



VILNIUS UNIVERSITY  
FACULTY OF MATHEMATICS AND INFORMATICS  
INSTITUTE OF COMPUTER SCIENCE  
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

Course project no. 2

**Fourier filter**  
**Based on Cooley-Tuckey algorithm**

Done by:

Kazimieras Vitkus

Supervisor:

Prof. Tadas Meskauskas

Vilnius  
2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Algorithm implementation</b>	<b>3</b>
2.1	Discrete Fourier Transform (DFT) . . . . .	3
2.2	Fast Fourier Transform (FFT) . . . . .	4
<b>3</b>	<b>Analysis of algorithms and implementation</b>	<b>5</b>
3.1	Comparison of DFT and FFT . . . . .	5
3.2	Comparison of FFT from scratch and numpy FFT . . . . .	6
3.3	Fitting requirements for algorithm reduces performance . . . . .	6
<b>4</b>	<b>Audio signal filtering</b>	<b>7</b>
4.1	High-pass filtering . . . . .	9
4.2	Low-pass filtering . . . . .	10
4.3	Mid-pass filtering . . . . .	11
<b>5</b>	<b>Additional signal examples</b>	<b>12</b>
5.1	Example 1 . . . . .	12
5.2	Example 2 . . . . .	14
5.3	Example 3 . . . . .	17
5.4	Example 4 . . . . .	19
<b>6</b>	<b>Conclusion</b>	<b>22</b>

# 1 Introduction

In the realm of digital signal processing, the Fast Fourier Transform (FFT) and its inverse (IFFT) are pivotal algorithms that enable efficient computation of the Discrete Fourier Transform (DFT) and its inverse (IDFT). These mathematical tools transform time-domain signals into their frequency-domain representations and vice versa, offering profound insights into the frequency components of signals. Implementing these transformations from scratch not only deepens the understanding of their mechanics but also highlights the computational optimizations that FFT introduces over the traditional DFT.

This report details the implementation of FFT, IFFT, DFT, and IDFT algorithms from the ground up. We will explore how these algorithms can be utilized for high-pass and low-pass filtering, allowing us to isolate and analyze specific frequency components within sound signals. By filtering and examining these signals, we gain a better understanding of the underlying structures and characteristics of different audio samples.

## 2 Algorithm implementation

### 2.1 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is a mathematical operation that transforms a finite sequence of equally spaced samples of a function into a sequence of complex numbers. The DFT is defined by the formula:

$$C_k = \sum_{j=0}^{N-1} f_j \cdot e^{-\frac{2\pi i}{N} kn} \quad (2.1)$$

where  $f_j$  is the input signal,  $C_k$  is the output signal, and  $N$  is the number of samples in the input signal. The DFT algorithm computes the frequency components of the input signal by summing the product of each sample with a complex exponential function. In my implementation, I used the following Python code to calculate the DFT of an input signal:

```
1 def dft(signal):
2
3     N = len(signal)
4
5     C_k = np.zeros(N, dtype=np.complex_)
6
7     for k in range(N):
8
9         for n in range(N):
10             C_k[k] += signal[n] * np.exp(-2j * np.pi * k * n / N)
11
12         C_k[k] /= N
13     return C_k
```

for the inverse DFT, we can use the following code:

```
1 def idft(signal):
2
3     N = len(signal)
4
5     f_j = np.zeros(N, dtype=np.complex_)
6
7     for j in range(N):
8
9         for k in range(N):
10             f_j[j] += signal[k] * np.exp(2j * np.pi * k * j / N)
11
12     return f_j
```

## 2.2 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an algorithm that computes the Discrete Fourier Transform (DFT) of a sequence of complex numbers. The FFT is an optimized version of the DFT that reduces the number of arithmetic operations required to compute the DFT from  $O(N^2)$  to  $O(N \log N)$ . The Cooley-Tukey algorithm is a popular implementation of the FFT that recursively divides the input sequence into smaller sub-sequences and combines the results to compute the DFT.

In my implementation, I used the following Python code to calculate the FFT of an input signal:

```
1 def fft(signal):
2     N = len(signal)
3
4     if N <= 1:
5         return signal
6
7     fft_even = fft(signal[::2])
8     fft_odd = fft(signal[1::2])
9
10    k = np.arange(N // 2)
11    W_n_k = np.exp(-2j * np.pi * k / N)
12
13    A_k = fft_even
14    B_k = fft_odd * W_n_k
15
16    C_k = np.zeros(N, dtype=np.complex_)
17    C_k[:N // 2] = A_k + B_k
18    C_k[N // 2:] = A_k - B_k
19
20    return C_k
```

For the inverse FFT, we can use the following code:

```
1 def ifft(signal):
2     N = len(signal)
3
4     if N <= 1:
5         return signal
6
7     ifft_even = ifft(signal[::2])
8     ifft_odd = ifft(signal[1::2])
9
10    k = np.arange(N // 2)
11    W_n_k = np.exp(2j * np.pi * k / N)
12
13    A_k = ifft_even
14    B_k = ifft_odd * W_n_k
15
16    f_j = np.zeros(N, dtype=np.complex_)
17    f_j[:N // 2] = A_k + B_k
18    f_j[N // 2:] = A_k - B_k
19
20    return f_j / N
```

## 3 Analysis of algorithms and implementation

### 3.1 Comparison of DFT and FFT

In theory and practice, the Fast Fourier Transform (FFT) is significantly faster than the Discrete Fourier Transform (DFT) for computing the frequency components of a signal. The FFT algorithm reduces the number of arithmetic operations required to compute the DFT from  $O(N^2)$  to  $O(N \log N)$ , making it much more efficient for large input signals. For the example of comparison, signal was generated using numpy random function with length varying from  $2^2$  to  $2^7$ .

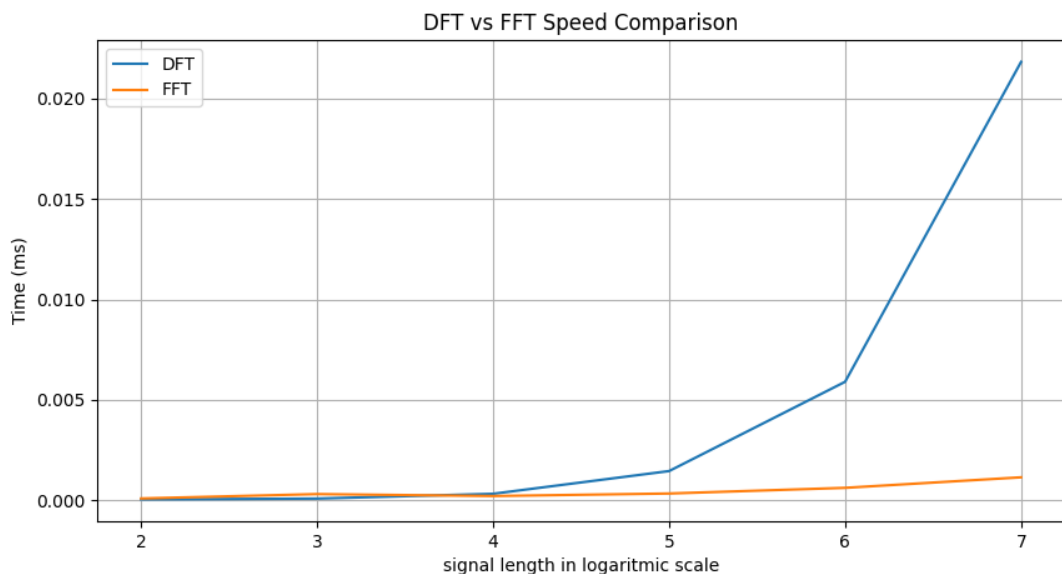


Figure 1. Comparison of computation time between DFT and FFT

As can be seen in figure 1, the FFT algorithm is significantly faster. The trend line for DFT seems to be increasing exponentially, while the FFT trend line is almost linear. This demonstrates the computational efficiency of the FFT algorithm compared to the DFT algorithm. However, the chart may vary depending on the length of the input signal used for comparison.

### 3.2 Comparison of FFT from scratch and numpy FFT

The numpy library provides a built-in FFT function that is highly optimized and efficient for computing the Discrete Fourier Transform (DFT) of a signal. In this section, we compare the performance of the FFT algorithm implemented from scratch with the numpy FFT function. The signal was generated using numpy random function with length varying from  $2^2$  to  $2^{16}$ .

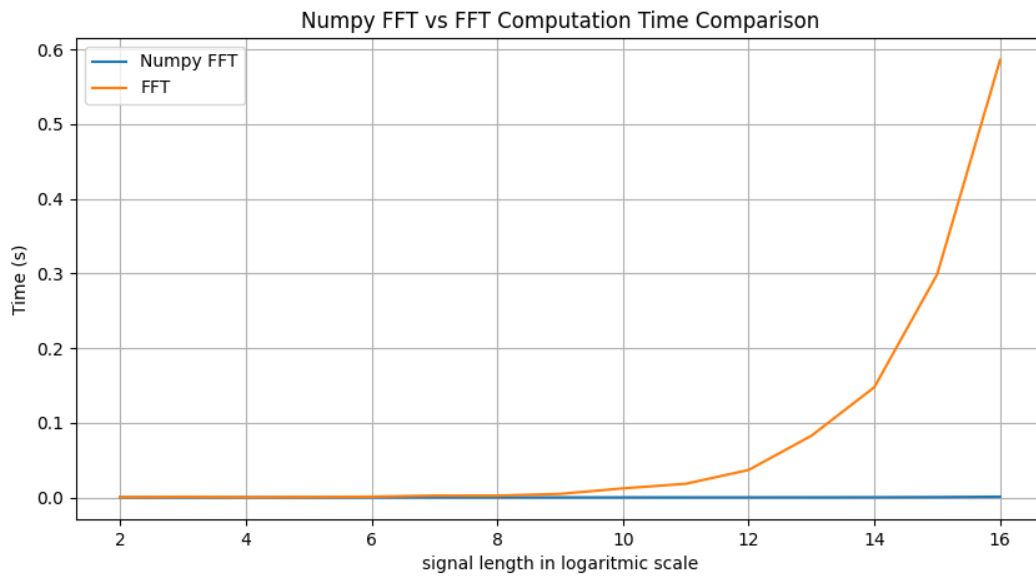


Figure 2. Comparison of computation time between FFT from scratch and numpy FFT

As can be seen in figure 2, the numpy FFT function is significantly faster than the FFT algorithm implemented from scratch. After certain length of the input, implementation from scratch seems to differ exponentially in computation times. It means, implementation quality is suboptimal and needs to be improved.

