

”Heaven’s Light is Our Guide”



Department of Computer Science & Engineering  
Rajshahi University of Engineering & Technology

## Lab Report-4

Digital Image Processing Sessional

Course Code: CSE 4106

Submitted By:	Submitted To:
Name: Sajidur Rahman Tarafder Roll: 2003154 Section: C Session: 2020-21 Department: CSE	Khaled Zinnurine Lecturer Department of CSE, RUET

# Table of Contents

---

Spatial Filtering and Edge Detection . . . . .	3
Helper Functions . . . . .	3
1 Problem (a): Mean (Box) Filter . . . . .	4
1.1 Code Snippet . . . . .	4
1.2 Implementation Approach . . . . .	4
1.3 Output Figure . . . . .	5
2 Problem (b): Weighted Average Filter . . . . .	5
2.1 Code Snippet . . . . .	5
2.2 Implementation Approach . . . . .	6
2.3 Output Figure . . . . .	7
3 Problem (c): Median Filter . . . . .	7
3.1 Code Snippet . . . . .	7
3.2 Implementation Approach . . . . .	8
3.3 Output Figure . . . . .	9
4 Problem (d): Min and Max Filters . . . . .	9
4.1 Code Snippet . . . . .	9
4.2 Implementation Approach . . . . .	10
4.3 Output Figure . . . . .	11
5 Problem (e): Laplacian Filter . . . . .	11
5.1 Code Snippet . . . . .	11
5.2 Implementation Approach . . . . .	13
5.3 Output Figure . . . . .	14
6 Problem (f): Unsharp Masking . . . . .	14
6.1 Code Snippet . . . . .	14
6.2 Implementation Approach . . . . .	15
6.3 Output Figure . . . . .	16
7 Problem (g): High-Boost Filtering . . . . .	16
7.1 Code Snippet . . . . .	16
7.2 Implementation Approach . . . . .	17
7.3 Output Figure . . . . .	18
8 Problem (h): Laplacian-based Sharpening . . . . .	18
8.1 Code Snippet . . . . .	18
8.2 Implementation Approach . . . . .	19
8.3 Output Figure . . . . .	20

9	Problem (i): Sobel Edge Detection . . . . .	20
9.1	Code Snippet . . . . .	20
9.2	Implementation Approach . . . . .	21
9.3	Output Figure . . . . .	22
	Overall Discussion and Conclusion . . . . .	23

# Spatial Filtering and Edge Detection

**Topic:** Implementation of various image filters including smoothing filters (Box, Weighted Average, Median), morphological filters (Min/Max), edge detection (Sobel), and sharpening filters (Laplacian, Unsharp Masking, High-Boost) using Python (NumPy/Matplotlib).

## Shared Utilities

### Code Snippet

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def pad_image(img, pad_y, pad_x, mode='replicate'):
6      h,w = img.shape
7      H = h + 2*pad_y
8      W = w + 2*pad_x
9      out = np.zeros((H,W), dtype=img.dtype)
10     out[pad_y:pad_y+h, pad_x:pad_x+w] = img
11     if mode == 'replicate':
12         out[:pad_y, pad_x:pad_x+w] = img[0:1,:]
13         out[pad_y+h:, pad_x:pad_x+w] = img[-1:,:]
14         out[:, :pad_x] = out[:, pad_x:pad_x+1]
15         out[:, pad_x+w:] = out[:, pad_x+w-1:pad_x+w]
16     elif mode == 'zero':
17         pass
18     return out
19
20 def convolution(img, kernel):
21     ky, kx = kernel.shape
22     py, px = ky//2, kx//2
23     pimg = pad_image(img, py, px, mode='replicate')
24     out = np.zeros_like(img, dtype=np.float32)
25     h,w = img.shape
26     for y in range(h):
27         for x in range(w):
28             region = pimg[y:y+ky, x:x+kx]
29             out[y,x] = np.sum(region * kernel)
30     return np.clip(out, 0, 255).astype(img.dtype)
```

Listing 1. Padding and Convolution Utilities

## Implementation Approach

First, I created two helper functions that all filters will use. The `pad_image` function adds extra pixels around the image borders by copying the edge pixels. This prevents dark

borders from appearing when we apply filters. The `convolution` function is the heart of spatial filtering - it slides a small kernel (like a  $3 \times 3$  window) across the entire image, multiplies the kernel values with the pixel values underneath, adds them all up, and stores the result. I made sure to clip the final values between 0 and 255 so we don't get invalid pixel intensities.

## 1 Problem (a): Mean (Box) Filter

---

**Objective:** *Smooth the image using a  $3 \times 3$  average kernel.*

### 1.1 Code Snippet

---

