Digital Filtering on Arduino: Real-Time IIR and FIR Filtering, Frequency-Domain Validation, and Simulink Comparison

Mohammed-salih Diyari
Master 2 E3A – Smart Aerospace & Autonomous Systems
Université d'Evry-Val-d'Essonne

***Abstract*—This lab implements and validates real-time digital filters on an Arduino Uno using an analog potentiometer input and recorded signals. The work progresses chronologically from (i) first-order low-pass filtering for basic noise reduction, to (ii) second-order low-pass filtering with damping control, to (iii) a second-order band-pass filter designed via prewarping and bilinear transformation, and finally to (iv) FIR low-pass filtering using both predefined coefficients and window-based design. Results are evaluated in time domain and frequency domain (FFT), and outputs are compared against Simulink/MATLAB reference implementations. The report emphasizes practical implementation details (sampling, recursion state, buffering) and explains the underlying DSP theory using accessible terminology.**

***Index Terms*—Arduino, digital signal processing, IIR, FIR, low-pass, band-pass, bilinear transform, FFT, Simulink.**

I. Introduction

Sensors and analog inputs often contain noise due to quantization, electrical interference, and environmental variation. Digital filters reduce unwanted components while preserving the useful part of the signal. In embedded systems, filtering must be performed in real time with limited memory and CPU resources.

This lab demonstrates multiple filtering strategies on Arduino:

- First-order low-pass filtering (simple, stable, minimal cost).
- Second-order low-pass filtering (better roll-off, damping control).
- Second-order band-pass filtering via bilinear transform (frequency-selective).
- FIR low-pass filtering (linear phase, stable, coefficient-based convolution).

The lab also validates results using frequency-response plots, impulse/step responses, and comparisons to Simulink/MATLAB.

II. Experimental Setup (Chronological Start)

A. Hardware and Data Acquisition

An Arduino Uno is connected to a potentiometer on analog pin A0. The Arduino reads voltage samples in real time. Because potentiometer readings are naturally noisy and hand motion introduces sudden changes, it is a good "live" input to visualize filtering performance.

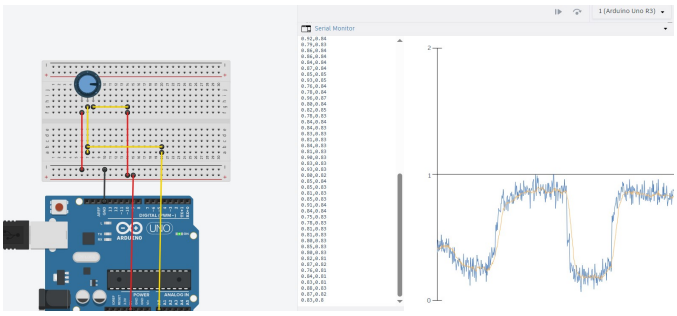

Fig. 1: Arduino + potentiometer setup and example live acquisition with noisy input and filtered output.

Fig. 1 shows the wiring and a representative live plot (raw vs filtered) during real-time acquisition.

B. Software Environment

The filtering is implemented and visualized using MATLAB scripts that:

- Connect to Arduino via serial/Arduino support package,
- Sample at a chosen sampling frequency f_s,
- Apply either a recursive difference equation (IIR) or convolution (FIR),
- Maintain a rolling display buffer for plotting.

III. Background Theory (Layman-Friendly)

A digital filter takes a sequence of samples $x[n]$ and produces an output $y[n]$ designed to reduce noise or isolate frequencies.

A. IIR vs FIR (Intuition)

IIR (Infinite Impulse Response): uses feedback; the output depends on past outputs. It is computationally cheap for strong filtering, but requires stability care.

FIR (Finite Impulse Response): uses only current and past inputs; it is always stable and can achieve linear phase, but may require more coefficients (more operations per sample).