### 3.3 Fitting requirements for algorithm reduces performance

One of requirements for signal that needs to be processed with FFT - it's length must be a power of 2. One of the provided solutions - zero padding, which means adding zeros to the end of the signal to make it's length a power of 2. However, this solution may reduce the performance of the FFT algorithm. In this section, we compare the performance of the FFT algorithm with and without zero padding. The signal was generated using numpy random function with length varying from  $2^2$  to  $2^{16}$  and for comparison, one signal has 0 appended to invoke zero padding function.

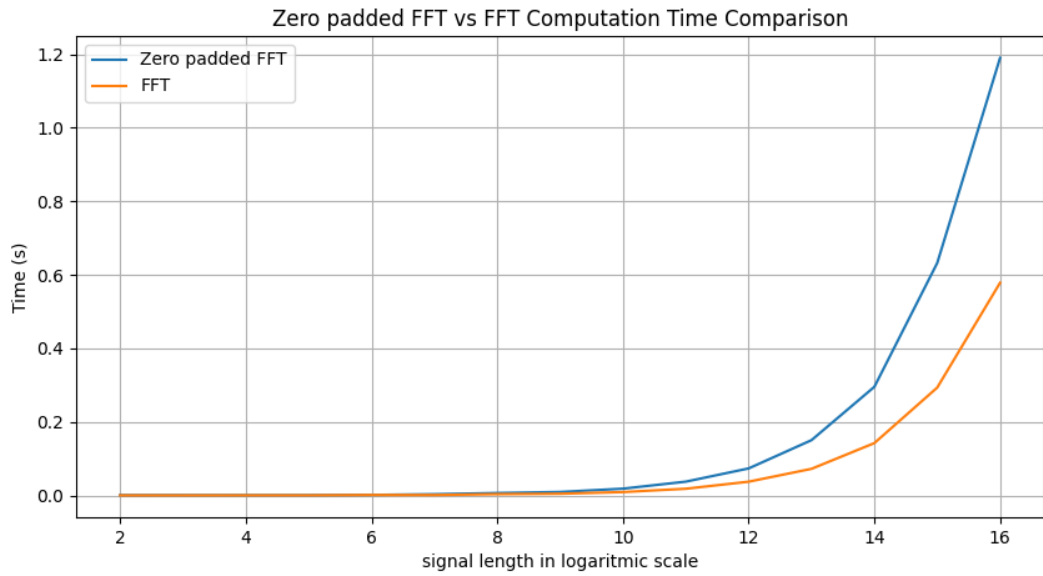


Figure 3. Comparison of computation time between FFT with and without zero padding

As can be seen in figure 3, the signal, that invokes zero padding, takes more time to compute the FFT algorithm. The relation between computation times, is that if zero padding was invoked, the computation equals to the computation time of signal with length  $2^{k+1}$ .

## 4 Audio signal filtering

For the demonstration of the Fourier filter, we will use a sample audio signal. The signal was downloaded from the following website: <https://freesound.org/people/InspectorJ/sounds/412067/>.

The signal in an unprocessed form is presented in figure 4. It is based on framerate of 44100 Hz and has 2 channels. The signal consists of 115327 frames and is of 2.5 seconds duration. The signal is not compressed and saved in the wave (WAV) format.

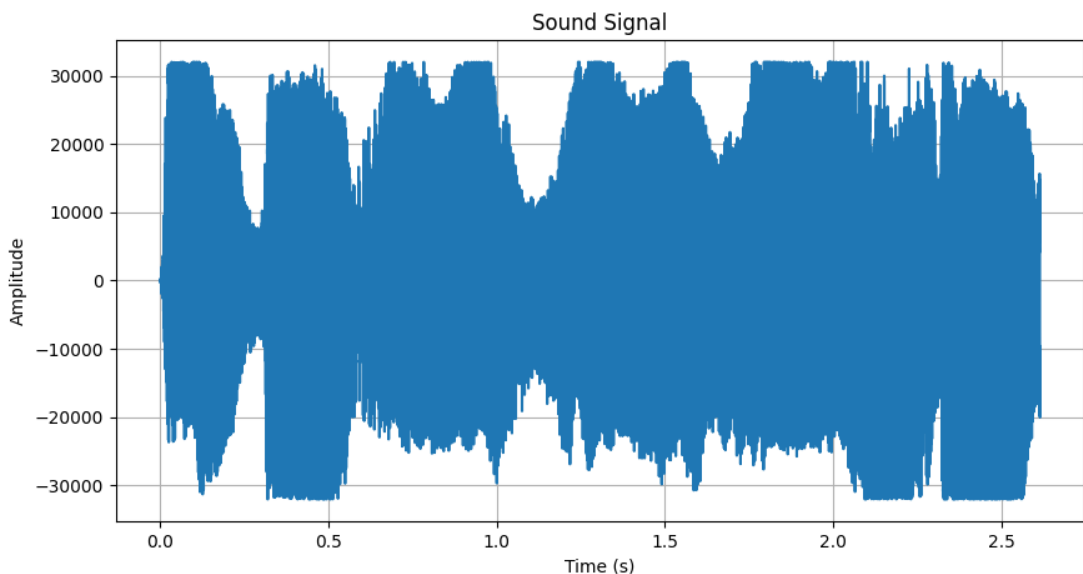


Figure 4. Unprocessed audio signal

The power spectrum of the signal is presented in figure 5. The power spectrum shows the frequency components of the signal. The x-axis represents the frequency in Hz, while the y-axis represents the power of the signal at each frequency in decibels (logarithmic scale). The power spectrum provides insights into the frequency components of the signal and can be used to identify specific frequency ranges for filtering.

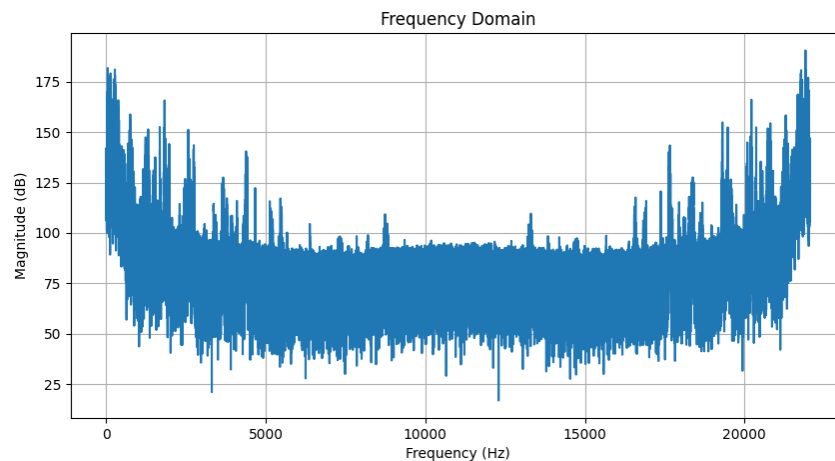


Figure 5. Power spectrum of the audio signal

Also signal when transformed to frequency domain, can also be plotted in histogram form. The histogram of the signal is presented in figure 6. The histogram shows the distribution of the signal values in the frequency domain. The x-axis represents the frequency bins, while the y-axis represents the number of occurrences of each frequency bin. The histogram provides insights into the distribution of frequency components in the signal and can be used to identify specific frequency ranges for filtering.

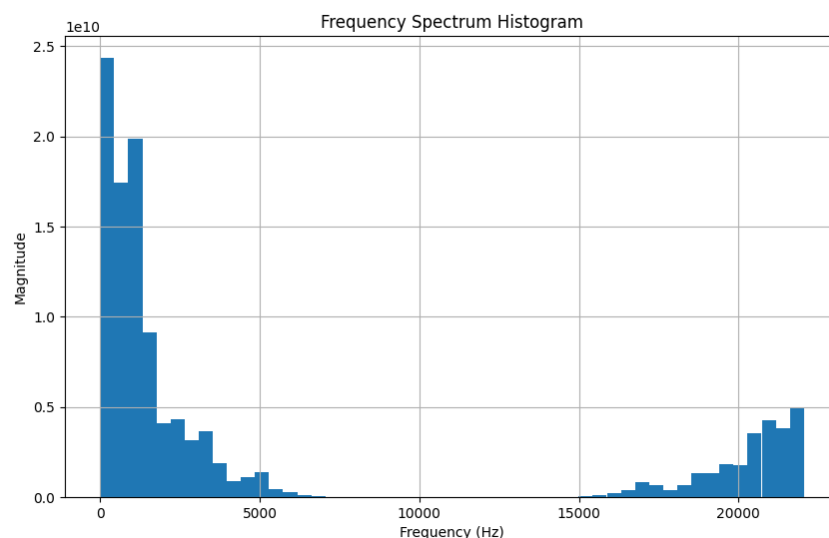


Figure 6. Signal in frequency domain and histogram



## 4.1 High-pass filtering

High-pass filtering is a signal processing technique that allows high-frequency components of a signal to pass through while attenuating low-frequency components. In this section, we will implement a high-pass filter using the Fourier transform and apply it to the audio signal. In the scope of this task, prebuilt function was made, that takes in signal and frequency that is The threshold frequency for the high-pass filter. The function returns the filtered signal.

