



Ministry of Education and Research of the Republic of
Moldova

Technical University of Moldova

Department of Software and Automation Engineering

REPORT

Individual Work No. 1

Discipline: Signal Processing

Elaborated:

Berco Andrei, FAF - 221

Checked:

Railean Serghei

Chişinău 2025

Purpose of the work

Generation of noise and its filtering using the Discrete-Time System "M-point Moving Average System."

Theoretical Notes

A system is defined as any device or algorithm that performs operations on a signal.

A **Discrete-Time System** is any device or algorithm that influences a discrete-time signal, called the **Input Signal** or **Excitation** – $x(n)$, according to well-defined rules to obtain another discrete-time signal called the **Output Signal** – $y(n)$ or **Response**.

The **input-output** relationship consists of mathematical expressions or fixed rules that define the connection between the input and output signals. The exact internal structure of the system is often unknown or ignored.

A system is called **Static** or **Memoryless** if the output signal at any moment n depends only on the input signal at the same moment n , and not on previous or future moments. Otherwise, the system is **Dynamic** or **with memory**. If the output signal at a given moment n depends on the input signal over the interval $n-N$ to n , then the system has memory of duration N . If $N = 0$, the system is **Static**.


A system is called **time-invariant** if its input-output characteristics do not depend on time. If we shift the input signal by k units – $x(n-k)$ – the output will also be shifted by k units: $y(n-k)$.

A system is called **linear** if it satisfies the **superposition principle** – meaning that the system's response to the sum of multiple input signals is equal to the sum of the system's responses to each individual input signal.

Practical Tasks

1. Study of Random Processes

1.1. **White noise** with a Gaussian dependence is generated using the rand procedure. Generate a random process as follows



```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # Parameters
5  Ts = 0.01
6  t = np.arange(0, 5, Ts)
7
8  # Generate white noise
9  x1 = np.random.rand(len(t))
10
11 # Plot the noise
12 plt.figure(figsize=(10, 6))
13 plt.plot(t, x1)
14 plt.title('1.1 White Noise')
15 plt.xlabel('Time (s)')
16 plt.ylabel('Amplitude')
17 plt.grid(True)
18 plt.show()
```

Figure 1: Code for task 1.1

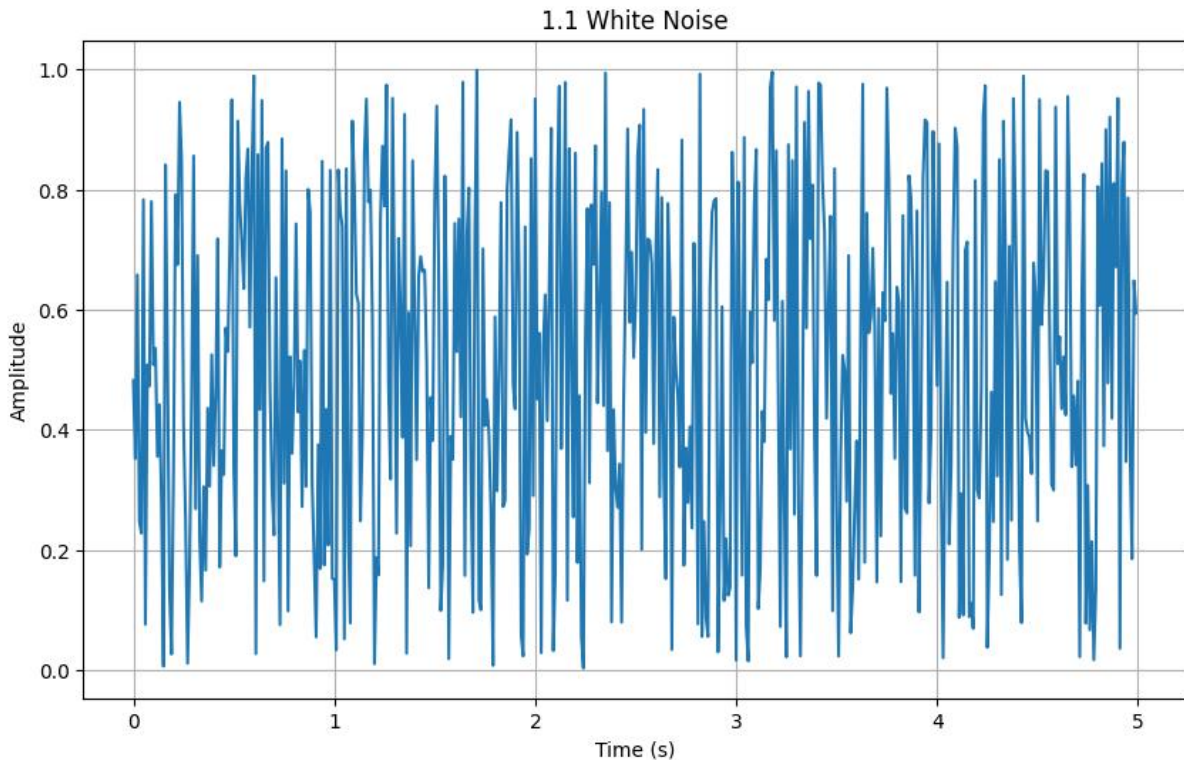


Figure 2: Result for 1.1

Parameters:

$T_s = 0.01$ sets the sampling interval to 0.01 seconds.

$t = np.arange(0, 5, T_s)$ creates an array of time values ranging from 0 to 5 seconds, with a step of 0.01 seconds.

$x1 = np.random.rand(len(t))$ generates an array of random values (white noise) with the same length as the time array t . The values are uniformly distributed between 0 and 1.

The resulting plot shows a random signal fluctuating rapidly over time, which is characteristic of white noise. The x-axis represents time in seconds (0 to 5), and the y-axis represents amplitude (0 to 1).

This plot visually demonstrates the concept of white noise, which has equal intensity at different frequencies. White noise is commonly used in various fields such as signal processing and communications.

1.2. Replace the plot function with hist to represent the histogram of the generated noise. Before doing this, change the time range to 1 and swap the variables t and x1.



Figure 1.3: Code for task 1.2

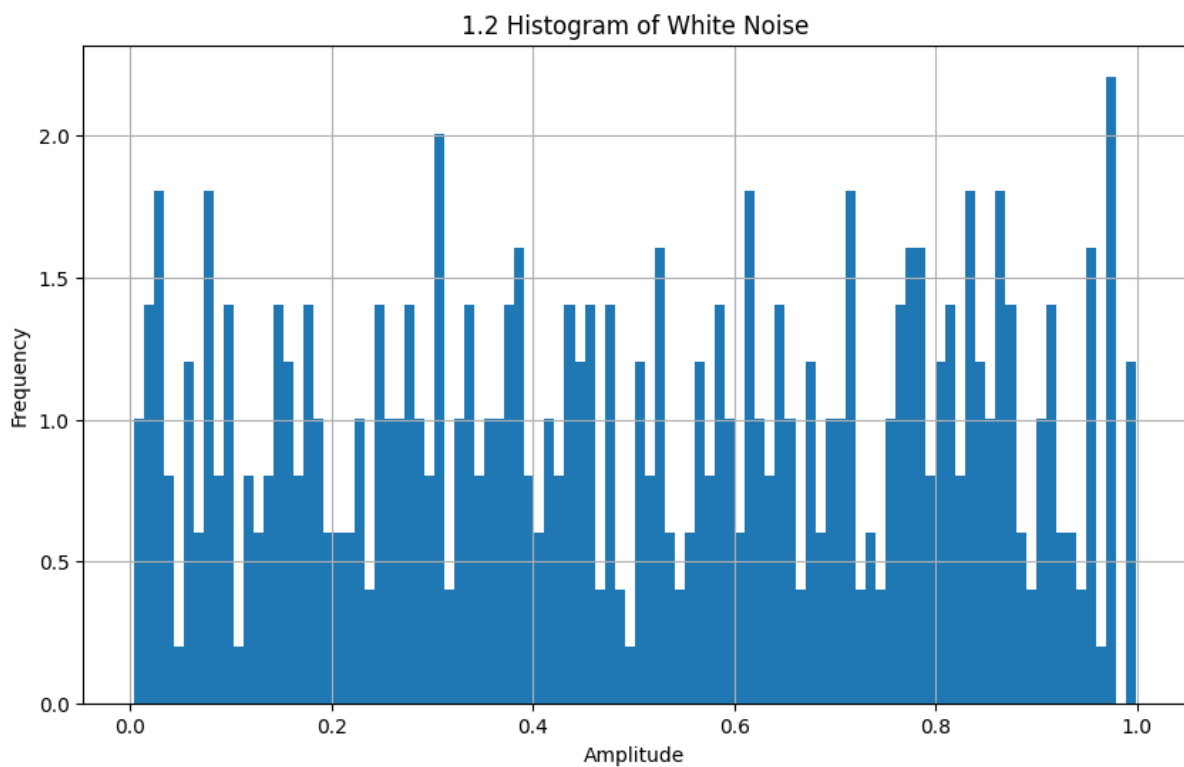


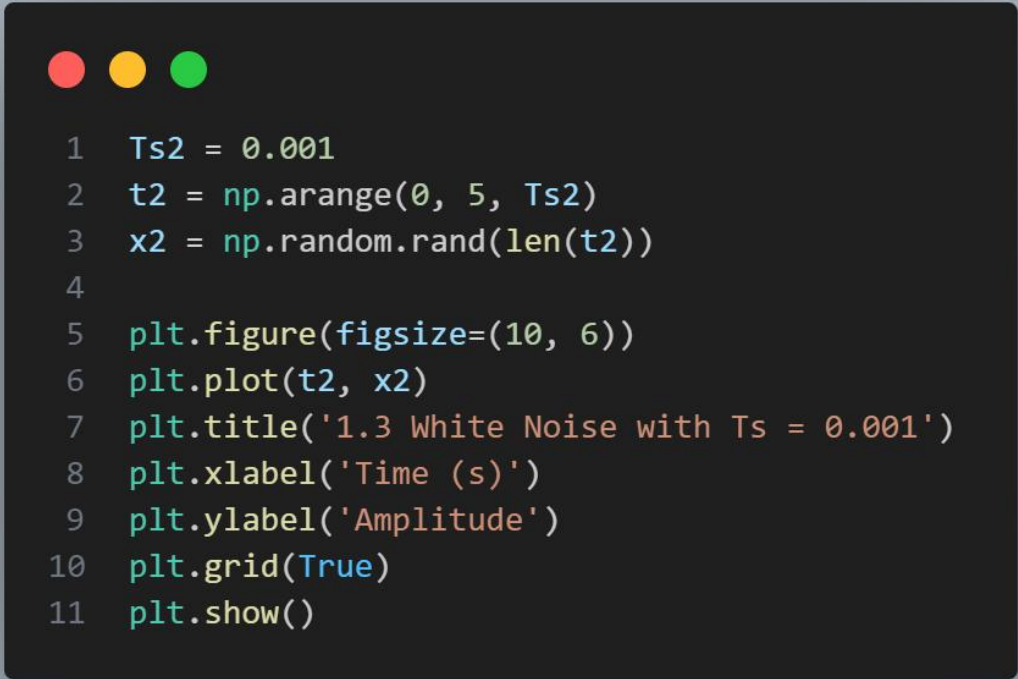
Figure 1.4: Result for task 1.2

`plt.hist(x1, bins=100, density=True)` creates a histogram with 100 bins. The `density=True` parameter normalizes the histogram, so the area under the histogram sums to 1, turning the count into a probability density function.

The resulting histogram shows the distribution of the amplitudes of the white noise signal. The x-axis represents amplitude values ranging from 0 to 1, and the y-axis represents the frequency (normalized) of these amplitude values.

The histogram reveals that the white noise has a relatively uniform distribution of amplitudes between 0 and 1. The bars of the histogram are fairly evenly distributed, indicating that the white noise is uniformly random. This uniformity is a characteristic property of white noise, where each amplitude value within the specified range is equally likely to occur. The grid lines make it easier to visualize the distribution pattern.