B. Frequency Domain View

Any signal can be seen as a mixture of sinusoids. A low-pass filter keeps slow variations and removes fast variations (noise). A band-pass filter keeps only a band around a target

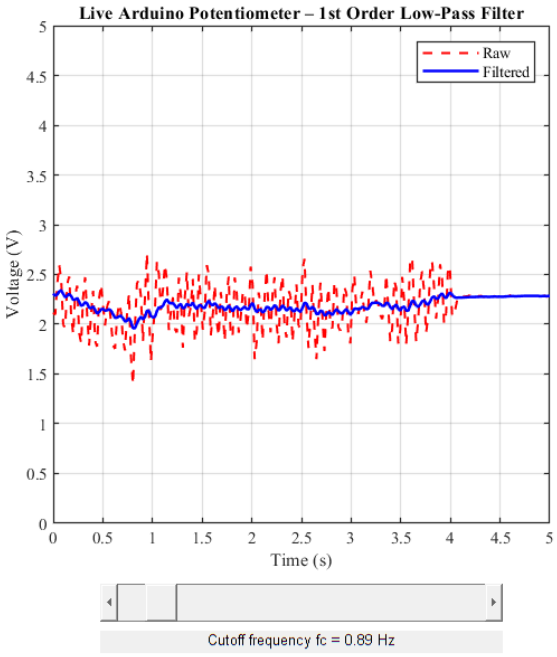

Fig. 2: First-order low-pass filtering on live Arduino potentiometer input (raw vs filtered).

frequency. FFT plots are used to confirm that unwanted high-frequency components are reduced.

IV. Part I — First-Order Low-Pass Filter (Real-Time)

A. Model and Difference Equation

A standard first-order low-pass IIR can be implemented as:

$$y[n] = \alpha x[n] + (1-\alpha)\, y[n-1], \tag{1}$$

where $0 < \alpha < 1$. **Meaning:** the output is a weighted average between the new sample and the previous output. Smaller α gives smoother output but more delay.

A cutoff frequency f_c can be linked to α using a time-constant approximation:

$$\alpha = \frac{2\pi f_c T_s}{1 + 2\pi f_c T_s}, \quad T_s = \frac{1}{f_s}. \tag{2}$$

B. Implementation Notes (Embedded Style)

In real-time code:

- Read $x[n]$ from A0,
- Update $y[n]$ using one stored state $y[n-1]$,
- Push data into a fixed plot buffer.

C. Results

Fig. 2 shows the noisy raw signal and the filtered output. The filtered curve is smoother and follows the general trend while rejecting rapid noise.

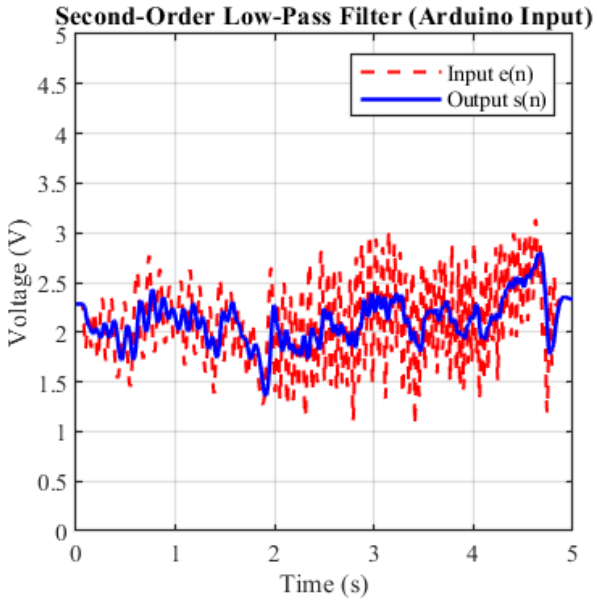

Fig. 3: Second-order low-pass filter on Arduino input: input $e(n)$ vs output $s(n)$.

V. Part II — Second-Order Low-Pass Filter (Arduino Input)

A. Analog Prototype

A common second-order low-pass analog prototype is:

$$H(s) = \frac{G_0\, \omega_c^2}{s^2 + 2m\omega_c s + \omega_c^2}, \tag{3}$$

where $\omega_c = 2\pi f_c$ and m is the damping factor. **Meaning:** compared to first order, a second-order filter attenuates high-frequency noise more strongly. The damping factor controls overshoot/ringing.

B. Discrete-Time Implementation

The filter is implemented using a second-order recursion, keeping two past input/output states (typical biquad structure):

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2]. \tag{4}$$

In practice, the coefficients come from discretizing the analog form (or directly using a digital design method). The key embedded requirement is storing $(x[n-1], x[n-2], y[n-1], y[n-2])$.

