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

BACHELOR FINAL THESIS

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

Summary

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

List of figures

List of tables

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

Chapter 1

Introduction

1.1 Objecte

Resultat final que es vol aconseguir. En aquest cas, l'objecte d'aquesta plantilla és donar les pautes d'estructura i contingut de la Memòria del TFE.

El cos tant d'aquest document "Memòria" com dels altres documents integrants del TFE (Pressupost, Annexos i Plec de condicions) serà amb lletra Times New Roman o Arial d'una mida d'11 punts, marge lateral esquerra de 3 cm, dret de 2,5, superior i inferior de 2,5 i espaiat senzill.

L'alumne/a ha de revisar l'ortografia i gramàtica de tots els documents del TFE; ha d'utilitzar les unitats del Sistema Internacional; ha d'utilitzar un nombre coherent de decimals; i ha d'identificar els eixos dels gràfics inclosos al llarg del text.

Es recomana que la memòria no superi una extensió màxima de 60-70 pàgines. Tant les taules com figures han d'estar enumerades i tenir un títol. Si s'han obtingut d'algun altre document consultat, s'haurà de dir la font d'on s'ha tret [UPC25].

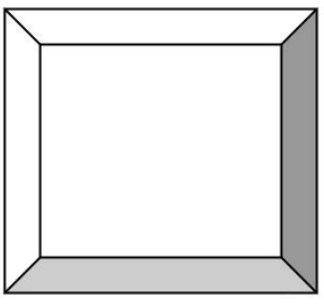


Figure 1.1: Imatge d'exemple

Table 1.1: Risks assessment

1	X2	X	X
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...

1.2 Abast

Paquets de treball i lliurables necessaris per arribar a la solució.

1.3 Requeriments

O especificacions bàsiques. Restriccions sobre la solució final.

1.4 Justificació

Plantejament de la necessitat del treball des d'una visió global i aproximant-lo a una visió més específica. Serveix per centrar i contextualitzar el treball.

1.4.1 Subsection

1.4.1.1 superseccion

divisiones.py

```
"""
Biblioteca con definiciones importantes para la división de números.
Se incluyen las funciones divideSiDivisible() y cocienteModulo().
"""

def divideSiDivisible(nume, deno):
"""
Si nume es divisible por deno, devuelve la división
entera. Si no lo es, devuelve None.
"""

if not nume % deno:
return nume // deno
```

```
14
15
16 def cocienteModulo(nume, deno):
17 """
18 Devuelve el conciente entero y el resto de
19 la divisi
20 ón entera (mod) de dos números.
21 """
22
23 return nume // deno, nume % deno
24
```

src/divisions.py

```
6 def divideSiDivisible(nume, deno):
7         """
8         Si nume es divisible por deno, devuelve la división
9         entera. Si no lo es, devuelve None.
10        """
11
12        if not nume % deno:
13        return nume // deno
```

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 [DONE]

Smaart, an acronym for *System Measurement Acoustic Analysis Real-time Tool* [25e], 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.