1.3.Repeat Step 1.1 for $T_s = 0.001$ and Generate a New Noise Signal x_2 .



```
1  Ts2 = 0.001
2  t2 = np.arange(0, 5, Ts2)
3  x2 = np.random.rand(len(t2))
4
5  plt.figure(figsize=(10, 6))
6  plt.plot(t2, x2)
7  plt.title('1.3 White Noise with Ts = 0.001')
8  plt.xlabel('Time (s)')
9  plt.ylabel('Amplitude')
10 plt.grid(True)
11 plt.show()
```

Figure 1.5: Code for task 1.3

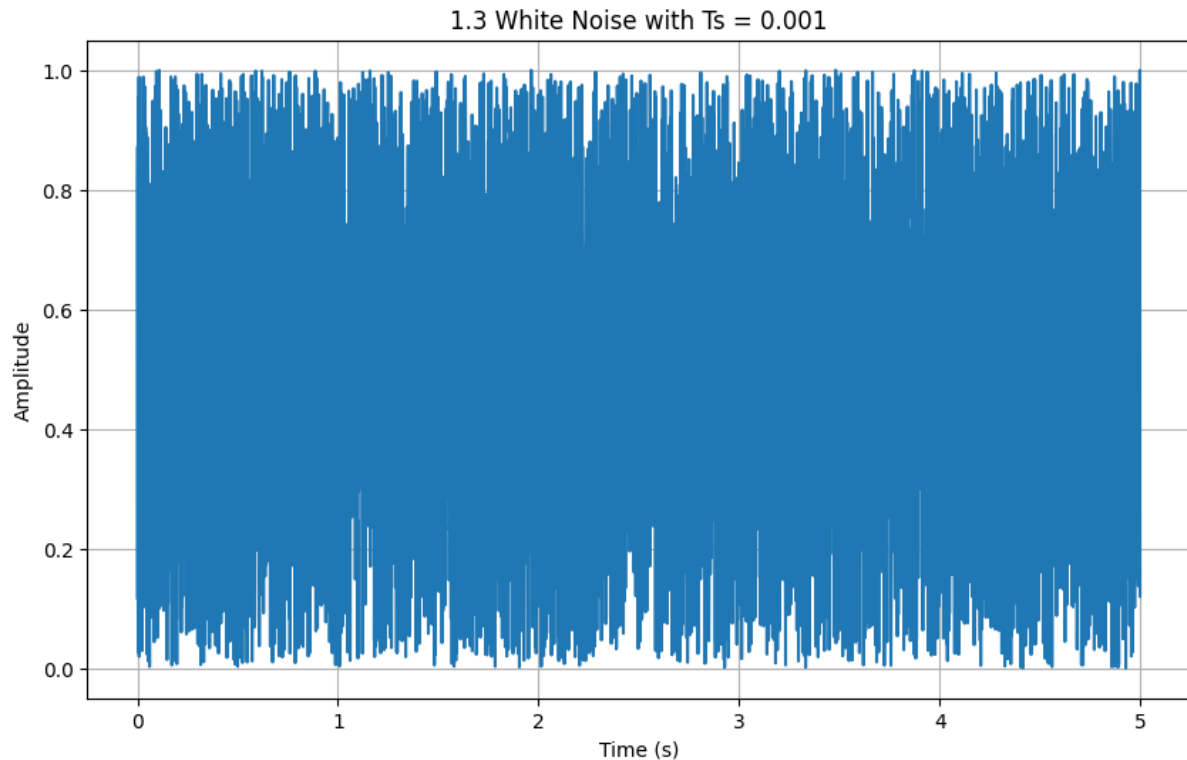


Figure 1.6: Result for task for 1.3

Parameters:

- $Ts2 = 0.001$ sets the sampling interval to 0.001 seconds, which is ten times smaller than the previous sampling interval.
- $t2 = np.arange(0, 5, Ts2)$ creates an array of time values ranging from 0 to 5 seconds, with a step of 0.001 seconds.

$x2 = np.random.rand(len(t2))$ generates an array of random values (white noise) with the same length as the time array $t2$. The values are uniformly distributed between 0 and 1

The resulting plot shows a much denser distribution of amplitude values over time, which is characteristic of white noise. The x-axis represents time in seconds (0 to 5), and the y-axis represents amplitude (0 to 1).

The plot demonstrates a random signal fluctuating rapidly over time with a higher resolution due to the smaller sampling interval. This higher resolution allows for more detailed analysis of the white noise properties. The increased density of data points shows a continuous and smooth pattern of randomness.

1.4. Represent the Histogram of the Generated Noise $x2$ from Step 1.3.



```
1 plt.figure(figsize=(10, 6))
2 plt.hist(x2, bins=500, density=True)
3 plt.title('1.4 Histogram of White Noise with Ts = 0.001')
4 plt.xlabel('Amplitude')
5 plt.ylabel('Frequency')
6 plt.grid(True)
7 plt.show()
```

Figure 1.7: Code for task 1.4

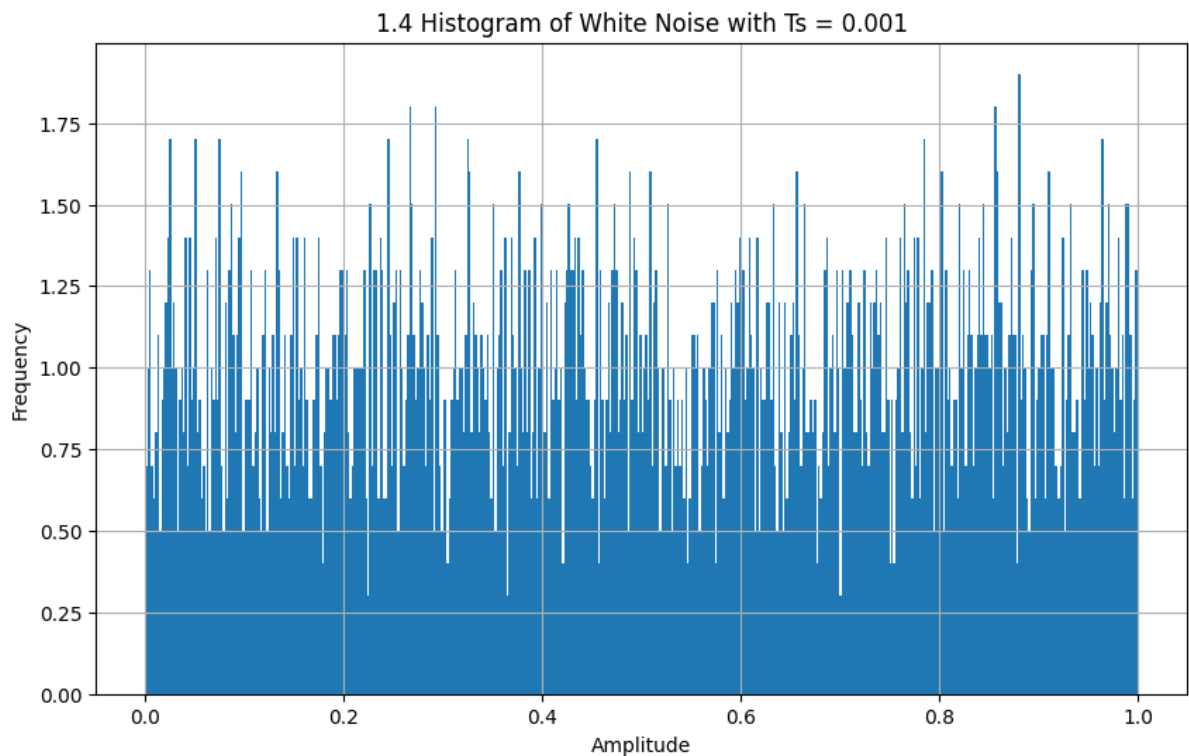


Figure 1.8: Result for task 1.4

The histogram reveals that the white noise has a relatively uniform distribution of amplitudes between 0 and 1. The bars are fairly evenly distributed, indicating that the white noise is uniformly random, a characteristic property of white noise. The normalized density allows for a clear visualization of the distribution pattern.

The higher number of bins (500) provides a more detailed view of the distribution, giving a smoother representation of the underlying random process. This detailed histogram helps in understanding the statistical properties of the white noise generated with the higher sampling rate

1.5.Design a Second-Order Digital Filter with a Natural Frequency of 1 Hz, Apply It to Signal x1, and Display the Output Signal.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.signal import lfilter
4
5 # Define parameters
6 Ts = 0.01 # Sampling time
7 om0 = 2 * np.pi # Natural frequency in rad/s
8 dz = 0.005 # Damping factor
9 A = 1 # Amplitude
10 oms = om0 * Ts # Normalized frequency
11
12 # Filter coefficients
13 a = [1 + 2 * dz * oms + oms**2, -2 * (1 + dz * oms), 1]
14 b = [A * 2 * oms**2]
15
16 # Generate time vector
17 t = np.arange(0, 50, Ts)
18
19 # Generate white noise signal x1
20 x1 = np.random.rand(len(t))
21
22 # Apply the filter
23 y1 = lfilter(b, a, x1)
24
25 # Plot the filtered signal
26 plt.figure(figsize=(10, 5))
27 plt.plot(t, y1, label='Filtered Signal (y1)')
28 plt.grid(True)
29 plt.xlabel('Time (s)')
30 plt.ylabel('Function y(t)')
31 plt.title('Filtering Noise with a Second-Order Filter')
32 plt.legend()
33 plt.show()
34
```

Figure 1.9: Code for task 1.5

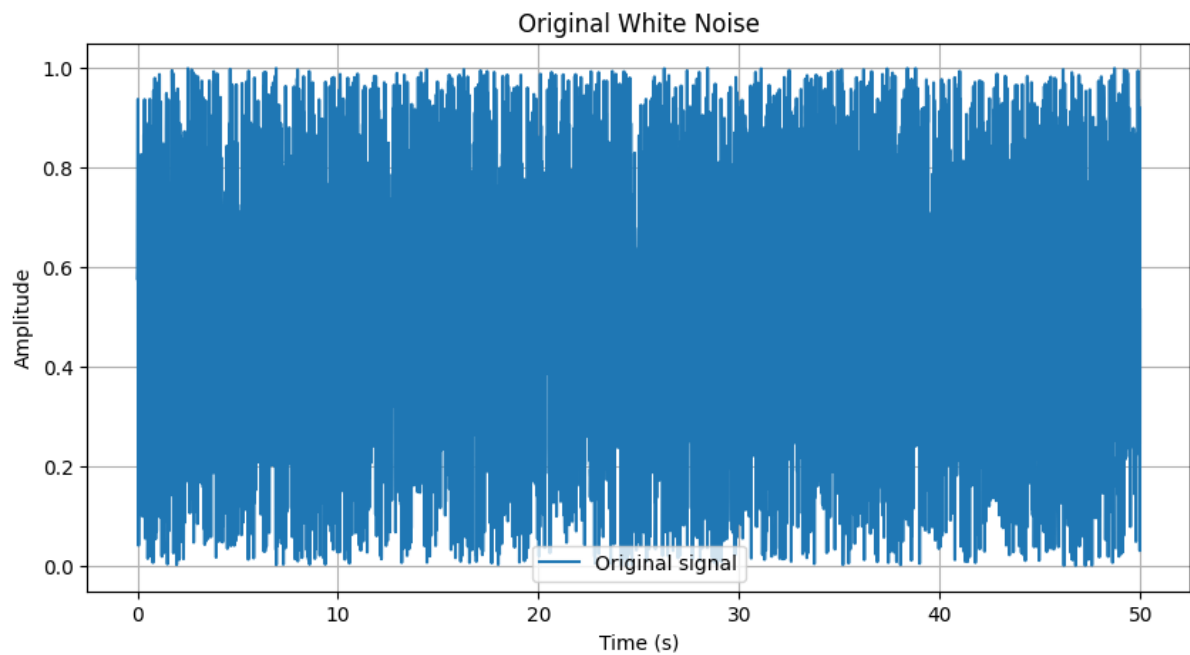


Figure 1.10: Original White Noise for task 1.5

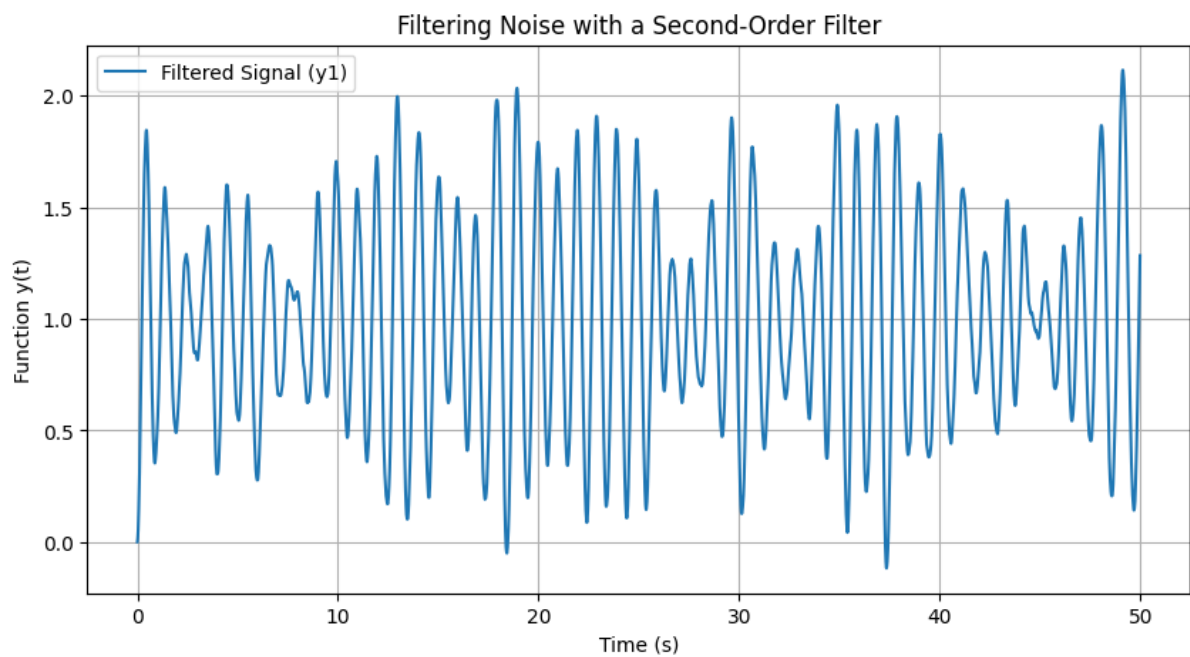


Figure 1.11: Filtered result for task 1.5

The filtered signal (y_1) is smoother than the original white noise signal. This indicates that the filter is effectively reducing the random fluctuations. The amplitude variations are more controlled and less erratic compared to the unfiltered noise. This suggests the filter is attenuating the noise.

The plot effectively illustrates how a second-order digital filter processes white noise, resulting in a smoother and more stable signal.

1.6.Repeat p 1.5 for $T_s = 0.001$ and generated noise x_2

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.signal import lfilter
4
5 # Define parameters
6 Ts = 0.001 # Sampling time
7 om0 = 2 * np.pi # Natural frequency in rad/s
8 dz = 0.005 # Damping factor
9 A = 1 # Amplitude
10 oms = om0 * Ts # Normalized frequency
11
12 # Filter coefficients
13 a = [1 + 2 * dz * oms + oms**2, -2 * (1 + dz * oms), 1]
14 b = [A * 2 * oms**2]
15
16 # Generate time vector
17 t = np.arange(0, 50, Ts)
18
19 # Generate white noise signal x1
20 x1 = np.random.rand(len(t))
21 plt.figure(figsize=(10, 5))
22 plt.plot(t, x1, label='Original signal')
23 plt.grid(True)
24 plt.xlabel('Time (s)')
25 plt.ylabel('Amplitude')
26 plt.title('Original White Noise')
27 plt.legend()
28 plt.show()
29
30 # Apply the filter
31 y1 = lfilter(b, a, x1)
32
33 # Plot the filtered signal
34 plt.figure(figsize=(10, 5))
35 plt.plot(t, y1, label='Filtered Signal (y1)')
36 plt.grid(True)
37 plt.xlabel('Time (s)')
38 plt.ylabel('Function y(t)')
39 plt.title('Filtering Noise with a Second-Order Filter')
40 plt.legend()
41 plt.show()
```