C. Results (Time Domain)

Fig. 3 shows improved smoothing compared to first order, especially when noise bursts occur.

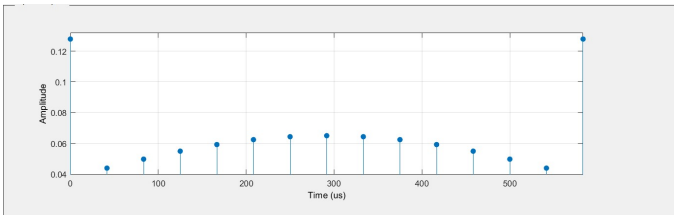

Fig. 4: Impulse response (time-domain characterization of the filter).

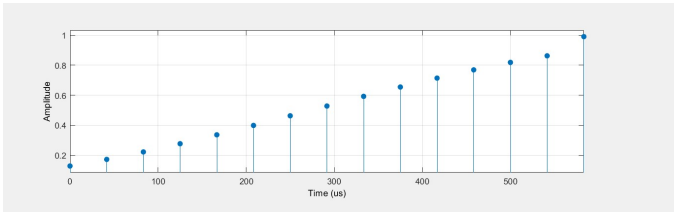

Fig. 5: Step response (shows settling time and overshoot behavior).

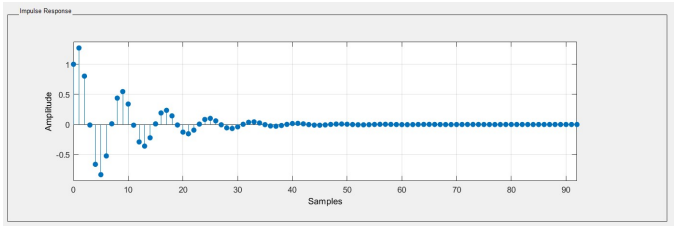

Fig. 6: Second-order filter response (impulse/response samples shown).

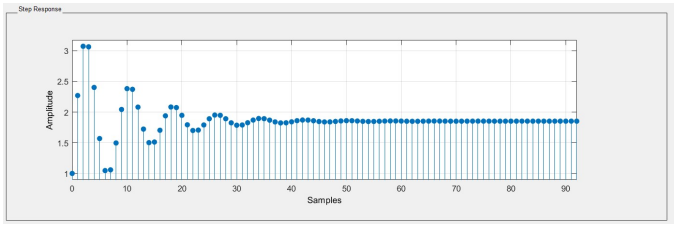

Fig. 7: Second-order step response (sample-domain).

D. Impulse and Step Responses

Impulse and step responses summarize how a filter behaves:

- **Impulse response:** output when the input is a single spike.
- **Step response:** output when the input jumps from 0 to 1 and stays there.

Additional second-order response plots are included in Fig. 6 and Fig. 7.

E. Frequency Response (Magnitude and Phase)

Magnitude response tells which frequencies are attenuated. Phase response indicates delay and waveform distortion risk.

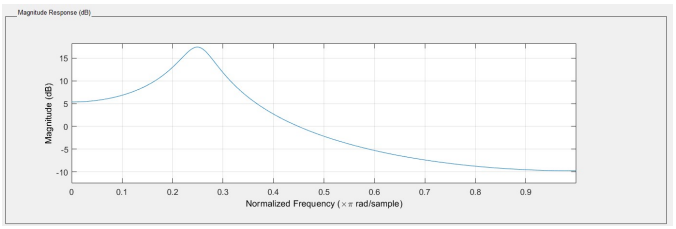

Fig. 8: Second-order magnitude response.

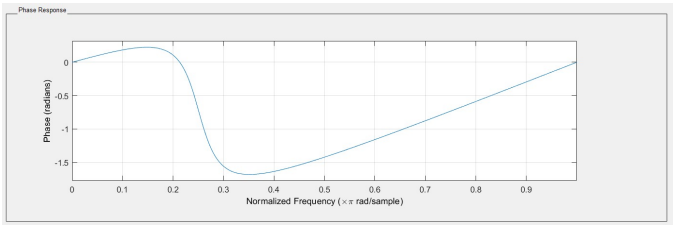

Fig. 9: Second-order phase response.