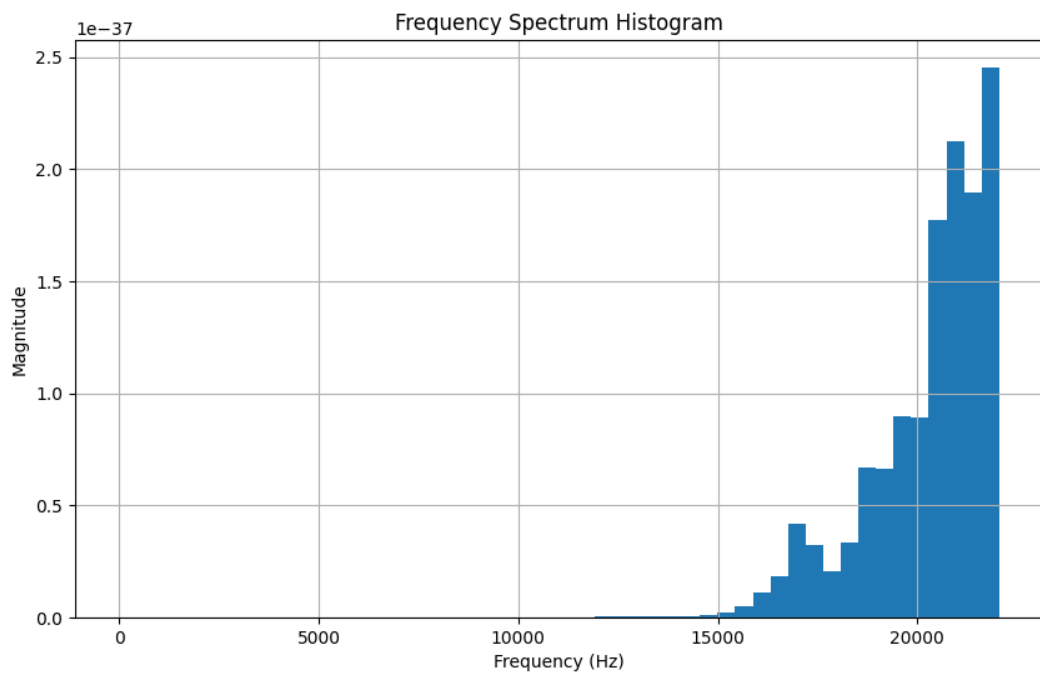


Figure 7. High-pass filtered audio signal

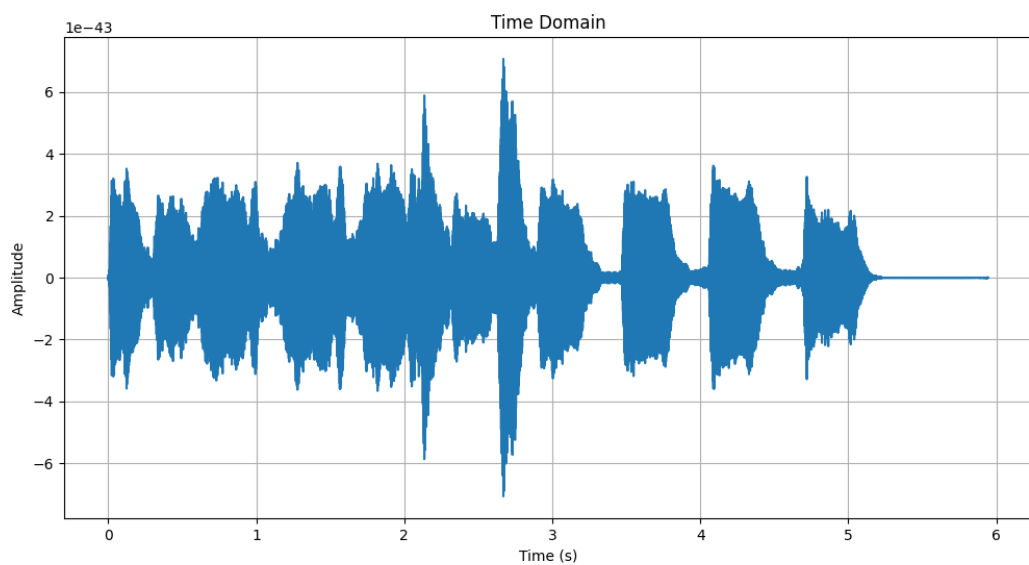


Figure 8. High-pass filtered audio signal reversed to time domain

## 4.2 Low-pass filtering

Low-pass filtering is a signal processing technique that allows low-frequency components of a signal to pass through while attenuating high-frequency components. In this section, we will implement a low-pass filter using the Fourier transform and apply it to the audio signal.

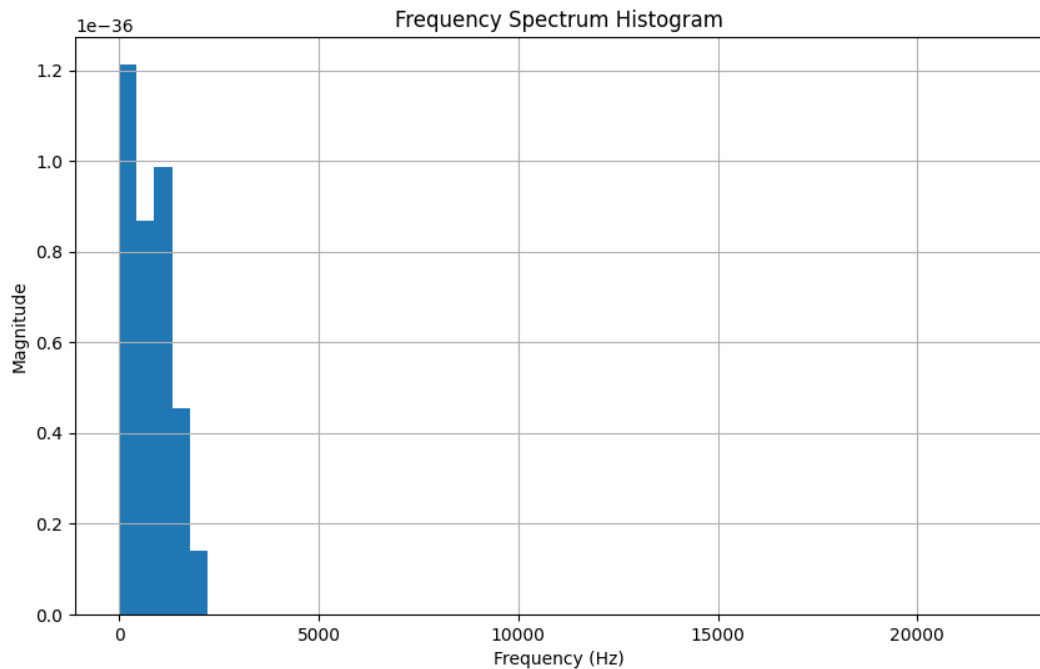


Figure 9. Low-pass filtered audio signal

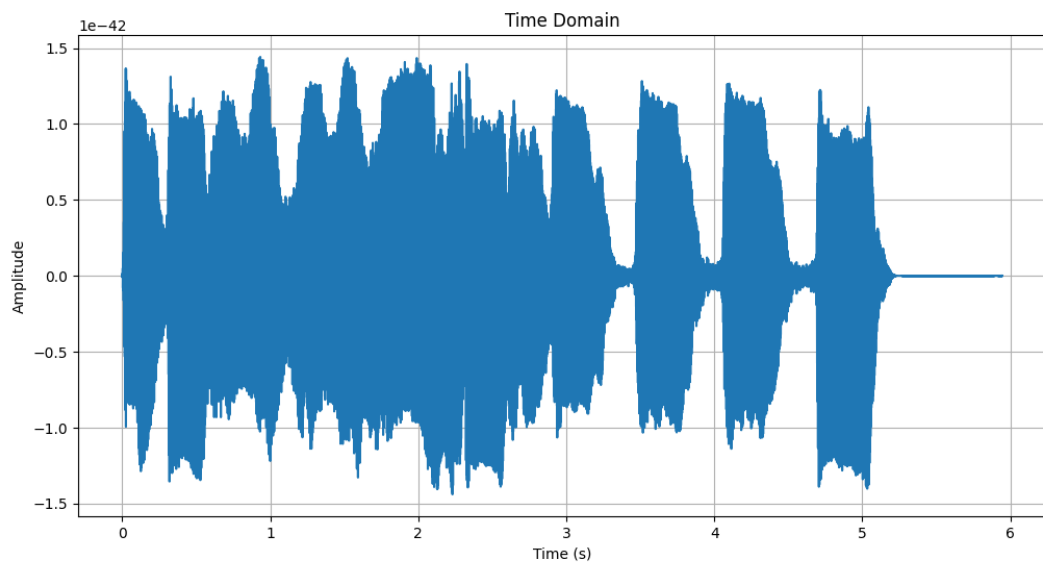


Figure 10. Low-pass filtered audio signal reversed to time domain

### 4.3 Mid-pass filtering

Mid-pass filtering is a signal processing technique that allows mid-frequency components of a signal to pass through while attenuating low and high-frequency components. In this section, we will implement a mid-pass filter using the Fourier transform and apply it to the audio signal.