Figure 1.12: Code for task 1.6

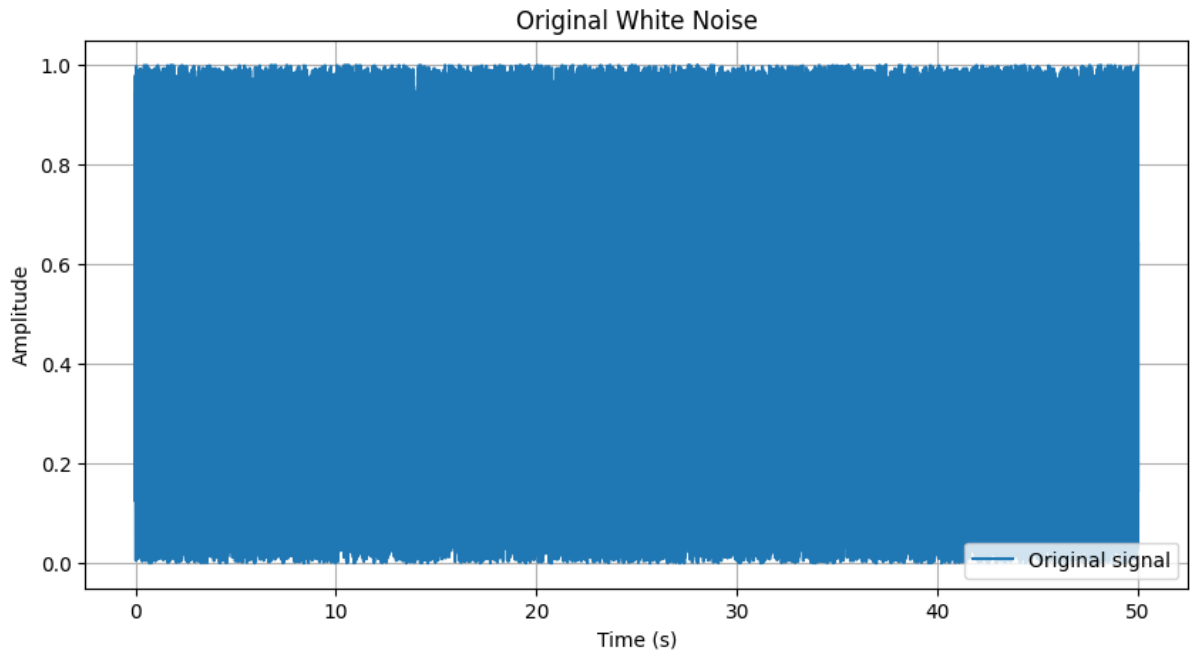


Figure 1.13: Original White Noise for task 1.6

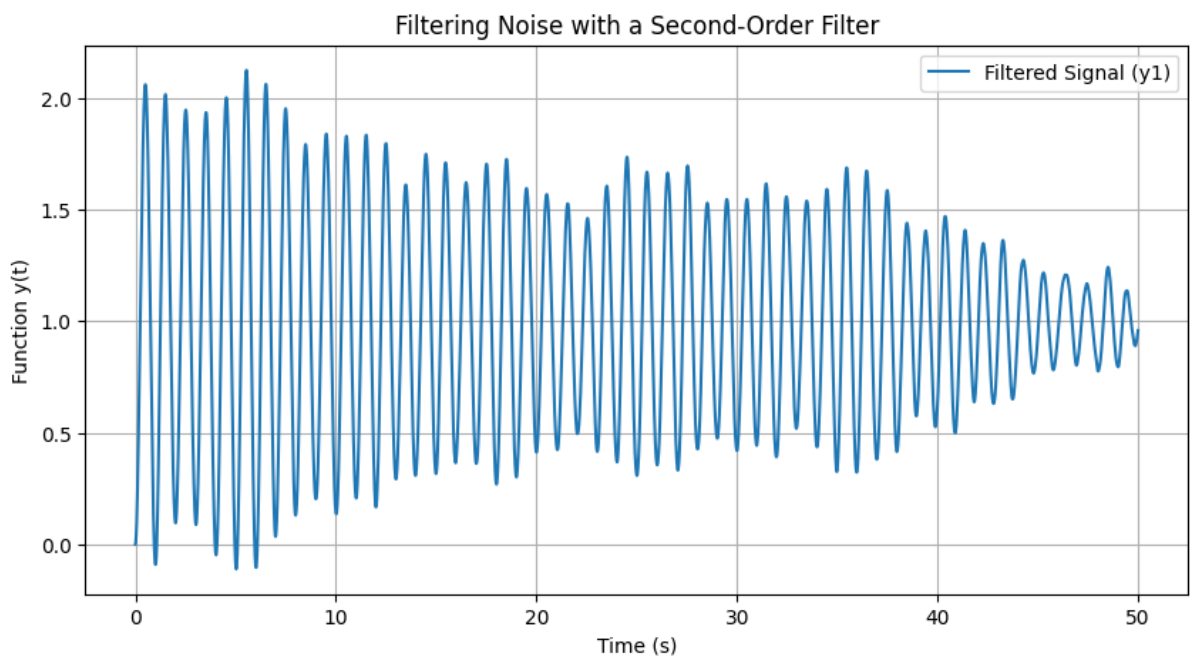


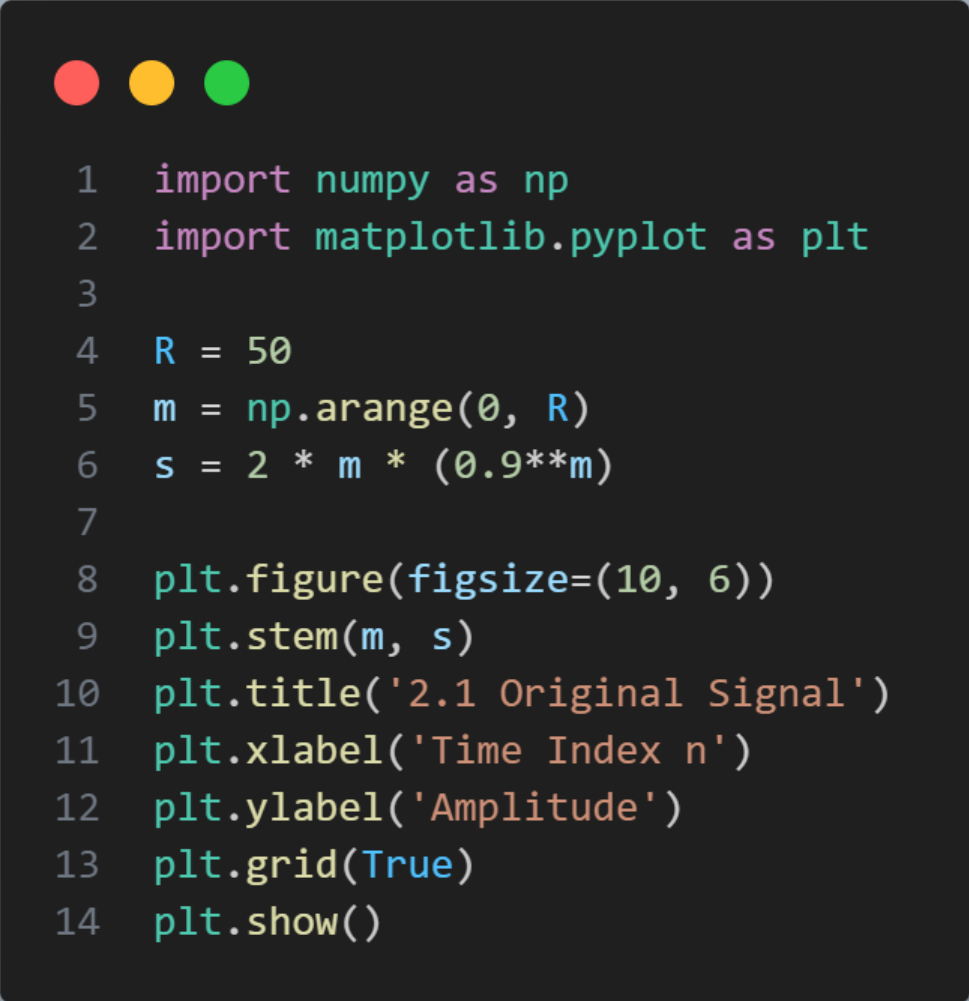
Figure 1.14: Filtered result for task 1.6

The filtered signal is significantly smoother than the original white noise. The filter reduces the random fluctuations, resulting in a more stable signal. The amplitude variations are more controlled, and the signal appears less erratic compared to the original noise. The grid and legend provide a clear visualization of the filtered signal.

The filtered noise plot demonstrates the effectiveness of the second-order filter in smoothing and attenuating the noise, producing a signal with reduced amplitude variations over time. This illustrates the filter's capability to enhance signal quality by reducing noise.

2. Filtering noise-affected signals using a MAF filter

2.1. Generate an original signal not affected by noise $s(m)$



```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  R = 50
5  m = np.arange(0, R)
6  s = 2 * m * (0.9**m)
7
8  plt.figure(figsize=(10, 6))
9  plt.stem(m, s)
10 plt.title('2.1 Original Signal')
11 plt.xlabel('Time Index n')
12 plt.ylabel('Amplitude')
13 plt.grid(True)
14 plt.show()
```

Figure 2.1: Code for task 2.1

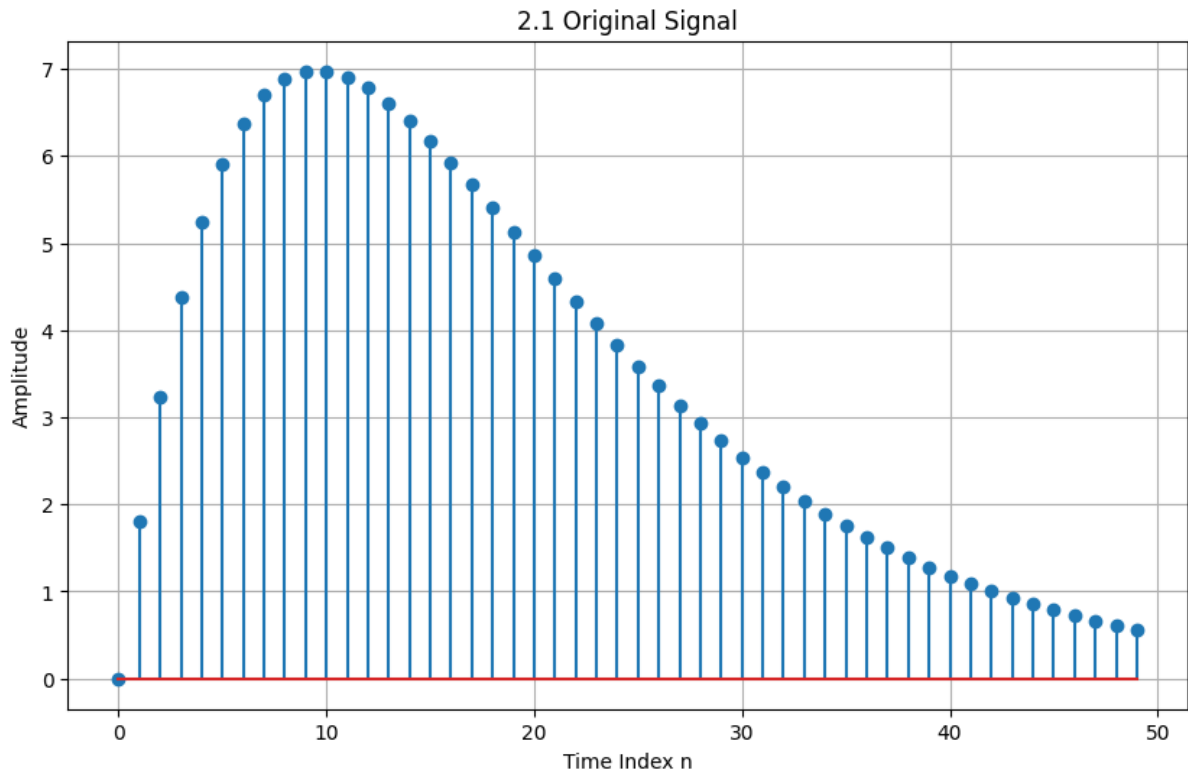


Figure 2.2: Result doe task 2.2

- The x-axis represents the **time index** m , ranging from 0 to 49.
- The y-axis represents the **amplitude** of the signal s .
- **Initial Increase:** The amplitude starts at 0 for $m=0$ and increases as m increases.
- **Peak Amplitude:** The signal reaches its peak amplitude around $m=10$.
- **Exponential Decay:** After reaching the peak, the amplitude gradually decreases due to the exponential decay factor (0.9^m) .