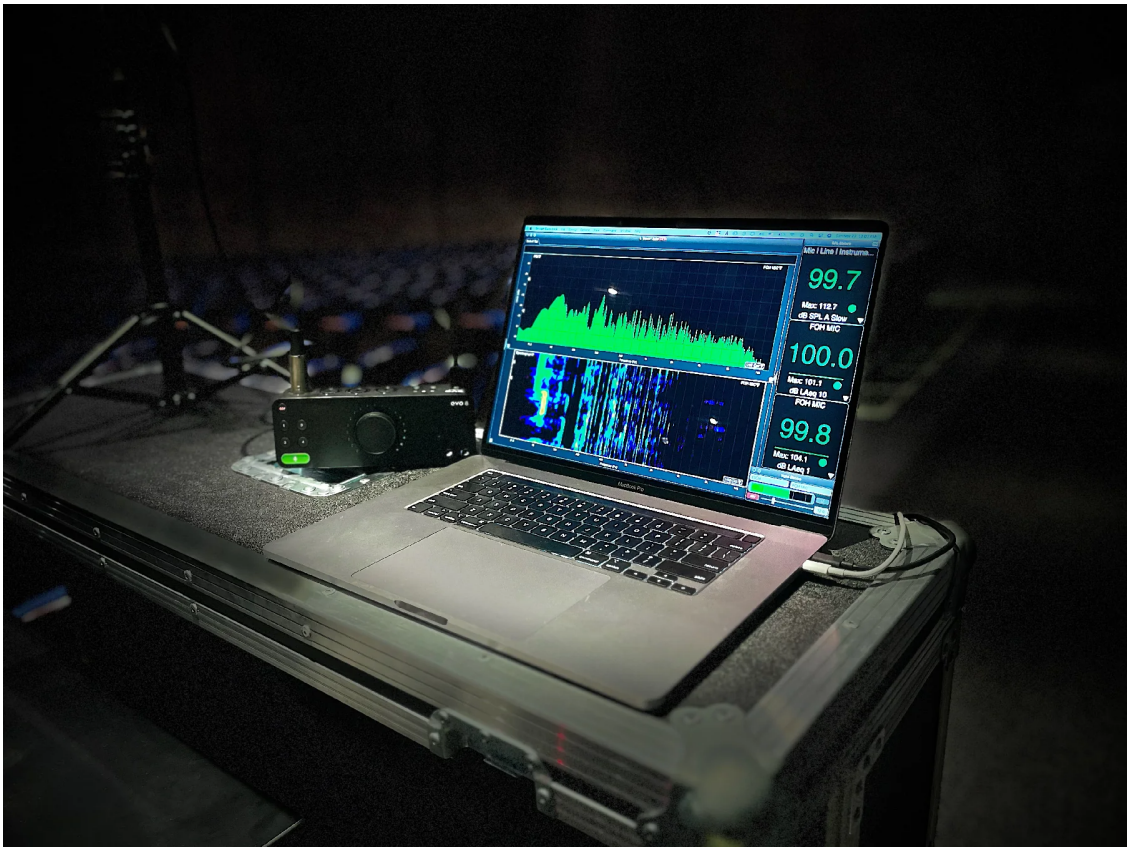


Figure 2.1: Notebook using Smaart, where we can see some of the tools it includes. The notebook is connected to the EVO 8 (a USB interface that acts as an external sound card).

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

Home correction solution ——————————————————————

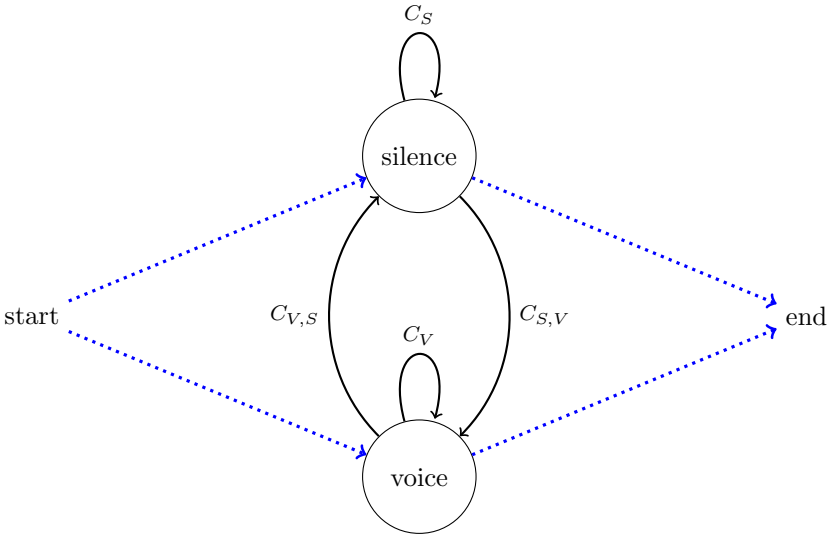
2.3 REW

Free software for room measurement ——————————————————————

2.4 Trinnov

Hardware base solution with correction ——————————————————————-

********** Exemple d'esquema **********



Chapter 3

Methodology, consideration and decision on alternative solutions [DONE]

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.

