的参数对于最后Unsharp的模板的效果。
 - 1. 探究了参数从1-15的结果对比


2. $\alpha = 1$

UnSharp

Open

Sharp

Unsharp



1.00

▲


▼

UnSharp

Open

Sharp

Unsharp



3.00

▲

▼


3. $\alpha = 3$

UnSharp

Open

Sharp

Unsharp



6.00

▲

▼


4. $\alpha = 6$

UnSharp

Open

Sharp

Unshar

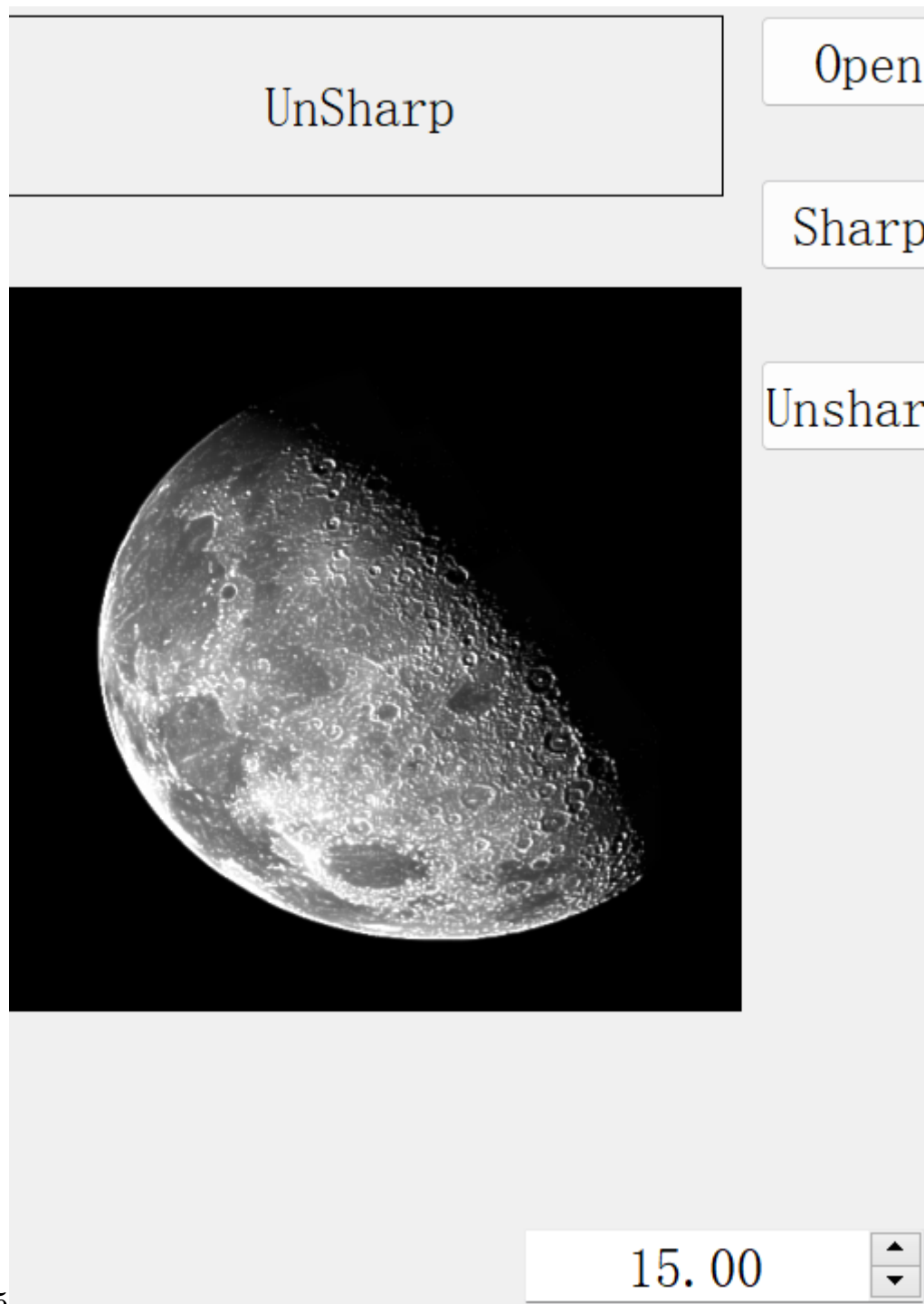


5. $\alpha = 7$

7.00

▲

▼



6. $\alpha = 15$

7. 原因

1. α 参数的值将影响最终增强图像的外观。这个参数控制了原图像与高频部分之间的权重，因此它会影响增强效果
2. α 值较小：当 α 的值较小时，增强效果会相对较弱，增强后的图像会更接近于原图像。这意味着边缘和细节的增强不会太明显。
3. α 值较大：当 α 的值较大时，增强效果会更明显，边缘和细节会更加强烈地凸显出来。然而，如果 α 值过大，可能会导致图像出现过度增强和噪声。

提高部分

一、实验内容 1.为什么自适应降噪效果更好？使用算法-自适应滤波器进行图像去噪处理，并分析原理。

二、实验原理

1. 自适应滤波器

1. 原理：自适应滤波器的特性根据 $m * n$ 矩形邻域 $S \times y$ 定义的滤波区域内的图像的统计特性变化。通常，自适应滤波器的性能优于前述的滤波器，但滤波器的复杂度和计算量也更大。