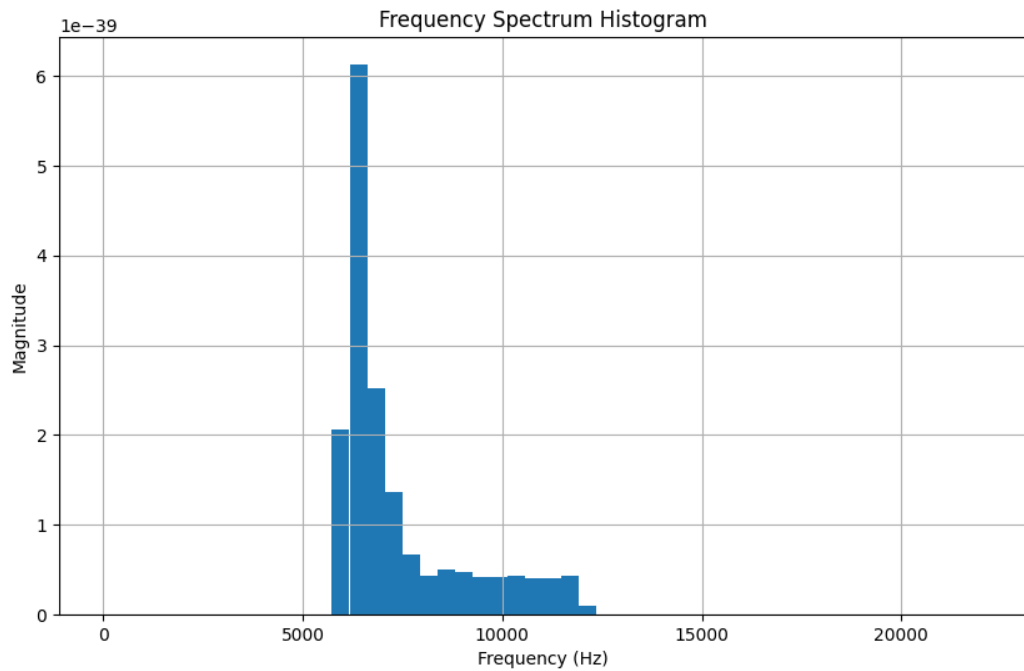


Figure 11. Mid-pass filtered audio signal

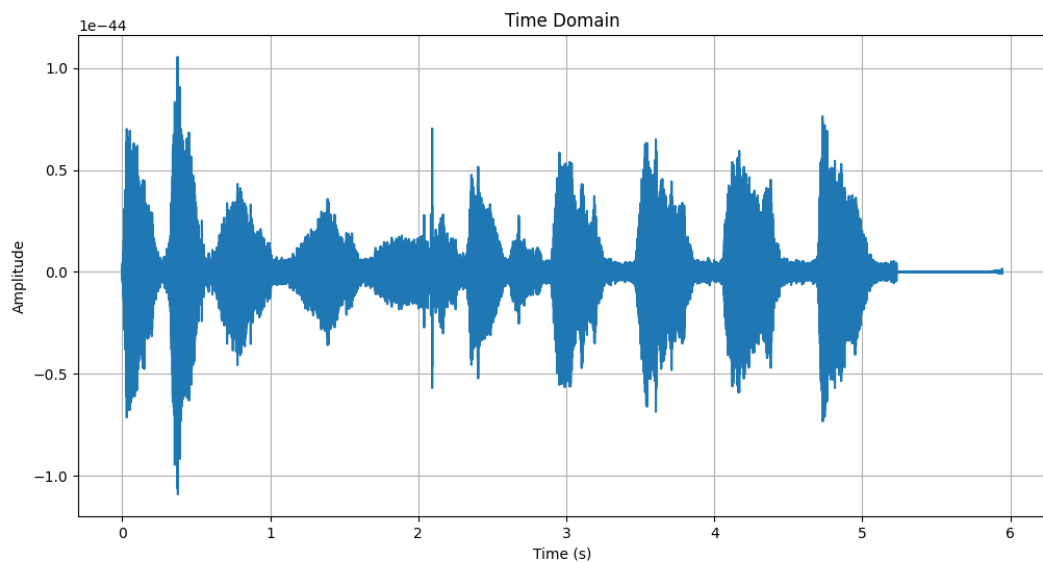


Figure 12. Mid-pass filtered audio signal reversed to time domain

## 5 Additional signal examples

### 5.1 Example 1

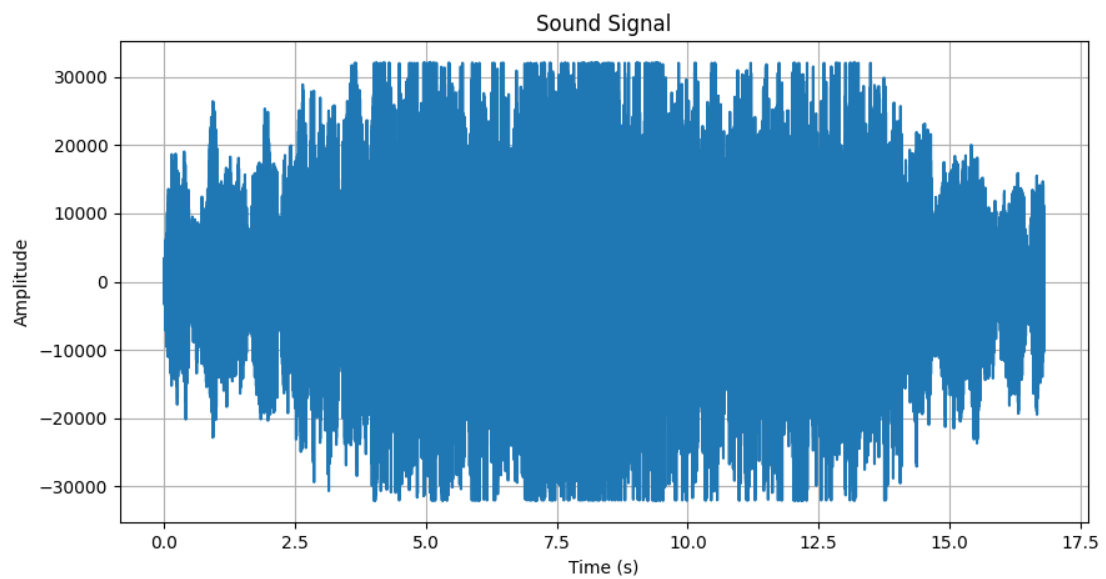


Figure 13. Unprocessed audio signal

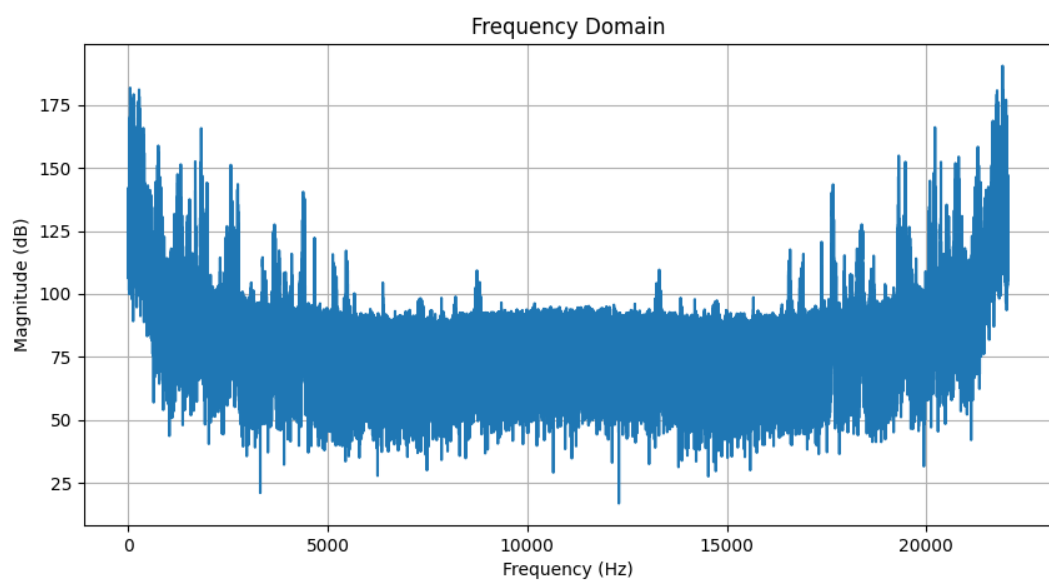


Figure 14. Power spectrum of the audio signal

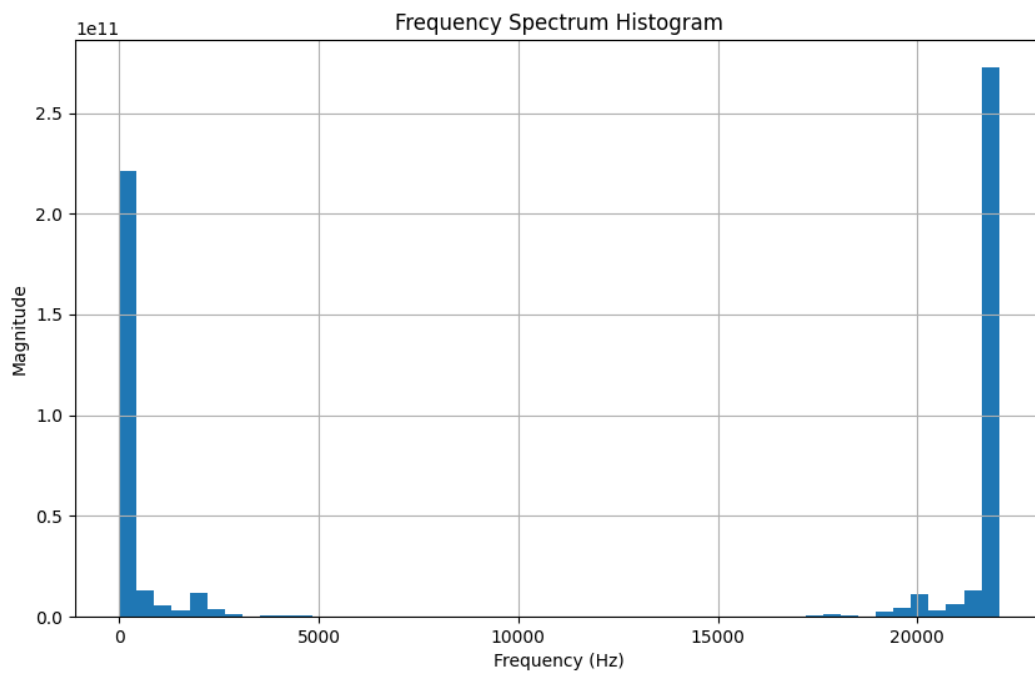


Figure 15. Signal in frequency domain and histogram

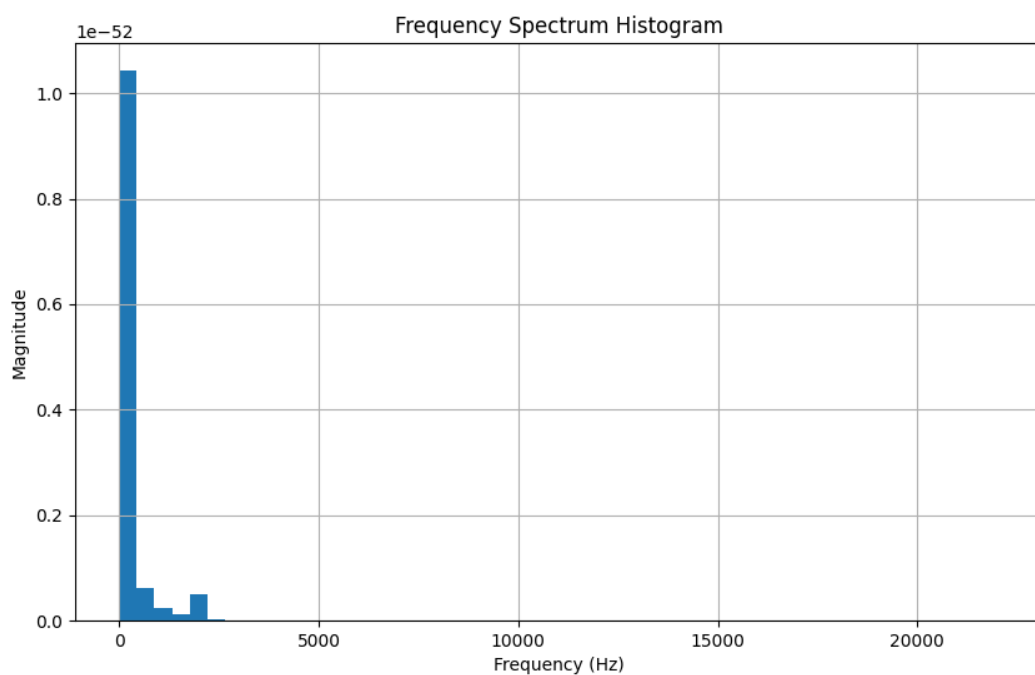


Figure 16. Low-pass filtered audio signal

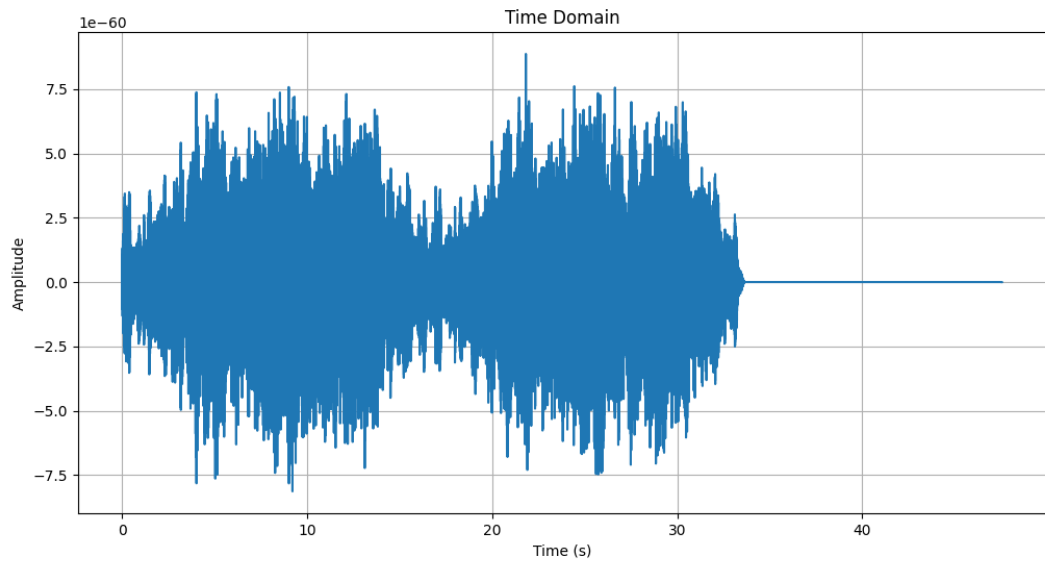


Figure 17. Low-pass filtered audio signal reversed to time domain