```
1 def box_filter(img):
2     kernel = np.array([[1,1,1],
3                        [1,1,1],
4                        [1,1,1]], dtype=np.float32) / 9.0
5     return convolution(img, kernel)
6
7 box_result = box_filter(img)
```

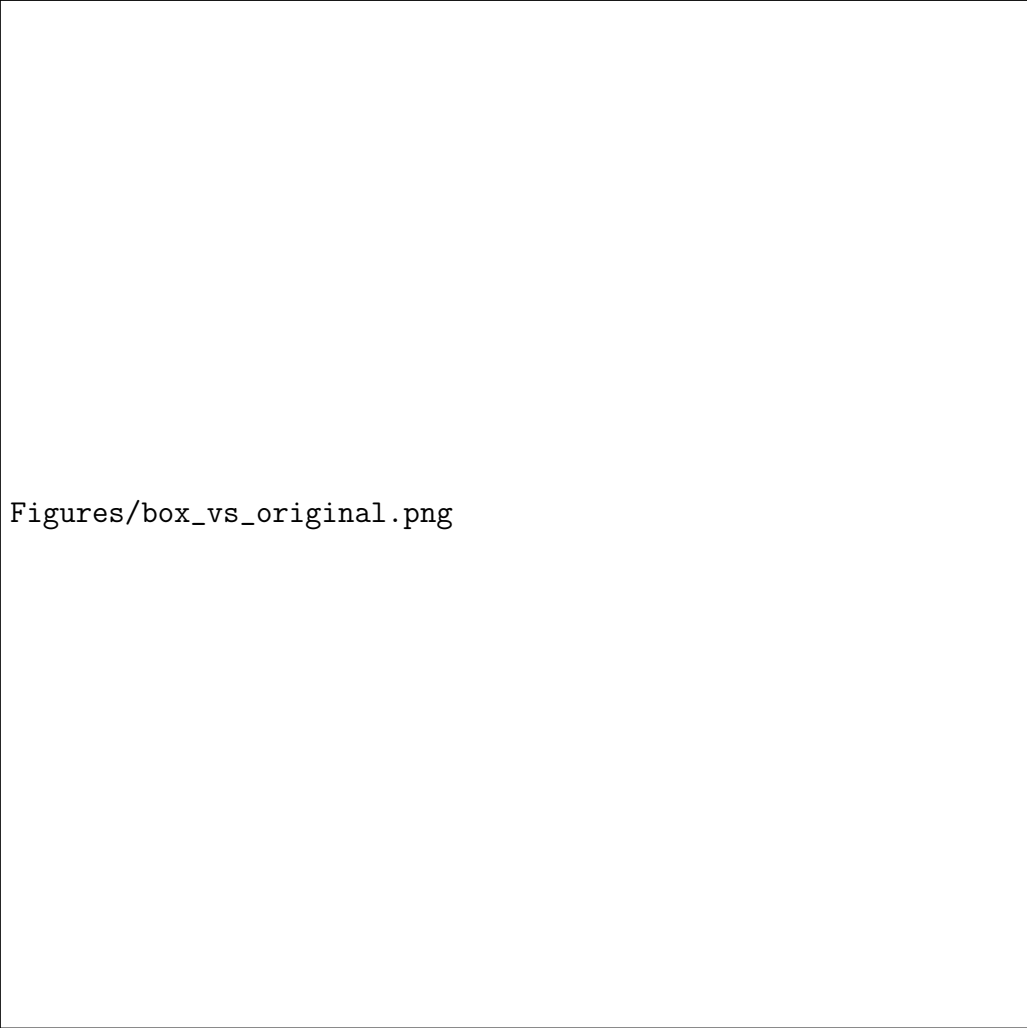
**Listing 2.** Mean (Box) Filter

### 1.2 Implementation Approach

---

The box filter is the simplest smoothing filter. I created a  $3 \times 3$  kernel where each element is  $1/9$ . When we apply this to the image, each pixel becomes the average of itself and its 8 neighbors. This smooths out noise nicely, but it also blurs the entire image including edges. The filter is normalized (all values add up to 1) so the image doesn't become darker or brighter overall.

### 1.3 Output Figure



Figures/box\_vs\_original.png

Figure 1. Original vs. Box Filtered Output

## 2 Problem (b): Weighted Average Filter

---

**Objective:** *Apply a separable weighted average that emphasizes the center.*

### 2.1 Code Snippet

---

```
1 # Weighted Average Filter
2 def weighted_average_filter(img):
3     kernel = np.array([[1,2,1],
4                        [2,4,2],
5                        [1,2,1]], dtype=np.float32) / 16.0
6     return convolution(img, kernel)
7
```

```
8 weighted_result = weighted_average_filter(img)
9
10 # Display original and filtered result
11 plt.figure(figsize=(12,5))
12 plt.subplot(1,2,1)
13 plt.imshow(img, cmap='gray')
14 plt.title('Original')
15 plt.axis('off')
16
17 plt.subplot(1,2,2)
18 plt.imshow(weighted_result, cmap='gray')
19 plt.title('Weighted Average Filter')
20 plt.axis('off')
21 plt.tight_layout()
22 plt.show()
```

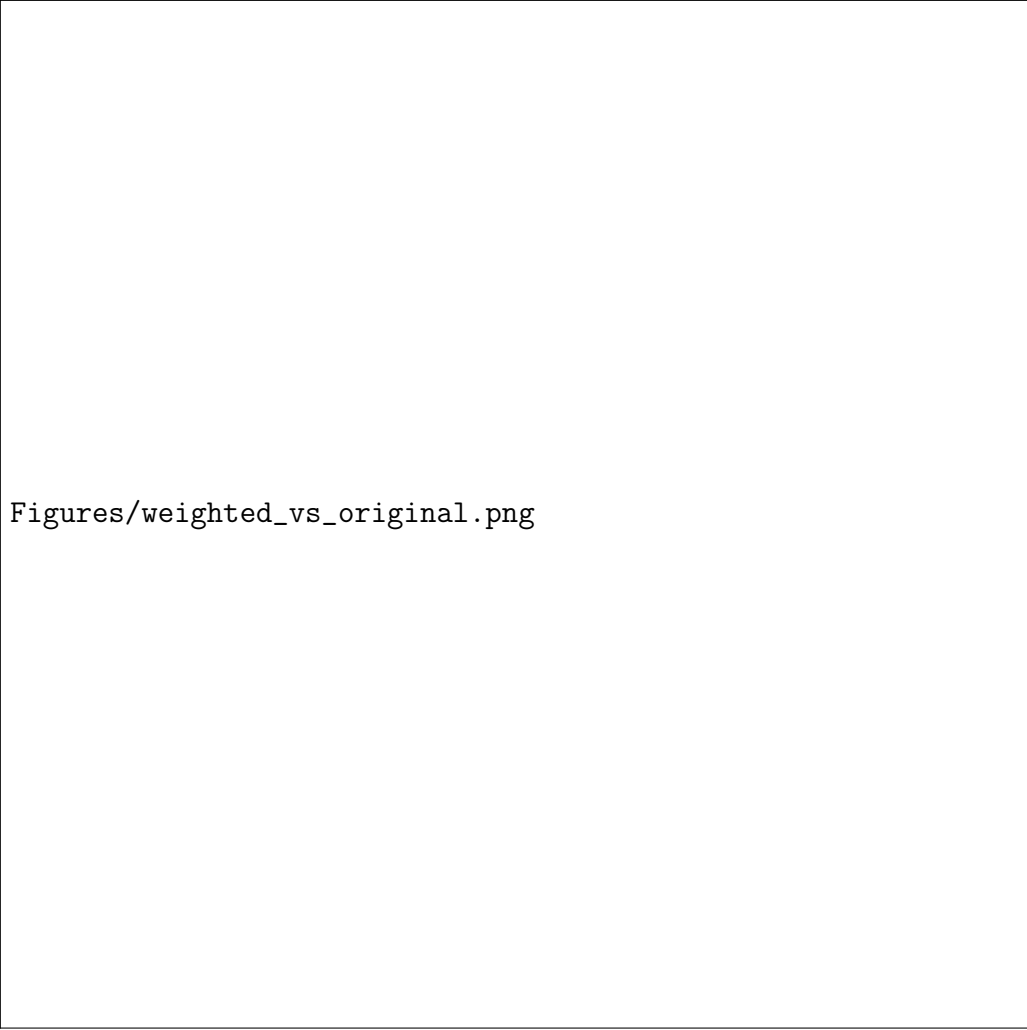
**Listing 3.** Weighted Average Filter (1-2-1 Kernel)

---

## 2.2 Implementation Approach

The weighted average filter is smarter than the box filter. Instead of treating all neighbors equally, it gives different weights to different positions. The center pixel gets the highest weight (4), the pixels directly above, below, left, and right get medium weight (2), and the diagonal corners get the lowest weight (1). After dividing by 16 (the sum of all weights), this creates a smoother blur that looks more natural and preserves edges better than the box filter. It's similar to a Gaussian blur but easier to compute.

## 2.3 Output Figure



Figures/weighted\_vs\_original.png

Figure 2. Original vs. Weighted Average Filter Output

### 3 Problem (c): Median Filter

**Objective:** *Remove salt-and-pepper noise while preserving edges.*

#### 3.1 Code Snippet

