UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Real time acoustic analysis and correction

Document:

Report

Author:

Alexandr Ramos Sundukov

Director:

Albino Nogueiras Rodríguez

Degree:

Bachelor's degree in Audiovisual Systems Engineering

Examintaion session:

Spring, 2025

BACHELOR FINAL THESIS

Abstract

This project addresses a common challenge faced by audio engineers and technicians: adjusting sound systems based on the acoustic properties of a given environment. While there are existing commercial solutions that offer advanced analysis tools, this work proposes the development of a custom, software-based alternative using open-source technologies.

The proposed solution is implemented in Python and designed to perform real-time acoustic analysis and correction. It integrates several scientific libraries, including NumPy and SciPy for signal processing, and uses Sounddevice for real-time audio input and output. The program interacts with the environment through a basic hardware setup consisting of a microphone, a speaker, and a sound card.

A graphical user interface is developed using Tkinter and Matplotlib to ensure usability and ease of interaction. This graphical user interface allows users to configure audio parameters, select input channels, visualize data, and monitor system behavior in real time. Features include Fourier transformation, filter-based processing, and measurement tools such as delay, phase, and frequency response analysis.

Throughout the development, special attention is given to flexibility and accessibility. The software is intended to run on standard hardware under Linux (specifically Ubuntu 22.04.5 LTS), making it usable in both professional and home studio environments.

Finally, the project reflects on the difficulties encountered during development, acknowledging that not all objectives have been fully achieved. It discusses the main technical and practical challenges, and offers constructive criticism to guide future iterations. Despite these limitations, the result is a solid foundation on which to build. While the tool is not yet fully functional, it provides a working base from which the missing features can be implemented and the existing ones improved.

Índex

Abstract iii

List of figures vi

List of tables viii

Abbreviations ix

1 Introduction 1

1.1 Context . . . 1
1.2 Objectives . . . 2
1.3 Project Structure . . . 2

2 Background and/or status of the matter 5

2.1 Smaart . . . 5
2.2 Dirac Live . . . 7
2.3 Trinnov . . . 8
2.4 Other Solutions . . . 9

3 Methodology, consideration and decision on alternative solutions 11

4 Development of the chosen solutions 13

4.1 Graphic interface . . . 13
4.2 Signal Path . . . 15
4.3 Settings page . . . 17
4.3.1 Device Settings . . . 17
4.3.2 Audio Settings . . . 17
4.4 Acoustics analysis . . . 18
4.4.1 Spectrogram (FT) . . . 20
4.4.2 RTA . . . 22
4.4.3 Delay . . . 25
4.5 Acoustic correction . . . 27

4.5.1 Bypass . . . 28
4.5.2 31 Bars . . . 29
4.6 Others . . . 30

5 Results **31**
5.1 Final tests . . . 32

6 Conclusions **51**

Bibliography **53**

Annex 1: Budget **I**

List of figures

2.1 Notebook using Smaart, where we can see some of the tools it includes. The notebook is connected to the EVO 8 (a USB interface that acts as an external sound card). 6
2.2 Screenshot of Dirac Live software. 7
2.3 Software used to configure and control Trinnov hardware processors. 8
2.4 Commercial set including a hardware processor, a proprietary four-capsule microphone, and a desktop remote control. 9

3.1 Home setup showing the sound card, the speaker, and the microphone. 12

4.1 GUI window schematic 14
4.2 Basic representation of the required signal path 16
4.3 Stream management 16
4.4 Device Settings page showing a red message because the user selected an invalid configuration. 17
4.5 Audio Settings page showing a green message. 18
4.6 Frequency Resolution as a Function of Time Window 20
4.7 Diagram of the architecture of the FT page 21
4.8 Analysis window - FT page. 22
4.9 Image of the front panel of the DBX 231s [25a], where we can observe that the center frequency bands are the same as those defined in IEC 61260. 23
4.10 Diagram of the architecture of the RTA page 24
4.11 Second-Order Sections for IIR filters with their parameters 24
4.12 Root Mean Square to calculate energy from filtered signal 24
4.13 Analysis window - RTA page. 25
4.14 Diagram of the architecture of the Delay page 26
4.15 Analysis window - Delay page. 27
4.16 Correction window - DSP. 28
4.17 Diagram of the architecture of the Bypass module 29
4.18 Diagram of the architecture of the EQ module 29

5.1 Microphone height relative to the seat 33
5.2 Microphone height = 1.1m 34

5.3 Microphone position relative to the speaker . . . 35
5.4 Horizontal distance between speaker and microphone . . . 36
5.5 Distance between microphone and speaker . . . 37
5.6 All equipment set up . . . 37
5.7 System connection overview . . . 38
5.8 Device Settings Window recognizing EVO 4 soundcard . . . 39
5.9 First RTA measurement over the External Input . . . 39
5.10 Audio Settings Window after changes . . . 40
5.11 Second RTA measurement over the External Input . . . 40
5.12 First Delay Measurement . . . 41
5.13 Second Delay Measurement . . . 41
5.14 FT Measurement Without Averaging Applied . . . 42
5.15 FT Measurement With Time Averaging Applied . . . 43
5.16 FT Measurement with Time and Frequency Averaging Applied . . . 43
5.17 X32 RTA Tool with Untreated Input from System Signal . . . 44
5.18 Reference RTA values, saved to apply in the Correction Window . . . 45
5.19 DSP window with EQ values applied from RTA analysis . . . 45
5.20 X32 RTA tool with Treated Input form System Signal . . . 46
5.21 RTA Results After Activating RTA+C EQ . . . 47
5.22 Configuring the X32 EQ tool with parameters obtained from RTA+C . . . 48
5.23 Input from System with X32 EQ treatment, visualized using X32 RTA tool . . . 48
5.24 RTA page, using music instead of pink noise . . . 49
5.25 FT page, using music instead of pink noise . . . 50

List of tables

4.1 Center frequencies for true 1/3 octave bands and IEC 61260 bands 23

List of abbreviations

UPC Universitat Politècnica de Catalunya

ESEIAAT Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa [UPC25].

RTA Real Time Analysis

SMAART System Measurement Acoustic Analysis Real-time Tool

FT Fourier Transform

FFT Fast Fourier Transform

DFT Discrete Fourier Transform

RFFT Real-valued Fast Fourier Transform

IEC International Electrotechnical Commision

RMS Root Mean Square

IIR Infinite Impulse Response

FIR Finite Impulse Response

GUI Graphical User Interface

OS Operating System

RT60 Reverberation time to decay 60 dB.

HI-FI High fidelity

DSP Digital Signal Processor

EQ Equalization or Equalizer

RTA+C Real Time Analysis plus Correction

DAW Digital Audio Workstation

USB Universal Serial Bus

Chapter 1

Introduction

This chapter presents the objectives and context of the project, which aims to develop a software-based system capable of performing real-time acoustic analysis and applying corrective digital processing to enhance sound quality in specific environments.

1.1 Context

To understand the proposed solution, we first need to understand what the problem is:

In any space where a sound system is present, there is an environment in which this system operates, as well as an area where the listeners are located. This means that the quality of the sound system and the acoustics of the environment where it is placed function together as a single system that directly affects the sound quality perceived by the listener.

- **About the Sound System:** It is considered that the human ear can perceive sound frequencies between 20 Hz and 20 kHz. The sound system acts as the source that plays all of these frequencies, and one thing the acoustic environment cannot do is add frequencies that were not already present (assuming the acoustic environment is a fully passive system—if there are other sound sources, they are not considered here). Therefore, a basic requirement for a high-quality sound system is the ability to reproduce the entire audible range and to do so with the flattest possible frequency response (in other words, equal fidelity across the whole range).

  Also, these systems are split across different speakers, which means the listener perceives multiple sound sources. It is considered important to keep all these sources in phase to ensure better sound quality—something that can sometimes be difficult to achieve.

- **About Acoustic Environment:** The system through which sound travels from the speaker to the listener can be extremely complex. Numerous acoustic phenomena can alter the perceived sound—and sometimes, these phenomena can completely ruin the listening experience. Some of the most important ones include: absorption, reflections, and diffractions, which can lead to reverberation, frequency boosts, cancellations, and resonances. Structural vibrations in the environment can also affect all of these phenomena.

Also, a very important factor is that the environment can be constantly changing, which directly affects these phenomena. For example, a window that is open or closed has a very different reflection and absorption coefficient, which directly impacts reflections and reverberation. Likewise, whether a room is empty or filled with people makes a difference—since the human body and clothing absorb sound. For this reason, in professional high-budget theaters and auditoriums, the seating is carefully designed to have a similar absorption coefficient whether or not someone is sitting in the seat. This helps ensure a stable, controlled, and consistent acoustic environment regardless of audience size or configuration. Unfortunately, such solutions are usually very expensive and not always feasible to implement.

Traditionally, achieving a good sound experience required both a high-quality sound system and a well-designed or acoustically treated environment. Nowadays, this is still a best practice to achieve the highest sound quality. However, in recent years, digital technology has also offered new solutions that can further improve the performance of such systems—and these are the kinds of solutions that will be investigated and developed in this project.

1.2 Objectives

The main objective is to develop a custom software-based solution to analyze audio systems and process the signal that will be played through them, in such a way that the processed signal counteracts the system's issues—improving the listener's experience in live situations.

This program must be usable by someone who did not participate in its development—not necessarily easy to use, but also not requiring expert-level knowledge.

And also, it is desirable to design the software in a modular and extensible way, allowing future improvements or the integration of new tools without requiring major changes to the existing structure. This will ensure the project remains scalable and adaptable to a wide range of use cases or evolving technical needs.

1.3 Project Structure

The project is divided into the following five chapters:

- **Background and/or status of the matter**, where similar solutions will be investigated and their strengths and weaknesses analyzed, in order to guide the development of a useful and effective solution.
- **Methodology, consideration and decision on alternative solutions**, where the characteristics of the final solution will be defined based on the previously analyzed alternatives.
- **Development of the choosen solution**, where the decisions made during the development process will be explained in detail, along with a full description of the implementation.

- **Results**, where the developed solution will be described and tested in a real-world scenario.
- **Conclusion**, where will be discussed the final results, and will be plantejat a possible future for these project.

Also, an **Annex 1: Budget** is included, where the estimated cost of all the components and resources required for the project will be discussed.

Chapter 2

Background and/or status of the matter

Nowadays, there are many solutions that can fit to solve our problem. Some are very expensive, and others have shortcomings. In this chapter, we will have a look at some of the most popular solutions.

2.1 Smaart

Smaart, an acronym for *System Measurement Acoustic Analysis Real-time Tool* [25f], is a software-based solution commercialized by Rational Acoustics. It is probably the most used and well-known solution for professional acoustic analysis, used in big venues, concert halls, stadiums, touring productions, as well as in professional audio studios and speaker development laboratories. Common uses are:

- **Speaker Alignment:** When we have multiple sound sources, this software helps us find the phase and delay between them. For example, it can be used to find the time and phase alignment between a subwoofer and a full-range speaker.
- **RTA, Frequency and Phase Response** used to view live spectrograms, phase deviation, or energy in frequency bands. One example of use is identifying resonances at specific frequencies.
- **Coherence Analysis** to evaluate the quality of the measured data. A common use is to detect reflections and background noise.
- **Delay Time** between different sources or signals. Widely used to synchronize different elements of the system.
- **Room and architectural acoustics** to identify the frequency and phase response of a room, as well as reverberation and echoes.

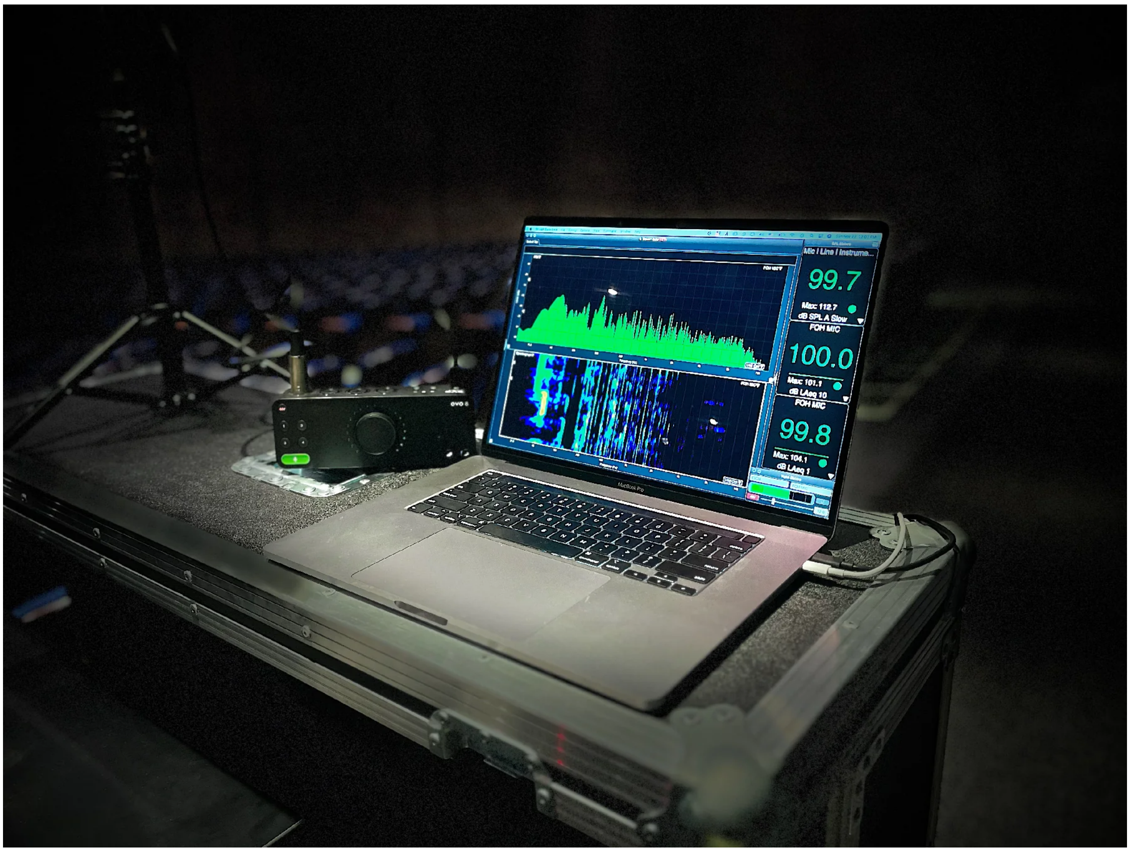

Figure 2.1: Notebook using Smaart, where we can see some of the tools it includes. The notebook is connected to the EVO 8 (a USB interface that acts as an external sound card).

The strongest points of this program are:

- **Flexibility:** As a software-based solution, it can run on any Windows or Mac computer (meeting the minimum required specs), and can be used with most external audio sound cards, allowing the connection of unlimited types of microphones or direct signals.

- **More than one channel:** This software can analyze and display information from more than one input channel at the same time, allowing comparisons between different channels. This is used to compare an original signal with the signal captured by a microphone inside a room with a sound system, helping to detect room acoustics or sound system issues. Another common use is to measure the sound in different places of the same room simultaneously.

- **Widely used:** It is very common to see professionals in the sector using this software, or at least being familiar with it. It has become a kind of standard, which leads other companies to ensure maximum compatibility with it. For example, Audix makes the Audix TM-1 Plus microphone [And23], which includes a file that can be imported into SMAART to apply microphone correction during analysis.

On the other hand, it requires a license, external hardware such as sound cards and microphones, and it does not have any correction capabilities—only analysis.

2.2 Dirac Live

Dirac Live [25b] is a software-based solution focused on room correction in home environments, aimed at users who want to improve the sound quality of their home theater systems or other audio setups without applying structural acoustic treatments. In many cases, it's not possible to perform physical acoustic modifications at home, so this software helps address such limitations by digitally correcting the room's response or enhancing the speaker performance.

Since it includes a correction processor, Dirac requires a dedicated computer to run continuously and process the audio being played through the system. Users can either use their own PC for this purpose or purchase a dedicated hardware unit provided by the company.

Additionally, setting up this solution requires a measurement microphone connected to the system, which captures the room and speaker response using excitation signals generated by the program itself. Dirac recommends specific microphones for this task and allows the import of calibration files to correct for minor microphone imperfections. Even so, it is acceptable to use other microphones as long as they come with measurement specifications, allowing for reasonably accurate results.