2. 算法步骤

1. 使用滑动窗口算法，便利图像的每一个点，根据kernel的大小，计算领域内的点的均值和方差。
2. 均值 $= \frac{\sum(g(i,j))}{\text{kernel} \times \text{kernel}}$, $\sigma^2 = \frac{\sum(g(i,j) - \text{mean})^2}{\text{kernel} \times \text{kernel}}$
3. 通过计算公式确定该点滤波后的结果。通过比较当前的计算出的 σ_l^2 以及全局的噪声的方差 σ_n^2 的大小关系。得出

$$f(x, y) = g(x, y) - \frac{\sigma_n^2}{\sigma_l^2} * (g(x, y) - \text{mean})$$
4. 需要注意的是我们人为规定噪声的方差 σ_n^2 是小于等于局部的灰度方差的所以在计算滤波后的灰度时，要进行判断如果小于的话，就需要为的规定局部方差的值 $\sigma_l^2 = \max(\sigma_l^2, \sigma_n^2)$
5. 计算均值 $\text{mean} += \text{inputImage.at}(y + wy, x + wx);$ 我们使用枚举整个窗口里的灰度值，然后相加，最后除窗口的大小。
6. 计算方差 $\text{double diff} = \text{inputImage.at}(y + wy, x + wx) - \text{mean};$ 我们使用求和当前值-均值。