```
1 # Median Filter
2 def median_filter(img):
3     padded = pad_image(img, 1, 1, mode='replicate')
4     h,w = img.shape
5     out = np.zeros_like(img)
6     for y in range(h):
7         for x in range(w):
```



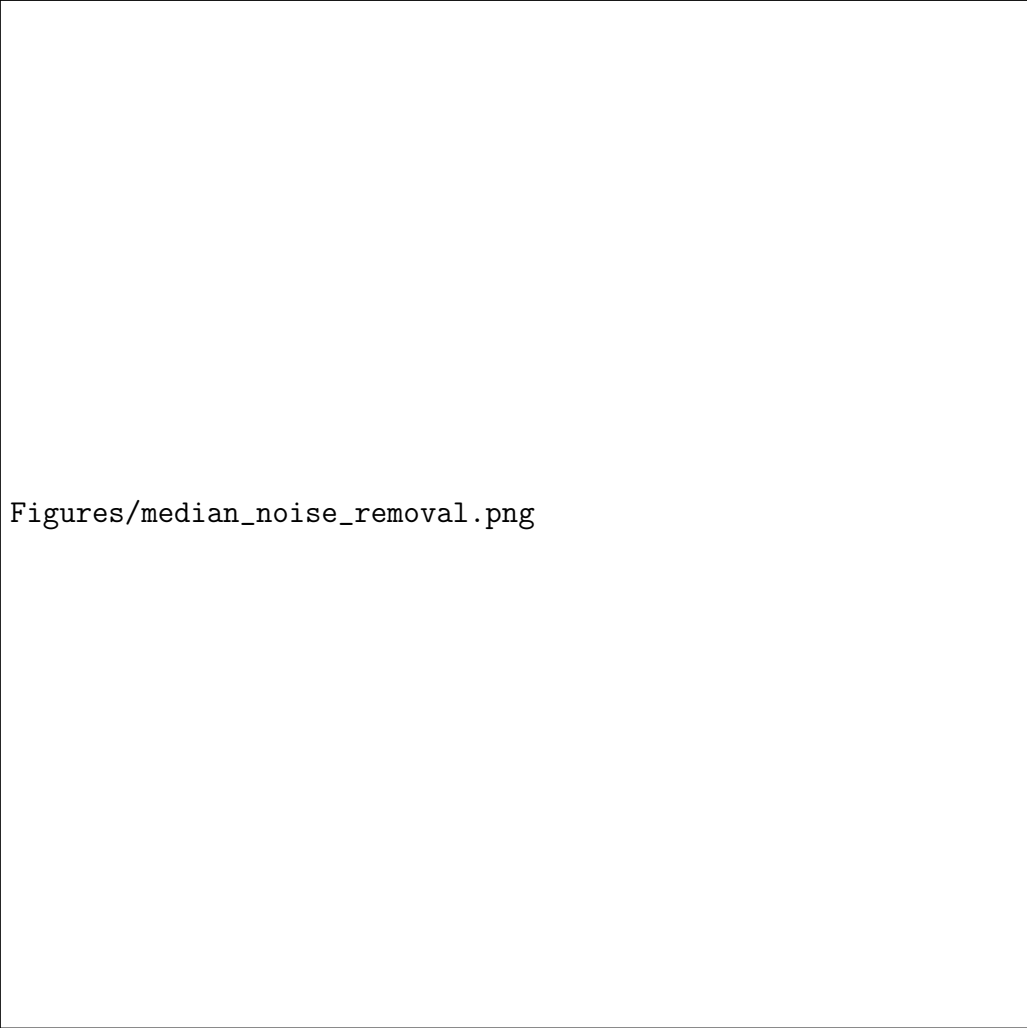
```
8         region = padded[y:y+3, x:x+3].flatten()
9         region.sort()
10        out[y,x] = region[4]  # median of 9 values
11    return out
12
13    median_result = median_filter(img)
14
15    # Display results
16    plt.figure(figsize=(12,5))
17    plt.subplot(1,2,1)
18    plt.imshow(img, cmap='gray')
19    plt.title('Original')
20    plt.axis('off')
21
22    plt.subplot(1,2,2)
23    plt.imshow(median_result, cmap='gray')
24    plt.title('Median Filter')
25    plt.axis('off')
26    plt.tight_layout()
27    plt.show()
```

Listing 4. Median Filter (3x3)

## 3.2 Implementation Approach

The median filter works differently from averaging filters - instead of calculating the mean, it picks the middle value. For each pixel, I look at its  $3 \times 3$  neighborhood (9 pixels total), sort them from smallest to largest, and pick the 5th value (the middle one). This is excellent for removing salt-and-pepper noise (those random black and white dots you sometimes see in images) because it ignores extreme values. Unlike blur filters, it keeps edges sharp because it doesn't blend values together.

## 3.3 Output Figure



Figures/median\_noise\_removal.png

Figure 3. Median Filter on Salt-and-Pepper Noise

## 4 Problem (d): Min and Max Filters

**Objective:** *Demonstrate nonlinear rank filtering for erosion-like and dilation-like effects.*

### 4.1 Code Snippet