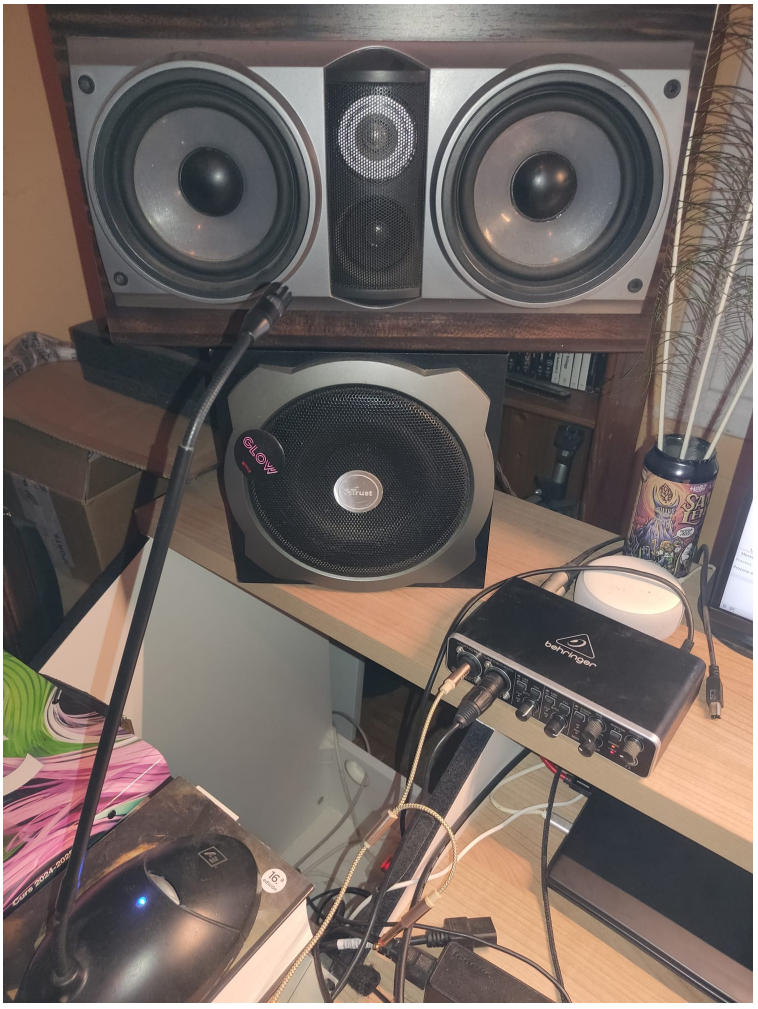


Figure 3.1: Home setup showing the sound card, the speaker, and the microphone.

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 [DONE]

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[25f] library for the window and page system, and the Matplotlib[25b] 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:

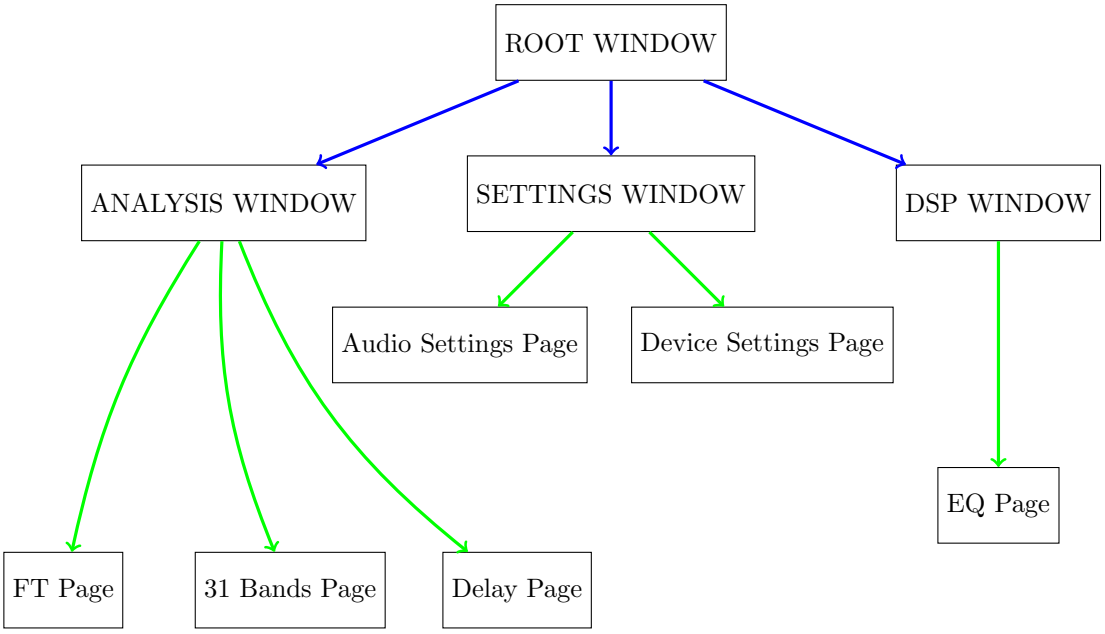


Figure 4.1: GUI window schematic

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

```
"""
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.

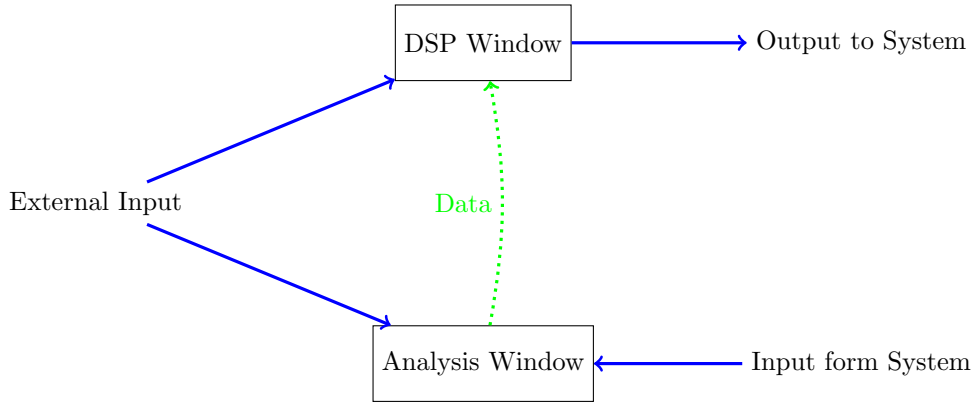


Figure 4.2: Basic representation of the required signal path

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.

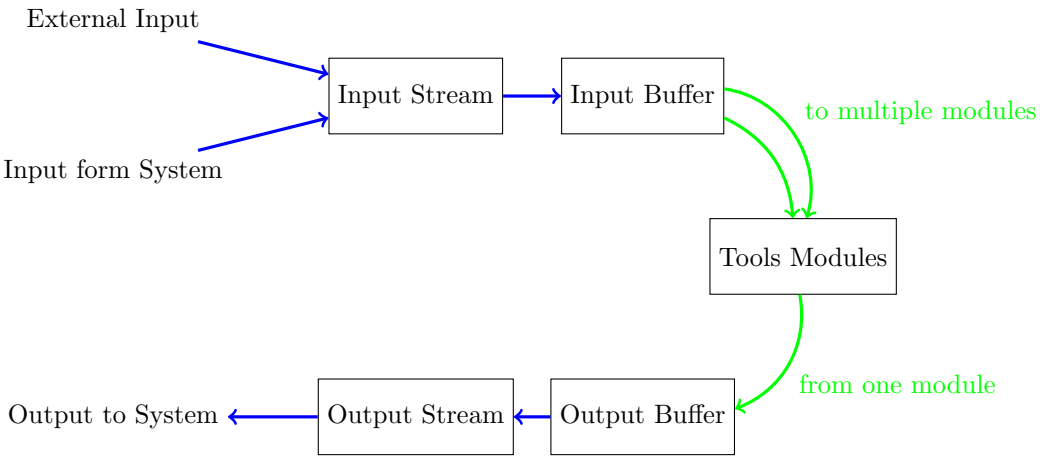


Figure 4.3: Stream management

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.

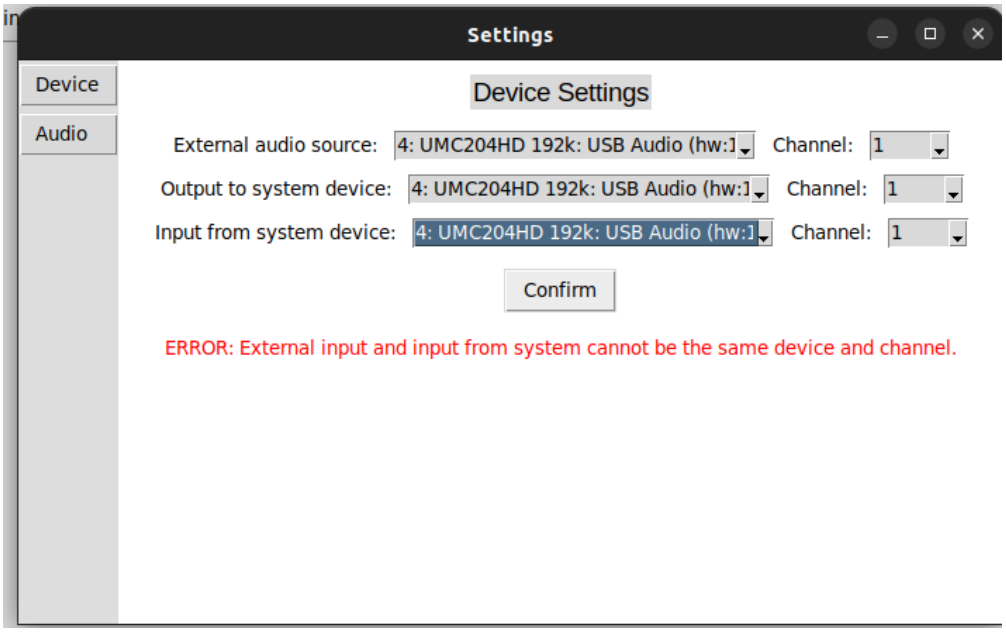


Figure 4.4: Device Settings page showing a red message because the user selected an invalid configuration.

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.

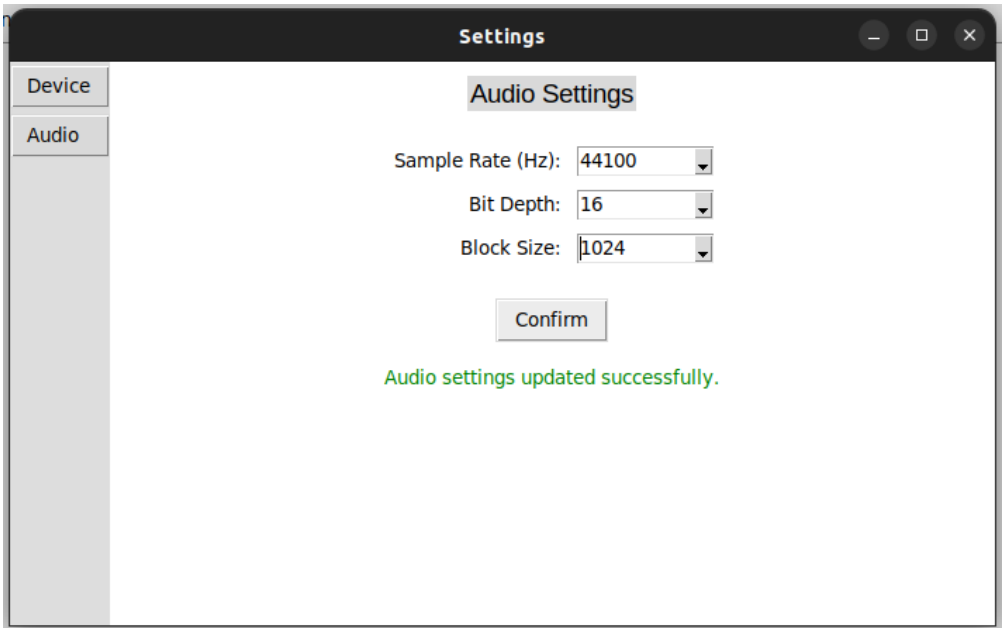


Figure 4.5: Audio Settings page showing a green message.

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

