POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Ad insertion dinamico e personalizzato in flussi radiofonici digitali

Relatore
Prof. Antonio Servetti

Candidato
Francesco Raitano

Anno accademico 2019-2020

A mia nonna,
grande esempio di
coraggio e tenacia

Sommario

La pubblicità personalizzata è un potente strumento ampiamente utilizzato nel mondo digitale, perché consente di fornire all'utente contenuti di suo interesse e di incrementare il ritorno sull'investimento delle aziende. Questa tesi mira ad integrare tale strumento all'interno degli stream radiofonici, senza alterare il palinsesto delle radio. La soluzione proposta prevede la sostituzione in tempo reale delle inserzioni già presenti con quelle personalizzate per l'utente in ascolto. A tale scopo è stato sviluppato un plugin audio, che tramite sola analisi dei campioni all'interno dello stream broadcast, consente l'individuazione dei punti di interesse e la sostituzione pubblicitaria in totale trasparenza. Due sono gli approcci implementati: nel primo l'inizio e la fine dell'inserzione sono individuate tramite rilevazione di toni non udibili, nel secondo è stata realizzata una tecnica di audio fingerprint ad hoc per il riconoscimento dei jingle, piccoli frammenti musicali, inseriti dalle stazioni radiofoniche per anticipare e concludere un momento pubblicitario. Il secondo approccio, che, per identificare univocamente i jingle implementa un algoritmo simile a quello utilizzato dal software Shazam per il riconoscimento delle canzoni, ha il vantaggio di non richiedere nessuna alterazione del flusso audio originale. Il plugin è stato sviluppato utilizzando il framework JUCE, in modo tale da generare dei formati multipiattaforma compatibili con le principali DAW in commercio.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Secondo il report "The Infinite Dial 2020"[1] ricavato da Edison Research e Triton Digital tramite sondaggio nazionale rivolto alla popolazione Americana sopra i 12 anni, circa il 60% degli Americani ascolta contenuti in streaming (stazioni radio online e musica on demand) con una media di 15 ore a settimana, il 62% afferma di avere utilizzato almeno una tipologia di assistente vocale, il 45% ha usufruito di uno smartphone per l'ascolto di audio online in macchina, ed il 18% della popolazione con età superiore ai 18 anni possiede auto con infotainment di nuova generazione (Android Auto, Apple CarPlay) con accesso diretto a internet. Dai report pubblicati di anno in anno, in cui sono raccolti dati sul comportamento degli utenti nei confronti dei digital media, è possibile notare come i contenuti audio (intrattenimento ed informazione) siano sempre più accessibili nella vita di tutti i giorni: è infatti ormai comune richiedere la riproduzione di una stazione radiofonica tramite comando vocale, rimanendo seduti sul divano di casa, o mantenendo ben salde le mani sul volante durante la guida. Dal 2017 al 2020 si stima che 220 milioni di persone in America (78% della popolazione) siano entrate in contatto con degli smart speaker, e 76 milioni (27% della popolazione) ne posseggono almeno uno, con un trend in forte crescita; nel 2020 sono stati utilizzati in ambito domestico una media di 2.2 smart speaker (dei tre principali competitor: Amazon, Google, Apple), contro una media di 1.7 registrata nel 2018. Il confronto in percentuale della popolazione che possiede smart speaker e/o dispositivi tradizionali per l'ascolto radio FM/AM è mostrato in figura 1.1.

[1] https://www.edisonresearch.com/the-infinite-dial-2020/

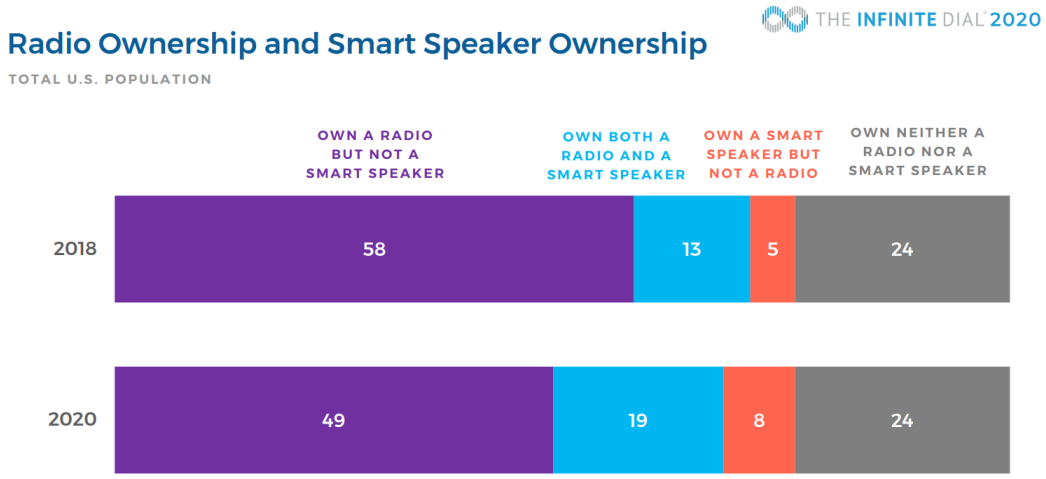

Figura 1.1: Percentuale della popolazione Americana in possesso di radio e smart speaker, secondo il sondaggio "The Infinite Dial 2020".

Connessioni a banda larga e nuovi end point per la fruizione multimediale stanno avendo un'influenza positiva sulla creazione di contenuti live, permettendone la trasmissione in tempo reale tramite HTTP streaming (protocolli HLS, Microsoft Smooth Streaming, DASH). Nonostante il principale canale di diffusione radio sia ancora la trasmissione terrestre FM/AM, le stazioni radiofoniche hanno allargato il bacino di utenza integrando la trasmissione dello stesso flusso su internet, in modo tale da generare ascolti attraverso un sito web, un'applicazione mobile, uno smart speaker o un infotainment auto di nuova generazione. In Italia, secondo il rapporto Censis 2018, la radio tradizionale perde il 2.9% di utenza (56.2% complessiva) mentre l'ascolto delle trasmissioni radiofoniche da pc e smartphone è in aumento (20.7% degli Italiani, +1.6% in un anno). Questi punti di accesso, a differenza dei tradizionali, aprono nuove possibilità, tra cui la più rilevante è la registrazione degli utenti tramite account. Ne consegue che attraverso un'analisi dei dati raccolti esplicitamente ed implicitamente mediante tecniche di profilazione, è possibile proporre loro contenuti e annunci personalizzati innalzando il livello di fidelizzazione. Lo strumento dei contenuti personalizzati è in utilizzo da anni all'interno di pagine web e social media, poiché consente di mantenere alta l'attenzione e l'interesse dell'utente migliorando allo stesso tempo il ROI delle aziende [1].

Profilazione degli utenti

L'user profiling può essere definito come il processo di identificazione dei dati inerenti agli interessi dell'utente, come spiegato in [2] è possibile classificare in tre tipologie le tecniche di profiling:

- le **esplicite** prevedono l'inserimento di alcuni dati da parte dell'utente come ad esempio età, sesso, città, regione di residenza tramite form o sondaggi. Questo approccio è chiamato anche static profiling, ed è soggetto a degradazione nel tempo.
- le **implicite** raccolgono passivamente dati in base al comportamento dell'utente all'interno della piattaforma, ad esempio la tipologia di contenuti visualizzata o ricercata.
- le **ibride** combinano le due precedenti con un maggior livello di efficacia.

Il filtraggio dei contenuti da proporre all'utente, con lo scopo di aumentare la sua soddisfazione, avviene invece tramite tecniche "Content Based Filtering" e "Collaborative Filtering". Nella prima tipologia si propongono contenuti molto simili a quelli che l'utente ha già visualizzato, richiesto o recensito, ed è fortemente dipendente dalle preferenze date da quest'ultimo sui vari prodotti. La difficoltà della tecnica di filtraggio basata sui contenuti sta nell'escludere le interazioni spurie, ad esempio le ricerche di un prodotto per conto di un amico e non di proprio interesse. La seconda tipologia d'altro canto raggruppa gli utenti con interessi simili, in modo tale da proporre gli stessi contenuti all'interno dello stesso gruppo; l'efficacia di questa tecnica tuttavia dipende dalla qualità della suddivisione in cluster degli utenti.

Obiettivi

Considerando la transizione delle radio nel mondo digitale, e i benefici attesi da una targetizzazione del pubblico, questa tesi mira ad integrare lo strumento delle pubblicità personalizzate, previa corretta profilazione dell'utente, all'interno di flussi radiofonici, senza alterare il palinsesto e la programmazione prestabilita. La soluzione proposta prevede infatti la sostituzione delle inserzioni generiche già presenti con quelle personalizzate per l'utente in ascolto. L'obiettivo finale è lo sviluppo di un software

che tramite sola analisi dei campioni in ingresso individui i punti di interesse coincidenti con l'inizio e la fine delle inserzioni, e consenta una sostituzione in totale trasparenza. Due sono gli approcci implementati: nel primo si ricercano marcatori audio, nel secondo frammenti musicali di breve durata chiamati jingle. Il primo approccio necessita una modifica dello stream prima della messa in onda, ovvero l'inserimento nel flusso dei toni, che hanno lo scopo di segnalare l'arrivo di una pubblicità. Il secondo approccio implementa una tecnica di audio fingerprint per la ricerca dei jingle, e ne consente una loro identificazione univoca, allo stesso modo in cui software audio effettuano il processo di song recognition. Questa implementazione rappresenta la parte più corposa di questo lavoro ed ha, a differenza della prima, il vantaggio di mantenere immutato lo stream di partenza; difatti i jingle sono già inseriti dalle stazioni radiofoniche per individuare momenti ricorrenti quali inserzioni pubblicitarie, aggiornamenti sul traffico e così via. La tecnica di audio fingerprint esegue una pre-analisi dei jingle per conservarne la loro "impronta", successivamente un'analisi in tempo reale dello stream in ingresso confrontando, attraverso una finestratura, le corrispondenze con le tracce analizzate in precedenza.

Organizzazione della tesi

Nel capitolo 2 sono esposti i concetti alla base del processamento dell'audio digitale. Nel capitolo 3 vengono introdotte ed analizzate le tecnologie utilizzate per l'implementazione delle soluzioni proposte. Nei capitoli 4 e 5 è descritto l'intero sviluppo dei vari componenti che permettono il corretto funzionamento del software finale, mentre nel capitolo 6 ne è descritto un breve manuale per l'utilizzo. Infine nei capitoli 7 e 8 sono riportati i risultati del riconoscimento e sostituzione dinamica al variare dei parametri dell'algoritmo di fingerprint implementato, seguiti da conclusioni e sviluppi futuri.

Capitolo 2

Il segnale audio

Prima di addentrarci nel mondo delle elaborazioni audio, in questo capitolo saranno espressi i concetti base riguardanti la musica ed il suono, partendo dalle loro caratteristiche fisiche fino alla loro rappresentazione analogica e digitale, affrontando i processi di campionamento e quantizzazione.

2.1 Il suono e le sue caratteristiche

Il suono è un'onda meccanica che si propaga attraverso un mezzo (come l'aria o l'acqua) mediante compressione e rarefazione dello stesso fino al raggiungimento dell'apparato uditivo. La sua rappresentazione è detta segnale audio, che ne consente quindi l'archiviazione, manipolazione e trasmissione. Un esempio di suono puro è ottenibile tramite sollecitazione del diapason[1], che genera un'onda sinusoidale caratterizzata da:

- una **ampiezza** rappresentante la massima estensione ottenuta dalla forma d'onda all'interno di un'oscillazione rispetto al valore di pressione locale medio.
- una **frequenza** rappresentante il numero di oscillazioni che l'onda compie in un secondo, misurata in *Hertz* (Hz).

L'intensità oggettiva di un suono è calcolata tramite il parametro Sound Pressure Level (SPL), che confronta la pressione del suono in esame con

[1]Strumento consistente in una forcella di acciaio che in fase di percussione genera un suono molto puro, privo di frequenze armoniche.

la fondamentale $p_0 = 2.5 \cdot 10^{-5} \frac{N}{m^2}$ nel seguente modo:

$$SPL = 20 \log_{10} \frac{p}{p_0} dB = 10 \log_{10} \frac{p^2}{p_0^2} dB = 10 \log_{10} \frac{I}{I_0} dB \tag{2.1}$$

Il nostro apparato uditivo è capace di percepire suoni nell'intervallo $[2.5 \cdot 10^{-5} \frac{N}{m^2}, 3 \cdot 10 \frac{N}{m^2}]$ ed essendo un range elevato di valori, si riporta su scala logaritmica (SPL). Tipicamente a 50 dB corrisponde il silenzio dentro casa, a 60 dB una normale conversazione, a 100 dB il rumore di un martello pneumatico, mentre i casi limite sono raggiunti da un aereo in fase di decollo che arriva a toccare la soglia dei 120 dB, chiamata anche soglia del dolore poiché da questo valore in su, in base al tempo di esposizione, il nostro apparato uditivo inizia a subire delle lesioni anche permanenti. Per quanto riguarda la frequenza invece, un individuo in perfette condizioni di salute riesce a percepire suoni nel range di frequenze 20 Hz - 20 KHz, che si riduce con l'aumentare dell'età. La perdita avviene specialmente nelle alte frequenze, come se al passare degli anni fosse applicato un filtro passa basso con soglia di attivazione in continua decrescita.

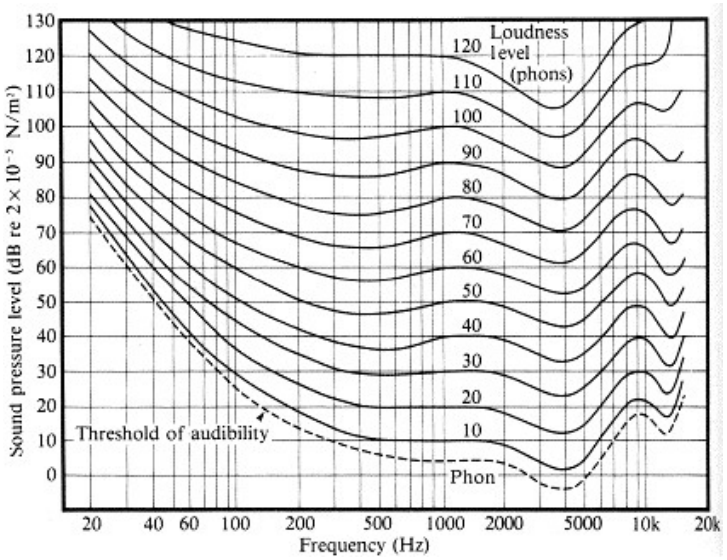

Figura 2.1: Curve isofoniche, al variare della frequenza.

Il volume percepito dall'apparato uditivo tuttavia non è uguale per tutte le frequenze, infatti a parità di SPL si percepisce una maggiore 'intensità' nelle medie / medio-alte frequenze ed una minore 'intensità' nelle basse ed alte frequenze. Essendo una misura soggettiva sono stati eseguiti

diversi test su gruppi di persone, ed infine stilate delle curve (mostrate in figura 2.1), dette isofoniche, che generalizzano la percezione sonora al variare delle frequenze. Per convenzione la curva 0 foni corrisponde alla minima soglia di udibilità che ad 1 KHz ha un valore di 0 dB, le successive curve mostrano all'incirca un raddoppio del volume percepito ogni 10 dB. Volendo infine fare un esempio, se ad 1 KHz servisse una sorgente sonora per generare un suono ad un determinato volume, a 200 Hz servirebbero 100 sorgenti sonore dello stesso tipo per percepire lo stesso volume.

2.1.1 I suoni reali

Difficilmente nella realtà si trovano suoni puri a singola frequenza, solitamente contengono altre componenti che possono essere di due tipi:

- **armoniche**, multiple intere di una frequenza fondamentale. In questo caso il suono rimane periodico, o quasi periodico, ed è caratterizzato da una frequenza fondamentale, o pitch (la più bassa). Come ad esempio il suono di un violino.
- **parziali** o componenti non armoniche, nello spettro viste come dei picchi senza una specifica relazione, ed in questo caso non è possibile riconoscere una chiara periodicità. Come ad esempio il suono di un tamburo.

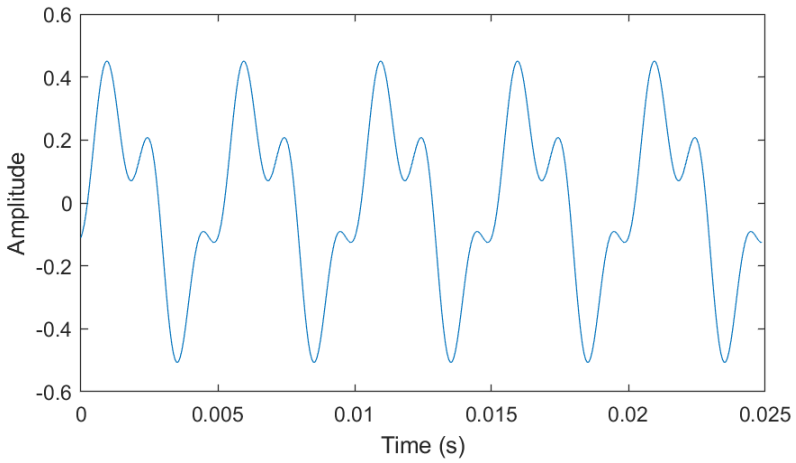

Figura 2.2: Suono periodico con frequenza fondamentale 200 Hz, e sole componenti armoniche.

In musica, per risultare un brano orecchiabile, in fase di composizione si seguono delle regole di 'armonia' che specificano il modo in cui è possibile combinare le note in accordi per ottenere delle successioni gradevoli al nostro orecchio, ed è sostanzialmente un'interfaccia per consentire la generazione di suoni con sole componenti armoniche e non parziali, in quest'ultimo caso si parla invece di dodecafonia. In figura 2.2 è mostrato l'esempio di un tono con frequenza fondamentale 200 Hz e due componenti armoniche, nello specifico la seconda e la terza, presenti ad una minore intensità. I suoni reali sono inoltre dei transienti, ovvero hanno una durata finita, ed il loro profilo nel tempo è detto inviluppo temporale nel quale è possibile riconoscere quattro fasi: attack, decay, sustain e release. La terza proprietà di un suono infine è il timbro, che ci consente a parità di frequenza fondamentale e volume di identificare due strumenti che suonano la stessa nota. Se cambia il contenuto in frequenza cambia anche il timbro, ma non è detto che a parità di contenuto in frequenza il timbro sia lo stesso, perché esso è fortemente dipendente sia dall'inviluppo temporale che da quello spettrale simultaneamente.

2.2 Dall'analogico al digitale

Il segnale audio, come anticipato, è la rappresentazione di un suono che ne consente quindi la conservazione su un supporto, la trasmissione o nel nostro caso di interesse l'elaborazione. In primis si effettua la rappresentazione analogica, che approssima le variazioni di ampiezza dell'onda di pressione acustica tramite un'analoga curva continua nel tempo i cui punti sono invece variazioni di tensioni elettrica. Questa seconda curva può essere scritta nel campo magnetico di un nastro oppure all'interno dei solchi di un disco in vinile, riproducendo fedelmente le variazioni di ampiezza del suono. Tuttavia ad ogni operazione di copia si induce un fenomeno di degradazione del segnale, poiché ogni volta è necessario approssimare con una curva continua la stessa che era stata memorizzata in precedenza. Dato che la rappresentazione analogica è soggetta a degradazione nella conservazione a lungo termine e a disturbi nelle trasmissioni, unito al fatto

che le elaborazioni su un segnale audio sono per lo più operazioni aritmetiche e quindi meglio applicabili su segnali discreti[2] si preferisce passare ad una rappresentazione digitale. Essa è una successione finita di numeri che approssima la curva delle variazioni di tensione elettrica; i numeri ottenuti sono chiamati campioni (samples) e vengono rappresentati in forma binaria. Pensando al processo di registrazione e riproduzione che utilizza un computer per l'acquisizione, elaborazione e memorizzazione di segnali audio bisogna tenere a mente lo schema rappresentato in figura 2.3. Il

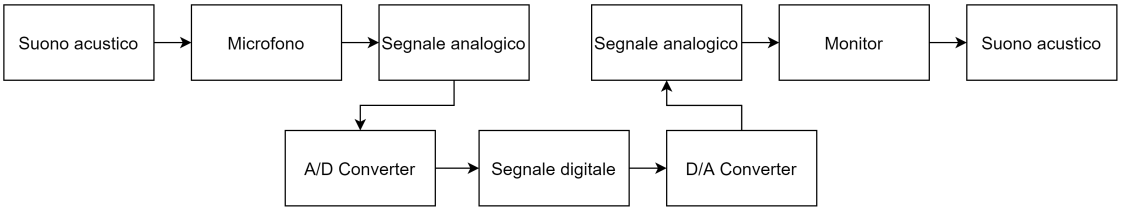

Figura 2.3: Processamento di un suono dalla rappresentazione analogica a quella digitale e viceversa.

microfono è un trasduttore che si occupa della trasformazione di un'onda acustica in un segnale analogico, mentre il convertitore A/D ha il compito di filtrare la banda del segnale analogico, campionare e quantizzare lo stesso, infine effettuarne una codifica. Il convertitore D/A ed il monitor effettuano le stesse operazioni in maniera simmetrica ai due precedenti.

2.2.1 Processo di campionamento

Campionare un segnale analogico significa effettuarne una lettura in istanti temporali ben precisi, equidistanti tra di loro. Da un segnale tempo continuo, si ottiene quindi un segnale tempo discreto, che approssima la forma d'onda analogica tramite campioni letti con una frequenza f_c (espressa in *Hertz*). Considerando T_c il tempo di campionamento, ovvero il tempo che intercorre tra un campione ed il successivo, f_c è così ottenuta:

$$f_c = \frac{1}{T_c} \tag{2.2}$$

[2]Banalmente se si vuole incrementare l'intensità di un suono basta moltiplicare i suoi campioni per il fattore desiderato di amplificazione.

