REPORT

Zajęcia: Analog and digital electronic circuits
Teacher: prof. dr hab. Vasyl Martsenyuk

Lab 3

20.03.2025

Topic: "Digital Filter Design and Analysis: Implementing FIR and IIR filters in Python. Adaptive Filtering: Applying adaptive filtering algorithms to noise reduction."

Variant: 13

Rafał Żmuda
Informatyka II stopień,
stacjonarne,
1 semestr,
Gr.2b

1. Problem statement:

- Design an FIR filter with the following coefficients and implement it in Python to reduce noise in a noisy sinusoidal signal.
FIR Filter Coefficients: b = {1, 0.5, 0.5}
- Design an IIR filter with the following coefficients and implement it in Python to reduce
noise in the same noisy sinusoidal signal.
IIR Filter Coefficients: b = {1, −0.5, 0.2}, a = {1, −0.3}
- Implement an adaptive LMS filter in Python with a step size μ = 0.1 and filter length M = 4 to reduce noise in the same noisy sinusoidal signal.

2. Input data:

FIR: b = {1, 0.5, 0.5},
IIR: b = {1, −0.5, 0.2}, a = {1, −0.3}
LMS: = μ = 0.1, M = 4

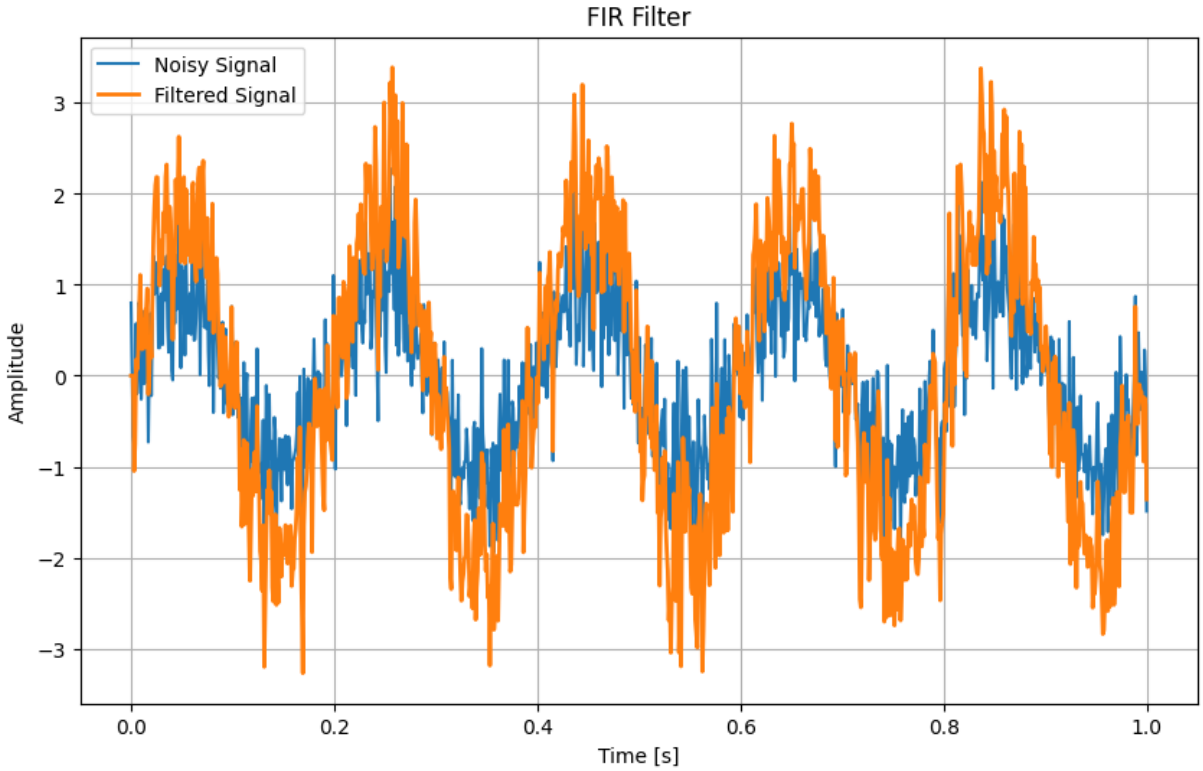
3. Commands used (or GUI):

3.1 FIR implementation

```python
import numpy as np
import matplotlib.pyplot as plt

def fir_filter(x, b):
    M = len(b)
    y = np.zeros(len(x))
    for n in range(M, len(x)):
        y[n] = np.dot(b, x[n-M+1:n+1][::-1])
    return y

fs = 1000  # Sampling frequency
t = np.linspace(0, 1, fs)
x = np.sin(2 * np.pi * 5 * t) + 0.5 * np.random.randn(len(t))  # Signal with noise
b = [1, 0.5, 0.5]  # FIR coefficients

y = fir_filter(x, b)

plt.figure(figsize=(10, 6))
plt.plot(t, x, label="Noisy Signal")
plt.plot(t, y, label="Filtered Signal", linewidth=2)
plt.legend()
plt.title("FIR Filter")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.grid()
plt.show()
```

