Measurement Systems

4th Lab Session

Professor: Dr. Amini

Amirhossein Ansari 810600050

Hana Shamsaei 810600097

Spring 1404

Department of Mechanical Engineering

University of Tehran

Contents

1 Assignment 1: Sampling Frequency Effect on Aliasing **2**

1.1 Circuit and Sampling Setup 2

1.2 Results 3

1.2.1 Sampling Frequency: $f_s = 12$ Hz 3

1.2.2 Sampling Frequency: $f_s = 20$ Hz 5

1.2.3 Sampling Frequency: $f_s = 12$ Hz 6

1.2.4 Sampling Frequency: $f_s = 46$ Hz 7

1.2.5 Sampling Frequency: $f_s = 92$ Hz 8

1.2.6 Sampling Frequency: $f_s = 120$ Hz 9

2 Assignment 2: Low-Pass Filter **11**

2.1 Introduction 11

2.2 What is sampling? 11

2.3 What is FFT? 11

2.4 What is hanning window? 11

2.5 Circuit description 11

3 Sampling the Signal **12**

4 Applying Hanning Window **13**

5 Applyuing Fast Fourier Transform (FFT) Analysis **14**

6 FFT Peak Detection and Attenuation Analysis **15**

7 Attachments **17**

7.1 Assignment 1 Arduino Code 17

7.2 Assignment 1 MATLAB Code 19

7.2.1 Assignment 1 MATLAB Code For Receiving Data 19

7.2.2 Assignment 1 MATLAB Code For Processing 19

7.3 Assignment 2 Arduino Code 21

7.4 Assignment 2 MATLAB Code 23

1 Assignment 1: Sampling Frequency Effect on Aliasing

In this experiment, we aim to observe the effect of sampling frequency on aliasing using a clean sinusoidal input signal. Before sampling, we used a low-pass filter composed of a 33 Ω resistor and a 100 µF capacitor to remove high-frequency noise from the signal. This setup helps us ensure that the signal being sampled is as close as possible to the desired waveform.

Although filtering is not the main focus of this assignment, we initially observed that the filter introduces a small attenuation. As shown in Figure 1, a 5 V peak-to-peak input signal is attenuated to approximately 4.5 V after passing through the filter. This is a minor but expected effect due to the nature of the RC filter.

The main objective of this task is to demonstrate how different sampling frequencies affect the representation of an analog signal in the digital domain. By applying various sampling rates to a 23 Hz sine wave, we study how undersampling leads to aliasing and distorts the frequency content. The results are visualized in both time and frequency domains using MATLAB.

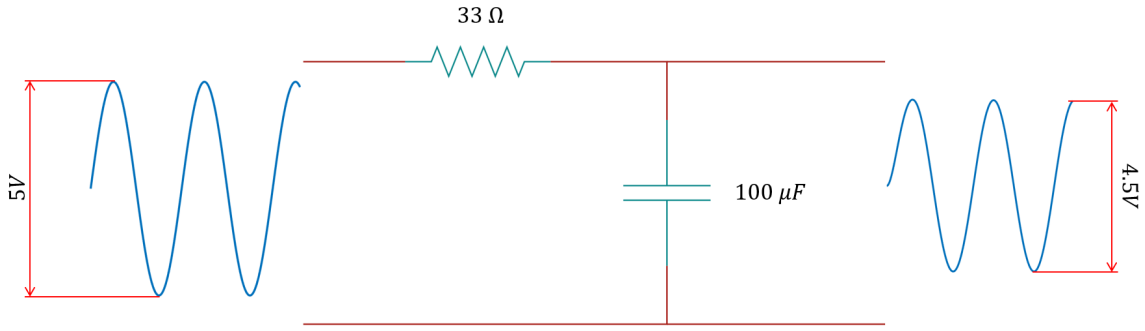

Figure 1: Low-pass filter circuit with 33 Ω resistor and 100 µF capacitor.

1.1 Circuit and Sampling Setup

To investigate the effect of sampling frequency on aliasing, we generated a sine wave using the PWM output of an Arduino Uno. The waveform had a frequency of 23 Hz, an amplitude of 2.5 V, and an offset of 2.5 V, producing a smooth 5 V peak-to-peak signal. This PWM signal was updated at 1000 Hz using the `MsTimer2` library.

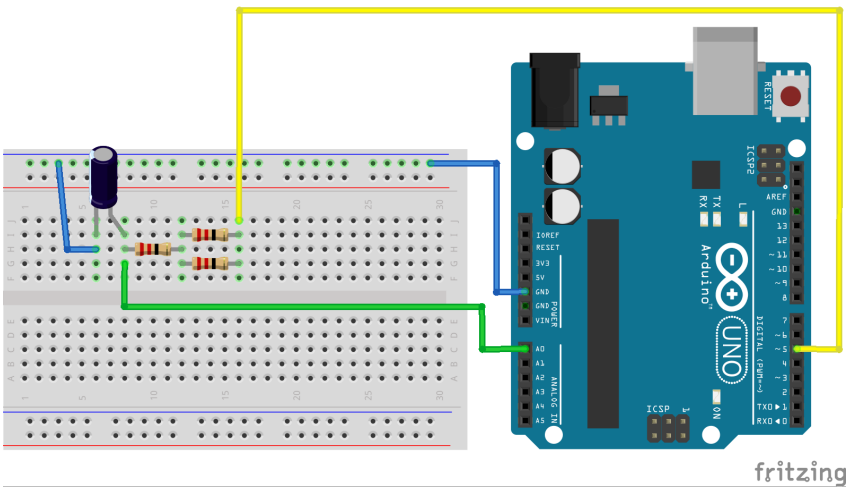

Figure 2: Breadboard view of the circuit

As illustrated in the breadboard setup, the PWM output is first passed through a passive RC filter to smooth the signal. This is crucial to reduce the high-frequency components generated by the PWM switching. The analog version of the waveform is then ready to be sampled.

The RC filter consists of three 22 Ω resistors — one in series and two in parallel — and a 100 µF capacitor connected to ground. This configuration improves the filter's roll-off and helps achieve a more stable analog voltage at the sampling point.

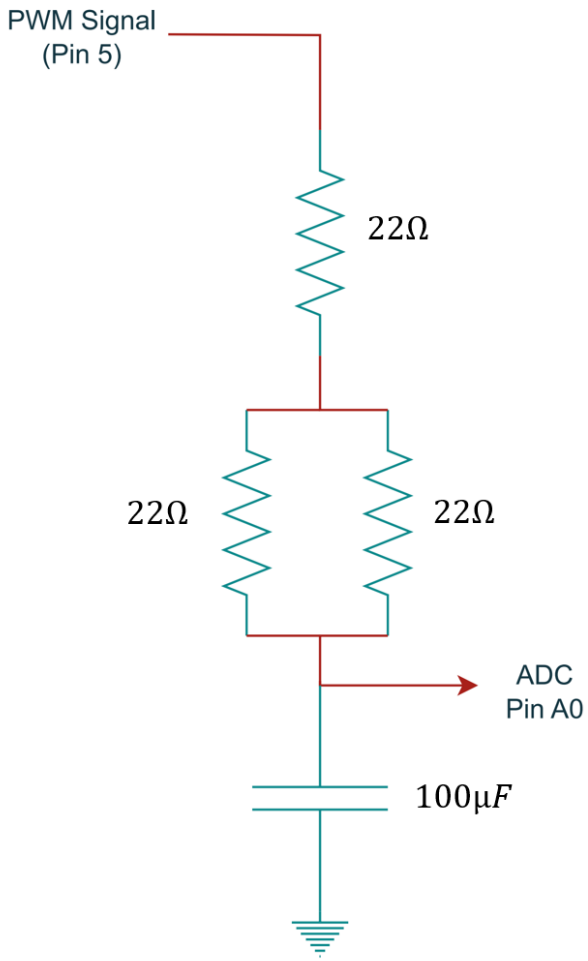

Figure 3: Schematic of the low-pass filter

To study aliasing, the filtered analog signal was sampled at multiple frequencies, including 12 Hz, 20 Hz, 26 Hz, 46 Hz, 92 Hz, and 120 Hz. For each sampling frequency, a total of 2 seconds worth of data was recorded. The samples were then sent to a PC for further analysis using MATLAB. This multi-rate sampling approach allows us to compare the signal's appearance in time and frequency domains and observe how undersampling distorts the true frequency content due to aliasing.

1.2 Results

1.2.1 Sampling Frequency: $f_s = 12$ Hz

At a sampling frequency of 12 Hz, the sampled signal deviates significantly from the original 23 Hz sine wave. Since the sampling rate is well below the Nyquist rate (which should be at least 46 Hz), aliasing occurs.

