Real-Time Noise Reduction on an ESP32-Based INMP441 Microphone System Using IIR Filtering

1st M. Adnan Bayu Firdaus
Faculty of Advanced Technology and Multidisciplinary
Airlangga University
Surabaya, Indonesia
muhammad.adnan.bayu-2021@ftmm.unair.ac.id

2nd Cahyan Irfan Syach
Faculty of Advanced Technology and Multidisciplinary
Airlangga University
Surabaya, Indonesia
cahyan.fan.syach-2021@ftmm.unair.ac.id

3rd Timo Widyanvolta
Faculty of Advanced Technology and Multidisciplinary
Airlangga University
Surabaya, Indonesia
timo.widyanvolta-2021@ftmm.unair.ac.id

4th Fatih Ulwan Annaufal
Faculty of Advanced Technology and Multidisciplinary
Airlangga University
Surabaya, Indonesia
fatih.ulwan.annaufal-2021@ftmm.unair.ac.id

***Abstract*—Voice-based interaction plays a crucial role in emerging IoT applications, yet the presence of background noise significantly impairs speech clarity and recognition accuracy. This paper examines the implementation of noise reduction techniques on a resource-constrained ESP32 microcontroller paired with an INMP441 microphone. By capturing audio signals via the I2S protocol and applying both time-domain and frequency-domain filtering methods—including Infinite Impulse Response (IIR) filters and Short-Time Fourier Transform (STFT)-based analysis—we assess their effectiveness in improving voice quality. While the system successfully acquires and processes audio data in real time, our findings show that conventional filtering alone provides limited noise suppression, underscoring the need for adaptive, time-frequency approaches..**

***Index Terms*—Voice Signal Processing, IoT, Noise Reduction, ESP32, I2S, IIR Filter, STFT**

I. Introduction

Voice input processing has become integral to modern IoT applications, enabling natural interactions through voice commands in scenarios such as smart homes, wearable devices, and industrial automation. However, a major hurdle to achieving accurate and efficient voice recognition in these systems is the pervasive presence of background noise. This noise, which can arise from various environmental sources, disrupts the clarity of the captured audio and reduces the reliability of voice-based interfaces [1].

In resource-constrained IoT devices, such as those built around the ESP32 microcontroller, implementing effective noise reduction methods poses additional challenges. Traditional filtering techniques, while conceptually simple, may not adequately adapt to time-varying signal characteristics or efficiently separate voice components from broadband noise [2]. Similarly, straightforward frequency-based approaches are often insufficient, as both desired speech and unwanted noise share overlapping frequency bands [3].

To address these issues, this work investigates the performance of various noise reduction techniques—from conventional infinite impulse response (IIR) filtering to more adaptive, time-frequency domain analyses—on an ESP32-based platform using the INMP441 microphone. By comparing these methods and examining their limitations, we aim to provide practical insights for improving voice clarity in noisy conditions.

II. Literature Review

A. Sound Signal

Sound signals are continuous variations in air pressure that the human ear interprets as audible information. To process these signals digitally, they must first be sampled and quantized, resulting in a discrete-time representation suitable for computational manipulation [4]. Understanding the fundamental properties—such as frequency components, amplitude, and phase—is essential for effective filtering and enhancement, particularly in environments where noise can obscure critical speech information.

B. Existing Noise Reduction Techniques

A wide array of noise reduction techniques have been developed to improve signal quality by attenuating unwanted components. Adaptive methods, such as the Wiener filter [5], can effectively deal with various noise conditions but often require careful parameter tuning, limiting their applicability in real-time, resource-constrained scenarios. Similarly, approaches like Harmonic Regeneration Noise Reduction (HRNR) [6] restore lost harmonics but lack robustness across diverse noise types and conditions.

The primary challenge with existing methods lies in adaptability and real-time deployment. Many algorithms assume relatively stable noise characteristics or depend on intensive computation, making them less suitable for embedded IoT platforms. Consequently, flexible, low-complexity solutions are needed to handle dynamic noise patterns without overburdening limited hardware resources.efficiency.