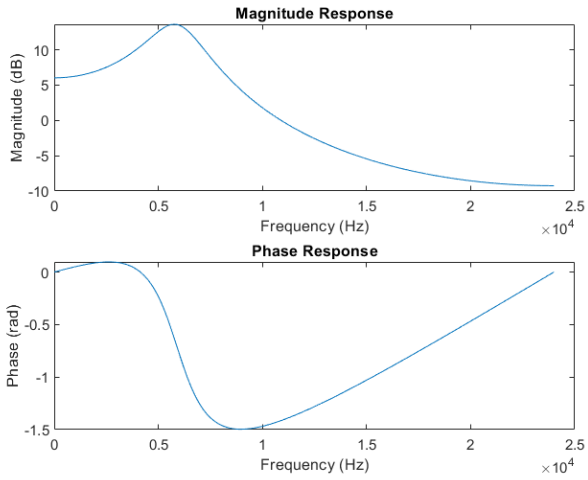

Fig. 10: Additional second-order low-pass filtering plot (Arduino input).

VI. Part III — Bilinear Band-Pass Filter (Arduino Input)

A. Goal

Unlike low-pass filtering (remove fast noise), a **band-pass** keeps only frequencies around a chosen center frequency and suppresses both low and high frequency components.

B. Analog Band-Pass Form and Design Parameters

A standard second-order band-pass analog form is:

$$H(s) = \frac{(\omega_0/Q)\, s}{s^2 + (\omega_0/Q)s + \omega_0^2}, \tag{5}$$

where:

- ω_0 = center angular frequency,
- Q = quality factor (controls bandwidth).

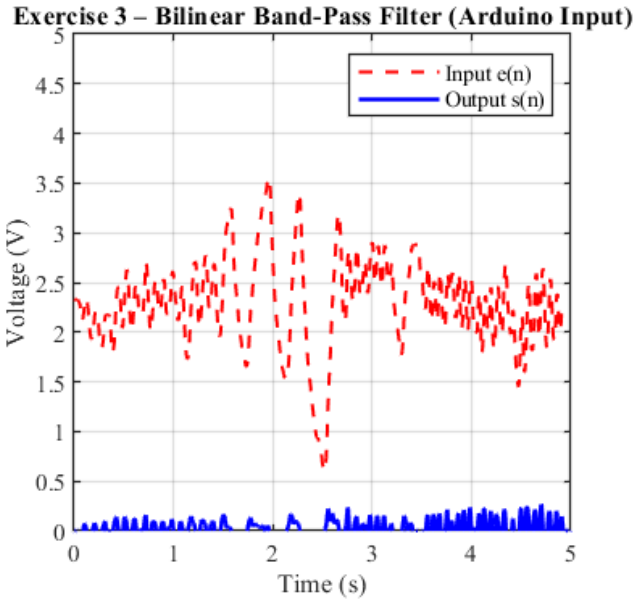

Fig. 11: Bilinear-designed band-pass filter: input $e(n)$ vs output $s(n)$.

C. Prewarping and Bilinear Transform

The bilinear transform maps s to z without instability, but it warps frequency. Prewarping corrects this at key design frequencies:

$$\Omega = \frac{2}{T_s} \tan\left(\frac{\omega}{2}\right), \tag{6}$$

then bilinear transform is applied:

$$s = \frac{2}{T_s} \frac{1 - z^{-1}}{1 + z^{-1}}. \tag{7}$$

This produces digital coefficients (b, a) used in the biquad recursion.

D. Results

Fig. 11 shows the Arduino input and the band-pass filtered output.

VII. Part IV — FIR Low-Pass Filtering (Two Approaches)

A. FIR Filtering by Convolution

An FIR filter computes:

$$y[n] = \sum_{k=0}^{N-1} b[k]\, x[n-k]. \tag{8}$$

Meaning: multiply recent samples by coefficients and sum. This is a moving weighted average (but with carefully designed weights).

```
Numerator:
0.12784033339363648495989878028922248631 7
0.04395968865180104045986553273905883543 2
0.04978779098637662464676978402167151216 4
0.05497073539613946124848808949536760337 7
0.05927106975711159825292995151357899885 6
0.06246375850857498190915606528506032191 2
0.06437279343266205655638145799457561224 7
0.06502279953729285522090464155553490854 8
0.06437279343266205655638145799457561224 7
0.06246375850857498190915606528506032191 2
0.05927106975711159825292995151357899885 6
0.05497073539613946124848808949536760337 7
0.04978779098637662464676978402167151216 4
0.04395968865180104045986553273905883543 2
0.12784033339363648495989878028922248631 7
```

Fig. 12: FIR numerator coefficients used in the coefficient-based FIR implementation.

