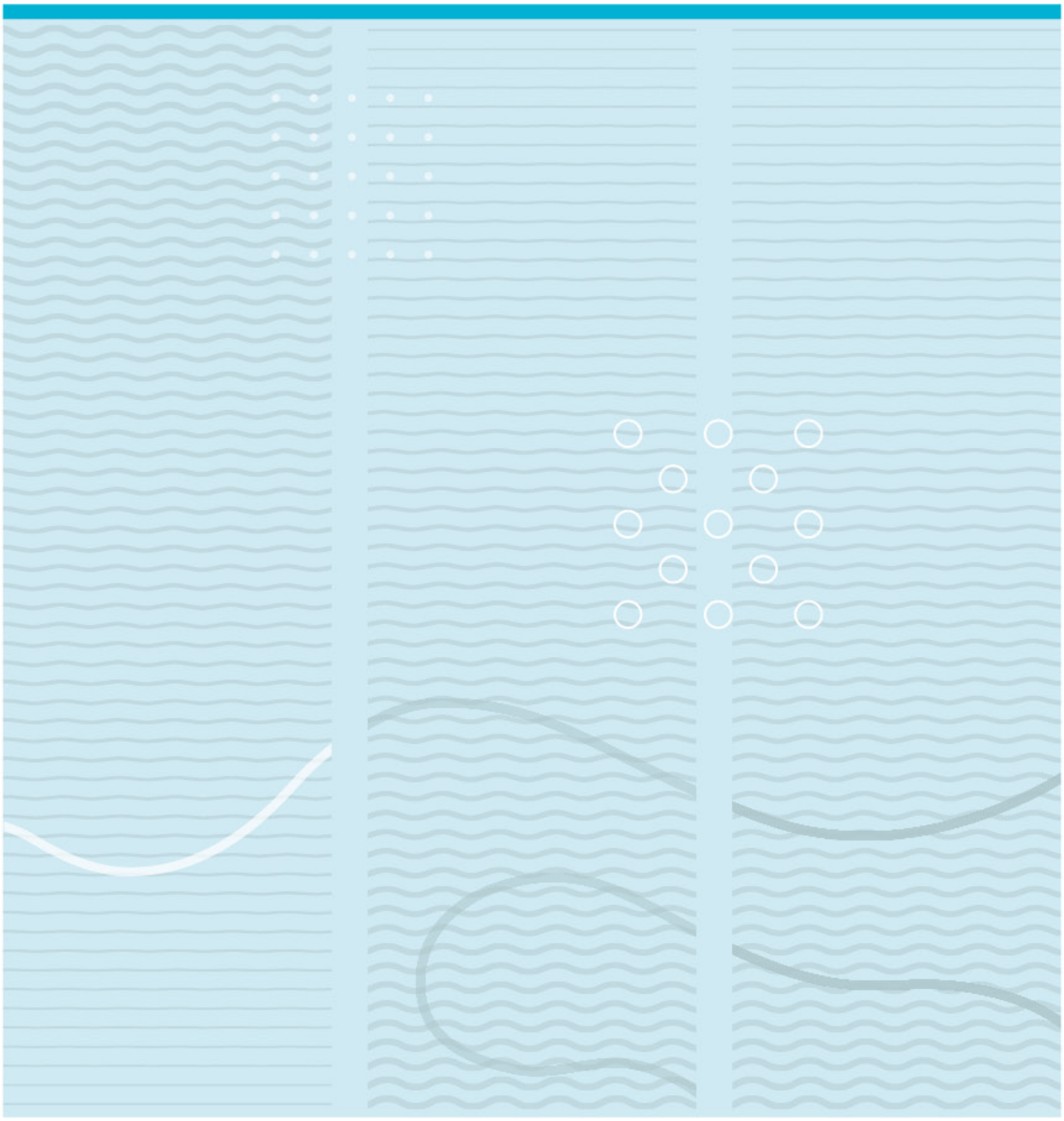
University of South-Eastern Norway
Faculty of Computer Science
–
Master's Thesis
Study programme: Master of Computer Science
Spring 2023

Md Muzibur Rahman

AudibleT - A real-time embedded tool for subjective assessment of audio parameters



University of South-Eastern Norway
Faculty of Computer Science
Department of Science and Industry Systems
PO Box 235
NO-3603 Kongsberg, Norway

http://www.usn.no

University of South-Eastern Norway

AudibleT - A real-time embedded tool for subjective assessment of audio parameters

Master’s Thesis in Computer Science

Md Muzibur Rahman

Supervisor(s): Sigmund Gudvangen

Department of Science and Industry Systems

University of South-Eastern Norway

Campus Kongsberg

May 26, 2023

Abstract

The project is focused on developing a real-time embedded application for the subjective assessment of audio parameters. This project's scope is limited to Just Noticeable Difference (JND) testing. The study around this project investigates the audio parameters suitable for JND while diving into different methods of testing such parameters. This tool is developed to allow real-time equalizer coefficients update as well as offer A/B/X testing of linear distortion. The system currently supports stereo audio output and a graphical user interface for running training and testing sessions with several configurable options and provides binomial distribution analysis and confusion matrix of each conducted ABX testing session. The tool utilized the JACK audio connection kit for low latency audio, and is optimized for general purpose computer as well as tested on Raspberry Pi 400.

Keywords: ABX Tests, Subjective Testing, Real-Time Audio, Embedded Systems, Just Noticable Difference, JACK Audio Connection Kit

Acknowledgements

I would like to commence by acknowledging the enormous contribution of my thesis supervisor, Sigmund Gudvangen. His guidance, mentorship, and endless support throughout the entire period of this thesis have been absolutely invaluable. His continuous encouragement and constructive feedback have been instrumental in refining my research and making this thesis what it is.

Furthermore, I extend my gratitude to all the program coordinators, professors of USN, and the people, resources, and circumstances that helped shape this thesis in direct and indirect ways, even if they cannot be enumerated individually. Their invisible touch has been indispensable.

Finally, this acknowledgment would not be complete without mentioning the immense support of my wife. Her unwavering belief, understanding, and patience provided a supportive backdrop against which the trials and tribulations of academic research could be weathered.

Last but not least, I would also like to thank myself for believing in me and continuously pushing myself hard to move forward, even when I felt utterly clueless.

Contents

List of Figures

List of Tables

Acronyms

JND	Just Noticable Difference
JACK	JACK Audio Connection Kit
UI	User Interface
JEC	JACK External Client
SoC	System–on–Chip
DSR	Design Science Research
IIR	Finite Impulse Response
JEC	Infinite Impulse Response
CSV	Comma–Separated Values
WAV	Waveform Audio File Format

1 Introduction

This project focuses on developing a real-time embedded application, AudibleT [i], for A/B/X testing [1] [2] [3] of specific audio parameters, addressing a critical need in the audio industry. Over the past three decades, listening tests have transitioned from tape recorders to computer software, enabling more sensitive and sophisticated evaluations. However, most existing software solutions are tailored to particular objectives, resulting in systems that neither allows on the fly equalizer alterations nor offer public access.

With recent advancements in computing power and CPU architectures, it is now possible to perform more complex and computationally intensive tests. Yet, many existing listening test tools do not leverage these developments. This project aims to develop an efficient audio processing and testing system comprising listener training, testing, and statistical analysis sections. The system will facilitate real-time filter parameter adjustments to allow subtle changes to audio stimuli, assess listeners' responsiveness to processed stimuli through interactive testing sessions, and present test results backed by statistical analysis.

Additionally, the system utilizes modern CPU resources through multi-threading and concurrent programming approaches, ensuring compatibility with both general-purpose systems and resource-constrained hardware like Raspberry Pi. Fundamentally, the project focuses on the system's real-time functionality, considering trade-offs between computational complexity, low latency, and dropout minimization. Furthermore, the goal extends to implementing tests for relevant audio parameters and ultimately providing an efficient, user-friendly, open-source application with the aim of turning subjective assessments into objective assessments of these parameters.

1.1 Motivation and Problem Statement

Significant work has been carried out in the field of audio testing in the past 50 years, but some of the work has become obsolete over the period. Researchers from different fields have been active in updating testing methods and building new tools. Still, most of them remained in-house and not accessible to the broader community. Moreover, silicon technology has evolved significantly in the last decades, and most of the software built before that is not able to utilize modern-day computing

[i]Given name of the software, will be used later in the report.

power. Besides, the majority of the tools built in recent years prepares the test stimuli offline before the testing takes place, and filter parameters cannot be changed in real time and are not particularly designed for cheap hardware. Therefore, the aim is to build a resource-efficient embedded tool that will allow users to run configurable A/B/X tests in real-time for listening tests and get statistics of the test results at the end. Besides, the tool is expected to be open-sourced so interested communities can access it for their purpose and contribute to further development.

Audio professionals could also use this tool in several industries, such as music, film, and gaming, to ensure that their audio products meet their specific standards and deliver their audiences the best possible sound experience. Additionally, this tool could be helpful for consumer electronics companies to test the performance of their audio products and improve their designs. This tool could also enable a more accurate and comprehensive evaluation of audio performance by allowing users to test multiple algorithms or configurations simultaneously and compare their results. Moreover, this tool could facilitate collaboration and sharing of ideas among audio professionals by providing a platform for conducting and sharing experiments and results.

The system primarily deals with the real-time aspects of the audio testing approaches, which are also related to embedded Linux applications. Therefore, this system's real-time core can also be widely adopted by automotive applications such as buses, trains, and even airplanes since the application can be ported on cheap single-boarded Linux computers.

1.2 Research Questions

The primary project goal was to build an embedded tool to enable configurable Just Noticeable Difference (JND) testing for two-channel audio in real-time, with the possibility of adding more testing methods, such as rank order tests [4]. The system should consist of multiple threads, where at least one thread should be dedicated to real-time audio processing. The real-time worker thread should also get the highest priority from the operating system. Furthermore, the system should process the audio buffers in suitable chunks to keep the latency to a minimum while preventing audio sample dropouts. Moreover, the system should be resource efficient, and the compatibility should be wide enough to be accessible by the interested communities. The tool is aimed to be built on top of JACK [5], a low latency audio server. However, the relatively newly developed PipeWire [6] has a JACK-compatible API layer that allows JACK-compatible applications to run on PipeWire. This also creates a possibility to analyze and compare the overall performance of the software utilizing both audio servers at some point in time. The project should also find appropriate audio test parameters, enable testing for such parameters, and produce statistics for the users.

Considering the initial goals of the project, a set of research questions is listed:

- **RQ1**: What is the appropriate buffer scheme with low overhead to prevent sample dropouts while maintaining a minimum or no audible latency in real-time audio processing and be designed efficiently for both general-purpose computers and cheap hardware?
- **RQ2**: What are the audio parameters suitable for JND testing, and how should those tests be conducted?
- **RQ3**: How to improve upon existing tools, in terms of processing audio in real-time, which statistics data should be generated, ease of use, and presentation of results?

1.3 Objectives

The project development is expected to have the following phases based on the initial goals and challenges.

- Investigation to find audio parameters suitable for JND testing and methods to perform the tests.
- Design the system in a resource-efficient manner to be able to deploy on low-performance processors.
- Create a user-friendly user interface to perform tests.
- Implement the audio processing engine with consideration of efficiency, performance, and real-time issues.
- Consider thread safety and proper compiler optimization level while writing the program. Improper optimization levels can affect the program behavior significantly on low-cost processors [7].
- Perform benchmark of the code using JACK API.
- Test, analyze, and compare the results from different hardware with profiling tool.

1.4 Assumptions and Limitations

This project is developed under a set of assumptions and comes with certain limitations for reasons such as aiming to complete a set of tasks realistic enough to be completed within the timeline provided. The constraints are acknowledged, as detailed below:

1.4.1 Assumptions

- Hardware Compatibility: The developed tool assumes a modern CPU architecture or System-on-Chip (SoC) as the underlying hardware. It may not function optimally on older or less capable systems.
- Operating Systems: The system has been primarily designed for and tested on Linux (Ubuntu 22.04.1 LTS) [8], MacOS Ventura [9], and Raspberry Pi OS [10], leveraging JACK. Compatibility with other operating systems and environments is not guaranteed.
- User Familiarity: The system assumes a certain level of user familiarity with the principles of audio testing and JND.
- Modular Development: The system has been developed using a modular approach, with separate, interchangeable components for different functionalities. This assumes that any changes or updates can be made to individual modules without affecting the overall design of the system.

1.4.2 Limitations

- Scope of Audio Testing: Currently, the system is primarily designed for JND testing of linear distortions through A/B/X testing. Other forms of testing, such as non-linear distortions or surround sound capabilities, are not included.
- Platform Dependency: While efforts have been made to design a resource-efficient system, the real-time performance may depend on the specific hardware and software environment in which it is used.
- User Interface: While the system does include a graphical user interface, its design and features have been optimized for functionality rather than extensive user experience design.

- Modular Dependency: While the modular development approach allows for a degree of flexibility, it may also lead to dependencies where changes in one module could require changes in others to maintain compatibility and seamless operation.

1.5 Thesis Contributions

The outcomes and contributions from the developed software are as follows:

- A multi-threaded real-time JACK client with the potential to be used for other real-time audio signal processing tasks and be released as an open-source application that is useful for others as a basis for similar applications.
- An efficient enough solution to be able to run and perform well on low-cost hardware and thus be deployed for embedded sound systems.
- A testing tool with an easy-to-use user interface that makes the testing suitable for users with both technical and non-technical backgrounds.
- It is also expected to be able to perform benchmarking/characterization of low-overhead real-time processing in the context of a general-purpose operating system (Linux).

1.6 Thesis Outline

This thesis is organized into the following main chapters, each serving a specific purpose in the overall narrative of this research project.

In **Chapter** 2 an overview of the field of audio quality assessment is provided. It explores the historical context and the current state of the audio industry to some extent, presenting a broad view of the importance of audio quality in various sectors. Prior research relevant to this project is reviewed, and the theoretical foundations that support the development of an audio-testing tool are examined.

In **Chapter** 3, the research methodology for this project is outlined. The chapter details the steps taken to develop the real-time embedded application for Just Noticeable Difference (JND) testing of audio. The strategies used to answer the research questions and accomplish the project objectives are explored, presenting the journey of creating an efficient, user-friendly system for real-time audio processing and statistical analysis.

In **Chapter** 4, the conceptual design and architecture of the proposed audio testing tool is presented. This chapter elaborates on the main components of the tool, their interactions, and the rationale behind key design decisions as well as the tools used to implement the software.

In **Chapter** 5, the process of implementing the tool and the necessary tests are discussed. This chapter walks through the methodologies and techniques employed during the development process and explores the challenges encountered and the solutions found.

In **Chapter** 6, the tests carried out to benchmark the tool, results of the various testing and evaluation activities are presented, followed by a detailed analysis of the results. The tool's performance, effectiveness, and potential impacts are critically examined in light of the research questions and project objectives.

Finally, **Chapter** 7 concludes the thesis with a summary of the research findings and their implications. The chapter also points out potential future research directions and enhancements to the audio testing tool.

2 Literature Review

Assessing audio quality is a pivotal element across various sectors within the audio industry. This not only includes areas like music and film but also permeates into sectors such as video gaming and even consumer electronics to a certain extend. Providing exceptional audio experiences to end users relies on developing reliable, efficient, and user-friendly tools for conducting these evaluations. This chapter presents an overview of the subject, diving into its historical context and examining the industry's current state. It also briefly reviews prior research related to this project and investigates the theoretical foundations underpinning an audio-testing tool's creation.

2.1 Historical Background

Since its inception, audio testing has advanced significantly. It has a rich history spanning several decades, marked by significant advancements in technology and methodologies. Early audio testing was primarily concerned with evaluating analog systems, and engineers and audio specialists relied on rudimentary methods and tools to assess audio quality. A few of the early significant events in the history of audio testing from different domains, such as psychoacoustics, testing methodologies, and introduction to the new technology, are as follows:

- The development of Equal-Loudness Contours by the Bell Telephone Laboratories in the 1930s [11], which offered a framework for comprehending how humans perceive sound levels at various frequencies and sound pressure levels.
- The invention of the first audio analyzers, such as the Hewlett-Packard 200 series [12] [13], in the 1950s and 1960s made it possible to measure audio characteristics like frequency response, distortion, and noise with greater accuracy. During this period, the Danish company Brüel & Kjær also contributed significantly to the field with their innovative audio measurement solutions, further improving the precision and capabilities of audio testing equipment [14].
- The emergence of digital audio technologies in the late 1970s and early 1980s prompted the development of new digital audio testing tools and significant advancements in audio testing procedures.
- The public release of the compact disc (CD) in 1982 [15] brought about a new digital audio format and necessitated more stringent both electronic and subjective testing techniques to guarantee the highest audio quality.

As audio technologies have become more complex, the field of audio testing has developed further, incorporating cutting-edge algorithms, digital signal processing methods, and automated testing procedures.

2.2 Current Industry and Field Conditions

The audio testing industry has seen significant advancements and widespread adoption of various techniques and methods. Perceptual coding techniques, such as the MPEG audio codec family (MP3, AAC), are prevalent [16], relying on psychoacoustic principles to compress audio data while maintaining perceptual quality. Objective quality measurements methods, like Perceptual Evaluation of Audio Quality (PEAQ) and POLQA (Perceptual Objective Listening Quality Assessment) [17], have also gained interest, providing numerical scores to assess audio quality based on mathematical models of human perception.

Subjective listening tests, such as ITU-R BS.1116 [18] and ITU-T P.800 [19], continue to be used, involving human listeners rating audio quality under controlled conditions. These tests remain a critical benchmark for validating the performance of objective quality measurement of perceived audio quality. Additionally, specialized software tools and platforms, like Audio Precision, MATLAB, and REW (Room EQ Wizard), have emerged, offering a broad range of audio testing and analysis capabilities, from basic measurements to advanced signal processing and simulation tasks.

Despite the advancements in the field, challenges remain. Some of the current limitations and challenges in the audio testing industry include:

- The lack of updating equalizer coefficients in real-time.
- Platform or hardware-specific compatibility issues that restrict the use of certain testing tools or methods.
- A lack of open-source or low-cost tools can limit access to advanced testing capabilities for smaller organizations, researchers, and hobbyists.
- The difficulty in ensuring the reliability and consistency of subjective listening tests due to factors such as listener fatigue, bias, and varying experimental conditions.

- The ongoing challenge to develop more accurate and reliable objective quality measurement methods that can closely approximate human perception of audio quality.

2.3 Theoretical Foundations

The creation of an audio testing tool necessitates a thorough comprehension of many fundamental theories, concepts, and frameworks, such as:
a) **Psychoacoustics**: Psychoacoustics is a branch of psychology and acoustics that studies the human perception of sound, including pitch, loudness, timbre, and localization. It helps understand how the human auditory system processes and interprets audio signals. Psychoacoustics plays a crucial role in the design of audio testing tools, as it provides insights into how listeners perceive audio artifacts and distortions, which can benefit the development of perceptual coding algorithms and subjective listening tests. This knowledge is vital for creating reliable and accurate audio testing tools that align with human auditory perception.

b) **Just Noticeable Difference (JND)**: Just Noticeable Difference [20], pp. 81-127, also known as the difference limen or differential threshold, is a key concept in psychoacoustics that quantifies the smallest change in a stimulus that a person can reliably detect. In audio testing, JND tests are often used to identify the minimum thresholds at which changes in audio parameters, such as loudness, frequency, or distortion, become perceptible to the human ear. Understanding JND enables the design of audio tests that accurately assess the audibility of audio artifacts or the quality of audio processing techniques.

c) **Metrics for Evaluating Audio Quality**: There exist multiple criteria for gauging the quality of audio. These include the Signal-to-Noise Ratio (SNR), a measure that assesses the desired signal level relative to the background noise level; and the Mean Opinion Score (MOS) [21], a subjective evaluation method that compiles individual listeners' judgments on audio quality. Grasping these criteria is vital for appropriately assessing audio quality. Moreover, it aids in deciphering test outcomes, which can subsequently guide the enhancement and fine-tuning of audio systems and processing algorithms.

d) **Real-time audio processing**: Real-time audio processing is the manipulation and transformation of audio signals with minimal latency, which is essential for applications such as live sound reinforcement, interactive audio systems, and virtual reality. Key principles of real-time audio processing include buffering, block processing, which processes audio data in chunks to balance computational efficiency and latency; and multithreading, which allows multiple processing tasks to run concurrently, increasing the overall processing efficiency. A comprehensive understanding of real-time audio processing principles is vital when developing an audio testing tool that aims to assess the real-time performance and responsiveness of audio systems and processing algorithms.

These theoretical underpinnings can be utilized to create an audio testing tool that can provide a more effective, and accessible solution for evaluating audio quality.

2.4 Human Perception in Subjective Testing

Schatz et al. [22] conducted a study to evaluate the impact of the duration of subjective testing on the listeners' exhaustiveness and on the quality of the assessments. Subjective testing results depend on the judgment and the perception of the listeners part of the test. Therefore, it is crucial to design the tests to not affect listeners' judgments from tiredness, boredom, or excessive tension. The authors initially evaluated the weariness of the listeners through a questionnaire and measured the eye blink rate (EBR) [23]. Later, ECG-based measurements were also employed to crossmatch the results from EBR testing. It was found that after 90 minutes, the listeners started to feel drowsiness, and EBR increased by 36% on average for one hypothesis and 17% on another. Besides, the heart rate also starts to drop, indicating that the listeners start to lose interest in the tests. Eventually that affects their judgment; hence the reliability of the tests conducted.

Liebetrau et al. [24] explored the challenges associated with evaluating emotions in multimedia system quality assessment, particularly regarding user affective states. They investigate the direct comparison of stimuli as a potentially more efficient and straightforward method for assessing emotions, as opposed to traditional self-report methods, which have inherent limitations such as demand characteristics, self-presentation biases, limited emotional awareness, and difficulties in articulating emotional perception. The experimental procedure consisted of two sessions in which subjects evaluated induced emotions in terms of arousal or valence [ii] using the direct comparison method. Tests were conducted in an audio lab designed in compliance with ITU-R BS.1116-1, [25], and DIN 15996 standards. The authors sought to replicate the results obtained from a Forced Choice Profiling (FCP) experiment by comparing items predominantly rated on a single emotion dimension. The assessment task was designed for easy comprehension by the subjects, enabling a more rapid evaluation process. The study's findings reveal that although the direct comparison method yielded reasonable results for the valence dimension, the outcomes for the arousal dimension were less interpretable. The authors ascribe this to the multidimensionality of the stimuli, which can influence ratings even when the assessment task is explicitly defined. Consequently, they suggest that the paired comparison method might be unsuitable for evaluating multidimensional items or problems commonly encountered in multimedia applications. The study highlights the need for further research to investigate this hypothesis and develop more reliable methodologies for assessing the emotional aspects of audio quality and user affective states in multimedia systems.

