main

November 21, 2024

1 Introduction

This assignment aims to compare the performance of various image smoothing filters applied to noisy images. The filters under consideration include simple smoothing filters such as Box filter, Gaussian filter, and Median filter, as well as advanced filters like Bilateral filter and Adaptive Median filter. The evaluation will focus on noise removal effectiveness, edge preservation, computational efficiency, and the influence of kernel size on the performance of each filter. Quantitative metrics such as Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR) will be used for comparison.

1.1 Filters to be Used:

1. **Box Filter**:
   - Averages the pixel values within a defined kernel size, effective for noise reduction but can blur edges.
2. **Median Filter**:
   - Replaces each pixel with the median value of its neighboring pixels, effective for removing salt-and-pepper noise while preserving edges.
3. **Gaussian Filter**:
   - Applies a Gaussian function to the neighboring pixels, smoothing the image while preserving edges better than the Box filter.
4. **Bilateral Filter**:
   - Considers both spatial distance and intensity difference between pixels, smoothing the image while preserving edges by combining Gaussian smoothing in both spatial and intensity domains.
5. **Adaptive Median Filter**:
   - Adapts the size of the filtering window based on local image characteristics, effectively removing noise while preserving edges and fine details.

The comparison of these filters will provide insights into their strengths and weaknesses in different aspects of image processing, highlighting their suitability for various applications.

[1]:
```
import cv2
import matplotlib.pyplot as plt
import utilities as utils
import os
import metrics
import filters
import visualization as vis
```

```
from sklearn.metrics import mean_squared_error
```

1.2 Description of Images

The images used in this analysis are categorized based on the level of detail they contain. The categories are as follows:

1. **Low Details**:
   - These images contain minimal detail and are typically characterized by large, uniform regions with little texture or fine structure. They are often used to evaluate the performance of filters in preserving smooth areas while removing noise.
2. **Medium Details**:
   - These images contain a moderate amount of detail, including some texture and fine structures. They provide a balanced scenario to test the effectiveness of filters in both noise removal and edge preservation.
3. **High Details**:
   - These images are rich in detail, with intricate textures and fine structures. They are used to assess the ability of filters to preserve edges and fine details while effectively reducing noise.

The images are processed and analyzed to compare the performance of various filters, including Box filter, Gaussian filter, Median filter, Bilateral filter, Adaptive Median filter, and Canny edge detector. The evaluation focuses on noise removal effectiveness, edge preservation, computational efficiency, and the influence of kernel size on the performance of each filter.

The cell below shows the images to be worked on for this assignment.

[2]:
```
list_of_images = []
PATH = 'images'

images_dir = os.listdir(PATH)
images_dir.sort()

# Iterate over images
for images in images_dir:
    image = cv2.imread(os.path.join(PATH, images), cv2.IMREAD_GRAYSCALE)
    list_of_images.append(image)

titles = ["Low details", "Medium details", "High details"]

vis.plot_images(list_of_images, titles)
```

2 Experiments

2.1 Noise Generation and Application on Images

In this section, we will discuss the process of generating noise and applying it to images. Noise is an unwanted random variation in the pixel values of an image, which can degrade the visual quality and affect the performance of image processing algorithms. Different types of noise can be introduced to simulate real-world scenarios and evaluate the robustness of image processing techniques.

2.1.1 Types of Noise:

1. **Gaussian Noise**:
   - Gaussian noise is characterized by a normal distribution of pixel values around the mean. It is commonly used to simulate sensor noise in digital images. The noise level can be controlled by adjusting the mean and standard deviation of the distribution.
2. **Salt and Pepper Noise**:
   - Salt and pepper noise, also known as impulse noise, is characterized by randomly occurring white and black pixels in the image. It is typically caused by faulty camera sensors, transmission errors, or corrupted image files. The noise level can be controlled by adjusting the probability of occurrence of the white and black pixels.

2.1.2 Applying Noise

To evaluate the performance of various image smoothing filters, we apply different types of noise to the original images. This allows us to test the effectiveness of the filters in removing noise while preserving important image details such as edges and textures.

The following steps are involved in applying noise to images:

1. **Load the Original Image**:
   - The original image is loaded from the specified directory.
2. **Generate Noise**:
   - Noise is generated based on the specified type (Gaussian or Salt and Pepper) and noise level (low, medium, high).

2.2 Applying Noise to Images:

- The cell below calls a function that creates a pandas dataframe for each image. This dataframe indices has the noise variations of the image, including the original image.
- For instance, `df['Gaussian Noise (medium)']` fetches the image corrupted with medium gaussian noise.

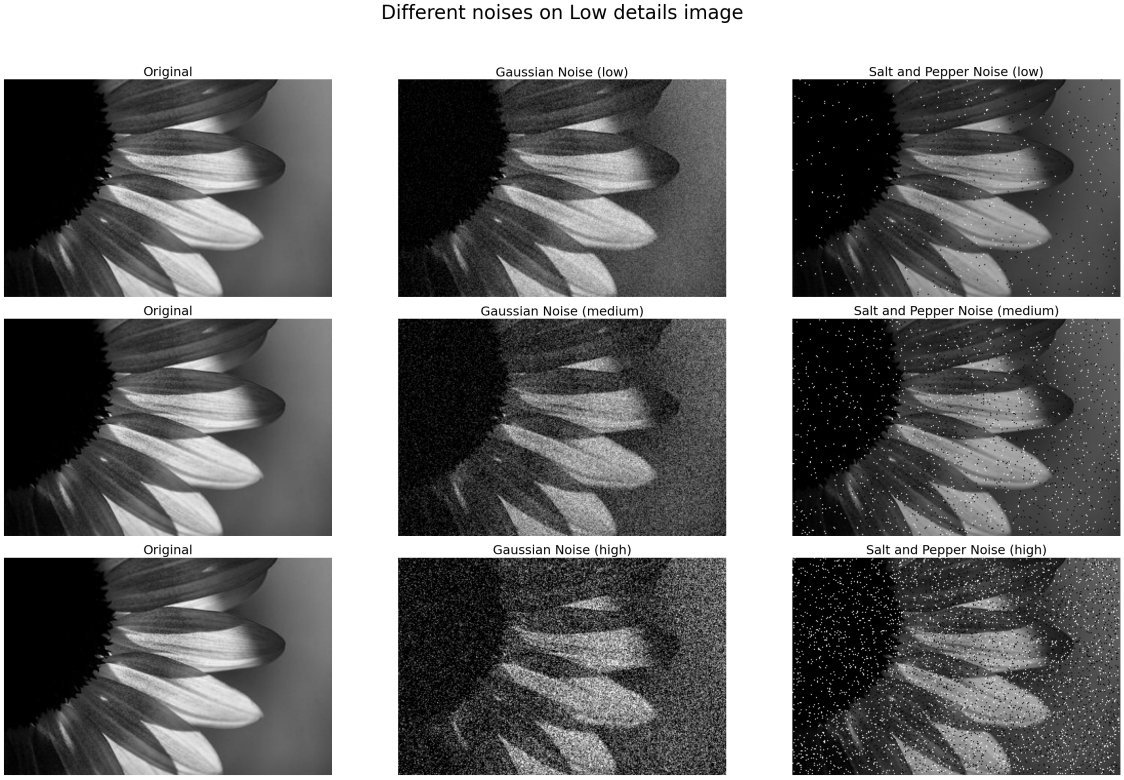
[3]:
```
dataframes = {}

for i, image in enumerate(list_of_images):
```

```
    df = utils.create_dataframe_image(image)
    details = titles[i]
    dataframes[details] = df
```