Shown above is the cell that implements the FIR filter that applies the filter coefficients to an input signal. The code generates a noisy sinusoidal test signal at 5 Hz with random noise added, then filters it using a 3-tap FIR filter with coefficients [1, 0.5, 0.5]

Results:



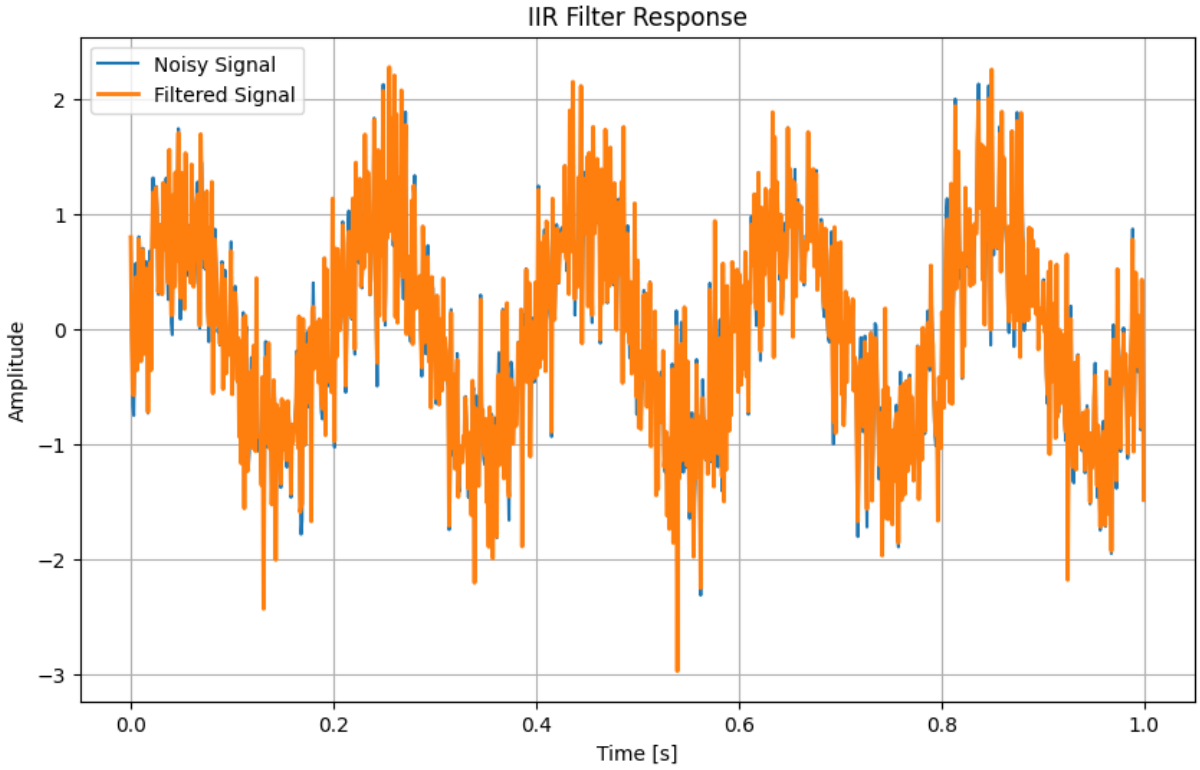
3.2 IIR Filter

```python
def iir_filter(x, b, a):
    M = len(b)  # Length of numerator coefficients (b)
    N = len(a)  # Length of denominator coefficients (a)
    y = np.zeros(len(x))  # Initialize output signal array

    for n in range(len(x)):
        x_slice = x[max(0, n-M+1):n+1]  # Input signal slice
        y[n] = np.dot(b[:len(x_slice)], x_slice[::-1])  # Apply reverse convolution for numerator

        if n >= 1:
            y_slice = y[max(0, n-N+1):n]  # Output signal slice
            y[n] -= np.dot(a[1:min(N, len(y_slice)+1)], y_slice[::-1])  # Apply reverse convolution for feedback

    return y

fs = 1000  # Sampling frequency
t = np.linspace(0, 1, fs)  # Time vector

# IIR filter coefficients
a = [1, -0.3]  # Denominator coefficients (a_0 = 1 by convention)
b = [1, -0.5, 0.2]  # Numerator coefficients
y = iir_filter(x, b, a)
```

A custom function called iir_filter that applies both feedforward (numerator) and feedback (denominator) coefficients to process an input signal. The function

handles boundary conditions by ensuring array indices remain valid when accessing past samples. The implementation uses a second-order IIR filter with numerator coefficients [1, -0.5, 0.2] and denominator coefficients [1, -0.3], applying it to the previously defined noisy sinusoidal signal. For each output sample, it calculates the weighted sum of current and previous input samples (using b coefficients) and subtracts the weighted sum of previous output samples (using a coefficients).