[ii]Valence: Describes the musical positiveness conveyed by a track.

In [26] the researchers investigated the preferences of recording professionals for high-frequency content in audio material. Fifteen trained subjects were asked to control a simple shelving equalizer, adjusting the high-frequency content of high-quality stereo program material over repeated trials. The results indicated a wide range of preferences for high-frequency content among the participants, with a total subject pool mean of +0.34 dB and a standard deviation of 2.8 dB. Interestingly, the researchers found no significant correlation between the participants' preferences and their years of experience or the musical genre they frequently worked on. The study also explored the effect of the randomized starting level of the equalizer on subjects' preferences. For two subjects, the initial high-frequency content had a statistically significant effect on their preferences, leading the researchers to exclude their data from the final analysis. After excluding these two subjects, the equalizer's starting position was found to be statistically insignificant at the p=.05 level [27]. This research highlights the diversity of preferences for high-frequency content among recording professionals and challenges some common beliefs about the influence of experience or muscial genre on these preferences.

Lund et al. [28] evaluated the importance of time and perception in subjective audio testing. The author focused on human sensation history and how consciousness works. Several factors, such as age, exhaustion, or even lack of attention, can drastically impact the perception bandwidth. However, experiences, previous learning, and expectations positively impact human perception. Nonetheless, the author iterated that subjective testing is not always dependent on the perception bandwidth [29], as it is still significantly hard to find out how human consciousness works.

Brook [30] presented an observation from a study conducted over 143 participants of different ages, gender, background, and experience. The goal of the author was to play two tracks, one unedited and another edited version of the original track, and see how users would rate the quality of the tracks. The original track was recorded in 1975, and then an edited version of the original track was prepared. After the survey, it was found that respondents below 24 years prefer the original track, while respondents from 25-44 years old prefer the processed and more polished edited track more. And respondents above 50 years have an equal preference for both tracks. However, musicians and audio professionals highly preferred the original track. It was also noticed that gender or educational background does not have a significant impact on audio preference. It is the listeners being accustomed to a certain quality of audio that has a higher impact on the listening tests.

2.5 Listening Tests and Methodologies

Toole [31] discusses subjective evaluations' crucial role and inherent variability in determining sound quality within the audio industry, suggesting potential avenues for improvement. He underscores the influential role that subjective evaluations play in shaping various sectors within the industry, including decisions about music production and consumer behavior. The author discusses the importance of listening tests in the audio industry and their considerable effect on shaping its trajectory. Despite their significance, the author argues that many tests need more controls and standardization to yield meaningful, reliable results. It explores the various sources of variability in subjective evaluations, such as technical factors like different audio hardware and listening environments, along with psychological elements that can affect listeners' opinions. The author also compares subjective and objective measurements, emphasizing the former's need for greater precision and standardization. While objective measurements are reliable due to their standardized procedures and calibrated instruments, subjective measurements depend on the human listeners as the "measuring devices", leading to potential biases and inconsistencies. Toole also highlights efforts by the International Electrotechnical Commission (IEC) to create standardized guidelines for conducting listening tests, ensuring uniformity in the acoustic and electrical requirements for these tests, and establishing specific procedures for conducting experiments and statistically processing data. This process seeks to enhance the consistency and overall value of the results obtained from listening tests.
Moreover, the concept of an ideal listening test is also introduced, which should be reproducible and accurately reflect the audible characteristics of the product or system under evaluation. However, the author acknowledges that accomplishing these goals entirely may pose some challenges. Furthermore, the choice of source material for listening tests is also discussed to a certain extent. Given the impact of recording manipulations on sound quality, selecting commercially available recordings for testing purposes presents its own challenges. The choice of music used for testing should be carefully evaluated for both its ability to reveal acoustic qualities and its entertainment value. The author suggests a few potential future research directions, which include exploring the relationship between measurements and perceptions, identifying the point of diminishing returns and just noticeable differences, and understanding the most influential subjective factors. It suggests that adopting standardized listening tests could elevate subjective testing to a scientific level, enabling more realistic comparisons and discussions across different tests carried out in various environments.

Clark [1] presented a detailed description of a system designed to perform double-blind audibility

tests in a practical, reliable manner. This system, referred to as the "A/B/X" method, addresses the weaknesses of traditional audio equipment listening tests, which often lack scientific rigor and don't adequately isolate variables. The A/B/X method is a robust system that incorporates techniques for improving discrimination ability, safeguards for upholding validity, and the use of purpose-built double-blind testing equipment. The author emphasizes the importance of rigorous scientific testing to generate meaningful data and stretches the necessity of controlling all potential influencing factors during the test, except for the specific variable under study. The author also highlighted the critical role of control experiments in providing a baseline for evaluating the results and in measuring random variations that could stem from the testing technique.
Moreover, several suggestions for optimizing the resolution of the A/B/X test are proposed. These include an immediate comparison of the signals being tested, emphasizing discerning differences rather than making qualitative judgments, ensuring sufficient time for the listener to decide, mandating a definitive decision without a "*no difference*" choice, offering the ability to compare similarity or dissimilarity, enabling listener control over the test, employing sensitizing tests or amplified differences to enhance sensitivity, allowing for perfect repetition of signals, and using statistical methods for analyzing the data from groups. Clack also emphasized maintaining validity in audibility testing. This involves managing factors such as maintaining equal volume levels, ensuring polarity consistency, and excluding unwanted noises and influences. They detail the criteria for frequency response matching and stress the necessity of careful control over these variables to ensure meaningful test results.
Furthermore, the author outlined the A/B/X comparator system's hardware components, which included the logic/display module, control module, and relay module. The logic/display module handles the random selection of components A and B. The control module enables the listener to make judgments and provide responses. The relay module assists in switching components for comparison purposes. The author's argument emphasizes the significance of frequency response and volume level matching in audibility testing and shares examples of tests conducted on various audio equipment like preamplifiers, amplifiers, pickup cartridges, and loudspeakers. The author also suggested potential areas for further research and underscored the educational benefits of the double-blind comparator in training individuals to identify and measure differences in audio signals.

Zielinski et al. [32] investigated the biases often encountered in modern audio quality listening tests. They highlights three primary types of biases that can distort test results: affective judgment bias, response mapping bias, and interface bias, while discussing several other biases. The authors also discussed potential bias due to the use of perceptually nonlinear graphic scales. They also explored the testing methods like ITU-R BS.1116 [18], MUSHRA (ITU-R BS.1534-1) [33], and ITU-T P.800 [19], and argued that despite considerable advancements in those testing methods. Biases persist and can affect the interpretation of results and the development of prediction algorithms. The authors emphasized biases can be introduced during the selection of audio stimuli or even from the listener's physiological and psychological characteristics. Besides, cognitive processes can also influence judgments. Furthermore, the design of the testing interface, whether traditional paper-based or computer-based, can also lead to errors. It is also argued that biases related to affective judgments can occur due to factors like the appearance of equipment, branding, expectation, and personal preference. The situational context can also introduce biases, as certain audio quality levels might be acceptable in one context but not another. When listeners translate their internal judgments into external responses, response mapping bias can occur. This bias can significantly distort listening test results and data interpretation. It's essential to consider the influence of these biases when designing and analyzing audio quality assessments. The authors focused on the importance of awareness of these biases and using mitigation strategies like blind listening tests, experimental design, and statistical analysis techniques. This can reduce the impact of biases and improve the reliability of results.

Jack et al. [34] presented a study exploring the impact of latency on the quality and interaction of a digital musical instrument. Musicians played a percussive digital instrument under various latency conditions, including no latency, 10 ms with jitter [iii], and 20 ms. The authors assessed the perceived instrument quality through improvisation tasks and measured timing accuracy with rhythmic tasks. The authors also emphasized the importance of minimal, consistent latency in digital musical instruments. They also explored the concept of 'control intimacy,' which emphasizes a close connection between a performer's action and the instrument's response. The authors also mentioned previous research on latency perception in a musical setting and how musicians react to timing differences. The instrument used, was a percussive instrument with ceramic tiles and piezo disks, controlled by the Bela platform [35] for low-latency processing. Eleven participants played the instrument under different latency conditions. The authors split the study into two parts: firstly, participants evaluated the instrument's quality based on various attributes while improvising. In the second part, the focus shifted to the objective performance measurement, where participants executed rhythmic exercises under different latency conditions, and their timing accuracy and dynamic performance were evalu-

[iii]Variability in the timing of data transmission due to network congestion, route changes, or other unforeseen complications.

ated. The results illustrated that higher latency or jitter negatively affected the subjective perception of instrument quality, while not significantly altering timing performance. The findings emphasized the significance of low, stable latency and its influence on subjective instrument quality assessment.

2.6 Listening Tests Parameters

Geddes et al. [36] introduced an innovative method to measure distortion in audio systems using the GedLee (Gm) metric, based on nonlinear system theory. Traditional metrics, Total Harmonic Distortion (THD) and Intermodulation Distortion (IMD), have been questioned for their correlation with perceived sound quality. In the experiment, thirty-seven normal hearing individuals rated twenty-one distinct stimuli for perceived distortion [iv]. The authors calculated THD, IMD, and Gm for each stimulus. The data showed weak negative correlations for THD and IMD with subjective ratings, suggesting their limited reliability in predicting sound quality. Conversely, Gm had a strong positive correlation with the ratings, demonstrating its potential as a reliable predictor. The Gm metric's correlation with subjective ratings improved significantly when focusing on stimuli with low to moderate distortion levels. The authors concluded that Gm outperforms THD and IMD in predicting perceived sound quality for nonlinear distortion. While acknowledging that specific stimuli could influence results, the authors argued that the general conclusions remain applicable.

Moore et al. [37] explored how different types of distortion in music and speech signals affect their perceived naturalness or quality. They focused on two types of distortion: linear distortion, which alters the tone quality of the signal, and nonlinear distortion, which adds unwanted noises to the signal. They sought to develop a model that could predict how these distortions affect perceived signal quality. The initial phase of the study looked at how various forms of nonlinear and linear distortion changed the perceived quality of music and speech signals. For linear distortion, they used filters that caused various spectral changes, such as tilts, ripples, and variations in the cutoff frequency. Listeners then rated the quality of the filtered signals. The findings showed consistent and reproducible effects on perceived quality based on the type of distortion.

The authors then developed a model to predict the perceived naturalness of these distortions. The model calculated the changes in the excitation pattern caused by the filtering. It also factored in how quickly the excitation pattern changed with frequency and how little effect low and high frequencies

[iv] Subjective experience or awareness of a listener regarding the alteration or modification of an audio signal from its original form.

had on perceived naturalness. This model was validated using new sets of spectral distortions, including those from actual transducers. The model, while successful, did not consider the impact of spectral distortion due to imperfect phase responses, as this was considered less perceptible compared to amplitude distortion. The authors concludes, the findings are most applicable to sound reproduction via headphones or earpieces, not sounds played via loudspeakers in typical listening environments.

Suzuki et al. [38] examined how changes in the phase response of loudspeakers impact the perceived sound quality. At the time of the research, there had been ongoing debate in the field of acoustics on whether phase response matters as much as amplitude response. The authors aimed to provide a more nuanced perspective on this debate. They suggested that both viewpoints might be correct, but the key factor is the extent to which phase response changes affect sound quality. This required detailed analysis for different kinds of phase responses and sound sources. The focus of the study was to assess whether the phase distortions seen in real-world loudspeakers (even those labeled as linear-phase) have a discernible impact on high-fidelity sound reproduction. The author used all-pass filters for their experiments because these have constant amplitude responses and can effectively simulate the phase responses of loudspeakers. This allowed them to isolate the phase response variable while keeping other factors constant, which would have been challenging using real-world loudspeakers due to their inherent variability. Their experiments revealed that phase change has a smaller effect on the quality of transient sound than one might expect from viewing waveform changes due to phase distortion. For musical signals, the phase distortions present in their experiment were too small to be detected by the listeners. The authors also pointed out that traditionally, the emphasis in audio quality analysis has been on amplitude response, partly because it's easy to measure and significantly influences sound quality. However, they suggested that the importance of phase response should not be overlooked, as it can affect the accurate reproduction of waveforms.

Gabrielsson et al. [39] studied how frequency response and sound level variations in sound-reproducing systems, such as headphones and loudspeakers, affect the perceived sound quality. Sound quality was analyzed on several perceptual dimensions including loudness, clarity, fullness, spaciousness, brightness, softness/gentleness, nearness, and fidelity. Three different audio types (female voice, jazz music, and pink noise) were played using four distinct frequency responses and two different sound levels. The researchers asked 14 participants with normal hearing to rate the sound quality they perceived through earphones on the aforementioned perceptual scales. Results indicated significant differences in perceived sound quality, attributable to variations in frequency response, sound level, or both. Also, the researchers observed a link between the program's spectrum and the frequency responses used, explaining the interactions observed between different reproductions and programs. The authors concluded that the influence of frequency response on perceptual dimensions is complicated and not fully understood. For instance, factors like brightness and sharpness increased with rising frequency response toward higher frequencies, while other characteristics like clarity, spaciousness, and nearness were favored by a broader frequency range. Similarly, the sound level was found to affect the perceived sound quality. Increasing sound level generally resulted in a perceived increase in fullness, spaciousness, and nearness as well as sharpness and brightness, with the reverse effects observed with a decrease in sound level. The authors also explored the interactions between sound level and the frequency response or the program's spectrum.

2.7 Tools for Listening Test

Hynninen [40] developed a system to enable easy access to subjective testing. Several test categories are supported by the system, including A/B/X testing. The system also supports multichannel audio output. Although the system addressed several aspects of audio testing, there are some major drawbacks of the system at the present day. The system was developed based around and limited to the SGI Hardware platform, which has become obsolete. Besides, The system utilized Alesis Digital Audio Tape (ADAT) interface, which also has become obsolete. Moreover, the system does not support real-time filter coefficient changes, among other limitations. However, the system was designed to be easily customizable and modular to support the creation of new customized tests not covered by existing standardized tests.

Ciba et al. [41] introduced a listening test tool called WhisPER, built for windows platform using MatLab. The authors illustrated the limitation of most of the commercial and non-commercial tools in terms of available test methods, playback options, and supported channels. The authors also iterated the limited availability of commercial listening software and their lack of customization options. Hence, the authors aimed to produce a listening test tool with customizable tests and a graphical user interface. The tool provided three types of tests, and they are a selection of adaptive psychophysical methods, the Repertory Grid Technique (RGT) and the Semantic Differential. The tool is also made available for free for the community to explore and continue the development of the tool, although, the MatLab licenses are not free.

The LIStening Test ENvironment (LisTEn) [42] offers a versatile and platform-independent system for the subjective evaluation of speech and audio signal processing algorithms, supporting various test types. However, despite its many benefits, the tool presents certain limitations, particularly in the test setup process and audio file preparation. The process of preparing audio files can be tedious, as it requires processing test files using candidate codecs and placing them in separate folders. The audio player's flexibility in handling different sampling rates and wordlengths is advantageous, but it does not negate the time-consuming nature of the preparation process. Streamlining the audio file preparation workflow could further enhance the user experience. Setting up the testing environment is another area that could be improved. While LisTEn outlines the requirements for an optimal testing environment, including low background noise, minimal visual distractions, and a high-quality audio reproduction system, it offers little assistance in facilitating the actual setup. Providing guidelines, recommendations, or tools to support users in creating the ideal testing environment would be a valuable addition to the system. Overall, while LisTEn has proven to be a valuable tool for researchers in audio engineering and related fields, addressing these limitations could make the test setup process more efficient and user-friendly, ultimately saving time and effort for those conducting subjective listening tests.

Johnston [43] presented an open-sourced framework for spatial audio perceptual testing to be able to take advantage of fast-growing virtual reality (VR) technology. The authors created a user-friendly interface for the users to create listening tests by simply dragging and dropping elements and sharing the test parameters and results between different users. The framework encompasses standard test paradigms, such as MUSHRA, 3GPP TS 26.259, and audio localization with different pointing methods with the capability of more customization by the users. For the user interface, the Unity 3D game engine is utilized. The test configuration of the environment can be exported as a JavaScript Object Notation (JSON) file with other users, and the test results also can be exported as a comma-separated values (CSV) file for further analysis. The audio rendering engine, which is a stand-alone application used on the SALTE framework, is explained by Rudzki [44]. The tool enables listening tests using Ambisonics stimuli and especially focuses on the development of spatial audio technology.

Murgela et al. [45] investigated the utilization of hearing loss simulations and extended the authors' previous prototype design to a working solution with added features such as real-time audio processing, two-channel audio support, and more customization. According to the authors, the tool performs well in terms of accuracy at a certain level, but it was compromised significantly to ensure good real-time performance. The digital signal processing algorithm used in the implementation needed to be more efficient to ensure good results and performance.

Gorzynski et al. [46] came up with a sophisticated listening test tool widely available for public users. The tool explored different standard audio testing methods for sound grading. It also supports multichannel audio processing. Besides, the authors built a VR-specific version of the tool to provide a better experience than the typical desktop testing experience for the user. However, the authors maintained two separate builds to achieve that, although it was mentioned that they aimed to unify the builds. Moreover, the tool focused on complex systems like spatial audio and virtual reality.

2.8 Hardware and Test Equipment

McPherson et al. [47] presented a new environment for hard real-time embedded audio processing application that demands ultra-low latency. It is based on BeagleRT, which at the time of writing, is known as the Bela. The environment is based on the cheap BeagleBone Black single-board computer [48]. The authors discussed the latency and dropout risks of audio processing in general-purpose computers and emphasized the dedicated environment for audio processing to provide high performance while keeping the latency below 1 ms for a round trip. The environment is built on Linux with the Xenomai [49] real-time kernel extensions to achieve such low latency by giving the Xenomai tasks higher priority than the kernel itself.

Langer [50] proposed an embedded Linux audio system that supports multichannel audio as well as addressed the real-time aspect of the audio processing. The author emphasized the need for more applications in this particular area since single-board computers like Raspberry Pi are cheap and popular nowadays. The authors chose Bela Platform to achieve ultra-low latency in the audio processing. The platform is suitable for hard real-time applications which require extremely low latency and good performance.

Coler [51] presented a real-time additive sound synthesis application based on the JACK [5] sound server. The software architecture of this application is designed to enable the use of additive synthesis or sinusoidal modeling in sound field synthesis systems or other audio reproduction setups. This allows for greater flexibility and versatility in the way the synthesized audio can be reproduced and experienced. An individual JACK client was developed to enable the connection of all individual synthesizer output channels to a JACK-capable renderer, such as the SoundScape Renderer (SSR), or [52]. This allows for the integration of the synthesizer with a range of different audio rendering technologies, giving users more options for how they can experience the synthesized audio. While the focus of this paper is primarily on additive sound synthesis and its real-time implementation, it provides valuable insights into the use of the Jack API and Open Source Control (OSC) interface for control in the context of audio processing applications.

Kuhr and Carôt [53] presented how JACK [5] can be utilized in audio video bridging (AVB) processing servers while illustrating the evolution of a media clocking scheme. The author aimed to minimize the latencies in real-time audio-video streaming. It was also argued that JACK is a suitable audio server solution when audio sample data has to be shared between applications in real time and with minimum latency. It can significantly minimize the latency as many Digital Signal Processing (DSP) applications and algorithms are available for JACK. Besides, the JACK server runs independently and communicates with client applications through an interface, which makes the client application architecture modular and easily adjustable.

Taymans [54] introduced PipeWire, which is a sound server system that aimed to combine the functionality of PulseAudio [55] and JACK [5]. PulseAudio in the Linux system is mainly used for consumer audio and video streaming, while JACK is utilized for Pro audio. The author initially aimed PipeWire to serve as a daemon that would separate access to the camera and the application, similar to existing audio daemons. Its purpose was to decouple these two elements and allow them to function independently. This approach is not unlike the way existing audio daemons work. But, later, with some design modifications, a necessary rewrite, and the support of the Linux Audio Developers (LAD) community, it adopted audio support as well. Eventually, PipeWire was able to provide the best features from both PulseAudio and JACK and added compatibility for PulseAudio and JACK applications by providing an API layer. PipeWire is a technology that aims to improve the way audio and video are handled on computers. PipeWire aimed to provice a solid foundation for new multimedia applications by unifying the audio stack [56]. This technology is designed to be future-proof, supporting the development of exciting new multimedia applications. And, with Fedora 34, it was planned to replace JACK and PulseAudio with PipeWire, and the authors' plan involved deploying PipeWire in other Linux distros too. From April 2021 onwards PipeWire has been the default sound server on Fedora.

Viganti et al. [57] presented a comparison between two different but actively used approaches of real-time audio processing in Linux audio. One is based on PREEMPT_RT kernel patch and the Advanced Linux Sound Architecture (ALSA) and JACK framework, and the other is the Xenomai patch and the Elk Audio OS. The authors mainly focused on comparing the most critical aspects, such as performance, real-time processing latency, scheduling latency, and digital signal processing load of each system. It was found on the comparison that the ALSA/JACK framework is suitable for soft real-time and not recommended for the safety-critical system due to comparatively low performance and higher latency than the Xenomai system. The Xemonai system provides uncompromising performance, which makes it the best candidate for a hard real-time system, and that is one of the main reasons high performing and low latency system like Bela is backed by Xenomai system.