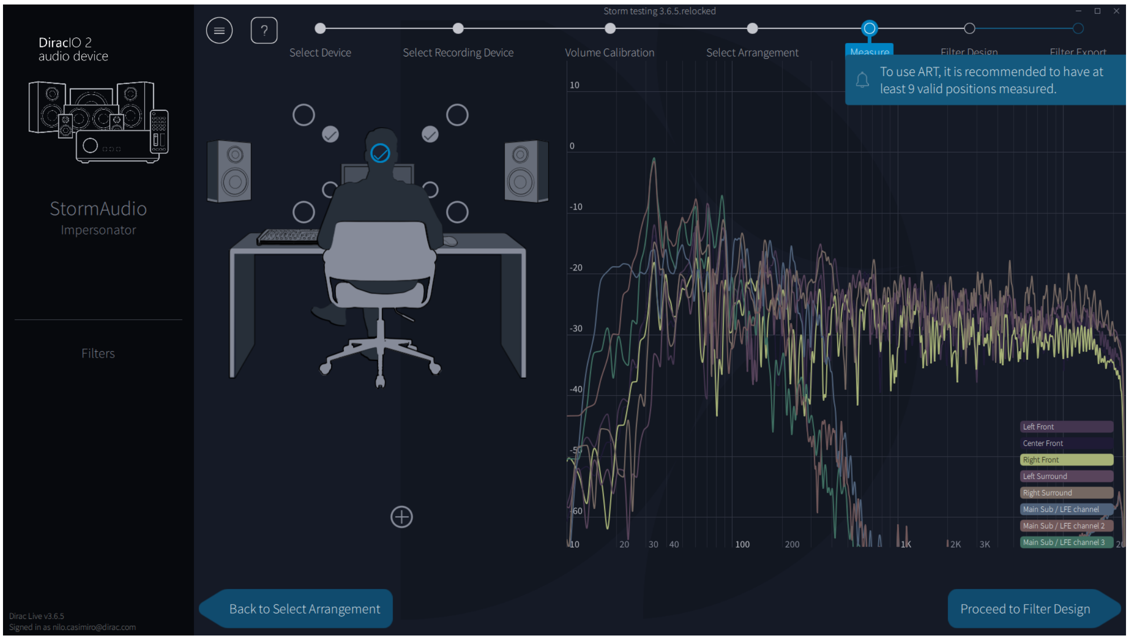

Figure 2.2: Screenshot of Dirac Live software.

As with any other solution, it has its downsides and upsides:

- On one hand, the biggest downside of this solution, in my opinion, is that it is heavily focused on users with little or no technical knowledge. While this is great because it makes the software easy to use and more accessible to any enthusiast, it also makes the system very opaque. You cannot know exactly what kind of correction is being applied or what analysis results the correction is based on. It provides very little information about what it is actually doing.
- On the other hand, the biggest upside is its broad compatibility with many different multichannel

system configurations, such as Stereo, LCR, Mono, 5.1, 6.0, 7.1, 5.1.2, and many more. Additionally, it can run as a standalone program, as a plugin within a DAW (*Digital Audio Workstation*), or even on dedicated hardware. Finally, it is very easy to use. All of these features make it a highly flexible and appealing solution.

To use Dirac Live, it is necessary to purchase a license. The company offers different license tiers that enable or restrict certain features, allowing the price to be adapted to different use cases and user needs.

2.3 Trinnov

Trinnov[Des25] is an expensive hardware-based solution with a complex correction module, aimed at high-end professional studios and premium home theaters. Unlike software-only solutions such as Dirac Live, Trinnov provides dedicated processing units that integrate directly into the audio signal chain. These units handle tasks such as room correction, speaker alignment, and advanced phase and delay compensation in real time.

Figure 2.3: Software used to configure and control Trinnov hardware processors.

An important particularity of this system is that it requires a specific and complex microphone provided by the same manufacturer. This microphone includes four measurement capsules arranged in a precise spatial configuration, allowing the system to determine the direction of incoming sound in a 3D space. This spatial information is used to correct complex acoustic issues and to create virtual sound sources, enhancing the immersive characteristics of the sound system within a specific room.

Figure 2.4: Commercial set including a hardware processor, a proprietary four-capsule microphone, and a desktop remote control.

It is also highly flexible, offering a wide range of options and detailed control. This solution is considered one of the most precise and reliable systems for digitally improving the acoustics of a room.

On the downside, Trinnov is significantly more expensive than most alternatives, as it relies on dedicated hardware and proprietary measurement tools. This cost places it outside the range of many semi-professional or enthusiast users.

2.4 Other Solutions

There are many more solutions available, but most of them are similar or offer the same tools with a similar workflow. However, I consider the following ones particularly worth mentioning:

- **REW** (Room EQ Wizard)[Mul24] is a free software-based solution that competes directly with **SMAART**, offering very similar features. However, it is more focused on non-live situations, such as studio or home audio analysis.
- **SoundID Reference** from the company Sonarworks [25g] is a software-based solution similar to **Dirac Live**, but more focused on home studio environments. It includes more advanced configuration options and offers additional features, such as a headphone calibration tool.

Chapter 3

Methodology, consideration and decision on alternative solutions

Looking at the "Background and/or status of the matter" chapter, I want to develop a solution that incorporates as many of the strengths of existing solutions as possible.

Considering the available resources, the most effective solution is a **software-based solution** that interacts with the sound card, display, mouse, and keyboard available in an OS (*Operating System*).

This decision allows me to implement the following strength points:

- **Analysis capabilities:** This will include: spectrogram, 31-band graphic display, delay measurement, phase measurement, RT60 measurement, and waterfall diagram.
- **Processing capabilities:** It must be able to process a live signal using an equalizer implemented with digital filters.
- **Signal management:** It must handle an external clean signal, a signal that has passed through the system, and output a processed signal — all in real-time.
- **Low resource usage:** Considering the time available for this project and the limited budget. In my case, I already have everything I need for development, but anyone with a PC, a sound card, a microphone, and a speaker can use this solution.

Of course, the type and quality of the sound card and microphone will be critical to obtaining accurate and realistic results. Under optimal conditions, a high-resolution sound card with good SNR, along with a high-quality measurement microphone, is required. These microphones are typically small-diaphragm, condenser, omnidirectional, with a flat frequency response and good signal-to-noise ratio.

The importance of the speaker depends on whether we consider it part of the system (a fixed speaker that is part of the room) or simply a tool used to excite the room. In the first case, the program should correct the speaker's imperfections, as it is part of the system being analyzed and corrected. The only requirement for this speaker is that it must be able to reproduce the full frequency spectrum that will be analyzed and corrected.

However, in the second case, it is very important for the speaker to have the flattest response possible, since any imperfections in the speaker will introduce false data about the room acoustics.

I will be using a Behringer U-Phoria UMC204HD sound card, an old Electro-Voice desktop microphone, and an old AIWA SX-NAVH1000 home hi-fi speaker. These are not the best hardware for analysis tools, but they are sufficient to develop the software under home conditions.

Figure 3.1: Home setup showing the sound card, the speaker, and the microphone.

As a computer, I will be using a laptop running Ubuntu 22.04.5 LTS and Python 3.

In order to achieve the overall goal, the program to be developed needs a set of tools that must communicate with each other. To achieve this, a **GUI** (*Graphical User Interface*) will be very helpful, and it should be easy and intuitive to use.

Chapter 4

Development of the chosen solutions

Once we have defined what we want to do in the previous chapter, **Methodology, Considerations, and Decisions on Alternative Solutions**, it is time to get to work.

4.1 Graphic interface

One important point is to provide a GUI (*Graphical User Interface*) that is intuitive and easy to use. Since Python is being used for development, the chosen solution is to use the Tkinter[25h] library for the window and page system, and the Matplotlib[25c] library to display graphical results of certain analysis algorithms.

As Tkinter works, all actions that need to be triggered or modified through user interaction must be handled inside a function. Therefore, the way this library operates is by calling predefined functions in response to any user interaction.

First, we need to define the graphical architecture, where each module will be integrated into the graphical user interface. Since the program functions as a set of tools, we must define where each tool will be placed within the interface.

These tools can be divided into two important groups, which will also correspond to two separate windows accessible from the root window:

- **Analysis Window:** This window will contain the tools that need to receive sound from both the external input and the system input (Input from System). It will then display the results of the analysis performed on those signals.
- **DSP Window:** This window, also called the **"Correction Window"**, contains the tools that receive sound from the external input, as well as data indicating how the signal should be processed, and then send the processed signal to the output (Output to System).

Each of these two windows will be divided into pages, with each page containing a specific tool. Additionally, a settings window is needed, where the user can set global parameters. With all of this, we arrive at the following GUI scheme:

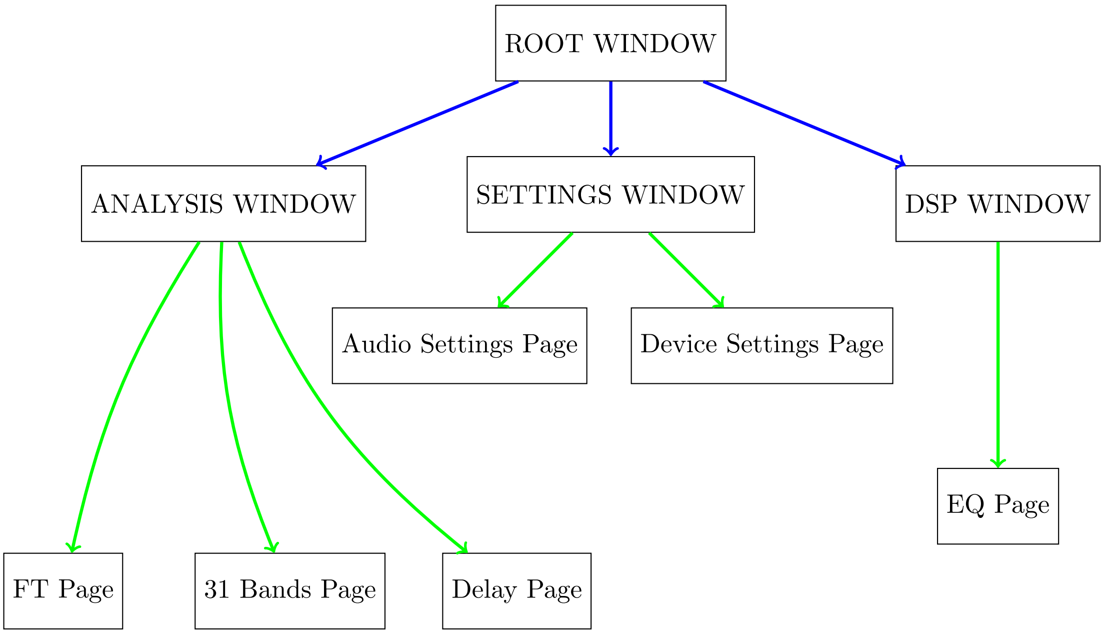

Figure 4.1: GUI window schematic

To make coding a bit easier, I will take advantage of the different pages to split the code across multiple Python files. In total, there are five files:

- **rta+c.py:** This is the root file that contains all initial instructions and the root window. It is the file that must be executed to start the program. Its name stands for **Real Time Analysis plus Correction**, which is also the name of the program.
- **settings.py:** This file contains the settings window and its associated pages.
- **dsp.py:** This file contains the DSP (correction) window.
- **analysis.py:** This file contains the analysis window and all its pages.
- **config.py:** This file declares some important variables that must be accessible from anywhere in the program. It does not contain any GUI-related elements.

Since the GUI layer is responsible for executing most of the program's modules, each window's file also contains all the necessary functions for analysis, processing, and plotting. In addition, all pages follow, as much as possible, the same structure in order to maintain the program's logic and readability. This structure is based on three parts:

1. **Initialize:** This section initializes the objects needed, sets up the plots with empty data, and defines the labels that will be displayed on the page.
2. **Update:** A function that contains the analysis algorithms, draws data on the plots, or runs DSP algorithms. This update function is called in every iteration of the algorithm loop using Tkinter methods.

3. **Control:** This final part contains the control parameters, flags, and buttons that affect the update function.

One very important aspect of using Tkinter is object management. By default, common Python classes may not work properly under certain GUI rendering conditions. To avoid this, when an object needs to be accessible from any page of the GUI, I use three different solutions: use of *config.py* as a shared configuration module containing predefined objects, which can be accessed from anywhere by importing this file; use of *global* and *nonlocal* objects inside the same window; and use of Tkinter classes such as `tkinter.IntVar`.

Finally, even though Tkinter offers a default way to close windows using the "X" symbol in the top left corner of the window, it does not always work properly—it may simply erase the window but not stop the process that is running from that window. To correctly close any window, it is important to define a close function. This function will be executed when the user clicks the "X" symbol, replacing the default function.

ipython

```python
"""
In this case, this is the function that closes the root window and, as a ...
↪  consequence, the entire program.
"""

# Close all
def on_close_all():
        config.update_enabled = False # Prevent all periodic updates
        stop_global_stream()  # Ensure the stream is stopped

        time.sleep(0.5)
        root.destroy()

root.protocol("WM_DELETE_WINDOW", on_close_all)

```

4.2 Signal Path

To perform any kind of analysis or signal processing, the first thing we need is data—more specifically, signal data captured from the sound card. This data is sent to different modules so that each one can carry out its specific task. Additionally, in the case of the correction module, it is necessary to send output data back to the sound card (the corrected signal). Additionally, there will be non-signal data containing the results of the analysis, which will be sent to the DSP (*Digital Signal Processor*) or the correction window.

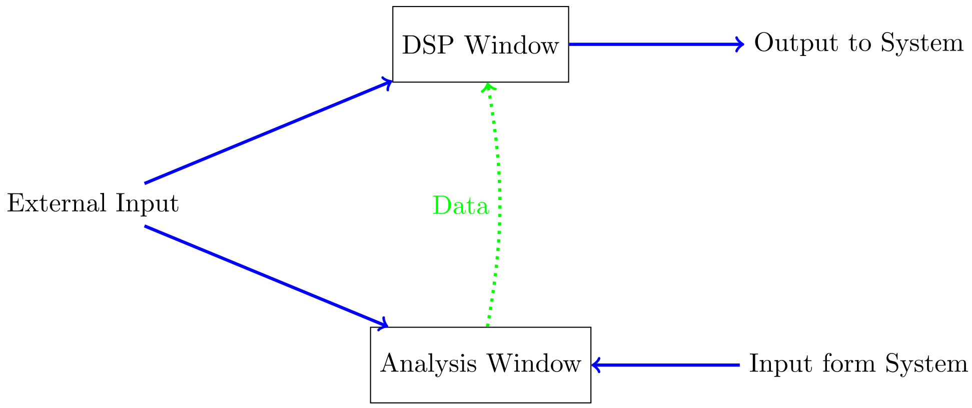

Figure 4.2: Basic representation of the required signal path

To achieve real-time processing, we must manage a constant flow of data. For this purpose, I will use a stream-style data management approach, which consists of capturing small blocks of data, processing them, and continuously sending the results. To capture and send this stream to and from the sound card, the Sounddevice library [24] will be used.

To simplify things, the two required inputs—**External Input** and **Input from System**—will be handled using a single input stream, while the output—**Output to System**—will be managed through a separate, independent output stream. All streams will share the same parameters, such as bit depth, sample rate, and block size. However, the audio device and channel parameters will be independently set for each stream. In the case of the input stream, each input channel must also be configured separately.

Once the streams are running and managed from the root window, we need to handle the data coming from the input streams and the data to be sent to the output stream. To achieve this, I will use one input buffer for both input channels and a separate output buffer. These buffers must be accessible to all tool modules in the program and will be locked whenever a module accesses them, in order to prevent simultaneous reads and writes.

