POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

Politecnico di Torino

Corso di Laurea Magistrale in

Ingegneria del Cinema e dei Mezzi di Comunicazione

TheMasker: EQUALIZZATORE DINAMICO BASATO SUL MASCHERAMENTO PSICOACUSTICO

Relatore: Prof. Antonio Servetti

Correlatore: Prof. Giorgio Presti

Tesi di Laurea di:

Nicola Degiorgi

Matr. Nr. 290367

anno accademico 2022-2023

Ringraziamenti

A Novembre 2021 avevo determinato di concludere la mia carriera universitaria con un progetto coraggioso, appassionante e stimolante per me, interessante e di valore per le persone. Chiunque mi conosce sa della mia passione dall'adolescenza per la musica e per la produzione musicale. L'imparare a sviluppare un plugin audio è sempre stato un desiderio, fin da quando ho iniziato ad approcciarmi alla programmazione e ai primi codici.

Sono estremamente grato di avere avuto la possibilità di studiare crescendo a livello intellettuale e umano, di inseguire i miei sogni e la fortuna di riuscire a realizzarne qualcuno.

Innanzitutto il primo pensiero va a mamma e papà che mi hanno sempre spronato a sfidarmi, sostenuto compiendo tutti i sacrifici che lo studiare e il vivere fuori sede richiede, e educato in modo da avere una collaborazione e convivenza pacifica con le persone. Tutta la gratitudine che Nicola è in grado di provare per l'universo, per l'essere umano e per la vita è dovuto a voi, al vostro amore e alla vostra dedizione. Abbiamo vinto mamma!!! Abbiamo vinto papà!!!

Un ringraziamento speciale ad Amedeo, con cui ho condiviso il percorso e pensato a ogni particolare del progetto. Grazie per il coraggio, la fiducia, la pazienza, il supporto, l'entusiasmo, la vittoria che ci siamo regalati. Abbiamo vinto!!!

Grazie ai relatori, il prof. Servetti e il prof. Presti, che ci hanno consentito di esaudire questo desiderio e realizzare il progetto con estrema leggerezza, libertà, supporto, guida e disponibilità.

Grazie a Silvia, che mi ha accompagnato per la gran parte del percorso di Magistrale, per il costante supporto e per la comprensione sempre dimostrata. Per il rapporto che abbiamo avuto e che conserviamo con coraggio, cura e tanta armonia e pace. Ci sei stata sempre, non potevi mancare alla fine di questo percorso. Ti voglio bene e te ne vorrò sempre. Abbiamo vinto!!!

Grazie a Riccio e Chia, che per tutto questo tempo ci sono stati, fonte di supporto, guida e svago. Siete bellissimi e siete speciali. Non dimenticherò mai questo bellissimo periodo di vita torinese, credo

che li ricorderemo come "gli anni d'oro". Ringraziamoci per questo presente. Come dice la parola stessa, è un regalo... quindi grazie.

Al mio maestro Daisaku Ikeda, al Gohonzon e alla famiglia Soka con cui condivido la missione di kosen rufu *e la fede che mi ha permesso di sfidarmi nel progetto e in tutti questi anni di università. Un grande grazie a tutti i compagni di fede, grandi e piccini: Ornella, Annick, Claudia, Maurizia, Eugenio, Eliana, Alessia, Veronica, Sabrina, Marilisa, Simona, Rita, Paola, Michele, Fabio, Piera, Rick, Dex, Alessandro, Alessio, Lisa (e Salva!), Samu, Vale, Giulia (e Nicole!), Giulia LiBa, Arge, Andrea Yuji, Sara, Simone, Alessadnro Cassutti, Giuseppe, Salvo, Giulia Campa, Simone, Gabri, Raffa, Luce, Carlo, Mirko, Valentino, Giuppy, Erica, Joy, Moffi, Alice, Elena, Beatrice, Laura. Abbiamo vinto!!!*

A nonna Nì, che non è riuscita a essere ancora tra noi per assistere a questa vittoria, per la quale avrebbe gioito tanto. Un po' speravo riuscissi a vedere come nonostante tutto anche il monellino di famiglia è riuscito a laurearsi a pieni voti e a trovare subito una buona occupazione. L'ho fatto anche per te. Riposa in pace

Alla famiglia materna e a quella paterna, da cui ricevo sempre tanto amore e sostegno. A Marta, Silvana, Livio, Piera, Amber, Dan, Fran (& families!), Patti e Luca, Annamaria, Marco & family, Silvia & family, Igna. A Carla, Marco, Francesca, Sandra, Luca (& family!), Aldo e Loriana, Chicco e Luisa.

A Nunzio (ormai quasi collega) e Lisa, che sono diventati la mia seconda famiglia al Nord, e che nonostante tutto continuano a volermi bene e sostenermi come fossi un terzo figlio. Vi voglio bene.

A Paolo e Zoe, che sono stati divertentissimi compagni di parte di questa Laurea Magistrale, e che stanno vincendo negli Stati Uniti. Vi abbraccio da qui. E mi raccomando, già che siete lì, State Uniti. Ok basta, scherzo.

Ad Alina, guerriera zen. Grazie per il sostegno di questo ultimo periodo di stress, sei una compagna preziosa e sono sicuro che sarai sempre sicuramente quantomeno una saggia guida per il futuro. Ho tanta stima di te, sei grande, ti voglio bene.

A Carla, saggia leonessa. Sei una persona incredibile, conoscerti è stato scoprire una specialità e una preziosità che hanno allargato il mio amore per gli esseri umani. Non cambiare mai, ti voglio bene.

A Mamu, per il rapporto unico che condividiamo anche a distanza, come se non passassero i mesi tra una chiacchiera e l'altra. Sono sicuro che vincerai in qualsiasi cosa desideri. Prenditi cura di te fratello. Ti voglio bene.

A La Palma Ent., a Less, Pak, Mirko, Grecu, Gabri Romano, Maistu, Spoli, Limo. Siete fortissimi

e spero davvero che vinceremo tutti assieme. Oggi uno di voi ha portato a casa una grande vittoria, spero di aver reso fiera tutta la squadra. Vi voglio bene.

Agli amici producer (e tester del progetto!) Glitchy, Ares, Cruel, Milvio, Goosaif, Tox, Colo. Condividere la passione della produzione musicale con voi ha portato fuori dalla cameretta questo sogno, che con questo progetto di tesi sento che abbia portato a una grande realizzazione, impensabile per me. Spero che anche voi abbiate vissuto con piacere questa condivisione. Grazie di cuore.

Ai colleghi e amici Marzio, Gabri, Nico, Lollo, Brando, Cinio, Bobbe, Antonio, Vlad, Drew, Andrea. Grazie per questi anni che ricorderemo con immenso piacere. I cinemini party, i caffè alle macchinette, i pranzi da 30 e Lode. Questa è la nostra storia e non vi dimenticherò mai. Cerchiamo di non perdere i legami ora che non abbiamo più una scusa valida palese per vederci! Vi voglio bene.

Ad Artin, Youna, e famiglie, per avermi accolto con calore nelle loro case in questa esperienza divertente ed estremamente formativa. Ne conserverò con piacere il ricordo. Moteshakeram!

Alla S.I.T.E., a Matte, Davi (anche tester!), Ciccio, Dimi, Giorgio, MAtti, Giulia e ad Angelo. Grazie per il tempo condiviso e tutte le risate. Non perdiamoci di vista!

Alla zona. Ad Andre, Ribi, Alepià, Pea, Anzo, Borsi, Claudiò, Davi, Juba, Albi, Coro, Lillo, Fedegò. Siete da sempre la family acquisita, vi voglio bene. Oggi un altro di voi ha portato a termine una grande vittoria. Se guardiamo a 10 anni fa, sarebbe sembrato uno scherzo. Nessuno ci avrebbe dato un €. E invece... Vorrei che questo episodio ci aiutasse a ricordare che niente è impossibile.

Abbiamo vinto!!!

Abstract

Il seguente elaborato ha come oggetto l'ideazione, la progettazione e lo sviluppo di un plugin audio, inteso come software esterno integrabile in una qualsiasi Digital Audio Workstation (DAW). Il software sviluppato prende il nome di TheMasker, e appartiene alla sottocategoria dei Virtual Studio Technology (VST), avendo dunque la funzione di elaborazione di un segnale.

L'idea di nasce dalla constatazione di una carenza nel mercato di software che tengano conto esplicitamente del mascheramento in frequenza. Esso è un fenomeno psicoacustico ben noto molto importante nella fase di mixing e mastering, sia essa musicale o di sonorizzazione di un prodotto visivo. Il lavoro del mix engineer consiste infatti in gran parte nella gestione della compresenza armoniosa delle diverse tracce, e il mascheramento costituisce un ostacolo di rilievo.

L'obiettivo ricercato è quello di permettere la visualizzazione e compensazione di tale fenomeno in real-time, mediante un'interfaccia di facile comprensione e comodo utilizzo. In particolare si vuole permettere il confronto tra il segnale mandato in input e un segnale esterno, collegato tramite catena laterale (sidechain). Con l'opportuna modulazione di quest'ultimo si ottiene una stima del mascheramento che esso introduce sul segnale in input. La curva calcolata diventa poi la soglia sulla quale effettuare una equalizzazione dinamica.

L'esigenza a cui si intende porre rimedio è dunque: *Quali parti dello spettro del segnale sono mascherate dal segnale secondario scelto? Come suonerebbe se compensassimo tale mascheramento con un filtro che cambia in real-time, in modo da ridurlo? Come suonerebbe se invece lo volessimo accentuare?*

Indice

CHAPTER

1 Introduzione

La fase di missaggio di un qualsiasi prodotto sonoro ha come obiettivo principale il permettere la coesistenza armoniosa e l'intelleggibilità delle diverse componenti dello stesso. Il lavoro del mix engineer è proprio quello di saper garantire un suono coerente con ciò che l'artista vuole trasmettere e valorizzare tutte le componenti del brano o del prodotto sottopostogli.

Il mascheramento in frequenza, noto fenomeno psicoacustico che approfondiremo in seguito, è particolarmente rilevante in questa fase in quanto influenza in maniera determinante la percezione delle componenti spettrali degli strumenti, o delle tracce della composizione sonora. La compresenza degli elementi sonori, siano essi strumenti musicali, voci, foley, o suoni sintetizzati, influenza la percezione degli stessi.

Attraverso la passione per la produzione musicale si è avuto modo di constatare l'importanza di tale fenomeno nel lavoro di elaborazione di una composizione musicale, una sonorizzazione di un prodotto visivo, e un qualsiasi prodotto sonoro che richieda la compresenza simultanea di più sorgenti. L'idea dello sviluppo nasce dall'importanza nel mix del fenomeno del mascheramento e dalla definizione scientifica dello stesso.

Grazie a numerosi studi precedenti e preesistenti svolti negli scorsi decenni, siamo in grado di quantificarlo precisamente e riprodurre - o meglio, prevedere e prevenire - il suo effetto, attraverso il calcolo numerico e la programmazione.

1.1 L'idea

TheMasker nasce come un analizzatore ed equalizzatore dinamico la cui soglia - frequenza per frequenza - viene calcolata automaticamente, tramite un modello psicoacustico che rappresenti le componenti non udibili di un segnale esterno in entrata.

Definiamo come *input* il segnale su cui si vuole operare un bilanciamento timbrico e come *sidechain* un segnale mandato al nostro plugin sull'entrata della catena laterale (o sidechain, appunto).

In primo luogo dunque l'utente ha modo di vedere in quali zone i due segnali "confliggono", ovvero condividono la stessa banda, e in cui dunque ha vita un potenziale mascheramento in frequenza. Questo aspetto è fondamentale in quanto l'intento del software è proprio quello di esplicitare questo fenomeno, spesso non noto ai neofiti dell'elaborazione del suono, ma estremamente importante.

Inoltre, grazie all'azione dell'equalizzatore dinamico, l'*input* verrà filtrato in base al mascheramento introdotto dal *sidechain*, provato a quantificare in real time in ogni frequenza dal nostro modello psicoacustico.

Questo permetterà di andare ad agire, in diversi modi a seconda delle scelte dell'utente, solo su parti di spettro non udibili o mascherate, appunto dal segnale *sidechain* (supponendo la presenza di quest'ultimo in uscita sul mixer).

> *TheMasker si può dunque definire come un equalizzatore dinamico a bande fisse i cui guadagni sono calcolati automaticamente in tempo reale, in base alla curva di mascheramento relativa a un segnale sidechain esterno, e in base al settaggio dei parametri scelto dall'utente.*

1.2 L'utilizzo previsto

È consigliato l'uso di TheMasker per le fasi di mix e di stem mastering di una traccia musicale.

Quando sommate sul mixer, le tracce, siano esse relative a singoli strumenti o gruppi di tracce (bus), facilmente si mascherano a vicenda. Questo è il motivo per cui nasce la cosiddetta tecnica del "ducking"[1], volta a far emergere un elemento del mix rispetto a un altro, attenuato in presenza del primo. L'utilizzo primario previsto per il plugin è proprio quello di attenuare un elemento nel momento in cui "sta per superare", ovvero è prossimo al mascherare, un secondo elemento.

Ad esempio, spesso nelle basse frequenze (sotto i 150 Hz) si sperimenta un fastidioso conflitto tra grancassa (o kick) e basso: la presenza simultanea delle componenti più basse in frequenza di questi due strumenti genera spesso percezione di confusione e poca definizione di entrambi.

Una tecnica usuale è infatti quella di inserire un compressore nella traccia del basso, pilotandolo con la magnitudine della traccia della grancassa. In questo modo si può aggiustare la soglia (in dB) in modo che questo venga compresso solo quando la grancassa supera tale livello in dB. Ciò permette di ottenere una maggiore chiarezza della cassa, senza di fatto percepire eccessive mancanze da parte del basso, che torna al suo livello naturale subito dopo che la cassa torna sotto la soglia scelta.

Questo metodo si può rendere ancora più preciso se eseguito con un compressore multi-banda, facendo agire la compressione solo nelle frequenze basse.

L'intento da cui nasce il suddetto software è quello di automatizzare la scelta delle frequenze e della soglia, in maniera che venga evitata la perdita di chiarezza di una traccia rispetto a un'altra dovuta al mascheramento. L'automatizzazione è quindi basata su criteri psicoacustici precisi, e di conseguenza ritenuta affidabile.

Tornando all'esempio, il plugin si potrebbe far lavorare, con funzione esclusivamente di compressione, sul basso, andando dunque a sostituire il compressore. A differenza di

[1] Ducking: In sound design, il ducking è un effetto audio comunemente utilizzato nella musica radiofonica e pop, soprattutto nella musica dance. Nel ducking, il livello di un segnale audio viene ridotto dalla presenza di un altro segnale.[1]

quest'ultimo, TheMasker valuterà automaticamente in quali istanti e in quali frequenze il basso viene superato dalla cassa ed eseguirà un bilanciamento timbrico (di sola attenuazione) in modo da non permettere al basso di raggiungere una soglia tale da causare il mascheramento della stessa. L'elemento percussivo sarà quindi sempre prevalente rispetto allo strumento a corde, senza avere minima ripercussione nei momenti in cui la traccia della cassa è silenziosa.

Un altro utilizzo previsto per TheMasker è quello di enfatizzare le frequenze che vengono superate e mascherate dal *sidechain*. La conseguenza di questa elaborazione è il tendere all'evitare tale superamento.

Nell'esempio precedente, anche se non sarebbe la scelta ottimale per il problema iniziale della poca definizione del low-end dei due elementi, si potrebbe farlo lavorare sulla cassa, settando però i parametri in modo da lavorare esclusivamente in espansione. La percussione sarebbe dunque enfatizzata, specie nelle basse frequenze. A seconda del dosaggio dei parametri, tale enfatizzazione "non permetterebbe" alla cassa di essere superata dal basso. Come già detto, però, non sarebbe la soluzione ottima del problema.

Un esempio più adatto potrebbe essere il far lavorare il software su una traccia dai livelli tendenzialmente bassi, ad esempio un violino, che si vuole far prevalere rispetto a una seconda traccia, ad esempio un coro. Quest'ultimo in alcuni momenti potrebbe sovrastare il violino.

Facendo in modo che si abbia esclusivamente un'espansione (tramite i parametri) si potrebbe applicare al violino l'effetto TheMasker, pilotandolo con il coro, per tirarlo su nel momento in cui quest'ultimo tende a mascherarlo. Si potrebbe dunque apprezzare una maggiore chiarezza nell'arco.

I parametri modificabili dall'utente andranno comunque a favorire un discreto livello di controllo dell'intervento, in modo da rendere il plugin adatto a elaborazioni più evidenti del timbro, come nel caso di un mix, così come a un effetto meno evidente e quasi trasparente, come spesso si intende lavorare in fase di mastering a stems. Qui si potrebbe voler "portar fuori" un gruppo di tracce (bus) senza intervenire eccessivamente, e quindi snaturare l'idea del missaggio degli strumenti.

1.3 Gli obiettivi

Possiamo fare una distinzione tra gli obiettivi del progetto relativamente ai diversi macroblocchi di cui si compone l'algoritmo pensato. Ne individuiamo tre:

1. In una prima fase occorre costruire un **modello psicoacustico** adeguato e scientificamente giustificato, in modo da costruire in modo più accurato possibile la soglia di mascheramento del segnale *sidechain*.

2. Successivamente bisognerà effettuare l'**elaborazione del segnale**, dipendente dai parametri esposti all'utente nell'interfaccia, senza inficiarne la qualità e facendo in modo che gli interventi risultino coerenti con i parametri scelti.

3. Infine occorre esporre all'utente un'**interfaccia** dotata degli strumenti adeguati per la comprensione, l'uso e la valutazione dell'elaborazione offerta dal plugin.

1.3.1 Gli obiettivi: il modello psicoaustico

I principali obiettivi della prima fase sono quelli di costruire un modello psicoacustico che andrà a rappresentare il mascheramento introdotto dal *sidechain*. Il risultato sarà una curva, che chiameremo *threshold* che andrà poi a essere comparata (con una differenza) con l'*input*. Tale differenza sarà la curva che andrà a pilotare l'equalizzatore dinamico, che definiamo *delta*. L'analisi psicoacustica deve essere in grado di fornire in output una curva:

- In tempi assimilabili al real-time (con latenza di massimo 80 ms)

- Costituita da un numero di punti in grado di rappresentare il segnale analizzato con sufficiente risoluzione [2]

- In grado di rappresentare la soglia di mascheramento introdotto dal segnale accuratamente sia nei valori di magnitudine (asse y) che di frequenza (asse x)

[2] Risoluzione minima: il numero di bande critiche secondo la teoria posizionale sviluppata da Hermann Von Helmoltz (metà '800) è 25 [2]

1.3.2 Gli obiettivi: l'elaborazione del segnale

Il segnale verrà poi elaborato a partire dal *delta* attraverso un'equalizzazione dinamica. Il processing del segnale dovrà:

- Essere abbastanza rapido da fornire un segnale in uscita in tempo reale (tempi inferiori ai 10 ms)
- Essere coerente con il settaggio dei parametri scelto dall'utente
- Garantire il più possibile la linearità di fase, il preservamento della qualità del suono e l'assenza di glitch, distorsioni e artifici digitali
- Fornire in uscita un segnale che presenti un'equalizzazione effettivamente utile per gestire il mascheramento causato del segnale *sidechain*.
- Avere un andamento nel tempo smussato e liscio, in modo da non avere variazioni repentine dell'effetto dell'elaborazione e garantirne la gradevolezza

1.3.3 Gli obiettivi: l'esperienza utente

La User Experience (UX) del plugin dev'essere gradevole, leggera e comoda. Attraverso l'interfaccia l'utente deve essere in grado di intuire le funzionalità offerte, metterle alla prova senza perplessità e valutarle attraverso una analisi del segnale in uscita. Essa deve dunque:

- Garantire la comprensibilità dell'analisi del segnale, dello stato dell'elaborazione e, in generale, del funzionamento del software
- Seguire la dinamica dei segnali e dell'elaborazione nel tempo in maniera comprensibile
- Permettere un controllo dei parametri consapevole, comodo e adeguato alla funzione proposta
- Fornire un'esperienza visiva coerente e gradevole

CHAPTER

2 Stato dell'arte

Competitor e comparable

Nel mercato attuale esistono innumerevoli plugin che si occupano di equalizzazione dinamica, intesa come equalizzazione con parametri di guadagno dei filtri dipendenti dalla magnitudine del segnale in entrata. Solo una piccola parte di essi basano la propria soglia sul fenomeno del mascheramento, attraverso un ulteriore accorgimento riguardo alla magnitudine dell'input, o segnale sidechain esterno designato per pilotare il gain dei filtri. Invece che utilizzare il valore di magnitudine puro, esso si può usare per definire una curva costruita secondo un modello psicoacustico: questa, rispetto al segnale puro, vedrà i propri picchi discendere con pendenza più lenta nel dominio della frequenza e un ammontare inferiore. Questo piccolo passaggio rappresenta più accuratamente la variazione di sensibilità che l'orecchio umano accusa durante la ricezione ed elaborazione di un suono mediante l'apparato uditivo.