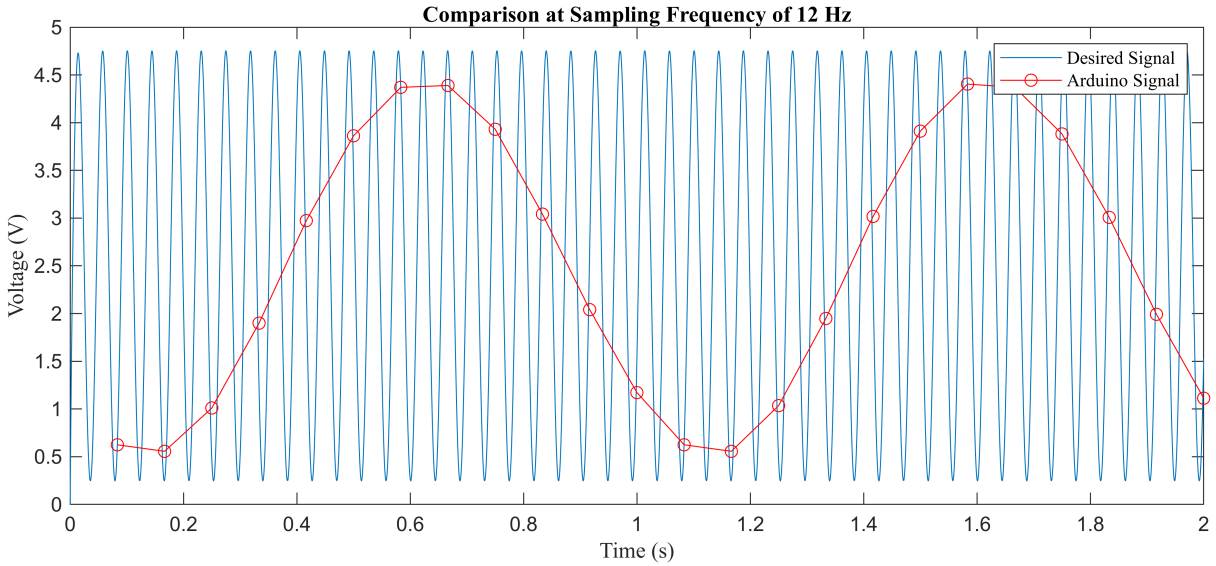

Figure 4: Comparison of the desired 23 Hz sine wave with the Arduino-sampled signal at 12 Hz.

As illustrated in Figure 4, the reconstructed waveform from the sampled data appears distorted and fails to preserve the original frequency. Instead of following a high-frequency oscillation, the sampled points form a much slower, misleading waveform. This is a clear indication of aliasing, where the signal appears to have a lower frequency than its true value due to insufficient sampling.

To further analyze the signal in the frequency domain, we applied a Fast Fourier Transform (FFT) to the acquired data. The original signal frequency of 23 Hz is not visible in the FFT result below:

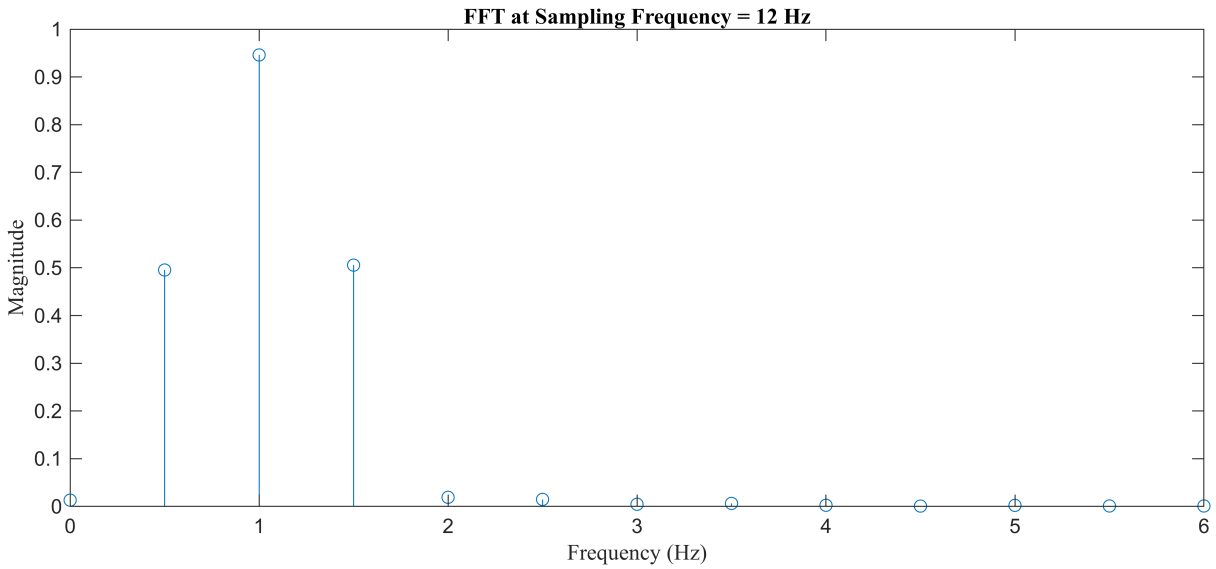

Figure 5: Frequency spectrum of the sampled signal at 12 Hz.

As seen in Figure 5, the dominant frequency components appear around 1 Hz and 2 Hz. These are not present in the actual input signal but are the result of aliasing. Specifically, the aliased frequency can be estimated using the relation:

$$f_{\text{alias}} = |f_{\text{signal}} - N \cdot f_s|$$

For $N = 2$, this gives $|23 - 2 \cdot 12| = |23 - 24| = 1$ Hz, which aligns with the FFT results.

This combined time-domain and frequency-domain analysis confirms the strong aliasing effect due to undersampling. Similar evaluations will be presented for other sampling frequencies in the upcoming sections.

1.2.2 Sampling Frequency: $f_s = 20$ Hz

When the sampling frequency was increased to 20 Hz, the sampled signal still failed to represent the original sine wave accurately. Although the waveform appears smoother than in the previous case (12 Hz), it still suffers from aliasing. This is because the signal frequency ($f = 23$ Hz) is above the Nyquist limit of $f_s/2 = 10$ Hz, causing distortion in the captured waveform.

As shown in Figure 6, the red markers representing the Arduino samples deviate noticeably from the continuous waveform. The reconstructed shape does not follow the expected sinusoidal form but instead takes a misleading, jagged profile.

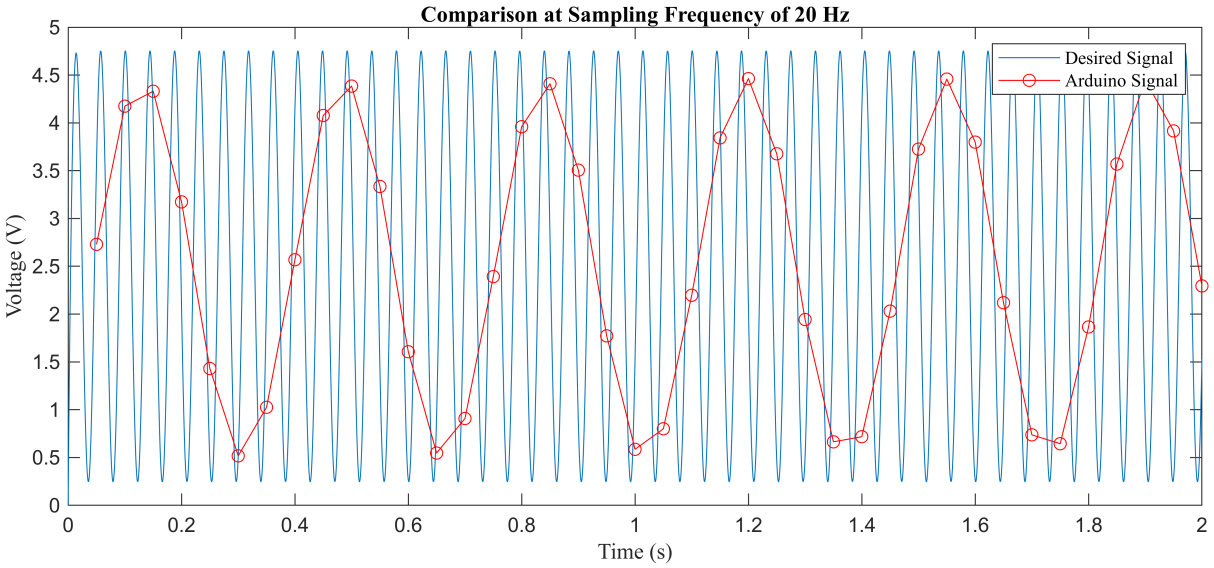

Figure 6: Comparison at Sampling Frequency of 20 Hz.

The FFT result in Figure 7 confirms this aliasing effect. A peak appears not at the original signal frequency (23 Hz), but at an aliased frequency of $|f_s - f| = 3$ Hz, which is consistent with the theoretical aliasing outcome. This further emphasizes that the sampling frequency must be more than twice the signal frequency to correctly capture the waveform's characteristics.

