题目：　　图像滤波算法的实现与对比

一、前文概述

图像滤波是遥感图像处理中一项重要的技术，它通过对图像进行空间域或频域处理，改善图像质量，去除噪声，增强图像细节或突出特征。在遥感应用中，图像数据往往受噪声、光照不均等因素影响，滤波技术在图像预处理阶段发挥着至关重要的作用。

常见的图像滤波算法包括均值滤波、中值滤波、Sobel 算子、Roberts 交叉梯度算子、拉普拉斯算子和高斯滤波等。这些算法各自基于不同的数学原理，通过调整滤波核和参数，达到去噪、边缘检测、特征提取等效果。滤波技术不仅可以平滑图像、去除高频噪声，还能有效地突出图像的边缘和纹理特征，尤其在遥感图像的解译和分析中具有广泛应用。

在遥感图像处理中，滤波不仅影响图像的视觉效果，还直接影响后续处理步骤的精度与效果。例如，在图像分类、目标识别等任务中，噪声的存在可能会导致误分类或识别错误，因此，选择合适的滤波算法至关重要。

本作业将通过实现并分析多种滤波算法，探讨它们在不同应用场景中的适用性、效果和计算效率，评估其在遥感图像处理中的作用。

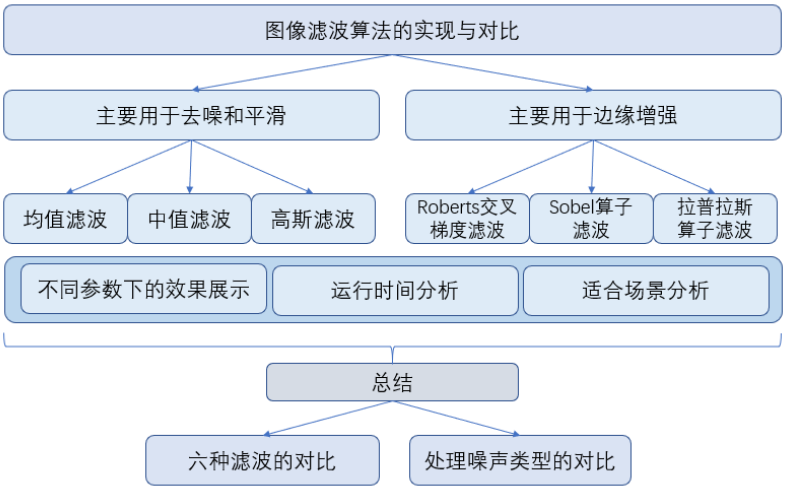


图 1：作业结构图

二、均值滤波

均值滤波是一种常见的图像平滑技术，广泛用于去除图像中的噪声。它通过取图像中每个像素邻域的平均值来替换该像素，从而平滑图像并减少高频噪声。均值滤波是一种局部的线性滤波器，它的核心思想是利用一个固定大小的滤波窗口（通常是一个方形窗口，如 3×3、5×5 等），对图像的每个像素进行操作。

设图像的像素矩阵为 $I(x,y)$，其中 x 和 y 表示图像中的坐标。均值滤波的输出图像 $J(x,y)$ 可以通过以下数学公式来描述：

$$J(x,y)=\frac{1}{N}\sum_{i=-k}^{k}\sum_{j=-k}^{k}I(x+i,y+j)$$

其中：

$J(x,y)$ 表示滤波后图像在位置 (x,y) 的像素值。

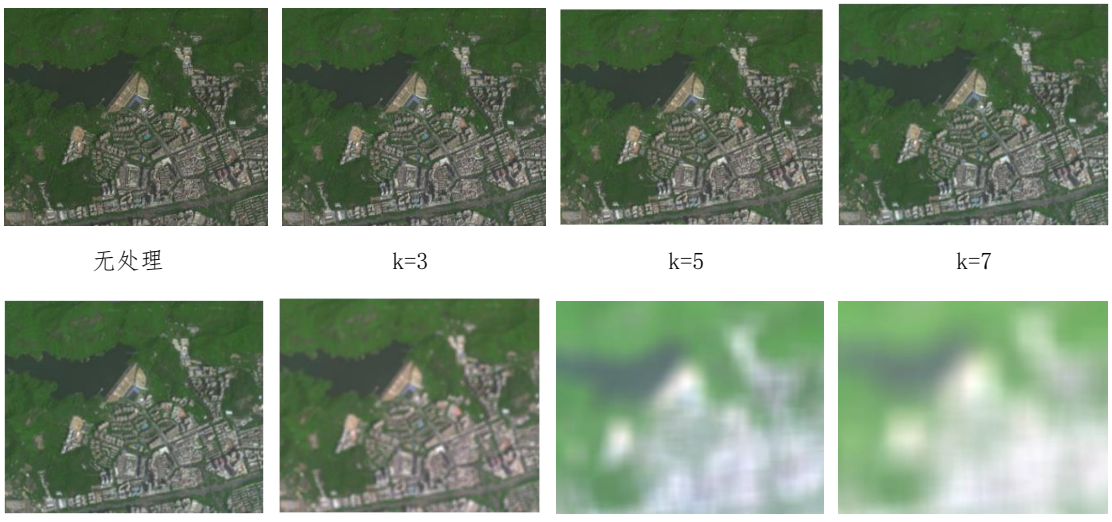
$I(x+i,y+j)$ 表示原始图像在 $(x+i,y+j)$ 位置的像素值。

k 是滤波窗口的半径。

N 是滤波窗口中包含的像素数量，通常为 $N=(2k+1)^2$

均值滤波是一种线性滤波器，这意味着它对图像的加权和进行操作。滤波过程中，每个像素都受到邻域像素的同等影响，即所有邻域像素的权重相同。因此，均值滤波的数学本质是对图像进行平滑或去噪。