C. Voice Processing in IoT Devices

Voice-enabled IoT devices—ranging from smart home assistants to industrial sensors—must reliably capture and interpret audio commands in noisy environments [7]. However, limited processing power, stringent energy requirements, and the need for low-latency operation constrain the complexity of noise reduction algorithms that can be implemented. Ensuring accurate voice recognition under these conditions remains a key research problem.

D. Digital Filters for Audio Enhancement

Digital filtering techniques like Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters are commonly employed to manipulate frequency content and reduce noise. FIR filters are inherently stable and exhibit linear-phase characteristics, making them attractive for audio applications [8]. However, achieving sharp frequency responses may demand a large number of coefficients and computational resources.

IIR filters require fewer coefficients and can provide steeper roll-offs, but they may introduce phase distortions and risk instability if not carefully designed. Still, IIR filters often present a favorable trade-off for embedded systems, offering significant computational savings over FIR filters.

E. Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is a pivotal algorithm in digital signal processing, offering a remarkably efficient method to compute the Discrete Fourier Transform (DFT) with computational complexity on the order of $O(N \log N)$, rather than the $O(N^2)$ operations required by a direct DFT calculation. Its development can be traced back to the seminal work of Cooley and Tukey in 1965 [9], which introduced a divide-and-conquer approach to factorizing the DFT. Since then, numerous variants and optimizations, such as the split-radix FFT and adaptive planning techniques, have been proposed to further enhance performance and adaptability on modern hardware [10].

III. Methodology

A. System Architecture

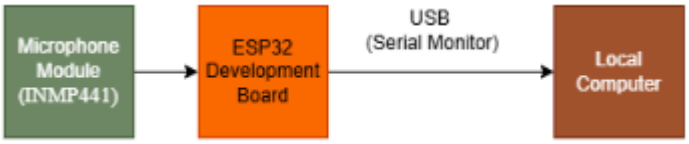


Fig. 1: System block diagram.

As shown in Figure 1, the system architecture consists of several components, including the microphone module (INMP441), the ESP32 development board, a USB cable, and a local computer. Microphone module acts as an input device with the input data being sound data and the sampling rate. The microphone module will be connected to an ESP32 development board, the ESP32 responsible for capturing sound data from the INMP441 microphone module.

B. Hardware Components

- Microphone module
  The INMP441 omnidirectional microphone is a high performance digital microphone used in this system to capture audio data. It is well-suited for applications that require clear voice input, as it offers excellent sound quality and sensitivity. Unlike traditional analog microphones, the INMP441 outputs digital data, reducing the need for complex analog circuitry and making it ideal for use with microcontrollers like the ESP32.

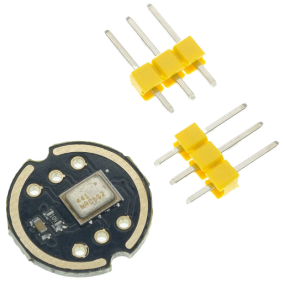


Fig. 2: INMP441 microphone module.

- ESP32 Microcontroller
  The ESP32 is a versatile microcontroller that plays a crucial role in this system, acting as the central processing unit for capturing and transmitting voice data. Known for its robust performance and integrated Wi-Fi and Bluetooth capabilities, the ESP32 is highly suited for IoT applications. In this project, the ESP32 manages the data collected from the INMP441 microphone, transmitting it efficiently over the MQTT protocol. Its dual-core architecture allows it to handle multiple tasks simultaneously, including real-time processing and data communication, ensuring low latency and efficient noise reduction. Additionally, the ESP32's low power consumption and small form factor make it an ideal choice for embedded systems and IoT devices.

Fig. 3: ESP32 microcontroller.

C. Capture Sound Signal

Data in this experiment was captured using the INMP441 microphone module and sent to a local computer using ESP32 via USB cable. ESP32 program to capture sound data from INMP441 written in C++ language with .ino format.