## 5.2 Example 2

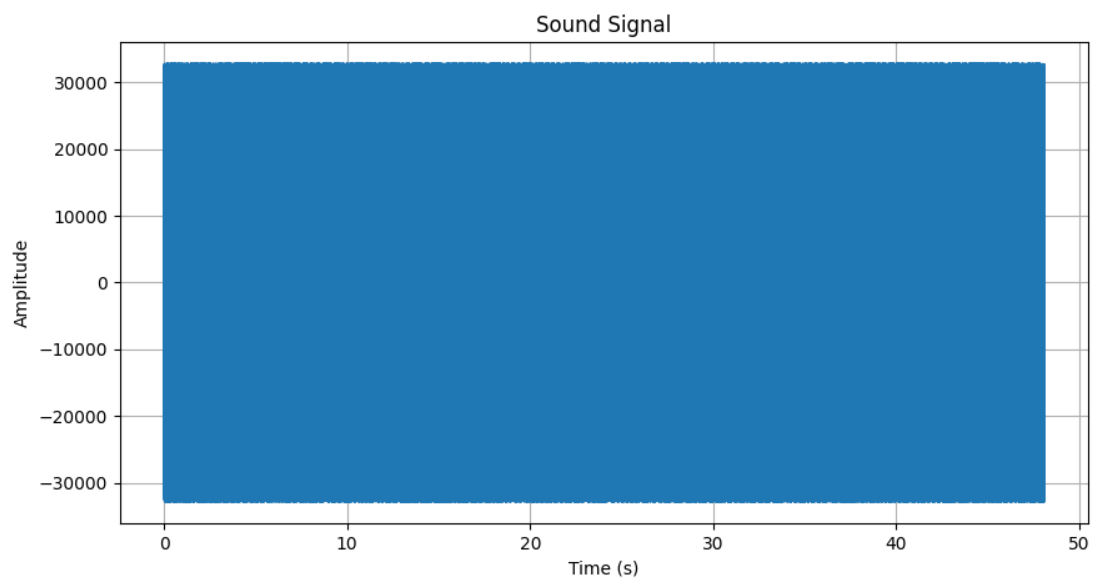


Figure 18. Unprocessed audio signal

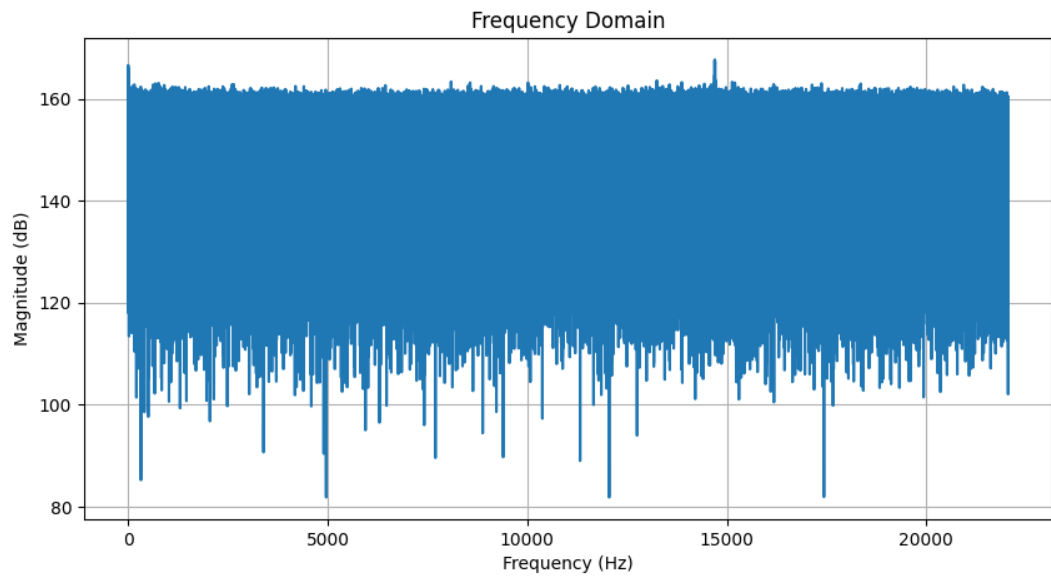


Figure 19. Power spectrum of the audio signal

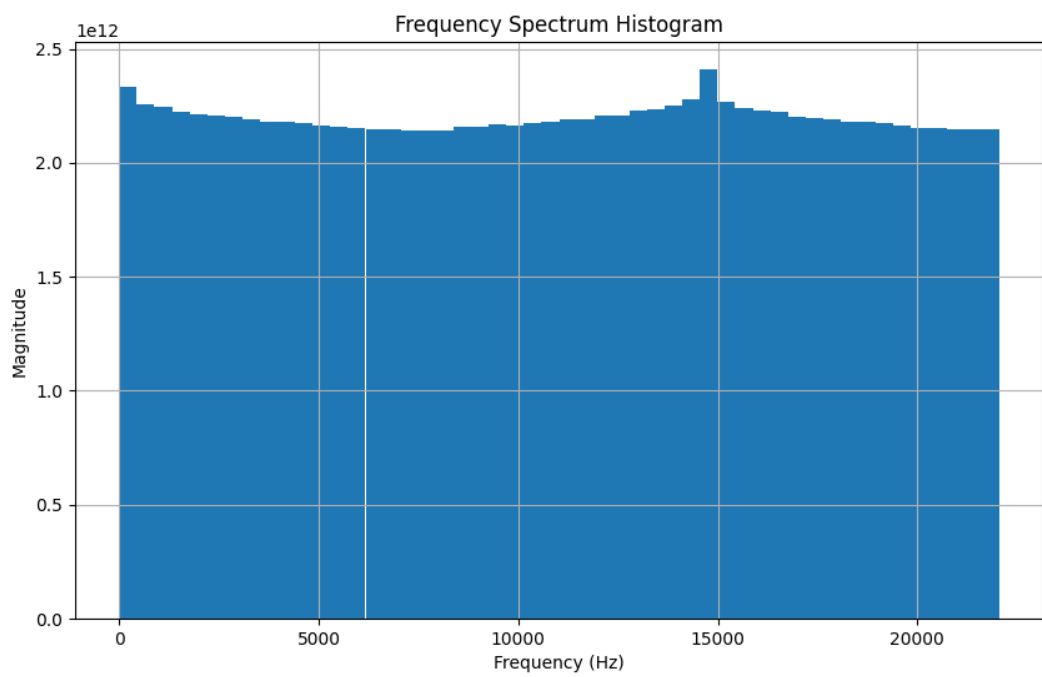


Figure 20. Signal in frequency domain and histogram

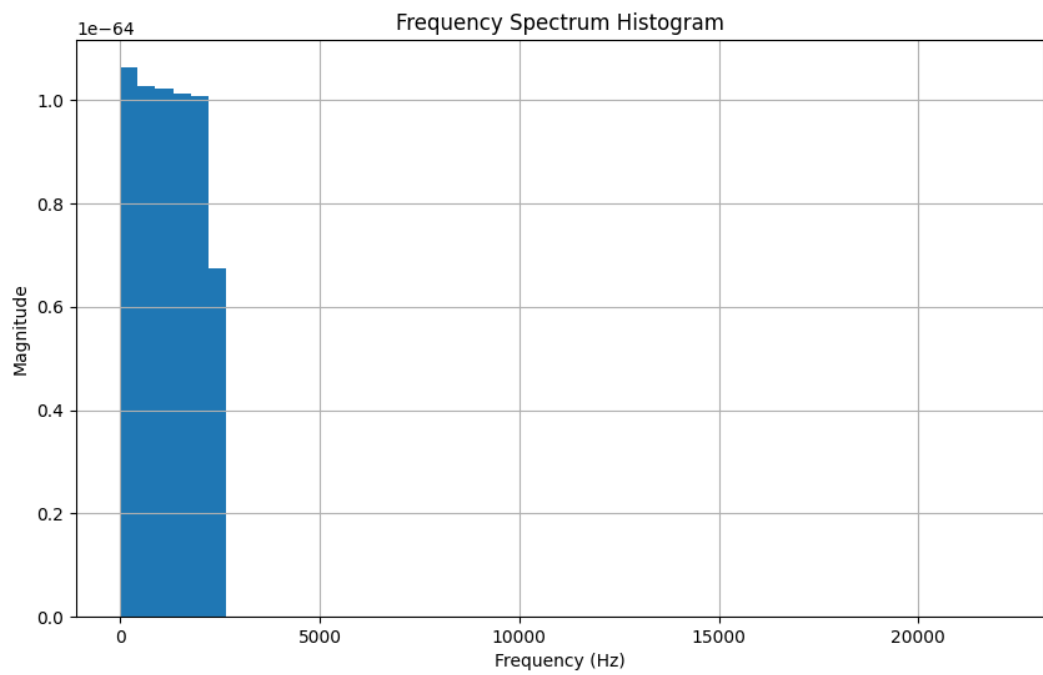


Figure 21. Low-pass filtered audio signal

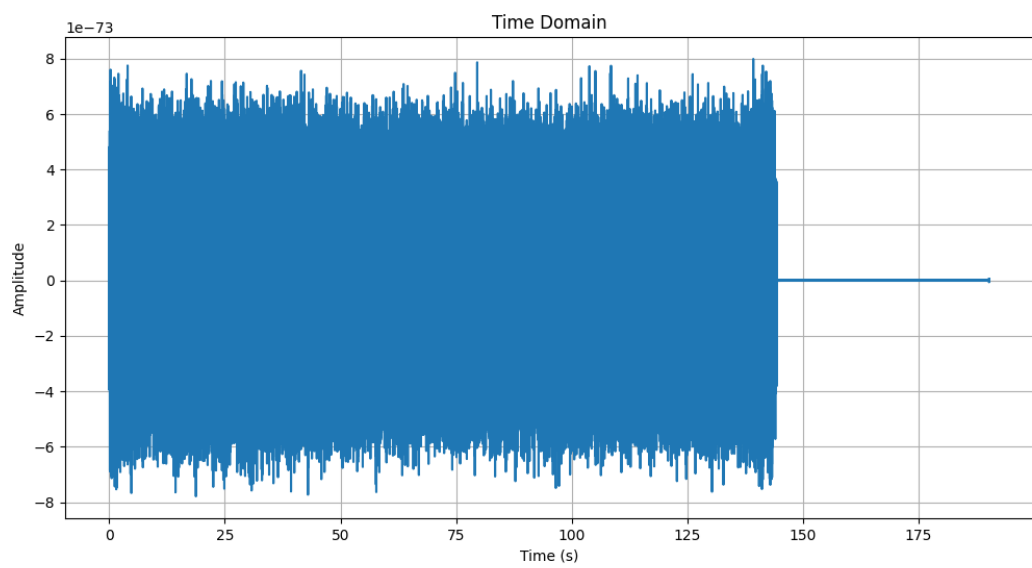


Figure 22. High-pass filtered audio signal reversed to time domain



### 5.3 Example 3

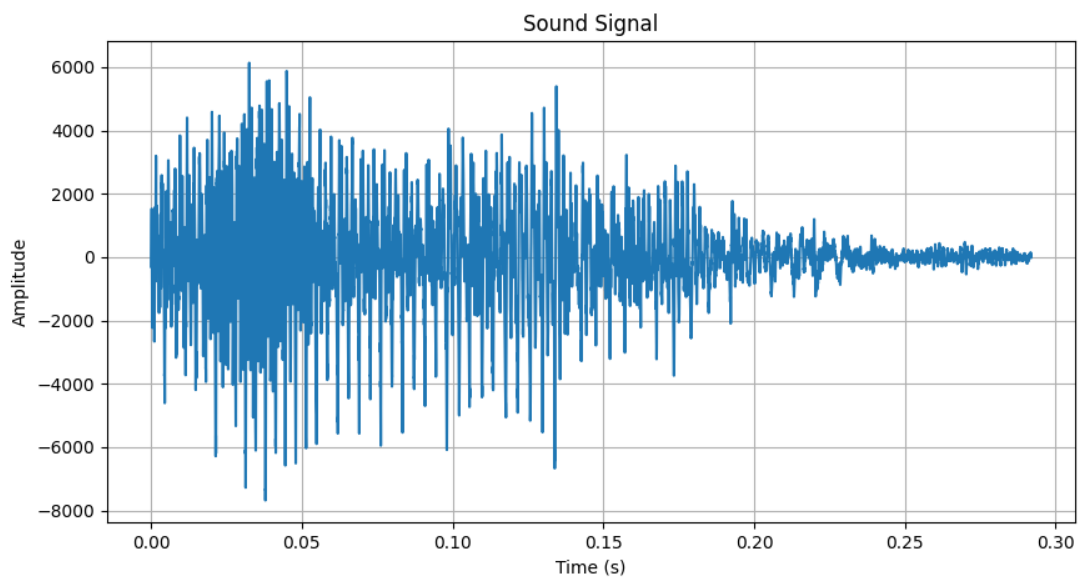


Figure 23. Unprocessed audio signal

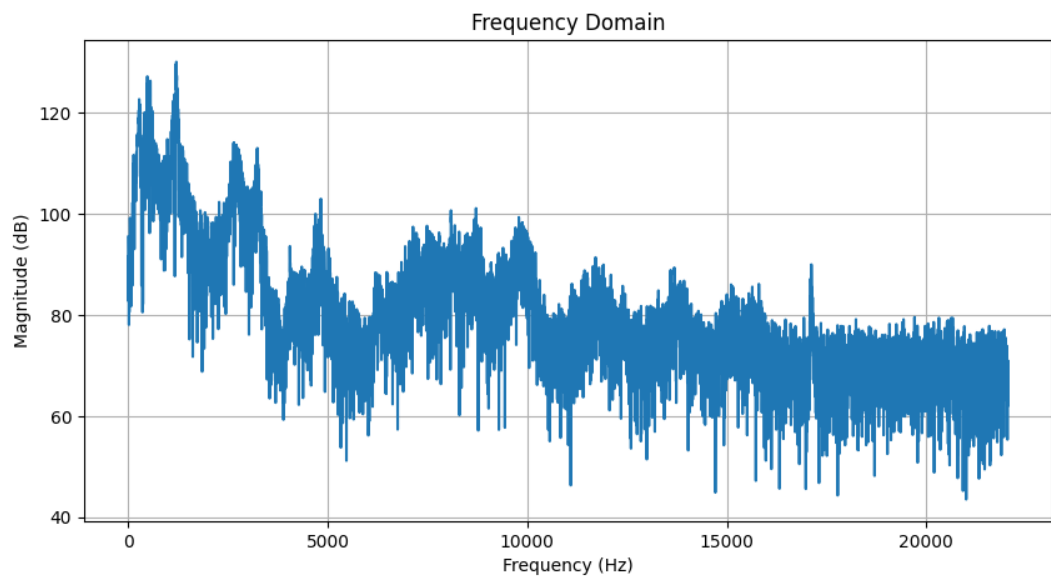


Figure 24. Power spectrum of the audio signal

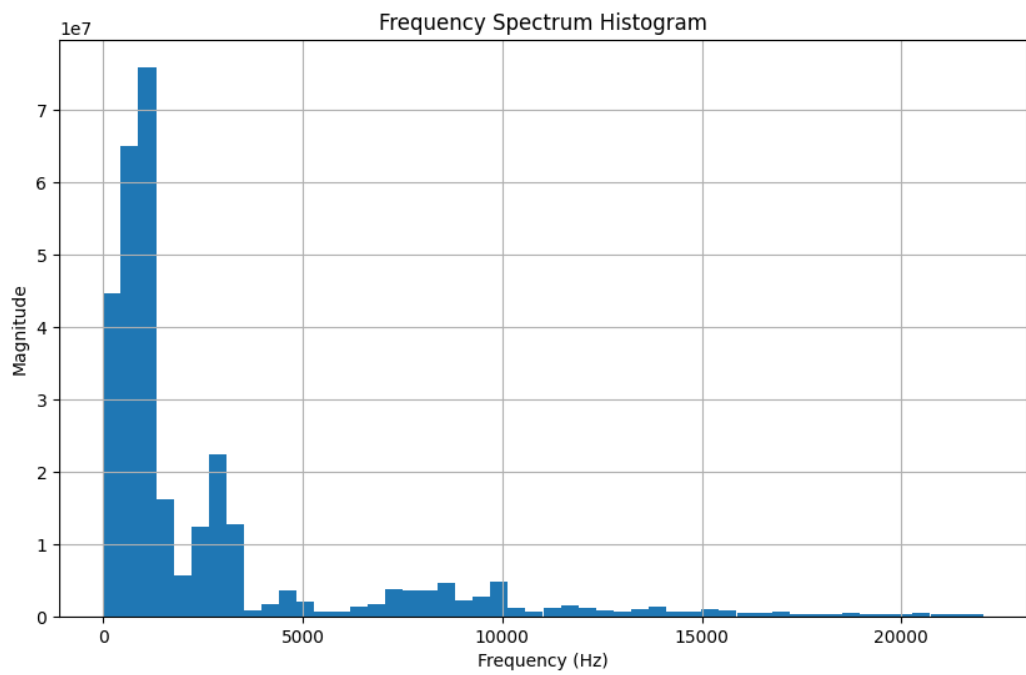