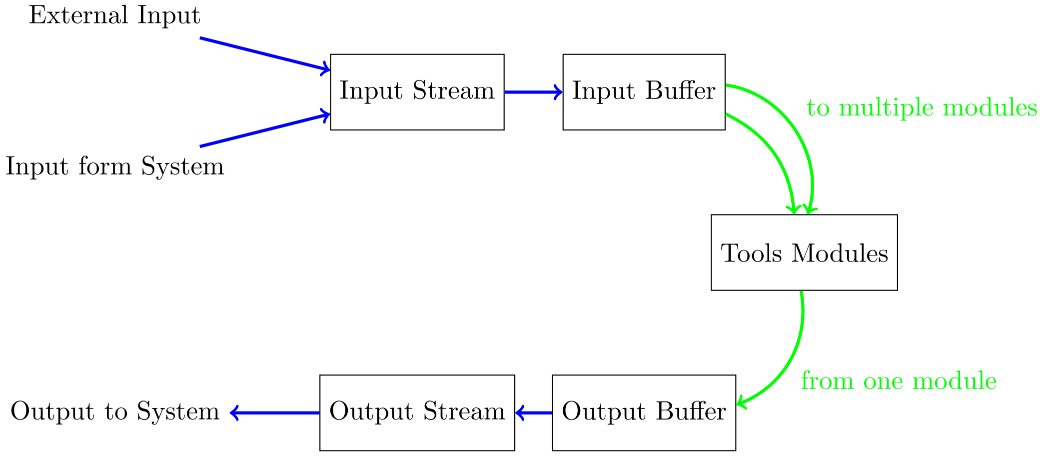

Figure 4.3: Stream management

4.3 Settings page

In order to be flexible with hardware, we need a settings window where we can tell the program which sound card, which channel, and which parameters we want to use. These parameters will also affect the resolution and fidelity of the analysis and processing, according to the sound card's capabilities.

I will splitt setting window in tow pages:

- **Device Settings:** Where we can define which sound card and which channels we want to use.
- **Audio Settings:** Where we can define which audio parameters we want to use.

4.3.1 Device Settings

I will use sounddevice methods to identify which sound cards and channels are available, and filter them into inputs and outputs. Then, using a questionnaire-style interface, I will let the user select which sound card and which channel they want for each required signal using a dropdown menu.

At the bottom of the page, there will be a "Confirm" button that performs some basic checks, such as verifying that the "external input" and the "input from system" are not the same input channel. Then, it will display a green message if the settings are applied successfully, or a red one if something goes wrong. Finally, it will update the values and labels on the root window to indicate which channels are currently selected.

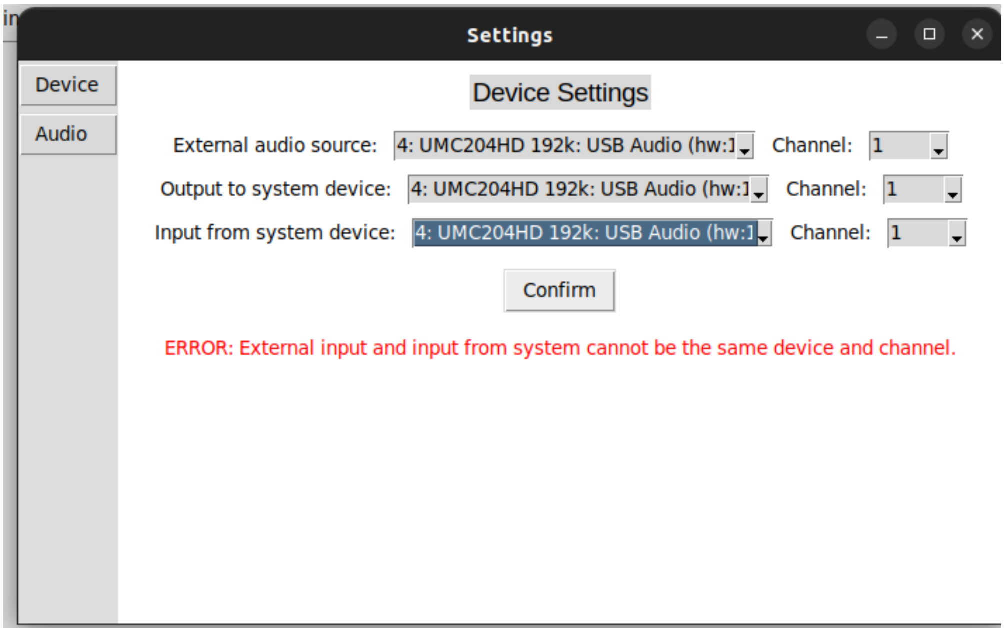

Figure 4.4: Device Settings page showing a red message because the user selected an invalid configuration.

4.3.2 Audio Settings

On this page, just like in the previous one, I will use a questionnaire-style interface with dropdown menus to define the Sample Rate, Bit Depth, and Block Size to be used.

For the Sample Rate and Bit Depth, I will suggest the most common values — such as 44,100 Hz, 48,000 Hz, 96,000 Hz, and 192,000 Hz for the sample rate, and 16, 24, and 32 bits for bit depth. However, the user is free to enter any value. The same applies to the Block Size.

At the bottom of the page, there is also a "Confirm" button, similar to the one in the "Device Settings" page. It performs basic checks, updates the selected values, and displays a message: green if everything is OK, orange in case of a warning (it applies the changes but alerts the user when a non-recommended or uncommon value is entered), and red if the Block Size is not a power of two — something required for the algorithms to work properly.

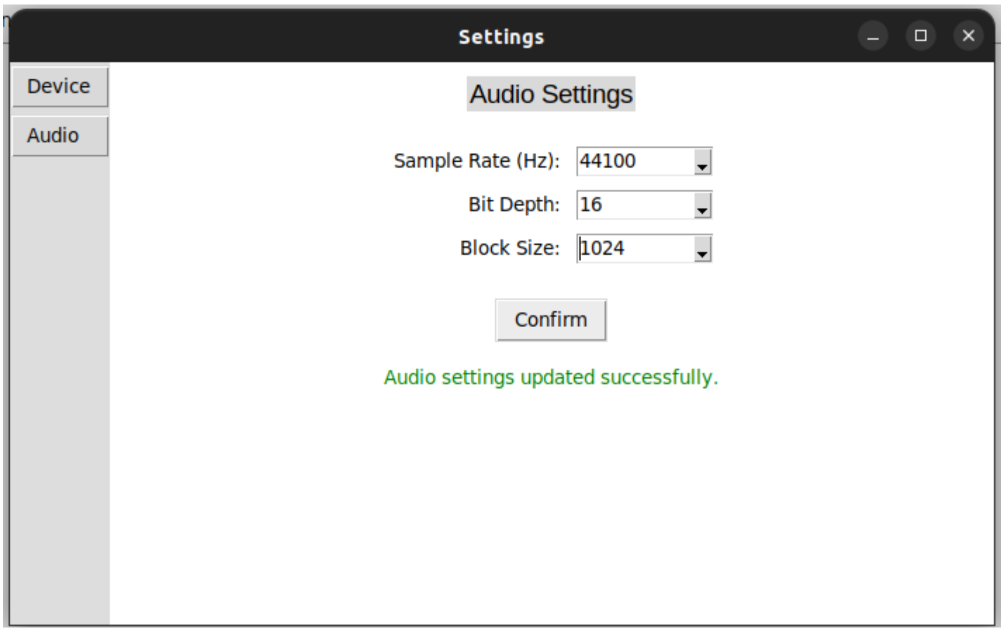

Figure 4.5: Audio Settings page showing a green message.

4.4 Acoustics analysis

This part represents the most complex part of the project, so it consists of different sets of tools, and some of them have to interact with each other. As was said before, each tool is inside an independent page.

So, first, apart from initial definitions, drawings, and labels, we need solid page management to ensure correct operation (without processes interfering with each other). The idea to develop is to close and erase everything related to one page every time the user changes pages, and then the new page will be executed. If something needs to be saved for later use on another page, it will be stored in a globally accessible object, managed from within each page. For example, the number of samples used to delay the external input signal is saved in the *config.py* module.

ipython

```python
"""
Close and Open Sequence for Page Switching
"""

```

```
# Destroy and unload current pages
config.update_enabled = False
time.sleep(0.3) #Give time for end whatever was executing

# Close all matplotlib figures to prevent memory leak
for fig in plt.get_fignums():
        plt.close(fig)
print("[INFO] Killed previous plots")

for name, frame in pages.items():
        frame.pack_forget()
        frame.destroy()  # Destroy the frame's widgets
pages.clear()
loaded_pages.clear()
print("[INFO] Cleared previous pages")

# Destroy any active page
for name in list(pages.keys()):
        pages[name].destroy()  # remove from memory
        del pages[name]
        del loaded_pages[name]
print("[INFO] Deleted previous pages")

time.sleep(0.3) #Give more time

config.update_enabled = True
print(page_name)

# Load and show new page
if page_name == "FT":
        load_ft_page()
elif page_name == "31 Bands":
        load_31bands_page()
elif page_name == "Delay":
        load_delay_page()
```

All signal and analysis algorithms are implemented using the **NumPy**[25d] and **SciPy**[25e] libraries.

4.4.1 Spectrogram (FT)

In studio or live sound situations, a spectrogram shows the energy of different frequencies over time using **FT** (*Fourier Transform*), displayed in the audible range (between 20 Hz and 20 kHz), and sometimes slightly extended beyond.

In my case, as a starting point, I'm using `numpy.fft.rfft`, which is the **RFFT** (*Real Fast Fourier Transform*) function from the NumPy library. It is an efficient variation of the **FFT** (*Fast Fourier Transform*) designed for real-valued input signals.

In order to achieve good resolution, the processing block size on this page is independent of the one set in the settings pages. It uses at least 100 ms of data (more precisely, the next power-of-two number of samples equivalent to 100 ms). This provides a minimum frequency resolution of 10 Hz.

Figure 4.6: Frequency Resolution as a Function of Time Window

$$\Delta f = \frac{1}{T}$$

But before starting the calculations, we first need to prepare the input data through two processes.

First, we have to delay the *External Input* signal to match it with the *Input from System* signal. This is managed using a set of instructions that operate on the delay buffer. Also, if there is not enough data to apply the required delay, the system waits for the next iteration.

ipython

```python
        """
        Set of Instructions to Manage the Delay Buffer
        """

    # Get delayed data
        if config.delay_samples == 0:
                ext_data = config.delay_buffer[-N_FFT:]
        else:
                ext_data = config.delay_buffer[-(config.delay_samples + N_FFT):-config.delay_samples]

        if ext_data is None or len(ext_data) < N_FFT:
                analysis_window.after(100, update_spectrogram)
                print("[DEBUG] Delayed ext_data too short")
```

```
            return

```

Secondly, I apply a windowing process to the acquired data in order to obtain more pleasant and understandable results to display. It is true that this kind of process slightly alters the final results, but in this case, it can help to better identify peaks that stand out—indicating possible resonances—and valleys, which can indicate cancellations, extra absorption, or a lack of energy in certain frequencies.

After trying several different window functions, I decided to use a *Blackman* window, implemented with the `numpy.blackman` function from the **NumPy** library.

After the FT calculations, I apply an averaging algorithm that can be performed either over time or over frequency. Time averaging is particularly useful when using non-varying sounds for frequency analysis to excite the system—for example, constant pink noise—allowing more stable and defined results.

Frequency averaging, on the other hand, is useful when analyzing the high-frequency range. In order to achieve good resolution in the low-frequency range, we often end up with unnecessarily high resolution in the high frequencies, since the data is displayed on a logarithmic scale. Therefore, if the user wants to focus on the high-frequency range, reducing the resolution through frequency averaging can make the visualization clearer and easier to interpret.

The final step is calculate the difference between External Input analysis and Input from System analysis, in order to get frequency response of the system. This is done with a simple resta of Analysis results.

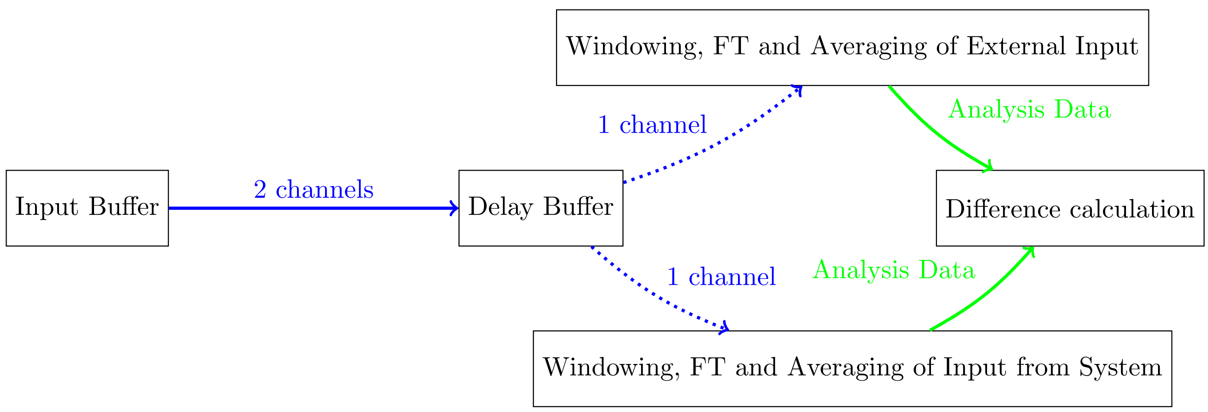

Figure 4.7: Diagram of the architecture of the FT page

At the end of the page, GUI elements are added for user interaction, allowing the user to set averaging values and to pause or resume the analysis.

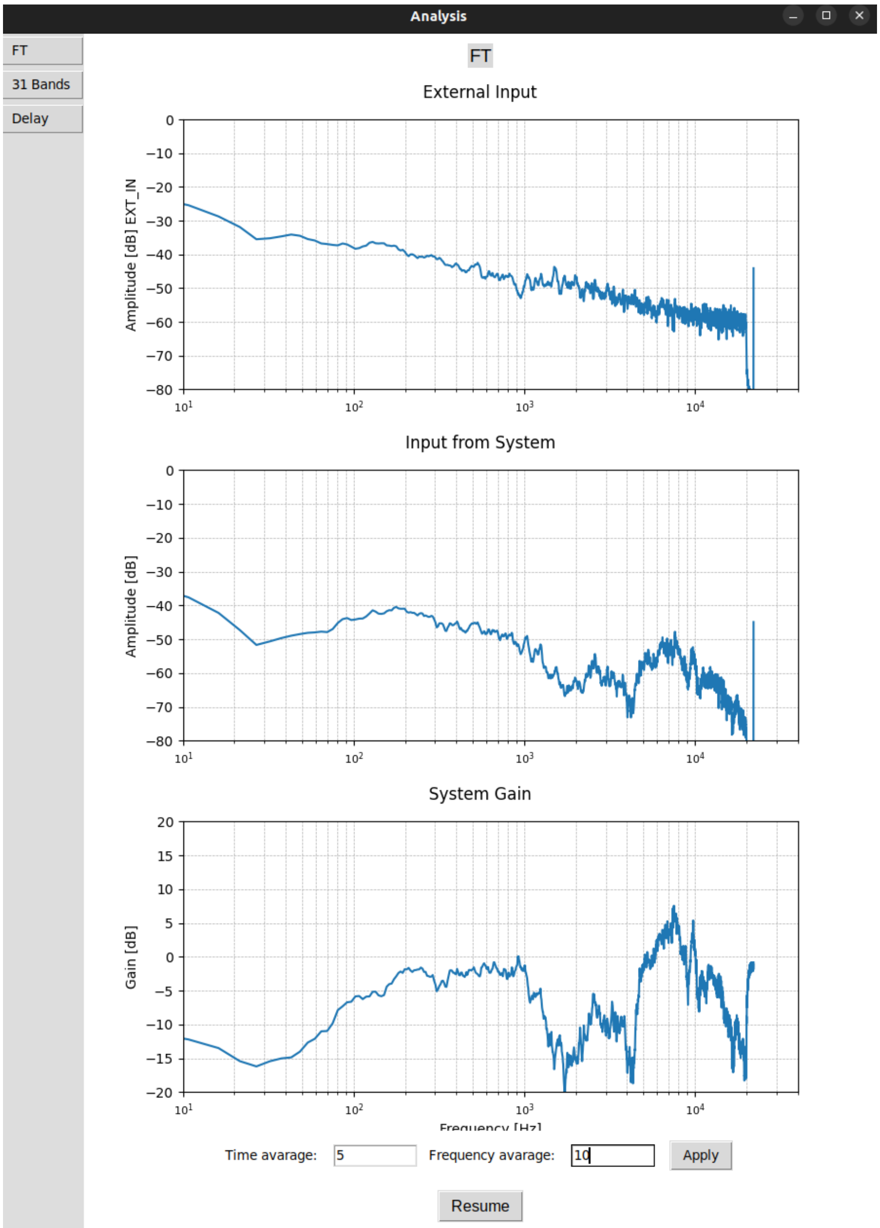

Figure 4.8: Analysis window - FT page.

