Digital Signal Processing Course Project

Adaptive Filtering in Noise- Cancelling Headphones

ECCE 402

Shayma Mohammed Alteneiji | 100063072

Rahika Jannat Roza | 100061286

Khalifa Yousif Almazrouei | 100059782

Course Instructor: Prof. Paschalis Sofatasios

Date: 12/12/2024

Abstract

This project explores the design, implementation of an adaptive noise cancellation system using the FxLMS algorithms. Adaptive filtering principles are leveraged to develop a real-time active noise control system, with a focus on mitigating secondary path effects through a pre-modelled secondary path filter. The system was implemented on a Raspberry Pi Model 3, utilizing MEMS microphones and custom 3D-printed casing to create a portable solution. Key parameters such as adaptation step size were optimized through simulations across various scenarios to balance convergence speed and stability. While we could successfully obtain a model for the secondary path effects, we could not successfully implement the headphones noise cancelling system due to some errors encountered in the code. The project provides a practical understanding of adaptive signal processing and highlights areas for future improvement, including advanced algorithm tuning for optimal performance in dynamic environments.

Table of Content

Introduction

Adaptive filters are dynamic systems whose parameters adjust to adapt to changes in the environment. In applications such as echo cancellation in phone calls, channel equalization, and producing a null in a specific direction for antenna characterization, the changes in the environment cannot be specified a priori, necessitating the use of adaptive filters [1]. Among the various adaptive filter algorithms, the Least-Mean-Square (LMS) algorithm is commonly employed due to its simplicity and practicality [1,2]. However, in real -world systems, processing delays and application-specific factors, referred to as secondary path effects, can degrade the performance of noise-cancelling systems using LMS [2-5].

To address these challenges, the FxLMS algorithm, an extension of LMS, that was developed to mitigate secondary path effects [2-5]. While both IIR and FIR filters can be used for adaptive filtering, FIR filters are generally preferred due to their guaranteed stability under any changing conditions [1].

This report details the design and implementation of an adaptive noise-canceling headphone system using the FxLMS algorithm. The report begins with the theoretical background of adaptive filters, focusing on LMS and FxLMS algorithms, followed by the system design, including hardware and software components. Finally, the methodology, results, and conclusions are presented.

Adaptive Filtering and Noise Cancellation Theory

This section discusses the details of the working of the LMS and the FxLMS adaptive filtering algorithms, their block diagram in adaptive noise cancellation systems.

The LMS algorithm

Adaptive filtering algorithms are based on some optimization criterion, each yielding a corresponding equation used to calculate the vector **w**[n+1] or the updated filter weights. The algorithm iteratively updates the filter coefficients until that criterion is optimized; at that stage we say convergence is reached, and the difference between $\boldsymbol{w}$[n+1] and $\boldsymbol{w}$[n] becomes insignificant. The LMS algorithm is based on an optimization of the mean squared error (MES), it works to minimize the MSE to reach the least squared error point [1,3-5]. The MSE is a multivariable function of the filter coefficients [4]. Figure 1 shows the MSE for an adaptive filter with two coefficients. Here, w_0 and w_1 are the filter coefficients. At the bottom of the bowl-shaped function, the MSE is at a minimum, and the corresponding values of the filter coefficients, w_0 and w_1, are optimal.

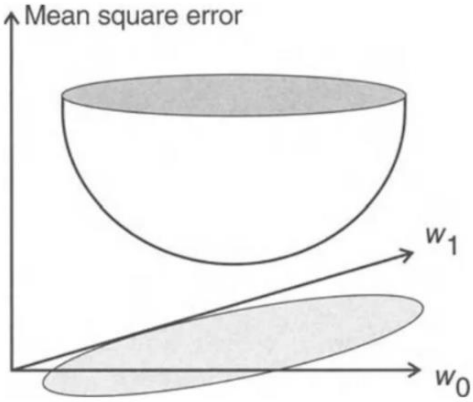

Figure 1. The MSE for an adaptive filter of two coefficients, [4]

The LMS algorithm identifies these optimal values (i.e., reach the minimum MSE point) using the gradient descent method [4,5]. Gradient descent is a method for minimizing a function (MSE) by iteratively adjusting its parameters in the direction of steepest descent [6]. Since computing the gradient of the MSE requires extensive data records and can be computationally intensive, instead, the instantaneous value of the squared error, e^2[n], serves as an approximation of the mean squared error, E ($e^2[n]$) [4]. In practical systems, this would enable computationally feasible real-time processing, where filter coefficients can be updated at every iteration.

Figure 2 shows the block diagram of adaptive filters in noise cancellation using the LMS algorithm. $P(z)$ represents an unknown system (plant), also called the primary path model. Both the plant system and the adaptive FIR model $W(z)$ are excited by the same input $x[n]$. The FIR model's output, $y[n]$, known as the anti-noise signal, is expressed as:

$$y[n] = \sum_{k=0}^{M-1} w[k]\, x[n-k]$$

Here, $w[k]$ are the adaptive filter coefficients, and M is the number of coefficients. The error, $e[n]$, is the difference between the plant output, $d[n]$, and the model's output, $y[n]$. Once convergence is reached, the MSE is minimized, and the FIR filter model equates to the primary path model.

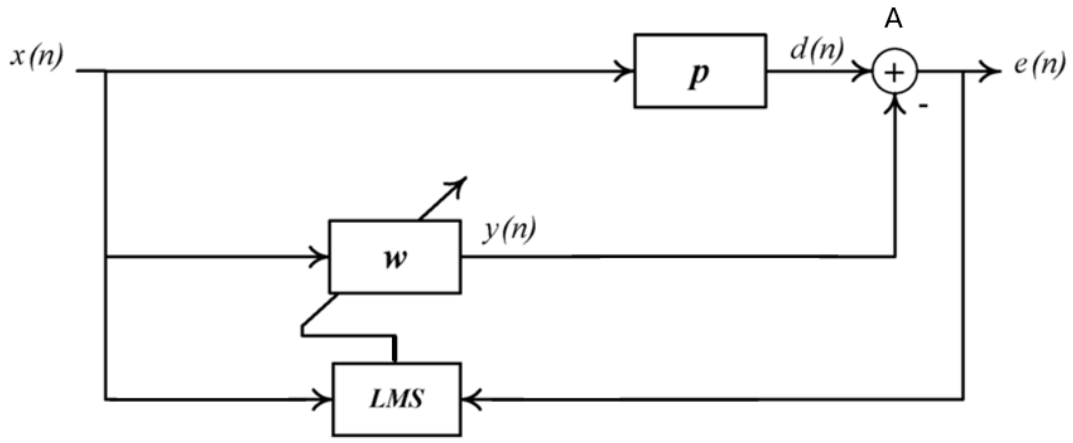

Figure 2. Adaptive filters in noise- cancellation using the LMS algorithm, [7]

In real-world systems, as stated previously, noise- cancelling systems using the LMS algorithm would yield disappointing results due to it overlooking of the secondary path effects. The next section discusses the FxLMS algorithm, which builds on the LMS algorithm to address these limitations. The FxLMS outperforms the LMS by integrating a secondary path effects model, S(z).