Ciò che ci si è posti come obiettivo a inizio progetto è stato proprio il rendere esplicita la considerazione del mascheramento in frequenza, e farne un punto di forza. Anche il nome "TheMasker" è stato scelto appositamente per garantirne l'accessiblità già dal primo incontro col software,in quanto esso vuole esplicitamente essere un de-mascheratore di segnali simultanei.

La sezione seguente è dedicata all'analisi dei principali competitor e comparable che sono stati individuati tramite una ricerca sui principali motori del web.

In particolare, i seguenti paragrafi sono dedicati a 5 plugin in commercio aventi caratteristiche simili a quelle pensate per TheMasker.

2.1 FabFilter - Pro-Q 3

Un grande esempio di equalizzatore digitale, che nella terza release diventa anche dinamico, è costituito dalla linea Pro-Q della FabFilter.

In particolare la terza e più recente versione, chiamata FabFilter Pro-Q 3, presenta feature aggiuntive molto comode e utili in tanti diversi processi del sound design, dalla produzione, al mixing, al mastering. Attualmente questo equalizzatore è tra i più utilizzati in tutto il mondo, sia a livello amatoriale che professionale, per via della semplicità e al contempo della versatilità e qualità delle features che presenta.

Figura 1: FabFilter Pro-Q 3 user interface

Sebbene l'aspetto dinamico del plugin sia ben curato, Pro-Q3 rimane ancora oggi uno strumento maggiormente di equalizzazione manuale e statica; inoltre, anche nell'ambito dell'equalizzazione dinamica, l'accento è prevalentemente spostato sula modifica della curva di equalizzazione da parte dell'utente: esso può decidere precisamente l'intervallo di frequenze su cui andrà a operare l'equalizzazione dinamica. La differenza principale risiede quindi

nel fatto che non avviene una equalizzazione dinamica obbligatoriamente sull'intero spettro, e soprattutto non in conseguenza a un calcolo della soglia di mascheramento del segnale sidechain. Per fare un esempio, dato un segnale sidechain con due forti picchi alle frequenze di 500Hz e 2000Hz, l'utente può decidere senza problemi di ignorare l'equalizzazione nel primo intorno di frequenze, ma invece di operare con abbondanza sul secondo: la direzione del progetto TheMasker è diversa, in quanto punta a una minore personalizzazione della curva, ma a una gestione automatica sull'intero spettro.

2.2 OekSound - Soothe 2

Figura 2: OekSound - Soothe 2 user interface

A prendere una strada più specifica rispetto al Pro-Q3 e più vicina a quella interessata è Soothe 2 della OekSound, altro prodotto molto popolare nel mondo della produzione musicale. Lo scopo per cui nasce questo plug-in è quello di "ripulire", in modo automatico, il segnale da

suoni indesiderati: risonanze non volute, asprezza delle sibilanti in una registrazione vocale, effetto di prossimità ecc. Oltre questi aspetti, che implicitamente riguardano un singolo segnale audio, Soothe 2 dà anche la possibilità di operare con un segnale sidechain ed eliminare i conflitti tra le frequenze dei due segnali paralleli.

Soothe 2 è stato il principale riferimento seguito per lo svilupppo di TheMasker, in quanto effettivamente condivide con esso gran parte delle caratteristiche. Tuttavia, nella documentazione disponibile non vi è alcun cenno relativo al fenomeno del mascheramento. L'effetto di pulizia del segnale sarà quindi fondato su basi diverse e di conseguenza parametri e configurazioni si differenzieranno, e sarà interessante analizzare i risultati di TheMasker confrontandoli, se possibile, con Soothe 2. Soothe 2 rimane comunque uno dei pochi reali competitors di TheMasker (consideriamo Pro-Q3 come comparable poiché, pur lavorando sull'equalizzazione dinamica non ha una gestione automatica delle bande di frequenza come intesa in questo studio), inoltre, il target a cui si riferisce è il settore professionale: per quanto non andremo ad analizzare il mercato, è bene comunque essere coscienti della minore accessibilità di Soothe 2, dato il suo valore economico, e della mancanza di valide alternative per il mondo dell'amatoriale e dell'home recording.

2.3 Altri comparables

I precedenti esempi sono i riferimenti maggiormente seguiti nello sviluppo del plugin, non solo per logiche di operazione, ma anche per scopi, qualità, risultati e scelte stilistiche. Seguono, comunque, altre menzioni d'onore, molto utili almeno per gli obiettivi parziali o per alternative che si è deciso di non seguire.

2.3.1 Hornet Plugins - Multifreqs

Parlando di obiettivi parziali, Multifreqs è un ottimo confronto per quanto riguarda l'analisi del segnale. Il plugin è, infatti, un dettagliato analizzatore di spettro, che permette il controllo contemporaneo di segnali paralleli per scoprirne le frequenze in conflitto. User friendly e accessibile, è un buono strumento per la prima parte del lavoro di equalizzazione, ma non si occupa del sound processing: obbligatoriamente deve seguire un altro plugin di equalizzazione dinamica per il risultato desiderato.

Figura 3: Hornet Plugins - Multifreqs

2.3.2 Melda Production - MAutoDynamicEQ

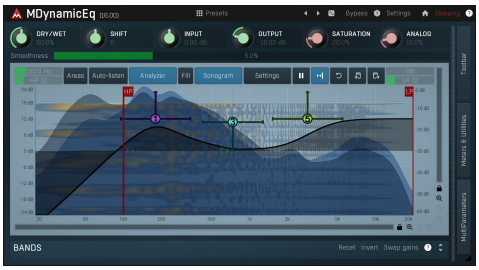


Figura 4: Melda Production - MAutoDynamicEQ

MAutoDynamicEQ è altro plugin che tende allo scopo desiderato, tuttavia manca di un fattore molto importante: l'analisi e l'esecuzione in tempo reale. MAutoDynamicEQ dispone, infatti, di un analizzatore di spettro che individua le frequenze in conflitto fra due segnali e in seguito ne permette un'equalizzazione dinamica. Si può vedere come la risultante dell'utilizzo seriale di Multifreqs, per analizzare, e di ProQ-3 per equalizzare. Da

non dimenticare la complicazione dell'interfaccia utente: il plugin dispone di innumerevoli parametrizzazioni e personalizzazioni, che da un lato permettono maggiore controllo, dall'altro richiedono all'utente una conoscenza approfondita non solo del plugin ma dell'argomento del mascheramento e dell'equalizzazione. La scelta è comprensibile, tuttavia, è nel nostro interesse cercare di riuscire a ridurre i parametri e rendere il più possibile automatico senza la preoccupazione dell'utente.

2.3.3 Wavesfactory - TrackSpacer

Un altro plugin professionale e con un'idea molto simile a TheMasker. Il paragone non è molto diverso da quello fatto con Soothe 2: l'obiettivo è simile e la qualità è ottima, ma di nuovo non si cita il mascheramento (il sito definisce il suo funzionamento "come per magia"). TrackSpacer potrà, anch'esso, essere preso come confronto per i risultati ottenuti da TheMasker.

Figura 5: WavesFactory - TrackSpacer

CHAPTER

3 Tecnologie utilizzate e basi teoriche

Il seguente capitolo si propone di presentare le scelte in quanto a tecnologie e tools utilizzati, riportare brevemente le basi teoriche su cui si basa il software e illustrare nel dettaglio la struttura scelta per l'algoritmo.

3.1 Organizzazione progetto e tools utilizzati

Per quanto riguarda la prima fase di schematizzazione dell'algoritmo ci si è mossi tramite la piattaforma Miro [3], che permette lo sviluppo di diagrammi a blocchi direttamente sul browser.

Successivamente si è passati a sviluppare un prototipo del software nell'ambiente (e linguaggio) MATLAB [4], andando a sviluppare la struttura del backend del plugin, le funzioni e le operazioni matematiche in grado di effettuare l'elaborazione designata. Il plugin vero e proprio è stato poi sviluppato tramite JUCE [5], un framework per applicazioni C++ multipiattaforma open source. Grazie alle sue librerie è ottimizzato per lo sviluppo di interfacce grafiche e plugin, e si è dunque potuto sviluppare un'applicazione in grado di elaborare il segnale in real-time.

3.1.1 Schema a blocchi - Miro

Miro è una piattaforma browser gratuita che offre numerosi tool di creazione di mappe mentali, schemi, diagrammi a blocchi, grafici e tanto altro. Avere un progetto condiviso e aggiornato in

tempo reale sulle modifiche effettuate ha permesso una collaborazione comoda, la definizione della struttura e di alcuni passaggi ancora non chiari, e ci ha dunque portato ad avere una base su cui sviluppare poi il prototipo.

Attraverso questo tool è stato infatti sviluppato un diagramma a blocchi [6] in grado di andare nel dettaglio dell'algoritmo da sviluppare: è stato rappresentato il flusso del segnale dall'inizio (input) alla fine (output), andando nel dettaglio delle funzioni da creare, delle operazioni matematiche e delle variabili necessarie a tali scopi.

3.1.2 Prototipo - MATLAB

MATLAB è un ambiente di calcolo numerico scritto in linguaggio C. I numerosi tools di cui dispone permettono di affrontare problemi tecnici e simulazioni su diversi campi, tra cui anche l'elaborazione dell'audio digitale. L'obiettivo del prototipo è l'analisi dei segnali audio in diverse fasi del ciclo di elaborazione, tramite grafici e file audio di output. L'ambiente è ottimo per la prototipazione, in quanto permette una gestione semplice e lineare dei segnali audio, senza le complicazioni strutturali e sintattiche che saranno necessarie per la scrittura di un codice funzionante in tempo reale. Questa infatti, sarà necessaria solo nello sviluppo del codice finale.

3.1.3 Sviluppo - JUCE

La creazione del codice finale si è usato JUCE, framework open source in linguaggio C++ molto comune nell'ambito dell'elaborazione dell'audio digitale per le sue numerose librerie di gestione dell'audio digitale e le Graphical User Interface (GUI). L'obiettivo iniziale è stato quello di tradurre il prototipo nel nuovo linguaggio e ottimizzarlo per renderlo il più leggero possibile a livello computazionale. Inoltre, il framework ha permesso anche la comoda gestione della GUI. Un altro vantaggio dell'utilizzo di JUCE è la possibilità di compilare il codice in diverse versioni, tra cui standalone (applicazione che è possibile avviare senza il bisogno di alcun software esterno) oppure nie formati .vst, .vst3 e .AU, i formati più comuni dei plugin audio, utilizzabili nelle più comuni DAW.

3.2 Struttura generale dell'algoritmo

Di seguito si illustrerà e spiegherà il diagramma a blocchi raggiungibile tramite il seguente link. Esso contiene due sottosezioni, dedicate rispettivamente all'algoritmo di analisi psicoacustica e di elaborazione del segnale, che verranno mostrate e approfondite in seguito.

3.2.1 Legenda

Il colore e il tratto delle frecce, così come la forma delle loro terminazioni o la forma dei blocchi è stata scelta in base a una specifica simbologia.

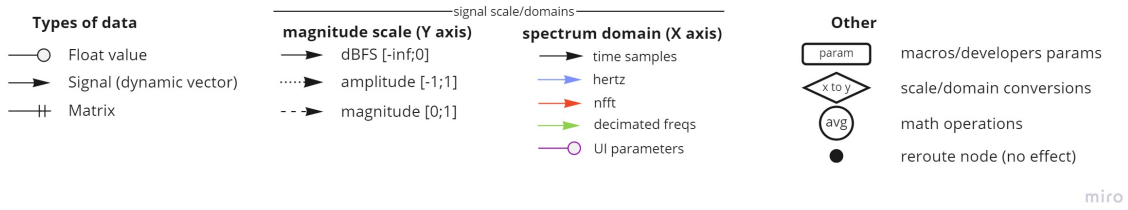


Figura 6: Legenda dello schema Miro

- **Terminazioni delle linee**: Tipi di dato
  - **Terminazione con pallino vuoto**: valore float scalare
  - **Terminazione con freccia**: vettore (di float)
  - **Terminazione con doppia linea**: matrice (di float)
- **Tratto delle linee**: Scala (dell'ampiezza) dei dati
  - **Tratto continuo**: dBFS [-inf;0]
  - **Tratto punteggiato**: Ampiezza [-1;+1]
  - **Tratto tratteggiato**: Magnitudine [0;+1]
- **Colore delle linee**: Dominio dei dati

  - **Nero**: Dominio del tempo
  - **Azzurro**: Dominio delle frequenze (Hz)
  - **Rosso**: Dominio dei punti Fast Fourier Transform (FFT) (val. scalari)
  - **Verde**: Dominio delle frequenze ”decimate” (Hz)
  - **Viola**: Parametri esposti nell’interfaccia (val. scalare float)

- **Forme dei nodi**: Tipi di operazione
  - **Rettangolo smussato**: Macro e parametri per gli sviluppatori
  - **Rombo**: Conversioni di scala o dominio
  - **Cerchio**: Operazione matematica (Hz)
  - **Cerchio nero pieno**: Nodo di rerouting (non ha effetto, ha una funzione puramente estetica volta a dare ordine al grafico)

3.2.2 Il flusso del segnale

Lo schema alla pagina seguente rappresenta il flusso del segnale di *input* e di quello in entrata in *sidechain*. In uscita in seguito alle dovute elaborazioni si ottiene il segnale di *output* e le curve da mostrare nell’interfaccia, qui chiamata ”UI out”. Esso mostra come i segnali di *input* e di *sidechain* (così come in uscita il segnale di *output*) vengono moltiplicati per un fattore esposto all’utente nell’interfaccia. Rispettivamente, si tratta dei guadagni (Gain) che definiamo come ”*inputGain*”, ”*sidechainGain*” e ”*outputGain*”.

Il *sidechain* ha l’unica funzione di definire, per ogni blocco di campioni (di conseguenza in tempo reale), una curva che andrà poi a determinare l’elaborazione del segnale. Essa è qui chiamata ”theshold”, ed è frutto dell’Analisi psicoacustica, che verrà illustrata in seguito. Tale soglia stima la curva di mascheramento che il segnale *sidechain* (dopo il rispettivo guadagno definito dall’utente) produce sull’ascoltatore.

L’elaborazione del segnale di basa su tale threshold e, chiaramente, accetta in entrata il segnale di *input*. Il segnale risultante (qui ”wet signal”) viene poi (dopo il rispettivo guadagno) mandato in uscita, sia in quanto segnale, che in quanto spettro in frequenza.

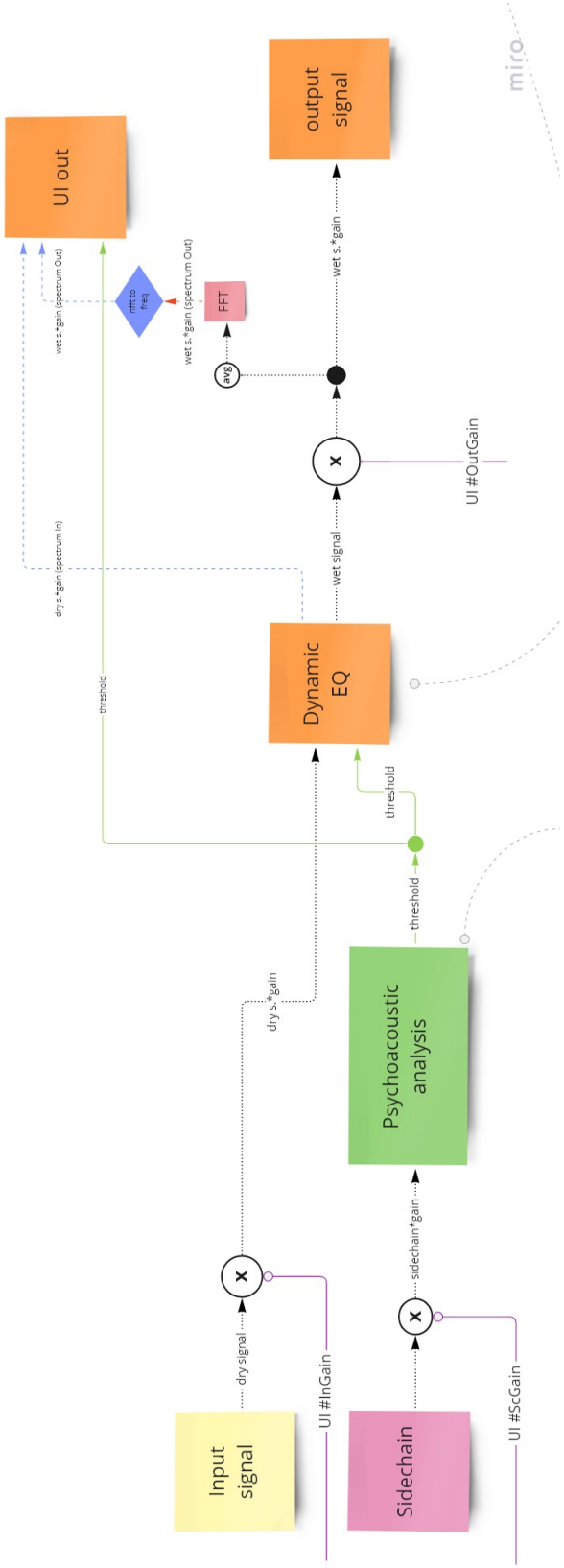


Figura 7: Struttura generale del plugin

Dopo un'operazione (avg) di media tra i due canali (supponendo un segnale stereo), si nota infatti un processo presente su più punti dell'algoritmo: il segnale viene processato tramite un'operazione di FFT e vengono poi convertiti i valori rappresentanti la magnitudine nei "punti FFT" in valori rappresentanti la magnitudine in frequenze specificatamente definite proceduralmente all'inizio dell'algoritmo, una tantum. Esse saranno utili in diversi punti del software, come vedremo in seguito.

3.2.3 Conversioni e frequenze

Conversioni di scala

Il segnale attraversa alcuni stadi in cui è espresso in una scala lineare, come ampiezza (tra -1 e 1) o come magnitudine (tra 0 e 1 - ottenuta tramite il valore assoluto dell'ampiezza), e altri in cui è espresso in scala logaritmica (dBFS).

La conversioni sono date dalle formule:

$$dB = 20 * \log_{10}(|amp|)$$

e

$$amp = 10^{\frac{dB}{20}}$$

dove amp esprime il valore di ampiezza del segnale e dB il valore in dBFS.

Conversioni di dominio

In alcune fasi i valori degli array che fluiscono nel software rappresentano il segnale espresso nel dominio del tempo (ad es. in entrata e in uscita, in cui è espresso come ampiezza pure, tra -1 e 1). È necessario però, in altri casi, suddividerlo in blocchi ed esprimerne le componenti in frequenza.