```
"""
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**[25c] and **SciPy**[25d] 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

```
        """
        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")
```

```
14                 return
15
```

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.

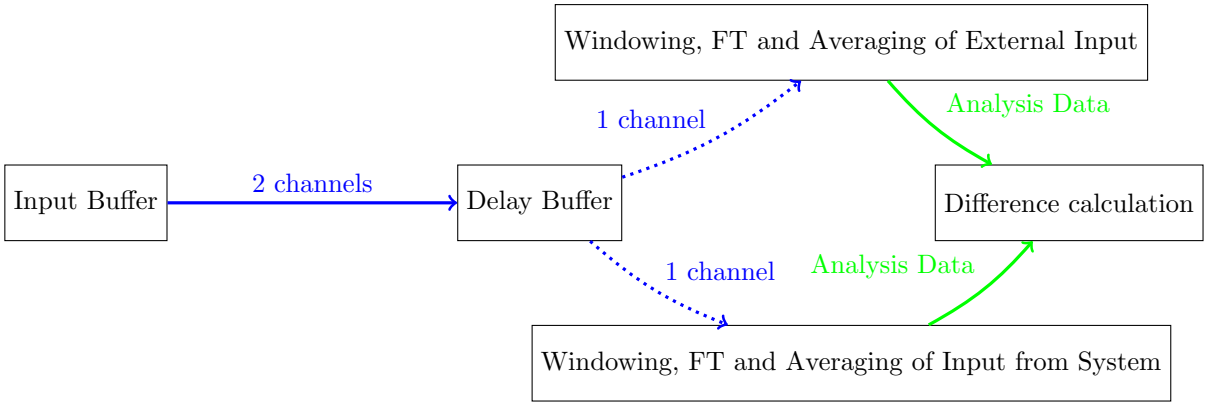


Figure 4.7: Diagram of the architecture of the FT page

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.