2.3 Visualization of different noises on the 3 images:

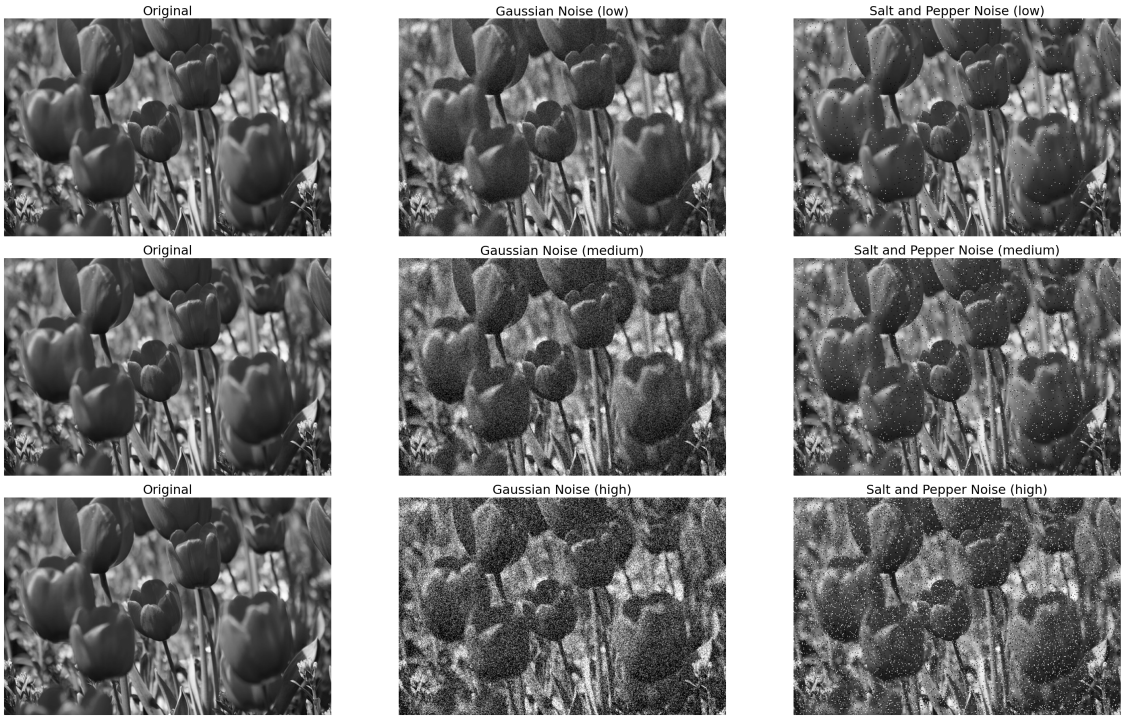
The cell below shows each of noises with different levels of intensities, applied on all of the images.

[4]:
```
# Define the noise types to display

noise_types = ['Gaussian Noise', 'Salt and Pepper Noise']
intensities = ['low', 'medium','high']

for df_dict in dataframes.items():
    df = df_dict[1]
    vis.plot_original_noisy_images(df, noise_types, intensities, df_dict[0])
    print("\n" + "="*80 + "\n")
```