Esse sono il risultato di operazioni di FFT, e rappresentano quindi il contenuto nei "*puntiFFT*", ovvero specifiche frequenze (Hz) linearmente distribuite tra 0 e la frequenza di Nyquist:

$$f_{nyquist} = \frac{sampleRate}{2} \tag{1}$$

Tali valori vengono in seguito interpolati (processo ”nfft to freq”) per stimare il contenuto acustico in frequenze scelte appositamente per rappresentare più accuratamente lo spettro rispetto alla psicoacustica umana: definiamo tali frequenze tramite un array che chiameremo ”*frequencies*”. Esse sono comprese tra una frequenza minima (definita a 20 Hz) e una frequenza massima (definita a 20000 Hz). Esse serviranno in diverse occasioni: sono le frequenze prese in considerazione per la definizione della curva di mascheramento, e per tutti i plot degli spettri in frequenza. Definiamo la dimensione di tale array come *npoints*, nel codice definito una tantum come 256.

L’array *frequencies* è formato tramite una distribuzione lineare nel dominio dei bark: la frequenza minima (20 Hz) e la massima (20000 Hz) sono convertire in bark (tramite la funzione hz2bark(2)) e fornite in input a una funzione che crea un array di 256 valori uniformemente distribuiti tra un minimo e un massimo. Tale funzione, in MATLAB nativamente presente come ”linspace”, è stata poi definita anche in C++. Si ottengono così 256 valori in bark uniformemente distribuiti tra i valori, in bark, della frequenza minima e massima. Tramite una riconversione nel dominio delle frequenze (funzione bark2hz(3)) si ottiene l’array *frequencies*.

$$bark = 6 * arcsinh(\frac{Hz}{600}) \tag{2}$$

$$Hz = 600 * \sinh(\frac{bark}{6}); \tag{3}$$

È inoltre importante citare un ulteriore array di frequenze inizializzato all’inizio dell’esecuzione del software, che definiamo come ”*fCenters*”. Anch’esso include valori in Hz compresi tra la frequenza minima e massima definite, ma è formato da un numero molto inferiore di frequenze. Esso è pari alla variabile che definisce quante saranno poi le bande su cui avverrà il filtraggio: per questo tale variabile la definiamo come ”*nfilts*”, pari a 32.

L’array *fCenters* viene creato contestualmente alla creazione della matrice *fBank*, la quale ha la funzione proprio di esprimere in un numero inferiore di punti (*nfilts*) il contenuto

in frequenza espresso nei 256 valori in uscita dalla FFT. Chiameremo "*decimazione*" la moltiplicazione puntuale per *fBank*, che prevede come secondo fattore un array di 256 valori e produce come risultato un array di 32 valori.

Tornando allo schema, le curve già "decimate" (o comunque di dimensione 32) sono rappresentate con una freccia verde, mentre le curve rappresentanti valori in corrispondenza delle frequenze definite da *frequencies* (di dimensione 256) con una freccia azzurra.

Nelle sezioni "Analisi psicoacutica" ed "Elaborazione del segnale" si entrerà nel dettaglio dei blocchi "Psychoacoustic Analysis" e "Dynamic EQ".

3.3 Analisi psicoacustica

3.3.1 Il mascheramento

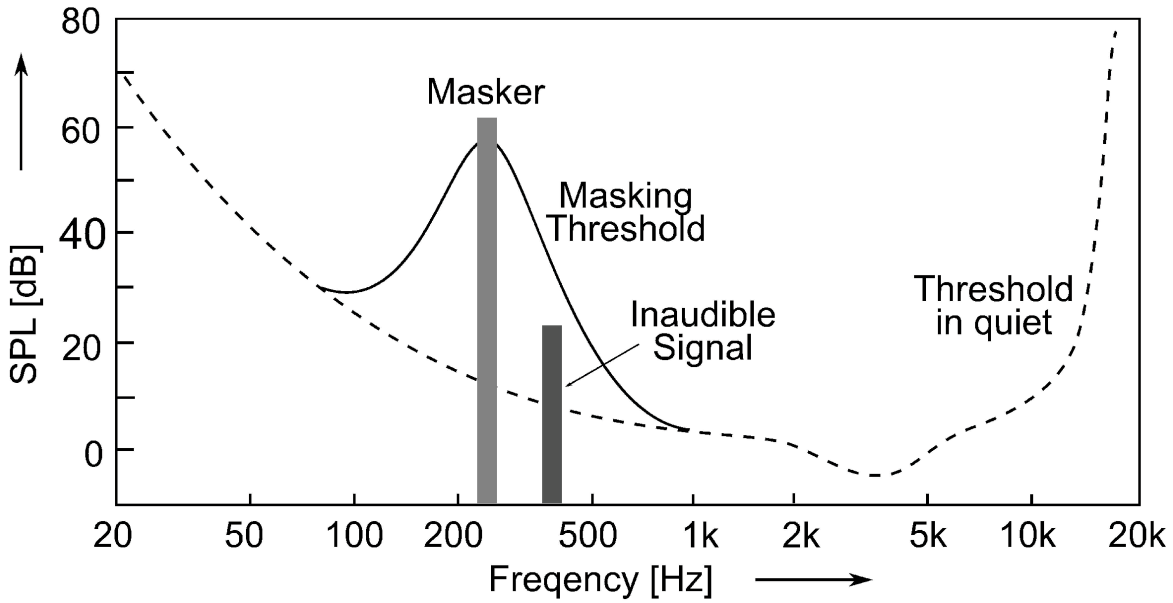


Figura 8: Soglia di mascheramento in frequenza introdotta da un tono puro

Il mascheramento in frequenza è un fenomeno psicoacustico ben noto: le componenti energetiche di un segnale sonoro che percepiamo influenzano la sensibilità al percepire altri segnali, se inferiori al segnale "*mascheratore*" di una certa soglia in dB. Tale scostamento in dB dipende

dalla natura del segnale: ha un valore minore per un mascheratore simile a un rumore (5 dB) rispetto a un mascheratore simile a un tono puro (25 dB)[7]. Essa è chiamata proprio soglia di mascheramento, ed è generalmente rappresentata sia nel dominio delle frequenze che in quello del tempo. Questo perchè il segnale *mascheratore* ridurrà la possibilità di percepire altri segnali (o determinate componenti di essi) se essi rispettano le seguenti condizioni:

1. appartenenti a frequenze adiacenti
2. inferiori al mascheratore di una specifica quantità di dB
3. simultanei al mascheratore, successivi (fino a 100 ms circa), o addirittura precedenti (fino a 20 ms circa) [8]

3.3.2 Scelte adottate e funzionamento

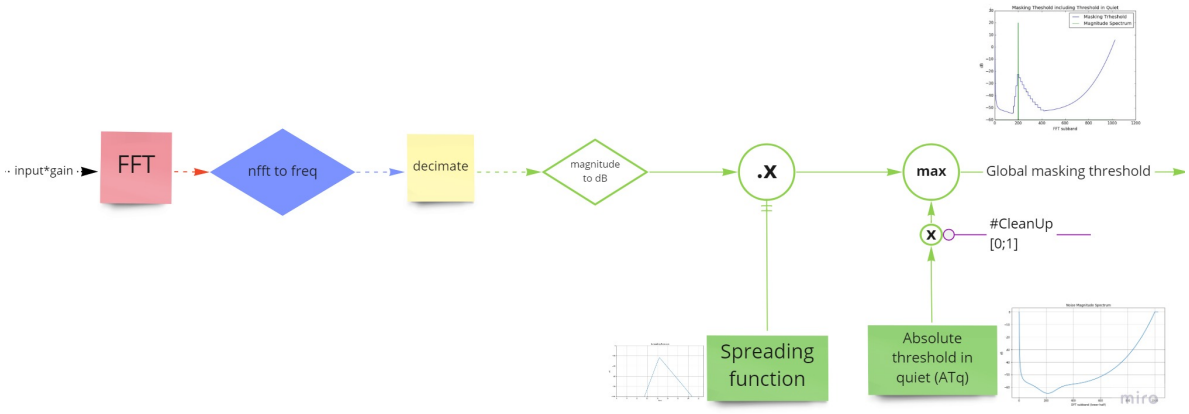


Figura 9: Blocco relativo all'analisi psicoacustica del segnale *sidechain*

Al fine di stimare il mascheramento introdotto dal segnale in *sidechain*, che in questo caso chiameremo *mascheratore*, sul segnale di *input*, ovvero il *mascherato*, in una prima fase si è proceduto considerando il solo *mascheratore*. Come si vede in figura, il blocco "Analisi psicoacustica" accetta in entrata solo il segnale *sidechain*. Innanzitutto attraverso un processo di FFT su un blocco di samples di dimensioni *nfft* (ad esempio 1024) ne viene calcolato il contenuto energetico in frequenza, il cui risultato è un array di $\frac{1024}{2}$ valori. Essi rappresentano

il contenuto acustico in frequenze specifiche, precedentemente menzionate come *puntiFFT*. Tali frequenze sono linearmente distribuite tra 0 e la frequenza di Nyquist(1).

Per ragioni psicoacustiche, però, come precedentemente accennato, tali valori sono poi interpolati per esprimere lo spettro delle frequenze in corrispondenza delle nostre *frequencies*. Si ottiene cosí un vettore di dimensione 256, che viene già mandato in uscita dal blocco per eseguirne un plot nell'interfaccia grafica.

Proseguendo l'elaborazione, tale array viene ulteriormente "interpolato" (moltiplicazione puntuale per la matrice *fBank*) in modo da eseguire la *decimazione* ed ottenere uno spettro di soli 32 valori, corrispondenti alle frequenze *fCenters*.

Avviene poi la conversione dalla scala lineare a quella logaritmica, in quanto da questo punto in poi sarà fondamentale effettuare le operazioni considerando i valori dello spettro espressi non più come magnitudine ma in dBFS.

La matrice di spreading

L'array decimato e convertito in dB viene poi moltiplicato puntualmente per una matrice di dimensione 32 x 32 chiamata *spreading matrix*.

Tale matrice ha la funzione di emulare il comportamento della soglia di mascheramento del modello psicoacustico umano, riportata nell'immagine.

Tale matrice effettua infatti uno "spreading", inteso come allargamento, andando a rendere dipendenti l'una dall'altra le frequenze adiacenti dell'array. In particolare, i valori alti dello spettro, dunque i picchi, trasmetteranno parzialmente il proprio valore alle frequenze limitrofe, cosí come faranno i valori bassi dello spettro. Il risultato è quindi una curva il cui andamento è piú smussato, e piú tendente al lineare, di come era prima dell'intervento della *spreading matrix*. Tale curva possiamo ora chiamare "*relative threshold*", ovvero soglia - di mascheramento - relativa al corrente blocco di samples. Essa interagisce subito dopo con la soglia di mascheramento assoluta, o Absolute Threshold in Quiet (ATQ).

La ATQ, soglia di mascheramento assoluta

Entra ora in gioco la ATQ, la quale viene ”messa a confronto” con la *relative threshold* tramite un operazione di *max()*. Nella curva risultante, che definiamo come *threshold*, i valori sono dunque i massimi tra i rispettivi valori delle due soglie, relativa e assoluta.

Per la formazione della curva di mascheramento è stata creata una funzione polinomiale, che accetta in entrata un array di frequenze (*fCenters*, di 32 valori tra 20 Hz e 20000 Hz) e restituisce in uscita un array di 32 valori in dB, pronti a essere confrontati con la soglia relativa.

Il parametro esposto nell'interfaccia con il nome *cleanUp* consente di regolare il grado di influenza della ATQ rispetto alla soglia relativa, attraverso uno scalamento prima dell'operazione di *max()*. Se tale parametro è nullo, la curva si ”appiattisce” a un valore minimo, che è stato definito come -100 dBFS.

Approssimazioni del modello

Al fine di garantire un costo computazionale sostenibile, il software non discrimina se i massimi locali dello spettro, cioè i picchi, appartengono a un segnale di natura simile al tono puro o se hanno banda larga e natura simile a un rumore. È stata dunque scelto un offset di 10 dB , in modo che fosse una quantità intermedia tra i 5 dB e i 25 dB.

Inoltre, gli spettri dei segnali, così come il calcolo del valore della ATQ, sono considerati in soli 32 valori, interpolando la FFT che restituisce un numero di punti pari a $\frac{1024}{2}$. Tale quantità, ovvero la variabile intera *nfilts*, è stata scelta in modo da essere superiore a 25 [9], che secondo la psicoacustica è il numero di bande critiche dell'orecchio umano, rappresentante il parametro psicoacustico della risoluzione in frequenza dell'orecchio umano. Esse sono definite come le bande di frequenze audio all'interno delle quali un secondo tono interferisce con la percezione del primo tono attraverso il mascheramento uditivo [10]. In altre parole, è l'intervallo di frequenze entro il quale due toni puri simultanei non possono essere percepiti come distinti. Questa scelta è volta a ridurre il peso computazionale del software, in quanto effettuare il filtraggio su 256 sarebbe stato eccessivo.

In ultima istanza, non è stata considerata con accuratezza la componente temporale del mascheramento: nell'arco temporale precedente (20 ms) e successivo (100 ms) al segnale *mascheratore* il mascheramento è sperimentato, ma il modello costruito approssima tale periodo attraverso lo smoothing dei valori del *delta* (attraverso una classe di JUCE di cui si parlerà in seguito) di un periodo di tempo di maggiore (80 ms per l'attacco, 250 ms per il rilascio).

3.4 Elaborazione del segnale

In entrata del blocco qui chiamato *dynamicEQ*, che si occupa della effettiva elaborazione del segnale, è mandato il segnale di *input* e la *threshold* calcolata tramite l'analisi psicoacustica del segnale *sidechain*.

3.4.1 Delta

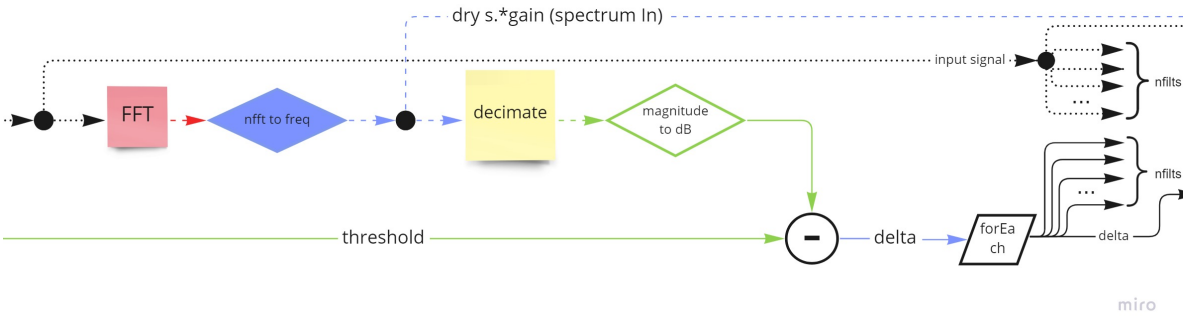


Figura 10: Prima parte del blocco relativo all'elaborazione del segnale di *input*

La prima parte del blocco *dynamicEQ* si occupa di mettere a confronto gli spettri, decimati e convertiti in dB, dei segnali *input* e *sidechain*: il secondo, come abbiamo visto, è già stato preparato con l'analisi psicoacustica, mentre il primo subisce un'elaborazione simile ma priva di *spreading matrix* e confronto con ATQ.

Viene quindi effettuata la FFT di un blocco di 1024 campioni, il risultato viene interpolato per ottenere il contenuto spettrale in corrispondenza dei valori contenuti in *frequencies*, avviene la *decimazione* e poi la conversione in dB.

In mezzo a questi passaggi, inoltre, l'array viene prelevato (una volta interpolato alle frequenze contenute in *frequencies*) e mandato in uscita del blocco, per essere poi destinato al plot nell'interfaccia grafica.

La seguente sottrazione misura la differenza tra la *threshold*, relativa al segnale in *sidechain*, e la curva cosí ottenuta, relativa al segnale di *input*:

$$delta = input - sidechain \tag{4}$$

Definiamo questo risultato "*delta*", inteso come differenza (in dB) tra le due curve. Esso è un array di 32 valori che andrà a determinare l'elaborazione del segnale, che avverrà in 32 bande, posizionate in corrispondenza delle frequenze contenuto in *fCenters*.

I delta positivi indicheranno le frequenze in cui il segnale di *input* supera la curva ottenuta dal segnale *sidechain*, i delta negativi indicheranno le frequenze in cui il segnale è mascherato dal *sidechain*.

3.4.2 Modulazione del Delta

Successivamente, la curva del *delta*, utile dunque a determinare l'elaborazione del segnale, viene modulata in base ai parametri esposti all'utente nell'interfaccia. Il risultato sarà un array i cui valori piloteranno i guadagni assegnati alle 32 bande in cui verrà scomposto il segnale. Infine, il segnale verrà ricostruito andando a sommare le diverse bande, ognuna avente ora una nuova intensità.

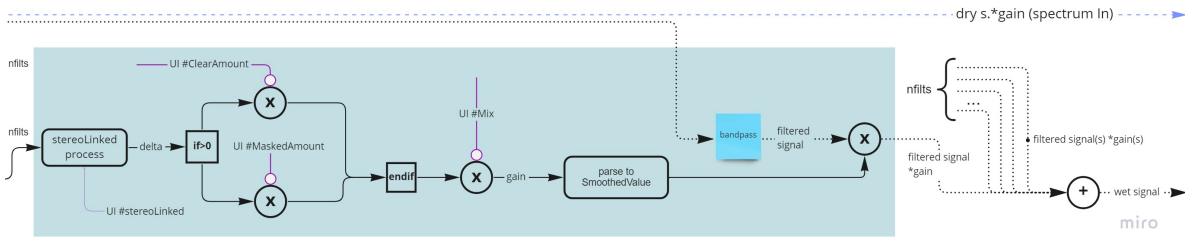


Figura 11: Seconda parte del blocco relativo all'elaborazione del segnale di *input*

Come si vede in figura, inizia ora un'elaborazione dell'array *delta* che vede ripetere le stesse operazioni su ogni elemento dell'array.

Definiamo *modulazione del delta* la moltiplicazione degli elementi dell'array per i parametri esposti all'utente (come sugerito dal colore viola), i quali sono valori float con un range compreso tra 0 e 1: rispettivamente, se l'elemento è positivo verrà moltiplicato per il parametro *clearAmount*, se negativo per *maskedAmount*. Questa scelta di design dell'esperienza utente è volta a differenziare il trattamento delle diverse frequenze. La discriminante, cioè il segno del *delta* nella specifica frequenza, non è altro che un indicatore del confronto tra il segnale *input* e il segnale *sidechain* in tale frequenza. Ricordiamo che il *delta* è il risultato della sottrazione (4).

- **Caso *input* $>$ *sidechain***: delta positivo, moltiplicazione per *clearAmount* (indicativo del fatto che in tale frequenza l'*input* è "clear", cioè non mascherato, dal *sidechain*)
- **Caso *input* $<$ *sidechain***: delta negativo, moltiplicazione per *maskedAmount* (indicativo del fatto che in tale frequenza l'*input* è "masked", cioè mascherato, dal *sidechain*)

Successivamente il risultato ulteriormente scalato in base al contenuto di *mixAmount*.

3.4.3 Filtraggio e ricostruzione del segnale

Il passaggio seguente è il vero artefice dell'elaborazione del segnale. Quest'ultimo, per ognuno degli 32 elementi dell'array *delta*, viene filtrato da un filtro bandpass (o meglio, come vedremo in seguito, un sistema in serie di low cut e high cut) in modo da isolare la rispettiva banda di segnale. Tale filtro ha frequenze di low cut e di high cut specifiche, calcolate in modo da essere equidistanti da elementi adiacenti di *fCenters*. Inoltre, una determinata banda ha la frequenza di low cut pari alla frequenza di high cut della banda immediatamente inferiore. In questo modo, si assicura il minimo intervento nella decostruzione del segnale.

Ognuna delle 32 bande, avente dunque centro nella rispettiva frequenza di *fCenters*, viene moltiplicata per il rispettivo coefficiente contenuto nella curva del *delta*, che ricordiamo essere calcolata in base al contenuto spettrale delle stesse 32 frequenze contenute in *fCenters*. Questo consiste quindi in una sorta di bilanciamento timbrico. Questa scelta è volta anche a ottimizzare il costo computazionale: se si fosse scelto un sistema di 32 peak filter, per ogni blocco di samples (o addirittura per ogni sample, nel caso in cui si fosse voluto interpolare

i coefficienti del *delta* in modo da avere un andamento nel tempo più smooth) si sarebbero dovuti ricalcolare tutti i coefficienti di tutte i 32 peak filter.