Figure 4.8 shows pink noise in the External Input, the signal captured with a microphone in the Input from System, and their difference, using various averaging options.

4.4.2 RTA

Usually, any form of analysis that is performed in real time can be considered **RTA** (*Real-Time Analysis*). This includes a wide range of operations such as spectrum monitoring, transfer function measurements, phase and coherence analysis, and more—all happening as the signal flows. However, in my experience, in common usage, when someone refers to "RTA", they are often specifically referring to the classic 31-band graphical spectrum display. Also, the data collected on this page is especially important because it will be used to set the correction parameters. For all of that, this page has been named **RTA**.

Also, there is another conflict. When someone defines the 31 bands and their bandwidth, it is common to use the definition provided in **IEC 61260**. However, this standard does not mathematically respect the logarithmic spacing between bands. The 31 bands are defined as 1/3 of an octave per band.

Table 4.1: Center frequencies for true 1/3 octave bands and IEC 61260 bands

1/3 octave	19.69	24.8	31.25	39.37	49.61	62.5	78.75	99.21	125	157.49	
IEC 61260	20	25	31.5	40	50	63	80	100	125	160	
1/3 octave	198,43	250	314.98	396.85	500	629.96	793.7	1000	1259.92	1587.4	
IEC 61260	200	250	315	400	500	630	800	1000	1250	1600	
1/3 octave	2000	2519.84	3174.8	4000	5039.68	6349.6	8000	10079.37	12699.21	16000	20158.74
IEC 61260	2000	2500	3150	4000	5000	6300	8000	10000	12500	16000	20000

ipython

```python
"""
In order to obtain the true 1/3 octave values, the following line of code was used with IPython3.
The results were rounded to the second decimal place.
"""

[round(1000*2**(band/3), 2) for band in range (-17,14)]

```

On the other hand, this standard is widely used in many professional devices and software. One important example is the DBX 231s graphic equalizer, which is commonly used in analog processing chains.

Figure 4.9: Image of the front panel of the DBX 231s [25a], where we can observe that the center frequency bands are the same as those defined in IEC 61260.

In order to achieve the greatest possible compatibility and coherence with industry standards, I prefer to use the IEC 61260 standard.

The signal path on this page is very similar to the one used on the "FT" page. We have a buffer with two blocks of input data ("Input from external device" and "Input from system").

First, we copy the buffer data to the "delay buffer", where, if needed, the data will be adjusted to make it coincide with the applied delay. If necessary, the adjustment will use data from previous blocks stored in the same "delay buffer".

I created this buffer with the capacity to store 1 second of data, which can be used to apply a maximum delay of (1 - "Block Size in seconds") seconds.

Once the data in the "delay buffer" is adjusted, we can start applying algorithms to perform the analysis.

The algorithm is based on the calculation of RMS (*root mean square*) to obtain the energy for each frequency band, which is divided using filters for each band.

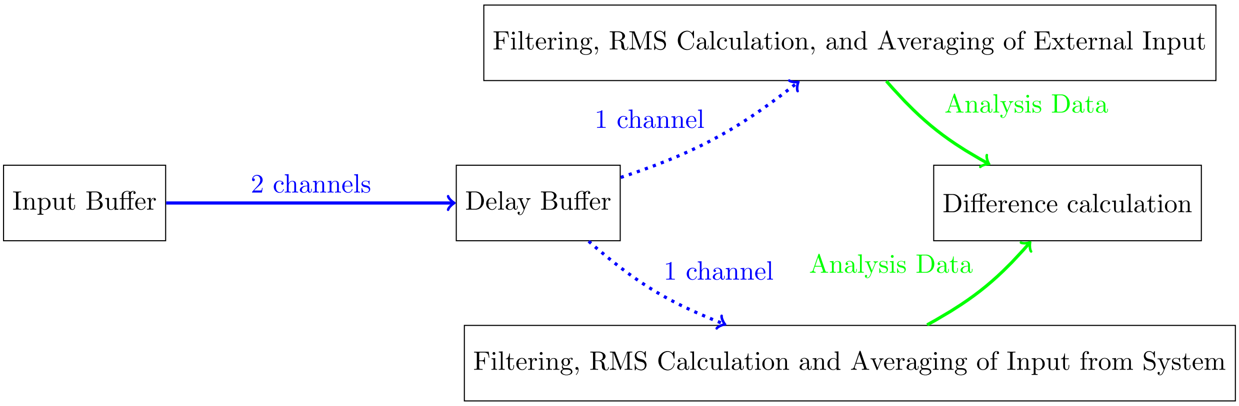

Figure 4.10: Diagram of the architecture of the RTA page

In this case, I'm not using any kind of windowing. As a starting point, I'm using 4th-order IIR (*infinite impulse response*) Butterworth band-pass filters for each band. All these filters are created using the `scipy.signal` library [25e], which returns SOS (*Second-Order Section*) parameters.

Figure 4.11: Second-Order Sections for IIR filters with their parameters

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

When the program applies each filter to the signal block, it also calculates the RMS and converts it to a logarithmic scale, which will be plotted on the graph and used to calculate the difference graph.

Figure 4.12: Root Mean Square to calculate energy from filtered signal

$$RMS = \sqrt{\frac{1}{N} \sum_{n=0}^{N-1} x^2[n]}$$

At the end, there is: a pause button, which blocks the update function and can be used to pause the graphics; a save button, to store the current values of the difference graph for later use in the correction window; and a time averaging section that works exactly the same as the averaging section from the FT page, except that it does not include frequency averaging (since it doesn't make sense to apply frequency averaging between bands).

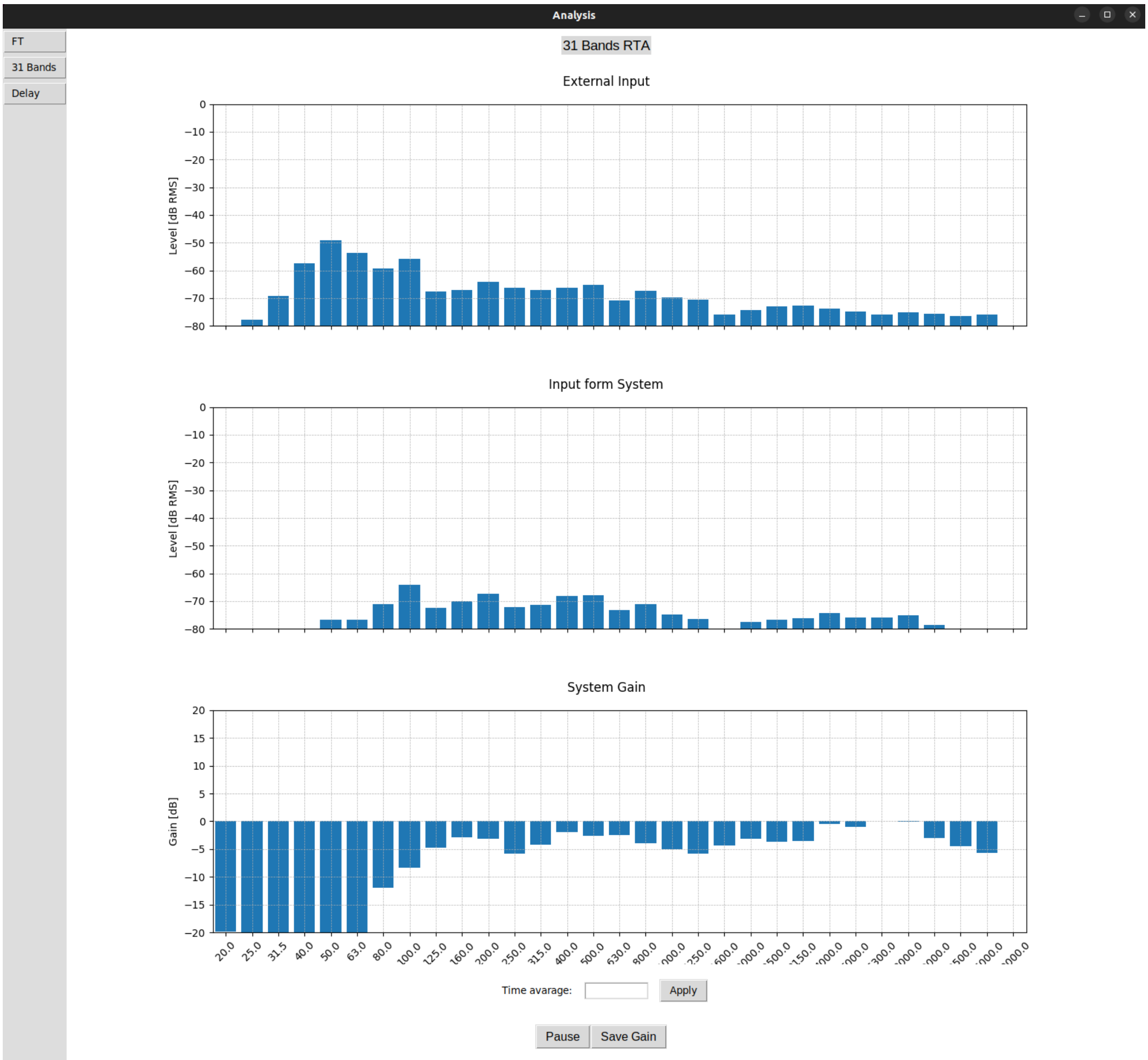

Figure 4.13: Analysis window - RTA page.

4.4.3 Delay

Usually, all systems introduce some delay, primarily caused by the travel time of sound between the speaker and the microphone. However, additional subsystems can also contribute to further delay. In cases where we want to perform analysis in a live situation with music (for example, measuring during a live show), where the excitation signal is not time-invariant in terms of frequency content, it is crucial to synchronize both inputs of the program in order to obtain an accurate representation from the analysis.

As explained in the FT and RTA pages, this synchronization is achieved using a dedicated buffer that allows shifting samples to align the signals. However, this mechanism needs to know how many samples the **External Input** must be delayed in order to compensate for the delay introduced by the system (**Input**

from System). The main goal of this page is to calculate that delay value.

For this purpose, I will use a correlation algorithm on both input signals to identify where they are most similar. However, to reduce the effect of amplitude changes introduced by the system, I first normalize each signal by its own RMS value before performing the correlation. For the correlation itself, I use the `scipy.signal.correlate` function from the *SciPy* library.

Once the correlation is obtained, I apply the absolute value to all results to eliminate the effect of possible polatity inversion introduced by the system (since an inverted signal would produce a negative correlation value). Then, I discard results corresponding to delays shorten than -10 ms. In the types of systems this program id designed to analyze, negative delays are physically impossible, as they would imply predicting a future signal. However, I allow a small negative range to help detect potential issues in case something goes wrong.

Now we can search for the maximum value of the correlation, and the position of this maximum will indicate the number of samples to delay. The result will be labeled both in samples and in the equivalent milliseconds. Additionally, I will plot the absolute correlation result to visually assess whether the detection is reliable. If the maximum value stands out clearly from the rest of the correlation, it means a strong and well-defined match has been found. However, if the peak does not stand out much, or if there are multiple peaks of similar value, the result may not be entirely reliable.

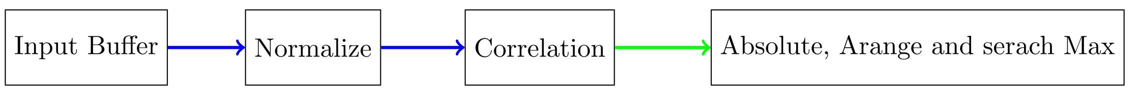

Figure 4.14: Diagram of the architecture of the Delay page

At the bottom of the page, I will add a “Pause / Resume” button and an “Apply” button, which saves the delay value in an object accessible to the other analysis pages. Once this parameter is saved, the other pages will automatically start applying the delay to their corresponding analyses.

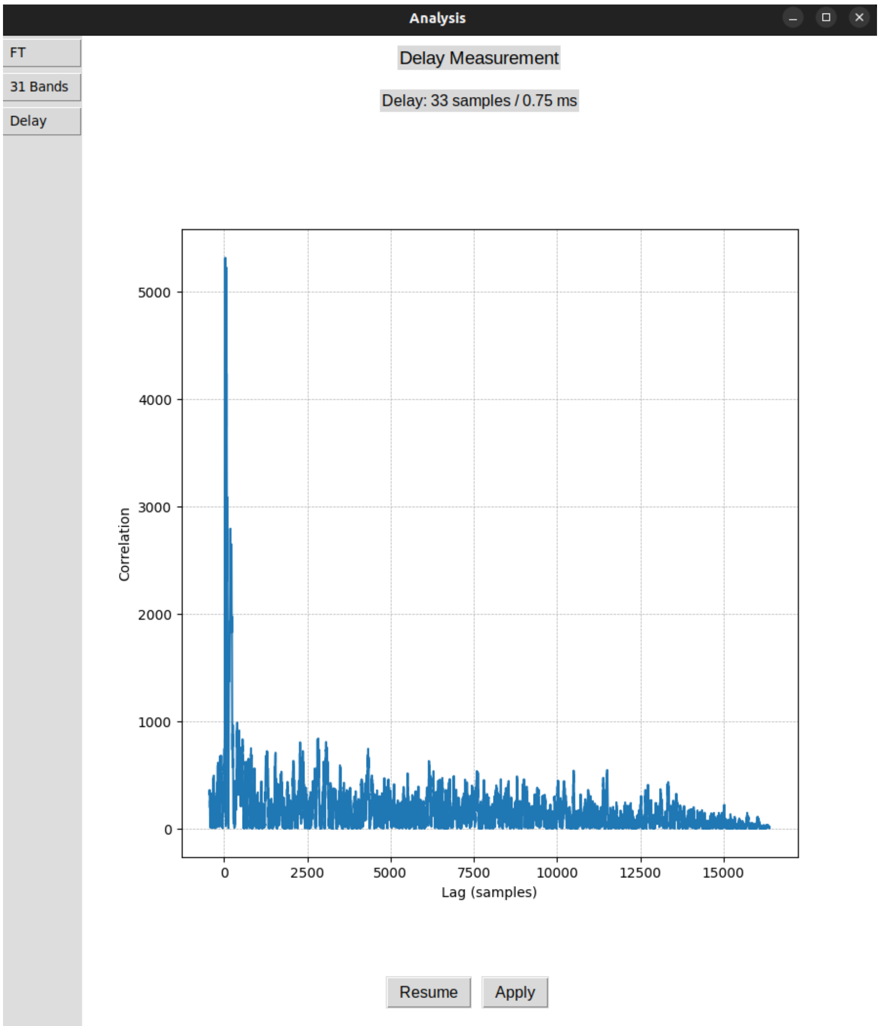

Figure 4.15: Analysis window - Delay page.

Figure 4.15 was captured using pink noise, with the microphone placed approximately 25 cm from the speaker, which corresponds to nearly 0.25 ms of delay.

4.5 Acoustic correction

The acoustic correction is based on a **DSP** (*Digital Signal Processor*) that processes the **External Input**, aiming to compensate for the imperfections of the analyzed system in order to improve the sound at the listening position. The processed signal is then sent through the **Output to System**.

This processing is based on a 31-band graphic equalizer using the IEC 61260-defined bands, to ensure consistency with the analysis and with common industry solutions (as explained in the Analysis–RTA section).

Furthermore, it must be able to bypass the processing, in case the user just wants to hear the unprocessed signal. It should also be possible to stop the **Output to System**, acting as a mute option. To switch between these three states, instead of using separate pages, I implemented a single button that cycles through the “Stop”, “Bypass”, and “EQ” states.

As the output stream is almost always active and continuously reads data from the output buffer, the “Stop” state is implemented by simply writing zeros to the output buffer. In fact, zeros are also the default content of the buffer.