均值滤波的计算复杂度取决于图像的尺寸和滤波窗口的大小。对于一个尺寸为 $M\times N$ 的图像和一个 $k\times \mathrm{k}$ 的滤波窗口，计算复杂度为 $O(M\times \mathrm{N}\times \mathrm{k}^2)$。均值滤波的主要作用是降低图像中的随机噪声，尤其是高频噪声。例如，图像中的椒盐噪声或随机噪声会被均值滤波器去除。然而，均值滤波的缺点是它可能会导致图像细节的丢失，特别是在处理边缘或细节较多的图像时。



无处理　　k=3　　k=5　　k=7

K=9　　k=10　　k=100　　k=150

图 2-9：原图像在不同窗口大小下均值滤波的不同处理结果

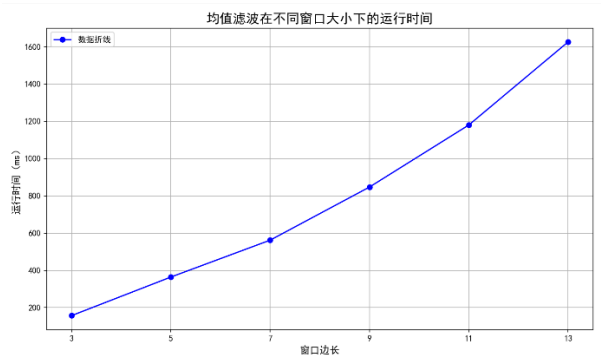


图 10：均值滤波在不同窗口大小下的运行时间

以上图像是在均值滤波处理下的结果。可以发现，在滤波窗口较小时，基本上图像没有什么变化。当滤波窗口逐渐增大后，图像越来越模糊，而且还可以发现整体颜色越来越亮，当窗口在超高值的情况下，基本上呈现亮色云雾状。均值滤波在处理过程中可以平滑图像的高斯噪声等随机噪声，但是不太适合处理含有脉冲噪声或者椒盐噪声的图像，因为均值会被异常值显著拉高或者拉低。可以看到，后期图像越来越亮，脱离初始状态也是与原图像中包含有很多高亮异常点有关，它们显著拉高了均值，使得图像变亮。均值滤波能够有效降低图像的整体噪声水平，但是就图像效果来看，包含建筑和水库在内的地物都被模糊了细节和边缘，这实际上反映了均值滤波对于图像中的均匀区域更加有效，但是对于边界和纹理丰富的区域，会导致信息丢失。另外，可以发现均值滤波在不同窗口大小下的运行时间，是随着窗口越来越大，运行时间也相应地越来越大，到那时整体运行时间还是非常快的，这也是均值滤波的优点，计算效率高，这也适合于处理大规模遥感图像。

三、中值滤波

中值滤波是一种非线性滤波方法，广泛应用于图像处理，特别是在去除椒盐噪声时表现良好。与均值滤波不同，中值滤波的核心思想是用邻域像素的中值来代替当前像素值，从而减少噪声影响，并保持图像的边缘特征。中值滤波通过在图像中每个像素位置的邻域内选择中位数值来平滑图像。在处理中值滤波时，邻域内的像素值不再是简单的平均值，而是排序后的中值。中值滤波主要应用于去除突出的离群值或噪声（如椒盐噪声），它对于边缘和细节的保留能力较强。

设图像的像素矩阵为 $I(x,y)$，其中 x 和 y 表示图像的坐标。对于输出图像 $J(x,y)$ 的每个像素，可以用以下公式描述中值滤波的过程：

$$J(x,y) = median(\{I(x+i,y+j) \mid -\mathrm{k} \le \mathrm{i} \le \mathrm{k}, -\mathrm{k} \le \mathrm{j} \le \mathrm{k}\})$$

其中：

$J(x,y)$ 表示滤波后图像在位置 (x,y) 的像素值。

$I(x+i,y+j)$ 表示原始图像在 $(x+i,y+j)$ 位置的像素值。

k 是滤波窗口的半径。

中值滤波的计算复杂度较高，因为每个像素需要对邻域内的像素进行排序。假设图像尺寸为 $M \times N$，滤波窗口大小为 $k \times k$，则计算复杂度为 $O(M \times N \times k^2 \log(k^2))$，其中 $k^2 \log(k^2)$ 是排序操作的时间复杂度。