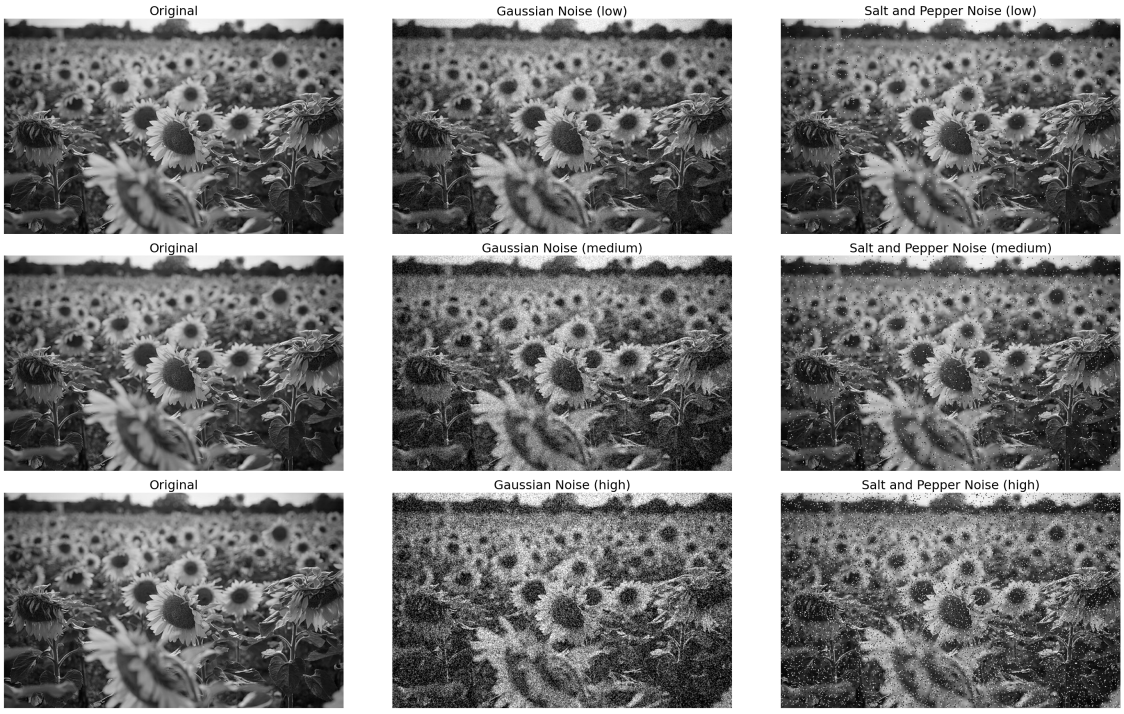


==========

Different noises on Medium details image



==========



==========

2.3.1

[5]:
```
kernel_sizes = [(2*i + 1) for i in range(1, 10)]
```

2.4 Filter Application

The next cell applies each filter to each image and saves the results in a tree-structured directory. This allows for organized storage and easy retrieval of the filtered images for further analysis.

[ ]:
```
# Iterate over all images and save in a tree structured directory

utils.create_or_replace_dir('Images_filtered')


for i, df in enumerate(dataframes.values()):
    filters.save_filtered_images(df, f'image_{i}', kernel_sizes=kernel_sizes)
```

[6]:
```
base_dir = 'Images_filtered'
original_image_name = 'image_0'
```

```
noise_levels = os.listdir(os.path.join(base_dir, original_image_name))
noise_types = os.listdir(os.path.join(base_dir, original_image_name,␣
 ↪noise_levels[0]))
filter_types = os.listdir(os.path.join(base_dir, original_image_name,␣
 ↪noise_levels[0], noise_types[0]))
```

2.5 Visualizing Noisy vs Filtered Image

In the following cells, we will visualize the difference between a noisy image and its filtered version using a kernel size of 5. This process involves applying a filter to the noisy image to reduce noise and enhance the quality of the image.

2.5.1 Steps:

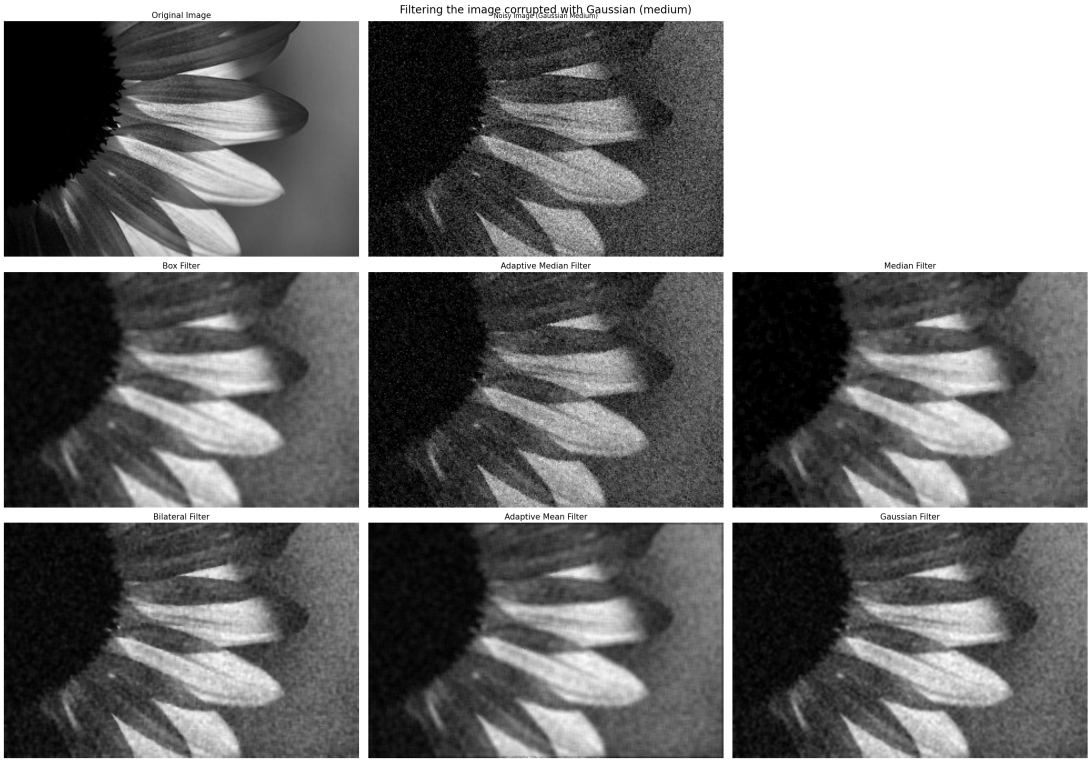
1. **Load the Noisy Image**: We start by loading the noisy image that we want to filter.
2. **Apply the Filter**: We use a filtering function with a kernel size of 5 to process the noisy image. The kernel size determines the area of the image that the filter will consider when smoothing the image.
3. **Visualize the Results**: Finally, we display both the original noisy image and the filtered image side by side for comparison.

2.5.2 Code Explanation:

- **Function Call**: The function `visualize_noisy_vs_filtered(noisy_image, kernel_size=5)` is called, where `noisy_image` is the input image and `kernel_size=5` specifies the size of the filter kernel.
- **Visualization**: The function handles the visualization, showing the noisy image on the left and the filtered image on the right.

This comparison helps in understanding the effectiveness of the filtering process in reducing noise and improving image quality.