```
1 # Min Filter
2 def min_filter(img):
3     padded = pad_image(img, 1, 1, mode='replicate')
4     h,w = img.shape
5     out = np.zeros_like(img)
6     for y in range(h):
7         for x in range(w):
```

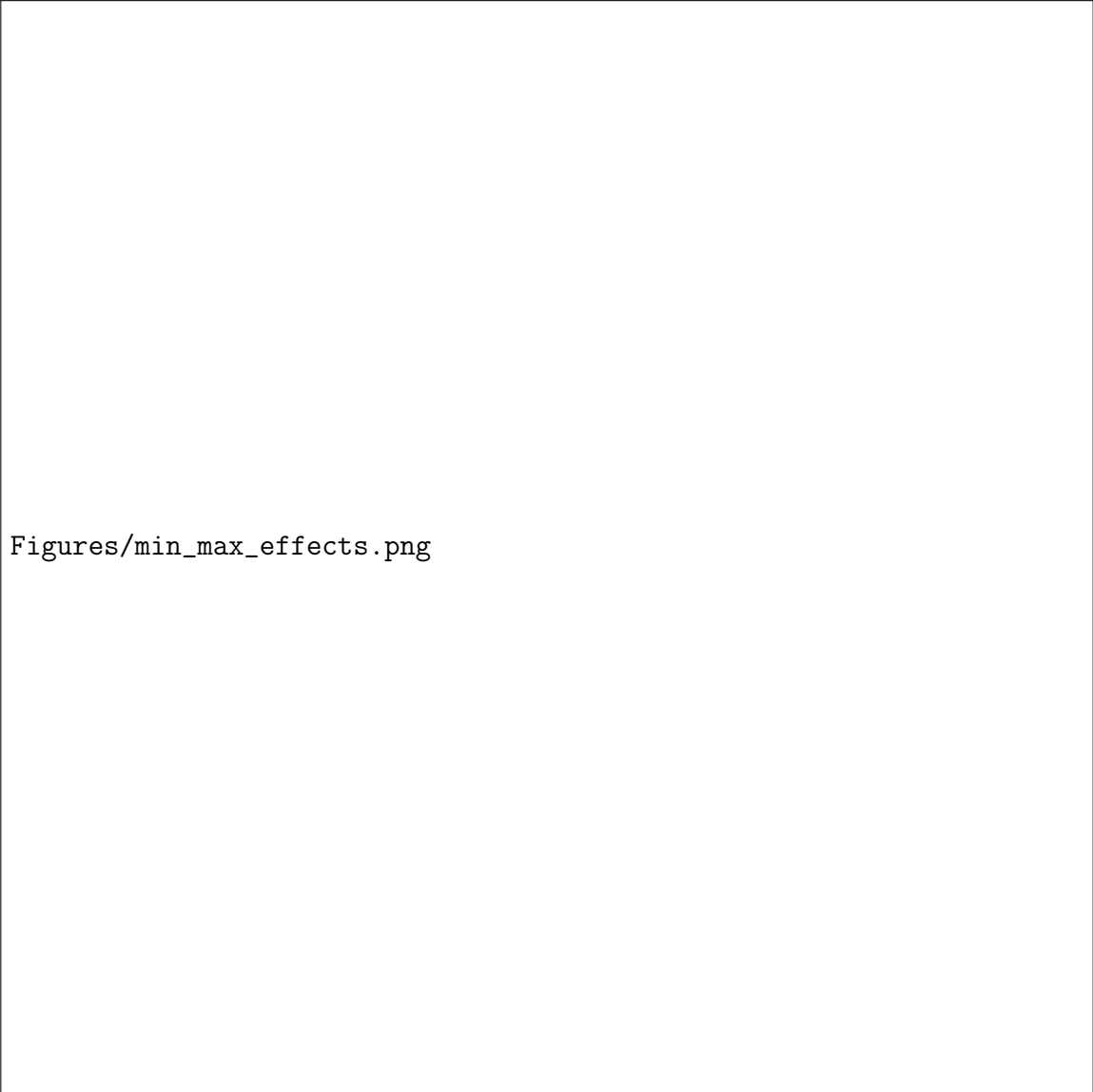
```
8         region = padded[y:y+3, x:x+3]
9         out[y,x] = np.min(region)
10     return out
11
12 # Max Filter
13 def max_filter(img):
14     padded = pad_image(img, 1, 1, mode='replicate')
15     h,w = img.shape
16     out = np.zeros_like(img)
17     for y in range(h):
18         for x in range(w):
19             region = padded[y:y+3, x:x+3]
20             out[y,x] = np.max(region)
21     return out
22
23 min_result = min_filter(img)
24 max_result = max_filter(img)
25
26 # Display results
27 plt.figure(figsize=(15,5))
28 plt.subplot(1,3,1)
29 plt.imshow(img, cmap='gray')
30 plt.title('Original')
31 plt.axis('off')
32
33 plt.subplot(1,3,2)
34 plt.imshow(min_result, cmap='gray')
35 plt.title('Min Filter')
36 plt.axis('off')
37
38 plt.subplot(1,3,3)
39 plt.imshow(max_result, cmap='gray')
40 plt.title('Max Filter')
41 plt.axis('off')
42 plt.tight_layout()
43 plt.show()
```

Listing 5. Min and Max Filters

## 4.2 Implementation Approach

Min and Max filters are simple but useful morphological operations. The Min filter replaces each pixel with the darkest pixel in its  $3 \times 3$  neighborhood. This makes the image darker overall and shrinks bright objects while expanding dark regions - great for removing bright "pepper" noise. The Max filter does the opposite, replacing each pixel with the brightest neighbor. This makes bright objects bigger and removes dark "salt" noise. They're like erosion and dilation operations you might learn about in morphology.

## 4.3 Output Figure



Figures/min\_max\_effects.png

Figure 4. Effects of Min and Max Filters

## 5 Problem (e): Laplacian Filter

---

**Objective:** *Detect rapid intensity changes via second-order derivatives.*

### 5.1 Code Snippet

---