The sound data captured using I2S (inter-integrated circuit sound) protocol. Sampling rate frequency used is 10 kHz, and the program will stop capturing data after 10 seconds. Thus

Algorithm 1 Audio Data Capture and Processing with I2S Microphone

Setup:
Configure I2S with the following microphone settings:
    - 16-bit samples
    - Mono
    - Sampling rate: 10,000 Hz
Start Serial communication.
Initialize variables and timers.
Loop:
while captured samples $<$ target samples **do**
    Read audio samples from the I2S microphone into the buffer.
    for each sample in the buffer **do**
        Convert the sample.
        Print the sample.
    end for
    Update the total samples counter.
end while
Stop:
Display a completion message.
Halt further processing.

the data captured will be 100.000 samples. The data printed in the serial monitor is then captured using a python program and saved in csv format.

Algorithm 2 Serial Data Logging to CSV

Setup:
Define serial port, baud rate, and CSV file path.
Open the serial port and the CSV file.
Write the CSV header.
Data Capture Loop:
while true **do**
    Continuously read data from the serial port.
    if valid data is received **then**
        Write the data to the CSV file.
        Optionally, print the data to the console.
    end if
end while
Interrupt Handling:
On `Ctrl + C`, stop capturing data.
Close all open resources.
Error Handling:
Handle any exceptions that occur.
Display appropriate error messages.
Close:
Close the serial connection and the CSV file.
Notify the user of the saved file location.

D. Infinite Impulse Response Filter Design

In this study, a Butterworth low-pass filter is used for noise reductions. The filter has a cutoff frequency of 3 Hz, ensuring effective suppression of high-frequency noise while preserving the clarity of the voice signal. The filter will use an order of 4 to provide a sharp transition between the passband and stopband. The transfer function $H(s)$ of the filter is expressed in equation (1):

$$H(s) = \frac{1}{\sqrt{1 + \left(\frac{s}{\omega_c}\right)^8}} \tag{1}$$

where $\omega_c = 2\pi f_c$ and $f_c = 3\,\text{KHz}$ is the cutoff frequency. This configuration ensures a smooth frequency response with very small distortion to the audio.

IV. Results and Discussion

A. Sound Signal

Data captured in this experiment is stored in csv file with earth row corresponding for one sample value, thus using 10 kHz sampling rate and 10 second record duration, there will be 100.000 data samples. This experiment captures 3 sounds to compare between voice only without noise, noise only without voice, and voice blended with noise (Figure 4).

To understand which frequency to cut off, fast fourier transform (FFT) of the sound data calculated. The result of FFT is visualized as a plot in the frequency domain.

From the visualization Figure 5, the noise is distributed across a wide frequency range and overlaps with the voice signal, making it difficult to isolate without affecting speech components. Since human speech primarily occupies lower frequencies, applying a low-pass filter at around 4.9 kHz helps reduce higher-frequency noise while preserving the core speech content (Figure 6).

Algorithm 3 FFT-Based Signal Processing and Filter Implementation

Setup:
Import sound data
Compute FFT of signal
Plot data and FFT
Filter Design:
Define cutoff frequency f_c and filter order n
Compute Butterworth filter coefficients
Adjust coefficients for frequency scaling
Derive analog transfer function $H(s)$
Discrete Filter:
Convert $H(s)$ to a discrete transfer function $H(z)$ (using GBT)
Apply Filter:
Extract numerator and denominator coefficients from $H(z)$
Apply filter coefficients to input samples

B. Infinite Impulse Response Filter Signal

Filter used in this experiment is the Butterworth-low pass filter with frequency cut off 4.9k in 2nd order. As shown in Algorithm 3, the filter coefficients are first calculated