Figure 25. Signal in frequency domain and histogram

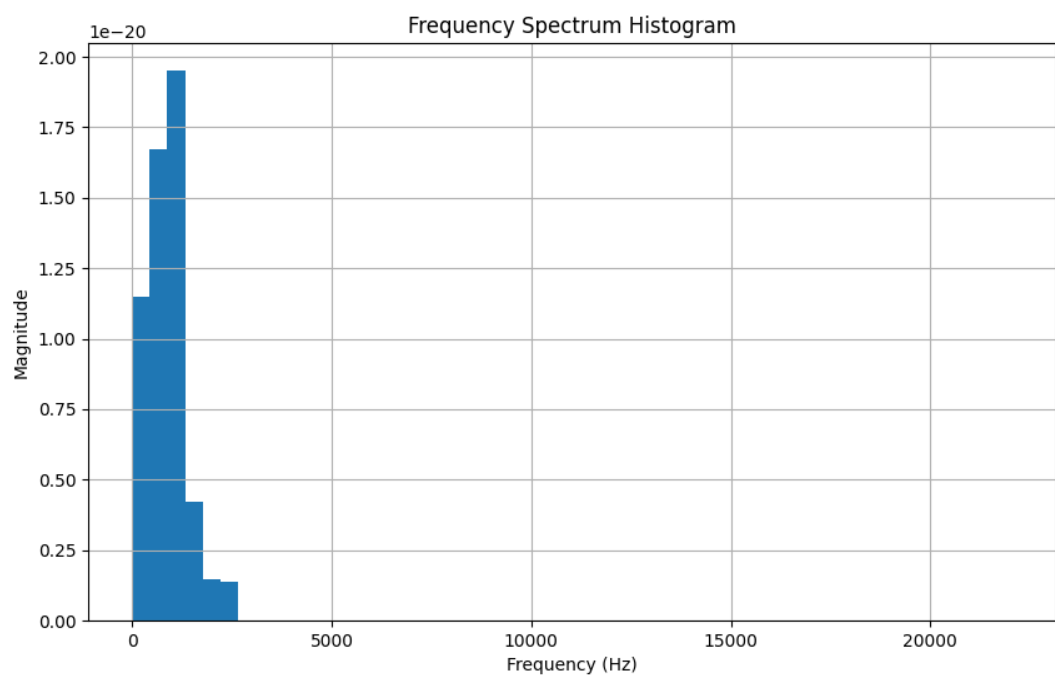


Figure 26. Low-pass filtered audio signal

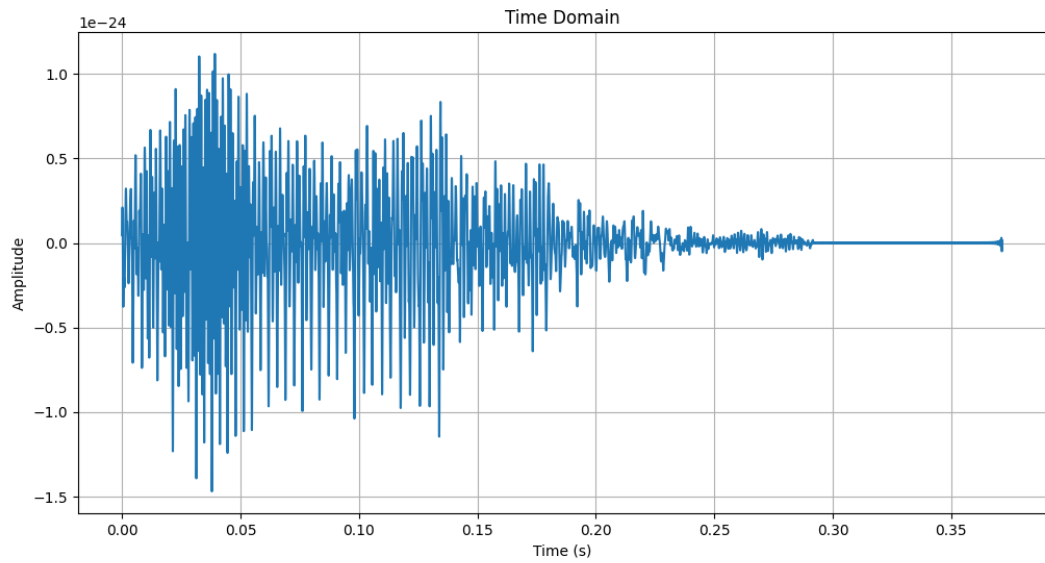


Figure 27. Low-pass filtered audio signal reversed to time domain

## 5.4 Example 4

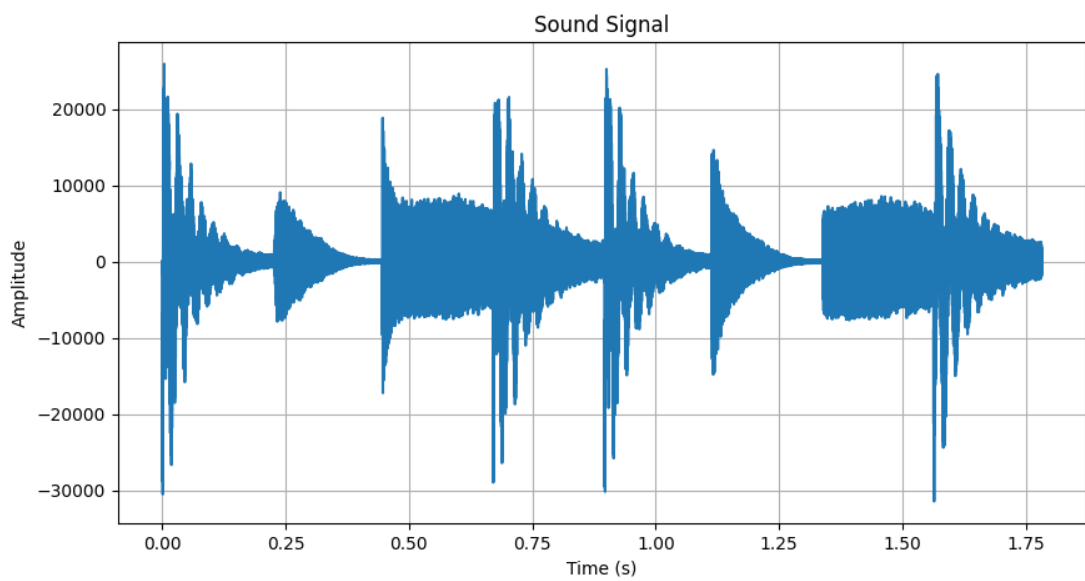


Figure 28. Unprocessed audio signal

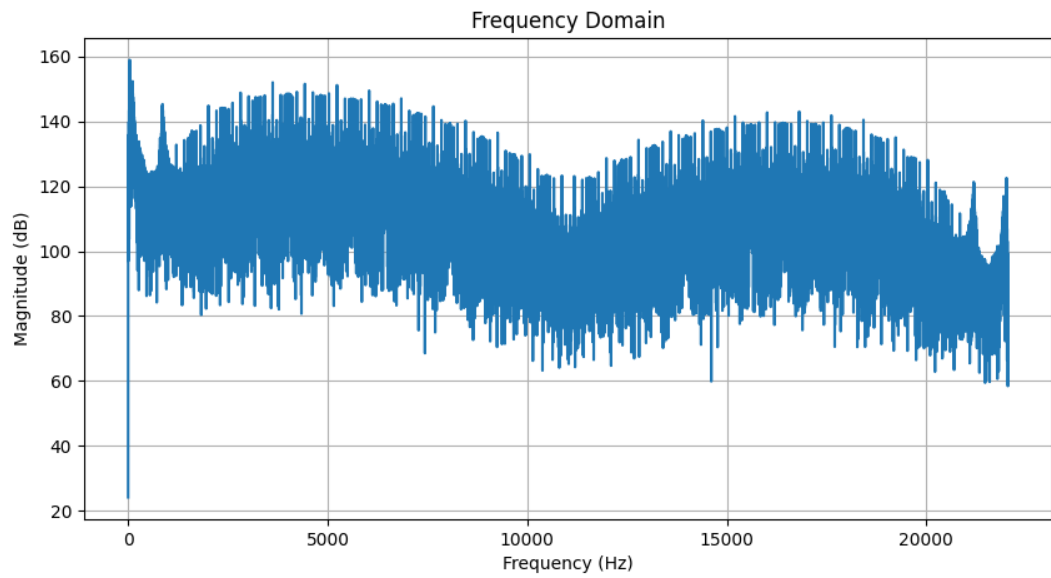


Figure 29. Power spectrum of the audio signal

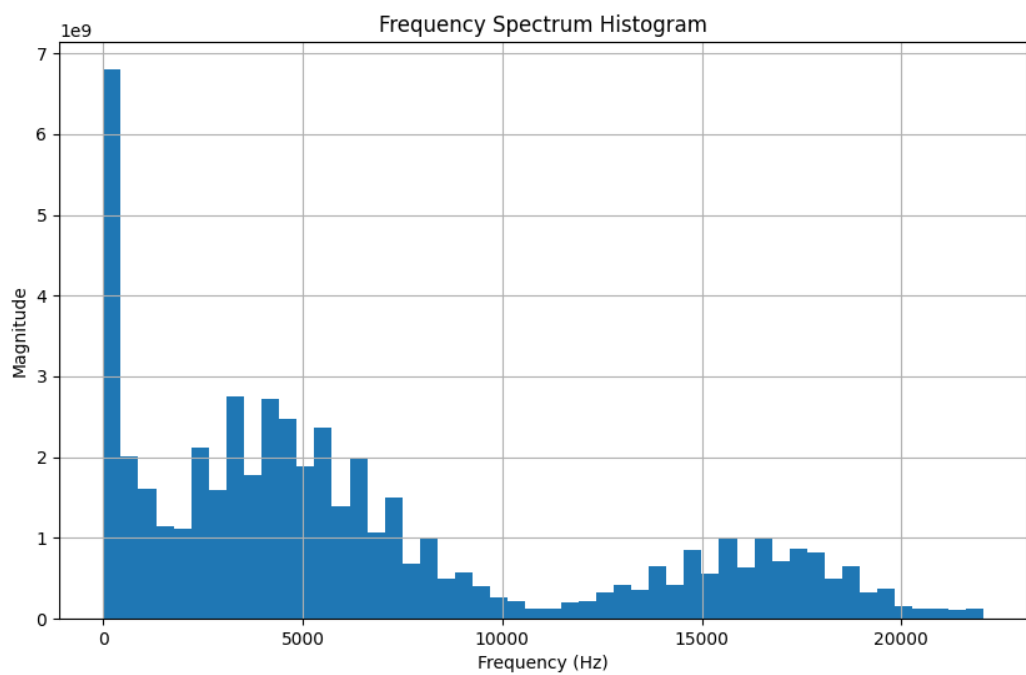


Figure 30. Signal in frequency domain and histogram

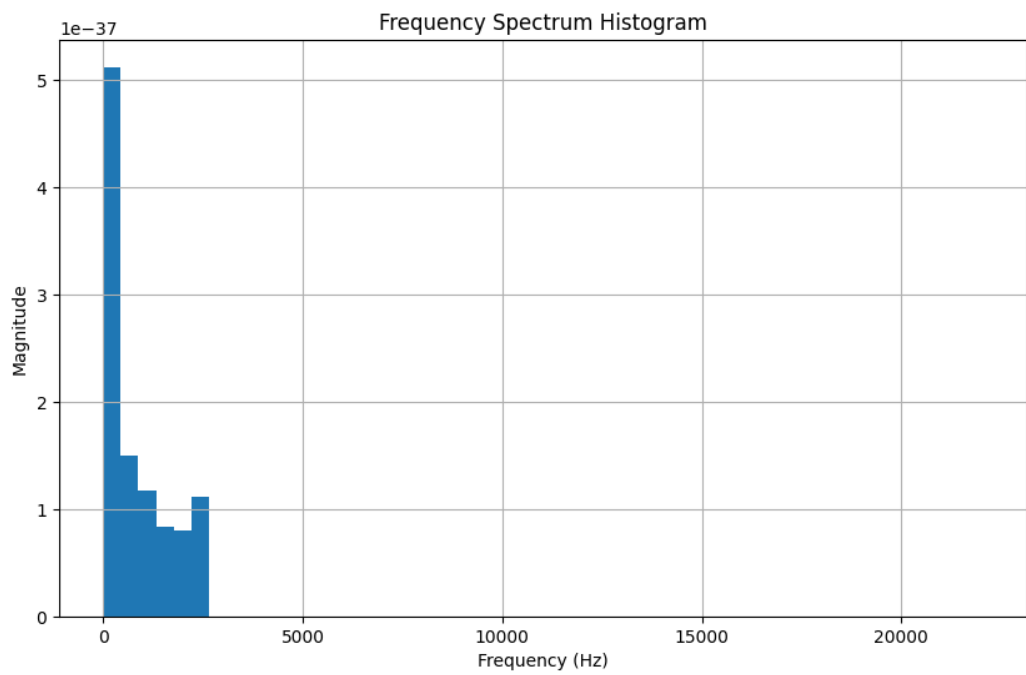


Figure 31. Low-pass filtered audio signal

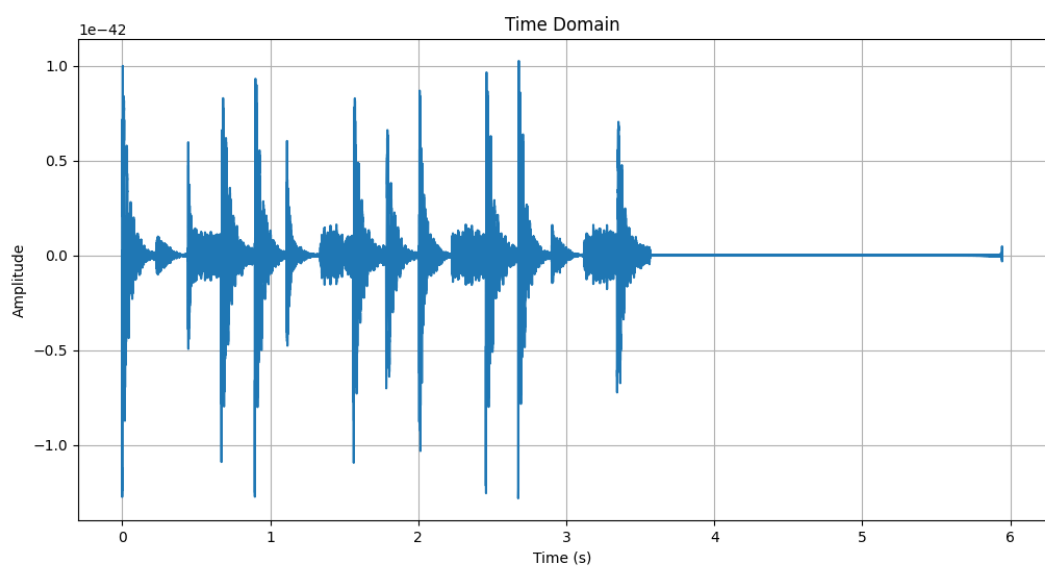


Figure 32. Low-pass filtered audio signal reversed to time domain

## 6 Conclusion

In this report, we implemented the Discrete Fourier Transform (DFT), Fast Fourier Transform (FFT), Inverse Discrete Fourier Transform (IDFT), and Inverse Fast Fourier Transform (IFFT) algorithms from scratch. We compared the performance of the DFT and FFT algorithms and analyzed the impact of zero padding on the FFT algorithm's performance. We also implemented high-pass, low-pass, and mid-pass filters using the Fourier transform and applied them to audio signals to isolate specific frequency components. The results demonstrate the effectiveness of the Fourier filter in isolating and analyzing frequency components of audio signals.

On the other hand, the analysis of algorithm performance showed that implementation of FFT from scratch is suboptimal and needs to be improved. The comparison of FFT from scratch and numpy FFT showed that numpy FFT is significantly faster. The comparison of DFT and FFT showed that FFT is significantly faster than DFT. The comparison of FFT with and without zero padding showed that zero padding reduces the performance of the FFT algorithm.

FFT is a powerful algorithm that enables efficient analysis and processing of time domain signals and their frequency components. By implementing and applying the FFT algorithm, we can gain insights into the frequency characteristics of signals and develop sophisticated signal processing techniques such as filtering, compression, and feature extraction.