2.2. Generate a noise, using the rand function by adding to point 2.1 the noise $d=\text{rand}(1,\text{length}(m))-0.5$.



```
1 d = np.random.rand(len(m)) - 0.5
2 plt.figure(figsize=(10, 6))
3 plt.plot(m, d, label='Random Signal')
4 plt.title('2.2 Random Signals')
5 plt.xlabel('Time Index n')
6 plt.ylabel('Amplitude')
7 plt.legend()
8 plt.grid(True)
9 plt.show()
```

Figure 2.3: Code for task 2.2

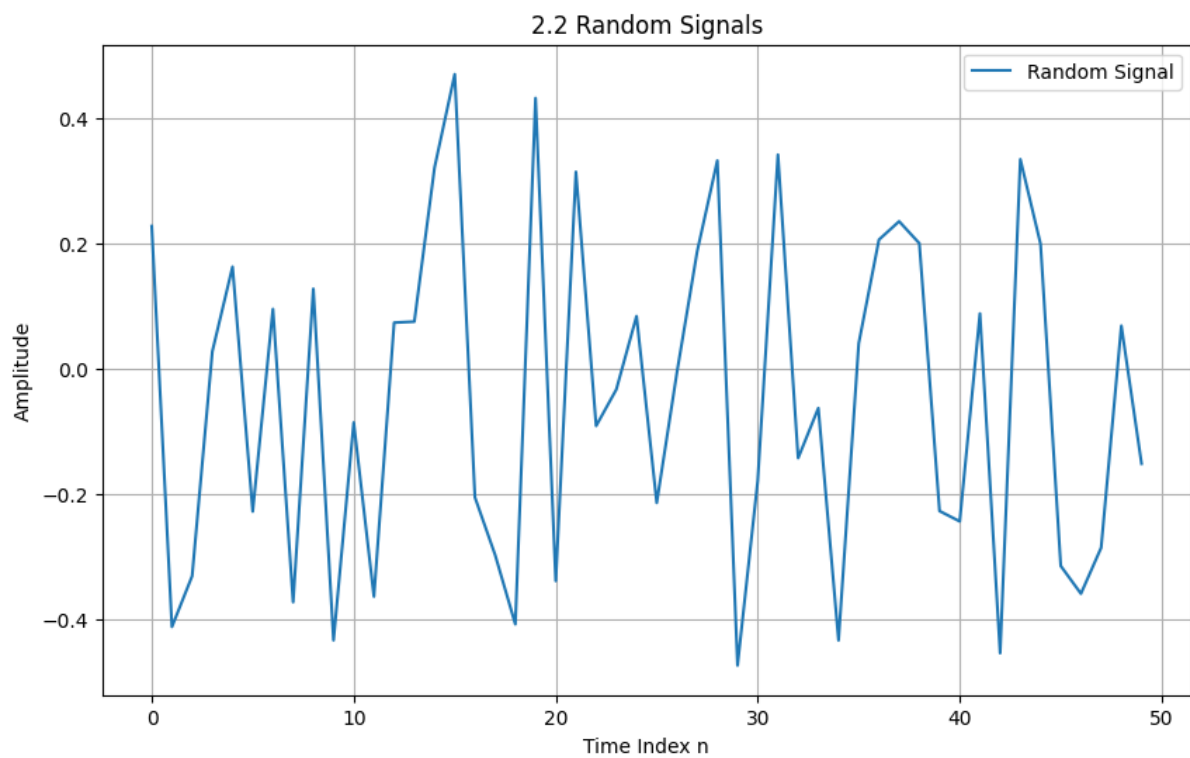
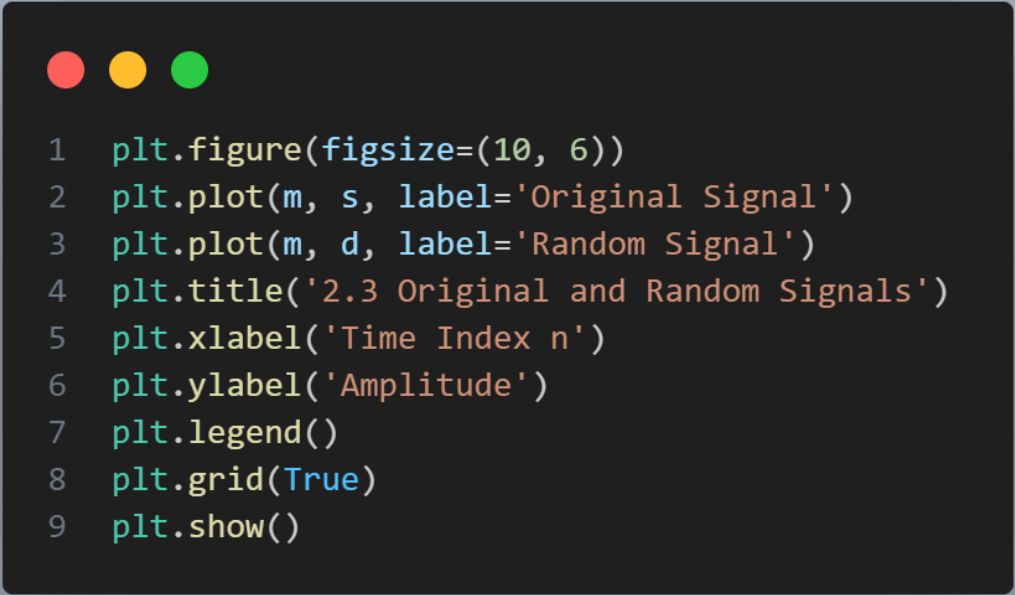


Figure 2.4: Result for task 2.2

The plot is a visual representation of a random signal, showcasing how the amplitude varies over different time indices.

- 2.3. Represent both of these signals in continuous form on a single graph, using the plot function.



```
1 plt.figure(figsize=(10, 6))
2 plt.plot(m, s, label='Original Signal')
3 plt.plot(m, d, label='Random Signal')
4 plt.title('2.3 Original and Random Signals')
5 plt.xlabel('Time Index n')
6 plt.ylabel('Amplitude')
7 plt.legend()
8 plt.grid(True)
9 plt.show()
```

Figure 2.5: Code for task 2.3

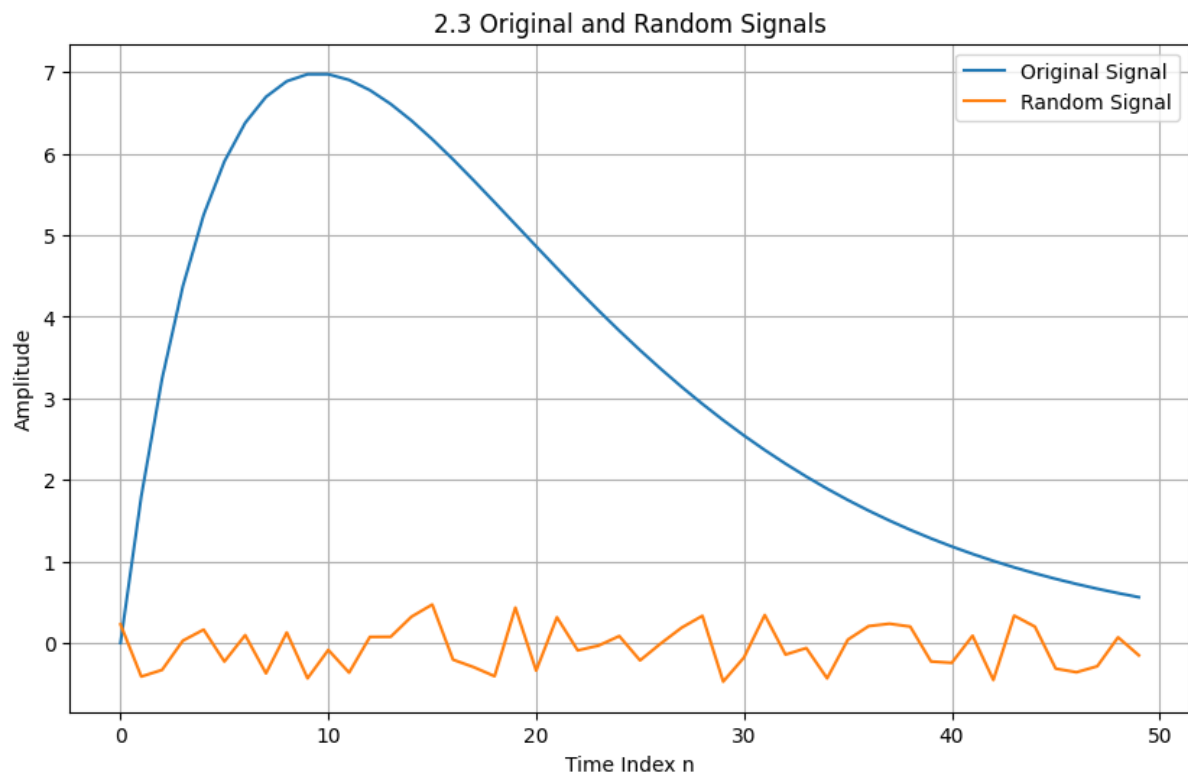


Figure 2.6: Result for task 2.3

The graph visually compares the behaviour of an original signal with a random signal over time, highlighting the differences in their patterns and amplitudes. The original signal shows a clear, structured rise and fall pattern, while the random signal displays irregular and unpredictable fluctuations.