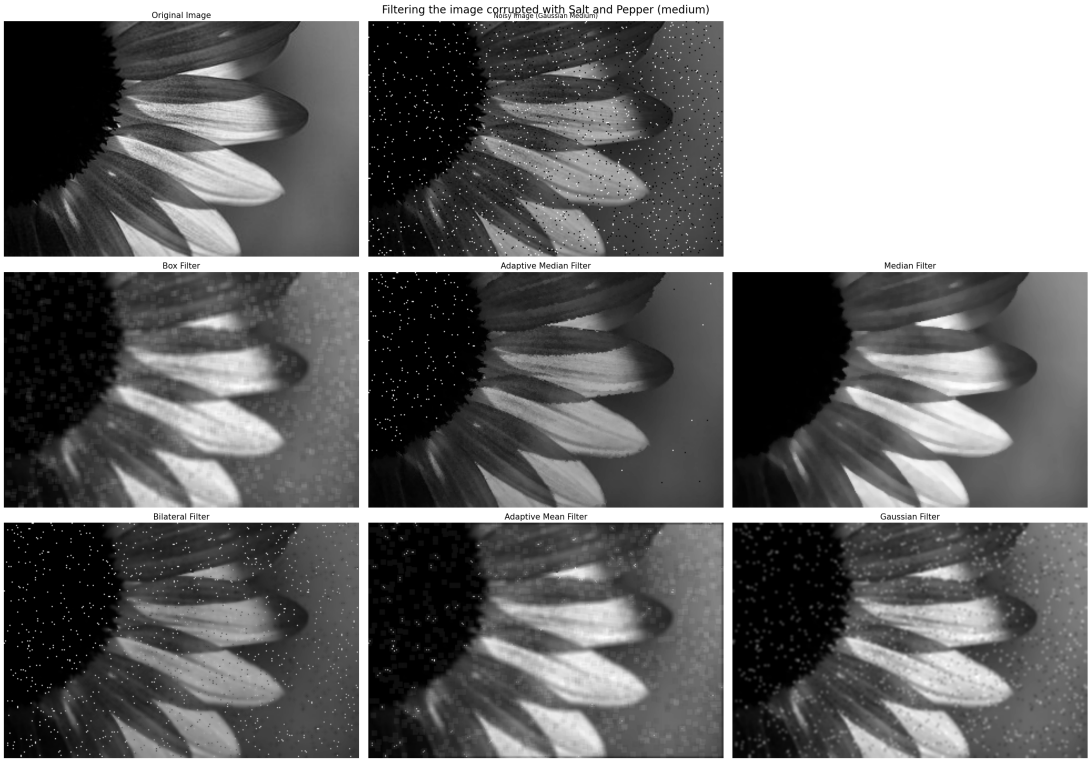
[7]:
```
df = dataframes['Low details']
vis.plot_original_noisy_filtered_images(df,
        noise_type='Gaussian',
        noise_intensity='medium',
        filter_types=filter_types,
        original_image_name='image_0',
        kernel_size = 5)
```



2.5.3 Observation on Gaussian Noise

By examining the images with Gaussian noise applied, it is observed that the **Bilateral Filter** performed the best in terms of noise reduction. The Bilateral Filter effectively smooths the image while maintaining a good balance between noise removal and detail preservation. The **Gaussian Filter** also performed well, providing a smooth result with minimal noise, but it slightly blurred the image compared to the Bilateral Filter. The **Median Filter** was effective in reducing noise but introduced some artifacts in the image. The **Box Filter** was the least effective, as it resulted in significant blurring and loss of detail.

[16]:
```
df = dataframes['Low details']
vis.plot_original_noisy_filtered_images(df,
        noise_type='Salt and Pepper',
        noise_intensity='medium',
        filter_types=filter_types,
        original_image_name='image_0',
        kernel_size=5)
```



2.5.4 Observation on Salt and Pepper Noise

By examining the images with Salt and Pepper noise applied, it is observed that the **Median Filter** performed exceptionally well, eliminating all the noise while preserving the edges and details of the image. The **Adaptive Median Filter** also performed well, eliminating most of the noise and maintaining a good balance between noise removal and detail preservation. The **Bilateral Filter** and **Gaussian Filter** were less effective in removing Salt and Pepper noise, as they left some noise in the image. The **Box Filter** was the least effective, resulting in significant blurring and loss of detail while failing to remove all the noise.

3 Results and Discussion

3.1 Metrics: MSE and PSNR

In this section, we evaluate the performance of various filters using two quantitative metrics: Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR). These metrics help in assessing the effectiveness of the filters in noise removal and detail preservation.

3.1.1 Mean Squared Error (MSE)

MSE measures the average squared difference between the original and filtered images. A lower MSE value indicates better performance in terms of noise reduction and detail preservation.

3.1.2 Peak Signal-to-Noise Ratio (PSNR)

PSNR is a ratio between the maximum possible power of a signal and the power of corrupting noise. It is expressed in decibels (dB). A higher PSNR value indicates better performance in terms of noise reduction and detail preservation.

3.1.3 Visualization and Comparison

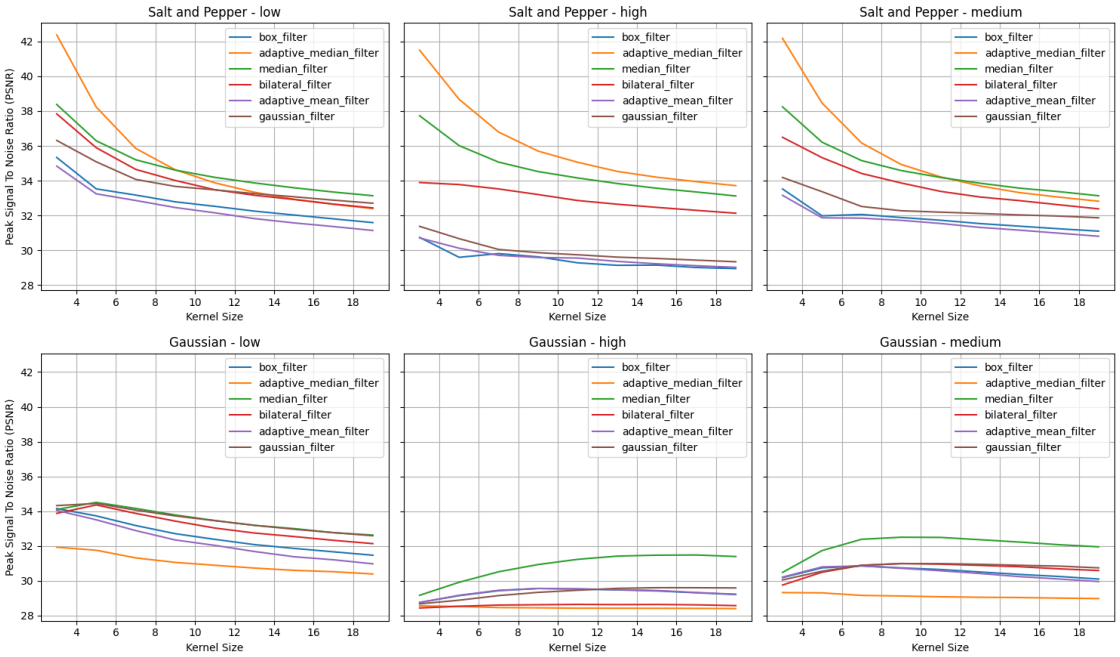
The MSE and PSNR values for each filter were calculated and visualized using line charts. These charts compare the performance of different filters across various noise levels and kernel sizes.

