

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica e delle Tecnologie digitali

---

Sviluppo di un sistema di audio immersivo per applicazioni di sicurezza alla guida

Development of an immersive audio system for driving safety applications

Tesi di Laurea di:
Chiara Menzietti

Relatore:
Dott. Alessandro Terenzi

Correlatrici:
Dott.ssa Valeria Bruschi
Prof.ssa Stefania Cecchi

A. A. 2025/2026



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica e delle Tecnologie digitali

---

Sviluppo di un sistema di audio immersivo per applicazioni di sicurezza alla guida

Development of an immersive audio system for driving safety applications

Tesi di Laurea di:
Chiara Menzietti

Relatore:
Dott. Alessandro Terenzi

Correlatrici:
Dott.ssa Valeria Bruschi
Prof.ssa Stefania Cecchi

A. A. 2025/2026

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica e delle Tecnologie digitali
Via Brecce Bianche – 60131 Ancona (AN), Italy

Ringraziamenti

Desidero esprimere il mio più sincero ringraziamento ai miei professori: i loro insegnamenti sono stati e saranno sempre per me fonte di ispirazione, la loro passione è diventata la mia.

Una menzione particolare va al mio relatore, il Dott. Alessandro Terenzi e alle mie correlatrici, la Dott.ssa Valeria Bruschi e la Prof.ssa Stefania Cecchi.

Vi ringrazio per la costante disponibilità, i preziosi consigli e il fondamentale supporto datomi nel realizzare questo lavoro di tesi.

Ancona, Ottobre 2026

Chiara Menzietti

Abstract

In driving scenarios, acoustic warnings play an important role in reporting dangerous conditions without diverting the driver's visual attention from the road. However, most warning systems used in vehicles today are limited to communicating the presence of a hazard or fault, without providing any indication of its spatial origin. This thesis work builds upon a communication system based on the MQTT protocol, which is capable of remotely controlling the playback of audio tracks via a plugin developed for a specific real-time processing environment real-time audio processing environment. This system only allows deciding which sound to play, returning a signal without any spatial information.

The objective of this thesis was to extend the system so that it could reproduce spatialized audio signals using techniques based on head-related transfer functions (HRTFs) filtering, with the aim of developing a warning system for driving safety. The system was designed to indicate not only which sound should be played, but also where the sound should come from in the three-dimensional space around the driver.

The work was developed in three main phases. First, a literature review was conducted regarding 3D audio and directional acoustic warnings in the automotive field. Based on these theoretical references, the extension of the existing communication system was designed and implemented as a new two-channel parallel binaural filtering stage based on FIR convolution.. The MQTT command was extended with an azimuth field to convey the requested direction, which is mapped via a circular angular distance rule onto a discrete set of reference directions; the corresponding HRTF pairs (taken from the MIT Media Lab KEMAR database) are preloaded into memory to comply with the real-time processing constraint imposed by the plugin's multithreaded architecture.

In an initial phase, the tests validated, the accuracy of the FIR filtering and the loading of the HRTFs on the two channels, as well as the correct end-to-end operation of the extended MQTT communication. Subsequently, a perception test conducted with the laboratory's driving simulator showed that the system is able to correctly communicate the direction of origin of a hazard in all tested cases, with a systematic confusion between the front and rear directions consistent with the known limitations of non-individualized HRTFs.

The work therefore demonstrates that it is possible to extend an existing audio communication system with a real-time HRTF spatialization stage, obtaining a system capable of conveying effectively perceivable directional information, which is particularly reliable in the lateral distinction of a hazard's origin.

Sommario

Negli scenari di guida, gli avvisi acustici rivestono un ruolo importante nel segnalare condizioni di pericolo senza distogliere l'attenzione visiva del conducente dalla strada. La maggior parte dei sistemi di allerta oggi impiegati nei veicoli si limita tuttavia a comunicare la presenza di un pericolo o guasto, senza fornire alcuna indicazione sulla sua provenienza spaziale.

Questo lavoro di tesi parte da un sistema di comunicazione, basato sul protocollo MQTT, in grado di comandare da remoto la riproduzione di tracce audio tramite un plugin sviluppato per uno specifico ambiente di elaborazione audio in tempo reale. Tale sistema permette di decidere unicamente quale suono riprodurre, restituendo un segnale privo di qualunque informazione spaziale. L'obiettivo della tesi è stato estendere il sistema affinché fosse in grado di riprodurre segnali audio spazializzati tramite ttecniche basate sul filtraggio con HRTF (Head Related Transfer Function), da impiegare come strumento di allerta per la sicurezza alla guida. Il sistema è stato progettato per indicare non solo quale suono debba essere riprodotto, ma anche da dove il suono debba provenire nello spazio tridimensionale attorno al guidatore.

Il lavoro si è sviluppato in tre fasi principali. In primo luogo, è stata condotta una rassegna della letteratura relativa all'audio 3D e agli avvisi acustici direzionali in ambito automotive. Sulla base di questi riferimenti teorici, è stata progettata l'estensione del sistema di comunicazione esistente, realizzata come un nuovo stadio di filtraggio binaurale a due canali paralleli, basato su convoluzione FIR: il comando MQTT è stato esteso con un campo di azimuth per veicolare la direzione richiesta, mappata tramite una regola di distanza angolare circolare su un insieme discreto di direzioni di riferimento, le cui coppie di HRTF (tratte dal database KEMAR del MIT Media Lab) vengono precaricate in memoria per rispettare il vincolo di elaborazione in tempo reale imposto dall'architettura multithread del plugin.

I test condotti hanno validato, in una prima fase, la correttezza del filtraggio FIR e del caricamento delle HRTF sui due canali, e il corretto funzionamento end-to-end della comunicazione MQTT estesa. Successivamente un test di percezione condotto con il simulatore di guida del laboratorio ha mostrato che il sistema è in grado di comunicare correttamente il lato di provenienza di un pericolo nella totalità dei casi testati, pur con una confusione sistematica tra le direzioni anteriore e posteriore coerente con i limiti noti delle HRTF non individualizzate.

Il lavoro dimostra quindi che è possibile estendere un sistema di comunicazione audio esistente con uno stadio di spazializzazione HRTF in tempo reale, ottenendo un

sistema in grado di veicolare un'informazione direzionale effettivamente percepibile, particolarmente affidabile nella distinzione laterale della provenienza di un pericolo.

Indice

1. Introduzione **1**

2. Stato dell'arte: audio immersivo per la sicurezza alla guida **3**

2.1. Fondamenti dell'audio 3D e delle HRTF 3

2.1.1. Dal semplice stereo alla spazializzazione tridimensionale . . . 3

2.1.2. Localizzazione uditiva 3

2.1.3. Sintesi binaurale e limiti delle HRTF generiche 4

2.1.4. Audio 3D tramite altoparlanti: il problema del crosstalk . . . 5

2.2. Modellazione dell'ambiente acustico 6

2.2.1. Riverbero 6

2.2.2. Effetto della distanza 7

2.2.3. Effetto Doppler 7

2.2.4. Assorbimento dell'aria e occlusione degli oggetti 7

2.2.5. Routing del segnale e virtual speaker 7

2.3. Perché la direzione conta per la sicurezza: dagli avvisi semantici agli avvisi spazializzati 8

2.3.1. Avvisi direzionali basati sul contenuto semantico: il caso della guida automatizzata 8

2.3.2. Avvisi acustici spazialmente predittivi e attenzione cross-modale 10

2.4. Audio immersivo object-based per l'HMI automotive 11

2.5. Fattibilità implementativa: rendering HRTF/BRIR in abitacolo reale 13

2.6. Sintesi e posizionamento della tesi 15

3. Sistema implementato **17**

3.1. Panoramica architetturale del sistema esteso 17

3.2. Principio di spazializzazione: filtraggio FIR binaurale 18

3.3. Mappatura delle direzioni: dalle misurazioni continue a un insieme discreto 19

4. Implementazione della comunicazione MQTT **21**

4.1. Obiettivi del sistema di comunicazione 21

4.2. Panoramica architetturale del sistema 22

4.3. Il protocollo MQTT: richiami essenziali 23

4.3.1. Broker, client, topic e payload 23

4.3.2. Livelli di QoS 23

4.4. Scelte progettuali 24
4.4.1. Linguaggio: Standard C++17 ISO 24
4.4.2. Piattaforma di compilazione a 32 bit 24
4.4.3. Librerie 25
4.4.4. Architettura multithread 26
4.5. Il meccanismo di comunicazione: dalla callback al plugin 26
4.5.1. Perché il plugin non riceve MQTT direttamente 27
4.5.2. La classe MqttCb e il meccanismo di override 27
4.5.3. Il puntatore self e l'accoppiamento callback–plugin 28
4.5.4. pendingAudio: il punto di sincronizzazione fra thread 28
4.5.5. Flusso reale del messaggio 28

5. Implementazione della spazializzazione **31**
5.1. Requisiti di progettazione 31
5.2. Struttura del plugin NuTech: analisi del codice 32
5.2.1. Costruttore e stato iniziale 32
5.2.2. Caricamento dei file audio in LEPlugin_Init() 33
5.2.3. Avvio del thread MQTT 33
5.2.4. Il ciclo di elaborazione audio: LEPlugin_Process() 34
5.2.5. Terminazione: LEPlugin_Delete 34
5.3. Vincolo di tempo reale: precaricamento delle HRTF 34
5.4. Evoluzione rispetto al sistema base 35
5.4.1. I file audio di riferimento e le costanti N1, N2 35
5.4.2. Numero di tap del filtro (firTaps) 36
5.5. Librerie e ambiente di sviluppo 36
5.5.1. Migrazione alla compilazione a 64 bit 36
5.5.2. Integrazione di Intel IPP 38
5.6. Il pattern GetSize/Init/Process applicato al filtro FIR binaurale . . 39
5.6.1. Variabili membro del plugin 40
5.6.2. Allocazione in LEPlugin_Init() 41
5.7. Caricamento delle HRTF: la funzione LoadHRTF 41
5.8. Precaricamento e cache delle direzioni 42
5.9. Selezione della direzione in tempo reale: ApplyHRTF 43
5.9.1. Perché una delay line esplicita 44
5.10. Estensione del protocollo MQTT: il campo azimuth 45
5.11. Il ciclo di elaborazione a due canali 45
5.12. Gestione della memoria 46
5.13. Una modalità di test dedicata: modalitaFileRead 47

6. Test e risultati **49**
6.1. Validazione del sistema di comunicazione 49
6.1.1. Configurazione di Mosquitto e MQTTX 49

6.1.2. Verifica della comunicazione MQTT 51
6.2. Validazione dell'algoritmo di spazializzazione 54
6.2.1. Verifica del filtraggio FIR 54
6.2.2. Verifica del filtraggio con le HRTF a due canali 60
6.3. Test sperimentali 61
6.3.1. Dalla difficoltà di percezione alla spazializzazione funzionante 61
6.3.2. Test di percezione della direzione: validazione con il simulatore di guida 65

7. Conclusioni e sviluppi futuri **73**
7.1. Conclusioni 73
7.2. Limiti del lavoro 74
7.3. Sviluppi futuri 75

A. Codice Sorgente del Plugin **77**
A.1. File `Plugin.h` 77
A.2. File `Plugin.cpp` 81

Elenco delle figure

2.1. Schema di misurazione delle HRTF (adattato da [1]). 4
2.2. Sintesi binaurale (adattato da [1]). 5
2.3. Fenomeno del crosstalk (adattato da [1]). 6
2.4. Configurazione surround a 5 canali (adattato da [2]). 12
2.5. Disposizione degli altoparlanti del veicolo. I cerchi indicano le dimensioni e la posizione approssimative degli altoparlanti , quelli evidenziate in verde sono aggiuntivi rispetto al sistema originale (adattato da [3]). 14

3.1. Struttura estesa del plugin: dal comando MQTT alla spazializzazione binaurale su due uscite indipendenti. 18

4.1. Architettura del sistema di comunicazione 22
4.2. Impostazione dello standard del linguaggio C++17 ISO nelle proprietà del progetto Visual Studio (Configurazione Debug, piattaforma Win32). 24
4.3. Configurazione della variabile d'ambiente PATH nelle proprietà di debug 26
4.4. Flusso reale del messaggio 29

5.1. Separazione di fase di inizializzazione e fase di esecuzione 35
5.2. Configurazione PATH per la configurazione Debug a 64 bit. 38
5.3. Setup per i test in modalitaFileRead 48

6.1. Mosquitto avviato in modalità local-only 50
6.2. Configurazione della connessione MQTTX al broker locale 50
6.3. Sottoscrizione del client MQTTX al topic sensori/comandi 51
6.4. Pubblicazione dei due messaggi di test 52
6.5. Panoramica dell'interfaccia MQTTX utilizzata per i test 52
6.6. Confronto visivo, tramite Viewer di NuTech, tra il segnale di ingresso (FileRead) e il segnale in uscita dal nuts (FileWrite), con filtro identità. 56
6.7. Differenza ingresso–uscita (blocco Mix) con filtro identità: linea piatta a zero. 58
6.8. Differenza ingresso–uscita (blocco Mix) con delta ritardata di 9 campioni: segnale residuo dovuto al disallineamento temporale. 58
6.9. Differenza tra Output 0 e Output 1 con firCoeffsL[0]=1.0 e firCoeffsR[0]=1.0: canali identici, differenza nulla. 59

6.10. Differenza tra Output 0 e Output 1 con firCoeffsL[0]=1.0 e firCoeffsR[9]=1.0: canali distinti, differenza non nulla. 60
6.11. Uscite Output 0 (L) e Output 1 (R) con HRTF a 90°: stessa struttura temporale del rumore, ampiezze diverse (particolare ingrandito a destra). 62
6.12. Confronto in MATLAB tra i due canali salvati separatamente: forme d'onda (ampiezze diverse), spettri in frequenza (colorazione diversa tra i due canali), differenza temporale non nulla. 63
6.13. Configurazione di NuTech impiegata per l'ascolto con cuffie Bluetooth: uscita del plugin salvata tramite FileWrite. 64
6.14. Configurazione del test al simulatore 67
6.15. Setup Nutech per i test finali 68

Elenco delle tabelle

4.1. Librerie utilizzate 25

6.1. Soggetto 1 : direzione reale vs. stimata (gradi). 69
6.2. Soggetto 2 : direzione reale vs. stimata (gradi). 69
6.3. Soggetto 3 : direzione reale vs. stimata (gradi). 70
6.4. Classificazione degli errori commessi nei tre test. 70

Capitolo 1.

Introduzione

Durante la guida, l'attenzione del conducente è una risorsa limitata, costantemente sollecitata da stimoli visivi, motori e cognitivi. In questo contesto, i sistemi di allerta acustica rivestono un ruolo importante: un avviso sonoro può segnalare una condizione di pericolo senza richiedere che il guidatore distolga lo sguardo dalla strada. Sebbene i sistemi acustici tradizionali si limitino a segnalare la presenza di un pericolo o un guasto, senza fornire alcuna indicazione su dove esso si trovi rispetto al guidatore, le soluzioni più recenti hanno iniziato ad integrare anche informazioni spaziali

Un segnale acustico capace di indicare direzionalmente la provenienza di un rischio (un veicolo che si avvicina lateralmente, un ostacolo posteriore, un guasto localizzato in una specifica area del veicolo) fornisce un'informazione aggiuntiva che un avviso generico non è in grado di veicolare.

Il presente lavoro di tesi si colloca in questo ambito, e nasce a partire da un sistema di comunicazione realizzato in una fase preliminare del lavoro di tirocinio: un meccanismo basato sul protocollo MQTT (Message Queuing Telemetry Transport)[4] che permette di comandare da remoto, tramite un plugin dedicato, la riproduzione di tracce audio all'interno di un ambiente di elaborazione in tempo reale.

Tale sistema, per quanto funzionale, si limita a decidere quale traccia audio riprodurre, restituendo un segnale privo di qualunque informazione spaziale: la sua architettura (la comunicazione asincrona tramite callback, la separazione tra thread di rete e thread audio, i vincoli di tempo reale nell'elaborazione dei campioni) costituisce però una base solida su cui costruire un sistema più complesso.

L'obiettivo di questa tesi è infatti estendere quel sistema affinché sia in grado di riprodurre segnali audio immersivi, spazializzati tramite tecniche di filtraggio con funzioni specifiche chiamate HRTF (Head Related Transfer Function), da utilizzare come strumento di allerta per la sicurezza durante la guida. Non si tratta più, quindi, soltanto di decidere quale suono riprodurre, ma anche da dove, nello spazio tridimensionale attorno al guidatore, quel suono debba provenire.

Per raggiungere questo obiettivo, il lavoro affronta tre problemi principali:

- comprendere i fondamenti teorici della spazializzazione audio tramite HRTF;

- progettare un'architettura che estenda il sistema di comunicazione preesistente con uno stadio di filtraggio binaurale, senza violare i vincoli di elaborazione in tempo reale già imposti dal contesto audio;
- realizzare tale estensione, integrando le librerie necessarie al filtraggio in tempo reale e un database di misurazioni HRTF reali, e verificarne il corretto funzionamento.

La tesi è organizzata come segue.

Il Capitolo 2 presenta lo stato dell'arte necessario a inquadrare il problema: i fondamenti dell'audio 3D e delle HRTF, la modellazione dell'ambiente acustico, e una rassegna della letteratura relativa all'impiego di avvisi acustici direzionali in ambito automotive.

Il Capitolo 3 descrive, dal punto di vista architetturale e teorico, il sistema esteso oggetto della tesi: la panoramica dell'architettura di spazializzazione, il principio matematico del filtraggio binaurale FIR, e la strategia di mappatura delle direzioni continue su un insieme discreto di direzioni di riferimento.

Il Capitolo 4 entra nel dettaglio implementativo del sistema di comunicazione di base: gli obiettivi, l'architettura, il protocollo MQTT impiegato, le scelte progettuali e il meccanismo di callback che collega l'evento di rete all'elaborazione audio.

Il Capitolo 5 descrive l'implementazione dell'estensione verso la spazializzazione HRTF: i requisiti di progettazione, la struttura del plugin NuTech, la libreria impiegata per la spazializzazione (Intel IPP) e le porzioni di codice più rilevanti per il precaricamento delle HRTF e la loro selezione in tempo reale.

Il Capitolo 6 presenta i test condotti per verificare il corretto funzionamento del sistema, distinguendo tra la configurazione dell'ambiente di test e la verifica del sistema di comunicazione di base, la validazione delle singole componenti e una fase di test sperimentali sul sistema completo, inclusa una validazione percettiva condotta con il simulatore di guida del laboratorio.

Il Capitolo 7, infine, trae le conclusioni del lavoro e ne discute i limiti e i possibili sviluppi futuri.

Il codice sorgente completo del plugin è riportato in Appendice A.

Capitolo 2.

Stato dell'arte: audio immersivo per la sicurezza alla guida

Introduzione

Questo capitolo presenta lo stato dell'arte necessario a inquadrare l'estensione, descritta nei capitoli successivi, del sistema di comunicazione verso una riproduzione audio spazializzata. Le prime due sezioni richiamano i fondamenti teorici dell'audio 3D e della modellazione dell'ambiente acustico tramite **HRTF**. Le sezioni seguenti presentano invece una rassegna della letteratura relativa all'impiego di segnali acustici direzionali e spazializzati come strumento di allerta per la sicurezza alla guida, con l'obiettivo di posizionare il lavoro di tesi rispetto a quanto già indagato in ambito automotive.

2.1. Fondamenti dell'audio 3D e delle HRTF

2.1.1. Dal semplice stereo alla spazializzazione tridimensionale

Un vero sistema audio 3D ha la capacità di posizionare un suono in punti arbitrari tutto attorno all'ascoltatore, inclusi i lati, il retro e le posizioni sopra e sotto la testa. Questo lo distingue nettamente dai tradizionali sistemi stereo o dai circuiti di *stereo enhancement*, i quali riescono unicamente ad ampliare il campo sonoro percepito di fronte all'utente [1]. Questa distinzione è particolarmente rilevante per un sistema di allerta pensato per la sicurezza alla guida: un avviso capace di indicare direzionalmente la provenienza di un pericolo o di un guasto fornisce al guidatore un'informazione che un segnale stereo tradizionale non può veicolare in modo altrettanto immediato.

2.1.2. Localizzazione uditiva

Il sistema uditivo umano localizza la provenienza dei suoni sfruttando tre indicatori (*cues*) principali che dipendono dalla conformazione di testa e orecchie [5], [6]:

- la **differenza interaurale di tempo** (*time delay*), ovvero il ritardo con cui l'onda sonora raggiunge un orecchio rispetto all'altro;

- la **differenza in ampiezza** (*amplitude difference*), dovuta all' "ombra acustica" generata dalla testa, che attenua il suono che deve aggirarla per raggiungere l'orecchio opposto;

- la **informazione spettrale** (*tonal information*), variazioni dello spettro del suono percepito dalle orecchie, determinate dalla direzione di provenienza della sorgente sonora e dall'interazione dell'onda acustica con il busto, la testa e, in particolare, con le pieghe del padiglione auricolare.

Queste trasformazioni, nel loro insieme, vengono misurate sperimentalmente tramite microfoni inseriti nei canali uditivi di un soggetto umano o di un manichino antropomorfo, e prendono il nome di **HRTF**. Come schematizzato in Figura 2.1, il segnale di misurazione viene riprodotto da un altoparlante e registrato dai microfoni. Un computer elabora i segnali registrati, per ricavare una coppia di HRTF (per l'orecchio sinistro e destro) corrispondenti alla posizione della sorgente sonora.

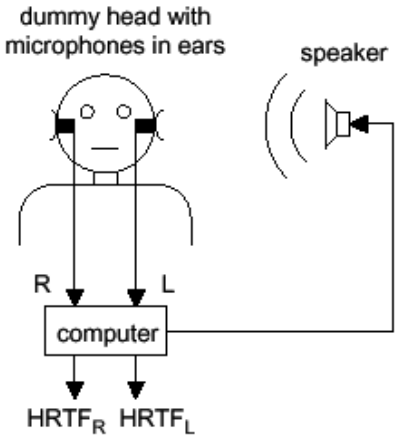


Figura 2.1.: Schema di misurazione delle HRTF (adattato da [1]).

Quindi ciascuna HRTF, tipicamente composta da diverse centinaia di numeri, descrive gli indicatori per una specifica posizione della sorgente sonora rispetto all'orecchio sinistro o destro del soggetto. La procedura di misurazione viene ripetuta per molte posizioni della sorgente sonora rispetto alla testa, ottenendo così un database di centinaia di HRTF che descrivono le caratteristiche di trasformazione del suono di una particolare testa [1].

2.1.3. Sintesi binaurale e limiti delle HRTF generiche

Un sistema di audio 3D crea la **sintesi binaurale** utilizzando una coppia di HRTF come filtri digitali per simulare il processo naturale di udito e far percepire al cervello il suono nel punto desiderato,[7], [8] come mostrato in Figura 2.2.

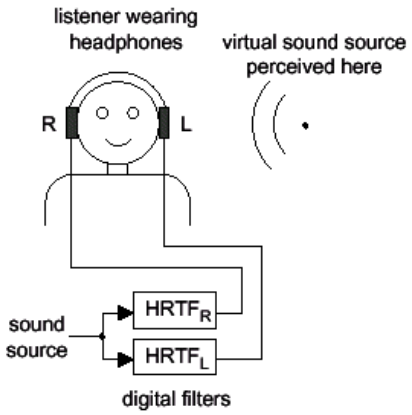


Figura 2.2.: Sintesi binaurale (adattato da [1]).

Poiché testa e orecchie variano da persona a persona, l'uso di HRTF non individualizzate riduce tuttavia le prestazioni di localizzazione.

I due errori più comuni sono :

1. la **confusione fronte/retro**: un suono generato davanti viene percepito come proveniente da dietro (o viceversa)[9];

2. gli **errori di elevazione**: la posizione verticale percepita non corrisponde a quella reale.

Questi limiti, ben documentati in letteratura e confermati anche in un contesto sperimentale più vicino a quello automotive (Sezione 2.4), sono un vincolo di cui è necessario tenere conto nella progettazione di un sistema di allerta acustico per la guida: un errore di localizzazione fronte/retro potrebbe indurre il guidatore a reagire nella direzione sbagliata rispetto a un pericolo reale. Per limitare tali problemi, Wave Arts utilizza HRTF ricavate da **710 misurazioni** effettuate su un manichino antropomorfo denominato KEMAR[10], all'interno della camera anecoica del MIT.

2.1.4. Audio 3D tramite altoparlanti: il problema del crosstalk

L'uso delle cuffie garantisce che il segnale destro arrivi esclusivamente all'orecchio destro e viceversa: è la condizione ideale per la sintesi binaurale.

Con gli altoparlanti, invece, si genera il fenomeno del **crosstalk**. Come visibile in Figura 2.3, gran parte del suono emesso dallo speaker sinistro raggiunge anche l'orecchio destro, compromettendo l'effetto 3D [1].

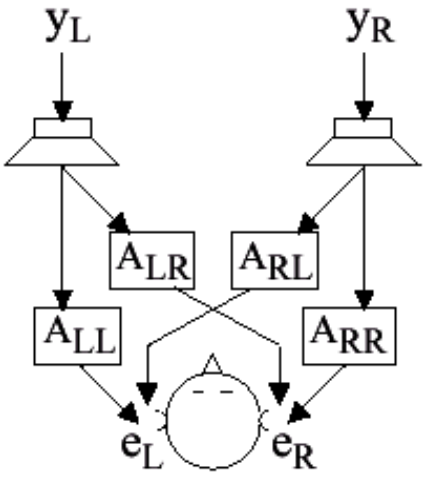


Figura 2.3.: Fenomeno del crosstalk (adattato da [1]).

Per risolvere questo problema si utilizza un elaborato filtro digitale detto **crosstalk canceller**. Tale filtro funziona perfettamente solo se l'utente si trova in una posizione centrale esatta chiamata *sweet spot*, rendendolo particolarmente indicato per il desktop computing; il sistema Wave Arts migliora la separazione dei canali di circa 20 dB nell'intervallo 100 Hz–6 kHz.

Questo aspetto ha una rilevanza diretta per un'applicazione automotive: l'abitacolo di un veicolo non è uno spazio di ascolto simmetrico, e la posizione del guidatore non coincide in generale con lo sweet spot ideale rispetto agli altoparlanti installati in fabbrica; un tema ripreso anche dalla letteratura specificamente automotive discussa nelle Sezioni 2.4 e 2.5.

2.2. Modellazione dell'ambiente acustico

La sola spazializzazione tridimensionale del suono non è sufficiente a ricreare una scena sonora percepita come naturale: è necessario combinarla con la simulazione dei comportamenti naturali del suono, in modo da ricreare scene acustiche realistiche [1].

2.2.1. Riverbero

Il riverbero è formato dall'energia sonora riflessa. Assumendo che esista un percorso diretto tra la sorgente e l'ascoltatore, quest'ultimo percepisce in sequenza:

- il **suono diretto** ;
- le **prime riflessioni** (*early reflections*), dovute alle pareti più vicine;
- il **riverbero diffuso** (*late reverberation*).

Il riverbero fornisce all'ascoltatore informazioni spaziali sulle dimensioni dell'ambiente e sui materiali che lo compongono: un decadimento con scarse componenti in alta frequenza indica la presenza di pareti morbide e assorbenti.

Le prime riflessioni possono essere calcolate tramite *ray-tracing*, mentre per il riverbero diffuso si utilizzano filtri a pettine o *all-pass* (filtri ricorsivi).

Wave Arts utilizza un algoritmo basato su *feedback all-pass* progettato per segnali binaurali, in cui le prime riflessioni si spostano insieme alla sorgente mentre il riverbero tardivo rimane diffuso in tutto lo spazio.

2.2.2. Effetto della distanza

La mente percepisce la lontananza valutando il rapporto tra il suono diretto e il riverbero.

Il suono diretto diminuisce di ampiezza rapidamente all'aumentare della distanza (6 dB per raddoppio della distanza), mentre il riverbero decresce molto più lentamente (3 dB per raddoppio della distanza); di conseguenza, suoni con elevata componente riverberata appaiono più lontani, mentre il solo volume non fornisce indicazioni affidabili sulla distanza.