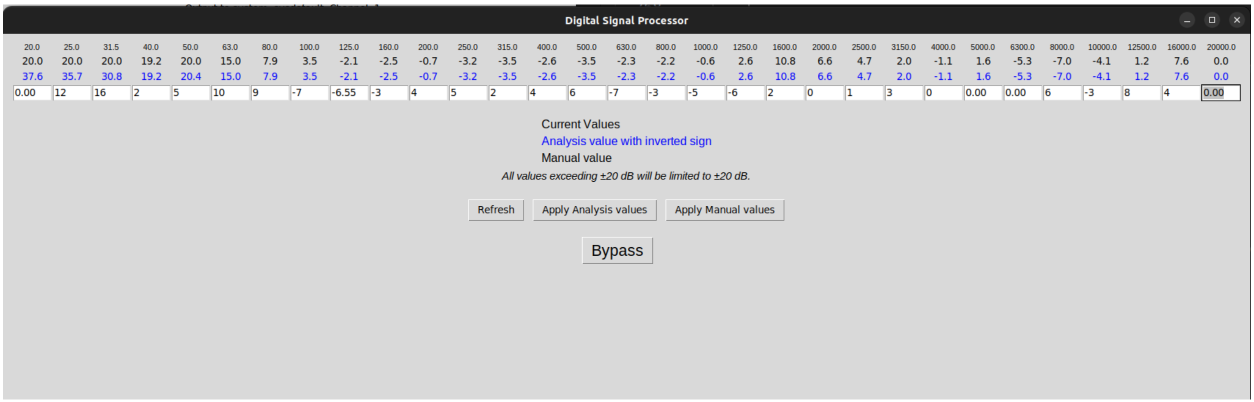

Figure 4.16: Correction window - DSP.

A very important aspect of this page is that it must take the data saved from the Analysis–RTA section and invert it, since compensating the system requires applying the inverse of the measured values. This is done automatically when the page is opened or when the user presses “Refresh.” Additionally, the user has the option to manually enter their own values.

4.5.1 Bypass

When I was developing one of the first versions of the program, I did not use shared buffers—each page managed its own stream, directly reading data from the input stream, processing it as needed, and, if necessary, writing directly to the output stream. For example, this was the code used for the Bypass function at that time: it took data from the input stream, stored it in a temporary buffer used only for this function, and then wrote that buffer to the output stream.

ipython
```python
"""
Core part of the code for the Bypass functionality that was working correctly.
"""

indata, _ = input_stream.read(block_size.get())
buffer[:] = indata[:, ext_in_ch.get() - 1]
out_block = np.zeros((block_size.get(), output_stream.channels), dtype='float32')
out_block[:, out_to_sys_ch.get() - 1] = buffer
output_stream.write(out_block)
```

This meant that every tool or page had to have its own input and output stream. However, while implementing the Bypass feature, I quickly realized that I couldn't use different streams simultaneously for the same physical input or output channels (for example, Bypass and FT analysis at the same time), because the Sounddevice library does not allow this.

Then, I switched to the current signal and path management approach, as explained earlier in the **Signal Path** section. I'm explaining this because this is the point where the Bypass functionality stopped working properly. The important issue that appeared will be discussed later in the **Results** chapter.

Nevertheless, the new signal path allows the program to simultaneously have a working Bypass while updating the FT analysis.

Specifically, the idea behind the Bypass is to take the shared input buffer and copy its data to the output buffer. This process, beyond displaying a label indicating that Bypass is active, does not require any additional GUI elements or feedback.

Figure 4.17: Diagram of the architecture of the Bypass module

As an attempt to solve the issue that will be discussed in the **Results** chapter, I added and managed an **Intermediate Buffer**, but it did not lead to any results.

4.5.2 31 Bars

This is the core of the DSP and uses the same set of filters as the RTA–Analysis page. First of all, it should be noted that it shares the same issue as the Bypass module. Nevertheless, I developed this module.

An important aspect is that this module must take data from the RTA–Analysis page, or user-defined data, to set the gain for each band. This serves as the basis for correction, aiming to level and compensate the room's frequency response. At the end, all filtered signal blocks with adjusted gains are written to the **Output Buffer**.

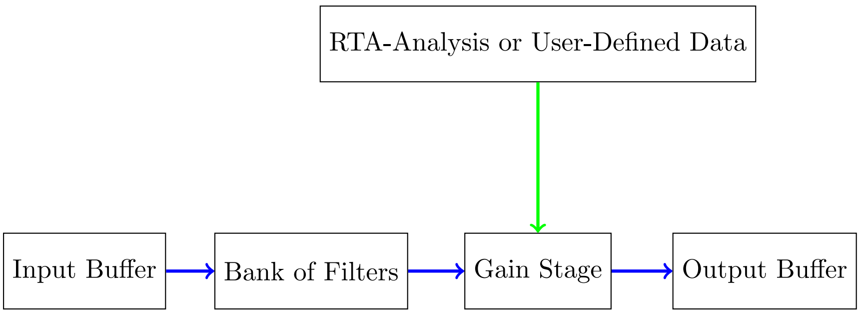

Figure 4.18: Diagram of the architecture of the EQ module

Most of the GUI is focused on this module. As shown in Figure 4.16, the user can refresh to load the data saved from the RTA analysis, apply those values, or manually define and apply their own values.

4.6 Others

There is a list of additional tools that still need to be implemented, such as phase analysis, RT60 measurement, waterfall diagram, and others. Unfortunately, I had to invest much more time than expected solving bugs, errors, and various issues—most of them related to **GUI** and **Signal Path** management. Some of these issues are still not fully resolved, as discussed in the **Results** chapter.

This left me without enough time to properly develop more advanced modules—especially in terms of researching and improving the filters used in the RTA and DSP modules—and to implement the remaining tools.

Chapter 5

Results

After four months of development, the program remains unfinished. Several unforeseen difficulties were encountered during the process, which required significantly more time than initially expected. As a result, there are not only some features still pending implementation, but also several existing issues that need to be resolved. The most important of these are:

- **GUI freeze:** Occasionally, the interface freezes when switching between pages in the analysis window. Debugging efforts have shown that the program continues to run in the background, but the graphical interface becomes completely unresponsive, leaving no option but to force quit and restart the program.
- **Output to System Glitch:** This is the main issue referred to several times in the Acoustic Correction chapter. It renders the Bypass and EQ modules nearly unusable in real-time situations, as it introduces a glitch effect in the Output to System signal. After conducting various tests, I concluded that the problem likely stems either from how data is managed within the DSP window or from how this window interacts with the output stream. The glitch appears to be caused by repeating or skipping certain data blocks in the output stream. When the program is configured with a very large block size, it becomes clearly audible that some blocks are played more than once, while others are skipped entirely.

Another minor issue is:

- **Filter Algorithms:** It would be beneficial to invest more time into researching and designing better filters in order to achieve more accurate and appropriate results, both for analysis and correction purposes.
- **User Limitations:** Additionally, the graphical user interface (GUI) could be more intuitive and requires refinement to offer clearer options and a better representation of ongoing processes. Furthermore, it is necessary to implement more constraints to prevent misconfigurations or malfunctions. For example, since the program uses the Sounddevice library and handles both inputs through a single input stream, both channels must originate from the same sound card. However, the current version of the program still allows users to select different sound cards for each input channel—an option that should be

restricted to avoid improper operation.

But not all news is bad. As will be shown in the **Final Test** section below, the program is already partially usable, and the developed tools work quite well.

5.1 Final tests

For the final test, which involved a real-world scenario, I had the privilege of accessing professional equipment and a real theater. The venue was the **Teatre del Coro**, located in Sentmenat, Spain, and managed by the non-profit cultural association *Societat Coral Obrera la Gloria Sentmenatenca*. The equipment available for the test was:

- **EVO 4:** External USB audio interface, used as the sound card for the program.
- **Audix TM1:** Measurement microphone, used as the source for the **Input from System** signal.
- **Mackie SRM-750:** Loudspeaker, responsible for playing the **Output to System** signal.
- **External Laptop:** Device used to generate the **External Input** signal.
- **Behringer X32 Compact:** The theater's digital mixing console. It is required to route signals to the speaker. Since it is part of the system, it will also be used to compare the analysis tools of the RTA+C program with the built-in tools of the mixer.

The sound card and the measurement microphone were kindly provided by *IMESDE, Integració, Distribució i Enginyeria Escènica, S.L.*, a private company located in La Garriga, Spain. All the devices used can be seen in Figure 5.6.

The first step was to place the measurement microphone. It was important to choose a good position where the microphone could capture the sound from a single representative point in the room. The first parameter to define was the microphone height. This venue is very versatile throughout the week, and the seats can be removed. However, events held without the seats typically do not require the use of the theater's sound system. Therefore, it was more appropriate to take measurements with the seats in place. Unfortunately, on the day I was able to perform the measurements, the seats had been partially removed. Nevertheless, I positioned the microphone at the average height of a seated person, as shown in Figure 5.1 and Figure 5.2.

Figure 5.1: Microphone height relative to the seat

Figure 5.2: Microphone height = 1.1m

Next, the microphone had to be placed at a representative point in the room. Since I was measuring only one speaker and the program operates in mono, the position had to be one where the speaker had good coverage and where the microphone was as close as possible to the center of the audience area. I decided to place the microphone approximately halfway through the depth of the room. Starting from a position where the speaker was directly in front of the microphone, I shifted it slightly to the right to bring it closer to the actual center of the room. As shown on figure 5.3 and figure 5.4.

Figure 5.3: Microphone position relative to the speaker

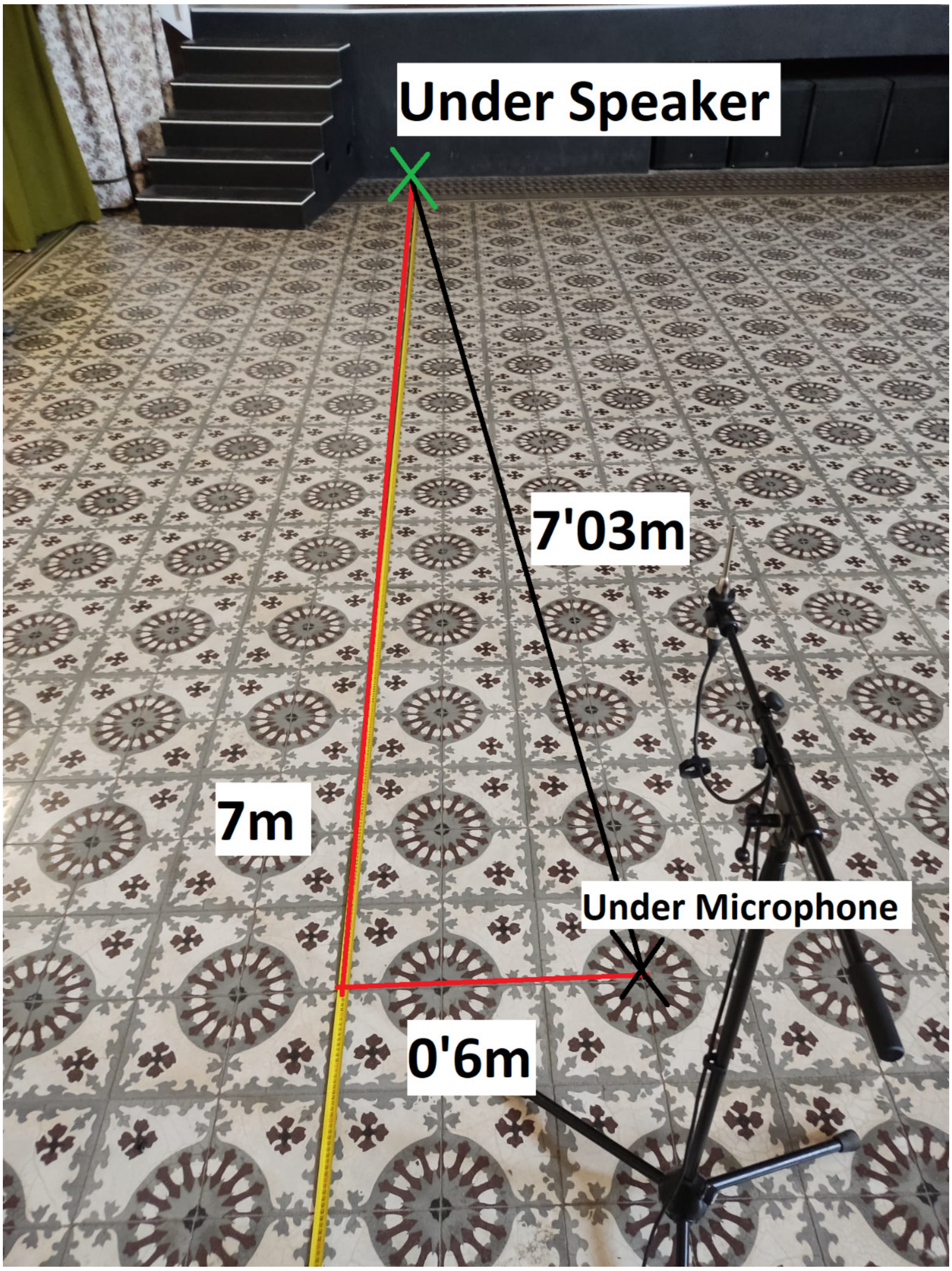

Figure 5.4: Horizontal distance between speaker and microphone

Knowing that the speaker is suspended from a rigging bar at a hieght of approximately 6.5 m, and substrating the microphone height, the vertical distance between the two is around 5.4 m. Additionally, the horizontal distance—shown in Figure 5.4—is 7.03 m. Using basic trigonometric calculations, we can determine that the total distance between the speaker and the microphone is approximately 8.86 m, as illustrated in Figure 5.5.

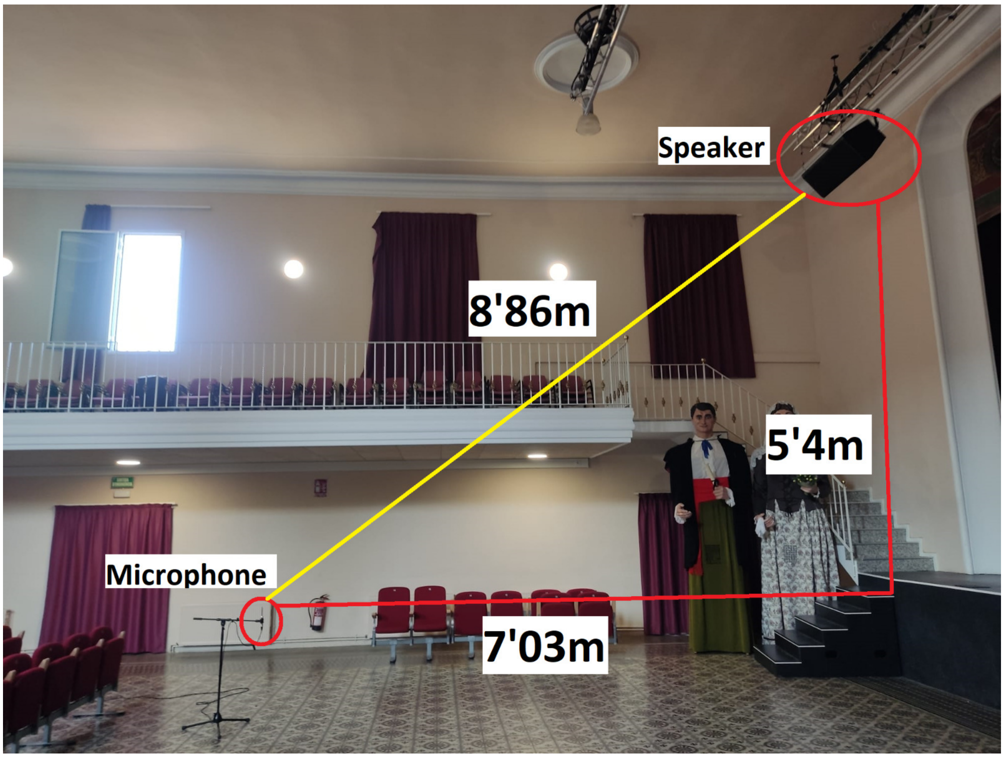

Figure 5.5: Distance between microphone and speaker

Once the microphone was positioned, it was time to set up the rest of the equipment, as shown in Figure 5.6.

