assignment1

April 22, 2025

1 Introduction

Image denoising is an essential process in computer vision, aimed at improving image quality by reducing noise while preserving important details. Various filters are commonly used to achieve this, each with unique characteristics in noise reduction, edge preservation, and computational efficiency. This assignment involves comparing the performance of simple smoothing filters: Box, Gaussian, and Median filters, with more advanced filters like adaptive mean, adaptive median, and bilateral filters. The filters are evaluated on noisy images across different metrics, including Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR), edge preservation, and processing speed. Additionally, the influence of different kernel sizes on filter performance is examined, providing insights into the trade-offs between noise reduction, detail preservation, and computational cost. This comparison aims to highlight the strengths and limitations of each filter type, aiding in the selection of appropriate filters for various image processing tasks.

- Libraries used in this assignment:

[1]:
```
from PIL import Image
import matplotlib.pyplot as plt
import cv2
import numpy as np
import os
import time
import pandas as pd
import seaborn as sns
```

2 Step 1. Generate or Load Noisy Images

3 Step 1.1. Clean Images Selection

- I used the Berkeley Segmentation Dataset 500 (BSDS500), which includes 200 training images. I selected 20 random images from various categories for testing and displayed the top 5 that produced the best results.

4 I. Natural High-Detail Images:

Images with intricate textures and fine details from nature, such as animal fur, flowers, and coral reefs.

[40]:
```
#list of image paths
image_paths = [
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/12074.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/124084.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/109034.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/134052.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/135037.
 ↪jpg"
]

plt.figure(figsize=(15, 10))  #set up the figure and size

#loop through the images and display them
for i, image_path in enumerate(image_paths):
    image = Image.open(image_path)
    plt.subplot(1, 5, i + 1)  # 1 row and 5 columns
    plt.imshow(image)
    plt.axis('off')
    plt.title(f"Image {i + 1}")

plt.show() #display all images
```

In this assignment, five images were processed, but only the results of Image 2, featuring the "red flower," are displayed and discussed, as it provided the best outcomes. The flower's petals have a clear structure and well-defined edges, making it easier for filters to reduce noise while preserving important details. The background is relatively plain and out of focus, which minimizes distractions and allows filters to concentrate on detail preservation in the foreground (the flowers). The contrast between the vibrant red petals and the green background helps the filters retain edge information and avoid excessive blurring.

5 II. Natural Low-Detail Images:

Nature images with simpler, uniform backgrounds, such as open skies, grassy fields, and blurred landscapes.

[41]:
```
#list of image paths
image_paths = [
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/130034.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/135069.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/126039.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/112082.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/105019.
 ↪jpg"
]

plt.figure(figsize=(15, 10)) #set up the figure and size

#loop through the images and display them
for i, image_path in enumerate(image_paths):
    image = Image.open(image_path)
    plt.subplot(1, 5, i + 1)  # 1 row and 5 columns
    plt.imshow(image)
    plt.axis('off')
    plt.title(f"Image {i + 1}")

plt.show() #display all images
```

Here, Image 2 (a bird flying against a clear, plain sky) was selected for display. The smooth, uniform sky background makes it ideal for evaluating the impact of noise and the effectiveness of filters in restoring uniformity after noise reduction. The bird stands out distinctly against the sky, allowing for a straightforward assessment of edge preservation without extra distractions. The minimal background details reduce the risk of losing subtle textures, enabling filters to concentrate on preserving the shape and edges of the bird.

6 III. Influenced by Human High-Detail Images:

Man-made scenes with complex textures and architectural details, including buildings, traditional attire, and textured materials.

[42]:
```
#list of image paths
image_paths = [
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/138032.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/118035.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/118020.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/122048.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/140075.
 ↪jpg"
]

plt.figure(figsize=(15, 10)) #set up the figure and size

#loop through the images and display them
for i, image_path in enumerate(image_paths):
    image = Image.open(image_path)
    plt.subplot(1, 5, i + 1)  # 1 row and 5 columns
    plt.imshow(image)
    plt.axis('off')
    plt.title(f"Image {i + 1}")

plt.show() #display all images
```

Image 1 (a close-up of coiled rope) was selected for display. The rope fibers exhibit a high level of detail, with distinct textures and patterns, making it ideal for evaluating how filters handle fine details. The coiled shape and texture create a structured pattern, useful for assessing how well filters preserve structural details after noise reduction. With the rope as the main subject against a relatively uniform background, filters can concentrate on preserving details without interference from other elements.

7 IV. Influenced by Human Low-Detail Images:

Scenes with clear human influence, featuring organized structures or objects within spacious, uncluttered environments, and without detailed backgrounds.

[43]:
```
#list of image paths
image_paths = [
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/157036.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/161062.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/169012.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/166081.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/172032.
 ↪jpg"
]

plt.figure(figsize=(15, 10)) #set up the figure and size

#loop through the images and display them
for i, image_path in enumerate(image_paths):
    image = Image.open(image_path)
    plt.subplot(1, 5, i + 1)  # 1 row and 5 columns
    plt.imshow(image)
    plt.axis('off')
    plt.title(f"Image {i + 1}")

plt.show() #display all images
```

Image 2 (the pyramids) was selected for display. The pyramids have a distinct shape and large, smooth surfaces, making them ideal for evaluating how filters handle noise on plain areas with minimal texture. The surrounding desert and sky are simple and lack detailed features, making this image suitable for testing how effectively filters preserve the simplicity of the scene without adding artifacts.

8 V. Edge-Rich Images:

Images with prominent, well-defined edges, such as animals against plain backgrounds, structured objects, and clear object boundaries.

[44]:
```
#list of image paths
image_paths = [
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/176035.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/176039.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/178054.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/183055.
 ↪jpg",
    "C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/images/train/187029.
 ↪jpg"
]

plt.figure(figsize=(15, 10)) #set up the figure and size

#loop through the images and display them
for i, image_path in enumerate(image_paths):
    image = Image.open(image_path)
    plt.subplot(1, 5, i + 1)  # 1 row and 5 columns
    plt.imshow(image)
    plt.axis('off')
    plt.title(f"Image {i + 1}")

plt.show() #display all images
```

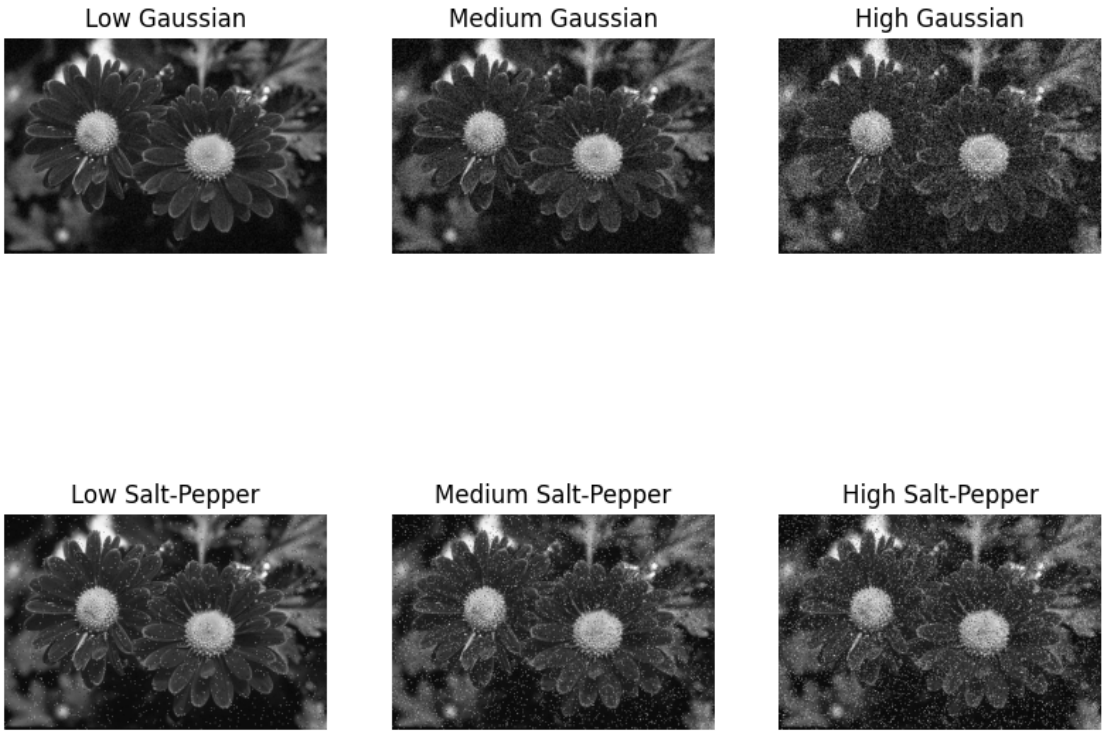
Image 4 (the polar bears) was selected for display. The polar bears have well-defined outlines, which makes it easy to assess how filters preserve edges when reducing noise. The white bears stand out against the snowy background, making it possible to evaluate edge preservation effectively, as any loss of edge definition will be noticeable. The background is relatively simple, allowing the focus to be on the edges of the polar bears rather than on background details.

9 Step 1.2. Add Noise to Images

In this part, Gaussian and Salt-and-Pepper noise were added, and the images were converted to grayscale for simplicity.

[2]:
```
#adding Gaussian noise
def gaussianNoise(image, mean=0, stddev=25):
    gaussian = np.random.normal(mean, stddev, image.shape)
    noisyImage = np.clip(image + gaussian, 0, 255).astype(np.uint8)
    return noisyImage

#adding Salt-and-Pepper noise
def saltAndPepperNoise(image, salt_prob=0.01, pepper_prob=0.01):
    noisyImage = np.copy(image)

    # Salt noise
    num_salt = np.ceil(salt_prob * image.size)
    coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.shape]
    noisyImage[coords[0], coords[1]] = 255

    # Pepper noise
    num_pepper = np.ceil(pepper_prob * image.size)
    coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.shape]
    noisyImage[coords[0], coords[1]] = 0

    return noisyImage

# Noisy Natural High-Detail Image
image = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/124084.jpg', cv2.IMREAD_GRAYSCALE)

#apply noises with different intensities
noisy_images = {
    "low_gaussian": gaussianNoise(image, stddev=10),
    "medium_gaussian": gaussianNoise(image, stddev=25),
    "high_gaussian": gaussianNoise(image, stddev=50),
    "low_sp": saltAndPepperNoise(image, salt_prob=0.01, pepper_prob=0.01),
    "medium_sp": saltAndPepperNoise(image, salt_prob=0.03, pepper_prob=0.03),
    "high_sp": saltAndPepperNoise(image, salt_prob=0.05, pepper_prob=0.05)
}

#display plots
plt.figure(figsize=(10, 8))
titles = ["Low Gaussian", "Medium Gaussian", "High Gaussian", "Low␣
 ↪Salt-Pepper", "Medium Salt-Pepper", "High Salt-Pepper"]
for i, (key, noisy_img) in enumerate(noisy_images.items()):
    plt.subplot(2, 3, i + 1)
```

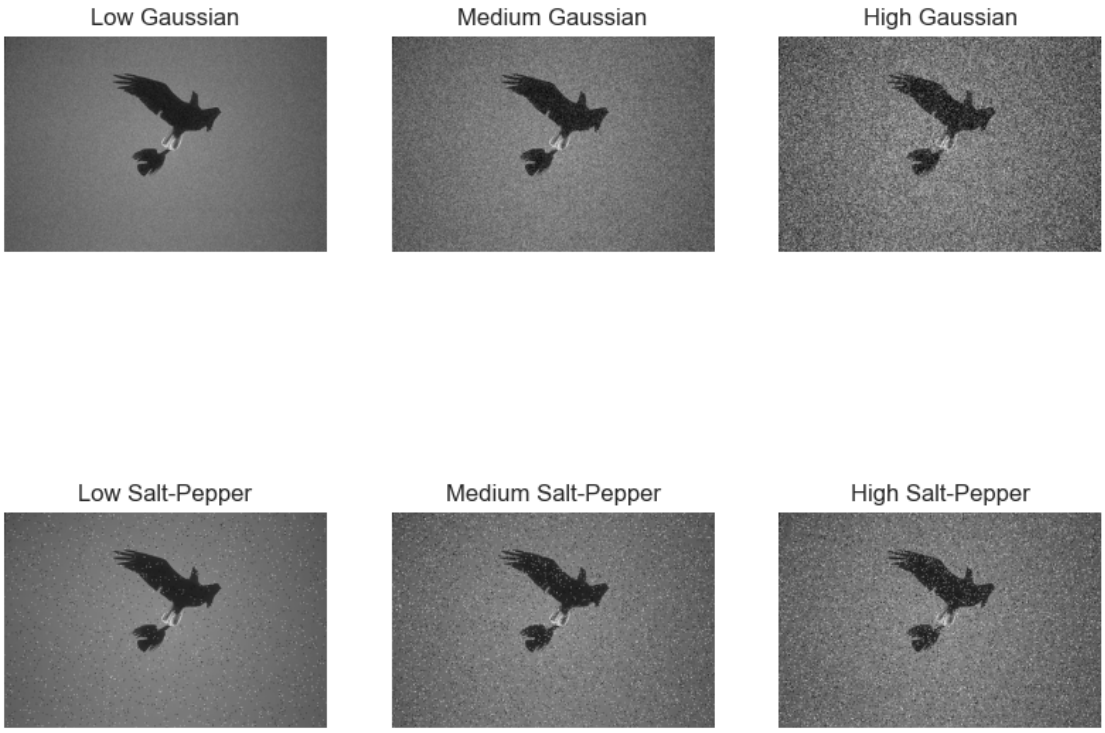
```
    plt.imshow(noisy_img, cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```



- Gaussian Noise: As the noise intensity increases from low to high, the intricate details in the flower petals gradually become obscured. The high level of Gaussian noise significantly disrupts the petal textures, causing a loss of fine details. The relatively plain background highlights the effect of noise on the flower edges.
- Salt-and-Pepper Noise: Salt-and-Pepper noise introduces random white and black specks across the image. The low level of this noise has minimal impact on the flower's structure, but as it increases to medium and high levels, the noise becomes very noticeable, masking details and interfering with the edges of the petals.

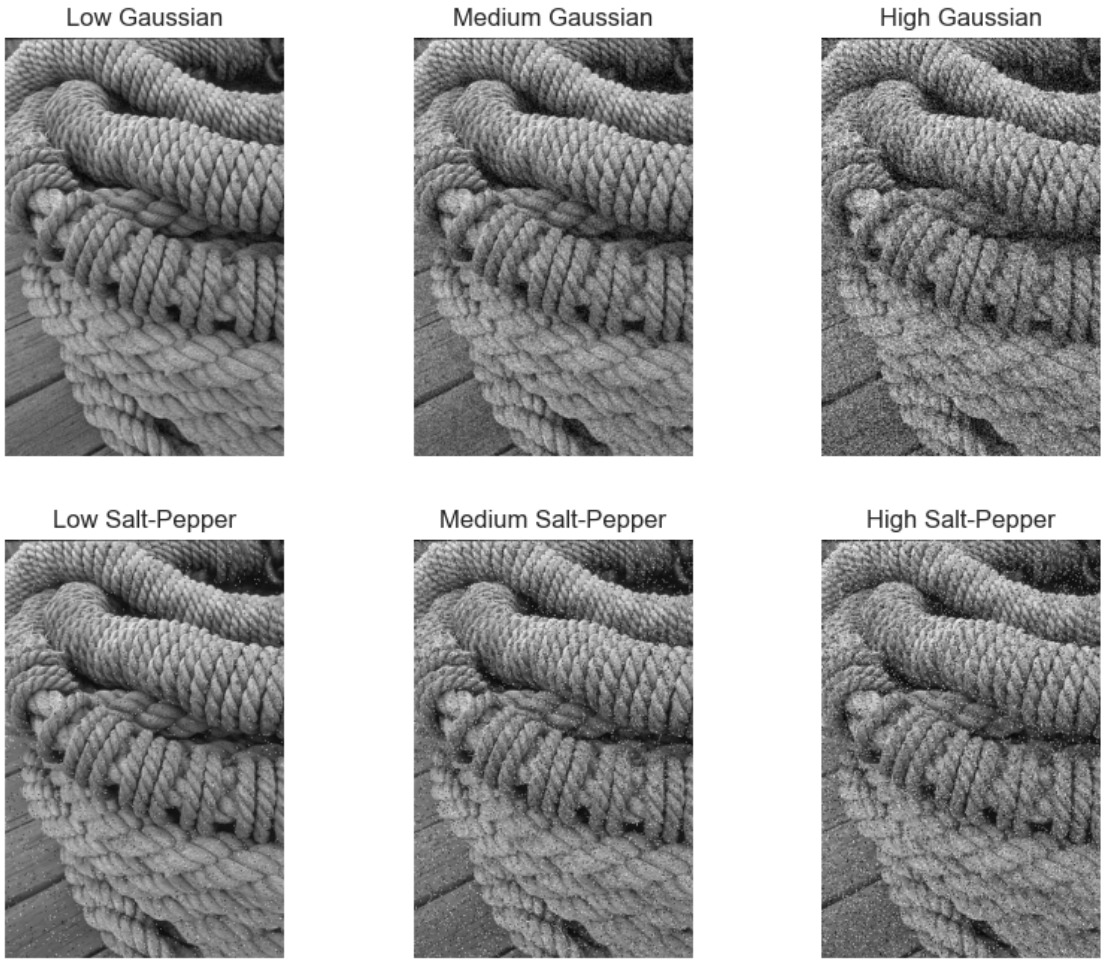
[8]:
```
# Noisy Natural Low-Detail Image
image2 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/135069.jpg', cv2.IMREAD_GRAYSCALE)

#apply noises with different intensities
noisy_images2 = {
    "low_gaussian": gaussianNoise(image2, stddev=10),
    "medium_gaussian": gaussianNoise(image2, stddev=25),
    "high_gaussian": gaussianNoise(image2, stddev=50),
```

```
    "low_sp": saltAndPepperNoise(image2, salt_prob=0.01, pepper_prob=0.01),
    "medium_sp": saltAndPepperNoise(image2, salt_prob=0.03, pepper_prob=0.03),
    "high_sp": saltAndPepperNoise(image2, salt_prob=0.05, pepper_prob=0.05)
}

#display plots
plt.figure(figsize=(10, 8))
titles = ["Low Gaussian", "Medium Gaussian", "High Gaussian", "Low␣
 ↪Salt-Pepper", "Medium Salt-Pepper", "High Salt-Pepper"]
for i, (key, noisy_img) in enumerate(noisy_images2.items()):
    plt.subplot(2, 3, i + 1)
    plt.imshow(noisy_img, cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```



- Gaussian Noise: The sky, being a uniform background, shows the added Gaussian noise more clearly as intensity increases. Low levels of noise create a grainy effect, while higher levels lead to significant distortion across the sky, impacting the smoothness of the background.
- Salt-and-Pepper Noise: The effect of Salt-and-Pepper noise is prominent against the plain background. Even low levels of this noise make the sky look speckled, while medium and high levels introduce more pronounced artifacts. The bird's silhouette stands out against the noisy background..

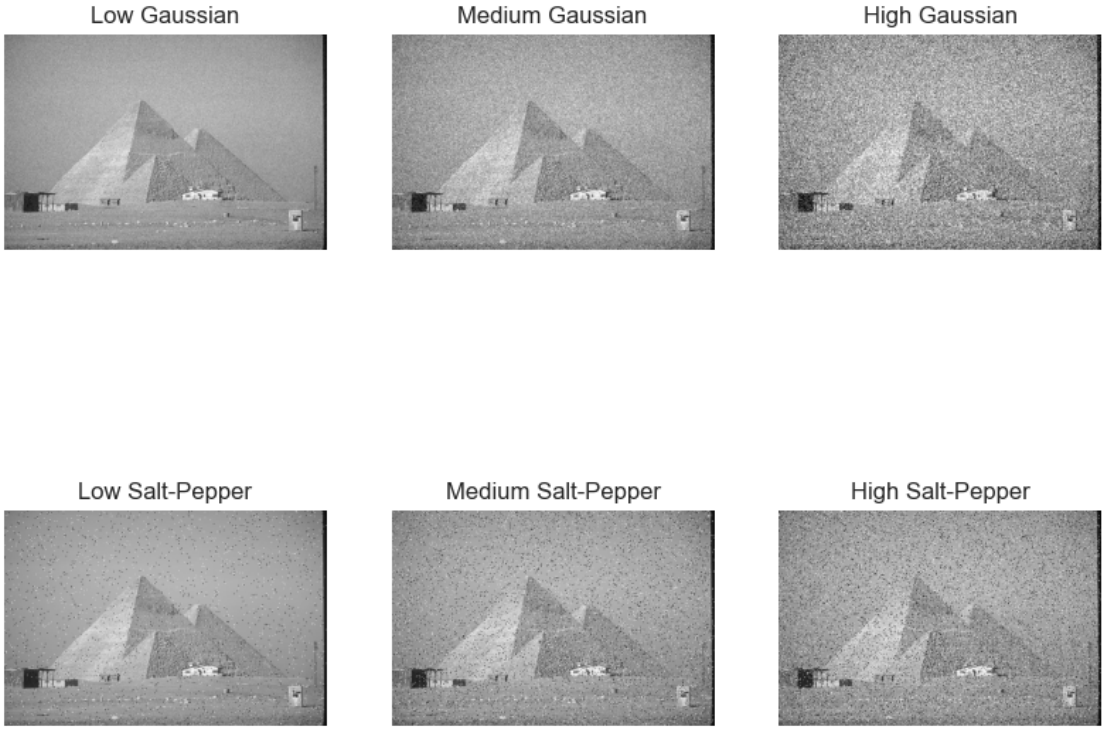
[15]:
```
# Noisy Influenced by Human High-Detail Image
image3 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/138032.jpg', cv2.IMREAD_GRAYSCALE)

#apply noises with different intensities
noisy_images3 = {
    "low_gaussian": gaussianNoise(image3, stddev=10),
    "medium_gaussian": gaussianNoise(image3, stddev=25),
    "high_gaussian": gaussianNoise(image3, stddev=50),
    "low_sp": saltAndPepperNoise(image3, salt_prob=0.01, pepper_prob=0.01),
    "medium_sp": saltAndPepperNoise(image3, salt_prob=0.03, pepper_prob=0.03),
    "high_sp": saltAndPepperNoise(image3, salt_prob=0.05, pepper_prob=0.05)
}

#display plots
plt.figure(figsize=(10, 8))
titles = ["Low Gaussian", "Medium Gaussian", "High Gaussian", "Low␣
 ↪Salt-Pepper", "Medium Salt-Pepper", "High Salt-Pepper"]
for i, (key, noisy_img) in enumerate(noisy_images3.items()):
    plt.subplot(2, 3, i + 1)
    plt.imshow(noisy_img, cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```



- Gaussian Noise: Gaussian noise affects the rope's detailed texture, especially at higher levels where the fiber patterns start to blur. The noise disrupts the fine structural details in the coiled rope, making it harder to distinguish between individual fibers at high noise levels.
- Salt-and-Pepper Noise: The rope's detailed texture makes it sensitive to Salt-and-Pepper noise. As intensity increases, the random white and black pixels interfere with the texture, making it challenging to maintain the rope's intricate appearance.

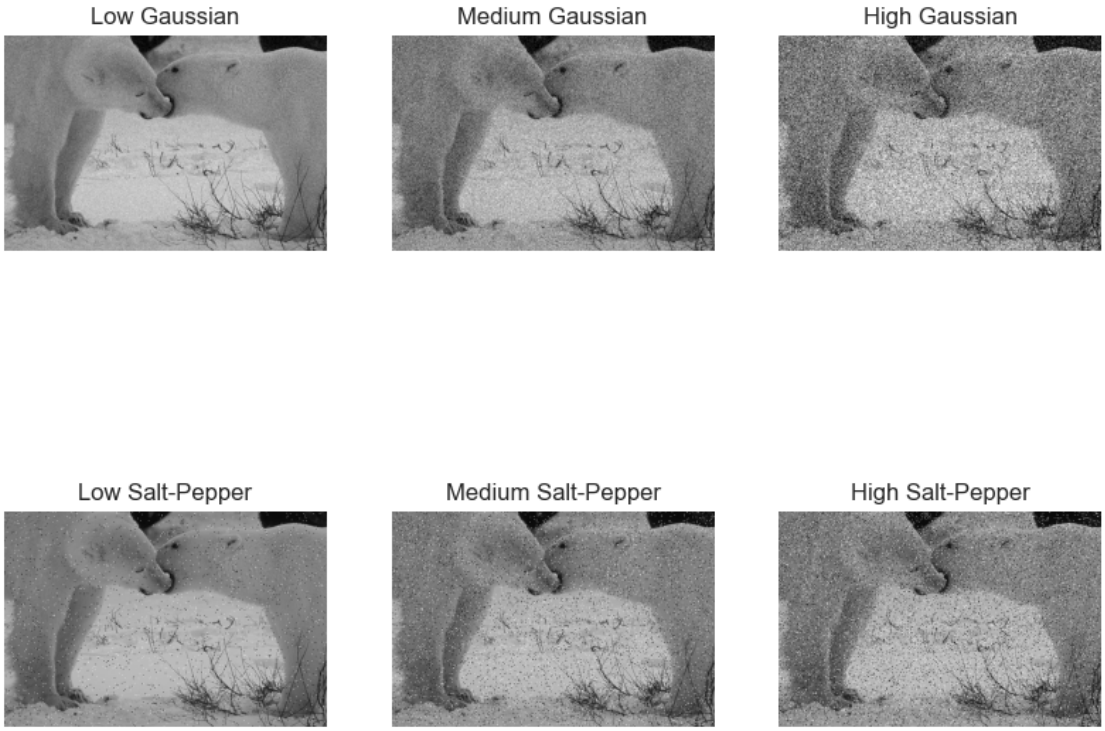
[18]:
```
# Noisy Influenced by Human Low-Detail Image
image4 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/161062.jpg', cv2.IMREAD_GRAYSCALE)

#apply noises with different intensities
noisy_images4 = {
    "low_gaussian": gaussianNoise(image4, stddev=10),
    "medium_gaussian": gaussianNoise(image4, stddev=25),
    "high_gaussian": gaussianNoise(image4, stddev=50),
    "low_sp": saltAndPepperNoise(image4, salt_prob=0.01, pepper_prob=0.01),
```

```
    "medium_sp": saltAndPepperNoise(image4, salt_prob=0.03, pepper_prob=0.03),
    "high_sp": saltAndPepperNoise(image4, salt_prob=0.05, pepper_prob=0.05)
}

#display plots
plt.figure(figsize=(10, 8))
titles = ["Low Gaussian", "Medium Gaussian", "High Gaussian", "Low␣
 ↪Salt-Pepper", "Medium Salt-Pepper", "High Salt-Pepper"]
for i, (key, noisy_img) in enumerate(noisy_images4.items()):
    plt.subplot(2, 3, i + 1)
    plt.imshow(noisy_img, cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```



- Gaussian Noise: The pyramids, with their smooth surfaces, display Gaussian noise quite prominently. At higher noise levels, the pyramid surfaces appear grainy, which distracts from the simplicity of the shapes. This image provides a straightforward canvas for evaluating how well filters can restore plain surfaces.
- Salt-and-Pepper Noise: Salt-and-Pepper noise causes random black and white specks on the uniform background and the pyramid surfaces. The noise stands out more on the simple background, making this image suitable for assessing filters' performance in removing high-contrast noise without impacting large, smooth areas.