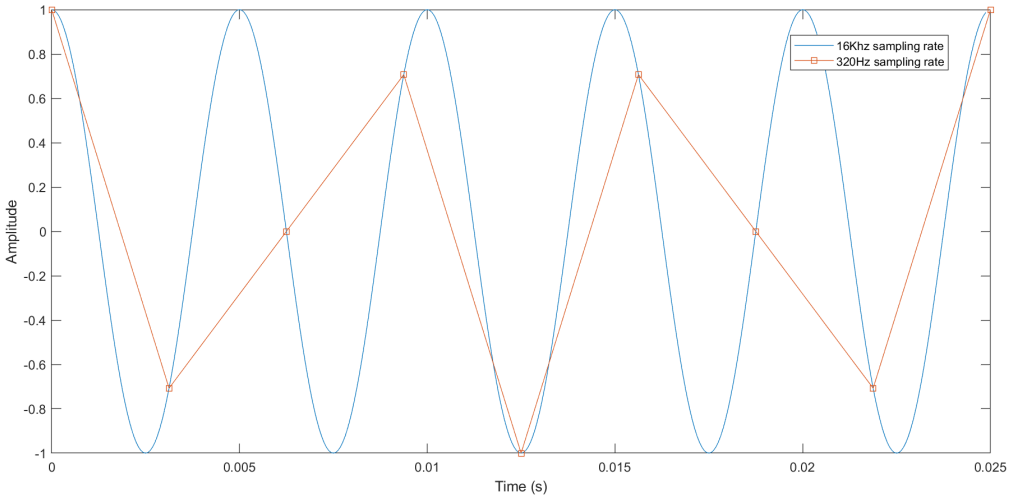

Figura 2.4: Onda sinusoidale campionata con due differenti sampleRate.

In figura 2.4 è mostrata un'onda sinusoidale pura con frequenza $200Hz$, campionata in celeste con $f_c = 16KHz$ ed in arancione con $f_c = 320Hz$. Come è facile notare, nel primo caso la rappresentazione digitale è fedele ed utilizzabile in fase di riproduzione, mentre nel secondo caso è scadente poiché interpolando i vari punti si ricava un'onda che ha un periodo differente dalla curva originale e di conseguenza in fase di riproduzione genererà un tutt'altro suono. Qual è quindi la frequenza di campionamento minima per avere una buona approssimazione del segnale di origine?

La risposta è data dal teorema del campionamento, o di Nyquist–Shannon [3], esprimibile come: $f_c > 2 \cdot f_0$, con f_0 la frequenza massima all'interno del segnale audio. Questo teorema afferma che a partire da un segnale tempo-discreto x[n] è possibile ricostruire un segnale tempo-continuo x(t), se x(t) è a banda limitata e la frequenza di campionamento utilizzata è almeno il doppio della componente in frequenza più elevata presente all'interno del segnale. Se, prima del processo di campionamento, non si filtra il segnale escludendo le frequenze al di sopra di $\frac{f_c}{2}$, come nel caso della curva arancione in figura 2.4, si incorre nel fenomeno dell'aliasing. Esso introduce delle frequenze spurie, non desiderate, non presenti nel segnale analogico di partenza. Volendo digitalizzare un segnale analogico con una componente in frequenza F_1, nel caso di sottocampionamento / aliasing possono verificarsi i seguenti casi:

- se $\frac{f_c}{2} < F_1 < f_c$ sentiremo una componente in frequenza spuria $\frac{f_c}{2} - F_1$, invertita di fase. Questo fenomeno è detto anche foldover, poiché la frequenza rimbalza indietro nello spettro.
- se $F_1 > f_c$ sentiremo la componente in frequenza spuria $F_1 - f_c$, senza inversione di fase.

2.2.2 Processo di quantizzazione

Il segnale analogico può assumere infiniti valori di ampiezza tra il limite massimo e minimo di tensione elettrica, eppure nel mondo digitale non si ha la possibilità di conservare una quantità infinita di informazioni. Pertanto i valori continui di ampiezza, letti alla frequenza di campionamento, vengono approssimati con dei valori discreti, corrispondenti ai livelli del quantizzatore. A partire da un segnale analogico tempo discreto ed ampiezza continua (ottenuto dal processo di campionamento), si ottiene quindi un segnale digitale, tempo e ampiezza discreti. Il numero di livelli

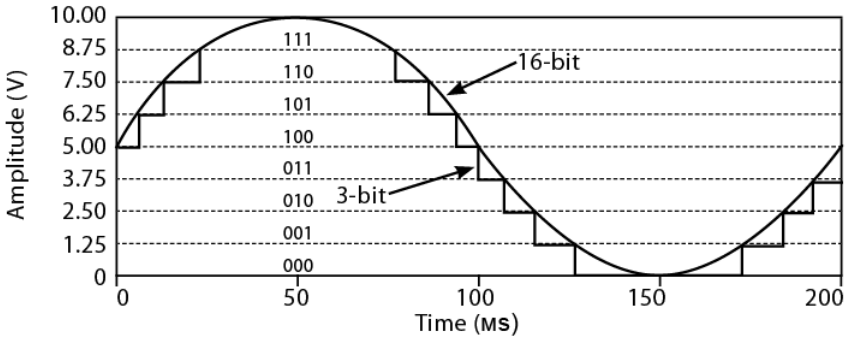

Figura 2.5: Rappresentazione di un segnale quantizzato con risoluzione 3 bit e 16 bit.

in cui è suddiviso il fondo scala del quantizzatore dipende dai bit a disposizione per la codifica. Come mostrato in figura 2.5, con 3 bit, si ottengono 8 livelli ed una scarsa approssimazione del valore di ampiezza, con 16 bit invece una buona approssimazione. In entrambi i casi viene introdotto un errore, detto di quantizzazione o anche rumore di quantizzazione, che è possibile ascoltare sottraendo il segnale ricostruito (dopo il processo di digitalizzazione) al segnale di partenza. Ogni bit in più corrisponde con un raddoppio dei livelli, un aumento del rapporto segnale rumore, e di

conseguenza della gamma dinamica. Infatti considerando una distribuzione uniforme del rumore, ed un quantizzatore con livelli uniformi, dall'SNR così calcolato:

$$SNR = 10 \log_{10} \frac{\sigma_v^2}{\sigma_e^2} \sim 6N - f(\frac{\sigma_v^2}{X_m^2}) \tag{2.3}$$

dove σ_v^2 è la varianza del segnale, σ_e^2 la varianza del rumore, X_m^2 il range del quantizzatore ed N i bit utilizzati, si evince che ad ogni bit in più corrisponde un aumento di guadagno di 6 dB.

Una volta che il segnale è stato convertito in digitale può essere facilmente registrato ed elaborato. Il file che si ottiene può avere caratteristiche differenti a seconda del formato. Lo standard CD (PCM lineare) ad esempio utilizza un sampleRate di 44100 Hz, 16 bit per campione, e due canali (destro e sinistro); volendo fare un rapido calcolo, questo equivale ad uno stream di $44100 \cdot 16 \cdot 2(canali) = 1.41 \frac{Mbit}{s}$.

Capitolo 3

Tecnologie utilizzate

Questo capitolo si concentrerà sulle fasi di studio e ricerca delle tecnologie necessarie al raggiungimento dell'obiettivo finale: un software per l'analisi e manipolazione in tempo reale di uno stream audio. Due saranno gli step fondamentali:

- identificazione dei punti di taglio contestuali all'inizio delle inserzioni pubblicitarie.
- sostituzione degli spot pubblicitari con altri personalizzati per l'utente al fine di una maggiore fidelizzazione.

Flussi audio possono essere manipolati attraverso due tipologie di prodotti finali:

- una audio application standalone, a bassa latenza, che accetti in ingresso un canale stereo e mandi in uscita n-canali processati.
- un plugin audio che possa cooperare con qualsiasi Digital Audio Workstation in commercio.

La scelta è ricaduta sulla seconda soluzione, e le motivazioni verranno spiegate nei paragrafi successivi.

3.1 Digital Audio Workstation

Le Digital Audio Workstation, o semplicemente DAW, nate negli anni '80 dall'unione di un registratore multitraccia ed un mixer, sono sistemi elettronici per la registrazione, monitoraggio e riproduzione dell'audio digitale. Oggi comunemente ci si riferisce al termine non più come sistema

elettronico ma come software dedicato all'editing audio, installabile all'interno di un personal computer, allo stesso modo di un qualsiasi programma di videoscrittura. Tra le potenzialità e funzionalità di questi software, ricordiamo la possibilità di combinare, tagliare ed incollare segnali audio digitali letti direttamente da disco o registrati e campionati tramite l'ausilio di interfacce audio di alta qualità (convertitori analogico-digitale). Come mostrato in figura 3.1, una DAW permette la creazione di più tracce in parallelo, abilitando un controllo individuale su ogni traccia sia per la gestione del volume sia per la gestione di eventuali elaborazioni realtime.

Figura 3.1: Arrangement View del software commerciale Ableton Live.

Le possibilità di elaborazione dei campioni audio diventano innumerevoli se si affida questo compito a del codice esterno. Un flusso audio continuo si presta perfettamente a questo utilizzo poiché i campioni possono essere raggruppati in buffer e affidati ad un altro software secondo le modalità dell'architettura plugin pattern. Produttori possono focalizzarsi in questo modo sullo sviluppo di plugin, siano essi sintetizzatori, equalizzatori, effetti che potranno girare indipendentemente dall'host e dall'ambiente che li ospita.

Il vantaggio di potere applicare un plugin a qualsiasi traccia all'interno di un software di editing che a sua volta può ricevere uno stream audio

attraverso un routing adeguato (tipico scenario nelle postazioni radiofoniche) ha fatto preferire questa soluzione rispetto ad una applicazione standalone. Nel paragrafo 3.2 sarà spiegata la comunicazione tra host e plugin.

3.2 Processamento di un audio plugin

Un host può caricare dinamicamente più istanze di uno stesso plugin, ed a sua volta un plugin può essere host di altri plugin. La comunicazione tra le due parti è descritta nelle API che specificano le funzioni e strutture dati da definire all'interno del codice del plugin, affinché l'host possa istanziarlo correttamente. Il prototipo della principale funzione chiamata dal plugin è di seguito mostrato:

```
1 void process(float *inBuffer, float *outBuffer, int samples)
    ;
```

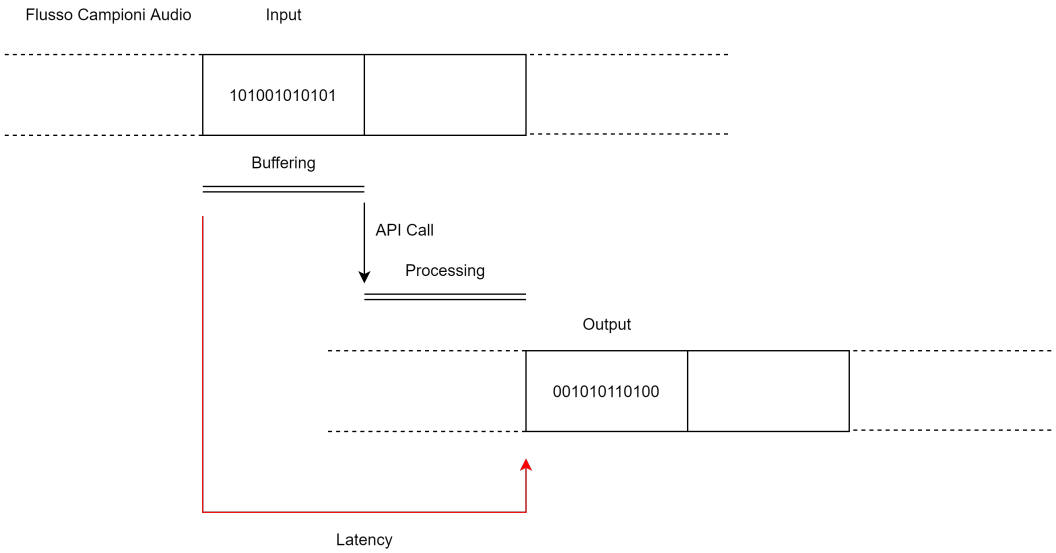

Figura 3.2: Schema di comunicazione tra host e plugin per l'elaborazione di campioni audio.

Questa funzione si occupa dell'elaborazione dei campioni dell'inBuffer, e della scrittura del risultato sull'outBuffer, in alcune implementazioni inBuffer ed outBuffer coincidono. Nello specifico l'host ad ogni ciclo di process, come è possibile notare dalla figura 3.2, prepara l'inBuffer che

andrà processato dal plugin. Quest'ultimo, al termine del processamento, scriverà il frame elaborato sul buffer di uscita. L'host potrà riprendere la riproduzione audio dai samples ricevuti nell'outBuffer e se necessario su di essi effettuare altre operazioni in cascata. Per un corretto funzionamento l'host è anche tenuto a comunicare il samplerate, il numero di canali e la profondità in bit di ogni campione. Il ritardo introdotto da $t_{processing}$ è un parametro critico perché se elevato causa la riproduzione di un salto, ovvero un click udibile. Da qui è possibile comprendere il motivo del perché si utilizzi come linguaggio C/C++ per quasi la totalità di queste tipologie di software: nelle fasi di elaborazione dei buffer si ricercano performance e bassa latenza per soddisfare i requisiti realtime richiesti.

Figura 3.3: Esempio di una GUI con knobs per controllare il processamento in tempo reale effettuato dal plugin.

Le funzionalità introdotte dagli audio plugin includono digital signal processing e sintesi sonora [4]. Solitamente a questa parte di elaborazione è associata una parte grafica che consente all'utente di interagire sul processamento tramite comuni widget quali slider, radio button, knob ecc. associati a specifiche soglie o parametri. Un esempio di audio Plugin GUI è mostrato in figura 3.3.

Per quanto concerne la sintesi sonora, i plugin potrebbero non ricevere in ingresso un buffer di campioni, ma un buffer di note MIDI[1], informazioni digitali che suggeriscono al ricevente il tipo di suono da generare, in

[1] https://www.midi.org/specifications

particolare:

- il pitch: frequenza fondamentale di una nota compresa, in valore numerico, tra 0 e 127.
- la velocity: intensità con cui si sta richiedendo quella nota, se si utilizza una tastiera midi questo valore corrisponde al livello di pressione che si applica sul tasto.
- la durata: un suono termina nel momento in cui il plugin riceve un messaggio di *note off*, oppure di *note on* con *velocity = 0*. L'utilizzo di due messaggi di *note on* consecutivi consente di risparmiare l'invio del secondo header che sarà identico al primo.

Il plugin a partire da queste informazioni sintetizza tramite signal processing un suono direttamente in digitale, ovvero discreto nel tempo. In questo caso il plugin viene comunemente chiamato virtual instrument, poiché va a simulare uno strumento acustico o elettronico. Un maggiore chiarimento su come funzioni il protocollo MIDI per il trasferimento di note in digitale sia monofoniche che polifoniche è spiegato in [5].

3.3 Sviluppo di un audio plugin in MATLAB

Il noto ambiente di sviluppo MATLAB mette a disposizione un add-on chiamato audio ToolBox[2] che fornisce strumenti per signal processing, speech analysis e acoustic measurement. Tra le altre cose l'audio Toolbox integra un tool per la creazione e hosting di audio plugin. L'algoritmo di signal processing deve essere implementato all'interno di una classe che eredita dalla classe base *audioPlugin*, e per esportarlo in un formato standard (VST, AU) sono necessari due passaggi:

- *validateAudioPlugin classname* che effettua una validazione del codice cercando eventuali problemi comuni tramite una test bench. È possibile specificare delle opzioni sfruttando la sintassi *validateAudioPlugin options classname*.
- *generateAudioPlugin className* che genera l'effettivo plugin secondo gli standard VST2 o AU, da utilizzare direttamente all'interno di una

[2] https://it.mathworks.com/help/audio/getting-started-with-audio-system-toolbox.html

DAW. Importante è l'opzione *-juceproject* che consente di generare un file zip contenente codice C++ e un progetto Projucer in modo tale da poter continuare lo sviluppo del plugin tramite il framework JUCE. Quest'ultimo è stato utilizzato per quasi la totalità di questo progetto e rappresenta lo standard nel settore plugin audio.

Dove *validateAudioPlugin* e *generateAudioPlugin* sono funzioni da utilizzare direttamente all'interno della Command Window di MATLAB. Un esempio di semplice plugin in MATLAB è mostrato nello spazio sottostante:

```
classdef stereoWidth < audioPlugin
  properties
    Width = 1;
  end
  properties (Constant)
    PluginInterface = ...
      audioPluginInterface(...
      audioPluginParameter('Width',...
      'Mapping',{'pow',2,0,4}))
  end
  methods
    function out = process(plugin, in)
      mid = (in(:,1) + in(:,2)) / 2;
      side = (in(:,1) - in(:,2)) / 2;
      side = side * plugin.Width;
      out = [mid + side, mid - side];
    end
  end
end
```

Il parametro *Width* viene controllato graficamente dall'utente tramite uno slider generato dalla *audioPluginInterface*, mentre la funzione *process(plugin, in)* è la stessa funzione di callback mostrata nel paragrafo 3.2 dove vengono passati il buffer di campioni in ingresso (per entrambi i canali) e l'oggetto *plugin* per accedere alle proprietà. L'elaborazione audio in questo caso consiste nell'espandere o ridurre l'immagine stereo con una semplice tecnica che prevede la somma e la differenza dei campioni presenti nei due canali destro e sinistro.

3.4 JUCE

JUCE è un framework parzialmente opensource scritto in C++ che offre diversi moduli per lo sviluppo di applicazioni audio desktop e mobile, rilasciato al pubblico per la prima volta nel 2004 dal singolo sviluppatore Jules Storer. In origine, come è possibile intuire dall'acronimo *Jules' Utility Class Extensions*, nasce come insieme di classi a supporto dello sviluppo della DAW alla quale Jules stava lavorando: Tracktion[3] (oggi Waveform). Questo spiega il perché JUCE è particolarmente focalizzato sull'audio, nello specifico fornisce supporto per audio device (CoreAudio, ASIO, ALSA, JACK, WASAPI, DirectSound) e protocollo MIDI, integra reader per la maggior parte delle codifiche audio (come WAV, AIFF, FLAC, MP3, Vorbis) e crea dei wrappers per diverse tipologie di audio plugin, essi siano effetti o virtual instrument.

Juce, tra gli altri vantaggi, è concepito per essere utilizzato allo stesso modo su più piattaforme e compilatori. Le versioni minime delle piattaforme supportate per il deployment sono:

- macOS versione 10.7
- Windows Vista
- Mainstream Linux distributions
- iOS 9.0
- Android Jelly Bean (API 16)

Mentre le versioni minime degli ambienti di sviluppo supportati sono:

- macOS/iOS: macOS 10.11 e Xcode 7.3.1
- Windows 8.1 e Visual Studio 2015 64-bit
- Windows: Linux: GCC 4.8
- iOS: iOS 9.0
- Android: Android Studio su Windows, macOS o Linux

Come molti altri frameworks (Qt, wxWidgets, GTK+ ecc.) JUCE non può mancare nel fornire delle classi per la creazione di elementi della user-interface, per il parsing XML e JSON, ed altre funzionalità per il networking, multi threading, crittografia e supporto video. Lo scopo finale del

[3] https://www.tracktion.com/

progetto JUCE è concentrare tutto ciò di cui lo sviluppatore ha bisogno in una sola libreria, velocizzandone il workflow. JUCE è stato acquisito nel novembre 2014 da ROLI, azienda di software e strumenti musicali con sede a Londra, conosciuta per aver portato alla luce prodotti che reinventano il concetto della tastiera di un pianoforte. Da quel momento ROLI inizia ad utilizzare il framework per lo sviluppo dei nuovi software da abbinare ai loro controller Seaboard, tra i quali Equator. Quest'ultimo, un soft synth e sound engine, come spiegato in [6], è passato da una alpha release ad un prodotto sofisticato in soli quattro mesi grazie al supporto che JUCE con i suoi strumenti già integrati ha saputo offrire.

3.4.1 Projucer

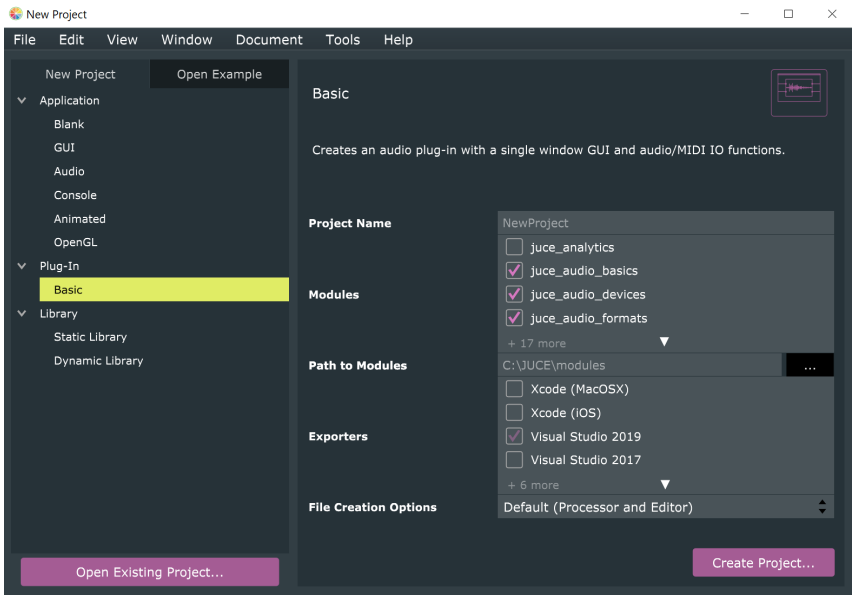

Figura 3.4: GUI di projucer nella selezione di un nuovo progetto.

Il poter compilare lo stesso progetto su IDE, sistemi operativi e compilatori differenti è garantito dal project manager di JUCE, chiamato Projucer. Questo tool consente di selezionare tramite una semplice GUI l'ambiente di sviluppo ed il tipo di prodotto che si vuole creare, occupandosi nella fase di esportazione di includere i file .cpp necessari ad una corretta compilazione nativa sulla piattaforma selezionata. Lo sviluppatore non deve preoccuparsi quindi della compilazione e linking di una libreria per Linux

piuttosto che per Windows prima di iniziare il suo lavoro e può effettivamente concentrarsi sul solo codice dell'applicazione. Come mostrato in 3.4 è possibile scegliere tra:

- un'applicazione con o senza GUI e con o senza parte audio.
- un audio plugin.
- una libreria statica o dinamica.

Al variare di queste tre tipologie, Projucer include i moduli JUCE necessari e tramite la stessa interfaccia sarà possibile aggiungerne altri nelle fasi successive dello sviluppo.