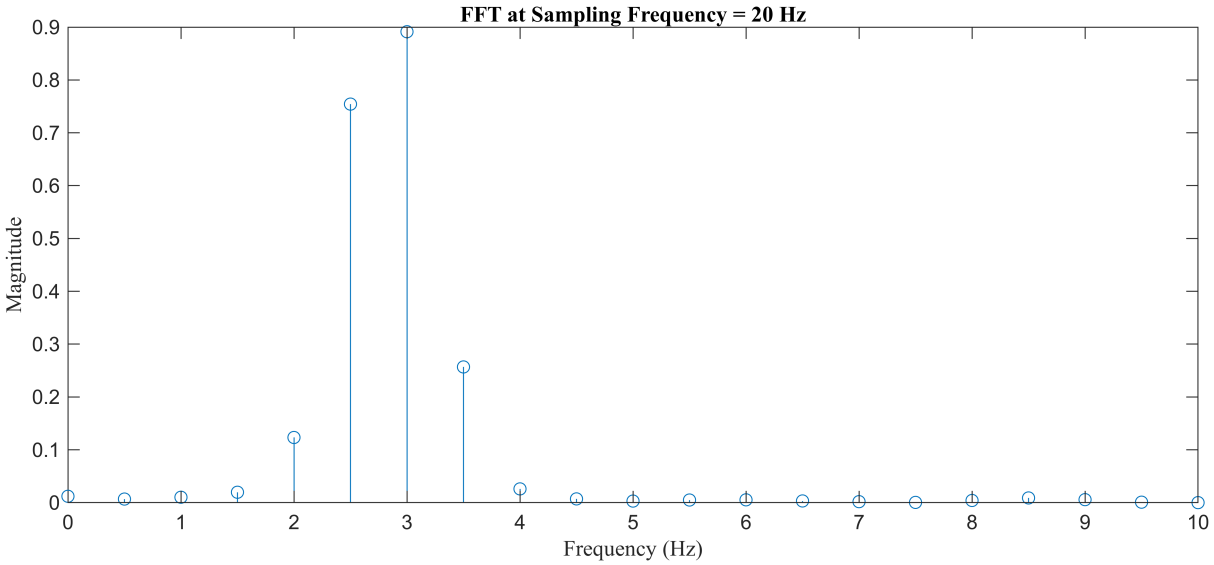

Figure 7: FFT at Sampling Frequency of 20 Hz.

This behavior continues to be observed as sampling frequency is varied and will be discussed for other values in the following sections.

1.2.3 Sampling Frequency: $f_s = 12$ Hz

As the sampling frequency increases further to 26 Hz, the system begins to satisfy the Nyquist criterion more comfortably for the target signal frequency of 23 Hz. The reconstructed waveform in Figure 8 begins to resemble the original sinusoidal waveform more closely, although minor aliasing and stair-step behavior are still present due to the discrete nature of the samples.

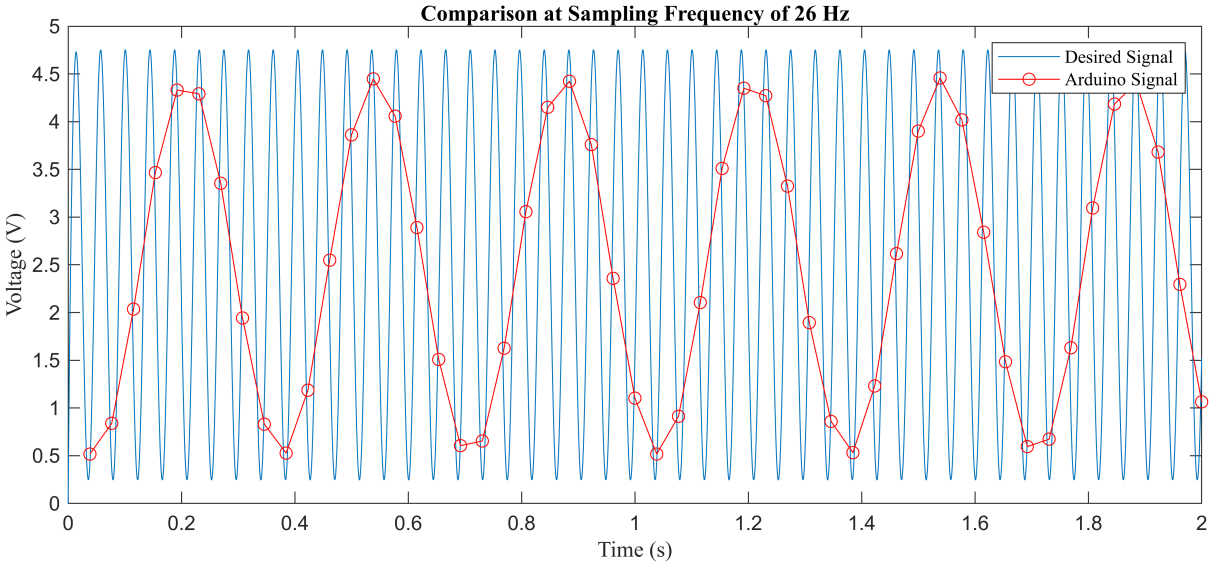

Figure 8: Comparison at Sampling Frequency of 26 Hz

The corresponding FFT result, shown in Figure 9, demonstrates a strong peak near 3 Hz. This is consistent with the expected aliasing behavior: since $f_s = 26$ Hz and the signal frequency is 23 Hz, the

aliased frequency observed in the FFT is:

$$f_{\text{alias}} = |f_{\text{input}} - f_s| = |23 - 26| = 3\,\text{Hz}$$

This confirms that the sampling captures the underlying signal content more effectively, and the magnitude of the alias component becomes dominant in the spectrum.

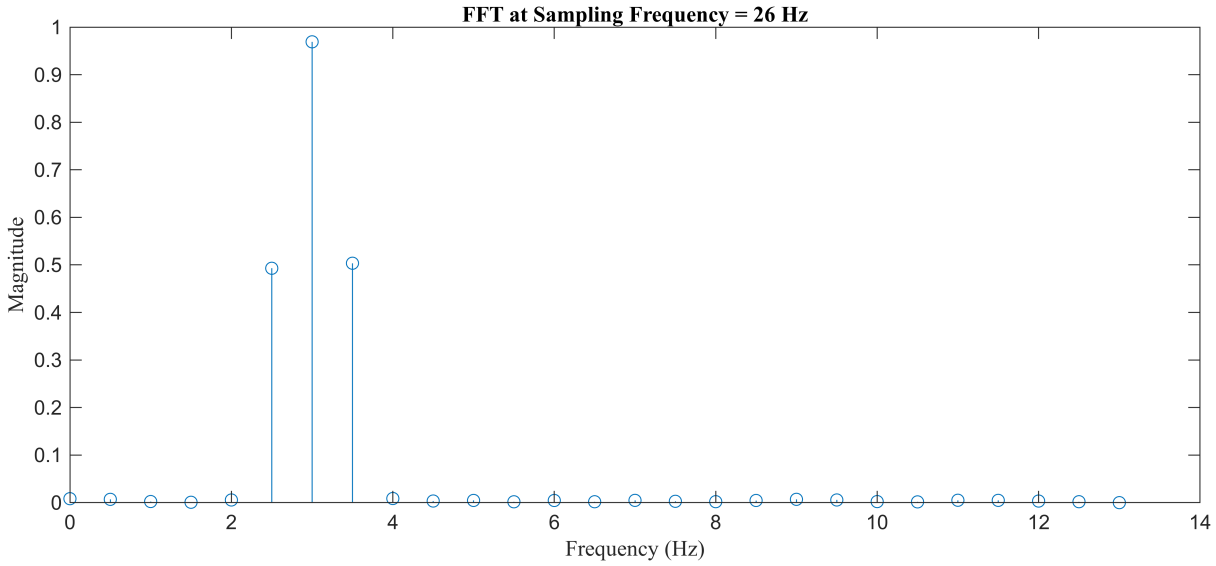

Figure 9: FFT at Sampling Frequency of 26 Hz

1.2.4 Sampling Frequency: $f_s = 46$ Hz

At a sampling frequency of 46 Hz, which is exactly twice the signal frequency ($f = 23$ Hz), we expect the sampling to be on the edge of satisfying the Nyquist criterion. However, since the sampling points are aligned such that they fall on opposite sides of the sine wave's phase, the result is a highly misleading flat signal.

As shown in Figure 10, the Arduino samples appear to oscillate very little, mostly showing a nearly constant voltage. This is a classic case of aliasing, where the sampling interval causes the signal to appear falsely as a constant or low-frequency component due to its misalignment with the original waveform.

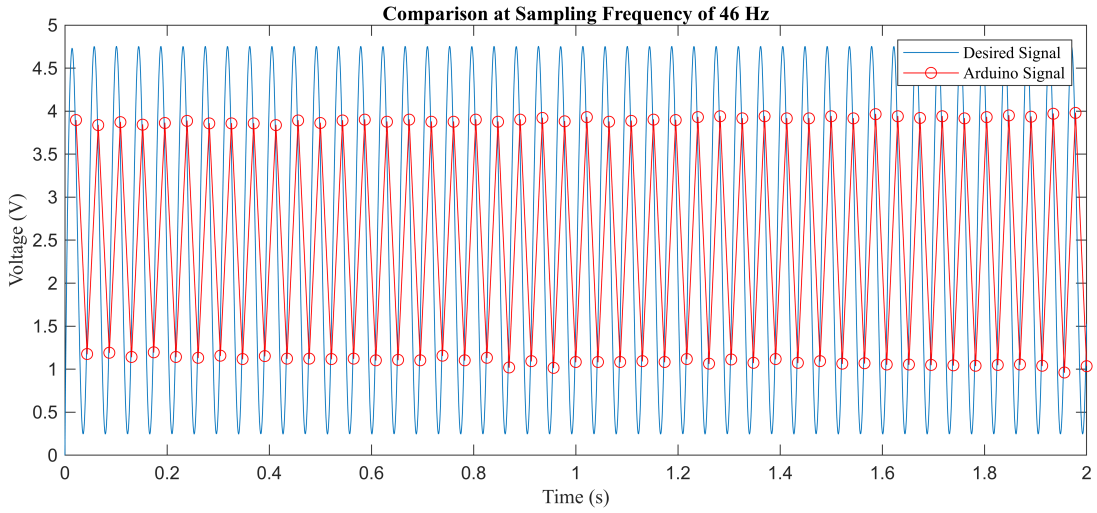

Figure 10: Comparison between generated sine signal and sampled signal at $f_s = 46$ Hz

The FFT result in Figure 11 shows two peaks near 23 Hz and its alias. This is a reflection of spectral leakage and the confusion introduced by sampling exactly at the Nyquist rate. The magnitude spectrum lacks the sharp dominance of the expected 23 Hz frequency due to phase misalignment and poor temporal resolution.