无处理　　k=3　　k=5　　k=7

k=9　　k=11　　k=20　　k=40

图 11-18：原图像在不同窗口大小下中值滤波的不同处理结果

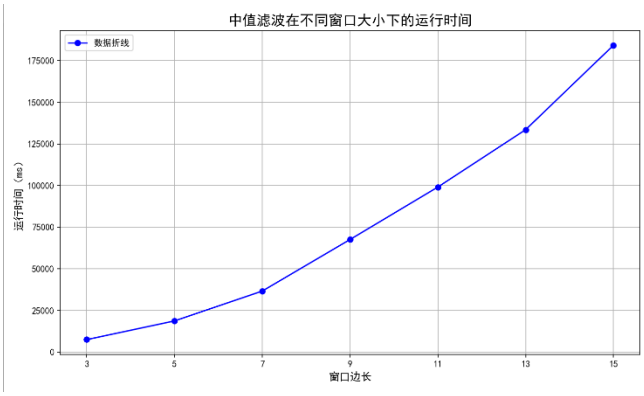


图 19：中值滤波在不同窗口大小下的运行时间

以上是中值滤波处理图像的对比效果，可以发现，在窗口逐渐增大时，图像也是越来越模糊，但是这种模糊效果与均值滤波有着显著的区别，可以发现尽管很多地物内部模糊了，但是很好地保留了边缘和轮廓，使得不同地物之间界限明显，而且一些突出的特征也保留了下来。中值滤波特别适合于处理脉冲噪声和椒盐噪声，尽管对于高斯噪声效果一般，但仍优于均值滤波。这意味着中值滤波能够在保留图像边缘和细节信息的同时去除离散异常点噪声。在遥感图像中，比较适合于去除雷达成像、传感器故障或者数据传输错误造成的噪声点，而不会显著模糊细节。我们看中值滤波在不同窗口下的运行时间可以发现，变化规律与均值滤波类似，也是随着窗口边长的增大，运行时间逐渐上升。但是其关键问题是时间复杂度较高，特别是在窗口较大时，耗时相当久，对于噪声密度较低的遥感图像，可能需要进一步调整。

四、Roberts 交叉梯度算子滤波

Roberts 交叉梯度算子是一种用于边缘检测和图像锐化的算子，主要通过计算图像灰度的梯度来突出边缘。它是一种基于一阶导数的算子，使用两个相邻像素的灰度值变化来估计图像的梯度。Roberts 交叉梯度算子尤其适用于检测图像中细小的边缘，具有较高的计算效率。其核心算子为两个卷积核（分别用于计算图像在两个方向上的梯度）：

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

其中，

G_x 用于计算图像在水平方向的梯度。

G_y 用于计算图像在垂直方向的梯度。

在 Roberts 算子中，通过将图像与卷积核 G_x 和 G_y 卷积，分别计算水平方向和垂直方向的梯度：

$$I_x(x,y) = G_x \times I(x,y), \quad I_y(x,y) = G_y \times I(x,y)$$

其中，$I(x,y)$ 是原始图像，$I_x(x,y)$ 和 $I_y(x,y)$ 分别是图像在水平方向和垂直方向的梯度。为了综合考虑水平方向和垂直方向的边缘强度，Roberts 算子通过以下公式计算图像的梯度幅值（即边缘强度）：

$$I_{edge}(x,y) = \sqrt{I_x(x,y)^2 + I_y(x,y)^2}$$

或者，也可以使用曼哈顿距离（绝对值）来近似：

$$I_{edge}(x,y) = |I_x(x,y)| + |I_y(x,y)|$$

该公式计算出每个像素点的边缘强度，表示图像中边缘的显著性。

在图像锐化中，Roberts 算子的输出通常被用来增强图像的边缘。图像锐化可以通过以下公式实现：

$$I_{sharpened}(x,y) = I(x,y) - \alpha \cdot I_{edge}(x,y)$$

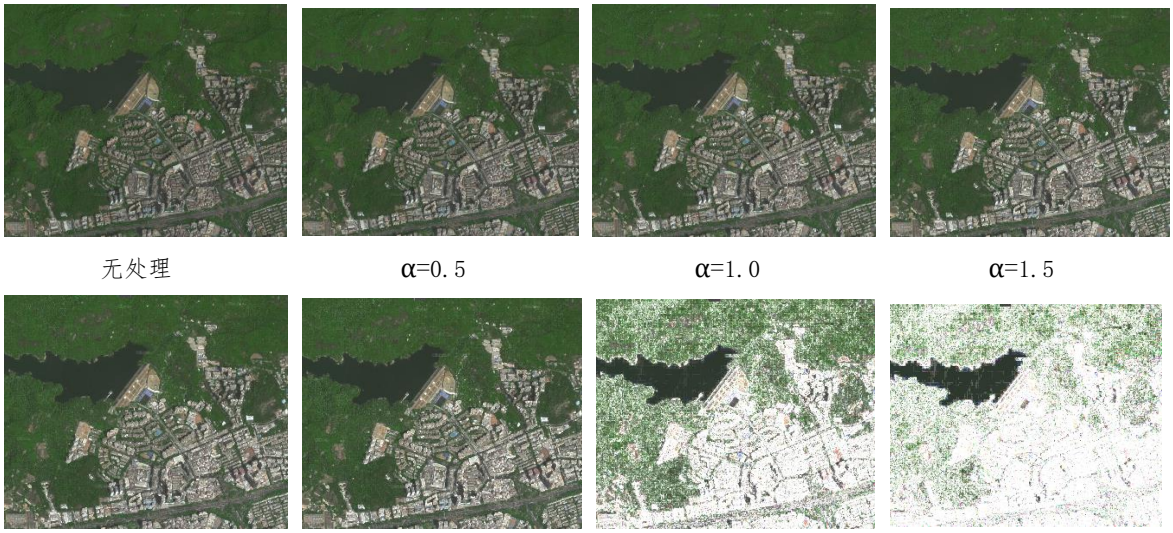
其中：

$I(x,y)$ 是原始图像。

$I_{sharpened}(x,y)$ 是锐化后的图像。

α是一个常数，控制锐化的强度。

Roberts 交叉梯度算子锐化滤波通过计算图像灰度的梯度来突出边缘细节。它通过两个卷积核 G_x 和 G_y 分别计算图像的水平方向和垂直方向的梯度，再通过结合这两个方向的梯度来估计边缘强度。最终，通过将边缘信息加回原图像，实现图像的锐化。Roberts 算子计算简单，适合用于检测细小边缘，但对于噪声较为敏感，常与其他平滑处理结合使用。