- 2.4. Represent the sum of these two signals $x = s + d$ and plot the resulting signal x and the initial signal s on a single graph, using the plot function.



```
1  x = s + d
2
3  plt.figure(figsize=(10, 6))
4  plt.plot(m, s, label='Original Signal')
5  plt.plot(m, x, label='Noisy Signal')
6  plt.title('2.4 Original and Noisy Signals')
7  plt.xlabel('Time Index n')
8  plt.ylabel('Amplitude')
9  plt.legend()
10 plt.grid(True)
11 plt.show()
```

Figure 2.7: Code for task 2.4

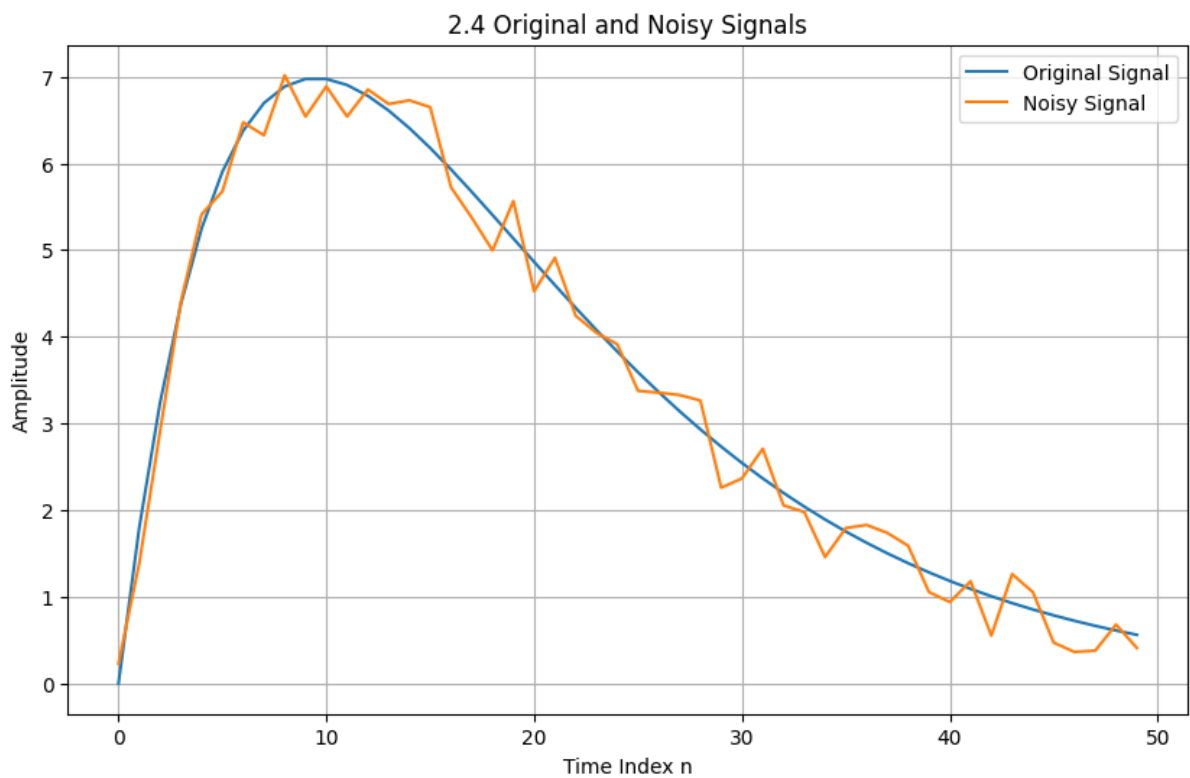



Figure 2.8: Result for task 2.4

Original Signal (Blue Line): This line represents the original signal without any noise. It has a smooth, sinusoidal shape that increases initially, reaches a peak, and then decreases gradually over time.

Noisy Signal (Orange Line): This line represents the same signal but with added noise. The noisy signal fluctuates around the original signal, showing irregular and random variations due to the noise. The overall shape follows the original signal's trend, but with added randomness.

This visual comparison helps in understanding the impact of noise on signal processing and the importance of filtering or noise reduction techniques.

2.5. Design a moving average filter (MAF) with the parameters $y = \text{filter}(b, 1, x)$, $b = \text{ones}(M, 1)/M$, and specify $M = 3$ in advance. Filter the signal affected by noise. Plot the filtered signal y , the noisy signal x , and the initial signal s on a single graph, using the plot function



```
1  from scipy.signal import lfilter
2
3  def moving_average_filter(signal, M):
4      b = np.ones(M) / M
5      return lfilter(b, 1, signal)
6
7  # Apply MAF with M=3
8  M = 3
9  y = moving_average_filter(x, M)
10
11 plt.figure(figsize=(10, 6))
12 plt.plot(m, s, label='Original Signal')
13 plt.plot(m, x, label='Noisy Signal')
14 plt.plot(m, y, label='Filtered Signal (M=3)')
15 plt.title('2.5 Original, Noisy, and Filtered Signals')
16 plt.xlabel('Time Index n')
17 plt.ylabel('Amplitude')
18 plt.legend()
19 plt.grid(True)
20 plt.show()
```

Figure 2.9: Code for task 2.5

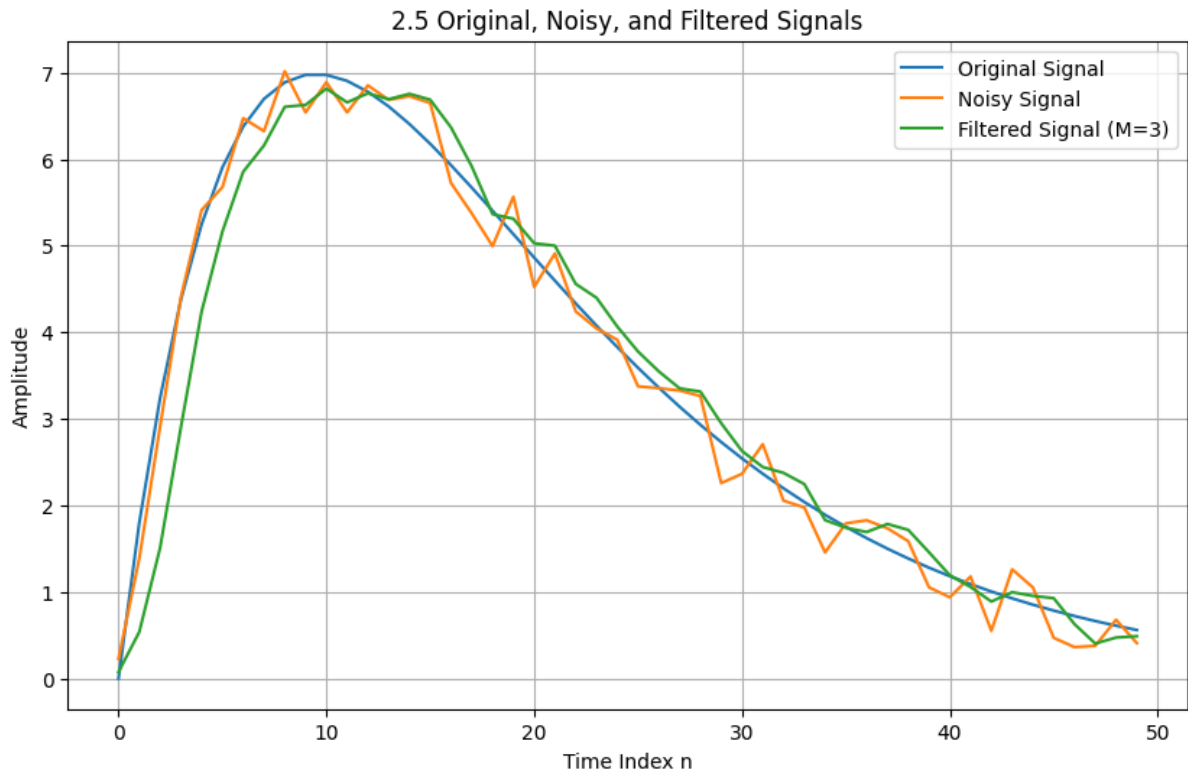


Figure 2.10: Result for task 2.5

Original Signal (Blue Line): Displays the original data points without any noise.

Noisy Signal (Orange Line): Shows the original signal with added noise, leading to fluctuations and irregularities.

Filtered Signal (Green Line): Demonstrates the effect of applying a moving average filter, which smooths out the noise, making the signal resemble the original one more closely.

The plot demonstrates the effectiveness of the moving average filter in reducing noise while preserving the overall shape of the original signal. This is relevant for signal processing applications where noise reduction is essential for accurate data analysis and interpretation.

2.6. Repeat step 2.5 for $M = 5$ and $M = 10$. Compare the results obtained.

```
1  # M=5
2  M = 5
3  y = moving_average_filter(x, M)
4
5  plt.figure(figsize=(10, 6))
6  plt.plot(m, s, label='Original Signal')
7  plt.plot(m, x, label='Noisy Signal')
8  plt.plot(m, y, label='Filtered Signal (M=5)')
9  plt.title('2.6.1 Original, Noisy, and Filtered Signals')
10 plt.xlabel('Time Index n')
11 plt.ylabel('Amplitude')
12 plt.legend()
13 plt.grid(True)
14 plt.show()
15
16 # M=10
17 M = 10
18 y = moving_average_filter(x, M)
19
20 plt.figure(figsize=(10, 6))
21 plt.plot(m, s, label='Original Signal')
22 plt.plot(m, x, label='Noisy Signal')
23 plt.plot(m, y, label='Filtered Signal (M=10)')
24 plt.title('2.6.2 Original, Noisy, and Filtered Signals')
25 plt.xlabel('Time Index n')
26 plt.ylabel('Amplitude')
27 plt.legend()
28 plt.grid(True)
29 plt.show()
```