2.9 Audio Processing Techniques

Henk L Muller [58] introduced an innovative approach to digital audio system design, which utilizes the predictability of hardware to minimize buffering. Muller introduces the application of multi-threading and multi-core design in real-time systems, notably in audio systems, and showcases their benefits through examples such as Asynchronous USB-Audio 2 and AVB over Ethernet. Muller's study emphasizes multi-threading and multi-core design as effective strategies for digital audio systems. It explores the advantages of using the XMOS XCore processor for instruction-level thread scheduling over traditional context-switching platforms like Linux. By partitioning the system into different threads, including network protocol stack, clock recovery, and DSP tasks, the efficiency and performance of the system are enhanced. The author also examines the concept of digital audio, highlighting its advantages over analog audio in terms of precision, reliability, transmission, and storage. Muller pays special attention to buffering in digital audio systems and the challenges in determining appropriate buffer sizes. Moreover, Muller underscores the importance of clocking in digital audio systems, suggesting a dedicated thread for clock measurement and frequency consistency. The author advocates for multi-threaded and multi-core design approaches for digital audio systems, arguing that they offer a promising and efficient route for real-time system design. However, he suggests that further research is needed to evaluate these methods' scalability and efficiency in larger scale applications.

Wang [59] presented a low latency multichannel audio processing evaluation platform with a substantial potential.Its standout features, such as precise latency control, comprehensive evaluation capabilities, and the ability to estimate synchronization and delay across channels, are foundational for optimizing live digital audio systems. The programmability of its Field Programmable Gate Array (FPGA) and DSP components allows for the implementation of diverse audio processing algorithms, a feature that significantly enhances its utility. Additionally, it presents a robust platform for research, opening avenues for investigation in areas like audio signal capturing, intelligent mixing, and latency measurement such as analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) digital filter architectures, buffer subsystem design, interrupt and scheduling, and high-level audio processing algorithms. Despite these promising attributes, there are some limitations that need consideration. The hardware prototype, being a first-version, might have potential limitations in performance and scalability, particularly in large-scale applications or real-world scenarios. Its complexity, combining multiple components, could present challenges in development and implementation. The paper does not clarify the cost or availability of the platform, which could impact its accessibility for users with limited resources. Additionally, the absence of comparisons with existing low latency multichannel audio processing platforms leaves an uncertainty in assessing the platform's distinctiveness or superiority.

2.10 Statistical Analysis

Leventhal [60] underscores the critical role of statistical significance testing in ascertaining the reliability of listening tests. It is commonly observed that these tests utilize a 0.05 significance level, mitigating the risk of type 1 errors [v] to 5%. However, they frequently neglect to report the risk of type 2 errors [vi] . This lapse can culminate in a high type 2 error risk and low power when the sample size (represented by the number of trials or listeners) is inadequate. The author introduces a novel fairness coefficient (FC), which serves as a comparative tool for the probabilities of type 1 and type 2 errors. This coefficient provides an objective measure of the fairness of the significance test employed in a listening study. Furthermore, this coefficient can be adjusted to assess the fairness of a conclusion of listening tests. The author also warns that combining biased studies might result in an overall more unfair conclusions.

The author also argues for equalizing the probabilities of type 1 and type 2 errors, primarily by augmenting the number of trials or listeners. If such an increase is not feasible, it was proposed to increase the level of significance to offset bias and balance the escalation in type 1 errors with a re-

[v]When listeners conclude the differences are audible when they are not actually audible.

[vi]When listeners find differences inaudible when they are actually audible.

duction in type 2 errors. Leventhal also acknowledged the challenge inherent in selecting a p-value to balance errors and suggest several remedies for this quandary. An arbitrary p-value, such as 0.7 or 0.8, could be chosen for the equal-error analysis, or the type 2 error could be reported for a range of p-values. By reporting statistical power [vii] or type 2 error for a variety of p-values in conjunction with type 1 error, the author believed that a balanced discussion and response to the work can be facilitated. Type 1 and Type 2 errors are discussed more in section 5.1.10.2.

Srednicki [61] used Bayesian statistical methods to analyze A-B listening tests, which are used to determine if there's an audible difference between two audio components. These tests are often debated due to the difficulties in designing a fair test and interpreting the results. The author's approach involved calculating the probability that a listener can hear a difference between the two components during the test. This probability is represented as a fraction (denoted as 'h') of the total number of trials. The goal was to determine the upper and lower limits of this fraction with a 95% confidence level. In the A-B-X testing arrangement, for some trials, the listener can hear a difference and correctly identify X; in other trials, they guess the identity of X. The probability of correctly identifying X is represented as 'p', which is calculated from the fraction 'h' and the fraction of guesses. The author provided a mathematical formulation for the probability of getting a certain number of correct answers, given the number of trials and the probability 'p'. A result is deemed significant if the cumulative probability of the result and all better results is less than 0.05 when 'h' is zero, suggesting that 'h' is greater than zero. The author argued that it's more informative to determine how much greater than zero 'h' could be. This is where Bayesian analysis comes into play. Based on Bayes' theorem, the author formulated a new probability distribution for 'p', called the posterior distribution, given the values of the number of trials and correct answers. The author then defines minimum and maximum values for 'p' at the 95% confidence level and demonstrates how these can be used to find the minimum and maximum values for 'h'. The results indicated that it is difficult to confidently establish the absence of any audible difference unless one either runs several hundred trials or chooses a prior distribution which is strongly biased. However, the author argued that, it is relatively easier to establish the absence of a reliably audible difference. The author also discussed the implications of changing the prior distribution for 'p', demonstrating that this could significantly impact the maximum value of 'h'. Srednicki's new statistical approach to interpreting the results of A-B listening tests provided insights into how these tests could be improved to yield more reliable and meaningful results.

[vii] Statistical power is the probability of correctly rejecting the null hypothesis in a future study when the alternative hypotesis is in fact true. The null hypothesis is the claim that there is no audible difference between the reference and the modified audio stream.

Boley [3] examines the effectiveness of signal detection theory in interpreting the results of ABX listening tests, a standard psychoacoustic test used to determine audible differences between two audio signals. Traditionally, the interpretation of ABX test results relies on percentage correct scores. However, the authors argue that the use of signal detection theory can help prevent potential false conclusions and provide more accurate results while discussing experimental design considerations and statistical analyses, providing insights into the appropriate ways to interpret and report results from ABX listening tests. They also suggest a set of reporting guidelines for all listening tests, which include a well-defined hypothesis, detailed method section, information on subject selection, trial numbers, and confidence levels. They also recommend avoiding certain pitfalls, such as combining results from different subjects or stimuli that are not quantitatively similar. The study concludes that while the binomial distribution method is appropriate for ABX test analysis, signal detection theory offers advantages in terms of accuracy and the need for fewer total trials. The authors emphasize that good experimental design is crucial and that statistical analysis cannot compensate for poor design. Overall, the authors provide valuable insights into enhancing the interpretation of ABX listening test results and suggests that the adoption of signal detection theory could lead to more accurate conclusions in audio engineering research.

3 Research Methodology

The purpose of this chapter is to outline the research methods used in the development of this real-time embedded application for subjective testing. This consists of the approach, design, and implementation of the system, as well as how the research questions were answered.

The project utilized the Design Science Research (DSR) methodology [62], which is commonly used in computer science and information systems research. The method involves an iterative process where each step informs and refines the next. Upon considering different research methodologies, the ones that stand out as potentially applicable to this project are Design Science Research, Experimental Research [63], and Case Study Research [64]. Although Experimental Research offers some congruence with this project due to its systematic approach to conducting experiments to validate hypotheses, it falls short of fully aligning with the project's core aim. This project's focal point extends beyond hypothesis testing, leaning more towards tool development and understanding its potential impact. Consequently, Experimental Research's scope seems limited for this undertaking. Case Study Research, which emphasizes the detailed exploration of a single instance or event, does resonate with some facets of this project. For example, applying the developed tool in a particular audio testing scenario could be viewed as a case study. However, the overall objective transcends the scrutiny of a solitary event and aims to craft a tool with broader applications. As such, the applicability of Case Study Research is circumscribed.

Weighing these alternatives brings DSR to the forefront as the most suitable methodology. With its primary focus on creating and evaluating IT artifacts intended to resolve identified issues, DSR perfectly matches the intent of this project. Furthermore, DSR's inherent iterative nature blends well with the project's development process, making it even more relevant. The systematic yet flexible approach afforded by DSR permits ongoing refinement and alignment with the project's evolving requirements. Thus, despite the potential merits of Experimental and Case Study Research, DSR provides this project's most comprehensive and fitting methodology. This led to the research methodology used for the development of the software. The approach, design, and implementation of the system were tailored to respond to the research questions comprehensively. An iterative process was followed, where each step informed and refined the subsequent ones.

The amount of audio signal processing overhead in real-time systems depends on several factors, such as the arithmetic and structural complexity of the processing algorithms, the sample rate, and the performance of the hardware and software being used. Real-time audio signal processing can be computationally intensive and require significant processing power, especially for complex algorithms or high sample rates. To minimize the overhead of audio signal processing in real-time systems, it is essential to carefully optimize the algorithms and use efficient implementations that enable the program to run on both high-performance and cheap hardware. Additionally, it may be necessary to carefully manage the amount of processing performed in real-time to ensure that the system can keep up with the incoming audio data. This can involve using techniques such as buffering, scheduling, and parallel processing to distribute the workload onto multiple cores and avoid overloading the system.

Although the scope of this project's implementation is confined to JND and linear distortion testing, which is conducted through A/B/X testing methods, initially, the study involved an extensive literature review in obtaining a deeper understanding of the relevant subject matter not limited to the scope particularly and explored areas such as human perception in subjective testing which has a significant impact in the tool and test design. It is also worth mentioning that not all of the literature studies directly contributed to the full scope of the project, but their overall contribution to the implementation can't be denied. The study dived into the previous work in the domain of audio testing methods, tools, hardware, and test equipment, as well as the digital signal processing to a certain extent that aided in identifying the theoretical base for the configurations for the software to be developed. Subsequent to this, the design and system architecture were conceptualized.

This takes us to the actual coding and implementation of the tool, focusing on creating a tool to meet the project's objectives. In this part, the first step was to design the tests and user interface, which was carried out by taking the various aspects of the listening tests into account observed from the literature study, such as listeners' experience in listening tests, sound preference, and headphone and speaker quality utilized on the test. The user interface design also considers the statistical analysis of the testing sessions. The next step was to implement the algorithms as well as making the tool functional. In this part, a suitable amount of experiments was carried out with different buffering schemes to find the one that provides the best balance of latency and efficiency. Besides, algorithms are also designed efficiently to not break the processors' instruction pipeline, which, in general, can potentially cause dropouts and latency. The applications' hard and soft real-time aspects are also considered simultaneously since the goal is to develop an efficient and responsive tool for different hardware. This is followed by a data collection and testing phase of the developed tool that facilitated the capture of relevant parameters for subsequent analysis and improvements of the tool. Later, the tool is benchmarked using JACK APIs to assess the performance under different conditions. Further, profiling tools are utilized to conduct thorough testing and analysis of the results from different hardwares and operating systems, enabling effective evaluation of the software's efficiency and performance.

Lastly, the results obtained from the testing and evaluation activities were analyzed to meet the project objectives. The tool's performance, effectiveness, and potential impacts were critically examined to provide insights into its capabilities and potential use in the audio industry.

4 System Design and Architecture

In this chapter, the conceptual design and architecture of the proposed embedded tool are presented. The tool aims to run efficiently on different hardware and evaluate the perception threshold for various stimuli, assisting researchers and people from interested fields in conducting audio testing to understand human sensory perception. The system design covers the main components, their interactions, and the rationale behind design decisions.

4.1 Requirements Analysis

The key functional and non-functional requirements of the tool are as follows:
Functional Requirements:

- FR1: Process audio in real-time utilizing multi-threading.
- FR2: Create and alter filter coefficients in real-time.
- FR3: Record participant responses from testing sessions.
- FR4: Generate reports with statistical analysis.

Non-Functional Requirements:

- NFR1: User-friendly interface for training and conducting tests.
- NFR2: Scalable for different types of filters and audio parameters.
- NFR3: Compatible with various hardware and computers.
- NFR4: Robust, resource-efficient, and reliable performance.

4.2 Technologies and Tools

The following tools, frameworks, and libraries were used in the project development:

- **QT:** Qt [65] is a widely used cross-platform framework for developing applications and graphical user interfaces (GUIs). It supports various platforms and operating systems, making it an ideal choice for our audio testing tool. With its extensive set of libraries and tools, Qt enables the creation of high-performance, responsive, and user-friendly interfaces. The decision to use Qt was based on its flexibility, ease of use, and compatibility with a wide range of hardware and software configurations.

- **JACK 2:** The JACK Audio Connection Kit (JACK) [5] is a professional sound server daemon that provides low latency, high-quality audio processing, and routing capabilities. It is written in C++, but it is based on C-style API to be compatible with JACK 1. It allows applications to connect and share audio streams, making it suitable for our real-time audio processing requirements. Using JACK, we can ensure that our testing tool has a reliable, efficient, and flexible audio processing infrastructure capable of handling complex audio routing and manipulation tasks with minimal latency.

- **C++:** C++ [66] is a versatile, high-performance programming language widely used in the development of system software, application software, and embedded systems. Its extensive set of features, including object-oriented programming and the Standard Template Library (STL), make it well-suited for implementing the testing tool's core functionality. The choice of C++ as the primary programming language for this project was influenced by its performance capabilities, compatibility with various platforms, and the availability of numerous libraries and frameworks, such as Qt and JACK.

- **JACK-Client C++ API (JACKCPP):** JACKCPP [67] is a C++ API for interacting with the JACK audio server. It provides an object-oriented interface to JACK, simplifying the process of connecting, controlling, and processing audio streams. The use of JACKCPP in the testing tool allowed for seamless integration with the JACK audio server, enabling efficient real-time audio processing and routing capabilities. By leveraging the JACKCPP library, we were able to streamline the development process and ensure reliable, high-performance audio processing within our tool.

- **AudioFile:** A simple library [68] to read audio files from local storage and load the audio samples into the memory.

4.3 High-Level Design

Fig. 1 illustrates the proposed tool's high-level architecture. The tool comprises four primary components: the User Interface (UI), the JACK External Client, and the Data Analysis and Reporting are the core components, and JACK2 is the sound server API that connects the tool with the system and runs on a separate process. One of the main design goals is to provide an abstraction layer between each module and perform the communication through simple API interfacing. The UI offers an interactive platform for users, enabling them to easily access the tool's functionalities, such as running training sessions and test sessions. Meanwhile, the JACK External Client module manages the tool's core operations, such as running the audio processing engine in a separate thread and processing audio in real-time with the user-configured filter coefficients. Lastly, the Data Analysis and Reporting module processes and summarizes the results obtained from the user response during the testing sessions, providing insights for further analysis and improvement.

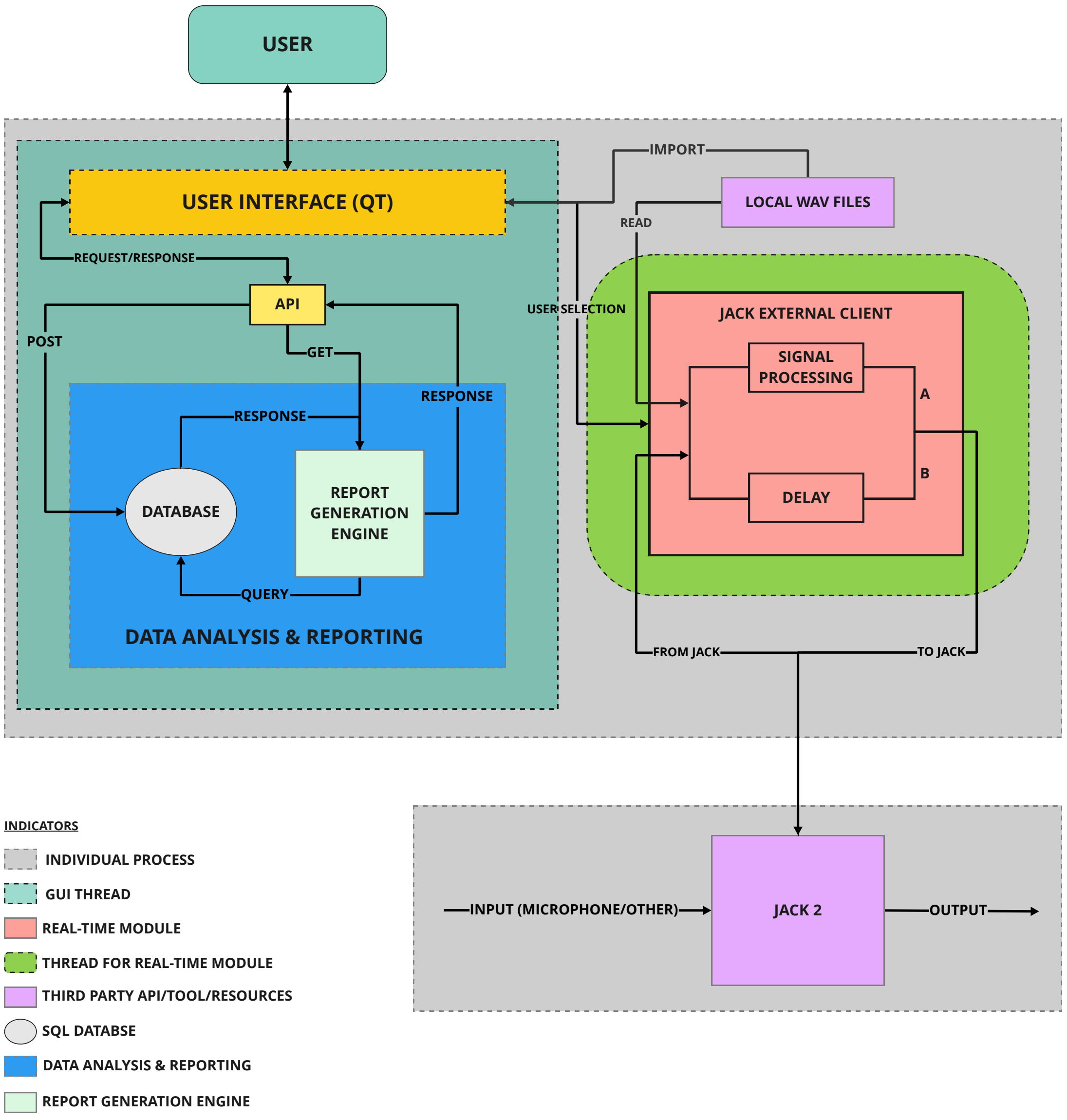


Figure 1: High Level Architecture of the Tool

4.4 Detailed Design

This section dives into the internal architecture and explains the tool's functionality.

4.4.1 User Interface (UI)

The User Interface (UI) module is crucial in enhancing user experience and facilitating seamless interaction with the system. Fig. 2 shows the internal architecture and flow of the UI module.
The tool aims to allow listeners to conduct training sessions and testing sessions and ultimately review the outcomes of the completed testing sessions in the form of various types of reports or statistics. To achieve this, the UI module is divided into three distinct sections, each represented by a separate tab: Training, Testing, and Statistics.

4.4.1.1 Training UI

This section of the UI module is dedicated to setting up and conducting training sessions for listeners. It provides listeners with the necessary tools and options to configure filter type, parameters, and the source listeners would prefer to train on, such as local audio files or any other input method compatible with JACK, and define other settings relevant to the training process. This window also shows the CPU load created by the software in real-time. The layout and design of this window aim to ensure that listeners can quickly and easily understand the process and navigate the various training options.

4.4.1.2 Testing UI

This tab focuses on facilitating the execution of testing sessions. Listeners can access and configure the testing duration and the number of guesses, select the appropriate audio samples and define other relevant settings to customize the testing process. This tab also provides the necessary functionalities to play the reference and modified tracks, choose their guesses, and monitor their progress in real-time.

4.4.1.3 Statistics UI

The Statistics section serves as a hub for reviewing and analyzing the results of completed testing sessions. Listeners can access a variety of reports and statistics that provide detailed insights into the performance and outcomes of the tests. This window offers visualization options, such as test session summary, graphs with session results to help listeners understand and interpret the data easily. The window also includes opportunities to export the test sessions data, allowing listeners to collaborate with others or use the results for further analysis.

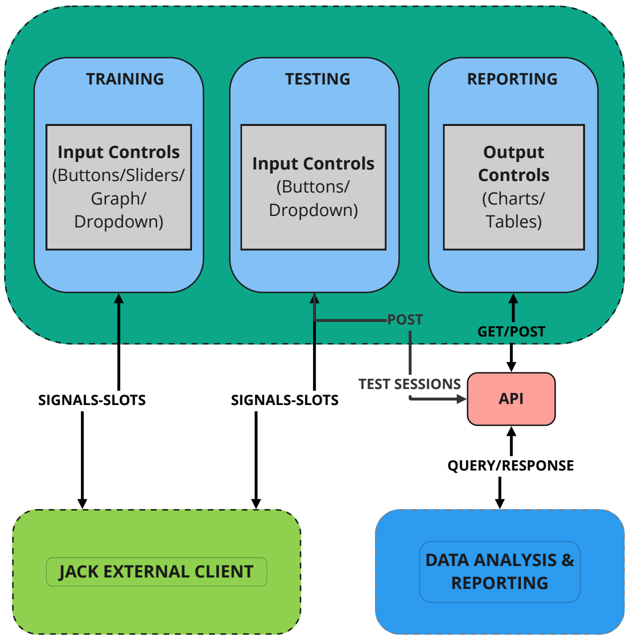

TRAINING
Input Controls (Buttons/Sliders/ Graph/ Dropdown)
TESTING
Input Controls (Buttons/ Dropdown)
REPORTING
Output Controls (Charts/ Tables)
POST
GET/POST
SIGNALS-SLOTS
SIGNALS-SLOTS
TEST SESSIONS
API
QUERY/RESPONSE
JACK EXTERNAL CLIENT
DATA ANALYSIS & REPORTING

Figure 2: User Interface

4.4.2 Interfacing

The interfacing of the UI module focuses on providing a seamless user experience while facilitating communication with other essential modules, such as the JACK external client and the Reporting and Analysis module. The following sections outline the design details of the interface for the Training, Testing, and Statistics UI, highlighting their interactions with the relevant external modules.