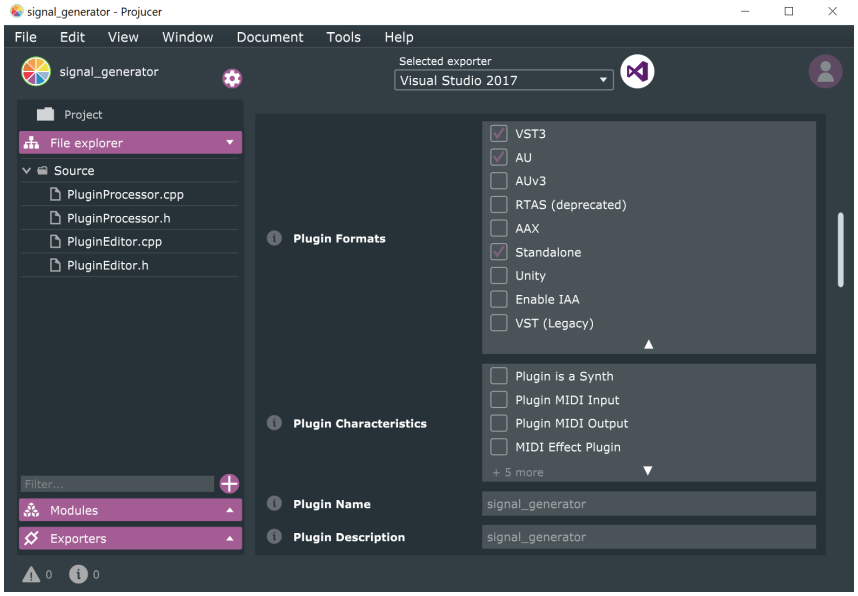

Figura 3.5: GUI di projucer nella fase di esportazione del progetto verso un IDE.

Selezionando ad esempio *audio plugin* nella schermata successiva, mostrata in 3.5, sarà possibile scegliere quale versione di plugin generare. Esistono vari standard e le DAW ne supportano solo alcuni, ad esempio Logic Pro, DAW di Apple, supporta solo plugin proprietari in formato AU, Pro Tools supporta solo il suo standard AAX. Per velocizzare lo sviluppo di un plugin, Projucer crea in automatico due classi:

- pluginEditor che si occupa della user interface del plugin, ereditando dalla classe base juce::AudioProcessorEditor.

- pluginProcessor che si occupa di gestire il thread audio e la sua elaborazione, ereditando dalla classe base juce::AudioProcessor.

Infine Projucer integra anche un code editor se si desidera modificare i file prima dell'esportazione sugli IDE supportati.

3.4.2 Primo plugin in JUCE

Per la realizzazione del software finale del lavoro di questa tesi, è stato necessario implementare un semplice sintetizzatore di onde sinusoidali, per generare toni ad una specifica frequenza. In questo sottoparagrafo sarà spiegato nel dettaglio il plugin che svolge il compito descritto, sfruttandolo come esempio per mostrare quali sono i principali metodi, ereditati dalle due classi AudioProcessorEditor ed AudioProcessor, dei quali fare l'override.

Un sintetizzatore di onde sinusoidali si basa sul segnale:

$$y(t) = A \cdot sin(2\pi ft + \varphi) \tag{3.1}$$

La rappresentazione grafica dell'onda da sintetizzare è mostrata in 3.6.

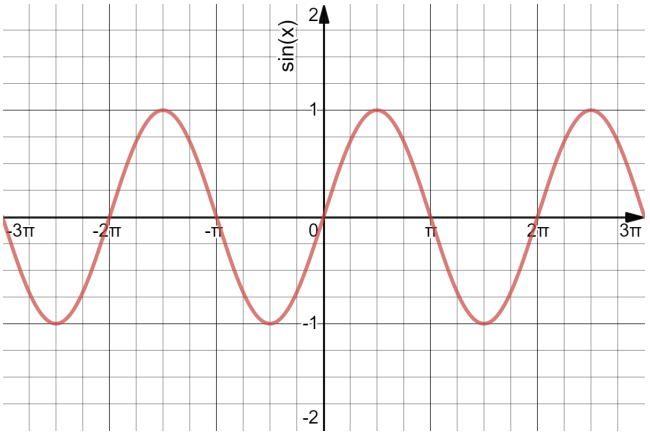

Figura 3.6: Rappresentazione grafica di un'onda sinusoidale con $\omega = 2\pi$ e $\varphi = 0$.

Per poter generare onde sinusoidali che percettivamente suonano in modo diverso, l'utente potrà variare in tempo reale i seguenti parametri:

- Ampiezza (A): andrà ad incidere sulla Sound Pressure Level, e di conseguenza sul volume percepito.

- Frequenza (f): numero di oscillazioni al secondo. Ad un'alta frequenza corrisponde un suono acuto, ad una bassa frequenza corrisponde un suono grave.
- Fase (φ): valore di sfasamento che comporta un ritardo temporale.

Il segnale sintetizzato è discreto nel tempo, quindi è necessario calcolare i valori di t associati ad ogni campione. Il Δt dipenderà dal samplerate dell'audio device in uscita, in particolare ogni campione disterà $1/samplerate$ secondi rispetto al successivo. Ad esempio con un $samplerate = 48000$:

$$\Delta t = \frac{1s}{48000samples} = 2.083 \cdot 10^{-5}s \tag{3.2}$$

Questa operazione può essere eseguita all'interno della funzione *prepareToPlay* dell'*audioProcessor*, chiamata dall'host prima dell'inizio del lavoro effettivo del plugin, affinché possa conoscere il sampleRate della macchina e il numero di samples che riceverà ad ogni ciclo di elaborazione. Solitamente ogni buffer contiene 10 ms di audio, corrispondenti a 480 campioni se si considera un samplerate di 48000 campioni al secondo. In questo caso la *prepareToPlay* è utilizzata anche per inizializzare gli altri parametri:

```
1 void SimpleSignalGeneratorAudioProcessor::prepareToPlay (
     double sampleRate, int samplesPerBlock) {
2   amplitude = 0.5;
3   frequency = 500;
4   phase = 0.0;
5   time = 0.0;
6   deltaTime = 1 / sampleRate;
7 }
```

É possibile che la *prepareToPlay* venga chiamata più volte dall'host, quindi è necessario evitare all'interno funzioni che producano un risultato dipendente da una chiamata precedente. La parte di sintesi vera e propria viene svolta dalla *processBlock* che svolge lo stesso ruolo della callback mostrata in 3.2, riceve un buffer di campioni, di messaggi MIDI e li elabora:

```
1 void SimpleSignalGeneratorAudioProcessor::processBlock (juce
     ::AudioBuffer<float>& buffer, juce::MidiBuffer&
     midiMessages) {
2
3   if (time >= std::numeric_limits<float>::max())
```

```
4       time = 0.0;
5
6     if (mute) {
7       buffer.clear();
8     }
9     else {
10      float *monoBuffer = new float[buffer.getNumSamples()];
11      //generate mono sin wave
12      for (int sample = 0; sample < buffer.getNumSamples(); ++
      sample) {
13          monoBuffer[sample] = amplitude * sin(2 * (double)M_PI
      * frequency * time + phase);
14          time += deltaTime;
15      }
16      //from mono to stereo
17      for (int channel = 0; channel < buffer.getNumChannels();
       ++channel) {
18          buffer.copyFrom(channel, 0, monoBuffer, buffer.
      getNumSamples());
19      }
20      delete[] monoBuffer;
21    }
22  }
```

In questo esempio la *processBlock* sovrascrive in ogni caso il buffer di uscita, con degli zeri se l'utente decide di riprodurre del silenzio, con il segnale sinusoidale altrimenti, copiato in maniera indistinta per i due canali destro e sinistro tramite il metodo *copyFrom* della classe *AudioBuffer* che richiede i seguenti parametri in sequenza: canale destinazione, sample iniziale di destinazione, sorgente di campioni in float, numero di campioni da copiare.

Il controllo dei parametri, nella GUI del plugin, viene gestito dalla classe SimpleSignalGeneratorAudioProcessorEditor che eredita da *AudioProcessorEditor*. Per controllare ampiezza, frequenza e fase è stato scelto l'utilizzo di uno slider, per attivare e disattivare la riproduzione del segnale un textButton. Questi widget sono forniti dal framework, ed è possibile compiere delle azioni in seguito ad una variazione del loro stato, poiché inviano delle notifiche ai listener, nel caso siano stati implementati. Per potere usufruire dei listener, bisogna ereditare dalle classi Slider::Listener e TextButton::Listener, mentre i widget vanno inizializzati nel costruttore in questo modo:

```
1 //freq slider
```

```
addAndMakeVisible(freqSlider);
freqSlider.setRange(10, 22000);
freqSlider.setTextValueSuffix(" Hz");
freqSlider.setValue(500.0);
freqSlider.addListener(this);

freqLabel.setText("Freq", dontSendNotification);
freqLabel.attachToComponent(&freqSlider, true);
```

L'AudioProcessorEditor, implementato da JUCE, riceve nel costruttore by reference l'oggetto AudioProcessor, questo abilita la comunicazione tra le due parti e quindi l'accesso a tutti i campi pubblici di quest'ultimo. Ad esempio tramite il listener associato allo slider della frequenza sarà possibile variare il parametro *frequency*, utilizzato all'interno della formula per la sintesi del segnale nella *processBlock*, in questo modo:

```
void SimpleSignalGeneratorAudioProcessorEditor::
    sliderValueChanged(Slider *slider) {
  if (slider == &freqSlider) {
    audioProcessor.frequency = (float)freqSlider.getValue();
  }
}
```

Svolgendo gli stessi passaggi per gli altri elementi dell'interfaccia si ottiene una semplice UI finale, mostrata in 3.7.

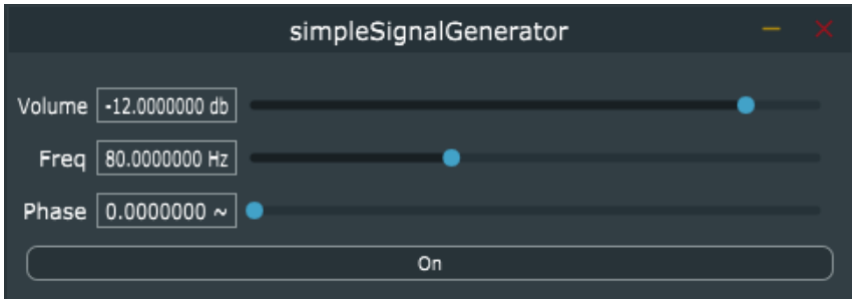

Figura 3.7: User Interface del sintetizzatore di onde sinusoidali.

Capitolo 4

Sviluppo del progetto

In questo ed il successivo capitolo sarà descritto nel dettaglio l'intero lavoro svolto, cercando di non tralasciare gli step evolutivi di progettazione e sviluppo che hanno portato alla realizzazione del prodotto finale. Verrà spiegato ogni componente, la sua funzione all'interno del progetto e come è stato possibile realizzarlo.
L'idea sviluppata può essere schematizzata in linea generale dalla figura 4.1 che illustra lo streaming radiofonico, contenente al suo interno un'inserzione pubblicitaria, ed una pubblicità custom che dovrà sostituire la precedente; i custom Ad risiedono all'interno del filesystem della macchina su cui sta girando il plugin.

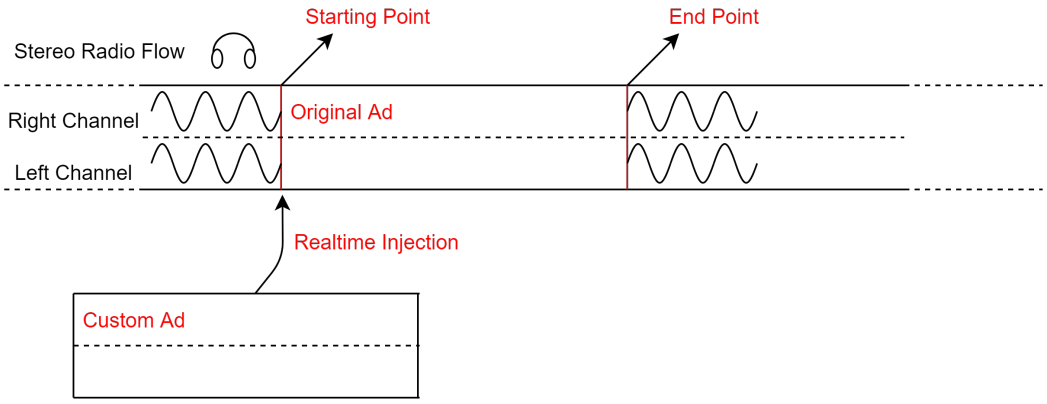

Figura 4.1: Schema di funzionamento dell'injection di un custom Ad.

Il plugin, nel seguente ordine, ha il compito di:

1. riconoscere lo starting point della pubblicità originale, tramite algoritmo di analisi in frequenza.
2. iniziare a sovrascrivere la pubblicità originale con quella custom.
3. continuare ad analizzare in frequenza la pubblicità originale per individuarne l'end point, riproducendo nel frattempo quella custom.
4. riagganciare lo stream iniziale nel preciso punto di end point, e tornare alla ricerca di una nuova pubblicità.

Tutte queste operazioni devono essere svolte in modo trasparente rispetto l'utente in ascolto, tuttavia alcune di esse richiedono una traslazione del punto di riproduzione ad n-campioni nel passato. Affinché l'utente non si accorga di questi movimenti temporali, è stato necessario introdurre un ritardo di riproduzione a monte, sufficiente a coprire lo svolgimento di queste azioni. Il componente che se ne occupa è un delay temporale e sarà il primo ad essere illustrato. Per il riconoscimento effettivo delle pubblicità, invece, sono stati implementati due approcci:

- il primo approccio associa allo starting point ed end point, un tono contenente specifiche frequenze. Esso funge da marcatore durante le fasi di riconoscimento e consente l'implementazione di uno switch on/off. Il componente che si occupa dell'analisi di questi marcatori è un tone analyzer e sarà il secondo componente descritto.
- il secondo approccio, sfrutta la presenza dei jingle, piccoli frammenti musicali, inseriti dalle stazioni radiofoniche per anticipare e concludere momenti pubblicitari. L'individuazione dei jingle è stata effettuata tramite tecniche di fingerprinting audio, in due fasi: conservazione delle caratteristiche offline e riconoscimento live. Questa implementazione rappresenta la parte più corposa di questo lavoro, ed oltre a consentire l'individuazione del punto di inizio della pubblicità, permette di identificare in maniera univoca il jingle riprodotto, allo stesso modo in cui software come Shazam[1] effettuano il riconoscimento delle canzoni. Per queste ragioni è stato dedicato l'intero capitolo 5 alla sua descrizione.

In entrambi i casi, queste soluzioni hanno lo scopo di segnalare la presenza di un punto rilevante all'interno del flusso audio. Le informazioni

[1] https://www.shazam.com/it

così ottenute, saranno passate al componente Ad Injection che si occuperà di attivare la logica di iniezione di uno spot custom.

4.1 Delay temporale

Il compito di questo componente è generare un ritardo temporale. É possibile realizzarlo, come mostra la figura 4.2, ricevendo in ingresso un campione corrispondente ad un tempo t_1, e mandando in uscita, quindi in riproduzione, un campione corrispondente ad un tempo $t_0 = t_1 - \Delta t$.

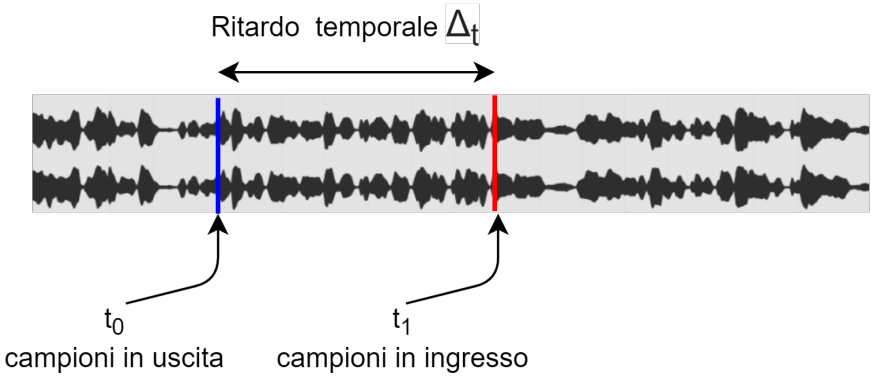

Figura 4.2: Schema di funzionamento di un delay temporale.

Sulla base di questo concetto vengono realizzati, in ambito effettistica, alcuni dei plugin più diffusi tra i quali l'echo ed il chorus, ffTapeDelay[2] ne è un'implementazione di esempio. Visto che nella funzione *processBlock* del plugin vengono passati dall'host 10 ms di audio alla volta, per poter creare un delay nell'ordine dei secondi è necessario conservare i campioni appena ricevuti in una nuova struttura dati e riutilizzarli in un secondo momento. Questo meccanismo è realizzabile utilizzando un buffer circolare, delle dimensioni del ritardo massimo che si vuole generare in campioni. Esso sarà una struttura di appoggio dove scrivere in maniera circolare i campioni in ingresso, e leggerli successivamente con un indice traslato n-campioni indietro nel passato. Il primo passaggio consiste nel copiare i campioni dal mainBuffer al delayBuffer ed aggiornare l'indice di scrittura

[2] https://github.com/ffAudio/ffTapeDelay

del delayBuffer. Questa azione è eseguita dal metodo *fillDelayBuffer* e nella fase di copia possono verificarsi due casi:

- nel primo, mostrato in fig. 4.3, il delayBuffer ha ancora spazio per ospitare nuovi campioni, quindi la copia avviene per intero.

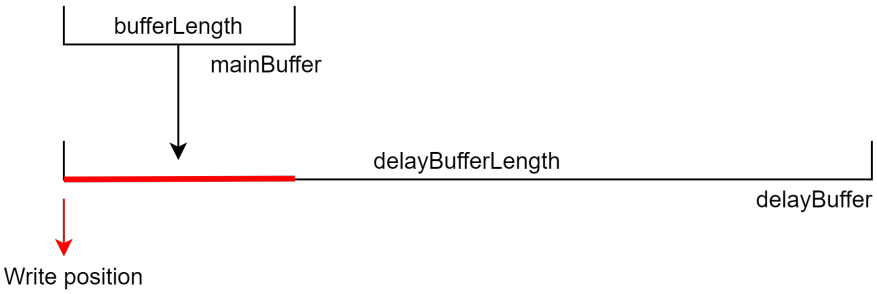

Figura 4.3: *fillDelayBuffer* nel caso in cui il buffer è copiato per intero.

- nel secondo, mostrato in fig. 4.4, l'indice di scrittura del delayBuffer è quasi alla fine di esso, quindi il buffer da copiare sarà spezzato in due frammenti. Uno inserito dall'indice di scrittura alla terminazione del delayBuffer, ed il secondo all'inizio del delayBuffer, creando a tutti gli effetti una scrittura circolare continua.

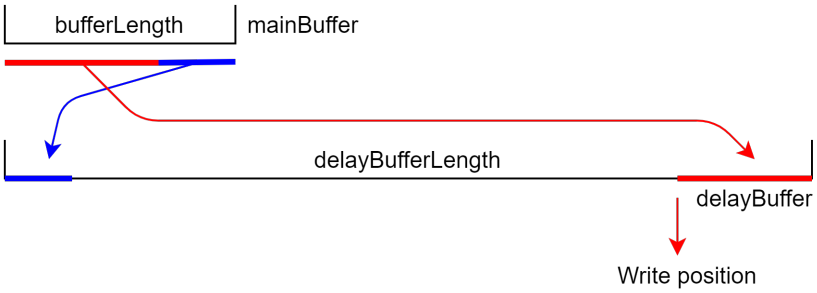

Figura 4.4: *fillDelayBuffer* nel caso in cui il buffer è spezzato in due frammenti.

Di seguito è mostrato il codice che si occupa di questa operazione:

```
1 void fillDelayBuffer(int channel, const int bufferLength,
    const int delayBufferLength, const float* bufferData,
    const float* delayBufferData) {
2
3   if (delayBufferLength > bufferLength + mWritePosition) {
```

```
4       mDelayBuffer.copyFrom(channel, mWritePosition,
        bufferData, bufferLength);
5     }
6     else {
7       const int bufferRemaining = delayBufferLength -
        mWritePosition;
8       mDelayBuffer.copyFrom(channel, mWritePosition,
        bufferData, bufferRemaining);
9       mDelayBuffer.copyFrom(channel, 0, bufferData +
        bufferRemaining, bufferLength - bufferRemaining);
10    }
11  }
```

Il secondo passaggio consiste nell'aggiornare l'indice di lettura del delayBuffer ad n-campioni indietro nel passato e copiare una quantità di campioni, dal delayBuffer al mainBuffer, pari alla dimensione del mainBuffer. Quest'operazione è svolta dalla *getFromDelayBuffer* e come per la *fillDelayBuffer* vi sono due casi:

- il primo, illustrato in fig. 4.5, in cui l'indice di lettura non è alla terminazione del delayBuffer e vengono quindi copiati un numero di campioni contigui dal delayBuffer al mainBuffer.

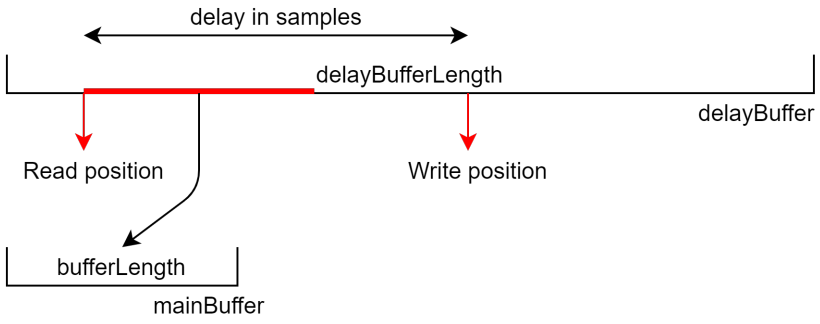

Figura 4.5: *getFromDelayBuffer* nel caso di una copia contigua.