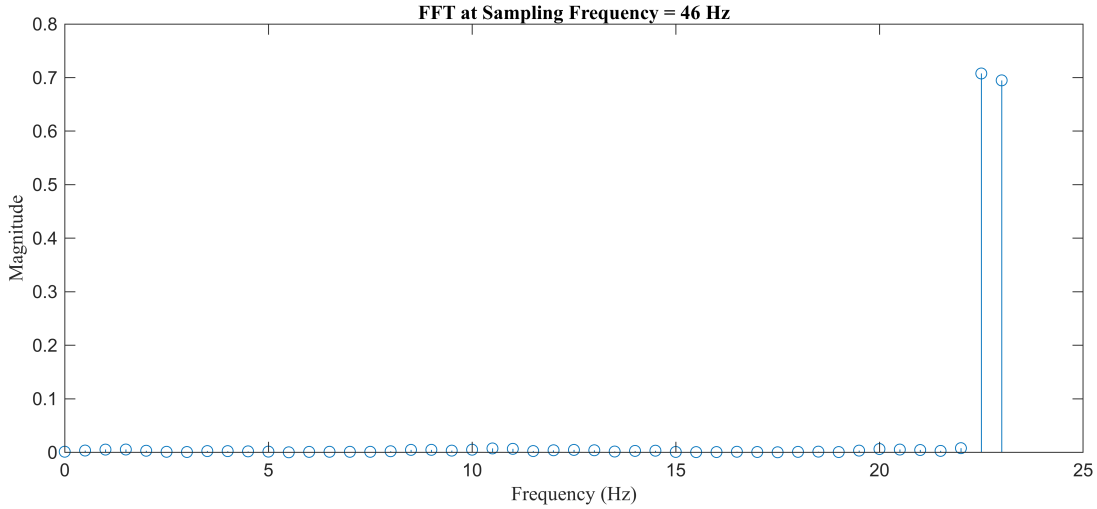

Figure 11: FFT of the signal sampled at $f_s = 46$ Hz

This scenario further emphasizes the importance of sampling at a frequency significantly higher than twice the maximum signal frequency to ensure accurate reconstruction and frequency domain interpretation.

1.2.5 Sampling Frequency: $f_s = 92$ Hz

In this case, the sampling frequency is set to 92 Hz, which is exactly four times the original signal frequency of 23 Hz. As shown in Fig. 12, the Arduino samples successfully follow the desired waveform with significantly better accuracy than the lower sampling frequencies. The signal's shape and periodic behavior are more clearly captured, with reduced distortion and less evidence of aliasing.

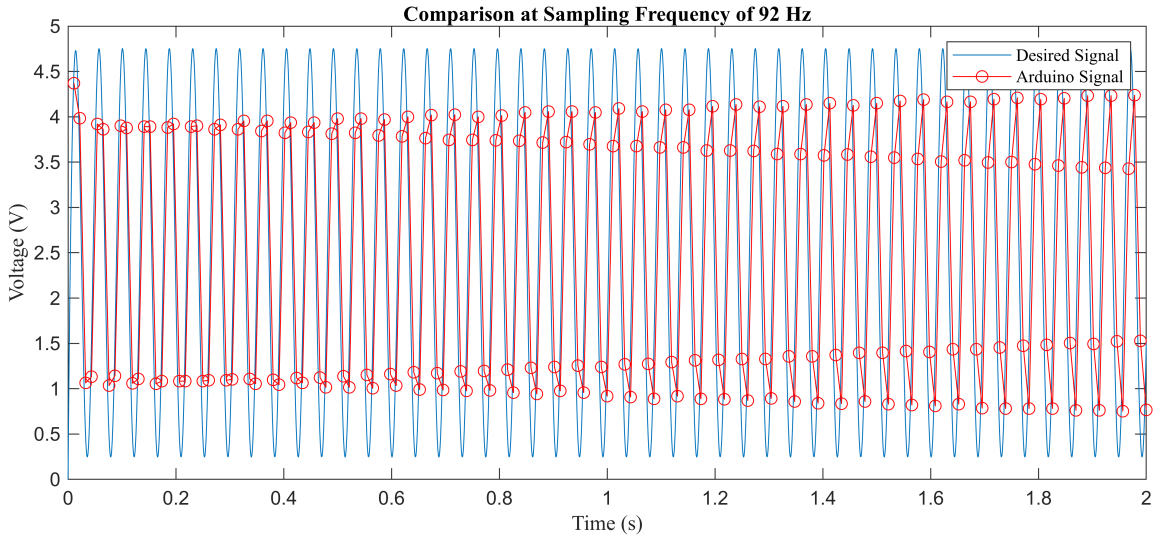

Figure 12: Comparison of desired and Arduino-sampled signals at 92 Hz sampling frequency.

The FFT result in Fig. 13 confirms this improvement, as the dominant spectral peak is sharply observed at 23 Hz, matching the sine wave generation frequency. The presence of minor spectral leakage or harmonic content is minimal, indicating that 92 Hz provides a suitable resolution and fidelity for both time and frequency domain representation of the signal.

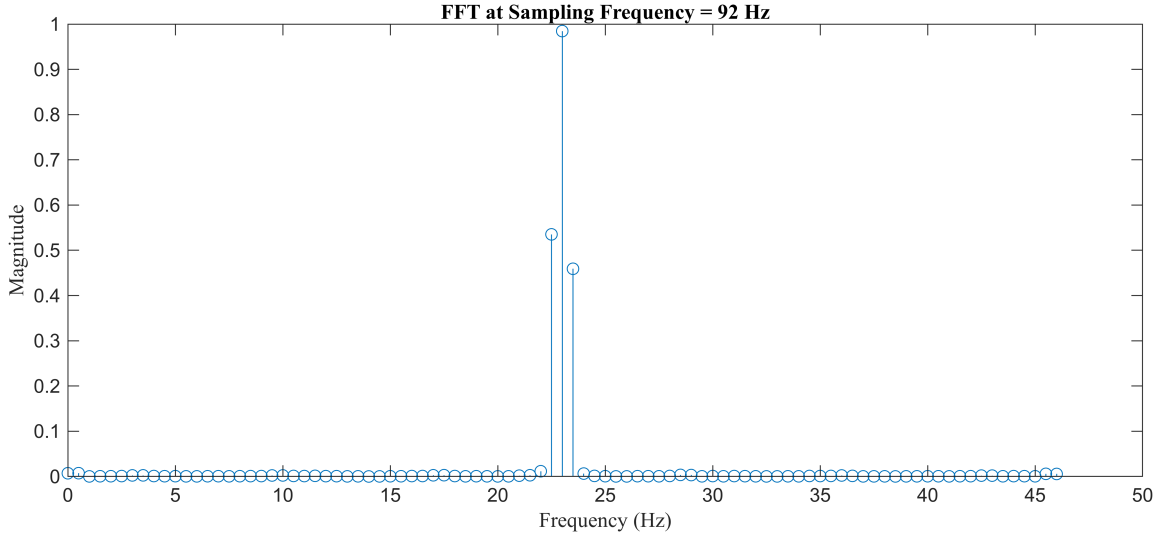

Figure 13: FFT of the signal sampled at 92 Hz. The main frequency component at 23 Hz is clearly identifiable.

1.2.6 Sampling Frequency: $f_s = 120$ Hz

In the final test, the sampling frequency is set to 120 Hz, which is significantly higher than the signal frequency of 23 Hz. According to the Nyquist criterion, this rate is sufficient to capture the waveform without aliasing.

Figure 14 shows a clear representation of the desired signal overlaid with the data acquired from the Arduino. While the number of samples is much greater than previous tests, the red curve (Arduino signal) appears more jagged. This is likely due to the reduced resolution and increased noise sensitivity in the analog-to-digital conversion process at high sampling speeds.