Nel modello Wave Arts, a brevi distanze il riverbero è di 20 dB inferiore al suono diretto; aumentando la distanza il rapporto diminuisce, fino a circa 30 m, dove il riverbero diventa più forte del suono diretto [1].

2.2.3. Effetto Doppler

L'effetto Doppler è il cambiamento di intonazione avvertito quando una sorgente sonora si avvicina (frequenze percepite più acute) o si allontana (frequenze percepite più gravi) rapidamente, ed è proporzionale alla velocità relativa fra ascoltatore e sorgente. Nei sistemi software viene simulato tramite una linea di ritardo variabile [1].

2.2.4. Assorbimento dell'aria e occlusione degli oggetti

Su grandi distanze, l'aria assorbe preferenzialmente le alte frequenze, che vengono dissipate più facilmente; per questo motivo, a grandi distanze si percepiscono prevalentemente le basse frequenze.

Se un ostacolo si interpone tra sorgente e ascoltatore, le frequenze acute vengono bloccate mentre quelle gravi riescono ad aggirare l'oggetto per diffrazione.

Entrambi gli effetti vengono ricreati con filtri passa-basso, la cui frequenza di taglio è, rispettivamente, inversamente proporzionale alla distanza e dipendente dalle dimensioni dell'oggetto occludente [1].

2.2.5. Routing del segnale e virtual speaker

Il motore di elaborazione tratta i segnali in ingresso (monofonici) facendoli transitare per una catena di effetti

1. Effetto Doppler

2. Assorbimento dell'aria
3. Occlusione
4. Spazializzazione 3D (vengono applicate le HRTF per sdoppiare il segnale in stereo, 2 canali)

Infine l'uscita viene distribuita in due bus in base ai parametri di distanza impostati:

- **direct bus**: veicola il segnale diretto;
- **reverb bus**: veicola il segnale di riverbero, che attraversa il riverberatore prima di ricongiungersi e sommarsi all'uscita del direct bus.

Nel caso di riproduzione su altoparlanti, il segnale così ricombinato attraversa poi il crosstalk canceller, uno stadio di equalizzazione dedicato (crosstalk EQ) e i controlli di tono finali (POST EQ); nel caso di riproduzione in cuffia, non essendovi crosstalk da compensare, l'uscita del direct/reverb bus va direttamente al POST EQ.

Un concetto particolarmente utile per un'applicazione di allerta automotive è quello di **virtual speaker**: un oggetto sonoro stazionario fisso nello spazio, immune agli effetti ambientali (non soggetto a Doppler, assorbimento, riverbero o occlusione), utilizzato per riposizionare sorgenti audio statiche preesistenti o per convertire segnali stereo/surround in audio 3D immersivo.

Alcuni virtual speaker consentono inoltre l'assegnazione diretta del segnale ai canali fisici, applicando in tal caso una compensazione energetica pari a $G_C = \sqrt{2}/2 \approx 0{,}707$ (circa $-3\,\mathrm{dB}$) per mantenere costante la potenza percepita quando un segnale mono viene sommato su entrambi i canali [1].

2.3. Perché la direzione conta per la sicurezza: dagli avvisi semantici agli avvisi spazializzati

I fondamenti richiamati nelle sezioni precedenti descrivono *come* un suono possa essere reso spazialmente realistico. Resta però da chiarire *se*, e *con quale meccanismo*, l'informazione sulla direzione di un pericolo trasmessa per via acustica sia effettivamente utile al guidatore.

La letteratura in ambito automotive distingue, a questo proposito, tra due modi molto diversi di veicolare la direzione: tramite un contenuto *semantico* (una parola pronunciata, come "sinistra") oppure tramite la posizione *acustica reale* del suono nello spazio. Le due sottosezioni seguenti presentano un lavoro rappresentativo per ciascun approccio.

2.3.1. Avvisi direzionali basati sul contenuto semantico: il caso della guida automatizzata

Con l'aumentare del livello di automazione dei veicoli stradali, il controllo operativo è il primo elemento della guida a essere disimpegnato. Nell'automazione di Livello

2 SAE (L2) [1], sebbene il conducente sia escluso dal ciclo di controllo operativo, deve comunque monitorare attivamente l'ambiente stradale e intervenire in caso di necessità. Vi è tuttavia evidenza che i movimenti oculari del conducente risultino alterati durante la guida L2 a mani libere (*hands-off*), con una maggiore dispersione dello sguardo e una ridotta attenzione verso le aree critiche per la sicurezza; questa condizione, nota come estraniamento dal ciclo di controllo (*out-of-the-loop*, OOTL), è generalmente associata a una riduzione della consapevolezza situazionale (*situation awareness*) e può portare a prestazioni scadenti al momento di una richiesta di ripresa del controllo (*takeover*) da parte dell'automazione [11].

In questo contesto, gli avvisi acustici direzionali che indirizzano l'attenzione del guidatore verso una posizione specifica si dividono in letteratura in due categorie [11]:

- gli avvisi **verso la corsia libera** (*towards-free-lane*, o "ipsilaterali"), che dirigono l'attenzione verso la posizione in cui è sicuro sterzare per evitare il pericolo;
- gli avvisi **verso il pericolo** (*towards-hazard*, o "controlaterali"), che informano invece sulla posizione del pericolo stesso, lasciando al guidatore la decisione di sterzare nella direzione opposta.

Studi precedenti avevano confrontato prevalentemente gli avvisi "verso il pericolo" con avvisi non direzionali, senza un quadro conclusivo: uno studio basato su scenari video ha rilevato risposte più rapide agli avvisi "verso il pericolo" [12]. Un secondo studio, condotto in un simulatore di cambio corsia, ha evidenziato che gli avvisi direzionali portano a reazioni più rapide rispetto a quelli non direzionali. Tuttavia, non sono state riscontrate differenze significative tra le due strategie, osservando anzi che l'allocazione dell'attenzione visiva dei partecipanti era guidata soprattutto dal contesto di guida (ostacoli, specchietti retrovisori) piuttosto che dal tipo di avviso acustico [13].

Arabian, Goodridge e Merat in [11], per colmare questa lacuna, hanno condotto un esperimento su 48 partecipanti presso il simulatore di guida a base fissa dell'Università di Leeds, confrontando avvisi "verso la corsia libera", avvisi "verso il pericolo" e avvisi non direzionali durante transizioni di controllo da un veicolo automatizzato SAE L2 a mani sul volante [11]. Gli avvisi, in linea con le linee guida NHTSA (National Highway Traffic Safety Administration)[2] sulla sinteticità dei messaggi vocali di allerta, consistevano in singole parole pronunciate[3][11].

I risultati hanno mostrato che entrambi i tipi di avviso direzionale hanno ridotto i tempi della prima reazione di sterzata, migliorato l'accuratezza della prima risposta

[1] La SAE, Society of Automotive Engineers, definisce 6 livelli di guida autonoma. Il livello 2 è un sistema a guida autonoma parziale, con intervento in frenata e accelerazione in caso di pericolo, ad esempio la frenata automatica di emergenza.

[2] sono normative, raccomandazioni e protocolli di sicurezza stradale che hanno lo scopo di ridurre incidenti, feriti e decessi sulle strade

[3] "Left"/"Right" per gli avvisi direzionali, "Look" per quelli non direzionali

e ridotto l'accelerazione laterale massima rispetto agli avvisi non direzionali, con gli avvisi "verso la corsia libera" leggermente più efficaci in termini di rapidità di reazione e minore carico di lavoro soggettivo [11]. Lo studio ha inoltre indagato l'effetto dell'asincronia temporale tra la componente acustica e quella visiva dell'avviso, mostrando che intervalli SOA[4] più lunghi riducono la dominanza dell'informazione visiva sull'elaborazione dell'informazione acustica, con un effetto di attenuazione lineare per gli avvisi non direzionali e moltiplicativo per quelli direzionali [11].

Questo lavoro dimostra con chiarezza l'importanza di fornire al guidatore un'informazione sulla **direzione**[5] del pericolo, piuttosto che un semplice avviso generico. Va tuttavia osservato un limite intrinseco a questo tipo di segnalazione: trattandosi di avvisi basati su parole pronunciate, essi richiedono comunque una decodifica linguistica, per quanto minima, da parte del guidatore, un processo cognitivo che si aggiunge al carico visivo e motorio già impegnato dalla guida.

Pertanto un segnale acustico spazializzato, la cui direzione di provenienza è percepita direttamente e non deve essere interpretata semanticamente, potrebbe in linea di principio sfruttare un canale di elaborazione più diretto e meno oneroso dal punto di vista cognitivo. È proprio questa ipotesi a essere indagata dal lavoro presentato nella sezione seguente.

2.3.2. Avvisi acustici spazialmente predittivi e attenzione cross-modale

Un riferimento centrale per comprendere il meccanismo attraverso cui la posizione *acustica reale* di un suono, e non solo il suo contenuto semantico, influenza l'attenzione del guidatore è lo studio di Ho e Spence[6] [14]. Lo studio si fonda sull'evidenza dell'esistenza di legami cross-modali nell'attenzione spaziale tra modalità sensoriali diverse: presentare informazioni rilevanti a sensi differenti (ad esempio udito e vista) approssimativamente dalla stessa posizione spaziale può potenziare l'efficienza dell'elaborazione multisensoriale, poiché attenzione uditiva e visiva non sembrano costituire risorse cognitive indipendenti, ma risultano invece collegate.

Gli autori hanno condotto cinque esperimenti in un compito di guida simulata, valutando la capacità di segnali acustici di facilitare la risposta (frenata o accelerazione) a situazioni di emergenza [7]. Sono stati confrontati suoni di clacson non spaziali e non predittivi (neutri), suoni spazialmente non predittivi (validi nel 50% dei casi) e suoni spazialmente predittivi (validi nell'80% dei casi), oltre a segnali vocali simbolici (le parole "front"/"back") presentati sia in modo non spaziale sia in posizione spaziale coerente con il significato.

I risultati hanno mostrato tempi di risposta significativamente più rapidi quando i segnali acustici erano spazialmente predittivi, con un effetto ancora maggiore quando

[4] *Stimulus Onset Asynchrony*, asincronia temporale fra gli avvisi)
[5] sia visivi, sia vocali/semantici
[6] pubblicato sul *Journal of Experimental Psychology: Applied*
[7] il rapido avvicinamento di un'auto dal davanti o da dietro, osservato attraverso il parabrezza o lo specchietto retrovisore

la predittività spaziale si combinava con un contenuto semanticamente rilevante: i segnali di avviso acustici spazialmente predittivi e semanticamente significativi si sono rivelati il mezzo più efficace nel catturare e orientare l'attenzione del guidatore verso la direzione corretta.

Un ulteriore studio richiamato dagli stessi autori, condotto da Spence e Read con un compito di ripetizione vocale (*shadowing*) concorrente a un compito di guida su simulatore, ha inoltre mostrato che il vantaggio nel processare informazioni presentate frontalmente rispetto a quelle laterali si accentua proprio quando il carico percettivo del compito di guida è più elevato [14]: un'indicazione che, in condizioni di carico cognitivo realistiche come quelle della guida, la corrispondenza spaziale tra la posizione del suono e la posizione dell'evento da attendere diventa ancora più rilevante, e non meno.

Questi risultati forniscono un fondamento sperimentale diretto all'idea alla base della presente tesi: mentre un avviso vocale come quello discusso nella sezione precedente richiede l'elaborazione cognitiva di un contenuto linguistico, un segnale acustico la cui posizione nello spazio uditivo, realizzata tramite HRTF[8], coincide con la posizione reale del pericolo può sfruttare il meccanismo di orientamento attentivo cross-modale riflesso descritto in questo studio, offrendo potenzialmente un canale di allerta più immediato e meno dipendente dalla decodifica linguistica.

2.4. Audio immersivo object-based per l'HMI automotive

Se gli studi presentati nella sezione precedente dimostrano, in condizioni sperimentali controllate, il valore della direzionalità (semantica prima, acustica poi) per la sicurezza, il lavoro di van Laack e Torschmied (Visteon) [2], affronta il problema da un punto di vista più applicativo: come rendere disponibile, in un veicolo reale, un audio realmente spazializzato in tutte le direzioni, superando i limiti dei formati audio tradizionali.

Gli autori osservano che i sistemi audio stereo e surround più comuni, come mostra la Figura 2.4, generano immagini sonore tramite la sovrapposizione di canali discreti, la cui corretta resa spaziale dipende dal fatto che la configurazione degli altoparlanti in riproduzione corrisponda al formato usato in produzione, e risulta piena solo in un punto di ascolto preciso, lo *sweet spot* [15].

[8] non più tramite un semplice panning stereo a due posizioni come nel caso del clacson sinistro/destro impiegato da Ho e Spence

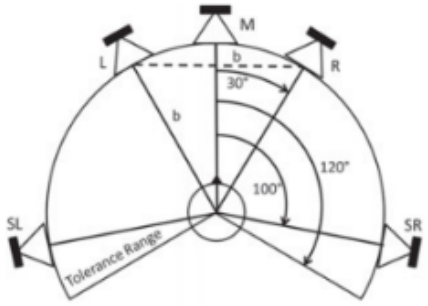


Figura 2.4.: Configurazione surround a 5 canali (adattato da [2]).

Inoltre, i formati surround derivano tipicamente da applicazioni home cinema, che presuppongono che la maggior parte dell'informazione acustica provenga dal fronte: una limitazione poco compatibile con un'HMI (Human-Machine Interface) veicolare che debba segnalare eventi provenienti anche dai lati o da dietro il veicolo.

Come alternativa, gli autori propongono l'adozione di tecnologia audio *object-based*, in cui i suoni sono trasmessi come canali mono indipendenti accompagnati da metadati che ne descrivono le proprietà fisiche e acustiche (posizione, ecc.); un algoritmo di rendering spaziale calcola quindi in tempo reale l'output per ciascun canale fisico, in modo sostanzialmente indipendente dalla configurazione fisica degli altoparlanti disponibili [15].

Per validare questo approccio dal punto di vista percettivo, gli autori hanno condotto una *customer clinic*[9] con 33 partecipanti, bendati per eliminare l'influenza del canale visivo, utilizzando una configurazione a 10.1 altoparlanti con un renderer audio object-based per creare suoni virtuali.

Lo studio ha indagato tre aspetti:

- la capacità dei partecipanti di determinare la posizione (direzione e distanza) di un suono statico;
- la capacità di determinare il movimento di un suono attraverso quattro settori attorno all'ascoltatore;
- la percezione soggettiva di diversi suoni, classificati come notifica informativa (*notification*), avvertimento (*warning*) o allarme (*alert*).

I risultati indicano che la distanza ha un'influenza quasi irrilevante sulla corretta localizzazione di un suono statico, mentre emerge una netta difficoltà nel localizzare suoni statici posizionati dietro la testa dell'ascoltatore, con meno del 50% dei partecipanti in grado di determinarne correttamente la posizione, una conferma sperimentale, in un contesto vicino a quello automotive, del limite di confusione fronte/retro già richiamato nella Sezione 2.1.3 a proposito delle HRTF generiche. I suoni dinamici, al

[9]Metodo di ricerca di mercato qualitativo in cui un gruppo selezionato di consumatori viene invitato a valutare in anteprima e a porte chiuse prodotti, concept o prototipi non ancora sul mercato. In questo caso è una sessione di test in cui un gruppo di persone prova direttamente il sistema audio

contrario, sono risultati identificabili con maggiore facilità rispetto a quelli statici, in particolare quando il movimento avveniva lateralmente rispetto all'ascoltatore. Sul piano percettivo, i suoni classificati come avvertimenti e allarmi sono stati valutati come più critici e meno amichevoli rispetto alle notifiche informative, indicando che le caratteristiche timbriche di un suono comunicano al guidatore anche il livello di urgenza della situazione, indipendentemente dalla sua posizione.

Questi risultati sono direttamente rilevanti per la progettazione di un sistema di allerta immersivo: da un lato confermano che l'audio object-based consente di posizionare oggetti sonori percepiti come tali dall'ascoltatore, in particolare quando in movimento e ai lati del guidatore[10]; dall'altro, evidenziano che i suoni statici posizionati dietro la testa restano problematici, suggerendo che un sistema di allerta per la sicurezza debba prevedere, per le direzioni posteriori, l'uso di suoni dinamici (in movimento) piuttosto che statici, oppure strategie di compensazione ulteriori[2].

2.5. Fattibilità implementativa: rendering HRTF/BRIR in abitacolo reale

Le sezioni precedenti hanno mostrato perché un audio spazializzato sia utile per la sicurezza del guidatore. Resta da chiarire se, e con quale hardware, un rendering HRTF realistico sia oggi effettivamente realizzabile all'interno di un abitacolo automobilistico. A questo proposito è utile il lavoro di Wolters, Giralt, Mundt e Biswas (Dolby Germany), presentato alla AES International Conference on Automotive Audio. È opportuno premettere che questo studio è motivato principalmente da obiettivi di intrattenimento (qualità di ascolto musicale) e non da obiettivi di sicurezza; viene qui richiamato esclusivamente come riferimento sulla fattibilità tecnica del rendering binaurale in tempo reale in un veicolo reale, non come ulteriore evidenza sperimentale sull'efficacia degli avvisi di sicurezza.

Gli autori osservano che l'abitacolo di un'automobile è un ambiente acusticamente difficile per la riproduzione 3D, a causa delle forti riflessioni delle superfici vetrate e dell'asimmetria delle posizioni di ascolto rispetto agli altoparlanti di serie. Una soluzione proposta per superare questi limiti è l'impiego di altoparlanti integrati nei poggiatesta (*headrest-integrated loudspeakers*): una tecnologia nota fin dagli anni '70, storicamente utilizzata per la sua vicinanza alle orecchie dell'ascoltatore in ambienti rumorosi, e oggi rivalutata per la riproduzione di audio immersivo grazie alla possibilità di creare zone sonore personali (*personal sound zones*). La prossimità dei diffusori alle orecchie dell'ascoltatore consente sia di creare zone acustiche personalizzate (personal sound zones), riducendo le distrazioni per gli

[10] Una condizione, questa, particolarmente utile per segnalare ad esempio un veicolo o una bicicletta in rapido avvicinamento laterale

altri occupanti, sia di applicare un'elaborazione binaurale basata su filtri HRTF e BRIR[11].[3]

Lo studio, condotto su una Volvo XC60 strumentata con un impianto audio premium a 15 driver più diffusori a soffitto e poggiatesta stereo indipendenti come mostrato in Figura 2.5, ha confrontato tre configurazioni di rendering[12]:

- un sistema discreto 7.1.4 con l'insieme degli altoparlanti dell'abitacolo, usato come riferimento[13];
- lo stesso sistema integrato con i diffusori del poggiatesta, verso cui vengono instradati i contenuti laterali e posteriori dopo un'elaborazione di **binauralizzazione basata su HRTF** che compensa il posizionamento fisico dei driver rispetto alle orecchie dell'ascoltatore;
- una configurazione ridotta con i soli altoparlanti anteriori più il poggiatesta, che si affida interamente ai driver del poggiatesta per i contenuti spaziali posteriori e in altezza[14].

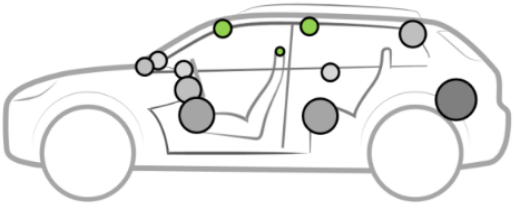


Figura 2.5.: Disposizione degli altoparlanti del veicolo. I cerchi indicano le dimensioni e la posizione approssimative degli altoparlanti , quelli evidenziate in verde sono aggiuntivi rispetto al sistema originale (adattato da [3]).

In entrambe le configurazioni con poggiatesta, oltre al filtraggio HRTF dipendente dalla direzione, viene applicata una seconda fase di elaborazione basata su misurazioni della risposta all'impulso binaurale dell'ambiente (*Binaural Room Impulse Response*, BRIR), che introduce le caratteristiche di riflessioni primarie e riverberazione dell'abitacolo (un parallelo diretto, in un contesto reale, con la modellazione del riverbero descritta nella Sezione 2.2.1 a proposito degli algoritmi Wave Arts) [3].

Attraverso test di ascolto a confronto di coppie, analizzati con il modello probabilistico Bradley-Terry-Luce, gli autori hanno riscontrato una netta preferenza degli

[11] *Binaural Room Impulse Response*, registra come un suono si propaga da una sorgente all'interno di una specifica stanza fino alle orecchie di un ascoltatore e filtra unendo le riflessioni e il riverbero dell'ambiente con forma della testa, del busto e dei padiglioni auricolari. Viene usato principalmente nell'audio spaziale per applicare filtri alle cuffie e ricreare un ascolto tridimensionale realistico in un ambiente virtuale

[12] progettate per rappresentare una gamma di scenari applicativi pratici in ambito automotive

[13] comprendente le posizioni anteriori, surround e a soffitto

[14] tale configurazione valuta in che misura gli altoparlanti nel poggiatesta possano sostituire un layout surround completo mantenendo al contempo una qualità spaziale accettabile, e quantifica eventuali compromessi percettivi rispetto al sistema discreto completo

ascoltatori per la configurazione con abitacolo completo più poggiatesta, e hanno osservato che la configurazione ridotta (soli anteriori più poggiatesta) raggiunge una qualità percepita paragonabile a quella del sistema discreto completo, suggerendo che i diffusori nel poggiatesta possano parzialmente sostituire un layout di altoparlanti distribuiti nell'abitacolo pur mantenendo una resa spaziale efficace .

Ai fini di questa tesi, il risultato più rilevante non riguarda tanto la preferenza d'ascolto in sé, quanto la dimostrazione che una pipeline di rendering basata su HRTF e BRIR, applicata in tempo reale in un veicolo reale e non solo in laboratorio, è oggi una tecnologia matura e oggetto di ricerca industriale attiva[3].

2.6. Sintesi e posizionamento della tesi

I contributi richiamati in questo capitolo, pur eterogenei per metodologia e contesto applicativo, permettono di tracciare un percorso logico che dai fondamenti dell'audio 3D conduce fino alle scelte progettuali della tesi:

- la letteratura tecnica sulle HRTF e sulla sintesi binaurale [1] fornisce gli strumenti matematici e concettuali per posizionare un suono di allerta in una direzione specifica dello spazio attorno al guidatore, ma evidenzia anche limiti concreti (HRTF non individualizzate, confusione fronte/retro, dipendenza dallo sweet spot nella riproduzione su altoparlanti) di cui è necessario tenere conto in fase di progettazione;

- lo studio sugli avvisi direzionali in guida automatizzata [11] dimostra che fornire al guidatore l'informazione sulla direzione di un pericolo, anche tramite un semplice contenuto vocale, migliora in modo significativo la qualità e la rapidità della risposta rispetto ad avvisi generici, ma tale approccio richiede comunque una decodifica linguistica del messaggio;

- lo studio di Ho e Spence [14] mostra che un segnale la cui posizione *acustica reale* coincide con la direzione dell'evento da segnalare sfrutta un meccanismo di orientamento attentivo cross-modale più diretto e riflesso, potenzialmente meno oneroso dal punto di vista cognitivo rispetto alla sola informazione semantica, un'evidenza che costituisce il fondamento sperimentale più diretto a sostegno dell'uso di audio spazializzato per la sicurezza;

- il lavoro Visteon sull'audio object-based [2] dimostra la fattibilità pratica, in un contesto sperimentale vicino a quello automotive, dell'uso della spazializzazione sonora per veicolare informazione direzionale, confermando al tempo stesso, su suoni statici posteriori, gli stessi limiti di localizzazione già discussi a livello teorico;

- infine, il lavoro di Wolters et al. [3], per quanto motivato da obiettivi di intrattenimento e non di sicurezza, dimostra che l'infrastruttura hardware e

> algoritmica necessaria a un rendering HRTF/BRIR in tempo reale all'interno di un abitacolo reale è oggi disponibile e attivamente sviluppata dall'industria.

Nessuno dei lavori qui presentati applica però una spazializzazione HRTF vera e propria come quella progettata in questa tesi.

È in questo spazio, all'incrocio tra l'evidenza sperimentale sul valore degli avvisi spazialmente predittivi e la maturità tecnologica del rendering binaurale in abitacolo, che si colloca il presente lavoro: i capitoli successivi descrivono l'estensione del sistema di comunicazione con un motore di spazializzazione audio basato su HRTF, progettato specificamente per la generazione di segnali di allerta immersivi in ambito automotive.

Capitolo 3.

Sistema implementato

3.1. Panoramica architetturale del sistema esteso

Il sistema base si limitava a decidere *quale* traccia audio riprodurre. L'obiettivo è estendere tale pipeline con uno stadio aggiuntivo di **spazializzazione binaurale**, inserito tra la selezione della traccia e la scrittura dell'uscita audio.

Concettualmente, la catena di elaborazione si articola ora in quattro fasi:

1. **ricezione del comando**: il plugin riceve, tramite la callback MQTT, descritta nel Capitolo 4, un messaggio JSON che specifica non più solo la traccia da riprodurre, ma anche la direzione da cui il suono deve provenire;

2. **selezione della coppia di filtri HRTF**: in base alla direzione richiesta, il plugin seleziona la coppia di risposte impulsive (orecchio sinistro/destro) più adatta tra quelle disponibili;

3. **filtraggio binaurale**: il segnale mono viene convoluto in tempo reale con le due risposte impulsive, generando due segnali distinti;

4. **uscita stereo spazializzata**: i due segnali filtrati vengono scritti sui due canali di uscita del plugin, ora resi disponibili come due pin distinti anziché uno solo.

La Figura 3.1 illustra questa estensione, ponendo in evidenza il nuovo stadio di spazializzazione si inserisca esattamente nel punto in cui, nel sistema base, il segnale veniva copiato direttamente nel buffer di uscita.

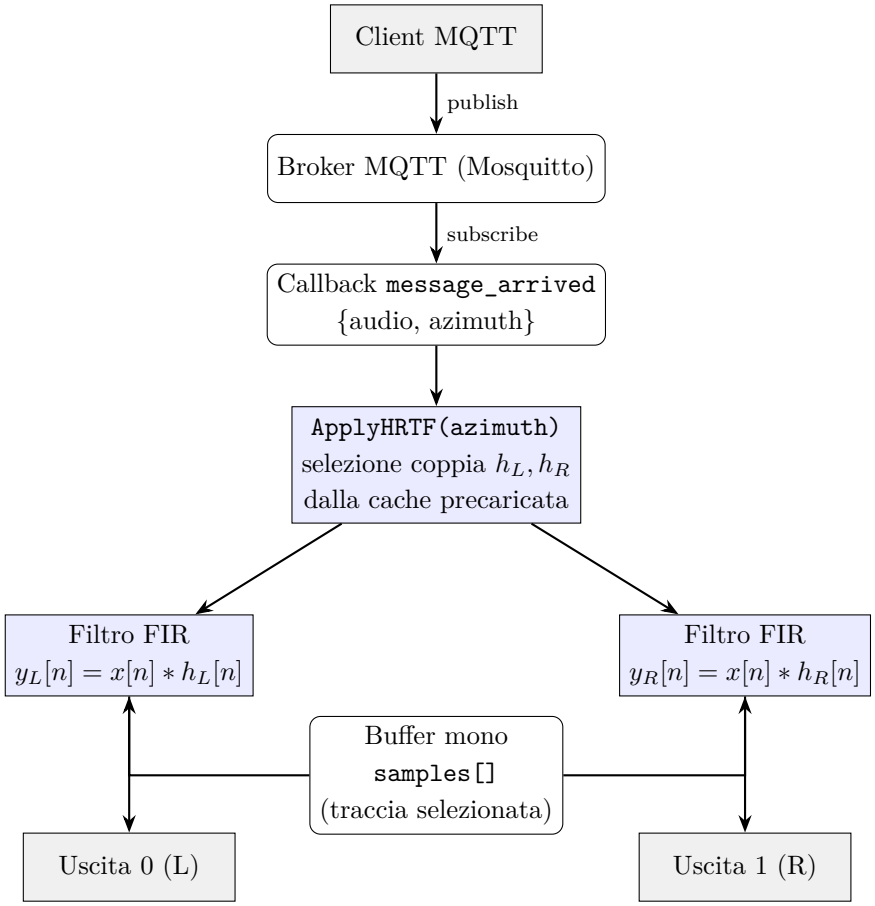


Figura 3.1.: Struttura estesa del plugin: dal comando MQTT alla spazializzazione binaurale su due uscite indipendenti.