- il secondo, illustrato in fig. 4.6, in cui l'indice di lettura è alla terminazione del delayBuffer. Verrà copiato nel mainBuffer un frammento di campioni, dall'indice di lettura alla terminazione del DelayBuffer, ed un frammento dall'inizio del delayBuffer.

Il codice della *getFromDelayBuffer* è così costituito:

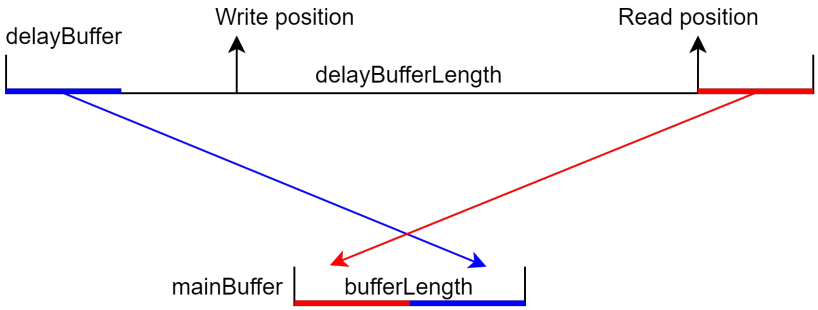

Figura 4.6: *getFromDelayBuffer* nel caso in cui l'indice di lettura è alla terminazione del delayBuffer.

```
1  void getFromDelayBuffer(AudioBuffer<float>& buffer, int
       channel, const int bufferLength, const int
       delayBufferLength, const float* bufferData, const float*
       delayBufferData) {
2    //delayTime -> ritardo in millisecondi
3    delayInSamples = mSampleRate * delayTime / 1000;
4    const int readPosition = static_cast<int> (
       delayBufferLength + mWritePosition - delayInSamples) %
       delayBufferLength;
5    if (delayBufferLength > bufferLength + readPosition) {
6      buffer.copyFrom(channel, 0, delayBufferData +
       readPosition, bufferLength);
7    }
8    else {
9      const int bufferRemaining = delayBufferLength -
       readPosition;
10     buffer.copyFrom(channel, 0, delayBufferData +
       readPosition, bufferRemaining);
11     buffer.copyFrom(channel, bufferRemaining,
       delayBufferData, bufferLength - bufferRemaining);
12   }
13 }
```

Importante notare a riga numero 4 che per ottenere un indice di lettura $mReadPosition = mWritePosition - delayInSamples$ che stia sempre nel range [0, delayBufferLength] si somma la lunghezza delayBufferLength e se ne fa il modulo con la stessa. Questo consente inoltre di riprodurre del silenzio all'inizio dello stream quando ancora il delayBuffer contiene degli zeri.

Le due funzioni, corrispondenti ai due passaggi di un delayer, sono così

chiamate dalla *processBlock* del plugin:

```
void processBlock(juce::AudioBuffer<float>& buffer, juce::
    MidiBuffer& midiMessages) {
  ...
  for (int channel = 0; channel < totalNumInputChannels; ++
    channel) {
     const float* bufferData = buffer.getReadPointer(channel)
    ;
     const float* delayBufferData = mDelayBuffer.
    getReadPointer(channel);
     //delay actions
     fillDelayBuffer(channel, bufferLength, delayBufferLength
    , bufferData, delayBufferData);
     getFromDelayBuffer(buffer, channel, bufferLength,
    delayBufferLength, bufferData, delayBufferData);
  }
  mWritePosition += bufferLength;
  mWritePosition %= delayBufferLength;
}
```

Anche qui l'indice di scrittura $mWritePosition$ viene incrementato della lunghezza bufferLength, ma rimane sempre confinato nel range [0, delayBufferLength].

4.2 Tone analyzer

Un tono, nel temperamento equabile, è un intervallo di seconda maggiore, nel nostro caso è inteso come suono a specifiche frequenze. Per la sintesi di toni a singola frequenza è stato utilizzato il tool descritto nel paragrafo 3.4.2, mentre per la somma di essi è stata utilizzata la DAW Ableton Live. Il loro riconoscimento è stato effettuato in una prima implementazione tramite analisi dello spettro del segnale, ottenuto applicando la trasformata di Fourier, ed in una seconda implementazione più efficiente tramite analisi di singole frequenze dello spettro, applicando l'algoritmo di Goertzel.

Le due tipologie di toni utilizzate per testare il funzionamento sono:

- toni udibili DTMF (Dual-tone multi-frequency signaling). Ogni tono è ottenuto come somma di due segnali a singole frequenze distanti tra loro, come mostrato in tabella 4.1. Si tratta di combinazioni non riprodotte in musica perché non fanno parte del temperamento equabile

e non armoniche. Storicamente erano utilizzati nei telefoni analogici per comunicare il tasto premuto alla centrale, ogni tasto corrispondeva ad un tono e veniva riconosciuto tramite algoritmo di Goertzel.

	1209 Hz	**1336 Hz**	**1477 Hz**	**1633 Hz**
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Tabella 4.1: Frequenze utilizzate da un tastierino DTMF.

- toni non udibili, in particolare nelle basse frequenze, da 16 Hz. Questa soluzione, ha dalla sua parte il vantaggio di non alterare lo stream audio dal punto di vista percettivo, poiché sono vibrazioni al di fuori del range di udibilità. In figura 4.7 è mostrato un confronto di una sezione con e senza tono da 16 Hz annegato all'interno, nella quale è possibile apprezzare la variazione dell'inviluppo temporale. Per captare la variazione dei 16 Hz lo svantaggio, tuttavia, è quello di dovere utilizzare una finestra temporale, da analizzare in frequenza, più grande rispetto ai toni udibili; il che comporta l'utilizzo di una trasformata di ordine superiore ed un carico di lavoro maggiore.

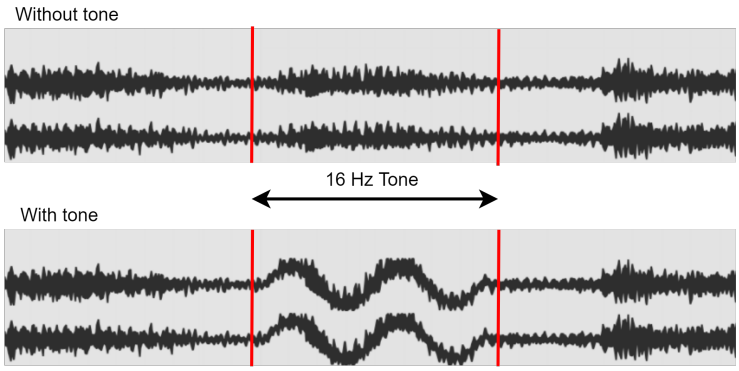

Figura 4.7: Inviluppo temporale di una sezione audio con e senza la presenza di un tono da 16 Hz all'interno.

4.2.1 Riconoscimento tramite trasformata di Fourier

La trasformata di Fourier:

$$X^F(\omega) = \int_{-\infty}^{+\infty} x(t) \cdot e^{-j\omega t}\, dt, \quad \omega \in \mathbb{R} \tag{4.1}$$

è un operatore che trasforma una funzione in un'altra funzione, nello specifico effettua il passaggio dal dominio del tempo a quello delle frequenze e viceversa, ed è:

- continua nel tempo.
- continua in frequenza.
- con supporto infinito in tempo e in frequenza.

Nel nostro caso di studio, il segnale in ingresso è una successione di campioni finita, si applica quindi la sua versione discreta DFT (Discrete Fourier Transform):

$$X[k] = \sum_{n=0}^{L-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}, \quad 0 \leq k \leq N-1 \tag{4.2}$$

che è rispettivamente:

- discreta nel tempo [n].
- discreta in frequenza [k].
- con supporto limitato in tempo ed in frequenza [L, 2π].

Se si utilizza un supporto temporale di pochi campioni, si avrà una maggiore precisione nel tempo ed una minore precisione in frequenza, viceversa se il supporto temporale è grande si avrà una maggiore precisione in frequenza ed una minore precisione temporale. In [7] vi è un esempio di algoritmo DFT scritto in c, ottenuto tramite diretta applicazione della formula:

```
void dft(float *in, float *out, int N) {
  for (int i = 0, k = 0; k < N; i += 2, k++) {
    out[i] = out[i + 1] = 0.0;
    for (int n = 0; n < N; n++) {
      out[i] += in[n] * cos(k*n*PI2 / N);
      out[i + 1] -= in[n] * sin(k*n*PI2 / N);
    }
    out[i] /= N; out[i + 1] /= N;
  }
}
```

Essendo composto da due cicli annidati che vanno da 0 ad N, questo algoritmo ha un costo computazionale $O(N^2)$. Per ottenere lo stesso risultato, si preferisce utilizzare un algoritmo più veloce che richiede un numero di operazioni inferiore $O(N \log(N))$, chiamato FFT, dall'inglese Fast Fourier Transform. La sua prima implementazione del 1965 sviluppata da Cooley-Tukey [8], ed anche la più diffusa, è ottimizzata per dimensioni di N che siano potenze di due; [9], [10] e [11] descrivono vari metodi per costruire efficienti algoritmi FFT e ne spiegano i relativi vantaggi.
Per lo sviluppo del componente tone analyzer e per il calcolo delle trasformate in tutte le altre parti del progetto è stata utilizzata la classe dsp::FFT di JUCE, fornita nel modulo dsp, che contiene i metodi:

- perform(), per calcolare una FFT, sia diretta che inversa, su un vettore di numeri complessi.
- performRealOnlyForwardTransform(), per calcolare una FFT diretta su un vettore di numeri reali. L'array di uscita conterrà in modo intervallato parte reale e immaginaria dei numeri complessi ottenuti.
- performRealOnlyInverseTransform(), per calcolare una FFT inversa allo stesso modo di performRealOnlyForwardTransform().
- performFrequencyOnlyForwardTransform(), per calcolare una FFT diretta ed ottenere direttamente i valori di intensità normalizzati per ogni bin di frequenze. TheFastFourierTransform[3] è un esempio di utilizzo della performFrequencyOnlyForwardTransform() all'interno di una audio application.

Dopo aver introdotto gli strumenti essenziali per ottenere lo spettro di un segnale audio discreto nel tempo, sono spiegati di seguito i passaggi realizzati all'interno del plugin per la rilevazione di un tono tramite FFT:

1. Ad ogni ciclo di *processBlock* vengono copiati i campioni in una struttura FIFO, di dimensione N pari al numero di punti della FFT (in JUCE N deve essere un multiplo di 2, poiché si basa sull'FFT di Cooley-Tukey).
2. Nel momento in cui l'array FIFO contiene N campioni, si calcola su di essi la performFrequencyOnlyForwardTransform().

[3] https://docs.juce.com/master/tutorial_simple_fft.html

3. Nell'array ottenuto dalla performFrequencyOnlyForwardTransform() si cerca il bin dove all'interno è presente la frequenza del tono di nostro interesse, e se ne valuta il valore di intensità corrispondente. Essendo l'FFT una trasformata discreta, restituisce un valore di intensità e fase per range di frequenze, chiamati bin, più o meno grandi in base al numero di punti utilizzati[4]. Il bin da cercare è così calcolato:

$$bin = freqTone \cdot \frac{N}{sampleRate} \tag{4.3}$$

Quindi utilizzando un tono da 440 Hz, una trasformata da 1024 punti ed un sampleRate di 48 KHz:

$$bin = 440Hz \cdot \frac{1024}{48000Hz} = 9.386 \tag{4.4}$$

bisognerà controllare il valore contenuto nel nono elemento dell'array ottenuto dalla performFrequencyOnlyForwardTransform().

4. Se l'intensità per quel bin è elevata, è garantita la presenza di quel tono in quell'istante di N campioni su cui è stata calcolata la trasformata. Di seguito il plugin può azionare la logica per l'injection pubblicitaria.

4.2.2 Riconoscimento tramite algoritmo di Goertzel

La Fast Fourier Transform elabora tutte le componenti in frequenza del segnale, tuttavia per il riconoscimento del singolo tono andiamo ad estrapolarne una sola parte, scartando le rimanenti. Questo è uno spreco computazionale non indifferente, che può essere colmato utilizzando non la generale FFT, ma uno strumento più adatto a questo use case: l'algoritmo di Goertzel. Esso consente di calcolare la DFT in uno o specifici punti, riducendo drasticamente il calcolo computazione rispetto ad una FFT dove soltanto un bin è analizzato, in [12] e [13] vi sono maggiori chiarimenti sulle implementazioni. L'algoritmo per eseguire il calcolo, utilizzato all'interno di questo lavoro, è il seguente:

```
1 float goertzel(int N, const float *data, int sampleRate, int
      fqRequired) {
```

[4]Ad esempio, con un sampleRate di 48 KHz ed una trasformata da 512 punti si ottengono bin da 93.75 Hz.

```
  float omega = static_cast<float> (PI2 * fqRequired /
   sampleRate);
  float sine = sin(omega);
  float cosine = cos(omega);
  float coeff = cosine * 2;
  float q0 = 0; float q1 = 0; float q2 = 0;
  for (int i = 0; i < N; i++) {
    q0 = coeff * q1 - q2 + data[i];
    q2 = q1;
    q1 = q0;
  }
  float real = (q1 - q2 * cosine) / (N / 2.0);
  float imag = (q2 * sine) / (N / 2.0);
  return 100 * sqrt(real * real + imag * imag);
}
```

Dove N è il numero di punti della DFT, *data* è il vettore di campioni ed *fqRequired* il punto della DFT da calcolare. La funzione ritorna il valore di magnitudine e tramite confronto con una soglia, sarà possibile dedurre se la frequenza *fqRequired* è presente o meno. Come nell'implementazione tramite FFT, all'interno della funzione pushSampleIntoFifo(), ad ogni ciclo di processBlock del plugin vengono copiati i campioni in una struttura FIFO:

```
void pushSampleIntoFifo(const float& sample) {
  if (fifoIndex == N) {
    memcpy(fftData.data(), fifo.data(), sizeof(fifo));
    window.multiplyWithWindowingTable(fftData.data(),
   GoertzelSize);

    detectGoertzelFrequencies(GoertzelSize, fftData.data());
    fifoIndex = 0;
  }
  fifo[fifoIndex] = sample;
  ++fifoIndex;
}
```

Al raggiungimento della dimensione N:

1. la *pushSampleIntoFifo()* prepara l'array di campioni da passare alla *detectGoertzelFrequencies()* per il calcolo dell'algoritmo Goertzel su tutte le frequenze di interesse contenute all'interno di un array *fq[]*.
2. i valori restituiti dall'algoritmo sono salvati in *fq_levels[]* e se ne affida il controllo, per il confronto con la soglia, alla funzione *checkGoertzelFrequencies()*.

3. nel caso del tono da 16 Hz, viene controllata la presenza di una singola frequenza, nel caso di un tono DTMF la presenza simultanea delle due frequenze di una delle combinazioni mostrate in tabella 4.1.

Setup della soglia di riconoscimento

Considerato che l'ampiezza di un'onda sinusoidale è ottenibile dal valore SPL nel seguente modo:

$$amplitude = 10^{\frac{ans(dB)}{20}} \tag{4.5}$$

Un segnale sinusoidale a -3 dBFS ha un'ampiezza di 0.70, a -6 dBFS un'ampiezza di 0.50, a -9 dBFS un'ampiezza di 0.354, a -12 dBFS un'ampiezza di 0.251. La soglia che decreta il riconoscimento di un tono è stata settata dopo opportune verifiche al variare dei parametri di ampiezza descritti. L'algoritmo di Goertzel restituisce la magnitudine di una frequenza in valore compreso tra 0 ed 1, per comodità questo è stato riscalato in percentuale. I test effettuati con Goertzel settato per l'individuazione di un suono a 16 Hz annegato nel flusso radiofonico, considerando anche il caso di sua assenza, sono riportati in tabella 4.2. É possibile osservare che il flusso radiofonico non disturba in alcun modo il riconoscimento del tono non udibile. Si è scelto quindi di utilizzare toni ad intensità -6 dBFS e di settare la soglia di riconoscimento a 35, in modo tale che tutti i suoni a 16 Hz con magnitudine al di sopra del 35% possano essere individuati come punti rilevanti per la sostituzione pubblicitaria. Utilizzando toni fino a 10 Hz, i valori di magnitudine sono del tutto simili al tono da 16 Hz.

Intensità Tono 16 Hz	Assente	-3 dBFS	-6 dBFS	-9 dBFS	-12 dBFS
Magnitude Goertzel	0.8%	70.4%	50.1%	35.5%	25.1%

Tabella 4.2: Magnitudine in percentuale, restituita dall'algoritmo di Goertzel, al variare dell'intensità dello stesso tono a 16 Hz.

La soglia settata con questo valore non ha generato falsi positivi, ed ha consentito sempre il riconoscimento di toni a 16 Hz, -6 dBFS all'interno

di una qualsiasi sezione del flusso radiofonico. Per toni udibili (quindi anche DTMF) invece i valori di magnitudine restituiti sono inferiori e più instabili, come è possibile notare in tabella 4.3 nel caso di un tono a 1209 Hz. Per il riconoscimento di questi suoni è necessario utilizzare una soglia di riconoscimento dimezzata rispetto ai toni non udibili, e questo insieme al conflitto delle stesse frequenze presenti nello stream radiofonico può causare dei falsi positivi. Dati gli svantaggi dei toni DTMF in questo contesto di utilizzo, per la versione finale del software che implementa il riconoscimento di marcatori audio, è stato scelto l'impiego di soli toni non udibili a 16 Hz.

Intensità Tono 1209 Hz	Assente	-3 dBFS	-6 dBFS	-9 dBFS	-12 dBFS
Magnitude Goertzel	Variabile	32.65% - 70%	22.7% - 49.8%	15.49% - 36.7%	11.6% - 24.5%

Tabella 4.3: Magnitudine in percentuale, restituita dall'algoritmo di Goertzel, al variare dell'intensità dello stesso tono a 1209 Hz.

4.3 Audio Injection: implementazione con Goertzel

Il componente Audio Injection, nel caso dell'implementazione tramite algoritmo di Goertzel, effettua un test continuo sul flusso audio per verificare la presenza di un nuovo tono di interesse. Nello specifico richiama ad ogni ciclo di processBlock() la funzione **checkGoertzelFrequencies**:

- se questa restituisce true, allora lo stato della injection passa da On a Off e viceversa.
- se questa restituisce false, lo stato rimane invariato.

Sia nel caso di apertura che di chiusura dello spot pubblicitario, viene ricercata la presenza dello stesso tono. Bisogna tenere conto però che, dopo il riconoscimento del primo, il controllo **checkGoertzelFrequencies** viene disabilito per un tempo prestabilito pari alla durata massima del tono stesso, cioè se ne attende una terminazione certa. Questo perché

un controllo immediatamente successivo ad un primo riconoscimento, andrebbe ad analizzare ancora lo stesso tono causando una variazione dello stato di injection errata. La figura 4.8 mostra un caso reale testato, nello specifico i due toni di interesse individuati da finestre in rosso, e l'inserzione pubblicitaria racchiusa all'interno di essi, mentre in alto ciò che viene restituito dalla funzione di controllo, che causa una variazione dello stato, illustrato in basso.

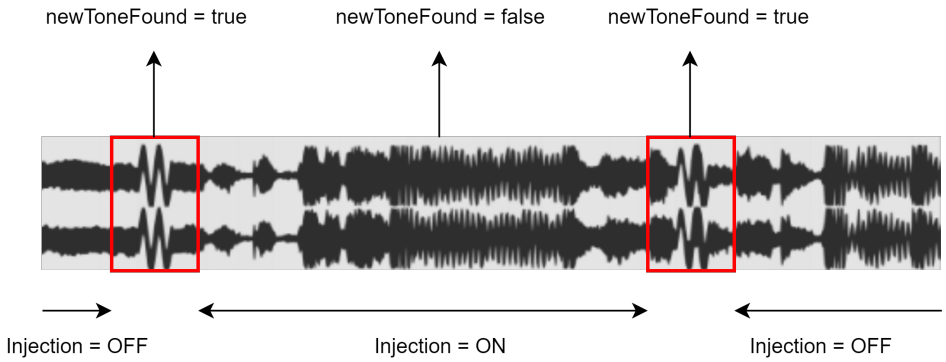

Figura 4.8: Schema della logica di injection, nel caso di riconoscimento di toni a 16 Hz tramite algoritmo di Goertzel.

La variazione dello stato può essere schematizzata con un automa a stati finiti, riportato in figura 4.9. Durante lo stato OFF, il componente Audio Injection sostituirà il buffer in uscita del plugin con lo stream audio in ingresso, nel caso dello stato ON leggerà dal filesystem uno spot pubblicitario custom (facendo una decodifica se necessario) e lo copierà a blocchi nel buffer in uscita, continuando ad effettuare l'analisi sui campioni in ingresso, fino al successivo stato di OFF. Il modo in cui è stato letto l'audio file è simile a quanto sarà descritto nel paragrafo 5.1.1.

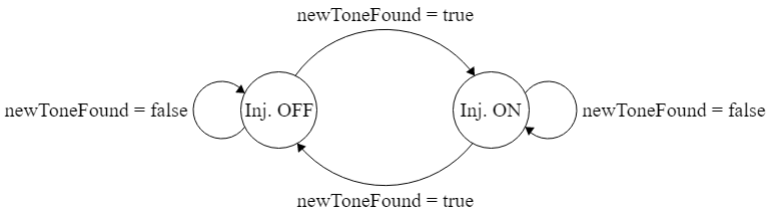

Figura 4.9: Automa a stati finiti della logica di injection al riconoscimento di toni tramite algoritmo di Goertzel.

Capitolo 5

Audio Fingerprint

L'audio fingerprint è un riassunto digitale di un segnale audio, che consente la sua univoca identificazione. A partire da un segmento, che sia percettivamente simile all'originale, non per forza coincidente bit per bit, l'audio fingerprint deve consentire il match con la traccia dal quale è stato estrapolato, tra diverse presenti all'interno di un database. Gli usi pratici di questo strumento vanno dal riconoscimento tramite smartphone di un brano musicale per soddisfare la curiosità di un utente nell'avere informazioni dettagliate (titolo, artista, album, data di uscita ecc.) su ciò che sta ascoltando, al riconoscimento dell'utilizzo illecito di contenuti multimediali, in mancanza di licenze o nel caso di violazioni di copyright. Servizi come Audioneex[1] ed Acoustid[2] si concentrano sul miglioramento continuo delle tecniche di audio recognition [14] [15] [16], fornendo degli strumenti, accessibili tramite API, a tutte le aziende che desiderano aggiungere delle orecchie ai loro applicativi.