Secondary Path Effects and the FxLMS Algorithm

Secondary path effects describe the modifications on the anti-noise, y[n], as it travels through the various components of the system. In noise cancellation systems, the error signal, e[n] is the signal resulting from the acoustic interfaces between the noise modified by the primary path model (headphone), d[n], and the anti-noise signal streamed via a speaker, y[n].

The effectiveness of noise cancellation depends on accurately modelling secondary path effects. For instance, if a delay was introduced by the secondary path effects, this would disrupt the exact 180-degree phase shift between the noise and anti-noise signals required for complete cancellation of the noise, resulting on a comprised performance of the system.

Specifically in this project, the secondary path effects include the impact on the signal, $y[n]$, as it travels through the acoustic path from the speaker to the noise source (e.g. phase shift), and also through the electro-acoustic path, which includes the effects of delays, quantization noise, and distortions introduced by the DAC (Digital-to-Analog Converter) and ADC (Analog-to-Digital Converter), phase shifts, delays introduced by smoothing filters, and noise generated by amplifiers and microphones [2,3].

The FxLMS algorithm is essential for real-time processing applications; it handles those effects by incorporating a secondary path effect model, $S(z)$, cascaded with the FIR adaptive filter model to counteract those effects. The FxLMS algorithm block diagram is shown in Figure 3. The next section derives the w[n+1] or updated filter coefficients equation.

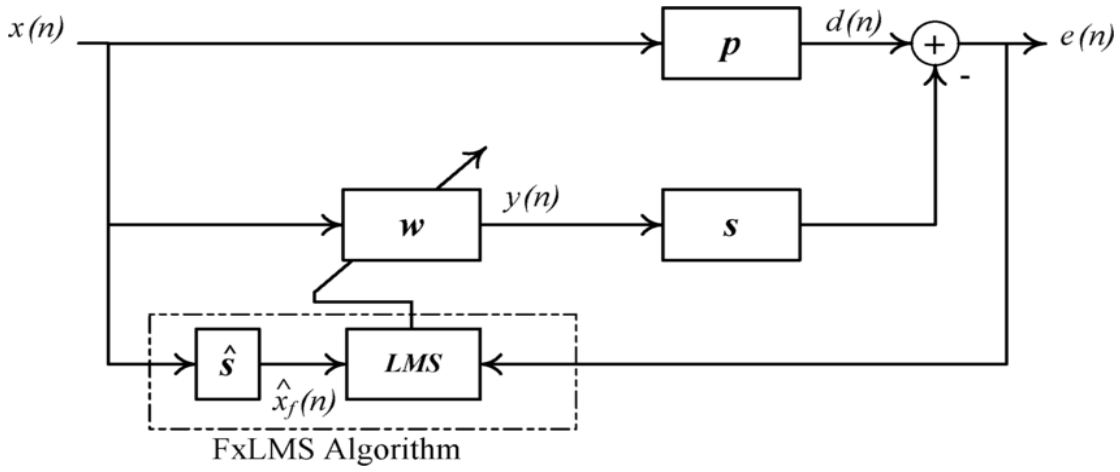

Figure 3. Adaptive filters in noise- cancellation using the FxLMS algorithm, [7]

Derivation of the Updated Adaptive Filter Coefficients Equation

Based on the gradient descent algorithm, the FIR model's updated coefficients, $\boldsymbol{w}$[n+1], are equal to their previous weights modified by a given percentage of the estimated gradient, as given below. The coefficients iteratively become updated in the direction of the negative of the gradient [1,3-6]

$$w[n+1] = w[n] + \mu\frac{\hat{\nabla}}{2}$$

Here, μ is the adaptation step size, a crucial parameter that determines by how much the filter coefficients are updated each iteration (the percentage of the estimated gradient that is being added to the current filter coefficients) [1,3-6]. The gradient and the estimated gradient are defined as follows, where $\frac{\partial}{\partial w}$ is the partial derivative with respect to the filter weights.

Gradient: $\nabla = \frac{\partial}{\partial w}[E(e^2[n])]$

Estimated Gradient: $\hat{\nabla} = \frac{\partial}{\partial w}[e^2[n]] = 2\frac{\partial}{\partial w}[e[n]]$

For the FxLMS algorithm, as derived in [6], the factor $\frac{\partial}{\partial w}[e[n]]$ is given by $-x_s$[n], which is the filtered x (Fx) or the reference input passed through $S(z)$. This explains the addition of a copy of the secondary path block at the input signal (Figure 3). The FxLMS calculates the updated filter coefficients as a function of x_s[n], and the feedback error signal, e[n]. The simplified updated coefficients of the FIR model equation then become:

$$\boldsymbol{w}[n+1] = \boldsymbol{w}[n] - \mu e[n]x_s[n]$$

The product of the error and the filtered input signal can be interpreted as the cross-correlation between the two signals [5,6]. Cross- correlation is a measure of the similarity between two signals as a function of their time shift. Therefore, the filter coefficients are updated such that the component of x_s[n] that is present on (or contributing to) the error signal is removed, reducing the cross-correlation between the two signals over time.

Having gained an understanding of the theoretical foundations on adaptive filtering and the noise cancellation theory, next we apply those concepts in the design and implementation of a practical noise - cancelling headphones system.

System Design and Overview

The system block diagram is shown in figure 4. The reference microphone captures the noise signal, s[n], this signal is then filtered through an anti-aliasing filter and digitized by an ADC. The digital signal is processed by the processing unit to generate the anti-noise, y[n]. The anti-noise then goes through a smoothing filter, and a DAC before being streamed through the headphones. The error microphone captures the resulting sound signal from the destructive interference of s'[n] and the streamed anti-noise signal. The feedback error signal together with the reference input signal filtered through the S(z) model are used by the LMS to update the FIR model coefficients, as was shown by the equation given above. This process repeats until convergence is reached, and the squared error is at a minimum.