```
1 # Laplacian with Negative Center
2 def laplacian_filter(img):
3     kernel = np.array([[0, 1, 0],
```

```
4         [1, -4, 1],
5         [0, 1, 0]], dtype=np.float32)
6     return convolution(img, kernel)
7
8     lap_result = laplacian_filter(img)
9     sharpened = img - lap_result # subtract to sharpen
10
11 # Display results
12 plt.figure(figsize=(15,5))
13 plt.subplot(1,3,1)
14 plt.imshow(img, cmap='gray')
15 plt.title('Original')
16 plt.axis('off')
17
18 plt.subplot(1,3,2)
19 plt.imshow(lap_result, cmap='gray')
20 plt.title('Laplacian Response (Negative)')
21 plt.axis('off')
22
23 plt.subplot(1,3,3)
24 plt.imshow(sharpened, cmap='gray')
25 plt.title('Sharpened (Original - Laplacian)')
26 plt.axis('off')
27 plt.tight_layout()
28 plt.show()
```

Listing 6. Laplacian Filtering - Negative Kernel

```
1 # Laplacian with Positive Center
2 def laplacian_filter(img):
3     kernel = np.array([[ 0, -1,  0],
4                        [-1,  4, -1],
5                        [ 0, -1,  0]], dtype=np.float32)
6     return convolution(img, kernel)
7
8     lap_result = laplacian_filter(img)
9     sharpened = img + lap_result # add to sharpen
10
11 # Display results
12 plt.figure(figsize=(15,5))
13 plt.subplot(1,3,1)
14 plt.imshow(img, cmap='gray')
15 plt.title('Original')
16 plt.axis('off')
17
18 plt.subplot(1,3,2)
19 plt.imshow(lap_result, cmap='gray')
20 plt.title('Laplacian Response (Positive)')
21 plt.axis('off')
22
23 plt.subplot(1,3,3)
24 plt.imshow(sharpened, cmap='gray')
25 plt.title('Sharpened (Original + Laplacian)')
```

```
26 plt.axis('off')
27 plt.tight_layout()
28 plt.show()
```

Listing 7. Laplacian Filtering - Positive Kernel


## 5.2 Implementation Approach

The Laplacian filter detects edges by measuring how quickly pixel intensities are changing. It uses second derivatives, which means it looks at the rate of change of the rate of change. I implemented two versions with different kernels:

- **Negative Center Kernel**  $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ : The center value is negative, so edges appear dark. To sharpen the image, we subtract this Laplacian result from the original.
- **Positive Center Kernel**  $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ : The center value is positive, so edges appear bright. To sharpen, we add this Laplacian result to the original.

Both methods work equally well - they just differ in whether you're adding or subtracting. The Laplacian is great at finding edges and fine details, and when we add them back to the original image, it makes everything look sharper and clearer.

## 5.3 Output Figure



Figures/laplacian\_edges.png

Figure 5. Laplacian Response (Edge Highlights)

## 6 Problem (f): Unsharp Masking

**Objective:** *Sharpen the image by adding back high-frequency details.*

### 6.1 Code Snippet

```
1 # Unsharp Masking with parameter k
2 def unsharp_masking(img, k=1):
3     blurred = box_filter(img)
4     mask = img - blurred
5     result = img + k * mask
6     return np.clip(result, 0, 255).astype(np.uint8)
7
```