Per questo lavoro è stata implementata una tecnica di fingerprint per il riconoscimento dei jingle pubblicitari al fine di:

- individuare il punto esatto di terminazione del jingle di apertura, e di conseguenza lo starting point dell'inserzione pubblicitaria.
- individuare il punto esatto di inizio del jingle di chiusura, e di conseguenza l'end point dell'inserzione pubblicitaria in modo tale da riagganciare lo stream radio.

[1] https://www.audioneex.com/

[2] https://acoustid.org/

- riconoscere i jingle in maniera univoca tra quelli più utilizzati da una stazione radiofonica.

Per confrontare i fingerprint di due file audio al fine di verificarne la loro somiglianza, è necessario effettuare lo stesso tipo di analisi in due fasi separate:

- una fase offline in cui si analizzano in frequenza tutti i jingle e se ne estraggono le feature da memorizzare in una struttura di supporto: file o database.
- una fase online in cui si effettua lo stesso tipo di analisi ma su blocchi dello streaming audio, confrontando le feature estrapolate da ognuno di essi con quelle memorizzate offline nella precedente fase.

Gli algoritmi di audio fingerprint sono un trade-off tra velocità e robustezza nel riconoscimento [17]. Una maggiore robustezza significa evitare falsi positivi o il non riconoscimento, ed un miglioramento di questo parametro solitamente richiede una finestra temporale maggiore di analisi, ovvero un peggioramento in velocità. Nel caso di questo lavoro, dato che:

- il segnale da analizzare viene ricevuto all'interno di una DAW e non presenta disturbi o sovrapposizioni di altro genere.
- il numero totale dei jingle utilizzati dalle radio non è dello stesso ordine di grandezza di una raccolta musicale globale.
- i jingle da riconoscere hanno una durata pressoché breve, tra i due ed i tre secondi.

si è preferito settare l'algoritmo in modo tale da favorire la velocità nel riconoscimento.

5.1 Analisi offline

Nella fase offline, tutti gli audio files di interesse devono essere schedati all'interno di un supporto riutilizzabile nella fase successiva (riconoscimento in tempo reale). Questo processo svolge i seguenti passaggi:

1. lettura degli audio files; operazione non banale se consideriamo la quantità di formati audio lossy e lossless esistente.
2. calcolo di una FFT al variare dei frame temporali, per ottenere uno spettrogramma del segnale.

3. determinazione dei punti chiave dello spettrogramma, che andranno associati ad un hash.
4. associazione dell'hash all'id del jingle in una hashmap.
5. scrittura della hashmap ed altre informazioni riguardanti i jingle su disco.

L'insieme di queste operazioni è stato implementato all'interno di una desktop application standalone, indipendente rispetto al plugin che si occuperà dell'analisi in tempo reale.

5.1.1 Lettura di un audio file

Oltre al formato standard PCM non compresso, nel corso degli anni sono nati diversi codificatori audio (ricordiamo tra i lossy l'MP3, tra i loseless il FLAC) con lo scopo di ridurre la dimensione dei file. JUCE, all'interno del modulo juce_audio_formats, si occupa della gestione dei diversi standard e fornisce delle API di alto livello per la lettura e scrittura di un generico audio, indipendentemente dal suo formato.

Tramite classe *AudioFormatManager* è possibile ottenere un oggetto di tipo *AudioFormatReader* che si occupa della lettura dei campioni e di effettuarne una copia all'interno di un buffer di supporto. Il codice che segue, mostra le operazioni da effettuare per la lettura di un generico audio file:

```
File file('example.wav');
AudioFormatManager formatManager;
formatManager.registerBasicFormats();
std::unique_ptr<juce::AudioFormatReader> reader(
   formatManager.createReaderFor(file));

if (reader.get() != nullptr) {
  AudioBuffer<float> fileBuffer;
  fileBuffer.setSize((int)reader->numChannels, (int)reader->
   lengthInSamples);
  reader->read(&fileBuffer, 0, (int) reader->LengthInSamples
   , 0, true, true);
}
```

Nello specifico, la *registerBasicFormats()* consente di registrare i formati più comuni (come ad esempio WAV, AIFF, Ogg Vorbis e così via), mentre la funzione *read()* gestisce le operazioni di lettura di basso livello caricando

in memoria l'intero audio file all'interno del buffer *fileBuffer* come array di float. Una sezione del buffer ottenuto tramite esecuzione del codice descritto è mostrata in figura 5.1.

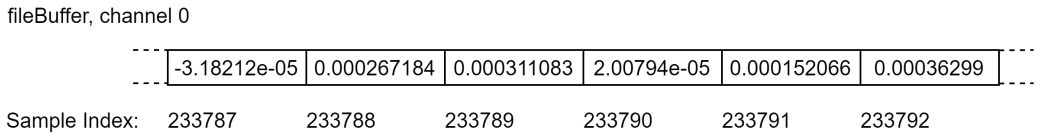

Figura 5.1: Sezione di un audioBuffer, canale 0, dopo lettura tramite *AudioFormatReader*.

Il sampleRate originale dell'audio file potrebbe non coincidere con quello utilizzato dalla DAW, e quindi dal plugin nella fase live di riconoscimento. Questo porterebbe a non ottenere alcuna tipologia di match anche se percettivamente il segnale audio fosse lo stesso. Per questo motivo, il tool sviluppato che si occupa nella fase offline di creare i file con gli hash di tutti i jingle, consente all'utente tramite combobox, come mostrato in figura 5.2, di selezione il sampleRate da utilizzare in uscita, cioè consente il resampling degli audio file letti.

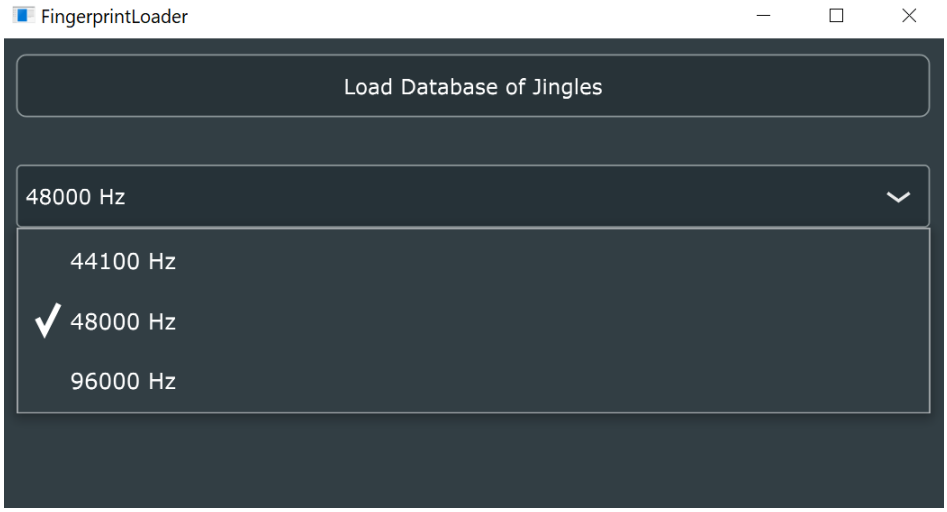

Figura 5.2: Fingerprint Loader, scelta del sampleRate, ai fini del resampling.

Nella stragrande maggioranza dei casi, gli audio file letti da disco hanno un sampleRate di 44100 Hz, per via dello standard CD [18], mentre le

DAW solitamente lavorano con un sampleRate di 48000 Hz. Trasformare un audioBuffer da 44100 Hz a 48000 Hz, significa effettuare un upsampling, e di conseguenza una interpolazione. JUCE mette a disposizione un Lagrange Interpolator, ed è stato sfruttato inizialmente all'interno della seguente funzione, che effettua la conversione di un buffer ad un nuovo sampleRate:

```
1  void resampleAudioBuffer(AudioBuffer<float>& buffer,
       unsigned int& numChannels, int64& samples, double&
       sampleRateIn, double& sampleRateOut) {
2    if (sampleRateIn != sampleRateOut) {
3      AudioBuffer<float> resampledBuffer;
4      double ratio = sampleRateIn / sampleRateOut;
5      resampledBuffer.setSize((int)numChannels, (int)(((double
       )samples) / ratio));
6      LagrangeInterpolator resampler;
7      resampler.reset();
8      resampler.process(ratio, buffer.getReadPointer(0),
       resampledBuffer.getWritePointer(0), resampledBuffer.
       getNumSamples());
9      buffer = std::move(resampledBuffer);
10   }
11 }
```

Il processo di resampling del Lagrange Interpolator richiede:

- una ratio, calcolata come:

$$speedRatio = \frac{sampleRateIn}{sampleRateOut} \tag{5.1}$$

- il buffer in ingresso.
- il buffer in uscita.
- il numero di output sample da produrre.

Tuttavia, visto che il processo di analisi sui campioni sarà effettuato frame dopo frame, all'interno di un ciclo, non si ha la necessità di tenere in memoria l'intero audio file. Si è preferito virare, quindi, su un approccio più efficiente che carica in memoria un blocco di campioni per volta, basato sull'utilizzo della classe *AudioFormatReaderSource*. Questa consente di leggere file audio a partire da un oggetto *AudioFormatRead* e di renderizzarne un singolo blocco (delle dimensioni desiderate) tramite la funzione *getNextAudioBlock()*, richiamabile fino alla terminazione dei

campioni. Inoltre, in questo approccio, il problema del resampling viene banalmente superato grazie all'utilizzo della classe *AudioTransportSource* che consente di settare tramite la funzione *prepareToPlay()* la dimensione del blocco da processare ad ogni *getNextAudioBlock()* e il sampleRate in uscita, facendo internamente la conversione se non dovesse coincidere con il sampleRate dell'audio file.

Ricapitolando, l'approccio finale realizzato per la lettura degli audio file sfrutta i seguenti strumenti:

1. un **formatManager** per la creazione di un reader.
2. un **audioFormatReaderSource** per il controllo sul reader e la lettura dinamica in memoria a blocchi.
3. un **audioTransportSource** per il controllo di playback sull'audioFormatReaderSource e la conversione del sampleRate.

5.1.2 Estrazione di informazioni dallo spettro

Analizzando gli algoritmi di audio fingerprint presenti in letteratura [19] [20] [21], in primis Shazam [22], è possibile notare come l'estrapolazione di informazioni rilevanti è effettuata a partire dallo spettro del segnale, e non dal dominio temporale. Questo perché dalla magnitudine delle frequenze è possibile estrarre caratteristiche rilevanti anche in caso di degradazione del segnale causata da rumore o codifica lossy. La logica implementata, nel seguente ordine:

- calcola lo spettrogramma del segnale, utilizzando una sliding window temporale adeguata.
- suddivide in quattro bande lo spettrogramma ottenuto, estraendo da ognuna di esse i massimi.
- conserva i bin e i valori temporali corrispondenti ai picchi, scartando questi ultimi.

Lo spettrogramma di un segnale è la rappresentazione grafica della sua intensità in funzione del tempo e della frequenza. Si ottiene suddividendo il segnale in t blocchi da *windowSize*[3] campioni e calcolando una FFT su

[3]Con *windowSize* si intende la dimensione in campioni della finestra temporale su cui applicare la FFT.

ognuno di essi, t è così ottenuto:

$$t = \frac{signalSamples}{windowSize} \tag{5.2}$$

In figura 5.3 è mostrato lo spettrogramma di un jingle della durata di quattro secondi, calcolato attraverso lo strumento *spectrogram* di MATLAB. Sull'asse delle ascisse è riportato il tempo in scala lineare, sull'asse delle ordinate è riportata la frequenza in scala lineare e a ciascun punto di data ascissa e data ordinata è assegnato un colore, che ne rappresenta l'intensità. Nello specifico si nota una maggiore intensità nelle basse frequenze e per brevi istanti temporali una distribuzione uniforme di energia da 0 a 14 KHz.

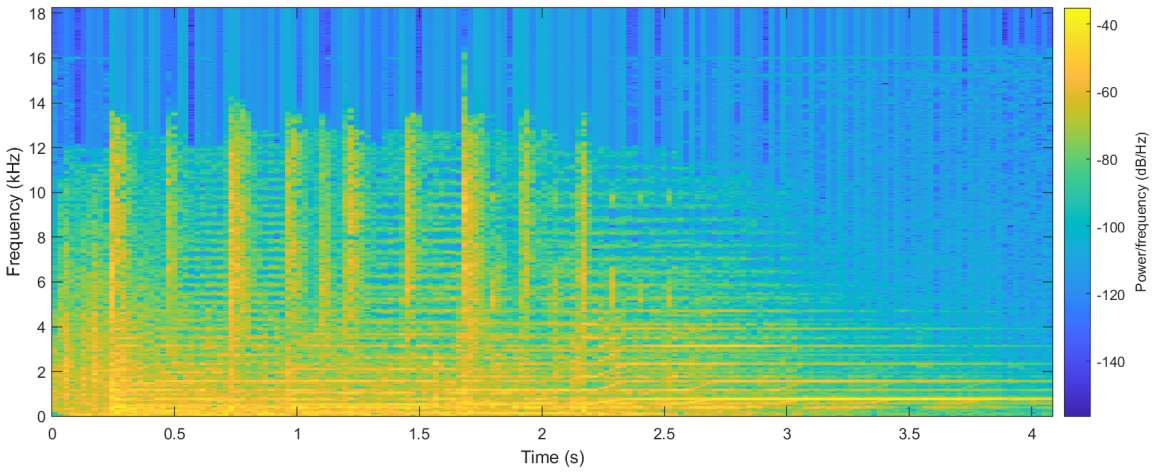

Figura 5.3: Spettrogramma di un jingle della durata di 4 secondi.

Implementazione

Nell'algoritmo implementato è stata utilizzata una FFT del nono ordine ($fftSize = 512$), che lavora su una *windowSize* di 512 campioni, e produce 512 bin in frequenza. Considerando che:

- con un $sampleRate = 44100Hz$ si ottiene un punto nello spettro ogni:

$$binRange = \frac{sampleRate}{fftSize} = 86.13Hz \tag{5.3}$$

- con un $sampleRate = 48000Hz$ si ottiene un punto nello spettro ogni:

$$binRange = \frac{sampleRate}{fftSize} = 93.75Hz \tag{5.4}$$

il parametro $fftSize$ scelto è stato ritenuto un buon valore per non avere un'eccessiva precisione in frequenza in modo tale da aumentare il numero di picchi ricadenti all'interno dello stesso *binRange*, e di conseguenza la possibilità di match. Inoltre un buffer da 512 e non più grande ci consente di ottenere una maggiore quantità di informazioni nel tempo (più hash da poter confrontare nella fase di riconoscimento).

Il segnale è stato suddiviso in blocchi, come illustrato in figura 5.4, corrispondenti ognuno ad un frame temporale t_n.

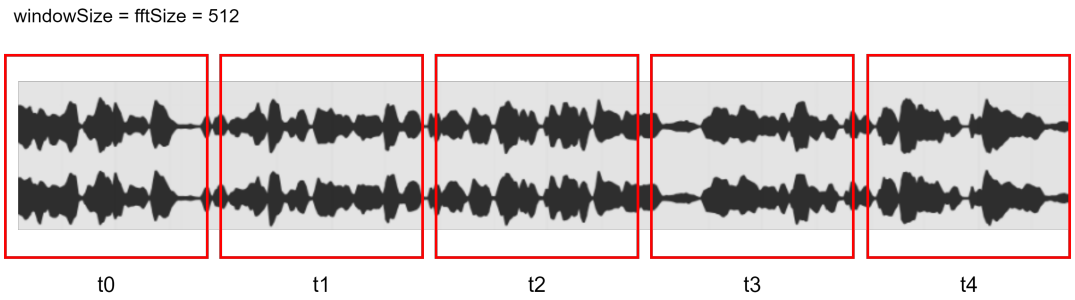

Figura 5.4: Segnale in ingresso finestrato con una sliding window, ai fini dell'estrazione di feature in frequenza.

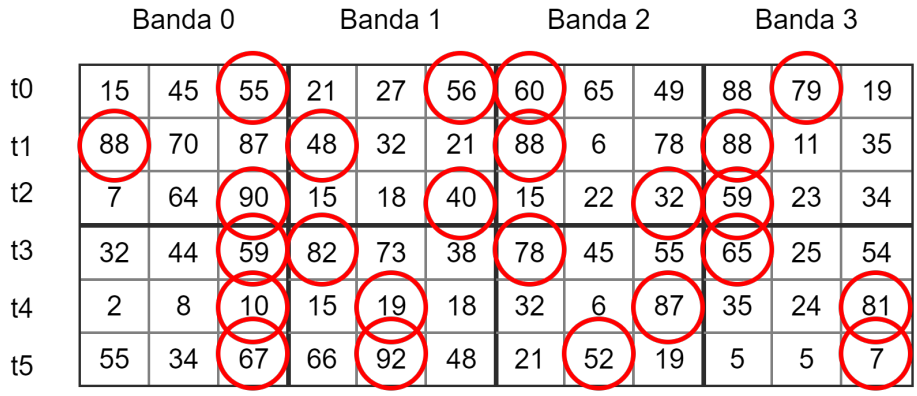

Figura 5.5: Estrapolazione picchi dallo spettrogramma per ogni sottobanda, all'istante di tempo t_n.

Successivamente, dal vettore della FFT, ad ogni istante temporale t_n è stato individuato un massimo per ognuna delle 4 sottobande selezionate, che comprendono la metà dei bin ottenuti dalla FFT, per via del teorema di Nyquist–Shannon [23]. Il processo di estrapolazione è visivamente mostrato in figura 5.5 mentre la suddivisione dello spettro in bande è così costituita:

- Banda0: da bin 40 a 79.
- Banda1: da bin 80 a 119.
- Banda2: da bin 120 a 179.
- Banda3: da bin 180 a 255.

A partire dai quattro massimi è stata calcolata una funzione di hash sui bin corrispondenti (indici del vettore della FFT). L'hash ottenuto rappresenta l'impronta digitale dello specifico jingle nell'istante temporale t_n, e sarà l'elemento chiave per il confronto nella fase di riconoscimento. In tabella 5.1 sono stati inseriti i valori dei bin e degli hash ottenuti dai massimi estrapolati nei primi 7 blocchi temporali dello stesso jingle utilizzato per il calcolo dello spettrogramma in figura 5.3.

	Pt1	**Pt2**	**Pt3**	**Pt4**	**Hash**
t0	68	88	136	180	18013608868
t1	44	98	151	193	19215009844
t2	46	82	149	191	19014808246
t3	42	83	168	209	20816808242
t4	40	94	151	192	19215009440
t5	43	87	142	183	18214208642
t6	41	81	125	194	19412408040

Tabella 5.1: Bin estrapolati nelle prime 7 finestre temporali di un jingle e relativo hash.

5.1.3 Storage degli Hash

Una volta ottenuti gli hash, questi devono essere salvati su disco e caricati in memoria (per questioni di performance), tutte le volte che il plugin eseguirà l'algoritmo di matching. Dato che:

- è possibile che all'interno di una traccia musicale (jingle) ci siano delle sezioni in cui lo spettrogramma è simile (vedi figura 5.3).
- è possibile che in due tracce musicali differenti vi siano sezioni simili nello spettrogramma.

può capitare che in frame temporali t_n differenti, o in jingle file differenti, siano estrapolati gli stessi massimi di energia, e di conseguenza sia generato un hash identico. Ogni hash ottenuto viene quindi associato a due valori:

- il **frame temporale** t_n.
- l'**ID** del jingle dal quale è stato ottenuto.

La loro collocazione in memoria è stata effettuata tramite l'utilizzo di un container unordered_map, che chiameremo banalmente **hashMap**, che ha come chiave l'hash effettivo, e come valore un oggetto *Point* contenente il **frame temporale** e l'**ID**, la struttura è mostrata in figura 5.6.

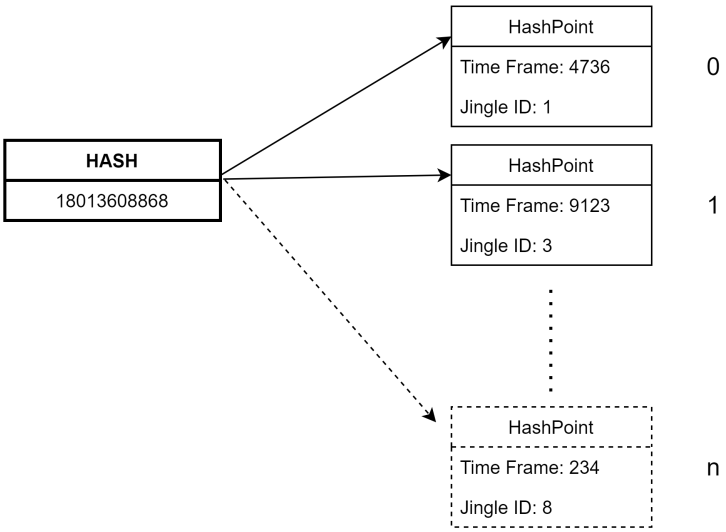

Figura 5.6: Schema di associazione uno a molti tra hash e relativi punti.

Serializzazione JSON

Per la scrittura di un file di testo su disco sono state utilizzate le API di JUCE, mentre per la serializzazione delle strutture dati è stato utilizzato il formato JSON. Questo formato si adatta perfettamente allo scopo prefissato perché è basato su due strutture:

- un insieme di coppie nome/valore, nel nostro caso gli hashPoint.
- un elenco ordinato di valori, nel nostro caso la hashMap.

JUCE, come scritto nel paragrafo 3.4, mette a disposizione una classe JSON nel modulo juce_core per la conversione dei tipi base, tuttavia si è preferito utilizzare la libreria nlohmann[4] "JSON for Modern C++" che consente anche la conversione da e verso tipi definiti dall'utente. Tramite i metodi **to_json()** e **from_json()**, è possibile specificare alla libreria come effettuare il parsing in caso di tipi non riconosciuti.