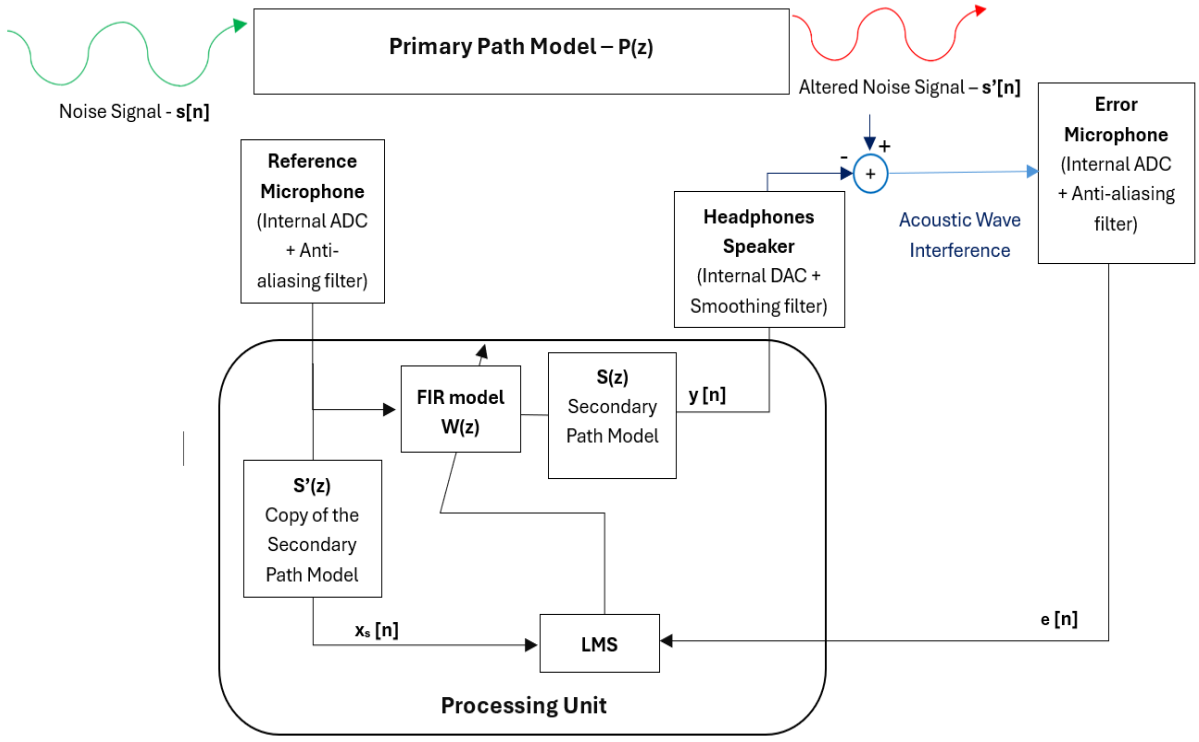

Figure 4. Block Diagram of Noise- Cancelling Headphones System

Figure 5 shows the system design, illustrating the position of every component. Figure 6 shows the 3D model of the casing. The error microphone is placed inside the earcup near the headphone speaker, while the reference microphone is mounted on the processing unit's casing such that its closest to the noise source. Since the reference microphone is in close proximity to the processing unit, it was decided to apply an aluminium tape to the casing to mitigate electromagnetic interference from the processing unit and to enhance system performance.

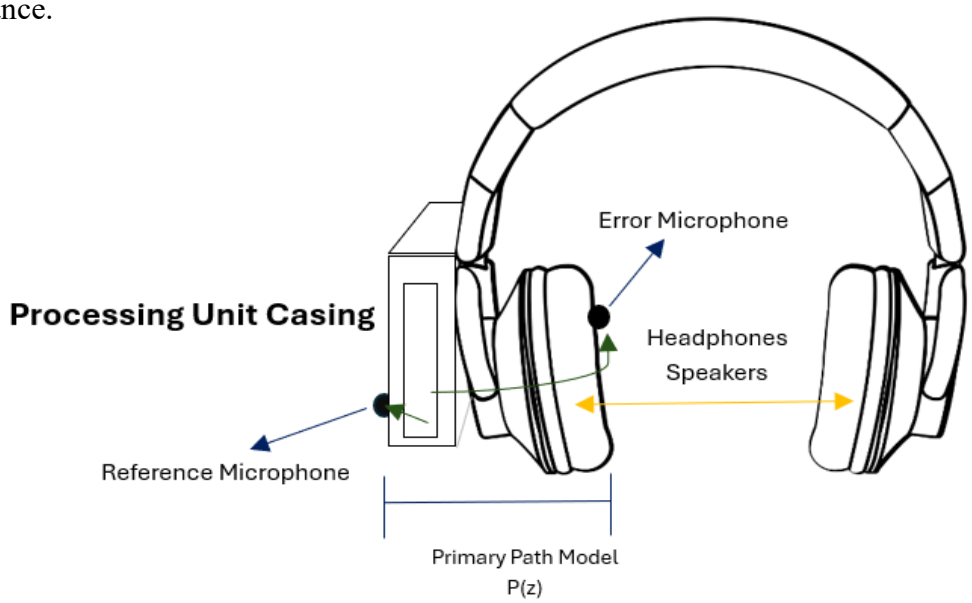

Figure 5. The Design of The Noise- Cancelling Headphones System

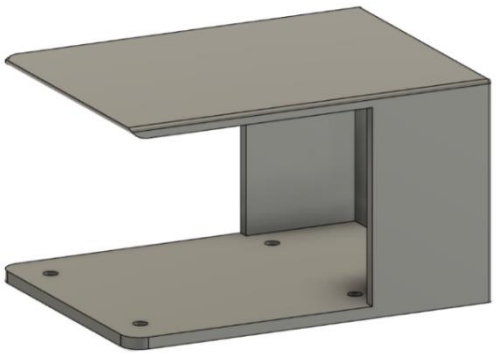

Figure 6. 3D Model Design of the Processing Unit Casin.

The next section details the hardware and software components used for the system implementation and the discuses the selection considerations.

Software and Hardware Specifications

The system uses a Raspberry Pi Model 3 as the processing unit, for it can handle complex and computationally- intensive programs required for this implementation. The microphones used are Micro-Electro-Mechanical-System (MEMS), featuring an internal anti-aliasing filter and an ADC, and outputting a 24- bit sample in the pulse width modulation (PCM) [2].

Python programming language was chosen for this implementation. Although C++ is more widely used because of its memory efficiency and speed [8], Python was preferred because to its ease of use, and extensive libraries. The main libraries used in this implementation include numpy, sounddevice, and adafilt. The real- time processing code used is given in Appendix A. Moreover, summarized below are the hardware components of the system:

- Headphones;
- Raspberry Pi Model 3;
- 3D printed casing of the processing unit;
- (2) - MEMS microphones (SPH0645LM4H);
- Jumper wires (male to male and female to male);
- 4-Pole Male to Male Audio Cable
- Adhesive suction holder (to mount the casing to the headphones);
- Aluminium tape;
- Plastic nails (to anchor the Raspberry Pi on the casing); and
- Double sided tape.

Methodology

This project implemented a systematic approach to design, optimize, and test an adaptive noise-cancellation system using the FxLMS algorithm. Key steps included identifying critical parameters, running simulations to determine optimal configurations, secondary path modelling, and testing the system in real-time conditions.

Parameter Optimization

The parameters considered for optimization are discussed below. The simulation was used to determine the optimal value of the adaptation step size (μ). The number of the adaptive filter coefficients and the buffer size parameters were selected based on the computational load capabilities of the processing unit used.

Adaptation Step Size: The step size was a primary focus of optimization, as it directly affects the system's balance between convergence speed and stability.