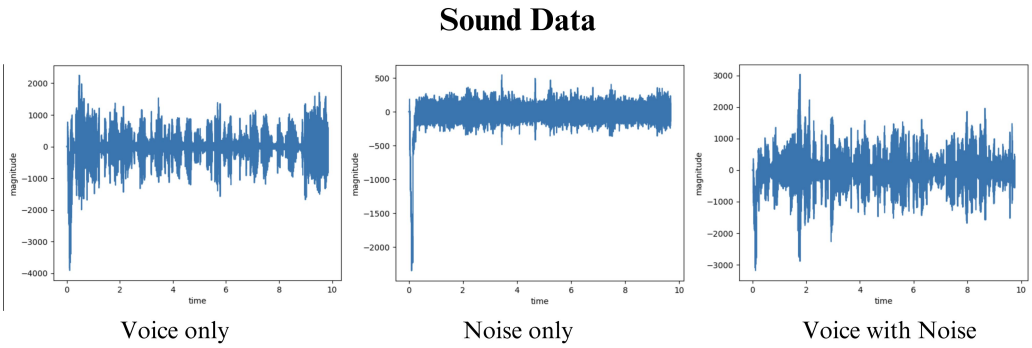


Fig. 4: Sound data.

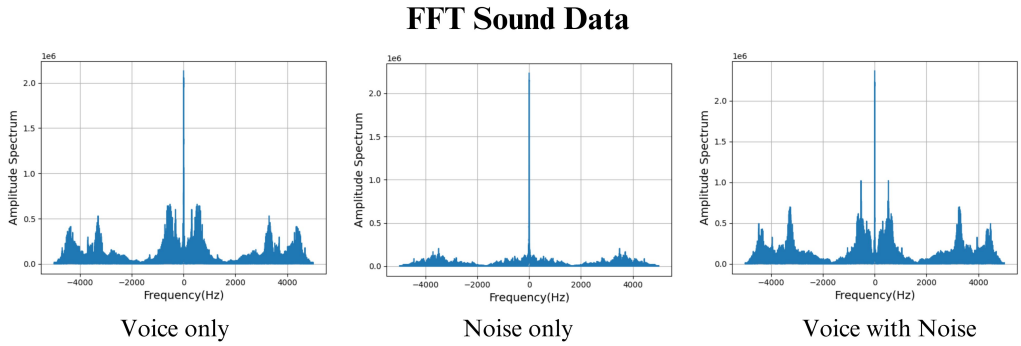


Fig. 5: FFT sound data.

in a Python program. Subsequently, in Algorithm 4, these coefficients are applied within the Arduino C++ environment.

Algorithm 4 I2S Audio Acquisition and Digital Filtering

Setup:
Configure I2S parameters: buffer size N, sample rate f_s, microphone channel
Assign GPIO pins for SCK, LRCK, and SD
Initialize filter states and counters
Install and configure I2S driver
Start Processing:
Begin serial debugging (optional)
Record start time
Main Loop:
while recording not complete **do**
    Read samples from I2S into buffer
    For each sample:
        Apply digital filter
        Update filter states
        If index % 3 = 0, output processed sample
        Increment sample counter
end while

From the computation the coefficient values can be seen in the table I.

TABLE I: Computed Coefficient Values for the Butterworth Filter

Coefficient	Value
a_1	-0.49387595
a_2	-0.21502486
b_0	0.4272252
b_1	0.8544504
b_2	0.4272252

Coefficient calculated applied to non continuous sound sample signal and continuous sound signal to do low pass filter, the result can be seen in the Figure 7 and Figure 8, respectively.

The result shows that the Butterworth-low pass filter doesn't work great for sound signals because sound signal frequency changes over time as we can see in the Figure 9.

It can be seen at the FFT result that the sound data fulfill all frequencies across the x axis.

C. ESP32 for Sound Processing

ESP32 for sound capturing using I2S seems to have a major problem. To record 10 seconds of sound data the program tends to always record it for more than 10 seconds. That condition results in data missing when recording. This happens because the I2S protocol requires you to read multiple data at a time before moving to read the next data.