无处理　　α=0.5　　α=1.0　　α=1.5

α=2.0　　α=1.7　　α=20　　α=50

图 20-27：原图像在不同锐化强度下 Roberts 交叉梯度算子滤波的不同处理结果

以上是 Roberts 交叉梯度算子滤波处理图像的对比效果，另外由于该滤波的处理时间与参数调整基本没有什么关系，所以没有绘制变化曲线图，处理时间基本稳定在 200ms 左右，可能会有非常小幅度的上下波动。从对比图像中可以发现，Roberts 交叉梯度算子滤波在锐化强度增强后，图像越来越呈现全局的白色，其中的边界信息也越来越明显，随着锐化强度增高，一些不明显的边界也被抹除，最后都是一些非常显著的边界信息。Roberts 交叉梯度算子滤波与前面的滤波截然不同，主要不是为了去噪，而是为了边缘检测，适合需要提取遥感图像中清晰边界的场景，可以发现图中植被、水库、建筑、道路的边界也越来越清晰。Roberts 交叉梯度算子滤波的时间复杂度相对于中值滤波来说比较低，和均值滤波的较小窗口状态的处理时间比较接近。该滤波能快速检测边缘，但对噪声非常敏感，特别是高斯噪声和椒盐噪声。在遥感应用中，常作为预处理步骤，用于后续特征提取。

五、Sobel 算子滤波

Sobel 算子是一种常用的边缘检测算子，它基于一阶梯度算子，能够检测图像中的边缘并增强图像的细节。与 Roberts 算子类似，Sobel 算子通过计算图像的梯度来识别边缘，但 Sobel 算子使用的是加权的卷积核，更能平滑噪声，适用于较大范围的边缘检测。

Sobel 算子通过计算图像在水平（x 方向）和垂直（y 方向）两个方向上的梯度来估计图像的边缘。其水平方向（G_x）的卷积核和垂直方向（G_y）的卷积核分别为：

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

其梯度计算与边缘强度计算与 Roberts 交叉梯度算子相同，不赘述。Sobel 算子与 Roberts 算子的不同之处在于它的卷积核有更多的权重（例如，G_x 中的-2 和 2），这使得它对边缘的检测更加平滑，并且对于噪声的抑制能力更强。由于其加权卷积核，Sobel 算子能够更好地平滑图像，适用于检测较大范围的边缘，尤其适合于中等噪声的图像。Sobel 算子计算出的梯度幅值可以用于图像锐化，通过增强边缘信息，提升图像的清晰度。

无处理　　α=0.1　　α=0.3　　α=0.5

α=0.7　　α=0.9　　α=3.0　　α=5.0

图 28-35：原图像在不同锐化强度下 Sobel 算子滤波的不同处理结果

以上是 Sobel 算子滤波的处理对比，可以发现 Sobel 算子的处理非常类似于 Roberts 交叉梯度算子，其用于边缘检测，但其抗噪声能力更强，对高斯噪声的鲁棒性优于 Roberts 交叉梯度算子，可以更好地提取边缘特征，生成分割或分类所需的边界信息。根据实验结果也发现其运行时间与 Roberts 交叉梯度算子差不多，所以总体上优于 Roberts 交叉梯度算子。

六、拉普拉斯算子滤波

拉普拉斯算子是一种常用于图像处理中的二阶导数算子，主要用于边缘检测和图像锐化。通过计算图像的二阶导数，拉普拉斯算子能够检测到图像中灰度变化较大的区域，即边缘部分。利用拉普拉斯算子进行图像锐化时，可以突出图像的细节和边缘，增强图像的对比度。在二维图像处理中，拉普拉斯算子通常表示为：

$$\Delta I(x,y) = \frac{\partial^2 I(x,y)}{\partial x^2} + \frac{\partial^2 I(x,y)}{\partial y^2}$$

其中：

$I(x,y)$ 是图像在位置 (x,y) 的灰度值。

$\frac{\partial^2 I(x,y)}{\partial x^2}$ 和 $\frac{\partial^2 I(x,y)}{\partial y^2}$ 分别表示图像在 x 和 y 方向上的二阶偏导数。

拉普拉斯算子本身可以突出图像的边缘，但由于它在边缘区域生成负值，因此通常会进行锐化处理，即通过对图像进行加法操作，增强图像细节。锐化操作可以通过以下公式实现：

$$I_{sharpened}(x,y) = I(x,y) - \alpha \cdot \Delta I(x,y)$$

其中：

$I_{sharpened}(x,y)$ 是锐化后的图像。

$I(x,y)$ 是原始图像。

α是一个常数，控制锐化的强度。较大的α值会增强锐化效果。

锐化操作通过将原始图像减去其二阶导数（即拉普拉斯算子的结果），来突出图像中的高频成分（边缘和细节）。在数字图像处理中，拉普拉斯算子通常通过离散化的卷积核来实现，常见的拉普拉斯算子的卷积核为：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \text{ 或 } \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