Number of Adaptive Filter Coefficients (Taps): The number of filter coefficients was another critical parameter. An 8192 - adaptive filter was chosen for the real – time processing implementation to accurately model the dynamic behavior of both primary paths, while ensuring computational efficiency.

Buffer Size: The buffer size is the number of data samples processed simultaneously during each iteration. The buffer size was set to 2048. This ensured maintaining a balance between processing speed and accuracy.

The simulation implementation discussed in the next section tested the performance of the system and enabled us to determine the optimal adaptation step size for the real-time processing implantation.

Simulation Implementation

The simulation implementation was conducted using predefined impulse response values for the primary (h_pri) and secondary (h_sec) paths in each scenario (Figure 3). These configurations reflect the complexity and characteristics of the system's environment, each presenting unique challenges. The system scenario specifications used in the simulation are summarized below. Detailed simulation code and the setup for testing various scenarios are provided in Appendix B.

Scenario 1 (*Simple Path Configurations*):

Primary Path (h_pri):

A single dominant impulse response was placed at index **50**, with a value of **0.8**.

Secondary Path (h_sec):

A smaller impulse response was placed at index 10, with a value of 0.5.

Scenario 2 (*Complex Configuration*):

Primary Path (h_pri):

Two impulse responses were placed at indices **30** and **40**, with values of **0.5** and **-0.3**, respectively.

Secondary Path (h_sec):

A single impulse response was placed at index **20**, with a value of **-0.4**.

Scenario 3 (Moderately *Complex Configuration)*:

Primary Path (h_pri):

A single impulse response was placed at index **25**, with a value of **0.7**.

Secondary Path (h_sec):

A single impulse response was placed at index **15**, with a value of **0.3**.

The output of the simulation showing the moving rms value of the error signal for each scenario as a function of the iterations of the system are given in Figures 7-9, for a step size of 0.01, 0.05, and 0.1 respectively. The simulations provided a nuanced understanding of how step size influences error curves and convergence rates. The results confirmed that a step size of around **0.05** delivered the best balance between speed and stability. The analysis of the results obtained is given below:

Scenario 1: Showed faster convergence but experienced higher oscillations, making it less stable for real-world applications.

Scenario 2: Offered a balanced performance, achieving both quick convergence and minimal oscillations, making it the most optimal choice.

Scenario 3: Demonstrated moderate performance, with slower convergence and increased residual error compared to Scenario 2

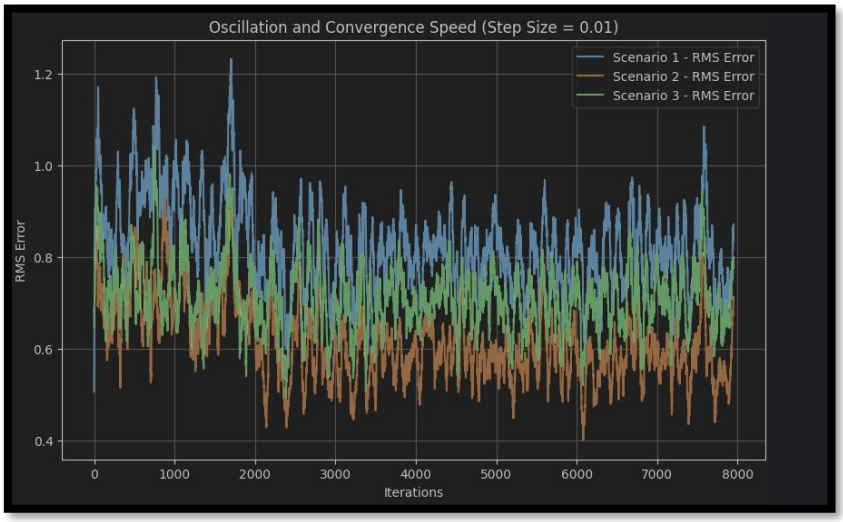

Figure 7 Simulation Result for a Step Size of 0.01

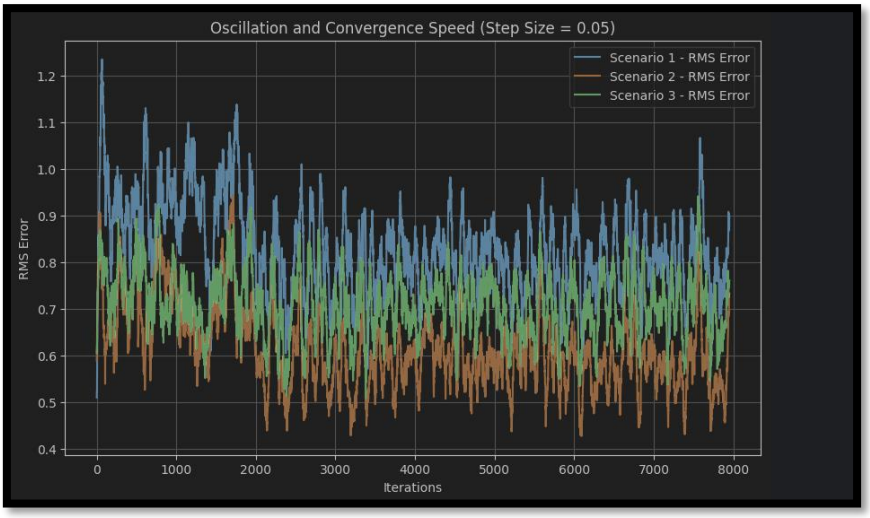

Figure 8 Simulation Result for a Step Size of 0.05

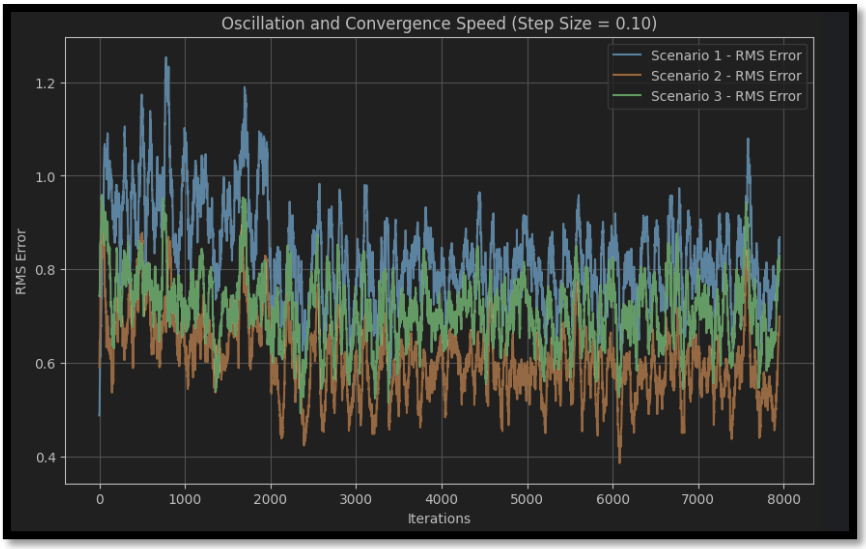

Figure 9 Simulation Result for a Step Size of 0.1

Real-Time Processing Implementation

This section discusses the circuit connections of the microphones and headphones to the Raspberry pi and presents the final prototype design. The details of setting up the Raspberry Pi and running the code is given in Appendix D.