```
8 k_value = 1
9 unsharp_result = unsharp_masking(img, k=k_value)
10
11 # Display results
12 plt.figure(figsize=(20,5))
13 plt.subplot(1,4,1)
14 plt.imshow(img, cmap='gray')
15 plt.title('Original')
16 plt.axis('off')
17
18 plt.subplot(1,4,2)
19 plt.imshow(box_filter(img), cmap='gray')
20 plt.title('Blurred')
21 plt.axis('off')
22
23 plt.subplot(1,4,3)
24 mask = img - box_filter(img)
25 plt.imshow(mask, cmap='gray')
26 plt.title('Mask (Original - Blurred)')
27 plt.axis('off')
28
29 plt.subplot(1,4,4)
30 plt.imshow(unsharp_result, cmap='gray')
31 plt.title(f'Unsharp Masking (k={k_value})')
32 plt.axis('off')
33 plt.tight_layout()
34 plt.show()
```

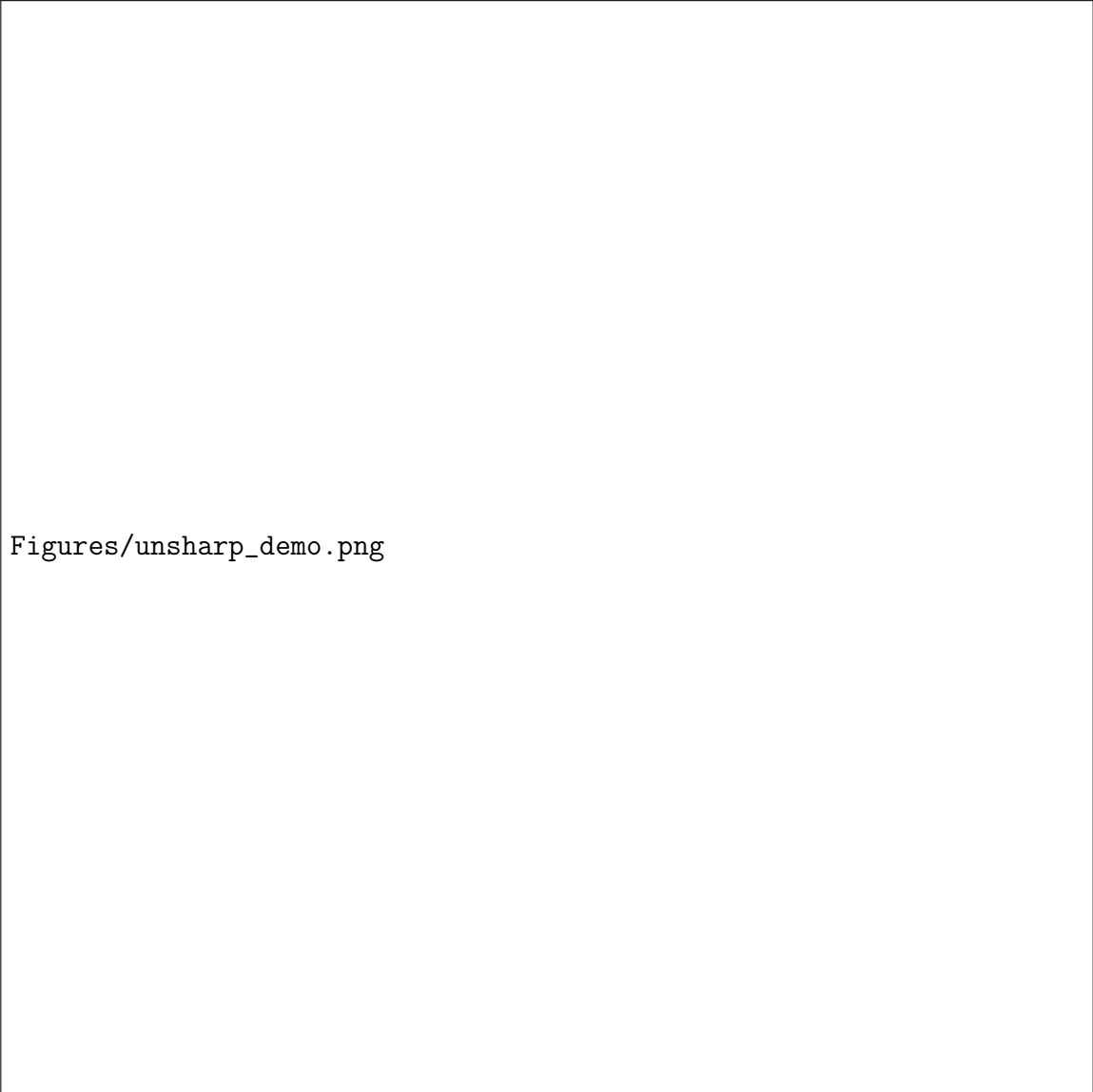
Listing 8. Unsharp Masking

## 6.2 Implementation Approach

Unsharp masking is a popular sharpening technique used by photographers. Here's how it works: First, I blur the image using a box filter. Then I subtract this blurred version from the original to create a "mask" that contains only the high-frequency details (edges and fine textures). Finally, I add this mask back to the original image, scaled by a parameter  $k$ . When  $k = 1$ , we get standard sharpening. When  $k > 1$ , the sharpening effect is stronger. The beauty of this method is that we're isolating and enhancing just the details we want to make stand out. I clamp the final result to stay within valid pixel values (0-255).

## 6.3 Output Figure





Figures/unsharp\_demo.png

Figure 6. Unsharp Masking: Original vs. Sharpened

## 7 Problem (g): High-Boost Filtering

**Objective:** Generalize unsharp masking with a boost factor  $k > 1$ .

### 7.1 Code Snippet

```
1 # High-Boost Filter with parameter A
2 def high_boost_filter(img, A=1.5):
3     blurred = box_filter(img)
```

```

4     mask = img - blurred
5     result = A * img - blurred
6     # Equivalent: result = img + (A-1)*img - blurred
7     #                 = img + A*(img - blurred) - (img - blurred)
8     #                 = img + (A-1)*(img - blurred)
9     return np.clip(result, 0, 255).astype(np.uint8)
10
11 A_value = 1.5
12 highboost_result = high_boost_filter(img, A=A_value)
13
14 # Display results
15 plt.figure(figsize=(20,5))
16 plt.subplot(1,4,1)
17 plt.imshow(img, cmap='gray')
18 plt.title('Original')
19 plt.axis('off')
20
21 plt.subplot(1,4,2)
22 plt.imshow(box_filter(img), cmap='gray')
23 plt.title('Blurred')
24 plt.axis('off')
25
26 plt.subplot(1,4,3)
27 mask = img - box_filter(img)
28 plt.imshow(mask, cmap='gray')
29 plt.title('Mask (Original - Blurred)')
30 plt.axis('off')
31
32 plt.subplot(1,4,4)
33 plt.imshow(highboost_result, cmap='gray')
34 plt.title(f'High-Boost Filter (A={A_value})')
35 plt.axis('off')
36 plt.tight_layout()
37 plt.show()