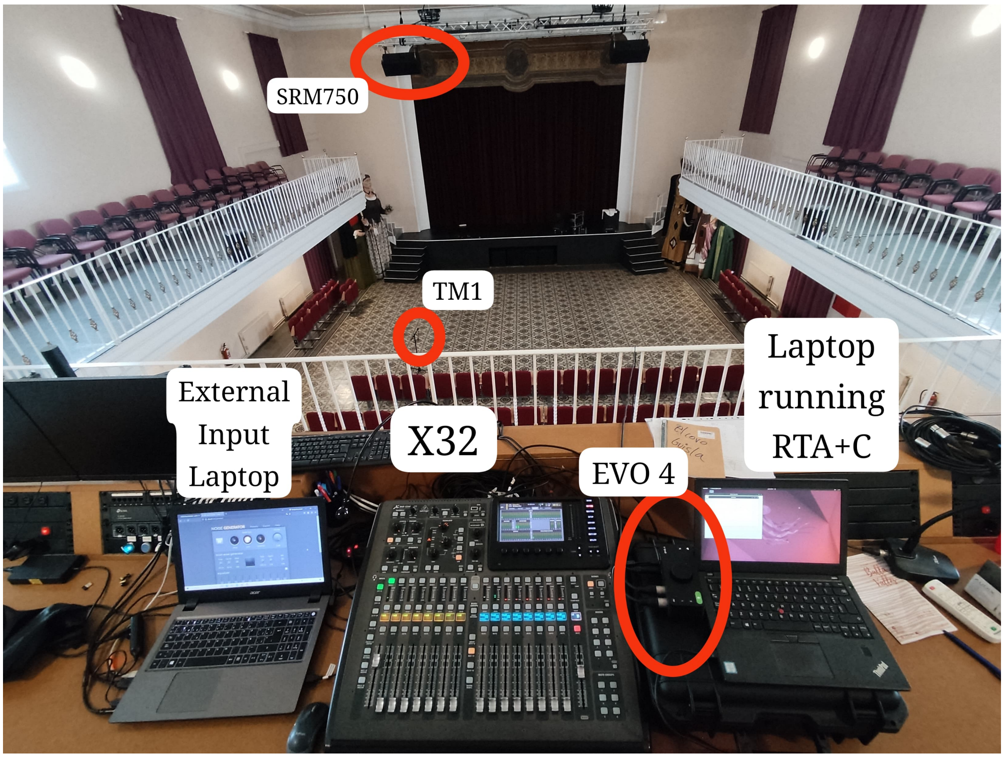

Figure 5.6: All equipment set up

In order to enable comparison between our software solution and the built-in tools of the digital mixing console (*X32*), all signals must be routed through the *X32*, as it offers advanced routing and distribution capabilities. Additionally, the console must be properly configured to route the signals correctly. Remember that we are using the *EVO 4* as the audio interface for the RTA+C software, as shown in the following connection diagram.

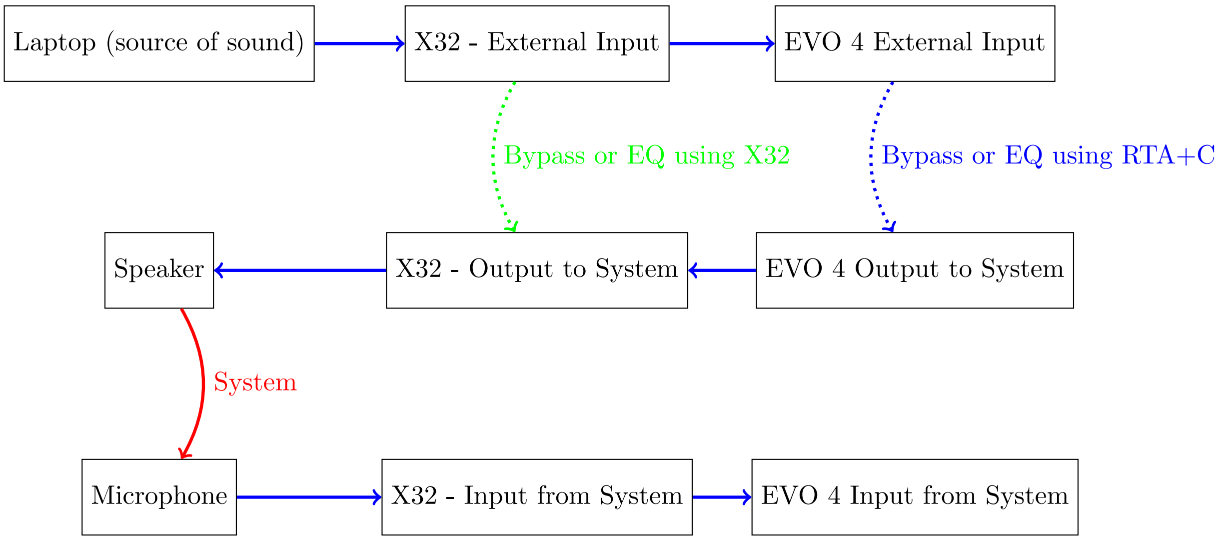

Figure 5.7: System connection overview

I compared my system with two basic tools available on the *X32* console:

- **RTA (Real-Time Analysis):** Shown in Figure 5.17, this tool provides a real-time spectrum display of the **Input from System** signal. Although the underlying algorithm is unknown, it is visually the most similar tool to the RTA page in my program.

- **Stereo GEQ (Graphic Equalizer):** A 31-band graphic EQ. Again, the specific algorithm used is not documented, but the concept is equivalent to the DSP window of the program.

Also, to avoid issues related to synchronization problems that cause the glitch effect in the Bypass and EQ modules of the **RTA+C** program, I will be taking measurements using the Bypass option of the **X32** console, which works flawlessly.

Once everything was connected, it was time to sit at the control desk (shown in Figure 5.6), perform basic checks on the signal path and gain staging, and disable or bypass all unnecessary options and tools on the mixing console. After that, the RTA+C program was launched.

The first step in the program was to configure the sound card and audio parameters using the "**Settings**" window, as shown in Figure 5.8 and Figure 5.10.

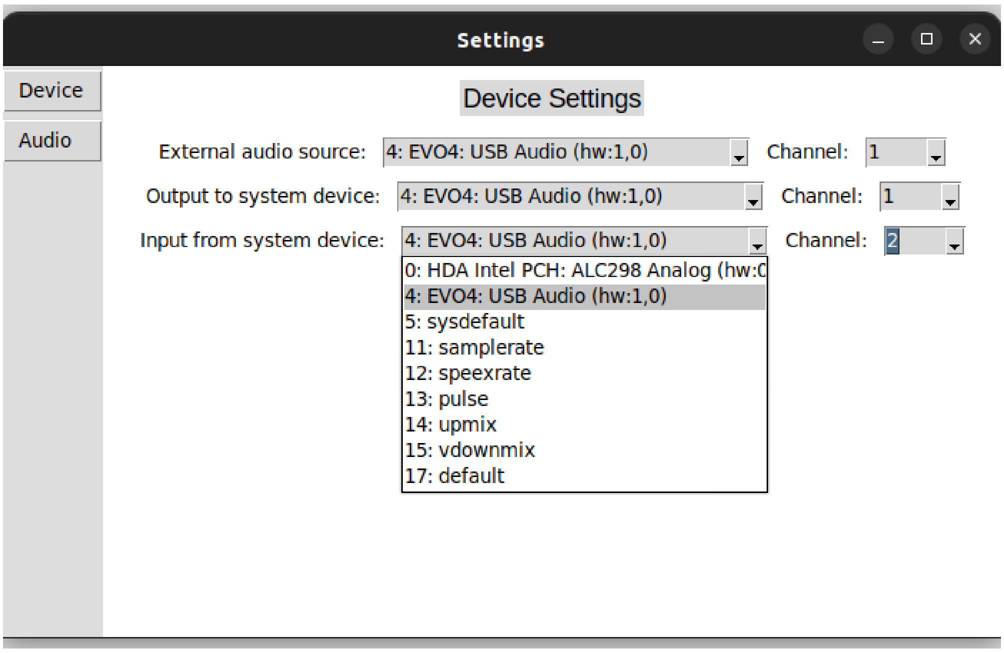

Figure 5.8: Device Settings Window recognizing EVO 4 soundcard

As we can see in Figure 5.8, the program automatically recognized the *EVO 4* soundcard without any prior configuration—just by plugging it in and setting the Device Settings window. At this point, I only changed the sample rate to 96 kHz from the Audio Settings page, confirming that the soundcard is compatible with this configuration. The other parameters were left at their default values.

Just to check if the parameters were working properly, I opened the RTA page, but something was not functioning correctly.

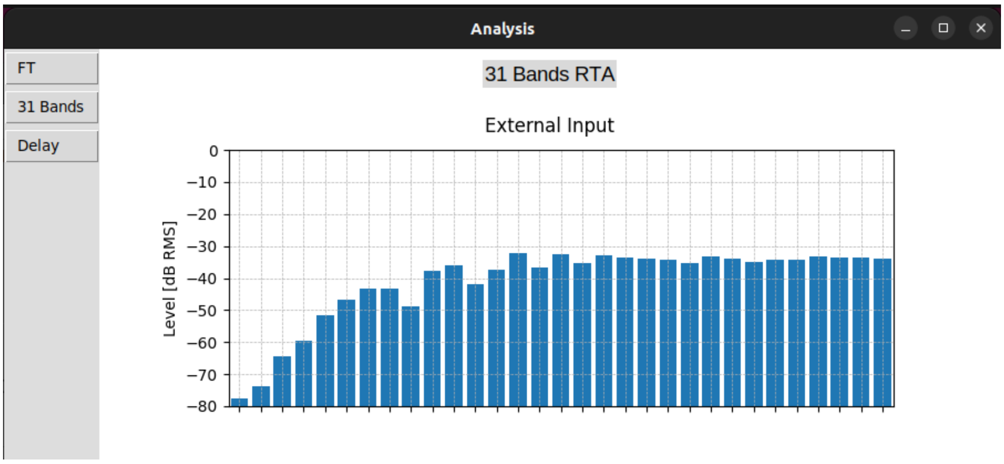

Figure 5.9: First RTA measurement over the External Input

As shown in Figure 5.9, with pink noise coming from the **External Input**, the External Input analysis shows a lack of low-frequency energy. I suspected that this might be due to the block size parameter being too small at this sample rate (which means a short analysis time window). To achieve a better representation of low frequencies, I increased the block size parameter, as shown in Figure 5.10, to 16,384 samples per block.

Figure 5.10: Audio Settings Window after changes

This block size, at a 96 kHz sampling rate, represents 170.7 ms, which in the worst case corresponds to approximately 3.4 wavelengths at 20 Hz (the lowest frequency considered). With this setting, we can observe in Figure 5.11 that the results are not perfect, but much better. I considered them good enough for this test.

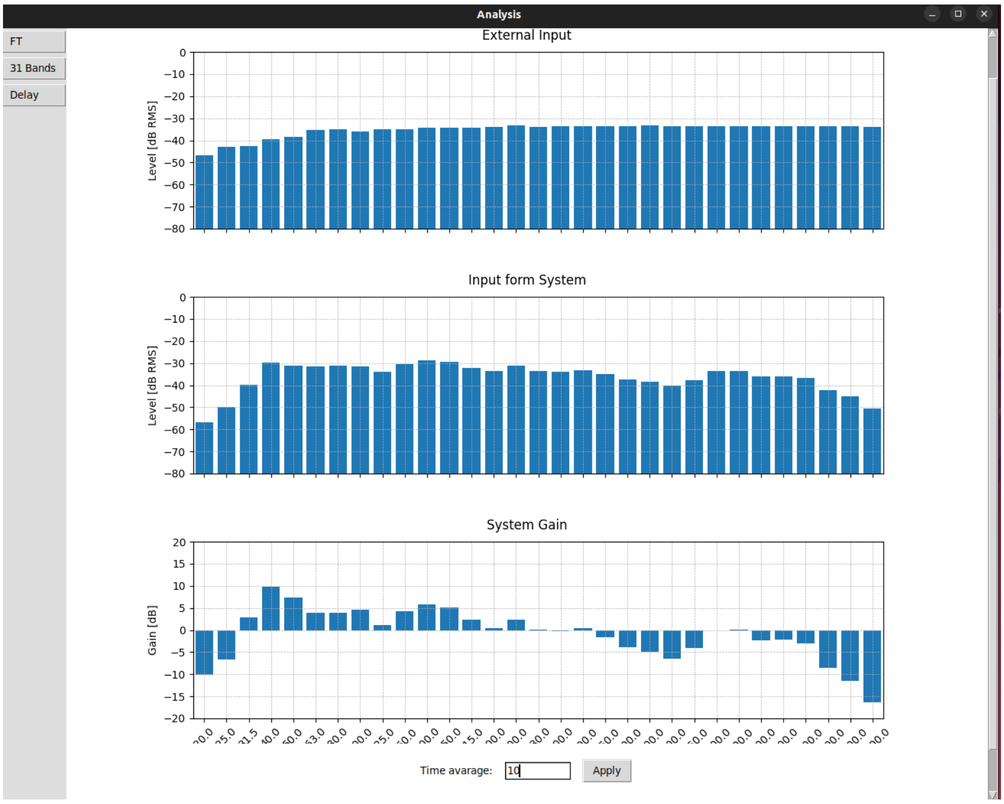

Figure 5.11: Second RTA measurement over the External Input

Once this was working properly, I started measuring the delay, which should correspond to the time it takes for sound to travel from the speaker to the microphone. Considering the speed of sound as 340 m/s, this value should represent the distance between the two devices.

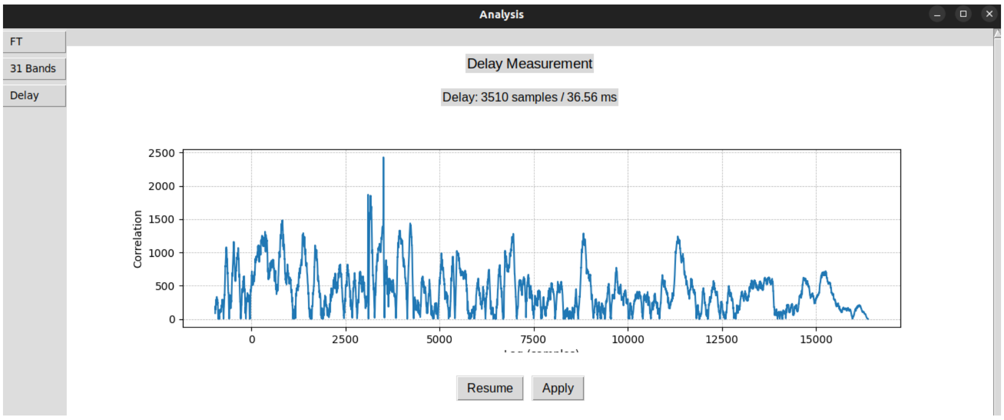

Figure 5.12: First Delay Measurement

The measurement was not very stable—most iterations gave different values. However, I paused the process and took the most consistent and frequently repeated value. Additionally, I considered the correlation plot to ensure the result was reliable, focusing on cases where the maximum value clearly stood out from the rest. This can be seen in Figure 5.12.

The measured delay was 36.56 ms, which—considering the speed of sound (340 m/s)—corresponds to a distance of 12.43 m. However, as shown in Figure 5.5, the actual physical distance between the speaker and the microphone is 8.86 m, which is 3.57 m more than expected.

It is logical to expect a slightly longer delay than the pure acoustic distance, since the signal passes through various systems before reaching the speaker—each of which may introduce some latency. For example, it is known that the mixing console adds approximately 0.3 ms of latency per output channel, which corresponds to only 0.1 m of additional delay. Even if this latency accumulates through multiple stages, it is still not enough to justify the extra 3.57 m (around 10.5 ms).

Because of this significant discrepancy, I continued measuring to investigate further.