4.4.2.1 Training UI Interfacing

The Training window is designed to allow users to configure various parameters and settings for the training sessions. The layout consists of input controls such as a slider for volume control, amplitude response alterations which leads to generation of the filter coefficients for the selected filter type. The filter type can also be selected from a drop-down menu. In addition, there is a drop-down menu to select the audio source, such as local files or JACK-compatible input, as well as buttons for starting or stopping the training session.
During the training session, the user interface communicates with the associated filter classes to obtain the filter coefficients necessary for the training process. The filter coefficients are sent to the real-time thread and audio engine. These coefficients are then applied to the audio input buffer to process the sound stimuli. This communication is facilitated through an abstraction layer supported by the QT signal and slots mechanism [69].

4.4.2.2 Testing UI Interfacing

The Testing window's interface design focuses on enabling users to configure and manage the testing sessions easily. It features a drop-down menu to select the audio file to be used to run the test on as well as an option to choose the number of trials, and three buttons, A, B, and X, to be able to play the unlabeled tracks. Listeners can play the modified and unmodified tracks as long as they want before making a choice. And finally, two more buttons to provide the listeners' response.
The filter coefficient generation takes place in the UI thread in soft real time. The communication between the testing window, the real-time thread, and the audio engine are similar to the training window. Additionally, during testing sessions, listeners' responses are logged into the database through an API layer which provides abstraction between the UI and the database.

4.4.2.3 Statistics UI Interfacing

The Statistics window interfacing is centered on providing listeners with an overview of the test results and related statistics. To generate the reports, the UI module communicates with the database through an API layer. The interface sends a request to the module, which then processes the test data and produces the relevant reports and statistics. Once the reports are generated, they are displayed within the Statistics window, allowing listeners to review and analyze the results.

4.5 JACK External Client (JEC)

JEC plays a pivotal role in the testing tool, as it handles the tool's core functionality processing audio stimuli based on the selected filter type and filter coefficients generated from the user-configured filter parameters in real time. This module runs on a separate worker thread that ensures that the main UI thread is not hindered, allowing for smooth and uninterrupted user interactions. Fig. 3 shows the internal components of the JEC.
The filter coefficient generation engine receives the user-selected filter type and filter parameters from the controls and executes the associated algorithm to compute the filter coefficients in real time. The filter-coefficient generations take place in the UI thread, with an abstraction layer to make the implementation robust and scalable. Upon completion, the calculated coefficients are used to update the frequency response graph with normalized values. Simultaneously, the generated filter coefficients are passed to the audio processing engine, which runs on a separate worker thread that manipulates the audio signals based on these coefficients. The processed audio is then played back

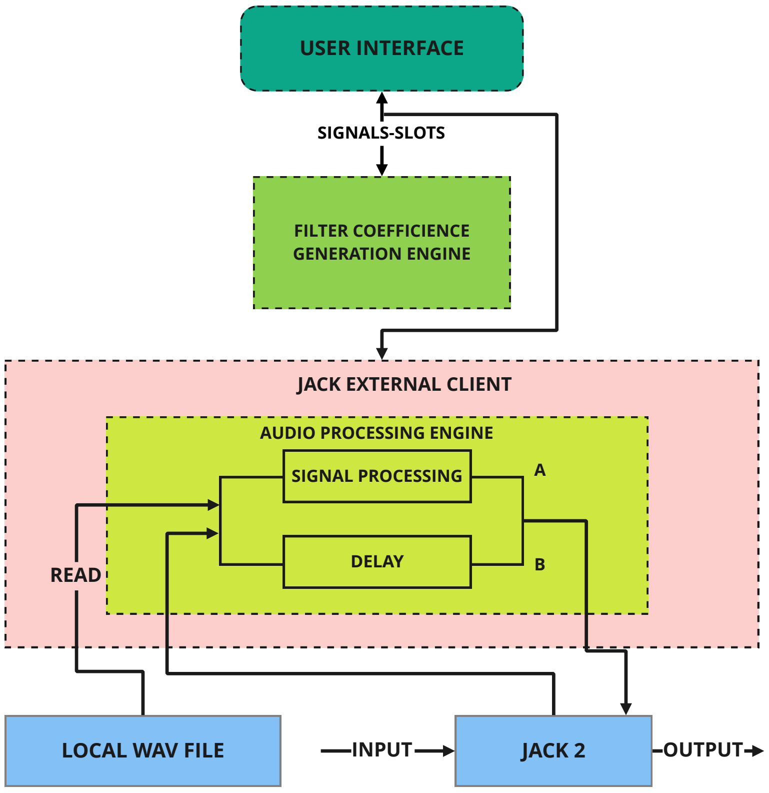


Figure 3: JACK External Client

to the user, reflecting the effects of the chosen filter settings.

4.5.1 Interfacing

JEC interfaces with both UI and the JACK audio server. The interfacing with the UI is described in sec. 4.4.2 UI Interfacing. On the other hand, the interfacing with the JACK audio server is performed utilizing the JACK API, which contains a comprehensive set of functions for connecting, processing, and managing audio streams.

Key components of this communication include creating a JACK client, registering callback functions, creating and connecting input and output ports, activating the client, processing audio in real-time, and cleaning up resources when the connection is no longer needed.

4.6 Data Analysis and Reporting

The module plays a crucial role in the proposed testing tool by processing the test results and presenting them in a meaningful way for the testers. This module contains a database to store user session data and is responsible for gathering data from the testing sessions, analyzing the data, and generating reports or statistics that provide insights into the audio tests performed.

Key components of this module include:

- **Data collection:** This component gathers relevant data from the testing sessions, such as user responses, time taken by user to respond, information of the audio file used foe testing, and number of trials configured for each test session. It organizes the data systematically for further analysis.
- **Data analysis:** This component processes the collected data to derive insights and statistics. It is to perform various calculations, such as identifying trends, calculating average values, or comparing different test parameters, to provide a better understanding of the test results.
- **Report generation:** This component generates reports based on the analyzed data. The API layer exposes a set of methods for this module to interact with the database and return processed data to render on the UI with a visual representation.
- **Export and sharing capabilities:** This component allows users to export the generated reports in CSV (Comma-Separate Value) format, facilitating easy sharing and collaboration among audio professionals. Additionally, this feature enables users to store the reports for future reference or comparison with other test results.

4.6.1 Interfacing

The module maintains an interface with only the UI module as described in 4.4.2. Fig. 4 illustrates the interfacing of the module with UI. The API layer is responsible for transporting the user commands to the database and data analysis engine and response from the reporting engine back to the UI.

4.6.2 Database Schema Design

Considering the tool's functional requirements and final aim, the data model is designed and illustrated in Fig. 5. The database schema for the tool consists of three tables: audio_samples, test_sessions, and trials. Each table is designed to store specific information related to the tool's operation.

- **audio_samples**: This table stores information about the audio samples the user imports for training and testing. It includes a unique sample_id (primary key), file_name, path, and other_properties. The sample_id uniquely identifies each audio sample. The file_name stores the name of the audio file. The path field contains the file path to the audio sample in local storage, and

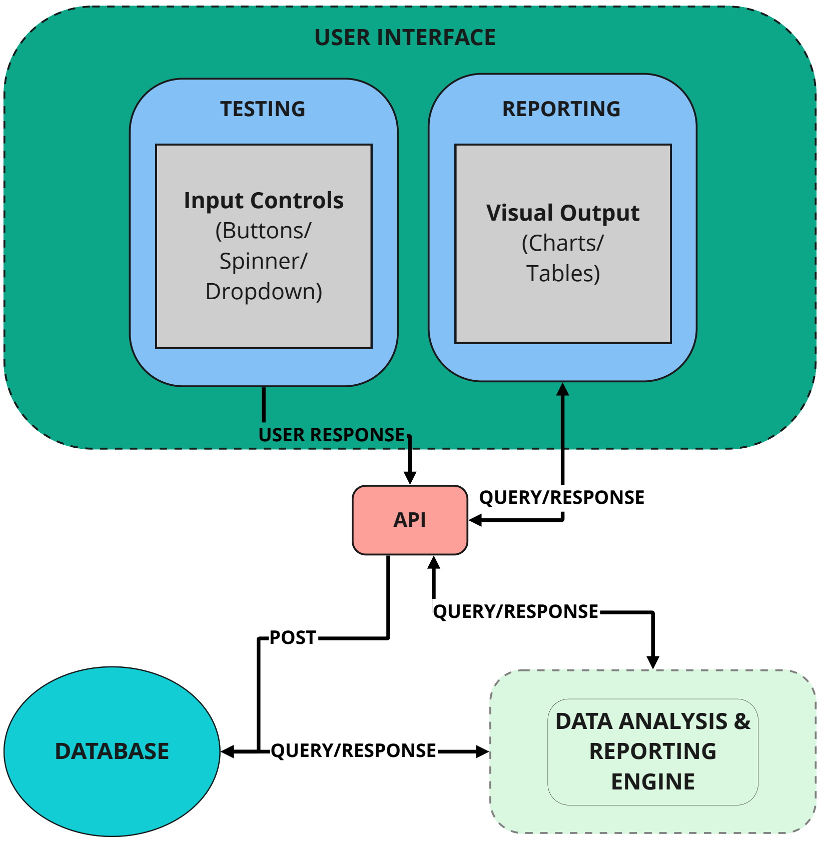
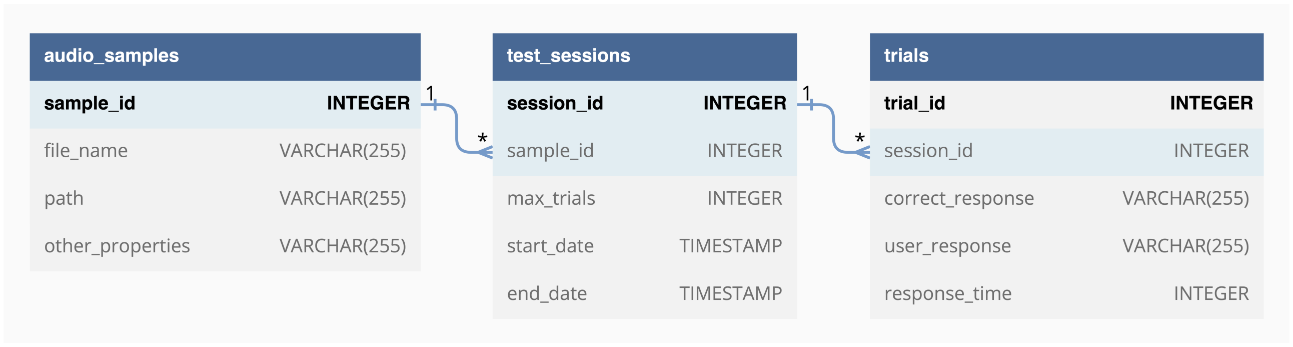


Figure 4: Data Analysis and Reporting Module



audio_samples	
sample_id	**INTEGER**
file_name	VARCHAR(255)
path	VARCHAR(255)
other_properties	VARCHAR(255)

test_sessions	
session_id	**INTEGER**
sample_id	INTEGER
max_trials	INTEGER
start_date	TIMESTAMP
end_date	TIMESTAMP

trials	
trial_id	**INTEGER**
session_id	INTEGER
correct_response	VARCHAR(255)
user_response	VARCHAR(255)
response_time	INTEGER

Figure 5: Data Model for Data Analysis and Reporting

other_properties is an optional field to store any additional information about the audio sample.

- **test_sessions**: This table contains information about each testing session conducted with the audio samples. It includes a unique session_id (primary key), sample_id (foreign key) referencing the associated audio sample, max_trials, start_date, and end_date. The session_id uniquely identifies each testing session, while the sample_id links the testing session to a specific audio sample. The max_trials field stores the maximum number of trials allowed for the session, and the start_time and end_time fields record the timestamps for the beginning and end of the testing session, respectively.

- **trials**: This table stores the individual trial data collected during the testing sessions. It includes a unique trial_id (primary key), session_id (foreign key) referencing the associated testing session, correct_response, user_response, and response_time. The trial_id uniquely identifies each trial, while the session_id links the trial to a specific testing session. The correct_response field stores the expected correct response for the trial, and the user_response field stores the user's actual response. The response_time field records the time taken by the user to make their response during the trial.

4.7 Design Rationale and Trade-offs

4.7.1 System Architecture

The modular architecture was chosen for the tool to ensure better organization, maintainability, and extensibility of the code, prevent performance bottlenecks, and ensure smooth operation throughout the system. This design choice enables easier future modifications and improvements to individual modules without affecting the overall system. Moreover, the tool is designed to run on multiple threads for simultaneous operations and take advantage of the modern CPUs. In order to make the source code modular and maintainable Qt's Model/View programming [70] paradigm is explored. However, this paradigm is more suited for application with complex data structure or applications where same data model could be used in different classes. In the context of AudibleT, the data structures were fairly simple and expected user experience and design components did not require the framework of the Model/View Programming at this phase.

4.7.2 Database Schema

The database schema design aims to efficiently store, organize, and retrieve data for the JND testing tool. The chosen schema structure allows for easy data access and management while maintaining a balance between performance, scalability, and ease of integration with the tool. The tool did not require any sophisticated database in particular to fit the requirements. The one fundamental requirement was to select a database to handle relational data insert and management because of the nature of the A/B/X testing. Therefore, Qt SQL [71] was a straightforward choice given that the tool was already utilizing the Qt [65] framework and its development environment.

4.8 Testing Strategy

A thorough testing plan has been developed to guarantee the dependability, effectiveness, and usability of the tool. The testing phases included in this strategy are unit testing, integration testing, system testing, and usability testing. The identification of potential problems and the verification of the tool's correct operation depend on each phase.

4.8.1 Unit Testing

Unit testing focuses on examining the accuracy of specific functions, classes, and components. Potential problems and bugs can be found and fixed early in the development process by separating these components and testing them in a controlled environment. A combination of automated test cases and manual inspection is used during unit testing to make sure that each function and component behaves as expected and complies with predetermined specifications.

4.8.2 Integration Testing

It examines the interactions between the tool's various modules, including the UI, the JACK external client, and the Data Analysis and Reporting module. This stage makes certain that these modules can effectively interact with one another and communicate with one another, resulting in a unified and seamless user experience. By developing test scenarios that mimic real-world use cases, integration testing enables the detection and correction of any potential problems that might arise when various tool components interact with one another.

4.8.3 System Testing

System testing uses end-to-end tests to verify the overall functionality and performance of the tool. This phase covers both functional and non-functional requirements to make sure the tool serves its intended purpose and functions properly in a variety of scenarios. System testing entails running a number of test scenarios designed to replicate real-world usage while evaluating aspects like dependability, performance, security, and compatibility with various hardware and software configurations.

4.8.4 Usability Testing

The main goal of usability testing is to assess how well the tool's user interface and overall user experience work. Aspects like usability, intuitiveness, aesthetic appeal, and responsiveness are evaluated during this phase. User comments, professional opinions, and heuristic assessments are used to conduct usability testing. Usability testing helps identify areas for improvement by gathering feedback from actual users and professionals, and it guarantees that the tool provides a satisfying user experience.

5 Implementation

This chapter presents the implementation process of the audio testing tool developed throughout this research project. The tool comprises three main modules described in sec 4.3. Each of these modules plays a vital role in ensuring the tool's functionality, performance, and usability. It aims to detail the implementation methodologies and techniques employed for each module, providing insights into the development process and the decisions made along the way, and detailed analysis of each modules functionalities.

5.1 User Interface (UI)

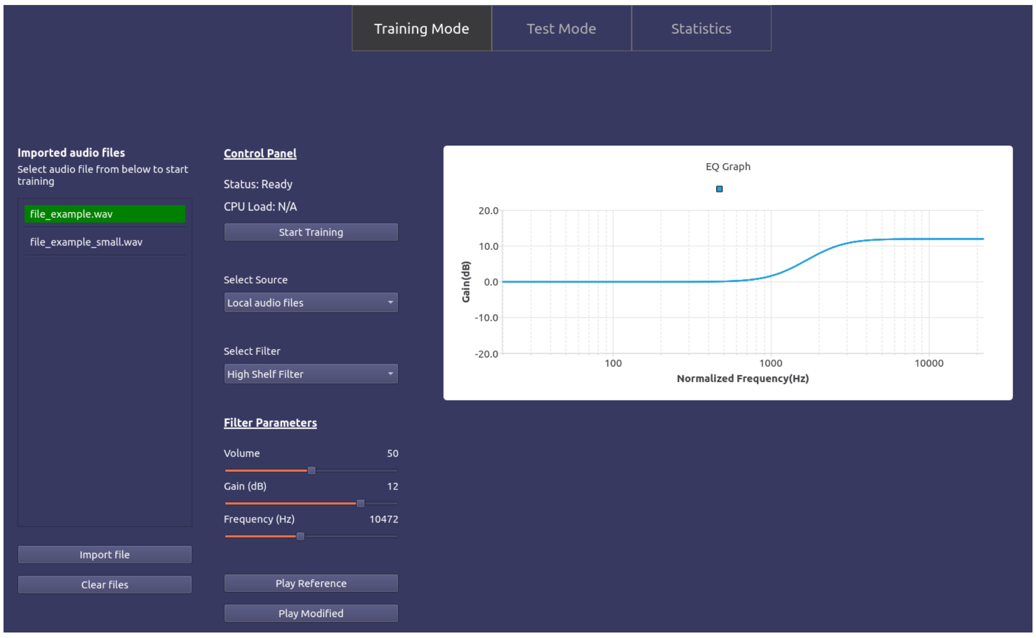
In this section, the implementation details of the User Interface (UI) for the tool is explored, focusing on the technical aspects that enable users to interact effectively with the application.

5.1.1 MainWindow

The main window serves as the central component of the application, providing the user interface for switching between different modes: Training Mode, Test Mode, and Statistics.

The custom *Window* class inherits from QWidget and is responsible for creating and managing the application's main window. The QMainWindow is not utilized since the UI is not sophisticated enough to need everything QMainWindow offers out of the box. It initializes the *TrainingWindow* [5.1.2], *TestingWindow* [5.1.6], and *StatisticsWindow* [5.1.7] objects and adds them to a *QStackedWidget*. This allows the application to easily switch between the different modes by changing the current widget in the widget container of type *QStackedWidget*. The *Window* class also sets up the database connection by creating a *DatabaseManager* object, which is responsible for managing the application's database interactions. Fig. 6 represents the software's main window.

The main window interface consists of three *QPushButton* objects that allow the user to switch between the different modes. The buttons are added to a *QHBoxLayout*, with leading and trailing *QSpacerItems* to keep the buttons centered horizontally. Finally, a *QVBoxLayout* is created to hold the widgets container, and another *QVBoxLayout* is created to combine the buttons layout and widgets layout vertically.



Status: Ready
CPU Load: N/A
Select Source: Local audio files
Select Filter: High Shelf Filter
Volume 50
Gain (dB) 12
Frequency (Hz) 10472

Figure 6: Main Window

The *QPushButton* objects are connected to their respective slot functions. These slots are responsible for changing the current widget in the widgets container and updating the button styles to reflect the currently active mode.

The *TrainingWindow*, *TestingWindow*, and *StatisticsWindow* objects are connected to various signals and slots to ensure proper communication and synchronization between them. For example, when the training session state changes, the *TestingWindow* and StatisticsWindow buttons are enabled or disabled accordingly. Similarly, when the filter settings or audio file selection in *TrainingWindow* changes, the *TestingWindow* is updated with the new information. Moreover, they are responsible for showing the appropriate widget based on the button pushed. Each of the push buttons comes with a distinctly selected and not selected style to help listeners understand which window is currently active.

5.1.2 TrainingWindow

The *TrainingWindow* class serves as the main window for training mode. It leverages the Qt framework and its widgets extensively. The *TrainingWindow* class inherits from the *QWidget* class, allowing it to act as a standalone window or be embedded within the main application, or be reuseable. The constructor initializes the window by creating various Qt widgets and arranging them in layouts.

The window interface consists of multiple sections, each represented by a Qt Widget. The *File List* section utilizes the custom *FileListWidget* [5.1.3] widget class, which encapsulates the list of imported audio files by the user. The *FileListWidget* is instantiated, and its corresponding content widget is added to the main layout using a QVBoxLayout.

The *Training Control Panel* [5.1.4] section includes a variety of widgets such as labels, drop-down menus (*QComboBox*), sliders (*QSlider*), and buttons (*QPushButton*). These widgets are arranged in nested layouts (*QVBoxLayout* and *QHBoxLayout*) to achieve the desired visual structure. The stylesheet mechanism provided by Qt is employed to define the appearance of the widgets, applying custom colors, font sizes, and padding.

The *Graph [5.1.5]* section employs the custom *EQGraph* widget, which visualizes the frequency response of the selected filter. The graph is updated dynamically by invoking the plot function with the relevant data.

5.1.3 FileListWidget

The *FileListWidget* class represents a widget responsible for managing and displaying the list of imported audio files by the users for training and testing purposes. It offers functionality to interact with the files, such as selecting a file for training in a training window.

A *QScrollArea* is created to accommodate the file list. The scroll-able content area is created as a child widget within the scroll area. It acts as a container for the individual file items, and each file item contains a *QLabel* for the file name. Besides, two buttons are created: *Import file* and *Clear files*, which allow users to import new audio files and clear the existing file list, respectively. The buttons are styled and connected to their respective slots using the clicked signal. With respect to the signal from the Import button, a *QFileDialog* window opens that allows listeners to select single or multiple audio files from their computer.

A public method is utilized to populate the file list based on the provided list of audio files from another parent class which helps make the class modular and to be used from other widgets if and when required. It begins by clearing the existing list first, and then, for each audio file, a label is created to display the file name. The label is customized with the file path stored as property and an event filter

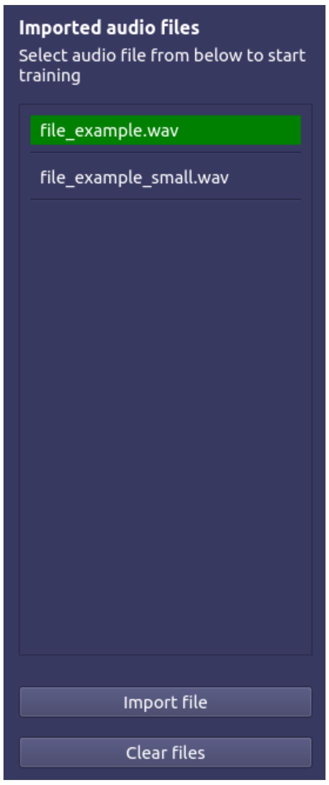


Figure 7: File List Widget

to detect mouse clicks. The styling of the label is determined based on the current selection state. A horizontal line is added below each label to separate the items visually. The label and line are added to the scroll layout to form each file list entry.

