VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 3: Smooth the image (image blurring, image smoothing)

Course: Digital Image and Video Processing

Supervisors:

Prof. Dr. Ly Quoc Ngoc

MSc. Nguyen Manh Hung

MSc. Pham Thanh Tung

Student:

23127266 - Nguyen Anh Thu

Ngày 12 tháng 12 năm 2025

Mục lục

Danh sách hình vẽ

Danh sách bảng

Chapter I: Introduction

1. Overview

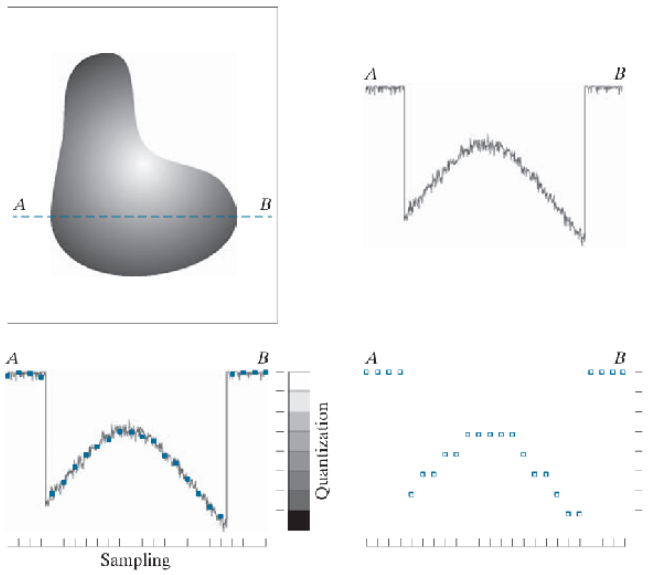
Smoothing, also known as blurring, constitutes a fundamental procedure in digital image processing. It is frequently used as a preprocessing operation to reduce noise, extract features, and enhance the reliability of edge-aware algorithms.

This assignment focuses on the manual implementation of multiple spatial smoothing techniques utilizing numerical computation with NumPy. All filters are applied to grayscale images $I(x, y)$, which are either obtained via conversion from RGB color images or originally available in single-channel format.

1.1. Definition

From Continuous Images to Digital Images

In the continuous domain, an image is represented as a smooth intensity function whose values vary gradually across spatial coordinates. However, when this continuous signal is converted into a discrete spatial image through sampling and quantization, its imperfections become more apparent.



Hình I.1: (a) Continuous mage. (b) A scan line showing intensity variations along line AB in the continuous image. (c) Sampling and quantization. (d) Digital scan line. (The black border in (a) is included for clarity. It is not part of the image).[1, p. 64]

As illustrated in I.1, the intensity profile measured along the line segment A-B shows small, irregular fluctuations that do not belong to the true scene. These fluctuations persist after digitization and are a direct manifestation of image noise in the spatial domain.[1, p. 64]

Thus, even before any processing takes place, the digital image already contains noise inherited from the continuous acquisition process. This observation motivates the need for spatial-domain smoothing techniques, whose objective is to suppress such unwanted variations while preserving the underlying structural information of the image.

What is an image noise?

In an ideal imaging system, each spatial location (x, y) of an image should reflect only the true intensity of the scene. However, real images often contain *noise*-random fluctuations that distort these true intensity values.

Formally, a noisy image can be expressed using the additive degradation model [1, p. 86]:

$$g(x, y) = f(x, y) + \eta(x, y),$$

where $f(x, y)$ is the unknown noiseless image, $g(x, y)$ is the observed corrupted image, and $\eta(x, y)$ denotes the noise component. In classical digital image processing, the noise $\eta(x, y)$ is typically assumed to satisfy:

- zero-mean: $\mathbb{E}[\eta(x, y)] = 0$,
- uncorrelated across pixel locations,
- statistically independent of the underlying image intensity.

These assumptions allow the noise to be treated analytically and support designing effective smoothing and estimation procedures.

1.2. Problem Statement

a) Input.

A grayscale image corrupted by additive noise:

$$f(x, y) : \Omega \subset \mathbb{Z}^2 \to [0, 255],$$

where Ω denotes the spatial domain of the image and $f(x, y)$ is the observed noisy intensity.

b) Output.

A noise-reduced (denoised) image:

$$g(x, y) : \Omega \subset \mathbb{Z}^2 \to [0, 255].$$

c) Goal.

Find an operator $\mathcal{F}$ such that:

$$g(x, y) = \mathcal{F}[f(x, y)],$$

where $\mathcal{F}$ reduces the noise contained in $f(x, y)$ while preserving essential structural elements of the scene, including edges, contours, and fine textures. In other words, the task is to recover an estimate $\hat{f}(x, y)$ of the true image $f(x, y)$ from its noisy observation $g(x, y)$ using appropriate smoothing techniques.

2. Approaches

Smoothing (or filtering) refers to a family of spatial-domain operations designed to reduce local intensity fluctuations, thereby suppressing noise while preserving meaningful structural information in the image. While many smoothing methods share the common goal of reducing noise, they differ fundamentally in how they combine pixel information. These methods fall into two major categories: *linear convolution-based smoothing* and *nonlinear or adaptive smoothing*.

2.1. Linear Smoothing (Convolution-Based)

Linear smoothing models the filtering operation directly in the spatial domain. A filter is defined by a kernel $w(s,t)$, whose coefficients specify how pixels in a local neighborhood contribute to the output. Because the kernel is fixed across the entire image, the filter does not adapt to edges or structural variations; every location is processed in exactly the same manner. This shift-invariant behavior is the reason linear smoothing reduces noise but also blurs edges.

With the kernel centered at location (x, y), the filtered output is obtained by combining neighboring pixels through a weighted sum [1, p. 154]:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x+s,\, y+t), \tag{I.1}$$

where $f(x,y)$ is the input image, $(2a+1) \times (2b+1)$ is the kernel size, and $w(0,0)$ corresponds to the kernel center. This formulation captures the essence of linear spatial filtering: each output pixel is a linear combination of intensities in its spatial neighborhood.

Re-indexing the kernel in non-centered coordinates yields the more common convolution notation:

$$g(x,y) = \sum_{i} \sum_{j} h(i,j)\, f(x-i,\, y-j), \tag{I.2}$$

where $h(i,j)$ is the same kernel expressed with positive index offsets. Both forms describe the same weighted sum-of-products operation that underlies all linear convolution-based smoothing methods.

Linear smoothing effectively suppresses high-frequency noise but, due to its fixed kernel

weights, inevitably smooths across intensity transitions, causing blurring at edges and reducing fine detail.

The two linear filters examined in this assignment are:

- **Uniform (average) filter** - equal weights across the neighborhood.
- **Gaussian low-pass filter** - weights decay with distance, producing smoother transitions.

2.2. Nonlinear and Adaptive Smoothing (Non-Convolution)

Not all smoothing operations can be expressed as convolution. Nonlinear and adaptive methods modify pixel intensities using rules that depend on statistics, intensity ordering, or local structural patterns. As a result, the effective kernel varies from pixel to pixel, and the filtering process is *spatially varying*.

These methods are often better suited for noise models that violate linear assumptions, such as impulse noise, or for tasks requiring edge preservation.

The nonlinear and adaptive techniques implemented in this assignment include:

- **Median filter**
- **Conditional averaging filter**
- **Gradient-weighted averaging**
- **Rotating mask filter**
- **MMSE filter**

2.3. Denoising Objective

The overarching objective of all smoothing techniques is to estimate the unknown clean image $f(x, y)$ from a noisy observation $g(x, y)$. Different noise sources require different filtering strategies; in this assignment, two noise models are considered:

- **Salt-and-pepper noise** (impulse noise),
- **Additive Gaussian noise**.