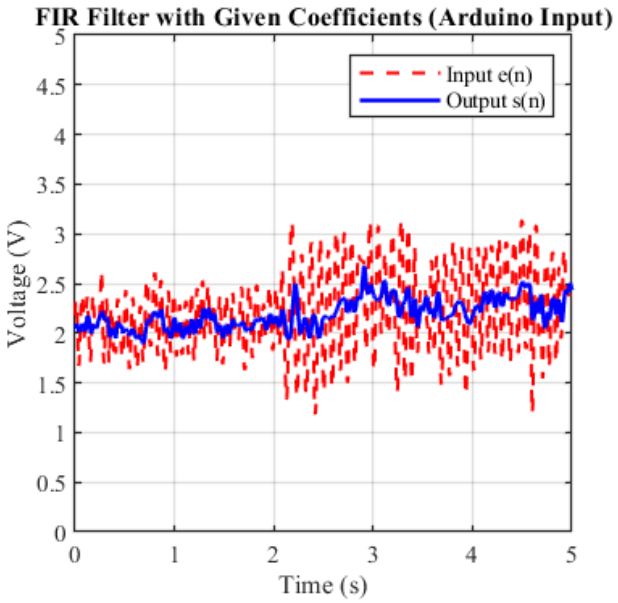

Fig. 13: Coefficient set / design preview (FDA tool export screenshot).

B. Approach A: FIR Using Given Coefficients

The lab uses a provided coefficient set (Fig. 12). These are loaded into the script and applied via a shift register buffer.

Time-domain effect of these coefficients is shown in Fig. **??**.

C. Approach B: FIR via Window Method (Hamming)

An ideal low-pass impulse response is a shifted sinc:

$$h_{\text{ideal}}[n] = 2\frac{f_c}{f_s}\,\text{sinc}\left(2\frac{f_c}{f_s}(n - M)\right), \tag{9}$$

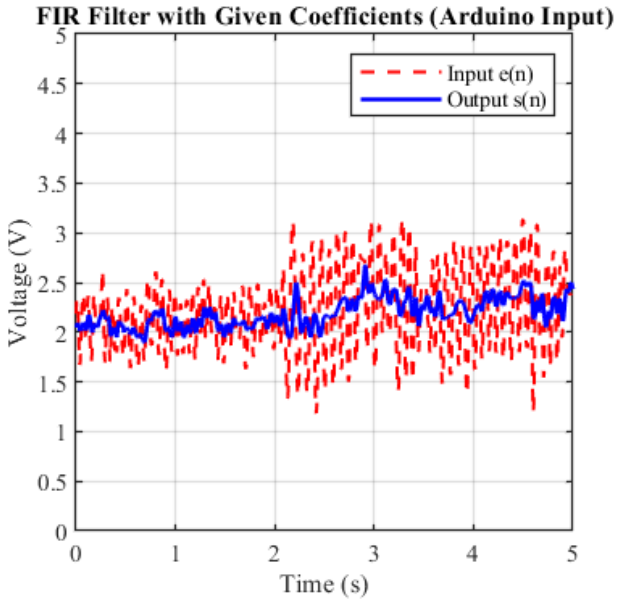

Fig. 14: Reference FIR design/coefficients confirmation (as provided).