[19]:
```
# Noisy Edge-Rich Image
image5 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/183055.jpg', cv2.IMREAD_GRAYSCALE)

#apply noises with different intensities
noisy_images5 = {
    "low_gaussian": gaussianNoise(image5, stddev=10),
    "medium_gaussian": gaussianNoise(image5, stddev=25),
    "high_gaussian": gaussianNoise(image5, stddev=50),
    "low_sp": saltAndPepperNoise(image5, salt_prob=0.01, pepper_prob=0.01),
    "medium_sp": saltAndPepperNoise(image5, salt_prob=0.03, pepper_prob=0.03),
    "high_sp": saltAndPepperNoise(image5, salt_prob=0.05, pepper_prob=0.05)
}

#display plots
plt.figure(figsize=(10, 8))
titles = ["Low Gaussian", "Medium Gaussian", "High Gaussian", "Low␣
 ↪Salt-Pepper", "Medium Salt-Pepper", "High Salt-Pepper"]
for i, (key, noisy_img) in enumerate(noisy_images5.items()):
    plt.subplot(2, 3, i + 1)
    plt.imshow(noisy_img, cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.show()
```



- Gaussian Noise: In this image, Gaussian noise particularly affects the bears' white fur, blending the edges with the snowy background at higher noise intensities. The increasing noise levels make it harder to preserve the contrast between the bears and the background.
- Salt-and-Pepper Noise: Salt-and-Pepper noise adds random specks across the image, which are quite noticeable on the plain background. The bears' outlines become harder to distinguish at high noise levels, challenging filters to maintain clarity around edges.

In summary, the two types of noise—Gaussian and Salt-and-Pepper—affect images in distinct ways across various image types. Gaussian noise, which adds a consistent graininess, impacts smooth backgrounds more noticeably, such as the sky in the bird image, making it easier to observe the effects of noise on uniform areas. Higher levels of Gaussian noise lead to a gradual loss of fine details in textured images, such as the flower and rope images, and make the edges in high-contrast regions, like the pyramids and polar bears, less distinct. Salt-and-Pepper noise, on the other hand, introduces random black and white specks, which are particularly disruptive to areas with large, plain surfaces (e.g., the sky and desert background around the pyramids). This noise type obscures fine details in textured images, making it challenging to preserve intricate patterns in the rope or flower petals.

10 Step 2. Apply Filters

In this section, I appllied simple filters (Box filter, Gaussian filter, and Median filter) and advanced filters (Adaptive mean filter, adaptive median filter, and Bilateral filter) on each noisy image with different kernel sizes.

[13]:
```
# Adaptive Mean Filter Function
def adaptive_mean_filter(image, kernel_size=3):
    height, width = image.shape      # Get the dimensions of the image
    pad_size = kernel_size // 2      # Pad the image to handle the borders
    padded_image = cv2.copyMakeBorder(image, pad_size, pad_size, pad_size,␣
 ↪pad_size, cv2.BORDER_REFLECT)

    output_image = np.zeros_like(image, dtype=np.float32)     # Create an empty␣
 ↪output image

    # Iterate through each pixel in the image
    for i in range(height):
        for j in range(width):
            local_region = padded_image[i:i+kernel_size, j:j+kernel_size]      ␣
 ↪       # Extract the local neighborhood

            local_mean = np.mean(local_region)             # Calculate the mean␣
 ↪of the neighborhood

            output_image[i, j] = local_mean              # Set the output pixel␣
 ↪value to the calculated mean
```

```
    output_image = np.clip(output_image, 0, 255).astype(np.uint8)      #␣
 ↪Convert back to uint8
    return output_image

# Adaptive Median Filter Function
def adaptive_median_filter(image, max_kernel_size=7):
    # Initialize output image
    output_image = np.zeros_like(image)
    padded_image = np.pad(image, max_kernel_size // 2, mode='reflect')

    # Process each pixel in the image
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            kernel_size = 3  # Start with the smallest kernel size
            pixel_filtered = image[i, j]  # Default to original pixel value

            while kernel_size <= max_kernel_size:
                # Extract the local neighborhood
                local_region = padded_image[i:i + kernel_size, j:j +␣
 ↪kernel_size]

                # Calculate median, minimum, and maximum values in the␣
 ↪neighborhood
                local_median = np.median(local_region)
                local_min = np.min(local_region)
                local_max = np.max(local_region)

                # Adaptive median filter conditions
                if local_min < local_median < local_max:
                    if local_min < image[i, j] < local_max:
                        pixel_filtered = image[i, j]  # Keep original pixel
                    else:
                        pixel_filtered = local_median  # Use median value
                    break  # Exit while loop as we have a valid result
                else:
                    # Increase kernel size
                    kernel_size += 2

            # Store the filtered pixel value
            output_image[i, j] = pixel_filtered

    return output_image

# Apply Filters Function
def applyFilters(image, kernel_size):
    # Simple filters
    box_filtered = cv2.blur(image, (kernel_size, kernel_size))  # Box Filter
```

```
    gaussian_filtered = cv2.GaussianBlur(image, (kernel_size, kernel_size),␣
↪sigmaX=0)  # Gaussian Filter
    median_filtered = cv2.medianBlur(image, kernel_size)  # Median Filter

    # Advanced filters
    adaptive_mean_filtered = cv2.boxFilter(image, -1, (kernel_size,␣
↪kernel_size), normalize=True)  # Adaptive Mean Filter
    adaptive_median_filtered = adaptive_median_filter(image,␣
↪max_kernel_size=kernel_size)  # Adaptive Median Filter
    bilateral_filtered = cv2.bilateralFilter(image, kernel_size, sigmaColor=75,␣
↪sigmaSpace=75)  # Bilateral Filter

    return {
        "Box": box_filtered,
        "Gaussian": gaussian_filtered,
        "Median": median_filtered,
        "Adaptive Mean": adaptive_mean_filtered,
        "Adaptive Median": adaptive_median_filtered,
        "Bilateral": bilateral_filtered
    }

# Kernel sizes to apply
kernel_sizes = [3, 5, 7]
```

11 I. Filtered Natural High-Detail Images:

[11]:
```
# Apply filters for each noisy image and each kernel size
for noise_type, noisy_image in noisy_images.items():
    for kernel_size in kernel_sizes:
        # Apply filters
        filtered_images = applyFilters(noisy_image, kernel_size)

        # Display results
        plt.figure(figsize=(15, 10))
        plt.suptitle(f'{noise_type.capitalize()} Noise - Kernel Size␣
↪{kernel_size}')
        for i, (filter_name, filtered_img) in enumerate(filtered_images.
↪items()):
            plt.subplot(2, 3, i + 1)
            plt.imshow(filtered_img, cmap='gray')
            plt.title(f"{filter_name} Filter")
            plt.axis('off')

        plt.show()
```

Low_gaussian Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_gaussian Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

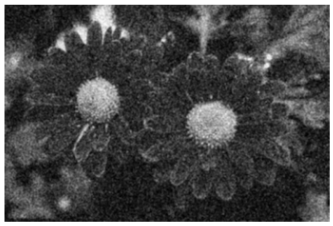
Adaptive Mean Filter

Adaptive Median Filter

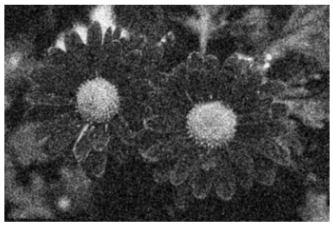
Bilateral Filter

High_gaussian Noise - Kernel Size 3

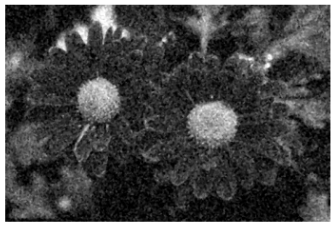
Box Filter



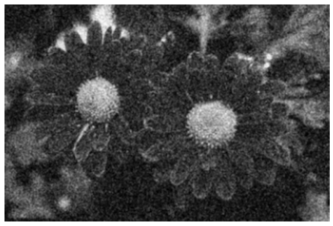
Gaussian Filter



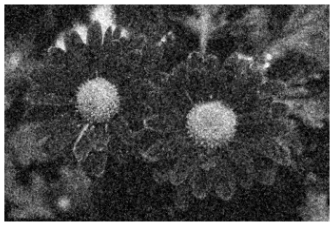
Median Filter



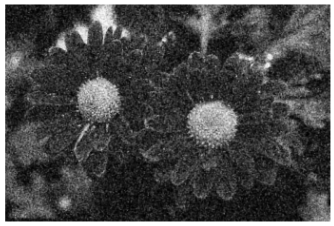
Adaptive Mean Filter



Adaptive Median Filter

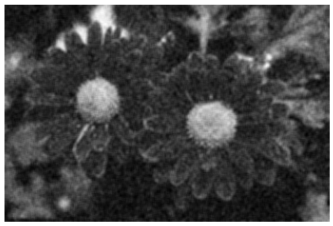


Bilateral Filter

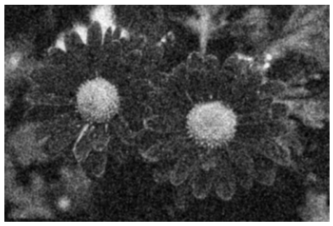


High_gaussian Noise - Kernel Size 5

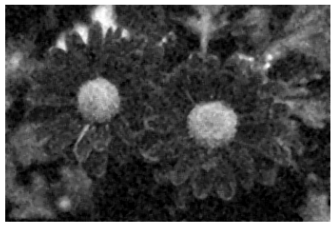
Box Filter



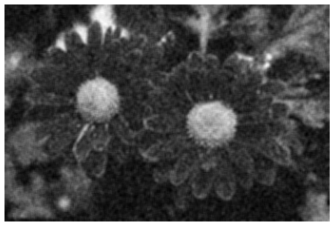
Gaussian Filter



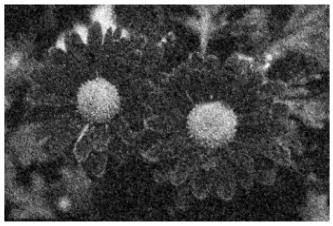
Median Filter



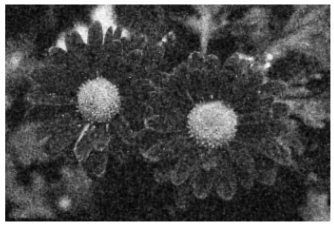
Adaptive Mean Filter



Adaptive Median Filter



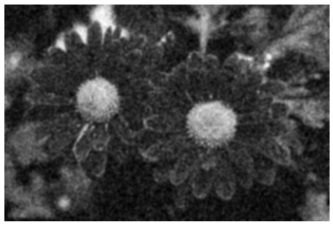
Bilateral Filter



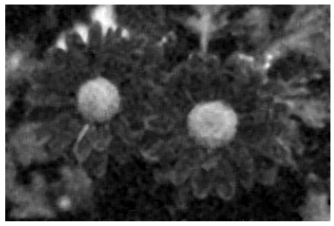
High_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

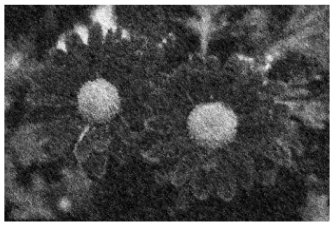


Median Filter

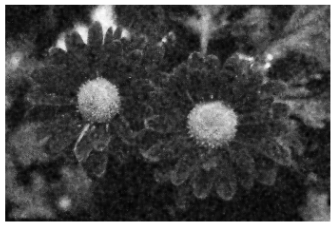


Adaptive Mean Filter

Adaptive Median Filter



Bilateral Filter



Low_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

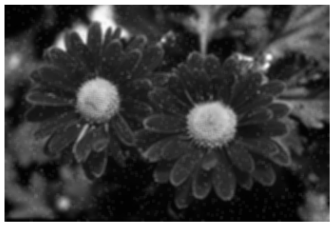
Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 5

Box Filter



Gaussian Filter

Median Filter

Adaptive Mean Filter

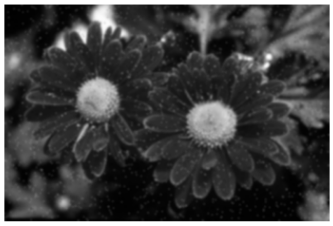
Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter



Median Filter

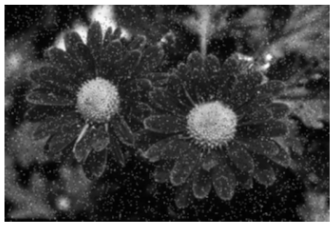
Adaptive Mean Filter

Adaptive Median Filter

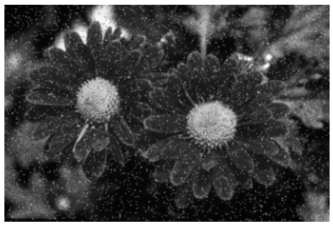
Bilateral Filter

Medium_sp Noise - Kernel Size 3

Box Filter

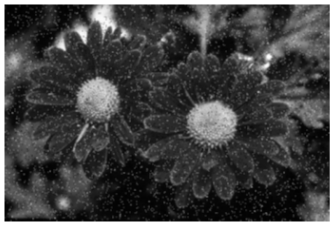


Gaussian Filter



Median Filter

Adaptive Mean Filter

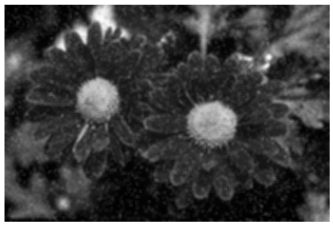


Adaptive Median Filter

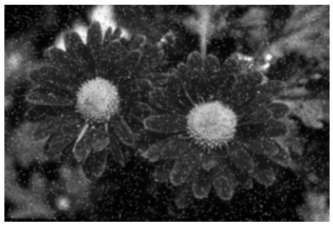
Bilateral Filter

Medium_sp Noise - Kernel Size 5

Box Filter

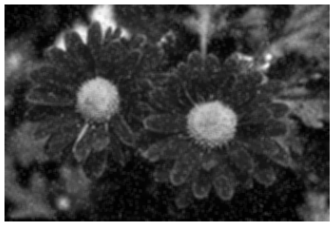


Gaussian Filter



Median Filter

Adaptive Mean Filter

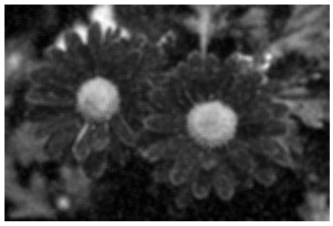


Adaptive Median Filter

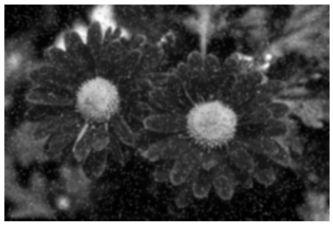
Bilateral Filter

Medium_sp Noise - Kernel Size 7

Box Filter

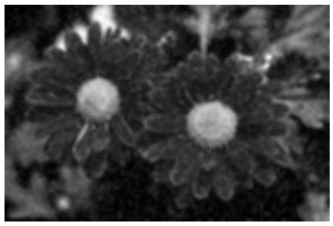


Gaussian Filter



Median Filter

Adaptive Mean Filter

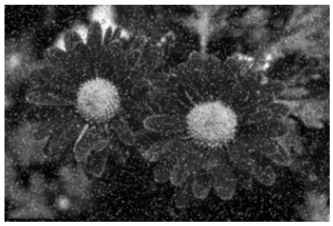


Adaptive Median Filter

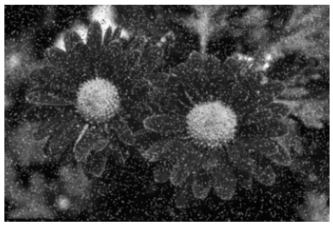
Bilateral Filter

High_sp Noise - Kernel Size 3

Box Filter

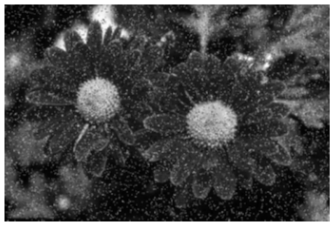


Gaussian Filter



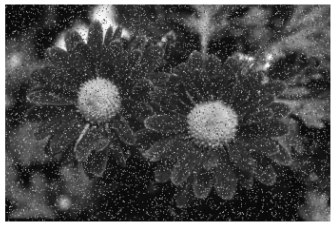
Median Filter

Adaptive Mean Filter



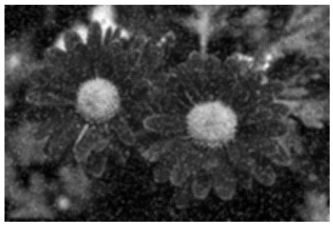
Adaptive Median Filter

Bilateral Filter

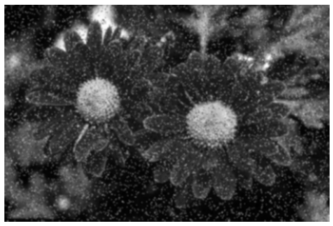


High_sp Noise - Kernel Size 5

Box Filter

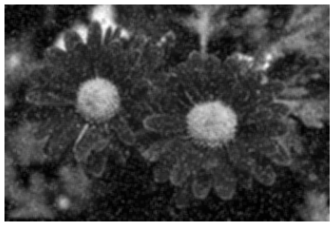


Gaussian Filter



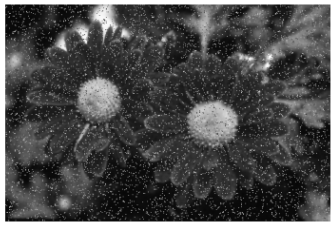
Median Filter

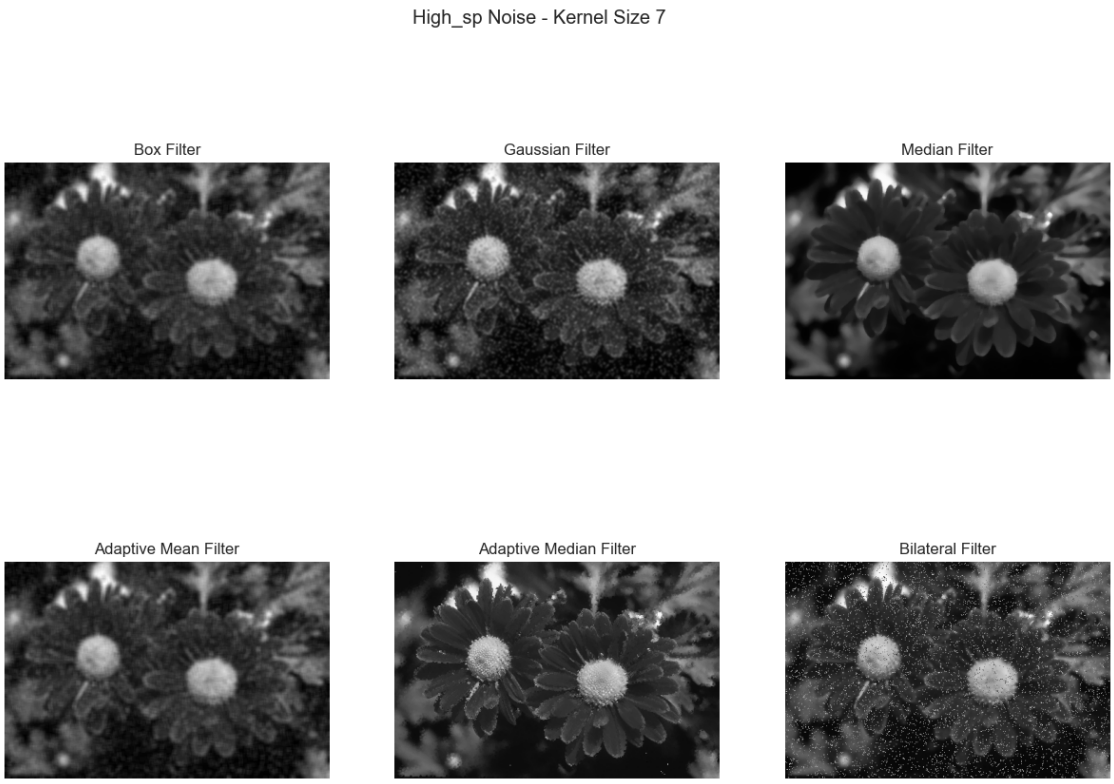
Adaptive Mean Filter



Adaptive Median Filter

Bilateral Filter





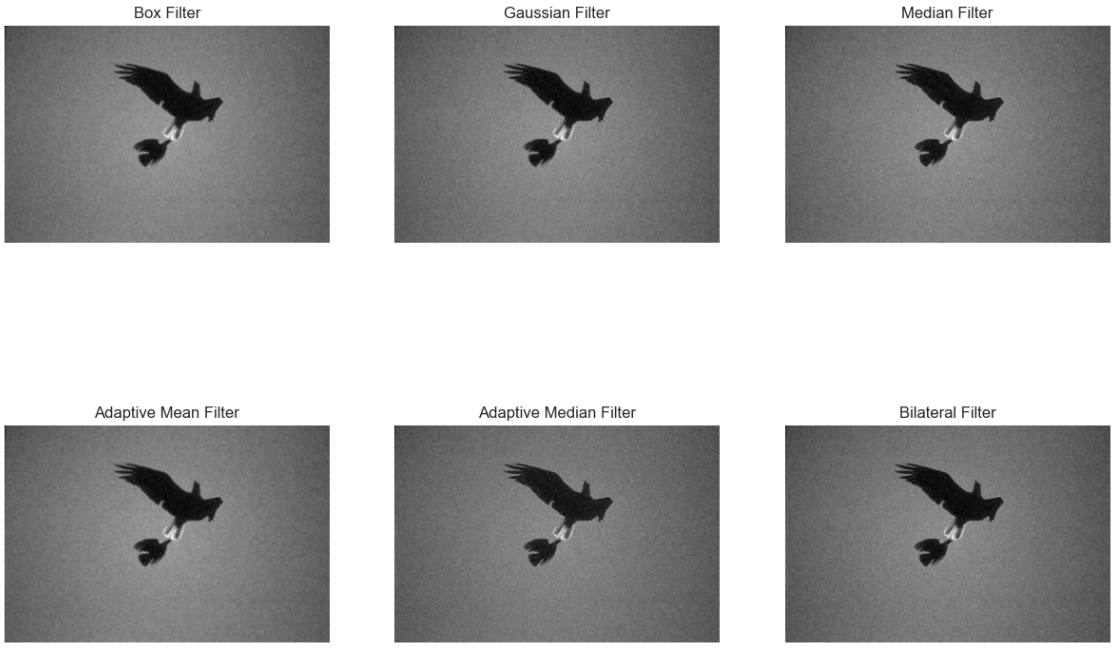
- For Gaussian noise, the Box filter smooth the image and reduce noise, though it tend to blur fine details, especially with larger kernel sizes and higher intensity levels. The Gaussian and Bilateral filters are more effective than the Median and the rest filters, as they reduce noise while better preserving fine details and edges. The adaptive Median filter seems to be the best in noise reduction especially at medium intinsity level.
- For Salt-and-Pepper noise, the Median and the Adaptive Median filters are the most effective across all kernel sizes, removing noise effectively while preserving edges. The Bilateral filter also provides good noise reduction with some edge preservation, though it's less effective than the Median filter at high noise levels. Meanwhile, the Box and Gaussian filters reduce noise but struggle with the distinct particles of salt-and-pepper noise, resulting in a blurrier effect without fully removing the noise.

12 II. Filtered Natural Low-Detail Images:

[12]:
```
# Apply filters for each noisy image and each kernel size
for noise_type, noisy_image in noisy_images2.items():
    for kernel_size in kernel_sizes:
        # Apply filters
        filtered_images = applyFilters2(noisy_image, kernel_size)
```

```
        # Display results
        plt.figure(figsize=(15, 10))
        plt.suptitle(f'{noise_type.capitalize()} Noise - Kernel Size␣
↪{kernel_size}')
        for i, (filter_name, filtered_img) in enumerate(filtered_images.
↪items()):
            plt.subplot(2, 3, i + 1)
            plt.imshow(filtered_img, cmap='gray')
            plt.title(f"{filter_name} Filter")
            plt.axis('off')

        plt.show()
```

Low_gaussian Noise - Kernel Size 3



Low_gaussian Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

High_gaussian Noise - Kernel Size 3

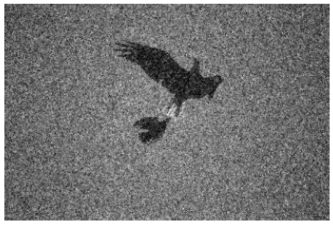
Box Filter

Gaussian Filter

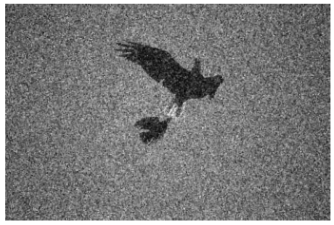
Median Filter

Adaptive Mean Filter

Adaptive Median Filter



Bilateral Filter



High_gaussian Noise - Kernel Size 5

Box Filter

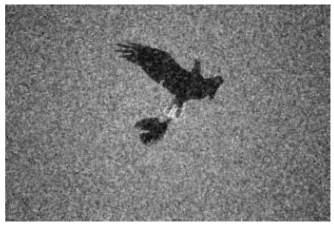
Gaussian Filter

Median Filter

Adaptive Mean Filter

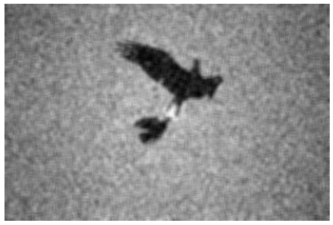
Adaptive Median Filter

Bilateral Filter



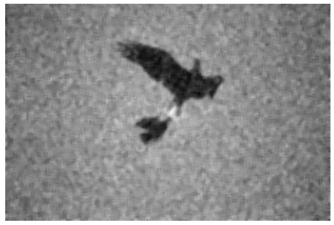
High_gaussian Noise - Kernel Size 7

Box Filter

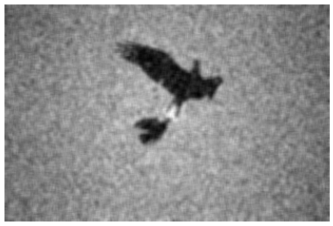


Gaussian Filter

Median Filter

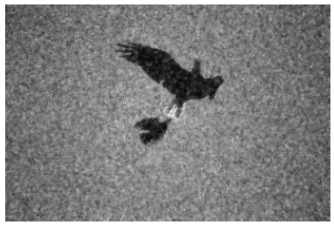


Adaptive Mean Filter



Adaptive Median Filter

Bilateral Filter

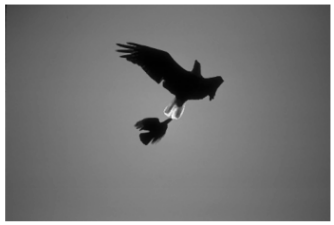


Low_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter



Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

High_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

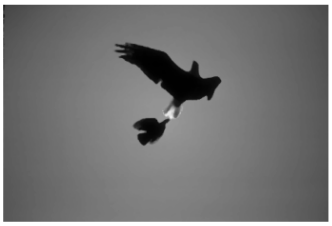
Bilateral Filter

High_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter



Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

High_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

- For Gaussian Noise, in low-detail images, both the Box and Gaussian filters reduce Gaussian noise but tend to blur the image, especially with larger kernel sizes and high intinsity levels, leading to a loss of clarity. The Median filter, also, is not very effective, especially at medium and high levels. Even advanced filters like the Bilateral and Adaptive Mean face limitations, especially at higher noise intensities and larger kernel sizes, where noise reduction leads to noticeable blurring and a loss of fine details.
- For Salt-and-Pepper Noise, the Median and Adaptive Mediam filters are notably effective in removing salt-and-pepper noise across all noise levels and kernel sizes, with the Median filter begin a little better than the Adaptive Median filter. The Bilateral filter also perform well in reducing noise while preserving some edges, though they are less effective than the Median and Adaptive Mediam filters at higher noise levels. Box and Gaussian filters are less suitable for salt-and-pepper noise, as they tend to blur the image without fully removing noise particles, particularly at higher noise levels and larger kernels.