In order to connect the I2S microphones to the Raspberry Pi the following connection (Figure 10) was used as reference. The headphone was connected to the Rassberry Pi's audio jack via a 4-pole male to male wire.

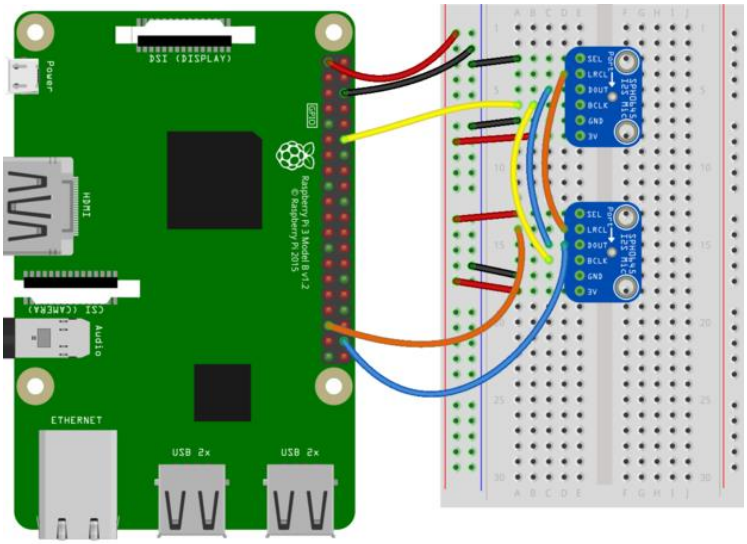

Figure 10. Circuit Connection of the microphones to the Raspberry Pi

The final prototype is shown in Figure 11. The Next section discusses secondary path effects modelling, the experimental setup used to obtain the model.

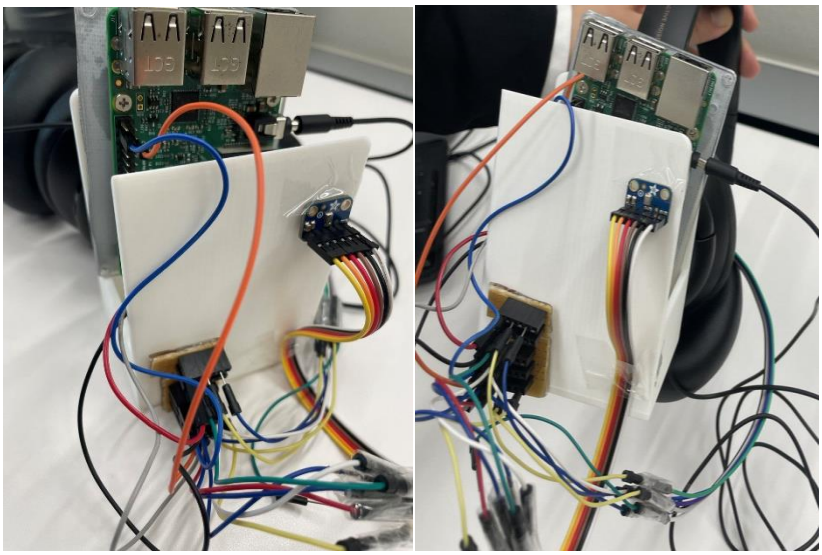

Figure 11. Prototype

Secondary Path Effects Modelling

The model S(z) can be obtained using adaptive filters in system identification as shown in figure 12. S(z) can be modelled either offline (separately) or simultaneously with the primary path model. Offline modelling removes the primary path during the identification of S(z). The simultaneous modelling approach yields a more effective noise-cancelling system, nevertheless, it requires a high-performance processing unit, as any processing delays during can diminish the system's effectiveness [3,4]. In this project, S(z) was modelled separately due to limited capabilities of the processing unit used.

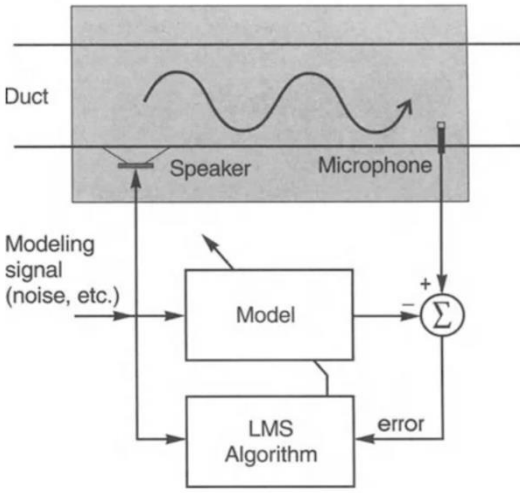

Figure 12. Adaptive filters in system identification, modelling the secondary path effects, S(z) [4].

As figure 4 illustrated, both the actual and adaptive FIR models are excited using a known signal, s[n]. (therefore, requiring no reference microphone to capture it). Ideally, the medium between the noise source and the microphone is a large duct, eliminating any environmental interferences in the modelling. Eventually, the converged model would account for delays, distortion and noise contributions by the non-idealizations of the system and the phase shift experienced by the sound signal as travels from the noise source to the microphone, which is P(z). Figure 13 shows the experimental setup used in modelling S(z). The code used in modelling the secondary path effects is given in Appendix C.

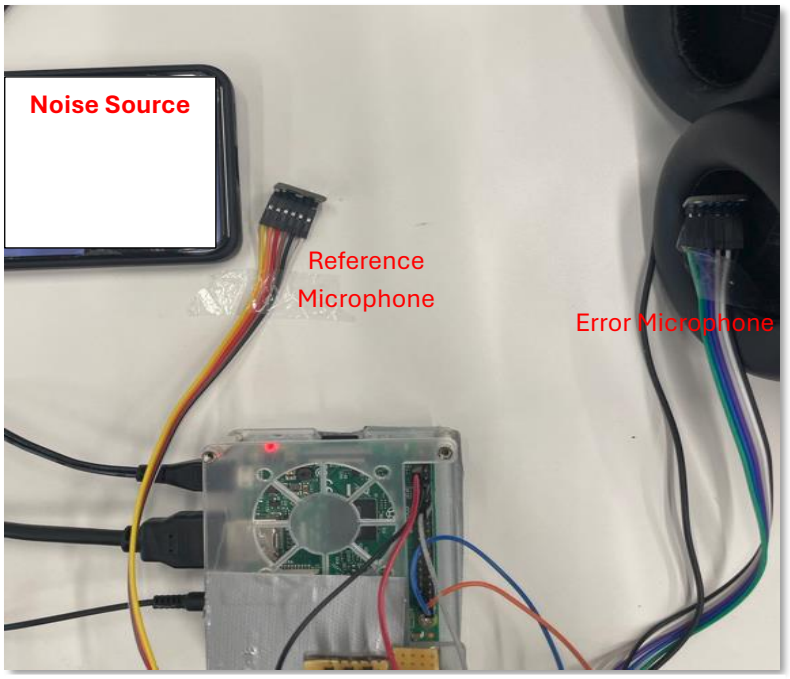

Figure 13. Experimental Setup to obtain the S(z) Model

Figure 14 shows the terminal output results. The obtained FIR coefficients of S(z) were saved in a txt file and were used as a fixed filter model in the main adaptive filter code. Due to an unresolved error in the main code, we could not verify the working of the adaptive filtering headphones. Since the codes for both the secondary path effects modelling and the main code are very similar, if given enough time, we would have been able to troubleshoot the issue easily.