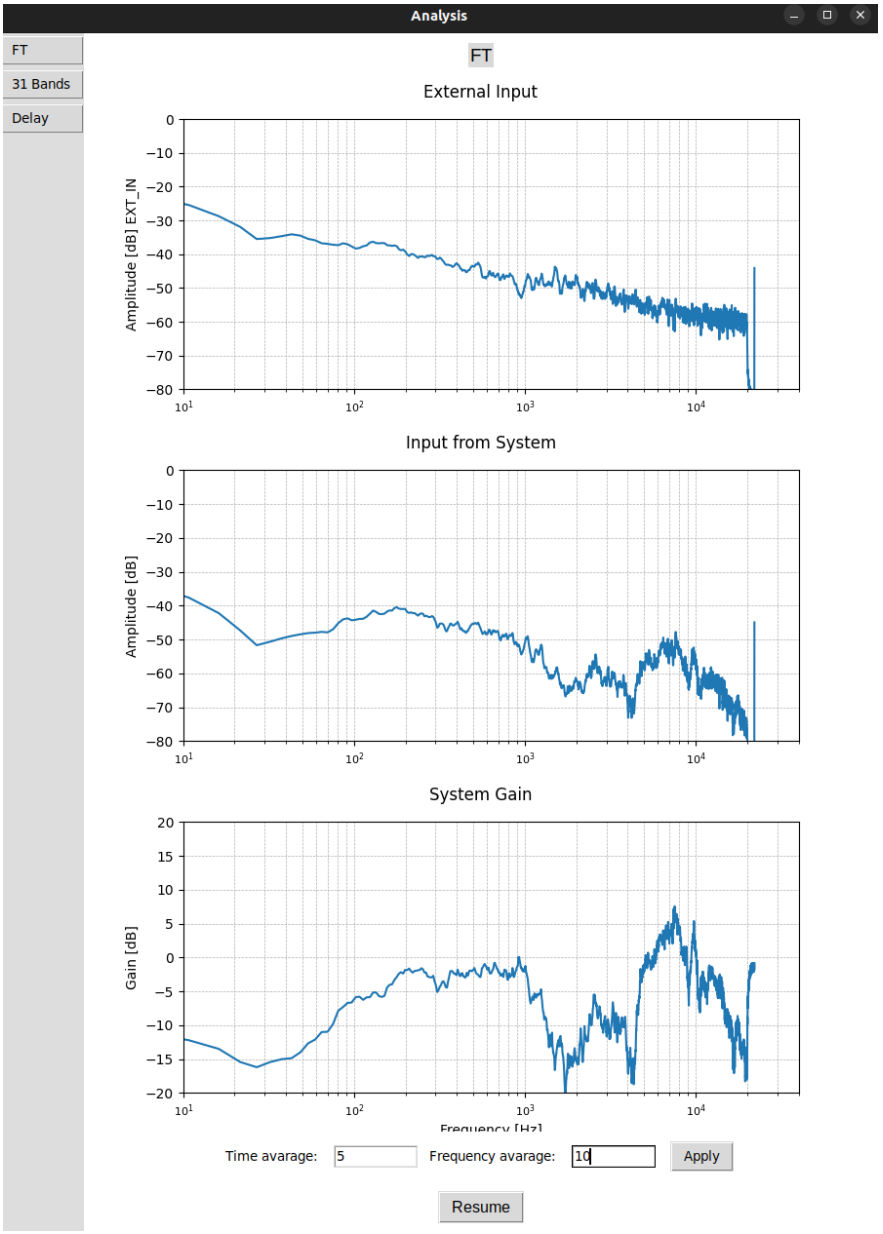


Figure 4.8: Analysis window - FT page.