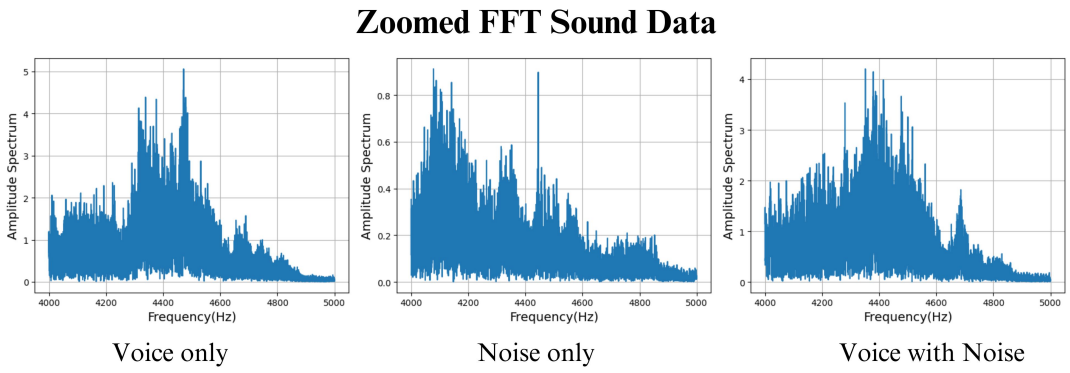


Fig. 6: Zoomed FFT sound data.

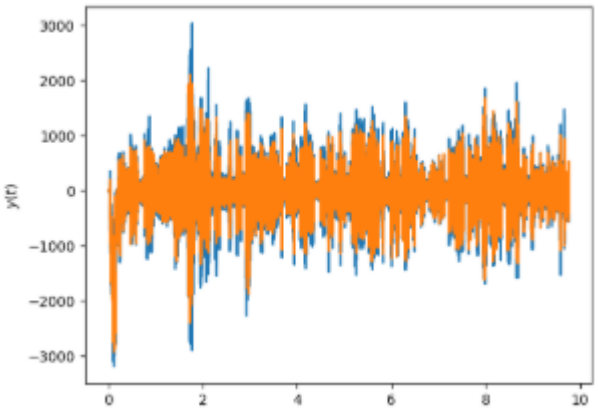
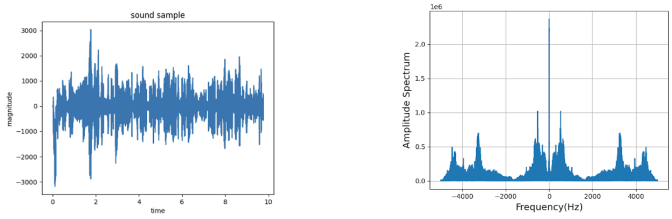


Fig. 7: Butterworth low pass filter to non continuous sound data sample.

Fig. 8: Butterworth low pass filter to continuous sound data.



(a) Sound signal time frequency

(b) Sound signal frequency domain

Fig. 9: Sound signal data visualization

Algorithm 5 Audio Data Capture Loop

```
while true do
    Check Completion:
    if captured samples ≥ target samples then
        Stop recording.
        Exit loop.
    end if
    Read Audio Data:
    Read audio samples from the I2S microphone into the buffer.
    Process Data:
    for each sample in the buffer do
        Convert the sample.
        Print the sample.
    end for
    Update the total samples counter.
end while
```

In the Algorithm 5, it can be seen that the program has to read multiple data and store it to buffer, then print it to serial monitor, then moving to the next capture process.

To better understand this condition, an experiment was conducted using generated sinusoid signal sound and then captured it using the system. The signal is using 500 Hz frequency in 5 second duration and 1 magnitude.

From the Figure 10 and 11 it can be seen the ESP32 only captures for 1 second, while it's supposed to be 5 seconds.

D. STFT for Sound Processing

Using regular fixed filters to remove certain frequencies from the entire signal is often ineffective because both the voice and noise components can span a broad range of frequencies. In contrast, the Short-Time Fourier Transform (STFT) provides a time-frequency representation that helps isolate when and where certain frequencies occur. The STFT

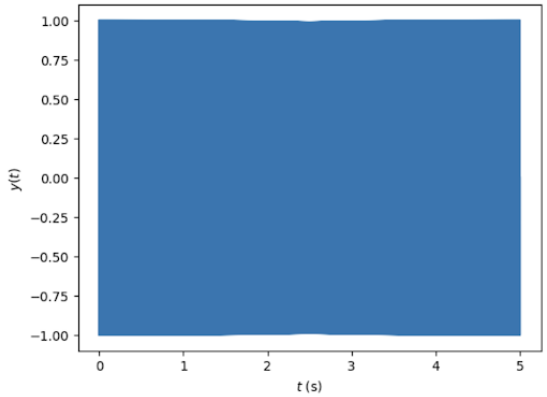


Fig. 10: Generated sinusoid signal.