Figure 2.11: Code for task 2.6

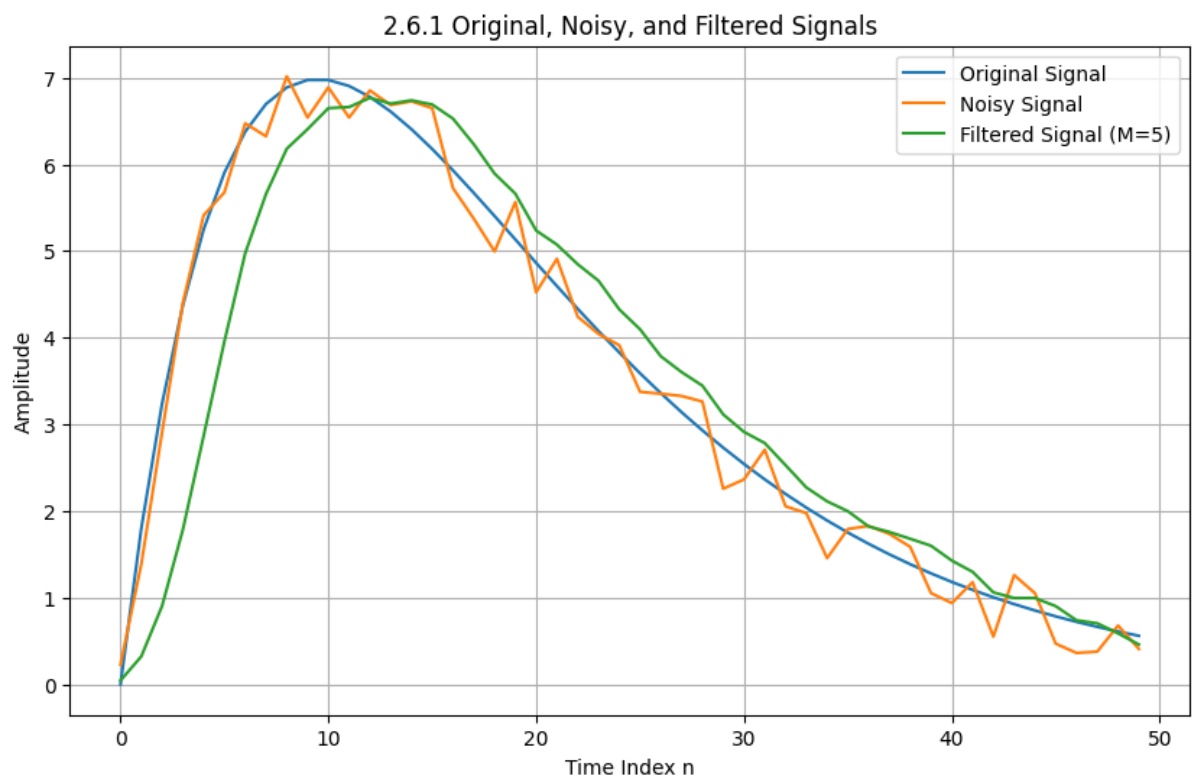


Figure 2.12.1: Result for task 2.6 with $M = 5$

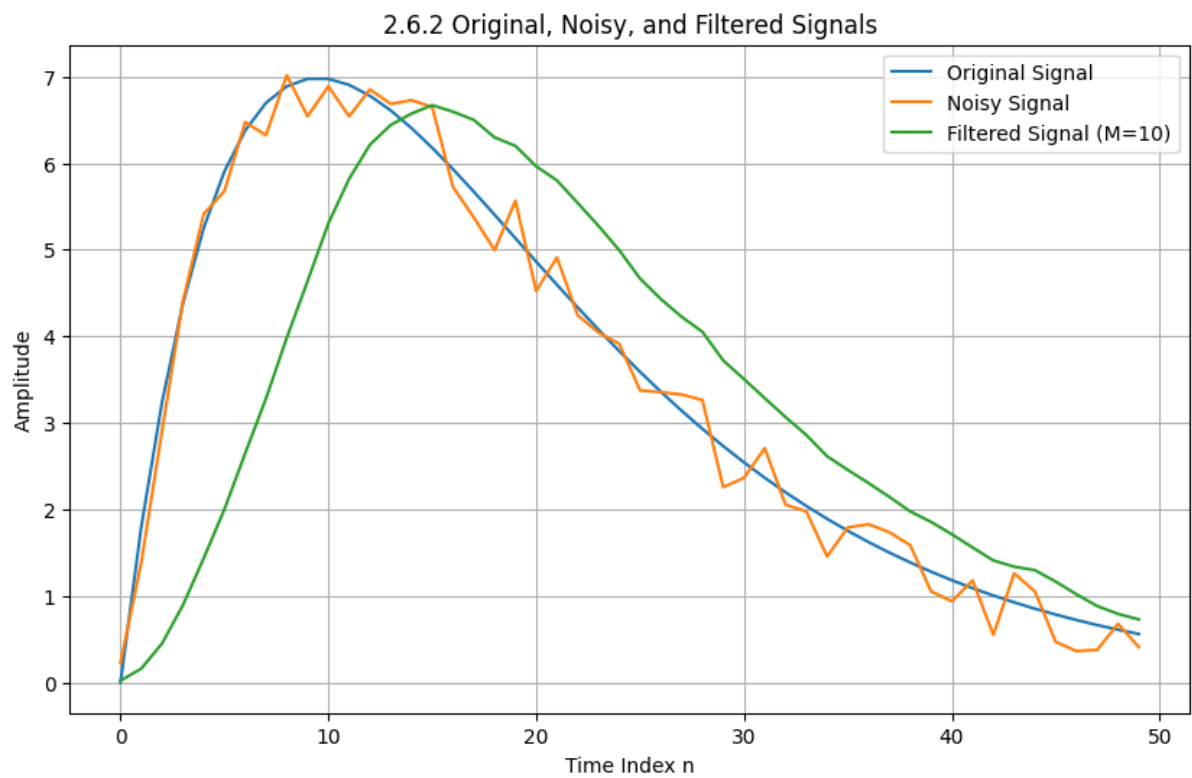


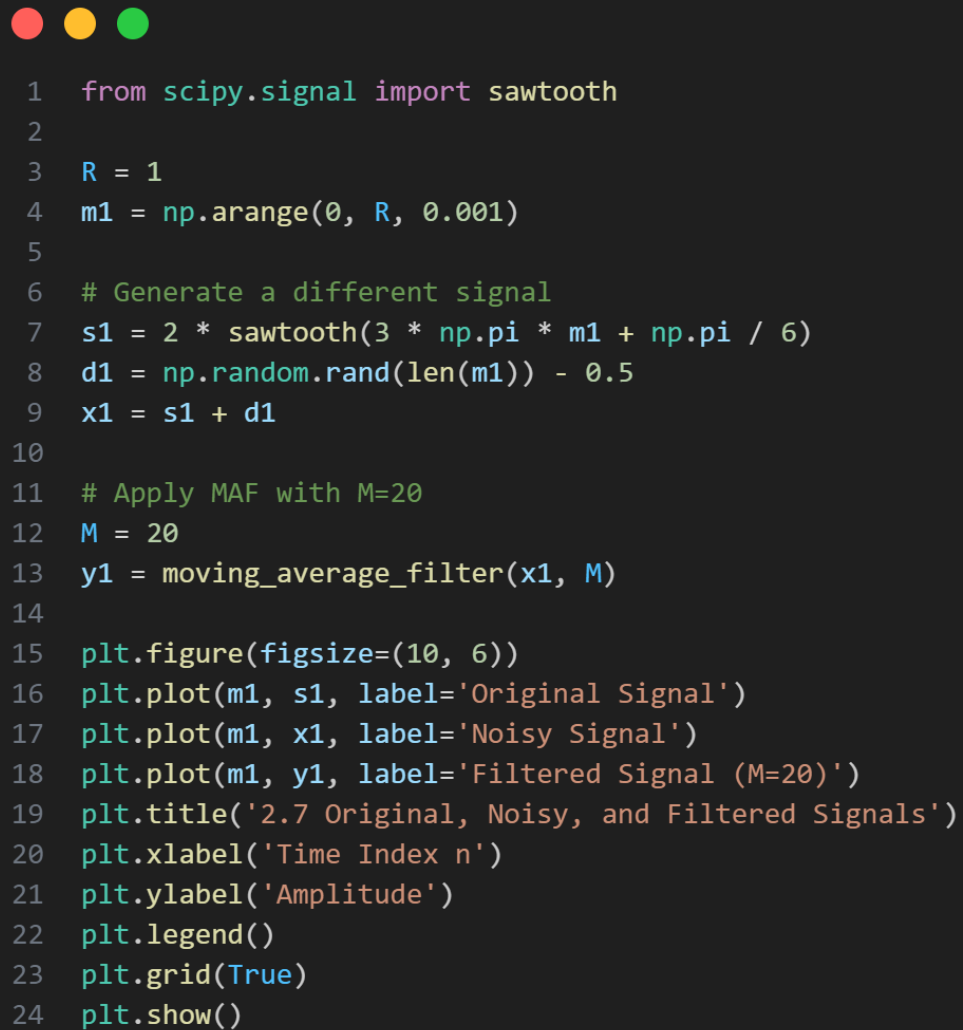
Figure 2.12.2: Result for task 2.6 with $M = 10$

Increasing the value of M from 5 to 10 in the moving average filter results in a smoother filtered signal, reducing more noise and better preserving the original signal's shape.

The plot with $M=10$ demonstrates better noise reduction compared to the plot with $M=5$, showing the trade-off between the filter window size and the smoothness of the signal.

These plots effectively illustrate the impact of different moving average filter sizes on a noisy signal, highlighting the benefits of using a larger filter window for more effective noise reduction.

2.7. Repeat step 2.5 for another signal – $s = 2 * \text{sawtooth}(3 * \pi * m + \pi / 6)$ and change the time step to a smaller one – $m = 0:0.001:R$ ($R = 1$) and set $M = 20$. Plot the filtered signal y , the noisy signal x , and the initial signal s on a single graph, using the plot function.



```
1  from scipy.signal import sawtooth
2
3  R = 1
4  m1 = np.arange(0, R, 0.001)
5
6  # Generate a different signal
7  s1 = 2 * sawtooth(3 * np.pi * m1 + np.pi / 6)
8  d1 = np.random.rand(len(m1)) - 0.5
9  x1 = s1 + d1
10
11 # Apply MAF with M=20
12 M = 20
13 y1 = moving_average_filter(x1, M)
14
15 plt.figure(figsize=(10, 6))
16 plt.plot(m1, s1, label='Original Signal')
17 plt.plot(m1, x1, label='Noisy Signal')
18 plt.plot(m1, y1, label='Filtered Signal (M=20)')
19 plt.title('2.7 Original, Noisy, and Filtered Signals')
20 plt.xlabel('Time Index n')
21 plt.ylabel('Amplitude')
22 plt.legend()
23 plt.grid(True)
24 plt.show()
```

Figure 2.13: Code for task 2.7

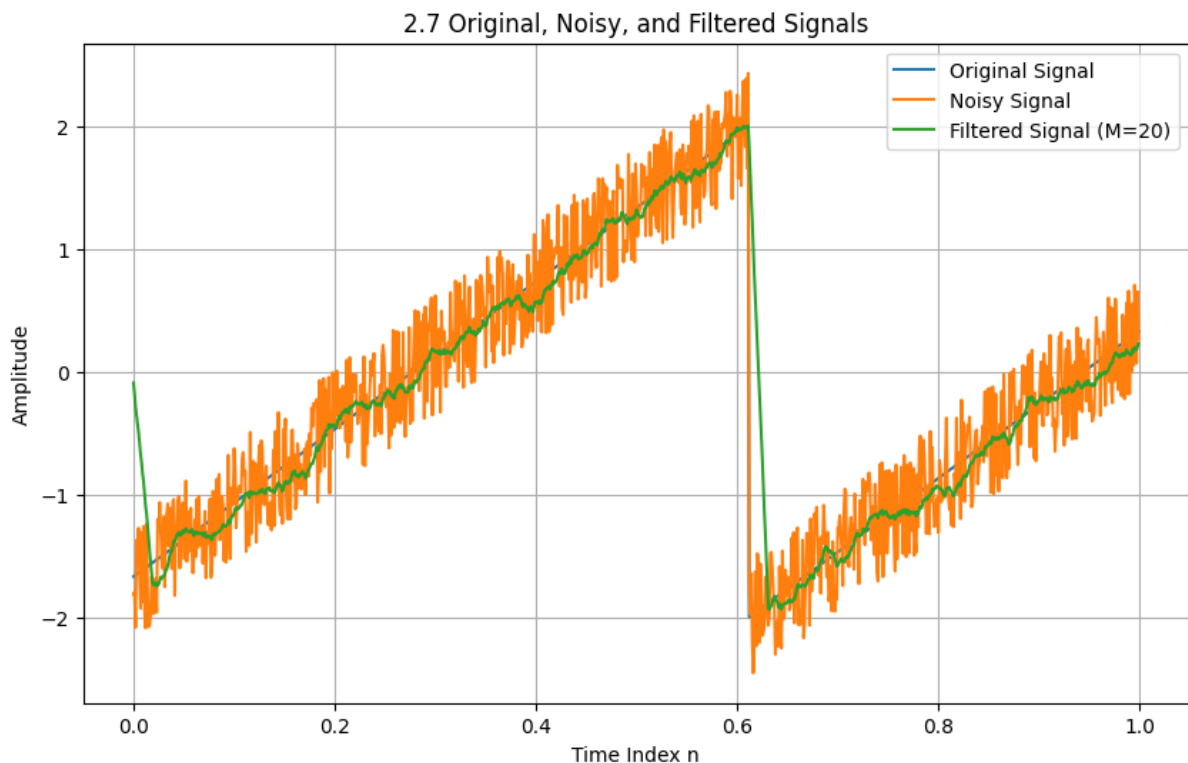


Figure 2.14: Result for task 2.7

Original Signal (Blue Line):

- The original signal is a sawtooth wave, generated using the sawtooth function from the scipy.signal module.
- It exhibits a linear rise and a sharp fall, characteristic of a sawtooth waveform.

Noisy Signal (Orange Line):

- The original sawtooth wave with added random noise.
- The noise causes irregular fluctuations, making the signal less smooth.

Filtered Signal (Green Line):

- This is the result of applying a Moving Average Filter with a window size of $M=20$ to the noisy signal.
- The filter smooths out the noise, making the signal closely resemble the original sawtooth wave.

This plot visually demonstrates the effectiveness of a Moving Average Filter in reducing noise and preserving the underlying structure of the original signal.

By averaging the data points within a window size of $M=20$, the filter smooths out random fluctuations and enhances the clarity of the signal.

2.8. Repeat step 2.7 for $M = 50$ and $M = 100$. Compare the results obtained.

```
1  # M=50
2  M = 50
3  y1 = moving_average_filter(x1, M)
4
5  plt.figure(figsize=(10, 6))
6  plt.plot(m1, s1, label='Original Signal')
7  plt.plot(m1, x1, label='Noisy Signal')
8  plt.plot(m1, y1, label='Filtered Signal (M=50)')
9  plt.title('Original, Noisy, and Filtered Signals')
10 plt.xlabel('Time Index n')
11 plt.ylabel('Amplitude')
12 plt.legend()
13 plt.grid(True)
14 plt.show()
15
16 # M=100
17 M = 100
18 y1 = moving_average_filter(x1, M)
19
20 plt.figure(figsize=(10, 6))
21 plt.plot(m1, s1, label='Original Signal')
22 plt.plot(m1, x1, label='Noisy Signal')
23 plt.plot(m1, y1, label='Filtered Signal (M=100)')
24 plt.title('Original, Noisy, and Filtered Signals')
25 plt.xlabel('Time Index n')
26 plt.ylabel('Amplitude')
27 plt.legend()
28 plt.grid(True)
29 plt.show()
```