3. Objectives

The main objectives of this assignment are:

- Implement a suite of spatial smoothing filters using NumPy, strictly following mathematical definitions.
- Evaluate how each filter behaves under different noise conditions (Gaussian vs Salt & Pepper).
- Analyze trade-offs involving noise reduction, edge preservation, and computational efficiency.
- Compare custom implementations with OpenCV's optimized filters (`cv2.blur`, `cv2.- GaussianBlur`, `cv2.medianBlur`, `cv2.bilateralFilter`) to validate correctness.

4. Program Structure

The program reads an input image, generates noisy versions, applies all filters, and saves outputs for comparison.

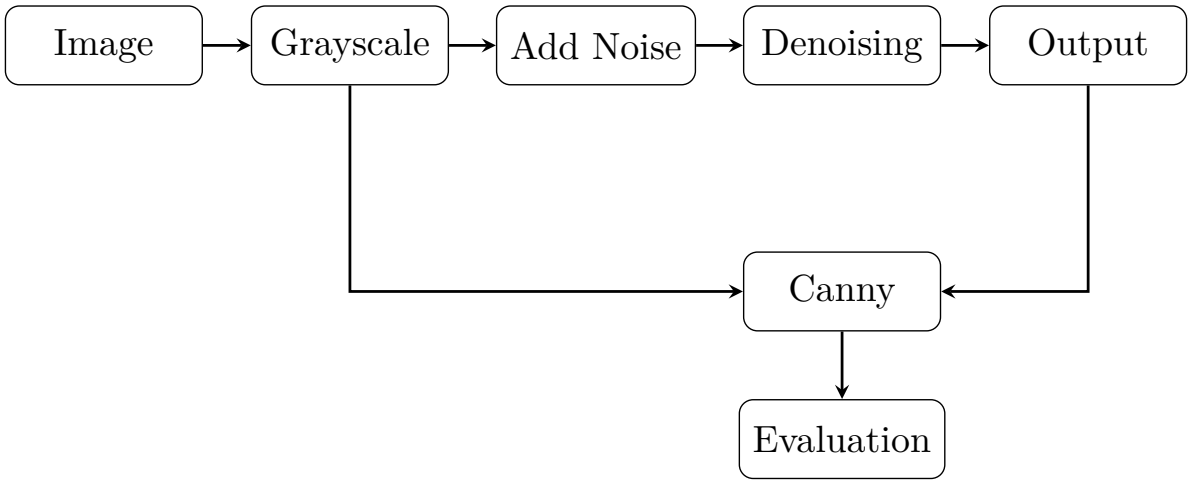
Input

- A single image file (PNG/JPG/JPEG) specified by the user or default Lena image.

Output

- Preprocessed noisy images,
- Filtered results for each smoothing method,
- Comparison tables and evaluation metrics.

The implementation is modularized: a base smoothing class provides shared utilities, specialized subclasses implement linear filters (mean, Gaussian, median) and adaptive filters (gradient-weighted, rotating mask, MMSE), while conditional filters for impulse and Gaussian noise reside in a dedicated conditional module.



Hình I.2: Processing Pipeline

Chapter II: Filtering Algorithms for Image Denoising

1. Average Filtering

1.1. Theory

Average filtering is a spatial smoothing technique in which each pixel value is replaced by the arithmetic mean of the intensities in a small neighborhood around that pixel. This tends to reduce random noise fluctuations but also introduces blurring of edges and fine details.

1.2. Method

Given a noisy grayscale image $g(x, y)$ and an odd kernel size k, average filtering in the spatial domain can be implemented as follows:

- **Step 1: Define the neighborhood size.** Choose an odd kernel size k and set

$$r = \frac{k-1}{2},$$

so that the square neighborhood of each pixel (x, y) is

$$\mathcal{N}_k(x, y) = \{(i, j) \mid x - r \leq i \leq x + r,\ y - r \leq j \leq y + r\}.$$

- **Step 2: Construct the average kernel.** The discrete average filter uses a uniform kernel

$$h(i, j) = \frac{1}{k^2}, \qquad (i, j) \in [-r, r] \times [-r, r].$$

For example, for a 3×3 kernel ($k = 3$, $r = 1$), we have [3]

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

- **Step 3: Apply convolution.** Pad $g(x, y)$ (e.g., by reflection) to obtain $\tilde{g}(x, y)$, then compute the average-filtered image as the discrete convolution

$$g_{\text{avg}}(x, y) = (h * g)(x, y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} h(i, j)\, \tilde{g}(x - i, y - j).$$

The resulting image $f_{\text{avg}}(x, y)$ has lower local variance than the input $g(x, y)$, because each output value is an average of k^2 noisy samples in the neighborhood, but the uniform weighting also causes noticeable edge blurring.

1.3. Pseudocode

```
Function mean_filter(image, kernel_size):
    r = floor(kernel_size / 2)
    Pad image with reflection to obtain image_pad
    Initialize output mean_img with zeros, same size as image

    For each pixel (x, y) in image:
        S = 0
        For i = -r to r:
            For j = -r to r:
                S += image_pad(x + i, y + j)
        mean_img(x, y) = S / (kernel_size * kernel_size)

    Return mean_img
```

2. Gaussian Filtering

2.1. Theory

Gaussian filtering smooths an image by averaging neighboring pixels with weights that follow a Gaussian distribution. The kernel is defined as [3]

$$h(i,j) = \frac{1}{\sqrt{2\pi}\,\sigma}\, e^{-\frac{i^2+j^2}{2\sigma^2}}$$

where σ controls the spread of smoothing. The filtered image is obtained by spatial convolution:

$$g = h * f.$$

Because Gaussian kernels act as low-pass filters, Gaussian smoothing is particularly effective for reducing additive Gaussian noise.

2.2. Method

Gaussian smoothing is applied to a noisy grayscale image $f(x,y)$ through the following steps:

- **Step 1: Choose kernel size.** Select an odd kernel size k. A common rule is

  $$k = \lceil 6\sigma \rceil + 1,$$

  ensuring the kernel captures significant Gaussian support.

- **Step 2: Construct the Gaussian kernel.** Let $r = (k-1)/2$. For each $(i,j) \in [-r,r]\times[-r,r]$ compute

  $$h(i,j) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{i^2+j^2}{2\sigma^2}\right).$$

  Normalize to ensure

  $$\sum_{i=-r}^{r} \sum_{j=-r}^{r} h(i,j) = 1.$$

- **Step 3: Example (3×3 kernel).** For $\sigma = 1$ and $k = 3$:

$$h = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

- **Step 4: Apply convolution.** Pad the image (e.g., reflection) to obtain $\tilde{f}(x, y)$, then compute

$$g(x, y) = (h * f)(x, y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} h(i, j)\, \tilde{f}(x - i,\, y - j).$$

2.3. Pseudocode

```
Function gaussian_filter(image, kernel_size, sigma):
    r = floor(kernel_size / 2)
    Construct coordinates (y, x) in [-r, r] x [-r, r]
    kernel = exp(-(x^2 + y^2) / (2 * sigma^2))
    Normalize kernel so sum(kernel) = 1

    Pad image with reflection to obtain image_pad
    g = convolve(image_pad, kernel)  # compute only on valid
   region
    Return g
```

3. Median Filtering

3.1. Theory

Median filtering is a nonlinear smoothing operation that replaces each pixel with the median intensity of its local neighborhood. Because the median is insensitive to isolated extreme values, this filter is particularly effective for removing salt–and–pepper noise while preserving edges.

3.2. Method

Given a noisy grayscale image $f(x, y)$ and an odd kernel size k, median filtering is performed in the spatial domain as follows:

- **Step 1: Define the neighborhood.** Choose an odd kernel size k and set

$$r = \frac{k-1}{2}.$$

  For each pixel (x, y), define the $k \times k$ neighborhood

$$\mathcal{N}_k(x, y) = \{(i, j) \mid x - r \leq i \leq x + r,\ y - r \leq j \leq y + r\}.$$

- **Step 2: Handle image boundaries.** Pad $f(x, y)$ (e.g., using reflection) to obtain $\tilde{f}(x, y)$ so that $\mathcal{N}_k(x, y)$ is well-defined for all (x, y) in the original image domain.