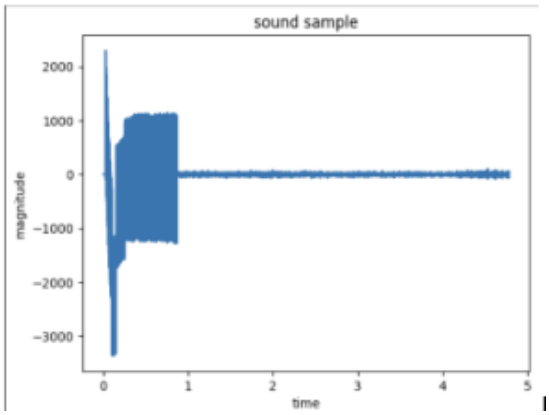


Fig. 11: Captured generated sinusoid signal.

involves segmenting the signal into short, overlapping frames, applying a window function to each frame, and then computing the Fourier transform of these frames. By doing so, it captures how the frequency content evolves over time, allowing more targeted and adaptive noise reduction strategies that can differentiate between transient voice signals and persistent background noise.

It must be underlined that STFT doesn't remove all the noise, but it reduces huge amounts of noise while maintaining the primary data (Figure 12. So it would work best with sound signals.

E. Performance Evaluation

The performance evaluation of the filtered signal is conducted using three key metrics: Mean Squared Error (MSE), Correlation Coefficient, and Signal-to-Noise Ratio (SNR). These metrics provide insights into the quality of the filtered signal and how well the filtering process has enhanced the signal in comparison to the noise.

- **Mean Squared Error (MSE)**: The MSE is calculated as the average of the squared differences between the original signal $x(t)$ and the filtered signal $\hat{x}(t)$:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$$

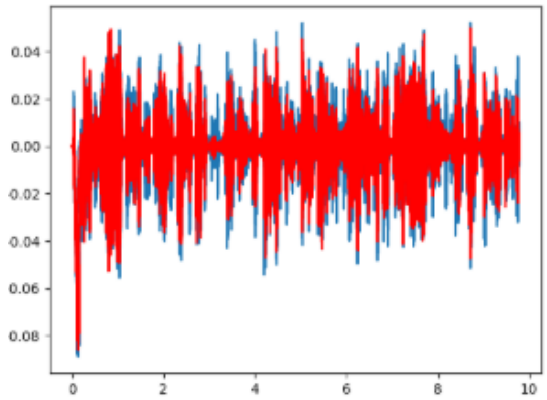


Fig. 12: STFT of voice with noise signal.

where N is the number of samples, x_i is the i-th sample of the original signal, and $\hat{x}_i$ is the i-th sample of the filtered signal.
In this evaluation:

$$\text{MSE} = 85111.6105$$

This high value suggests that there is still a significant difference between the filtered and original signals, likely due to the presence of noise or limitations in the filtering process.

- **Correlation Coefficient (CC)**: The Correlation Coefficient (r) measures the linear relationship between the original signal $x(t)$ and the filtered signal $\hat{x}(t)$:

$$r = \frac{\text{cov}(x, \hat{x})}{\sigma_x \sigma_{\hat{x}}}$$

where $\text{cov}(x, \hat{x})$ is the covariance between the original and filtered signals, and σ_x and $\sigma_{\hat{x}}$ are their respective standard deviations.
The computed correlation coefficient matrix is:

$$\text{Correlation Coefficient} = \begin{bmatrix} 1 & 0.675 \\ 0.675 & 1 \end{bmatrix}$$

A correlation coefficient of $r = 0.6755$ indicates a moderate positive linear relationship between the original and filtered signals. This shows that while some similarities exist, the filtering process has not fully preserved the original signal's characteristics.