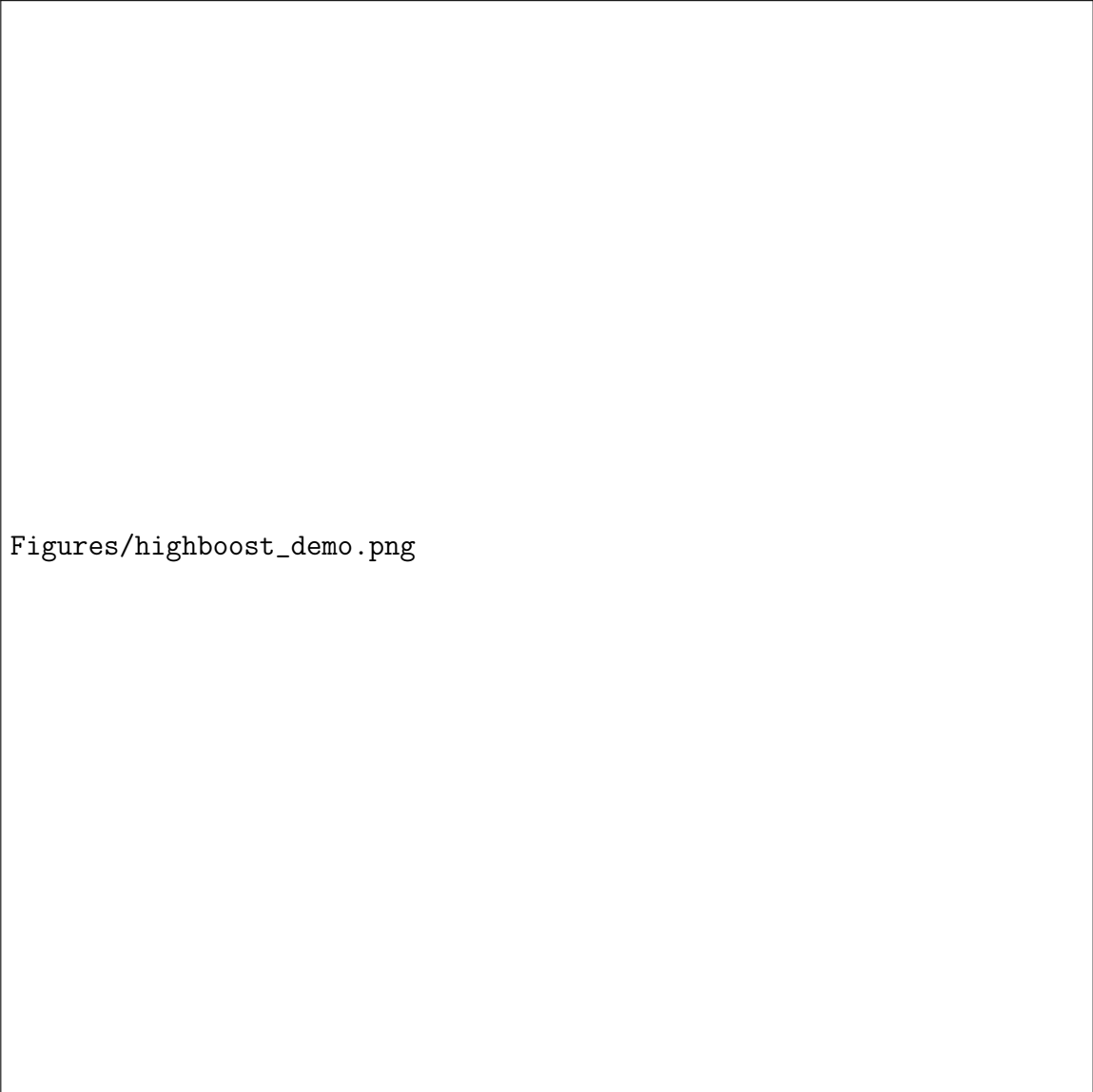
```

Listing 9. High-Boost Filtering

## 7.2 Implementation Approach

High-boost filtering takes unsharp masking to the next level. Instead of just adding the detail mask back to the original, we amplify the original image first using a factor  $A$ . The formula is:  $\text{result} = A \times \text{original} - \text{blurred}$ . When  $A = 1$ , this is exactly the same as regular unsharp masking. When  $A > 1$ , we get both a brighter image and stronger edge enhancement. Think of  $A$  as a boost dial - the higher you turn it, the more dramatic the effect. This is useful when you want to really make details pop, but you need to be careful not to overdo it and create artifacts. As always, I clip the result to keep pixel values valid.

## 7.3 Output Figure



Figures/highboost\_demo.png

Figure 7. High-Boost Filtering with Different  $k$  Values

## 8 Problem (h): Laplacian-based Sharpening

---

**Objective:** *Use Laplacian response to enhance edges.*

### 8.1 Code Snippet

---

```
1 # Note: This section demonstrates the sharpening effect already
2 # shown in Problem (e) with both Laplacian kernel versions.
3
```

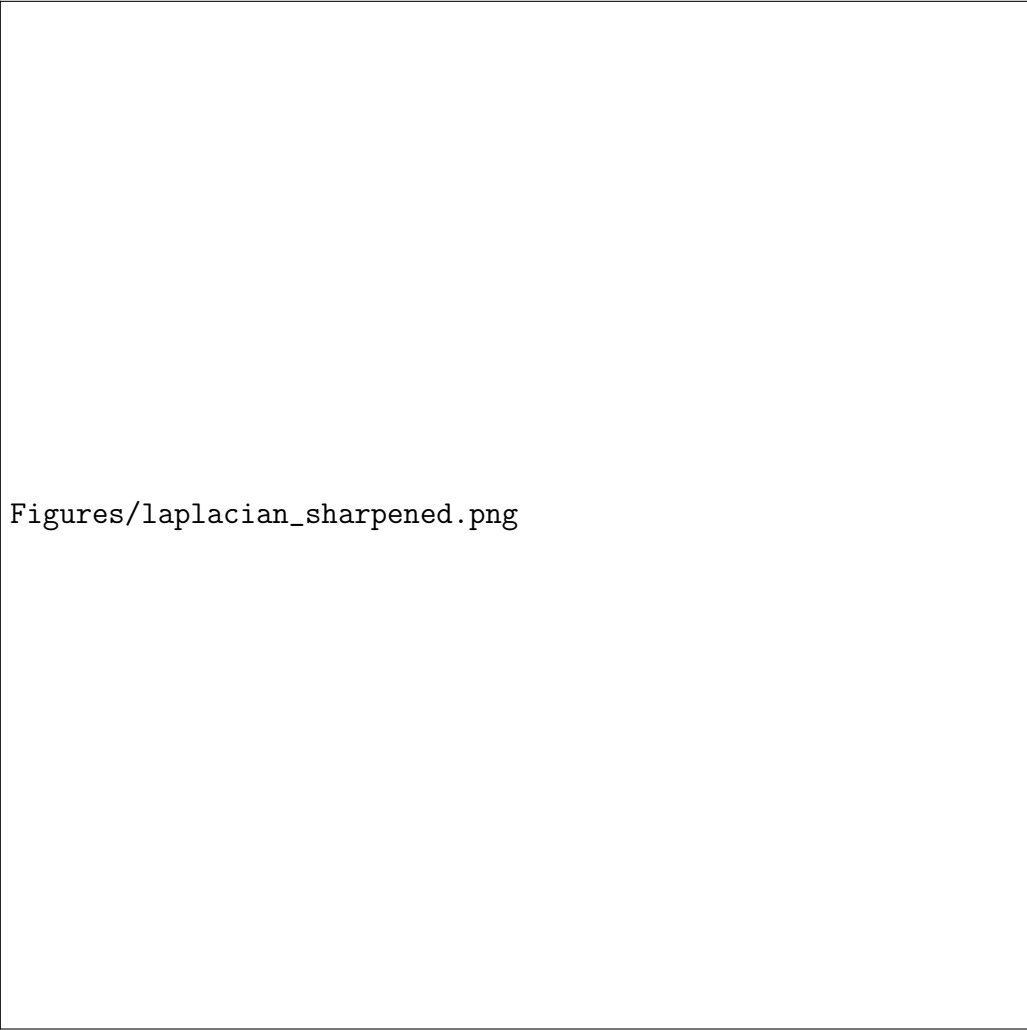
```
4 # Using Negative Center Kernel:
5 # lap_result = laplacian_filter(img)
6 # sharpened = img - lap_result
7
8 # Using Positive Center Kernel:
9 # lap_result = laplacian_filter(img)
10 # sharpened = img + lap_result
11
12 # Both approaches shown in Problem (e) with full visualization
```

**Listing 10.** Laplacian Sharpening

## 8.2 Implementation Approach

This section demonstrates Laplacian-based sharpening, which I already showed in Problem (e). The key idea is simple: the Laplacian filter finds edges, and when we add those edges back to (or subtract them from) the original image, we get a sharpened result. Whether you add or subtract depends on which kernel version you use. The negative center kernel gives negative edge values (so we subtract), while the positive center kernel gives positive edge values (so we add). Both approaches work great for bringing out fine details and making images look crisper. This is actually one of the oldest and most widely-used sharpening techniques in image processing.

## 8.3 Output Figure



Figures/laplacian\_sharpened.png

Figure 8. Laplacian-based Sharpening

## 9 Problem (i): Sobel Edge Detection

---

**Objective:** *Compute gradient magnitude (and optionally orientation) using Sobel operators.*

### 9.1 Code Snippet

---