- **Step 3: Form the sample set.** For each pixel (x, y), collect the intensities within the neighborhood into a set

$$S_{x,y} = \{\tilde{f}(i, j) \mid (i, j) \in \mathcal{N}_k(x, y)\}.$$

  This set contains k^2 values.

- **Step 4: Compute the median.** Sort the elements of $S_{x,y}$ in nondecreasing order,

$$S_{x,y} = \{v_1 \leq v_2 \leq \cdots \leq v_{k^2}\},$$

  and assign the median value to the output:

$$g(x, y) = v_{(k^2+1)/2}.$$

The resulting image $g(x, y)$ has most impulse outliers suppressed, while edges and structures are better preserved than with simple averaging for the same kernel size.

3.3. Pseudocode

```
Function median_filter(image, k):
Compute r = floor(k / 2)
For each pixel (x, y):
window = []
For i = -r to r:
For j = -r to r:
window.append(image(x + i, y + j))
g(x, y) = median(window)
Return g
```

4. Conditional Averaging

4.1. Theory

Conditional averaging is a nonlinear smoothing operator in which only 'reliable' neighbors inside a window contribute to the output. Instead of averaging all pixels, the filter first applies a *selection rule* to reject corrupted samples, then averages the remaining ones. In this work we consider two variants: a range-based operator designed for impulse (salt & pepper) noise and a difference-based operator tailored to Gaussian noise.

4.2. Range-Based Conditional Averaging

4.2.1 Method

Given an input image $f(x, y)$ and a $k \times k$ window $\mathcal{N}_k(x, y)$:

- **Step 1:** Choose a global valid intensity interval $[\min, \max]$.
- **Step 2:** For each neighbor $(x + i, y + j) \in \mathcal{N}_k(x, y)$, keep it only if its gray level lies inside this interval.
- **Step 3:** If the center pixel $f(x, y)$ is outside $[\min, \max]$, replace it by the mean of all retained neighbors; otherwise keep it.

The selection function is defined as [3]

$$h(i,j) = \begin{cases} 1, & f(x+i, y+j) \in [\min, \max], \\ 0, & \text{otherwise.} \end{cases}$$

The filtered output is

$$g(x,y) = \frac{\sum_{(i,j)\in\mathcal{N}_k(x,y)} h(i,j)\, f(x+i, y+j)}{\sum_{(i,j)\in\mathcal{N}_k(x,y)} h(i,j)}.$$

4.2.2 Pseudocode (Range-Based Variant)

```
Function conditional_range(image, k, min_val, max_val):
    r = floor(k / 2)
    For each pixel (x, y):
        if image[x, y] in [min_val, max_val]:
            g[x, y] = image[x, y]      # keep valid center
        else:
            S = 0; count = 0
            For i = -r to r:
                For j = -r to r:
                    v = image[x+i, y+j]
                    If v in [min_val, max_val]:
                        S += v; count += 1
            g[x, y] = S / count        # replace only out-of-
   range center
    Return g
```

4.2.3 Limitation

(a) Conditional range on Salt & Pepper noise with kernel size 3x3.

(b) Conditional range on Gaussian noise with kernel size 3x3.

As illustrated in Figures II.1a and II.1b, the range-based conditional averaging method assumes that a single global intensity interval $[\min, \max]$ can effectively separate valid pixels from corrupted impulses.

For Salt & Pepper noise, this assumption holds well: outliers near 0 or 255 are clearly distinguishable from valid pixels, enabling the filter to selectively exclude corrupted samples and achieve effective noise removal, as demonstrated in Figure II.1a.

However, for Gaussian noise, this approach becomes ineffective because most pixels remain within the same global intensity range, making it impossible to distinguish noise from signal using a fixed threshold. As a result, the filter barely reduces the noise level, as evident in Figure II.1b. This limitation is further illustrated in the comprehensive comparison shown in Figure IV.4.

4.3. Difference-Based Conditional Averaging

To overcome the dependence on a fixed global range, we modify the selection step so that it is *relative to the center pixel* instead of absolute in the gray-level axis.

4.3.1 Method

Given a threshold T:

- **Step 1:** For each center pixel $f(x, y)$, scan the window $\mathcal{N}_k(x, y)$.
- **Step 2:** Keep neighbors whose intensity differs from the center by less than T.
- **Step 3:** Replace the center by the mean of all selected neighbors.

The similarity-based selection rule is

$$\phi(f(i,j), f(x,y)) = \begin{cases} 1, & |f(i,j) - f(x,y)| < T, \\ 0, & \text{otherwise}, \end{cases}$$

with selected set

$$\mathcal{S}(x,y) = \{(i,j) \in \mathcal{N}_k(x,y) \mid |f(i,j) - f(x,y)| < T\}.$$

The output is

$$g(x,y) = \frac{1}{|\mathcal{S}(x,y)|} \sum_{(i,j)\in\mathcal{S}(x,y)} f(i,j).$$

4.3.2 Pseudocode (Difference-Based Variant)

```
Function conditional_diff(image, k, T):
    r = floor(k / 2)
    For each pixel (x, y):
        S = 0; count = 0
        c = image[x, y]
        For i = -r to r:
            For j = -r to r:
                v = image[x+i, y+j]
                If abs(v - c) < T:
                    S += v; count += 1
        g[x, y] = S / count
    Return g
```

5. Gradient-Weighted Averaging

5.1. Theory

Gradient-weighted averaging is a smoothing technique in which neighboring pixels are weighted according to their similarity to the center pixel. Pixels whose intensities are close to the center receive larger weights, while dissimilar pixels receive smaller ones. This produces smoothing within homogeneous regions while preserving edges. The weights are computed from intensity differences and normalized to form a valid averaging kernel.

5.2. Method

- **Step 1: Compute similarity weights $\delta(i,j)$.**

  For a center pixel $f(m,n)$ and a neighbor $f(i,j)$, the similarity measure is defined as

$$\delta(i,j) = \begin{cases} \dfrac{1}{|f(m,n) - f(i,j)|}, & \text{if } f(m,n) \neq f(i,j), \\ 2, & \text{if } f(m,n) = f(i,j). \end{cases}$$

  with the constraint

$$\delta(i,j) \in (0,2].$$

- **Step 2: Normalize weights to obtain kernel coefficients $h(i,j)$.** /

  For a neighborhood $O(m,n)$,

$$h(i,j) = 0.5 \cdot \frac{\delta(i,j)}{\sum\limits_{(i',j') \in O(m,n)} \delta(i',j')}, \qquad h(m,n) = 0.5.$$

  The weights satisfy the normalization condition

$$\sum_{(i,j) \in O(m,n)} h(i,j) = 1.$$

[3]

- **Step 3: Compute the filtered output.**

  The smoothed pixel value is computed as

$$g(m,n) = \sum_{(i,j)\in O(m,n)} h(i,j)\, f(i,j).$$

5.3. Pseudocode

```
Function gradient_weighted_average(image, k, eps):
Compute gradients gx, gy
Compute r = floor(k / 2)
For each pixel (x, y):
numerator = 0
denominator = 0
For i = -r to r:
For j = -r to r:
gmag = sqrt(gx(x+i, y+j)^2 + gy(x+i, y+j)^2)
w = 1 / (gmag + eps)
numerator += w * image(x+i, y+j)
denominator += w
g(x, y) = numerator / denominator
Return g
```

6. Rotating Mask Filtering

6.1. Theory

The rotating–mask operator computes the output

$$g(x,y) = \mathcal{F}[f(x,y)]$$

by selecting, among several directional masks, the one whose local variance is minimal.