Results:

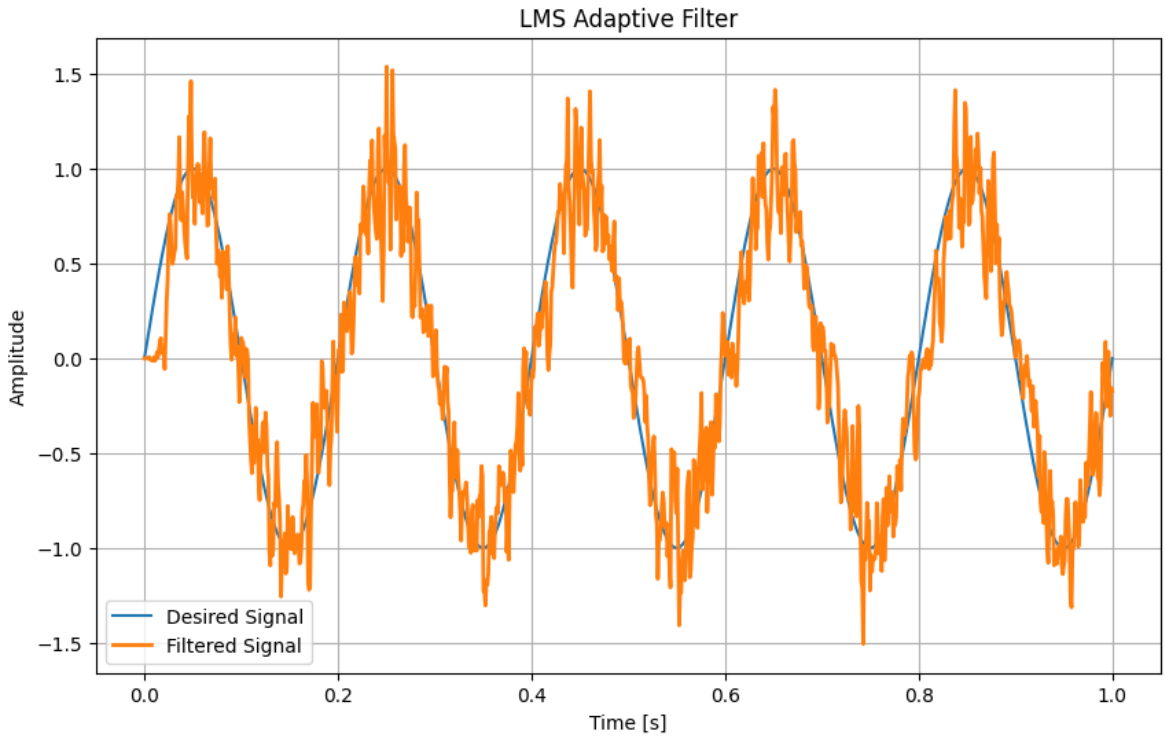


3.3 Adaptive LMS Filter

```python
def lms_filter(x, d, mu, num_taps):
    n = len(x)
    w = np.zeros(num_taps)
    y = np.zeros(n)
    e = np.zeros(n)

    for i in range(num_taps, n):
        x_segment = x[i-num_taps:i][::-1]
        y[i] = np.dot(w, x_segment)
        e[i] = d[i] - y[i]
        w += mu * e[i] * x_segment

    return y, e, w

d = np.sin(2 * np.pi * 5 * t )  # Desired signal
mu = 0.1  # Step size
num_taps = 4  # Number of filter taps

y, e, w = lms_filter(x, d, mu, num_taps)
```

We define a function called lms_filter that adaptively updates filter coefficients to minimize the error between the filtered output and a desired reference signal. The implementation initializes filter weights (w), output signal (y), and error signal (e) arrays, then processes the input sample by sample. For each iteration, it calculates the current output by applying the filter weights to a segment of the input signal, computes the error, and updates the weights proportionally to the error and input using the LMS algorithm.

Results:



Link to repository: https://github.com/RafalZmu/School/tree/main/Lab%203

5. Conclusions:

Each filter type suits different applications: FIR for precise phase requirements, IIR for efficient filtering with minimal coefficients, and adaptive filtering for unknown or changing signal environments.

The implementation complexity increases from FIR to IIR to adaptive methods, but this corresponds with greater flexibility and potentially improved performance in complex signal processing scenarios.