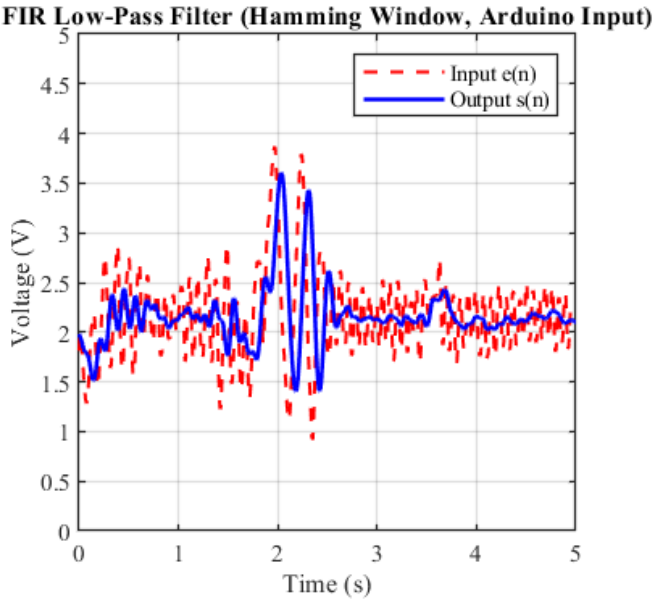

Fig. 15: FIR low-pass filtering using Hamming window (Arduino input): input vs output.

with $M = (N-1)/2$. To reduce ripples, multiply by a window (Hamming):

$$w[n] = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \tag{10}$$

$$b[n] = h_{\text{ideal}}[n] \cdot w[n]. \tag{11}$$

The resulting FIR behavior is shown in Fig. 15.

VIII. Part V — Frequency Response Validation (Magnitude and Phase)

To confirm filter design beyond time plots, magnitude and phase responses are plotted.

IX. Part VI — DO Signal Filtering + Spectral Verification

A. Time-Domain Effect on DO Signal

The DO signal is filtered using FIR low-pass filtering to reduce high-frequency noise while retaining the main oscillation structure.

B. FFT-Based Validation (Spectrum Before/After)

A good filter should reduce unwanted high-frequency spectral components. FFT comparisons are shown in Fig. 22 and Fig. 23.

X. Part VII — Simulink/MATLAB vs Embedded FIR Comparison

To ensure the embedded-style FIR recursion (shift buffer + dot product) matches a reference implementation, the output is compared against Simulink FIR.

Additional comparisons and views are provided in Fig. 25–Fig. 27.

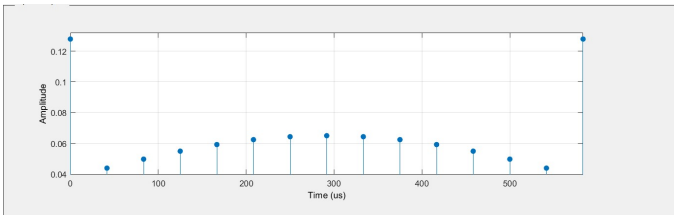

Fig. 16: Impulse response visualization used to validate FIR behavior.

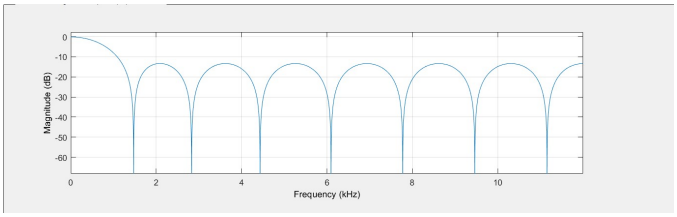

Fig. 17: Magnitude response of designed filter (dB vs frequency).

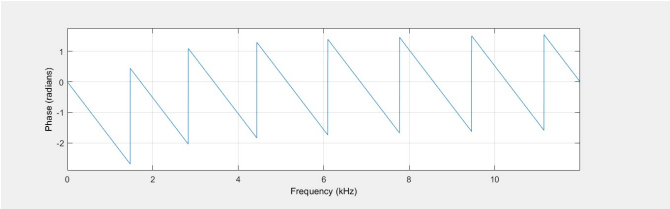

Fig. 18: Phase response of designed filter.

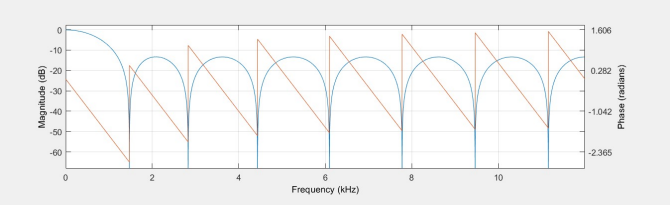

Fig. 19: Combined magnitude and phase plot (same design).