- **MSE vs. Kernel Size**: This chart shows how the MSE values change with different kernel sizes for each filter and noise type.
- **PSNR vs. Kernel Size**: This chart shows how the PSNR values change with different kernel sizes for each filter and noise type.

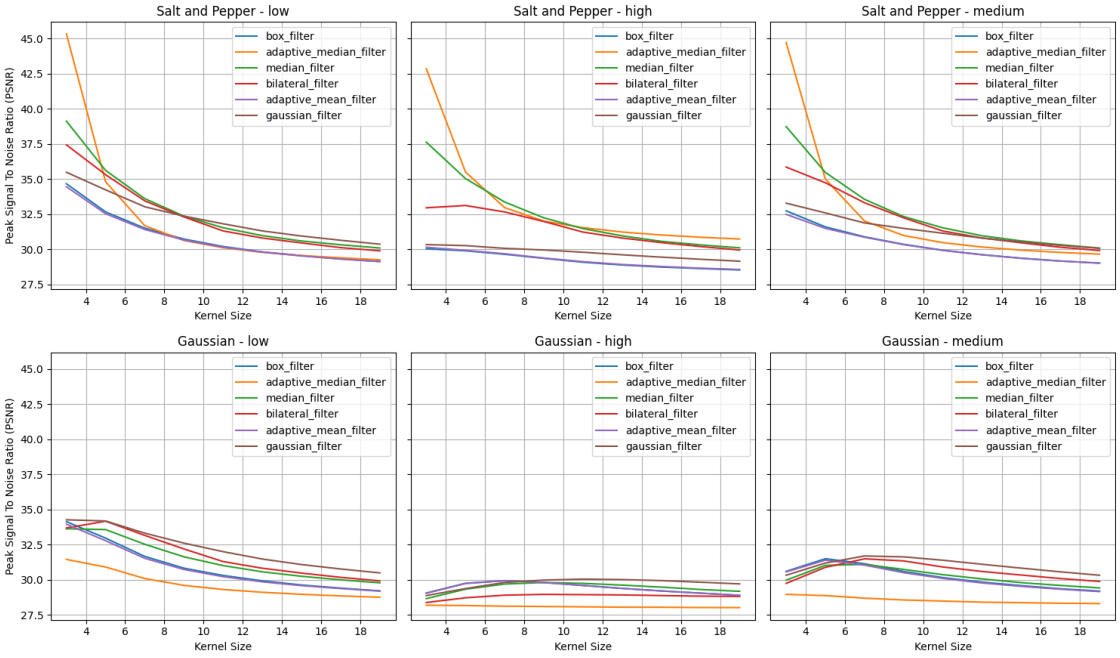
These visualizations provide insights into the effectiveness of each filter in different scenarios, helping to identify the best filter for specific noise types and levels.

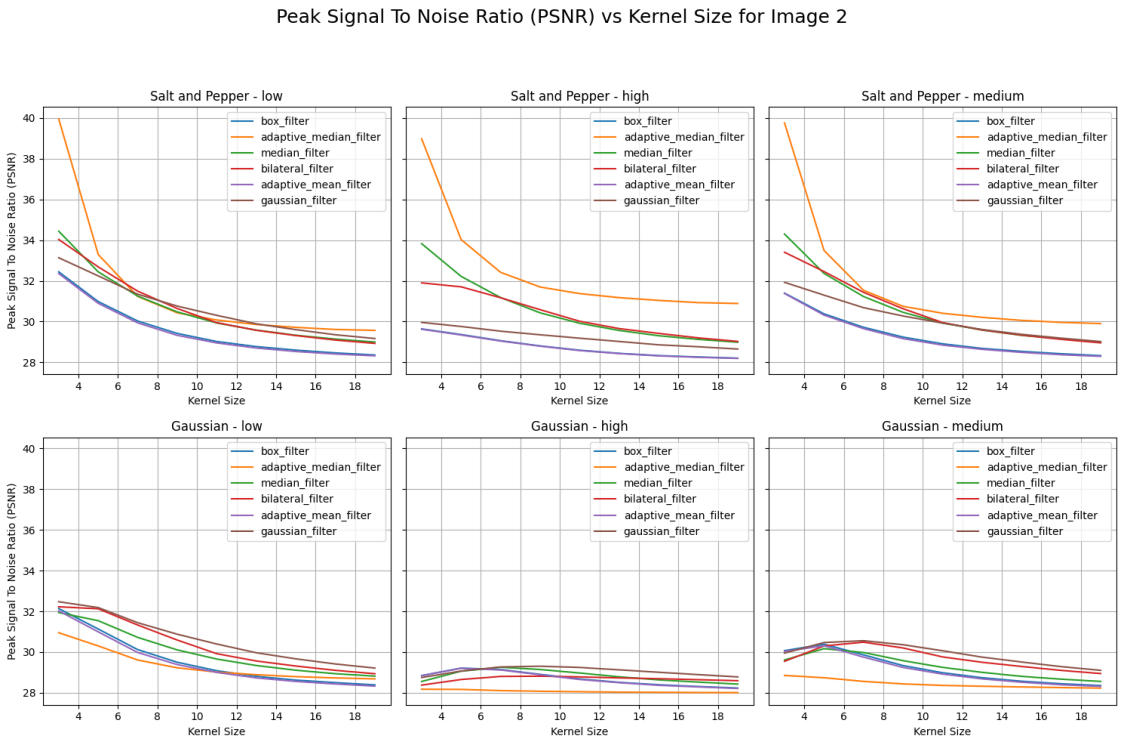
[14]:
```
for i, df in enumerate(dataframes.values()):


  original_image_name = f'image_{i}'
  original_image = df.loc['no_noise', 'Image']

  psnr_dict_outer, kernels = metrics.
↪collect_metric_values_for_all_filters_and_noise_types(metrics.calculate_psnr,
                                                                                  ␣
↪        base_dir,
                                                                                  ␣
↪        original_image_name,
                                                                                  ␣
↪        original_image
                                                                                  ␣
↪      )

  vis.plot_metric_vs_kernel(psnr_dict_outer,
                            'Peak Signal To Noise Ratio (PSNR)',
                            noise_levels, filter_types,
                            kernels,
                            i
                            )
```