The *eventFilter* function from Qt handles mouse-click events on the file labels. When a label is clicked, it is identified as the clicked label. If the clicked label differs from the current selection, the visual styling of the labels is updated to highlight the selection. The current selection is updated to the clicked label, and a signal is emitted to indicate the selected audio file. Lastly, a *signal* is emitted with the file path of the currently selected audio file, if any, when that particular item is selected. It retrieves the file path stored in the label's property. The parent class then uses the file path to load the selected audio file data into the memory using AudioFile [68].

5.1.4 Training Control Panel

The *Select Source* drop-down menu lets users choose the audio source for the training session. It offers two options: *Local audio files* and *Microphone/JACK*. Selecting the *Local audio files* option reveals the file list widget, allowing users to import and select audio files from their local storage. On the other hand, choosing *Microphone/JACK* hides the file list widget and indicates that the audio input will be sourced from the microphone or JACK audio server.

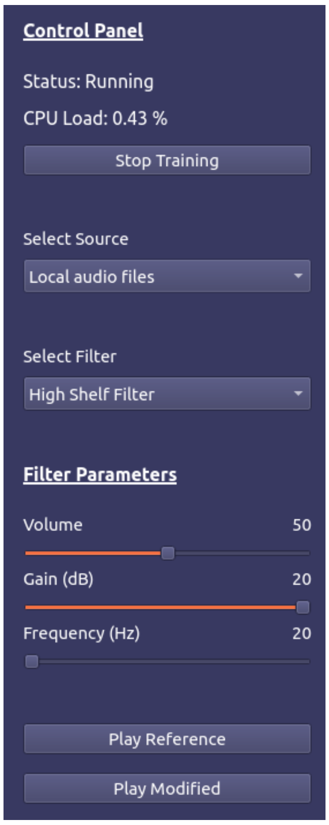


Figure 8: Training Control Panel

The *Select Filter* drop-down menu allows users to choose the type of filter to apply during the training session. It provides two options: ***High Shelf Filter*** and ***Low Shelf Filter*** at the moment. Selecting either option triggers the corresponding filter selection functionality and updates the audio processing accordingly in real-time with a barely noticeable delay.

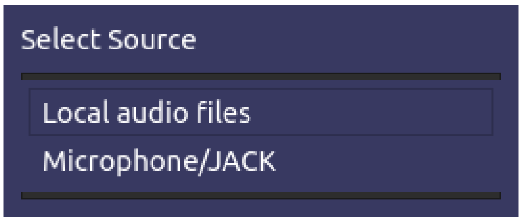


Figure 9: Source Selection Drop-down Menu

The *Filter Parameters* section displays the current settings of the selected filter. It consists of three sliders: *Volume*, *Gain (dB)*, and *Frequency (Hz)*. The *Volume* slider controls the overall volume level of the audio output. *The Gain (dB)* slider adjusts the gain applied by the selected filter, allowing users to modify the magnitude of the filter effect. The *Frequency (Hz)* slider determines the crossover of

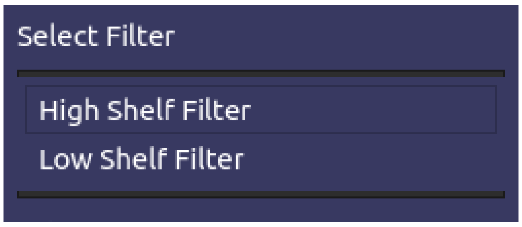


Figure 10: Filter Type Selection Drop-down Menu

the filter setting, influencing the frequency range affected by the filter.

As users interact with the sliders, the associated slot function is triggered, updating the equalizer parameters accordingly. The function sets the gain and cutoff frequency values based on the slider positions and emits a *signal* to inform other components about the updated values.

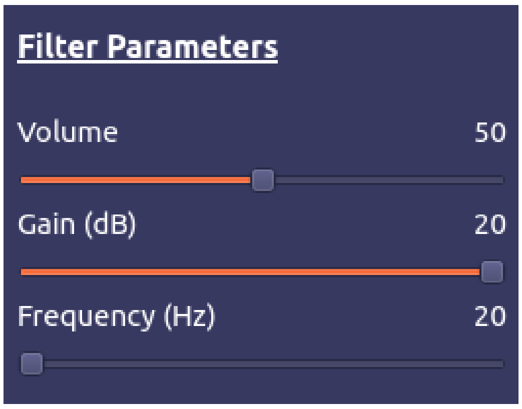


Figure 11: Filter Parameters Control

Additionally, the control panel includes two buttons: *Play Reference* and *Play Modified*, enabling users to play the selected audio file in its original state or with the applied filter effect. When clicked, the buttons invoke the corresponding actions through the JackWorker 5.1.8.1 instance, which handles audio playback.

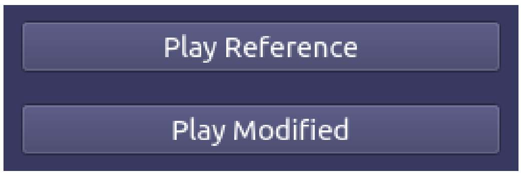


Figure 12: Reference and Modified track selection buttons

Moreover, the code introduces a separate *JackWorker* [5.1.8.1] class and a worker thread (QThread) to handle audio-related operations. The *JackWorker* class encapsulates the setup and control of the

JACK audio server. It is moved to the worker thread using the QObject's class' *moveToThread* function to offload audio processing tasks from the main thread, ensuring a responsive user interface. Signals and slot connections are established between the *JackWorker*, the worker thread, and the *TrainingWindow* class to facilitate communication, such as starting/stopping the audio engine and updating the engine's status label [72].

Consequently, The control panel section in the *TrainingWindow* class also includes the *Start Training / Stop Training* button, which allows users to initiate or halt the training session. Initially labeled as *Start Training*, the button serves as a trigger to begin the training process. When clicked, it invokes the corresponding slot function connected to the button's clicked signal. Upon activation, the button title changes to *Stop Training*, indicating the active training state and updating necessary state variables. Upon starting the training session, the button triggers several actions.

It starts the JACK audio server by invoking the appropriate function of the *JackWorker* class, which handles the audio processing engine. The audio source, either as a local audio file or JACK input, is set properly before starting the JACK audio server since the audio input and output port setup holds dependency with the said configuration. The active filter type is also set and passed to the audio processing engine. Simultaneously, the button disables the *Select Source* drop-down menu to prevent changes during the ongoing training session. It updates the button's tooltip to inform users that audio source selection is disabled while the training session is active.

Conversely, when the training session is already running, clicking the button triggers the *Stop Training* functionality. It stops the JACK audio server using the corresponding function of the *JackWorker* class. The button text reverts to *Start Training*, indicating that the training session has been halted. The *Select Source* drop-down menu becomes enabled again, allowing users to change the audio source for future sessions.

5.1.5 EQ Graph

The *EQGraph* widget within the *TrainingWindow* class is responsible for displaying an equalizer graph representing the selected filter's frequency response. It utilizes *QtCharts* module to produce a graphical representation that could help users visualize and understand the impact of the applied filter on different frequency ranges.

The *EQGraph* widget sits on the right side of the control panel section of the *TrainingWindow*. The behavior of the *EQGraph* widget is driven by the selected filter type and the corresponding filter parameters set by the user. When the user changes the filter parameters using the gain and cutoff frequency sliders or when the selected filter type changes from the drop-down menu, the corresponding function is invoked. The appropriate *FilterType* [5.1.12.3] and the corresponding gain and cutoff frequency values are determined inside the method. The necessary filter coefficients (B and A) are computed based on the filter type where B and A refers to non-recursive and and recursive coefficient vectors respectively. These coefficients represent the filter's transfer function and determine the filter's effect on the audio signal.

Next, a set of frequency points is chosen to represent the frequency range of interest, spanning from 20 Hz to 24 kHz. The gain (in dB) at each frequency point is calculated by applying the filter's transfer function to the selected frequency values. This determines how the filter modifies the audio signal's amplitude at different frequency regions.

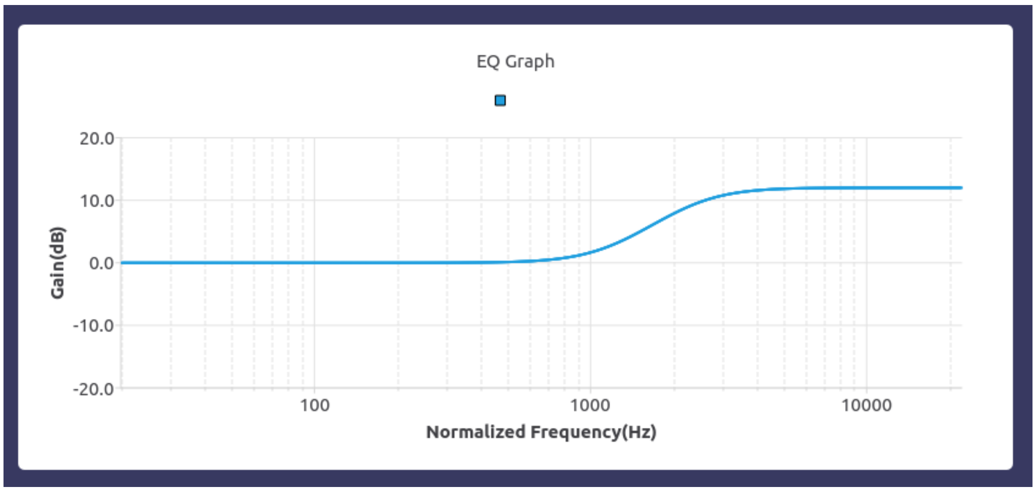


Figure 13: EQ Graph

In the end, the *EQGraph* widget is updated by invoking its plot function. The frequency points and corresponding gain values, along with axis labels and graph titles, are passed as parameters to generate the visual representation of the frequency response. The graph is displayed within the *EQGraph* widget, allowing users to observe the changes in the frequency response as they adjust the filter parameters.

5.1.6 TestingWindow

The *TestingWindow* component is the core part of the tool that allows listeners to conduct audio testing sessions. The implementation involves creating a *QWidget*-based class that encapsulates the functionality and visual layout of the window. The *Control Panel* section of the testing window is designed using *QLabel*, *QPushButton*, and *QSpinBox* widgets. It allows users to configure the testing session by selecting the number of trials and the audio file to test with for that particular session. The Start/Stop button triggers the initiation or termination of the testing session. The *Control Panel* layout is organized using *QVBoxLayout* and *QHBoxLayout* to achieve the desired vertical and horizontal alignment of the widgets.

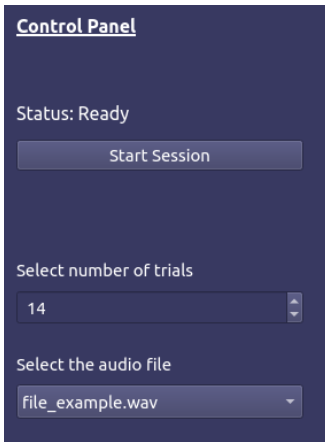


Figure 14: Test Control Panel

Fig. 15 shows the Testing Panel section, that provides real-time feedback during the testing session. It displays how many attempts are left out of desired trials and the currently playing audio track and provides buttons to play the respective tracks (Track A, Track B, Track X). It is worth mentioning that the currently playing status only shows the abstract label of the track, such as *Track A*, *Track B*, or *Track X*. It doesn't indicate which one is the modified track or the reference track. Assigning of track label, whether modified or reference track, will get *Track A* or *Track B* label is controlled by a separate method which performs a seed generation and random label distribution after each trial, hence, making the labeling is entirely random. Finally, listeners can select their response indicating which track (A or B) they perceive as most similar to Track X. The Testing Panel layout is also structured using *QVBoxLayout* and *QHBoxLayout* to arrange the widgets in a visually appealing and intuitive manner.

The implementation includes event handling and signal-slot connections to enable interaction with the user interface. For example, when the Start/Stop button is clicked, it triggers the corresponding slot function to start or stop the testing session. Similarly, selecting an audio file from the dropdown

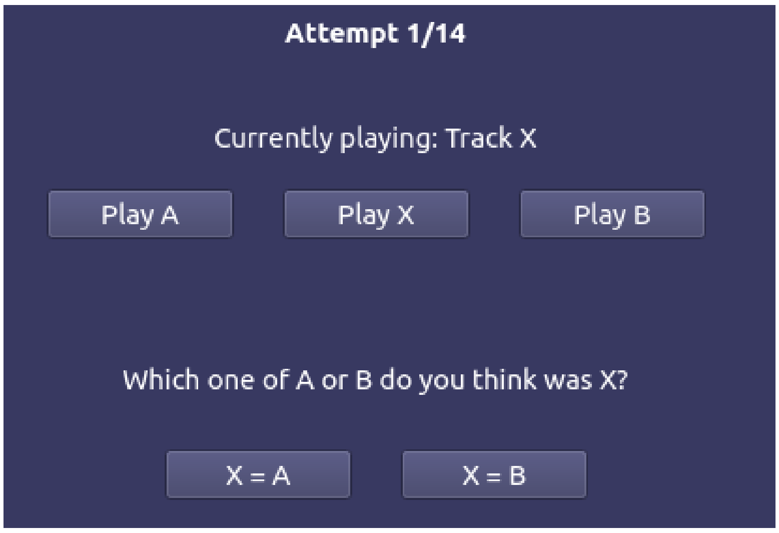

Attempt 1/14
Currently playing: Track X
Play A
Play X
Play B
Which one of A or B do you think was X?
X = A
X = B

Figure 15: Test Area

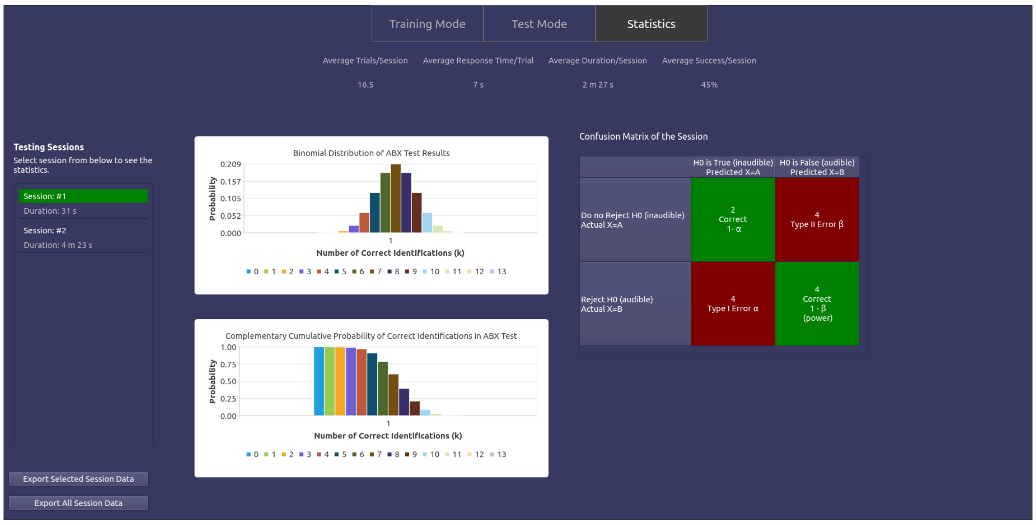
menu emits a signal that triggers the associated slot function to update the selected audio file.

Furthermore, the window interacts with the Jack server to handle audio playback and processing, as stated in 5.1.2. Additionally, the Play buttons (Play A, Play B, Play X) are connected to slots that trigger the playback of the respective audio tracks through the *JackWorker*. These buttons interact with the *JackWorker* to set the audio file to be played, and the currently playing label is updated accordingly. The window also involves integrating additional components, such as a *DatabaseManager*, to handle data storage and retrieval. The Testing Window interacts with the *DatabaseManager* to populate the available audio files imported by the listeners from the local computer for selection in a testing session. It also creates a new testing session record in the database when a new session is started and logs the listener responses and other information, such as expected response, into a separate table associated with the corresponding session. These records are later utilized for report generation, which is explained in 5.1.10.

5.1.7 StatisticsWindow

The Statistics Window is responsible for displaying statistical graphs and session information related to the testing sessions. It provides visual representations of the data gathered during the testing process. The implementation of the Statistics Window which is illustrated in fig. 16 involves the following key components:

- **Statistics Summary Widget**: This widget displays some specific summary of the total sessions conducted within the tool such as average trials per session, average response time per trial,


Training Mode
Test Mode
Statistics
Testing Sessions
Select session from below to see the statistics.
Session: #1
Duration: 31 s
Session: #2
Duration: 4 m 23 s
Export Selected Session Data
Export All Session Data
Binomial Distribution of ABX Test Results
Probability
Number of Correct Identifications (k)
Complementary Cumulative Probability of Correct Identifications in ABX Test
0 1 2 3 4 5 6 7 8 9 10 11 12 13
Confusion Matrix of the Session

Average Trials/Session	Average Response Time/Trial	Average Duration/Session	Average Success/Session
16.5	7 s	2 m 27 s	45%

	H0 is True (inaudible) Predicted X=A	H0 is False (audible) Predicted X=B
Do no Reject H0 (inaudible) Actual X=A	2 Correct 1- α	4 Type II Error β
Reject H0 (audible) Actual X=B	4 Type I Error α	4 Correct 1 - β (power)

Figure 16: Statistics Window

average duration per session, and average success rate per session. A custom class *StatisticsSummaryWidget* is utilized to create the widget using *QLabel*, *QVBoxLayout* and *QHBoxLayout*.

- **Session List Widget**: This widget displays a list of testing sessions. It is created using the *SessionListWidget* class, which provides the necessary UI elements to present the session information. The implementation is similar to *FileListWidget* [5.1.3]. The main difference between the *SessionListWidget* and *FileListWidget* is the *SessionListWidget* contains one extra label for each item in the scroll view that shows the duration of each session in a user-readable format. The session list is populated by retrieving session data from the database through the *DatabaseManager*. The corresponding binomial graphs and confusion matrix are updated upon selecting a testing session from the list.

- **Session Data Export**: The statistics window allows listeners to export both single test session data as well as all sessions data into a CSV file. A custom class called *CSVExporter* is created that communicates to the *DatabaseManager* to prepare the export data based on listeners' selection from the UI. When the data is ready, a system modal opens on the listeners computer to select the location for the file to store. Fig. 18 shows the buttons available for exporting the session data.

- **Binomial Graph**: A custom class is created utilizing *QtCharts* bar chart modules to represent a graphical display of the binomial distribution. It visualizes the results of the trials conducted during a selected testing session. The graph is updated by retrieving the trial data for the selected session from the *DatabaseManager* and passing it to the object created from the Bino-

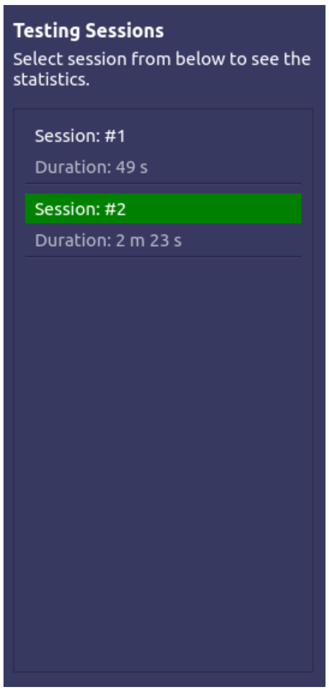


Figure 17: Session List Widget

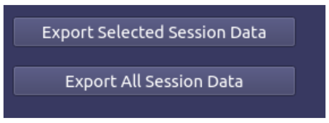


Figure 18: Session Data Export Options

mial Graph class for display through the corresponding method for creating and populating the chart. The reasoning behind implementing Binomial Graph is discussed in 5.1.10.3.

- **Inverse Binomial Graph**: For inverse binomial graph, the same Binomial Graph class is utilized but with a different method to create the chart and populate the data since the algorithm to generate inverse binomial graph is fairly different than the binomial graph. More in-depth insight into the data analysis is presented in 5.1.10.3.

- **Confusion Matrix**: The presentation of the each test session data confusion matrix required creating a custom widget class *ConfusionMatrixWidget* with utilizing *QtableWidget* to create a 3×3 table. The class exposed a public method which is called from the *StatisticsWindow* class when user selects an item from the *SessionListWidget* to update the table with with confusion matrix data prepared by the data analysis module. An in-depth theory behind this analysis is presented in 5.1.10.2.

5.1.8 JACK External Client (JEC)

JEC is one of the core parts of the tool that does the heavy lifting, such as processing the audio in real-time with the equalizer configuration and filter coefficients generated in the UI real-time. However, it is worth mentioning that the JACK client can be both external and internal [72]. An external JACK client is appropriate in this scope as it simplifies the tool design.

The *JackClientAudioEngine* class is the heart of the project's real-time audio processing functionality. It extends an AudioIO [67] class that provides a framework for real-time audio input/output (I/O) using the JACK Audio Connection Kit. The real-time audio processing operation is set up, customized, and executed in this class. This class is managed by *JackWorker*, which runs in a separate worker thread. The constructor of the class configures the audio *I/O* by setting the client name and the number of input and output channels. It also created a filter object with default filter configuration settings and sets the initial position of the audio sample to 0 (zero) to reset the sample position at the beginning of a session.

The most significant method in the class is the audio callback function, where the real-time audio processing takes place. The JACK server invokes the function whenever there are an incoming block of audio frames to process. The function begins by creating an instance of a *BaseFilter* [5.1.9.2] object, the type of which depends on the user's selection of filter type [5.1.12.3]. The filter is used to process the audio data in real time.

The audio data processed in the function can come from an external JACK input or an audio file. If the source is an external input, each sample of the incoming audio data is fetched, optionally processed by the filter (depending on the user's choice), and scaled by the user-set volume level. The processed sample is then sent to the output buffer.