Due elementi architetturali cambiano in modo sostanziale: il numero di uscite del plugin, che passa da una a due (`LESetNumOutput(2)`), e la natura dell'elaborazione applicata al buffer campioni prima della scrittura in uscita, che non è più una semplice copia con guadagno ma una convoluzione FIR indipendente per ciascun canale.

Per veicolare anche l'informazione di direzione, il formato del messaggio JSON è stato esteso con un secondo campo, `azimuth`, espresso in gradi:

```
{"audio": 1, "azimuth": 90}
```

Questa scelta non altera la struttura concettuale: l'unica differenza è che la callback `message_arrived`, oltre a leggere il campo `audio`, legge ora anche il campo `azimuth` tramite lo stesso meccanismo di parsing JSON, con un valore di default (90°) qualora il campo non fosse presente, in linea con l'approccio "robusto ai messaggi malformati".

3.2. Principio di spazializzazione: filtraggio FIR binaurale

Come richiamato nel Capitolo 2, le HRTF sono, nel dominio digitale, filtri a risposta impulsiva finita (FIR): i coefficienti del filtro coincidono con i campioni

della risposta impulsiva misurata per una determinata direzione. Applicare una spazializzazione HRTF a un segnale mono $x[n]$ significa quindi calcolare due segnali distinti, uno per ciascun orecchio, tramite convoluzione discreta con le rispettive risposte impulsive $h_L[n]$ e $h_R[n]$:

$$y_L[n] = x[n]*h_L[n] = \sum_{k=0}^{M-1} h_L[k]\, x[n-k], \qquad y_R[n] = x[n]*h_R[n] = \sum_{k=0}^{M-1} h_R[k]\, x[n-k] \tag{3.1}$$

dove M è il numero di coefficienti (*tap*) del filtro. Nel sistema qui sviluppato, i coefficienti h_L e h_R sono letti da un database di misurazioni reali: il database KEMAR del MIT Media Lab già richiamato nel Capitolo 2, costituito da 710 coppie di risposte impulsive misurate a 44.1 kHz mediante sequenze pseudo-casuali a lunghezza massima, per altrettante posizioni di un altoparlante attorno a un manichino antropomorfo [1].

Il sistema richiede quindi due filtraggi FIR indipendenti e paralleli, ciascuno con il proprio stato interno (la propria *delay line*), applicati allo stesso segnale mono di ingresso. Questo raddoppio è la diretta conseguenza del principio di sintesi binaurale descritto nel Capitolo 2 [1]: è la differenza tra i due segnali filtrati a veicolare l'informazione di direzione al sistema uditivo dell'ascoltatore.

3.3. Mappatura delle direzioni: dalle misurazioni continue a un insieme discreto

Il database KEMAR contiene misurazioni per un numero elevato ma discreto di direzioni (710 posizioni, come visto nel Capitolo 2), mentre il campo `azimuth` del messaggio MQTT può assumere qualsiasi valore continuo tra 0° e 360°. Si è reso quindi necessario definire una strategia per mappare l'angolo richiesto sulle coppie di filtri effettivamente disponibili.

L'obiettivo del sistema è indicare al guidatore la provenienza approssimativa di un pericolo o di un guasto[1] non è pertanto richiesta una risoluzione angolare pari a quella del database originale. Si è scelto di limitare il sistema a un set ridotto di **direzioni di riferimento**. Come descritto successivamente, queste direzioni sono state selezionate tramite test di ascolto in laboratorio per massimizzarne la distinguibilità.

L'insieme finale comprende sei direzioni: 0° (anteriore centrale), 70° (anteriore destra), 160° (posteriore destra), 180° (posteriore centrale), 200° (posteriore sinistra) e 290° (anteriore sinistra).

Quando il plugin riceve un angolo θ qualsiasi, calcola la distanza angolare circolare $d(\theta, \theta_i)$ rispetto a ciascuna direzione di riferimento θ_i:

$$d(\theta, \theta_i) = \min\left(|\theta - \theta_i|,\, 360^\circ - |\theta - \theta_i|\right) \tag{3.2}$$

[1] ad esempio: davanti, dietro o ai lati

dove il minimo garantisce di considerare il percorso più breve lungo la circonferenza.

Successivamente, viene selezionata la direzione θ_{scelto} che minimizza tale distanza:

$$\theta_{\text{scelto}} = \arg \min_{\theta_i \in \{0°,70°,160°,180°,200°,290°\}} d(\theta, \theta_i) \tag{3.3}$$

Questa scelta ha una duplice motivazione. Da un lato, riduce la complessità del sistema: solo sei coppie di filtri devono essere gestite, invece di un numero arbitrario, semplificando sia la gestione della memoria sia i test di validazione (Capitolo 6). Dall'altro, è coerente con i limiti di localizzazione uditiva richiamati nel Capitolo 2: dato che l'orecchio umano commette comunque errori sistematici di localizzazione, anche con HRTF misurate con precisione, una risoluzione maggiore comporterebbe un notevole costo computazionale senza offrire reali benefici percettivi.

Capitolo 4.

Implementazione della comunicazione MQTT

Introduzione

Questo capitolo descrive, dal punto di vista implementativo, il sistema di comunicazione di base su cui si innesta l'estensione presentata nel Capitolo 3. Vengono illustrati l'architettura complessiva della comunicazione tra client MQTT, broker e plugin NuTech, i richiami essenziali al protocollo MQTT impiegato, e le scelte progettuali adottate per integrare le librerie. Una parte centrale del capitolo è dedicata al meccanismo di comunicazione asincrona tramite callback, che collega l'evento di rete all'elaborazione audio in tempo reale. La configurazione dell'ambiente di sviluppo e i test di verifica della comunicazione sono riportati nel Capitolo 6.

4.1. Obiettivi del sistema di comunicazione

Il punto di partenza del lavoro è un sistema in grado di far comunicare NuTech[16], ambiente di elaborazione audio in tempo reale basato su plugin nativi Windows, con un'interfaccia di comando remoto. Poiché NuTech non dispone nativamente di alcun meccanismo di comunicazione di rete, è stato necessario progettare e implementare un plugin dedicato che, sfruttando il protocollo MQTT [4], permettesse di comandare da remoto la riproduzione audio.

L'obiettivo di questa prima fase implementativa era dunque realizzare un sistema di comunicazione MQTT capace di:

- ricevere, tramite un client MQTT, l'indice della traccia audio da riprodurre, pubblicato come messaggio JSON su un topic concordato;
- instradare tale comando, tramite il plugin NuTech, verso l'avvio della riproduzione della traccia corrispondente.

Questo obiettivo ha richiesto la risoluzione di una serie di problemi tecnici non banali: l'integrazione di una libreria di rete moderna (Paho MQTT C++)[17] in un ambiente a 32 bit, la gestione di un thread di rete e un thread audio real-time. Questo

sistema di comunicazione di base costituisce il punto di partenza su cui, nelle sezioni successive, viene costruita l'estensione verso la riproduzione audio spazializzata.

4.2. Panoramica architetturale del sistema

Il sistema si articola in una catena di componenti che collega l'azione dell'utente alla riproduzione audio effettiva sul lato NuTech. La Figura 4.1 di seguito riassume graficamente questa catena.

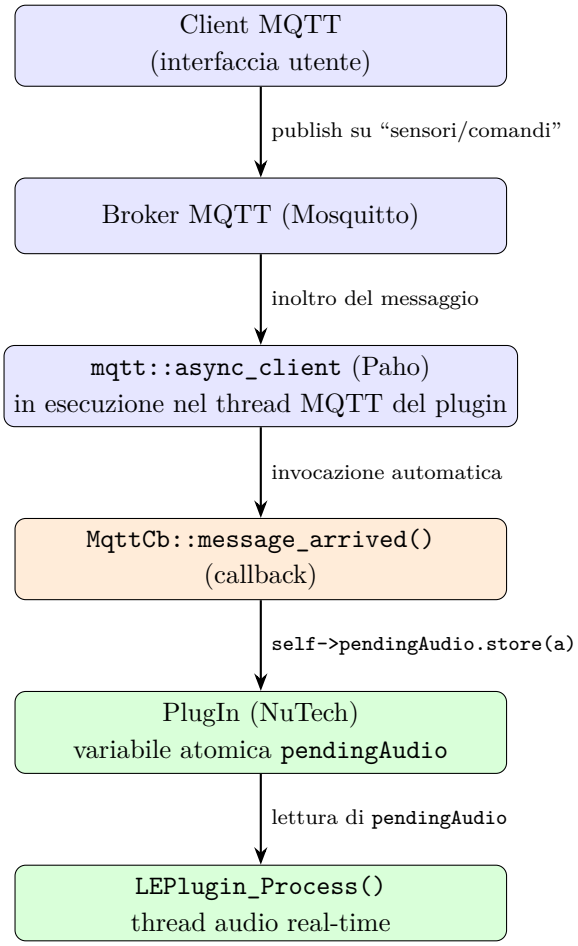


Figura 4.1.: Architettura del sistema di comunicazione

Un aspetto concettuale centrale è il seguente: nessuno dei due sistemi conosce direttamente l'altro. Il client pubblica un messaggio sapendo solo che esiste un broker MQTT in ascolto su un determinato indirizzo; il broker inoltra il messaggio a qualunque client sottoscritto al topic, senza sapere nulla del plugin NuTech; il plugin NuTech, a sua volta, non sa nè del client da cui è stato inviato il messaggio, né conosce i dettagli del broker oltre all'indirizzo di connessione. L'unico punto in cui i due mondi si incontrano è la callback MQTT, descritta nel Paragrafo 4.5.2, che

funge da collante fra il mondo asincrono della rete e il mondo sincrono e real-time dell'elaborazione audio.

4.3. Il protocollo MQTT: richiami essenziali

MQTT (Message Queuing Telemetry Transport) è un protocollo di messaggistica publish/subscribe, molto diffuso in ambito domotico. Il progetto ha permesso di applicarne concretamente i concetti fondamentali, richiamati di seguito.

4.3.1. Broker, client, topic e payload

L'architettura MQTT prevede un nodo centrale, il broker [1] verso cui si connettono più client. Ogni client può pubblicare messaggi su un topic [2], una stringa gerarchica che identifica un canale logico, e/o sottoscriversi a uno o più topic per ricevere i messaggi pubblicati su di essi. Il broker non fa altro che ricevere i messaggi pubblicati e inoltrarli a tutti i client attualmente sottoscritti al topic corrispondente. Il contenuto del messaggio, detto Payload, non ha un formato imposto dal protocollo: nel progetto si è scelto di codificarlo come JSON[3], per la sua semplicità di generazione (lato client) e di ricezione (lato plugin, tramite la libreria nlohmann/json[18]).

4.3.2. Livelli di QoS

MQTT definisce tre livelli di Quality of Service (QoS), che regolano il compromesso fra affidabilità della consegna e overhead di rete:

- **QoS 0, "at most once"**: il messaggio viene inviato senza richiedere conferma; è il più leggero, ma un messaggio può andare perso senza che il publisher se ne accorga. Adatto a telemetria ad alta frequenza dove la perdita occasionale di un dato è irrilevante.
- **QoS 1, "at least once"**: il broker conferma la ricezione; se la conferma non arriva, il publisher ritrasmette. Ciò garantisce che il messaggio arrivi, al prezzo di possibili duplicati lato subscriber.

  È il livello scelto per questo progetto, in quanto un comando di riproduzione audio deve arrivare a destinazione, mentre un'eventuale duplicazione del comando non comprometterebbe il funzionamento.
- **QoS 2, "exactly once"**: il livello più robusto, basato su un handshake a quattro passaggi che garantisce la consegna singola e senza duplicati; ha però un costo maggiore in termini di overhead e latenza, giustificabile solo per comandi realmente critici (es. apertura di una serratura).

[1] nel nostro caso è stato utilizzato il software open-source Mosquitto
[2] ad esempio nel nostro caso il topic è sensori/comandi
[3] ad esempio {"audio": 2}

Il progetto adotta QoS 1 sia lato subscribe del plugin NuTech sia, nei test, lato publish da MQTTX.

4.4. Scelte progettuali

La realizzazione del plugin ha comportato una serie di scelte tecniche vincolate dalla necessità di integrarsi con l'ambiente NuTech e con una libreria di terze parti (Paho MQTT C++)[17] dai requisiti specifici.

4.4.1. Linguaggio: Standard C++17 ISO

La libreria Paho MQTT C++ richiede funzionalità disponibili solo a partire da C++17. È stato pertanto necessario elevare lo standard del linguaggio nelle proprietà di progetto di Visual Studio, come mostrato in Figura 4.2.

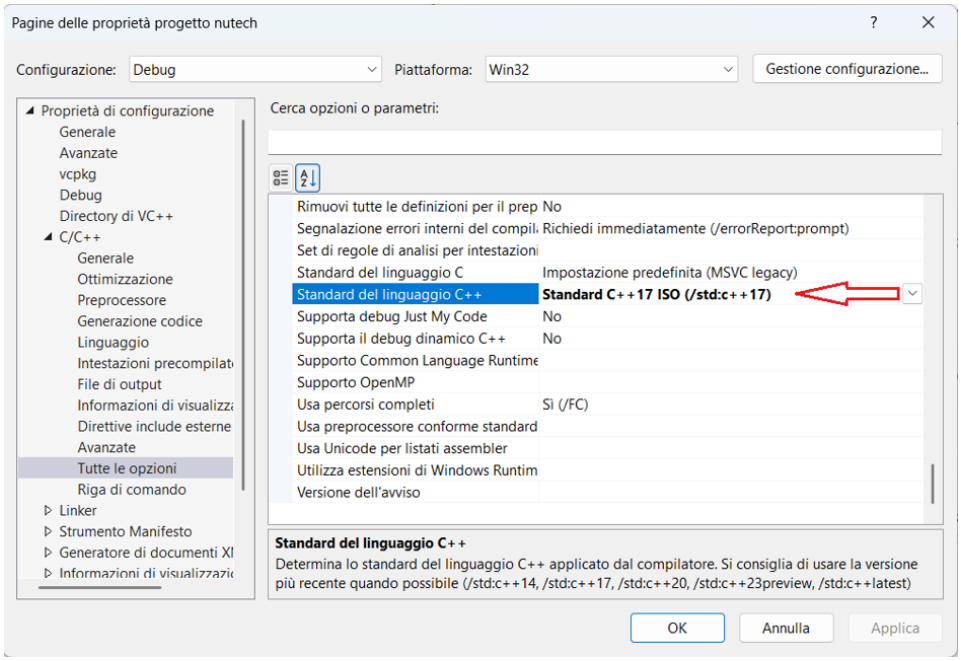


Figura 4.2.: Impostazione dello standard del linguaggio C++17 ISO nelle proprietà del progetto Visual Studio (Configurazione Debug, piattaforma Win32).

4.4.2. Piattaforma di compilazione a 32 bit

NuTech è un host per plugin audio compilato a 32 bit; ogni plugin caricato al suo interno deve pertanto essere compilato come libreria a 32 bit (x86) e collegato esclusivamente a librerie di terze parti anch'esse compilate per la stessa architettura. Questo vincolo si è ripercosso sulla scelta e sulla configurazione di tutte le librerie esterne utilizzate, incluse le rispettive versioni installate tramite vcpkg (variante x86-windows).

4.4.3. Librerie

Sono state selezionate tre librerie esterne, descritte in Tabella 4.1:

Utilizzo	**Libreria**	**Motivazione della scelta**
File audio WAV	dr_wav[19]	Header-only, molto veloce e leggera; nessuna dipendenza da configurare a parte l'inclusione dell'header.
Comunicazione MQTT	Paho MQTT C++[17]	Libreria ufficiale Eclipse, thread-safe, supporta sia MQTT 3.1.1 sia MQTT 5; ampiamente utilizzata in ambito embedded e desktop, con documentazione e community consolidate.
Parsing JSON	nlohmann/json[18]	Header-only (un solo file .hpp), sintassi molto pulita e parsing estremamente semplice, riducendo il codice di gestione degli errori.

Tabella 4.1.: Librerie utilizzate

L'installazione delle librerie Paho MQTT C++[17] e nlohmann/json[18] è stata gestita tramite vcpkg, il package manager ufficiale per C/C++, che si occupa di scaricare automaticamente le versioni corrette delle librerie (in questo caso, target x86-windows).

Durante l'integrazione è stata inoltre necessaria una piccola modifica a un file della libreria Paho stessa: nella riga 96 di thread_queue.h, la costante MAX_CAPACITY è stata ridefinita come segue, per evitare un problema di compilazione legato ai limiti numerici sul target a 32 bit:

```
static constexpr size_type MAX_CAPACITY = (std::numeric_limits<size_type>::
    max)();
```

Infine, affinché NuTech fosse in grado di caricare correttamente il plugin (.dll) risolvendo tutte le dipendenze esterne a runtime, è stato necessario estendere la variabile d'ambiente PATH nelle proprietà di debug del progetto, includendo la cartella delle DLL installate da vcpkg (Figura 1.3):

```
PATH=C:\Users\chiar\vcpkg\installed\x86-windows\debug\bin;%PATH%
```

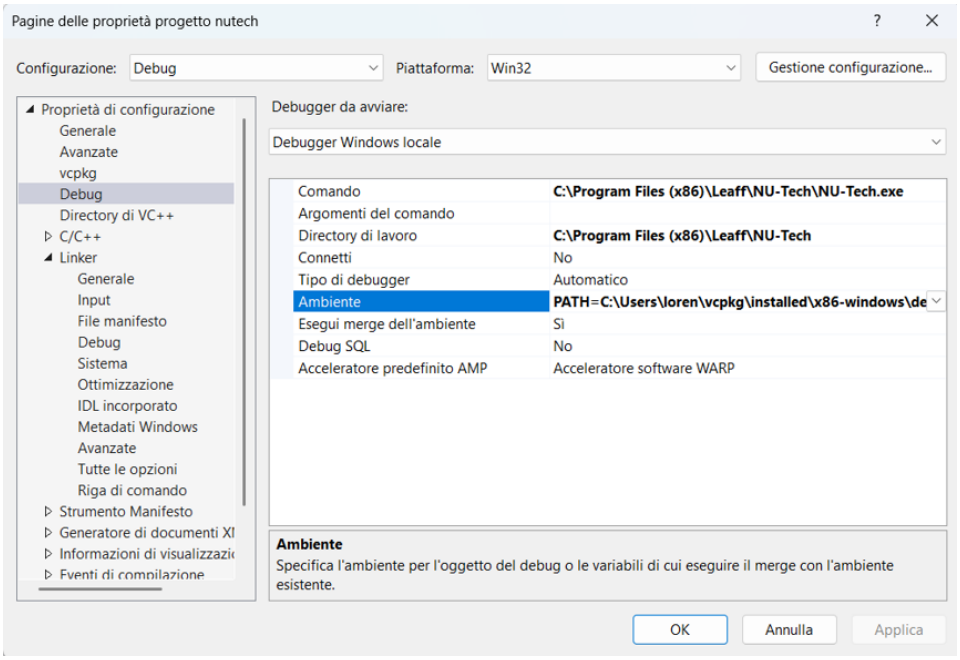


Figura 4.3.: Configurazione della variabile d'ambiente PATH nelle proprietà di debug

4.4.4. Architettura multithread

Una scelta progettuale centrale è stata l'adozione di un'architettura multithread, con due thread distinti che condividono lo stato del plugin:

- **il thread audio**, gestito direttamente da NuTech, che invoca periodicamente LEPlugin_Process() per generare i campioni audio da inviare in output;
- **il thread MQTT**, creato esplicitamente dal plugin in LEPlugin_Init(), dedicato alla connessione al broker, alla sottoscrizione al topic e alla gestione asincrona dei messaggi in arrivo.

La separazione è motivata da un vincolo tipico dell'elaborazione audio in tempo reale: il thread audio non può tollerare blocchi o attese, poiché anche una pausa di 10-50 millisecondi produce artefatti udibili (glitch) nel segnale in uscita. Le operazioni di rete, come connessione, attesa di conferme, gestione di eventuali disconnessioni, hanno invece tempi di risposta non deterministici e potenzialmente lunghi, del tutto incompatibili con i vincoli del thread audio. Isolando la gestione MQTT in un thread separato, il thread audio può limitarsi a leggere una variabile condivisa (pendingAudio), senza mai attendere direttamente la rete.

4.5. Il meccanismo di comunicazione: dalla callback al plugin

Il nucleo concettuale del progetto risiede nel meccanismo che permette a un evento asincrono di rete (l'arrivo di un messaggio MQTT) di influenzare in modo sicuro l'elaborazione audio in corso in un thread differente. I tre punti chiave sono:

- il plugin NON riceve MQTT direttamente;
- il broker NON conosce il plugin;
- il collante fra i due mondi è la callback.

4.5.1. Perché il plugin non riceve MQTT direttamente

La libreria Paho MQTT C++[17] è progettata attorno a un modello a callback: il client MQTT (mqtt::async_client) richiede che l'utilizzatore fornisca un oggetto, derivato dalla classe base mqtt::callback, che la libreria stessa invocherà automaticamente, sul proprio thread interno, ogni volta che si verifica un evento (arrivo di un messaggio, perdita della connessione, conferma di consegna, ecc.). Il plugin NuTech, dal canto suo, non ha alcuna conoscenza di questo meccanismo: espone soltanto i metodi richiesti dall'host NuTech. È quindi necessario un oggetto intermedio che sappia "parlare" sia il linguaggio della libreria Paho sia quello del plugin: questo oggetto è la classe MqttCb.

4.5.2. La classe MqttCb e il meccanismo di override

La classe MqttCb eredita pubblicamente da mqtt::callback e ne sovrascrive il metodo virtuale message_arrived:

```
    class MqttCb : public mqtt::callback {
    PlugIn* self;
public:
    explicit MqttCb(PlugIn* p) : self(p) {}

    void message_arrived(mqtt::const_message_ptr msg) override {
        auto payload = msg->to_string();
        auto j = nlohmann::json::parse(payload);
        int a = j.value("audio", 0);
        if (a == 1 || a == 2) {
            self->pendingAudio.store(a, std::memory_order_release);
        }
    }

    void connection_lost(const std::string&) override { }
};
```

La parola chiave override riveste un ruolo importante come controllo di sicurezza offerto dal compilatore: se il metodo dichiarato non corrisponde esattamente, per firma, a un metodo virtuale della classe base, il compilatore genera un errore invece di compilare silenziosamente una funzione che non sovrascrive nulla. Senza questa garanzia, un semplice errore di firma comporterebbe che, all'arrivo di un messaggio, Paho invocherebbe l'implementazione vuota della classe base anziché quella di MqttCb, e la callback non produrrebbe alcun effetto.

All'interno di message_arrived, il payload del messaggio viene convertito in stringa e sottoposto a parsing tramite nlohmann::json::parse. Il valore associato alla chiave "audio" viene letto con il metodo value, che restituisce un valore di default (0) qualora la chiave non fosse presente, evitando eccezioni in caso di messaggi malformati o non pertinenti. Se il valore letto corrisponde a uno dei due indici di traccia validi (1 o 2), viene aggiornata la variabile condivisa pendingAudio.

4.5.3. Il puntatore self e l'accoppiamento callback–plugin

Affinché la callback sappia a quale istanza di plugin comunicare il comando ricevuto, essa mantiene un puntatore privato self al plugin che l'ha creata, impostato nel costruttore di MqttCb. Nel plugin, la callback viene istanziata passando this come argomento:

```
mqttCb = std::make_unique<MqttCb>(this);
```

In altre parole, l'istanza di PlugIn si registra presso la callback come unico destinatario dei messaggi che questa riceverà.

4.5.4. pendingAudio: il punto di sincronizzazione fra thread

Il campo condiviso fra il thread MQTT e il thread audio è dichiarato come **std::atomic<int>pendingAudio**. L'uso di un tipo atomico è motivata dal vincolo di tempo reale già discusso: il thread audio non può mai bloccarsi in attesa dell'acquisizione. La scrittura, lato callback, utilizza esplicitamente la semantica:

```
self->pendingAudio.store(a, std::memory_order_release);
```

mentre la lettura, lato thread audio, nella versione attuale del codice avviene tramite una semplice assegnazione (int req = pendingAudio;). Solo dopo aver verificato che la riproduzione precedente sia terminata, avviene l'azzeramento di pendingAudio con un'assegnazione separata (pendingAudio = 0;) .

4.5.5. Flusso reale del messaggio

Ricomponendo i tasselli descritti nei paragrafi precedenti, il flusso completo che porta dalla selezione di una traccia alla riproduzione audio nel plugin NuTech si articola nei sette passaggi illustrati in Figura 4.4.

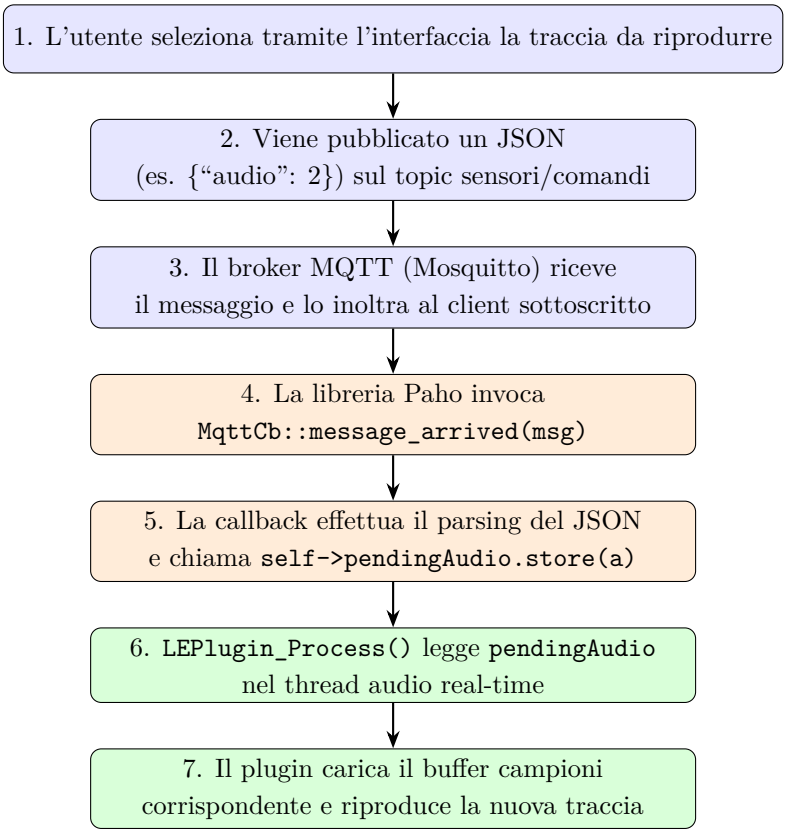


Figura 4.4.: Flusso reale del messaggio

Capitolo 5.

Implementazione della spazializzazione

Introduzione

Questo capitolo entra nel dettaglio implementativo dell'intero plugin: le configurazioni adottate e le porzioni di codice più rilevanti per il precaricamento delle HRTF e la loro selezione in tempo reale.

Per non appesantire la trattazione, non viene riportato il codice per intero, ma solo i frammenti significativi ai fini della comprensione delle scelte implementative; il codice completo del plugin è disponibile nell'appendice A.

5.1. Requisiti di progettazione

Prima di descrivere l'implementazione del sistema esteso, è utile esplicitare i requisiti che ne hanno guidato la progettazione, derivati in parte dall'obiettivo della tesi e in parte dai vincoli già emersi in precedenza:

- **compatibilità con l'architettura esistente**: il sistema deve estendere, senza stravolgere, il plugin NuTech-MQTT descritto nella sezione precedente, riutilizzandone il ciclo di vita (`LEPlugin_Init`/`LEPlugin_Process`/`LEPlugin_Delete`) e il meccanismo di comunicazione già validato;
- **rispetto del vincolo real-time**: come per il sistema di base, nessuna operazione dal tempo di esecuzione imprevedibile può essere eseguita all'interno del thread audio;
- **fedeltà fisica della spazializzazione**: la direzione percepita del suono deve derivare da misurazioni HRTF reali, come discusso nel Capitolo 2;
- **robustezza**: un eventuale problema nel caricamento di una particolare HRTF (file mancante, corrotto, percorso errato) non deve causare l'interruzione della riproduzione audio, ma optare per un comportamento noto.
- **ottimizzazione adeguata all'applicazione**: poiché l'obiettivo finale è segnalare la provenienza approssimativa di un pericolo rispetto all'abitacolo, non è necessaria la risoluzione angolare completa del database di misurazione, ma un numero limitato di direzioni significative;

Le sezioni seguenti mostrano come ciascuno di questi requisiti si sia tradotto in scelte precise.