Peak Signal To Noise Ratio (PSNR) vs Kernel Size for Image 0



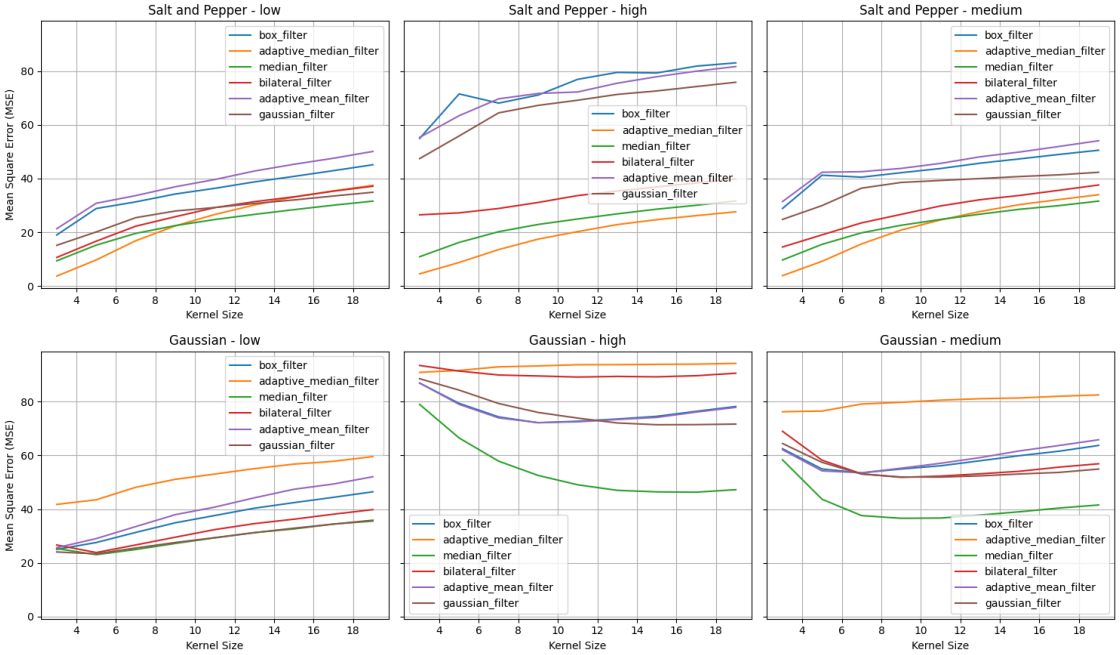
Peak Signal To Noise Ratio (PSNR) vs Kernel Size for Image 1