Figure 2.15: Code for task 2.8

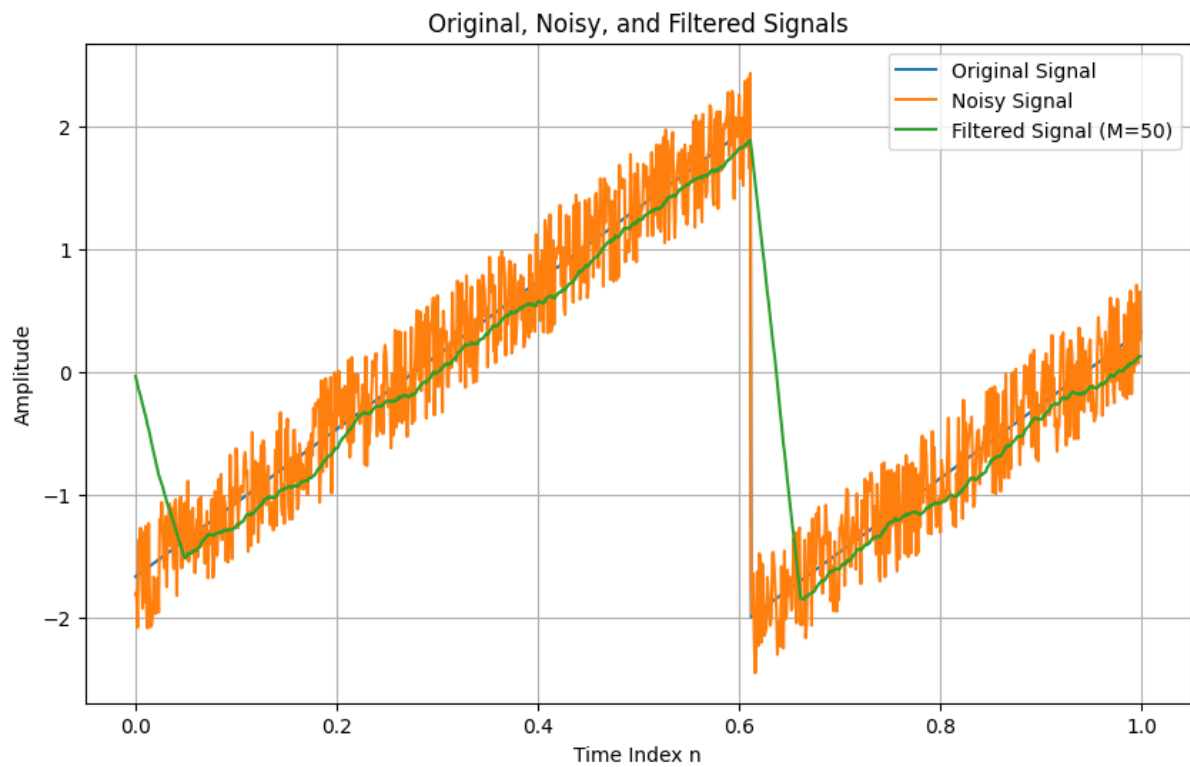


Figure 2.16.1: Result for task 2.8 for $M = 50$

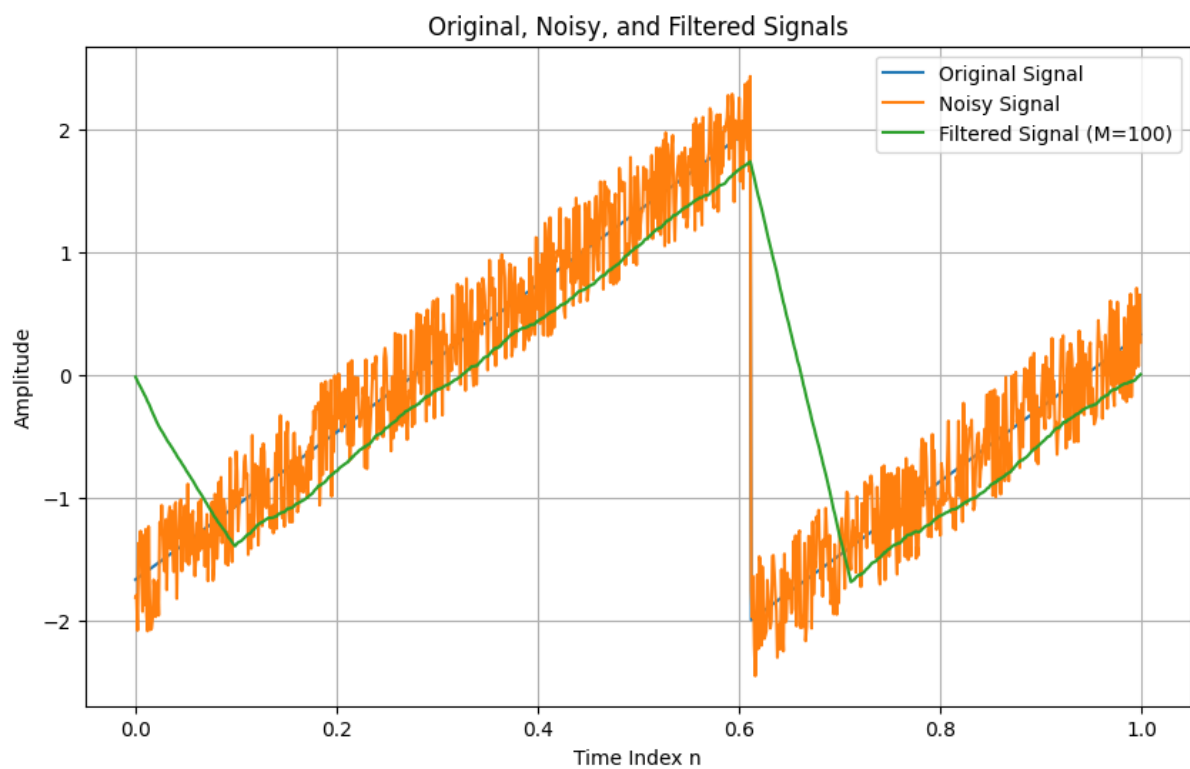


Figure 2.16.2: Result for task 2.8 for $M = 100$

Plot 1: Original, Noisy, and Filtered Signals (M=50)

- **Original Signal (Blue Line):** Represents the unfiltered sawtooth signal.
- **Noisy Signal (Orange Line):** Displays the sawtooth signal with added random noise.
- **Filtered Signal (Green Line, M=50):** Shows the result of applying a moving average filter with a window size of 50. The filtered signal is smoother than the noisy signal, with reduced random fluctuations, but some noise still persists.

Plot 2: Original, Noisy, and Filtered Signals (M=100)

- **Original Signal (Blue Line):** Again, represents the unfiltered sawtooth signal.
- **Noisy Signal (Orange Line):** Shows the noisy version of the original signal.
- **Filtered Signal (Green Line, M=100):** Shows the result of applying a moving average filter with a window size of 100. This filtered signal is significantly smoother compared to the one with M=50, closely resembling the original signal with minimal noise.

Increasing the window size M results in a smoother filtered signal. The larger the window size, the more points are averaged, which better reduces noise.

While larger M values yield smoother signals, they may also slightly lag behind rapid changes in the original signal due to the increased averaging.

3. Research the filtering processes using the LMS and RLS algorithm.

As a result of not having access to MATLAB and its applications / packages, for this task was used Python.

LMS Algorithm

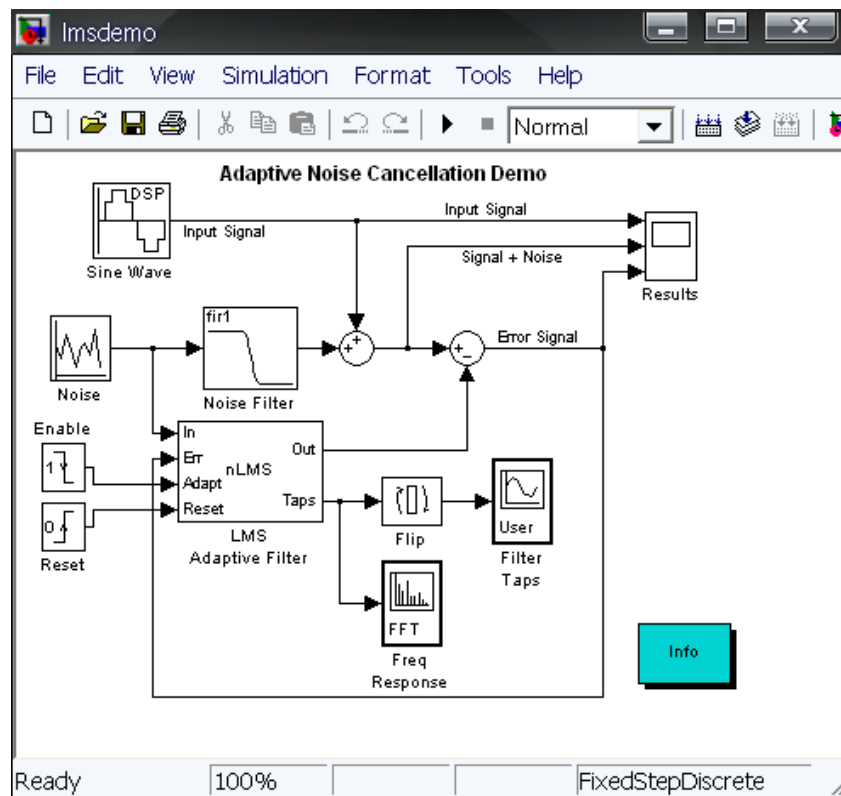


Figure 3.1: Block diagram of LMS algorithm

RLS Algorithm

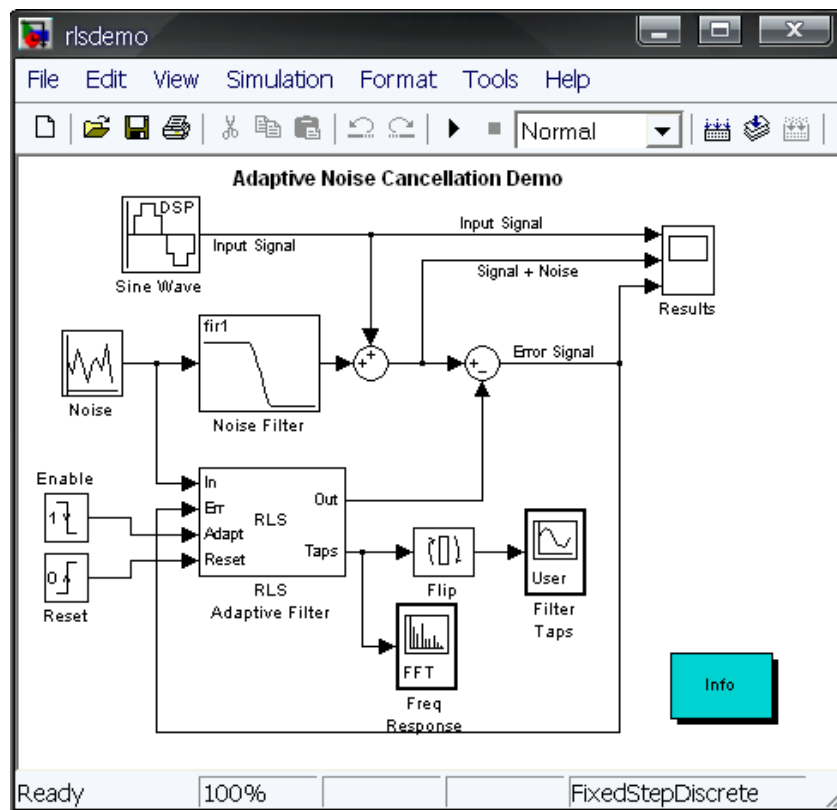


Figure 3.2: Block diagram for RLS Algorithm

Code:

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt

class LMSFilter:
    def __init__(self, filter_length, mu):
        self.L = filter_length
        self.mu = mu
        self.w = np.zeros(self.L)

    def filter(self, x, d):
        N = len(x)
        y = np.zeros(N)
        e = np.zeros(N)

        # Pad input for initial filter delay
        x_padded = np.pad(x, (self.L-1, 0), 'constant')

        for n in range(N):
            # Get current input window
            x_window = x_padded[n:n+self.L][::-1]
```



```

        # Calculate filter output
        y[n] = np.dot(self.w, x_window)

        # Calculate error
        e[n] = d[n] - y[n]

        # Update filter weights
        self.w = self.w + self.mu * e[n] * x_window

    return y, e

class RLSFilter:
    def __init__(self, filter_length, lambda_forget=0.99,
delta=0.1):
        self.L = filter_length
        self.lambda_forget = lambda_forget
        self.w = np.zeros(self.L)
        self.P = np.eye(self.L) / delta

    def filter(self, x, d):
        N = len(x)
        y = np.zeros(N)
        e = np.zeros(N)

        # Pad input for initial filter delay
        x_padded = np.pad(x, (self.L-1, 0), 'constant')

        for n in range(N):
            # Get current input window
            x_window = x_padded[n:n+self.L][::-1]

            # Calculate Kalman gain
            k = self.P @ x_window / (self.lambda_forget +
x_window.T @ self.P @ x_window)

            # Calculate filter output
            y[n] = np.dot(self.w, x_window)

            # Calculate error
            e[n] = d[n] - y[n]

            # Update filter weights
            self.w = self.w + k * e[n]

            # Update inverse correlation matrix
            self.P = (self.P - np.outer(k, x_window.T @ self.P))
/ self.lambda_forget

```

```

        return y, e

# Example usage and demonstration
def demo_noise_cancellation():
    # Generate sample signals
    t = np.linspace(0, 10, 600)
    desired_signal = np.sin(2 * np.pi * 10 * t) # 10 Hz sine
wave
    noise = 0.5 * np.random.randn(len(t))
    input_signal = desired_signal + noise

    # Apply LMS filter
    lms = LMSFilter(filter_length=32, mu=0.01)
    lms_output, lms_error = lms.filter(input_signal,
desired_signal)

    # Apply RLS filter
    rls = RLSFilter(filter_length=32)
    rls_output, rls_error = rls.filter(input_signal,
desired_signal)

    # Plot results
    plt.figure(figsize=(12, 8))

    plt.subplot(4, 1, 1)
    plt.plot(t, desired_signal)
    plt.title('Original Signal')
    plt.grid(True)

    plt.subplot(4, 1, 2)
    plt.plot(t, input_signal)
    plt.title('Noisy Signal')
    plt.grid(True)

    plt.subplot(4, 1, 3)
    plt.plot(t, lms_output)
    plt.title('LMS Filtered Signal')
    plt.grid(True)

    plt.subplot(4, 1, 4)
    plt.plot(t, rls_output)
    plt.title('RLS Filtered Signal')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":

```