La ricostruzione del segnale consiste in una banale somma di tutti i 32 segnali, corrispondenti alle bande, ora aventi una nuova intensità, dovuta al rispettivo elemento dell'array *delta*.

CHAPTER

4 Prototipo MATLAB

La fase di prototipazione è stata sviluppata in ambiente MATLAB, su cui si è maturata una discreta esperienza in ambito accademico. Considerando la necessità di sviluppare ex novo un algoritmo fortemente basato sull'andamento nel tempo dello spettro del segnale, la comodità di questo linguaggio nell'effettuare i plot di vettori e matrici è stata una delle motivazioni principali della scelta. Inoltre il linguaggio MATLAB permette la facile manipolazione di vettori e matrici. Per un prototipo in grado di elaborare un segnale audio, sono state fondamentali le toolbox Audio, Communications, DSP System e Signal Processing.

Nella fase di prototipazione si è considerato di operare tramite la lettura di un dato file audio e non tramite un input real-time, in quanto sarebbe stato molto più complicato lo sviluppo senza una ragione valida. In JUCE, infatti, sarebbe poi stato banale lo sviluppo di un algoritmo real-time, data la natura e la main purpose del suddetto framework.

Inoltre in questo frangente ci si è limitati a lavorare sul back-end del software, trattando i parametri da esporre all'utente come semplici variabili, modificabili esclusivamente da codice.

Inizialmente c'è stata una fase di esplorazione, nella quale si sono reperiti online diversi esempi di programmi MATLAB che contenevano una parte dell'algoritmo simile a ciò che ci si proponeva di realizzare. Per la parte di analisi psicoacustica abbiamo combinato i seguenti progetti:

- *Prof. Dr. Ing. G. Schuller - Audio Coding 05: Psychoacoustics Models* [11]:

  un progetto Google Colab (in linguaggio phyton) in grado di costruire la soglia di

mascheramento di un segnale dato, dotato anche di spiegazione testuale e tramite videolezioni su Youtube. Attraverso la traduzione in codice MATLAB prima, e C++ poi, è stata la principale fonte di ispirazione per l'algoritmo operante l'analisi psicoacustica, nonchè per alcune funzioni di conversione di scala o dominio, usate in più occasioni.

- *F. Petitcolas - MPEG for Matlab* [12]: un progetto MATLAB emulante la compressione MPEG, la quale codifica il file tramite un'approssimazione dipendente da una soglia di mascheramento calcolata in ogni blocco di samples (frame). La costruzione di tale soglia, ad esclusione di alcuni accorgimenti superflui per il caso di TheMasker, era esattamente l'obiettivo del blocco operante l'analisi psicoacustica.

4.1 Struttura e impostazione del lavoro

Si è deciso di impostare il progetto restando il più possibile fedeli alla struttura dello schema a blocchi teorizzato. Per questo motivo il prototipo MATLAB è stato strutturato su diversi files: il principale (TheMasker_Main.m) racchiude lo scheletro generale del progetto, altri files (dependencies) svolgono singoli blocchi di operazioni e infine il file Shared.m che contiene diverse costanti e funzioni utili in diverse parti del codice, come ad esempio le conversioni da un dominio ad un altro.

Il file TheMasker_Main.m, dopo l'import dei files, l'inizializzazione delle variabili, la lettura dei file audio input e sidechain, si divide fondamentalmente nelle seguenti sezioni:

- una prima parte di setup, (che verrà chiamata "*Prepare to play*" per uniformità con il progetto JUCE) la quale viene eseguita solo una volta, all'inizio della compilazione del codice;

- una seconda parte (definita "*Process Block*", sempre per uniformità con il progetto JUCE), consistente in un loop che processa ogni sample del segnale audio, in base alla grandezza del buffer. Questa è la sezione fondamentale in cui avviene il vero e proprio processing dei segnali audio.

- una parte finale che si occupa di creare un plot relativo all'andamento nel tempo della soglia calcolata, di riprodurre l'audio del file una volta elaborato, e di eventualmente salvare quest'ultimo in formato wav e i grafici in immagini di formato jpg.

La divisione attualmente è solo concettuale: *Prepare to play* e *Process Block* non sono due funzioni, né file separati, questo perché a livello prototipale non c'è un concreto bisogno di questa divisione, ma è molto comoda come riferimento per il successivo sviluppo del progetto finale, anche perché le principali funzioni che caratterizzano un progetto in JUCE, sono, per l'appunto *Prepare to play* e *Process Block*. Approfondiamo le due sezioni nei seguenti paragrafi.

4.1.1 Prepare to Play

La sezione *Prepare to play* si occupa dell'inizializzazione di tutti i parametri che caratterizzano il prototipo e di svolgere tutte quelle funzioni che non hanno bisogno di essere ripetute nel ciclo di elaborazione, poiché non dipendenti dall'evoluzione dei segnali nel tempo né da altre variabili. Le funzioni più importanti riguardano:

- L'inizializzazione dei segnali di input e sidechain tramite la funzione offerta da MATLAB *audioread*. Inoltre vengono inizializzati la soglia di mascheramento e il segnale di output, al momento uguali a un vettore di elementi pari a 0.

- L'ottentimento della ATQ ovvero la soglia di mascheramento "in quiet": quando non c'è alcun segnale. La ATQ viene in seguito anche decimata per semplificare alcune difficoltà computazionali.

4.1.2 Process Block

La sezione *Process Block*, a differenza della precedente, è caratterizzata da un principale ciclo for in cui viene eseguito l'intero algoritmo per ogni campione (sample) del segnale audio, con dimensione del buffer fissa. In ordine le funzioni che lo compongono:

- **getDelta**: calcolo della soglia che verrà applicata al singolo campione audio, dipendente dal segnale input, dal segnale sidechain e dalla precedente menzionata ATQ.

  In primis al segnale sidechain viene applicata la *spreadingFunctionMatrix*: una matrice che si occupa di individuare le frequenze principali del campione nel determinato momento, e ”spalmarne” il valore sullo spettro delle frequenze, in modo da calcolarne la soglia di mascheramento dinamica. La funzione *spreadingFunctionMatrix* verrà approfondita in seguito nel prossimo paragrafo, per il momento basta sapere che serve ad aggiungere alla ATQ statica la componente variabile nel tempo dovuta alle frequenze che caratterizzano il campione audio in quel momento.

  In secundis viene calcolato il massimo tra ogni valore del campione sidechain e ogni valore della soglia ATQ:

  $$threshold = max(SC, ATQ)$$

  Infine viene calcolata la differenza tra il segnale di input e la threshold appena calcolata. È possibile effettuare il calcolo dal momento che entrambi i segnali sono in scala di bark e sono stati decimati considerando la stessa quantità di frequenze.

- **stereoLinkedProcess**: nel caso in cui il campione *delta* appena calcolato fosse stereo, viene calcolata la media tra i canali destro e sinistro, in modo da evitare grandi differenze tra questi ultimi e creare effetti spiacevoli all'orecchio.

- **modulateDelta**: diminuisce, aumenta o inverte il segno del *delta* in base a parametri che possa decidere l'utente.

- **scaleDelta**: il segnale *delta* viene moltiplicato per una curva sigmoide che si occupa di portare a 0 i valori del *delta* quando questi sono estremamente bassi (es: valori dai –40 dB in poi). In questo modo i valori paragonabili al silezio o al rumore di fondo non vanno a influenzare il risultato del *delta*, e non causano artifici indesiderati.

- **filterBlock**: si occupa di equalizzare il segnale di ingresso in base al *delta* precedentemente calcolato. Avviene separatamente per entrambi i canali.

- Ricostruzione dei segnali e della threshold tramite concatenazione del blocco corrente con i precendenti.

- Infine, essendo *Process Block* un ciclo di funzioni, abbiamo l'incremento del campione di uno step predefinito, per passare al sample successivo, o saltarne alcuni per superare iniziali momenti di silenzio o di azioni poco interessanti per l'anialisi.

$$blockNumber = blockNumber + step\_block;$$

4.2 Dependencies

Dopo aver descritto genericamente le funzioni principali del prototipo, si andrà ora ad analizzare nel dettaglio alcune delle funzioni utilizzate più interessanti, raccolte nella cartella "Dependencies". Le funzioni utilizzate in *Prepare to Play*:

- **getATQ**: funzione per ottenere la ATQ. Essa è praticamente la curva isofonica più bassa, sotto la quale l'orecchio umano medio non percepisce alcun suono. Si ricorda che essa deriva da misure empiriche, di conseguenza è stato necessario fare utilizzo di una funzione [11] che possa simulare questa curva con una discreta precisione; per cui, data una frequenza f:

$$ATQ = 3.64 * \left(\frac{f}{1000}\right)^{-0.8} - 6.5 * exp\left(-0.6 * \left(\frac{f}{1000} - 3.3\right)^2\right) + .00015 * \left(\frac{f}{1000}\right)^4$$

  La soglia calcolata è un vettore di 32 valori corrispondenti al livello in dB SPL per ogni centro di frequenza interessato.

  La ATQ è indipendente dal segnale di input, ma è legata esclusivamente al modello di mascheramento psicoacustico derivante dall'orecchio umano e può essere creata in questa sezione senza che debba modificarsi nel ciclo di processing dei blocchi di segnale. A variare, invece, sarà la soglia di mascheramento (chiamata semplicemente *threshold*),

che si compone della ATQ e dalle soglie calcolate ad ogni campione dipendenti dalle frequenze di quest'ultimo.

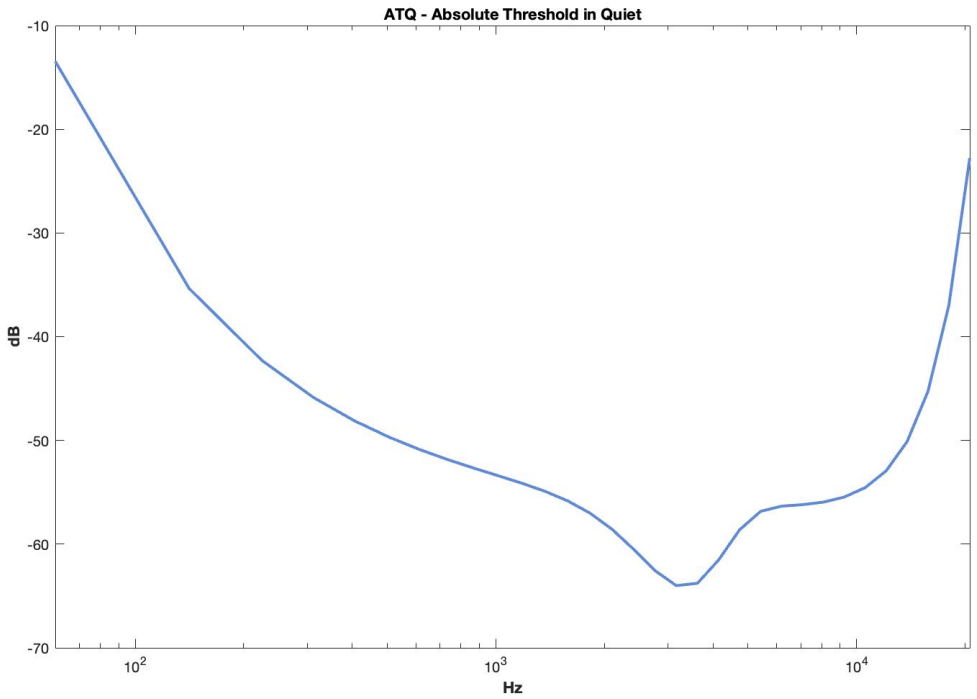


Figura 12: ATQ - Absolute Threshold in Quiet

Come si può notare, il grafico della ATQ è molto simile a quello delle curve isofoniche di cui sopra.

- **scaleATQ**: Permette di controllare l'effetto che ha la ATQ sul segnale, può essere anche diminuita tramite un parametro disponibile all'utente, chiamato *cleanUp* (valori da 0 a 1).

  Nota: A livello teorico, si può notare che, essendo quest'ultima una funzione dipendente da un parametro esterno, potenzialmente variabile nel tempo, essa dovrebbe essere implementata nella sezione *Process Block*. Tuttavia, dal momento che il prototipo non è pensato per essere eseguito in real time, i parametri esterni sono modificabili solo pre esecuzione. Inoltre, data la natura ciclica della sezione *Process Block*, si è deciso di anticipare la funzione in modo da evitare che questa venga ripetutamente

eseguita in ripetizione, senza che a tutti gli effetti ci possano essere variazioni nel corso dell'elaborazione. Nello sviluppo del progetto JUCE, invece, è stata correttamente spostata nel loop per poter essere variata in tempo reale.

Sono descritte ora le funzioni utilizzate in *Process Block*, in ordine di esecuzione:

- **getMagnitudeFD**: Funzione per ottenere i segnali nel dominio della frequenza, con valori di ampiezza in "Magnitude": valori tra 0 e 1. Inserendo come ultimo parametro *fbank*, la funzione applica la *decimazione* del segnale passato come parametro; *fbank* è una matrice NxM dove N è il numero di bande in cui viene decimato il campione e M il numero di campioni della FFT. Nel nostro caso si decima un segnale con 2048 campioni FFT in 32 campioni. La *decimazione* avviene prima del calcolo del vettore *delta*, proprio per poter effettuare il calcolo della soglia, uguale minimo tra sidechain e ATQ (già decimata a 32 valori) e, in seguito, la differenza tra input e soglia (uguale al vettore *delta*).

- **spreadingFunctionMatrix**: Gli effetti del mascheramento non si applicano mai a una singola banda critica di frequenza, ma influenzano anche le frequenze vicine, inferiori e superiori.

  Con un esempio matematico, dato un impulso di Dirac, la matrice spreading function restituisce una curva discreta con punto di massimo l'impulso stesso e due punti di minimo 0, uno precedente e e uno successivo all'impulso, creando una forma triangolare nel suo intorno, come mostrato in figura.

  Moltiplicata ad un vettore di bande di frequenze, permette di calcolare l'effetto del mascheramento di ogni banda nelle bande vicine, con una pendenza che definisce la diminuzione di intensità dell'effetto all'allontanarsi dalla frequenza considerata. È la matrice che, dato un segnale in dominio di frequenza, ne restituisce a tutti gli effetti la soglia di mascheramento dinamica, ovvero dovuta alle componenti armoniche del segnale: vedremo infatti, nella maggior parte dei casi, una soglia con punti di massimo relativi nelle frequenze che caratterizzano il segnale.

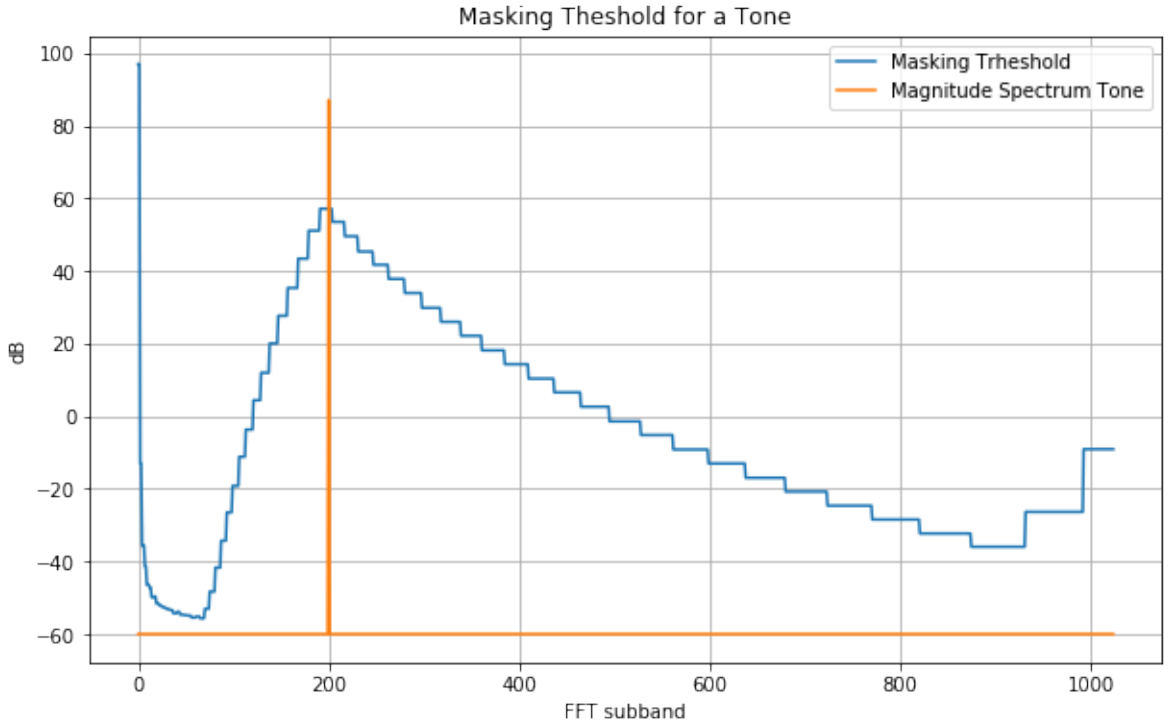


Figura 13: Spreading function relativa a un inpulso

La spreading function viene applicata inizialmente in scala di Bark . Nel passaggio alla scala in Hertz, le pendenze della soglia, prima lineari, formeranno delle cuspidi attorno ai punti di massimo: questo perché la scala di Bark non è lineare a differenza della rappresentazione in Hertz, ma segue l'andamento percettivo dell'orecchio umano medio, quindi con più divisioni sulle alte frequenze .

- **getDelta**: Comprende le due funzioni precedenti. Come già anticipato nel paragrafo precedente, al segnale sidechain viene applicata la matrice spreading function per calcolare la soglia di mascheramento dinamica dovuta alle componenti armoniche del segnale. Ne viene calcolato il massimo fra ATQ e soglia appena calcolata: dove la ATQ è maggiore è giusto che questa venga presa in considerazione, dal momento che il suo valore è sempre influente nel tempo allo stesso modo.

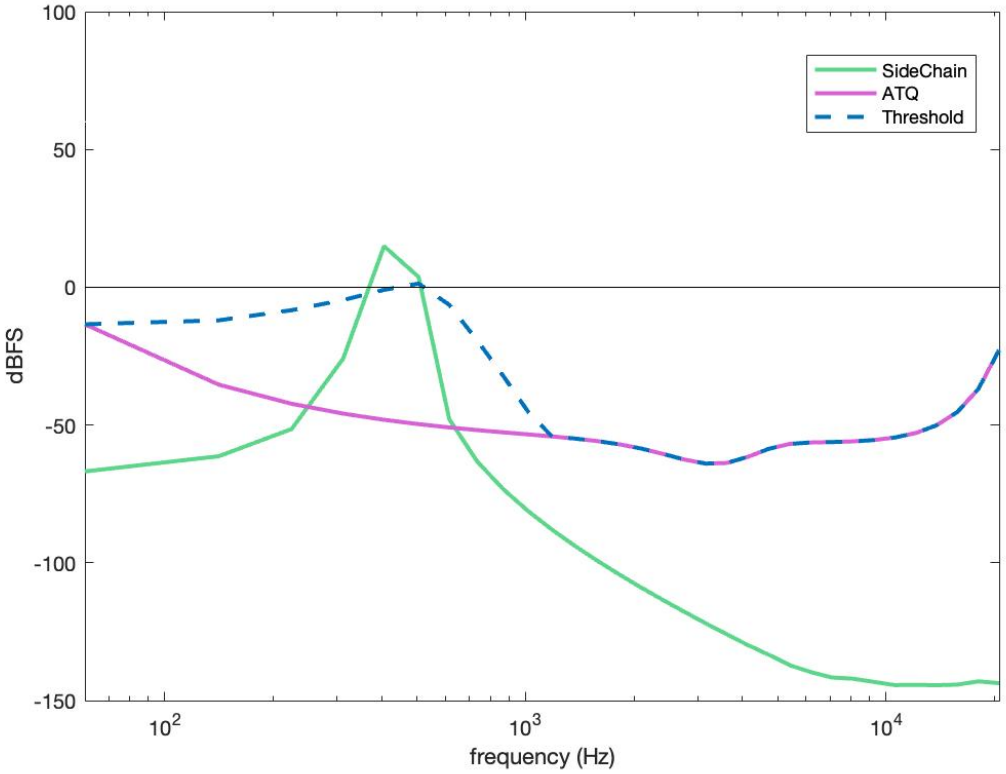


Figura 14: Esempio di Threshold ottenuta da una sinusoide come segbale sidechain

Infine, si ottiene il vettore *delta*, vettore delle differenze tra il segnale di input e la soglia attuale in ogni banda di frequenza.

$$delta = input - threshold$$

dalla precedente figura, si può notare come nel punto di massimo del vettore *threshold*, questo sia maggiore in ampiezza rispetto al segnale di input, generando, quindi, dei valori *delta* negativi.