Figure 14. Terminal Output Showing the FIR Coefficient of the Secondary Path Effects model, S(z)

Conclusion and Future Work

Although we were unable to resolve the issue in the main code, this project provided a valuable journey into adaptive noise cancellation systems. In this project, we explored the concept of adaptive noise cancellation systems, and their practical implementation through the FxLMS algorithm. Addressing realistic problems and testing real-world systems taught us a lot about noise-cancellation technology. The project gave us the opportunity to take theoretical knowledge in adaptive signal processing and apply it to a practical application. Simulation, optimization of parameters, and testing of real-time systems are some of the critical skills that we acquired, which will be helpful in future engineering projects. The errors encountered in both hardware and software integration improved our problem-solving ability and teamwork.

Future work may give a deeper focus on optimizing the three key parameters, filter tap size, buffer size, and step size, which can yield better noise cancellation performance. Additionally, more complex and efficient methods found in the literature may be investigated, including adaptive gain algorithms and FPGA-based implementations for possible inclusions into the design [9,10]. This will further enhance system performance, lessen computational loads, and enrich real-time responsiveness.

Moreover, better and a wider range of testing methodologies shall be developed for the assessment of the performance of the system based on approaches considered in earlier works. Such advanced metrics would give further insights into how noise cancellation would perform in field conditions under variable environments.

References

[1]. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing*. Pearson, 2007, pp. 880–909.

[2]. Piotr Sykulski and Konrad Jędrzejewski, "Adaptive Active Noise Cancelling System for Headphones on Raspberry Pi Platform," Oct. 2020, doi: https://doi.org/10.23919/spw49079.2020.9259141.

[3]. L. Lu *et al.*, "A survey on active noise control in the past decade—Part I: Linear systems," *Signal Processing*, vol. 183, p. 108039, Jun. 2021, doi: https://doi.org/10.1016/j.sigpro.2021.108039.

[4]. Dr. Steve Arar, "More on Noise-Canceling Headphones: Adaptive Controllers in Active Noise Control Systems," *Allaboutcircuits.com*, Nov. 18, 2020. https://www.allaboutcircuits.com/news/more-on-noise-canceling-headphones-adaptive-controllers-in-active-noise-control-systems/ (accessed Dec. 07, 2024).

[5]. A. K. Wang, B. Tse, and W. Ren, "Adaptive Active Noise Control for Headphones Using the TMS320C30 DSP," Jan. 1997.

[6]. R. Kwiatkowski, "Gradient Descent Algorithm — a deep dive," *Medium*, May 24, 2021. https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21.

[7]. I. T. Ardekani and W. H. Abdulla, "Effects of Imperfect Secondary Path Modeling on Adaptive Active Noise Control Systems," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 5, pp. 1252–1262, Sep. 2012, doi: https://doi.org/10.1109/tcst.2011.2161762.

[8]. C. Oddy, "C++ vs Python - What You Need to Know," *KO2 Recruitment*, Apr. 29, 2021. https://www.ko2.co.uk/c-plus-plus-vs-python/ (accessed Dec. 07, 2024).

[9]. B. Reshma and K. A. Kiran, "Active noise cancellation for in-ear headphones implemented on FPGA," IEEE Xplore, Jun. 01, 2017. https://ieeexplore.ieee.org/abstract/document/8250533 (accessed Dec. 10, 2022).

[10]. X. Shen, D. Shi, W.-S. Gan, and Santi Peksi, "Adaptive-gain algorithm on the fixed filters applied for active noise control headphone," Mechanical Systems and Signal Processing, vol. 169, pp. 108641–108641, Dec. 2021, doi: https://doi.org/10.1016/j.ymssp.2021.108641.