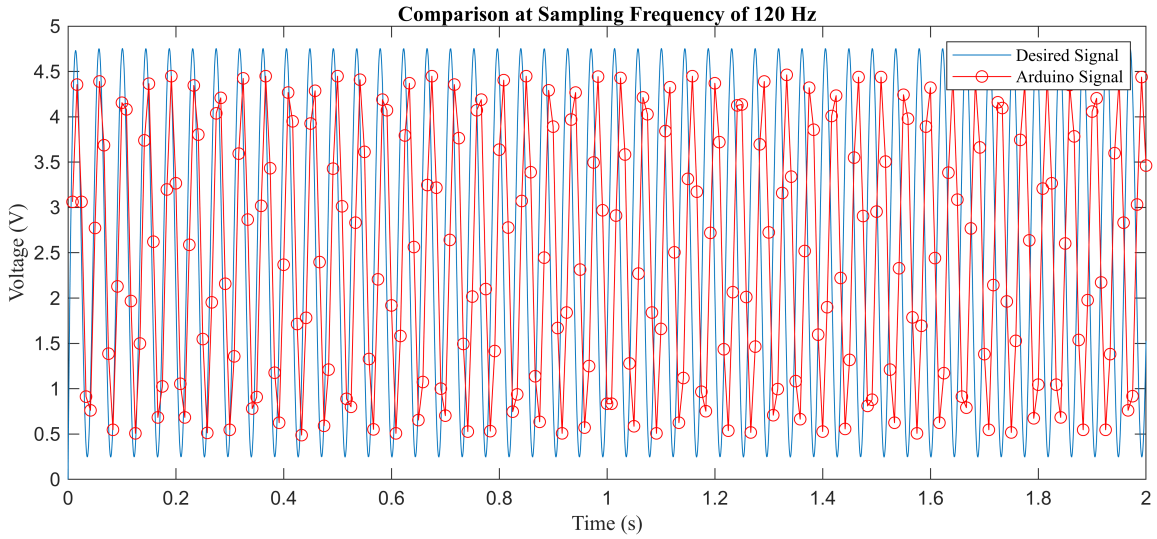

Figure 14: Comparison between desired and acquired signal at $f_s = 120$ Hz

The corresponding FFT result in Figure 15 indicates a dominant frequency component centered very close to 23 Hz, as expected. Despite minor side peaks, the overall frequency content aligns well with the original signal. This confirms that the sampling process accurately captures both the time-domain and frequency-domain behavior of the waveform when an appropriate sampling frequency is used.

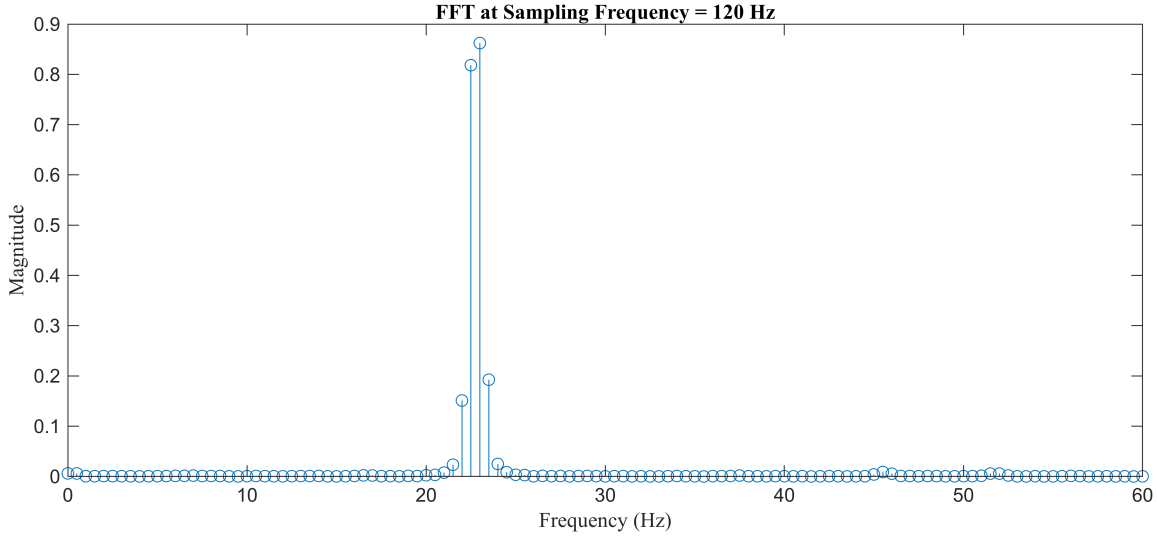

Figure 15: FFT at Sampling Frequency = 120 Hz

2 Assignment 2: Low-Pass Filter

2.1 Introduction

In this lab session, we aim to analyze a signal consisting of a low-frequency sinusoid masked by a high-frequency sinusoid of equal amplitude, focusing on sampling and frequency analysis. The tasks include generating this signal, sampling it with Arduino Uno at a fixed frequency (limited to 800 samples), applying a Hanning window to reduce spectral leakage, and using MATLAB to acquire, plot, and analyze the data. By performing a Fast Fourier Transform (FFT) on the sampled data, we will extract frequency components and determine the attenuation of the high-frequency signal, which is theoretically calculated using a low-pass filter transfer function. The results will be compared with experimental observations, and any discrepancies will be discussed, reinforcing our understanding of signal processing and hardware interfacing concepts.

2.2 What is sampling?

Sampling is the process of converting a continuous-time signal into a discrete-time signal by measuring its amplitude at regular intervals, known as the sampling period. This is essential for analyzing and processing signals using digital systems such as computers and microcontrollers, which can only handle discrete data. The sampling frequency, defined as the number of samples taken per second, must be chosen carefully to capture the essential information in the signal without introducing distortion or aliasing.

2.3 What is FFT?

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT) of a discrete-time signal. It decomposes a signal into its constituent frequency components, revealing how much of each frequency is present in the signal. By transforming the time-domain sampled data into the frequency domain, we can analyze the signal's spectral characteristics, identify dominant frequencies, and measure attenuation at specific frequencies. In this lab session, we use the FFT to observe the frequency spectrum of the sampled signal.

2.4 What is hanning window?

A Hanning window is a type of window function used in signal processing to reduce spectral leakage when performing a Fast Fourier Transform (FFT). It is applied to the sampled signal to smoothly taper the signal's edges towards zero, which minimizes discontinuities at the signal's boundaries. Without a window, sharp transitions at the edges can cause leakage, where energy from one frequency spreads into others, making the frequency analysis less accurate. The Hanning window is defined mathematically to have a cosine-shaped envelope, ensuring a smooth transition. In this lab session, we use the Hanning window on the sampled data before applying the FFT, which enhances the clarity of the frequency peaks and improves the accuracy of the attenuation measurements by reducing spectral leakage.

2.5 Circuit description

The circuit for the second assingment is just the circuit for the first assignment as shown in Figure 2 and Figure 3. The signal that was generated in assignment 2 of lab session 4 was a combination of two signals, one of which was a low frequency signal with a frequency of 6 Hz and the other was a hight frequency signal with a frequency of 84 Hz. The combination of these two signals which was the desired result of assignment 2 of lab session 3 is shown in Figure 16

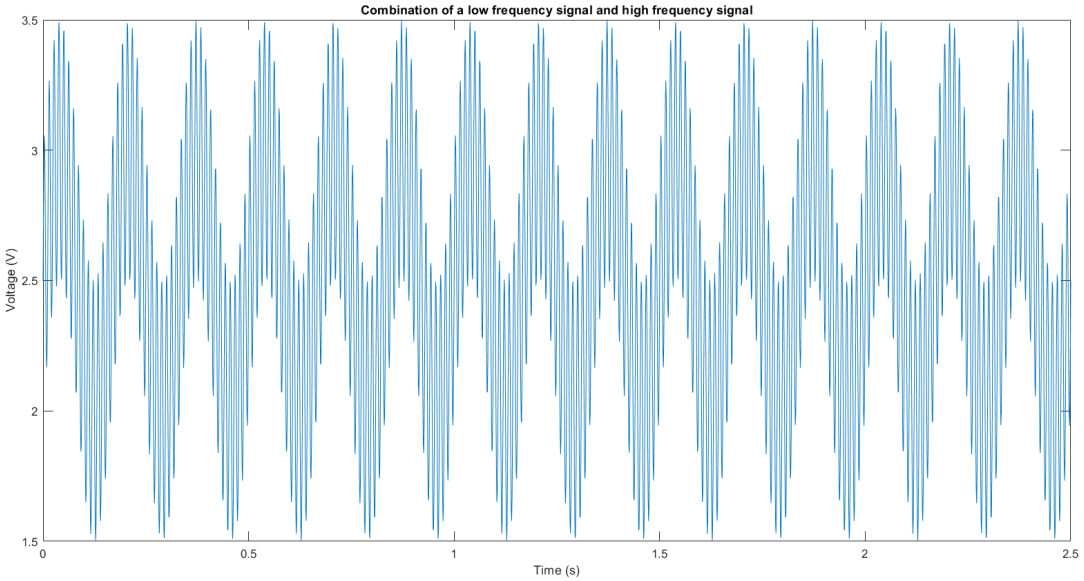

Figure 16: Combination of the high- and low-frequency signals in the sampled signal

The input and output of the signal throught the low pass filter with the corner frequency of:

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi \times 33 \times 100 \times 10^{-6}} = 48.2288 \text{ Hz}$$

is shown in Figure 17

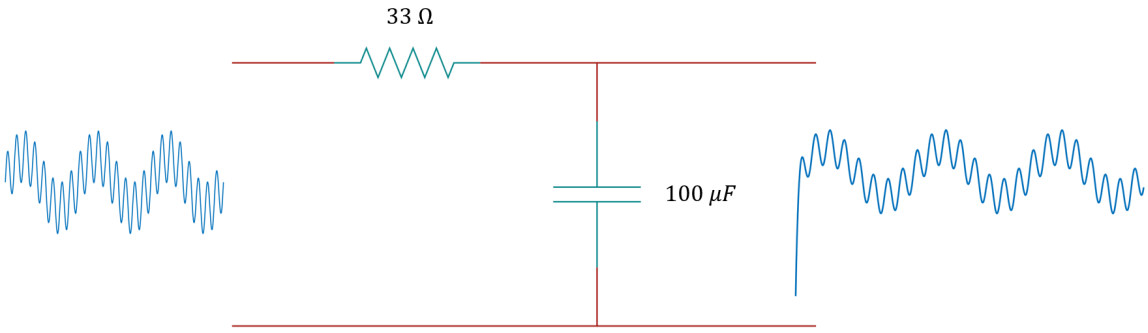

Figure 17: Low-pass filter circuit with 33 Ω resistor and 100 µF capacitor for assignment 2.