- **stereoLinkedProcess**: semplice processo di modulazione che calcola la media in ampiezza tra i canali destro e sinistro. Il valore viene assegnato a entrambi canali, costruendo quello che è effettivamente un segnale stereo. Un compromesso scelto onde

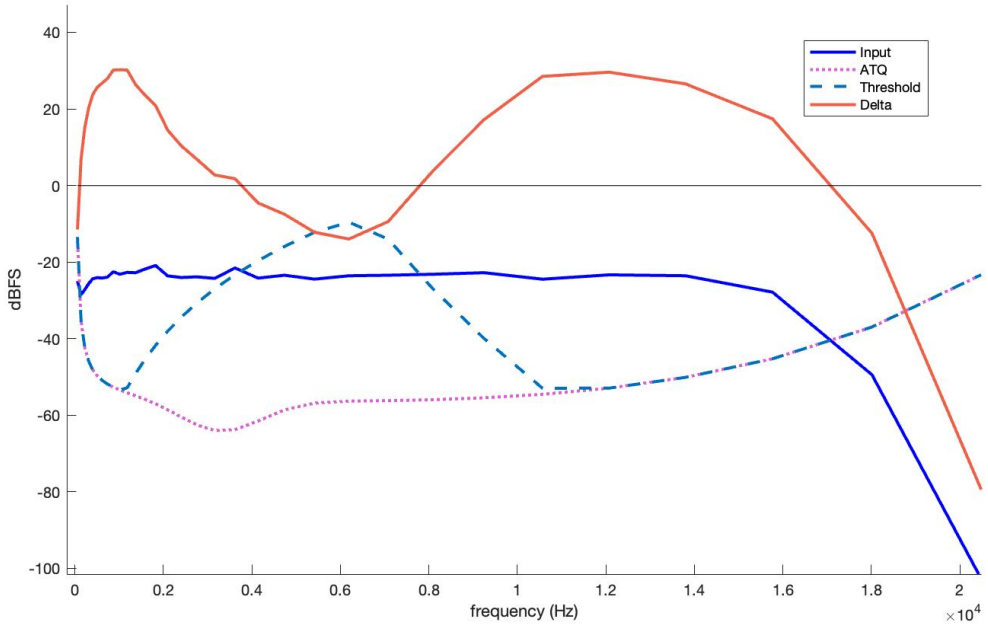


Figura 15: Esempio evidente del vettore *delta*

evitare artifici o effetti spiacevoli qualora si presentassero grandi differenze di ampiezza tra i canali.

- **modulateDelta**: funzione che permette di diminuire o addirittura invertire i valori del vettore *delta* in base ai parametri *comp* ed *exp* disponibili per l'utente

- **filterBlock**: qui avviene l'equalizzazione del segnale di ingresso non decimato, sulla base dei valori ottenuti dal *delta*, modulato e tagliato.

  Tramite la funzione MATLAB *butter()* [13], vengono generati dei filtri di tipo "Butterworth", detti anche "massivamente piatti", con lo scopo di mantenere il più possibile piatta la loro risposta in frequenza. Dalla funzione si ottengono gli zeri, i poli e i valori in ampiezza del filtro. In seguito la funzione MATLAB *zp2sos()* [14] riceve in ingresso gli zeri e i poli calcolati e restituisce un filtro di secondo ordine ("sos": "second order section"), che servirà per l'equalizzazione vera e propria del segnale, tramite la funzione MATLAB *filtfilt()* [15]. Quest'ultima funzione garantisce nessuna distorsione di fase.

- **Shared**: file che raccoglie generiche funzioni e definizioni di variabili utili una o più volte all'interno del prototipo, come, per esempio, la frequenza di campionamento, il numero di sottobande, il numero di campioni in frequenza e la dimensione del buffer.

4.3 Plot dei risultati

I due grafici stampati nel prototipo mettono a confronto i diversi elementi precedentemente descritti, per ogni campione audio analizzato nel ciclo della *Process Block*.

Il primo grafico rappresenta, in ordine, il segnale di input decimato, la soglia ATQ e la threshold derivata da quest'ultima, in seguito mostra l'evoluzione del segnale *delta*: originario (*delta*_raw), modulato e infine "clippato" (funzione del clipping approfondita nel capitolo successivo).

Il secondo, invece, confronta i segnali input e sidechain con il segnale output generato dal prototipo, senza decimazioni particolari in frequenza o ampiezza.

Nell'esempio qui riportato, il segnale input è un rumore bianco con volume basso (-64 dBFS, tipico "rumore di fondo"), mentre il segnale sidechain è una sinusoide di circa 500 Hz di volume alto (-3 dBFS). Il risultato finale del prototipo è quello desiderato: nell'intorno del picco del seidechain notiamo un abbassamento del segnale di output, nel resto dello spettro, l'output rimane intatto.

Analizzando i due grafici possiamo notare l'effetto della matrice *spreading function*: il segnale sidechain, con un picco in frequenza molto stretto, genera una soglia (threshold, nel primo plot) molto ampia, che indica il mascheramento causato da questo segnale. Infine, notiamo come a tutti gli effetti il *delta* finale non abbia una grande variazione in ampiezza, a differenza del vettore originario *delta*_raw: soprattutto i valori delle alte frequenze a dispetto di quanto possa sembrare, non hanno alcun effetto sull'output finale, che rimane praticamente identico all'input. Il motivo di questo effetto è dovuto ad una particolare funzione per la gestione del rumore di fondo, spiegata nel capitolo successivo.

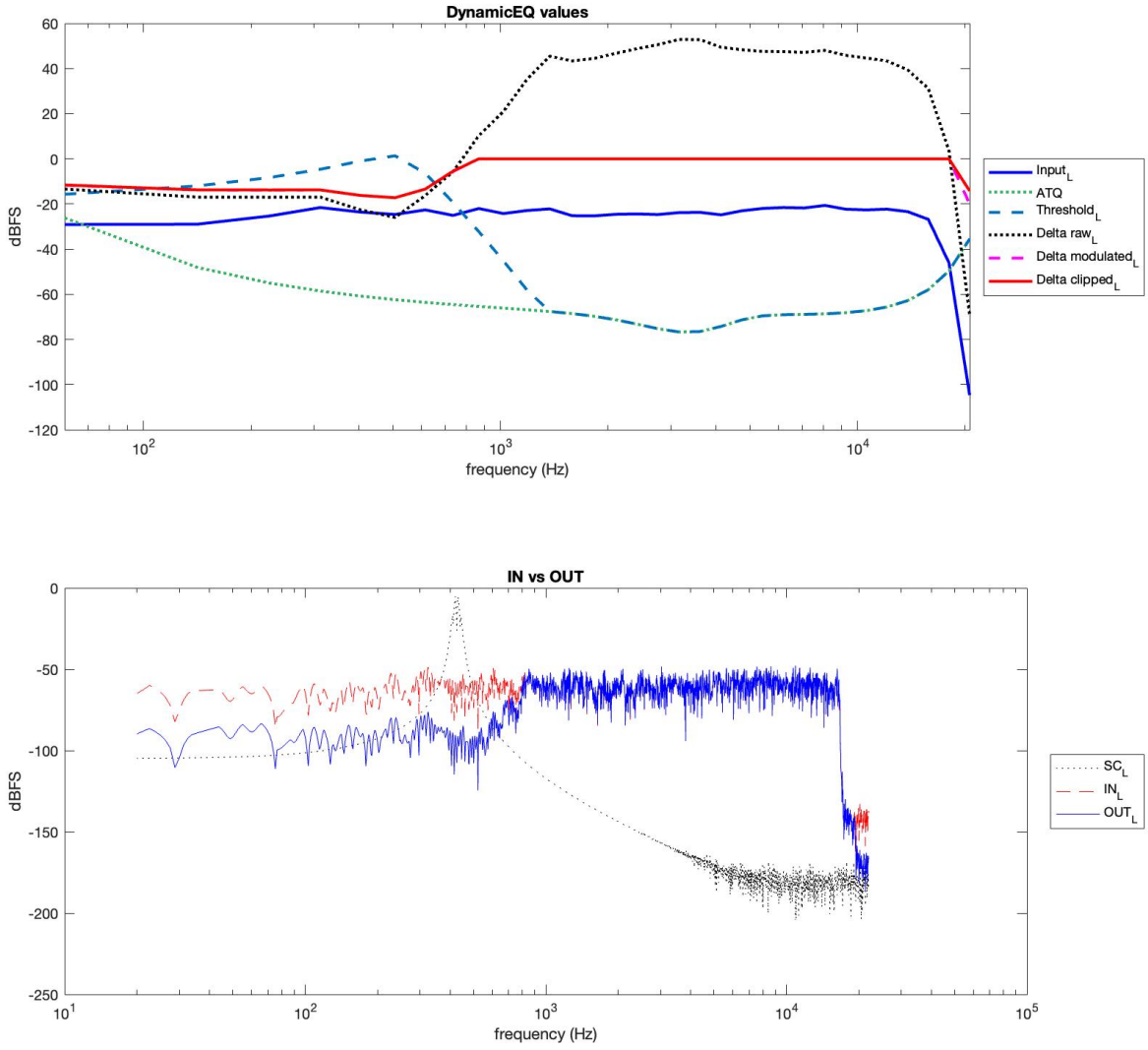


Figura 16: Confronto generale fra input, SC e output

4.4 Problemi e accorgimenti

4.4.1 Gestione del silenzio

Nella gestione di qualsiasi segnale di input è inevitabile imbattersi in situazioni con campioni audio con volumi molto bassi o nulli. In casi simili, per come è strutturato l'algoritmo, si otterrebbero dei valori di *delta* molto ampi (la differenza tra un segnale di input alto e un a threshold bassa). L'aspettativa, tuttavia, è quella di un comportamento minimo se non nullo del plugin: sarebbe inaspettata alle orecchie dell'utente una forte equalizzazione dell'input, in presenza di un segnale di silenzio in sidechain. In questa occasione viene in soccorso la funzione scaleDelta(), che si occupa di azzerare i valori del vettore *delta* ed il conseguente effetto del plugin in questi casi di silenzio.

Al vettore *delta* viene applicata una funzione di soft-clipping: un sigmoide che si occupa di ridurre lentamente a 0 i valori di ampiezza molto bassi (es: intorno ai -40dB) e a 1 quelli più alti. Dati il vettore della soglia *thresh* e due costanti: *dGating_thresh* e *dGating_knee* , la funzione è la seguente:

$$THclip = \left(1 + tanh\left(\frac{thresh - dGating\_thresh}{dGating\_knee}\right)\right) * \frac{1}{2}$$

- *dGating_thresh* è una costante di traslazione orizzontale: in questo caso il valore assegnato è 40, quindi i valori inferiori a -40 inizieranno a tendere a 0.

- *dGating_knee* indica la pendenza e la "morbidezza" della curva, ovvero la velocità con cui essa tende a 0 e 1. Il termine molto ricorrente in produzione musicale per questo tipo di parametro è proprio "knee".

Infine il sigmoide *THclip* viene moltiplicato al vettore *delta* per azzerarne i valori quando il segnale di *input* presenta valori molto bassi: *delta_clipped*

$$delta\_clipped = delta * THclip;$$

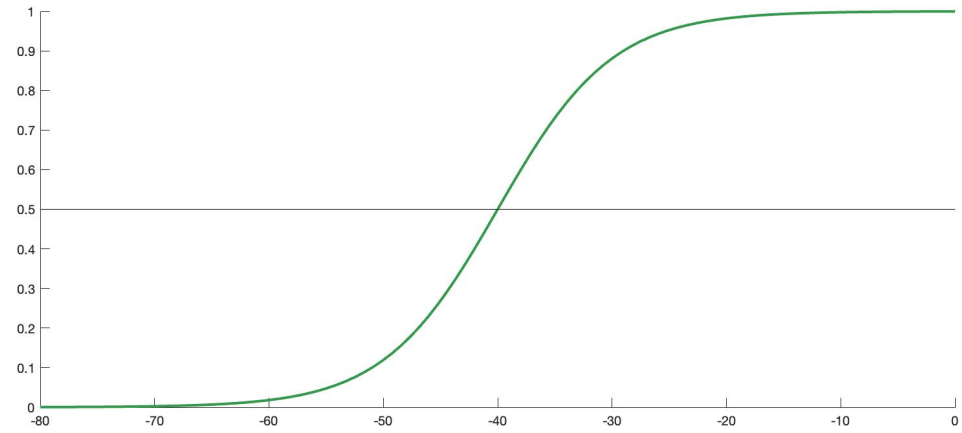


Figura 17: THclip, sigmoide applicato al vettore *delta*

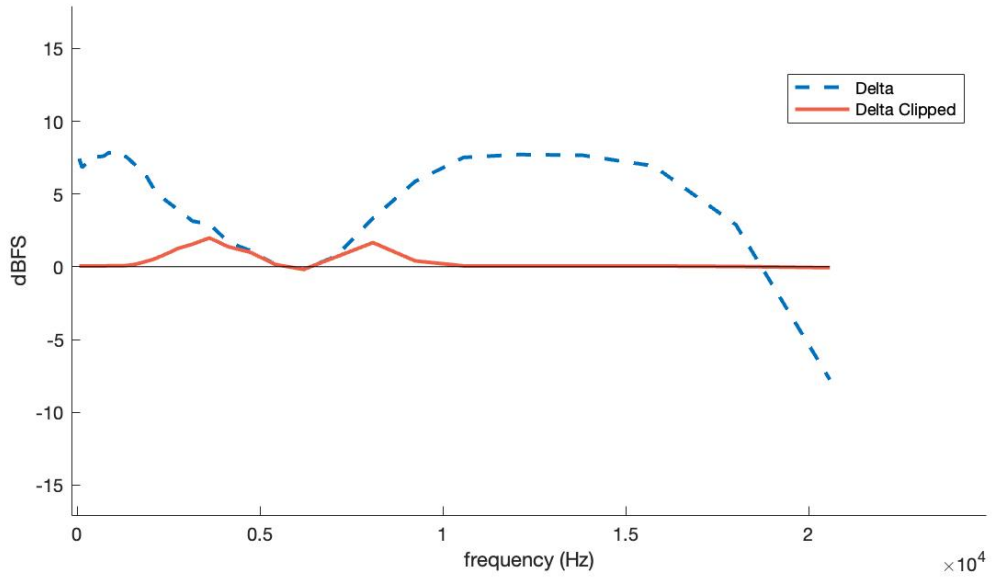


Figura 18: Vettori *delta* prima e dopo il clipping

In questa immagine possiamo notare come i valori tendenzialmente bassi restino invariati, mentre all'aumentare dell'ampiezza essi vengano in fretta portati allo 0, per evitare esagerate differenze di volume all'interno dello spettro. La qualità dell'effetto di clipping non è rilevante in questa fase della prototipazione: l'interesse è quello di gestire corner cases, ma la valutazione qualitativa sarà maggiormente considerata in fase di ultimazione del progetto finale (fase di *fine tuning*).

CHAPTER

5 Progetto JUCE

JUCE è un framework open-source multipiattaforma per applicazioni C++, utilizzato per lo sviluppo di applicazioni desktop e mobile. È utilizzato in particolare per le sue librerie di GUI e per plug-in. Ha una doppia licenza, la GPLv3 e una licenza commerciale.

Dopo la fase di prototipazione, si è costruito un Unified Modeling Language (UML) delle classi, in quanto in MATLAB se ne era fatto a meno, non dovendo far girare l'algoritmo in tempo reale. Per questa ragione, però, è stato fondamentale definire prima le classi utili all'algoritmo, senza escludere l'uso del concetto dell'ereditarietà. Successivamente si è proceduto con lo sviluppo tramite l'Integrated Development Environment (IDE) Visual Studio.

5.1 Funzionamento di JUCE

Il framework JUCE consente lo sviluppo di plugin audio offrendo librerie ottimizzate per l'elaborazione e la visualizzazione di interfacce real-time. Esso permette l'integrazione con gli IDE di Visual Studio e Xcode che consentono un comodo debug, lanciando il file .exe (versione standalone del plugin creato durante la build, assieme alle estensioni .vst3 ed eventuali altre designate).

Nell'utilizzo di un plugin audio, è importante considerare separatamente la User Interface (UI) e l'elaborazione del segnale. La prima viene consultata saltuariamente, eventualmente per ritoccare il settaggio dei parametri, e spesso chiusa subito dopo, mentre l'elaborazione dev'essere continua e non interrompersi mai. Per questa ragione il suddetto framework

separa l'esistenza di due oggetti fondamentali su cui si basa tutto l'algoritmo: essi sono il PluginProcessor e il PluginEditor. Il primo si occupa dell'elaborazione del segnale, il secondo dell'interfaccia grafica. È il PluginProcessor ad avere priorità nell'allocazione delle risorse del processore, che se avanzano vengono dedicate al PluginEditor, di cui è impostata una frequenza di aggiornamento massima.

Per quanto riguarda il PluginProcessor viene separata in due funzioni una fase di inizializzazione e istanziazione degli oggetti, in cui viene chiamata la *prepareToPlay()*, da quella che invece è la fase del loro aggiornamento nel tempo, chiamata la funzione *processBlock()*. La *prepareToPlay()* viene chiamata saltuariamente, cioè quando viene lanciato il plugin o quando ad esempio viene cambiato il setting della scheda audio. È infatti quest'ultima a determinare, ad esempio, la dimensione del buffer che l'host (e di conseguenza il plugin) riceverà continuamente, oltre che il sampleRate.

Nella prepareToPlay viene dunque rilevato il dato (passato dall'host, che lo riceve dalla scheda audio) della dimensione massima che il buffer avrà e la frequenza di campionamento. Definiamo questi due valori *maxNumSamples* e *sampleRate*.

La *processBlock()* riceve come parametro la referenza del buffer di samples su cui si può agire andando a elaborare il segnale di tale blocco di samples, che, appunto, avrà una dimensione massima di *maxNumSamples*. Tale buffer contiene uno o più bus, che rappresentano le diverse tracce in entrata al plugin, tipicamente la traccia di *input* (che verrà processata) ed eventualmente una traccia in entrata in *sidechain*. Ogni bus contiene un buffer di dati, ognuno con il rispettivo numero di canali da cui è composto.

5.2 UML e struttura codice

Prima di procedere con lo sviluppo, è stato strutturato un diagramma delle classi in forma di UML, tramite il tool online diagrams.net. Questo è stato fondamentale per strutturare il progetto secondo le classi pensate, utilizzando i concetti di ereditarietà e polimorfismo peculiari del linguaggio C++.

Seguirà un'analisi dello schema al fine di illustrare della struttura del codice implementato.

Figura 19: La struttura generale del diagramma delle classi

Per quanto riguarda gli oggetti ”PluginProcessor” e ”DynamicEQ” è stato specificato già nello schema il contenuto delle principali funzioni (*prepareToPlay()* e *processBlock()*) in quanto molto importanti per la progettazione delle classi.