If the source is an audio file, the function fetches each audio sample from the file, optionally processes it with the filter, and scales it with the volume level. The processed sample is then written to the output buffer. The function keeps track of the current position in the audio file and moves to the next sample position after processing each sample, looping back to the beginning of the file if the end is reached.

The class also provides methods for configuring the audio processing operation. These methods allow the user to set the volume level, the parameters for the filter, the audio file to be processed, whether the audio data should be filtered, the source of the audio data, and the type of filter to use. The audio data filter flag is utilized both in the training mode [5.1.2] and testing mode [5.1.6]. When listeners choose to play a modified track, the filter flag is set to *true*, which then processes the audio sample based on the selected filter and configurations. In contrast, when a reference track is chosen, the filter flag is set to *false*, which writes back the original input buffer to the output buffer without any filters being applied. These methods are all expected to ensure that the audio engine is flexible and adaptable enough to various use cases, allowing listeners to customize the real-time audio processing operation to their specific needs.

5.1.8.1 JackWorker

This class handles the project's audio processing management as this serves as an intermediary between the user interface and the JEC [5.1.8]. This class takes responsibility for initiating the setup of the real-time audio server. Once instantiated, the class sends a status update indicating the readiness of the audio engine to the UI by Qt signal, and the UI then notifies this class to start the JACK audio server as soon as listeners push the start button. Qt's signal and slot mechanism enables real-time inter-object communication as well as being capable of communication across different threads.

Upon receiving a signal to start the audio server, the intermediary class checks whether the audio engine is active. If it is not, the intermediary resets the audio engine, initiates its operation, and connects the audio engine's outputs and inputs to the physical ports. When the audio source is an external input, the class ensures that the source ports are also connected to the audio engine's inputs. After successfully starting the audio engine, it sends a status update to the parent class to indicate that the audio server is now running.

Similarly, when it receives a signal to stop the audio server, it first checks if the audio engine is active. If it is, it stops the engine, resets it, and sends a status update indicating that the audio server has been stopped.

This class also manages various configurations for the audio engine. These configurations include setting the audio source, the type of active filter, the volume level, and the filter parameters. In addition, it sets the audio file for playback and a flag that indicates whether the audio file's playback state has been modified as described in 5.1.8. After any change in these configurations, it checks whether the audio engine is active before applying the changes to prevent any unexpected behavior from the tool.

Should the audio engine need a reset, the *JackWorker* class deletes the current instance of the audio engine, creates a new one, and sends a status update.

5.1.9 Digital Filters

A discrete-time signal can be selectively enhanced or suppressed using a digital filter. Digital filters' main objective is frequently changing a digital signal's frequency content by damping or enhancing undesired frequencies, boosting specific frequency bands, or changing the phase relationships between different frequency components.

Finite impulse response (FIR) filters and Infinite impulse response (IIR) filters are the two basic subtypes of digital filters. Because FIR filters have a finite impulse response, their output for a given input will eventually reach zero after a predetermined number of samples. IIR filters have an infinite impulse response, meaning their output might theoretically continue long after the input stops. They are often more computationally economical than FIR filters [73, p.364-368]

However, this project focused on designing shelving filters, and updating filter coefficients in real-time.

5.1.9.1 ShelfFilter

Digital filters are employed in the context of audio processing to modify the signal's frequency spectrum to provide various effects, including equalization, noise reduction, and spatialization. To create a frequency response that resembles a shelf, shelving filters, a particular form of the digital filter, can modify the amplitude of frequencies in a signal above or below a given cutoff frequency [74].
The *ShelfFilter* class in this project is implemented to represent such a shelving filter. The filter is initialized with key parameters, including the sample rate (Fs), cutoff frequency (Fc), gain in decibels (g), and filter type. After initialization, the filter coefficients are updated based on these parameters. The gain multiplier (G) is computed from the gain in decibels using the formula:

$$G = 10^{g/20}, \tag{1}$$

where G is the gain.

The cutoff frequency is normalized to the sample rate and represented in radians per sample using the formula:

$$\omega = 2\pi(Fc/Fs). \tag{2}$$

Two sets of coefficients are calculated - one set for scaling the input samples (b_0, b_1, b_2) and the other for scaling the output samples (a_0, a_1, a_2). These coefficients are determined based on the filter type, which can either be a high-shelf or low-shelf filter.

For low-frequency region, the coefficients are calculated from the transfer function:

$$H_{\mathrm{LS}}(z) = G^{1/2}\frac{G^{1/2}\Omega^2+\sqrt{2}\Omega G^{1/4}+1+2\left(G^{1/2}\Omega^2-1\right)z^{-1}+\left(G^{1/2}\Omega^2-\sqrt{2}\Omega G^{1/4}+1\right)z^{-2}}{G^{1/2}+\sqrt{2}\Omega G^{1/4}+\Omega^2+2\left(\Omega^2-G^{1/2}\right)z^{-1}+\left(G^{1/2}-\sqrt{2}\Omega G^{1/4}+\Omega^2\right)z^{-2}}. \tag{3}$$

For a high-frequency region, the coefficients are calculated from the transfer function:

$$H_{\mathrm{HS}}(z) = G^{1/2}\frac{G^{1/2}+\sqrt{2}\Omega G^{1/4}+\Omega^2-2\left[G^{1/2}-\Omega^2\right]z^{-1}+\left(G^{1/2}-\sqrt{2}\Omega G^{1/4}+\Omega^2\right)z^{-2}}{G^{1/2}\Omega^2+\sqrt{2}\Omega G^{1/4}+1+2\left[G^{1/2}\Omega^2-1\right]z^{-1}+\left(G^{1/2}\Omega^2-\sqrt{2}\Omega G^{1/4}+1\right)z^{-2}}. \tag{4}$$

The notations are given respectively in [74], (18)-(19). These coefficients are normalized by the a_0 coefficient to ensure a unity gain at mid frequency. Normalizing the coefficients in this way guarantees that the filter does not alter the gain of the mid-frequency region of the signal, effectively preserving the signal's average value.

The filter processes input samples by storing them in a buffer and calculating the output sample based on the stored input and output samples, scaled by their respective coefficients. The output sample is computed using the equation which is a representation of a liner time-invariant system [75], pp. 20-34:

$$y[n] = b_0x[n]+b_1x[n-1]+b_2x[n-2]-a_1y[n-1]-a_2y[n-2], \tag{5}$$

where x[n] is the current input sample, x[n-1] and x[n-2] are previous input samples, and y[n-1] and y[n-2] are previous output samples.

In the end, the filter provides access to the computed vectors of input and output scaling coefficients which are utilized to calculate normalized frequency response through *FrequencyResponseHelper* [5.1.12.1] class to plot the EQGraph [5.1.5].

5.1.9.2 BaseFilter

This is an abstract class that provides skeleton of a basic audio filter. It forms a blueprint for creating different types of audio filters. The class provides a common interface for all filters in the form of a method that processes an audio block. The *process_sample()* method of this abstract class takes each audio block as input and returns the processed audio block.

However, the actual implementation of this function is left to the derived classes. This allows each derived class to implement its own version of the audio processing function, which might involve applying different filtering algorithms or transformations to the audio sample. In this project, *ShelfFlter* [5.1.9.1] inherits from this class.

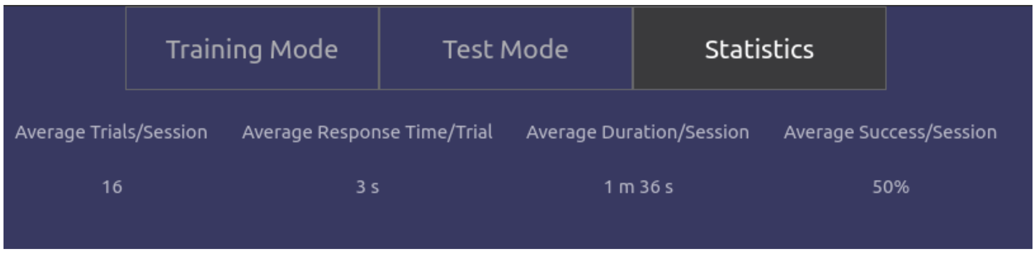
This abstract class effectively sets a standard interface for all audio filters and allows the integration of different types of filters in the audio processing pipeline to keep the implementation consistent.

5.1.10 Data Analysis and Reporting

This section presents the approach followed to implement another vital part of the tool, Data Analysis and Reporting. The type of analysis offered within the tool is detailed as follows:

5.1.10.1 Session Summary

In the statistics window, a simple summary stat is presented at the top. The summary data is calculated from database records such as the total number of sessions, listeners response time for each trial, duration for each session and the actual correct and incorrect response from the trials. The *DatabaseManager* exposes different methods that returns the computed average number of trials a listener attempter in each test sessions, average time(seconds) taken by the listener to choose A/B in an attempt, average time duration of each testing session, and average success rate listener had for each testing session. These values then presented through *StatisticsSummaryWidget*.



Average Trials/Session	Average Response Time/Trial	Average Duration/Session	Average Success/Session
16	3 s	1 m 36 s	50%

Figure 19: Sessions Summary Widget

5.1.10.2 Confusion Matrix

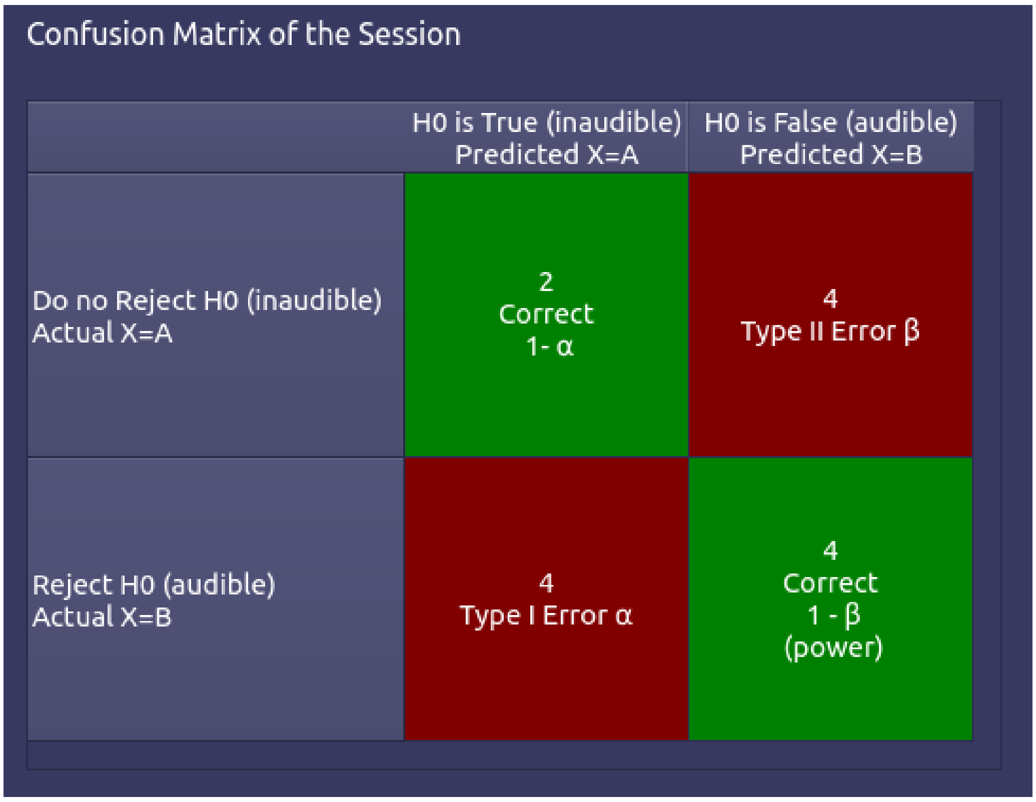
This section provides a detailed explanation of Type I and Type II errors, both of which are common terms in statistical hypothesis testing and how it is utilized in this software.

A Type I error marked by mistake, also known as a false positive, occurs when one rejects a null hypothesis that is actually true. The probability of making a Type I error is determined by the significance level, also known as alpha (α). This error may lead to false conclusions about statistical significance. On the other hand, a Type II error, or false negative, happens when one fails to reject a null hypothesis that is actually false. This error occurs when one overlooks an actual effect due to lack of statistical power, which could be a result of smaller effect size, larger measurement errors, small sample size, or low significance level. Beta (β) represents the probability of making a Type II error. While this error might lead to missed opportunities for innovation or improvement, its impact is generally considered less severe than a Type I error [60].

The concept of Type I and Type II error is utilized to develop a confusion matrix and present the results of an audio testing sessions. In the context of ABX testing, listeners' responses produce a confusion matrix as follows. Fig. 20 provides visual representation of a testing session of 14 trials using confusion matrix.

- Cell $[1, 1]$: True Positives (TP): The listener correctly chosen Track A as X.
- Cell $[1, 2]$: False Positives (FP): The listener incorrectly chosen Track A as X when it was actually B.
- Cell $[2, 1]$: True Negatives (TN): The listener correctly chosen Track B as X.
- Cell $[2, 2]$: False Negatives (FN): The listener incorrectly chosen Track B as X when it was actually A.

Through this, the confusion matrix can provide detailed insights about the listener's ability to correctly identify the difference. It can also highlight the occurrence of Type I and Type II errors in the test. A Type I error, in this context, would be when a listener incorrectly identifies the original track as distorted (False Positive). A Type II error would occur when the listener fails to identify the distorted track and believes it to be the original (False Negative).


Confusion Matrix of the Session

	H0 is True (inaudible) Predicted X=A	H0 is False (audible) Predicted X=B
Do no Reject H0 (inaudible) Actual X=A	2 Correct 1- α	4 Type II Error β
Reject H0 (audible) Actual X=B	4 Type I Error α	4 Correct 1 - β (power)

Figure 20: Confusion Matrix of a Test Session Result

5.1.10.3 Binomial and Inverse Binomial Distribution

The ABX testing methodology evaluates whether an audible difference exists between two audio signals, "*A*" and "*B*", by asking listeners to identify an unknown signal "*X*" as either "*A*" or "*B*". It is also worth mentioning that, It can only be verified that a difference is audible, but it is not possible to prove that is is inaudible. The premise of ABX testing aligns with the principles of binomial distribution: if there is no audible difference between the signals, the listener's responses are expected to be binomially distributed, with equal probability for both "*X=A*" and "*X=B*". Therefore, the correct interpretation of ABX test results fundamentally depends on an appropriate statistical analysis. This need is thoroughly addressed by Boley [3], where the authors emphasize the role of proper statistical analysis in drawing meaningful conclusions from ABX tests.

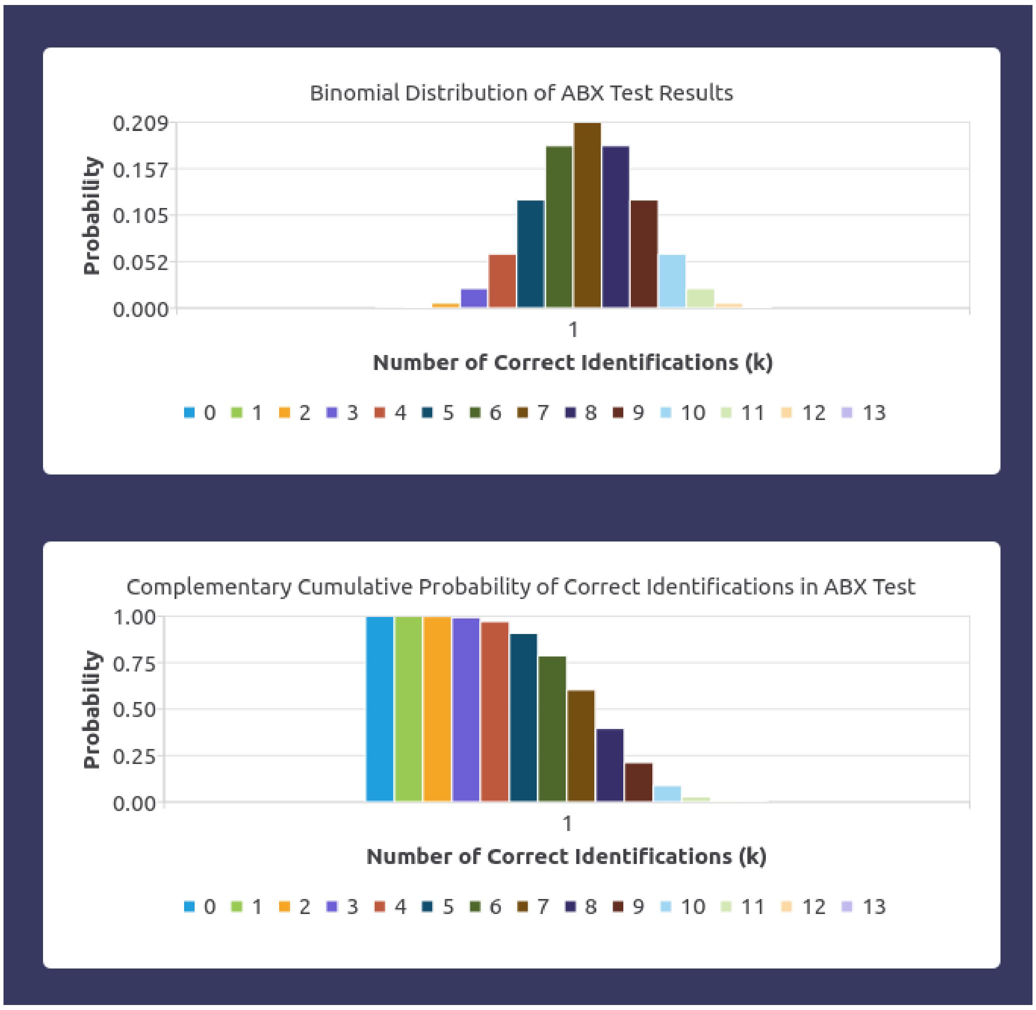


Figure 21: Binomial and Inverse Binomial Distribution

As outlined in the research, if there is no perceptible difference between the two audio signals under test, listeners' responses to whether stimulus "*X*" is the same as "*A*" or "*B*" will follow a binomial distribution, even when there a difference is detected. The binomial distribution can effectively visualisation the probability of a certain number of successes (correct identifications of "*X*") in a fixed number of trials (total number of identifications), making it a practical and valid choice for statistical analysis in this context. Similarly, the use of inverse binomial distribution is justified to determine the confidence with which a perceived difference between two signals can be declared significant. The inverse binomial distribution, or the negative binomial distribution, is typically used to model the number of trials required to achieve a specified number of successes randomly. In the context of an ABX test, it can be utilized to approximate the number of trials needed before a certain number of correct identifications (or "successes") occur. This aids in understanding the confidence level of the test results, which is critical for the interpretation of the results. *BinomialHelper* [5.1.12.2] discusses more the implementation of the distribution algorithms. Additionally, it is worth noting that as the number of trials increases significantly, the binomial distribution approximates a Gaussian distribution, also known as the normal distribution. Fig. 21 shows the visual representation of Binomial and Inverse Binomial of a test session with 14 trials.

The decision to include binomial and inverse binomial distribution in the ABX audio testing tool as well as the confusion matrix was notably influenced by previous academic research in the field [60]. They ultimately serve to improve the accuracy and reliability of the implemented tool's results, thereby contributing to our goal of enhancing the understanding of perceptual differences in audio signals.

5.1.11 Database

5.1.11.1 DatabaseManager

This acts as a middleware between *QtSql* database and the UI, such as the training, testing or statistics modules. The class encapsulates the database operations such as opening and closing the database, executing queries, and setting up the database schema, and exposes necessary public methods accessible from outside.

The database schema consists of three tables illustrated in fig 5. The class provides the capability to insert audio file paths into the database. This operation requires the path and the name of the audio file. This method is particularly used when listeners import audio files from their local computers. In addition to inserting audio files, the class also exposed a method to retrieve all audio files from the database. It does this by returning a list of *AudioSample* [5.1.11.2] objects which represent the audio files. Moreover, If there is a need to delete all audio files, the class also provides a method that removes all records of the audio sample from the database.
Furthermore, the class also provides functionalities related to the management of test sessions, such as creating a testing session and updating a session when it ends. A method is exposed to retrieve all test sessions along with their duration from the database. It does this by returning a vector of *TestingSession* [5.1.11.3] objects which represent the test sessions.

Moreover, this class also provides functionalities for managing trials within test sessions. It can insert a new trial record into the database. This operation requires the unique identifier of the session for the trial, the correct response for the trial, the listener's response for the trial, and the time the user takes to respond. Consequently, there's a method to retrieve all trials for a specific session from the database. This operation requires the unique identifier of the session for which the trials are to be retrieved, and it returns a vector of *SessionTrial* [5.1.11.4] objects representing the trials that are utilized in constructing the *Binomial Graph* and *Inverse Binomial Graph*.

5.1.11.2 AudioSample

This structure represents an audio sample fetched from a database. It's a simple blueprint that holds crucial information about each audio sample. Each audio sample is uniquely identified by an ID, which is stored in *sampleId*. This is important for differentiating between the samples and allows for efficient retrieval from the database. The *fileName* field represents the name of the audio file corresponding to the sample. This name can be used to present the file to the user or to perform operations on the file. The *path* field stores the exact location of the file within the file system. This information is essential when the audio file needs to be read, manipulated or moved.

5.1.11.3 TestingSession

This struct represents a single testing session. The *sessionId* is a unique identifier for each testing session, allowing for the tracking and management of individual sessions within the database. The *sampleId* refers to the specific audio sample being tested in this particular session. This allows for the association of testing sessions with specific audio samples, which is vital for interpreting the outcomes within the framework of the audio sample under test. The *trialsCount* field represents the maximum number of trials configured for this session. This gives an indication of the length and extent of the session. The *duration* field represents the total length of the testing session in seconds. This provides a measurement of the total time that the user spent on the testing session, offering insights into the user's engagement and persistence.