这些卷积核用于计算图像每个像素点及其邻域的加权和，得到每个像素的二阶导数值。拉普拉斯算子能突出图像中的边缘信息，因为图像的边缘通常具有较大的灰度变化率。通过锐化处理，图像中的细节会被增强，特别是在边缘区域。拉普拉斯锐化可能会增强图像中的噪声，特别是在低对比度区域。

无处理　　α=3　　α=5　　α=7

α=9　　α=11　　α=100　　α=150

图 36-43：原图像在不同锐化强度下拉普拉斯算子滤波的不同处理结果

以上是拉普拉斯算子滤波的处理对比，可以发现拉普拉斯算子滤波在锐化强度增强之后不断增强图像边缘，而且似乎多出了很多无效的奇怪边缘，在锐化强度过高时，边缘过大，最终形成一个非常混乱的情形，只有一些内部比较统一的地物会有所缓和。拉普拉斯算子滤波本身也是为了锐化而生，而非去噪，因此对于噪声异常敏感，在锐化强度过高时产生了大量的伪边缘。拉普拉斯算子滤波显著增强了边缘对比度，突出目标物体，比较适合用于分析的模糊区域，比如说边界不清的云层、山脉等，但是对于上述实验图像这种细节复杂的图像可能效果比较糟糕，尤其是在锐化强度过高时，一般来讲，拉普拉斯算子会在去噪处理之后进行，如果进行了平滑和去噪之后，一些无关紧要的边缘被抹除，拉普拉斯算子滤波才可以更好地显示重要的边缘，而非伪边缘影响总体效果。

七、高斯滤波

高斯滤波是一种广泛应用的图像平滑（去噪）技术，其主要目的是通过加权平均周围像素来减少图像中的噪声，特别是在图像的平滑区域。高斯滤波基于高斯函数，其滤波效果遵循高斯分布，能够保留图像中的边缘特征同时有效去除噪声。高斯滤波基于高斯函数，该函数定义为：

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

其中：

$G(x,y)$ 是二维高斯函数的值，表示每个像素的加权值。

σ 是标准差，控制高斯函数的宽度。较小的 σ 值意味着滤波器对邻近像素的加权更强，较大的 σ 值则意味着加权较为平滑。

x 和 y 是像素在二维空间中的位置。

高斯滤波的实现是通过卷积操作完成的，即用高斯滤波器与图像进行卷积。在图像中的每个像素点，卷积操作会将其周围邻域的像素值与高斯滤波器中的对应权重值相乘，然后将这些乘积加起来得到新的像素值。

$$I_{filtered}(x,y) = \sum_{i=-k}^{k}\sum_{j=-k}^{k} G(i,j)\cdot I(x+i,y+j)$$

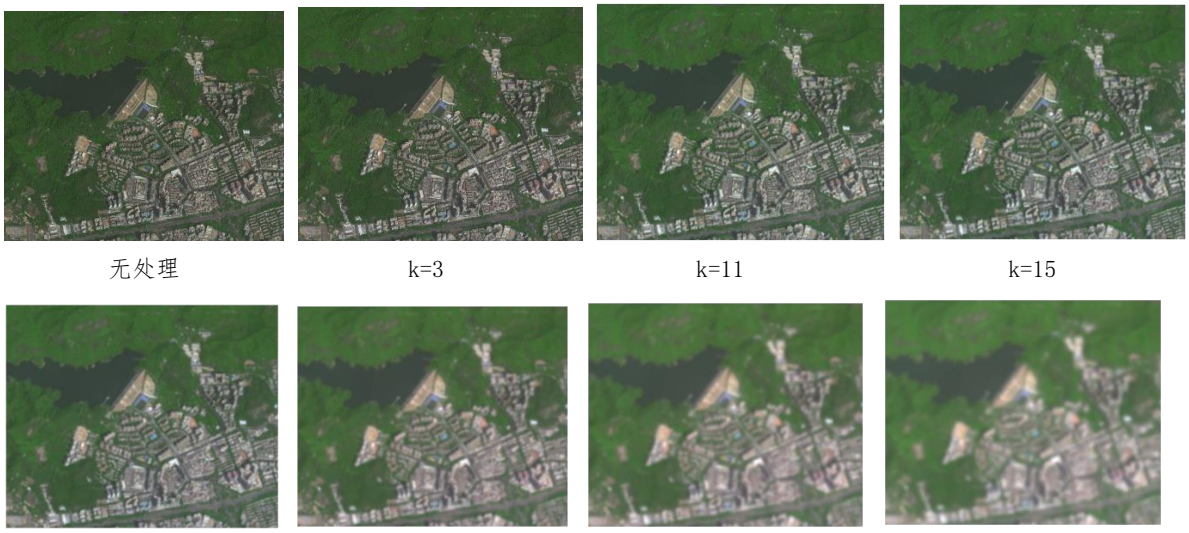
其中：

$G(i,j)$ 是高斯滤波器的值（权重）。

$I(x+i,y+j)$是图像中对应位置的像素值。

k 是滤波器半径的一半，决定了滤波器的大小。

高斯滤波通过加权平均去除图像中的高频噪声，同时保留了低频信息。由于高斯函数的权重随着距离中心像素的增大而逐渐减小，滤波效果通常会较为平滑。与均值滤波不同，高斯滤波能够更好地保留图像中的边缘，因为其加权函数使得图像中心像素的影响最大，远离中心的像素对滤波结果的影响逐渐减小。