3 Sampling the Signal

In the plot below, the red signal represents the output signal generated by the Arduino system, while the blue signal corresponds to the desired signal generated in MATLAB. The Arduino output is the sampled signal processed through the implemented low-pass filter, capturing both the low-frequency and high-frequency components but attenuating the high-frequency part. The blue signal, generated using theoretical models in MATLAB, represents the ideal response of the system to the combined low-frequency (6 Hz) and high-frequency (84 Hz) input signal. To ensure that the signal is accurately sampled and the high-frequency component is preserved, a sampling frequency of 400 Hz was selected, which exceeds twice the highest frequency component as required by the Nyquist criterion. Due to the limit of 800 data points, this sampling frequency resulted in a total sampling time of 2 seconds. By comparing

the two signals, we can evaluate how closely the Arduino hardware replicates the expected filter behavior and identify any discrepancies or distortions introduced during the physical implementation.

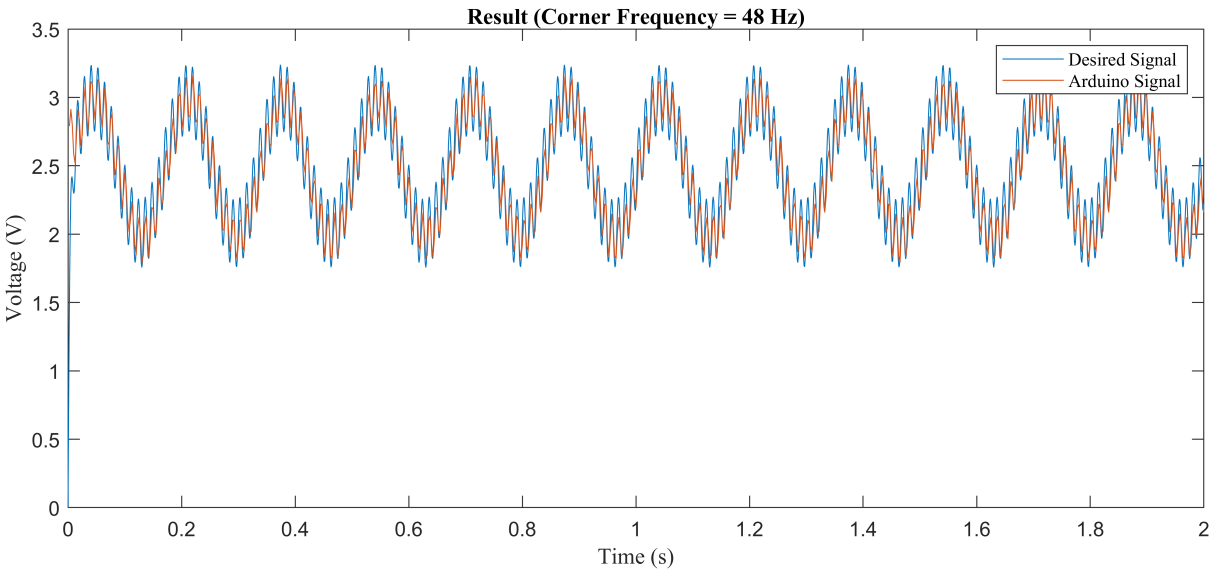

Figure 18: MATLAB generated signal VS. Arduino sampled signal.

4 Applying Hanning Window

In the MATLAB code, the Hanning window is applied to the Arduino-sampled signal to reduce spectral leakage before performing the Fast Fourier Transform (FFT). The signal stored in the variable `b` represents raw analog-to-digital conversion values read from the Arduino Uno. It is first scaled to voltage using `b1 = b/1023*5`, where 1023 is the maximum ADC value and 5 V is the reference voltage. The adjustment `b1 - 2.5` centers the signal around zero by subtracting the DC offset of 2.5 V, which is the midpoint of the signal's range. This centering is crucial because the FFT assumes the input signal fluctuates around zero. The Hanning window, created with `hann(N)`, is then multiplied with the zero-centered signal to create a tapered version of the data, stored in `hanned`. This tapering reduces discontinuities at the signal's edges, thereby minimizing the leakage of energy across frequency bins in the FFT output and improving the clarity and accuracy of the spectral analysis.

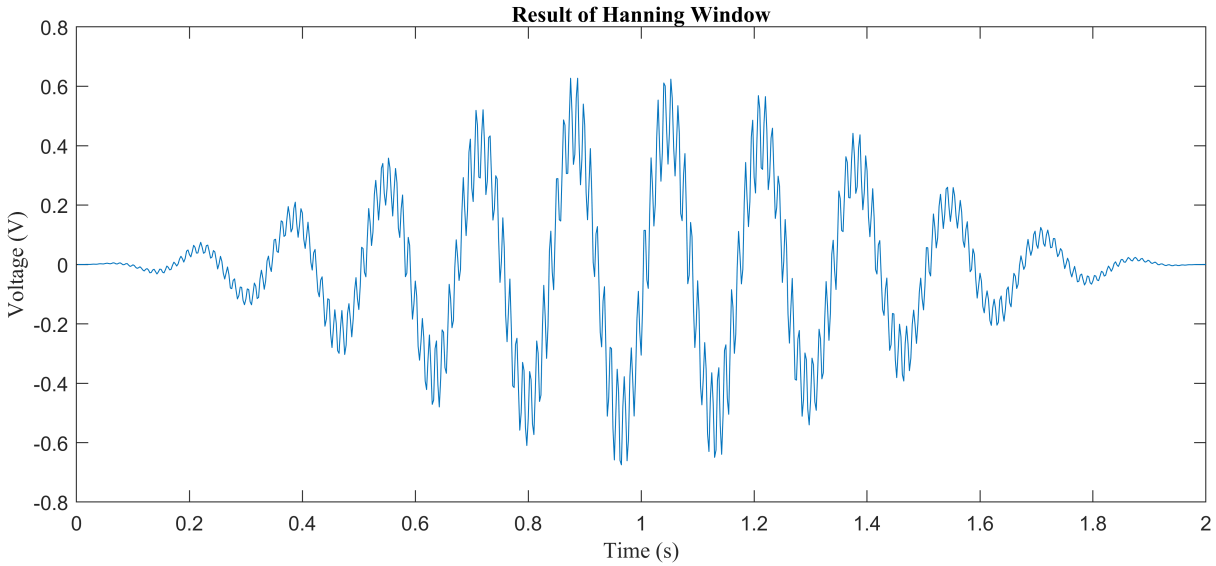

Figure 19: The result of implementing hanning window on the sampled signal generated by arduino

5 Applyuing Fast Fourier Transform (FFT) Analysis

The Fast Fourier Transform (FFT) is applied to the sampled signal `b1` to analyze its frequency components. The FFT is computed using `Y = fft(b1)`, transforming the time-domain signal into its frequency-domain representation. The DC component (at index 1) is halved with `Y(1) = Y(1)/2` to correct its magnitude. The two-sided amplitude spectrum is computed with `P2 = abs(Y/(N/2))`, and the single-sided spectrum is extracted using `P1 = P2(1:N/2+1)`. To account for energy loss due to symmetry, the amplitudes (except the first and last) are doubled with `P1(2:end-1) = 2*P1(2:end-1)`. The frequency axis `f` is computed based on the sampling frequency `Fs` and the total number of points `N`. The resulting frequency spectrum is visualized using a stem plot with labeled axes and an appropriate title, showing the signal's attenuation at different frequencies. The result is shown in Figure 20