5.2. Struttura del plugin NuTech: analisi del codice

Come ogni plugin audio per NuTech, anche questo si inserisce in un ciclo di vita: all'atto del caricamento vengono invocati il costruttore e LEPlugin_Init(), che si occupano dell'inizializzazione; durante l'esecuzione, viene richiamato ripetutamente LEPlugin_Process() per ottenere ogni singolo blocco di campioni audio; viene infine invocato LEPlugin_Delete(), responsabile del rilascio delle risorse. Nei paragrafi seguenti si analizzano nel dettaglio le parti di questo ciclo di vita più rilevanti ai fini della comunicazione MQTT.

5.2.1. Costruttore e stato iniziale

Il costruttore di PlugIn interroga l'host NuTech per ottenere la dimensione del buffer audio (FrameSize) e la frequenza di campionamento (SampleRate) correnti, quindi inizializza i contatori di riproduzione. Le dimensioni dei due file audio disponibili (N1 e N2) sono impostate come costanti, ricavate a priori dal numero esatto di campioni dei rispettivi file .wav:

```
PlugIn::PlugIn(InterfaceType _CBFunction,void * _PlugRef,HWND ParentDlg):
    LEEffect(_CBFunction,_PlugRef,ParentDlg) //classe costruttore
{
        FrameSize = CBFunction(this,NUTS_GET_FS_SR,0,(LPVOID)AUDIOPROC);
        SampleRate = CBFunction(this,NUTS_GET_FS_SR,1,(LPVOID)AUDIOPROC);
        k = -1;                      //inizializzato a -1 perchè al primo
            ciclo viene subito incrementato
        N1 = 262094;          //qui vanno scritti il numero di campioni
            precisi verificando la lunghezza del file wav (es su matlab)
        N2 = 1860560;
        N = 0;                       //inizialmente a 0 poi sarà assegnata
            la lunghezza in base alla traccia selezionata giusta nel
            controllo sotto
        samples1 = 0;
        samples2 = 0;
        samples = 0;
        audio = 1;                   //temporaneo
        //pendingAudio = 1;

        LESetNumInput(1);     //obbligatorio a 1 anche se non riceve niente
            in input fisico
        LESetNumOutput(1);
}
```

5.2.2. Caricamento dei file audio in LEPlugin_Init()

All'inizializzazione del plugin, i due file .wav vengono caricati integralmente in memoria tramite la libreria dr_wav[19], che restituisce i campioni come vettore[1] di float.

```
float* data1 = drwav_open_file_and_read_pcm_frames_f32(
"C:\Users\chiar\OneDrive\Desktop\progetto con lorenzo\
    file_example_WAV_1MG.wav",
&channels, &sampleRate, &totalFrames, nullptr);
for (int i = 0; i < N1; i++) samples1[i] = data1[i];
drwav_free(data1, nullptr);
```

I percorsi dei file audio sono specifici del dispositivo di sviluppo.

5.2.3. Avvio del thread MQTT

Sempre in LEPlugin_Init(), viene avviato il thread dedicato alla gestione MQTT. All'interno del thread vengono, in sequenza: istanziati il client MQTT (mqtt::async_client) puntando all'indirizzo del broker; creata e collegata la callback MqttCb; impostate le opzioni di connessione; effettuata la connessione al broker; effettuata la sottoscrizione al topic sensori/comandi con QoS 1; infine, il thread rimane in attesa (con un breve sleep periodico) finché il plugin non viene scaricato, momento in cui la connessione viene chiusa in modo ordinato:

```
    mqttThread = std::thread([this]() {
    mqttClient = std::make_unique<mqtt::async_client>(
        "tcp://192.168.1.46:1883", "nutech_plugin");
    mqttCb = std::make_unique<MqttCb>(this);
    mqttClient->set_callback(*mqttCb);

    mqtt::connect_options opts;
    opts.set_clean_session(true);
    mqttClient->connect(opts)->wait();

    mqttClient->subscribe("sensori/comandi", 1)->wait();

    while (mqttRunning.load())
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

    mqttClient->disconnect()->wait();
});
```

L'indirizzo del broker è cablato nel codice sorgente[2]. La gestione degli errori si limita, per il momento, alla cattura delle eccezioni mqtt::exception con interruzione silenziosa del thread (tramite un log di debug).

[1] i due array dedicati (samples1, samples2)sono allocati in base alle dimensioni N1 e N2

[2] 192.168.1.46, l'indirizzo assegnato al dispositivo che ospitava l'interfaccia di comando nella rete di laboratorio durante lo sviluppo

5.2.4. Il ciclo di elaborazione audio: LEPlugin_Process()

La funzione LEPlugin_Process() viene invocata dall'host NuTech a ogni blocco audio (di dimensione FrameSize) e costituisce il thread real-time del plugin. A ogni chiamata, la funzione: incrementa il contatore di blocco k; verifica se è stato ricevuto un nuovo comando (pendingAudio diverso da zero) e se la riproduzione precedente è terminata; in tal caso, seleziona il buffer campioni corretto (samples1 o samples2) copiandolo nel buffer di riproduzione attivo (samples) e aggiorna la lunghezza totale N; infine copia nel buffer di output i campioni del blocco corrente, applicando un guadagno fisso e completando con zeri quando i campioni disponibili si esauriscono:

```
int req = pendingAudio;
if (req != 0) {
    if ((k - 1) * FrameSize >= N) {
        k = 0;
        pendingAudio = 0;
        if (req == 1) { N = N1; /* copia samples1 -> samples */ }
        else if (req == 2) { N = N2; /* copia samples2 -> samples */ }
    }
}
if (start + FrameSize <= N) {
    for (int n = 0; n < FrameSize; ++n)
        OutputData[n] = 100000 * samples[start + n];
    ...
}
```

5.2.5. Terminazione: LEPlugin_Delete

Alla chiusura del plugin, LEPlugin_Delete() dealloca i buffer audio e, soprattutto, arresta in modo ordinato il thread MQTT. Questa sequenza è importante per evitare fughe di risorse o accessi a memoria non più valida qualora il plugin venga scaricato mentre il thread di rete è ancora attivo.

5.3. Vincolo di tempo reale: precaricamento delle HRTF

La necessità di non introdurre blocchi nel thread audio si applica anche allo stadio di spazializzazione: la lettura dei coefficienti HRTF da un file su disco è un'operazione di I/O, e come tale non è compatibile con l'esecuzione diretta all'interno del ciclo di elaborazione audio (`LEPlugin_Process()`).

Per questo motivo, il caricamento dei file HRTF dal disco è stato collocato esclusivamente nella fase di inizializzazione del plugin (`LEPlugin_Init()`), analogamente a quanto già avviene per il caricamento dei file audio. All'avvio, il plugin legge dal database KEMAR[10] le sei coppie di risposte impulsive corrispondenti alle direzioni di riferimento definite nella sezione precedente, e le mantiene in una struttura dati in memoria. Quando, in fase di esecuzione, arriva un comando MQTT con un nuovo

valore di `azimuth`, il thread audio non accede più al disco: si limita a individuare, tramite la regola di Equazione 3.3, quale coppia di coefficienti già in memoria utilizzare, e a copiarla nei buffer attivi del filtro. Questa copia in memoria, a differenza di una lettura da disco, ha un tempo di esecuzione deterministico e trascurabile, compatibile con i vincoli del thread audio.

La Figura 5.1 riassume questa separazione tra fase di inizializzazione (I/O da disco, eseguita una sola volta) e fase di esecuzione (accesso alla sola memoria, eseguita a ogni cambio di direzione).

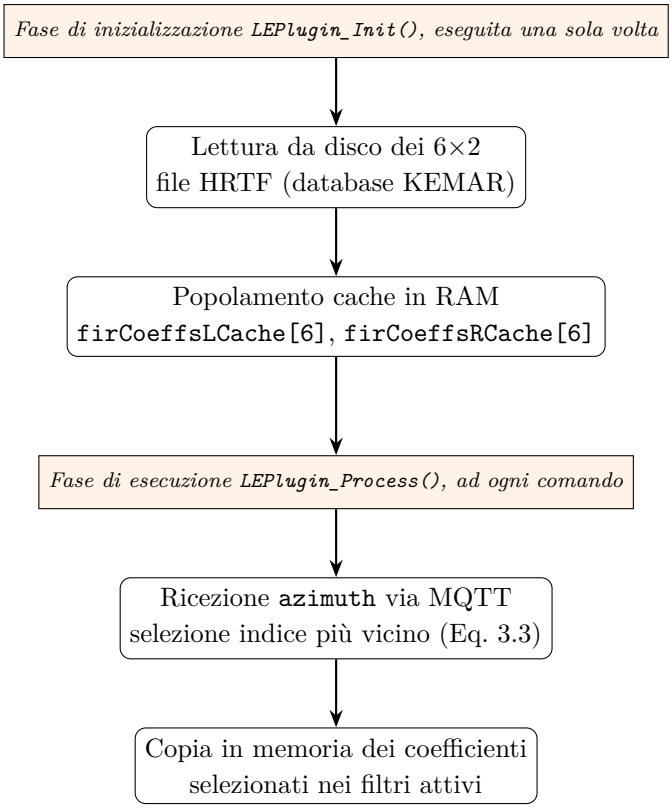


Figura 5.1.: Separazione di fase di inizializzazione e fase di esecuzione

5.4. Evoluzione rispetto al sistema base

È opportuno chiarire due apparenti discrepanze numeriche rispetto a quanto dichiarato in precedenza, che non sono errori ma il riflesso di due fasi distinte del lavoro.

5.4.1. I file audio di riferimento e le costanti N1, N2

Nel sistema di comunicazione di base descritto nella sezione precedente, i due file audio disponibili avevano lunghezza $N_1 = 262094$ e $N_2 = 1860560$ campioni. Nella fase di estensione verso la spazializzazione, questi stessi file sono stati sostituiti con nuovi segnali di test più adatti a verificare in modo rigoroso il corretto funzionamento

del filtro FIR: un segnale di rumore bianco di durata nota (tre secondi a 44.1 kHz, generato in MATLAB) al posto del primo file, e un brano musicale convertito in mono al posto del secondo. Le costanti sono state quindi aggiornate di conseguenza:

$$N_1 = 132300, \qquad N_2 = 10196352 \tag{5.1}$$

Questa sostituzione è stata necessaria perché il rumore bianco è il segnale di test ideale per verificare che un filtro identità lasci l'uscita invariata e, più avanti, che le due HRTF applichino trasformazioni effettivamente diverse ai due canali. Nel seguito del capitolo e nel successivo, ogni riferimento a N_1 e N_2 è da intendersi relativo a questi nuovi valori, non ai precedenti.

5.4.2. Numero di tap del filtro (firTaps)

Analogamente, nei primi test del filtro FIR (Sezione **??**, filtro identità e delta ritardata) il numero di coefficienti `firTaps` era stato fissato arbitrariamente a 128, un valore scelto solo per semplicità nella fase in cui i coefficienti erano ancora sintetici (un impulso di Dirac). Nella versione finale del sistema, che utilizza le risposte impulsive reali del database KEMAR[10], `firTaps` è stato portato a 512:

```
firTaps = 512;
```

Questo valore non è una scelta di progetto ma un vincolo imposto dal formato dei dati sorgente: ogni file del database KEMAR contiene esattamente 512 campioni per risposta impulsiva, motivo per cui la funzione `LoadHRTF` (Sezione 5.7) verifica esplicitamente che il file letto contenga almeno `nTaps` campioni prima di procedere.

5.5. Librerie e ambiente di sviluppo

L'estensione del plugin per l'audio immersivo non si è limitata all'aggiunta della sola libreria Intel IPP[20], ma ha richiesto anche una modifica più ampia dell'ambiente di compilazione rispetto a quanto descritto in precedenza.

5.5.1. Migrazione alla compilazione a 64 bit

Il sistema di base era compilato interamente a 32 bit (piattaforma Win32), un vincolo dettato dall'host NuTech impiegato in quella fase (Paragrafo 4.4.2). Tuttavia, per poter integrare la parte di audio immersivo è necessaria la migrazione dell'intero progetto alla compilazione a 64 bit (x64), sia per le configurazioni Debug sia per quella Release.

Lo standard del linguaggio è stato mantenuto a ISO C++17, come nel progetto originario.

Questa migrazione ha richiesto la reinstallazione, per il nuovo target, di tutte le librerie di terze parti già impiegate. Tramite Windows PowerShell, le librerie Paho

MQTT C++ e nlohmann/json sono state reinstallate specificando esplicitamente la versione 64 bit di vcpkg:

```
.\vcpkg install paho-mqttpp3:x64-windows
.\vcpkg install paho-mqtt-c:x64-windows
.\vcpkg install nlohmann-json:x64-windows
```

Conseguentemente, tutti i percorsi configurati nelle proprietà del progetto Visual Studio sono stati aggiornati per puntare alla nuova cartella di installazione:

- **C/C++ → General → Additional Include Directories**:
  `vcpkg\installed\x64-windows\include`
- **Linker → General → Additional Library Directories**:
  `vcpkg\installed\x64-windows\lib` (configurazione Release),
  `vcpkg\installed\x64-windows\debug\lib` (configurazione Debug)[3];
- **Linker → Input → Additional Dependencies**:
  `paho-mqtt3as.lib; paho-mqttpp3.lib; paho-mqtt3a.lib`

La migrazione ha richiesto la risoluzione di alcune criticità non banali, in larga parte legate a incompatibilità tra la libreria Paho MQTT C++ e le impostazioni di compilazione del progetto NuTech:

- alla riga 96 del file `thread_queue.h` della libreria Paho, la costante `MAX_CAPACITY` è stata ridefinita esplicitamente come
  ```
  static constexpr size_type MAX_CAPACITY =
      (std::numeric_limits<size_type>::max)();
  ```
  per un problema di compilazione analogo a quello già incontrato, sulla stessa libreria, nell'implementazione del sistema di comunicazione (vedi Paragrafo 4.4.3), qui ripresentatosi nella versione a 64 bit;
- in **Progetto → Proprietà → C/C++ → Code Generation → Runtime Library** è stato necessario impostare, per le due configurazioni:

  Multi-threaded Debug DLL (`/MDd`) per Debug,

  Multi-threaded DLL (`/MD`) per Release,

  per allineare il runtime del progetto a quello con cui erano state compilate le librerie vcpkg;
- la macro `PAHO_MQTTPP_IMPORTS`, anziché essere definita nel codice sorgente, è stata aggiunta globalmente in

  Project → Properties → C/C++ → Preprocessor → Preprocessor Definitions

[3] le librerie Debug e Release, essendo compilate con impostazioni del runtime differenti, sono infatti mantenute in cartelle separate

per entrambe le configurazioni, e inserita manualmente anche nel file di progetto `.vcxproj` [4], tra `<PreprocessorDefinitions>` e `WIN32`, per entrambe le righe di configurazione Debug e Release;

Una volta risolti gli errori del linker, il plugin compilava correttamente ma non veniva caricato da NuTech (il "nuts" non appariva). La causa è stata individuata nell'assenza, nella variabile d'ambiente di debug, del percorso alle DLL installate da vcpkg per il nuovo target.

È stato quindi necessario estendere, in

Project → Properties → Configurazione → Debug → Ambiente

per entrambe le configurazioni, l'ambiente:

`PATH=C:\Users\chiar\vcpkg\installed\x64-windows\debug\bin`

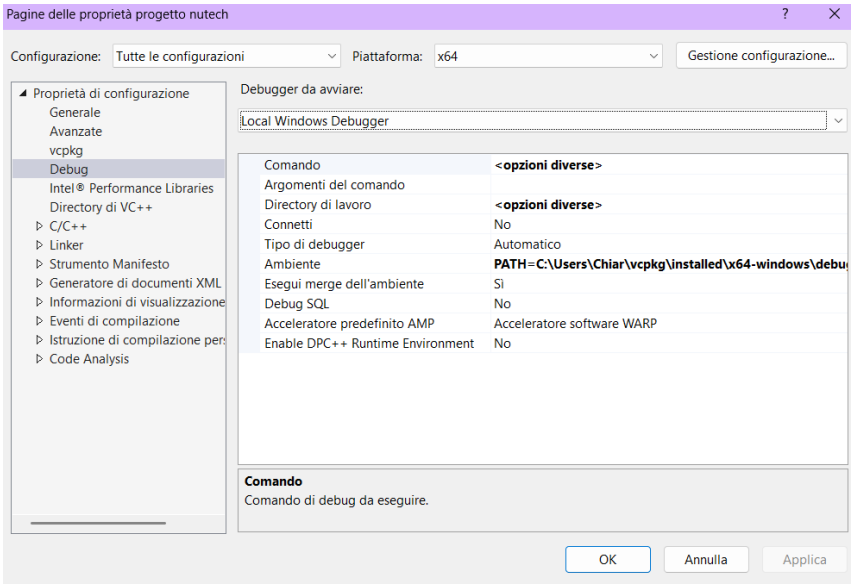


Figura 5.2.: Configurazione PATH per la configurazione Debug a 64 bit.

5.5.2. Integrazione di Intel IPP

Come richiamato in apertura di sezione, la migrazione a 64 bit è stata condizione necessaria per poter integrare **Intel® Integrated Performance Primitives (IPP)**, parte del pacchetto Intel oneAPI Base Toolkit, unica libreria aggiuntiva richiesta rispetto all'ambiente precedente.

Dopo l'installazione, Visual Studio dovrebbe presentare automaticamente l'integrazione con le librerie Intel: è stato quindi verificato, nelle proprietà del progetto (configurazione Debug), che comparisse l'opzione *Intel Libraries for oneAPI* e che le impostazioni correlate[5] fossero impostate sulle versioni più recenti disponibili.

[4] aperto come testo e modificato tramite Blocco Note

[5] versione del Windows SDK e set di strumenti della piattaforma (*platform toolset*)

Verificata l'integrazione automatica, si è comunque proceduto, a una configurazione esplicita dei percorsi Include, Lib e delle dipendenze del linker, per avere piena tracciabilità della configurazione del progetto indipendentemente dal plugin Intel.

- **Include Directories**:
  `C:\Program Files (x86)\Intel\oneAPI\ipp\latest\include`
- **Library Directories**:
  `C:\Program Files (x86)\Intel\oneAPI\ipp\latest\lib`
- **Additional Dependencies**: `ippi.lib; ipps.lib; ippcore.lib`

Le tre dipendenze corrispondono ai moduli IPP effettivamente utilizzati dal plugin: `ippcore`, che fornisce le funzioni di supporto e inizializzazione, e `ipps`, il modulo di *signal processing* che contiene le funzioni di filtraggio FIR impiegate per l'HRTF [6].

Nel codice è quindi sufficiente includere l'header principale della libreria, `#include <ipp.h>`, per avere accesso a tutte le funzioni dei moduli configurati.

In fase di *Linking* è stata scelta la variante *Static Library* delle IPP.

In fase di prima esecuzione, IPP non individuava correttamente due DLL specifiche: il problema è stato risolto copiando manualmente le due DLL dalla cartella `bin` dell'installazione oneAPI alla cartella dell'eseguibile NuTech.

5.6. Il pattern GetSize/Init/Process applicato al filtro FIR binaurale

Ogni funzione con stato della libreria Intel IPP[7] segue un pattern a tre fasi a livello di singola funzione:

- `GetSize`, che calcola la quantità di memoria necessaria per lo stato del filtro;
- `Init`, che inizializza tale stato con i coefficienti;
- `Process`, che applica il filtro a un blocco di campioni.

Questo pattern non va confuso con il ciclo di vita del plugin NuTech descritto nel Paragrafo 5.2[8] sebbene i due schemi siano strettamente correlati: nel codice del plugin, sia `GetSize` sia `Init` [9] vengono eseguite in sequenza, una sola volta, all'interno di `LEPlugin_Init()` mentre `Process`[10] corrisponde direttamente a `LEPlugin_Process()`, chiamata a ogni blocco audio nel thread real-time.

[6] `ippi`, il modulo per l'elaborazione di immagini, non è di fatto utilizzato dal codice del plugin ma è stato mantenuto tra le dipendenze in fase di configurazione iniziale del progetto

[7] tra cui il filtro FIR impiegato in questo lavoro

[8] costruttore e `LEPlugin_Init()` eseguiti una sola volta, `LEPlugin_Process()` invocata ripetutamente a ogni blocco audio, `LEPlugin_Delete()` eseguita allo scarico

[9] le prime due fasi del pattern IPP

[10] la terza fase del pattern IPP

Questa collocazione non è casuale: allocare la memoria per lo stato del filtro (`GetSize`) e inizializzarlo (`Init`) sono operazioni con un costo computazionale non trascurabile e non deterministico, che devono avvenire fuori dal loop audio esattamente come il caricamento dei coefficienti stessi, per non violare il vincolo di tempo reale già discusso.

Questo principio, per un sistema con due canali indipendenti (sinistro e destro), si traduce nella necessità di mantenere *due* stati distinti del filtro, ciascuno con la propria delay line, poiché i due canali applicano in generale coefficienti diversi.

5.6.1. Variabili membro del plugin

In `Plugin.h` sono state introdotte le seguenti variabili (private alla classe `PlugIn`), che sostituiscono la precedente versione a canale singolo descritta nei primi test:

```
// coefficienti dei due filtri
Ipp64f* firCoeffsL;
Ipp64f* firCoeffsR;

// stato persistente dei due filtri
IppsFIRSpec_64f* firStateL;
IppsFIRSpec_64f* firStateR;

// buffer temporanei di lavoro
Ipp8u* firBufL;
Ipp8u* firBufR;

// delay line esplicite: gestite a parte per poter
// essere azzerate al cambio di direzione senza reinizializzare lo stato
Ipp64f* dlyL;
Ipp64f* dlyR;

int firTaps; // numero di coefficienti (= 512)
```

A queste si aggiungono, per il precaricamento descritto in Sezione 5.3, i vettori di cache per le sei direzioni di riferimento (0°, 70°, 160°, 180°, 200°, 290°):

```
static const int NUM_AZIMUTHS = 6;
int azimuthSet[NUM_AZIMUTHS] = { 0, 200, 160, 70, 290, 180 };

Ipp64f* firCoeffsLCache[NUM_AZIMUTHS];
Ipp64f* firCoeffsRCache[NUM_AZIMUTHS];
```

Tutti i puntatori vengono inizializzati a `nullptr` nel costruttore, in modo che il controllo di sicurezza in `LEPlugin_Process()` possa distinguere in modo affidabile uno stato non ancora allocato da uno stato valido.

5.6.2. Allocazione in LEPlugin_Init()

L'allocazione dello stato dei due filtri avviene una sola volta, in `LEPlugin_Init()`, secondo il pattern GetSize/Init:

```
firTaps = 512;

firCoeffsL = new Ipp64f[firTaps];
firCoeffsR = new Ipp64f[firTaps];

int specSize = 0, bufSize = 0;
ippsFIRSRGetSize(firTaps, ipp64f, &specSize, &bufSize);

firStateL = (IppsFIRSpec_64f*) new Ipp8u[specSize];
firBufL  = new Ipp8u[bufSize];
firStateR = (IppsFIRSpec_64f*) new Ipp8u[specSize];
firBufR  = new Ipp8u[bufSize];

dlyL = ippsMalloc_64f(firTaps - 1);
ippsZero_64f(dlyL, firTaps - 1);
dlyR = ippsMalloc_64f(firTaps - 1);
ippsZero_64f(dlyR, firTaps - 1);
```

Si noti che `ippsFIRSRGetSize` viene chiamata una sola volta: le dimensioni richieste (`specSize`, `bufSize`) dipendono unicamente dal numero di tap e dal tipo di dato, non dai coefficienti specifici, quindi sono identiche per i due canali.

Oltre allo stato dei filtri, in `LEPlugin_Init()` vengono allocati anche i buffer persistenti usati nel ciclo di elaborazione (`filteredBlockL`, `filteredBlockR`, `lastBlock`), come membri della classe anziché come variabili locali di `LEPlugin_Process()`: allocarle a ogni chiamata introdurrebbe infatti un costo non deterministico nel thread audio, esattamente il tipo di operazione che il vincolo di tempo reale impone di evitare.

5.7. Caricamento delle HRTF: la funzione LoadHRTF

Ogni file del database KEMAR (Capitolo 2) contiene 512 campioni a 16 bit, che la libreria dr_wav già in uso per i file WAV standard è in grado di leggere.

La funzione `LoadHRTF`, definita come funzione libera (non membro della classe) nel file `Plugin.cpp`, incapsula questa lettura:

```
static bool LoadHRTF(const char* path, Ipp64f* out, int nTaps)
{
    unsigned int channels, sampleRate;
    drwav_uint64 totalFrames;

    float* data = drwav_open_file_and_read_pcm_frames_f32(
        path, &channels, &sampleRate, &totalFrames, nullptr);
```

```
    if (!data) {
        OutputDebugStringA(("[HRTF] File non trovato: " +
            std::string(path) + "\n").c_str());
        return false;
    }
    if ((int)totalFrames < nTaps) {
        OutputDebugStringA("[HRTF] File troppo corto\n");
        drwav_free(data, nullptr);
        return false;
    }

    for (int i = 0; i < nTaps; i++)
        out[i] = data[i]; // gia' normalizzato in [-1.0, +1.0] da dr_wav

    drwav_free(data, nullptr);
    return true;
}
```

Il valore di ritorno booleano permette di gestire in modo robusto l'eventualità che un file non sia presente o sia corrotto, requisito esplicitamente indicato tra i requisiti di progettazione del Paragrafo 5.1: in caso di fallimento, il chiamante sostituisce i coefficienti mancanti con un filtro identità (`firCoeffs[0] = 1.0`), garantendo che il plugin continui comunque a riprodurre audio, seppur privo di spazializzazione per quella direzione, invece di arrestarsi o produrre silenzio.

5.8. Precaricamento e cache delle direzioni

Come motivato in Sezione 5.3, l'accesso al disco non può avvenire nel thread audio. In `LEPlugin_Init()`, subito dopo l'allocazione dello stato dei filtri, il plugin carica quindi tutte e sei le coppie di HRTF corrispondenti alle direzioni di riferimento, popolando la cache:

```
for (int i = 0; i < NUM_AZIMUTHS; i++) {
    firCoeffsLCache[i] = new Ipp64f[firTaps];
    firCoeffsRCache[i] = new Ipp64f[firTaps];

    char pathL[256], pathR[256];
    sprintf(pathL, "...\\full\\elev0\\L0e%03da.wav", azimuthSet[i]);
    sprintf(pathR, "...\\full\\elev0\\R0e%03da.wav", azimuthSet[i]);

    if (!LoadHRTF(pathL, firCoeffsLCache[i], firTaps)) {
        ippsZero_64f(firCoeffsLCache[i], firTaps);
        firCoeffsLCache[i][0] = 1.0;
    }
    if (!LoadHRTF(pathR, firCoeffsRCache[i], firTaps)) {
        ippsZero_64f(firCoeffsRCache[i], firTaps);
```

```
        firCoeffsRCache[i][0] = 1.0;
    }
}
```

Il nome dei file segue la convenzione del database KEMAR[11] con l'azimuth codificato su tre cifre tramite `sprintf`.