```
demo_noise_cancellation()
```

Results

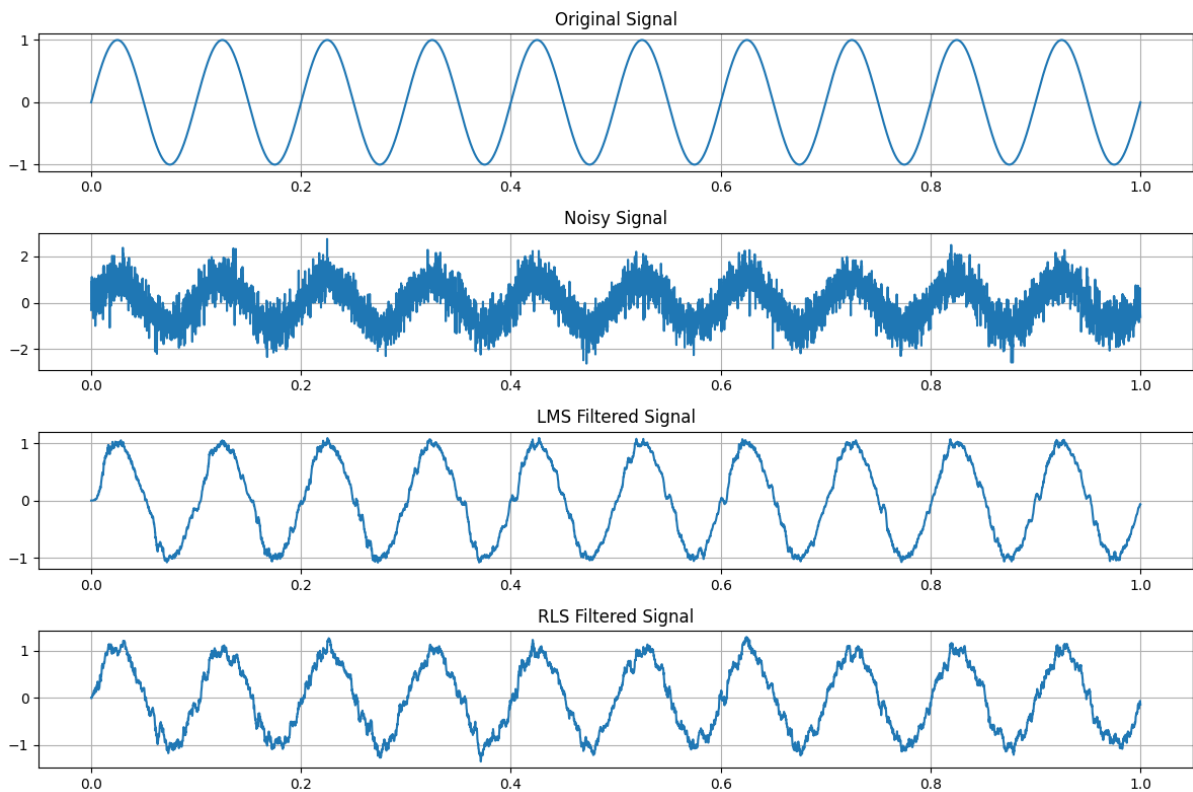


Figure 3.3: Results for $t = 1$ and filter length equal with 32

- **Original Signal:** This plot shows a clean sinusoidal wave oscillating between -1 and 1 over a time period from 0 to 1.
- **Noisy Signal:** This plot shows the same sinusoidal wave but with added noise, causing the signal to fluctuate more erratically around the original wave.
- **LMS Filtered Signal:** This plot shows the noisy signal after being processed by an LMS (Least Mean Squares) filter, which reduces the noise and makes the signal more closely resemble the original sinusoidal wave.
- **RLS Filtered Signal:** This plot shows the noisy signal after being processed by an RLS (Recursive Least Squares) filter, which also reduces the noise but with different characteristics compared to the LMS filter.

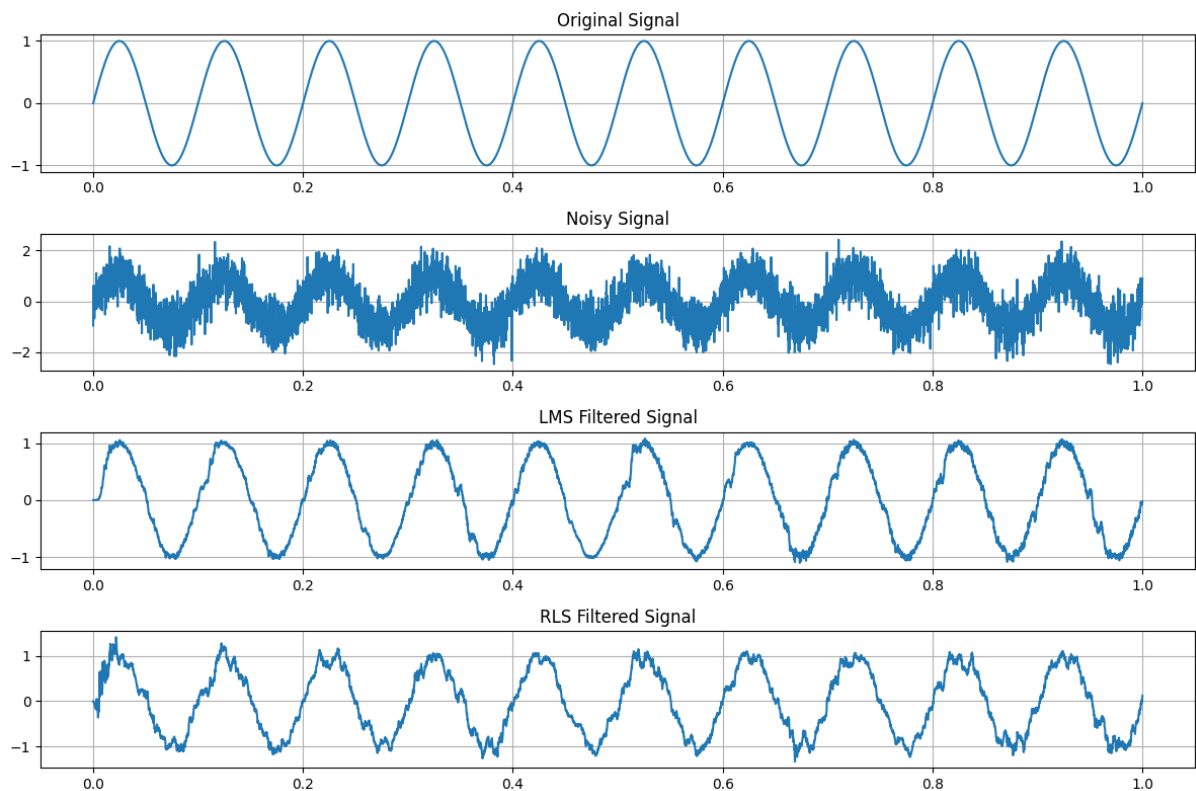


Figure 3.4: Results for $t = 1$ and filter length equal with 64

- **Original Signal:** This plot still shows the clean 10 Hz sine wave, oscillating smoothly over the time period.
- **Noisy Signal:** This plot still shows the sine wave with added noise, causing it to appear more erratic.
- **LMS Filtered Signal (64):** With a filter length of 64, the LMS filter have more coefficients to work with, which improve its performance in reducing noise. The filtered signal appears cleaner and more closely resemble the original sine wave than with the shorter filter length.
- **RLS Filtered Signal (64):** Similarly, the RLS filter with a filter length of 64 have more coefficients, which can enhance its ability to reduce noise.

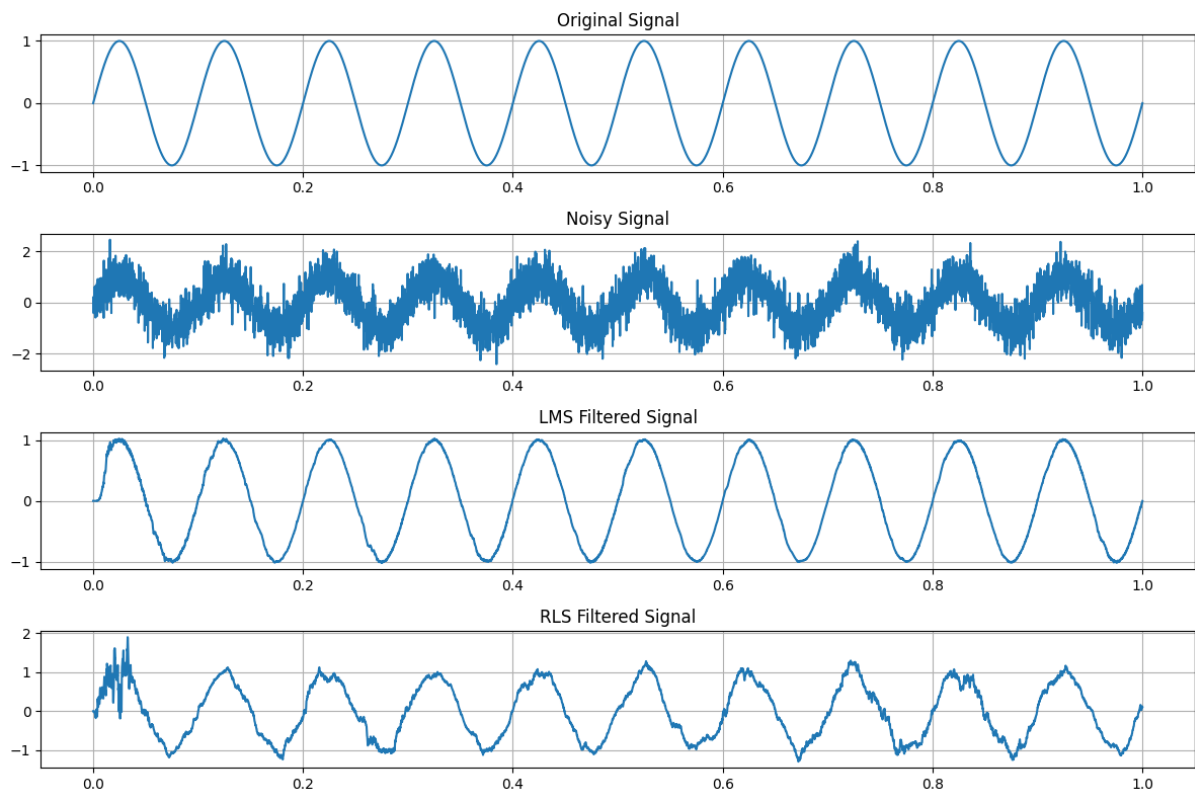


Figure 3.5: Results for $t = 1$ and filter length equal to 128

- **Original Signal:** This plot still shows the clean 10 Hz sine wave, oscillating smoothly over the time period.
- **Noisy Signal:** This plot still shows the sine wave with added noise, causing it to appear more erratic.
- **LMS Filtered Signal (128):** With a filter length of 128, the LMS filter have even more coefficients to work with, which can further improve its performance in reducing noise. The filtered signal appears significantly cleaner and more closely resemble the original sine wave compared to shorter filter lengths. The LMS filter is able to capture more of the signal's characteristics, resulting in better noise reduction.
- **RLS Filtered Signal (128):** Similarly, the RLS filter with a filter length of 128 have more coefficients, enhancing its ability to reduce noise even further. The filtered signal is very close to the original sine wave, with minimal noise and smooth oscillations. The RLS filter will perform even more effectively in capturing the signal's characteristics and reducing noise.

Conclusions

In this individual work, we explored noise generation and filtering techniques using the Discrete-Time System, focusing on the M-point Moving Average System. The study involved generating white noise with different sampling intervals, analyzing its characteristics through histograms, and applying various filtering methods to enhance signal quality. These experiments provided a deeper understanding of how noise behaves in digital signal processing and how filtering techniques can be used to extract meaningful signals from noisy environments.

We began by generating white noise using a random function and visualizing it through time-domain plots. By modifying the sampling interval, we observed how different resolutions affect the representation of noise. We then represented the statistical properties of noise using histograms, which helped in understanding the uniform and Gaussian distributions of random processes. To mitigate noise, we implemented digital filtering techniques. A second-order digital filter was applied to white noise, significantly reducing its randomness and producing a smoother signal. The impact of different filter parameters on signal clarity and smoothness was examined, demonstrating how filtering can enhance signal quality while preserving important features.

Further, we designed and applied a Moving Average Filter (MAF) to noisy signals. By adjusting the filter window size (M), we analyzed its effect on noise reduction. A smaller M provided less smoothing, preserving more details but allowing more noise to pass through, whereas a larger M resulted in a significantly smoother signal with reduced noise but also introduced a slight delay in signal response. These findings highlighted the trade-off between noise reduction and signal distortion in practical signal processing applications.

Additionally, we explored adaptive filtering techniques using the Least Mean Squares (LMS) and Recursive Least Squares (RLS) algorithms. These algorithms dynamically adjusted filter weights to minimize the error between the desired and output signals. Through various experiments, we found that RLS converged faster and provided more accurate filtering than LMS, making it more effective for real-time noise reduction. Increasing the filter length further improved performance for both algorithms, reducing noise and making the filtered signal more closely resemble the original.

Overall, this lab provided a comprehensive understanding of noise modeling, filtering techniques, and adaptive signal processing. The experiments

demonstrated the importance of digital filtering in various applications, from audio processing to communication systems, where noise reduction is essential for improving signal clarity. By implementing and comparing different filtering methods, we gained valuable insights into how to effectively manage noise in real-world signal processing scenarios.