5.1.11.4 SessionTrial

The SessionTrial struct represents a single trial within a testing session. It contains several properties. The *trialId* is a unique identifier for a trial within a given session. The *sessionId* represents the identifier of the session to which this trial belongs. The *responseTime* is an integer value representing the time the listener takes to respond to the trial, measured in milliseconds. This is crucial in understanding listeners' performance and responsiveness during the testing session. The *correctResponse* is a string value representing the correct response for this trial. This mainly represents the expected output or reaction from the listener in response to the audio stimulus presented during the trial. The *userResponse* is another string value that represents the actual response provided by the listener for this trial. By comparing the *correctResponse* and *userResponse*, the application can evaluate the accuracy and correctness of the listeners' response.

5.1.12 Utils

This subsection explains the utility classes, structures, and enumerations used throughout the tool's implementation to make the source code scalable, maintainable, and clean.

5.1.12.1 FrequencyResponseHelper

This class possesses two methods that work together to compute and return the frequency response of the shelving filters for a logarithmically spaced sequence of frequencies.

The first method A generates a sequence of points that are spaced logarithmically between a given start and end point, with each successive frequency increasing by a constant ratio (e.g., double the previous frequency to get an octave or some fraction thereof). This aims to create a set of frequencies that are distributed more densely at the lower end of the spectrum. This is particularly useful in audio processing, where our perception of pitch is logarithmic, indicating that we interpret equal frequency ratios as equal pitch variations.

The second operation calculates the frequency response of the filter, see listing. A. The frequency response is a crucial characteristic of a filter, showing how it affects the amplitude and phase of different frequencies in a signal. The implementation followed the concept of complex exponential signals, which are fundamental to the analysis of linear time-invariant (LTI) systems in the frequency domain. The general form of the complex exponential signal is $e^{j\omega t}$, where *j* is the imaginary unit ($\sqrt{-1}$), ω is the angular frequency, and t is the time variable. In the context of signal processing and systems analysis, $e^{j\omega t}$ represents a signal that oscillates at a frequency ω [75]. The process starts by defining the time between each sample, known as the sampling period, and a representation of the imaginary unit. It then iterates over the list of frequencies for which we want to calculate the frequency response, see listing. A.

The process transposes the filter's coefficients into the frequency domain at each frequency. This is achieved by employing complex exponentials that represent the frequency on the unit circle in the z-domain, where $z = r.exp(j\omega)$, and the radius r is set to 1. These complex exponentials are computed by taking the exponential of a product involving the imaginary unit, the frequency, and the sampling period.

The process then evaluates the filter's transfer function at the given frequency. The transfer function of a filter is a mathematical representation of how the filter transforms input signals into output signals.

Once the transfer function is evaluated, the process computes the magnitude of the result. This magnitude represents the gain of the filter at the current frequency, showing how much the filter amplifies or attenuates the signal at that frequency.

Finally, the gain is converted to decibels to express it on a logarithmic scale. This is more suitable for audio applications, as our perception of loudness is also logarithmic [76] - we perceive equal ratios of amplitudes as equal differences in loudness.

5.1.12.2 BinomialHelper

The class is fundamentally designed for calculations related to the binomial distribution. The binomial distribution represents a probability distribution outlining the count of successful outcomes in a predetermined number of independent trials that follow the Bernoulli distribution. In this tool, the binomial distribution is presented as part of *Data Analysis and Reporting* module.

The class begins by calculating the logarithm of a factorial. Factorials, even for relatively small inputs, can become very large. This can potentially lead to issues with numerical stability and precision in calculations. By calculating and using the natural logarithm of a factorial instead of the factorial itself, those issues can potentially be mitigated. The mathematical formula used here is given in [77]:

$$ln(n!) = ln(1) + ln(2) + ... + ln(n). \tag{6}$$

The class then uses this concept to calculate the natural logarithm of a combination. In mathematics, a combination refers to the method of choosing elements from a larger group where the sequence of selection is irrelevant, distinguishing it from permutations where order matters. This is often symbolized as *n choose k*, which represents the number of possible selections of k items from a set of

n. The combination calculation is critical in binomial distribution calculations because it determines the number of ways a certain number of successes can occur in a given number of trials. The formula used is:

$$ln[C(n,k)] = ln(n!) - [ln(k!) + ln((n-k)!)]]. \tag{7}$$

In binomial probability, the combination part of the equation represents the number of ways we can have exactly 'k' successes in 'n' trials. The rest of the equation represents the probability of any one of those ways occurring. Therefore, the combination is used in the calculation of binomial probability. The binomial probability is calculated in the class using these concepts: the number of combinations and the probabilities of success and failure. The calculation followed the formula for the binomial distribution. This probability reflects the likelihood of seeing a specific number of successes (k) in a specific number of Bernoulli trials (n). The formula used is:

$$P(X = k) = C(n,k) \times (p^k) \times ((1-p)^{n-k}), \tag{8}$$

where 'p' is the success probability in a single trial.
The class also provides a method to calculate the cumulative distribution function (CDF) for a binomial distribution. The Cumulative Distribution Function (CDF) at a specific point represents the likelihood that the variable assumes a value that is less than or equal to this particular value. The CDF is computed by summing up the probabilities of all outcomes less than or equal to the given number of successes. The formula used here is:

$$P(X \leq k) = \sum P(X = i), \tag{9}$$

for i = 0 to k
Moreover, the class provides an inverse function for the CDF. This function finds the smallest number of successes such that the cumulative probability is greater than or equal to a target probability.

5.1.12.3 FilterType

This enumeration provides two distinct types of audio filters at this moment: HighShelf and LowShelf. It is utilized in several places of the application, such as the audio processing engine and the user interface, to control components like computing filter coefficients, generating normalized frequency, or plotting frequency graphs.

The *HighShelf* enumeration value represents a low-frequency region. This filter allows frequencies above a certain cutoff point to pass through unaffected, while frequencies below that point are either boosted or attenuated.

The *LowShelf* enumeration value, on the other hand, represents a high-frequency region. This is essentially the opposite of a high-shelf filter. It allows frequencies below a certain cutoff point to pass through, while frequencies above that point are either boosted or attenuated.

5.1.12.4 SessionType

The *SessionType* enumeration is used to represent the current state of a testing session. This state information is crucial as it affects what operations can be performed on the session at any given time. It consists of three states - Stopped, Running, and Paused.

The *Stopped* state indicates that the session is currently not active. This could mean that the session has not yet started or has been completed and stopped. During this state, adjusting session settings or preparing for a new session is typically possible. The *Running* state indicates that the session is currently active and ongoing. The listeners would typically interact with the application during this time, and certain operations (like changing session parameters) are not permissible to ensure the integrity of the session. The *Paused* state is used when a session is temporarily halted but not completely stopped. This state might be used if the user needs to take a break or if there's an interruption that requires temporarily halting the session. In the paused state, the session can be resumed at the point it was halted.

5.1.12.5 SourceType

This enumeration represents the source of audio input within the application. It's used to determine how the application accesses and processes the audio data it needs for training and testing. Besides, the user interface [2] modules several components depending on the audio source type.

There are two possible sources of audio input at this moment:

- Local - state signifies that the audio input source is the local audio files of *Waveform Audio File Format(WAV)* stored somewhere within the listeners' computer.

- JACK - indicates that the audio input is sourced from a Jack Audio Connection Kit (JACK) input. JACK is a professional-grade sound server daemon offering real-time, low-latency pathways for audio and MIDI data exchanges between various applications. This scenario is more common in professional or studio setups, where audio data might be routed between different applications or hardware devices.

5.1.12.6 TrackType

This enumeration is fundamentally used to differentiate between two different states of the audio tracks used for training and testing sessions, each serving a distinct purpose. The *Reference* value indicates that the track is a reference track. This is the original or standard track against which others are compared or evaluated. It is the original form of the audio track without applying any kind of alterations. The *Modified* value indicates that the track is a modified track. This track has been changed or altered in some way from the reference track. Changes could include effects added or alterations in frequency and amplitude, among others.

5.1.13 Tests

The key components of AudibleT are covered by unit testing and benchmarking test with the goal of ensuring the tool behaves functionally correctly and performs optimally. Both benchmarking and unit testing was implemented for classes such as DatabaseManager, FrequencyResponseHelper, BinomialHelper, and JackClientAudioEngine since these are considered the most critical parts of the software that shapes the behavior of the whole software and user experience. QtTest framework is utilized to write and perform the tests.

DatabaseManager, tasked with managing database operations, was subjected to speed tests for various operations like database setup, audio file handling, test session management, and trial handling. The performance of statistical queries, such as the average number of trials per session and the average response time per trial was also benchmarked. The intention was to locate potential bottlenecks and optimize database operations for enhanced performance. The *FrequencyResponseHelper* class was analyzed to confirm the accurate calculation of the frequency response for given coefficients. Similarly, the accuracy of the binomial coefficient calculations in the *BinomialHelper* class is also benchmarked and covered by unit testing.

Besides, for the *JackClientAudioEngine* class, which deals with audio processing and interaction with the JACK Audio Connection Kit, its performance is somewhat assessed by creating mock input and output buffers and running the audio callback function under different conditions. The intention of the conducted tests was to verify the effective functionality of audio callbacks and the application of audio filters to a certain extent. It is worth mentioning that this is a primary level of benchmarking for real-time audio processing as this doesn't necessarily emulate the real-life scenario. A more in-depth benchmarking for this class is discussed in chapter 6.1.

Beyond performance tests, unit tests for the *DatabaseManager* are employed to validate each database operation, including creation, manipulation, and statistical data retrieval. *FrequencyResponseHelper*, *BinomialHelper*, and *JackClientAudioEngine* underwent unit tests with the aim of making sure the tool behaves as expected.

6 Results and Discussion

This chapter presents the experimental tests carried out to measure the tool's real-time performance under different hardware and test environments. It analyzes and compares the results in section 6.1. A critical discussion is followed in detail in section 6.2 by examining the results in the context of the objectives of this project as well as the previous works in this field.

6.1 Results

This section presents the results of the real-time tool developed. Two benchmarking processes were followed to test the performance of the tool. The first is to run the benchmarking of the code execution of the critical part of the tool to get an overview of the execution of those methods. This helps find general performance bottlenecks in the code and improve them. This benchmarking is carried out by the QtTest [78] framework. Next, benchmarking the CPU load of the developed JACK External Client is carried out using different hardware and buffer sizes. It utilized the $jack\_cpu\_load()$ [79] function provided by JACK API. This provides a real-time estimate of how much load JACK is creating on the CPU. It's calculated by tracking how long it takes to carry out all necessary processing tasks within each given cycle. This is then compared to the total time available for these tasks, which is determined by the buffer size and sample rate. The result is presented as a percentage, indicating the proportion of available time that is being used. However, this returns the CPU load for all running clients, therefore the benchmarking is carried out ensuring only one client is active. This refers to one of the main aspects of this thesis. The sample rate used for benchmarking is fixed at 44100 Hz.

6.1.1 Test Hardware

Several general-purpose computers were utilized for tests, and one Raspberry Pi computer is also used to verify the usability of the tool in the embedded system. The specification of the hardware and environments are detailed as follows in table 1. For additional information about the hardwares, see listing. B.

6.1.2 Benchmarks

The real-time performance of the tool has been tested for different hardware devices with different buffer size, as shown in table 2, 3, 4, 5, 6 and 7.

Table 1: Test Hardwares

Configuration Type	**Apple Mac Studio**	**MSI Creator**	**Raspberry Pi 400**
Vendor	Apple	Micro-Star International Co., Ltd.	N/A
Model Name	Mac Studio	Creator 15 A10SDT (16V2.1)	Raspberry Pi 400 Rev 1.1
CPU/Chip	Apple M1 Max (Apple Silicon based on ARM64)	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz	Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
Total Number of Cores	10 (8 performance and 2 efficiency)	6	4
Memory	32 GB LPDDR5	16 GB	4 GB LPDDR4-3200
System Version	macOS 13.3.1	Ubuntu 22.04.1	Raspberry Pi OS with Desktop
Kernel Version	Darwin 22.4.0	5.19.0-42-generic	6.1
GCC Version	Apple clang version 14.0.3 (clang-1403.0.22.14.1)	11.3.0	10.1.0
JACK Version	jackdmp 1.9.22	jackdmp 1.9.20	jackdmp 1.9.17
CMake Version	3.26.3	3.26.20230513-g7f64e92	3.18.4

Table 2: Results from Different Hardware with Buffer Size of 64 Samples

Hardware	**Buffer Size**	**CPU Load (%) (Filter Off)**	**CPU Load (Local File)**	**CPU Load (JACK - Microphone/Others)**
Mac Studio	64	≈ 2.5	≈ 4.6	≈ 4.6
MSI	64	≈ 6.6	≈ 9.2	≈ 9.6
Raspberry Pi	64	≈ 3.1	≈ 4.0	≈ 5.6

Table 3: Results from Different Hardware with Buffer Size of 128 Samples

Hardware	**Buffer Size**	**CPU Load (%) (Filter Off)**	**CPU Load (Local File)**	**CPU Load (JACK - Microphone/Others)**
Mac Studio	128	≈ 1.5	≈ 3.2	≈ 3.4
MSI	128	≈ 3.7	≈ 6.8	≈ 7.5
Raspberry Pi	128	≈ 2.4	≈ 3.8	≈ 3.7

Table 4: Results from Different Hardware with Buffer Size of 256 Samples

Hardware	**Buffer Size**	**CPU Load (%) (Filter Off)**	**CPU Load (%) (Local File)**	**CPU Load (%) (JACK - Microphone/Others)**
Mac Studio	256	≈ 1.0	≈ 3.4	≈ 3.5
MSI	256	≈ 1.9	≈ 5.0	≈ 5.1
Raspberry Pi	256	≈ 1.5	≈ 3.0	≈ 2.9

Table 5: Results from Different Hardware with Buffer Size of 512 samples

Hardware	Buffer Size	CPU Load (%) (Filter Off)	CPU Load (%) (Local File)	CPU Load (%) (JACK - Microphone/Others)
Mac Studio	512	≈ 0.7	≈ 2.9	≈ 3.0
MSI	512	≈ 1.35	≈ 4.2	≈ 3.8
Raspberry Pi	512	≈ 1.1	≈ 2.4	≈ 2.3

Table 6: Results from Different Hardware with Buffer Size of 1024 Samples

Hardware	Buffer Size	CPU Load (%) (Filter Off)	CPU Load (%) (Local File)	CPU Load (%) (JACK - Microphone/Others)
Mac Studio	1024	≈ 0.5	≈ 2.6	≈ 2.6
MSI	1024	≈ 0.8	≈ 3.6	≈ 3.4
Raspberry Pi	1024	≈ 0.5	≈ 2.0	≈ 2.2

Table 7: Results from Different Hardware with Buffer Size of 2048 Samples

Hardware	Buffer Size	CPU Load (%) (Filter Off)	CPU Load (%) (Local File)	CPU Load (%) (JACK - Microphone/Others)
Mac Studio	2048	≈ 0.3	≈ 2.4	≈ 2.3
MSI	2048	≈ 0.5	≈ 3.2	≈ 3.1
Raspberry Pi	2048	≈ 0.3	≈ 2.0	≈ 1.9

A key observation is that as the buffer size increases, the CPU load decreases for all tested hardware. For the buffer size of 64 samples, the Apple Studio showed the lowest CPU load, with approximately 2.5% when the equalization filter is off, 4.6% when the filter is processing sample from the file from disk storage, and 4.6% when the filter is processing audio samples from JACK compatible input sources. This is followed by Raspberry Pi with 3.1%, 4.0%, and 5.6% for the same conditions, respectively. The MSI showed the highest CPU loads with 6.6%, 9.2%, and 9.6% for the same conditions, respectively. As the buffer size is doubled to 128 samples, all devices showed decreased CPU loads. Again, the Mac Studio showed the lowest CPU load, while the MSI showed the highest. This trend continued consistently as the buffer size was doubled successively to 256, 512, 1024, and 2048 samples. The Mac Studio consistently showed the lowest CPU load, followed by Raspberry Pi, and then MSI showed the highest. For the largest buffer size of 2048 samples, CPU loads dropped to less than 1% across all tested conditions for all hardware, indicating insignificant improvements in CPU efficiency. However, overall the test data proves the impact of the buffer size on the CPU load, with larger buffer sizes resulting in lower CPU loads because of the less frequent demand of real-time processing. Among the tested hardware, Mac Studio consistently showed the best performance in terms of CPU load, followed by Raspberry Pi, while MSI demonstrated the highest CPU load. However, it is also worth mentioning that these CPU load values are approximate and collected while the computer may or may not be running non-JACK processes. Besides, the values fluctuate consistently; therefore, an overall approximate average value is chosen for the benchmarking. This real-time performance benchmark reflects the performance of both the training and testing window of the tool. Other parts of the application, such as database CRUD (Create, Read, Update, Delete) operations and computing frequency response and filter coefficients, are benchmarked and profiled using Qt Test. Besides, the implemented unit testing suit also helps ensure the tool behaves functionally correctly in other UI areas, such as filter configuration settings, updating the equalizer graph, playing the correct track with respect to the option selected, and showing the correct statistical data through statistical analysis. However, even though integration and automated system testing was planned initially, it wasn't covered due to time constraints. Qt framework provides out of the box solution to set up integration and system testing that as well.

6.1.3 Report Generation and Export

The tool provides visual presentation of the test sessions through statistical analysis. And, the test sessions data can be exported into a CSV file. There are two separate export options, one exports the all session trials into a single CSV file and another option exports the trials from the selected session. The file contains the following fields:

- **SessionId**: The unique session identifier for the conducted test session.
- **SessionStartDate**: The start date of the testing session, a timestamp.
- **SessionEndDate**: The end date of the testing session, a timestamp.
- **SampleFileName**: Name of the audio file used on that particular testing session.
- **SessionEndDate**: The end date of the testing session, a timestamp.
- **SessionDuration**: Session duration in seconds.
- **TrialID**: The unique trial id for the trial within the session. Each trial has an unique id.
- **CorrectResponse**: The expected response of that trial.
- **ActualResponse**: The actual response of that trial listener provided.

- **CorrectResponse**: The expected response of that trial.
- **ResponseTime**: The time user taken in milliseconds to response for that particular trial.

6.2 Discussion

This section analyses the findings of this project and provides critical insight into how the objectives and research questions have been addressed.

One addtional project goal was to investigate the appropriate buffer scheme to balance latency and sample dropouts as illustrated in the Benchmark section 6.1.2, a suitable amount of tests carried out with different buffer sizes and sample rates, which goes further beyond the presented data at the development and the testing phase of the tool to find out the best and optimal performance point of the tool that performs in cheap hardware as good as the general consumer computers. Although the audio processing latency is not benchmarked as the real-time CPU load, a substantial amount of test sessions have taken place to avoid sample dropouts. Based on the lab results, it was found that a buffer size of 256 samples coupled with 44100 Hz performs well in general-purpose computers and doesn't show any sample dropouts while frequency response is altered and audio samples are processed in real time, which gives a latency of 5.8ms in theory and below 10ms which the preferred latency by the musicians [34]. However, in the cheap hardware, such as the Raspberry Pi, used to run tests, the tool performs quite similarly to the general-purpose computers, although it occasionally drops a few samples. In such circumstances, setting up a buffer size of 512 samples shows no sample dropouts when audio samples are processed through the filter, although latency is increased, and in most cases, the difference could be inaudible.

As a part of the system and test design, the project also focused on identifying suitable audio parameters for subjective testing and methods to perform the tests. The outcome of this was highly influenced by the background study conducted at the beginning of the project, where it was decided to implement ABX test design for JND to test linear distortion, such as subtle alterations of the frequency response, such as in the high-frequency region of microphones, phonograph cartridges used for digitizing LPs, loudspeakers, etc. However, it could also be interesting to expand the tool to non-linear distortions too with more sophisticated test designs. Moreover, AudibleT is currently limited to only shelving filters, although the ambition at the beginning of the project was to include FIR filters too. Hence, fig. 3 shows a delay signal in the tool design, is then required when FIR filter is used. Due to the time constraints and the project's scope getting more comprehensive, that could not be considered in the end. Besides, the frequency response generation approach was not trivial and works only for a particular filter design. This also limits the approach being applied to any other filter designs and can be overcome by designing filter with an impulse response and subsequently calculating the frequency response using the Fast Fourier Transform (FFT). Overall, AudibleT is designed and implemented so that it could be extensible and scalable as it could accommodate new filter types or even test types with minimum changes. The goal of such design decisions was made from the thinking of modularity and scalability. Efforts have been made to structure and maintain a clean, well-documented code base for the tool along the way. However, there's still room for a lot of improvements that can be done by refactoring the code, making each of the modules even smaller. Since the focus was to build a working prototype, such details were omitted during the development phase.

However, to make the tool compatible with different types of hardware and environment, cross-platform tools were chosen accordingly, such as Qt framework. It allowed us to build a program for different CPU architectures, such as x86-64 and ARM64 Cortex, as well as a relatively recently introduced custom Apple Silicon SoC based on ARM64 architecture from a single codebase. One of the primary goals of the project was to build a solution with wider compatibility, which was met by using Qt and JACK and implementing the algorithms and audio processing pipeline efficiently, leveraging the multiple cores of the modern CPUs, which was presented in Benchmark section 6.1.2, where it was noticed that even though Raspberry Pi is the weakest CPU in the comparison by a significant margin, the CPU load of the Raspberry Pi is relatively low while maintaining similar performance. However, the tool is built on top of JACK, which makes the tool limited to JACK-supported systems, although JACK is widely available on all major Linux distros, Windows and MacOS. Therefore, the tool can be ported into Windows too. While the tool is built considering real-time issues, and optimizations for different hardware, it is also worth mentioning that AudibleT is currently limited to shelving filters which are more computationally efficient than FIR filters. Implementing FIR filters and benchmarking under different testing conditions would also be interesting.

Moreover, for benchmarking the tool, only a specific JACK API function is utilized alongside the built-in Qt Test framework for code execution time and performance. JACK also provides another function *jackiodelay*, which provides the round trip latency measurement. That could be utilized for the latency measurement of the tool alongside more sophisticated tests to analyze the real-time performance of the filter and audio callback method. Besides, several other comparatively better third-party profiling tools are available for general profiling, memory graphs, CPU load for the process, or such, which could be employed to improve the tool's performance by running advanced analysis.