5.9. Selezione della direzione in tempo reale: ApplyHRTF

La funzione `ApplyHRTF`, membro della classe `PlugIn`, è il punto di incontro fra il valore di azimuth ricevuto via MQTT e i coefficienti già in memoria. Nella sua versione finale, ottimizzata rispetto alla prima implementazione (che leggeva i file da disco a ogni cambio di direzione), la funzione non compie alcuna operazione di I/O:

```
void PlugIn::ApplyHRTF(int azimuth)
{
    // normalizza l'angolo richiesto in [0, 360)
    azimuth = ((azimuth % 360) + 360) % 360;

    // trova, tra le 6 direzioni disponibili, quella a distanza
    // angolare circolare minima (Equazione 3.2)
    int bestIdx = 0, bestDist = 361;
    for (int i = 0; i < NUM_AZIMUTHS; i++) {
        int d = abs(azimuth - azimuthSet[i]);
        d = (std::min)(d, 360 - d);
        if (d < bestDist) { bestDist = d; bestIdx = i; }
    }

    // copia dalla cache in memoria: nessun accesso al disco
    memcpy(firCoeffsL, firCoeffsLCache[bestIdx], firTaps * sizeof(Ipp64f));
    memcpy(firCoeffsR, firCoeffsRCache[bestIdx], firTaps * sizeof(Ipp64f));

    // reinizializza lo stato del filtro con i nuovi coefficienti
    ippsFIRSRInit_64f(firCoeffsL, firTaps, ippAlgAuto, firStateL);
    ippsFIRSRInit_64f(firCoeffsR, firTaps, ippAlgAuto, firStateR);

    // azzera le delay line per evitare artefatti alla transizione
    ippsZero_64f(dlyL, firTaps - 1);
    ippsZero_64f(dlyR, firTaps - 1);

    currentAzimuth = azimuthSet[bestIdx];
}
```

Si osservi che il ciclo di ricerca dell'indice `bestIdx` non è altro che l'implementazione diretta dell'Equazione 3.2 definita in Sezione 3.3: per ciascuna delle direzioni disponibili in `azimuthSet`, il codice calcola la distanza angolare circolare $d(\theta, \theta_i)$

[11] L0e090a.wav = orecchio sinistro, elevazione 0°, azimuth 90°

e mantiene traccia dell'indice a distanza minima, esattamente come formalizzato dall'operatore di minimizzazione dell'equazione. La normalizzazione dell'angolo richiesto nell'intervallo [0°, 360°), eseguita nella prima riga della funzione, garantisce che la formula resti valida anche per valori di azimuth negativi o superiori a 360°eventualmente ricevuti via MQTT.

5.9.1. Perché una delay line esplicita

Nel filtro a canale singolo, la delay line[12] veniva lasciata gestire interamente a IPP, passando `nullptr` come quarto e quinto parametro di `ippsFIRSR_64f`:

```
ippsFIRSR_64f(pSrc, pDst, numIters, firState, nullptr, nullptr, firBuf);
```

Questo approccio funziona correttamente finché i coefficienti del filtro restano fissi per tutta la durata dell'esecuzione, come nei test di continuità iniziali (Paragrafo **??**).

Con l'introduzione del cambio di direzione a runtime, questo schema è però diventato insufficiente.

Per poter azzerare la delay line indipendentemente dal resto dello stato, essa è stata quindi estratta e allocata esplicitamente come buffer a parte, tramite le funzioni di allocazione allineata di IPP:

```
dlyL = ippsMalloc_64f(firTaps - 1);
ippsZero_64f(dlyL, firTaps - 1);
dlyR = ippsMalloc_64f(firTaps - 1);
ippsZero_64f(dlyR, firTaps - 1);
```

e passata esplicitamente a `ippsFIRSR_64f` sia come delay line di ingresso sia come delay line di uscita:

```
ippsFIRSR_64f(samples + start, filteredBlockL, FrameSize, firStateL, dlyL,
    dlyL, firBufL);
```

In questo modo, `ApplyHRTF` può azzerare esplicitamente la "storia" del filtro subito dopo aver caricato i nuovi coefficienti:

```
ippsFIRSRInit_64f(firCoeffsL, firTaps, ippAlgAuto, firStateL);
ippsZero_64f(dlyL, firTaps - 1); //nessun mix fra HRTF diverse
```

Questa è dunque un'evoluzione reale del codice: la gestione automatica della delay line (`nullptr`, `nullptr`) resta corretta e più semplice per un filtro a coefficienti fissi, mentre la gestione esplicita si è resa necessaria per supportare il cambio di HRTF a runtime.

[12] cioè la "memoria" degli ultimi `firTaps`−1 campioni di ingresso, necessaria per garantire la continuità della convoluzione tra un blocco e il successivo

5.10. Estensione del protocollo MQTT: il campo azimuth

Coerentemente con l'estensione del formato del comando descritta in Sezione 3.1, la callback `MqttCb::message_arrived` è stata modificata per leggere, oltre al campo `audio` già presente nel sistema originario, anche il nuovo campo `azimuth`:

```
void message_arrived(mqtt::const_message_ptr msg) override {
    try {
        auto payload = msg->to_string();
        auto j = nlohmann::json::parse(payload);

        int a = j.value("audio", 0);
        int az = j.value("azimuth", 90); // default 90 se assente

        if (a == 1 || a == 2) {
            self->pendingAzimuth.store(az, std::memory_order_release);
            self->pendingAudio.store(a, std::memory_order_release);
        }
    }
    catch (...) { DBG("[MQTT] JSON invalido"); }
}
```

La variabile `pendingAzimuth`, dichiarata come `std::atomic<int>` in `Plugin.h` analogamente a `pendingAudio` (Paragrafo 4.5.4), viene scritta dal thread MQTT e letta dal thread audio in `LEPlugin_Process()` solo nel momento in cui inizia la riproduzione di una nuova traccia:

```
if (req == 1) {
    N = N1;
    for (int n = 0; n < N1; ++n) samples[n] = samples1[n];
    ApplyHRTF(pendingAzimuth.load());
}
else if (req == 2) {
    N = N2;
    for (int n = 0; n < N2; ++n) samples[n] = samples2[n];
    ApplyHRTF(pendingAzimuth.load());
}
```

In questo modo la direzione richiesta viene applicata una sola volta per riproduzione, all'atto del cambio traccia.

5.11. Il ciclo di elaborazione a due canali

La funzione `LEPlugin_Process()` è stata estesa per operare su due uscite indipendenti (`Output[0]`, `Output[1]`), dichiarate tramite `LESetNumOutput(2)` nel costruttore e riflesse nel campo `NumOutStartUp = 2` della struttura `StartUpNUTSProps` restituita da `LENUTSDefProps`.

All'inizio della funzione, un controllo di sicurezza verifica che entrambi gli stati dei filtri siano stati correttamente allocati prima di qualunque elaborazione:

```
if (firStateL == nullptr || firStateR == nullptr ||
    firBufL == nullptr || firBufR == nullptr)
    return COMPLETED;
```

Per il blocco corrente, i due filtraggi vengono eseguiti in sequenza sullo stesso segmento di ingresso (`samples + start`), ciascuno con il proprio stato e la propria delay line:

```
ippsFIRSR_64f(samples + start, filteredBlockL, FrameSize,
              firStateL, dlyL, dlyL, firBufL);
ippsFIRSR_64f(samples + start, filteredBlockR, FrameSize,
              firStateR, dlyR, dlyR, firBufR);

for (int n = 0; n < FrameSize; ++n) {
    OutputDataL[n] = filteredBlockL[n] * 32768.0;
    OutputDataR[n] = filteredBlockR[n] * 32768.0;
}
```

Il fattore di guadagno 32768.0 (pari a 2^{15}) non è un'amplificazione arbitraria: come emerso durante i test descritti nel Capitolo 6, la scheda audio utilizzata si aspetta campioni nell'intervallo a 16 bit con segno (-32768 / $+32767$), mentre i campioni prodotti dal filtraggio IPP restano normalizzati. Il fattore 2^{15} riporta quindi il segnale nell'intervallo atteso dall'hardware, senza introdurre alcuna distorsione percettibile[13], a differenza dei tentativi preliminari di normalizzazione o guadagno applicati direttamente sui coefficienti del filtro, discussi in dettaglio nel Capitolo 6.

5.12. Gestione della memoria

In coerenza con il pattern GetSize/Init/Process, il rilascio delle risorse allocate è centralizzato in `LEPlugin_Delete()`:

```
if (firCoeffsL) { delete[] firCoeffsL; firCoeffsL = nullptr; }
if (firCoeffsR) { delete[] firCoeffsR; firCoeffsR = nullptr; }
if (firBufL)  { delete[] firBufL;  firBufL = nullptr; }
if (firBufR)  { delete[] firBufR;  firBufR = nullptr; }
if (firStateL) { delete[] (Ipp8u*)firStateL; firStateL = nullptr; }
if (firStateR) { delete[] (Ipp8u*)firStateR; firStateR = nullptr; }
if (dlyL) { ippsFree(dlyL); dlyL = nullptr; }
if (dlyR) { ippsFree(dlyR); dlyR = nullptr; }

for (int i = 0; i < NUM_AZIMUTHS; i++) {
    if (firCoeffsLCache[i]) { delete[] firCoeffsLCache[i]; firCoeffsLCache[i
        ] = nullptr; }
```

[13] gravi problemi di instabilità, rumore di quantizzazione e overflow interni

```
    if (firCoeffsRCache[i]) { delete[] firCoeffsRCache[i]; firCoeffsRCache[i
        ] = nullptr; }
}
```

Si noti l'uso di `ippsFree` anziché `delete[]` per i buffer allocati con `ippsMalloc_64f` (`dlyL`, `dlyR`): IPP fornisce funzioni di allocazione/deallocazione dedicate, allineate in memoria in modo ottimale, che non devono essere miscelate con l'allocatore standard.

5.13. Una modalità di test dedicata: modalitaFileRead

Per isolare la verifica del solo stadio di filtraggio FIR dal resto[14], è stato introdotto un flag booleano, membro della classe `PlugIn`:

```
bool modalitaFileRead = false; // true = test con FileRead/FileWrite di
    NuTech
```

Quando impostato a `true`, il ramo corrispondente in `LEPlugin_Process()` filtra direttamente il segnale ricevuto dall'`InputData` del pin di ingresso, cioè un blocco `FileRead` di NuTech collegato esternamente al nuts, come in Figura 5.3, anziché dal buffer interno `samples` caricato nel costruttore, e ignora del tutto il meccanismo di comando via MQTT:

```
if (modalitaFileRead) {
    ippsFIRSR_64f(InputData, OutputDataL, FrameSize, firStateL, dlyL, dlyL,
        firBufL);
    ippsFIRSR_64f(InputData, OutputDataR, FrameSize, firStateR, dlyR, dlyR,
        firBufR);
    return COMPLETED;
}
```

[14] caricamento dei file audio, gestione del thread MQTT

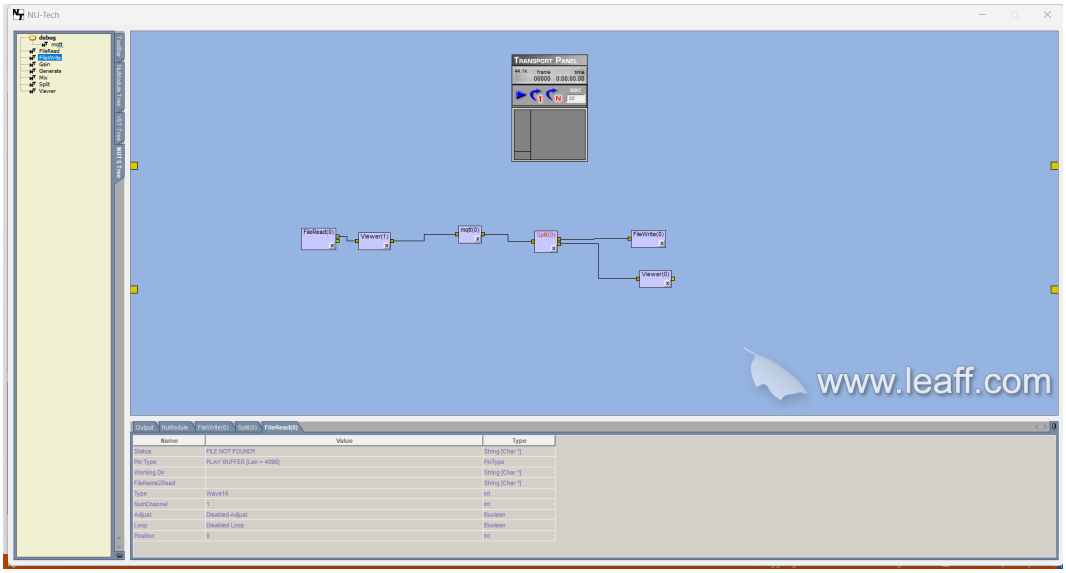


Figura 5.3.: Setup per i test in modalitaFileRead

Questa modalità è stata impiegata esclusivamente nella fase di verifica del filtraggio FIR di base (Sezione **??**): continuità dello stato tra blocchi consecutivi, confronto campione per campione con MATLAB, e confronto tra i due canali tramite un blocco Mix di NuTech con guadagni opposti.

In tutti questi test l'obiettivo era isolare la correttezza matematica della convoluzione IPP [15] da qualunque altra componente del sistema, incluso il meccanismo di comando remoto.

Una volta verificato che il filtraggio funzionasse correttamente in questa configurazione semplificata, tutti i test relativi alla direzionalità percepita del suono e al funzionamento del sistema (Capitolo 6 e , incluso il test con MQTTX) sono stati condotti con `modalitaFileRead = false`, cioè nella modalità di funzionamento reale del plugin: segnale caricato dai file WAV interni, comando di traccia e direzione ricevuto via MQTT, filtraggio.

Il flag resta presente nel codice come strumento di debug per eventuali verifiche future dello stadio di filtraggio isolato dal resto della pipeline, ma non interviene nel funzionamento operativo del sistema.

[15] identità, delta ritardata, coefficienti HRTF caricati correttamente

Capitolo 6.

Test e risultati

Introduzione

Questo capitolo presenta la verifica sperimentale del sistema, condotta per gradi di complessità crescente. Si parte dal sistema di comunicazione di base descritto nel Capitolo 4: la Sezione 6.1.1 illustra la configurazione dell'ambiente di sviluppo e di test, e la Sezione 6.1.2 riporta i test funzionali della comunicazione MQTT in modo da verificare sia il funzionamento di base della comunicazione MQTT sia la sua estensione con il campo di azimuth.

Seguono la verifica della correttezza del filtraggio FIR, trattata in Sezione 6.2, e la verifica delle HRTF sui due canali, descritta in 6.2.2.Chiude il capitolo la sezione dei **test sperimentali** (Sezione 6.3), che verifica il sistema completo, prima in condizioni di ascolto isolato e poi in un test di percezione condotto con il simulatore di guida del laboratorio.

Come anticipato in Sezione 5.13, le verifiche sul filtraggio (Sezione 6.2.1) sono state condotte con `modalitaFileRead = true`, cioè in isolamento dal resto della pipeline: continuità dello stato tra blocchi, comportamento con filtro identità e con delta ritardata, confronto quantitativo con un'implementazione di riferimento in MATLAB, indipendenza dei due canali e caricamento delle HRTF reali dal database KEMAR. Le verifiche successive sul sistema completo (comunicazione MQTT estesa e test sperimentali) sono state condotte con `modalitaFileRead = false`, cioè nella modalità di funzionamento reale del plugin.

Aver verificato la correttezza del filtro separatamente dalla catena di comunicazione e dall'ascolto ha permesso, come emerso in Sezione 6.3.1, di individuare più rapidamente la causa di problematiche che altrimenti sarebbe stato difficile diagnosticare, avendo troppe componenti in gioco contemporaneamente.

6.1. Validazione del sistema di comunicazione

6.1.1. Configurazione di Mosquitto e MQTTX

Per poter testare il funzionamento del plugin indipendentemente dal client scelto, isolando così eventuali problemi del plugin da eventuali problemi di configurazione di esso, è stata impiegata la combinazione Mosquitto + MQTTX: Mosquitto[21] come

broker MQTT locale, MQTTX come client con interfaccia grafica per pubblicare manualmente i messaggi JSON sul topic concordato.

Una parte rilevante del lavoro ha riguardato la configurazione dell'ambiente necessario per testare il sistema in condizioni realistiche.

Per riprodurre fedelmente le condizioni dell'installazione finale, il codice NuTech veniva eseguito su un primo calcolatore (PC A), mentre broker e client MQTTX erano installati su un secondo calcolatore (PC B), entrambi connessi alla medesima rete Wi-Fi.

Di default, Mosquitto[21] su Windows si avvia in modalità local-only, restando in ascolto esclusivamente sull'indirizzo di loopback (127.0.0.1) e rifiutando quindi connessioni da altri host della rete locale, come mostrato in Figura 6.1.

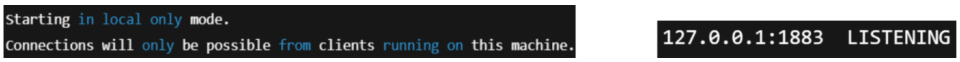


Figura 6.1.: Mosquitto avviato in modalità local-only

È stato pertanto necessario aggiungere al file di configurazione di Mosquitto le due righe seguenti:

```
listener 1883
allow_anonymous true
```

La prima riga apre l'ascolto del broker su tutte le interfacce di rete alla porta standard MQTT (1883), rendendolo raggiungibile da qualunque host della LAN all'indirizzo IP_PC_B:1883; la seconda consente la connessione di client MQTT privi di credenziali.

Il client MQTTX[22] è stato configurato con una connessione al broker locale su indirizzo 127.0.0.1, porta 1883, SSL/TLS disattivato, come illustrato in Figura 6.2.

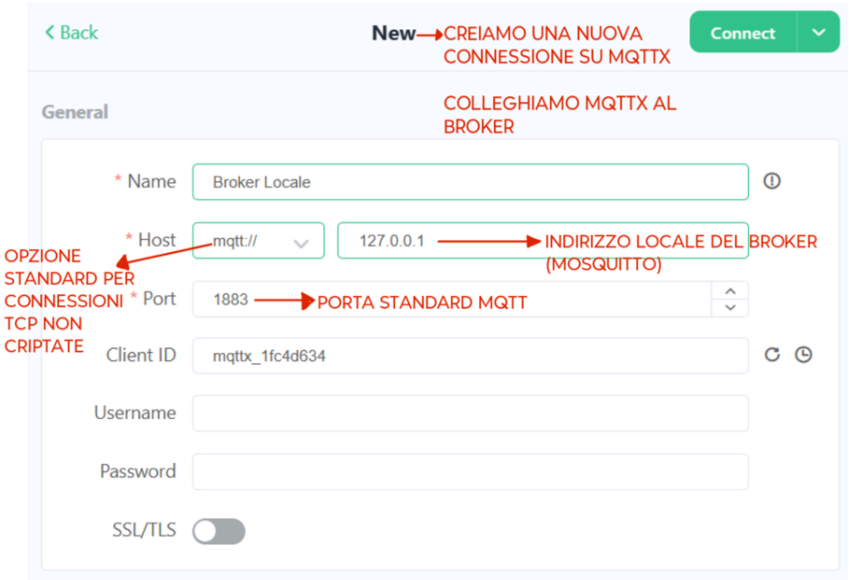


Figura 6.2.: Configurazione della connessione MQTTX al broker locale

Per verificare che il messaggio pubblicato transitasse effettivamente per il broker, il client MQTTX[22] è stato inoltre iscritto (subscribe) allo stesso topic sensori/comandi utilizzato per la pubblicazione, con QoS 1 (Figura 6.3): costituisce conferma diretta del corretto funzionamento del broker.

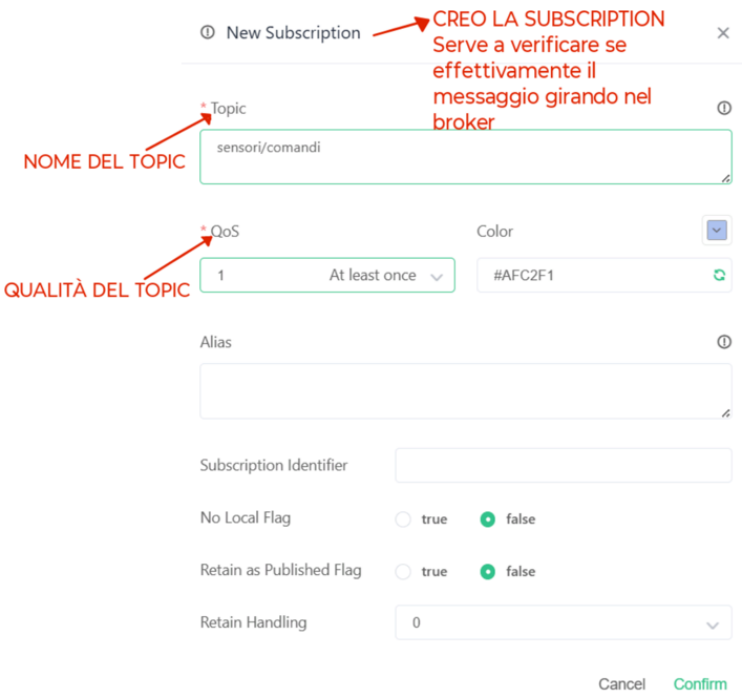


Figura 6.3.: Sottoscrizione del client MQTTX al topic sensori/comandi

6.1.2. Verifica della comunicazione MQTT

Il test funzionale del sistema è consistito nella pubblicazione, tramite MQTTX, di due messaggi distinti sul topic sensori/comandi, rispettivamente {"audio": 1} e {"audio": 2}, verificando che il plugin NuTech, in esecuzione su PC A, avviasse effettivamente la riproduzione della traccia audio corrispondente.

La Figura 6.4 mostra i due messaggi pubblicati, entrambi ricevuti dallo stesso client grazie alla sottoscrizione al medesimo topic, a conferma dell'avvenuto transito attraverso il broker.

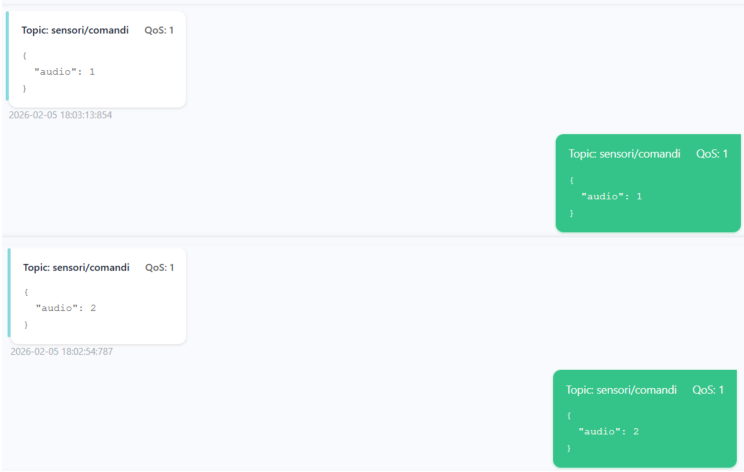


Figura 6.4.: Pubblicazione dei due messaggi di test

La Figura 6.5 riassume, in un'unica schermata annotata, l'intero setup di test lato MQTTX: la connessione al broker locale, la sottoscrizione di verifica al topic, e il pannello di pubblicazione con cui sono stati inviati i due comandi in formato JSON con QoS 1.

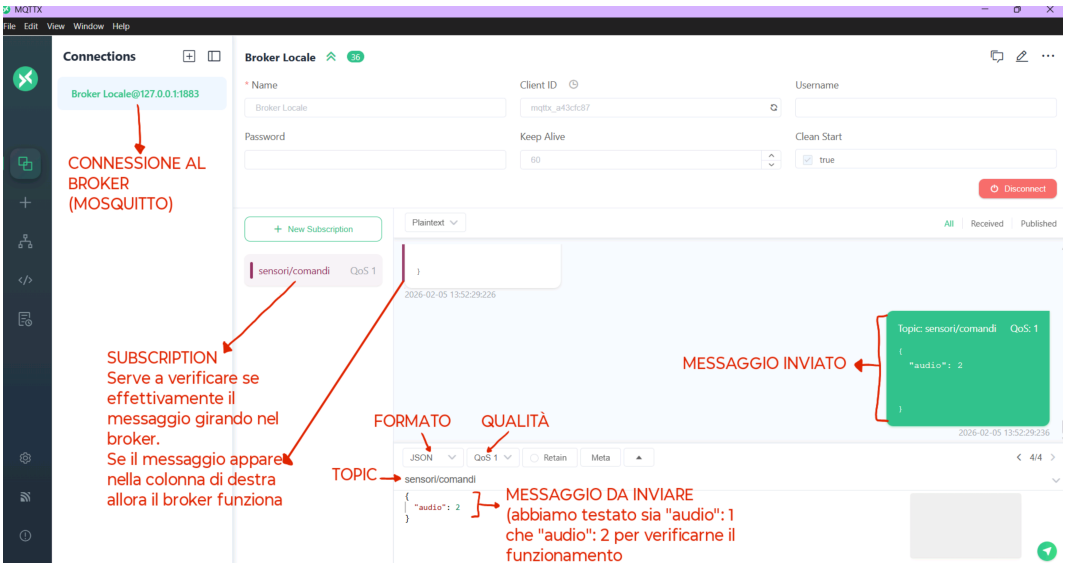


Figura 6.5.: Panoramica dell'interfaccia MQTTX utilizzata per i test

Il test, condotto in condizioni di laboratorio controllate (PC A e PC B sulla stessa rete Wi-Fi), ha confermato il corretto funzionamento dell'intera catena: pubblicazione da MQTTX, inoltro da parte del broker Mosquitto, ricezione da parte della callback MqttCb, aggiornamento della variabile pendingAudio e conseguente cambio di traccia riprodotta da LEPlugin_Process().

Verificata la comunicazione di base, si è passati a verificare il funzionamento della comunicazione estesa, la ricezione di un comando MQTT contenente traccia e azimuth, alla selezione della HRTF più vicina tra le sei precaricate.

Prima di procedere, è stata verificata l'infrastruttura di rete locale: il broker Mosquitto risultava in esecuzione (`sc query mosquitto` → `STATO: RUNNING`) e in ascolto sulla porta standard (`netstat -an | findstr 1883` → `0.0.0.0:1883 LISTENING`), e un test preliminare con i client a riga di comando `mosquitto_sub`/`mosquitto_pub` ha confermato che un messaggio pubblicato sul topic `sensori/comandi` veniva correttamente recapitato al sottoscrittore.

Con il plugin caricato, il log di debug ha confermato l'avvio corretto della componente MQTT e della cache HRTF:

```
[HRTF] Azimuth richiesto=270 -> usato=270 (idx=4)
[MQTT] Thread partito
[MQTT] Connesso al broker
[MQTT] Subscribe OK su sensori/comandi
[DEBUG] FrameSize = 4096
```

Impiegando MQTTX come client di test (connesso allo stesso broker locale su `127.0.0.1:1883`), sono stati pubblicati diversi messaggi JSON sul topic concordato. Il primo messaggio di verifica,

```
{"audio": 1, "azimuth": 0}
```

ha prodotto in uscita di debug:

```
MQTT msg: {"audio": 1, "azimuth": 0}
[HRTF] Azimuth richiesto=0 -> usato=0 (idx=0)
```

confermando la corretta ricezione del comando, il parsing del JSON e la selezione della direzione corrispondente in cache.

È stato inoltre verificato il comportamento del meccanismo di snap alla direzione più vicina descritto in Sezione 3.3 (Equazione 3.3), inviando un valore di azimuth non presente tra le sei direzioni di riferimento:

```
[HRTF] Azimuth richiesto=55 -> usato=70 (idx=1)
```

Il valore 55° viene correttamente mappato sulla direzione di riferimento più vicina (70°), risultato atteso dalla distanza angolare circolare calcolata nell'Equazione 3.2, essendo 55° più vicino a 70° che a 0°.

Infine, è stato verificato che comandi ravvicinati nel tempo vengano gestiti correttamente dal meccanismo basato sulla variabile atomica `pendingAudio`: se un nuovo comando arriva mentre la riproduzione precedente è ancora in corso, esso non viene riprodotto fino al termine della traccia corrente, come mostra la sequenza di log seguente, generata inviando quattro comandi (traccia/azimuth) a distanza ravvicinata:

```
MQTT msg: { "audio": 1, "azimuth": 90 }
[HRTF] Azimuth richiesto=90 -> usato=90 (idx=3)
[DEBUG] FrameSize = 4096
MQTT msg: { "audio": 2, "azimuth": 270 }
[HRTF] Azimuth richiesto=270 -> usato=270 (idx=4)
```

```
[DEBUG] FrameSize = 4096
MQTT msg: { "audio": 1, "azimuth": 270 }
[HRTF] Azimuth richiesto=270 -> usato=270 (idx=4)
[DEBUG] FrameSize = 4096
MQTT msg: { "audio": 2, "azimuth": 90 }
[HRTF] Azimuth richiesto=90 -> usato=90 (idx=3)
```

6.2. Validazione dell'algoritmo di spazializzazione

6.2.1. Verifica del filtraggio FIR

Filtro identità con impulso di Dirac Il primo test verifica che un filtro FIR i cui coefficienti sono un impulso di Dirac ($h[0] = 1$, $h[n] = 0$ altrove) si comporti come un filtro identità, cioè lasci il segnale di ingresso inalterato. È stato filtrato un vettore di test di 128 campioni, anch'esso un impulso di Dirac (`testInput[0] = 1.0`, tutto il resto a zero), e il risultato è stato verificato tramite file di log:

```
testOutput[0] = 1.000000
testOutput[1] = 0.000000
testOutput[2] = 0.000000
```

Il risultato conferma che filtrare un impulso con un filtro identità restituisce esattamente l'impulso di ingresso, come atteso dalla teoria: la convoluzione di un segnale con la delta restituisce il segnale stesso.