13 III. Filtered Influenced by Human High-Detail Image:

[17]:
```
# Apply filters for each noisy image and each kernel size
for noise_type, noisy_image in noisy_images3.items():
    for kernel_size in kernel_sizes:
        # Apply filters
        filtered_images = applyFilters(noisy_image, kernel_size)

        # Display results
        plt.figure(figsize=(15, 10))
        plt.suptitle(f'{noise_type.capitalize()} Noise - Kernel Size␣
↪{kernel_size}')
        for i, (filter_name, filtered_img) in enumerate(filtered_images.
↪items()):
            plt.subplot(2, 3, i + 1)
            plt.imshow(filtered_img, cmap='gray')
            plt.title(f"{filter_name} Filter")
            plt.axis('off')

        plt.show()
```

Low_gaussian Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_gaussian Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

High_gaussian Noise - Kernel Size 3

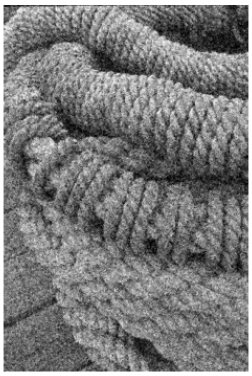
Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter



Bilateral Filter

High_gaussian Noise - Kernel Size 5

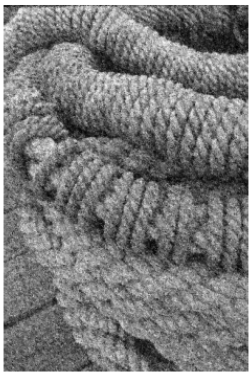
Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter



Bilateral Filter

High_gaussian Noise - Kernel Size 7

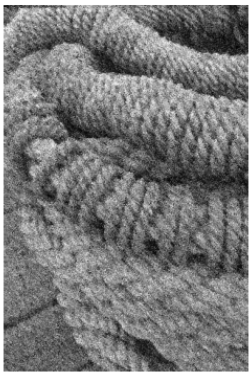
Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter



Bilateral Filter

Low_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

High_sp Noise - Kernel Size 3

Box Filter

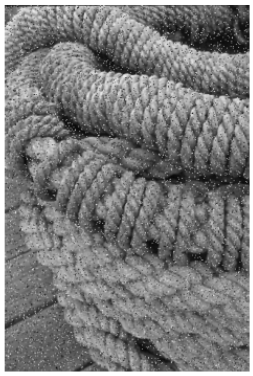
Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter



High_sp Noise - Kernel Size 5

Box Filter

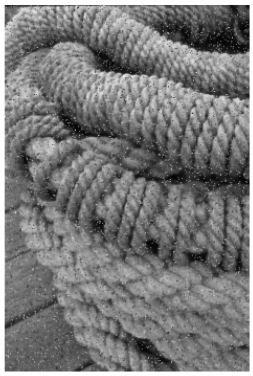
Gaussian Filter

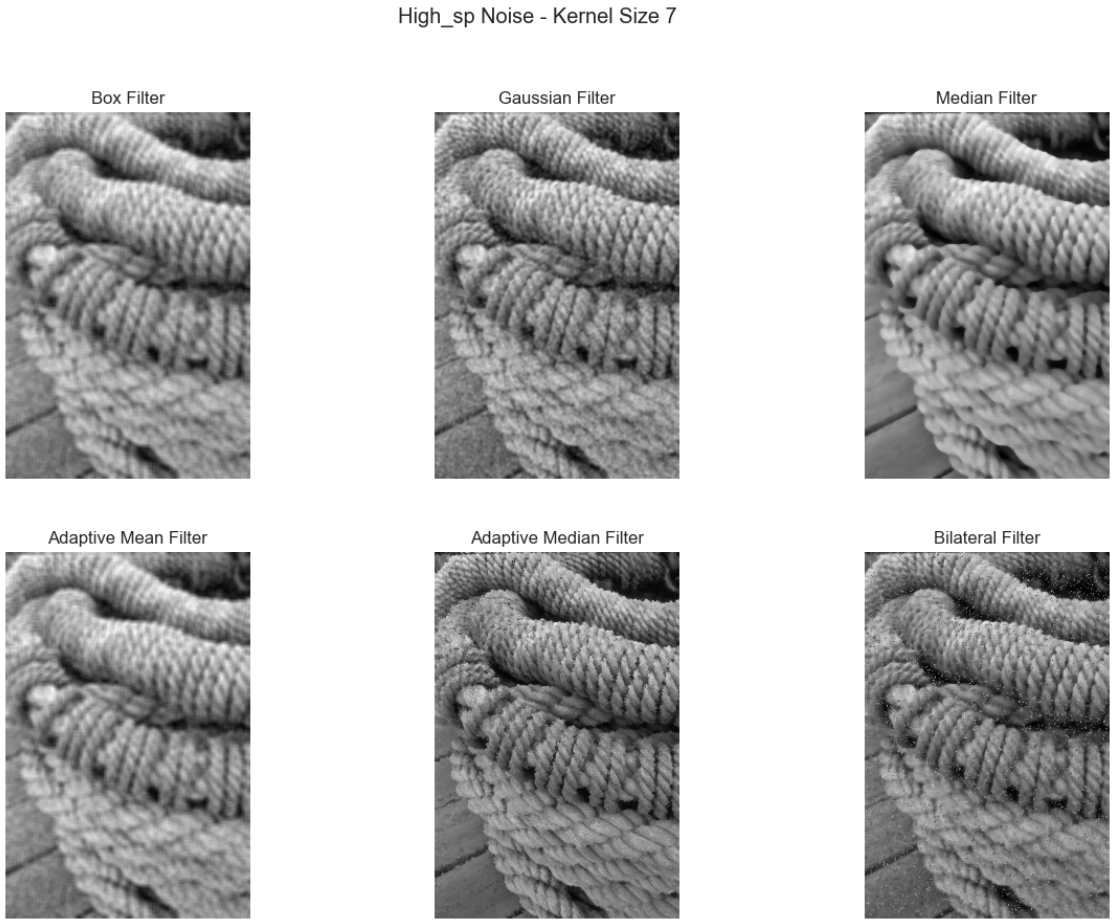
Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

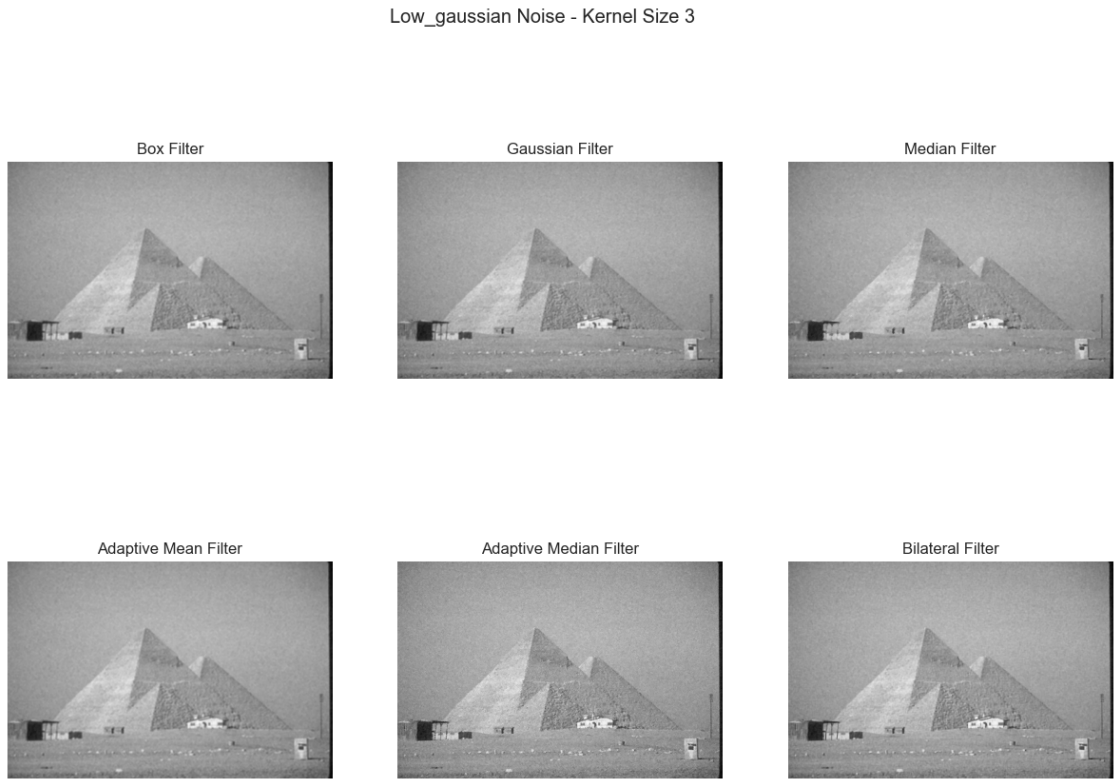




- For Gaussian noise, the Gaussian, the Adaptive Median and Biletral filters perform well in reducing Gaussian noise while preserving image details, with the Gaussian filter providing a good balance between smoothing and detail retention, especially at lower noise levels. However, as kernel size increases, all filters tend to blur the finer details to some extent, with the box and the adaptive mean filters causing the most significant blurring.
- In the case of salt-and-pepper noise, the median-based filters (simple and adaptive) excel at removing noise without overly blurring the image, as they are particularly suited to this type of noise, especially at lower noise and kernel levels. The bilateral filter also performs decently, but it retain some noise spots due to its emphasis on preserving detail. Meanwhile, the box and Gaussian filters reduce noise but at the cost of significant blurring, which diminishes the clarity of fine textures. Thus, for salt-and-pepper noise, median filters are the most effective choice.

14 IV. Filtered Influenced by Human Low-Detail Image:

[21]:
```
# Apply filters for each noisy image and each kernel size
for noise_type, noisy_image in noisy_images4.items():
    for kernel_size in kernel_sizes:
        # Apply filters
        filtered_images = applyFilters(noisy_image, kernel_size)

        # Display results
        plt.figure(figsize=(15, 10))
        plt.suptitle(f'{noise_type.capitalize()} Noise - Kernel Size␣
 ↪{kernel_size}')
        for i, (filter_name, filtered_img) in enumerate(filtered_images.
 ↪items()):
            plt.subplot(2, 3, i + 1)
            plt.imshow(filtered_img, cmap='gray')
            plt.title(f"{filter_name} Filter")
            plt.axis('off')

        plt.show()
```



Low_gaussian Noise - Kernel Size 5

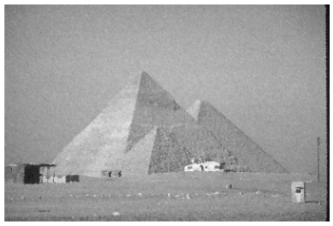
Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter



Bilateral Filter

Low_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

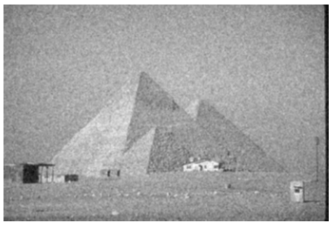
Adaptive Mean Filter

Adaptive Median Filter

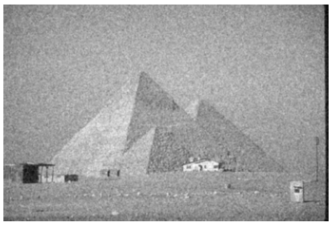
Bilateral Filter

Medium_gaussian Noise - Kernel Size 3

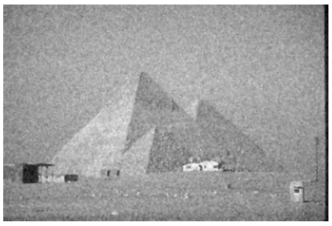
Box Filter



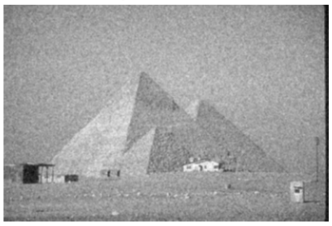
Gaussian Filter



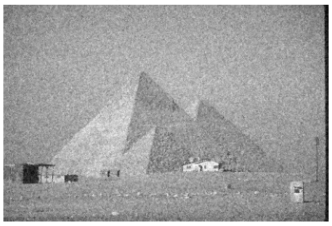
Median Filter



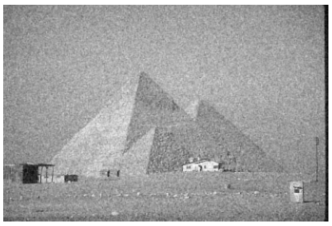
Adaptive Mean Filter



Adaptive Median Filter



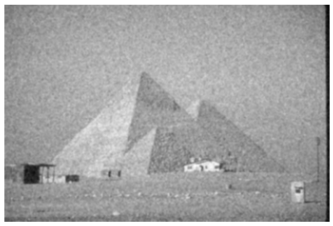
Bilateral Filter



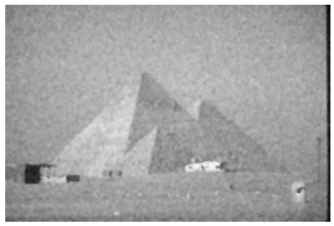
Medium_gaussian Noise - Kernel Size 5

Box Filter

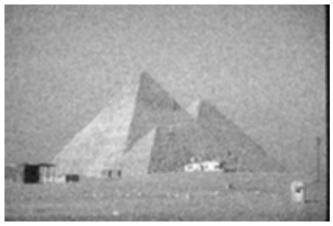
Gaussian Filter



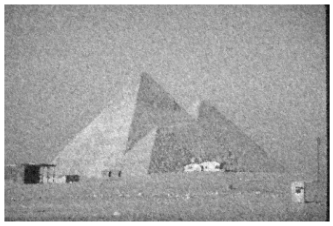
Median Filter



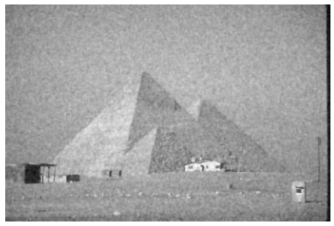
Adaptive Mean Filter



Adaptive Median Filter

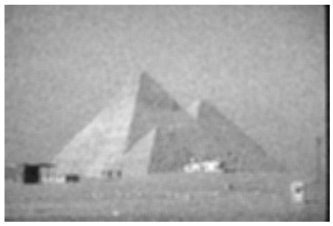


Bilateral Filter

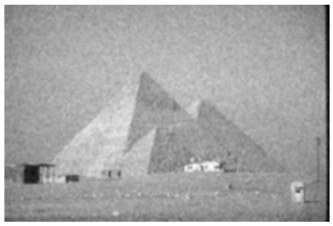


Medium_gaussian Noise - Kernel Size 7

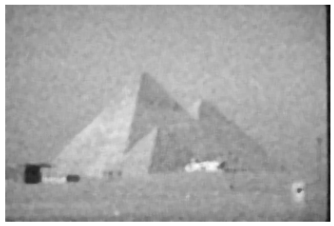
Box Filter



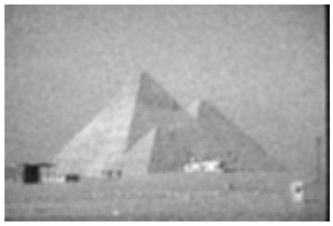
Gaussian Filter



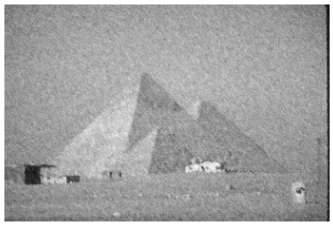
Median Filter



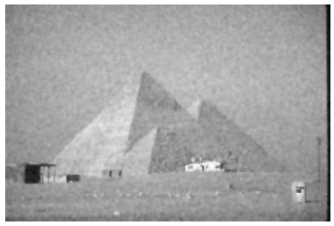
Adaptive Mean Filter



Adaptive Median Filter

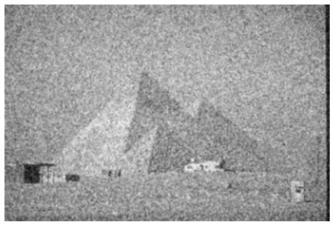


Bilateral Filter

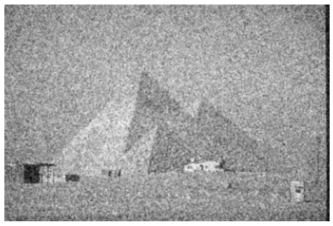


High_gaussian Noise - Kernel Size 3

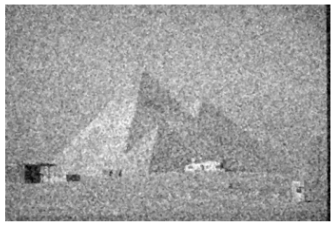
Box Filter



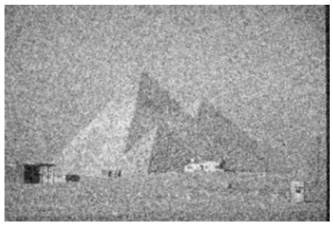
Gaussian Filter



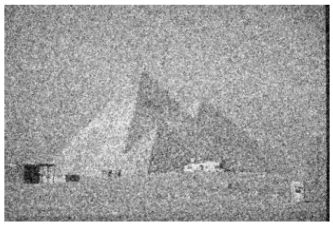
Median Filter



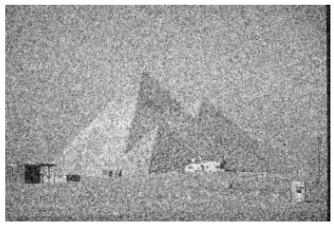
Adaptive Mean Filter



Adaptive Median Filter

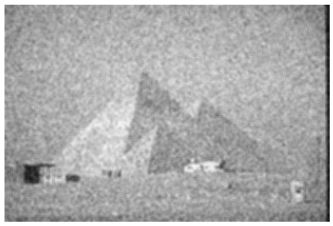


Bilateral Filter

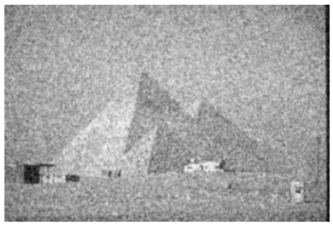


High_gaussian Noise - Kernel Size 5

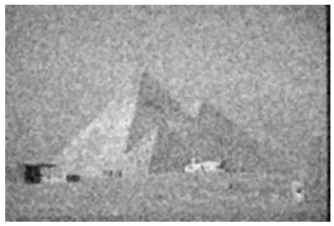
Box Filter



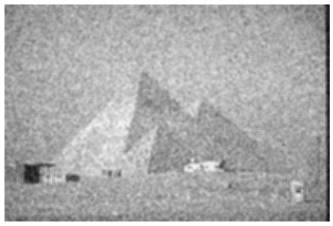
Gaussian Filter



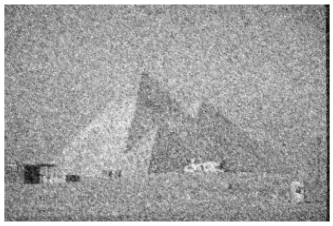
Median Filter



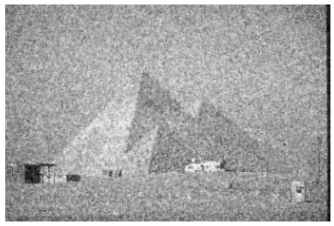
Adaptive Mean Filter



Adaptive Median Filter

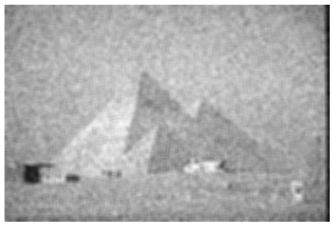


Bilateral Filter

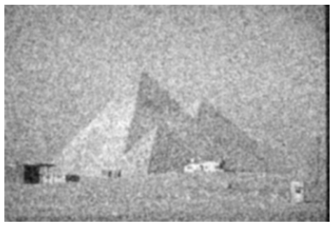


High_gaussian Noise - Kernel Size 7

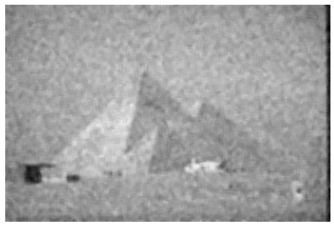
Box Filter



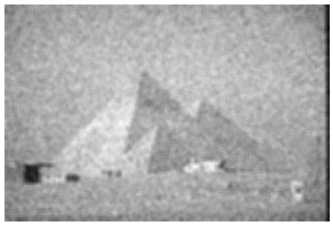
Gaussian Filter



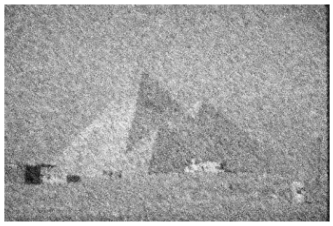
Median Filter



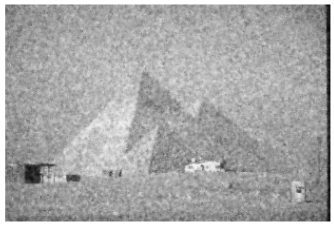
Adaptive Mean Filter



Adaptive Median Filter



Bilateral Filter



Low_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 3

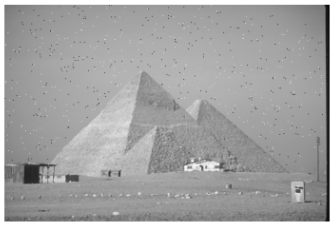
Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter



Bilateral Filter

Medium_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

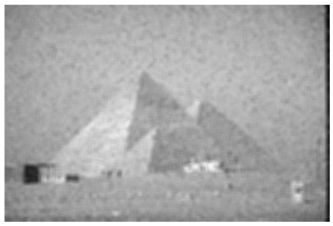
Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 7

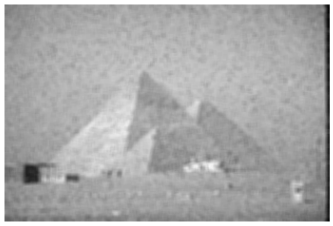
Box Filter



Gaussian Filter

Median Filter

Adaptive Mean Filter

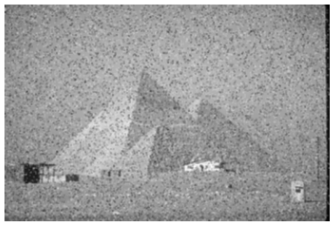


Adaptive Median Filter

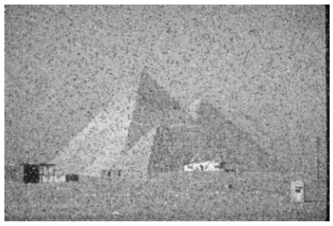
Bilateral Filter

High_sp Noise - Kernel Size 3

Box Filter

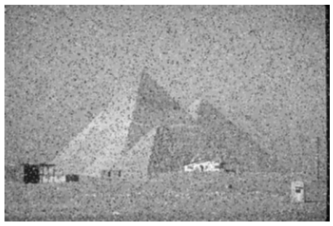


Gaussian Filter

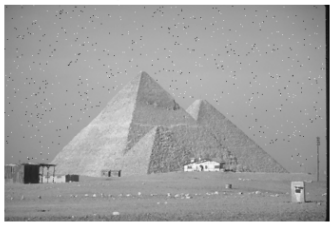


Median Filter

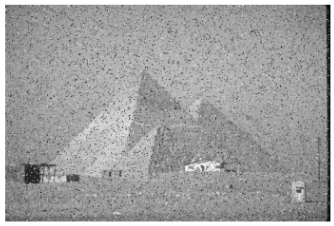
Adaptive Mean Filter



Adaptive Median Filter

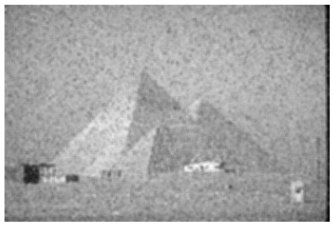


Bilateral Filter

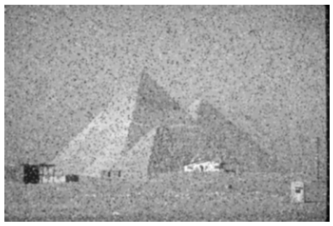


High_sp Noise - Kernel Size 5

Box Filter

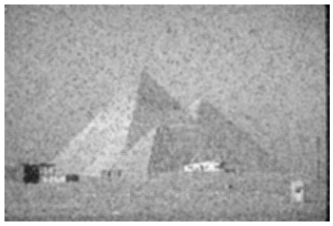


Gaussian Filter



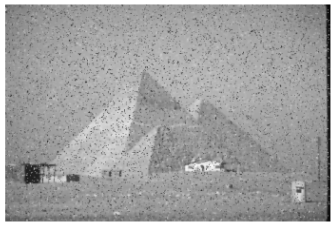
Median Filter

Adaptive Mean Filter



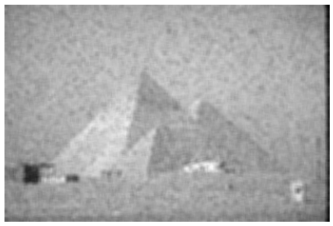
Adaptive Median Filter

Bilateral Filter

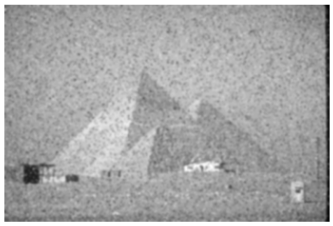


High_sp Noise - Kernel Size 7

Box Filter

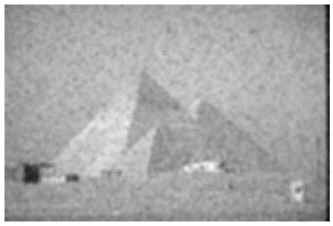


Gaussian Filter



Median Filter

Adaptive Mean Filter



Adaptive Median Filter

Bilateral Filter

- For Gaussian noise, the Gaussian and Bilateral filters provide consistent smoothing but blur fine details, which are essential for retaining image clarity, but they are the best among the rest here, especially at lower intensity and kerenl levels. The box filter, while straightforward, has a limited noise-reduction effect and introduce a uniform blur across the image. The Median filter is less effective for noise reduction compared to other filters, even the adaptive filters are not fully maintain texture at higher noise intensities and larger kernel sizes, where some blurring is inevitable. The Adaptive Median filter seems to be the best choice here.
- In the case of salt-and-pepper noise, the Median and Adaptive Median filters excel in removing the scattered noise points effectively, with the Median filter being slightly better. The box and Gaussian filters offer smoothing but struggle to remove high-contrast noise spots completely. The adaptive mean filter performs similarly to the Gaussian filter, balancing smoothing with some level of detail retention. The bilateral filter, while preserving edges, shows limitations in fully suppressing salt-and-pepper noise, especially in areas of uniform color or low texture.