- **Signal-to-Noise Ratio (SNR)**: The SNR is defined as the ratio of the signal power to the noise power, typically expressed in decibels (dB). It is calculated as:

$$\text{SNR} = 10 \cdot \log_{10}\left(\frac{\text{Signal Power}}{\text{Noise Power}}\right)$$

where the signal power is the variance of the original signal $x(t)$, and the noise power is the variance of the difference between the original and filtered signal, $e(t) = x(t) - \hat{x}(t)$.
The SNR for this evaluation is:

$$\text{SNR} = 2.4252\,\text{dB}$$

A relatively low SNR value suggests that while the filtering process has reduced some noise, a significant amount of noise still remains in the signal. This could be due to the recording itself being excessively noisy.

V. Conclusion

A. Summary of Findings

In this study, the authors have developed a sound signal processing system using the INMP441 microphone module and the ESP32 microcontroller. Although this system was designed to address the challenges of noise reduction in noisy environments, experimental results indicate that the ESP32 is not optimal for sound processing. Limitations in the sound signal processing capabilities of the ESP32 have led to difficulties in achieving effective noise reduction. Despite implementing various noise reduction techniques, including IIR Butterworth filters and FFT analysis, the system still faces challenges in producing clear sound amidst noise. These findings highlight the need for further exploration of more suitable platforms for sound processing in IoT applications.

B. Implications

The findings of this study have significant implications for the development of voice-based applications within the IoT ecosystem. With the increasing use of voice commands in smart devices, it is crucial to have systems that can perform well in noisy conditions. The limitations of the ESP32 in sound processing indicate that developers need to consider other, more powerful and efficient platforms for applications requiring complex sound signal processing. This research provides insights into the challenges faced in sound processing in noisy environments and emphasizes the need for better algorithms and hardware development to enhance user interaction with IoT devices.

C. Future Work

For future research, authors recommend several directions that can be pursued. First, exploring the use of more advanced microcontrollers or signal processing platforms, such as Raspberry Pi or FPGA, which can provide better processing capabilities for sound applications. Second, developing more adaptive and real-time noise reduction algorithms to improve performance in various environmental conditions. Additionally, further research could focus on integrating advanced signal processing techniques, such as deep learning, for more accurate and efficient voice recognition. Thus, future research can concentrate on finding more effective solutions to the challenges of sound processing in IoT applications.

References

[1] Paz Penagos, H., Mahecha, E. M., Camargo, A. M., Jimenez, E. S., Sarmiento, D. A. C., and Salazar, S. V. H. (2024). Detection, recognition and transmission of snoring signals by ESP32. Measurement: Sensors, 36, 101397.

[2] Bernal-Ruiz, C., García-Tapias, F. E., Martín-Del-Brío, B., Bono-Nuez, A., and Medrano-Marqués, N. J. (2005). Microcontroller implementation of a voice command recognition system for human-machine interface in embedded systems. IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 1 2 VOLS.

[3] Guo, Z., Dai, H., Ye, J., and Song, Y. (2023). STM32 Microcontroller-Based Voice-Activated Anti-Throttle Mis-Stepping System. 2023 IEEE 3rd International Conference on Data Science and Computer Application, ICDSCA 2023.

[4] Oppenheim, A. V., and Schafer, R. W. (2010). Discrete-Time Signal Processing. Prentice Hall.

[5] Lim, J. S., and Oppenheim, A. V. (1979). Enhancement and bandwidth compression of noisy speech. Proceedings of the IEEE.

[6] Salmute, V. B., and Agrawal, N. G. (2013). Harmonic regeneration noise reduction in speech signals. International Journal of Engineering and Advanced Technology (IJEAT).

[7] Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions.

[8] Proakis, J. G., and Manolakis, D. G. (2006). Digital Signal Processing: Principles, Algorithms, and Applications. Pearson Prentice Hall.

[9] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. Comput., 1965.

[10] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," Proc. IEEE, 2005.