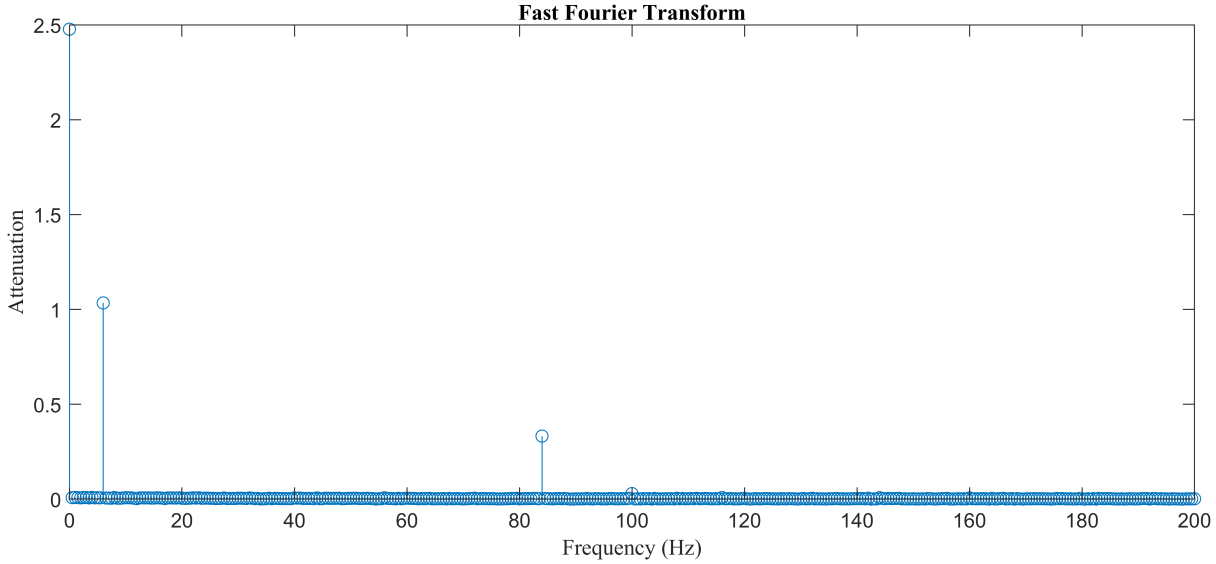

Figure 20: Applying FFT on the arduino signal to show the frequency spectrum

6 FFT Peak Detection and Attenuation Analysis

Experimental Peak Detection and Results

The peaks in the frequency spectrum were identified using the `findpeaks` function in MATLAB:

```
[pks, locs] = findpeaks(P1, 'MinPeakHeight', 0.01);
peak_freqs = f(locs);
peak_amps = pks;
disp(table(peak_freqs', peak_amps, 'VariableNames', {'Frequency_Hz', 'Amplitude'}));
```

This code finds local maxima in the amplitude spectrum `P1` above a threshold of 0.01, retrieves the corresponding frequencies `f(locs)`, and prints them in a table. The detected peaks and their amplitudes were:

Frequency (Hz)	Amplitude
6	1.0343
84	0.3323
100	0.0292

Table 1: Experimental FFT Peaks

Theoretical Attenuation Calculations

The cutoff frequency of the low-pass filter is given by:

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi \times 33 \times 10^{-4}} = 48.2288 \text{ Hz}$$

The theoretical transfer function (magnitude response) of a first-order low-pass filter is:

$$H(f) = \frac{1}{\sqrt{1 + \left(\frac{f}{f_c}\right)^2}}$$

Thus, the theoretical attenuations at 6 Hz and 84 Hz are:

$$H(6) = \frac{1}{\sqrt{1 + \left(\frac{6}{48.2288}\right)^2}} = \frac{1}{\sqrt{1 + 0.0155}} = \frac{1}{\sqrt{1.0155}} \approx 0.9924$$

$$H(84) = \frac{1}{\sqrt{1 + \left(\frac{84}{48.2288}\right)^2}} = \frac{1}{\sqrt{1 + 3.0313}} = \frac{1}{\sqrt{4.0313}} \approx 0.4979$$

These theoretical values were calculated in MATLAB using:

```
fc = 1/(2*pi*R*C);
H_theoretical = @(f) 1 ./ sqrt(1 + (f./fc).^2);
atten_6 = H_theoretical(6);
atten_84 = H_theoretical(84);
```

which produced:

Theoretical attenuation at 6 Hz : 0.9924

Theoretical attenuation at 84 Hz : 0.4979

Comparison and Explanation

Comparing the experimental and theoretical results:

- At 6 Hz, the experimental amplitude was 1.0343 and the theoretical attenuation was approximately 0.9924. The slight difference may be due to system noise, sampling inaccuracies, or windowing effects.
- At 84 Hz, the experimental amplitude was 0.3323 and the theoretical attenuation was approximately 0.4979. The experimental result shows more attenuation than theoretically predicted. This could be caused by hardware imperfections in the Arduino setup, signal distortion, or limitations in the sampling process.

The presence of an additional peak at 100 Hz with a small amplitude (0.0292) is likely due to noise or harmonic components not accounted for in the theoretical model. Overall, while the experimental data aligns reasonably well with theoretical predictions, minor discrepancies highlight the impact of real-world imperfections.

7 Attachments

7.1 Assignment 1 Arduino Code

```
#include <TimerOne.h>      // TimerOne library for precise timing using Timer1
#include <MsTimer2.h>      // MsTimer2 library for using Timer2 with millisecond
    resolution

/******** Constants ********/
#define PWMOUT 5           // PWM output pin
#define ADC_CHANNEL A0     // ADC input pin

// Sine Wave Generation
const int SINE_FREQ = 23;                       // Frequency of the generated
    sine wave (Hz)
const int SINE_GEN_FREQ = 1000;                 // Frequency of the sine
    generation routine (Hz)
const float SINE_AMP = 2.5;                     // Amplitude of sine wave in
    volts
const float SINE_OFFSET = 2.5;                  // Offset of sine wave in volts (
    centers around 2.5 V)

// Sampling
const int SAMPLING_FREQ = 120;                  // Sampling frequency (Hz)
const float TOTAL_SAMPLE_TIME = 2;              // Total sampling duration (
    seconds)

// Precomputed coefficients
const float SINE_COEFF     = SINE_FREQ*TWO_PI/SINE_GEN_FREQ;     // Angular step
    for sine wave
const float SINE_AMP_A     = SINE_AMP/5.00*255;                  // Scaled
    amplitude for 8-bit PWM
const float SINE_OFFSET_A = SINE_OFFSET/5.00*255;                // Scaled
    offset for 8-bit PWM
const int NUMBER_OF_SAMPLES = TOTAL_SAMPLE_TIME*SAMPLING_FREQ;  // Total number
    of samples to acquire

// Global variables
int DUTY_CYCLE, SAMPLES[NUMBER_OF_SAMPLES];  // PWM duty and sample array
long SINE_GEN_COUNTER, SAMPLING_COUNTER;     // Counters for sine generation
    and sampling
bool FINISHED=false;                         // Flag to indicate data
    acquisition complete

void setup () {
  // Initialize sample array with -1
  for (int i = 0 ; i < NUMBER_OF_SAMPLES ; i++) {
    SAMPLES[i] = -1;
  }

  // Set Timer0 to normal fast PWM
  TCCR0B = TCCR0B & B11111000 | B00000001;

  pinMode(PWMOUT, OUTPUT);  // Set PWM output pin

  Serial.begin(57600);      // Initialize serial communication
  Serial.begin(57600);      // Redundant, but harmless

  // Setup sine wave generation timer interrupt
  MsTimer2::set(1000/SINE_GEN_FREQ, PWM_DT); // Call PWM_DT every 1 ms
  MsTimer2::start();
```

```
  // Setup sampling timer interrupt
  Timer1.initialize(1e6/SAMPLING_FREQ);       // Configure Timer1 based on
      sampling frequency
  Timer1.attachInterrupt(SAMPLING);           // Attach the sampling function to
       Timer1
}

void loop() {
  // Once sampling is complete, send data via Serial
  if (FINISHED) {
    for (int i = 0 ; i < NUMBER_OF_SAMPLES ; i++) {
      Serial.println(SAMPLES[i]);  // Send each sample
    }
    FINISHED = false;  // Reset flag after transmission
  }
}

void PWM_DT () {
  // Generate PWM duty cycle value based on sine wave
  DUTY_CYCLE = SINE_AMP_A * sin(SINE_COEFF*SINE_GEN_COUNTER) + SINE_OFFSET_A;
  analogWrite(PWMOUT, DUTY_CYCLE);  // Output PWM signal
  SINE_GEN_COUNTER++;               // Increment phase
}

void SAMPLING () {
  if (SAMPLING_COUNTER == NUMBER_OF_SAMPLES) {
    // Stop timers when enough samples are collected
    MsTimer2::stop();
    Timer1.detachInterrupt();
    FINISHED = true;
  } else {
    // Read analog value from ADC and store it
    SAMPLES[SAMPLING_COUNTER] = analogRead(ADC_CHANNEL);
    SAMPLING_COUNTER++;
  }
}
```

7.2 Assignment 1 MATLAB Code

7.2.1 Assignment 1 MATLAB Code For Receiving Data

```
clc, clear
fs = 120;
SamplingTime = 2;
N = SamplingTime*fs;
a = strings(N,1);

s = serialport("COM8",57600);
configureTerminator(s,"CR/LF");

for i = 1:N
    a(i) = readline(s);
end

Lab4_Ass1_fs120 = double(a);
```