Let $\{\mathcal{M}_p\}$ be a set of masks centered at (x, y). For each mask $\mathcal{M}_p$, define the local mean

$$\mu_p(x, y) = \frac{1}{|\mathcal{M}_p|} \sum_{(i,j)\in\mathcal{M}_p} f(i, j),$$

and the local variance

$$\sigma_p^2(x, y) = \frac{1}{|\mathcal{M}_p|} \sum_{(i,j)\in\mathcal{M}_p} \left(f(i, j) - \mu_p(x, y)\right)^2 .$$

The optimal mask is the one with minimum variance:

$$p^* = \arg\min_p \ \sigma_p^2(x, y).$$

Thus the filtering operator becomes

$$g(x, y) = \mu_{p^*}(x, y),$$

which preserves directional edges by smoothing only along the least–variant orientation.

6.2. Method

Given K directional masks $\mathcal{M}_1, \ldots, \mathcal{M}_K$, the filtering at pixel (x, y) is performed through the following mathematical steps:

- **Step 1:** Extract mask samples:

$$S_p(x, y) = \{\, f(i, j) \mid (i, j) \in \mathcal{M}_p \,\}.$$

- **Step 2:** Compute mean of mask p:

$$\mu_p(x, y) = \frac{1}{|S_p(x, y)|} \sum_{v\in S_p(x,y)} v.$$

- **Step 3:** Compute variance of mask p:

$$\sigma_p^2(x, y) = \frac{1}{|S_p(x, y)|} \sum_{v\in S_p(x,y)} (v - \mu_p(x, y))^2.$$

[3]

- **Step 4:** Select the mask with minimal variance:

$$p^* = \arg\min_p \sigma_p^2(x, y).$$

- **Step 5:** Assign filtered value:

$$g(x, y) = \mu_{p^*}(x, y).$$

6.3. Pseudocode

```
Function rotating_mask_filter(image, masks):
  For each pixel (x, y):
    best_var = infinity
    best_mean = 0
    For each mask M in masks:
        values = sample(image, M around (x, y))
        mu  = mean(values)
        var = variance(values)
        If var < best_var:
            best_var = var
            best_mean = mu
    g(x, y) = best_mean
  Return g
```

7. MMSE Filtering

7.1. Theory

The MMSE operator computes

$$g(x, y) = \mathcal{F}[f(x, y)]$$

by optimally estimating the clean signal under additive noise

$$f(x,y) = u(x,y) + n(x,y), \qquad \mathbb{E}[n] = 0, \quad \mathrm{Var}(n) = \sigma_n^2.$$

Let $\mu(x,y)$ and $\sigma^2(x,y)$ denote the local mean and variance within a window centered at (x,y). The MMSE estimator is a linear shrinkage of the noisy pixel toward the local mean:

$$g(x,y) = f(x,y) - \frac{\sigma_n^2}{\sigma^2(x,y)} \left(f(x,y) - \mu(x,y)\right).$$

Equivalently,

$$g(x,y) = \mu(x,y) + \left(1 - \frac{\sigma_n^2}{\sigma^2(x,y)}\right) \left(f(x,y) - \mu(x,y)\right).$$

When the local variance is small (flat region), the estimator pulls strongly toward $\mu(x,y)$; near edges (large variance), the correction becomes weak.

7.2. Method

For each pixel (x,y), the MMSE filtering consists of the following mathematical steps:

- **Step 1:** Collect the local window:

$$W(x,y) = \{\, f(i,j) \mid (i,j) \in \text{window around } (x,y) \,\}.$$

- **Step 2:** Compute local mean:

$$\mu(x,y) = \frac{1}{|W(x,y)|} \sum_{v \in W(x,y)} v.$$

- **Step 3:** Compute local variance [3]:

$$\sigma^2(x,y) = \frac{1}{|W(x,y)|} \sum_{v \in W(x,y)} (v - \mu(x,y))^2.$$

- **Step 4:** Compute MMSE shrinkage coefficient:

$$\alpha(x,y) = 1 - \frac{\sigma_n^2}{\sigma^2(x,y)}.$$

- **Step 5:** Assign output:

$$g(x,y) = \mu(x,y) + \alpha(x,y)\Big(f(x,y) - \mu(x,y)\Big).$$

7.3. Pseudocode

```
Function mmse_filter(image, k, sigma_n2):
  r = floor(k / 2)
  For each pixel (x, y):
      window = collect neighborhood of size k x k
      mu  = mean(window)
      var = variance(window)
      alpha = 1 - (sigma_n2 / var)
      g(x, y) = mu + alpha * (image(x, y) - mu)
  Return g
```

Chapter III: Installation and Usage

This section describes how to set up the environment, install required libraries, and execute the program in all three supported run modes.

1. Environment Setup

1.1. System Requirements

This spatial smoothing filters project requires the following system components:

- Python 3.7 or higher;
- RAM: Minimum 4GB (recommended 8GB);
- Disk space: Approximately 100MB for dependencies and outputs.

1.2. Python and Library Installation

To set up the development environment, perform the following steps:

a) Create a Virtual Environment.

Create a virtual environment to isolate the project's dependencies.

```
python -m venv .venv
```

Activate the virtual environment:

```
# Windows:
.venv\Scripts\activate

# Linux/Mac:
source .venv/bin/activate
```

b) Install Required Libraries.

Install the dependencies using the requirements.txt file:

```
pip install -r requirements.txt
```

Alternatively, install packages individually:

```
pip install numpy>=1.21.0 opencv-python>=4.5.0 Pillow>=9.0.0
```

1.3. Main Libraries

The project uses the following Python libraries:

- `numpy`: Multidimensional array processing and mathematical operations for image filtering;
- `opencv-python`: Image processing, I/O operations, and baseline filter implementations;
- `Pillow`: Enhanced text rendering for visualization (optional but recommended).

1.4. Default Test Image

The program uses the Lena image as the default test image. If the image is not found in the `assets/` directory (as `Lena.jpg` or `Lenna.jpg`), it will be automatically downloaded from a remote server. Alternatively, users can provide their own image using the `-image` command-line argument.

1.5. Installation Check

To verify that the installation was successful, run the main script:

```
cd src
python main.py
```

If the installation is correct, the program will process the default Lena image and generate output files in the `outputs/` directory.

2. How to Run the Program

2.1. Run Main Processing Pipeline

Script `main.py` processes images with spatial smoothing filters, generates noisy versions, applies all filters, and creates comparison tables.

a) Run with default image.

By default, the script uses the Lena image from `assets/Lena.jpg` (downloads automatically if missing) and saves results to `outputs/`:

```
cd src
python main.py
```

b) Run with custom image.

Specify the input image path:

```
python main.py --image path/to/your_image.jpg
```

c) Specify custom output directory.

Change the output directory:

```
python main.py --out_dir custom_outputs
```

d) Combine options.

Run with custom image and output directory:

```
python main.py --image data/camera.png --out_dir results_test
```

2.2. Run Evaluation Script

Script `evaluation.py` provides quantitative evaluation of all filters with metrics including MSE, PSNR, and edge preservation.

a) Run evaluation.

Execute the evaluation script from the `src/` directory:

```
cd src
python evaluation.py
```

b) Output.

The evaluation script generates:

- Filtered images saved to `outputs/eval/`;
- Summary tables printed to console showing:
  - Metrics table: MSE, PSNR, and edge preservation (IoU) for each filter;
  - Execution time table: processing time for each filter with ranking.

c) Evaluation metrics.

The script computes the following metrics for each filter:

- Mean Squared Error (MSE): Measures pixel-wise difference between original and filtered images;
- Peak Signal-to-Noise Ratio (PSNR): Measures image quality in decibels;
- Edge preservation: Evaluates edge preservation using Canny edge detection and IoU metric.