5.2 Matching real time

Il plugin dopo aver caricato in memoria gli hash ed altre informazioni sui jingle (come ad esempio titolo e durata), tramite deserializzazione JSON delle strutture dati create nella fase precedente, inizia a ricevere nella **processBlock()** i buffer dello stream audio da analizzare. Arrivati a questo punto, l'immagine 5.7 descrive le operazioni che il plugin compie per completare il processo di matching:

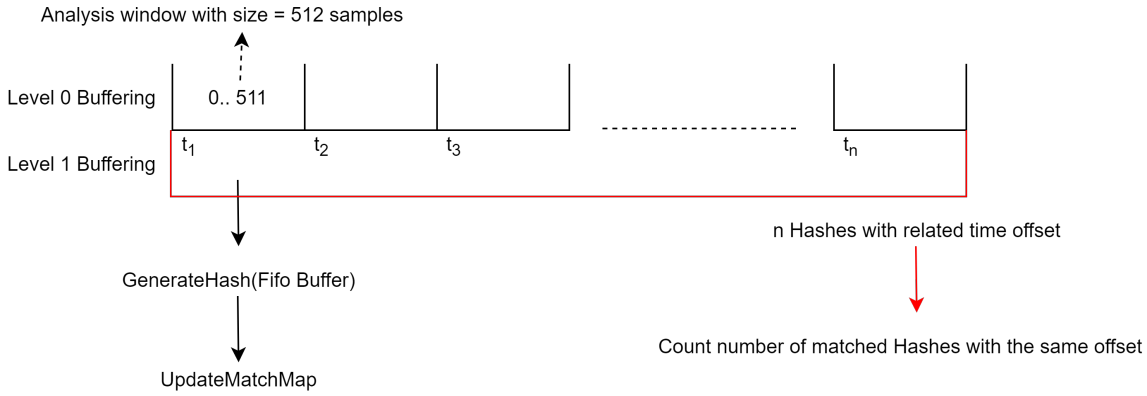

Figura 5.7: Schema di bufferizzazione e delle operazioni svolte dal plugin, nella fase di riconoscimento real time.

Nel level 0 Buffering:

[4] https://github.com/nlohmann/json

1. come prima operazione, implementa una sliding window attraverso un buffer FIFO con $size = 512$, dal quale estrapola ad ogni frame t_n un hash, applicando lo stesso tipo di analisi descritta nel paragrafo 5.1.2.
2. successivamente confronta l'hash con quelli calcolati nella fase offline, tramite algoritmo di matching, generando una nuova struttura chiamata matchMap.

Nel level 1 Buffering:

1. raccoglie n hash (vedi finestra in rosso in fig. 5.7), calcolando le informazioni di confronto all'interno della matchMap, dove n è un parametro variabile, trade-off tra robustezza e velocità di matching.
2. esegue una funzione per il controllo della matchMap, la quale verifica se ci sono dei risultati validi per decretare il riconoscimento di un jingle.

5.2.1 Algoritmo di matching

A partire da un hash **h** ottenuto tramite sliding window, l'algoritmo svolge i seguenti passaggi:

1. cerca **h** all'interno dell'**hashMap** (calcolata nella fase offline). Se lo trova estrapola tutti gli **hashPoint** associati.
2. per ogni **hashPoint** ottenuto calcola la differenza temporale, che chiameremo **offset**, nel seguente modo:

$$offset = hashPoint.getTimeFrame() - h.getTimeFrame() \quad (5.5)$$

3. se è il primo match associato ad un nuovo jingle (*hashPoint.getJingleID()*), crea un oggetto temporaneo contenente (offset, 1), che indica la presenza di un singolo match ad un determinato **offset**, ed inserisce una nuova entry all'interno della matchMap:

$$tmpMap.insert(offset, 1); \quad (5.6)$$

$$matchMap.insert(hashPoint.getJingleID(), tmpMap); \quad (5.7)$$

altrimenti estrapola il numero di match **count** già presenti per quel jingle per quel relativo **offset**, e lo incrementa di 1:

$$matchMap.find(hashPoint.getJingleID()) -> second = count + 1; \quad (5.8)$$

La struttura ottenuta, è rappresentabile come in figura 5.8, dove si nota che il valore effettivo degli hash che generano un match viene scartato perché non più utile ai fini del riconoscimento. Viene conservato però l'**offset** temporale ed il numero di **count** in cui quell'offset si ripresenta.

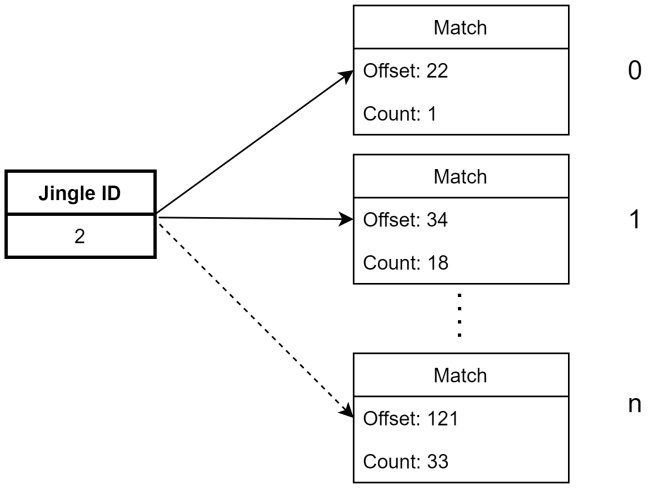

Figura 5.8: Struttura matchMap utilizzata nel conteggio dei match per ogni jingle.

5.2.2 Riconoscimento dei jingle

Come spiegato anche nell'algoritmo di Shazam [22], ai fini del riconoscimento, non è tanto rilevante il numero di match totali corrispondenti ad un jingle di riferimento ma il numero di match con lo stesso offset temporale. Poiché in questo modo si evitano i match spuri, e si tengono in considerazione quelli avvenuti consecutivamente sull'asse temporale, cioè quelli maggiormente rilevanti che ci danno alte probabilità di riconoscimento. Confrontando, al variare del tempo, gli offline hash di una traccia audio sull'asse delle ascisse, con quelli di un suo estratto analizzato nella fase live sull'asse delle ordinate, con l'aiuto della figura 5.9, ci accorgiamo che ad un determinato offset Δ iniziano ad esserci dei match consecutivi. Questi stanno sulla diagonale:

$$y = x + \Delta \tag{5.9}$$

e indicano che la traccia live che si sta analizzando coincide con una di

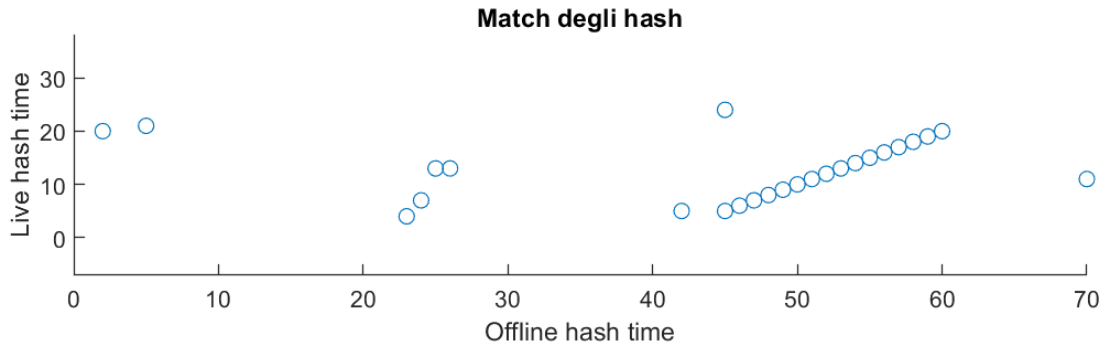

Figura 5.9: Match degli hash tra una traccia audio ed un suo estratto a partire da un offset temporale.

riferimento, a partire dal frame $t_n = \Delta$. L'algoritmo di riconoscimento ha il compito di trovare il jingle di riferimento, tra quelli computati nella fase offline, con la maggiore quantità di match consecutivi, scartando quelli spuri.

5.2.3 Accorgimenti per le performance

Per migliorare le performace nel riconoscimento sono state implementate due modifiche a quanto descritto precedentemente. Lo scopo comune di entrambe è quello di incrementare il numero di hash da calcolare e confrontare, nel momento in cui si scorre lo stream audio, al fine di avere dei risultati consistenti nel minor tempo possibile. All'inizio di questo paragrafo 5.2 è stata fatta una distinzione tra level 0 buffering, e level 1 buffering: il primo bufferizza i campioni da analizzare, il secondo gli hash da confrontare. Le modifiche apportate agiscono su entrambi i livelli.

Overlap sulle finestre di analisi

La prima modifica migliora le performance nel caso in cui la sezione da riconoscere sia shiftata nel tempo, di pochi campioni rispetto alla traccia di riferimento. Osservando la fig. 5.7 si può notare che il caso peggiore si otterrebbe con uno shift pari a $size/2$, poiché l'FFT, e quindi l'hash, verrebbe calcolato su un buffer che contiene solo per metà gli stessi campioni. Per ridurre il problema, è stato implementato un overlap del 50% sulle finestre di analisi (level 0 buffering), facendole scorrere solo per metà della loro dimensione. La figura che chiarisce il concetto è mostrata in 5.10,

in questo caso vengono generati il doppio degli hash, e di conseguenza è atteso un aumento del numero di match.

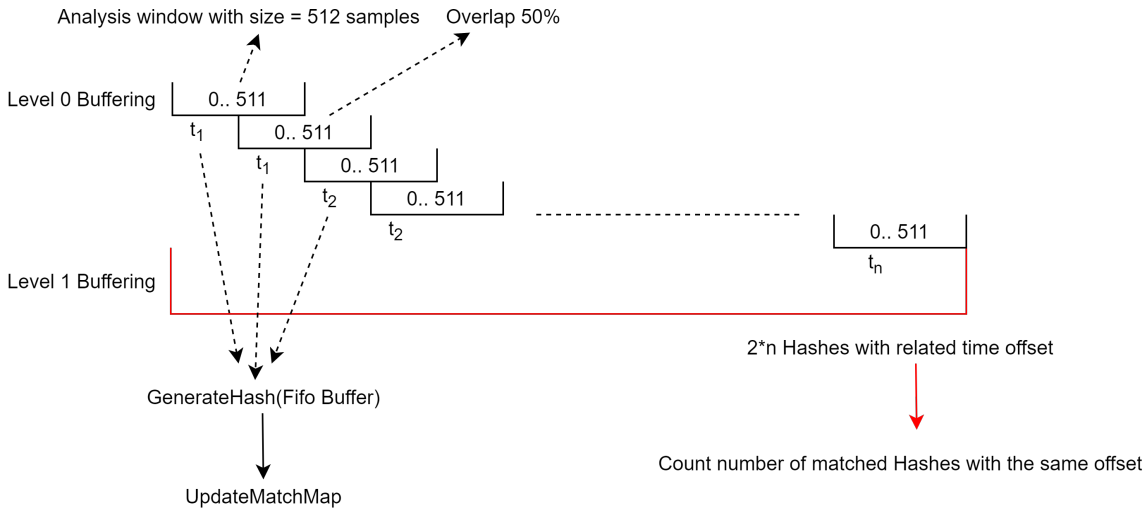

Figura 5.10: Schema di analisi in tempo reale, nel caso di overlap sulle finestre di analisi del 50%.

Overlap sulle finestre di matching

Ogni finestra di matching (level 1 buffering) raccoglie $2 \cdot n$ hash, prima che venga chiamata la funzione **computeBestMatch** per il calcolo della migliore corrispondenza tra le tracce offline. Va da se che più grande è n, più informazioni saranno utilizzate durante il confronto e maggiore sarà la probabilità di riconoscimento. Tuttavia, incrementando n, e quindi la finestra di matching, si introduce un maggiore ritardo temporale, prima di un ipotetico riconoscimento, poiché la **computeBestMatch** sarà richiamata con una minore frequenza. Facendo un esempio pratico:

- con $n = 128$ ed un $sampleRate = 48000$, la **computeBestMatch** è richiamata ogni:

$$t_{frequency} = \frac{n \cdot frameAnalysisSize}{sampleRate} = \frac{128 \cdot 512}{48000} = 1.365s \qquad (5.10)$$

- con $n = 256$ ed un $sampleRate = 48000$, ci si aspetta il doppio del ritardo:

$$t_{frequency} = \frac{n \cdot frameAnalysisSize}{sampleRate} = \frac{256 \cdot 512}{48000} = 2.730s \qquad (5.11)$$

Volendo raddoppiare il valore di n ma allo stesso tempo mantenere invariata la frequenza, con cui viene richiamata la **computeBestMatch**, si è arrivati alla modifica che prevede la sovrapposizione delle finestre di matching del 50%, come la figura 5.11 illustra. Per implementarla è stato necessario utilizzare due matchMap, che saranno quindi riempite contemporaneamente con gli stessi hash ma con indici temporali differenti (l'indice è uguale a zero all'inizio di ogni finestra di matching, in rosso e in blu nella fig. 5.11).

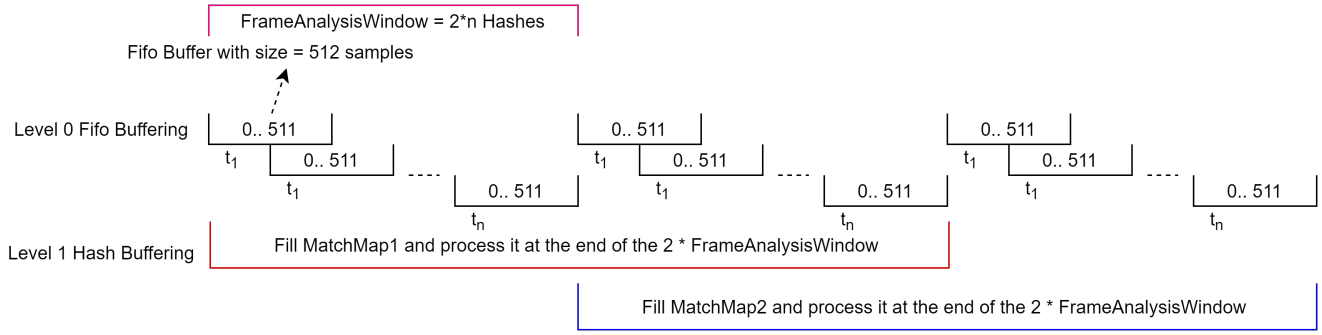

Figura 5.11: Schema di analisi in tempo reale, nel caso di overlap sulle finestre di matching del 50%.

5.3 Audio Injection: implementazione tramite audio Fingerprint

Come anticipato, l'inserimento dinamico nel caso di implementazione tramite audio fingerprint è subordinato al riconoscimento di un sample di apertura ed uno di chiusura (jingle). Al fine di rendere l'operazione il più trasparente possibile nei confronti dell'utente, è stato scelto di non sovrascrivere questi momenti musicali, e di mandarli in riproduzione nello stream editato. Questo approccio è più critico rispetto a quello implementato con Goertzel poiché:

- l'inserimento dinamico deve avvenire alla fine esatta del jingle di apertura, e non immediatamente dopo il rilevamento di un tono. Tuttavia come è possibile conoscere il punto esatto di taglio se i jingle hanno una durata variabile e il loro riconoscimento può avvenire in qualsiasi finestra di matching al loro interno?

- il riaggancio dello stream originale deve coincidere con l'inizio del jingle di chiusura, e non nel momento esatto di rilevazione di un secondo tono. Ma come è possibile effettuare una tale operazione se il jingle di chiusura deve ancora essere analizzato?

La figura 5.12 mostra un estratto pubblicitario della radio nazionale 'Rai Radio 1' ed evidenzia in rosso i jingle J1, J2, e l'inserzione tra i due. É possibile notare la durata variabile dei tre elementi (J1, J2, inserzione) a sostegno delle problematiche esposte nei punti precedenti, a cui sarà data una soluzione nei sottoparagrafi seguenti.

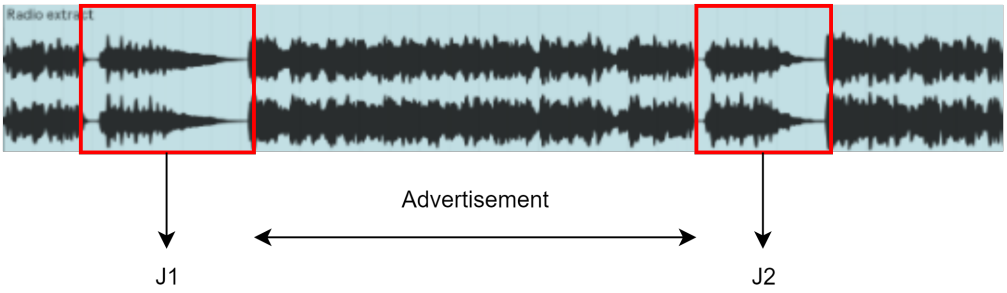

Figura 5.12: Schema di una sezione pubblicitaria estratta dalla radio nazionale 'Rai Radio 1'.

Individuazione del punto di inizio dell'inserzione

Per individuare il punto esatto di taglio, in termini pratici quanti campioni attendere prima dell'avvio dell'inserzione custom, è necessario conoscere:

- durata del Jingle.
- miglior offset della finestra di matching in cui è avvenuto il riconoscimento.

Nonostante si sia adottato un meccanismo di lettura degli audio file a blocchi, la durata in samples del jingle è stata facilmente ottenuta, nella fase offline 5.1.1 tramite la classe **transportSource** di JUCE, con il seguente metodo:

```
1 int samples = transportSource.getTotalLength();
```

Il miglior offset è invece lo sfasamento tra la finestra di matching e la traccia di riferimento che ha generato più match consecutivi, ottenuto

come descritto nel paragrafo 5.2.2. Esso può assumere un valore positivo o negativo e la figura 5.13 mostra un esempio di entrambi i casi:

- se ha un valore negativo (fig. 5.13 A) significa che il jingle è riconosciuto nel minor tempo possibile poiché ha inizio nella finestra di matching corrente[5].
- se ha un valore positivo (fig. 5.13 B) il jingle ha avuto inizio in una finestra di matching passata.

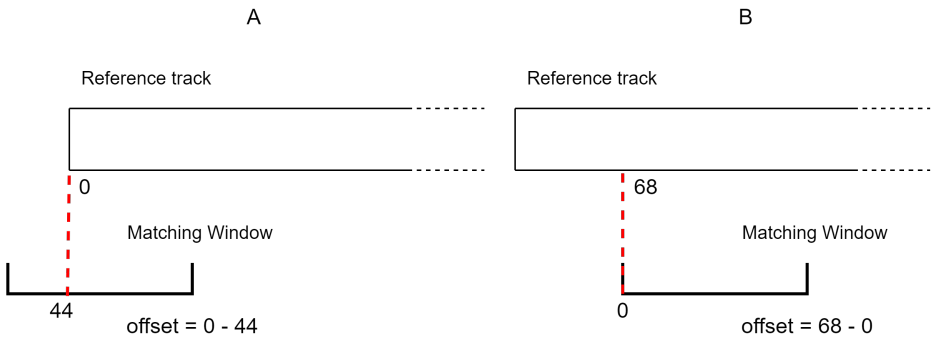

Figura 5.13: Esempio di match di un reference jingle, nel caso di miglior offset positivo e negativo.

Dal **bestOffset** si ottiene il punto di riconoscimento in campioni del jingle dal suo inizio, e sottraendolo alla lunghezza totale, si ottengono i campioni rimanenti alla sua fine, e quindi all'inizio della pubblicità custom, come mostrato nel seguente codice[6]:

```
1 nSamplesOffset = matchMapFrame * matchingWindoSize * fftSize
      + bestOffset * fftSize;
2 remaining = duration - nSamplesOffset;
```

Per comodità ad ogni riconoscimento tramite fingerprint viene restituito un oggetto di tipo **RecognizedJingle** che contiene tutte le informazioni utili (che in parte saranno mostrare nella GUI) per effettuare le operazioni di editing, nello specifico:

[5]Con $matchingWindowSize = 128$ ed un $sampleRate = 48000$ il riconoscimento avviene dopo 1.365s.

[6]Considerando che **matchMapFrame** indica se vi è overlap nelle matchMap, **matchingWindoSize** la dimensione della finestra di matching, ed **fftSize** la dimensione della finestra di analisi.

- durata in campioni.
- offset di riconoscimento in campioni.
- tempo rimanente alla fine in campioni.
- ID.
- titolo con estensione.

Individuazione del punto di fine dell'inserzione

Una volta avviato l'inserimento dinamico (alla terminazione di J1), il plugin ricerca J2 allo stesso modo di J1 sullo stream non editato. Sarebbe possibile interrompere l'inserzione custom e riprendere la riproduzione dall'offset di riconoscimento di J2, ma questo causerebbe un artefatto udibile e fastidioso. Rispondendo alla seconda domanda esposta all'inizio di questo paragrafo, è possibile invece riprendere lo stream in maniera trasparente lato utente, terminando correttamente l'inserzione e riproducendo il jingle di chiusura nella sua interezza, sfruttando un ritardo di riproduzione inserito a monte, grazie al componente 'Delay Temporale' descritto nel paragrafo 4.1. Nel momento in cui si riconosce J2, la riproduzione dei campioni sarà quindi $\Delta_{samples}$ nel passato, in questo modo si potrà riagganciare lo stream quando essa raggiungerà l'inizio di J2.
Dato un delay in campioni $\Delta_{samples}$, l'individuazione perfetta della fine dell'inserzione, ovvero tra quanti campioni terminare l'injection, sarà ottenuta come:

$$samplesAdRemaining = \Delta_{samples} - J2.getOffsetInSamples() \quad (5.12)$$

Struttura Generale

La logica generale dell'injection tramite audio fingerprint può essere schematizzata con un automa a stati finiti rappresentato in figura 5.14. Dove con il valore 0/1 si indica se il riconoscimento tramite fingerprint è andato a buon fine, mentre con le etichette le seguenti corrispondenze:

- A: (fSearchingJ1) è lo stato in cui viene ricercato J1.
- B: (fRecognizedJ1) è lo stato in cui J1 è riconosciuto e se ne attende la terminazione prima di ritornare ad effettuare l'analisi in A'.
- A': (fSearchingJ2) è lo stato in cui si ricerca J2.

- B': (fRecognizedJ2) come B, ma per J2.
- C: (waitJ1DeleyedEnd) è lo stato in cui si attende la terminazione di J1 in fase di riproduzione, al fine di mostrare nella GUI il tempo mancante alla sua terminazione.
- D: (waitAdEnd) è lo stato in cui si attende la terminazione dell'inserzione, in fase di riproduzione, al fine di riagganciare lo stream originale.
- E: (waitJ2End) è lo stato in cui si attende la terminazione di J2 in fase di riproduzione, al fine di mostrare nella GUI il tempo rimanente alla sua terminazione.