Filtro identità con rumore bianco Il secondo test sostituisce l'impulso di ingresso con un segnale di rumore bianco (256 campioni generati con `rand()`, normalizzati in $[-1, 1]$). Il confronto campione per campione tra ingresso e uscita, con filtro identità, ha prodotto:

```
[FIR WN] i=0 input=-0.997497 output=-0.997497 diff=0.000000
[FIR WN] i=1 input=0.127171 output=0.127171 diff=0.000000
[FIR WN] i=2 input=-0.613392 output=-0.613392 diff=0.000000
[FIR WN] i=3 input=0.617481 output=0.617481 diff=0.000000
[FIR WN] i=4 input=0.170019 output=0.170019 diff=0.000000
```

La differenza nulla per tutti i campioni conferma che il filtro identità non altera in alcun modo un segnale a banda larga come il rumore bianco.

Delta ritardata Per verificare che il filtro riproduca correttamente anche un comportamento non banale, i coefficienti sono stati modificati ponendo a 1.0 il decimo campione anziché il primo (`firCoeffs[9] = 1.0`), il che equivale, a un ritardo puro di 9 campioni: $y[n] = x[n - 9]$. Confrontando l'ingresso con l'uscita shiftata della quantità attesa:

```
[FIR WN] i=9  input=-0.997497 output=-0.997497 diff=0.000000
[FIR WN] i=10 input=0.127171 output=0.127171 diff=0.000000
```

```
[FIR WN] i=11 input=-0.613392 output=-0.613392 diff=0.000000
[FIR WN] i=12 input=0.617481 output=0.617481 diff=0.000000
[FIR WN] i=13 input=0.170019 output=0.170019 diff=0.000000
```

la differenza è stabilmente nulla, confermando che l'uscita corrisponde all'ingresso traslato esattamente di 9 posizioni, in accordo con la posizione dell'impulso nei coefficienti del filtro.

Continuità della delay line tra blocchi consecutivi Un aspetto critico per un filtro che opera in streaming a blocchi, già discusso dal punto di vista teorico in Sezione 5.9.1, è che la delay line venga mantenuta correttamente da una chiamata di `ippsFIRSR_64f` alla successiva, evitando discontinuità ai bordi dei blocchi. Per verificarlo, è stato filtrato, con filtro identità, un segnale progressivo (campioni numerati $0, 1, 2, \ldots$) suddiviso in due blocchi consecutivi da 256 campioni, senza reinizializzare lo stato fra i due:

```
for (int i = 0; i < 256; i++) block1In[i] = (double)i;
for (int i = 0; i < 256; i++) block2In[i] = (double)(i + 256);

ippsFIRSR_64f(block1In, block1Out, 256, firState, nullptr, nullptr, firBuf);

ippsFIRSR_64f(block2In, block2Out, 256, firState, nullptr, nullptr, firBuf);
```

Il risultato,

```
[CONT] block1Out[255]=255.000000 block2Out[0]=256.000000 atteso_b2
   =256.000000
```

mostra che l'ultimo campione del primo blocco (255) e il primo campione del secondo blocco (256) si susseguono senza soluzione di continuità, esattamente come nel segnale di ingresso originale. Questo conferma che IPP gestisce correttamente lo stato interno del filtro tra chiamate successive, requisito indispensabile per l'elaborazione audio in tempo reale a blocchi.

Verifica su tutti i campioni tramite confronto con MATLAB I test precedenti verificano solo i primi campioni di un segnale sintetico breve. Per una verifica esaustiva è stato generato in MATLAB un file WAV di rumore bianco di durata nota (tre secondi, $N_1 = 132300$ campioni a 44.1 kHz):

```
fs = 44100; durata = 3;
rumore = randn(fs*durata, 1);
rumore = rumore / max(abs(rumore)) * 0.99;
audiowrite('rumore_bianco_3s.wav', rumore, fs);
```

Il plugin, con `modalitaFileRead = true` e filtro identità (`firCoeffs[0]=1.0`), ha elaborato l'intero file a blocchi di 512 campioni e salvato l'uscita filtrata su un secondo file WAV. Il confronto campione per campione tra i due file, condotto in MATLAB su tutti i 132300 campioni, ha prodotto:

```
Differenza massima: 1.3878e-16
Differenza media: 3.3583e-21
I file sono identici (differenza < 1e-6)
```

La differenza massima, dell'ordine di 10^{-16} è trascurabile: il filtraggio FIR con IPP non introduce alcuna alterazione sistematica del segnale, confermando su scala completa quanto già osservato sui primi campioni.

Verifica tramite gli strumenti nativi di NuTech (FileWrite/Mix) Come passo successivo, la stessa verifica è stata ripetuta impiegando esclusivamente gli strumenti nativi di NuTech, senza dipendere da un file scritto su disco tramite dr_wav. Il nuts è stato collegato in ingresso a un blocco `FileRead` (configurato sul file di rumore bianco) e in uscita a un blocco `FileWrite`, entrambi con un `Viewer` in parallelo per l'ispezione visiva del segnale (Figura 6.6).

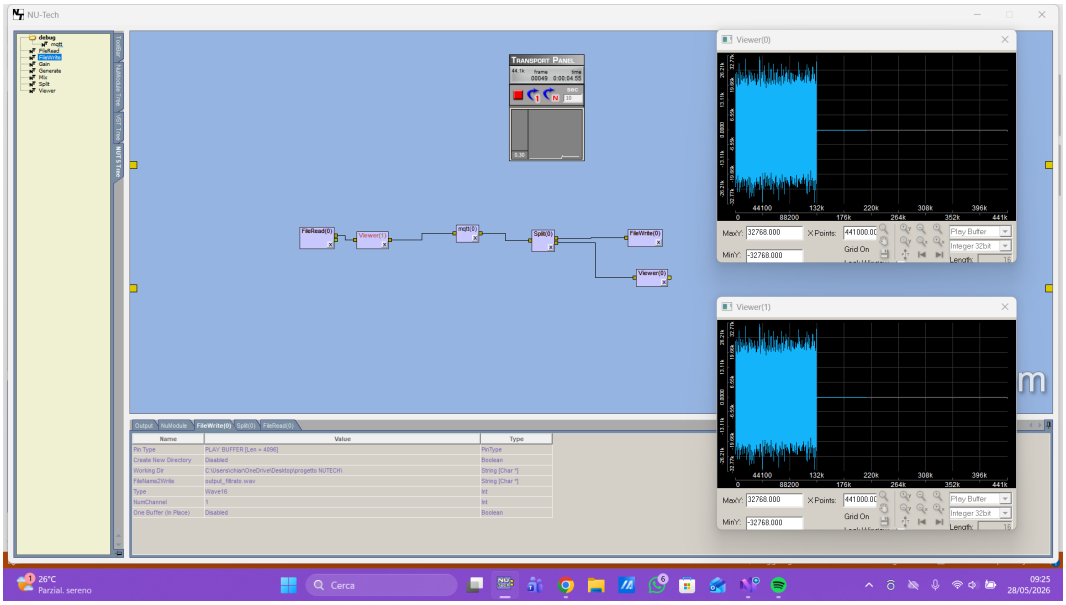


Figura 6.6.: Confronto visivo, tramite Viewer di NuTech, tra il segnale di ingresso (FileRead) e il segnale in uscita dal nuts (FileWrite), con filtro identità.

Questo primo esperimento ha richiesto una modifica: il ramo di elaborazione MQTT, in condizioni normali, scrive `OutputData` solo quando arriva un comando (`pendingAudio != 0`); poiché in questo test non veniva inviato alcun messaggio MQTT, l'uscita restava silenziosa. È stato quindi necessario introdurre la modalità dedicata `modalitaFileRead` (Sezione 5.13), che bypassa il meccanismo MQTT e legge direttamente da `InputData`.

Il confronto quantitativo in MATLAB tra il file di ingresso e il file scritto da `FileWrite` ha richiesto un ulteriore accorgimento: poiché NuTech continua a elaborare blocchi (riempiendoli di zero) anche dopo l'esaurimento del file letto da `FileRead`, finché non si interrompe manualmente la riproduzione, il file di uscita risulta più

lungo di quello di ingresso (184320 campioni contro 132300). Il confronto è stato quindi condotto troncando entrambi i segnali alla lunghezza minima comune:

```
N = min(length(ingresso), length(uscita));
ingresso = ingresso(1:N);
uscita  = uscita(1:N);
```

Con questa correzione, il caso a filtro identità ha prodotto una differenza sostanzialmente nulla salvo un breve transitorio finale (attribuibile alla coda del filtro FIR in prossimità della fine del segnale), mentre nel caso a delta ritardata (`firCoeffs[9]=1.0`) è stato necessario confrontare l'ingresso con l'uscita shiftata di 9 posizioni:

```
diff_corretta = ingresso(1:end-9) - uscita(10:end);
```

L'analisi statistica della differenza residua ha mostrato:

```
Campioni con diff > 0.01 con shift=0: 129000 su 132291
Campioni con diff > 0.01 con shift=9: 1455 su 132291
  di cui negli ultimi 1500 campioni (transitorio finale): 1182
  nel resto del segnale: 273 (0.2% del totale)
Distanza media tra campioni sbagliati: 88 campioni
```

Il fatto che il confronto con shift nullo produca un numero di campioni errati prossimo alla totalità del segnale, mentre lo shift di 9 lo riduca a poche centinaia concentrate quasi per intero nel transitorio finale, conferma che il ritardo di 9 campioni introdotto dalla delta è quello corretto e uniforme lungo tutto il segnale. I rimanenti 273 campioni (0.2% del totale), distribuiti con una periodicità media di circa 88 campioni, sono compatibili con un errore di quantizzazione a 16 bit introdotto ai bordi di ciascun blocco elaborato da NuTech.

Un secondo tipo di verifica, più diretto, è stato condotto instradando l'ingresso e l'uscita del nuts in un blocco `Mix` di NuTech, configurato con guadagno $+1$ su un canale e -1 sull'altro, in modo da calcolare direttamente ingresso $-$ uscita e osservarne l'andamento sul Viewer in tempo reale. Con filtro identità il Viewer mostra una linea piatta a zero (Figura 6.7).

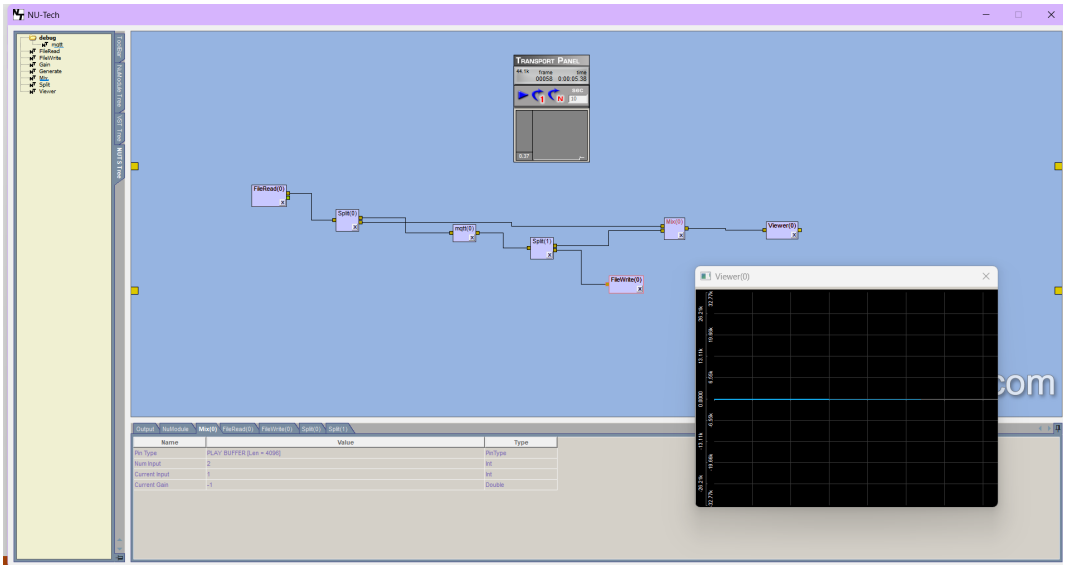


Figura 6.7.: Differenza ingresso–uscita (blocco Mix) con filtro identità: linea piatta a zero.

Con la delta ritardata, poiché i due segnali non sono allineati temporalmente, la differenza non è nulla ma mostra un residuo strutturato (Figura 6.8), come atteso.

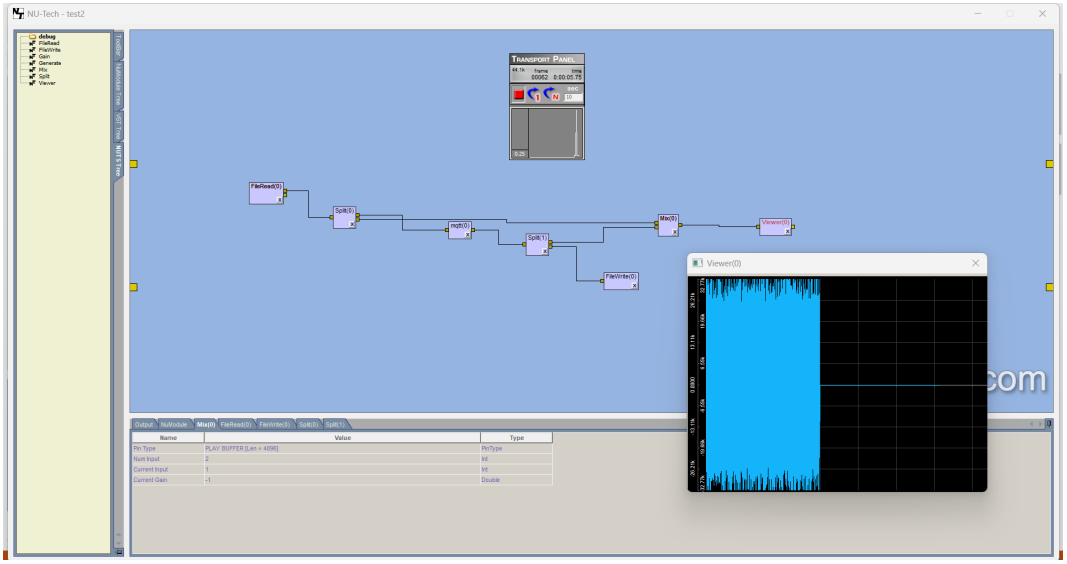


Figura 6.8.: Differenza ingresso–uscita (blocco Mix) con delta ritardata di 9 campioni: segnale residuo dovuto al disallineamento temporale.

I test descritti in questa sezione, nel loro complesso, hanno permesso di verificare in modo indipendente e con più metodi (log di debug, confronto quantitativo in MATLAB, confronto visivo tramite gli strumenti nativi di NuTech) che il pattern GetSize/Init/Process di Intel IPP applicato al filtro FIR si comporta esattamente come previsto, fornendo così una base solida su cui innestare, nella sezione successiva, il filtraggio con le HRTF reali.

Estensione a due canali paralleli

Una volta verificato il filtro a canale singolo, il sistema è stato esteso alla configurazione a due uscite (`firStateL`, `firStateR`, due filtraggi FIR indipendenti sullo stesso ingresso mono). La correttezza di questa estensione è stata verificata con lo stesso approccio del blocco Mix descritto sopra, questa volta confrontando tra loro le due uscite del nuts (Output 0 e Output 1) anziché ingresso e uscita.

Impostando entrambi i canali con lo stesso filtro identità (`firCoeffsL[0]=firCoeffsR[0]=1.0`), la differenza tra le due uscite è risultata una linea piatta a zero, confermando che i due filtraggi paralleli, se alimentati con gli stessi coefficienti, producono risultati identici (Figura 6.9).

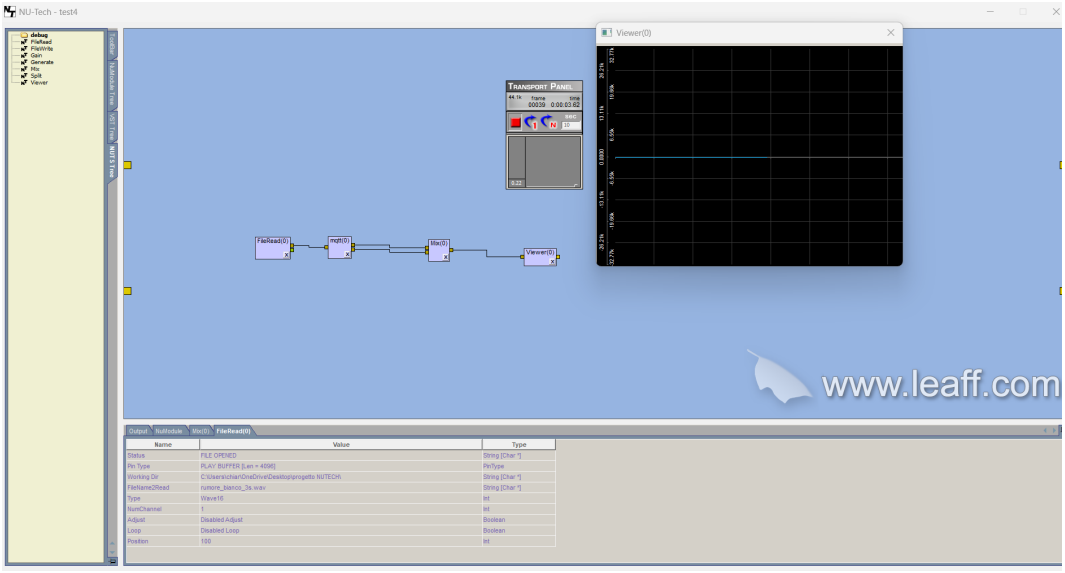


Figura 6.9.: Differenza tra Output 0 e Output 1 con firCoeffsL[0]=1.0 e firCoeffsR[0]=1.0: canali identici, differenza nulla.

Impostando invece coefficienti diversi (`firCoeffsL[0]=1.0`, `firCoeffsR[9]=1.0`), la differenza è diventata un segnale non nullo e chiaramente strutturato (Figura 6.10), a conferma che i due stati del filtro (`firStateL`, `firStateR`) restano effettivamente indipendenti e applicano trasformazioni distinte allo stesso segnale di ingresso.

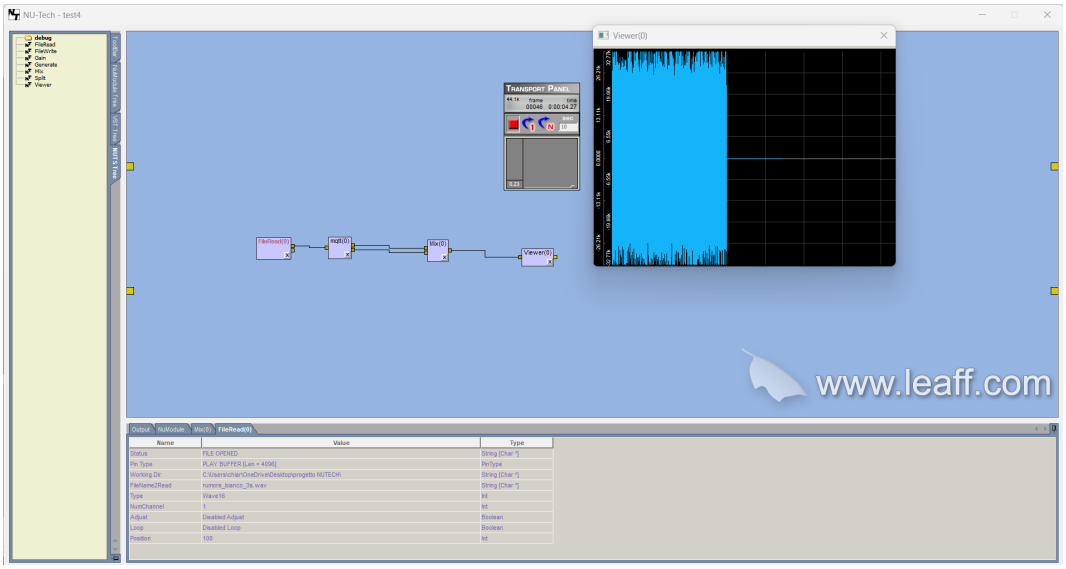


Figura 6.10.: Differenza tra Output 0 e Output 1 con firCoeffsL[0]=1.0 e firCoeffsR[9]=1.0: canali distinti, differenza non nulla.

6.2.2. Verifica del filtraggio con le HRTF a due canali

Sostituiti i coefficienti sintetici con le risposte impulsive reali del database KEMAR tramite `ApplyHRTF` (Sezione 5.9), una prima verifica ha riguardato la plausibilità dei valori caricati. Con azimuth 90° (sorgente sonora concettualmente a sinistra), i primi coefficienti dei due canali risultano dello stesso ordine di grandezza ma non identici:

```
Azimuth=90
firCoeffsL[0..9]: -0.000061, -0.000031, 0.000000, 0.000031, ...
firCoeffsR[0..9]: -0.000153, -0.000214, -0.000122, -0.000031, ...
```

Esaminando i coefficienti più avanti nella risposta impulsiva (indici 20–30), la differenza tra i due canali diventa netta:

```
firCoeffsL[20..30]: valori quasi nulli (~0.00003)
firCoeffsR[20..30]: picco fino a 0.214996 (indice 29)
```

Questo comportamento è fisicamente coerente con la teoria richiamata nel Capitolo 2: con la sorgente sonora a 90° (a sinistra dell'ascoltatore), l'orecchio sinistro riceve il suono per via diretta, con una risposta più semplice, mentre l'orecchio destro, il più lontano, la cui HRTF deve tenere conto dell'ombra acustica della testa e del percorso più lungo, presenta una risposta più complessa e con energia significativa anche in campioni successivi. Il caricamento dei log (`channels=1, sampleRate=44100, totalFrames=512`) ha inoltre confermato che i file venivano letti correttamente nel formato atteso.

6.3. Test sperimentali

6.3.1. Dalla difficoltà di percezione alla spazializzazione funzionante

Nonostante la correttezza confermata a livello di segnale (Sezione 6.2.2), l'ascolto in cuffia del segnale filtrato non permetteva di percepire alcuna direzionalità. Questo disallineamento tra la correttezza verificata a livello di segnale e l'assenza di un effetto percettivo ha richiesto una serie di verifiche incrociate per isolarne la causa, qui riportate nell'ordine in cui sono state condotte, e costituisce un test del sistema completo[1] poiché coinvolge l'intera catena audio fino all'ascolto.

Guadagno e normalizzazione dei coefficienti. Poiché il segnale filtrato con le HRTF reali risultava di ampiezza molto ridotta, un primo tentativo ha previsto l'applicazione di un guadagno diretto sui coefficienti, che tuttavia introduceva distorsione udibile. È stata quindi provata una normalizzazione indipendente[2] dei due canali, ma questo approccio ha mostrato un effetto collaterale rilevante, le forme d'onda risultavano più bilanciate in ampiezza pur mantenendo una struttura temporale identica, come atteso da due filtraggi dello stesso segnale mono con HRTF diverse. In nessuno dei due casi, tuttavia, la direzionalità diventava percepibile in ascolto, il che ha permesso di escludere la normalizzazione dei coefficienti come causa del problema.

Dai test abbiamo pertanto rimosso tale normalizzazione, ma l'abbiamo sostituita con un guadagno fisso pari a 2^{15} descritto in Sezione 5.11 e in caso di necessità da un blocco `Gain` esterno al plugin per portare il segnale a un volume udibile.

Verifica indipendente dei due canali in NuTech. Osservando le due uscite del nuts tramite due `Viewer` separati, le forme d'onda dei due canali risultavano visibilmente diverse in ampiezza pur mantenendo la stessa struttura temporale (Figura 6.11), un'ulteriore conferma che il filtraggio a livello di segnale era corretto.

[1] non unicamente del filtraggio

[2] cioè ciascuno rispetto al proprio massimo. Facendo una normalizzazione comune si elimina la differenza fra i due canali e quindi anche la spazializzazione

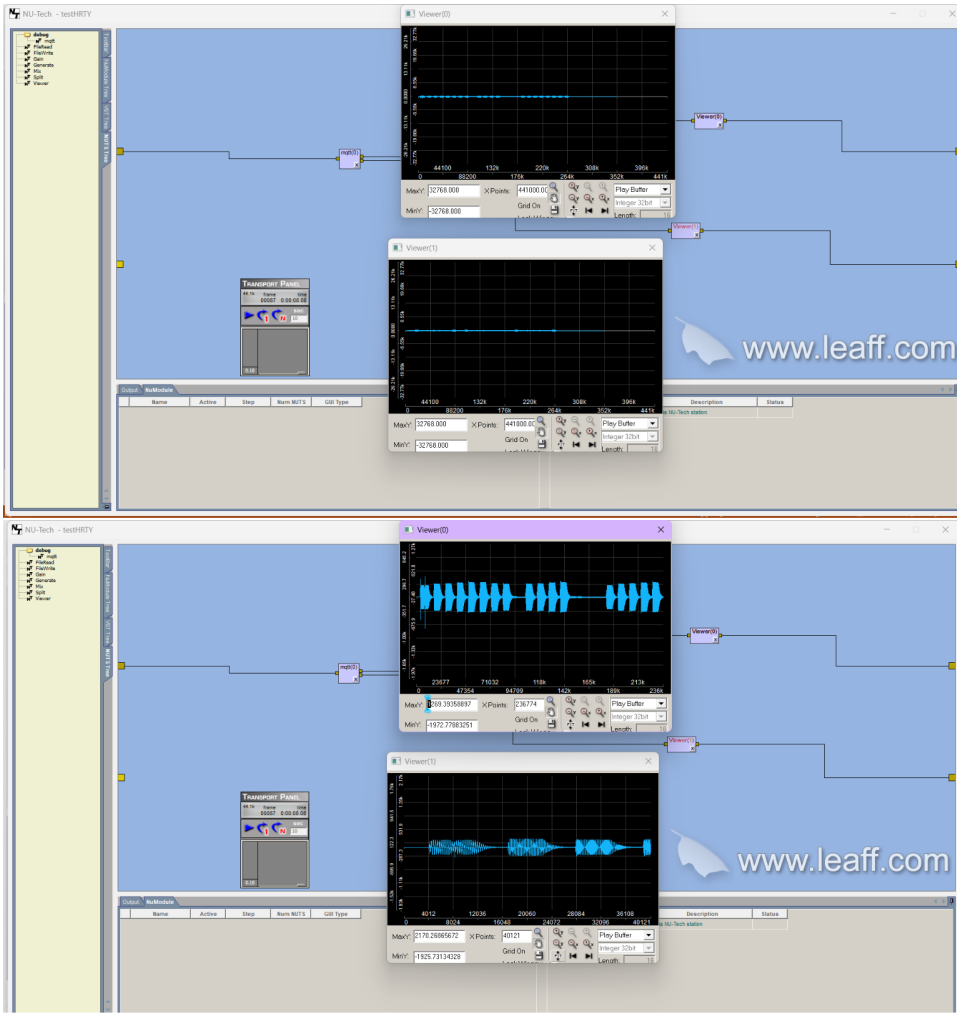


Figura 6.11.: Uscite Output 0 (L) e Output 1 (R) con HRTF a 90°: stessa struttura temporale del rumore, ampiezze diverse (particolare ingrandito a destra).