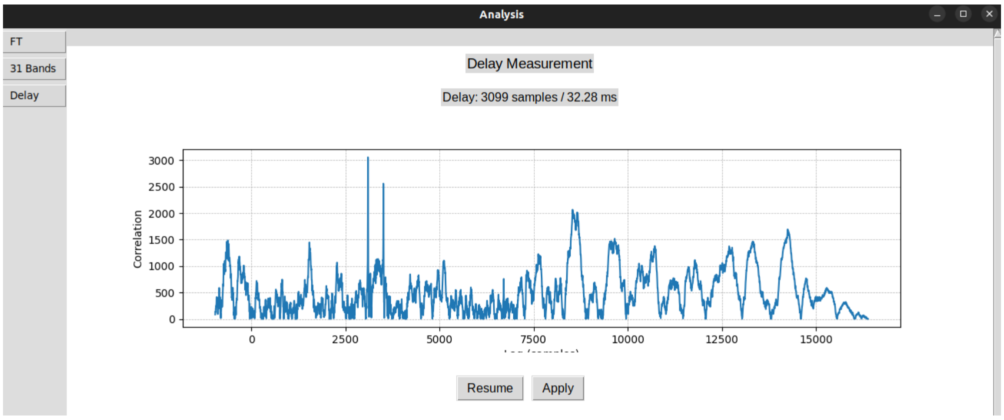

Figure 5.13: Second Delay Measurement

After observing the iterations of the plot and the measurement results for a while, I realized that there was a second value that appeared repeatedly across many iterations and also made sense visually on the plot. As shown in Figure 5.13, the plot clearly highlights two peaks with significant correlation values. At the right moment I stopped the measurement, the lower time value among the two had the highest correlation, which corresponded to 32.28 ms. This value translates to a distance of 10.98 m, which is 2.12 m more than the actual physical distance between the speaker and the microphone.

Although not fully conclusive, this second measurement seems more reasonable than the previous one, so I decided to use this value and apply it over the rest of the measurements.

Next step is open the FT page.

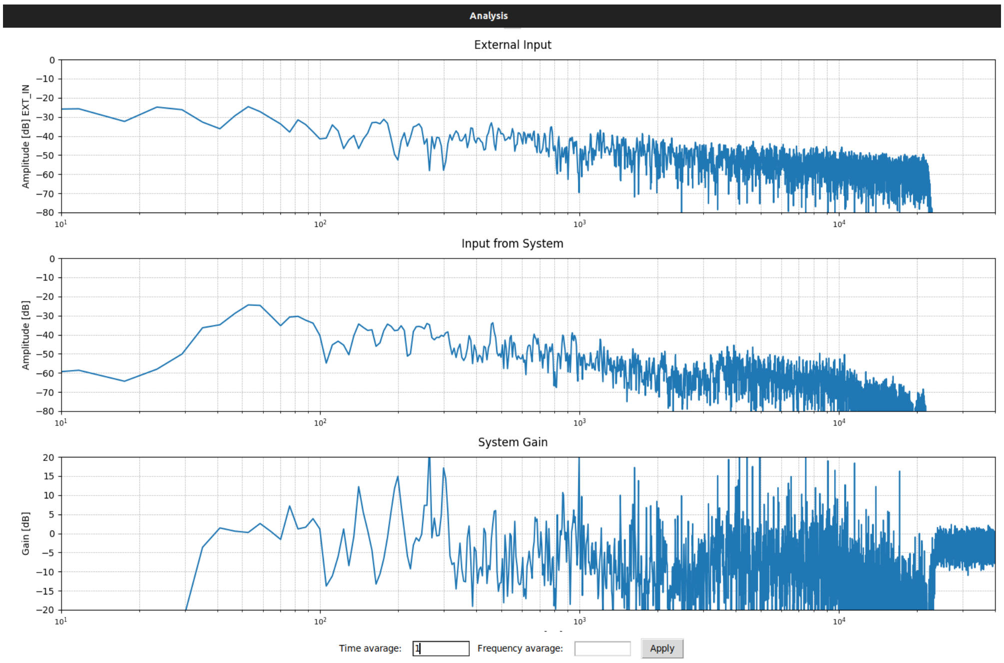

Figure 5.14: FT Measurement Without Averaging Applied

Continuing with Pink noise excitation from **External Input**. I can see the FT page results without applying any avarage option, as shown on Figure 5.14. But the plots are very anarchic, changes a lot between iterations, and is difficult to make a solid interpretaion.

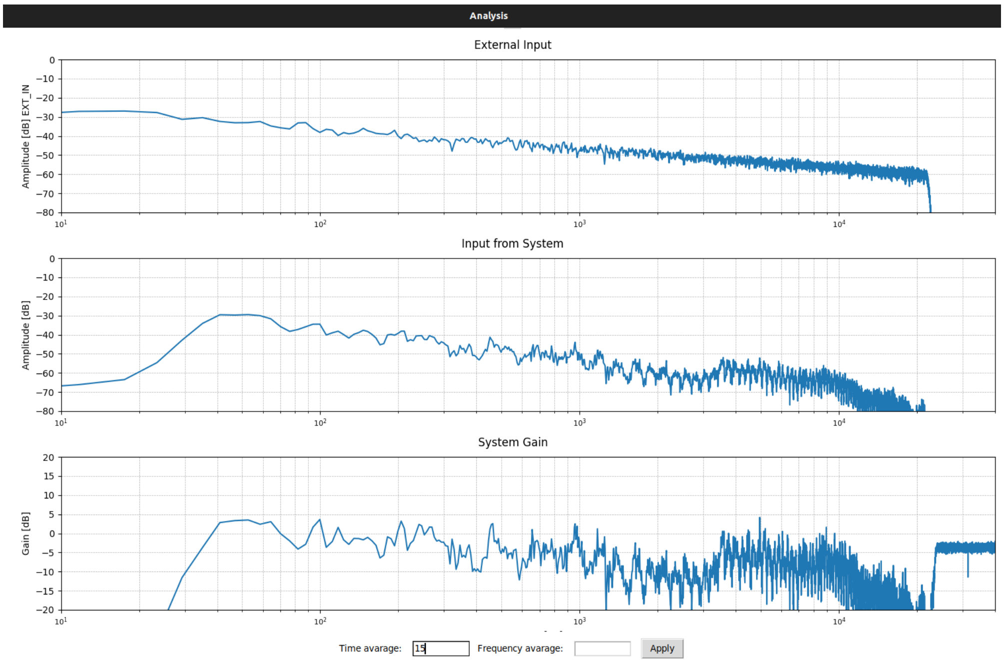

Figure 5.15: FT Measurement With Time Averaging Applied

In Figure 5.15, a time average is applied, which greatly improves the consistency between iterations and makes it much easier to interpret the displayed data. However, interpreting the data in the high-frequency range is still somewhat confusing. To address this, I applied frequency averaging.

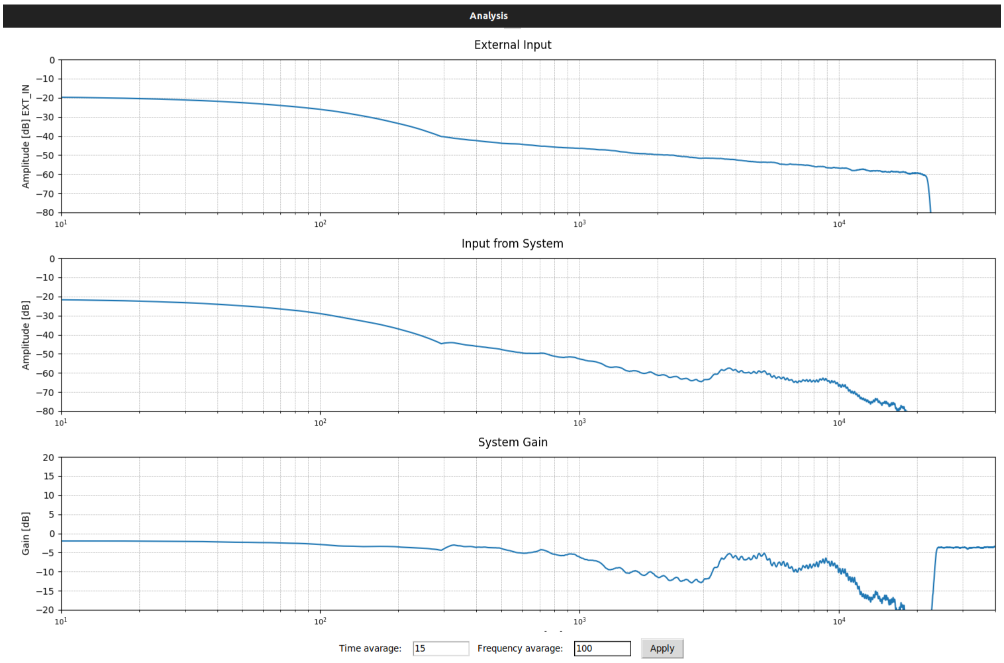

Figure 5.16: FT Measurement with Time and Frequency Averaging Applied

With all averaging options applied, we obtain the result shown in Figure 5.16. While we lose considerable resolution in the low-frequency range, the high-frequency representation becomes much more visually understandable.

Figure 5.17: X32 RTA Tool with Untreated Input from System Signal

To check whether the results are reasonably close to reality, we use the **X32** RTA tool as a reference and compare it with the **Input from System** analysis. As shown in Figure 5.17, and comparing it with the FT page of the RTA+C program (see previous figures), the results look very similar—especially the dip between 1000 Hz and 4000 Hz. Therefore, I consider that the FT page works pretty well.

Next step: open the RTA page.

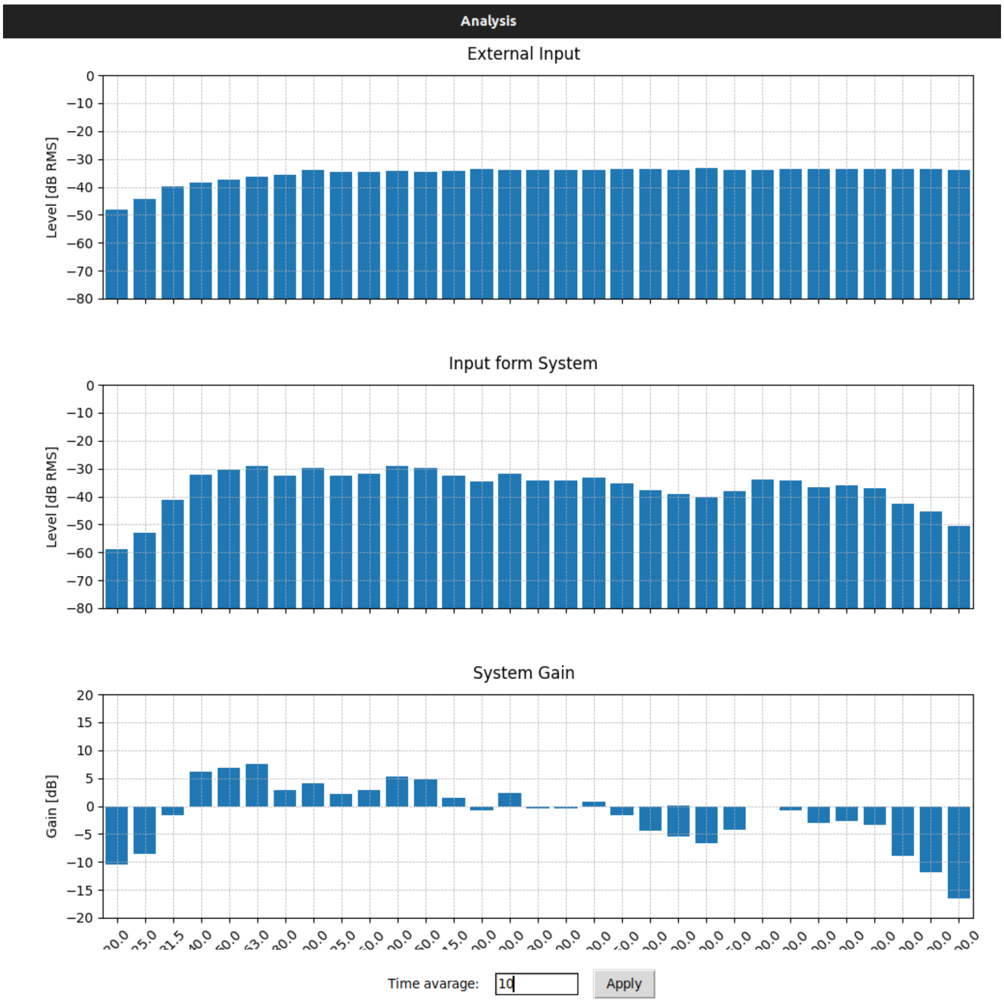

Figure 5.18: Reference RTA values, saved to apply in the Correction Window

Referencing Figure 5.18, with 10 iterations averaged, I obtained a reasonably stable plot that is easy to interpret. Additionally, when compared with Figure 5.17, we can clearly observe similarities with the **Input from System** analysis. The plot also accurately displays a flat response for the **External Input**, which is expected given that this signal is a clean Pink Noise.

Considering these results satisfactory, I saved the **System Gain** values using the "*Save Gain*" button. Now, it's time to open the **DSP** window.

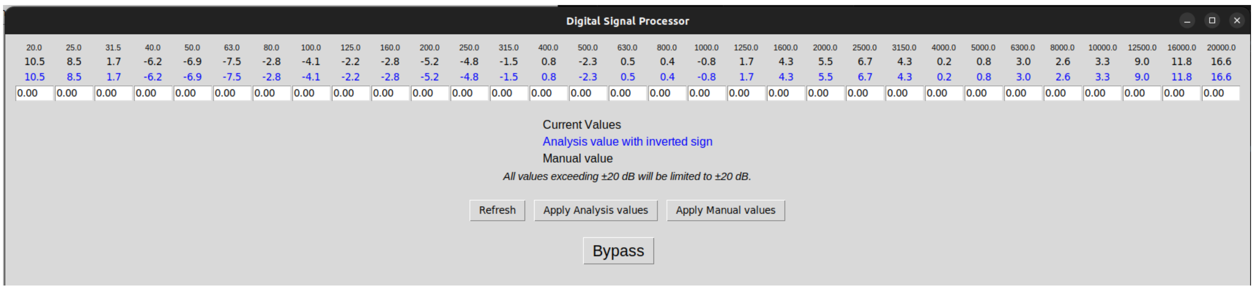

Figure 5.19: DSP window with EQ values applied from RTA analysis

Once the **DSP** window is opened, since the values from the RTA were previously saved, it automatically loads with the *Analysis Values* imported with inverted sign (to apply correction, we need to compensate the analysis values, which is done by inverting their sign). If we overwrite the saved ones, or if there were no values when this page was first opened, it is possible to import the latest saved values using the *Refresh* button. Then, simply press *Apply Analysis values* to send them to the **EQ** modules. At this point, the page is configured as shown in Figure 5.19. Additionally, it is also possible to manually enter values; in that case, pressing the *Apply Manual values* button will activate them.

At this point, I deactivated the **X32** Bypass to activate the EQ module from the RTA+C program with the applied values. Luckily, with pink noise, the glitch effect is almost unnoticeable.

Figure 5.20: X32 RTA tool with Treated Input form System Signal

First, I took a look at the **X32** RTA tool, shown in Figure 5.20, to compare it with the non-correction applied results from Figure 5.17. Clearly, the graph looks flatter, which indicates that the correction is working properly and yielding good results. It's not perfect, but the dip between 1000 and 4000 Hz has improved significantly.