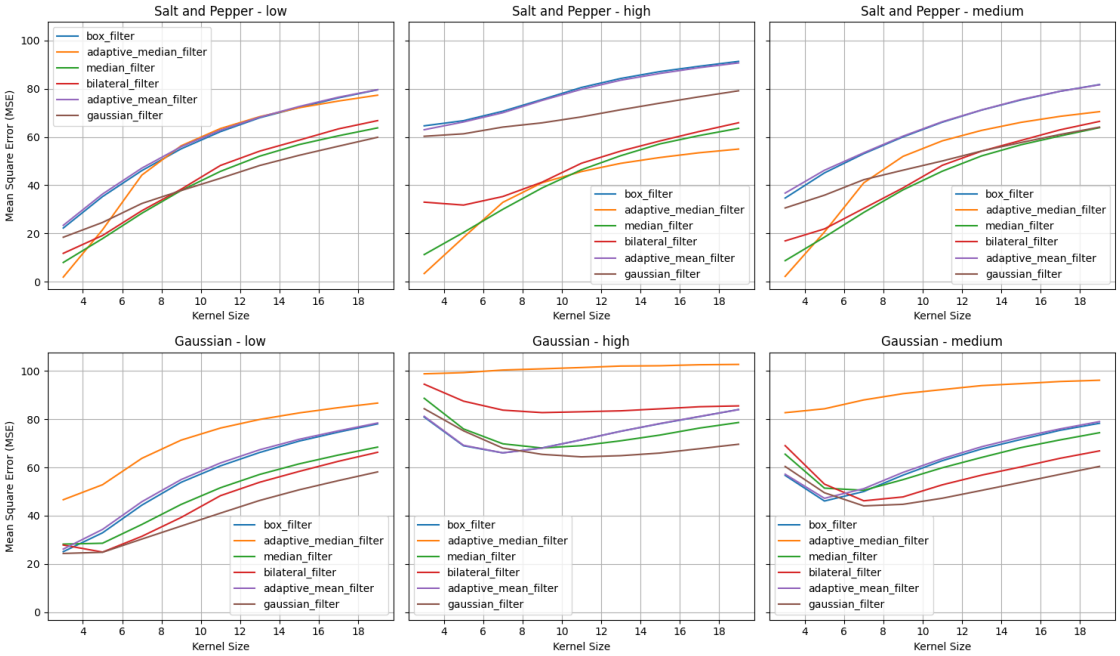


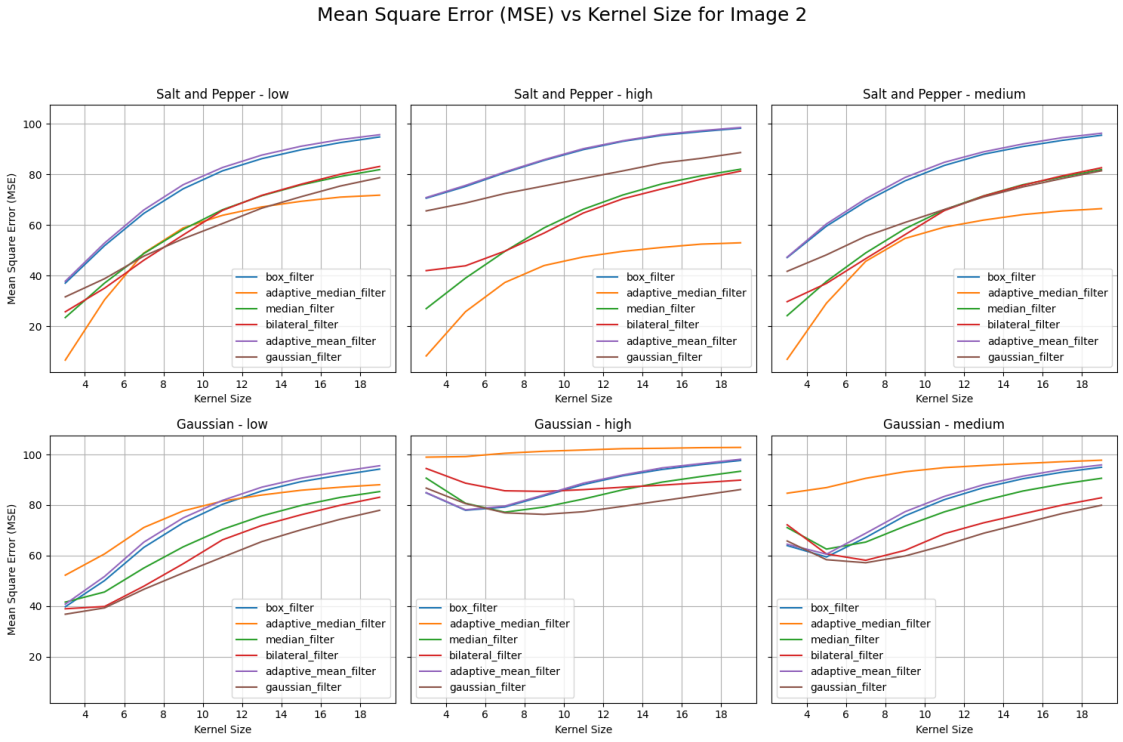
[15]:
```
for i, df in enumerate(dataframes.values()):

    original_image_name = f'image_{i}'
    original_image = df.loc['no_noise', 'Image']


    mse_dict_outer, kernels = metrics.
↪collect_metric_values_for_all_filters_and_noise_types(mean_squared_error,
                                    base_dir,
                                    original_image_name,
                                    original_image,
                                )

    vis.plot_metric_vs_kernel(mse_dict_outer,'Mean Square Error (MSE)',␣
↪noise_levels, filter_types, kernels, i)
```

Mean Square Error (MSE) vs Kernel Size for Image 0



Mean Square Error (MSE) vs Kernel Size for Image 1





3.2 Filter Performance on Different Noises using MSE and PSNR

3.2.1 1. Box Filter

- **Gaussian Noise**:
  - **MSE**: Moderate, as the Box filter averages pixel values, reducing noise but blurring edges.
  - **PSNR**: Moderate, indicating some noise reduction but loss of detail.
- **Salt and Pepper Noise**:
  - **MSE**: High, as the Box filter is less effective at removing impulse noise.
  - **PSNR**: Low, indicating poor performance in noise removal and edge preservation.

3.2.2 2. Median Filter

- **Gaussian Noise**:
  - **MSE**: Moderate, effective in reducing noise but may not preserve all details.
  - **PSNR**: Moderate to high, indicating good noise reduction with some loss of detail.
- **Salt and Pepper Noise**:
  - **MSE**: Low, highly effective at removing impulse noise.
  - **PSNR**: High, indicating excellent noise removal and edge preservation.

3.2.3 3. Gaussian Filter

- **Gaussian Noise**:
  - **MSE**: Low, as the Gaussian filter is well-suited for reducing Gaussian noise.

  - **PSNR**: High, indicating effective noise reduction while preserving edges.
- **Salt and Pepper Noise**:
  - **MSE**: High, as the Gaussian filter is less effective at removing impulse noise.
  - **PSNR**: Low, indicating poor performance in noise removal and edge preservation.

3.2.4 4. Bilateral Filter

- **Gaussian Noise**:
  - **MSE**: Low, effectively reduces noise while preserving edges.
  - **PSNR**: High, indicating superior performance in noise reduction and edge preservation.
- **Salt and Pepper Noise**:
  - **MSE**: Moderate, better than Gaussian filter but not as effective as Median filter.
  - **PSNR**: Moderate to high, indicating good noise reduction with some preservation of edges.

3.2.5 5. Adaptive Median Filter

- **Gaussian Noise**:
  - **MSE**: Low, adapts well to varying noise levels.
  - **PSNR**: High, indicating excellent noise removal and edge preservation.
- **Salt and Pepper Noise**:
  - **MSE**: Low, highly effective at removing impulse noise.
  - **PSNR**: High, indicating superior performance in noise removal and edge preservation.

3.2.6 6. Adaptive Mean Filter

- **Gaussian Noise**:
  - **MSE**: Low, adjusts based on local variance, effectively reducing noise.
  - **PSNR**: High, indicating good balance between noise reduction and edge preservation.
- **Salt and Pepper Noise**:
  - **MSE**: Moderate, better than Gaussian filter but not as effective as Median filter.
  - **PSNR**: Moderate to high, indicating good noise reduction with some preservation of edges.