Analisi spettrale in MATLAB su file salvati separatamente. Le due uscite del plugin sono state salvate su due file WAV distinti tramite `FileWrite` e confrontate in MATLAB, calcolandone forma d'onda, spettro in frequenza e differenza temporale:

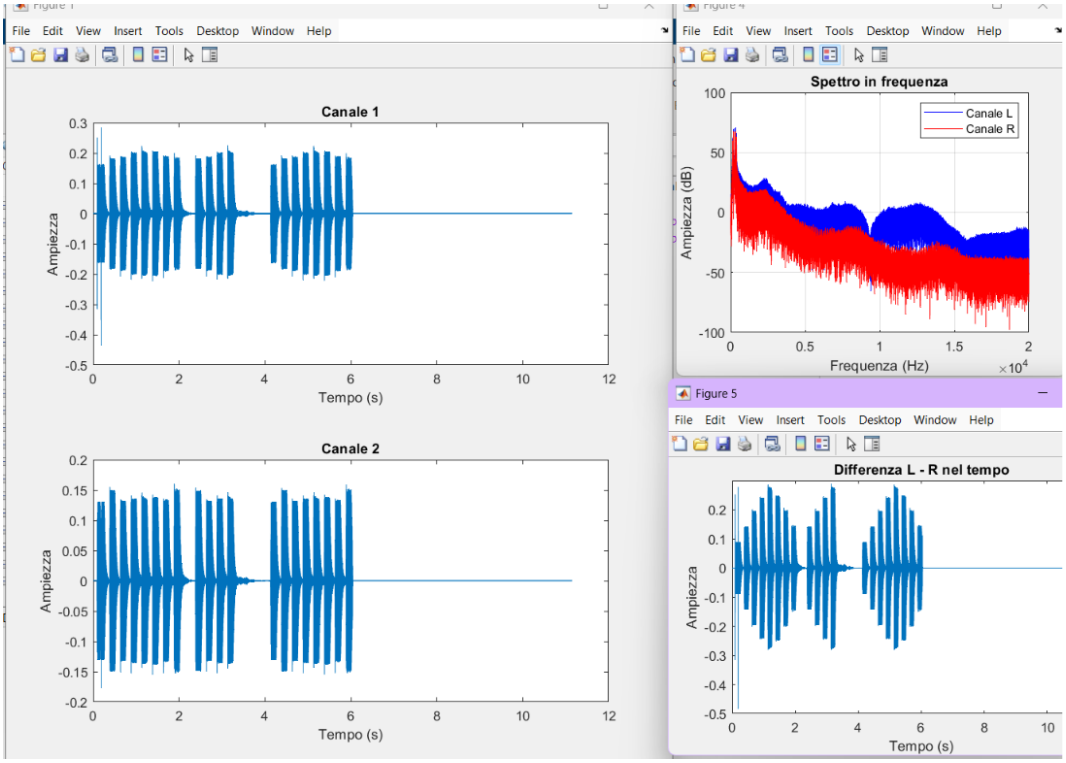


Figura 6.12.: Confronto in MATLAB tra i due canali salvati separatamente: forme d'onda (ampiezze diverse), spettri in frequenza (colorazione diversa tra i due canali), differenza temporale non nulla.

Il confronto ha mostrato che il canale L presenta un'ampiezza maggiore su tutte le frequenze rispetto al canale R, il quale risulta più attenuato specialmente alle alte frequenze, comportamento coerente con la teoria, e che la differenza temporale tra i due segnali non è piatta a zero. Questi tre indizi confermavano concordemente che la spazializzazione era corretta a livello di segnale, restringendo ulteriormente il campo delle possibili cause della mancata percezione in ascolto.

Verifica indipendente con MATLAB e ascolto tramite mixaggio. Quindi in laboratorio abbiamo condotto un confronto diretto tra l'uscita del filtraggio in MATLAB e i valori del filtro ispezionati tramite breakpoint in Visual Studio: i coefficienti e i campioni filtrati risultavano identici, a meno dell'offset di indicizzazione tra MATLAB[3] e C++[4]. Successivamente, i due file WAV salvati separatamente (`canale_1.wav`, `canale_2.wav`), che ascoltati singolarmente non presentavano alcuna direzionalità percepibile, sono stati combinati in un unico segnale stereo direttamente in MATLAB:

```
[x1,fs] = audioread('canale_1.wav');
[x2,fs] = audioread('canale_2.wav');
```

[3]che parte da 1
[4]che parte da 0

```
xx = [x1 x2];
sound(xx,fs)
```

L'ascolto di questo segnale stereo ricombinato ha permesso, per la prima volta, di percepire chiaramente la direzionalità del suono. Questo risultato ha confermato in modo definitiva che il problema non risiedeva nel filtraggio né nel caricamento delle HRTF.

Causa del problema: instradamento e conversione dei livelli audio. L'indicazione decisiva è stata di moltiplicare l'uscita del filtro per il fattore $2^{15} = 32768$, corrispondente alla normalizzazione tipica di una scheda audio a 16 bit: i coefficienti HRTF, letti tramite dr_wav, risultano infatti normalizzati nell'intervallo $[-1, +1]$, mentre la catena audio a valle si aspetta segnali a 16 bit. Applicando questo fattore sia sul primo blocco sia sull'ultimo, il segnale è diventato udibile a un volume adeguato senza distorsione.

Il setup di ascolto impiegato per questa verifica (Figura 6.13) ha reso necessario salvare l'uscita del plugin su file tramite `FileWrite` e riprodurla successivamente con il lettore multimediale del sistema operativo o tramite `FileRead` da Nutech. Se si collegava direttamente i pin di uscita del nuts alla scheda audio del computer, le cuffie Bluetooth impiegate per l'ascolto non permettevano di percepire alcuna direzionalità, pur funzionando correttamente con la riproduzione tramite file.

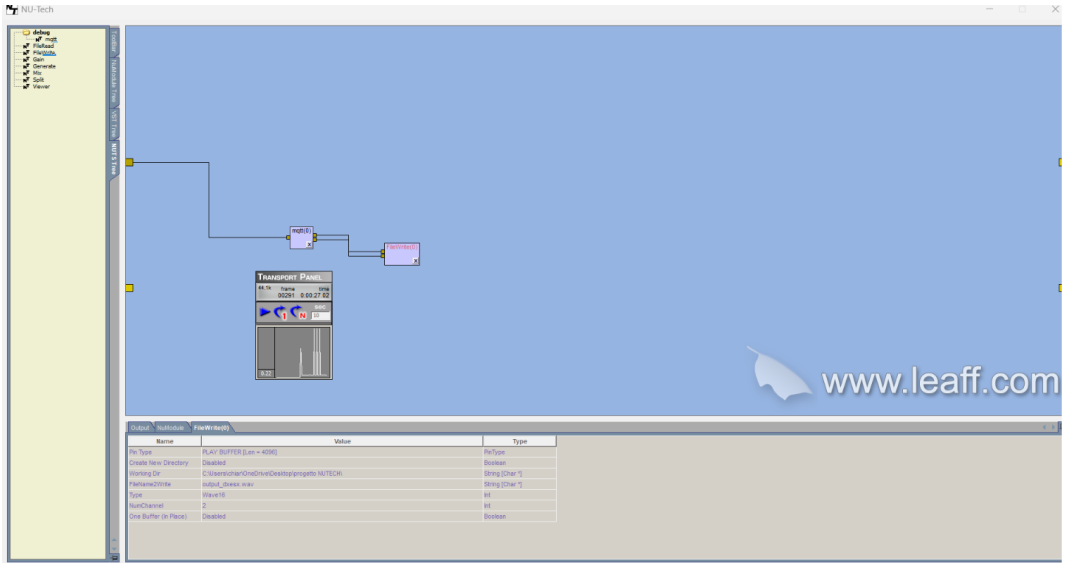


Figura 6.13.: Configurazione di NuTech impiegata per l'ascolto con cuffie Bluetooth: uscita del plugin salvata tramite FileWrite.

Una volta individuata questa causa, i test successivi sono stati condotti collegando direttamente i pin di uscita del plugin alla scheda audio, impiegando cuffie cablate anziché Bluetooth, senza necessità del passaggio intermedio tramite file.

6.3.2. Test di percezione della direzione: validazione con il simulatore di guida

Revisione dell'insieme di direzioni di riferimento

I test descritti nelle sezioni precedenti hanno validato la correttezza del filtraggio HRTF a livello di segnale e la funzionalità end-to-end del comando MQTT, ma non ancora la capacità del sistema di trasmettere un'informazione direzionale effettivamente distinguibile da un ascoltatore. A questo scopo, in una sessione di laboratorio, l'insieme delle cinque direzioni di riferimento $\{0°, 45°, 90°, 270°, 315°\}$, è stato sottoposto a una serie di ascolti comparativi tramite MQTTX, dai quali è emerso che non costituiva la scelta più efficace ai fini applicativi.

In particolare, le direzioni laterali a 90° e 270° risultavano poco distinguibili dalle direzioni diagonali immediatamente più prossime. L'insieme di direzioni è stato quindi rivisto, sostituendo le cinque direzioni originarie con sei nuove direzioni, scelte tramite un processo iterativo di ascolto comparativo per massimizzare la distinguibilità percettiva reciproca, pur mantenendo una copertura ragionevolmente uniforme attorno all'abitacolo:

- 0°: anteriore centrale;
- 70°: anteriore destra;
- 160°: posteriore destra;
- 180°: posteriore centrale;
- 200°: posteriore sinistra;
- 290°: anteriore sinistra.

Questa revisione ha comportato l'aggiornamento della cache di direzioni descritta in Sezione 5.9:

```
static const int NUM_AZIMUTHS = 6;
int azimuthSet[NUM_AZIMUTHS] = { 0, 200, 160, 70, 290, 180 };
```

La formula di selezione della direzione più vicina (Equazione 3.3) resta invariata: l'aumento a sei direzioni non richiede alcuna modifica alla logica di `ApplyHRTF`.

Generazione degli avvisi audio di sicurezza

Per avvicinare il sistema al caso d'uso finale, un avviso di sicurezza spazializzato[5], sono stati generati, tramite un sintetizzatore vocale, due file audio corrispondenti a due scenari di guasto tipici in ambito automotive:

[5] anziché un segnale di test come il rumore bianco impiegato nelle sezioni precedenti

- il primo, associato al comando `{audio: 1}`, corrisponde all'avviso

  "ATTENZIONE: guasto gomma!";

- il secondo, associato al comando `{audio: 2}`, corrisponde all'avviso

  "ATTENZIONE: anomalia al sistema motore rilevata".

I due file hanno sostituito i segnali di rumore bianco e il brano musicale impiegati nella fase di verifica del filtro (Sezione 6.2.1), in un'ulteriore evoluzione delle costanti N_1 e N_2 rispetto a quanto già discusso in quella sede: dopo aver verificato in MATLAB la lunghezza esatta dei due nuovi file,

$$N_1 = 102312, \qquad N_2 = 178164 \tag{6.1}$$

I due file, generati dal sintetizzatore vocale a una frequenza di campionamento diversa da quella impiegata nel resto del sistema, sono stati inoltre ricampionati in MATLAB a 44.1 kHz, per coerenza con la frequenza di campionamento configurata nel resto della pipeline audio.

Configurazione del test: simulatore di guida e interfaccia audio

Il test è stato condotto collegando l'uscita audio del simulatore di guida del laboratorio e l'uscita del PC che ospita il plugin NuTech a un'interfaccia audio esterna (Focusrite Scarlett 18i20, seconda generazione), che sommava i due segnali e li inviava a un unico paio di cuffie indossato dal soggetto alla guida (Figura 6.14).

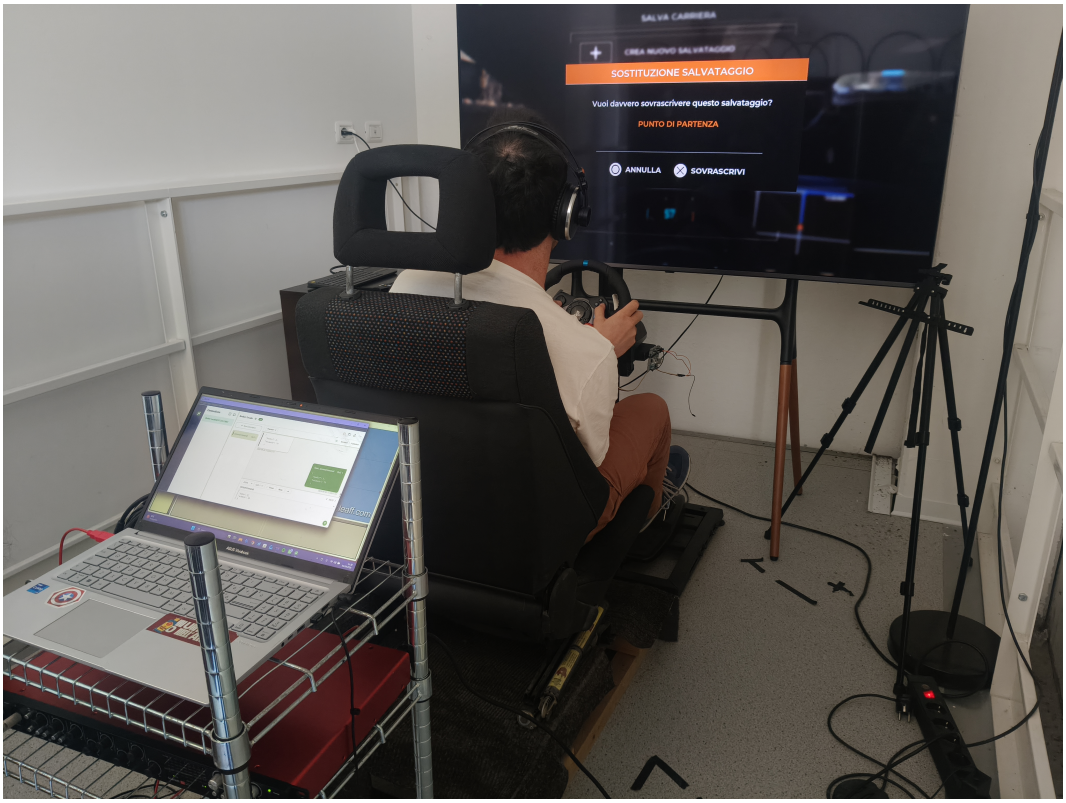


Figura 6.14.: Configurazione del test al simulatore

In questo modo, il soggetto percepiva contemporaneamente l'audio di gioco del simulatore e gli avvisi di sicurezza spazializzati inviati dal plugin, in una condizione più vicina a un reale scenario di guida rispetto agli ascolti isolati delle sezioni precedenti.

Il setup finale di Nutech, utilizzato nei test con il simulatore, è mostrato in Figura 6.15

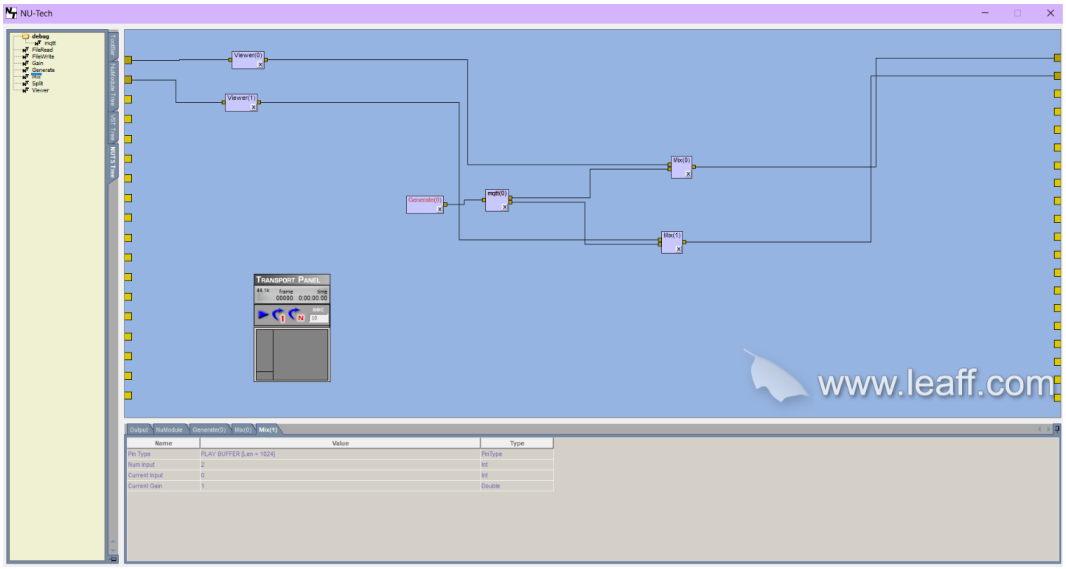


Figura 6.15.: Setup Nutech per i test finali

Metodologia

Il test è stato condotto su tre soggetti. Per ciascun soggetto, mentre era impegnato nella guida del simulatore, sono stati inviati dieci avvisi di sicurezza in sequenza, ciascuno con una direzione (azimuth) scelta tra le sei di riferimento; dopo ogni avviso, al soggetto è stato chiesto di indicare verbalmente la direzione percepita di provenienza del suono, registrata e confrontata con la direzione realmente inviata via MQTT.

Deliberatamente, gli avvisi impiegati nel test non specificavano il tipo di guasto insieme alla direzione[6]: l'obiettivo era isolare la sola capacità del soggetto di localizzare la provenienza spaziale del suono, indipendentemente da qualunque indizio semantico aggiuntivo che un messaggio più specifico avrebbe potuto fornire. L'eventuale combinazione tra contenuto semantico e posizione spaziale, il cui beneficio congiunto è discusso in letteratura (Capitolo 2), resta un possibile sviluppo futuro.

Risultati

Le Tabelle 6.1–6.3 riportano, per ciascun soggetto, la direzione realmente inviata e quella stimata dal soggetto per ciascuna delle dieci prove.

[6]ad esempio non è stato usato un messaggio come "guasto gomma anteriore destra"

Tabella 6.1.: Soggetto 1 : direzione reale vs. stimata (gradi).

Prova	Reale	Stimata	Esito
1	70	70	corretto
2	180	180	corretto
3	290	200	errore
4	160	160	corretto
5	0	180	errore
6	70	70	corretto
7	200	200	corretto
8	160	160	corretto
9	290	200	errore
10	0	0	corretto

Tabella 6.2.: Soggetto 2 : direzione reale vs. stimata (gradi).

Prova	Reale	Stimata	Esito
1	160	70	errore
2	200	290	errore
3	0	0	corretto
4	180	180	corretto
5	70	160	errore
6	200	290	errore
7	290	290	corretto
8	70	70	corretto
9	160	160	corretto
10	180	0	errore

Tabella 6.3.: Soggetto 3 : direzione reale vs. stimata (gradi).

Prova	Reale	Stimata	Esito
1	70	70	corretto
2	160	180	errore
3	200	200	corretto
4	0	0	corretto
5	290	290	corretto
6	180	0	errore
7	160	0	errore
8	70	70	corretto
9	200	200	corretto
10	180	180	corretto

I tre soggetti hanno ottenuto rispettivamente 7, 5 e 7 risposte corrette su 10, per un totale complessivo di 19 risposte corrette su 30 prove, pari a un'accuratezza media del 63,3%.

Analisi degli errori

Un'analisi qualitativa degli 11 errori complessivi, riportata in Tabella 6.4, permette di individuare un pattern ricorrente.

Tabella 6.4.: Classificazione degli errori commessi nei tre test.

Tipo di errore	**Descrizione**
Confusione anteriore/posteriore	Il lato (destra/sinistra/centro) viene identificato correttamente, ma la direzione stimata si trova sull'asse opposto anteriore-posteriore rispetto a quella reale (es. $70° \rightarrow 160°$, $0° \rightarrow 180°$, $290° \rightarrow 200°$). Questa categoria comprende 8 dei 11 errori totali.
Confusione laterale/centrale	Una direzione posteriore laterale (160° o 200°) viene confusa con la direzione posteriore centrale (180°), o viceversa, oppure in un caso con la direzione anteriore centrale (0°). Questa categoria comprende i restanti 3 errori.

Un risultato particolarmente rilevante, non immediatamente evidente dalla sola percentuale di accuratezza complessiva, è che **in nessuno degli** 11 **errori il soggetto ha confuso il lato destro con il lato sinistro**: ogni errore, quando riconducibile a una singola direzione laterale, ha mantenuto correttamente l'emisfe-

ro destro/sinistro, sbagliando esclusivamente sull'asse anteriore/posteriore o sulla distinzione laterale/centrale all'interno dello stesso emisfero.

Questo risultato è coerente con la teoria della localizzazione uditiva richiamata nel Capitolo 2: esattamente il fenomeno di *confusione fronte/retro* già discusso in Sezione 2.1.3 e osservato sperimentalmente anche nello studio Visteon richiamato in Sezione 2.4.

Va inoltre osservato che il campione impiegato (tre soggetti, dieci prove ciascuno) è ridotto: i risultati riportati hanno quindi valore indicativo e qualitativo, utile a evidenziare un pattern di errore coerente con le conoscenze teoriche, ma non consentono conclusioni statisticamente robuste sulla percentuale di riconoscimento del sistema in condizioni di guida reale.

Implicazioni per l'uso come sistema di allerta

La sistematicità della confusione fronte/retro, unita alla sua assenza sull'asse laterale, ha un'implicazione pratica diretta per l'impiego del sistema come strumento di allerta per la sicurezza: se un avviso è generico[7], l'eventuale confusione tra 0° e 180° ha un impatto limitato, poiché entrambe le direzioni sono comunque codificate come "centrali" e non richiedono una discriminazione laterale ai fini della sicurezza.

Al contrario, per un'applicazione in cui la direzione laterale del pericolo è l'informazione critica[8], coerentemente con lo scenario discusso in Sezione 2.4, il sistema ha mostrato un'affidabilità del 100% nel riconoscimento del lato corretto sull'intero campione di test, un risultato che ne supporta l'impiego per questa classe di avvisi.

[7] ad esempio un avviso di anomalia del veicolo che non richiede una reazione immediata e direzionata
[8] ad esempio un veicolo in avvicinamento da un lato

Capitolo 7.

Conclusioni e sviluppi futuri

7.1. Conclusioni

Questo lavoro di tesi è partito da un sistema di comunicazione in grado di comandare da remoto, tramite protocollo MQTT, la riproduzione di tracce audio in un plugin NuTech, capace di decidere soltanto *quale* suono riprodurre. L'obiettivo era estendere questo sistema affinché fosse in grado di riprodurre segnali audio spazializzati tramite tecniche HRTF, da utilizzare come strumento di allerta per la sicurezza durante la guida: non più soltanto quale suono, ma anche da dove, nello spazio tridimensionale attorno al guidatore, quel suono debba provenire.

Il lavoro ha richiesto, in primo luogo, di inquadrare il problema nella letteratura relativa all'audio 3D e agli avvisi acustici direzionali in ambito automotive (Capitolo 2), da cui è emerso un fondamento sperimentale a sostegno dell'uso di segnali spazializzati per la sicurezza, insieme ai limiti noti delle HRTF non individualizzate, in particolare la confusione fronte/retro, che sarebbe poi riemersa nei test sperimentali condotti nell'ultima parte del lavoro.

Su queste basi è stata progettata l'estensione del sistema di comunicazione (Capitolo 3), realizzata come un nuovo stadio di filtraggio binaurale a due canali paralleli, basato su convoluzione FIR tramite la libreria Intel IPP. Il comando MQTT è stato esteso con un campo di azimuth per veicolare la direzione richiesta, la quale viene mappata, tramite una regola di distanza angolare circolare, su un insieme ristretto di direzioni di riferimento; le rispettive coppie di HRTF, tratte dal database KEMAR del MIT Media Lab, vengono precaricate in memoria all'avvio del plugin, in modo da rispettare il vincolo di elaborazione in tempo reale già imposto dall'architettura multithread del sistema di base (Capitoli 4 e 5).

I test condotti (Capitolo 6) hanno confermato, in una prima fase di validazione, la correttezza matematica del filtraggio FIR realizzato tramite Intel IPP in ogni sua componente, e la correttezza dell'estensione a due canali paralleli con stati indipendenti, verificata sia nel caso di coefficienti identici sia nel caso di coefficienti diversi tra i due canali. Le HRTF del database KEMAR sono risultate caricate correttamente, producendo a livello di segnale differenze tra i due canali coerenti con la fisica dell'ascolto binaurale descritta nel Capitolo 2; la comunicazione MQTT estesa con il campo di azimuth è stata verificata, incluso il meccanismo di snap alla

direzione più vicina e la corretta gestione di comandi ravvicinati nel tempo. Ciascuno di questi aspetti è stato verificato in isolamento dalle altre componenti del sistema.

In una seconda fase di test sperimentali, un problema inizialmente imprevisto, la mancata percezione della direzionalità in ascolto, pur a fronte di un filtraggio verificato come corretto a livello di segnale, ha richiesto un lavoro di debug approfondito, che ha permesso di escludere il filtraggio HRTF e il suo caricamento come causa, isolando invece il problema in una combinazione di fattori a valle del plugin: un fattore di conversione di livello mancante e una sensibilità del sistema all'instradamento del segnale audio e al dispositivo di ascolto impiegato. Risolti questi aspetti, un test di percezione condotto con il simulatore di guida del laboratorio ha mostrato un'accuratezza media del 63,3% nel riconoscimento della direzione esatta su sei alternative, ma un'accuratezza del 100% nel riconoscimento del lato del guasto o pericolo, con errori concentrati quasi esclusivamente sulla confusione anteriore/posteriore, coerente con i limiti delle HRTF generiche discussi nel Capitolo 2. Questi risultati confermano che il sistema soddisfa i requisiti di progettazione enunciati in Sezione 5.1.

Nel complesso, il lavoro dimostra che è possibile estendere un sistema di comunicazione audio esistente con uno stadio di spazializzazione HRTF in tempo reale, rispettando i vincoli di un'elaborazione audio, e che il risultato, pur con i limiti discussi di seguito, è in grado di veicolare un'informazione direzionale effettivamente percepibile da un ascoltatore, particolarmente affidabile sulla distinzione laterale del pericolo.

7.2. Limiti del lavoro

Il lavoro presenta alcuni limiti, in parte già segnalati nel Capitolo 6.

- Il test di percezione con il simulatore di guida è stato condotto su un campione ridotto di tre soggetti: un numero sufficiente a evidenziare un pattern di errore coerente con la letteratura, ma non a trarre conclusioni statisticamente robuste sulla percentuale di riconoscimento del sistema in condizioni di guida reale. All'interno di questo campione, il sistema ha mostrato errori frequenti di *confusione fronte/retro*: come discusso nel Capitolo 2, questo è un limite noto delle HRTF non individualizzate, e non una criticità specifica dell'implementazione qui realizzata; resta tuttavia un limite pratico dell'affidabilità del sistema quando la distinzione anteriore/posteriore del pericolo è essa stessa l'informazione critica da comunicare.

- Il sistema, inoltre, non implementa un *crosstalk canceller*: l'intero setup attuale è stato validato esclusivamente in ascolto tramite cuffie, che per costruzione garantiscono la separazione dei due canali binaurali tra le due orecchie. La riproduzione su altoparlanti fisici, come discusso nel Capitolo 2, introduce invece il fenomeno del crosstalk, che senza un filtro dedicato comprometterebbe l'efficacia della spazializzazione; il sistema realizzato in questa tesi non è stato

quindi validato per un impiego su altoparlanti, condizione più realistica per un abitacolo automobilistico.

7.3. Sviluppi futuri

A partire da questi limiti, e dai risultati ottenuti, il lavoro suggerisce diverse direzioni di sviluppo futuro.

HRTF individualizzate. La confusione sistematica tra le direzioni anteriore e posteriore, osservata nel test di percezione, è un limite noto delle HRTF non individualizzate; l'impiego di HRTF personalizzate, o di tecniche di adattamento a partire da un piccolo numero di misurazioni individuali, potrebbe ridurre questo tipo di errore e migliorare l'affidabilità del sistema come strumento di allerta.

Combinazione tra contenuto semantico e posizione spaziale. Come discusso nel Capitolo 2, la letteratura suggerisce che un avviso spazialmente predittivo e semanticamente rilevante sia più efficace di uno dei due soli elementi. Un possibile sviluppo è quindi l'integrazione esplicita di un contenuto vocale, ad esempio la natura del guasto segnalato, con la spazializzazione HRTF già implementata, per verificare se la combinazione migliori ulteriormente i tempi di reazione e l'accuratezza di localizzazione.