7.2.2 Assignment 1 MATLAB Code For Processing

```
clc, clear


fs = 120;  % Change this only

SamplingTime = 2;
N = SamplingTime * fs;

% --- Load data ---
dataVarName = sprintf("Lab4_Ass1_fs%d", fs);  % Variable name in .mat file
fileName = sprintf("Lab4_Ass1_fs%d.mat", fs);  % File name
load(fileName);  % loads variable with name Lab4_Ass1_fsXX
rawData = eval(dataVarName);  % Extract that variable dynamically

% --- Time vector and scaling ---
t = 1/fs:1/fs:SamplingTime;
y = rawData / 1024 * 5;
tt = 0:0.001:2;
syms t_sym s
R = 33;
C = 1e-4;
G1 = 1/(1+R*C*s);
y_sym = 2.5 * sin(2*pi*23*t_sym) + 2.5;
G2 = laplace(y_sym);
out(t_sym) = ilaplace(G1*G2,t_sym);
yy = out(tt);

% --- Plot 1: Raw Sample Plot ---
fig1 = figure('Name', sprintf("fs = %d Hz", fs));
plot(t, y)
title(sprintf("Result with a Sampling Frequency of %d Hz", fs), 'FontName', 'Times New Roman');
xlabel("Time (s)", 'FontName', 'Times New Roman');
ylabel("Voltage (V)", 'FontName', 'Times New Roman');
```

```
grid on
set(fig1, 'Units', 'Inches', 'Position', [0, 0, 10, 4], 'PaperPositionMode', 'auto');
% exportgraphics(fig1, sprintf('f%d.png', fs), 'Resolution', 600);

% --- Plot 2: Compare with Desired Signal ---
fig2 = figure();
plot(tt, yy);
hold on;
plot(t, y, '-o', 'Color', 'red');
title(sprintf("Comparison at Sampling Frequency of %d Hz", fs), 'FontName', 'Times New Roman');
xlabel("Time (s)", 'FontName', 'Times New Roman');
ylabel("Voltage (V)", 'FontName', 'Times New Roman');
legend("Desired Signal", "Arduino Signal", 'FontName', 'Times New Roman');
hold off;
set(fig2, 'Units', 'Inches', 'Position', [0, 0, 10, 4], 'PaperPositionMode', 'auto');
exportgraphics(fig2, sprintf('f%d_compare.png', fs), 'Resolution', 600);

% --- Plot 3: FFT Analysis ---
hann_win = hann(N);
Y = fft((y - 2.5) .* hann_win);
P2 = abs(Y/N);
P1 = P2(1:N/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = fs*(0:(N/2))/N;


fig3 = figure();
stem(f, P1');
title(sprintf("FFT at Sampling Frequency = %d Hz", fs), 'FontName', 'Times New Roman');
xlabel("Frequency (Hz)", 'FontName', 'Times New Roman');
ylabel("Magnitude", 'FontName', 'Times New Roman');
set(fig3, 'Units', 'Inches', 'Position', [0, 0, 10, 4], 'PaperPositionMode', 'auto');
exportgraphics(fig3, sprintf('f%d_fft.png', fs), 'Resolution', 600);
```

7.3 Assignment 2 Arduino Code

```
#include <TimerOne.h>
#include <MsTimer2.h>

/******** Constants ********/
#define PWMOUT 5
#define ADC_CHANNEL A0


// Sine Wave Generation
const int SINE_GEN_FREQ = 1000;                 // Frequency of the sine wave
    generation system. Hz.
const float SINE_AMP = 0.5;                    // Sine wave's amplitude to be
    generated. Volts.
const float SINE_OFFSET = 2.5;                 // Sine wave's offset to be
    generated. Volts.

// Sampling
const int SAMPLING_FREQ = 400;
const float TOTAL_SAMPLE_TIME = 2;

const float SINE_AMP_A    = SINE_AMP/5.00*255;
const float SINE_OFFSET_A = SINE_OFFSET/5.00*255;
const int NUMBER_OF_SAMPLES = TOTAL_SAMPLE_TIME*SAMPLING_FREQ;

int DUTY_CYCLE, SAMPLES[NUMBER_OF_SAMPLES];
long SINE_GEN_COUNTER, SAMPLING_COUNTER;
bool FINISHED=false;

int Sine_Freq_Low = 6;      // Low-frequency sine wave (Hz)
int Sine_Freq_High = 84;    // High-frequency sine wave (Hz)

float Sine_Coeff_Low = Sine_Freq_Low * TWO_PI / 1000.00;
float Sine_Coeff_High = Sine_Freq_High * TWO_PI / 1000.00;


void setup () {
  for (int i = 0 ; i < NUMBER_OF_SAMPLES ; i++) {
    SAMPLES [i] =- 1;
  }
  TCCR0B = TCCR0B & B11111000 | B00000001;
  pinMode(PWMOUT, OUTPUT);

  Serial.begin(57600);
  Serial.begin(57600);

  MsTimer2::set(1000/SINE_GEN_FREQ, PWM_DT) ;
  MsTimer2::start();

  Timer1.initialize(1e6/SAMPLING_FREQ);
  Timer1.attachInterrupt(SAMPLING);
}


void loop() {
  // put your main code here, to run repeatedly:
  if (FINISHED) {
    for (int i = 0 ; i < NUMBER_OF_SAMPLES ; i++) {
      Serial.println(SAMPLES[i] ) ;
    }
    FINISHED=false;
  }
```

```
}


void PWM_DT () {
  DUTY_CYCLE = SINE_AMP_A * sin(Sine_Coeff_Low * SINE_GEN_COUNTER) +
               SINE_AMP_A * sin(Sine_Coeff_High * SINE_GEN_COUNTER) +
               SINE_OFFSET_A;
  analogWrite (PWMOUT, DUTY_CYCLE) ;
  SINE_GEN_COUNTER++;
}
void SAMPLING () {
if (SAMPLING_COUNTER == NUMBER_OF_SAMPLES) {
  MsTimer2::stop();
Timer1.detachInterrupt() ;
FINISHED=true;
}else {
SAMPLES [SAMPLING_COUNTER]=analogRead(ADC_CHANNEL);
SAMPLING_COUNTER++;
}
}
```

7.4 Assignment 2 MATLAB Code

```
clc, clear
load('Lab4_Ass2.mat');
R = 33;
C = 1e-4;
Time = 2;
Fs = 400;
N = Time*Fs;

% a = strings(N,1);
%
% s = serialport("COM8",57600);
% configureTerminator(s,"CR/LF");
% for i = 1:N
%     a(i,:) = readline(s);
% end
% b = double(a);
% clear s
b = Lab4_Ass2;

close all
syms t s
tt = 0 : 0.001 : 2;
generated = 0.5*sin(2*pi*6*t)+0.5*sin(2*pi*84*t)+2.5;
G1 = laplace(generated);
G = 1/(R*C*s+1);
clear f
f(t) = simplify(ilaplace(G1*G,t));

fig1 = figure("Name","b");
plot(tt,f(tt));
hold on
b1 = b/1023*5;
tt = 1/Fs : 1/Fs : Time;
plot(tt, b1)
title("Result (Corner Frequency = 48 Hz)", "FontName", "Times New Roman");
xlabel("Time (s)", "FontName", "Times New Roman");
ylabel("Voltage (V)", "FontName", "Times New Roman");
set(fig1, 'Units', 'Inches');
legend("Desired Signal", "Arduino Signal", "FontName", ...
    "Times New Roman")
set(fig1, 'Position', [0, 0, 10, 4]);
set(fig1, 'PaperPositionMode', 'auto');
exportgraphics(fig1, 'Assignment2_Response.png', 'Resolution', 600);
hold off
hnning_var = hann(N);

hanned = (b1-2.5).*hnning_var;

plot(tt, b1)
plot(tt, hanned)


Y=fft(b1);

Y(1) = Y(1)/2;
P2 = abs(Y/(N/2));
```

```
P1 = P2(1:N/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(N/2))/N;
fig2 = figure("Name","a");
stem(f,P1)
title("Fast Fourier Transform", "FontName", "Times New Roman");
xlabel("Frequency (Hz)", "FontName", "Times New Roman");
ylabel("Attenuation", "FontName", "Times New Roman");
set(fig2, 'Units', 'Inches');
set(fig2, 'Position', [0, 0, 10, 4]);
set(fig2, 'PaperPositionMode', 'auto');
exportgraphics(fig2, 'FFT_Assignemt2.png', 'Resolution', 600);

[pks, locs] = findpeaks(P1, 'MinPeakHeight', 0.01); % You may need to adjust threshold
peak_freqs = f(locs);
peak_amps = pks;

% Display peaks
disp(table(peak_freqs', peak_amps, 'VariableNames', {'Frequency_Hz', 'Amplitude'}));

fc = 1/(2*pi*R*C);  % Example cutoff frequency in Hz

H_theoretical = @(f) 1 ./ sqrt(1 + (f./fc).^2);

% Calculate theoretical gains at 6 Hz and 84 Hz
atten_6 = H_theoretical(6);
atten_84 = H_theoretical(84);

fprintf("Theoretical attenuation at 6 Hz: %.4f\n", atten_6);
fprintf("Theoretical attenuation at 84 Hz: %.4f\n", atten_84);
```