15 V. Filtered Edge-Rich Image:

[22]:
```
# Apply filters for each noisy image and each kernel size
for noise_type, noisy_image in noisy_images5.items():
    for kernel_size in kernel_sizes:
        # Apply filters
        filtered_images = applyFilters(noisy_image, kernel_size)

        # Display results
        plt.figure(figsize=(15, 10))
        plt.suptitle(f'{noise_type.capitalize()} Noise - Kernel Size␣
↪{kernel_size}')
        for i, (filter_name, filtered_img) in enumerate(filtered_images.
↪items()):
            plt.subplot(2, 3, i + 1)
            plt.imshow(filtered_img, cmap='gray')
            plt.title(f"{filter_name} Filter")
            plt.axis('off')

        plt.show()
```

Low_gaussian Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_gaussian Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

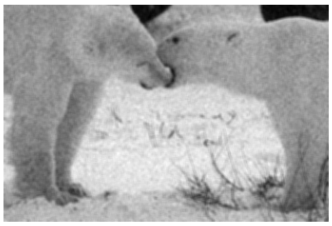
Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

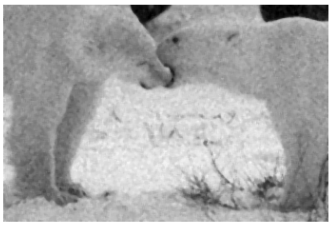
Medium_gaussian Noise - Kernel Size 5

Box Filter

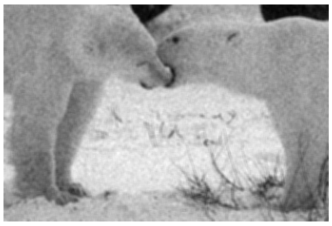


Gaussian Filter

Median Filter



Adaptive Mean Filter



Adaptive Median Filter

Bilateral Filter

Medium_gaussian Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

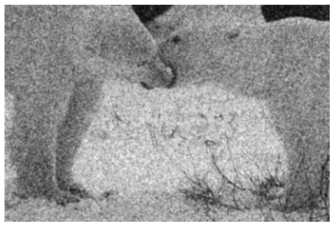
Adaptive Mean Filter

Adaptive Median Filter

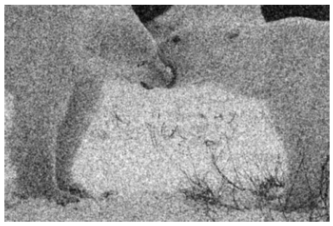
Bilateral Filter

High_gaussian Noise - Kernel Size 3

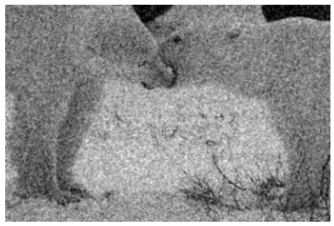
Box Filter



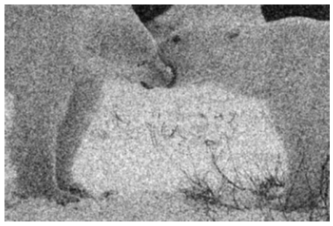
Gaussian Filter



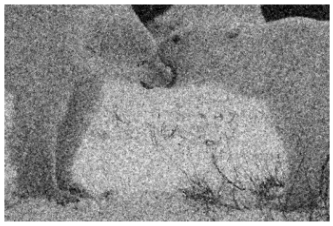
Median Filter



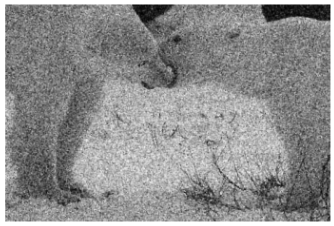
Adaptive Mean Filter



Adaptive Median Filter

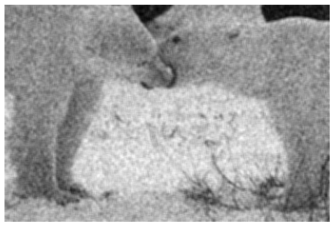


Bilateral Filter

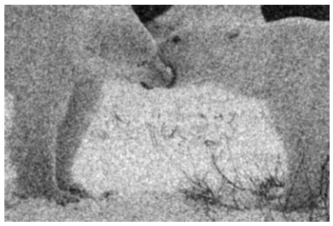


High_gaussian Noise - Kernel Size 5

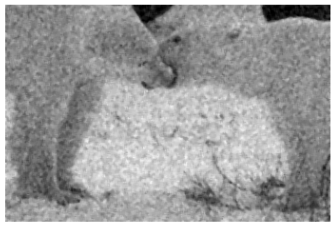
Box Filter



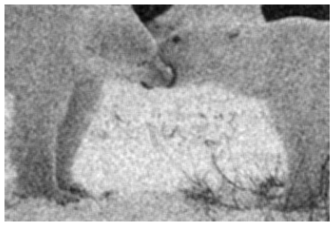
Gaussian Filter



Median Filter

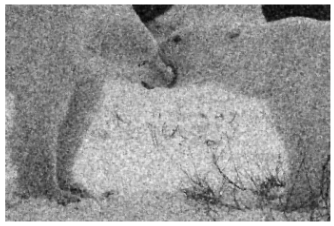


Adaptive Mean Filter



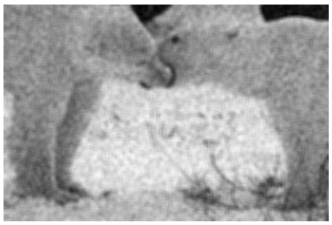
Adaptive Median Filter

Bilateral Filter

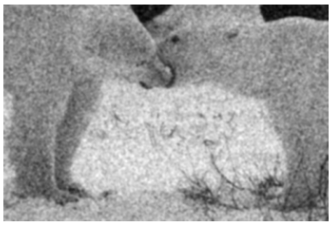


High_gaussian Noise - Kernel Size 7

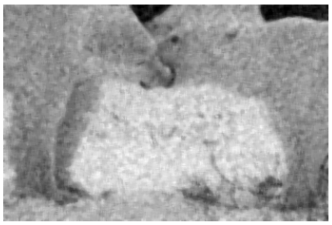
Box Filter



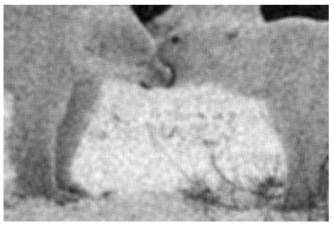
Gaussian Filter



Median Filter

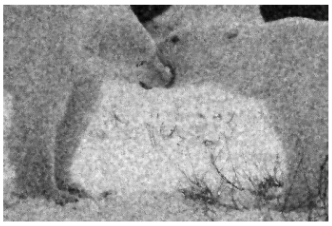


Adaptive Mean Filter



Adaptive Median Filter

Bilateral Filter



Low_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Low_sp Noise - Kernel Size 7

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 3

Box Filter

Gaussian Filter

Median Filter

Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 5

Box Filter

Gaussian Filter

Median Filter

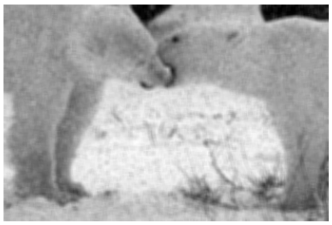
Adaptive Mean Filter

Adaptive Median Filter

Bilateral Filter

Medium_sp Noise - Kernel Size 7

Box Filter



Gaussian Filter

Median Filter

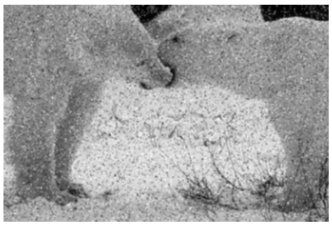
Adaptive Mean Filter

Adaptive Median Filter

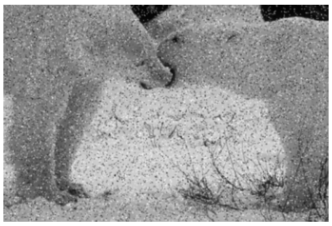
Bilateral Filter

High_sp Noise - Kernel Size 3

Box Filter

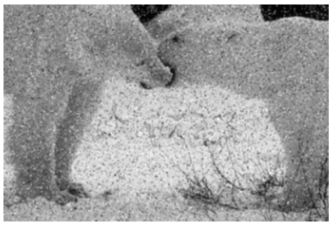


Gaussian Filter



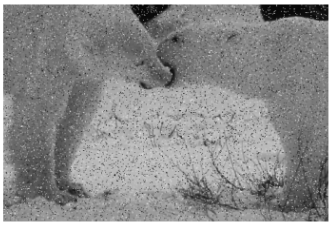
Median Filter

Adaptive Mean Filter



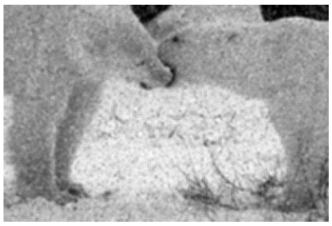
Adaptive Median Filter

Bilateral Filter

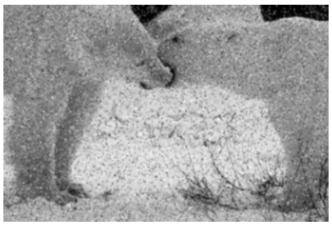


High_sp Noise - Kernel Size 5

Box Filter

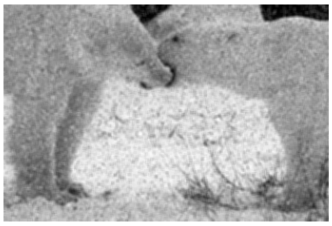


Gaussian Filter



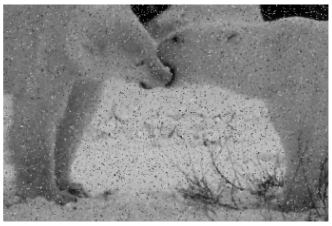
Median Filter

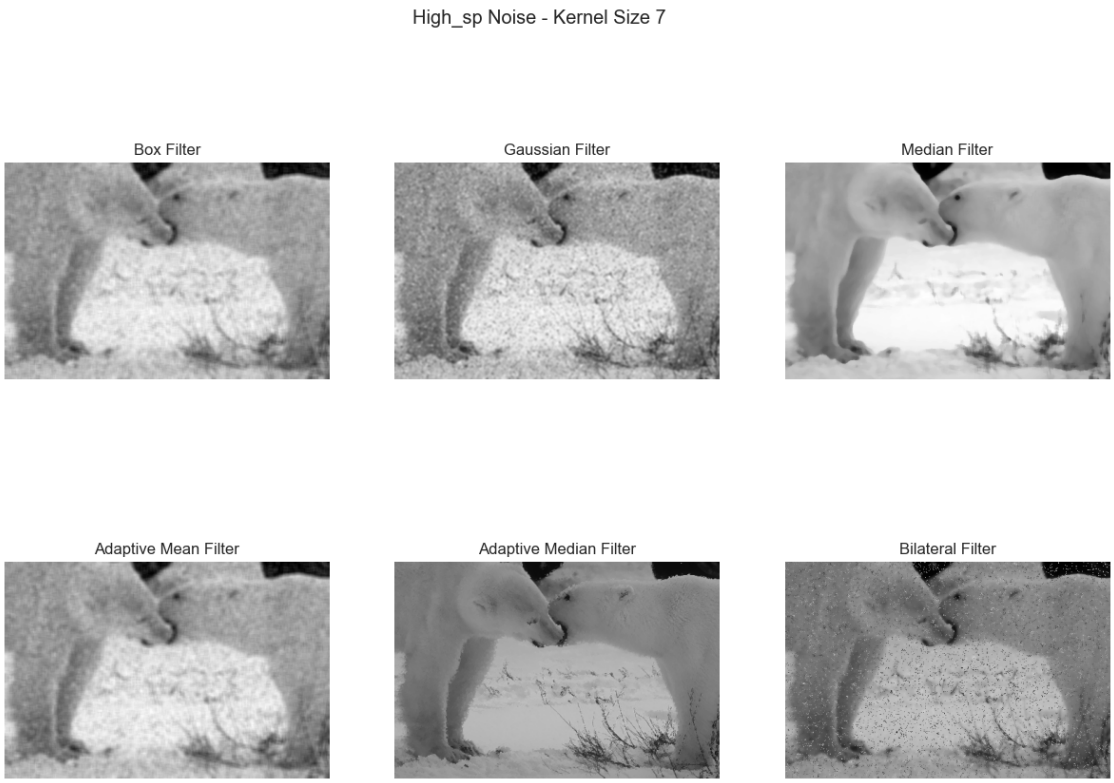
Adaptive Mean Filter



Adaptive Median Filter

Bilateral Filter





- For the Gausssain noise, the Gaussian filter, which applies a weighted average, smooths noise with minimal blurring, and is more effective in maintaining some edge details, though larger kernels still lead to more blurring, while the Bilateral filter reduces noise in flat areas and preserves edges effectively. The Box and Adaptive Mean filters reduce noise but tend to blur details more. The Box filter applies a simple averaging across pixels, which reduces noise but tends to blur details, especially as the kernel size increases.
- In the case of salt-and-pepper noise, the Median and Adaptive Mean filters dynamically adjusts based on local intensity, effectively balancing noise reduction and detail retention. Larger kernels increase smoothing but also slightly blur edges. The Bilateral filter is unique in preserving edges by weighting pixels based on both spatial distance and intensity difference, maintaining fine details even as the kernel size grows, though its effectiveness decreases with very high noise levels. The rest of filters have a poor performance on the noisy images with different intensities and kernels levels.

Across all varied image types, larger kernels enhance noise reduction but often at the cost of blurring edges. However, the specific effects are more noticeable in high-detail and edge-rich images, where edge clarity is critical, and in low-detail images, where excessive blurring can remove subtle textures. Salt-and-pepper noise is easier to filter than Gaussian noise because it appears as distinct black and white specks, allowing filters like the Median and the Adaptive Median filters to remove it by targeting isolated pixels without affecting surrounding details. Gaussian noise, however, spreads

evenly across the image as a fine grain, making it harder to separate from real details, so removing it often requires more advanced filters, such as the Gaussian or Bilateral filter, that balance noise reduction with edge preservation. In addition, smaller kernels tend to retain edge details better, while larger kernels enhance noise reduction at the cost of some blurring.

16 Step 3.1. MSE and PSNR

- MSE measures the average squared difference between pixel intensities in the original (clean) and filtered images. A lower MSE value indicates that the filtered image is closer to the original.
- PSNR is derived from MSE and represents the ratio between the maximum possible pixel value and the MSE. It is measured in decibels (dB). A higher PSNR indicates better image quality and less distortion.

MSE and PSNR for the Natural High-Detail Image:

[26]:
```
# MSE and PSNR calculations for the Natural High-Detail Image
def calculate_mse(original, filtered):
    return np.mean((original - filtered) ** 2)

def calculate_psnr(original, filtered):
    mse = calculate_mse(original, filtered)
    if mse == 0: #perfect match
        return float('inf')  #infinite PSNR if images are identical
    max_pixel = 255.0
    return 10 * np.log10((max_pixel ** 2) / mse)
```

[27]:
```
# Natural High-Detail Image
original_image = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/124084.jpg', cv2.IMREAD_GRAYSCALE)

#apply filters with different kernel sizes and calculate MSE, PSNR
kernel_sizes = [3, 5, 7]
results = []

for noise_type, noisy_img in noisy_images.items():
    for kernel_size in kernel_sizes:
        filtered_images = applyFilters(noisy_img, kernel_size)

        for filter_name, filtered_img in filtered_images.items():
            mse_value = calculate_mse(original_image, filtered_img)
            psnr_value = calculate_psnr(original_image, filtered_img)

            # Store results
            results.append({
                "Noise Type": noise_type,
                "Filter": filter_name,
```

```
                "Kernel Size": kernel_size,
                "MSE": mse_value,
                "PSNR": psnr_value
            })

#convert results to a DataFrame for easy analysis
results_df = pd.DataFrame(results)
results_df
```

[27]:

	Noise Type	Filter	Kernel Size	MSE	PSNR
0	low_gaussian	Box	3	25.359836	34.089339
1	low_gaussian	Gaussian	3	24.984210	34.154147
2	low_gaussian	Median	3	29.078665	33.495059
3	low_gaussian	Adaptive Mean	3	25.359836	34.089339
4	low_gaussian	Adaptive Median	3	45.177713	31.581561
..	…	…	…	…	…
103	high_sp	Gaussian	7	66.566149	29.898269
104	high_sp	Median	7	24.175957	34.296967
105	high_sp	Adaptive Mean	7	74.844813	29.389187
106	high_sp	Adaptive Median	7	25.427646	34.077742
107	high_sp	Bilateral	7	31.232816	33.184692

[108 rows x 5 columns]

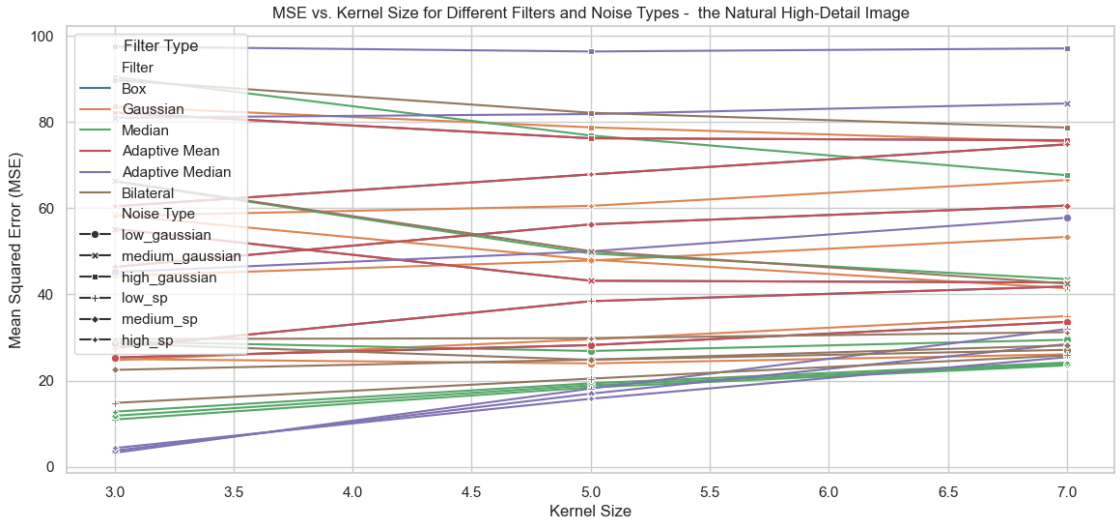
17 Due to the large number of cases to compare and discuss, I employed the following two approaches:

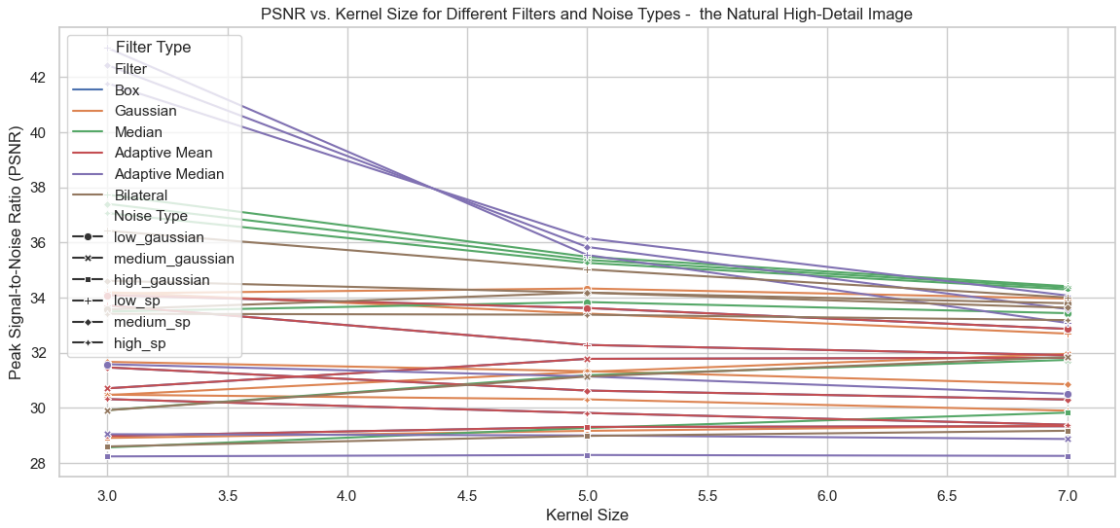
18 a. Line plot to visualize the trends and relationships in MSE and PSNR values across the filtered results:

- The Natural High-Detail Image:

[28]:
```
# Convert results to DataFrame
results_df = pd.DataFrame(results)

# Set up the plotting style
sns.set(style="whitegrid")

# Plot MSE for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df, x="Kernel Size", y="MSE", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("MSE vs. Kernel Size for Different Filters and Noise Types -  the␣
 ↪Natural High-Detail Image")
plt.ylabel("Mean Squared Error (MSE)")
plt.xlabel("Kernel Size")
```

```
plt.legend(loc="best", title="Filter Type")
plt.show()

# Plot PSNR for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df, x="Kernel Size", y="PSNR", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("PSNR vs. Kernel Size for Different Filters and Noise Types -  the␣
 ↪Natural High-Detail Image")
plt.ylabel("Peak Signal-to-Noise Ratio (PSNR)")
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()

# Check the total number of rows
total_rows = results_df.shape[0]
print(f"Total rows in DataFrame: {total_rows}")
```





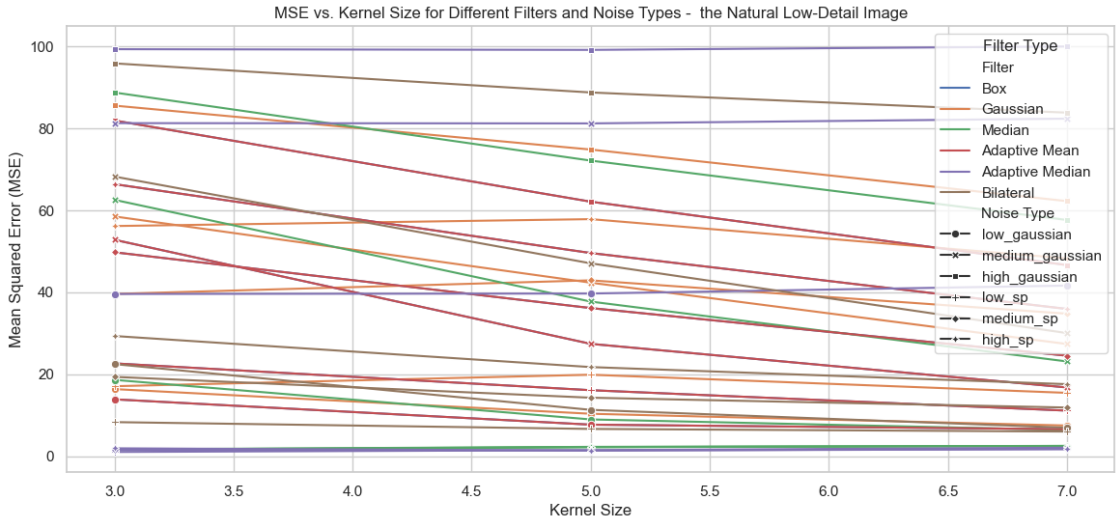

```
Total rows in DataFrame: 108
```

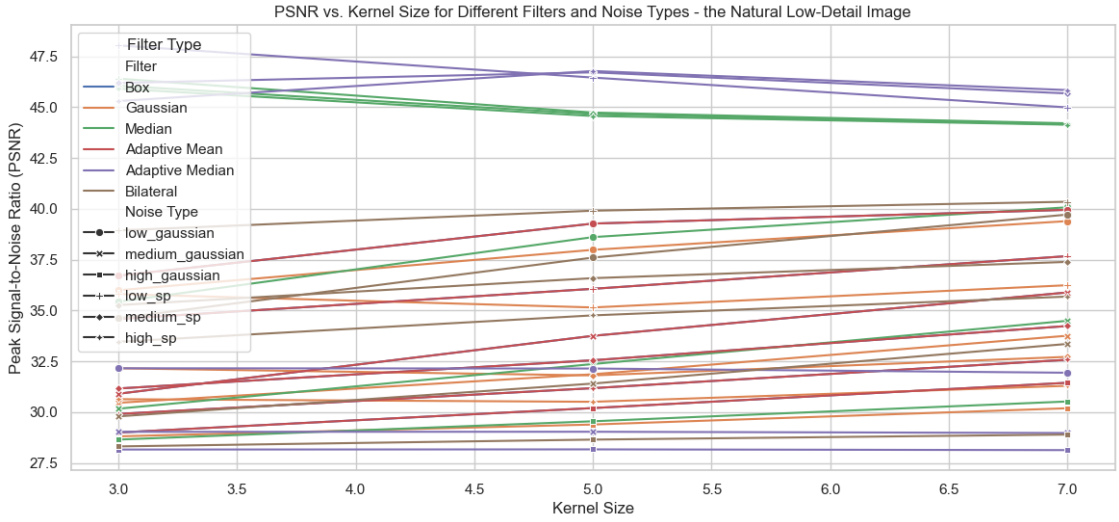
- MSE Analysis: As kernel size increases, the Mean Squared Error tends to decrease for most filters and noise types, indicating an improved noise reduction effect. However, in high-detail images or images with edge-rich content, larger kernel sizes also introduce blurring, which affects the clarity of the edges. This effect is more pronounced in filters like the Box and Gaussian filters. Adaptive filters, such as the Adaptive Mean and Adaptive Median, show a gradual reduction in MSE but manage to retain more structural details, especially in images with high-detail content.
- PSNR Analysis: For Peak Signal-to-Noise Ratio, an increase in kernel size generally results in a reduction in PSNR across most filters, indicating a trade-off between noise removal and detail preservation. High PSNR values are usually associated with low-noise images and well-preserved details, while the decline in PSNR with larger kernels highlights the potential loss in edge clarity. The Bilateral filter stands out with relatively stable PSNR values across kernel sizes, suggesting its effectiveness in balancing noise reduction and edge preservation, especially in high-detail images.
- Comparison of Noise Types: Salt-and-pepper noise tends to be better filtered by Median and Adaptive Median filters, as these filters specifically target outlier pixels. For Gaussian noise, filters like Gaussian and Bilateral perform better, as they are designed to smooth continuous noise patterns without severely impacting edge details. Hence, the choice of filter and kernel size impacts the balance between noise reduction and edge preservation. Adaptive filters and the Bilateral filter offer a good balance for high-detail images, while median-based filters are particularly effective for salt-and-pepper noise.
- The Natural Low-Detail Image:

[29]:
```
# Natural Low-Detail Image
original_image2 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
  ↪archive/images/train/135069.jpg', cv2.IMREAD_GRAYSCALE)
```

```
# Apply filters with different kernel sizes and calculate MSE, PSNR
kernel_sizes = [3, 5, 7]
results2 = []  # Use results2 to store results for the Natural Low-Detail Image

for noise_type, noisy_img in noisy_images2.items():
    for kernel_size in kernel_sizes:
        filtered_images = applyFilters(noisy_img, kernel_size)

        for filter_name, filtered_img in filtered_images.items():
            mse_value = calculate_mse(original_image2, filtered_img)
            psnr_value = calculate_psnr(original_image2, filtered_img)

            # Store results in results2
            results2.append({
                "Noise Type": noise_type,
                "Filter": filter_name,
                "Kernel Size": kernel_size,
                "MSE": mse_value,
                "PSNR": psnr_value
            })

# Convert results2 to a DataFrame for easy analysis
results_df2 = pd.DataFrame(results2)

# Set up the plotting style
sns.set(style="whitegrid")

# Plot MSE for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df2, x="Kernel Size", y="MSE", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("MSE vs. Kernel Size for Different Filters and Noise Types -  the␣
 ↪Natural Low-Detail Image")
plt.ylabel("Mean Squared Error (MSE)")
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()

# Plot PSNR for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df2, x="Kernel Size", y="PSNR", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("PSNR vs. Kernel Size for Different Filters and Noise Types - the␣
 ↪Natural Low-Detail Image")
plt.ylabel("Peak Signal-to-Noise Ratio (PSNR)")
plt.xlabel("Kernel Size")
```

```
plt.legend(loc="best", title="Filter Type")
plt.show()
```



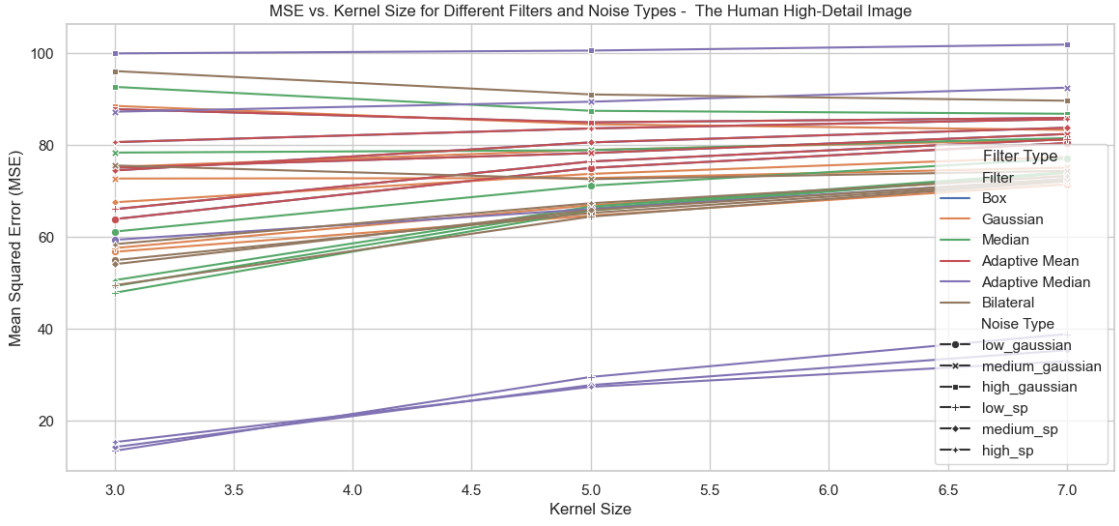


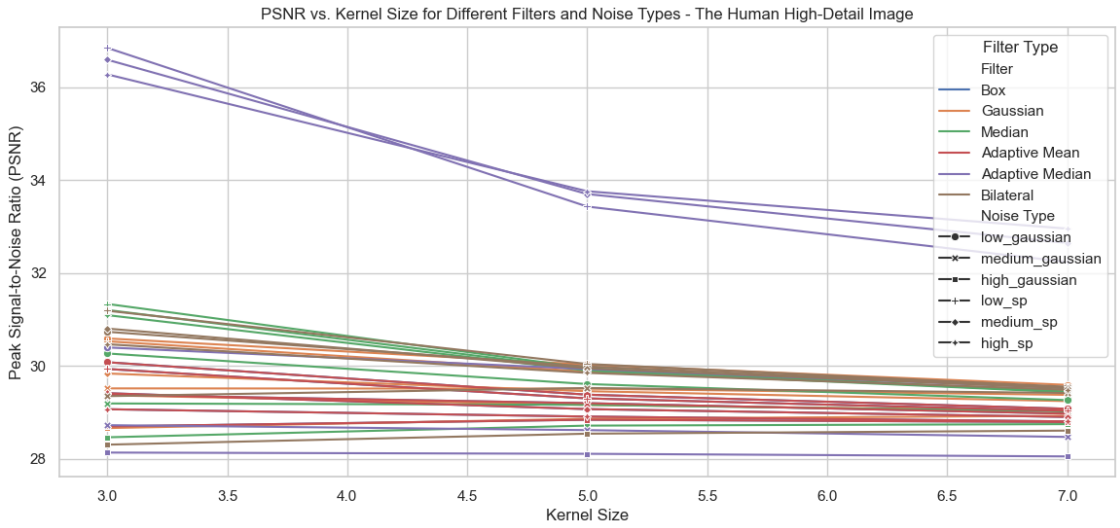
- MSE Analysis: The MSE trends indicate that larger kernel sizes generally reduce noise, as evidenced by the downward-sloping lines for each filter. However, this reduction in MSE comes with a trade-off: while noise is diminished, there is a tendency to blur the image, which can lead to a loss in detail. The Gaussian and Adaptive Mean filters exhibit consistent noise reduction across different kernel sizes, while the Bilateral filter demonstrates a better ability to maintain lower MSE values, especially for salt-and-pepper noise at higher kernel sizes. Adaptive Median, meanwhile, shows minimal MSE values, effectively handling salt-and-pepper noise even with smaller kernel sizes.

- PSNR Analysis: In the PSNR plot, higher values generally correspond to better image quality after filtering. Similar to the MSE trends, larger kernels improve PSNR for most filters, especially the Adaptive Median and Bilateral filters, which maintain the highest PSNR values across kernel sizes. However, with high-intensity noise (especially Gaussian noise), PSNR improvements diminish at higher kernel sizes, indicating the filters struggle to fully retain details. Notably, Bilateral and Adaptive Median filters achieve higher PSNR values, suggesting they strike a better balance between noise removal and edge preservation.
- Comparison of Noise Types: Salt-and-pepper noise is consistently easier to filter out than Gaussian noise, as shown by the lower MSE and higher PSNR values achieved by most filters for this noise type. For Gaussian noise, all filters exhibit higher MSE and lower PSNR, suggesting greater difficulty in removing this type of noise without affecting image details. The Adaptive Median filter is particularly effective for salt-and-pepper noise, even at smaller kernel sizes, while the Gaussian and Bilateral filters show more stable performance across noise intensities for Gaussian noise, albeit at higher kernel sizes.
- The Influenced by Human High-Detail Image:

[30]:
```
# Natural Low-Detail Image
original_image3 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/138032.jpg', cv2.IMREAD_GRAYSCALE)

# Apply filters with different kernel sizes and calculate MSE, PSNR
kernel_sizes = [3, 5, 7]
results3 = []  # Use results2 to store results for the Influenced by Human␣
 ↪High-Detail Image

for noise_type, noisy_img in noisy_images3.items():
    for kernel_size in kernel_sizes:
        filtered_images = applyFilters(noisy_img, kernel_size)

        for filter_name, filtered_img in filtered_images.items():
            mse_value = calculate_mse(original_image3, filtered_img)
            psnr_value = calculate_psnr(original_image3, filtered_img)

            # Store results in results2
            results3.append({
                "Noise Type": noise_type,
                "Filter": filter_name,
                "Kernel Size": kernel_size,
                "MSE": mse_value,
                "PSNR": psnr_value
            })

# Convert results2 to a DataFrame for easy analysis
results_df3 = pd.DataFrame(results3)

# Set up the plotting style
```

```
sns.set(style="whitegrid")

# Plot MSE for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df3, x="Kernel Size", y="MSE", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("MSE vs. Kernel Size for Different Filters and Noise Types -  The␣
 ↪Human High-Detail Image")
plt.ylabel("Mean Squared Error (MSE)")
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()

# Plot PSNR for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df3, x="Kernel Size", y="PSNR", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("PSNR vs. Kernel Size for Different Filters and Noise Types - The␣
 ↪Human High-Detail Image")
plt.ylabel("Peak Signal-to-Noise Ratio (PSNR)")
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()
```



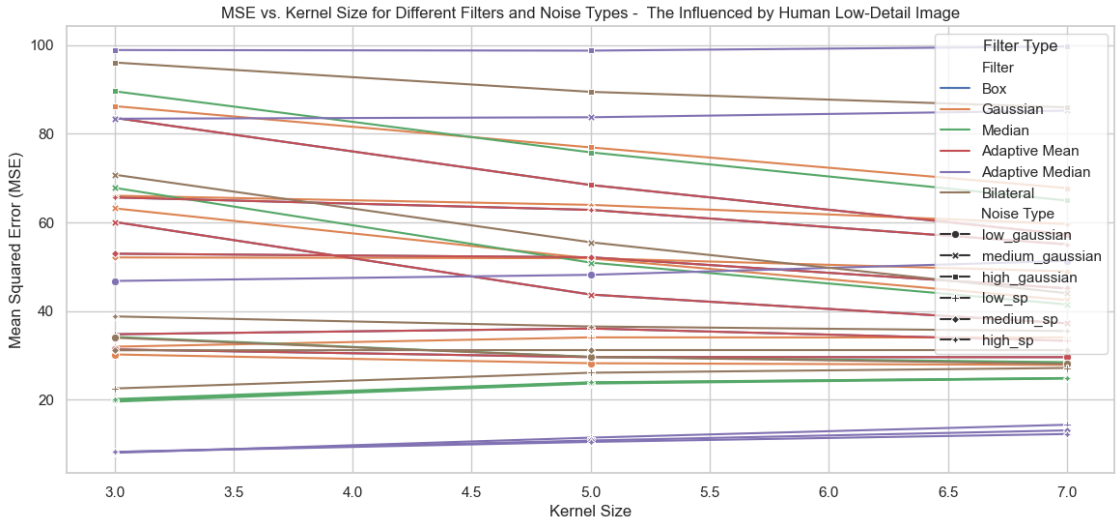


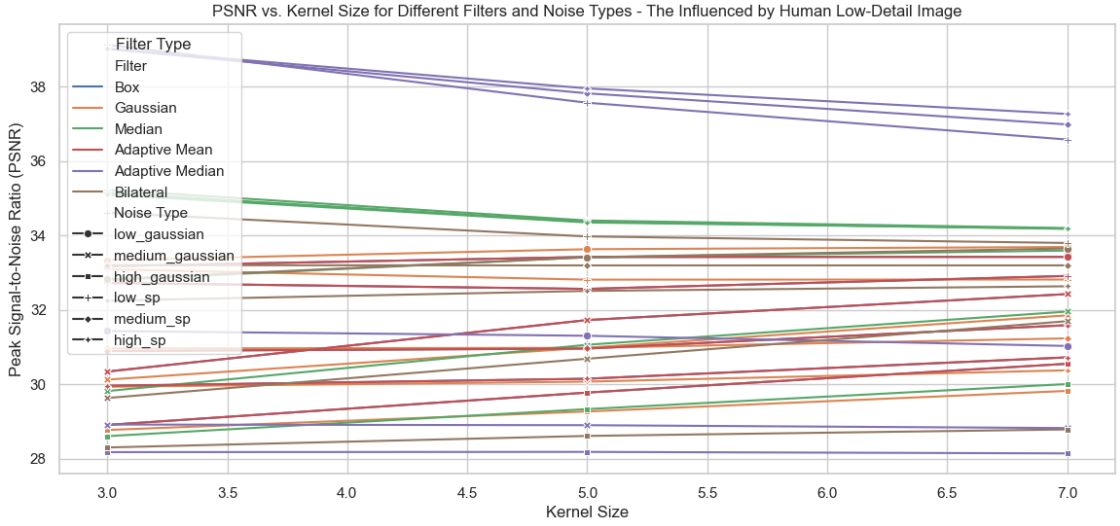
- MSE Analysis: In the MSE plot for the Human High-Detail Image, the MSE generally decreases as the kernel size increases, particularly for the Adaptive Mean and Adaptive Median filters. This trend indicates that larger kernels are more effective at reducing noise in high-detail images, albeit at the cost of slightly more blurring. For salt-and-pepper noise types (low, medium, and high), the Adaptive Median filter consistently achieves lower MSE values, suggesting it is particularly well-suited for this noise type. For Gaussian noise, the Gaussian and Bilateral filters demonstrate a gradual improvement in MSE as kernel size increases, although their effectiveness plateaus at larger sizes.
- PSNR Analysis: In the PSNR plot, higher kernel sizes correlate with lower PSNR for most filters, particularly in the high-noise cases. The Adaptive Median filter maintains a relatively higher PSNR across various kernel sizes, especially for salt-and-pepper noise, which aligns with its ability to preserve edges while filtering out isolated noise specks. For Gaussian noise, however, the Bilateral filter performs better in terms of preserving image quality, maintaining a relatively high PSNR as kernel size increases, though it gradually drops with larger kernel sizes.
- Comparison of Noise Types: Salt-and-pepper noise generally shows better results with filters designed for isolated noise (such as Adaptive Median), while Gaussian noise requires more nuanced filters like the Gaussian and Bilateral filters for effective reduction without compromising image details. The choice of filter and kernel size becomes a trade-off between noise reduction and detail preservation. Salt-and-pepper noise benefits from adaptive filters that can specifically target noisy pixels, whereas Gaussian noise, being more uniformly distributed, benefits from filters like Gaussian and Bilateral that smooth over broader areas while retaining edge information.
- Influenced by Human Low-Detail Image:

[31]:
```
# Natural Low-Detail Image
original_image4 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
↪archive/images/train/161062.jpg', cv2.IMREAD_GRAYSCALE)
```

```
# Apply filters with different kernel sizes and calculate MSE, PSNR
kernel_sizes = [3, 5, 7]
results4 = []  # Use results2 to store results for the Influenced by Human␣
 ↪Low-Detail Image

for noise_type, noisy_img in noisy_images4.items():
    for kernel_size in kernel_sizes:
        filtered_images = applyFilters(noisy_img, kernel_size)

        for filter_name, filtered_img in filtered_images.items():
            mse_value = calculate_mse(original_image4, filtered_img)
            psnr_value = calculate_psnr(original_image4, filtered_img)

            # Store results in results2
            results4.append({
                "Noise Type": noise_type,
                "Filter": filter_name,
                "Kernel Size": kernel_size,
                "MSE": mse_value,
                "PSNR": psnr_value
            })

# Convert results2 to a DataFrame for easy analysis
results_df4 = pd.DataFrame(results4)

# Set up the plotting style
sns.set(style="whitegrid")

# Plot MSE for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df4, x="Kernel Size", y="MSE", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("MSE vs. Kernel Size for Different Filters and Noise Types -  The␣
 ↪Influenced by Human Low-Detail Image")
plt.ylabel("Mean Squared Error (MSE)")
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()

# Plot PSNR for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df4, x="Kernel Size", y="PSNR", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("PSNR vs. Kernel Size for Different Filters and Noise Types - The␣
 ↪Influenced by Human Low-Detail Image")
plt.ylabel("Peak Signal-to-Noise Ratio (PSNR)")
```

```
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()
```



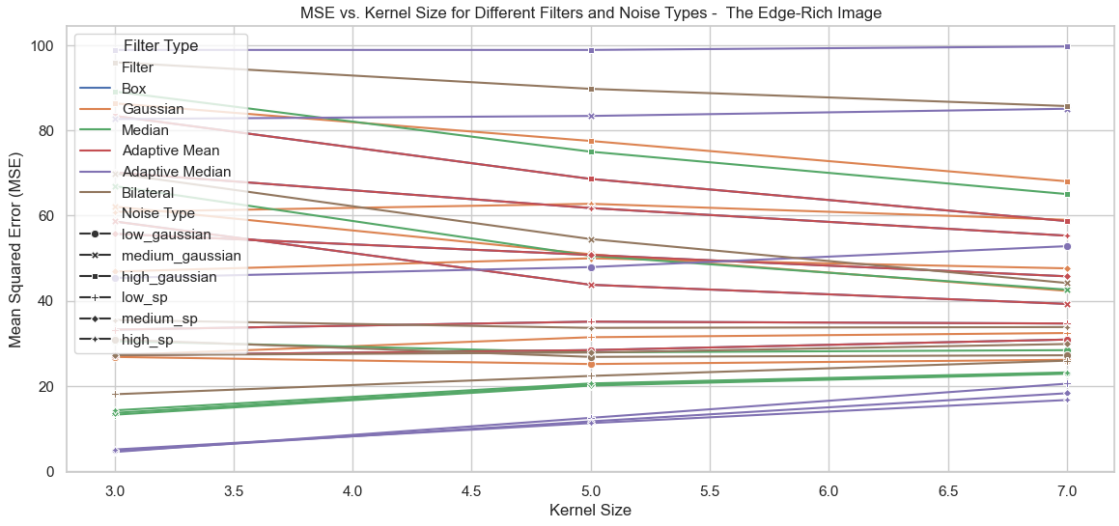


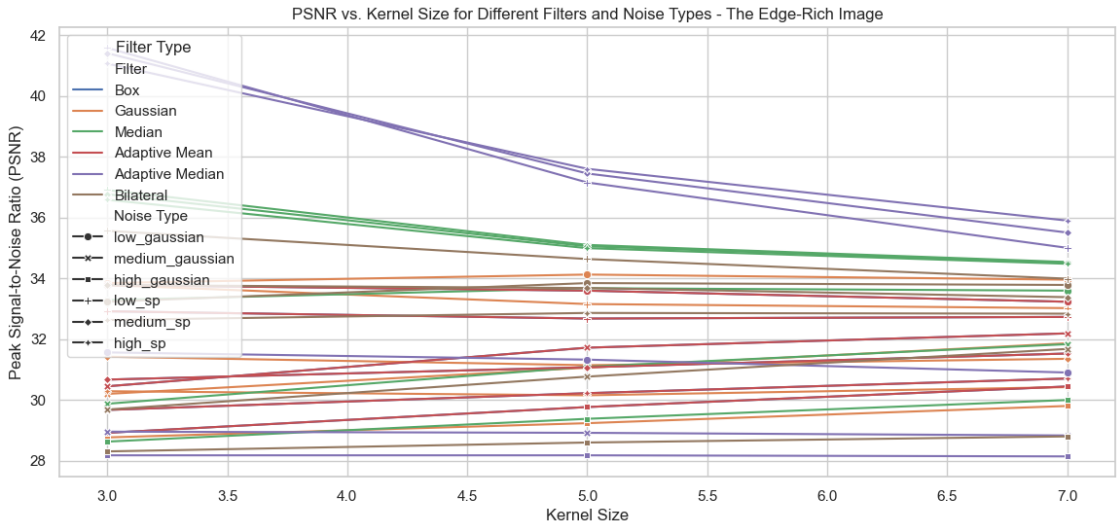
- MSE Analysis: For the Human Low-Detail Image, the MSE values generally decrease as the kernel size increases, indicating better noise reduction with larger kernels. However, this improvement in MSE comes at the cost of image sharpness. The Adaptive Median filter has the lowest MSE across all kernel sizes for salt-and-pepper noise, particularly at higher noise levels, highlighting its effectiveness in handling this noise type. For Gaussian noise, the Bilateral filter shows lower MSE, especially at smaller kernel sizes, due to its edge-preserving properties.

- PSNR Analysis: the Adaptive Median and Bilateral filters perform the best across different noise types. PSNR values tend to decrease as the kernel size increases, reflecting a trade-off between noise reduction and detail preservation. Smaller kernels tend to maintain higher PSNR values, particularly for the Bilateral filter with Gaussian noise, indicating a balance between reducing noise and preserving image details.
- Comparison of Noise Types: For salt-and-pepper noise, the Adaptive Median filter stands out, especially at higher noise levels, as it specifically targets isolated noise pixels without overly blurring the image. Conversely, for Gaussian noise, the Bilateral filter achieves a better balance in both MSE and PSNR, as it smooths noise while preserving edges. Overall, salt-and-pepper noise is easier to reduce without sacrificing detail, whereas Gaussian noise requires a more careful balance, with the Bilateral filter being the preferred choice for maintaining image quality in low-detail images.
- Edge-Rich Image:

[32]:
```
# Natural Low-Detail Image
original_image5 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/183055.jpg', cv2.IMREAD_GRAYSCALE)

# Apply filters with different kernel sizes and calculate MSE, PSNR
kernel_sizes = [3, 5, 7]
results5 = []  # Use results2 to store results for the Edge-Rich Image

for noise_type, noisy_img in noisy_images5.items():
    for kernel_size in kernel_sizes:
        filtered_images = applyFilters(noisy_img, kernel_size)

        for filter_name, filtered_img in filtered_images.items():
            mse_value = calculate_mse(original_image5, filtered_img)
            psnr_value = calculate_psnr(original_image5, filtered_img)

            # Store results in results2
            results5.append({
                "Noise Type": noise_type,
                "Filter": filter_name,
                "Kernel Size": kernel_size,
                "MSE": mse_value,
                "PSNR": psnr_value
            })

# Convert results2 to a DataFrame for easy analysis
results_df5 = pd.DataFrame(results5)

# Set up the plotting style
sns.set(style="whitegrid")

# Plot MSE for each filter and kernel size for different noise types
```

```
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df5, x="Kernel Size", y="MSE", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("MSE vs. Kernel Size for Different Filters and Noise Types -  The␣
 ↪Edge-Rich Image")
plt.ylabel("Mean Squared Error (MSE)")
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()

# Plot PSNR for each filter and kernel size for different noise types
plt.figure(figsize=(14, 6))
sns.lineplot(data=results_df5, x="Kernel Size", y="PSNR", hue="Filter",␣
 ↪style="Noise Type", markers=True, dashes=False)
plt.title("PSNR vs. Kernel Size for Different Filters and Noise Types - The␣
 ↪Edge-Rich Image")
plt.ylabel("Peak Signal-to-Noise Ratio (PSNR)")
plt.xlabel("Kernel Size")
plt.legend(loc="best", title="Filter Type")
plt.show()
```





- MSE Analysis: the Adaptive Median and Median filters consistently achieve lower MSE values, especially with salt-and-pepper noise. This indicates their effectiveness in reducing isolated noise while preserving image edges. As kernel size increases, MSE generally decreases for these filters, highlighting enhanced noise reduction at the cost of slight blurring. However, Bilateral and Gaussian filters show a moderate reduction in MSE for Gaussian noise, which reflects their smoothing capabilities but also introduces slight blurring to edge details as the kernel size grows.
- PSNR Analysis: the Adaptive Median and Bilateral filters achieve higher values, particularly with lower kernel sizes, demonstrating their balance between noise reduction and edge preservation. For salt-and-pepper noise, Adaptive Median maintains higher PSNR values, emphasizing its efficiency for this type of noise. For Gaussian noise, Bilateral filter maintains relatively stable PSNR, though it gradually declines as kernel size increases, indicating a trade-off between noise reduction and edge sharpness.
- Comparison of Noise Types: Salt-and-pepper noise is more effectively handled by Adaptive Median and Median filters, which both maintain low MSE and high PSNR across different kernel sizes. In contrast, Gaussian noise is better reduced by Bilateral and Gaussian filters, though increasing kernel size slightly blurs fine details, affecting PSNR. As kernel size increases, Adaptive Mean filter shows a decline in PSNR for both noise types, suggesting it may overly smooth edges in an attempt to reduce noise.