3. Project Directory Structure

3.1. Project Directory Structure

The project is organized as follows:

```
src/
|-- main.py              # Entry point: SmoothingApp class
|-- smoothing/           # Package: base + derived filters + facade
|   |-- __init__.py      # SpatialSmoothing facade
|   |-- base.py          # BaseSmoothing helpers
|   |-- mean.py          # MeanSmoothing
```

```
|   |-- gaussian.py       # GaussianSmoothing
|   |-- median.py         # MedianSmoothing
|   |-- adaptive.py       # AdaptiveSmoothing
|   |-- conditional.py    # ConditionalSmoothing
|-- utils.py              # Utilities
|-- evaluation.py         # Evaluation script with quantitative metrics
|-- requirements.txt      # Python dependencies
|-- readme.md             # Project documentation
|-- assets/
|   |-- Lenna.jpg         # Default test image (auto-downloaded if missing)
|-- outputs/              # Generated outputs
```

3.2. Key Classes and Functions

3.2.1 SmoothingApp Class (main.py)

The `SmoothingApp` class serves as the main application wrapper for running the smoothing pipeline.

Method	Description
`__init__`	Initialize app and parse command-line arguments.
`_parse_args`	Parse CLI arguments (`-image`, `-out_dir`).
`_fetch_default_lena`	Ensure default Lena image exists locally; download if missing.
`_load_image`	Read image as BGR format; raise error if not found.
`_choose_image`	Resolve image path: CLI provided or default Lena.
`_run_pipeline`	Add noise, run all smoothing filters, and write outputs and comparisons.
`run`	Execute the app.

Bảng III.1: SmoothingApp Class Methods

3.2.2 SpatialSmoothing Facade (smoothing/)

Method	Description
`mean`	Mean/average filter with given odd kernel size.
`gaussian`	Gaussian smoothing with odd kernel size and sigma parameter.
`median`	Median filter using odd window size.
`conditional`	Difference-based conditional filter (modern style, threshold-based).
`conditional_range_impulse`	Range-based conditional filter for impulse noise (textbook style, replaces pixels outside range).
`conditional_range_smooth`	Range-based smoothing filter (textbook style, smooths all pixels using valid neighbors).
`gradient_weighted`	Gradient/gray-level weighted averaging with bilateral-style weights.
`gradient_weighted_impulse`	Impulse-robust gradient weighting using local median and thresholding.
`rotating_mask`	Rotating mask averaging using variance-based orientation selection (kernel_size=3 only).
`mmse`	MMSE filter using local mean/variance and known noise variance.

Bảng III.2: SpatialSmoothing Facade Methods

3.2.3 Utility Classes and Functions (utils.py)

Utility classes and functions for filesystem handling, noise generation, image processing, and visualization.

Class/Function	Description
`FileIOHandle`	Handle filesystem layout for outputs and common save helpers.
`FileIOHandle.save_single_image`	Save one filtered image into structured result directories.
`ensure_dir`	Create directory (and parents) if missing.
`add_noise`	Add synthetic noise for filter testing (Gaussian or salt & pepper).
`_to_bgr`	Ensure 3-channel BGR format for visualization.
`make_tile`	Resize and add header label for grid visualization.
`save_grid`	Save a labeled grid of images with uniform tile size and padding.
`_render_text_pil`	Render text using PIL for better font support.
`save_table_grid`	Save a table-style grid with header row and kernel size column.
`_save_comparison_grid`	Save comparison items in a grid layout for adaptive filters.

Bảng III.3: Utility Classes and Functions

3.2.4 Evaluation Functions (evaluation.py)

Functions for quantitative evaluation of filters with metrics.

Function	Description
`mse`	Calculate Mean Squared Error between two images.
`psnr`	Calculate Peak Signal-to-Noise Ratio between two images.
`detect_edges`	Detect edges in image using Canny or Sobel method.
`evaluate_edge_preservation`	Evaluate edge preservation capability using edge detection and IoU metric.
`load_filtered_image`	Load filtered image from result directory for evaluation.
`format_filter_info`	Format filter information string with parameters.
`main`	Run evaluation for all filters and print summary tables.

Bảng III.4: Evaluation Functions

3.2.5 Main Scripts

Main scripts to run the program:

Script	Description
`main.py`	Entry point for processing images with all spatial smoothing filters, generates noisy versions, applies filters, and creates comparison tables with OpenCV baselines.
`evaluation.py`	Script for quantitative evaluation of all filters with metrics (MSE, PSNR, edge preservation) and execution time analysis.

Bảng III.5: Main Scripts

Chapter IV: Results and Evaluation

1. Evaluation Methodology

This section describes the evaluation methodology used to assess the performance of spatial smoothing filters. The evaluation process involves both quantitative metrics and qualitative visual assessment to comprehensively analyze filter effectiveness.

1.1. Experimental Setup

The evaluation pipeline processes images with two types of noise: Salt & Pepper noise (5% pixel corruption) and Gaussian noise (standard deviation $\sigma = 12.0$). All filters are tested with a 3×3 kernel size to ensure fair comparison, except for the MMSE filter which uses a 5×5 kernel as specified in its implementation. The original grayscale image serves as the ground truth for quantitative metric computation.

1.2. Quantitative Metrics

To objectively evaluate filter performance, three categories of metrics are employed: reconstruction quality metrics, edge preservation metrics, and computational efficiency metrics.

1.2.1 Mean Squared Error (MSE)

The Mean Squared Error measures the average squared difference between the filtered image and the original image. It is defined as [2]:

$$\text{MSE} = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - \hat{I}(i,j)]^2$$

where $I(i,j)$ represents the pixel value at position (i,j) in the original image, $\hat{I}(i,j)$ is the corresponding pixel in the filtered image, and $M \times N$ is the image size. Lower MSE values indicate better reconstruction quality, as the filtered image is closer to the original.

1.2.2 Peak Signal-to-Noise Ratio (PSNR)

The Peak Signal-to-Noise Ratio provides a logarithmic measure of image quality relative to the maximum possible pixel value. PSNR is computed as [2]:

$$\text{PSNR} = 20 \log_{10} \left(\frac{\text{MAX}_I}{\sqrt{\text{MSE}}} \right)$$

where MAX_I is the maximum possible pixel value (typically 255 for 8-bit images). PSNR is expressed in decibels (dB), with higher values indicating better image quality. A PSNR value above 30 dB is generally considered good quality, while values below 20 dB indicate significant degradation.

1.2.3 Edge Preservation Evaluation

To assess the ability of filters to preserve important image structures, edge preservation is evaluated using edge detection and Intersection over Union (IoU) metrics. The evaluation process involves [2]:

- **Step 1:** Detecting edges in the original image using the Canny edge detector with thresholds (50, 150), producing the ground truth edge map E_{gt}.
- **Step 2:** Detecting edges in the filtered image using the same Canny parameters, producing the filtered edge map E_{filtered}.
- **Step 3:** Computing the Intersection over Union (IoU) between the two edge maps:

$$\text{Edge IoU} = \frac{|E_{\text{gt}} \cap E_{\text{filtered}}|}{|E_{\text{gt}} \cup E_{\text{filtered}}|}$$

The IoU metric ranges from 0 to 1, where 1 indicates perfect edge preservation and 0 indicates complete edge loss. Higher Edge IoU values indicate better preservation of image structures and boundaries.

Additionally, edge preservation is assessed through:

- **Edge count ratio**: The ratio of detected edge pixels in the filtered image to those in the original, indicating whether edges are preserved or lost.

- **Visual inspection**: Qualitative assessment of whether edge boundaries remain sharp and continuous or become blurred or broken.

1.2.4 Computational Efficiency

Execution time is measured for each filter to assess computational efficiency. The timing measurement captures the total processing time for both Salt & Pepper and Gaussian noise cases, providing insight into the practical applicability of each method.