In sintesi, gli stati A, B, A', B' agiscono sui campioni prima delle azioni di delay e servono ad attivare gli stati C, D, E che invece agiscono sullo stream ritardato (ascoltato dall'utente finale).

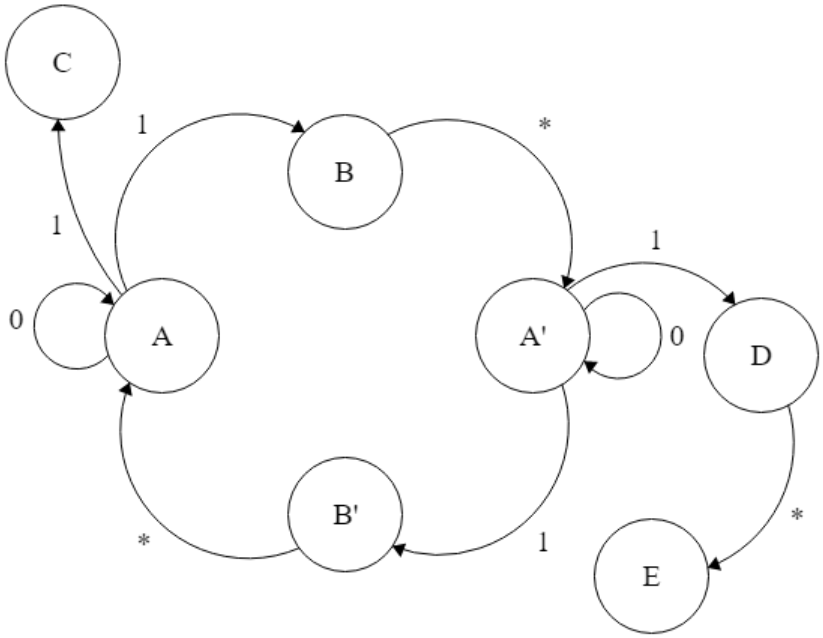

Figura 5.14: Automa a stati finiti della logica di injection al riconoscimento di J1 e J2 tramite algoritmo di fingerprint.

Capitolo 6

Utilizzo del software

Il Software utilizza una directory di supporto: **audio-ad-insertion-data**, che per comodità viene ricercata automaticamente dal plugin all'interno della directory documenti della macchina su cui esso girerà. I sorgenti sono compilabili previa installazione di JUCE ed uno tra gli IDE supportati all'esportazione (Xcode, Visual Studio, Code::Blocks, Clion) con relativi compilatori. La compilazione è stata testata con JUCE 6.0.1, Visual Studio Community 2017 ed il kit "Desktop development with c++".

Analisi offline

Per la generazione del database degli hash, è necessario:

1. creare una directory **audio-ad-insertion-data\audioDatabase** contenente tutti i jingle che saranno presenti all'interno dello stream radiofonico da analizzare. Possono essere utilizzati file audio dei formati più comuni loseless e lossy.
2. aprire il progetto projucer **FingerprintLoader**, esportarlo su un IDE di preferenza tra quelli disponibili, ed avviare il processo di compilazione.
3. avviare l'eseguibile **FingerprintLoader** ottenuto dal processo di compilazione, e dall'interfaccia, come mostrato in 5.2, selezionare il sampleRate di default che sarà utilizzato all'interno della DAW / host che ospiterà il plugin.
4. una volta completata l'operazione di caricamento degli hash, all'interno di audio-ad-insertion-data sarà creata una directory dataStructures

contenente i file json delle strutture dati, che il plugin leggerà in fase di avvio.

Esecuzione del Plugin all'interno di una DAW

Per attivare il processo di riconoscimento e sostituzione degli spot pubblicitari:

1. aprire il progetto projucer audio-ad-insertion, dal quale selezionare il formato del plugin che si desidera ottenere (VST3, AU, AUv3, RTAS, AAX, Standalone, Unity, VST (Legacy)), tenendo presente che per il formato VST(Legacy) è necessario prima fornirsi dell'SDK VST (versione testata: vstsdk367_03_03_2017_build_352); ed esportare il tutto su un IDE come nel caso precedente.
2. avviare la compilazione, e spostare il binario ottenuto nella directory utilizzata dalla DAW per la ricerca dei plugin 64bit.
3. creare una traccia all'interno della DAW su cui istanziare il plugin, ed infine collegare lo stream radiofonico in ingresso alla traccia tramite routing. La DAW su cui è stato eseguito il plugin è Ableton Live, con il formato VST(legacy).

Il plugin in fase di injection pubblicitaria, leggerà i file audio presenti all'interno della directory **audio-ad-insertion-data\audioInjection**.

Demo

Per una veloce demo in ambiente Windows, nel software è già fornito un eseguibile FingerprintLoader.exe ed il VST(legacy) audio-ad-insertion.dll, dei jingle di test in **audio-ad-insertion-data\audioDatabase** con i relativi hash a 44100 Hz in **audio-ad-insertion-data\dataStructures**, e delle pubblicità personalizzate in **audio-ad-insertion-data\ audioInjection**. Per comodità in **audio-ad-insertion-data\stream** sono state inserite anche delle sezioni stream di esempio, contenenti i jingle da riconoscere, che possono essere caricate all'interno della DAW, simulando uno streaming in ingresso, su cui applicare il plugin per la sostituzione in tempo reale. Basterà quindi spostare la cartella **audio-ad-insertion-data** in documenti, importare il plugin nella DAW, settare in essa lo stesso samplerate usato per la generazione degli hash (44100 HZ), caricare uno stream di esempio, ed avviare la riproduzione.

Capitolo 7

Risultati

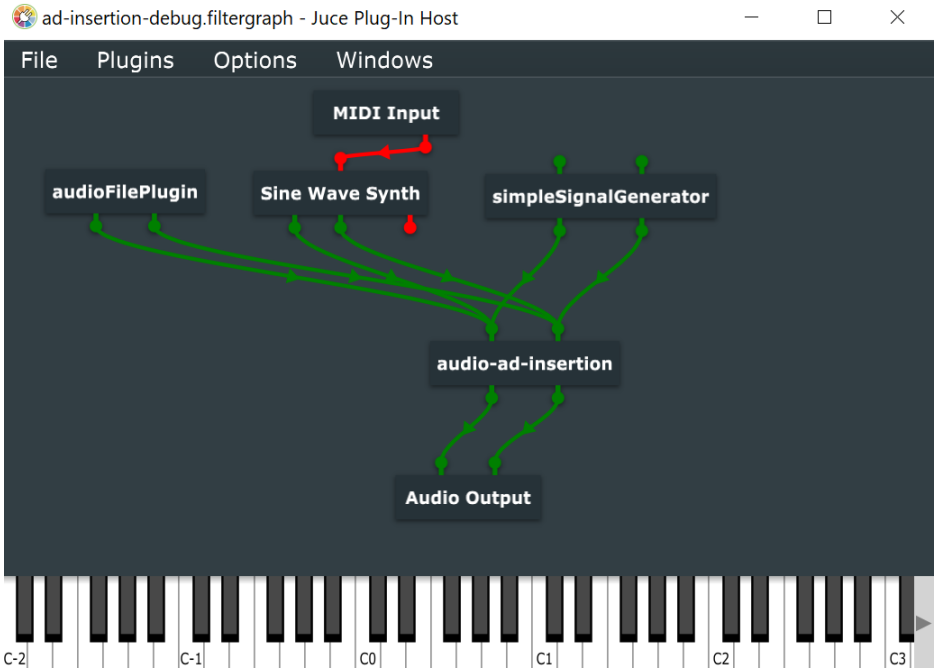

Figura 7.1: Schema di routing all'interno dell'audioPluginHost per il test del plugin audio-ad-insertion.

Nella fase di testing il plugin **audio-ad-insertion** è stato eseguito all'interno dell'audioPluginHostDebug fornito da JUCE. Questo strumento è un host che consente di istanziare diversi plugin audio all'interno, creando specifici modelli di routing. Dalla figura 7.1 si osserva la struttura di supporto utilizzata:

- 'SimpleSignalGenerator' e 'Sine Wave Synth' sono stati impiegati nella prima parte del lavoro, come spiegato nel paragrafo 4.2, per la generazione di toni a specifiche frequenze e conseguente riconoscimento.
- 'AudioFilePlugin' è stato creato ed utilizzato appositamente per la lettura degli stream audio da passare all'**audio-ad-insertion**, in modo tale da verificare il suo corretto funzionamento.

7.1 Effetti del disallineamento

Per valutare l'efficacia della prima modifica introdotta nel paragrafo 5.2.3, ovvero l'overlap sulle finestre di analisi, è stato applicato l'algoritmo di fingerprint su una traccia musicale con energia in tutte le frequenze udibili, prima per la conservazione degli hash offline, successivamente per il confronto in tempo reale. Le finestre di analisi, ricordiamo essere dei buffer che nel tempo raccolgono $fftsize = 512$ campioni, sui quali è applicata l'FFT, con conseguente generazione di un hash che identifica quella traccia in quel frame. Cosa succede però se la traccia da riconoscere è disallineata rispetto a quella di riferimento di pochi campioni fino ad un massimo di 512, e cosa cambia se si abilita l'overlap?

Per rispondere a questa domanda, è stato volutamente introdotto uno shift incrementale, da 32 a 512, sulla traccia di analisi, ed è stato analizzato il numero di occorrenze dei match raccolti all'interno dei primi 10.88 secondi; in un primo test con finestre di analisi contigue (overlap disabilitato), in un secondo test abilitando l'overlap del 50% (ogni 256 campioni). I risultati sono riportati in tabella 7.1.

Shift	**0**	**32**	**64**	**96**	**128**	**160**	**192**	**221**	**256**	**288**	**320**	**352**	**384**	**416**	**448**	**480**	**512**
No Overlap	934	415	196	88	47	25	16	12	9	9	20	20	37	76	170	413	934
Overlap 50%	939	423	216	107	84	100	186	424	943	421	201	89	51	83	176	418	941

Tabella 7.1: Valori delle occorrenze, applicando uno shift temporale tra la traccia di analisi da 10.88 secondi e quella di riferimento.

Considerando che:

- dalla traccia offline, nei primi 10.88 secondi sono disponibili 937 hash.

- dalla traccia di analisi, senza overlap sono generati 937 hash.
- dalla traccia di analisi, con overlap sono generati 1874 hash.

Con shift pari a 0, gli hash di riferimento vengono matchati quasi nella totalità sia con che senza overlap (939 e 934, considerando qualche match spurio nel caso di overlap). Aumentando invece il valore di shift, il numero di occorrenze decresce fino al caso peggiore che, come ci si aspettava, con overlap disabilitato coincide con $fftsize/2$, con overlap abilitato coincide con $fftsize/4$. Dal grafico in figura 7.2, è più facile comprendere l'andamento: quando a 256 campioni di shift il caso no overlap raggiunge il minimo assoluto (9), il caso overlap raggiunge invece il picco massimo (943). I valori minimi raggiunti (9 e 51) in entrambi i casi non devono preoccupare, infatti riescono lo stesso ad essere discriminatori, se consideriamo che nello stesso test le altre tracce nel DB (non quella di riferimento) hanno prodotto in tutti i frame al massimo 3 match spuri senza overlap, e 4 match spuri con overlap.

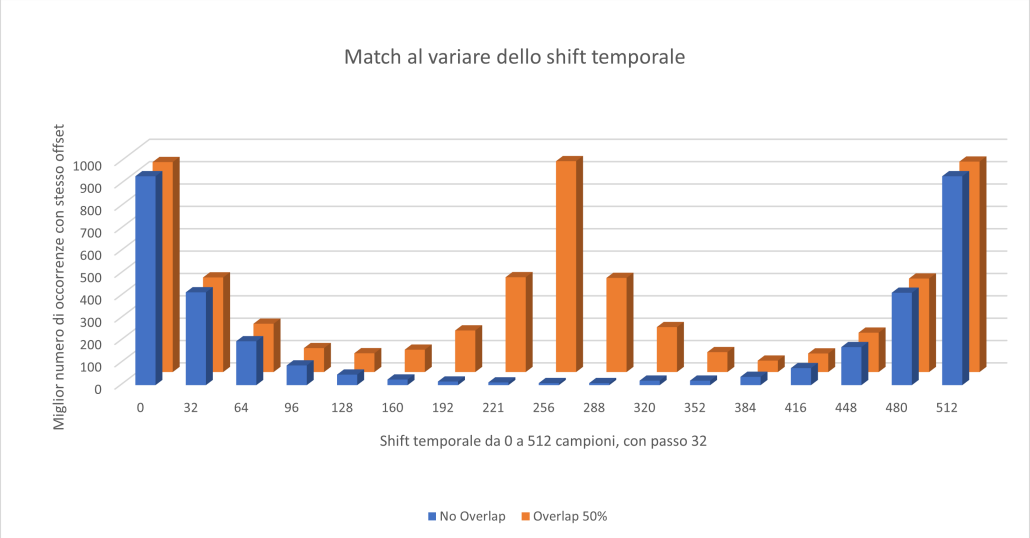

Figura 7.2: Miglior numero di match con stesso offset, al variare del disallineamento tra la traccia di analisi e quella di riferimento.

Tuttavia visto l'andamento del grafico, considerato il risultato peggiore overlap (51 occorrenze) circa 5 volte superiore al risultato peggiore no overlap (9 occorrenze), e considerata la media dei valori overlap nei casi intermedi più alta, è possibile dedurre che l'overlap sulle finestre di analisi è sicuramente da preferire. Per questo motivo è stato sempre abilitato

in tutti i test reali su stream audio, che saranno esposti nel successivo paragrafo.

7.2 Riconoscimento dei jingle ed inserimento dinamico

In questa sezione ci concentreremo sull'inserimento dinamico tramite algoritmo di audio fingerprint, che è stato testato prima su dei prototipi audio, creati ex novo, annegando all'interno di essi dei jingle di media durata (~10s), successivamente su delle sezioni audio, estratte direttamente da radio broadcast nazionali. Nel secondo caso, è stato necessario individuare ed estrapolare ai fini dell'analisi, i jingle utilizzati dalle stazioni radio in contesti reali. I test sono stati effettuati mantenendo fissa la soglia di riconoscimento del conteggio degli offset pari a $countThreshold = 5$ e la dimensione della finestra di analisi (level 0 buffering) a 512 campioni con overlap del 50%, variando però la dimensione delle finestre di matching (level 1 buffering, 128 e 256), con e senza overlap.

7.2.1 Test su uno stream prototipo

Il prototipo utilizzato in questo esempio è una sezione audio da 38 secondi, con all'interno un jingle di apertura della durata di 8 secondi, ed uno di chiusura da 9 secondi. Inseriti a distanza di 5 secondi per verificare il corretto funzionamento delle azioni di delay, anche nel caso limite in cui si debba riconoscere un nuovo jingle prima che ancora il precedente abbia completato la sua riproduzione. Nel primo test, è stata utilizzata una matchingWindowSize di 128, e disabilitato l'overlap sulle finestre di matching: come è possibile osservare in figura 7.3, il riconoscimento è andato a buon fine consentendo una corretta sostituzione pubblicitaria, difatti i jingle n.4 e n.5 superano la soglia $countThreshold$, rispettivamente nei frame t_6 e t_{11}. Gli offset ottenuti negli stessi frame temporali sono mostrati nella tabella 7.2.

Utilizzando invece una matchingWindowSize di 256 (figura 7.4), si osserva un aumento di robustezza, ovvero un aumento dei match dovuto al maggior numero di hash raccolti nelle finestre di matching. Dalla tabella 7.3, si evince che il numero di match sale vertiginosamente da 9 a 20 e da 35 a 96, a discapito di un ritardo leggermente superiore nel riconoscimento

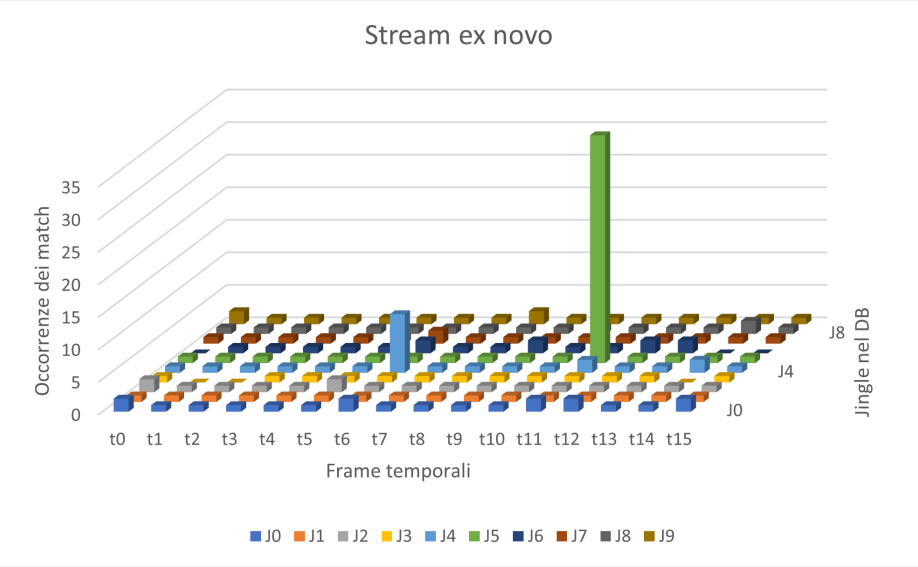

Figura 7.3: Riconoscimento in tempo reale dei jingle di un prototipo pubblicitario, con $matchingWindowSize = 128$ e Overlap disabilitato.

	J0	J1	J2	J3	J4	J5	J6	J7	J8	J9
t6	2(857)	1(901)	1(1133)	1(129)	9(31)	1(257)	2(86)	2(351)	1(480)	1(1178)
t11	2(6384)	1(360)	1(537)	1(-80)	2(424)	35(-39)	1(81)	1(203)	1(1235)	1(588)

Tabella 7.2: Stream Prototipo: BestOffset e relativo count nelle finestre di avvenuto riconoscimento, $matchingWindowSize = 128$ e Overlap disabilitato.

(ad esempio nel calcolo del primo jingle si attendono 4 finestre da $256 \cdot 2$ hash al posto di 7 finestre da $128 \cdot 2$ hash).

Infine è stato effettuato un test abilitando l'overlap sulle finestre di matching con matchingWindowsize = 256. In questo caso si è ottenuto un riposizionamento degli offset, dovuto alla diversa alternanza delle finestre rispetto ai casi precedenti ed un numero di match invariato rispetto allo stesso caso senza overlap (riconoscimento in $t_6 : 9(-97)$ e $t_{11} : 35(-167)$; anche l'andamento grafico è molto simile e per questo motivo non riportato). Come spiegato nel paragrafo 5.2.3, l'overlap sulle finestre di matching, come quello sulle finestre di analisi, aiuta infatti a migliorare le performace di riconoscimento in caso di differenti allineamenti temporali del segnale, non è quindi certa una sua effettiva efficacia, come in quest'ultimo test. In conclusione, tra i tre setup, il primo ed il terzo (essendo simili in costo

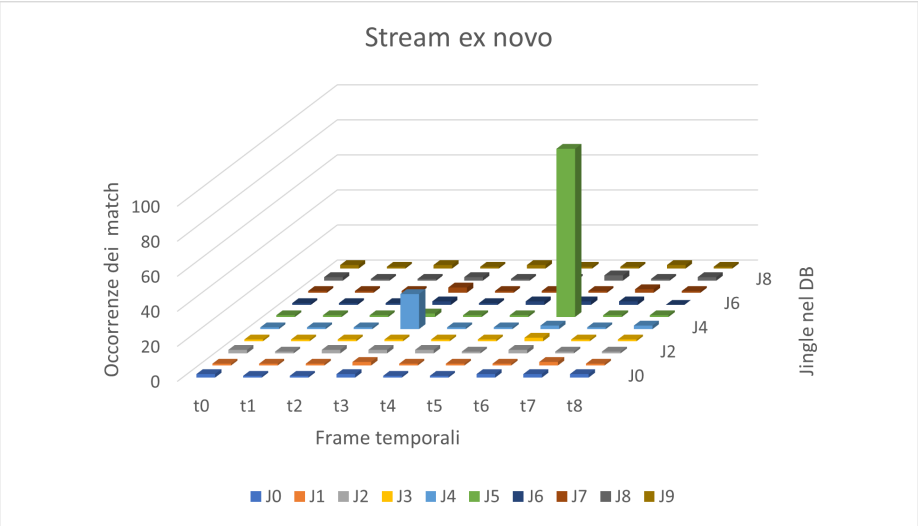

Figura 7.4: Riconoscimento in tempo reale dei jingle di un prototipo pubblicitario, con $matchingWindowSize = 256$ e Overlap disabilitato.

	J0	J1	J2	J3	J4	J5	J6	J7	J8	J9
t3	2(2160)	2(123)	2(344)	1(-62)	20(31)	2(426)	2(9)	3(351)	2(1137)	1(1178)
t6	2(6384)	1(360)	2(703)	2(-47)	2(424)	96(-39)	2(107)	1(203)	3(1250)	1(588)

Tabella 7.3: Stream Prototipo: BestOffset e relativo count nelle finestre di avvenuto riconoscimento, $matchingWindowSize = 256$ e Overlap disabilitato.

computazionale e performance) sono considerati i migliori, mentre il secondo nonostante il vantaggio dei match è sconsigliabile poiché il margine aggiuntivo che fornisce non porta delle migliorie al funzionamento finale, introducendo del ritardo aggiuntivo nel riconoscimento ($128 \cdot 2$ hash).