19 b. Displaying just the best result:

- The Natural High-Detail Image:

[33]:
```
# Find the best result: lowest MSE and highest PSNR for each noise type and␣
↪filter
best_mse_results = results_df.loc[results_df.groupby(['Noise Type',␣
↪'Filter'])['MSE'].idxmin()]
```

```
best_mse_results = best_mse_results.reset_index(drop=True) #reset index for␣
 ↪better display

print("Best Results (Lowest MSE and Highest PSNR for each Noise Type and␣
 ↪Filter):")
display(best_mse_results)
```

Best Results (Lowest MSE and Highest PSNR for each Noise Type and Filter):

	Noise Type	Filter	Kernel Size	MSE	PSNR
0	high_gaussian	Adaptive Mean	7	75.807786	29.333665
1	high_gaussian	Adaptive Median	5	96.412776	28.289458
2	high_gaussian	Bilateral	7	78.746880	29.168470
3	high_gaussian	Box	7	75.807786	29.333665
4	high_gaussian	Gaussian	7	75.479634	29.352506
5	high_gaussian	Median	7	67.676518	29.826424
6	high_sp	Adaptive Mean	3	60.449926	30.316846
7	high_sp	Adaptive Median	3	4.311520	41.784500
8	high_sp	Bilateral	3	29.625352	33.414168
9	high_sp	Box	3	60.449926	30.316846
10	high_sp	Gaussian	3	58.246864	30.478078
11	high_sp	Median	3	12.766757	37.069998
12	low_gaussian	Adaptive Mean	3	25.359836	34.089339
13	low_gaussian	Adaptive Median	3	45.177713	31.581561
14	low_gaussian	Bilateral	5	24.810247	34.184493
15	low_gaussian	Box	3	25.359836	34.089339
16	low_gaussian	Gaussian	5	23.982124	34.331927
17	low_gaussian	Median	5	26.858019	33.840064
18	low_sp	Adaptive Mean	3	27.626926	33.717478
19	low_sp	Adaptive Median	3	3.209604	43.066290
20	low_sp	Bilateral	3	14.797158	36.429020
21	low_sp	Box	3	27.626926	33.717478
22	low_sp	Gaussian	3	24.734205	34.197824
23	low_sp	Median	3	10.949029	37.737048
24	medium_gaussian	Adaptive Mean	7	42.770332	31.819377
25	medium_gaussian	Adaptive Median	3	81.045855	29.043496
26	medium_gaussian	Bilateral	7	42.530476	31.843801
27	medium_gaussian	Box	7	42.770332	31.819377
28	medium_gaussian	Gaussian	7	41.428281	31.957834
29	medium_gaussian	Median	7	43.555825	31.740341
30	medium_sp	Adaptive Mean	3	46.380619	31.467438
31	medium_sp	Adaptive Median	3	3.705494	42.442342
32	medium_sp	Bilateral	3	22.474213	34.613959
33	medium_sp	Box	3	46.380619	31.467438
34	medium_sp	Gaussian	3	44.277686	31.668954
35	medium_sp	Median	3	11.818602	37.405142

As we can see through the table, for the high Gaussian noise, the Adaptive Mean and Bilateral filters with larger kernel sizes performed better, balancing noise reduction and detail preservation. In the case of salt-and-pepper noise (low and high), the Adaptive Median filter with smaller kernel sizes provided optimal performance, highlighting its effectiveness in removing isolated noise. Low Gaussian noise showed better results with Gaussian and Bilateral filters, indicating these filters' proficiency in handling distributed noise while retaining image details. In short, filters that adapt to noise structure, like the Adaptive Median and Bilateral filters, tend to perform best across varied noise intensities and types.

- The Natural Low-Detail Image:

[34]:
```
# Find the best result: lowest MSE and highest PSNR for each noise type and␣
 ↪filter
best_mse_results2 = results_df2.loc[results_df2.groupby(['Noise Type',␣
 ↪'Filter'])['MSE'].idxmin()]

best_mse_results2 = best_mse_results2.reset_index(drop=True) #reset index for␣
 ↪better display

print("Best Results (Lowest MSE and Highest PSNR for each Noise Type and␣
 ↪Filter):")
display(best_mse_results2)
```

```
Best Results (Lowest MSE and Highest PSNR for each Noise Type and Filter):
```

	Noise Type	Filter	Kernel Size	MSE	PSNR
0	high_gaussian	Adaptive Mean	7	46.631453	31.444014
1	high_gaussian	Adaptive Median	5	99.148989	28.167921
2	high_gaussian	Bilateral	7	83.833013	28.896653
3	high_gaussian	Box	7	46.631453	31.444014
4	high_gaussian	Gaussian	7	62.239331	30.190154
5	high_gaussian	Median	7	57.640870	30.523498
6	high_sp	Adaptive Mean	7	35.924210	32.576931
7	high_sp	Adaptive Median	5	1.366293	46.775365
8	high_sp	Bilateral	7	17.570327	35.683005
9	high_sp	Box	7	35.924210	32.576931
10	high_sp	Gaussian	7	48.259513	31.294974
11	high_sp	Median	3	1.668623	45.907222
12	low_gaussian	Adaptive Mean	7	6.580851	39.947983
13	low_gaussian	Adaptive Median	3	39.582127	32.155812
14	low_gaussian	Bilateral	7	6.942280	39.715782
15	low_gaussian	Box	7	6.580851	39.947983
16	low_gaussian	Gaussian	7	7.476266	39.393956
17	low_gaussian	Median	7	6.393890	40.073152
18	low_sp	Adaptive Mean	7	11.114377	37.671952
19	low_sp	Adaptive Median	3	1.022364	48.034749
20	low_sp	Bilateral	7	6.005706	40.345163
21	low_sp	Box	7	11.114377	37.671952
22	low_sp	Gaussian	7	15.449116	36.241767

23	low_sp	Median	3	1.490133	46.398554
24	medium_gaussian	Adaptive Mean	7	16.769095	35.885707
25	medium_gaussian	Adaptive Median	5	81.209668	29.034726
26	medium_gaussian	Bilateral	7	30.032869	33.354835
27	medium_gaussian	Box	7	16.769095	35.885707
28	medium_gaussian	Gaussian	7	27.369849	33.758080
29	medium_gaussian	Median	7	23.116262	34.491627
30	medium_sp	Adaptive Mean	7	24.519563	34.235676
31	medium_sp	Adaptive Median	5	1.387102	46.709718
32	medium_sp	Bilateral	7	11.865798	37.387834
33	medium_sp	Box	7	24.519563	34.235676
34	medium_sp	Gaussian	7	34.771446	32.718576
35	medium_sp	Median	3	1.614536	46.050326

Hence, the Bilateral and Adaptive Median filters show the best adaptability across noise types, with Bilateral particularly excelling in Gaussian noise scenarios and Adaptive Median in salt-and-pepper noise cases. Increasing the kernel size generally enhances noise removal at the expense of detail, but the Bilateral filter minimizes this trade-off effectively.

- The Influenced by Human High-Detail Image:

[35]:
```
# Find the best result: lowest MSE and highest PSNR for each noise type and␣
 ↪filter
best_mse_results3 = results_df3.loc[results_df3.groupby(['Noise Type',␣
 ↪'Filter'])['MSE'].idxmin()]

best_mse_results3 = best_mse_results3.reset_index(drop=True) #reset index for␣
 ↪better display

print("Best Results (Lowest MSE and Highest PSNR for each Noise Type and␣
 ↪Filter):")
display(best_mse_results3)
```

Best Results (Lowest MSE and Highest PSNR for each Noise Type and Filter):

	Noise Type	Filter	Kernel Size	MSE	PSNR
0	high_gaussian	Adaptive Mean	5	84.986904	28.837284
1	high_gaussian	Adaptive Median	3	100.006243	28.130532
2	high_gaussian	Bilateral	7	89.697780	28.602987
3	high_gaussian	Box	5	84.986904	28.837284
4	high_gaussian	Gaussian	7	83.344156	28.922052
5	high_gaussian	Median	7	86.858621	28.742674
6	high_sp	Adaptive Mean	3	80.679775	29.063157
7	high_sp	Adaptive Median	3	15.341662	36.272079
8	high_sp	Bilateral	3	58.462426	30.462035
9	high_sp	Box	3	80.679775	29.063157
10	high_sp	Gaussian	3	75.279804	29.364019
11	high_sp	Median	3	50.605838	31.088797
12	low_gaussian	Adaptive Mean	3	63.910687	30.075069

13	low_gaussian	Adaptive Median	3	59.397685	30.393108
14	low_gaussian	Bilateral	3	54.955894	30.730661
15	low_gaussian	Box	3	63.910687	30.075069
16	low_gaussian	Gaussian	3	56.796562	30.587583
17	low_gaussian	Median	3	61.197965	30.263434
18	low_sp	Adaptive Mean	3	66.073367	29.930539
19	low_sp	Adaptive Median	3	13.452244	36.842856
20	low_sp	Bilateral	3	49.508332	31.184021
21	low_sp	Box	3	66.073367	29.930539
22	low_sp	Gaussian	3	57.603817	30.526291
23	low_sp	Median	3	47.873272	31.329873
24	medium_gaussian	Adaptive Mean	3	75.123782	29.373029
25	medium_gaussian	Adaptive Median	3	87.307576	28.720284
26	medium_gaussian	Bilateral	5	72.622736	29.520078
27	medium_gaussian	Box	3	75.123782	29.373029
28	medium_gaussian	Gaussian	3	72.752981	29.512296
29	medium_gaussian	Median	3	78.399576	29.187666
30	medium_sp	Adaptive Mean	3	74.512561	29.408509
31	medium_sp	Adaptive Median	3	14.266922	36.587501
32	medium_sp	Bilateral	3	54.092266	30.799452
33	medium_sp	Box	3	74.512561	29.408509
34	medium_sp	Gaussian	3	67.582742	29.832446
35	medium_sp	Median	3	49.325173	31.200117

Thus, the Adaptive Median filter is highly effective for salt-and-pepper noise, while the Bilateral and Adaptive Mean filters provide a balanced approach for both noise types, making them versatile choices depending on the specific noise intensity and type.

- The Influenced by Human Low-Detail Image:

[36]:
```
# Find the best result: lowest MSE and highest PSNR for each noise type and␣
 ↪filter
best_mse_results4 = results_df4.loc[results_df4.groupby(['Noise Type',␣
 ↪'Filter'])['MSE'].idxmin()]

best_mse_results4 = best_mse_results4.reset_index(drop=True) #reset index for␣
 ↪better display

print("Best Results (Lowest MSE and Highest PSNR for each Noise Type and␣
 ↪Filter):")
display(best_mse_results4)
```

```
Best Results (Lowest MSE and Highest PSNR for each Noise Type and Filter):
```

	Noise Type	Filter	Kernel Size	MSE	PSNR
0	high_gaussian	Adaptive Mean	7	57.250944	30.552977
1	high_gaussian	Adaptive Median	5	98.750753	28.185399
2	high_gaussian	Bilateral	7	85.941244	28.788787
3	high_gaussian	Box	7	57.250944	30.552977

4	high_gaussian	Gaussian	7	67.722288	29.823487
5	high_gaussian	Median	7	64.909508	30.007720
6	high_sp	Adaptive Mean	7	55.028504	30.724927
7	high_sp	Adaptive Median	3	8.158697	39.014595
8	high_sp	Bilateral	7	35.454492	32.634091
9	high_sp	Box	7	55.028504	30.724927
10	high_sp	Gaussian	7	59.610560	30.377572
11	high_sp	Median	3	20.104365	35.097900
12	low_gaussian	Adaptive Mean	7	29.561376	33.423557
13	low_gaussian	Adaptive Median	3	46.763402	31.431743
14	low_gaussian	Bilateral	7	28.054864	33.650722
15	low_gaussian	Box	7	29.561376	33.423557
16	low_gaussian	Gaussian	7	27.810027	33.688789
17	low_gaussian	Median	7	28.426254	33.593607
18	low_sp	Adaptive Mean	7	33.257272	32.911937
19	low_sp	Adaptive Median	3	7.964670	39.119126
20	low_sp	Bilateral	3	22.494362	34.610067
21	low_sp	Box	7	33.257272	32.911937
22	low_sp	Gaussian	3	31.924152	33.089610
23	low_sp	Median	3	19.521830	35.225598
24	medium_gaussian	Adaptive Mean	7	37.204513	32.424847
25	medium_gaussian	Adaptive Median	3	83.379298	28.920221
26	medium_gaussian	Bilateral	7	43.995972	31.696674
27	medium_gaussian	Box	7	37.204513	32.424847
28	medium_gaussian	Gaussian	7	42.438676	31.853185
29	medium_gaussian	Median	7	41.462478	31.954251
30	medium_sp	Adaptive Mean	7	45.091599	31.589847
31	medium_sp	Adaptive Median	3	8.168652	39.009300
32	medium_sp	Bilateral	3	31.113419	33.201326
33	medium_sp	Box	7	45.091599	31.589847
34	medium_sp	Gaussian	7	48.928569	31.235178
35	medium_sp	Median	3	19.899988	35.142276

Increasing the kernel size generally improves noise reduction (lower MSE) but at the cost of detail preservation, which affects PSNR. This effect is especially pronounced in filters like the Adaptive Mean and Median, which tend to perform better with larger kernels for higher noise levels. Salt-and-pepper noise is best handled by the Adaptive Median filter, while Gaussian noise favors the Bilateral and Adaptive Mean filters.

- The Edge-Rich Image:

[37]:
```
# Find the best result: lowest MSE and highest PSNR for each noise type and␣
 ↪filter
best_mse_results5 = results_df5.loc[results_df5.groupby(['Noise Type',␣
 ↪'Filter'])['MSE'].idxmin()]

best_mse_results5 = best_mse_results5.reset_index(drop=True) #reset index for␣
 ↪better display
```

```
print("Best Results (Lowest MSE and Highest PSNR for each Noise Type and␣
 ↪Filter):")
display(best_mse_results5)
```

```
Best Results (Lowest MSE and Highest PSNR for each Noise Type and Filter):
```

	Noise Type	Filter	Kernel Size	MSE	PSNR
0	high_gaussian	Adaptive Mean	7	58.731550	30.442089
1	high_gaussian	Adaptive Median	5	98.910519	28.178379
2	high_gaussian	Bilateral	7	85.732307	28.799358
3	high_gaussian	Box	7	58.731550	30.442089
4	high_gaussian	Gaussian	7	68.068821	29.801321
5	high_gaussian	Median	7	65.071658	29.996885
6	high_sp	Adaptive Mean	7	55.302530	30.703354
7	high_sp	Adaptive Median	3	5.095569	41.058876
8	high_sp	Bilateral	5	33.659905	32.859675
9	high_sp	Box	7	55.302530	30.703354
10	high_sp	Gaussian	7	59.029948	30.420080
11	high_sp	Median	3	14.287038	36.581382
12	low_gaussian	Adaptive Mean	3	27.254370	33.776442
13	low_gaussian	Adaptive Median	3	45.379039	31.562251
14	low_gaussian	Bilateral	5	26.845914	33.842022
15	low_gaussian	Box	3	27.254370	33.776442
16	low_gaussian	Gaussian	5	25.169040	34.122137
17	low_gaussian	Median	5	27.909560	33.673274
18	low_sp	Adaptive Mean	3	33.239914	32.914205
19	low_sp	Adaptive Median	3	4.523261	41.576287
20	low_sp	Bilateral	3	18.077623	35.559390
21	low_sp	Box	3	33.239914	32.914205
22	low_sp	Gaussian	3	27.248360	33.777400
23	low_sp	Median	3	13.270296	36.901997
24	medium_gaussian	Adaptive Mean	7	39.286604	32.188359
25	medium_gaussian	Adaptive Median	3	82.688409	28.956357
26	medium_gaussian	Bilateral	7	44.198665	31.676712
27	medium_gaussian	Box	7	39.286604	32.188359
28	medium_gaussian	Gaussian	7	42.342860	31.863002
29	medium_gaussian	Median	7	42.634782	31.833163
30	medium_sp	Adaptive Mean	7	45.761141	31.525835
31	medium_sp	Adaptive Median	3	4.707787	41.402636
32	medium_sp	Bilateral	3	27.250730	33.777022
33	medium_sp	Box	7	45.761141	31.525835
34	medium_sp	Gaussian	3	46.880428	31.420888
35	medium_sp	Median	3	13.699834	36.763651

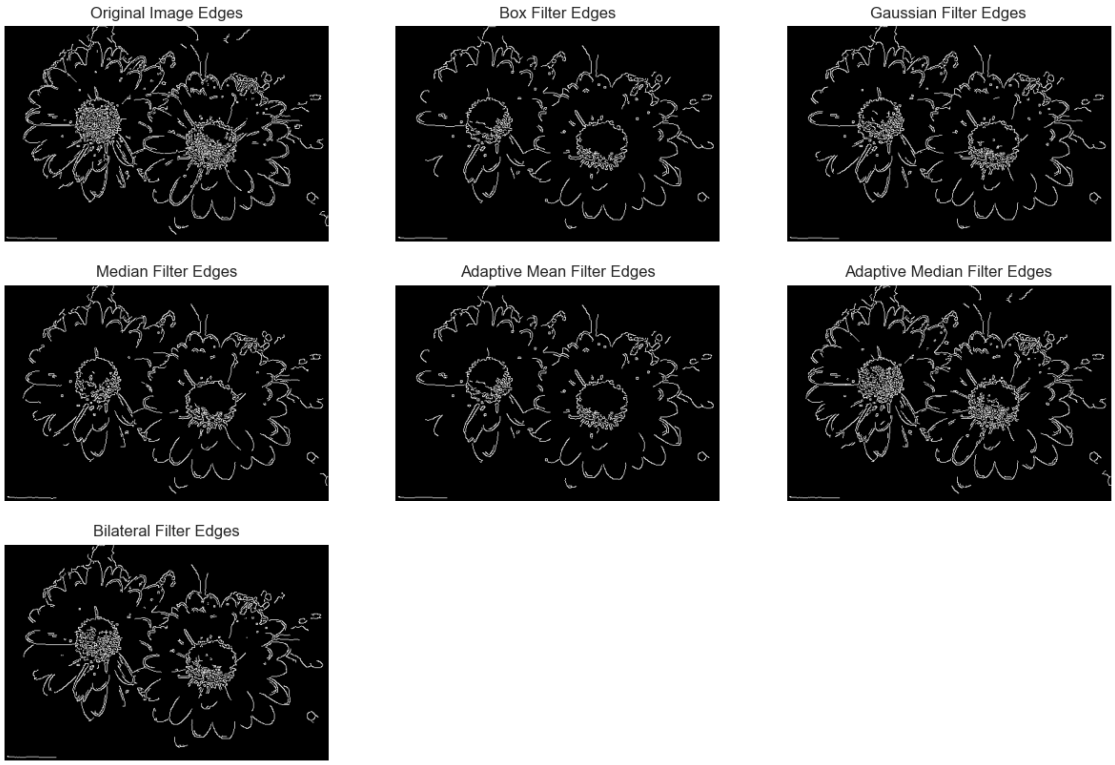
For salt-and-pepper noise, Adaptive Median and Median are preferable, while Bilateral and Adaptive Mean filters are better suited for Gaussian noise, especially at medium noise levels. The kernel size also influences the balance between noise reduction and edge preservation, with larger kernels favoring noise reduction at the expense of finer details.

20 Step 3.2. Edge Preservation:

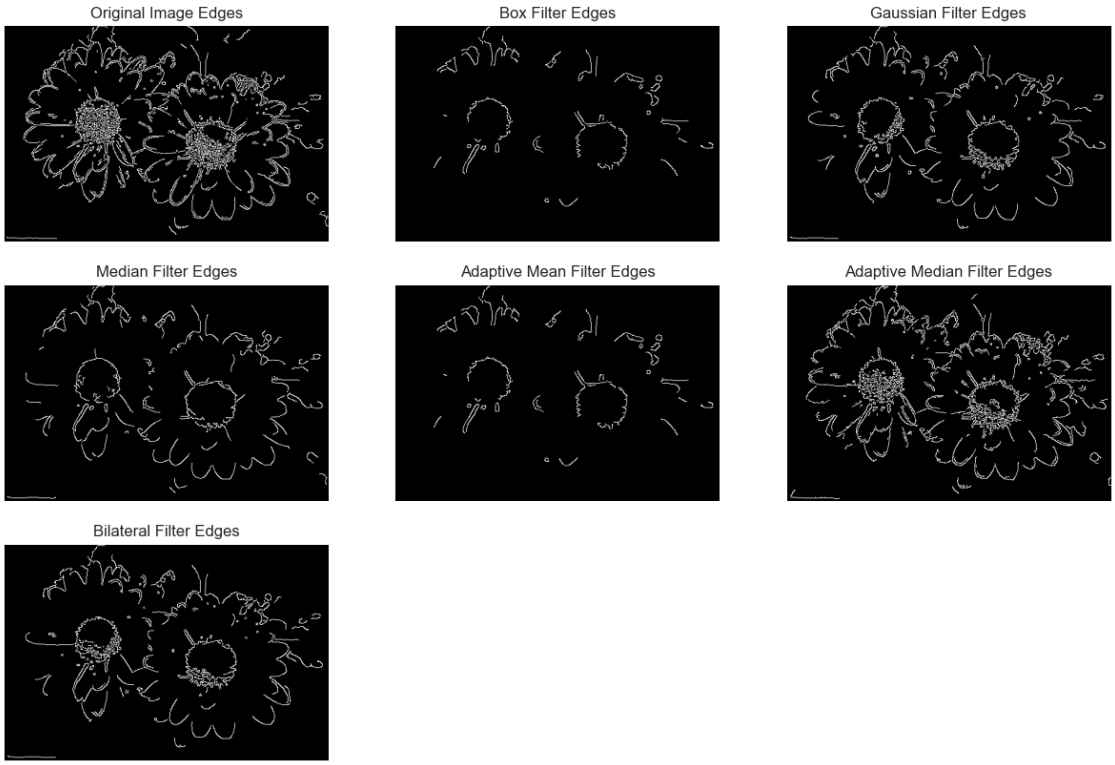
21 - The Natural High-Detail Image:

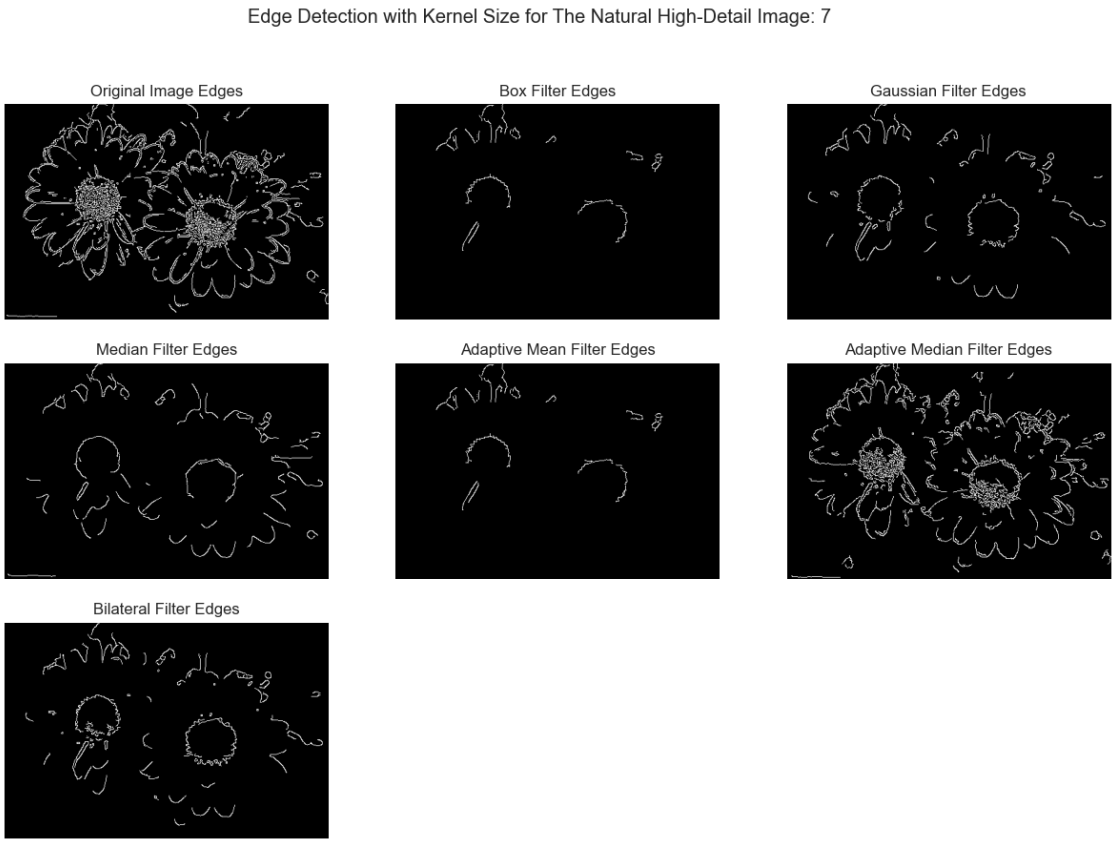
[38]:
```
# Apply Canny edge detection
def apply_canny(image, low_threshold=100, high_threshold=200):
    return cv2.Canny(image, low_threshold, high_threshold)

# Load the original image
original_image = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/124084.jpg', cv2.IMREAD_GRAYSCALE)

# Define kernel sizes
kernel_sizes = [3, 5, 7]

# Apply filters and perform Canny edge detection
for kernel_size in kernel_sizes:
    filtered_images = applyFilters(original_image, kernel_size)

    # Display and save results for edge preservation analysis
    plt.figure(figsize=(15, 10))
    plt.suptitle(f"Edge Detection with Kernel Size for The Natural High-Detail␣
 ↪Image: {kernel_size}")

    # Apply Canny edge detection to the original image for comparison
    original_edges = apply_canny(original_image)
    plt.subplot(3, 3, 1)
    plt.imshow(original_edges, cmap='gray')
    plt.title("Original Image Edges")
    plt.axis('off')

    for i, (filter_name, filtered_img) in enumerate(filtered_images.items(),␣
 ↪start=2):
        # Apply Canny edge detection on the filtered image
        edges = apply_canny(filtered_img)

        # Display the edge-detected image
        plt.subplot(3, 3, i)
        plt.imshow(edges, cmap='gray')
        plt.title(f"{filter_name} Filter Edges")
        plt.axis('off')

    plt.show()
```

Edge Detection with Kernel Size for The Natural High-Detail Image: 3



Edge Detection with Kernel Size for The Natural High-Detail Image: 5





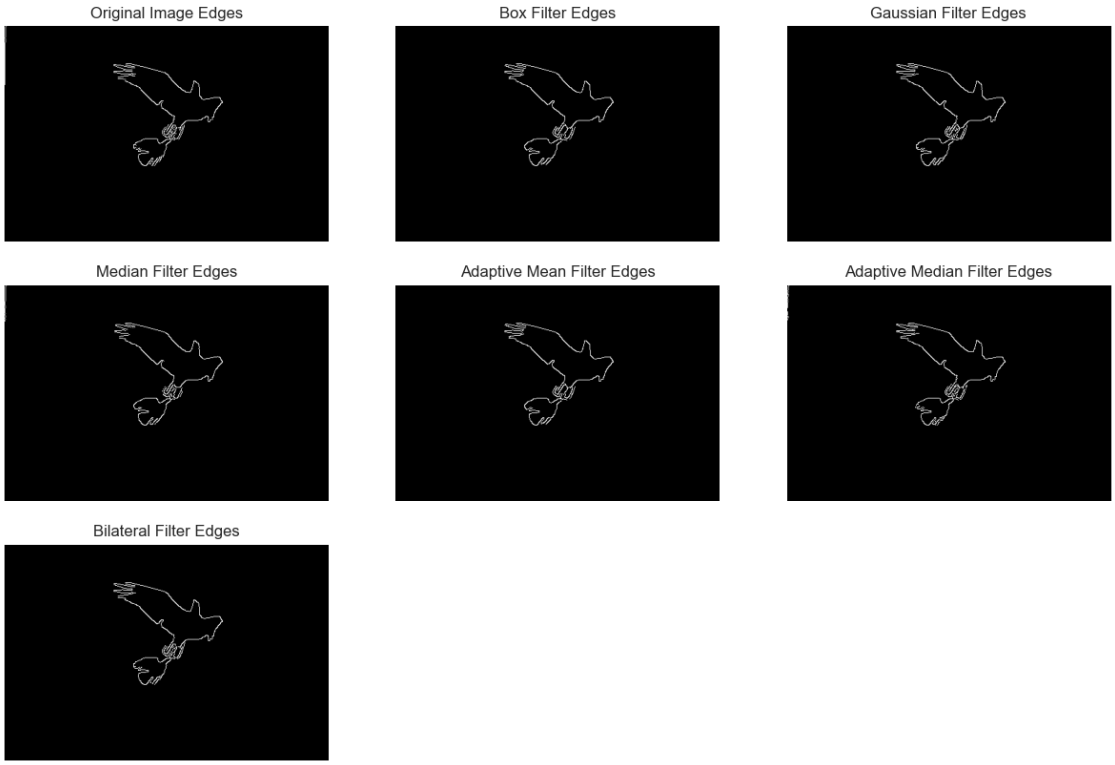
- Box Filter: As the kernel size increases, the Box filter increasingly blurs the edges, resulting in significant loss of detail. This is evident by the fading and broadening of edges, especially at kernel size 7, where most fine details are lost.
- Gaussian Filter: The Gaussian filter maintains some edge structure better than the Box filter, but edge sharpness decreases with larger kernel sizes. At kernel size 7, edge details are smoothed out, although the main outlines remain somewhat intact.
- Median Filter: The Median filter is effective in preserving edges even at larger kernel sizes. It retains strong edges and removes noise effectively without overly blurring the edges, which makes it suitable for images with distinct edges.
- Adaptive Mean Filter: Similar to the Box filter, the Adaptive Mean filter blurs edges progressively with larger kernel sizes, leading to significant edge loss at kernel size 7. The filter is not ideal for applications where fine edge details need to be preserved.
- Adaptive Median Filter: This filter is one of the best in preserving edges across all kernel sizes. It retains most of the fine details, especially at kernel size 3, where edges are clear and well-defined. Even with larger kernels, it maintains edge integrity better than other filters.
- Bilateral Filter: The Bilateral filter preserves edges well while reducing noise, especially for smaller kernel sizes. At kernel size 3, it maintains edge details similar to the Adaptive Median filter. However, with larger kernels, there is a slight loss of finer edges, although it

still performs better than simpler filters like Box and Gaussian.