无处理　　k=3　　k=11　　k=15

K=21　　k=31　　k=41　　k=51

图 44-51：原图像在不同核大小下高斯滤波的不同处理结果

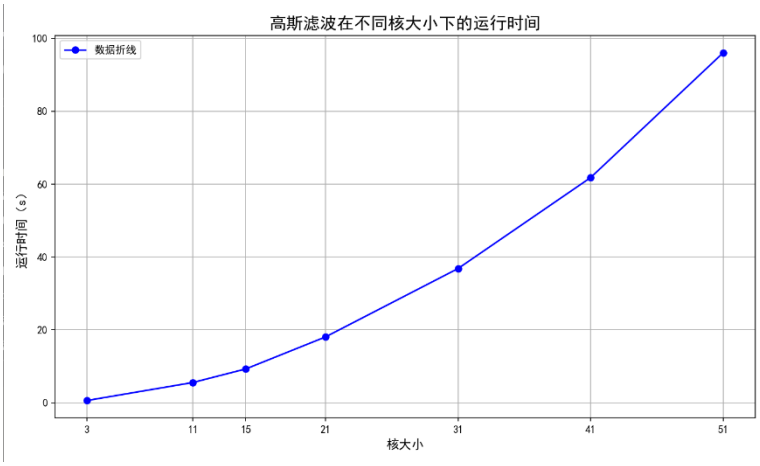


图 52：高斯滤波在不同核大小下的运行时间

以上是高斯滤波的处理对比，可以发现随着核大小的增加，高斯滤波能够将图像进行更加强的平滑处理，其效果会与均值滤波的效果类似，但实际上会比均值滤波更加自然。高斯滤波是处理高斯噪声的理想选择，同时具有良好的平滑效果，但是不适合处理含有显著异常点的遥感图像。在遥感图像处理中，高斯滤波适合于大气纠正和背景平滑的任务。从高斯滤波在不同核大小下的运行时间来说，低于中值滤波，但会高于均值滤波，适合需要较高平滑效果且不丢失太多细节的场景。

八、总结

以下是六种滤波对比和噪音分类对比：

六种滤波对比

滤波方法	底层原理	优点	缺点	计算复杂度	参数影响
均值滤波	通过用窗口内像素的平均值替代中心像素值实现平滑，降低噪声。	简单易实现；对高斯噪声有效；平滑效果较好。	模糊边缘；对脉冲噪声效果较差。	$O(n^2 \cdot k^2)$	核越大，平滑效果越强，细节丢失越严重。
中值滤波	用窗口内像素的中值替代中心像素值，有效去除噪声同时保留边缘。	对脉冲噪声去除效果好；边缘保留效果较好。	对高斯噪声去除效果较差；计算复杂度较高。	$O(n^2 \cdot k^2 \log k)$	窗口越大，去噪效果越好，但计算更耗时，可能导致过度平滑。
Roberts 交叉梯度算子滤波	使用对角方向的梯度计算图像边缘强度，检测图像边缘的变化。	对边缘检测快速有效；实现简单。	易受噪声影响；对粗糙边缘效果差。	$O(n^2)$	锐化强度越大，边缘检测越明显，但噪声增强可能性更高。
Sobel 算子滤波	通过卷积核对图像水平和垂直方向求梯度幅值，用于边缘检测。	对边缘和纹理检测性能较好；抗噪声能力较 Roberts 强。	对复杂纹理效果一般；边缘可能不够锐利。	$O(n^2)$	锐化强度越大，边缘和梯度增强，但噪声可能被放大。
拉普拉斯算子滤波	使用二阶导数计算图像的边缘变化，通过零交叉点检测边缘。	对边缘检测更敏感；适合锐化模糊图像。	极易受噪声影响；需先去噪处理；可能产生假边缘。	$O(n^2)$	锐化强度越大，边缘突出效果更强，但可能引入更多伪边缘。
高斯滤波	用高斯核权重像素值，平滑图像，抑制噪声。	对高斯噪声平滑效果好；权重分布平滑，保留细节较均值滤波好。	模糊边缘；对脉冲噪声效果较差。	$O(n^2 \cdot k^2)$	核越大，平滑效果越强；标准差越大，模糊范围越广，保留边缘能力减弱。

注：n 是图像宽高，k 是滤波窗口的大小

噪声分类对比

噪声种类	脉冲噪声	高斯噪声	椒盐噪声
分布形式	离散、随机出现	正态分布	随机位置，值为最小值或最大值
对图像的表现	随机孤立点	整体随机小幅波动	图像上散布黑点和白点
典型影响	局部突变影响明显	整体模糊，细节损失	边缘和纹理丢失

适合去除的方法	中值滤波	均值滤波、高斯滤波	中值滤波、对噪声检测后局部修复

在本次作业中，进行了六种滤波的对比实验和分析，总的来看，对于含高斯噪声的遥感图像，高斯滤波是优选；含脉冲噪声或椒盐噪声时，中值滤波效果最佳。Roberts 和 Sobel 适合目标边界提取，但需考虑噪声敏感性；拉普拉斯更适合在增强边缘细节时使用。均值和 Sobel 计算最快，适合实时处理需求；中值和高斯适合对精度要求较高但计算时间允许的情况。