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

```
"""
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.

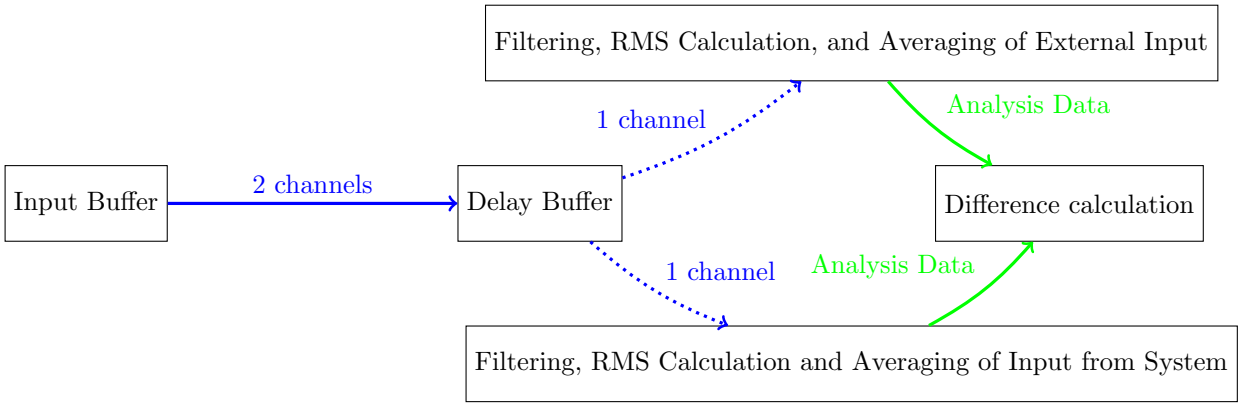


Figure 4.10: Diagram of the architecture of the RTA page

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 [25d], 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).

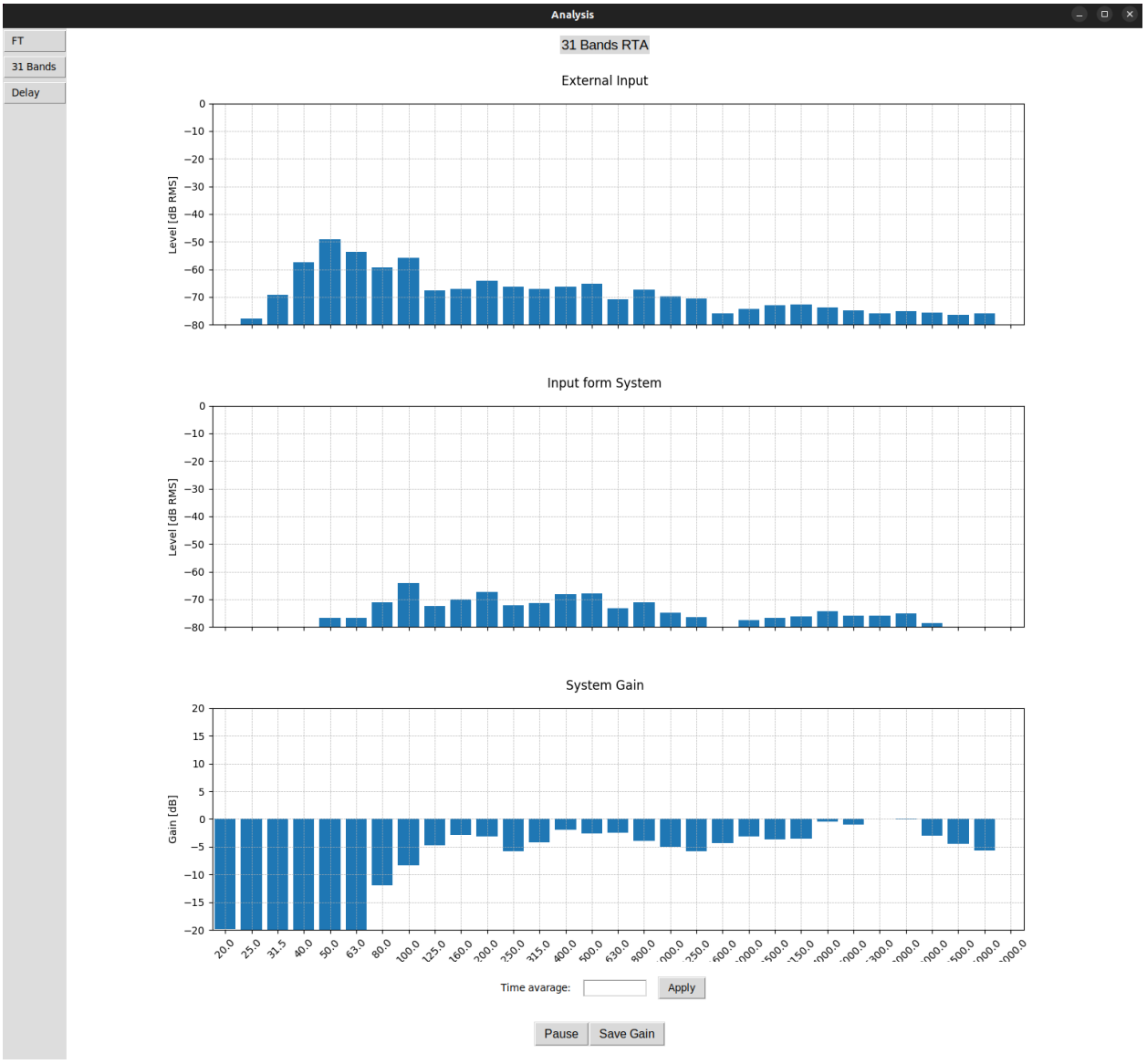


Figure 4.13: Analysis window - RTA page.

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.

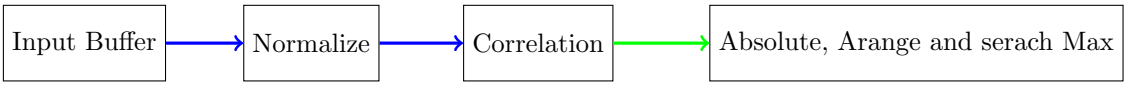


Figure 4.14: Diagram of the architecture of the Delay page

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.

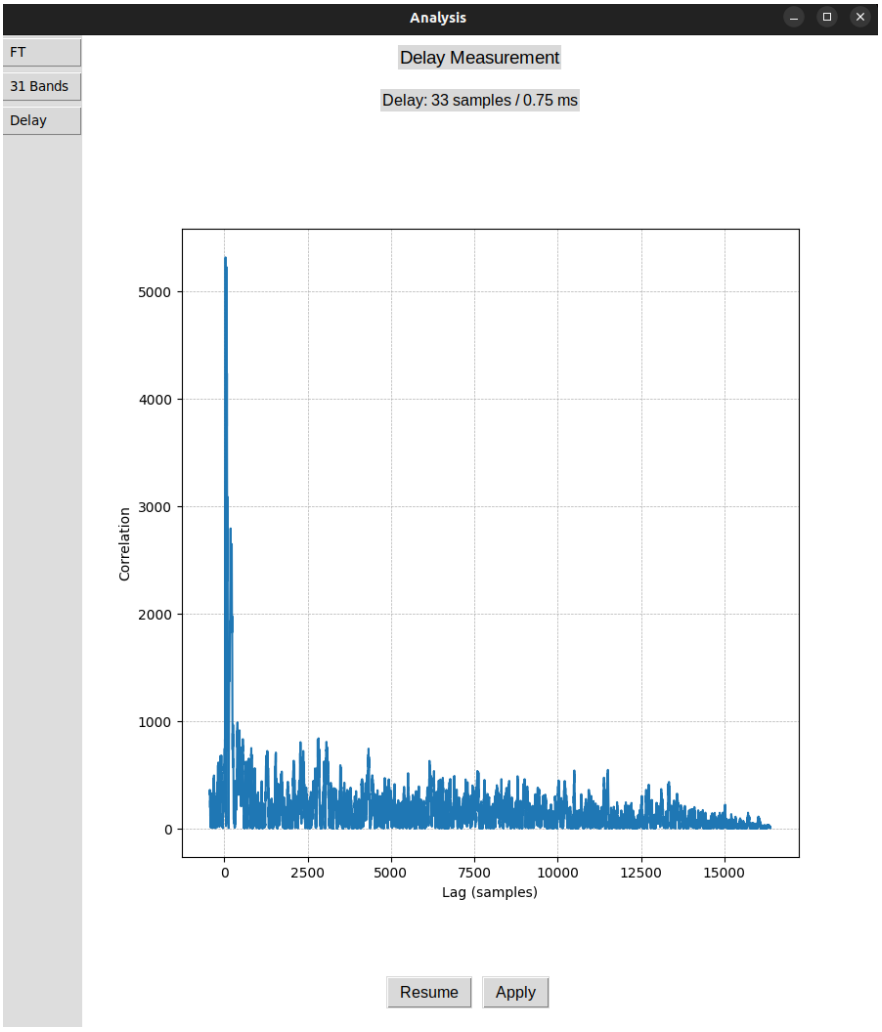


Figure 4.15: Analysis window - Delay page.

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.

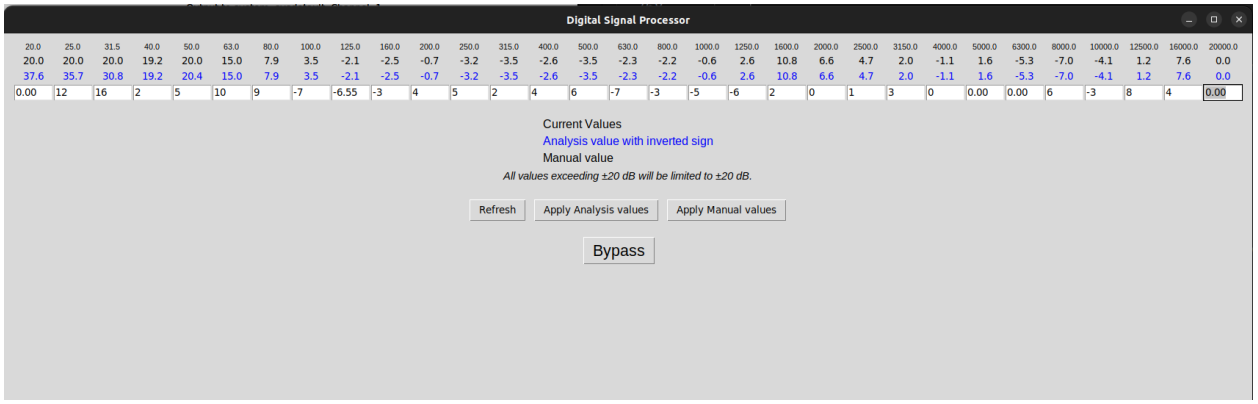


Figure 4.16: Correction window - DSP.

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