Validazione su un campione più ampio. Un test di percezione condotto su un numero maggiore di soggetti, e in condizioni di guida più prossime a uno scenario reale, permetterebbe di quantificare con maggiore affidabilità l'accuratezza del sistema e di verificare se le percentuali di errori osservati in questo lavoro si confermano su scala più ampia.

Estensione della risoluzione direzionale. L'insieme di sei direzioni di riferimento impiegato in questo lavoro è stato scelto come compromesso tra semplicità implementativa e distinguibilità percettiva. Un ampliamento del numero di direzioni disponibili, o l'introduzione di una componente di elevazione oltre al solo azimuth, potrebbe aumentare la precisione con cui il sistema comunica la posizione di un pericolo, a fronte di un maggiore costo in termini di memoria e di test di validazione necessari.

Appendice A.

Codice Sorgente del Plugin

A.1. File `Plugin.h`

Listing A.1: Interfaccia PlugIn.h

```
#pragma once
#include "LEEffect.h"
#include <vector>


#define DR_WAV_IMPLEMENTATION
#include "dr_wav.h"

#include <atomic>
#include <thread>
#include <memory>

#include <ipp.h>

#define PLAYBUFFER_TXT "PlayBuffer"
#define SCALAR_TXT "Scalar"
#define VECTOR_TXT "Vector"
#define VARLEN_TXT "VarLen"

#define INT16_TXT "Integer␣16bit"
#define INT32_TXT "Integer␣32bit"
#define FLT32_TXT "Float␣32bit"
#define DBL64_TXT "Double␣64bit"

#define DATATYPEBITMASK VECTOR|SCALAR|BUFFSTREAMING|VARLEN
#define DATALENBITMASK DATAINT32|DATAINT16|DATAINT8|DATAFLOAT32|
    DATADOUBLE64|CUSTOM

#define WIDTHDEF 75

#define NUTS_NAME     "mqtt"
```

```
class MqttC
namespace mqtt { class async_client; }

class PlugIn : public LEEffect
{
public:
	PlugIn(InterfaceType _CBFunction,void * _PlugRef,HWND ParentDlg);

	~PlugIn(void);

	void __stdcall LESetName(char *Name);

	void __stdcall LEPlugin_Init();
	int  __stdcall LEPlugin_Process(PinType **Input,PinType **Output,
	    LPVOID ExtraInfo);
	void __stdcall LEPlugin_Delete(void);

	bool __stdcall LEInfoIO(int index,int type, char *StrInfo);

	int __stdcall LEGetNumInput(){return LEEffect::LEGetNumInput();};
	int __stdcall LEGetNumOutput(){return LEEffect::LEGetNumOutput();};
	int __stdcall LESetNumInput(int Val,PinType *TypeNewIn=0){return
	    LEGetNumInput();};
	int __stdcall LESetNumOutput(int Val,PinType *TypeNewOut=0){return
	    LEGetNumOutput();};

	void __stdcall LESetParameter(int Index,void *Data,LPVOID
	    bBroadCastInfo);
	int  __stdcall LEGetParameter(int Index,void *Data);

	int __stdcall LESetDefPin(int index,int type, PinType *Info);

	HWND __stdcall LEGetWndSet(){return 0;};
	int __stdcall LEWinSetStatusChange(int NewStatus){return 0;};

	void __stdcall LESaveSetUp();
	void __stdcall LELoadSetUp();

	void __stdcall LESampleRateChange(int NewVal,int TrigType);
	void __stdcall LEFrameSizeChange (int NewVal,int TrigType);
	void __stdcall LEConnectionChange(int IOType,int Index,bool
	    Connected){};
	int  __stdcall LEConnectionRequest(int IOType,int Index,PinType *
	    NewType);
```

```
        int __stdcall LEExtraInfoPinChange(int IOType,int Index,
            PinExtraInfoType ExInfo){return 0;};

        void __stdcall LERTWatchInit();
        LPVOID __stdcall LEOnNUTechMessage(int MessageType,int MessageID,
            WPARAM wParam,LPARAM lParam);


        void ApplyHRTF(int azimuth);

private:



        int FrameSize,SampleRate;

        unsigned int channels;
        unsigned int sampleRate;
        drwav_uint64 totalFrames;


        double* samples1;
        double* samples2;
        double* samples;

        int k;
        int N1;
        int N2;
        int N;
        int audio;

        int firTaps;
        Ipp64f* firCoeffsL;
        Ipp64f* firCoeffsR;



        IppsFIRSpec_64f* firStateL;
        IppsFIRSpec_64f* firStateR;
        Ipp8u* firBufL;
        Ipp8u* firBufR;

        Ipp64f* filteredBlockL = nullptr;
        Ipp64f* filteredBlockR = nullptr;
        Ipp64f* lastBlock = nullptr;

        Ipp64f* dlyL;
```

```
        Ipp64f* dlyR;

        bool modalitaFileRead = false;


        int currentAzimuth;

        static const int NUM_AZIMUTHS = 6;
        int azimuthSet[NUM_AZIMUTHS] = { 0, 200, 160, 70, 290, 180};

        Ipp64f* firCoeffsLCache[NUM_AZIMUTHS];
        Ipp64f* firCoeffsRCache[NUM_AZIMUTHS];


////////////////////////////////MQTT\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

    std::unique_ptr<mqtt::async_client> mqttClient;

        std::thread mqttThread;
        std::unique_ptr<MqttCb> mqttCb;
        std::atomic<int> pendingAudio{ 0 };
        std::atomic<bool> mqttRunning{ false };
        std::atomic<int> pendingAzimuth{ 0 };

        friend class MqttCb;

////////////////////////////////\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
};
```

A.2. File Plugin.cpp

Listing A.2: Implementazione PlugIn.cpp

```
\label{appendice}
#include "StdAfx.h"
#include ".\plugin.h"

#include <vector>
#include <iostream>

#include <windows.h>
#define DBG(msg) OutputDebugStringA(msg "\n")

#define DR_WAV_IMPLEMENTATION
#include "dr_wav.h"

#include <nlohmann/json.hpp>

#include <mqtt/async_client.h>

#include <atomic>
#include <thread>
#include <memory>

#include <chrono>

#include <ipp.h>

class MqttCb : public mqtt::callback {
	PlugIn* self;

public:

	explicit MqttCb(PlugIn* p) : self(p) {}

	void message_arrived(mqtt::const_message_ptr msg) override {

		DBG("[MQTT] messaggio ricevuto");

		try {

			auto payload = msg->to_string();
			auto j = nlohmann::json::parse(payload);

			int a = j.value("audio", 0);
			int az = j.value("azimuth", 90);
```

```
                        if (a == 1 || a == 2) {
                                self->pendingAzimuth.store(az, std::
                                    memory_order_release);
                                self->pendingAudio.store(a, std::
                                    memory_order_release);

                        }

                        OutputDebugStringA(("MQTT␣msg:␣" + payload + "\n").
                            c_str());
                }
                catch (...) {

                        DBG("[MQTT]␣JSON␣invalido");
                }
        }

        void connection_lost(const std::string&) override {

        }
};

static bool LoadHRTF(const char* path, Ipp64f* out, int nTaps)
{
        unsigned int channels;
        unsigned int sampleRate;
        drwav_uint64 totalFrames;

        float* data = drwav_open_file_and_read_pcm_frames_f32(
                path, &channels, &sampleRate, &totalFrames, nullptr);

        if (!data) {
                OutputDebugStringA(("[HRTF]␣File␣non␣trovato:␣" + std::string
                    (path) + "\n").c_str());
                return false;
        }

        char dbg[256];

        sprintf(dbg, "[HRTF]␣channels=%d␣sampleRate=%d␣totalFrames=%llu␣max
            =%f\n",
                channels, sampleRate, totalFrames, data[0]);
        OutputDebugStringA(dbg);


        if ((int)totalFrames < nTaps) {
                OutputDebugStringA("[HRTF]␣File␣troppo␣corto\n");
```

```
                drwav_free(data, nullptr);
                return false;
        }


        for (int i = 0; i < nTaps; i++)
                out[i] = data[i];

        drwav_free(data, nullptr);
        return true;
}

void PlugIn::ApplyHRTF(int azimuth)
{

        azimuth = ((azimuth % 360) + 360) % 360;


        int bestIdx = 0;
        int bestDist = 361;
        for (int i = 0; i < NUM_AZIMUTHS; i++) {
                int d = abs(azimuth - azimuthSet[i]);
                d = (std::min)(d, 360 - d);
                if (d < bestDist) { bestDist = d; bestIdx = i; }
        }


        memcpy(firCoeffsL, firCoeffsLCache[bestIdx], firTaps * sizeof(Ipp64f
            ));
        memcpy(firCoeffsR, firCoeffsRCache[bestIdx], firTaps * sizeof(Ipp64f
            ));

        ippsFIRSRInit_64f(firCoeffsL, firTaps, ippAlgAuto, firStateL);
        ippsFIRSRInit_64f(firCoeffsR, firTaps, ippAlgAuto, firStateR);

        ippsZero_64f(dlyL, firTaps - 1);
        ippsZero_64f(dlyR, firTaps - 1);

        currentAzimuth = azimuthSet[bestIdx];

        char buf[128];
        sprintf(buf, "[HRTF] Azimuth richiesto=%d -> usato=%d (idx=%d)\n",
            azimuth, azimuthSet[bestIdx], bestIdx);
        OutputDebugStringA(buf);
}

PlugIn::PlugIn(InterfaceType _CBFunction,void * _PlugRef,HWND ParentDlg):
```

```
    LEEffect(_CBFunction,_PlugRef,ParentDlg)
{
        FrameSize = CBFunction(this,NUTS_GET_FS_SR,0,(LPVOID)AUDIOPROC);
        SampleRate = CBFunction(this,NUTS_GET_FS_SR,1,(LPVOID)AUDIOPROC);
        k = -1;
        N1 = 132300;
        N2 = 10196352;
        N = 0;
        samples1 = 0;
        samples2 = 0;
        samples = 0;
        audio = 1;
        pendingAudio = 0;

        firTaps = 0;
        firCoeffsL = nullptr;
        firCoeffsR = nullptr;
        firBufL = nullptr;
        firBufR = nullptr;
        firStateL = nullptr;
        firStateR = nullptr;

        dlyL = nullptr;
        dlyR = nullptr;

        for (int i = 0; i < NUM_AZIMUTHS; i++) {
                firCoeffsLCache[i] = nullptr;
                firCoeffsRCache[i] = nullptr;
        }

        LESetNumInput(1);
        LESetNumOutput(2);
}

int __stdcall PlugIn::LEPlugin_Process(PinType **Input,PinType **Output,
    LPVOID ExtraInfo)
{
        double* InputData = ((double*)Input[0]->DataBuffer);
        double* OutputDataL = ((double*)Output[0]->DataBuffer);
        double* OutputDataR = ((double*)Output[1]->DataBuffer);

        if (k == 0) {
                char bufFS[64];
                sprintf(bufFS, "[DEBUG] FrameSize = %d", FrameSize);
                OutputDebugStringA(bufFS);
        }
```

```
if (firStateL == nullptr || firStateR == nullptr || firBufL ==
    nullptr || firBufR == nullptr)
        return COMPLETED;

if (modalitaFileRead)
{
        double* OutputDataL = ((double*)Output[0]->DataBuffer);
        double* OutputDataR = ((double*)Output[1]->DataBuffer);
        ippsFIRSR_64f(InputData, OutputDataL, FrameSize, firStateL,
            dlyL, dlyL, firBufL);
        ippsFIRSR_64f(InputData, OutputDataR, FrameSize, firStateR,
            dlyR, dlyR, firBufR);

        for (int n = 0; n < FrameSize; ++n)
                OutputDataR[n] = 0.0;

        return COMPLETED;
}

++k;

int req = pendingAudio;
if (req != 0) {


        if ((k - 1) * FrameSize >= N) {
                k = 0;
                pendingAudio = 0;

                if (req == 1) {
                        N = N1;

                        for (int n = 0; n < N1; ++n) {
                                samples[n] = samples1[n];
                        }
                        ApplyHRTF(pendingAzimuth.load());
                }

                else if (req == 2) {
                        N = N2;
                        for (int n = 0; n < N2; ++n) {
                                samples[n] = samples2[n];
                        }
                        ApplyHRTF(pendingAzimuth.load());
                }

                req = 0;
```

```
        }
    }

    const int start = k * FrameSize;

    ippsZero_64f(filteredBlockL, FrameSize);
    ippsZero_64f(filteredBlockR, FrameSize);

    if (start + FrameSize <= N)
    {
        ippsFIRSR_64f(samples + start, filteredBlockL, FrameSize,
            firStateL, dlyL, dlyL, firBufL);
        ippsFIRSR_64f(samples + start, filteredBlockR, FrameSize,
            firStateR, dlyR, dlyR, firBufR);

        double* OutputDataL = ((double*)Output[0]->DataBuffer);
        double* OutputDataR = ((double*)Output[1]->DataBuffer);

        for (int n = 0; n < FrameSize; ++n) {
            OutputDataL[n] = filteredBlockL[n] * 32768.0;
            OutputDataR[n] = filteredBlockR[n] * 32768.0;
        }
        return COMPLETED;
    }

    if (start + FrameSize >= N)
    {
        ippsZero_64f(lastBlock, FrameSize);
        ippsZero_64f(filteredBlockL, FrameSize);
        ippsZero_64f(filteredBlockR, FrameSize);

        for (int n = 0; n < FrameSize; ++n)
            if (start + n < N)
                lastBlock[n] = samples[start + n];

        ippsFIRSR_64f(lastBlock, filteredBlockL, FrameSize, firStateL
            , dlyL, dlyL, firBufL);
        ippsFIRSR_64f(lastBlock, filteredBlockR, FrameSize, firStateR
            , dlyR, dlyR, firBufR);

        for (int n = 0; n < FrameSize; ++n) {
            OutputDataL[n] = filteredBlockL[n] * 32768.0;
            OutputDataR[n] = filteredBlockR[n]* 32768.0;
        }

        return COMPLETED;
    }
```

```
        return COMPLETED;
}

void __stdcall PlugIn::LEPlugin_Init()
{
        if (samples1 == 0)
        {
                samples1 = new double[N1];
        }

        if (samples2 == 0)
        {
                samples2 = new double[N2];
        }

        if (samples == 0)
        {
                samples = new double[N2];
        }

        float* data1 = drwav_open_file_and_read_pcm_frames_f32(
                "C:\\Users\\chiar\\OneDrive\\Desktop\\progetto␣NUTECH\\
                    rumore_bianco_3s.wav", &channels, &sampleRate, &
                    totalFrames, nullptr);
        if (!data1) {
                std::cerr << "Errore␣apertura␣file␣WAV\n";
                return;
        }

        for (int i = 0; i < N1; i++) {
                samples1[i] = data1[i];
        }
        drwav_free(data1, nullptr);

        float* data2 = drwav_open_file_and_read_pcm_frames_f32(
                "C:\\Users\\chiar\\OneDrive\\Desktop\\song_mono.wav", &
                    channels, &sampleRate, &totalFrames, nullptr );

        if (!data2) {
                std::cerr << "Errore␣apertura␣file␣WAV\n";
                return;
        }

        for (int i = 0; i < N2; i++) {
                samples2[i] = data2[i];
        }
```

```
drwav_free(data2, nullptr);

firTaps = 512;

firCoeffsL = new Ipp64f[firTaps];

firCoeffsR = new Ipp64f[firTaps];

int specSize = 0;
int bufSize = 0;
ippsFIRSRGetSize(firTaps, ipp64f, &specSize, &bufSize);

firStateL = (IppsFIRSpec_64f*) new Ipp8u[specSize];
firBufL = new Ipp8u[bufSize];
firStateR = (IppsFIRSpec_64f*) new Ipp8u[specSize];
firBufR = new Ipp8u[bufSize];

filteredBlockL = new Ipp64f[FrameSize];
filteredBlockR = new Ipp64f[FrameSize];
lastBlock = new Ipp64f[FrameSize];

dlyL = ippsMalloc_64f(firTaps - 1);
ippsZero_64f(dlyL, firTaps - 1);
dlyR = ippsMalloc_64f(firTaps - 1);
ippsZero_64f(dlyR, firTaps - 1);

for (int i = 0; i < NUM_AZIMUTHS; i++) {
        firCoeffsLCache[i] = new Ipp64f[firTaps];
        firCoeffsRCache[i] = new Ipp64f[firTaps];

        char pathL[256], pathR[256];
        sprintf(pathL, "C:\\Users\\chiar\\OneDrive\\Desktop\\progetto
            ␣NUTECH\\full\\elev0\\L0e%03da.wav", azimuthSet[i]);
        sprintf(pathR, "C:\\Users\\chiar\\OneDrive\\Desktop\\progetto
            ␣NUTECH\\full\\elev0\\R0e%03da.wav", azimuthSet[i]);

        if (!LoadHRTF(pathL, firCoeffsLCache[i], firTaps)) {
                ippsZero_64f(firCoeffsLCache[i], firTaps);
                firCoeffsLCache[i][0] = 1.0;
        }
        if (!LoadHRTF(pathR, firCoeffsRCache[i], firTaps)) {
                ippsZero_64f(firCoeffsRCache[i], firTaps);
                firCoeffsRCache[i][0] = 1.0;
        }
}

ApplyHRTF(270);
```

```
for (int i = 0; i < N1; i++)
       samples[i] = samples1[i];

mqttRunning.store(true);

mqttThread = std::thread([this]()
       { DBG("[MQTT] Thread partito");
              try
              {
                     mqttClient = std::make_unique<mqtt::
                         async_client>(
                            "tcp://127.0.0.1:1883",

                            "nutech_plugin"

                     );

                     mqttCb = std::make_unique<MqttCb>(this);
                     mqttClient->set_callback(*mqttCb);


                     mqtt::connect_options opts;
                     opts.set_clean_session(true);


                     mqttClient->connect(opts)->wait();


                     DBG("[MQTT] Connesso al broker");

                     mqttClient->subscribe("sensori/comandi", 1)->
                         wait();
                     DBG("[MQTT] Subscribe OK su sensori/comandi");

                     while (mqttRunning.load())
                     {
                            std::this_thread::sleep_for(std::chrono
                                ::milliseconds(50));
                     }

                     mqttClient->disconnect()->wait();
              }
              catch (const mqtt::exception& e) {
                     DBG("[MQTT] errore di connessione");
                     return;
              }
```

```
            });
}

void __stdcall PlugIn::LEPlugin_Delete()
{
        if (samples1 != 0)
        {
                delete[] samples1;
                samples1 = 0;
        }

        if (samples2 != 0)
        {
                delete[] samples2;
                samples2 = 0;
        }

        if (samples != 0)
        {
                delete[] samples;
                samples = 0;
        }

        if (firCoeffsL) { delete[] firCoeffsL; firCoeffsL = nullptr; }
        if (firCoeffsR) { delete[] firCoeffsR; firCoeffsR = nullptr; }
        if (firBufL) { delete[] firBufL; firBufL = nullptr; }
        if (firBufR) { delete[] firBufR; firBufR = nullptr; }
        if (firStateL) { delete[](Ipp8u*)firStateL; firStateL = nullptr; }
        if (firStateR) { delete[](Ipp8u*)firStateR; firStateR = nullptr; }

        if (filteredBlockL) { delete[] filteredBlockL; filteredBlockL =
            nullptr; }
        if (filteredBlockR) { delete[] filteredBlockR; filteredBlockR =
            nullptr; }
        if (lastBlock) { delete[] lastBlock; lastBlock = nullptr; }

        if (dlyL) { ippsFree(dlyL); dlyL = nullptr; }
        if (dlyR) { ippsFree(dlyR); dlyR = nullptr; }

        for (int i = 0; i < NUM_AZIMUTHS; i++) {
                if (firCoeffsLCache[i]) { delete[] firCoeffsLCache[i];
                    firCoeffsLCache[i] = nullptr; }
                if (firCoeffsRCache[i]) { delete[] firCoeffsRCache[i];
                    firCoeffsRCache[i] = nullptr; }
        }

        mqttRunning.store(false);
```

```
	if (mqttThread.joinable())
		mqttThread.join();

	mqttClient.reset();
	mqttCb.reset();
}

PlugIn::~PlugIn(void)
{

}

bool __stdcall PlugIn::LEInfoIO(int index,int type, char *StrInfo)
{
	if(type==INPUT) sprintf(StrInfo,"In[%d]",index);
	if(type==OUTPUT) sprintf(StrInfo,"Out[%d]",index);
	return true;
}

int __stdcall PlugIn::LESetDefPin(int index,int type, PinType *Info)
{
	if (type==OUTPUT)
	{
		Info->DataType=PLAYBUFFER;
		Info->Exclusive=false;
		Info->DataLen=FrameSize;
		Info->MaxDataLen=FrameSize;
		return OUTPUT;
	}

	if (type==INPUT)
	{
		Info->DataType=PLAYBUFFER;
		Info->Exclusive=false;
		Info->DataLen=FrameSize;
		Info->MaxDataLen=FrameSize;
		return INPUT;
	}

	return -1;
}

int __stdcall PlugIn::LEConnectionRequest(int IOType,int Index,PinType *
    NewType)
{
	return 0;
```

```
}

LPVOID __stdcall PlugIn::LEOnNUTechMessage(int MessageType,int MessageID,
    WPARAM wParam,LPARAM lParam)
{
       return 0;
}

void __stdcall PlugIn::LESetName(char *Name)
{
       strncpy(Name, NUTS_NAME, MAXNAME);
}

void __stdcall PlugIn::LESetParameter(int Index,void *Data,LPVOID
    bBroadCastInfo)
{

}

int  __stdcall PlugIn::LEGetParameter(int Index,void *Data)
{
       return 0;
}

void __stdcall PlugIn::LESaveSetUp()
{

}

void __stdcall PlugIn::LELoadSetUp()
{

}

void __stdcall PlugIn::LERTWatchInit()
{

}

void __stdcall PlugIn::LESampleRateChange(int NewVal,int TrigType)
{
       if(TrigType==AUDIOPROC)
       {
              if(NewVal!=SampleRate)
              {
                     SampleRate = NewVal;
              }
```

```
	}
}

void __stdcall PlugIn::LEFrameSizeChange (int NewVal,int TrigType)
{
	if (TrigType == AUDIOPROC)
	{
		if (NewVal != FrameSize)
		{
			FrameSize = NewVal;

			if (filteredBlockL) delete[] filteredBlockL;
			if (filteredBlockR) delete[] filteredBlockR;
			if (lastBlock) delete[] lastBlock;

			filteredBlockL = new Ipp64f[FrameSize];
			filteredBlockR = new Ipp64f[FrameSize];
			lastBlock = new Ipp64f[FrameSize];

			ippsZero_64f(filteredBlockL, FrameSize);
			ippsZero_64f(filteredBlockR, FrameSize);
		}
	}
}

LEEffect * __stdcall LoadEffect(InterfaceType _CBFunction,void *PlugRef,
    HWND ParentDlg)
{
	return new PlugIn(_CBFunction,PlugRef,ParentDlg);
}

int __stdcall UnLoadEffect(PlugIn *effect)
{
	delete effect;
	return TRUE;
}

void __stdcall LENUTSDefProps(char *NameEffect,int *Width, void *data)
{
	strncpy(NameEffect,NUTS_NAME,MAXNAME);
	*Width=WIDTHDEF;

	if (data!=0)
	{
		StartUpNUTSProps *Info=(StartUpNUTSProps *)data;
		Info->NumInStartUp=1;
		Info->NumOutStartUp=2;
```

```
                Info->BitMaskProc = AUDIOPROC;
                Info->BitMaskDriver = OFFLINEDRIVER | DIRECTDRIVER |
                    ASIODRIVER;
        }
}
```

Bibliografia

[1] William G. Gardner. *3D Audio and Acoustic Environment Modeling*. Wave Arts, Inc., 1999. Disponibile su: https://headwizememorial.wordpress.com/2018/03/16/3d-audio-and-acoustic-environment-modeling/.

[2] A. van Laack and A. Torschmied. Immersive audio HMI to improve situational awareness. Visteon Innovation & Technology GmbH, 2015. https://www.researchgate.net/publication/346106276_Immersive_Audio_HMI_to_Improve_Situational_Awareness.

[3] M. Wolters, J. Giralt, H. Mundt, and A. Biswas. Evaluation of headrest-integrated loudspeakers for enhanced spatial audio immersion in automotive cabins. Dolby Germany GmbH, 2026. https://www.researchgate.net/publication/407645534_Evaluation_of_Headrest-Integrated_Loudspeakers_for_Enhanced_Spatial_Audio_Immersion_in_Automotive_Cabins.

[4] MQTT protocol specification. OASIS Standard, 2019. Disponibile su: https://mqtt.org/.

[5] Jens Blauert. *Spatial Hearing*. MIT Press, Cambridge, MA, 1983.

[6] Robert H. Gilkey and Timothy R. Anderson, editors. *Binaural and Spatial Hearing in Real and Virtual Environments*. Lawrence Erlbaum Associates, Mahwah, NJ, 1997.

[7] Frederic L. Wightman and Doris J. Kistler. Headphone simulation of free-field listening. ii: Psychophysical validation. *Journal of the Acoustical Society of America*, 85(2):868–878, 1989.

[8] Frederic L. Wightman and Doris J. Kistler. Headphone simulation of free-field listening. i: Stimulus synthesis. *Journal of the Acoustical Society of America*, 85(2):858–867, 1989.

[9] Elizabeth M. Wenzel, Marianne Arruda, Doris J. Kistler, and Frederic L. Wightman. Localization using nonindividualized head-related transfer functions. *Journal of the Acoustical Society of America*, 94(1):111–123, 1993.

[10] William G. Gardner and Keith D. Martin. Hrtf measurements of a kemar dummy-head microphone. https://sound.media.mit.edu/resources/KEMAR.html. MIT Media Lab. Dataset consultato nel 2026.

[11] A. Arabian, C. M. Goodridge, and N. Merat. Asynchrony of directional auditory warnings and visual information in hands-on SAE level 2 automated vehicles: Analysis of driver responses. *PLoS ONE*, 20(11):e0336941, 2025. Disponibile su: `https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0336941`.

[12] J. Chen, E. Šabić, S. Mishler, C. Parker, and M. Yamaguchi. Effectiveness of lateral auditory collision warnings: Should warnings be toward danger or toward safety? *Human Factors*, 64(2):418–435, 2022.

[13] J. de Winter, J. Hu, and B. Petermeijer. Ipsilateral and contralateral warnings: effects on decision-making and eye movements in near-collision scenarios. *Journal on Multimodal User Interfaces*, 16(3):303–317, 2022.

[14] C. Ho and C. Spence. Assessing the effectiveness of various auditory cues in capturing a driver's visual attention. *Journal of Experimental Psychology: Applied*, 11(3):157–174, 2005.

[15] Francis Rumsey. *Spatial Audio*. Focal Press, Oxford, 2001.

[16] F. Lattanzi, F. Bettarelli, and S. Cecchi. Nu-tech: The entry tool of the hartes toolchain for algorithms design. In *124th Convention of the Audio Engineering Society (AES)*, pages 1–8, May 2008.

[17] Eclipse Foundation. Eclipse Paho MQTT C++ client library. Eclipse Project, 2023. Disponibile su: `https://github.com/eclipse/paho.mqtt.cpp`.

[18] Niels Lohmann. JSON for modern C++. GitHub Repository, 2024. Disponibile su: `https://github.com/nlohmann/json`.

[19] David Reid. dr_wav: single file audio decoding library for C/C++. GitHub Repository, 2023. Disponibile su: `https://github.com/mackron/dr_libs`.

[20] Intel Corporation. Intel® integrated performance primitives (intel® ipp) developer guide and reference. `https://www.intel.com/content/www/us/en/docs/ipp/developer-guide-reference/`. Consultato nel 2026.

[21] Roger Light. Eclipse Mosquitto: An open source MQTT broker. Eclipse Project, 2023. Disponibile su: `https://mosquitto.org/`.

[22] EMQ Technologies Co., Ltd. MQTTX: Advanced MQTT 5.0 desktop client. Open Source software, 2024. Disponibile su: `https://mqttx.app/`.