```
1 # Sobel Edge Detection
2 def sobel_filters(img):
3     sobel_h = np.array([[ -1, -2, -1],
4                         [ 0,  0,  0],
5                         [ 1,  2,  1]], dtype=np.float32)
6
```

```

7     sobel_v = np.array([[ -1,  0,  1],
8                        [ -2,  0,  2],
9                        [ -1,  0,  1]], dtype=np.float32)
10
11     Gh = convolution(img, sobel_h)
12     Gv = convolution(img, sobel_v)
13     Gmag = np.sqrt(Gh**2 + Gv**2)
14     Gmag = np.clip(Gmag, 0, 255).astype(np.uint8)
15
16     return Gh, Gv, Gmag
17
18 Gh, Gv, Gmag = sobel_filters(img)
19
20 # Display results
21 plt.figure(figsize=(20,5))
22 plt.subplot(1,4,1)
23 plt.imshow(img, cmap='gray')
24 plt.title('Original')
25 plt.axis('off')
26
27 plt.subplot(1,4,2)
28 plt.imshow(Gh, cmap='gray')
29 plt.title('Sobel Horizontal (Gh)')
30 plt.axis('off')
31
32 plt.subplot(1,4,3)
33 plt.imshow(Gv, cmap='gray')
34 plt.title('Sobel Vertical (Gv)')
35 plt.axis('off')
36
37 plt.subplot(1,4,4)
38 plt.imshow(Gmag, cmap='gray')
39 plt.title('Sobel Magnitude')
40 plt.axis('off')
41 plt.tight_layout()
42 plt.show()

```

Listing 11. Sobel Filters (H &amp; V, Magnitude)

## 9.2 Implementation Approach

Sobel edge detection is one of the most popular edge detection methods. It uses two special  $3 \times 3$  kernels - one for horizontal edges and one for vertical edges. The horizontal

kernel  $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$  responds strongly to horizontal edges (where the intensity changes

from top to bottom), while the vertical kernel  $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$  responds to vertical edges (where intensity changes from left to right).

After applying both kernels, I combine them using the Pythagorean theorem:  $G_{mag} = \sqrt{G_h^2 + G_v^2}$ . This gives us the overall edge strength at each point, regardless of direction. The beautiful thing about Sobel is that it's simple, fast, and produces clean edge maps that work well for most images. Higher magnitude values mean stronger edges.

### 9.3 Output Figure

Figures/sobel\_h\_v\_mag.png

**Figure 9.** Sobel Horizontal, Vertical, and Magnitude

## Overall Discussion and Conclusion

---

In this lab, I implemented a comprehensive set of spatial filters and edge detection techniques from scratch using Python. The experience gave me a much deeper understanding of how these filters actually work, rather than just calling library functions.

I started by building the fundamental building blocks - a padding function to handle image borders and a general convolution function that can work with any kernel. These utilities became the foundation for everything else. The smoothing filters (box and weighted average) showed me how different kernel designs affect the blur quality. The box filter is simple but effective, while the weighted average produces smoother, more natural-looking results.

The median filter was particularly interesting because it works so differently from the averaging filters. Instead of blending pixel values together, it picks the middle value, which makes it excellent at removing salt-and-pepper noise while keeping edges sharp. The min and max filters introduced me to morphological operations - I could see how they erode or dilate features in the image.

For edge detection and sharpening, the Laplacian filter taught me about second-order derivatives and how they respond to rapid intensity changes. I implemented both kernel versions and saw how the same mathematical concept can be implemented in different ways. Unsharp masking and high-boost filtering showed me how we can decompose an image into low and high frequency components, then manipulate them separately to enhance details.

Finally, the Sobel filter demonstrated first-order derivative approaches to edge detection. By using separate horizontal and vertical kernels, it can detect edges in different orientations and combine them into a single edge strength map. This is more sophisticated than the Laplacian because it gives us directional information.

Overall, this lab showed me that image processing is really about understanding what mathematical operations do to pixel neighborhoods, and how we can design kernels to achieve specific effects. The modular design I used - building reusable padding and convolution functions first, then implementing specific filters on top - made the code clean and easy to experiment with. Each filter has its strengths and trade-offs, and choosing the right one depends on what you're trying to achieve and what kind of noise or features you're dealing with.