九、参考资料

[1] 数字图像处理(11): 图像平滑 (均值滤波、中值滤波和高斯滤波)_高斯滤波,均值滤波,中值滤波-CSDN 博客

[2] 数字图像处理(19): 边缘检测算子(Roberts 算子、Prewitt 算子、Sobel 算子 和 Laplacian 算子) ubuntu 写出 roberts、sobel、laplace 边缘检测算法-CSDN 博客

[3] [CV] Roberts, Prewitt, Sobel 边缘检测算子 (原理 & C++实现) - 知乎

[4] 遥感数字图像处理（实验三）——滤波、锐化、平滑_convolutions and morphology-CSDN 博客

[5] 不同图像的噪声，选用什么滤波器去噪，图像处理的噪声和处理方法 高斯噪声用什么滤波方法去除-CSDN 博客

[6] 图像处理学习笔记（十四）——图像边缘锐化的基本方法(理论篇) - 知乎

附录（代码展示）

1.均值滤波完整代码

```
#include <iostream>
#include <sstream>
#include <stdio.h>
#include "gdal_priv.h"
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

void meanFilter(unsigned char* pData, unsigned char* pFilteredData, int width, int height, int kernelSize)
{
    int halfSize = kernelSize / 2;
    int kernelArea = kernelSize * kernelSize;
    for (int y = halfSize; y < height - halfSize; y++)
    {
        for (int x = halfSize; x < width - halfSize; x++)
        {
            int sum = 0;
            for (int i = -halfSize; i <= halfSize; i++)
            {
```

```
                for (int j = -halfSize; j <= halfSize; j++)
                {
                    sum += pData[(y + i) * width + (x + j)];
                }
            }
            pFilteredData[y * width + x] = sum / kernelArea;
        }
    }
}

void normalize(unsigned char* pData, int width, int height)
{
    unsigned char minVal = 255, maxVal = 0;

    for (int i = 0; i < width * height; i++)
    {
        if (pData[i] < minVal) minVal = pData[i];
        if (pData[i] > maxVal) maxVal = pData[i];
    }

    for (int i = 0; i < width * height; i++)
    {
        pData[i] = (unsigned char)(((double)(pData[i] - minVal) / (maxVal - minVal)) * 255);
    }
}

int main()
{
    GDALAllRegister();
    const char* inputFilename = "D:\\download\\gdal\\ConsoleApplication1\\InitialImage.tif";

    GDALDataset* pDatasetRead = (GDALDataset*)GDALOpen(inputFilename, GA_ReadOnly);

    if (pDatasetRead == NULL)
    {
        std::cout << "Cannot open file." << std::endl;
        return -1;
    }

    int lWidth = pDatasetRead->GetRasterXSize();
    int lHeight = pDatasetRead->GetRasterYSize();
    int nBands = pDatasetRead->GetRasterCount();

    GDALDriver* pDriver = GetGDALDriverManager()->GetDriverByName("GTiff");
    if (pDriver == NULL)
    {
        std::cout << "GTiff driver not available." << std::endl;
        GDALClose(pDatasetRead);
```

```
        return -1;
    }
    for (int kernelSize = 3; kernelSize <= 15; kernelSize += 2)
    {
        std::cout << "Testing kernelSize = " << kernelSize << std::endl;
        std::ostringstream outputFilenameStream;
        outputFilenameStream <<
"D:\\download\\gdal\\ConsoleApplication1\\MeanFiltering_kernelSize_"
            << kernelSize << ".tif";
        std::string outputFilename = outputFilenameStream.str();

        GDALDataset* pDatasetWrite = pDriver->Create(outputFilename.c_str(), lWidth, lHeight,
nBands, GDT_Byte, NULL);
        if (pDatasetWrite == NULL)
        {
            std::cout << "Cannot create output file for kernelSize = " << kernelSize <<
std::endl;
            continue;
        }
        auto start = high_resolution_clock::now();

        for (int band = 1; band <= nBands; band++)
        {
            std::cout << "Processing band " << band << " with kernelSize = " << kernelSize <<
std::endl;
            GDALRasterBand* pBand = pDatasetRead->GetRasterBand(band);
            unsigned char* pData = (unsigned char*)CPLMalloc(lWidth * lHeight * sizeof(unsigned
char));
            pBand->RasterIO(GF_Read, 0, 0, lWidth, lHeight, pData, lWidth, lHeight, GDT_Byte,
0, 0);
            unsigned char* pFilteredData = (unsigned char*)CPLMalloc(lWidth * lHeight *
sizeof(unsigned char));
            meanFilter(pData, pFilteredData, lWidth, lHeight, kernelSize);
            normalize(pFilteredData, lWidth, lHeight);
            GDALRasterBand* pOutBand = pDatasetWrite->GetRasterBand(band);
            pOutBand->RasterIO(GF_Write, 0, 0, lWidth, lHeight, pFilteredData, lWidth, lHeight,
GDT_Byte, 0, 0);
            CPLFree(pData);
            CPLFree(pFilteredData);
        }
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<milliseconds>(stop - start);
        std::cout << "kernelSize " << kernelSize << " finished in " << duration.count() << "
ms" << std::endl;
        GDALClose(pDatasetWrite);
    }
    GDALClose(pDatasetRead);
    std::cout << "Processing completed. Check output files." << std::endl;
```

```
    return 0;
}
```