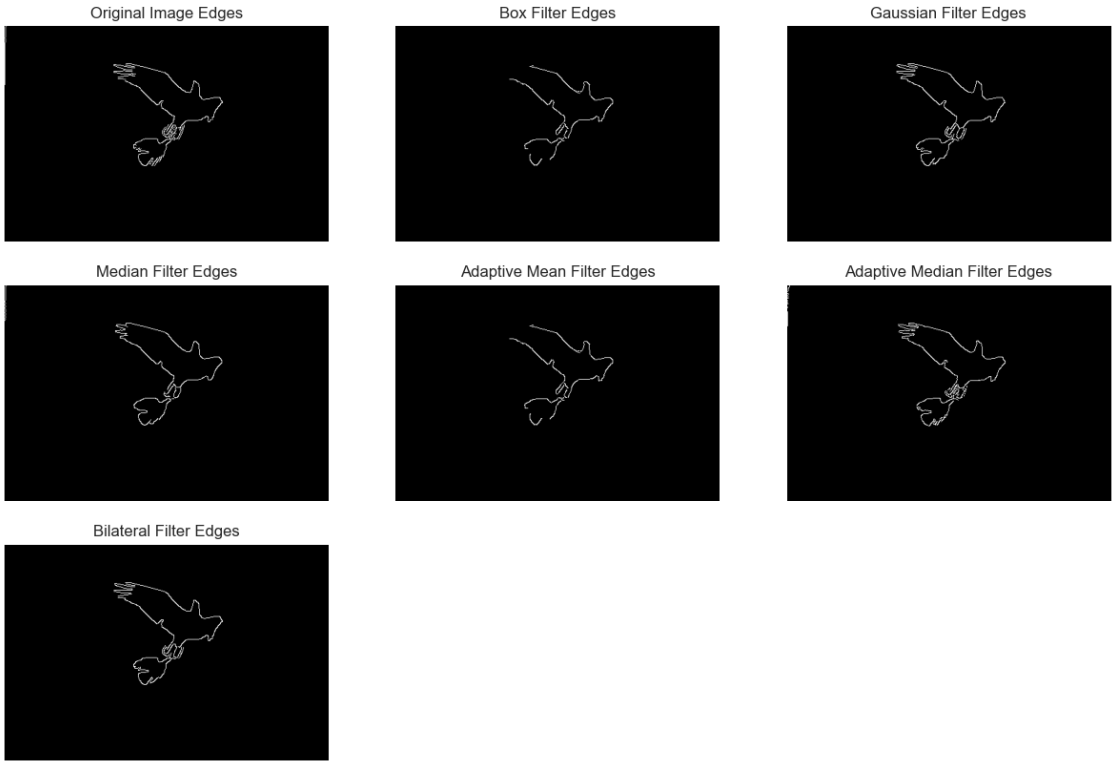
22 - The Natural Low-Detail Image:

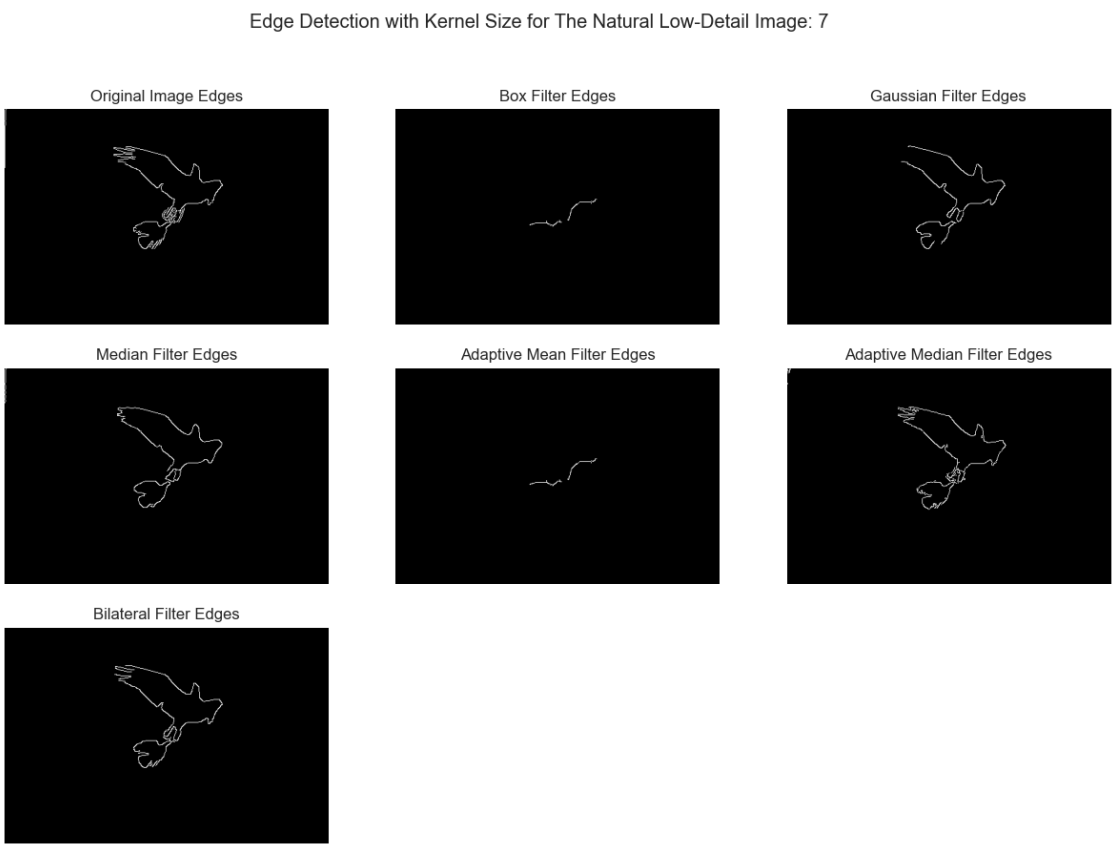
[41]:
```
# Apply Canny edge detection
def apply_canny(image, low_threshold=100, high_threshold=200):
    return cv2.Canny(image, low_threshold, high_threshold)

# Load the original image
original_image2 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/135069.jpg', cv2.IMREAD_GRAYSCALE)

# Define kernel sizes
kernel_sizes = [3, 5, 7]

# Apply filters and perform Canny edge detection
for kernel_size in kernel_sizes:
    filtered_images = applyFilters(original_image2, kernel_size)

    # Display and save results for edge preservation analysis
    plt.figure(figsize=(15, 10))
    plt.suptitle(f"Edge Detection with Kernel Size for The Natural Low-Detail␣
 ↪Image: {kernel_size}")

    # Apply Canny edge detection to the original image for comparison
    original_edges = apply_canny(original_image)
    plt.subplot(3, 3, 1)
    plt.imshow(original_edges, cmap='gray')
    plt.title("Original Image Edges")
    plt.axis('off')

    for i, (filter_name, filtered_img) in enumerate(filtered_images.items(),␣
 ↪start=2):
        # Apply Canny edge detection on the filtered image
        edges = apply_canny(filtered_img)

        # Display the edge-detected image
        plt.subplot(3, 3, i)
        plt.imshow(edges, cmap='gray')
        plt.title(f"{filter_name} Filter Edges")
        plt.axis('off')

    plt.show()
```

Edge Detection with Kernel Size for The Natural Low-Detail Image: 3



Edge Detection with Kernel Size for The Natural Low-Detail Image: 5





- Box Filter: The Box Filter performs poorly in preserving edges, especially as the kernel size increases. At kernel size 7, much of the edge detail is lost, resulting in significant blurring. This is due to the averaging nature of the filter, which smooths out details without prioritizing edge preservation.
- Gaussian Filter: The Gaussian Filter maintains edge details slightly better than the Box Filter, but still experiences noticeable blurring as kernel size increases. The Gaussian filter's weighting structure helps to some extent, but with larger kernels, edge sharpness deteriorates.
- Median Filter: The Median Filter effectively retains edges, particularly at smaller kernel sizes. It handles isolated noise well, maintaining the main edges up to kernel size 5. However, at kernel size 7, edge clarity diminishes somewhat, though it still outperforms simple smoothing filters like the Box Filter.
- Adaptive Mean Filter: The Adaptive Mean Filter produces results similar to the Box Filter, as it also averages pixel values. It struggles with edge preservation and shows significant blurring at larger kernel sizes.
- Adaptive Median Filter: The Adaptive Median Filter is effective at preserving edges even at larger kernel sizes. It adapts to variations in local intensity, which allows it to maintain edge sharpness better than the standard Median Filter and the simpler smoothing filters.
- Bilateral Filter: The Bilateral Filter provides the best balance between noise reduction and

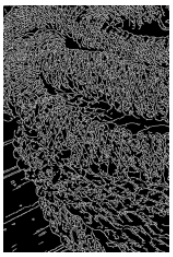
edge preservation, especially at larger kernel sizes. Its use of spatial and intensity differences helps to maintain edges while smoothing flat regions, resulting in clearer edges compared to other filters at kernel size 7.

23 - The Influenced by Human High-Detail Image:

[42]:
```
# Apply Canny edge detection
def apply_canny(image, low_threshold=100, high_threshold=200):
    return cv2.Canny(image, low_threshold, high_threshold)

# Load the original image
original_image3 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/138032.jpg', cv2.IMREAD_GRAYSCALE)

# Define kernel sizes
kernel_sizes = [3, 5, 7]

# Apply filters and perform Canny edge detection
for kernel_size in kernel_sizes:
    filtered_images = applyFilters(original_image3, kernel_size)

    # Display and save results for edge preservation analysis
    plt.figure(figsize=(15, 10))
    plt.suptitle(f"Edge Detection with Kernel Size for The Influenced by Human␣
 ↪High-Detail Image: {kernel_size}")

    # Apply Canny edge detection to the original image for comparison
    original_edges = apply_canny(original_image3)
    plt.subplot(3, 3, 1)
    plt.imshow(original_edges, cmap='gray')
    plt.title("Original Image Edges")
    plt.axis('off')

    for i, (filter_name, filtered_img) in enumerate(filtered_images.items(),␣
 ↪start=2):
        # Apply Canny edge detection on the filtered image
        edges = apply_canny(filtered_img)

        # Display the edge-detected image
        plt.subplot(3, 3, i)
        plt.imshow(edges, cmap='gray')
        plt.title(f"{filter_name} Filter Edges")
        plt.axis('off')

    plt.show()
```

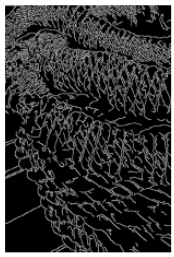
Edge Detection with Kernel Size for The Influenced by Human High-Detail Image: 3

Original Image Edges

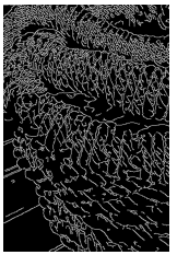


Box Filter Edges

Gaussian Filter Edges

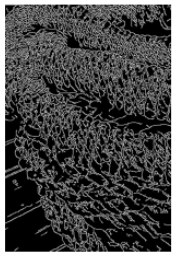


Median Filter Edges

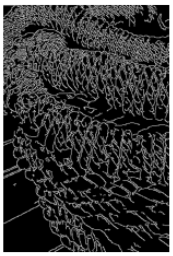


Adaptive Mean Filter Edges

Adaptive Median Filter Edges

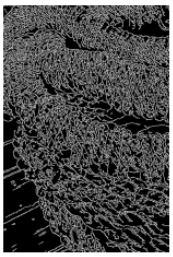


Bilateral Filter Edges



Edge Detection with Kernel Size for The Influenced by Human High-Detail Image: 5

Original Image Edges



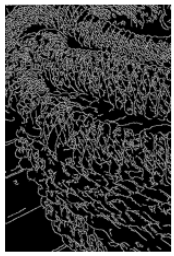
Box Filter Edges

Gaussian Filter Edges

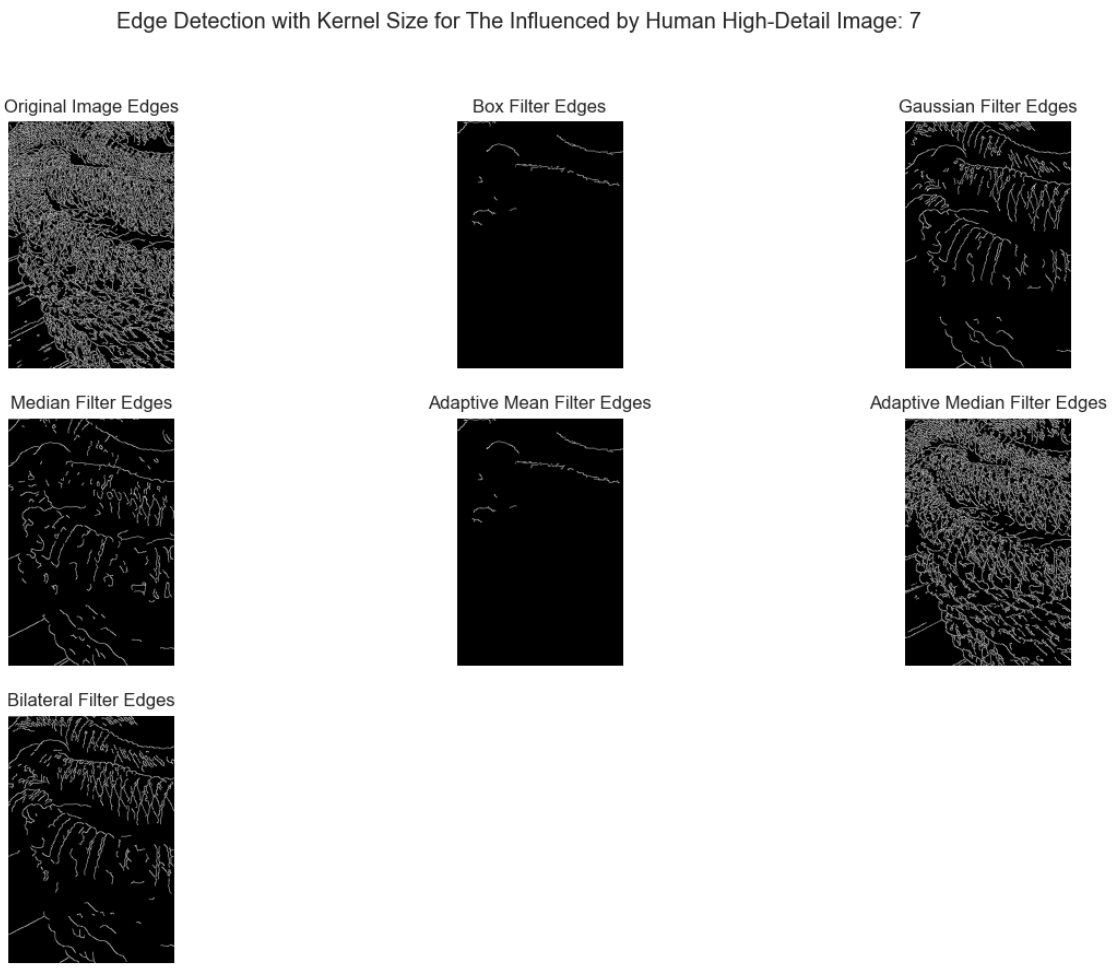
Median Filter Edges

Adaptive Mean Filter Edges

Adaptive Median Filter Edges



Bilateral Filter Edges

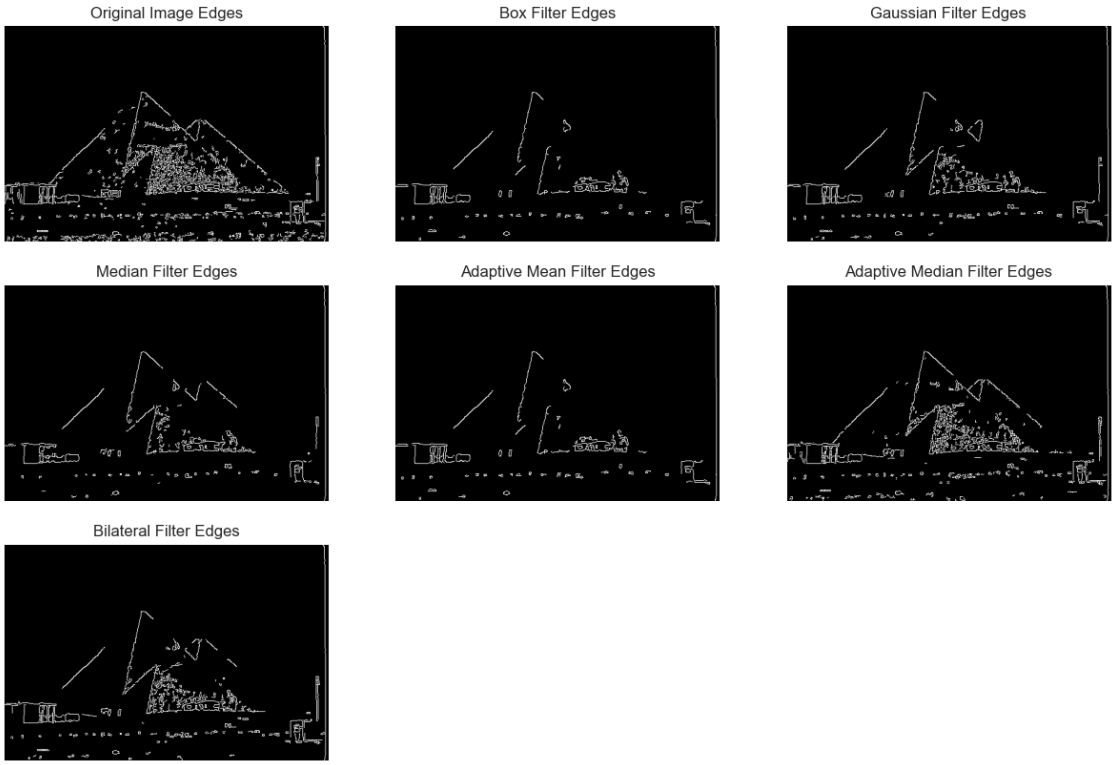


- Box Filter: Blurs edges significantly as kernel size increases. At smaller sizes, some edges are retained, but finer details are mostly lost at larger kernels.
- Gaussian Filter: Smoothly blurs edges and retains slightly more detail than the box filter at smaller sizes, but loses fine details at larger kernels as it averages surrounding pixels.
- Median Filter: Good at preserving strong edges and reducing noise, especially with small kernels. However, it loses finer details as kernel size increases.
- Adaptive Mean Filter: Similar to the box filter, it shows considerable edge blurring as kernel size grows, though it adapts slightly better to local variations.
- Adaptive Median Filter: Retains edges well, even with larger kernels. It preserves textures and edges better than most filters, thanks to its local median approach.
- Bilateral Filter: Excellent at edge preservation across all kernel sizes. It smooths noise while keeping edges intact, making it ideal for balancing noise reduction and edge retention.

24 - The Influenced by Human Low-Detail Image:

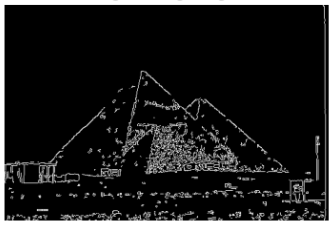
[45]:
```
# Apply Canny edge detection
def apply_canny(image, low_threshold=100, high_threshold=200):
    return cv2.Canny(image, low_threshold, high_threshold)

# Load the original image
original_image4 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/161062.jpg', cv2.IMREAD_GRAYSCALE)

# Define kernel sizes
kernel_sizes = [3, 5, 7]

# Apply filters and perform Canny edge detection
for kernel_size in kernel_sizes:
    filtered_images = applyFilters(original_image4, kernel_size)

    # Display and save results for edge preservation analysis
    plt.figure(figsize=(15, 10))
    plt.suptitle(f"Edge Detection with Kernel Size for The Influenced by Human␣
 ↪Low-Detail Image: {kernel_size}")

    # Apply Canny edge detection to the original image for comparison
    original_edges = apply_canny(original_image4)
    plt.subplot(3, 3, 1)
    plt.imshow(original_edges, cmap='gray')
    plt.title("Original Image Edges")
    plt.axis('off')

    for i, (filter_name, filtered_img) in enumerate(filtered_images.items(),␣
 ↪start=2):
        # Apply Canny edge detection on the filtered image
        edges = apply_canny(filtered_img)

        # Display the edge-detected image
        plt.subplot(3, 3, i)
        plt.imshow(edges, cmap='gray')
        plt.title(f"{filter_name} Filter Edges")
        plt.axis('off')

    plt.show()
```