Come vedremo in seguito, tante di queste classi possiedono le funzioni *prepareToPlay()* e *processBlock()*, che vengono chiamate a loro volta dentro le funzioni omonime dell'oggetto in cui sono istanziate.

5.2.1 Plugin Processor

Il plugin processor si occupa in primo luogo di creare il pluginEditor e di istanziare l'oggetto *dynEQ* della classe DynamicEQ, che rappresenta il nucleo del funzionamento del plugin.

Nella *prepareToPlay()*, inoltre, attraverso il metodo getFrequencies() crea l'array di 256 frequenze che verrà utilizzato in diverse fasi dell'algoritmo. Grazie a un sistema di listener, attraverso il metodo parameterChanged() richiama le funzioni setter dei parametri esposti all'utente: esse si trovano dentro *dynEQ*, e aggiornano il contenuto delle variabili su cui si basa l'elaborazione. Inoltre, fondamentale è la chiamata nella *prepareToPlay()* della funzione setLatencySamples(), con la quale il processor comunica all'host che ha bisogno di ricevere il buffer della traccia in anticipo del numero di samples indicato. In questo caso, le FFT presenti nell'algoritmo eseguono un rebuffering [1] a 1024 samples, su cui viene elaborata l'analisi FFT. Per questa ragione viene introdotto un ritardo di 1024 samples, che si è provveduto a compensare in questo modo.

Nella *processBlock()*, il plugin processor riceve il buffer passato dall'host. Suddivide il buffer per bus, copiandone il contenuto in mainBuffer e in auxBuffer. Nel caso in cui il bus del sidechain sia vuoto, ci copia lo stesso segnale di input. Inoltre esso si occupa di acquisire il numero di canali presenti nel buffer (che possono cambiare durante il runtime) e chiamare la funzione *numChannelsChanged()* di *dynEQ*, che andrà a ri-settare tutte le variabili associate al numero dei canali.

[1] Rebuffering: costruzione di un buffer di un numero fisso di punti

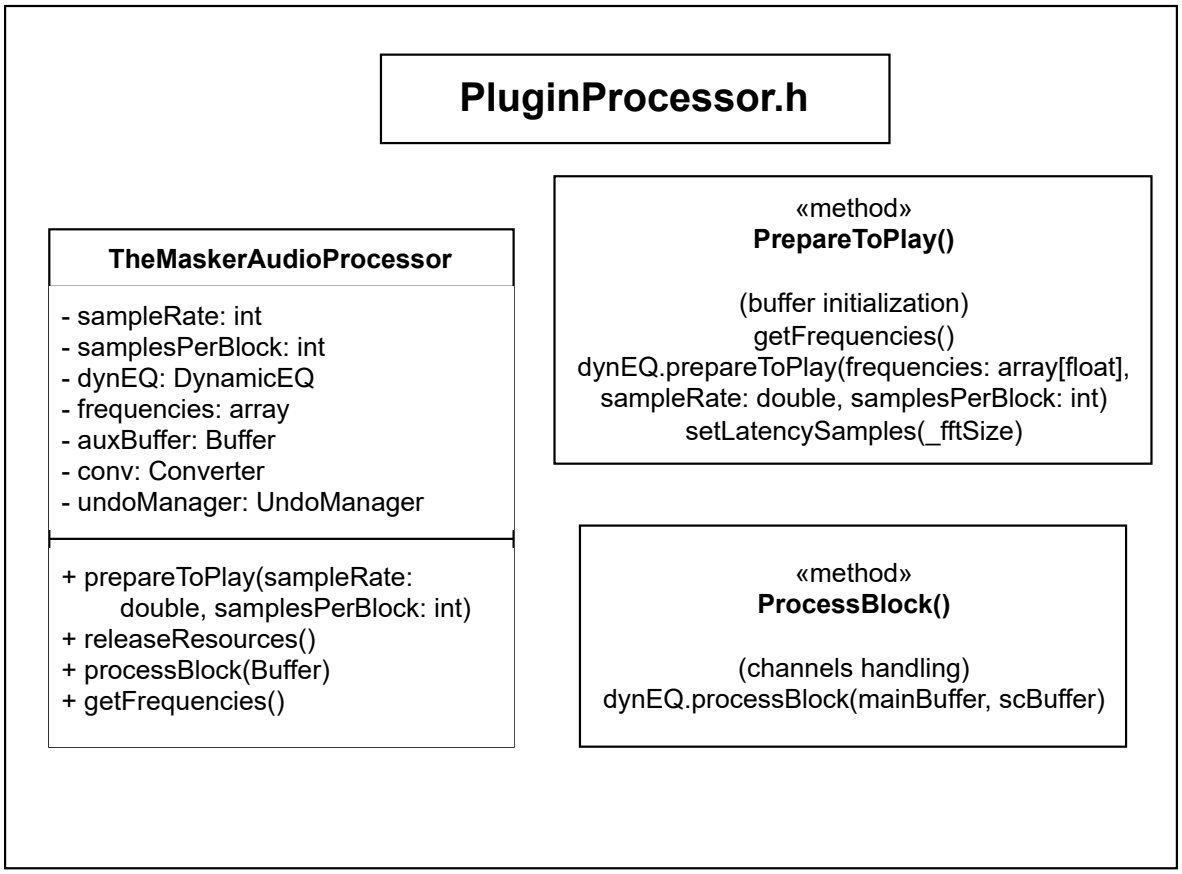


Figura 20: L'oggetto PluginProcessor e le principali funzioni

5.2.2 Constants e Converters

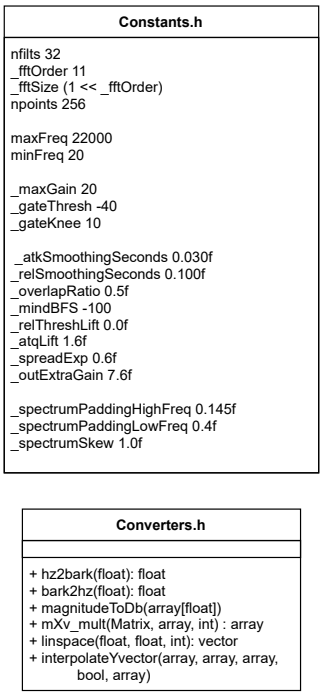


Figura 21: Il file Constants.h e la classe Converters

Prima di entrare nel vivo dell'algoritmo, occorre citare due file di uso frequente: il primo è il file Constants.h, che contiene tutte le macro utili in numerosi punti del codice (*nfilts* o *FFTsize*, per citarne alcune).

La classe Converters, invece, è istanziata in diversi punti del codice in quanto contiene funzioni fondamentali di conversione (*hz2bark()* ad esempio, che converte un valore in Hz nel corrispettivo nel dominio di Bark), o di scopo generale, come *linspace()* (esistente in MATLAB ma non in C++), o *mXv_mult()*, che si occupa di effettuare una moltiplicazione puntuale tra una matrice e un vettore.

5.2.3 DynamicEQ: il nucleo del plugin

La classe DynamicEQ, di cui l'unica istanza è creata nel pluginProcessor, è il motore di tutto l'algoritmo. Qui sono istanziati tutti gli oggetti che si occupano di analizzare ed elaborare il segnale, tutti i parametri su cui si basa l'elaborazione aventi le rispettive funzioni setter, e anche gli oggetti utili all'interfaccia grafica, che richiama la funzione *drawFrame()* 25 volte al secondo (se il plugin editor è aperto e se la cpu ha abbastanza risorse).

Nella *prepareToPlay()* viene popolata la matrice dell'oggetto FilterBank, che si basa su *nfilts* e *npoints*, tramite la funzione *getFBank()*. Essa inoltre crea dentro l'oggetto un'array di 32 frequenze, che viene copiato anche nell'array *fCenters* di dynamic EQ. Vengono inoltre chiamate le omonime funzioni di molteplici oggetti, ovvero quelli per i quali l'inizializzazione dipende da *nfilts*, *fCenters*, *frequencies* o *fBank*, e via dicendo.

Nella *processBlock()* i due buffer di *input* e *sidechain*, chiamati mainBuffer e scBuffer, vengono analizzati ed elaborati.

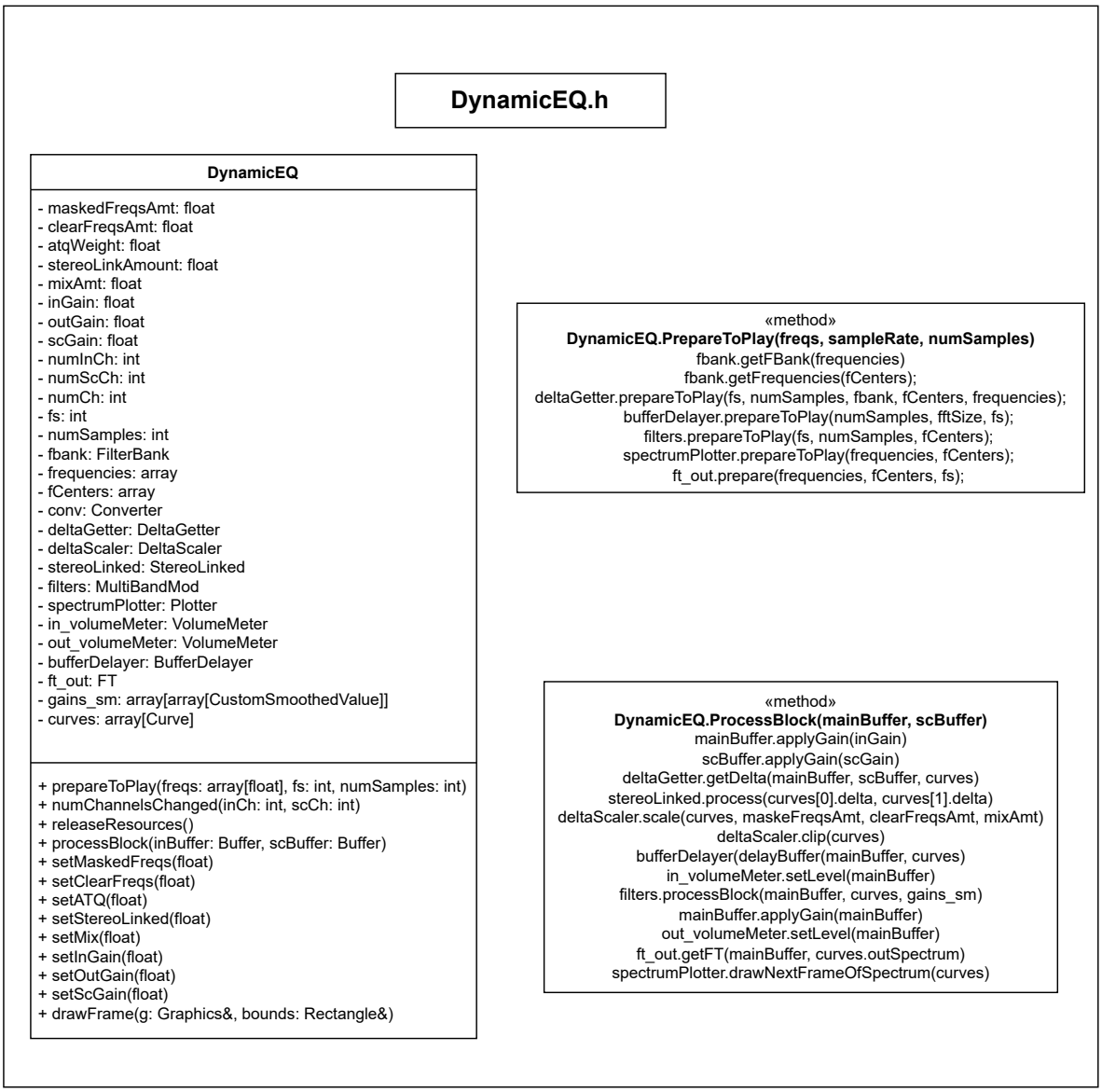


Figura 22: Il file DynamicEQ.h

Curve

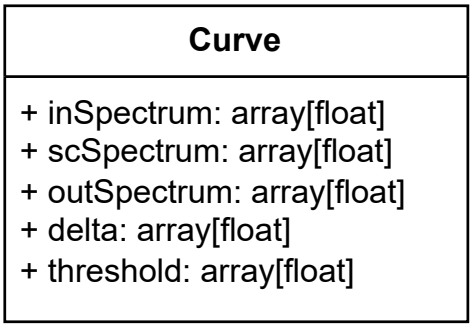


Figura 23: Lo struct Curve

Dopo un guadagno iniziale, l'oggetto di classe DeltaGetter si occupa di popolare con il contenuto in frequenza l'oggetto *curves*. Esso è un array di 2 (numero massimo di canali accettati dal plugin) istanze dello struct Curve, illustrato in figura. Esso contiene 5 array di float: i primi 3 conterranno ciò che verrà plottato nella UI - e sono di dimensione 256 - mentre i restanti due, di dimensione 32 , sono il frutto della *decimazione* operata tramite moltiplicazione puntuale per la matrice di *fBank*, la FilterBank sopra citata.

CustomSmoothedValue.h

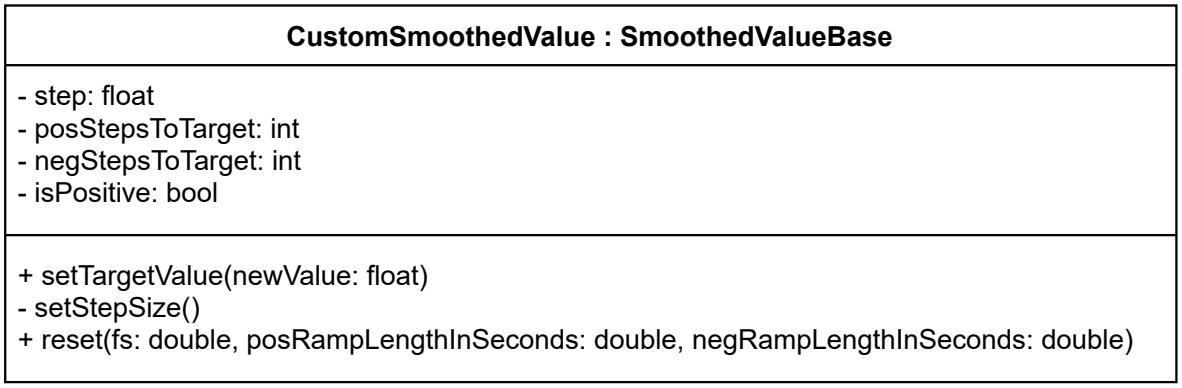


Figura 24: La classe CustomSmoothedValue

Un'altra importante classe è costituita da CustomSmoothedValue, che è stata creata a partire da una copia di SmoothedValue, classe di juce che approfondiremo in seguito. Tale classe viene istanziata in *dynEQ* dall'oggetto gains_sm, che consiste in un array composto da due (numero di canali) array di 32 (numero di punti del *delta*) CustomSmoothedValues. Esso contiene dunque i 32 valori (smoothed) del *delta*, ed è colui che verrà interpellato per effettuare poi l'effettivo filtraggio.

SmoothedValue permette il setting di una variabile non istantaneo ma graduale nel tempo tramite una rampa di lunghezza in secondi definita una tantum. La modifica di tale classe è consistita nel definire due diversi valori in secondi di lunghezza della rampa, uno preso in considerazione quando il valore target è maggiore del valore corrente (il valore è in aumento, la rampa è ascendente), l'altro quando minore (valore in diminuzione, rampa discendente). Il motivo di tale scelta è infatti la possibilità di definire due durate diverse della rampa di smoothing: una relativa all'"attacco" del *delta*, fissata a 80 ms, l'altra relativa al "rilascio" del *delta*, fissato a 250 ms, per le ragioni psicoacustiche già citate nel capitolo relativo all'analisi psicoacustica.

5.2.4 DeltaGetter

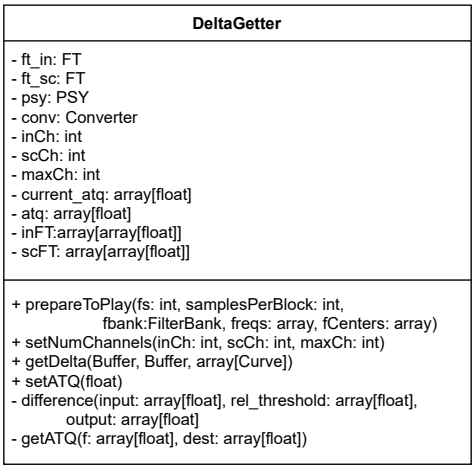


Figura 25: La classe DeltaGetter

I segnali contenuti rispettivamente in mainBuffer e in scBuffer vengono dunque analizzati dal DeltaGetter. Esso istanzia al proprio interno due oggetti della classe FT (che operano una FFT), che si occupano di analizzare il segnale *input* e quello *sidechain*, restituendone 256 valori in magnitudine, che vengonopoi convertiti in dB. Lo spettro del *sidechain*, in particolare, prima di tale conversione viene elaborato dalla *spreading matrix*, con la funzione *spread()* che vedremo in seguito. Esso viene poi confrontato attraverso la funzione *max()* con l'ATQ, array di float rappresentante la soglia di mascheramento "in quiet". Essa viene calcolata nella *prepareToPlay()* grazie alla funzione *getATQ()*, e viene poi pesata in funzione del parametro *cleanUp* esposto nell'interfaccia dalla funzione *setATQ()*. Quando tale parametro è a 0, essa si appiattirà ai -100 dBFS, e non avrà dunque influenza (considerando la funzione *max()*) rispetto alla soglia relativa al corrente blocco, quando a 1, andrà ad appiattire il *delta* nelle alte e nelle basse frequenze.

Dopo questa fase il deltaGetter si occupa inoltre di risolvere un importante problema relativo al numero di canali: il plugin consente di avere *input* e *sidechain* sia mono che stereo,

mentre l'uscita è mono solo nel caso in cui entrambi lo siano. Ciò che fa la funzione getDelta è dunque anche duplicare gli spettri di *input* e *sidechain* appena calcolati, nei casi in cui uno sia mono e l'altro stereo.

Il deltaGetter infine aggiorna l'oggetto curves, referenziato dal *dynEQ*, con gli spettri di *input* e *sidechain* (di 256 punti) e con gli spettri di 32 punti dell'*input* (ovvero lo spettro "decimato", chiamato threshold, utile poi nel deltaScaler) e del *delta*, che nasce dalla differenza tra lo spettro decimato dell'*input* (threshold, appunto) e del *sidechain*, come dalla formula (4).

FT e Analyser

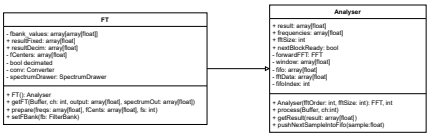


Figura 26: La classe Analyser e la sua sottoclasse FT

La classe FT, che estende la classe Analyser, svolge il suo principale ruolo nella funzione *getFT()*. Riceve il buffer da cui calcola la FFT, e si occupa di restituirne il contenuto spettrale all'interno dell'oggetto curves, referenziato da *dynEQ*. Il risultato della FFT viene però prima interpolato in modo da ottenere i contenuti spettrali nelle frequenze definite in *frequencies*: si ottiene così uno spettro da 256 punti a partire da uno di $\frac{1024}{2}$ punti. Inoltre, nel caso in cui nella *prepareToPlay()* sia stata chiamata la funzione *setFBank()*, allora è stata riempita la matrice *fbank_values* ed è stato anche posto a true il booleano *decimated*. Esso distingue i casi in cui c'è bisogno anche di una *decimazione* a 32 punti (nei casi di analisi dell'*input* e del *sidechain*, al fine di calcolare poi il *delta*) oppure basta ottenere lo spettro da 256 punti. Esso è passato in ogni caso all'oggetto curves, in quanto serve per il plotting dello spettro nella UI.

Il processing della FFT è delegato alle funzioni della classe madre (Analyser), che si occupa di riempire un buffer da 1024 samples, e solo quando è pieno assegnare a true il valore del booleano pubblico *nextFFTBlockReady*. Solo quando è a true esso permetterà a FT di leggere il valore di *result* e calcolarne le diverse interpolazioni.