1.3. Filter Implementations

The evaluation includes both custom implementations and OpenCV baseline methods. Custom filters are implemented from first principles, while OpenCV filters serve as optimized reference implementations. All filters are configured with identical parameters to ensure fair comparison:

- **Mean/Average Filter**: Uniform averaging within a 3×3 neighborhood.
- **Gaussian Filter**: Gaussian-weighted averaging with $\sigma = 1.0$ and 3×3 kernel.
- **Median Filter**: Non-linear filtering using median value in 3×3 neighborhood.
- **Conditional Averaging**: Adaptive filtering with threshold range [80, 200] for 3×3 kernel.
- **Gradient-weighted Averaging**: Weighted averaging based on local gradient magnitude, 3×3 kernel.
- **Rotating Mask Averaging**: Directional averaging using rotating mask patterns, 3×3 kernel.
- **MMSE Filter**: Minimum Mean Squared Error filter with noise variance $\sigma_n^2 = 20.0$, 5×5 kernel.
- **OpenCV Baselines**: `cv2.blur`, `cv2.GaussianBlur`, `cv2.medianBlur`, and `cv2.bilateralFilter` with matching parameters.

The evaluation process systematically applies each filter to both noise-corrupted images, computes all metrics, and saves individual filtered images for visual inspection. This comprehensive approach enables both quantitative ranking and qualitative assessment of filter performance.

2. Qualitative Comparison

This section presents qualitative visual comparisons of filter outputs, examining how each method handles different noise types and preserves image details.

2.1. Mean Filter

The mean filter applies uniform averaging within a local neighborhood, effectively reducing noise by replacing each pixel with the average of its neighbors. As shown in the results, the mean filter demonstrates moderate noise reduction capabilities. For Salt & Pepper noise, the filter achieves MSE of 198.08 and PSNR of 25.16 dB, while for Gaussian noise, it achieves MSE of 102.61 and PSNR of 28.02 dB. The filter effectively smooths both noise types but tends to blur image details and edges, resulting in Edge IoU values of 0.266 for Salt & Pepper and 0.428 for Gaussian noise. The uniform averaging process reduces high-frequency components, including both noise and fine image structures.

(a) Mean filter on Salt & Pepper noise. (b) Mean filter on Gaussian noise.

Hình IV.1: Mean filter results.

As illustrated in Figure IV.1, the mean filter successfully removes noise artifacts but introduces significant blurring, particularly visible in edge regions. The uniform averaging operation

treats all pixels equally, leading to loss of fine details and edge sharpness.

2.2. Gaussian Filter

The Gaussian filter employs weighted averaging with a Gaussian kernel, providing better edge preservation compared to the mean filter. The Gaussian weighting emphasizes central pixels while smoothly decreasing influence toward the kernel periphery. Results show MSE of 187.79 (S&P) and 81.76 (Gaussian), with corresponding PSNR values of 25.39 dB and 29.01 dB. The Edge IoU values of 0.279 (S&P) and 0.476 (Gaussian) indicate improved edge preservation compared to the mean filter. The Gaussian weighting provides a good balance between noise suppression and detail retention, making it suitable for applications requiring smooth transitions while maintaining structural information.

(a) Gaussian filter on Salt & Pepper noise. (b) Gaussian filter on Gaussian noise.

Hình IV.2: Gaussian filter results.

Figure IV.2 demonstrates the Gaussian filter's superior edge preservation compared to uniform averaging. The smooth weighting function reduces noise while maintaining sharper boundaries, resulting in visually more appealing results.

2.3. Median Filter

The median filter, a non-linear operator, excels at removing Salt & Pepper noise while preserving edges. By selecting the median value within the neighborhood, the filter effectively removes impulse noise without significantly affecting edge pixels. The results demonstrate excellent performance for Salt & Pepper noise: MSE of 59.45 and PSNR of 30.39 dB, with Edge IoU of 0.509. For Gaussian noise, performance is moderate with MSE of 87.31, PSNR of 28.72 dB, and Edge IoU of 0.419. The median filter's strength lies in its ability to remove outliers (impulse noise) while maintaining sharp boundaries, making it particularly effective for Salt & Pepper noise removal.

(a) Median filter on Salt & Pepper noise. (b) Median filter on Gaussian noise.

Hình IV.3: Median filter results.

As shown in Figure IV.3, the median filter produces clean results for Salt & Pepper noise with minimal edge degradation. The non-linear operation effectively removes impulse noise while preserving edge structures, demonstrating the filter's robustness for impulse noise scenarios.

2.4. Conditional Averaging

The conditional averaging filter applies averaging only when pixel values fall within a specified threshold range, providing adaptive noise reduction. However, the results indicate limited

effectiveness: MSE of 3820.96 (S&P) and 484.44 (Gaussian), with low PSNR values of 12.31 dB and 21.28 dB respectively. The Edge IoU values of 0.151 (S&P) and 0.158 (Gaussian) are the lowest among all evaluated filters. The threshold-based approach appears too restrictive, causing significant information loss and poor reconstruction quality. The filter struggles to distinguish between noise and image content, leading to excessive smoothing or insufficient noise removal.

Hình IV.4: Comparison between conditional-range and conditional-diff on both noise types.

As shown in Figure IV.4, between the two variants, conditional-range provides more stable

and superior quality. For Salt & Pepper noise, it removes outliers more effectively and reconstructs flat regions better while preserving natural edges. For Gaussian noise, it continues to demonstrate advantages through its value-range selection mechanism, reducing noise grains while limiting excessive blurring.

In contrast, conditional-diff strongly depends on local deviation, making it prone to miss salt-pepper noise and only achieving moderate Gaussian noise reduction. Its effectiveness is less stable and tends to retain more residual noise.

In conclusion, conditional-range is the stronger and more consistent variant across both noise types; conditional-diff is only suitable when prioritizing structure preservation and accepting lower noise reduction levels.

2.5. Gradient-weighted Averaging

The gradient-weighted averaging filter adjusts smoothing strength based on local gradient magnitude, providing stronger smoothing in uniform regions and preserving edges in high-gradient areas. Results show MSE of 906.00 (S&P) and 136.63 (Gaussian), with PSNR values of 18.56 dB and 26.78 dB. Edge IoU values of 0.316 (S&P) and 0.332 (Gaussian) indicate moderate edge preservation. The adaptive weighting mechanism helps preserve edges but may not provide sufficient noise reduction in low-gradient regions, resulting in moderate overall performance.

(a) Gradient-weighted averaging on S&P noise. (b) Gradient-weighted averaging on Gaussian noise.

Hình IV.5: Gradient-weighted averaging results.

The adaptive weighting preserves edges but shows limited noise reduction effectiveness. Figure IV.5 shows the gradient-weighted averaging results.

2.6. Rotating Mask Averaging

The rotating mask filter applies directional averaging using multiple mask orientations, providing enhanced edge preservation through directional selectivity. Results demonstrate good performance: MSE of 165.00 (S&P) and 80.71 (Gaussian), with PSNR values of 25.96 dB and 29.06 dB. Edge IoU values of 0.382 (S&P) and 0.413 (Gaussian) show improved edge preservation compared to simple averaging methods. The rotating mask approach effectively preserves edges aligned with mask directions while providing noise reduction, making it suitable for images with dominant edge orientations.

(a) Rotating mask on Salt & Pepper noise. (b) Rotating mask on Gaussian noise.

Hình IV.6: Rotating mask results.

Figure IV.6 shows that the rotating mask filter achieves balanced performance, effectively reducing noise while preserving edge information through directional processing.

2.7. MMSE Filter

The Minimum Mean Squared Error filter optimizes the trade-off between noise reduction and detail preservation based on estimated noise statistics. Using a 5×5 kernel with noise variance $\sigma_n^2 = 20.0$, the filter achieves MSE of 890.52 (S&P) and 123.48 (Gaussian), with PSNR values of 18.63 dB and 27.21 dB. Edge IoU values of 0.313 (S&P) and 0.371 (Gaussian) indicate moderate edge preservation. The MMSE approach theoretically optimizes reconstruction quality, but practical performance depends on accurate noise variance estimation and may be sensitive to noise model assumptions.