Edge Detection with Kernel Size for The Influenced by Human Low-Detail Image: 3

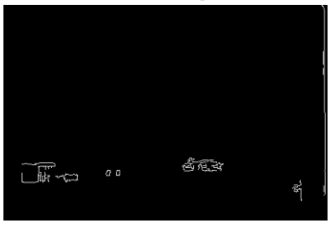


Edge Detection with Kernel Size for The Influenced by Human Low-Detail Image: 5

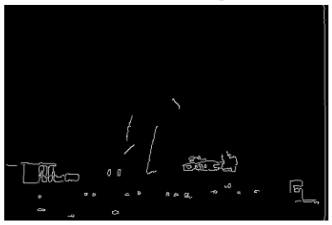
Original Image Edges



Box Filter Edges

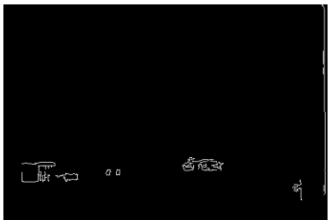


Gaussian Filter Edges

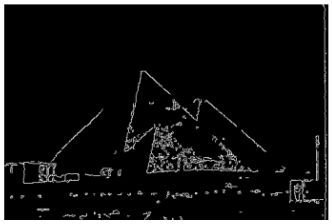


Median Filter Edges

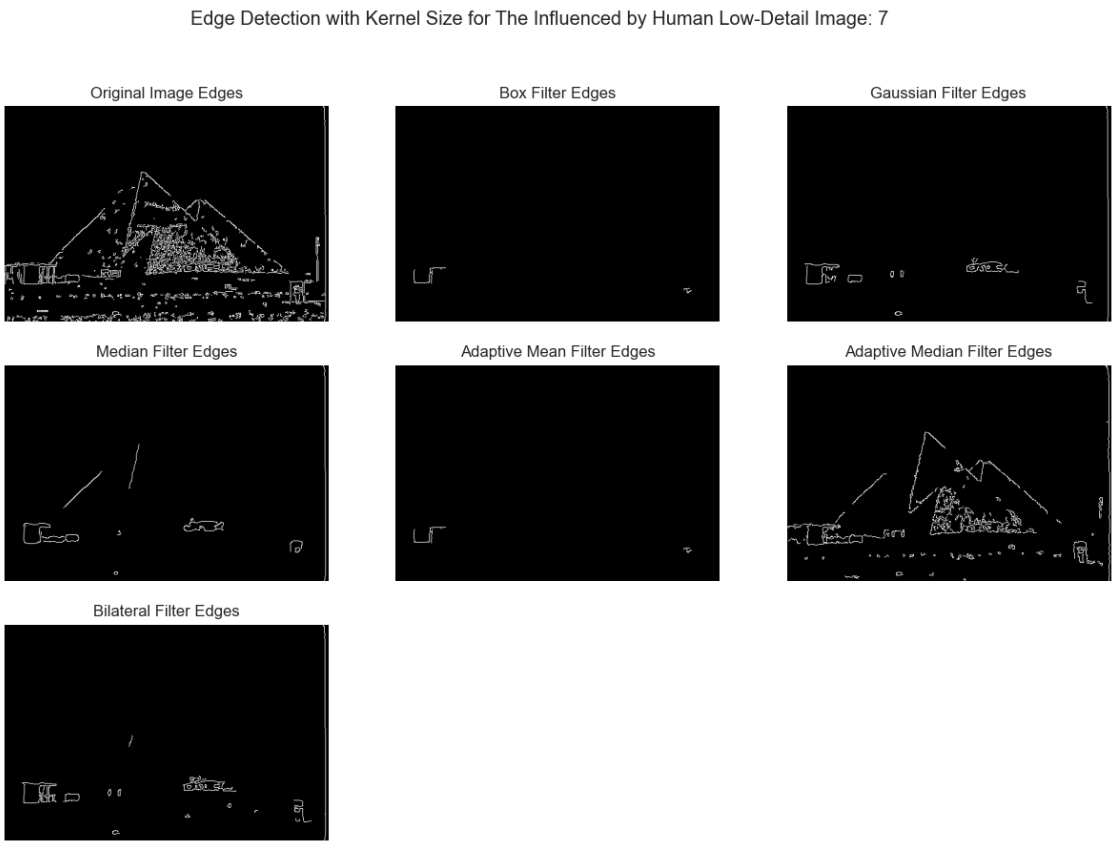
Adaptive Mean Filter Edges



Adaptive Median Filter Edges



Bilateral Filter Edges



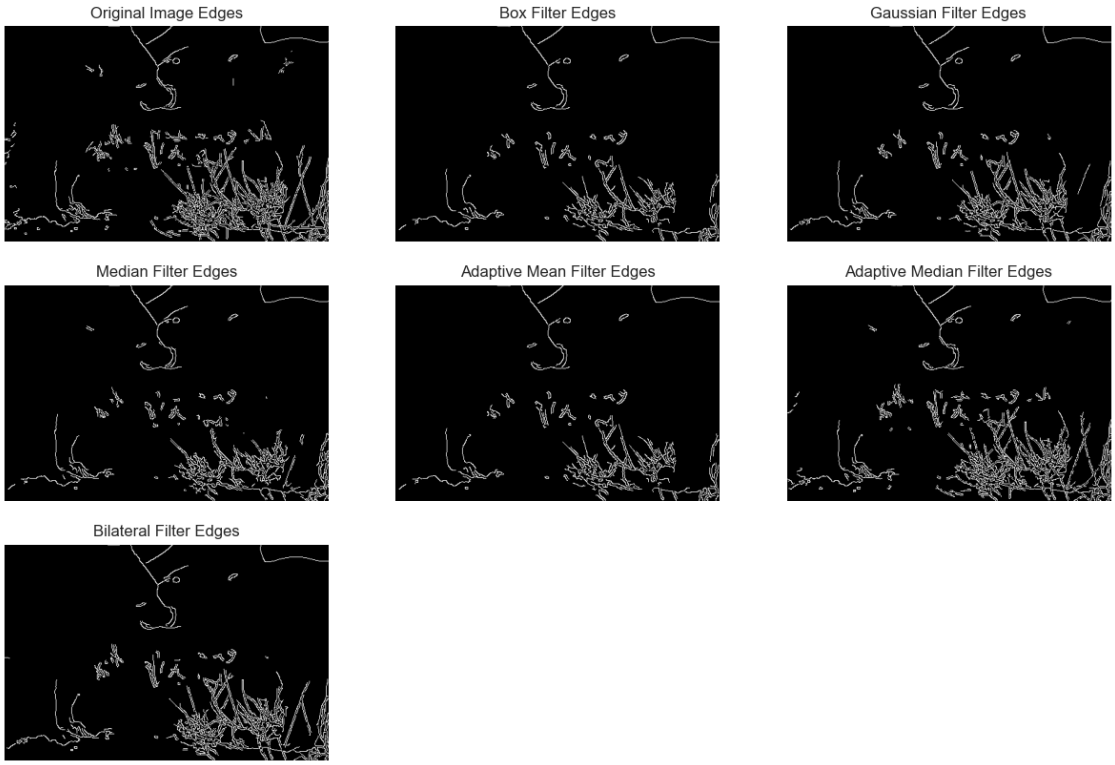
- Box Filter: The box filter tends to blur edges, particularly as the kernel size increases. By kernel size 7, most fine edges are lost, and only larger, less detailed structures remain. This filter is not effective for edge preservation.
- Gaussian Filter: The Gaussian filter also blurs edges with increasing kernel size but preserves more detail than the box filter at smaller kernels. By kernel size 7, edges are significantly softened, losing smaller details, making it less suitable for high-detail edge preservation.
- Median Filter: The median filter maintains edges better than the simple smoothing filters (box and Gaussian), especially at smaller kernel sizes. However, it also loses finer edges as the kernel size increases, though it remains somewhat more resilient than the Gaussian filter.
- Adaptive Mean Filter: The adaptive mean filter performs similarly to the box filter but shows slightly better edge preservation at smaller kernel sizes. By kernel size 7, however, it blurs edges almost as much as the box filter.
- Adaptive Median Filter: The adaptive median filter performs well at maintaining edge structures across kernel sizes, even at size 7. It preserves more fine details than other filters, making it effective for edge preservation in low-detail images.
- Bilateral Filter: The bilateral filter offers good edge preservation at smaller kernel sizes by keeping edges sharp while smoothing other areas. Even at larger kernel sizes, it retains more edge information than the simple smoothing filters, making it a strong choice for edge

preservation.

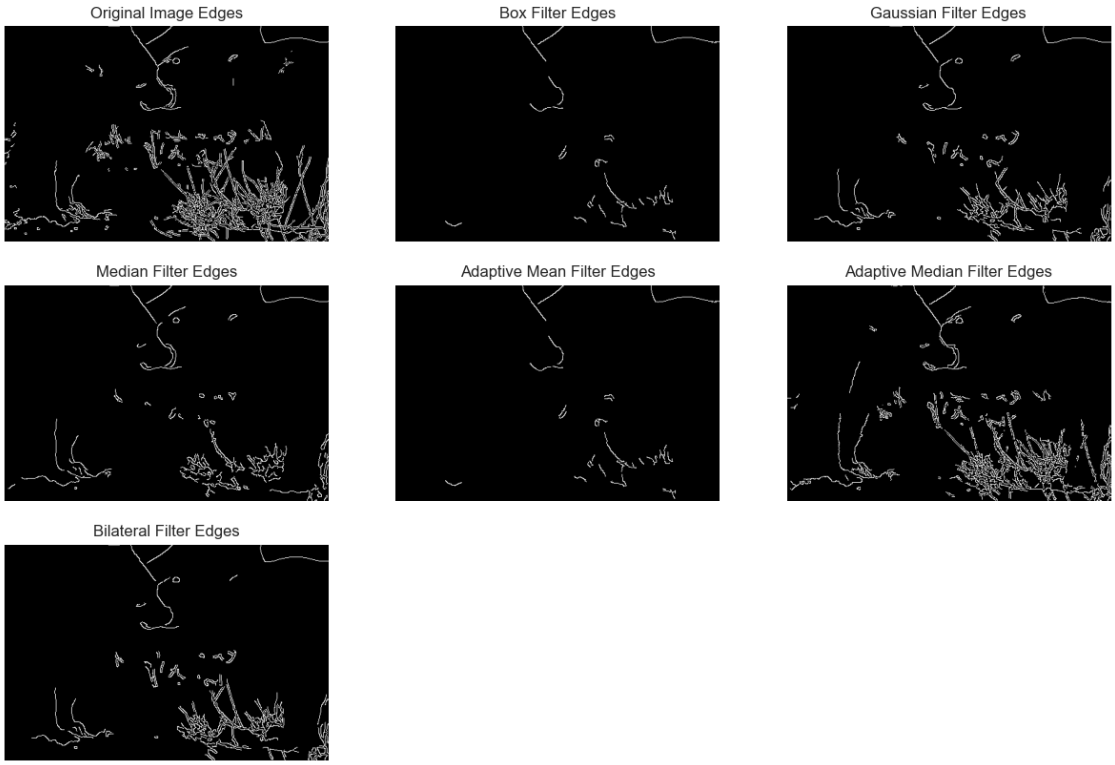
25 - The Edge-Rich Image:

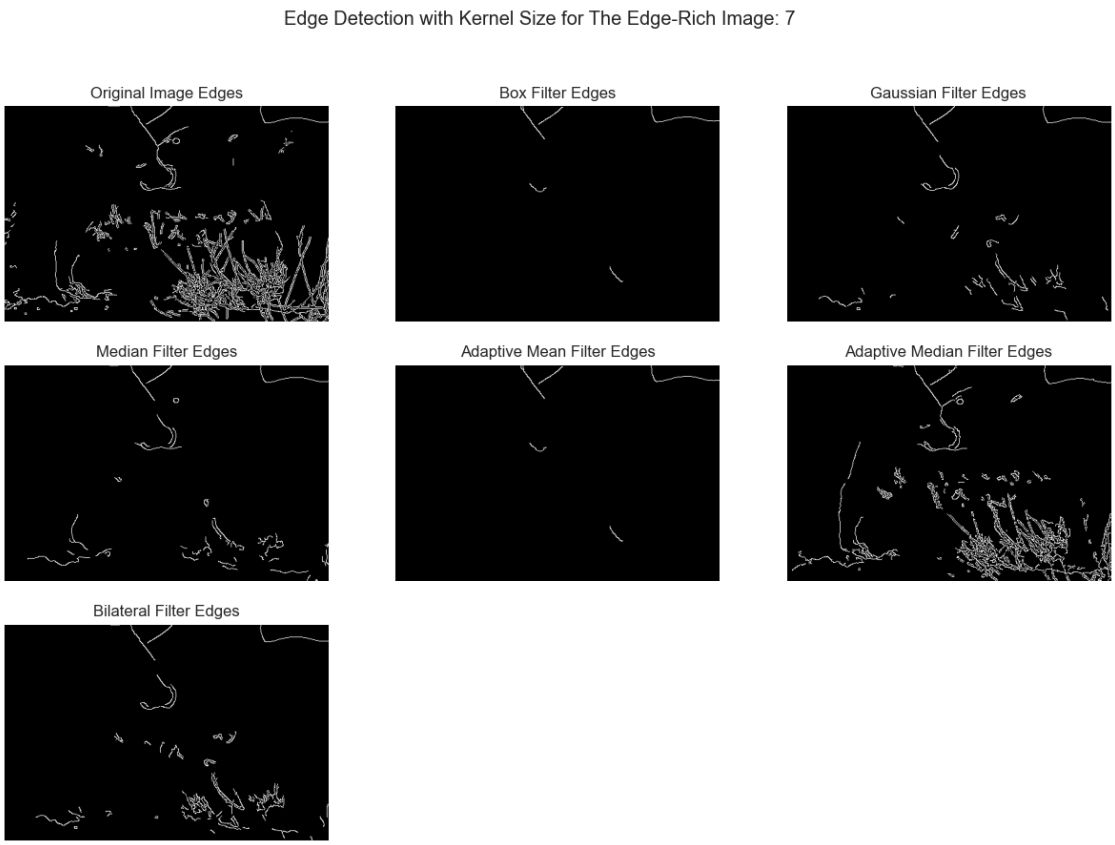
[46]:
```
# Apply Canny edge detection
def apply_canny(image, low_threshold=100, high_threshold=200):
    return cv2.Canny(image, low_threshold, high_threshold)

# Load the original image
original_image5 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/
 ↪archive/images/train/183055.jpg', cv2.IMREAD_GRAYSCALE)

# Define kernel sizes
kernel_sizes = [3, 5, 7]

# Apply filters and perform Canny edge detection
for kernel_size in kernel_sizes:
    filtered_images = applyFilters(original_image5, kernel_size)

    # Display and save results for edge preservation analysis
    plt.figure(figsize=(15, 10))
    plt.suptitle(f"Edge Detection with Kernel Size for The Edge-Rich Image:␣
 ↪{kernel_size}")

    # Apply Canny edge detection to the original image for comparison
    original_edges = apply_canny(original_image5)
    plt.subplot(3, 3, 1)
    plt.imshow(original_edges, cmap='gray')
    plt.title("Original Image Edges")
    plt.axis('off')

    for i, (filter_name, filtered_img) in enumerate(filtered_images.items(),␣
 ↪start=2):
        # Apply Canny edge detection on the filtered image
        edges = apply_canny(filtered_img)

        # Display the edge-detected image
        plt.subplot(3, 3, i)
        plt.imshow(edges, cmap='gray')
        plt.title(f"{filter_name} Filter Edges")
        plt.axis('off')

    plt.show()
```

Edge Detection with Kernel Size for The Edge-Rich Image: 3



Edge Detection with Kernel Size for The Edge-Rich Image: 5





- Box Filter: The box filter performs basic averaging, which causes significant loss of edge details as kernel size increases. It's not effective at preserving fine edges, making it less suitable for edge-rich images.
- Gaussian Filter: The Gaussian filter smooths out noise while maintaining some edge details. However, it still causes notable edge blurring at larger kernel sizes, resulting in loss of fine details similar to the box filter.
- Median Filter: The median filter is better at preserving edges than box and Gaussian filters, especially at smaller kernel sizes. It removes noise without excessively blurring the edges, but larger kernels reduce sharpness over fine details.
- Adaptive Mean Filter: The adaptive mean filter behaves like the box filter in terms of edge blurring, as it applies a simple average within local regions. It doesn't maintain edges well, especially at larger kernel sizes, and shows limited adaptability.
- Adaptive Median Filter: The adaptive median filter effectively preserves edges, even at larger kernel sizes. It adapts to local variations, which makes it robust against noise while retaining edge details better than the standard median filter.
- Bilateral Filter: The bilateral filter excels at preserving edges while reducing noise, especially in complex or edge-rich images. It retains sharpness better than other filters as it combines spatial and intensity information, making it highly effective across kernel sizes.

26 Step 3.3. Computational Time

[49]:
```
# Define a function to apply filters and measure the timing
def apply_filters_with_timing(image, kernel_size):
    timings = {}

    # Box filter
    start_time = time.time()
    cv2.blur(image, (kernel_size, kernel_size))
    timings["Box Filter"] = time.time() - start_time

    # Gaussian filter
    start_time = time.time()
    cv2.GaussianBlur(image, (kernel_size, kernel_size), sigmaX=0)
    timings["Gaussian Filter"] = time.time() - start_time

    # Median filter
    start_time = time.time()
    cv2.medianBlur(image, kernel_size)
    timings["Median Filter"] = time.time() - start_time

    # Adaptive mean filter
    start_time = time.time()
    adaptive_mean_filter(image, kernel_size)
    timings["Adaptive Mean Filter"] = time.time() - start_time

    # Adaptive median filter
    start_time = time.time()
    adaptive_median_filter(image, kernel_size)
    timings["Adaptive Median Filter"] = time.time() - start_time

    # Bilateral filter
    start_time = time.time()
    cv2.bilateralFilter(image, kernel_size, sigmaColor=75, sigmaSpace=75)
    timings["Bilateral Filter"] = time.time() - start_time

    return timings
```

27 I. Natural High-Detail Image:

[50]:
```
image = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/124084.jpg', cv2.IMREAD_GRAYSCALE)
kernel_sizes = [3, 5, 7]
results = []

for kernel_size in kernel_sizes:
```

```
    timings = apply_filters_with_timing(image, kernel_size)
    for filter_name, duration in timings.items():
        results.append({
            "Filter": filter_name,
            "Kernel Size": kernel_size,
            "Time (s)": duration
        })

results_df = pd.DataFrame(results)
results_df
```

[50]:

	Filter	Kernel Size	Time (s)
0	Box Filter	3	0.000000
1	Gaussian Filter	3	0.000000
2	Median Filter	3	0.001008
3	Adaptive Mean Filter	3	1.522937
4	Adaptive Median Filter	3	9.037359
5	Bilateral Filter	3	0.000000
6	Box Filter	5	0.003665
7	Gaussian Filter	5	0.000000
8	Median Filter	5	0.000000
9	Adaptive Mean Filter	5	3.651036
10	Adaptive Median Filter	5	12.664929
11	Bilateral Filter	5	0.001599
12	Box Filter	7	0.000000
13	Gaussian Filter	7	0.000000
14	Median Filter	7	0.018737
15	Adaptive Mean Filter	7	3.620040
16	Adaptive Median Filter	7	13.031401
17	Bilateral Filter	7	0.015908

The computational performance results indicate that built-in OpenCV filters (Box, Gaussian, and Bilateral) are highly efficient, with near-instant processing times across all kernel sizes due to OpenCV's optimizations. In contrast, the custom adaptive mean and adaptive median filters require significantly more processing time, especially as kernel size increases, reflecting their higher computational complexity. These findings suggest that while adaptive filters are effective for specialized noise reduction, they can be quite slow, thus they should be used thoughtfully, especially with high-detail images or larger kernel sizes, where processing time can quickly increase.

28 II. Natural Low-Detail Image:

[52]:
```
image2 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/135069.jpg', cv2.IMREAD_GRAYSCALE)
kernel_sizes = [3, 5, 7]
results2 = []

for kernel_size in kernel_sizes:
```

```
    timings = apply_filters_with_timing(image2, kernel_size)
    for filter_name, duration in timings.items():
        results2.append({
            "Filter": filter_name,
            "Kernel Size": kernel_size,
            "Time (s)": duration
        })

results_df2 = pd.DataFrame(results2)
results_df2
```

[52]:

	Filter	Kernel Size	Time (s)
0	Box Filter	3	0.002556
1	Gaussian Filter	3	0.000000
2	Median Filter	3	0.000839
3	Adaptive Mean Filter	3	1.467049
4	Adaptive Median Filter	3	9.184597
5	Bilateral Filter	3	0.000000
6	Box Filter	5	0.000000
7	Gaussian Filter	5	0.000000
8	Median Filter	5	0.005273
9	Adaptive Mean Filter	5	3.580047
10	Adaptive Median Filter	5	16.566093
11	Bilateral Filter	5	0.003754
12	Box Filter	7	0.000000
13	Gaussian Filter	7	0.004632
14	Median Filter	7	0.010431
15	Adaptive Mean Filter	7	3.629869
16	Adaptive Median Filter	7	18.976285
17	Bilateral Filter	7	0.015738

For low-detail images, the results show that box and Gaussian filters work quickly and efficiently, with almost no processing delay, regardless of kernel size. The median filter offers a good balance, providing solid noise reduction without too much extra processing time. On the other hand, adaptive filters, especially the adaptive median filter, take much longer to process, particularly with larger kernels, which may make them less practical for tasks where speed is essential. The bilateral filter stands out for delivering effective noise reduction with minimal time impact, making it a great option when both quality and quick results are needed.

29 III. Influenced by Human High-Detail Image:

[53]:
```
image3 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/138032.jpg', cv2.IMREAD_GRAYSCALE)
kernel_sizes = [3, 5, 7]
results3 = []

for kernel_size in kernel_sizes:
```

```
    timings = apply_filters_with_timing(image3, kernel_size)
    for filter_name, duration in timings.items():
        results3.append({
            "Filter": filter_name,
            "Kernel Size": kernel_size,
            "Time (s)": duration
        })

results_df3 = pd.DataFrame(results3)
results_df3
```

[53]:

	Filter	Kernel Size	Time (s)
0	Box Filter	3	0.000000
1	Gaussian Filter	3	0.000000
2	Median Filter	3	0.000000
3	Adaptive Mean Filter	3	1.481490
4	Adaptive Median Filter	3	9.236687
5	Bilateral Filter	3	0.000000
6	Box Filter	5	0.000000
7	Gaussian Filter	5	0.000000
8	Median Filter	5	0.009953
9	Adaptive Mean Filter	5	3.000093
10	Adaptive Median Filter	5	9.750248
11	Bilateral Filter	5	0.000000
12	Box Filter	7	0.000000
13	Gaussian Filter	7	0.000000
14	Median Filter	7	0.009428
15	Adaptive Mean Filter	7	1.540299
16	Adaptive Median Filter	7	6.269878
17	Bilateral Filter	7	0.000000

Here, computational time varied widely among filters. Box and Gaussian filters were the quickest, with virtually no delay across all kernel sizes. Median and bilateral filters also performed efficiently, with only a slight increase in time at larger kernel sizes. However, adaptive filters, especially the adaptive median filter, required significantly more processing time, especially as the kernel size increased. This trade-off suggests that while adaptive filters can improve noise reduction, their higher computational cost may not be ideal for high-detail images where speed is crucial.

30 IV. Influenced by Human Low-Detail Image:

[55]:
```
image4 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/161062.jpg', cv2.IMREAD_GRAYSCALE)
kernel_sizes = [3, 5, 7]
results4 = []

for kernel_size in kernel_sizes:
    timings = apply_filters_with_timing(image4, kernel_size)
```

```
    for filter_name, duration in timings.items():
        results4.append({
            "Filter": filter_name,
            "Kernel Size": kernel_size,
            "Time (s)": duration
        })

results_df4 = pd.DataFrame(results4)
results_df4
```

[55]:

	Filter	Kernel Size	Time (s)
0	Box Filter	3	0.000000
1	Gaussian Filter	3	0.000000
2	Median Filter	3	0.000000
3	Adaptive Mean Filter	3	1.505180
4	Adaptive Median Filter	3	9.189925
5	Bilateral Filter	3	0.000000
6	Box Filter	5	0.000000
7	Gaussian Filter	5	0.000000
8	Median Filter	5	0.005131
9	Adaptive Mean Filter	5	2.248376
10	Adaptive Median Filter	5	9.196568
11	Bilateral Filter	5	0.005014
12	Box Filter	7	0.000000
13	Gaussian Filter	7	0.000000
14	Median Filter	7	0.010931
15	Adaptive Mean Filter	7	1.533556
16	Adaptive Median Filter	7	12.250568
17	Bilateral Filter	7	0.014967

Here, the computational times for different filters showed a similar trend as with high-detail images. The Box and Gaussian filters were extremely fast, maintaining nearly zero processing time across all kernel sizes. The Median and Bilateral filters also performed efficiently, with minor increases at larger kernel sizes. In contrast, the Adaptive Mean and Adaptive Median filters required significantly more processing time, especially as kernel size increased, with the Adaptive Median filter being the most time-intensive. These results highlight the efficiency of simpler filters for low-detail images, where processing speed might outweigh the benefits of complex filtering.

31 V. Edge-Rich Image:

[56]:
```
image5 = cv2.imread('C:/Users/asus/OneDrive/Desktop/Computer Vision/archive/
 ↪images/train/138032.jpg', cv2.IMREAD_GRAYSCALE)
kernel_sizes = [3, 5, 7]
results5 = []

for kernel_size in kernel_sizes:
    timings = apply_filters_with_timing(image5, kernel_size)
```

```
    for filter_name, duration in timings.items():
        results5.append({
            "Filter": filter_name,
            "Kernel Size": kernel_size,
            "Time (s)": duration
        })

results_df5 = pd.DataFrame(results5)
results_df5
```

[56]:

	Filter	Kernel Size	Time (s)
0	Box Filter	3	0.000000
1	Gaussian Filter	3	0.000000
2	Median Filter	3	0.000000
3	Adaptive Mean Filter	3	1.529896
4	Adaptive Median Filter	3	8.684910
5	Bilateral Filter	3	0.000000
6	Box Filter	5	0.000000
7	Gaussian Filter	5	0.000000
8	Median Filter	5	0.010370
9	Adaptive Mean Filter	5	1.972001
10	Adaptive Median Filter	5	6.317603
11	Bilateral Filter	5	0.000000
12	Box Filter	7	0.000000
13	Gaussian Filter	7	0.000000
14	Median Filter	7	0.010393
15	Adaptive Mean Filter	7	3.469904
16	Adaptive Median Filter	7	11.479854
17	Bilateral Filter	7	0.015022

For edge-rich images, the Box and Gaussian filters again performed exceptionally fast, showing zero or near-zero processing time across all kernel sizes. Median and Bilateral filters were also efficient, maintaining low processing times even with larger kernels. However, Adaptive Mean and Adaptive Median filters required considerably more time, with the Adaptive Median filter being the most time-consuming, especially at larger kernel sizes. These results suggest that simpler filters are suitable for edge-rich images where speed is crucial, while adaptive filters offer more refined processing at a cost of longer computation time.

32 Summary of Findings and Recommendations

33 PART A: Kernel Size Sensitivity Analysis

- Box and Gaussian Filters: Highly sensitive to kernel size, with larger kernels leading to excessive smoothing and loss of edge details. Best suited for uniform noise reduction in low-detail regions.
- Median and Adaptive Median Filters: Perform well at smaller kernel sizes (3 or 5), maintaining edges while reducing noise. However, they lose finer details and become computationally

expensive with larger kernels.
- Adaptive Mean Filter: Moderately sensitive to kernel size. Effective in smoothing noise but tends to blur edges, especially at larger kernel sizes. Computationally heavier than Box or Gaussian filters.
- Bilateral Filter: Least sensitive to kernel size. Maintains edge sharpness even with larger kernels by preserving intensity differences while smoothing flat regions. Computational cost is low compared to Adaptive Median and Adaptive Mean filters.

34 Recommendations:

- For high-detail or edge-critical applications, prefer smaller kernel sizes (3 or 5) for filters like Median, Adaptive Median, or Bilateral.
- Use Adaptive Mean or Box filters for general noise reduction where edge detail is less critical.
- Larger kernels should be used only in extreme noise cases where edge blurring is acceptable.

35 PART B: Exploring Trade-offs

- Box and Gaussian Filters: Offer the best computational efficiency and are ideal for uniform noise reduction. However, they lack edge preservation and are unsuitable for detail-sensitive applications.
- Median and Adaptive Median Filters: Effective for impulse noise and preserving edges, but their computational cost rises significantly with kernel size, especially for Adaptive Median.
- Adaptive Mean Filter: Provides balanced noise reduction but struggles with edge sharpness. Computational cost is higher than simple filters like Box or Gaussian.
- Bilateral Filter: Excels in balancing edge preservation and noise reduction, even at larger kernel sizes, with a low computational cost.

36 Recommendations:

- For edge-sensitive tasks, prioritize Median or Bilateral filters.
- For computationally efficient solutions, Box or Gaussian filters work best.
- Use Adaptive Mean or Adaptive Median filters when more sophisticated noise reduction is needed but ensure resources are sufficient to handle the computational load.