2.中值滤波核心代码（由于其他内容相似，仅展示中值滤波函数）

```
void medianFilter(unsigned char* pData, unsigned char* pFilteredData, int width, int height,
int kernelSize)
{
    int halfSize = kernelSize / 2;
    std::vector<unsigned char> window;
    for (int y = halfSize; y < height - halfSize; y++)
    {
        for (int x = halfSize; x < width - halfSize; x++)
        {
            window.clear();
            for (int i = -halfSize; i <= halfSize; i++)
            {
                for (int j = -halfSize; j <= halfSize; j++)
                {
                    window.push_back(pData[(y + i) * width + (x + j)]);
                }
            }
            std::sort(window.begin(), window.end());
            pFilteredData[y * width + x] = window[window.size() / 2];
        }
    }
}
```

3.Roberts 交叉梯度算子滤波核心代码

```
void robertsSharpen(unsigned char* pData, unsigned char* pSharpenedData, int width, int height,
float sharpenFactor)
{
    for (int y = 0; y < height - 1; y++)
    {
        for (int x = 0; x < width - 1; x++)
        {
            int Gx = pData[y * width + x] - pData[(y + 1) * width + (x + 1)];
            int Gy = pData[y * width + (x + 1)] - pData[(y + 1) * width + x];
            int gradientValue = (int)(sqrt(Gx * Gx + Gy * Gy));
            int sharpenedValue = pData[y * width + x] + (int)(sharpenFactor * gradientValue);
            pSharpenedData[y * width + x] = std::min(255, std::max(0, sharpenedValue));
        }
    }
}
```

4.Sobel 算子滤波核心代码

```
void sobelFilter(unsigned char* pData, unsigned char* pFilteredData, int width, int height,
float filterFactor)
{
    int GxKernel[3][3] = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };
    int GyKernel[3][3] = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    int kernelSize = 3;
    int halfSize = kernelSize / 2;

    for (int y = halfSize; y < height - halfSize; y++)
    {
        for (int x = halfSize; x < width - halfSize; x++)
        {
            int Gx = 0, Gy = 0;

            for (int i = -halfSize; i <= halfSize; i++)
            {
                for (int j = -halfSize; j <= halfSize; j++)
                {
                    int pixelValue = pData[(y + i) * width + (x + j)];
                    Gx += GxKernel[i + halfSize][j + halfSize] * pixelValue;
                    Gy += GyKernel[i + halfSize][j + halfSize] * pixelValue;
                }
            }

            int gradientValue = (int)(sqrt(Gx * Gx + Gy * Gy));
            int filteredValue = pData[y * width + x] + (int)(filterFactor * gradientValue);
            pFilteredData[y * width + x] = std::min(255, std::max(0, filteredValue));
        }
    }
}
```

5.拉普拉斯算子滤波核心代码

```
void laplacianSharpen(unsigned char* pData, unsigned char* pSharpenedData, int width, int
height, float sharpenFactor)
{
    int kernel[3][3] = {
        {0,  -1, 0},
        {-1,  4, -1},
```

```
        {0,  -1, 0}
    };

    int kernelSize = 3;
    int halfSize = kernelSize / 2;
    for (int y = halfSize; y < height - halfSize; y++)
    {
        for (int x = halfSize; x < width - halfSize; x++)
        {
            int laplacianValue = 0;
            for (int i = -halfSize; i <= halfSize; i++)
            {
                for (int j = -halfSize; j <= halfSize; j++)
                {
                    int pixelValue = pData[(y + i) * width + (x + j)];
                    laplacianValue += kernel[i + halfSize][j + halfSize] * pixelValue;
                }
            }
            int sharpenedValue = pData[y * width + x] + (int)(sharpenFactor * laplacianValue);
            pSharpenedData[y * width + x] = std::min(255, std::max(0, sharpenedValue));
        }
    }
}
```

6.高斯滤波核心代码

```
std::vector<std::vector<float>> generateGaussianKernel(int kernelSize, float sigma) {
    int halfSize = kernelSize / 2;
    std::vector<std::vector<float>> kernel(kernelSize, std::vector<float>(kernelSize, 0));
    float sum = 0;

    for (int i = -halfSize; i <= halfSize; ++i) {
        for (int j = -halfSize; j <= halfSize; ++j) {
            kernel[i + halfSize][j + halfSize] = exp(-(i * i + j * j) / (2 * sigma * sigma));
            sum += kernel[i + halfSize][j + halfSize];
        }
    }
    for (int i = 0; i < kernelSize; ++i) {
        for (int j = 0; j < kernelSize; ++j) {
            kernel[i][j] /= sum;
        }
    }
    return kernel;
}

void gaussianFilter(unsigned char* pData, unsigned char* pFilteredData, int width, int height,
const std::vector<std::vector<float>>& kernel) {
    int kernelSize = kernel.size();
    int halfSize = kernelSize / 2;
```

```
    for (int y = halfSize; y < height - halfSize; ++y) {
        for (int x = halfSize; x < width - halfSize; ++x) {
            float sum = 0;
            for (int i = -halfSize; i <= halfSize; ++i) {
                for (int j = -halfSize; j <= halfSize; ++j) {
                    int pixelValue = pData[(y + i) * width + (x + j)];
                    sum += pixelValue * kernel[i + halfSize][j + halfSize];
                }
            }
            pFilteredData[y * width + x] = static_cast<unsigned char>(std::min(255.0f,
std::max(0.0f, sum)));
        }
    }
}
```