```
"""
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**.

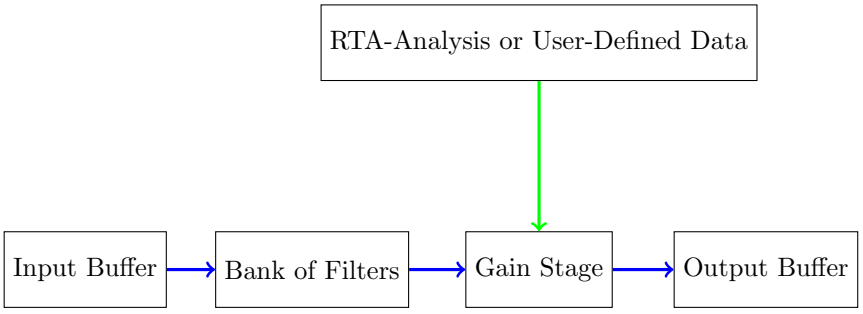


Figure 4.18: Diagram of the architecture of the EQ module

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

About the final program results

There ar not just things to finish or implement, also, it is nedded to solve some actual problems. Most important of them are:

- **Filter Algorithms:**
- **User Limitations:** For example, since we are using the Sounddevice library and managing both inputs as a single input stream, we have the limitation that both channels must come from the same sound card. However, the program currently allows the selection of different sound cards for each input channel.

5.1 Final tests

About testing program on real situations

5.2 User experiance

About user experiance.

Chapter 6

Conclusions

Fa la funció de síntesi final i s'elabora a partir de la interpretació dels resultats assolits. La conclusió sol ser breu i s'ha de relacionar directament amb els objectius del treball.

També s'hi han d'incloure les recomanacions de continuació del treball i la planificació i programació del treball futur proposat.

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

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

[25d] *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.

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

[25f] *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.

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