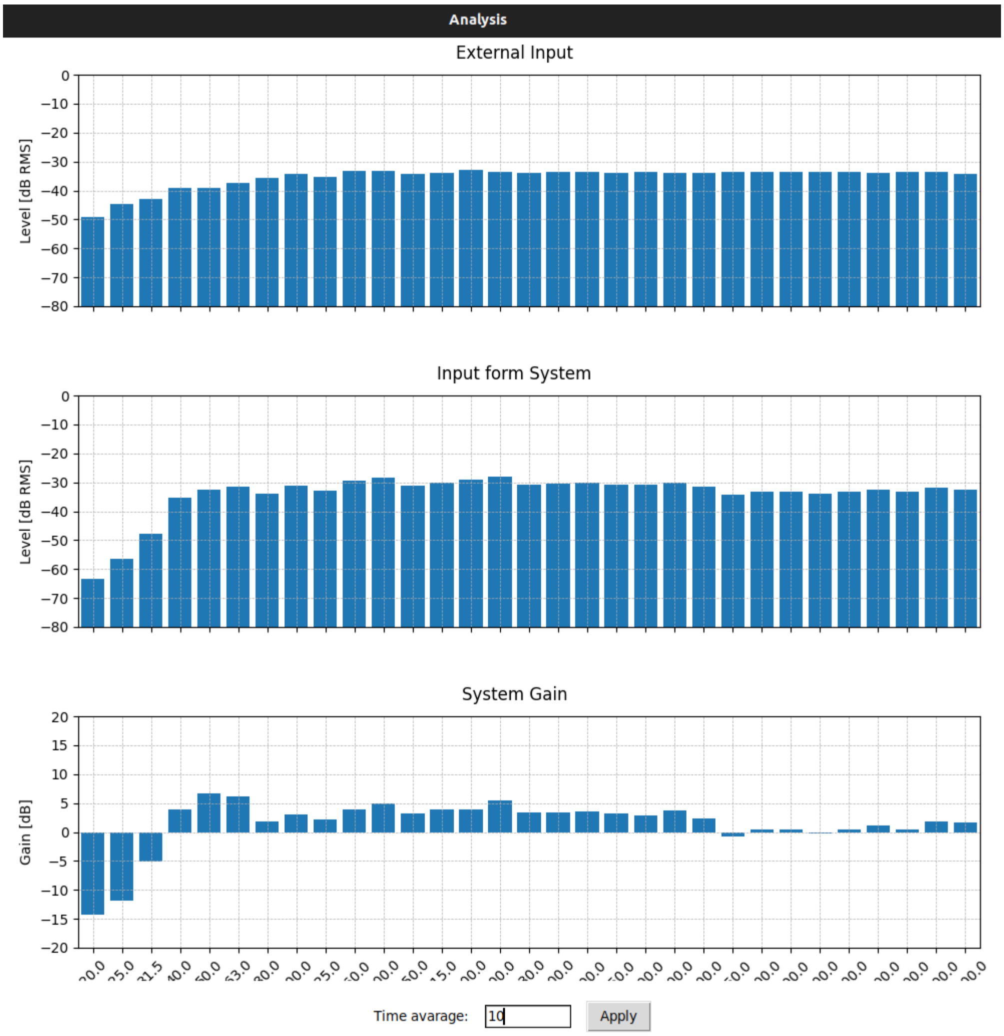

Figure 5.21: RTA Results After Activating RTA+C EQ

Then, I looked at the RTA module page from **RTA+C**, shown in Figure 5.21, and the results looked very promising. The plot appeared much flatter compared to previous measurements, which indicates that the EQ module is functioning correctly. This also confirms that using the same filters for both analysis and processing was a good design choice. What remains pending is further research and the implementation of improved filters to achieve a more natural and realistic representation of the real acoustic response.

Figure 5.22: Configuring the X32 EQ tool with parameters obtained from RTA+C

Also, I wanted to compare the correction applied by **RTA+C** with the *Stereo GEQ* tool of the **X32** console. To do so, I configured the console's EQ tool with the same parameters used in the EQ module of the software (taken directly from the RTA page of **RTA+C**). The configuration is shown in Figure 5.22.

Figure 5.23: Input from System with X32 EQ treatment, visualized using X32 RTA tool

And in Figure 5.23, we can see the **Input from System** signal displayed by the RTA tool of the **X32** console. This result was obtained after processing the **Output to System** signal with the console's `Stereo GEQ` tool using the same parameters taken from the **RTA+C** program. The result looks pretty good, and when comparing it with Figure 5.20, the response is very similar. This suggests that the filters currently used in the **RTA+C** program are quite effective.

As I was testing a software solution intended to work properly in live situations, for the final test I replaced the pink noise with music.

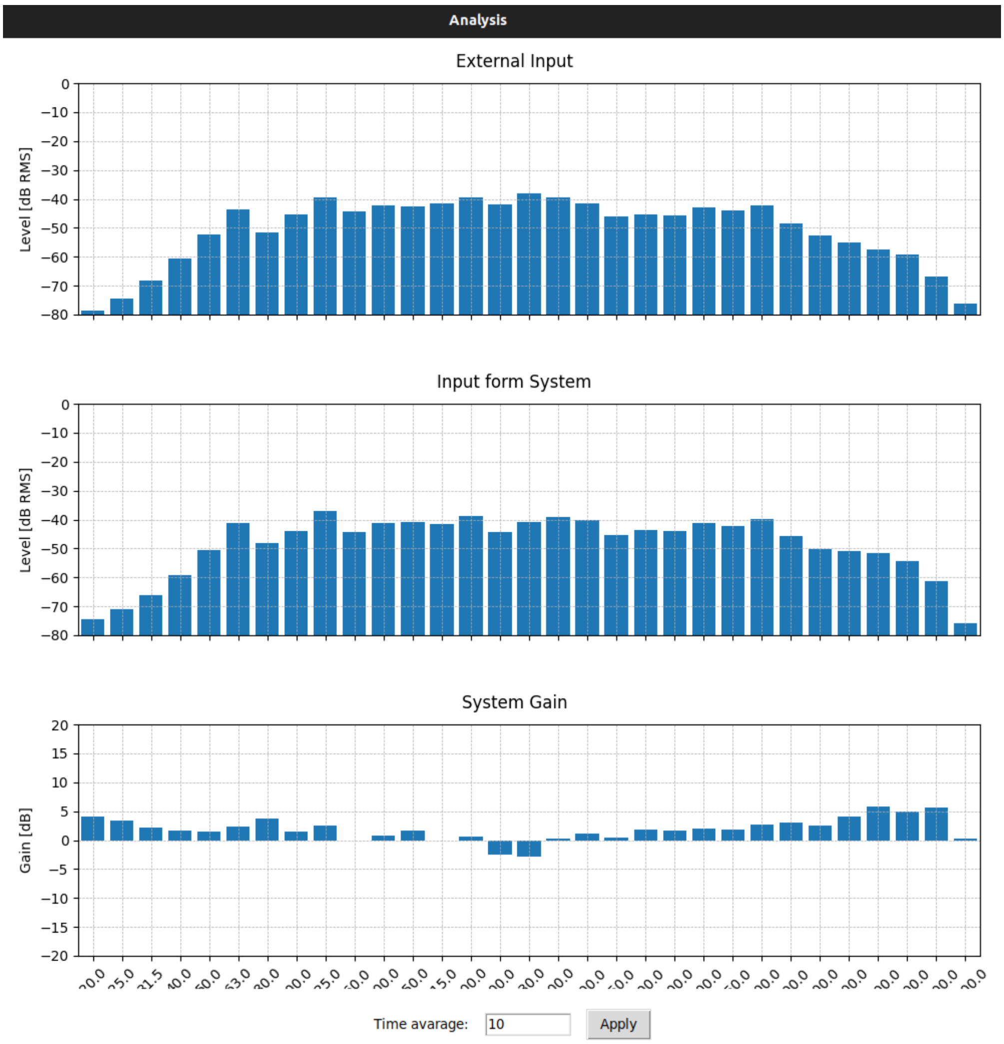

Figure 5.24: RTA page, using music instead of pink noise

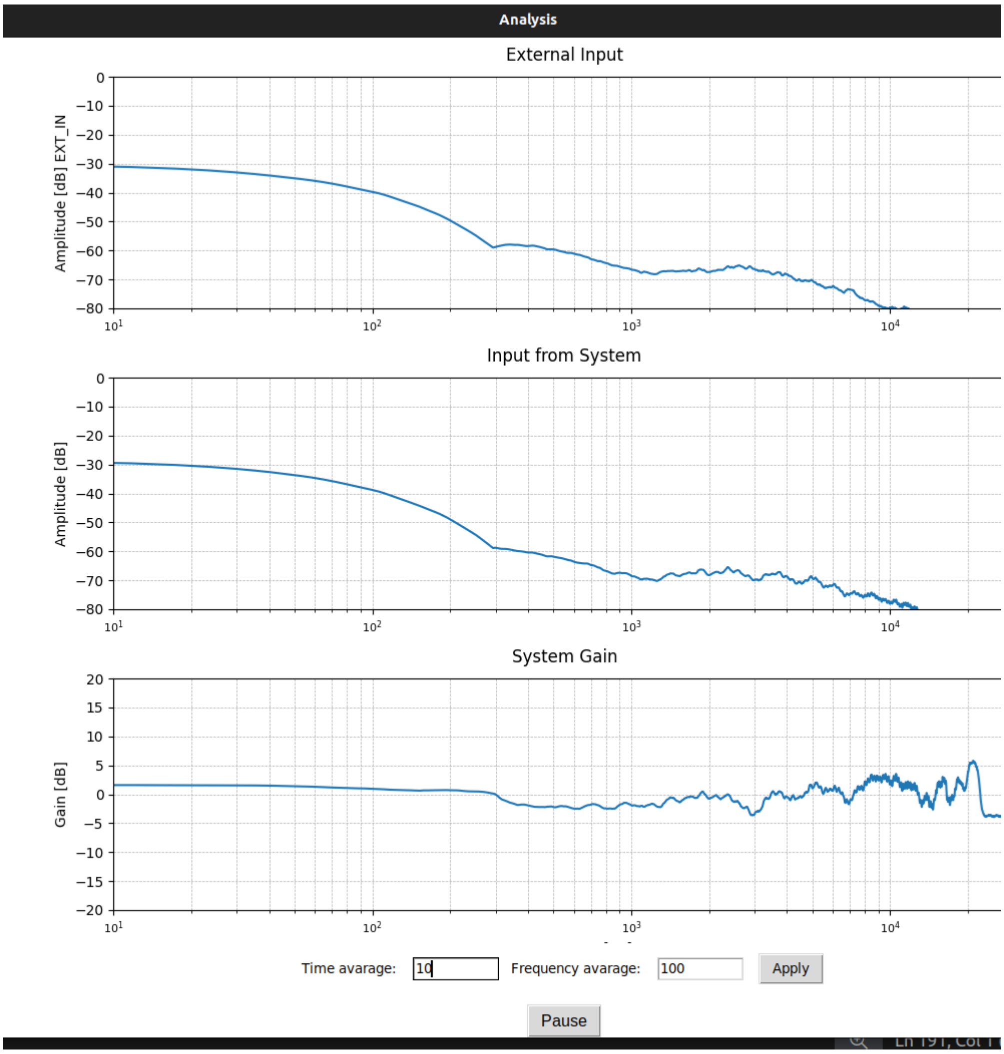

Figure 5.25: FT page, using music instead of pink noise

Using music and ignoring the glitch issue, I obtained the results shown in Figures 5.24 and 5.25. It was necessary to apply some averaging to make the results more understandable. Even so, the system performed quite well: although both the **External Input** and **Input from System** signals are constantly changing, the System Gain plots show only minor variations— which makes sense, since the analyzed system remains unchanged despite the music content varying. Moreover, because these signals are synchronized using the delay adjustment, the averaged results from different signal frames consistently represent the same slice of music.

Despite significant issues such as signal desynchronization (which causes glitch effects) and GUI malfunctions (which can freeze the program), the developed software solution works well.

Chapter 6

Conclusions

After more than four months of work on this project, not all objectives were achieved. The overall goal of developing a software-based solution for acoustic analysis and correction was not fully completed. Several tools are still pending implementation (such as phase analysis, RT60 measurement, waterfall diagram, among others), and some of the tools already implemented could benefit from improvements (such as the **GUI** or the filters used). In addition, there are two major issues that must be resolved for the software to be fully usable: GUI freezing and signal desynchronization in the **DSP** window.

Even so, if we ignore the two main issues, the rest of the developed program can be considered a success. As shown in the final test section, it is functional and performs quite well. Moreover, a solid foundational architecture was developed, which makes it easy to add or modify tools and functionalities in the future. This extensible and modular design is, in my opinion, one of the most solid and valuable achievements of the project.

On the other hand, if I—or someone else—were to continue this project with another four months of development time, I would recommend starting the program from scratch. The same structure and toolset could be implemented following the same logic, but with a serious rethinking of the GUI system (to avoid unexplained bugs and improve code organization) and the signal path management (to eliminate the glitch effect). If these two key issues were properly addressed in a new version of the program, it would result in a very strong foundation to keep building on—allowing for easier expansion, refinement of existing tools, and unlocking the full potential of the project.

From a market perspective, this project—if fully developed with all its planned functionalities working correctly—could be a strong candidate for commercial release. As discussed in the **Background and/or status of the matter** chapter, there are already several existing solutions that offer many of the tools proposed in this project. However, these alternatives are often either too expensive (because they are hardware-based), or they lack flexibility, such as not integrating analysis and correction in the same environment, or offering fewer tools overall.

If additional features were implemented beyond the current scope—such as multichannel system analysis and correction (for stereo setups or distributed measurement points) or a microphone calibration system—this solution could not only remain at a much more competitive price point compared to hardware-based options, but also become one of the most complete and practical software tools available on the market.

Ultimately, beyond the technical aspects, this project has represented a valuable learning journey. It has provided deep insight into real-time audio processing, signal analysis, and software design challenges—especially considering that I had never developed a graphical user interface before. Facing this for the first time added an extra layer of complexity to the project, but also made the experience more enriching. The process of solving unexpected issues, iterating on solutions, and building a complex tool from scratch has contributed significantly to my personal and professional growth. Regardless of its current limitations, the project lays down a strong conceptual and technical foundation that can serve as the starting point for more advanced developments in the future—and that, in itself, is a success.

Acknowledgements

I would like to thank my project director for the guidance and support provided throughout the development of this work.

I am also grateful to the Societat Coral Obrera la Glòria Sentmenatenca for allowing me to use their theater for the final testing, and to IMESDE, Integració, Distribució i Enginyeria Escènica, S.L. for lending me part of their equipment and offering flexibility in my schedule, which allowed me to dedicate more time to this project.

Finally, I want to thank my parents and friends for their constant support throughout the entire process.

Bibliography

[24] *API Documentation — python-sounddevice, version 0.5.1*. [Online; accessed 31. May 2025]. Oct. 2024. URL: https://python-sounddevice.readthedocs.io/en/0.5.1/api/index.html.

[25a] *DBX 231s*. [Online; accessed 26. May 2025]. May 2025. URL: https://www.thomann.es/dbx_231s.htm.

[25b] *Dirac Live - Digital Room Correction software*. [Online; accessed 15. Jun. 2025]. June 2025. URL: https://www.dirac.com/live.

[25c] *Matplotlib documentation — Matplotlib 3.10.3 documentation*. [Online; accessed 2. Jun. 2025]. May 2025. URL: https://matplotlib.org/stable/index.html.

[25d] *NumPy user guide — NumPy v2.2 Manual*. [Online; accessed 7. Jun. 2025]. Jan. 2025. URL: https://numpy.org/doc/stable/user/index.html#user.

[25e] *Signal processing (scipy.signal) SciPy v1.15.3 Manual*. [Online; accessed 26. May 2025]. May 2025. URL: https://docs.scipy.org/doc/scipy/reference/signal.html.

[25f] *Smaart Home*. [Online; accessed 25. May 2025]. May 2025. URL: https://www.rationalacoustics.com/pages/smaart-home.

[25g] *SoundID Reference - Speaker & Headphone Calibration*. [Online; accessed 15. Jun. 2025]. June 2025. URL: https://www.sonarworks.com/soundid-reference.

[25h] *tkinter — Python interface to Tcl/Tk*. [Online; accessed 2. Jun. 2025]. June 2025. URL: https://docs.python.org/3/library/tkinter.html.

[And23] Karen Anderson. "Audix TM1 Plus Now Available". In: *Rational Acoustics* (Sept. 2023). URL: https://www.rationalacoustics.com/blogs/news/audix-tm1-plus-now-available.

[Des25] Presto Web Design. "Trinnov | Digital Room Correction, Monitoring and Immersive Sound Solutions". In: *Trinnov* (June 2025). URL: https://www.trinnov.com.

[Mul24] John Mulcahy. *REW room acoustics and audio device measurement and analysis software*. [Online; accessed 15. Jun. 2025]. Dec. 2024. URL: https://www.roomeqwizard.com/index.html.

[UPC25] UPC. *The School of Industrial, Aerospace and Audiovisual Engineering of Terrassa*. Spring of 2025. URL: https://eseiaat.upc.edu/en?set_language=en.

Annex 1: Budget