Appendices

Appendix A: Real – Time Processing Code

```
import sys
import threading
import numpy as np
import sounddevice as sd
from adafilt import MultiChannelBlockLMS
from vispy import app, scene
from vispy.scene.visuals import Text
from vispy.scene.widgets import Grid, ViewBox

# Parameters for real-time processing
# Query the available sound devices and print the information
print(sd.query_devices())

# Set input and output device indices (microphones and speaker)
input_device = 6  # Input device (microphones)
output_device = 6  # Output device (speaker via headphone jack)

# Get information about the selected input device
device = sd.query_devices(input_device)
print(device)

# Get the sample rate from the input device (sampling rate for audio
processing)
samplerate = device["default_samplerate"]
print(samplerate)

# Set latency for the audio stream (low latency for real-time processing)
latency = "low"

# Set blocksize, the number of samples to process in one iteration (affects
real-time performance)
blocksize = 2048  # Size of the audio block for processing (e.g., number of
samples)

# Data type for the audio processing (32-bit floating point)
dtype = "float32"

# Define the number of channels for input (2 channels: reference and error
microphones) and output (1 channel: speaker)
channels = inch, outch = (2, 1)  # 2 input channels (reference and error),
1 output channel (speaker)

# Adaptive filter (LMS)
# MultiChannelBlockLMS is used here, which is an adaptive filter that
processes blocks of data.
filt = MultiChannelBlockLMS(
    Nin=2,  # Number of input channels (reference and error signals)
    length=blocksize * 4,  # Length of the adaptive filter (number of taps)
    blocklength=blocksize,  # Block size to process at a time
    leakage=0.99999999,  # Leakage factor for stability in filtering
    stepsize=0.1,  # Stepsize for adapting the filter
    constrained=True,  # Apply constraints to ensure stability
)

# Initialize arrays for error signal (e) and output signal (y)
e = np.zeros((blocksize, 1))  # Error signal (difference between desired
```

```
and actual)
y = np.zeros((blocksize, 1))  # Output signal (anti-noise generated by
filter)

# Initialize the secondary path model, initially set to zero
h_sec = np.zeros(blocksize * 4)  # Model of the secondary path (for
filtering the reference signal)

# Define a Filter Monitor for visualization
class FilterMonitor:
    def __init__(self):
        # Create arrays to store the filter weights for plotting
        N = blocksize * 4  # Filter length
        self.pos1 = np.zeros((N, 2))  # Positions for plotting filter
weights
        self.pos2 = np.zeros((N, 2))
        self.pos1[:, 0] = np.arange(N) / samplerate  # Time axis
        self.pos2[:, 0] = np.arange(N) / samplerate  # Time axis

        # Colors for the filter weights plots
        self.color1 = (1, 0, 0)  # Red for filter weight 1
        self.color2 = (0, 1, 0)  # Green for filter weight 2

        # Canvas setup for visualizing the filter weights
        canvas = scene.SceneCanvas(keys="interactive", show=True)
        main_grid = canvas.central_widget.add_grid()

        grid = Grid(border_color="r")  # Grid layout for the plot
        info = scene.widgets.ViewBox(border_color="b")  # Viewbox for
additional information
        info.camera = "panzoom"  # Set the camera type
        info.camera.rect = -1, -1, 2, 2  # Set initial view area
        info.stretch = (1, 0.1)  # Stretch factor for layout

        main_grid.add_widget(grid, row=0, col=0)
        main_grid.add_widget(info, row=1, col=0)

        # Add axis widgets for plotting
        x_axis = scene.AxisWidget(orientation="bottom")  # Bottom axis for
time
        x_axis.stretch = (1, 0.1)

        y_axis = scene.AxisWidget(orientation="left")  # Left axis for
filter weights
        y_axis.stretch = (0.1, 1)

        grid.add_widget(x_axis, row=1, col=1)
        grid.add_widget(y_axis, row=0, col=0)

        # Create viewbox for plotting the filter weights
        viewbox = grid.add_view(row=0, col=1, camera="panzoom")
        x_axis.link_view(viewbox)
        y_axis.link_view(viewbox)

        # Add CPU load information display
        text = Text(text="TEXT", color=(1, 1, 1, 1), parent=info.scene)
        text.font_size = 18
        self.text = text

        # Line plot for the filter weights
        self.line1 = scene.Line(self.pos1, self.color1,
```

```
parent=viewbox.scene)
        self.line2 = scene.Line(self.pos2, self.color2,
parent=viewbox.scene)

        # Auto-scale view
        viewbox.camera.set_range()

    # Update method to refresh the visual representation of filter weights
and CPU load
    def update(self, w, load):
        self.pos1[:, 1] = w[:, 0, 0]  # Update first filter weight plot
        self.pos2[:, 1] = w[:, 0, 1]  # Update second filter weight plot
        self.line1.set_data(pos=self.pos1, color=self.color1)
        self.line2.set_data(pos=self.pos2, color=self.color2)
        self.text.text = f"CPU load: {load * 100:.1f}%"  # Update CPU load
display

# Initialize the FilterMonitor instance for visualization
filter_monitor = FilterMonitor()

# Callback function for audio stream (real-time audio processing)
def callback(indata, outdata, frames, time, status):
    global e, y, h_sec

    if status:
        print("Callback status:", status)

    try:
        # Reference signal from microphone 1 (input channel 1)
        x = indata[:, 0:1]  # Reference microphone (index 0)

        # Error signal from microphone 2 (input channel 2)
        d = indata[:, 1:2]  # Error microphone (index 1)

        # Generate the output signal (anti-noise) through the adaptive
filter
        # Filter the reference signal using the secondary path model
        fx = np.convolve(x[:, 0], h_sec, mode="same")  # Filtered reference
signal

        # Use the adaptive filter to predict the output signal (anti-noise)
        y[:] = filt.filt(fx[:, None])

        # Calculate the error signal (difference between desired signal and
actual output)
        e[:] = d - y

        # Adapt the filter weights based on the error signal
        filt.adapt(fx[:, None], e)

        # Send the anti-noise signal to the speaker (output channel)
        outdata[:] = y

        # Update the filter visualization (show filter weights and CPU
load)
        filter_monitor.update(filt.w, stream.cpu_load)

    except StopIteration:
        raise sd.CallbackAbort  # If iteration stops, abort the callback
    except Exception as e:
        print(type(e).__name__ + ": " + str(e))  # Handle any other
```

```
exceptions
        raise sd.CallbackAbort  # Abort the callback on error

# Event for thread synchronization (ensure stream has finished processing)
callback_finished_event = threading.Event()

# Stream setup for real-time audio processing (using the selected
input/output devices)
stream = sd.Stream(
    device=(input_device, output_device),  # Set input and output devices
    samplerate=samplerate,  # Set the sample rate
    blocksize=blocksize,  # Set the block size (number of samples per
block)
    dtype=dtype,  # Set the data type (32-bit float)
    latency=latency,  # Set the latency for real-time processing
    channels=channels,  # Set the number of input and output channels
    callback=callback,  # Set the callback function for processing audio
    finished_callback=callback_finished_event.set,  # Event when the stream
finishes
)

try:
    # Run the audio stream and start the real-time FxLMS process
    with stream:
        app.run()  # Run the Vispy app for visualization
except KeyboardInterrupt:
    sys.exit(0)  # Exit gracefully on keyboard interrupt
 plt.plot(filt.w)
plt.show()
```

Appendix B: Real – Simulation Code

```
import numpy as np
import matplotlib.pyplot as plt
from adafilt import FastBlockLMSFilter, FIRFilter
from adafilt.io import FakeInterface
from adafilt.utils import wgn

# Define impulse responses for different scenarios
def setup_paths(scenario):
    h_pri, h_sec = np.zeros(64), np.zeros(64)
    if scenario == "Scenario 1":
        h_pri[50] = 0.8
        h_sec[10] = 0.5
    elif scenario == "Scenario 2":
        h_pri[30], h_pri[40] = 0.5, -0.3
        h_sec[20] = -0.4
    elif scenario == "Scenario 3":
        h_pri[25] = 0.7
        h_sec[15] = 0.3
    return h_pri, h_sec

# General parameters
length = 64
blocklength = 4
n_buffers = 2000
estimation_phase = 500

# Generate a predefined noise signal
np.random.seed(123)  # Fix seed for reproducibility
predefined_noise_signal = np.random.normal(0, 1, size=n_buffers *
blocklength)

# Step sizes to iterate through
step_sizes = np.arange(0.01, 0.11, 0.01)

# Run simulations for each step size
all_results = {}

for step_size in step_sizes:
    scenario_results = {}
    for scenario in ["Scenario 1", "Scenario 2", "Scenario 3"]:
        # Set up paths
        h_pri, h_sec = setup_paths(scenario)

        # Initialize simulation with predefined noise signal
        sim = FakeInterface(
            blocklength, predefined_noise_signal, h_pri=h_pri, h_sec=h_sec,
noise=wgn(predefined_noise_signal, 20, "dB")
        )
        filt = FastBlockLMSFilter(length, blocklength, stepsize=step_size,
leakage=0.99999, power_averaging=0.9)
        filt.locked = True
        plant_model = FIRFilter(np.zeros(blocklength + length))
        adaptive_plant_model = FastBlockLMSFilter(length, blocklength,
stepsize=step_size, leakage=0.99999)

        # Logging
        elog = []

        # Simulation loop
```

```
        y = np.zeros(blocklength)
        for i in range(n_buffers):
            if i < estimation_phase:
                v = np.random.normal(0, 1, blocklength)  # Estimation noise
            else:
                v = np.random.normal(0, 0.01, blocklength)  # Operation
noise
                adaptive_plant_model.stepsize = step_size

            x, e, u, d = sim.playrec(-y + v)
            plant_model.w[blocklength:] = adaptive_plant_model.w
            fx = plant_model(x)

            if i >= estimation_phase:
                filt.adapt(fx, e)
                y = filt.filt(x)

            elog.append(np.abs(e))

        # Store results for each scenario
        scenario_results[scenario] = np.array(elog)

    # Store results for each step size
    all_results[step_size] = scenario_results

# Generate and save graphs for each step size
for step_size, results in all_results.items():
    plt.figure(figsize=(10, 6))
    for scenario, elog in results.items():
        # Calculate moving RMS error
        rms_error = np.sqrt(np.convolve(elog.flatten() ** 2, np.ones(50) /
50, mode="valid"))
        plt.plot(rms_error, label=f"{scenario} - RMS Error")

    plt.title(f"Oscillation and Convergence Speed (Step Size =
{step_size:.2f})")
    plt.xlabel("Iterations")
    plt.ylabel("RMS Error")
    plt.legend()
    plt.grid(True)
    plt.show()
```