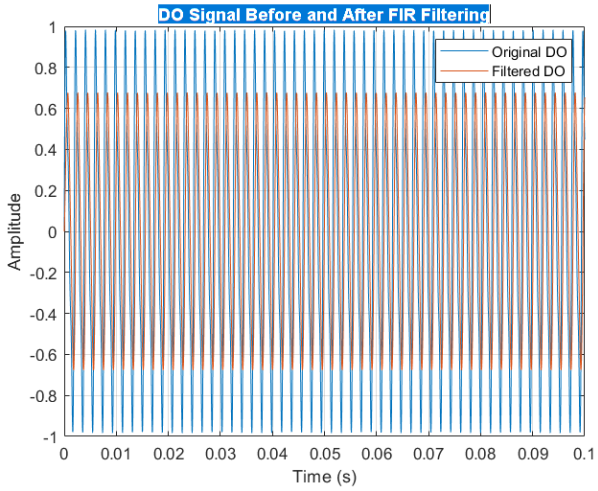

Fig. 20: DO signal before and after FIR filtering (time domain).

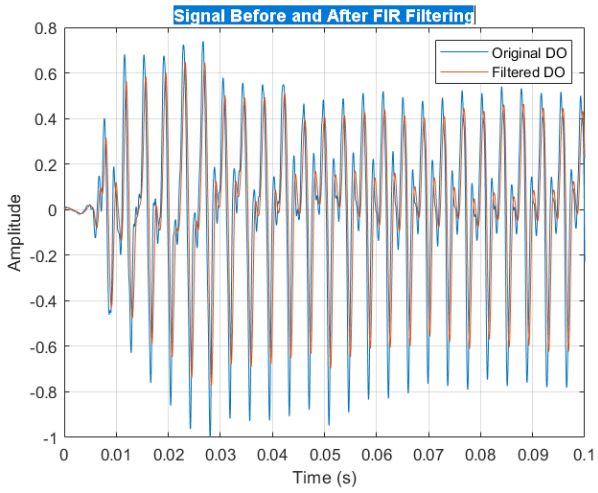

Fig. 21: Signal before and after FIR filtering (alternative view).

XI. Discussion

Chronologically, each stage improved control and validation depth:

- **First-order LPF:** simplest real-time denoising; minimal memory; best for quick smoothing.
- **Second-order LPF:** stronger attenuation; damping factor influences overshoot and settling; requires more state variables.

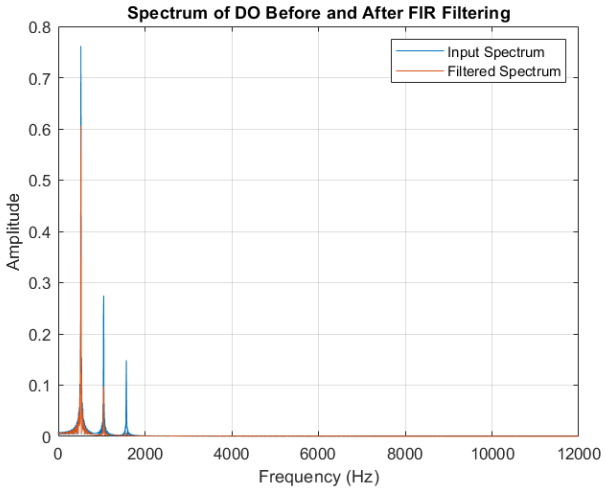

Fig. 22: Spectrum of DO signal before/after FIR filtering (FFT amplitude).

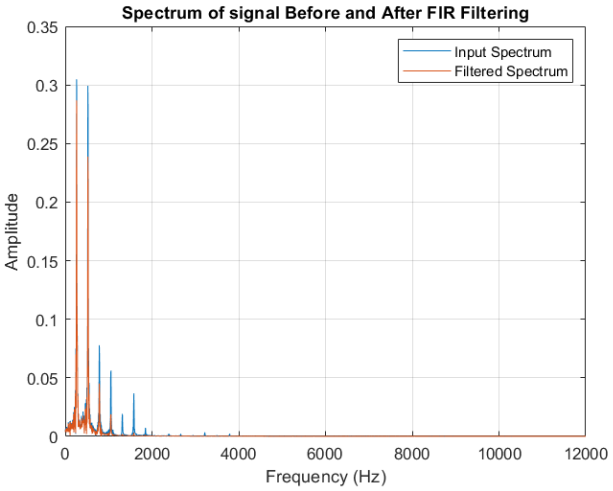

Fig. 23: Spectrum of signal before/after FIR filtering (FFT amplitude).