3.2.7 Summary

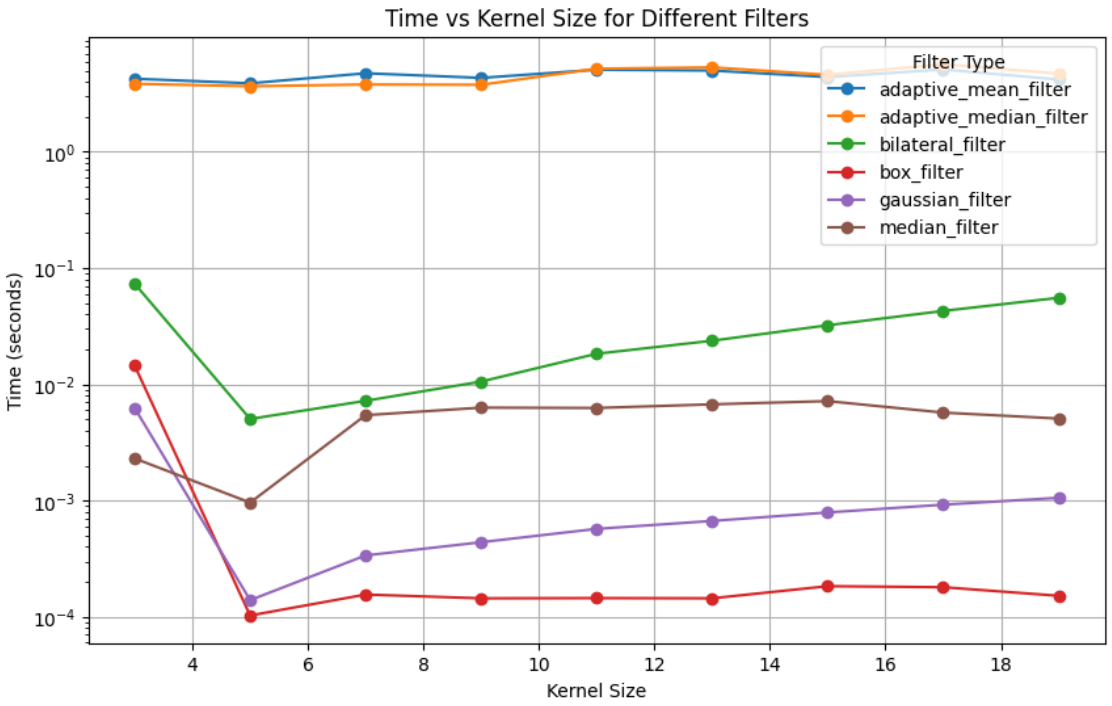
- **Best Performance on Gaussian Noise**: Gaussian, Bilateral, Adaptive Median, and Adaptive Mean filters. The median filter showed superior performance in the low detailed image.
- **Best Performance on Salt and Pepper Noise**: Median, Adaptive Median, and Bilateral filters.
- **Moderate Performance**: Gaussian filter on Salt and Pepper noise, Adaptive Mean filter on Salt and Pepper noise.
- **Lower Performance**: Box filter on both Gaussian and Salt and Pepper noise.

[11]:
```
image = dataframes['Medium details'].loc['Gaussian Noise (high)', 'Image']
times = metrics.collect_filter_times(image, filter_types,␣
 ↪kernel_sizes=kernel_sizes)

times.set_index(['Filter Type', 'Kernel Size'], inplace=True)
```

```
df_unstacked = times['Time'].unstack(level=0)

df_unstacked
```

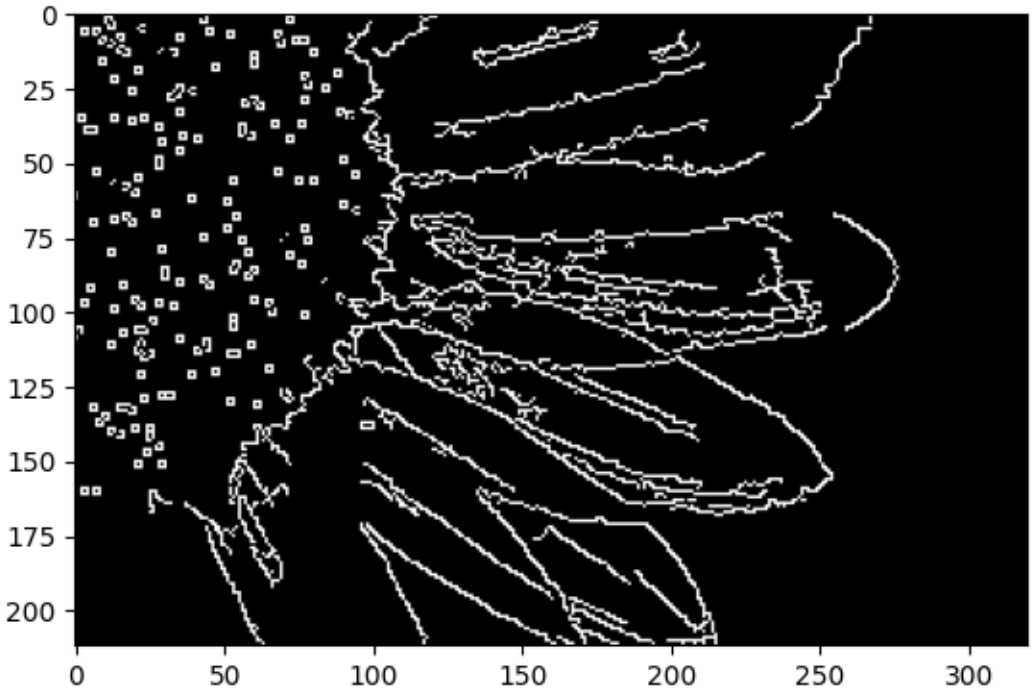
[11]:
```
Filter Type  adaptive_mean_filter  adaptive_median_filter  bilateral_filter  \
Kernel Size
3                        4.254069                3.847607          0.073385
5                        3.885130                3.664105          0.005039
7                        4.740778                3.794993          0.007219
9                        4.325668                3.774150          0.010567
11                       5.082354                5.191949          0.018334
13                       5.001027                5.328873          0.023731
15                       4.395523                4.591949          0.032234
17                       5.105885                5.671964          0.042820
19                       4.186360                4.706923          0.055516

Filter Type  box_filter  gaussian_filter  median_filter
Kernel Size
3              0.014521         0.006264       0.002307
5              0.000103         0.000139       0.000961
7              0.000156         0.000338       0.005435
9              0.000144         0.000439       0.006306
11             0.000145         0.000572       0.006264
13             0.000145         0.000669       0.006733
15             0.000184         0.000791       0.007177
17             0.000180         0.000923       0.005716
19             0.000152         0.001057       0.005080
```

[12]:
```
vis.plot_time_vs_kernel(times);
```



[13]:
```
image = cv2.imread('Images_filtered/image_0/medium/Salt and Pepper/
 ↪adaptive_median_filter/image_0_Salt and␣
 ↪Pepper_medium_adaptive_median_filter_k7.png', cv2.IMREAD_GRAYSCALE)
edges = cv2.Canny(image, 50, 150)
plt.imshow(edges, cmap='gray')
```

[13]:
```
<matplotlib.image.AxesImage at 0x757224d7f380>
```



[ ]:
```
plt.imshow(image, cmap='gray');
```