Appendix C: Secondary Path Effects Modelling Code

```
import numpy as np
import sounddevice as sd
from adafilt import FastBlockLMSFilter

# Parameters for real-time processing
print(sd.query_devices())  # Query available sound devices

# Set input and output device indices (microphones)
input_device_a = 6  # Input device (reference microphones)
input_device_b = 6  # Input device (error microphone)

# Get information about the selected input device
device = sd.query_devices(input_device_a)
print(device)

# Get the sample rate from the input device (sampling rate for audio
processing)
samplerate = device["default_samplerate"]
print(f"Samplerate: {samplerate}")
blocksize = 2048  # Block size for real-time processing
filter_length = blocksize * 4  # Length of the adaptive filter (number of
taps)
latency = "low"  # Low latency for real-time processing
channels = 2  # Two channels: reference and error microphones

# Create an adaptive filter for estimating the secondary path (h_sec)
filt = FastBlockLMSFilter(length=filter_length, blocklength=blocksize)

# This will hold the estimated secondary path model (h_sec)
h_sec_estimate = np.zeros(filter_length)

# Convergence check parameters
threshold = 1e-6  # Small change to determine if weights have converged
stable_iterations = 50  # Number of iterations to check for convergence
convergence_counter = 0  # Counter for iterations where weights change is
below threshold

# Track previous weights for comparison
previous_weights = np.zeros(filter_length)

# Callback function for real-time audio processing
def callback(indata, outdata, frames, time, status):
    global h_sec_estimate, filt, previous_weights, convergence_counter

    if status:
        print("Callback status:", status)

    # Reference signal comes from the reference microphone (index 0)
    x = indata[:, 0]  # Reference microphone signal

    # Error signal comes from the error microphone (index 1)
    d = indata[:, 1]  # Error microphone signal

    # Use the adaptive filter to estimate the secondary path
    # The filter output is used to estimate the secondary path
    yhat = filt.filt(x)  # Apply filter to the reference signal
```

```
    # Compute the error signal (difference between desired and predicted)
    e = d - yhat  # The error between the desired signal and the predicted
output

    # Adapt the filter weights to minimize the error signal
    filt.adapt(x, e)

    # Check the change in filter weights to detect convergence
    weight_change = np.linalg.norm(filt.w - previous_weights)  # Euclidean
distance between current and previous weights

    if weight_change < threshold:
        convergence_counter += 1
    else:
        convergence_counter = 0  # Reset counter if the change is too large

    # Store the current weights for the next iteration
    previous_weights = filt.w.copy()

    # If the weights have been stable for a sufficient number of
iterations, print the coefficients
    if convergence_counter >= stable_iterations:
        print("Filter coefficients have converged!")
        print("Estimated Secondary Path Model (FIR Coefficients):")
        print(filt.w)  # This prints the filter weights representing the
secondary path (h_sec)

# Create a stream for real-time audio processing
stream = sd.Stream(
    device=(input_device_a, input_device_b),
    samplerate=samplerate,
    blocksize=blocksize,
    dtype="float32",
    channels=channels,
    latency=latency,
    callback=callback,
)

# Start the audio stream and process in real-time
try:
    with stream:
        print("Starting real-time secondary path estimation...")
        sd.sleep(10000)  # Run for 10 seconds (or however long needed for
system identification)
except KeyboardInterrupt:
    print("Stream interrupted.")
```

Appendix D: Steps for Setting up the Raspberry Pi and Running the Code

Setting Up the Raspberry Pi

- Using the Raspberry Pi Imager application the operating system was downloaded onto a SD card. The RASPBERRY PI OS (32-BIT) was chosen for this project as it a better fit for Raspberry Pi 3 that is used for this project.
- The Microphone was configured on to the Raspberry Pi by going to the configuration file of the Raspberry pi.
    - On the terminal the following needs to types to access that file
    boot/firmware/config.txt
    - Once in the file the following text was added to ended of the document to configure the microphones.
    dtoverlay=googlevoicehat-soundcard
- The headphones did not require configuring.

Running the Program

- In order to run the program on Raspberry Pi a Python virtual in environment was required and it had to created on the terminal.
    1. The package required to use virtual environment is first downloaded.
    pip install virtualenv
    2. Then a virtual environment is created named *myenv*.
    python -m venv myenv
    3. To access the environment the through the terminal the following is typed.
    source myenv/bin/activate
    4. Before running the code all the necessary libraries need to downloaded within the environment. The ones needed for running the system_identification.py and real-time-visuals-2-channel.py were the following.
    pip install numpy
    pip install matplotlib
    pip install sounddevice
    pip install vispy

5. To run a code within the environment the terminal needs to run the program.

python /home/dsp/projectdsp/system_identification.py

- The system_identification.py program was then run using the environment. This code estimates the secondary path model in real-time for adaptive noise cancellation by using an adaptive filter to process reference and error microphone signals, identifying filter coefficients when convergence is achieved.

python /home/dsp/projectdsp/system_identification.py

- The real-time-visuals-2-channel.py which is meant to implement the real-time adaptive noise cancellation system using the FxLMS algorithm, processing audio input to generate anti-noise output while visualizing filter weights and CPU load. It uses the coefficients found in the system_idenfication.py program which is stored in on a txt file to do the calculations.

python /home/dsp/projectdsp/ real-time-visuals-2-channel.py