In terms of the user-friendly interface to perform the tests, the focus was to create a simple yet customizable training, testing, and statistics interface. But, user-friendliness is a subjective term and varies from person to person, based on one's taste, appeal, judgment, background knowledge of the tools being used, and so on. AudibleT is built with the aim of being usable by people from both technical and non-technical backgrounds. However, it also assumes that the interested parties in using this software possess sufficient knowledge of subjective audio testing. The interface is created with a combination of simple buttons, sliders, labels, and charts, avoiding complicity and best efforts made to make it easy to use. The software also allows visualizing the testing session reports generated encouraged by statistical analysis [3]. It allows listeners to see the confusion matrix of a particular test session's result as well as a binomial and inverse binomial graph. Besides, it shows the summary of all conducted sessions, such as average trials per session, the average time is taken to respond per trial, the average length in the duration of each testing session, and the average success rate per training session. While these might be useful for some listeners, for some, this might be inadequate. To accommodate that, this software also allows exporting the testing session data in CSV format, which can be used to generate more reports from other available third-party tools. However, one downside of the user interface is, it is not fully responsive to all screen sizes and resolutions a typical display would offer. According to the tests, the software runs well with a screen resolution of 1080p and above.

Furthermore, one of the initial ambitions was to get the software out to the real-world and, do usability testing, and gather test data. But, the scope of the project was limited to building the functional proof of concept. Consequently, the software has not yet undergone extensive user testing outside of the lab environment. Besides, the main goal was to offer a framework and functional prototype for wider hardware compatibility and community to be able to convert subjective testing into objective testing. To achieve that goal, it was required to study different areas such as psychoacoustics, subjective testings, digital signal processing, and most importantly, real-time aspects of the hardware and software, which the literature study reflects. It is also evident that the previous work from this field is limited to specific areas and unlike the scope of this tool, which tries to tie the abovementioned areas into one software. Consequently, AudibleT doesn't allow new customized tests like [40] Guineapig2, but AudibleT provides real-time filter coefficient changes in real-time while processing the audio as well in real-time, which is not available Guineapig2. Similarly, LisTEn [42] supports several test types, although one has to prepare the audio file using candidate codes before each test session and based on the tests type. In contrast, AudibleT testing sessions are fairly easy, as a listener can train using importing WAV files directly from their storage, as well as can use JACK compatible source such as a microphone or third-party audio player like Audacious [80] which can send audio to the JACK client. Listeners are also allowed to run testing sessions from their imported audio files. There's no preparation of files or filter coefficients required on AudibleT. Besides, while Salte [43] allows different types of tests using standard methodologies like MUSHRA, it focuses only on VR technology; on the other hand, AudibleT focuses more on hardware compatibility of general purpose computer and cheap hardware used on the embedded systems. It is also evident from the literature study that tools built dedicated to hard real-time systems such as Bela [47] would essentially perform better than the general purpose CPUs for real-time processing. However, the aim of AudibleT is to build a real-time system for versatile hardware and systems, and not limited to certain hardware or platforms to allow access to a wider community and adaptability. Therefore, it is also evident that AudibleT made an effort to improve certain areas of the existing tools, such as many of the previous tools do not offer proper statistical analysis of the test results while AudibleT does, and many of the tools are quite dated while AudibleT runs on the latest systems and development environments. At the same time AudibleT falls behind in some areas too, such as customizability of the tests, testing non-linear parameters with other standardized methods, more than 2 channels audio output, support spatial audio or VR technology.

In summary, this project has made effective efforts to bridge the gap by offering a comprehensive and versatile subjective audio testing tool designed for both general-purpose and low-cost hardware systems. While there are areas for improvement and expansion, AudibleT aligns real-time audio processing, user interface design, cross-platform compatibility, and statistical analysis into a single accessible platform. The tool is also made available to the wider community through open-sourcing the source code [81], which will allow interested communities to try the software and improve it.

7 Conclusion and Future Work

This chapter brings the conclusion to the thesis by providing a summary of the overall objectives and how they are met and shed some light into how this software can be improved further.

7.1 Conclusion

In this thesis, the AudibleT system is developed and introduced, which is a real-time ABX testing tool capable of running on general-purpose computers and also on resource-constrained hardware such as Raspberry Pi. It also allows users to conduct training sessions before running a testing session using audio files from their computers and any JACK-compatible input sources, such as microphones or third-party audio players, to allow listeners to familiarize themselves with the test and configurations. The software also allows equalizer coefficient changes in real-time on the UI thread with a minimum delay while processing the audio sample through the IIR filter in a separate worker thread to prevent the UI from becoming non-responsive at the time of processing audio. The software also passed the compatibility test for running on different hardware, operating systems, and Raspberry Pi, followed by extended CPU load testing with a variable sample buffer size. AudibleT also offers visualization of the test results through various reports generated via statistical analysis and provides the option to export the test session data in CSV.

7.2 Proposed Future Work

During the literature study and development of the project, several interesting areas were recognized to make further improvements to the tool. Some of them are as follows:

- There could be a panel introduced for test supervisors, who could potentially run and manage a listening test session for a group of listeners. Only supervisors will be able to configure the tests, while the listeners will only hear and use the buttons to provide their responses.
- Introduce FIR filter using Parks–McClellan filter design algorithm to generate filter coefficient vectors that apply subtle frequency response aberrations. It also allows for the flexible design to simulate loudspeakers with limited bandwidth.
- Non-linear distortions testing can be introduced.

- PipeWire provides a JACK-compatible API layer that allows running JACK client on PipeWire. Running the software with PipeWire and benchmarking the overall performance using the PipeWire API can also be carried out.
- The unit testing suit can be improved further, and integration, and system testing can be introduced to cover more edge cases of the performance and resource usage bottleneck.
- The code base can be improved by refactoring, avoiding repetition of codes, and utilizing various design paradigms.

References

[1] D. Clark, “High-Resolution Subjective Testing Using a Double-Blind Comparator,” *Jou. Audio Eng. Soc.*, vol. 30, pp. 330–338, May 1982.

[2] S. P. Lipshitz and J. Vanderkooy, “The Great Debate: Subjective Evaluation,” *Jou. Audio Eng. Soc.*, vol. 29, pp. 482–491, Aug 1981.

[3] J. Boley and M. Lester, “Statistical Analysis of abx Results Using Signal Detection Theory,” 127^{th} AES Conv. Oct 1, 2009, New York, USA.

[4] N. C. Otto, “Listening Test Methods for Automotive Sound Quality,” 103^{rd} AES Conv. Sep 26-29, 1997, New York, USA.

[5] “JACK.” `https://jackaudio.org/`.

[6] “PipeWire.” `https://pipewire.org/`.

[7] “Memory Access Ordering - An Introduction.” `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/memory-access-ordering---an-introduction`.

[8] “Ubuntu Desktop.” `https://ubuntu.com/download/desktop`.

[9] “MacOS Ventura.” `https://www.apple.com/macos/ventura/`.

[10] “Raspberry Pi OS.” `https://www.raspberrypi.com/software/`.

[11] W. A. Yost, “Psychoacoustics : A Brief Historical Overview from Pythagoras to Helmholtz to Fletcher to Green and Swets , a Centuries-Long historical Overview of Psychoacoustics,” *Acoustics Today*, vol. 11, 2015.

[12] “HP’s First Product: The 200A.” `https://www.hewlettpackardhistory.com/item/hewlett-packards-first-product/`.

[13] “The 200AB and 200CD: Always Improving.” `https://www.hewlettpackardhistory.com/item/always-improving/`.

[14] J. Deery, “The ’Real’ History of Real-Time Spectrum Analyzers - a 50-Year Trip Down Memory Lane,” *Sound & Vibration*, vol. 41, pp. 54–59, Jan 2007.

[15] V. R. Melchior, "High-Resolution Audio: A History and Perspective," *Jou. Audio Eng. Soc.*, vol. 67, pp. 246–257, May 2019.

[16] K. Brandenburg, "MP3 and AAC Explained," 17^{th} Int. Conf. on High-Quality Audio Coding, Sep 2-5, 1999, Florence, Italy.

[17] "Perceptual Objective Listening Quality Assessment(POLQA)." `http://www.polqa.info/`.

[18] "ITU-R BS.1116." `https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.1116-1-199710-S!!PDF-E.pdf`.

[19] "ITU-T P.800." `https://www.itu.int/rec/T-REC-P.800-199608-I`.

[20] M. Long, "3 - Human Perception and Reaction to Sound," in *Architectural Acoustics (Second Edition)* (M. Long, ed.), Boston: Academic Press, 2nd ed., 2014.

[21] M. Welvaert and Y. Rosseel, "On The Definition of Signal-To-Noise Ratio and Contrast-To-Noise Ratio for fMRI Data," *PLOS ONE*, vol. 8, pp. 1–10, Nov 2013.

[22] R. Schatz, S. Egger, and K. Masuch, "The Impact of Test Duration on User Fatigue and Reliability of Subjective Quality Ratings," *Jou. Audio Eng. Soc.*, vol. 60, pp. 63–73, Jan 2012.

[23] C. Ranti, W. Jones, A. Klin, and S. Shultz, "Blink rate patterns provide a reliable measure of individual engagement with scene content," *Scientific Reports*, vol. 10, p. 8267, May 2020.

[24] J. Liebetrau, J. Nowak, T. Sporer, M. Krause, M. Rekitt, and S. Schneider, "Paired Comparison as a Method for Measuring Emotions," 135^{rd} AES Conv. Oct 17-20, 2013, New York City, USA.

[25] "EBU 3276." `https://tech.ebu.ch/docs/tech/tech3276.pdf`.

[26] R. King, B. Leonard, S. Bremner, and G. Sikora, "Consistency of High Frequency Preference Among Expert Listeners," 136^{th} AES Conv. Apr 26-29, 2014, Berlin, Germany.

[27] G. Di Leo and F. Sardanelli, "Statistical Significance: p value, 0.05 threshold, and Applications to Radiomics—Reasons for a conservative Approach," *European Radiology Experimental*, vol. 4, p. 18, Mar 2020.

[28] T. Lund, A. Mäkivirta, and S. Naghian, "Time for Slow Listening," *Jou. Audio Eng. Soc.*, vol. 67, pp. 636–640, Sep 2019.

[29] M. A. Cohen, D. C. Dennett, and N. Kanwisher, "What is the Bandwidth of Perceptual Experience?," *Trends in Cognitive Sciences*, vol. 20, pp. 324–335, May 2016.

[30] A. Brook, "How Good is Too Good? Audience Evaluation of Mixes Composed of Edited Audio Versus Unedited Audio.," *Jou. Audio Eng. Soc.*, 153^{rd} AES Conv. Oct 19-20, 2022, New York, USA.

[31] F. E. Toole, "Listening Tests-Turning Opinion into Fact," *Jou. Audio Eng. Soc.*, vol. 30, pp. 431–445, Jun 1982.

[32] S. Zielinski, F. Rumsey, and S. Bech, "On Some Biases Encountered in Modern Audio Quality Listening Tests-A Review," *Jou. Audio Eng. Soc.*, vol. 56, pp. 427–451, Jun 2008.

[33] "MUSHRA." `https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.1534-3-201510-I!!PDF-E.pdf`.

[34] R. H. Jack, T. Stockman, and A. McPherson, "Effect of Latency on Performer Interaction and Subjective Quality Assessment of a Digital Musical Instrument," in *Proc. of the Audio Mostly 2016*, AM '16, (New York, USA), p. 116–123, Association for Computing Machinery, 2016.

[35] "Bela." `https://bela.io/`.

[36] E. R. Geddes and L. W. Lee, "Auditory Perception of Nonlinear Distortion - Theory," 115^{th} AES Conv. Oct 10-13, 2003, New York, USA.

[37] B. C. J. Moore and C.-T. Tan, "Development and Validation of a Method for Predicting the Perceived Naturalness of Sounds Subjected to Spectral Distortion," *Jou. Audio Eng. Soc.*, vol. 52, pp. 900–914, Sep 2004.

[38] H. Suzuki, S. Morita, and T. Shindo, "On the Perception of Phase Distortion," *Jou. Audio Eng. Soc.*, vol. 28, pp. 570–574, Sep 1980.

[39] A. Gabrielsson, B. Hagerman, T. Bech-Kristensen, and G. Lundberg, "Perceived Sound Quality of Reproductions with Different Frequency Responses and Sound Levels," *The Jou. of the Acoustical Society of America*, vol. 88, pp. 1359–1366, Sep 1990.

[40] J. Hynninen, "A Software-based System for Listening Tests," Master's thesis, Helsinki University of Technology, May 2001. [Online]. Available: `http://research.spa.aalto.fi/publications/theses/hynninen_mst.pdf`.

[41] S. Ciba, A. Wlodarski, and H. Maempel, “Whisper – A New Tool for Performing Listening Tests,” 126th AES Conv. May 7–10, 2009, Munich, Germany.

[42] M. Schäfer, C. Schnelling, B. Geiser, and P. Vary, “A Listening Test Environment for Subjective Assessment of Speech and Audio Signal Processing Algorithms,” *Studientexte zur Sprachkommunikation: Elektronische Sprachsignalverarbeitung 2011*, pp. 237–244, 2011.

[43] D. Johnston, B. Tsui, and G. Kearney, “Salte Pt. 1: A Virtual Reality Tool for Streamlined and Standardized Spatial Audio Listening Tests,” 147th AES Conv. Oct 16–19, 2019, New York, USA.

[44] T. Rudzki, C. Earnshaw, D. Murphy, and G. Kearney, “Salte pt. 2: On the Design of the Salte Audio Rendering Engine for Spatial Audio Listening Tests in VR,” 147th AES Conv. Oct 16–19, 2019, New York, USA.

[45] A. Mourgela, J. Reiss, and T. R. Agus, “Investigation of a Real-Time Hearing Loss Simulation for Use in Audio Production,” 149th AES Conv. Oct 27-30, 2020, Online Con.

[46] S. Gorzynski, N. Kaplanis, and S. Bech, “A Flexible Software Tool for Perceptual Evaluation of Audio Material and VR Environments,” 149th AES Conv. Oct 27-30, 2020, Online Conf.

[47] A. Mcpherson and V. Zappi, “An Environment for Submillisecond-Latency Audio and Sensor Processing on Beaglebone Black,” 138th AES Conv. May 7–10, 2015, Warsaw, Poland.

[48] “BeagleBone Black.” `https://beagleboard.org/black`.

[49] “Xenomai.” `https://xenomai.org/documentation/xenomai-2.2/html/xenomai/`.

[50] H. Langer and R. Manzke, “Embedded Multichannel Linux Audiosystem for Musical Applications,” *Jou. Audio Eng. Soc.*, vol. 66, pp. 286–291, Apr 2018.

[51] H. von Coler, “A JACK-Based Application for Spectro-Spatial Additive Synthesis,” Proc. 17th Linux Audio Conference (LAC-19), Mar 23-26, 2019, Stanford University, CA, USA.

[52] “Panoramix.” `https://forum.ircam.fr/media/uploads/forumnet-legacy/2016/12/Panoramix-QuickStart2.pdf`.

[53] C. Kuhr and A. Carôt, “A JACK Sound Server Backend to Synchronize to an IEEE 1722 AVTP Media Clock Stream,” Proc. 17th Linux Audio Conference (LAC-19), Mar 23-26, 2019, Stanford University, CA, USA.

[54] W. Taymans, "Pipewire: A Low-level Multimedia Subsystem," Proc. 18th Linux Audio Conference (LAC-20), Nov 25-27, 2020, Université de Bordeaux, France.

[55] "PulseAudio." `https://www.freedesktop.org/wiki/Software/PulseAudio/`.

[56] "PipeWire." `https://docs.pipewire.org/page_overview.html`.

[57] L. Vignati, S. Zambon, and L. Turchet, "A Comparison of Real-time Linux-based Architectures for Embedded Musical Applications," *Jou. Audio Eng. Soc.*, vol. 70, pp. 83–93, Jan 2022.

[58] H. L. Muller, "Designing Multithreaded and Multicore Audio Systems," 24th Conference: The Ins & Outs of Audio , Jun, 2011, Bristol, UK.

[59] Y. Wang, X. Zhu, and Q. Fu, "A Low Latency Multichannel Audio Processing Evaluation Platform," 132nd AES Conv. Apr 26-29, 2012, Budapest, Hungary.

[60] L. Leventhal, "Type 1 and Type 2 Errors in The Statistical Analysis of Listening Tests," *Jou. Audio Eng. Soc.*, vol. 34, pp. 437–453, Jun 1986.

[61] M. Srednicki, "A Bayesian Analysis of A-B Listening Tests," *Jou. Audio Eng. Soc.*, vol. 36, pp. 143–146, Mar 1988.

[62] J. vom Brocke, A. Hevner, and A. Maedche, "Introduction to design science research," in *Design Science Research. Cases* (J. vom Brocke, A. Hevner, and A. Maedche, eds.), ch. 1, Cham: Springer, 2020.

[63] S. Ross and G. Morrison, "Experimental research methods," in *Handbook of Research on Educational Communications and Technology* (D. Jonassen, ed.), pp. 1021–1043, Mahwah: Lawrence Erlbaum Associates, 2nd ed., 2004.

[64] S. Crowe, K. Cresswell, A. Robertson, G. Huby, A. Avery, and A. Sheikh, "The Case Study Approach," *BMC Medical Research Methodology*, vol. 11, p. 100, Jun 2011.

[65] "QT." `https://www.qt.io/`.

[66] "C++." `https://cplusplus.com/`.

[67] Alex Norman, "JACKCPP." `https://github.com/x37v/jackcpp`.

[68] Adam Stark, "AudioFile." `https://github.com/adamstark/AudioFile`.

[69] "Signals & Slots." https://doc.qt.io/qt-6/signalsandslots.html.

[70] "Model/View Programming." https://doc.qt.io/qt-6/model-view-programming.html.

[71] "Qt SQL." https://doc.qt.io/qt-5/qtsql-index.html.

[72] "JACK Client Functions." https://jackaudio.org/api/group__ClientFunctions.html.

[73] S. Mitra and S. Mitra, *Digital Signal Processing: A Computer-based Approach*. McGraw-Hill, 2011.

[74] V. Välimäki and J. D. Reiss, "All About Audio Equalization: Solutions and Frontiers," *Applied Sciences*, vol. 6, no. 5, 2016.

[75] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. USA: Prentice Hall Press, 3rd ed., 2009.

[76] L. R. Varshney and J. Z. Sun, "Why Do We Perceive Logarithmically?," *Significance*, vol. 10, no. 1, pp. 28–31, 2013.

[77] J. Stirling, *The Differential Method: Or, A Treatise Concerning Summation and Interpolation of Infinite Series. By James Stirling ... Translated Into English, with the Author's Approbation, by Francis Holliday*. E. Cave, 1749.

[78] "Qt Test." https://doc.qt.io/qt-6/qttest-index.html.

[79] "$jack\_cpu\_load()$." https://jackaudio.org/api/group__ServerControl.html.

[80] "Audacious." https://audacious-media-player.org/.

[81] "AudibleT." https://github.com/priyonto/AudibleT.

Appendices

Appendix A Frequency Response Helper Class Source

```
std::vector<double> FrequencyResponseHelper::logspace(double start, double end,
    int num_points) {
    std::vector<double> result;

    // Convert start and end to logarithmic scale
    double start_log10 = std::log10(start);
    double end_log10 = std::log10(end);

    // Compute the step size in the logarithmic scale
    double step_log10 = (end_log10 - start_log10) / (num_points - 1);
    for (int i = 0; i < num_points; ++i) {
        // Compute the frequency in the logarithmic scale and convert it back to
    linear scale
        double frequnecy = std::pow(10, start_log10 + i * step_log10);
        if (frequnecy < start)
            continue;
        result.push_back(frequnecy);
    }
    return result;
}
```

Listing 1: Method to create a logarithmically spaced vector

```cpp
std::vector<double> FrequencyResponseHelper::calculate_frequency_response(
    const std::vector<double> &B,
    const std::vector<double> &A,
    double sample_rate,
    const std::vector<double> &frequencies
    ) {
    std::vector<double> gain_db;
    // Calculate the sampling period
    double T = 1.0 / sample_rate;
    // Define a complex number j
    std::complex<double> j(0, 1);

    for (double f : frequencies) {
        // Calculate the complex exponentials for the current frequency
        std::complex<double> e_neg_j2pi_fT = std::exp(-j * 2.0 * M_PI * f * T);
        std::complex<double> e_neg_j4pi_fT = std::exp(-j * 4.0 * M_PI * f * T);

        // Compute the frequency response H(f) for the current frequency
        std::complex<double> H_f = (B[0] + B[1] * e_neg_j2pi_fT + B[2] *
   e_neg_j4pi_fT) / (A[0] + A[1] * e_neg_j2pi_fT + A[2] * e_neg_j4pi_fT);
        // Calculate the magnitude of H(f)
        double magnitude = std::abs(H_f);
        // Convert the magnitude to decibels
        double gain = 20 * std::log10(magnitude);
        gain_db.push_back(gain);
    }

    return gain_db;
}
```

Listing 2: Method to calculate the frequency response of a filter

Appendix B Additional Information of Test Hardware

B.1 Apple Mac Studio

Output Device: Built-in

Output Channel: 2

Transport: Built-in

Target: arm64-apple-darwin22.4.0

Thread Model: POSIX

B.2 MSI Creator

CPU Size: 3762 MHz

CPU Capacity: 5 GHz

CPU Clock: 100 MHz

Audio Devices:

- Product: TU116 High Definition Audio Controller
  Vendor: NVIDIA Corporation
  Width: 32 bits
  Clock: 33 MHz

- Product: Comet Lake PCH cAVS
  Vendor: Intel Corporation
  Width: 64 bits
  Clock: 33 MHz

B.3 Raspberry Pi 400

Hardware: BCM2835

Debian Version: 11 (bullseye)

Audio Devices:

- card 0: vc4hdmi0 [vc4hdmi0], device 0: MAI PCM i2shifi0 [MAI PCM i2shifi0]

- card 1: Device [USB PnP Sound Device], device 0: USB Audio [USB Audio]

- card 2: vc4hdmi1 [vc4hdmi1], device 0: MAI PCM i2shifi0 [MAI PCM i2shifi0]