(a) MMSE filter on Salt & Pepper noise.

(b) MMSE filter on Gaussian noise.

Hình IV.7: MMSE filter results.

The MMSE filter results in Figure IV.7 demonstrate moderate performance. The theoretical optimization may not fully translate to practical effectiveness when noise characteristics deviate from assumed models.

2.8. OpenCV Baseline Comparisons

OpenCV implementations serve as optimized reference baselines. The `cv2.bilateralFilter` achieves the best Gaussian noise reduction with MSE of 56.98 and PSNR of 30.57 dB, along with the highest Edge IoU of 0.510 for Gaussian noise. The bilateral filter's edge-preserving property makes it particularly effective. `cv2.medianBlur` excels for Salt & Pepper noise with MSE of 59.11 and PSNR of 30.41 dB, matching the custom median implementation. `cv2.GaussianBlur` and `cv2.blur` show performance similar to their custom counterparts, validating the correctness of custom implementations while highlighting OpenCV's computational optimizations.

Hình IV.8: Comparison between custom mean filter (3x3) and `cv2.blur`.

As shown in Figure IV.8, the custom mean filter and `cv2.blur` produce visually indistinguishable results, confirming that the custom implementation correctly performs uniform averaging. The performance metrics are nearly identical, with only minor differences due to numerical precision.

Hình IV.9: Comparison between custom Gaussian filter (3x3, $\sigma = 1.0$) and `cv2.GaussianBlur`

Figure IV.9 demonstrates that the custom Gaussian filter implementation matches `cv2.GaussianBlur` performance. Both methods employ Gaussian-weighted averaging, resulting in similar noise reduction and edge preservation characteristics. The slight differences in metrics are within expected numerical precision ranges.

Hình IV.10: Comparison between custom median filter (3x3) and `cv2.medianBlur`

The comparison in Figure IV.10 shows that both median filter implementations excel at removing Salt & Pepper noise while maintaining edge sharpness. The custom implementation achieves slightly better MSE (61.55 vs 62.27) and PSNR (30.24 vs 30.19 dB) for Salt & Pepper noise, demonstrating correct implementation of the median operation. For Gaussian noise, both methods show moderate performance, as the median filter is primarily designed for impulse noise removal.

Hình IV.11: Comparison between MMSE filter (5x5) and `cv2.GaussianBlur` (3x3)

Figure IV.11 compares the MMSE filter against `cv2.GaussianBlur`. The MMSE filter, using a 5×5 kernel with noise variance estimation, demonstrates moderate performance. While theoretically optimal under certain assumptions, the practical effectiveness depends on accurate noise variance estimation, which may limit performance when noise characteristics deviate from the assumed model.

2.9. All Filters Comparison

Hình IV.12: All filters comparison on Salt & Pepper noise.

For Salt & Pepper noise, nonlinear order-based filters (Median, `cv2.medianBlur`) achieve superior performance: noise is effectively removed, edge details are well preserved, and images remain sharp without excessive blurring.

Linear filters (Mean, Gaussian, `cv2.GaussianBlur`) generally produce the poorest quality with salt-pepper noise: noise reduction is minimal and edges suffer significant blurring due to averaging operations.

Conditional filters and custom adaptive methods (gradient-weighted, rotating mask) perform intermediately: they reduce noise better than linear filters and preserve details more effectively, but still fall short of median filter performance. Conditional-range provides the most stable quality within this group, yet does not achieve the "clean and sharp" results of median filters.

Among OpenCV library filters, `cv2.medianBlur` delivers the best results, consistent with theoretical expectations: median is the optimal choice for outlier removal.

Hình IV.13: All filters comparison on Gaussian noise.

For Gaussian noise, as shown in Figure IV.13, weight-based smoothing filters (Gaussian blur, gradient-weighted, rotating mask) demonstrate superior performance because the nature of Gaussian noise aligns well with distribution-based and gradient-based smoothing mechanisms. These filters provide uniform noise reduction, preserve relatively smooth edges, and achieve the most stable visual quality.

Conditional filters perform less effectively compared to their performance on salt-pepper noise. They maintain edges reasonably well but do not reduce Gaussian noise grains as effectively as weight-based smoothing filters.

Among library filters, `cv2.GaussianBlur` delivers results as expected: effective noise reduction with the smoothest output within the OpenCV group, while `cv2.medianBlur` becomes less suitable since median filters are not designed for continuously distributed noise.

Overall, Gaussian-based smoothing is the optimal choice for this noise type; custom methods (gradient-weighted, rotating mask) also achieve high effectiveness, approaching or exceeding linear Gaussian filters in edge preservation.

3. Quantitative Comparison

This section provides quantitative analysis through statistical tables comparing all evaluated filters across multiple metrics.

3.1. Quantitative Metrics Comparison

Table IV.1 presents a comprehensive comparison of all evaluated filters across noise reduction metrics (MSE and PSNR) for both Salt & Pepper and Gaussian noise. The table is organized into two groups: OpenCV filters with their custom equivalents, and advanced custom filters without direct OpenCV counterparts.

Bảng IV.1: Noise Reduction Performance: MSE and PSNR for Denoising Filters on Salt & Pepper and Gaussian Noise. Lower MSE and higher PSNR indicate better noise reduction performance.

Filter	**MSE (S&P)↓**	**PSNR (S&P)↑**	**MSE (Gauss)↓**	**PSNR (Gauss)↑**
OpenCV filters and my implementations equivalents				
cv2.bilateralFilter	946.51	18.37	**71.64**	**29.58**
cv2.GaussianBlur	197.18	25.18	81.99	28.99
gaussian	197.34	25.18	103.19	27.99
cv2.medianBlur	**61.47**	**30.24**	87.78	28.70
median	61.91	30.21	88.51	28.66
cv2.blur	206.96	24.97	102.58	28.02
mean	207.11	24.97	103.49	27.98
Advanced filters				
rotating_mask	178.85	25.61	82.82	28.95
conditional_diff	955.62	18.33	92.84	28.45
mmse	921.79	18.48	127.69	27.07
gradient_weighted	936.24	18.42	140.68	26.65

Analysis of Table IV.1 reveals several key findings regarding noise reduction performance. For Salt & Pepper noise, `cv2.medianBlur` achieves the best performance with MSE of **61.47** and PSNR of **30.24 dB**, followed closely by the custom median filter implementation with MSE of 61.91 and PSNR of 30.21 dB. The median filter's non-linear nature makes it particularly effective at removing impulse noise, as it replaces each pixel with the median value in its neighborhood, effectively eliminating salt and pepper outliers.

For Gaussian noise, `cv2.bilateralFilter` demonstrates superior performance with the lowest MSE of **71.64** and highest PSNR of **29.58 dB**. The bilateral filter's adaptive weighting mechanism, which considers both spatial and intensity similarity, makes it highly effective for Gaussian noise reduction while preserving edges.

The custom implementations (mean, gaussian, median) show performance comparable to

their OpenCV counterparts, with the custom gaussian filter achieving MSE of 197.34 and PSNR of 25.18 dB for Salt & Pepper noise, closely matching `cv2.GaussianBlur` (MSE: 197.18, PSNR: 25.18 dB). Small differences arise from numerical precision and implementation details, but the overall behavior validates the correctness of the implementations.

Among advanced custom filters, the rotating mask method shows the best overall noise reduction performance with MSE of 178.85 and PSNR of 25.61 dB for Salt & Pepper noise, and MSE of 82.82 and PSNR of 28.95 dB for Gaussian noise. The conditional_diff filter performs similarly to bilateral filtering for Salt & Pepper noise (MSE: 955.62, PSNR: 18.33 dB) but shows better performance for Gaussian noise (MSE: 92.84, PSNR: 28.45 dB) compared to other advanced methods.