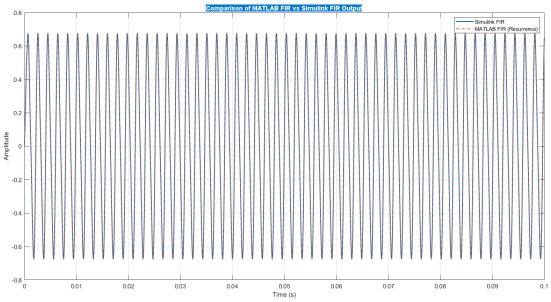

Fig. 24: Comparison of FIR output: MATLAB (recurrence/convolution) vs Simulink reference.

- **Band-pass (bilinear):** targets a narrow frequency range; prewarping is necessary because bilinear transform warps

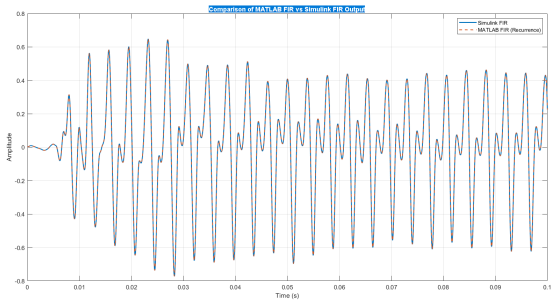

Fig. 25: Unfiltered vs filtered signal (Simulink FIR) — time domain.

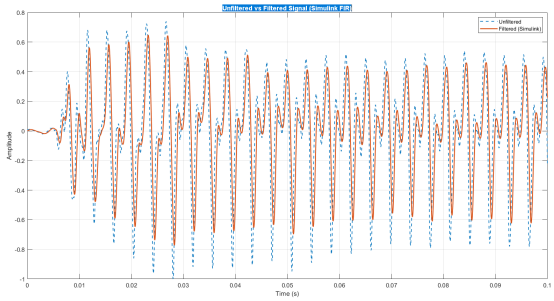

Fig. 26: Unfiltered vs filtered signal (Simulink FIR) — alternative view.

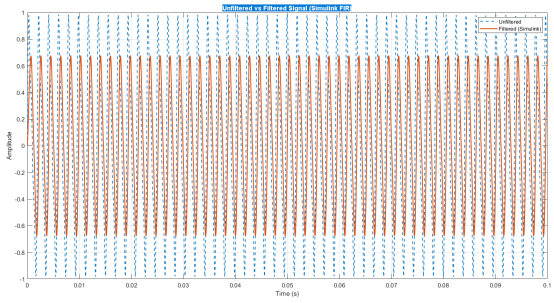

Fig. 27: Unfiltered vs filtered DO signal (Simulink FIR).

frequency.

- **FIR filters:** stable and predictable; performance depends on coefficient choice and length N; window method is a standard practical design technique.
- **FFT validation:** confirmed that filtering reduces unwanted spectral components rather than just making the time plot "look smooth."
- **Simulink comparison:** verified implementation correctness against a trusted reference model.

XII. Conclusion

This lab implemented multiple digital filters on Arduino-driven signals and validated them using time-domain plots, impulse/step responses, magnitude/phase responses, FFT spectra, and Simulink comparisons. The results demonstrate that real-time embedded filtering is practical when the difference equation and buffering are designed carefully. Future work could include measuring execution time per sample, fixed-point implementation, and automated coefficient generation for different filter requirements.

Appendix A: Practical Implementation Notes (What Was Done)

- Sampling frequency f_s was selected per experiment (e.g., 30 Hz, 50 Hz, 200 Hz depending on script).
- Real-time plotting used a rolling buffer of length $N = f_s \times T_{\text{plot}}$.
- IIR filters used stored past outputs (and sometimes past inputs) to compute $y[n]$.
- FIR filters used a shift register buffer storing the last N input samples and computed a dot product with coefficient vector b.
- FFT plots compared the spectrum of the original signal with the filtered signal to confirm attenuation of undesired frequency components.

References

[1] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. Pearson, 2010.
[2] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms and Applications*, 4th ed. Prentice Hall, 2006.
[3] MathWorks, "Digital Filters and Filter Design," MATLAB Documentation, accessed in lab environment.