PSY

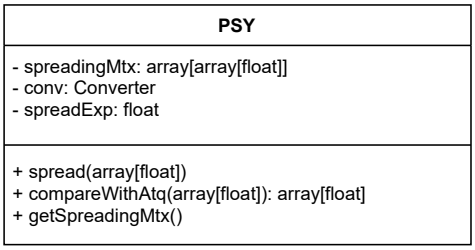


Figura 27: La classe PSY

La classe PSY viene istanziata dentro l'oggetto *deltaGetter* ed è l'artefice dello *spreading* effettuato tramite la moltiplicazione puntuale del *sidechain* per la *spreading matrix*, di dimensione 32 x 32, calcolata una tantum nella *prepareToPlay()*. Essa è fondamentale dal punto di vista psicoacustico in quanto attenua i picchi e le valli dello spettro del *sidechain* in modo da simulare il comportamento della soglia di mascheramento psicoacustico.

Tale classe si occupa altresì dell'operazione di *max()* che va a confrontare la soglia relativa e quella assoluta (ATQ), all'interno della funzione *compareWithAtq()*.

5.2.5 StereoLinked e DeltaScaler

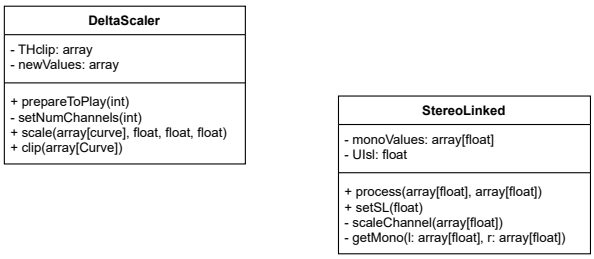


Figura 28: Le classi DeltaScaler e StereoLinked

Ottenuto il *delta* grazie all'oggetto deltaGetter, l'oggetto *curves* è riempito in entrambi i canali (salvo i casi in cui *input* e *sidechain* sono tracce mono) con gli array di *inSpectrum*, *scSpectrum*, *delta* e *threshold*. Questi ultimi due, in particolare, espressi in dB e relativi alle 32 frequenze definite in *fCenters*, andranno a definire il filtraggio effettuato successivamente nell'oggetto *MultiBandMod*.

Prima però, a seconda dei valori contenuti nei parametri esposti all'utente, l'oggetto *stereoLinked* prima, e *deltaScaler* poi, andranno a elaborare il contenuto del *delta*. Infatti, escluso il caso in cui *input* e *sidechain* siano mono, il parametro esposto nella UI chiamato *stereoLinkedAmount* andrà a pesare il grado di influenza reciproca dei punti del *delta* tra i due

diversi canali: se tale parametro è settato a 0, essi agiranno in maniera indipendente (saranno dunque ”unlinked); se invece esso è settato a 1, entrambi i canali conterrano una media tra i valori, in quel punto del *delta*, tra i due canali.

L'oggetto *deltaScaler*, invece, con la funzione *scale()* effettuerà un importante elaborazione del *delta*: eseguirà in prima battuta una traslazione del delta in modo che il valore medio tra i suoi 32 punti sia nullo (per compensare problematiche dovute ai diversi livello di *input* e *sidechain*); in secondo luogo distinguerà i punti del delta positivi (aventi quindi $input > sidechain$, detti ”clear frequencies”) e quelli negativi (aventi $input < sidechain$, detti ”masked frequencies”), e utilizzerà i parametri rispettivamente di *clearAmount* e *maskedAmount* (aventi range da -1 a 1) per scalarne il valore. Il parametro *mixAmount* (avente range 0-1), scalerà il contenuto di tutti i punti del delta. Infine, il delta sarà limitato a un valore di $\pm 12dB$:

$$y = \tanh(\frac{x}{12dB}) * 12dB;$$

Questa operazione effettuerà dunque un clipping graduale dei valori più estremi del delta, positivi e negativi, come illustrato in figura.

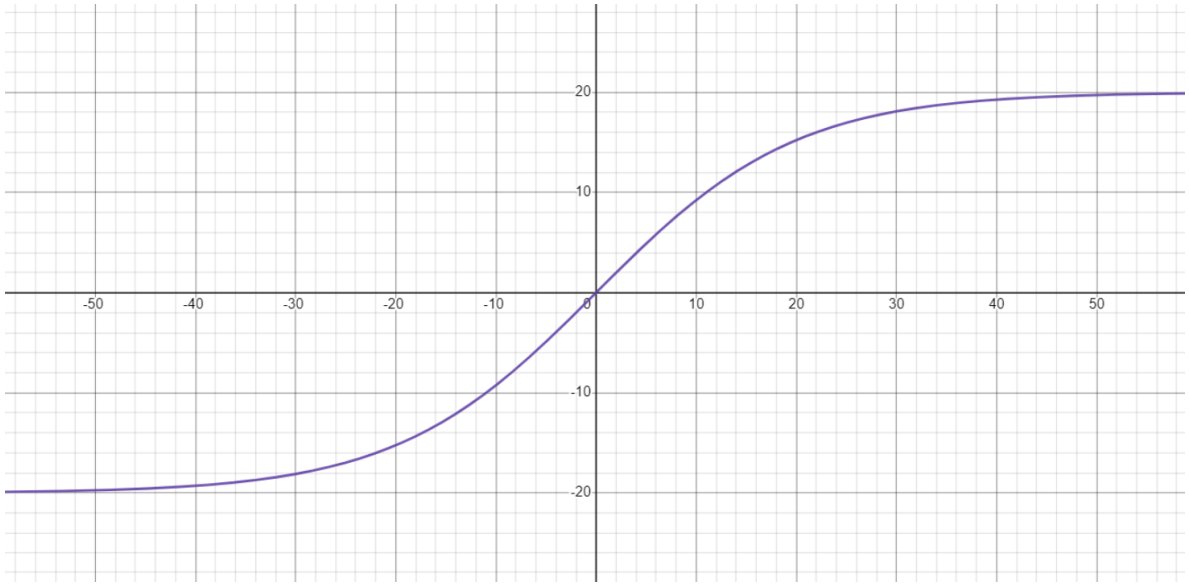


Figura 29: La funzione di clipping del Delta

La funzione *clip()* del medesimo oggetto invece si occupa di effettuare un gating del delta nei punti in cui il segnale di *input* sia classificato come silenzioso: vediamo ora nel dettaglio

in che modo avviene questa analisi.

All'interno della classe *DeltaScaler* è presente un array chiamato *THclip* che viene aggiornato per ogni aggiornamento della *threshold* (cioè il segnale di *input* "decimato"). Esso mappa i valori della *threshold* in una scala da 0 a 1: quando essa è inferiore al valore di -40 dB viene mappato a 0 in *THclip*, altrimenti viene mappato a 1. Ancora una volta questo passaggio è graduale: si hanno infatti 10 dB di "knee": come si evince dalla funzione, mostrata poi nella seguente figura, il passaggio da 1 a 0 non è immediato, ma comincia ai -20 dB e si esaurisce ai -60dB. Ecco come viene calcolato il contenuto di THclip per ogni *i* (rappresentante ognuno degli 32 punti della threshold):

$$THclip[i] = \{1.0 + \tanh[(threshold[i] - 40dB) * \frac{1}{10dB}]\} * 0.5;$$

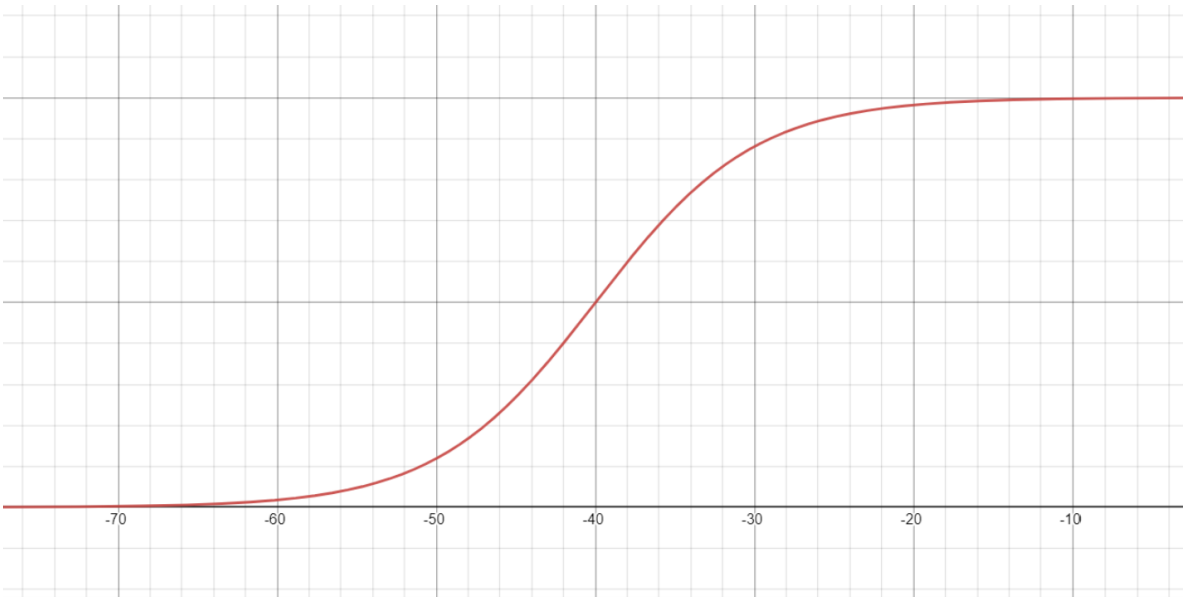


Figura 30: La funzione di gating della threshold

La moltiplicazione puntuale del *delta* per il THclip provoca un annullamento dell'effetto del *delta* nei punti di input "silenzioso" (o quasi).

5.2.6 BufferDelayer

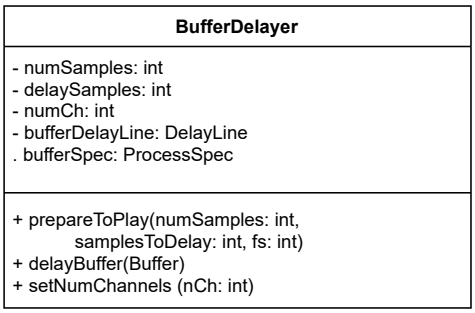


Figura 31: La classe BufferDelayer

A questo punto il *delta* è pronto per influenzare l'elaborazione del segnale. Va fatto però un accorgimento temporale. Per essere calcolato, il delta richiede 1024 samples. Essendo la dimensione del buffer passato dall'host variabile, e generalmente inferiore a tale valore, abbiamo visto come nella classe Analyser viene fatto un rebuffering. Solo quando tale buffer è pieno avremo un *delta* in uscita. Definendo come *numSamples* il numero di samples contenuti nel buffer, tale delta, di conseguenza, non è sempre relativo al buffer corrente, ma all'insieme di $\lceil \frac{1024}{numSamples} \rceil$ precedenti al buffer corrente. Per questa ragione, occorre ora ritardare il *mainBuffer* di 1024 samples, in modo da effettuare l'elaborazione in base al delta ottenuto dal buffer corrispondente e non a quello corrente, che rappresenta un istante successivo rispetto al delta contenuto in *curves*.

Questo è lo scopo dell'oggetto *BufferDelayer*, che attraverso la classe di juce *DelayLine* conserva i campioni correnti e restituisce in *mainBuffer* quelli precedenti di 1024 campioni.

5.2.7 MultiBandMod

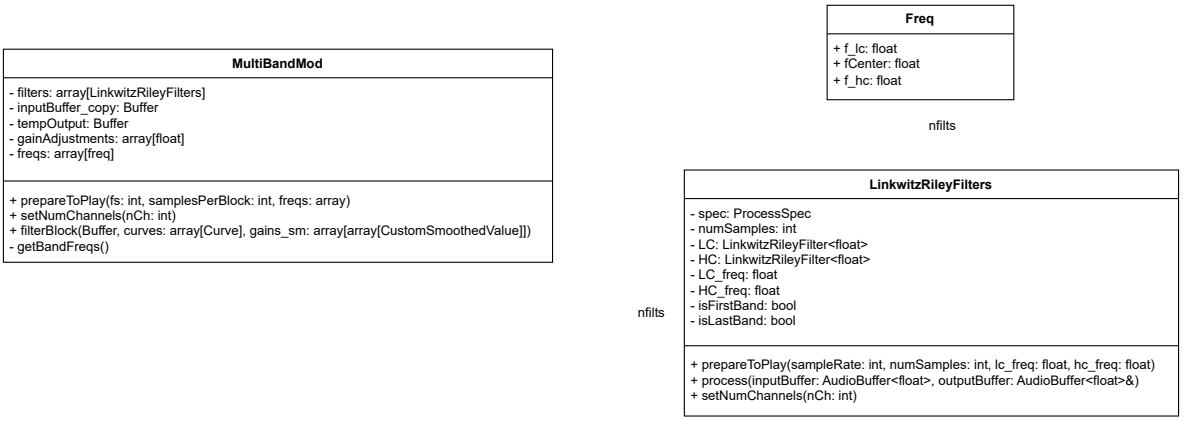


Figura 32: Le classi MultiBandMod, LinkwitzRileyFilters e lo struct freq

A questo punto avviene l'effettiva elaborazione tramite l'oggetto istanza della classe *MultiBandMod*. Come già menzionato, questo plugin è a tutti gli effetti un equalizzatore dinamico a 32 bande fisse. Viene effettuato un sezionamento dello spettro del segnale in 32 bande attraverso la classe *LinkwitzRileyFilters*, che al proprio interno contiene un filtro passa-basso e uno passa-alto. Possiamo dunque per ora considerarli come dei filtri passa-banda.

Freq

Tali bande hanno centro nelle frequenze contenute in *fCenters* e da esse otteniamo a questo punto le frequenze di crossover. Dentro *MultiBandMod* viene dichiarato uno struct chiamato *freq* utile a raggruppare tra loro la frequenza centrale e le due di crossover. Tramite la funzione *getBandFreqs()* chiamata nella *prepareToPlay()* l'array *freqs*, contenente 32 oggetti di tipo *freq*, viene riempito in questo modo: la frequenza centrale è già contenuta in *fCenters*, mentre per quelle di crossover viene fatta la media tra le frequenze centrali dell'elemento dell'array *freqs* precedente (nel caso della frequenze di crossover inferiore, che verrà poi assegnata al passa-alto relativo) e successivo (nel caso della frequenza superiore, assegnata poi al relativo passa-basso). Come frequenza inferiore del primo elemento di *freqs* viene assegnato il valore di 20 Hz e invece 20000 Hz è il valore assegnato alla frequenza superiore dell'ultimo elemento dell'array.

LinkwitzRileyFilters

La classe *LinkwitzRileyFilters* contiene al proprio interno due oggetti di tipo *LinkwitzRileyFilter* (classe nativa di juce), ottimizzata per la suddivisione in bande non distruttiva, di cui il primo (chiamato *LC*) è di tipo passa-alto (e gli verrà assegnata la frequenza di crossover inferiore) e il secondo (chiamato *HC*) è di tipo passa-basso (a cui è assegnata la frequenza superiore). Unica eccezione la hanno il primo e l'ultimo elemento dell'array di elementi di questa classe, per i quali viene assegnato un filtro di tipo all-pass, rispettivamente ad *LC* e a *HC*.

Questa classe ha poi una funzione *process()* che fa agire i due filtri sul buffer passatole, e ne copia il risultato su un array anch'esso referenziato come parametro della funzione.

Filtraggio e ricostruzione del segnale

Tornando all classe *MultiBandMod*, essa istanzia 32 oggetti di tipo *LinkwitzRileyFilters* in un array chiamato *filters*. Alla funzione *filterBlock()*, chiamata nella *processBlock()* di *dynEQ*, vengono passati il buffer da filtrare, l'oggetto *gains_sm* (che ricordiamo essere una matrice 2x32 di oggetti *CustomSmoothedValue*) e l'oggetto *curves*. Tale funzione, dopo aver copiato il buffer corrente dentro l'oggetto *inputBuffer_copy*, ne cancella il contenuto: il segnale verrà infatti poi ricostruito dopo il filtraggio. A questo punto attraverso un ciclo for su ognuno dei 32 filtri, avviene questa sequenza di chiamate:

- Il contenuto dell'oggetto *tempOutput*, di classe *AudioBuffer* viene cancellato
- Tale oggetto viene riempito con il risultato della *process()* del rispettivo oggetto di tipo *LinkwitzRileyFilters*, nell'array *filters*. Ciò che gli viene passato come buffer da filtrare è *inputBuffer_copy*
- Per ognuno dei canali (for annidato) avvengono queste operazioni:
  - Come target value del corrispondente elemento di *gains_sm* viene asssegnato il valore contenuto del *delta* contenuto in *curves*.
  - Per ognuno dei sample di *tempOutput* viene effettuata la moltiplicazione (regolazione di guadagno) per il valore corrente del corrispondente elemento di *gains_sm*, che viene fatto avanzare a ogni sample (operazione richiesta dalla classe *SmoothedValue* per progredire nella rampa associata, che tende al valore target.
  - il contenuto di *tempOutput* viene sommato dentro il buffer passato alla *filterBlock()*, che viene in questo modo ricostruito.

5.2.8 Le classi relative all'interfaccia grafica

Le classi mostrate in figura sono le principali utilizzate dall'interfaccia grafica. Esistono due istanze della classe VolumeMeter, una per il volume di ingresso e una per il volume di uscita: essa si occupa di inizializzare e aggiornare il componente contenente una barra verticale che

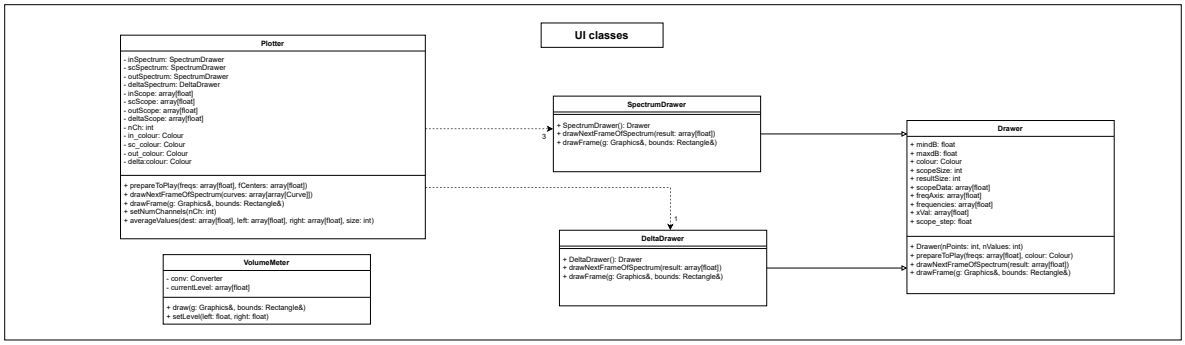


Figura 33: Le classi relative alla UI

mostra il volume Root Mean Square (RMS) del segnale, oltre che indicarne in basso il valore numerico corrente.

Le altre classi sono dedicate al plotting degli spettri dei segnali di *input* , *sidechain* e di *output*, e anche del *delta* corrente. È la classe *Plotter* a essere istanziata dentro l'oggetto *dynEQ* e ad aggiornarne i valori tramite la funzione *drawNextFrameOfSpectrum()*, chiamata durante la *processBlock()*. È invece il *PluginEditor.h* a ottenere una referenza di *dynEQ* e chiamare (con una frequenza variabile, con massimo di 25Hz) la funzione *drawFrame()* dell'istanza dell'oggetto di tipo *Plotter*. Questa funzione rasterizza le linee dei 4 oggetti in base al contenuto di un array che ha memorizzato l'ultimo aggiornamento.

Classi di juce::dsp utilizzate Alcune classi appartenenti al framework JUCE sono state particolarmente utili per lo sviluppo del plugin. In particolare, per quanto riguarda il back-end del software, è importante citare e spiegare brevemente il funzionamento delle seguenti classi.