Bảng IV.2: Edge Preservation Performance: Edge IoU for Denoising Filters on Salt & Pepper and Gaussian Noise. Higher Edge IoU indicates better edge preservation.

Filter	**Edge IoU (S&P)↑**	**Edge IoU (Gauss)↑**
Group 1: OpenCV filters and my implementations equivalents		
cv2.medianBlur	**0.512**	0.403
median	0.510	0.399
cv2.bilateralFilter	0.293	0.434
cv2.GaussianBlur	0.281	**0.465**
gaussian	0.280	0.423
cv2.blur	0.263	0.425
mean	0.266	0.423
Group 2: Advanced my implementations filters (no direct OpenCV equivalent)		
rotating_mask	0.386	0.391
conditional_diff	0.289	0.394
mmse	0.320	0.349
gradient_weighted	0.321	0.337

Table IV.2 evaluates the edge preservation capability of all filters using Edge IoU metrics. For Salt & Pepper noise, `cv2.medianBlur` achieves the highest Edge IoU value of **0.512**, followed closely by the custom median filter with Edge IoU of 0.510. The median filter's non-linear operation effectively removes impulse noise while maintaining sharp boundaries, making it the best choice for preserving edges when dealing with Salt & Pepper noise.

For Gaussian noise, `cv2.GaussianBlur` achieves the best edge preservation with Edge IoU of **0.465**, followed by `cv2.blur` (0.425) and `cv2.bilateralFilter` (0.434). The Gaussian filter's smooth weighting function preserves edge structures better than other methods for Gaussian noise reduction, while the bilateral filter's edge-preserving property adaptively weights pixels based on both spatial distance and intensity similarity.

Among advanced custom filters, the rotating mask method shows the best edge preservation performance with Edge IoU values of 0.386 for Salt & Pepper noise and 0.391 for Gaussian noise. The directional selectivity of the rotating mask approach helps preserve edges aligned with mask orientations. The conditional_diff filter achieves Edge IoU of 0.289 for Salt & Pepper noise and 0.394 for Gaussian noise, showing reasonable edge preservation capabilities.

3.2. Computational Efficiency Comparison

Table IV.3 presents execution time measurements for all filters, providing insight into computational efficiency and practical applicability.

Bảng IV.3: Execution Time Comparison for All Filters. Times are measured in milliseconds (ms), with filters ranked by speed (fastest first).

Filter	**Time (ms)**	**Rank**
cv2.GaussianBlur	**3.2**	1
conditional_diff	3.6	2
gradient_weighted	4.0	3
cv2.medianBlur	4.2	4
cv2.blur	4.9	5
median	5.9	6
gaussian	6.7	7
cv2.bilateralFilter	6.8	8
rotating_mask	15.0	9
mean	19.4	10
mmse	**32.6**	11

As shown in Table IV.3, `cv2.GaussianBlur` demonstrates the fastest execution time of **3.2 ms**, making it the most computationally efficient filter. The conditional_diff filter follows closely at 3.6 ms, ranking second overall. OpenCV implementations generally show computational advantages, with `cv2.medianBlur` completing in 4.2 ms and `cv2.blur` in 4.9 ms.

Custom implementations show varying execution times. The gradient_weighted filter achieves excellent performance at 4.0 ms, ranking third overall. The custom median filter executes in 5.9 ms, approximately 1.4 times slower than `cv2.medianBlur`, while the custom gaussian filter takes 6.7 ms, about 2.1 times slower than `cv2.GaussianBlur`. The custom mean filter requires 19.4 ms, significantly slower than `cv2.blur`.

The slowest method is mmse at **32.6 ms**, followed by rotating_mask at 15.0 ms. The performance gap between optimized OpenCV implementations and custom filters highlights the importance of optimization in production systems. While custom implementations provide educational value and algorithm transparency, OpenCV's optimized code is essential for real-time applications. The trade-off between implementation clarity and computational efficiency must be considered based on application requirements.

3.3. Overall Assessment

The comprehensive evaluation reveals that filter selection depends on multiple factors: noise type, quality requirements, edge preservation needs, and computational constraints. For Salt & Pepper noise, median filters (both custom and OpenCV) provide the best balance of noise removal and edge preservation, with `cv2.medianBlur` achieving the lowest MSE of 61.47 and highest Edge IoU of 0.512. For Gaussian noise, bilateral filtering offers superior quality with the lowest MSE of 71.64 and PSNR of 29.58 dB, while `cv2.GaussianBlur` provides the best edge preservation with Edge IoU of 0.465.

Computational efficiency varies significantly across filters. `cv2.GaussianBlur` is the fastest at 3.2 ms, while advanced custom filters like mmse require 32.6 ms, approximately 10 times slower. The conditional_diff filter demonstrates excellent efficiency, ranking second at 3.6 ms while providing reasonable noise reduction performance.

The evaluation demonstrates that kernel size significantly affects results: larger kernels provide better noise reduction but increase edge blurring, as larger kernels average over larger regions, reducing high-frequency details including both noise and edges. This trade-off is fundamental to spatial smoothing and must be carefully balanced based on application requirements. The choice between OpenCV's optimized implementations and custom filters depends on whether computational efficiency or algorithm transparency is prioritized.

Chapter V: Conclusion

1. Summary of Achievements

This assignment successfully implemented and evaluated a comprehensive set of spatial smoothing filters, ranging from simple linear kernels to advanced adaptive methods. Each technique was derived from its mathematical formulation and translated into a working NumPy implementation without relying on external optimized routines.

The overall completion of the lab is shown in Table:

Section	**Completion**
Average Filter	100%
Gaussian Filter	100%
Median Filter	100%
Conditional Averaging	100%
Gradient-Weighted Averaging	100%
Rotating Mask Filter	100%
MMSE Filter	100%
Comparison	100%

Bảng V.1: Summary of Achievements

2. Discussion

The results highlight several important observations:

- Median filtering is the most effective method for removing Salt & Pepper noise while maintaining edge quality, confirming its robustness to impulse outliers.
- Gaussian filtering provides a good compromise between noise reduction and edge preservation, particularly for additive Gaussian noise.
- Gradient-weighted and rotating mask filters preserve structural information better than uniform averaging, though at higher computational cost.

- MMSE filtering, while theoretically optimal for known noise statistics, is highly dependent on accurate noise variance estimates and may produce suboptimal results when assumptions are violated.
- Conditional averaging performs poorly unless threshold parameters are tuned carefully, showing that overly restrictive adaptive rules may degrade image quality.

The comparison with OpenCV reveals that custom implementations-while correct—are significantly slower due to lack of low-level optimizations. This demonstrates the educational value of manual implementations while reinforcing the practicality of using optimized libraries in real-world systems.

Future work may include exploring bilateral filtering, non-local means, and deep learning-based denoising networks to further investigate advanced smoothing paradigms.

Reference

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 4 edition, 2018.

[2] M. Manju, P. Abarna, U. Akila, and S. Yamini. Peak signal to noise ratio & mean square error calculation for various images using the lossless image compression in ccsds algorithm. *International Journal of Pure and Applied Mathematics*, 119(12):14471–14477, 2018. *Special Issue*.

[3] Ly Quoc Ngoc. Digital image & video processing - lecture 5: Làm trơn ảnh dựa trên miền không gian, 2025. Lecture slides.

Acknowledgment

I would like to thank Prof. Dr. Ly Quoc Ngoc for the clear and rigorous theoretical lectures on spatial smoothing and noise reduction. I also appreciate MSc. Nguyen Manh Hung and MSc. Pham Thanh Tung for their support in the practical sessions, including hands-on guidance and prompt answers to implementation questions. Their combined instruction helped bridge theory and practice throughout this assignment.