2. 代码实现滤波器

```
Mat adaptiveAvg(Mat &inputImage, const int windowSize, double v)
{
    Mat outputImage = Mat::zeros(inputImage.size(), inputImage.type());

    int border = windowSize / 2;

    for (int y = 0; y < inputImage.rows - 0; y++) {
        for (int x = 0; x < inputImage.cols - 0; x++) {
            int border = windowSize / 2;

            double mean = 0.0;
            for (int wy = -border; wy <= border; wy++) {
                for (int wx = -border; wx <= border; wx++) {

                    if(y+wy>=0&&x+wx>=0&&y+wy<inputImage.rows&&x+wx<inputImage.cols)
                        mean += inputImage.at<uchar>(y + wy, x + wx);
                }
            }
            mean /= (windowSize * windowSize); // 计算均值

            double variance = 0.0;
            for (int wy = -border; wy <= border; wy++) {
                for (int wx = -border; wx <= border; wx++) {

                    if(y+wy>=0&&x+wx>=0&&y+wy<inputImage.rows&&x+wx<inputImage.cols)
                    {
                        double diff = inputImage.at<uchar>(y + wy, x + wx) - mean;
                        variance += diff * diff;
                    }
                }
            }
        }
    }
}
```

```

    }
    variance /= (windowSize * windowSize); // 计算方差

    variance=max(variance,v);
    variance= v/variance;

    outputImage.at<uchar>(y, x) = saturate_cast<uchar>
(inputImage.at<uchar>(y, x)-
                                                    (variance)*
(inputImage.at<uchar>(y, x)-mean));
    }
}

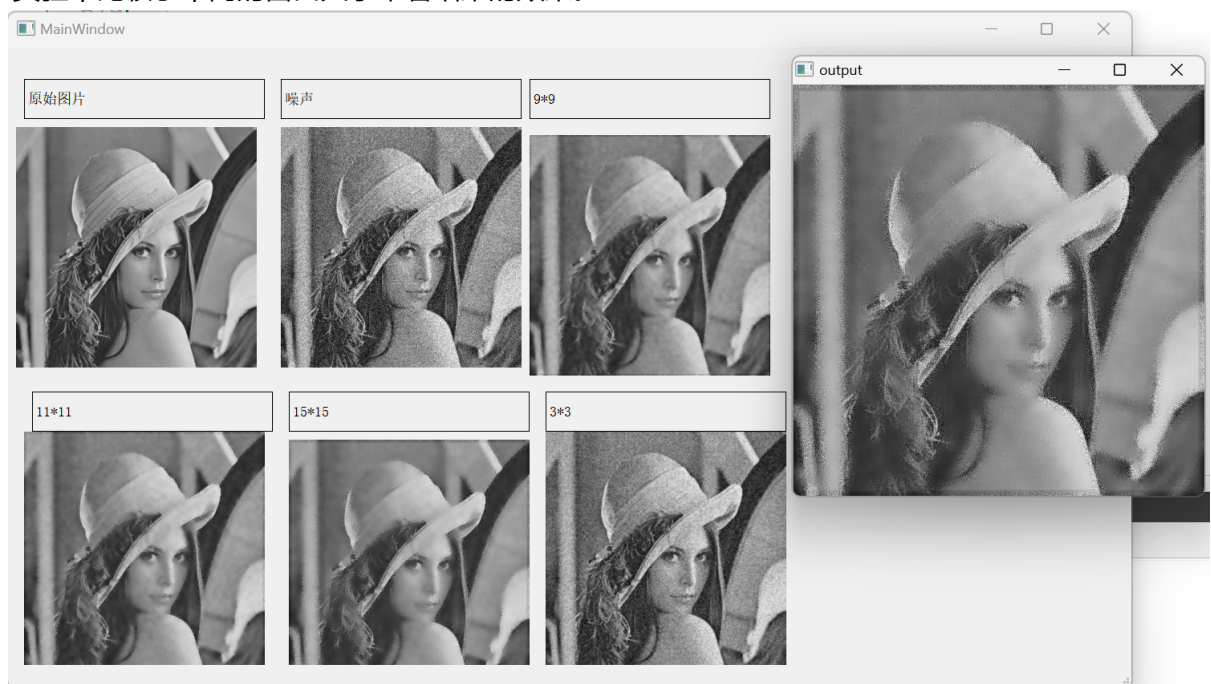
return outputImage;
}

```

最后计算当前点的结果值时我们使用的是前文的公式。

三、对比实验

1. 我们在实验中比较了不同的窗口大小来看结果的效果。



- 1.
2. 我们枚举了 3*3, 9*9, 11*11, 15*15 的大小对于结果的影响。可以发现在窗口大小很小时，效果很不明显。随着大小的增大，图像逐渐清晰。但是最右边显示了 15*15 的图像，可以发现图像出现了虚化效果。
3. 分析
 1. 调整窗口大小对自适应滤波器的效果具有重要影响。窗口大小决定了在进行自适应滤波时考虑的局部像素区域的大小。
 2. 较大的窗口大小将包括更多的像素，导致更强的平滑效果。这意味着较大的窗口能够去除更多的噪声，但可能会丧失一些图像细节。
 3. 在自适应均值滤波中，较大的窗口可能会导致更接近于全局均值的结果。较小的窗口会更敏感于局部均值，产生更多的局部对比度
4. 为什么自适应更好

1. 自适应滤波器根据局部像素的特性自动调整滤波参数。这意味着它可以在不同的图像区域使用不同的滤波策略，从而更好地处理不均匀的噪声分布和图像细节。
2. 适应滤波器可以有效地抑制噪声，因为它考虑了局部噪声水平。与普通滤波器不同，自适应滤波器不会过度平滑包含噪声的区域，因此可以更好地保留图像细节。
3. 自适应滤波器通常能够更好地保留图像细节，因为它根据局部对比度来调整滤波强度。这使得它在去噪的同时能够保持图像的清晰度和锐度。