7.2.2 Test su uno stream reale

Dopo aver registrato diverse ore di broadcast streaming da alcune radio principali, sono stati estratti i jingle ai fini della memorizzazione offline, e delle sezioni audio su cui effettuare dei test di riconoscimento. In questo sottoparagrafo sono mostrati i risultati ottenuti analizzando una sezione da 33 secondi di una trasmissione 'Rai Radio 1' contenente un'inserzione pubblicitaria di 15 secondi anticipata e conclusa da jingle di breve durata (4 secondi). Come fatto precedentemente sullo stream prototipo, in

questi test sono stati variati i parametri matchWindowSize e overlap, valutando vantaggi e svantaggi. La tabella 7.4 riporta i frame in cui è stata trovata una corrispondenza con un jingle di riferimento, nel caso A utilizzando $matchingWindowSize = 128$ e *overlap* disabilitato, nel caso B con $matchingWindowSize = 256$ e *overlap* disabilitato, nel caso C con $matchingWindowSize = 256$ e *overlap* abilitato. In tutti e tre i casi i jingle sono stati riconosciuti correttamente, e l'offset ottenuto è risultato sempre valido ai fini del calcolo della rimanenza in campioni necessaria alle azioni di inserimento dinamico.

Test	Frame	J0	J1	J2	J3	J4	J5	J6	J7	J8	J9
A1	t2	1(7459)	1(476)	1(470)	17(-24)	1(465)	1(585)	1(314)	1(95)	1(1603)	1(381)
A2	t14	2(6928)	1(976)	2(985)	1(17)	2(28)	2(501)	16(83)	2(345)	1(703)	2(-40)
B1	t1	2(8705)	2(307)	1(932)	45(-24)	1(465)	2(478)	1(17)	1(95)	2(1114)	1(381)
B2	t7	2(6800)	2(162)	2(560)	1(217)	2(435)	2(429)	19(-46)	2(217)	2(587)	2(55)
C1	t2	2(3669)	2(127)	1(1379)	23(-23)	2(265)	1(348)	2(-8)	1(271)	2(1315)	2(254)
C2	t14	2(6800)	2(162)	2(560)	1(217)	2(435)	2(429)	19(-46)	2(217)	2(587)	2(55)

Tabella 7.4: Stream Reale: **bestOffset** e **count** dei frame in cui si verifica una corrispondenza nei casi **A**: $matchingWindowSize = 128$ e Overlap disabilitato, **B**: $matchingWindowSize = 256$ e Overlap disabilitato, **C**: $matchingWindowSize = 256$ e Overlap abilitato.

I jingle all'interno dell'estratto: il n.3 ed il n.5, sono stati rilevati con un buon valore di *count* dell'offset al di sopra della soglia prefissata (> 5), mentre si può notare che in tutti gli altri il valore non supera i 2 match spuri. Nel caso **A** si hanno 17 match con l'offset -24, e 16 match con l'offset 83, nel caso **B** si ha un aumento significativo di match, dovuto alla dimensione più ampia della finestra di matching come nel test su stream prototipo, con 45 match sull'offset -24 e 19 match sull'offset -46. Infine nel caso **C** si trae vantaggio dall'overlap abilitato, esso è un buon compromesso tra i due precedenti ed è stato ritenuto il migliore: si hanno infatti delle prestazioni simili al caso **B** (23 match sull'offset -23 e 19 match sull'offset -46) ma con un controllo dei valori ogni $128 \cdot 2$ hash, più frequente e reattivo del caso **B** ogni $256 \cdot 2$ hash. La presenza di offset negativi in quasi tutti i risultati ottenuti è un buon indice di funzionamento, poiché equivale ad un riconoscimento nella prima finestra disponibile per quel setup dell'algoritmo, e fa la differenza in jingle con pochi secondi a disposizione da analizzare, come questo caso in esame. In figura 7.5 è mostrato l'andamento del riconoscimento nel caso **B**, mentre

in figura 7.6 il caso **C**.

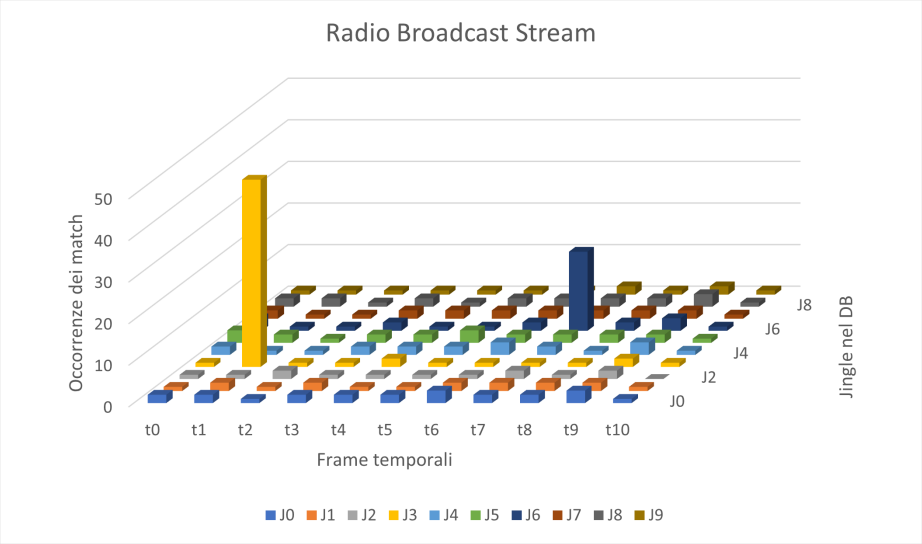

Figura 7.5: Riconoscimento in tempo reale dei jingle di uno stream reale, con $matchingWindowSize = 256$ e Overlap disabilitato.

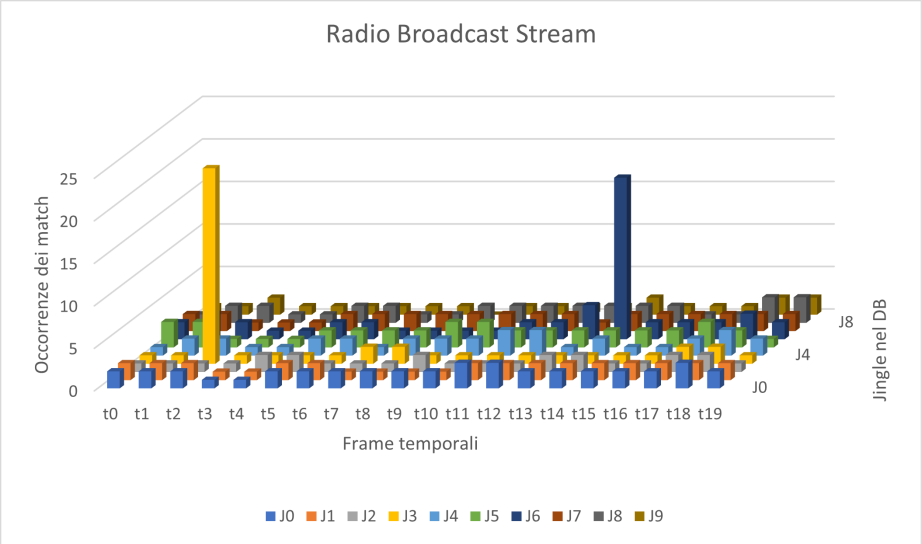

Figura 7.6: Riconoscimento in tempo reale dei jingle di uno stream reale, con $matchingWindowSize = 256$ e Overlap abilitato.

Capitolo 8

Conclusioni

Il primo approccio di questo lavoro, nel quale è stato implementato il riconoscimento di toni a specifiche frequenze tramite algoritmo di Goertzel, è risultato efficace per l'individuazione del punto di inizio e fine dell'inserzione al superamento di una soglia, opportunamente settata al 35% di magnitudine. L'utilizzo di un tono a frequenza non udibile (non presente quindi in musica) garantisce sempre un rilevamento e non genera falsi positivi, grazie all'assenza di 'interferenze' (altri toni) all'interno della stessa banda. Tuttavia necessita un operatore che modifichi lo stream broadcast dalla sorgente prima della diffusione, inserendo i toni come marcatori, parallelamente alla programmazione degli spot pubblicitari. Una volta fatto ciò il plugin a destinazione, lavora senza l'ausilio di alcun supporto (DB) che necessiti aggiornamento nei mesi e negli anni. Il secondo approccio invece, consente di mantenere inalterato lo stream già prodotto, lasciando intatto il sistema di produzione lato sorgente, poiché i jingle sono già presenti nel modello AS IS. Tuttavia aggiunge un'operazione a destinazione necessaria prima della fruizione del flusso radiofonico, ovvero l'hashing dei jingle (non così fastidiosa se consideriamo che i jingle utilizzati da una radio non superano la decina, sono facilmente reperibili, ed il tool FingerprintLoader impiega pochi secondi per effettuare l'analisi). Se i jingle all'interno dello stream non subiscono un'alterazione invadente (filtri EQ, compressione upword e downword elevata ecc.), il confronto con quelli all'interno del database tramite l'algoritmo sviluppato produce dei risultati validi per decretare un match o un miss in maniera accurata. Nei test effettuati, non sono stati rilevati dei falsi positivi, infatti le

altre tracce non candidate al match, hanno prodotto pochissime occorrenze spurie in ogni finestra temporale (2 o 3), con la soglia *countThreshold* per il decretamento di un match valido settata su 5 occorrenze. Dai test si è dedotto che il miglior compromesso tra velocità di riconoscimento e robustezza dell'algoritmo di audio fingerprint è ottenibile con overlap del 50% sulle finestre di analisi (level 0 buffering) da 512 campioni, ed overlap del 50% sulle finestre di matching (level 1 buffering) da 256 frame, infatti nel test **C** (50% overlap matching) eseguito su uno stream reale si sono verificate 23 e 19 occorrenze rispettivamente per il jingle di apertura e di chiusura (raccolti all'interno della prima finestra di matching, entrambi offset negativi) più alte rispetto al caso **A** (no overlap matching) a parità di campioni analizzati nel riconoscimento.

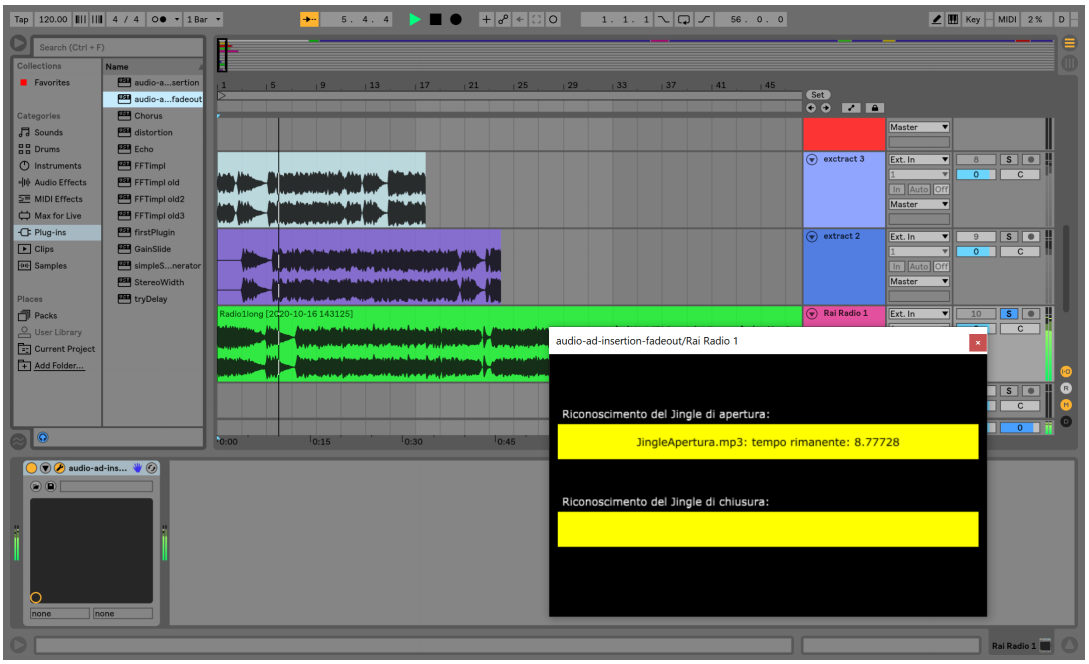

Figura 8.1: Audio-ad-insertion in esecuzione all'interno di Ableton Live, nel momento di riconoscimento di un jingle di apertura, identificato univocamente e corrispondente al file "JingleApertura.mp3" nel DB.

Con questo setup e la logica implementata:

- la lunghezza minima di un jingle affinché possa essere garantito un suo riconoscimento, è pari ad almeno una finestra di matching. Con un $sampleRate = 48KHz$ equivale a 2.73 s, con un $sampleRate = 44.1KHz$ equivale a 2.97 s.

- la lunghezza minima di un'inserzione pubblicitaria può essere anche 0 s, infatti alla terminazione del primo jingle la logica di injection torna subito alla ricerca del secondo, che per assurdo può essere quindi concatenato al primo.
- il nome del file del jingle riconosciuto viene mostrato nella GUI, in anticipo rispetto al momento dell'ascolto (grazie alle azioni di delay), insieme al tempo in secondi rimanente alla sua terminazione, che per J1 coinciderà con l'inizio dell'inserzione personalizzata, mentre per J2 con il riaggancio dello stream originale.
- se la pubblicità personalizzata ha una durata diversa da quella generica già presente, viene applicato un fadeout della traccia iniettata, in modo da evitare un taglio brusco al ritorno nello stream di partenza.
- se per qualche raro motivo il jingle di chiusura non dovesse presentarsi o non dovesse essere riconosciuto, allora viene gestito il caso con un timeout. Allo scadere di un conteggio in campioni (inizializzato al momento di riconoscimento del jingle di apertura), pari alla durata massima della pubblicità iniettabile, la logica di injection ritorna allo stream di partenza con fadeout annesso.

In figura 8.1 è mostrato il plugin in azione all'interno di una DAW, al momento del riconoscimento di un jingle di apertura. La soluzione con audio fingerprint è sicuramente più potente ed attuabile in contesti reali rispetto al riconoscimento di marcatori audio (Goertzel) o metadati, aprendo la strada a diverse possibilità future, alcune descritte nel successivo paragrafo.

8.1 Sviluppi futuri

Contenuti On Demand

Il meccanismo dei jingle non è esclusivo per le inserzioni pubblicitarie, lo stesso è infatti applicato ad altri momenti del palinsesto radiofonico, con lo scopo di rendere riconoscibili e familiari elementi ricorrenti, come ad esempio un programma condotto in studio, il giornale orario o l'aggiornamento sul traffico. Data la grande diffusione dei podcast ed in generale dei contenuti on demand, il sistema di individuazione dei jingle implementato potrebbe essere riadattato, per individuare altre sezioni di interesse dello

stream radiofonico, con il fine di estrarne una copia in locale, da rendere automaticamente disponibile su un'eventuale piattaforma web della stessa radio. Questo potrebbe avvenire, come per il riconoscimento di pubblicità, in tempo reale senza l'intervento di alcun operatore. Tramite una ricodifica della sezione audio di interesse (ad esempio un programma di punta della radio in questione) e upload automatico sulla piattaforma di riferimento, è difatti possibile concedere agli utenti la fruizione in differita del contenuto estratto, tutte le volte che lo desiderano.

Analisi statistiche

Abilitare un'analisi continua sullo stream radiofonico in ingresso significa introdurre anche la possibilità di estrapolare in modo dinamico dati a fini statistici, come ad esempio il numero di volte in cui un determinato contenuto viene riprodotto. Per i musicisti indipendenti e per le piccole etichette discografiche è infatti rilevante conoscere quante volte un loro determinato brano musicale è stato proposto in una stazione radio in un arco temporale e in che specifici orari, in modo tale da verificare il rispetto dei contratti pattuiti. L'audio fingerprint in questo caso potrebbe essere riadattato per il riconoscimento di alcuni brani musicali, utilizzando delle finestre di matching più ampie, prediligendo in questo modo la robustezza, data la lunghezza media delle canzoni superiore a quella dei jingle. Lo stesso principio applicato direttamente alle pubblicità interessa le agenzie pubblicitarie che desiderano avere informazioni sulla diffusione di un'inserzione: numeri, orari di riproduzione, contesti di riproduzione all'interno del palinsesto e così via.

Ringraziamenti

Alla fine dei cinque anni di questo percorso universitario che mi ha indubbiamente dato tanto in termini di conoscenze ed esperienze di vita, vorrei porre dei sentiti ringraziamenti.

Ringrazio il Professore Antonio Servetti, per il meraviglioso corso “Elaborazione dell’audio digitale” nel quale ho avuto finalmente l’opportunità di approfondire in modo efficace argomenti di cui nutro forte interesse sin da quando la mia passione per il pianoforte ha incontrato quella per l’informatica, ed in particolar modo per avermi guidato in questo lavoro, con grande professionalità e disponibilità, anche nel mese di Agosto ed in piena pandemia, dandomi sempre preziosi consigli.

Ringrazio Giovanni, Enrico e Rinaldo, talentuosi ingegneri, per avermi spronato a fare meglio e a guardare il lato positivo della realtà, nelle varie sfide che il Politecnico ci ha posto. Tra l’altro, senza di loro il progetto C.A.R.T.E non avrebbe mai visto la luce; non dimenticherò mai il volto di Giovanni sull’Ipad alle 8:30 di mattina, i meeting recap di sabato e le infinite ore di videochiamata facetime per risolvere bug in pair programming.

Ringrazio tutti i docenti, i colleghi e le persone che ho incontrato in questi anni, perché in diversi modi hanno arricchito la mia persona.

Ringrazio i ragazzi del collegio SD, Giacomo, Claudio, Alessandra, Annachiara, Ludovica, Francesco, Leonardo, Giovanni, Federico per avermi fatto sentire sin da subito a casa, per essere stati una seconda famiglia. "Beppe" vi ringrazia.

Ringrazio mio fratello Vincenzo, per essere stato più di un fratello, per

aver sempre creduto in me e avermi sempre aiutato nonostante, negli ultimi anni, la distanza fisica. Il mio più profondo ringraziamento va ai miei genitori, mia madre e mio padre, per avermi insegnato i valori più importanti della vita e per essere stati sempre e dico sempre presenti dandomi massimo supporto ed investendo nel mio futuro. Mi ritengo davvero una persona fortunata e non smetterò mai di ringraziarli per tutto quello che continuano a fare per me.

Ringrazio i miei zii, ed i miei cugini Anna Maria, Maria Letizia, Filippo, Maria Pia, Rossella, Maria Concetta e Antonella, con i quali sono cresciuto ed ho condiviso i momenti più importanti della mia vita.

Consegno virtualmente questa tesi a mia nonna, per essere stata fonte inesauribile di insegnamenti, facendomi crescere in ogni aspetto della mia persona.

Nonna ricordo benissimo quella sera del 23 Aprile 2019, il giorno prima del mio rientro a Torino, quando accompagnandoti a letto dissi "Ci vediamo a Luglio" e tu, con tristezza, rispondesti "Mi raccomando, stai attento". A Luglio non ci siamo più rivisti però oggi ti consegno questa tesi sperando tu possa essere orgogliosa di me, il tuo Francesco.

Bibliografia

[1] A. Bleier and M. Eisenbeiss, "Personalized online advertising effectiveness: The interplay of what, when, and where," *Marketing Science*, vol. forthcoming, 07 2015.

[2] S. Kanoje, S. Girase, and D. Mukhopadhyay, "User profiling trends, techniques and applications," *International Journal of Advance Foundation and Research in Computer*, vol. 1, pp. 2348–4853, 11 2014.

[3] C. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, jan 1949. [Online]. Available: https://doi.org/10.1109/jrproc.1949.232969

[4] V. Goudard and R. Muller, "Real-time audio plugin architectures," 11 2020.

[5] H. de Oliveira and R. Oliveira, "Understanding midi: A painless tutorial on midi format," 05 2017.

[6] ROLI, "Building equator with juce," 03 2018. [Online]. Available: https://juce.com/discover/stories/building-equator-with-juce

[7] T. H. Park, *Introduction To Digital Signal Processing: Computer Musically Speaking*. USA: World Scientific Publishing Co., Inc., 2009.

[8] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965, uRL: `http://cr.yp.to/bib/entries.html#1965/cooley`.

[9] E. O. Brigham and R. E. Morrow, "The fast fourier transform," *IEEE Spectrum*, vol. 4, no. 12, pp. 63–70, 1967.

[10] H. Sorensen, D. Jones, M. Heideman, and C. Burrus, "Real-valued fast fourier transform algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 6, pp. 849–863, 1987.

[11] U. Oberst, "The fast fourier transform," *SIAM J. Control and Optimization*, vol. 46, pp. 496–540, 01 2007.

[12] Z. Gao, P. Reviriego, X. Li, J. Maestro, M. Zhao, and J. Wang, "A fault tolerant implementation of the goertzel algorithm," *Microelectronics Reliability*, vol. 54, p. 335–337, 01 2014.

[13] A. Dabrowski and T. Marciniak, "Canonic goertzel algorithm and drawbacks of various goertzel algorithm formulations," in *2017 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, 2017, pp. 259–262.

[14] D. Jang, C. D. Yoo, S. Lee, S. Kim, and T. Kalker, "Pairwise boosted audio fingerprint," *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 4, pp. 995–1004, 2009.

[15] F. Kurth and M. Muller, "Efficient index-based audio matching," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 16, no. 2, pp. 382–395, 2008.

[16] A. Gramaglia, "A binary auditory words model for audio content identification," https://github.com/a-gram/audioneex, 2014.

[17] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system with an efficient search strategy," *Journal of New Music Research*, vol. 32, pp. 211–221, 06 2003.

[18] A. Pras and C. Guastavino, "Sampling rate discrimination: 44.1 khz vs. 88.2 khz," vol. 2, 05 2010.

[19] P. Cano, E. Batlle, T. Kalker, and J. Haitsma, "A review of algorithms for audio fingerprinting," 03 2003.

[20] C. Neves, A. Veiga, L. Sá, and F. Perdigão, "Audio fingerprinting system for broadcast streams," 01 2009.

[21] Wei Xiong, Xiaoqing Yu, and Jianhua Shi, "An improved audio fingerprinting algorithm with robust and efficient," in *IET International Conference on Smart and Sustainable City 2013 (ICSSC 2013)*, 2013, pp. 377–380.

[22] A. L. Wang, "An industrial-strength audio search algorithm," in *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, 2003, pp. 7–13, in , S. Choudhury and S. Manus, Eds., The International Society for Music Information Retrieval. http://www.ismir.net: ISMIR, October , pp. . [Online]. Available: http://www.ee.columbia.edu/ dpwe/papers/Wang03-shazam.pdf.

[23] M. H. Weik, *Nyquist theorem.* Boston, MA: Springer US, 2001, pp. 1127–1127. [Online]. Available: https://doi.org/10.1007/1-4020-0613-6_12654