AudioBuffer

La classe AudioBuffer, come è chiaro dallo stesso nome, è ottimizzata per memorizzare blocchi di sample audio, leggerne il contenuto, sostituirlo, o modificarlo. Già dal *PluginProcessor*, oggetto di partenza di qualsiasi progetto JUCE, vediamo una referenza a un oggetto di questa classe: si tratta del buffer audio passato dall'host come input del plugin.

Essa prevede molteplici canali e fornisce comode funzioni per modificarne il numero, così come la dimensione del buffer, ottenere puntatori di lettura o di scrittura, o applicare un guadagno (passando un valore float) ai sample del buffer.

SmoothedValue

SmoothedValue, classe già citata in precedenza, è una classe volta a contenere valori aggiornabili nel tempo in maniera graduale, ”smooth”, senza discontinuità. Ad esempio, se ci fossero discontinuità tra un sample e l'altro, si udirebbero glitch causati dalla altissima frequenza prodotta da questa discontinuità. Per questa ragione è spesso usata per aggiornare in maniera graduale i parametri esposti all'utente.

Il funzionamento è semplice: definito una tantum il periodo di tempo desiderato per far avvenire l'aggiornamento - e il sampleRate del segnale - il valore può essere aggiornato tramite un setter del cosiddetto *targetValue* dell'oggetto. L'effettivo valore contenuto nello *SmoothedValue*, che occorrerà aggiornare per ogni sample tramite la funzione *getNextValue()*, impiegherà il tempo definito per arrivare al valore target.

In questo caso, la classe è stata copiata e leggermente modificata per ottenere un funzionamento molto simile: l'unica differenza consiste nella definizione di due tempi diversi, uno per le fasi di incremento del valore e uno per la fasi di decremento.

FFT

Questa classe, come dice il nome stesso, è volta a operare una FFT del buffer di campioni che viene passato alla funzione *processforwardFFTtransform()*. Occorre però che tale buffer sia di dimensione fissa, e spesso si ricorre a un rebuffering in quanto i blocchi provenienti dall'host hanno dimensione variabile. La dimensione da passare alla funzione citata poc'anzi dev'essere doppia rispetto a all'array che si intende ricevere come risultato.

LinkwitzRileyFilter

La classe LinkwitzRileyFilter è stata di fondamentale importanza nel caso di TheMasker: essa è ottimizzata per la suddivisione in bande del segnale, considerandone una successiva ricotruzione, come è avvenuto in questa occasione. Ciò comporta una particolare attenzione al design dei filtri, che devono garantire un filtraggio non distruttivo della topologia del segnale.

Inventato da Siegfried Linkwitz e Russ Riley, il filtro di tipo Filtri Linkwitz-Riley (LR) consiste in un doppio filtraggio in serie con un Butterworth filter: per questo chiamato anche

Butterworth squared filter. Quest'ultimo è un filtro Infinite Impulse Response (IIR) avente risposta in frequenza pari a -3dB nella frequenza di cutoff. Utilizzando in parallelo due filtri LR, uno come high-pass e uno come low-pass, la risposta in frequenza nel punto di cutoff è di 0 dB, ottimale quindi per ottenere una risposta in frequenza piatta. La fase, invece, risente del filtraggio, anche se in minime quantità. Con filtri LR di secondo ordine la pendenza è di 12 dB per ottava, mentre con quelli di quarto ordine è di 24 dB per ottava.

La classe implementata da JUCE, in particolare, realizza un filtro LR di quarto ordine.

DelayLine

L'utilizzo della classe DelayLine è stato comodo per la classe BufferDelayer, volta a ritardare il segnale per compensare il ritardo dovuto all'elaborazione delle FFT. Questa, infatti, salva in un buffer provvisorio i sample passati con la funzione *pushSample()* e li restituisce tramite la funzione *popSample()*, ritardati di un numero di punti definito in fase di setting iniziale.

CHAPTER

6 UI/UX

L'interfaccia grafica è un elemento fondamentale del plugin. Per quanto possa esserne semplice lo sviluppo, l'esperienza utente non deve essere sottovalutata in quanto decisiva per un corretto nonché piacevole utilizzo. Oltre la scelta dei parametri disponibili all'utente, ci si è concentrati sulla grafica e sul "feel", ovvero la sensazione dell'effetto al tocco dei parametri.

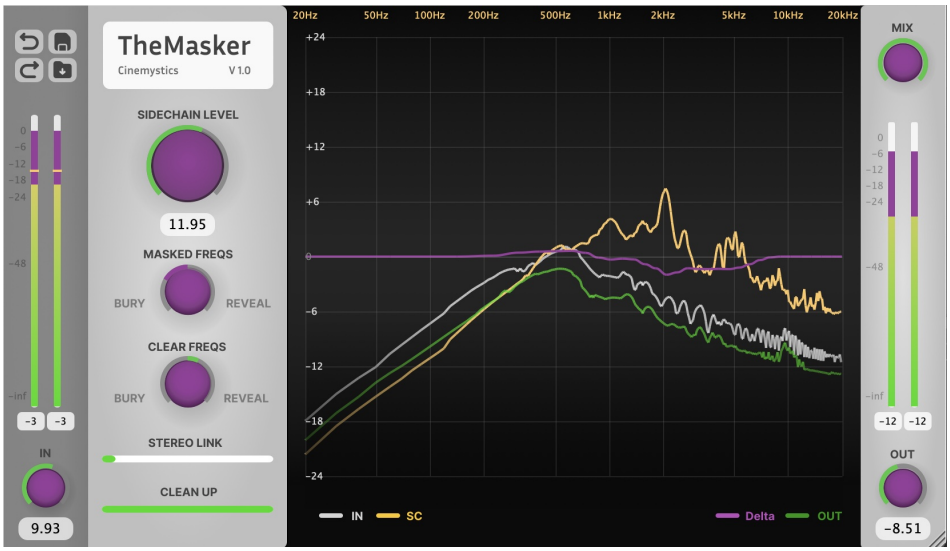


Figura 34: Interfaccia grafica di TheMasker

Per quanto JUCE permetta una buona separazione tra elaborazione del segnale e interfaccia grafica, si è preferito realizzare quest'utlima solo in seguito a una versione abbastanza solida e strutturata del codice. Questo perché alcuni parametri sono particolarmente dipendenti dalla loro implementazione e struttura, disposizione o semplicemente nomenclatura sarebbe potuta

cambiare nel corso dello sviluppo. In parallelo allo sviluppo del codice JUCE, comunque, sono stati realizzati i mockup grafici relativi, per avere un'idea generale dell'aspetto grafico desiderato.

6.1 Nome, palette e decisioni stilistiche

Il nome "TheMasker", come già detto, è un gioco di parole che accomuna il fenomeno del mascheramento, a cui è legato, e il personaggio dei fumetti *The Mask*. Grazie a questo gioco di parole il nome rimane efficace e facile da ricordare, tuttavia, per il resto non vi sono altri legami tra il personaggio e l'idea del plugin, anzi, si possono considerare molto discostanti: da un lato abbiamo una personalità che suggerisce creatività, sregolatezza ed esagerazione, dall'altro un plugin tecnico e preciso. Per questi motivi si è scelto uno stile comunque sobrio e geometrico, riprendendo soltanto il verde di The Mask come colore di accentuazione e il viola come colore secondario. Il giallo rimane utilizzato per alcuni dettagli che devono essere messi in risalto ma sicuramente non ha un ruolo centrale.

Figura 35: palette

Per quanto riguarda invece la struttura, le forme e le disposizioni, i riferimenti sono stati i plugins Soothe2 e Pro-Q3, gli stessi a cui ci siamo affidati per l'idea iniziale. In particolare Soothe2 presenta una interfaccia molto pulita e minimalista, ottima per contrastare la natura del plugin, complesso e ricco di parametri.

Le due figure mettono a paragone l'interfaccia di TheMasker con Soothe2:

si può notare immediatamente la somiglianza della disposizione delle aree degli slider circolari e dello spettro, oltre allo stile degli slider stessi. Per il resto le decisioni sono pratiche:

- lo slider relativo al guadagno del sidechain è più grande perché gli è stata attribuita un'importanza maggiore: senza di esso l'effetto sarebbe effettivamente nullo.

Figura 36: Soothe2

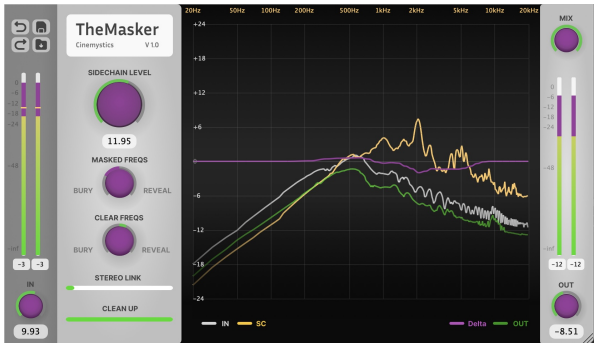


Figura 37: theMasker

- Gli sliders circolari successivi agiscono da moltiplicatori sulle frequenze mascherate (”masked freqs”) o non mascherate (”clear freqs”). In positivo avranno un’azione di ”rivelazione”, quindi di enfasi di queste frequenze, in negativo andranno a ”sotterrare” queste ultime. Abbiamo scelto, quindi, i termini ”reveal” e ”bury”, che possono rendere metaforicamente in modo efficace l’effetto che vanno a porre.

- I due slider lineari in basso, relativi allo stereolink e alla ATQ (definito clean up), sono sicuramente meno rilevanti degli parametri precedenti, ma comunque disponibili in primo piano e non nascosti.

- Gli spettri visibili sono quelli relativi a input, sidechain e output, inoltre si è ritenuto estremamente importante mostrare il vettore *delta*: è possibile mostrare o nascondere ognuno di essi, per poter concentrarsi esclusivamente sui segnali di interesse.

- Gli indicatori di volume a destra e sinistra sono relativi al segnale di input e di output: il segnale con gradiente dal verde al giallo indica il valore attuale in dBFS, mentre il viola indica l’RMS del segnale stesso. La linea orizzontale gialla,sovrapposta ai precedenti, indica il volume del sidechain, a cui non abbiamo dato uno spazio dedicato, ma che comunque può essere comodo da paragonare al segnale di input.

6.2 Mockup grafici

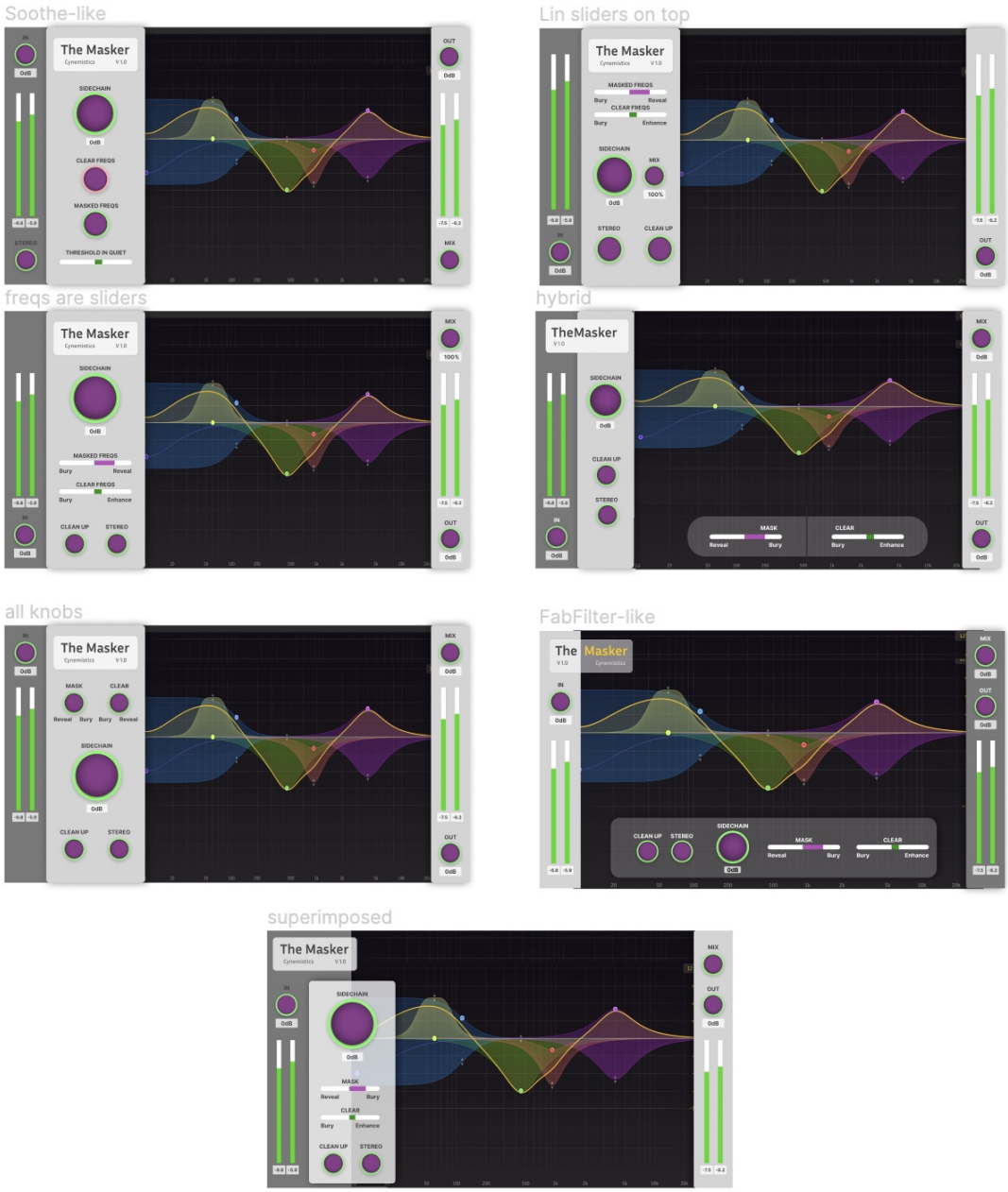


Figura 38: mockup grafici

I mockup grafici sono stati realizzati con l'editor online Figma [16], gratuito e ottimo per la creazione di immagini vettoriali. Si può notare come ognuno dei mockup grafici condivida gli stessi componenti e colori; l'obiettivo, d'altronde, non è un aspetto rivoluzionario e creativamente eccessivo, ma semplice, comprensibile e funzionale.

Confrontando con le immagini precedenti, il mockup più vicino alla soluzione finale è il primo in alto a sinistra. Quelle più differenti vedono lo spostamento dei parametri in orizzontale, al fondo dello spettro, in un'impostazione più vicina al Pro-Q3 di FabFilter. La scelta è stata scartata perché i parametri andavano a coprire parzialmente lo spettro nei valori più bassi, e allo stesso tempo non si è voluto ridurre l'area della risposta in frequenza perché si sarebbe persa una buona definizione della variazione del segnale in verticale, quindi in ampiezza. Si è deciso, inoltre, di tenere i nomi degli effetti quali "bury" e "reveal", ritenuti utili alla comprensione. Infine, si è mantenuta la simmetria tra gli indicatori del volume di input e output, in modo da non ingannare percettivamente l'utente: è probabile che un indicatore posizionato più in alto possa dare un'idea di un volume più alto e viceversa. Per i mockup lo spettro non è stato realizzato: si è deciso temporaneamente di inserire un'immagine indicativa presa dalla risposta in frequenza del Pro-Q3. Lo spettro è effettivamente una componente complessa a livello grafico, implementativo e computazionale.

6.3 Implementazione

L'interfaccia grafica finale differisce poco dal mockup. Il motivo è molto semplice: come detto in precedenza, Figma permette la creazione di immagini vettoriali; allo stesso tempo JUCE accetta sia immagini rasterizzate sia vettoriali. Sappiamo che le immagini in formato raster sono facilmente soggette ad aliasing nel momento in cui si vuole cambiare la dimensione dell'applicativo. Siccome si è voluta dare la possibilità di ridimensionare il plugin anche in corso di utilizzo, si è ricorsi all'utilizzo di immagini vettoriali, che invece vengono ricalcolate senza perdita di definizione. E' bastato, quindi, esportare il mockup in formato .svg e inserirlo in JUCE come sfondo per mantenere la massima fedeltà al mockup, di cui si era soddisfatti a livello visivo. Le uniche parti non esportate sono quelle dinamiche: indicatori di volume,

amount degli sliders e spettro (che comunque non è stato sviluppato nel prototipo grafico) vengono create direttamente con l'editor di JUCE.

6.3.1 La classe LookAndFeel

Per la realizzazione delle componenti dinamiche sopra citate ci si è serviti della classe Look And Feel (LnF), che mette a disposizione JUCE per il controllo grafico di sliders e volumi che altrimenti avrebbero uno stile di default non modificabile. La classe LookAndFeel presenta a sua volta diversi metodi, tra cui *drawLinearSlider()* e *drawRotarySlider()*, per disegnare rispettivamente uno slider lineare e uno circolare. Semplificano il codice permettendo di impostare in ingresso i limiti dello slider, intesi come variabili float per le posizioni massima e minima nello spazio o gli angoli massimi e minimi. Inoltre richiedono in ingresso il valore che ha al momento il parametro, mappato tra i limiti precedenti. Creando un'instanza LnF all'interno dell'oggetto dello Slider, è sufficiente chiamare i precedenti metodi e, all'interno di essi, disegnare path lineari o ad arco che partano dalla posizione minima al valore attuale del parametro e colorarli come si preferisce.

6.3.2 Il "feel" degli sliders

Oltre all'aspetto grafico, è stata creato un determinato mapping all'interno degli sliders che potesse seguire un'andamento particolare:

$$skew = \left(\frac{pos^3 + pos}{2} \right) \tag{5}$$

Dove *skew* è la nuova posizione mappata e *pos* è la posizione dello slider se fosse lineare, con intervallo [-1,1]. Otteniamo, in questo modo, maggiore definizione nei valori prossimi allo 0, e minore agli estremi, e conseguente maggiore precisione da parte dell'utente nei valori centrali dello slider. Questo perché comunque la scala su cui agiscono gli sliders non è mai lineare, e inoltre perché abbiamo considerato importante dare controllo sul "dosaggio" degli effetti, ma senza comunque vietare all'utente la possibilità di abbondare con valori alti del parametro.

6.3.3 Volume Meters

Gli indicatori di volume (Volume Meters) non hanno un comportamento molto diverso dagli sliders lineari, a parte, ovviamente, dover mostrare valori di volume e non dover assegnare valori al pluginProcessor. Il volume meter dell'input rappresenta contemporaneamente l'input, il sidechain e l'RMS. In particolare, il valore RMS non è una semplice variabile float, ma un oggetto di tipo SmoothedValue, già affrontato in precedenza, con un tempo di decadimento molto lento, in modo da rimanere visibile più a lungo e dare un feedback più utile all'utente che spesso vuole avere un'idea della potenza del segnale.

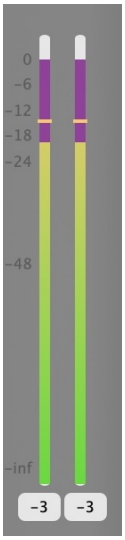


Figura 39: volume

6.3.4 Realizzazione degli spettri

Per la rappresentazione grafica degli spettri dei segnali input, output e sidechain, si è creata la classe virtuale Drawer da cui ereditano le classi SpectrumDrawer e DeltaDrawer. Le classi sono molo simili, per cui analizzeremo solo la prima.

Essa contiene i metodi:

- *DrawNextFrameOfSpectrum*, che riceve in ingresso il vettore campioni che compongono lo spettro, li mappa nello spazio disponibile secondo un andamento logaritmico che possa rappresentare meglio le frequenze e associa a ognuno di essi il valore in ampiezza corrispondente, pronto per essere disegnato.

- *DrawFrame*, che prende i valori precedentemente calcolati e crea una interpolazione tra i punti, per disegnare effettivamente lo spettro. Per DeltaDrawer la curva che interpola i punti è una di Bezier (funzione *cubicTo*, nativa di JUCE).

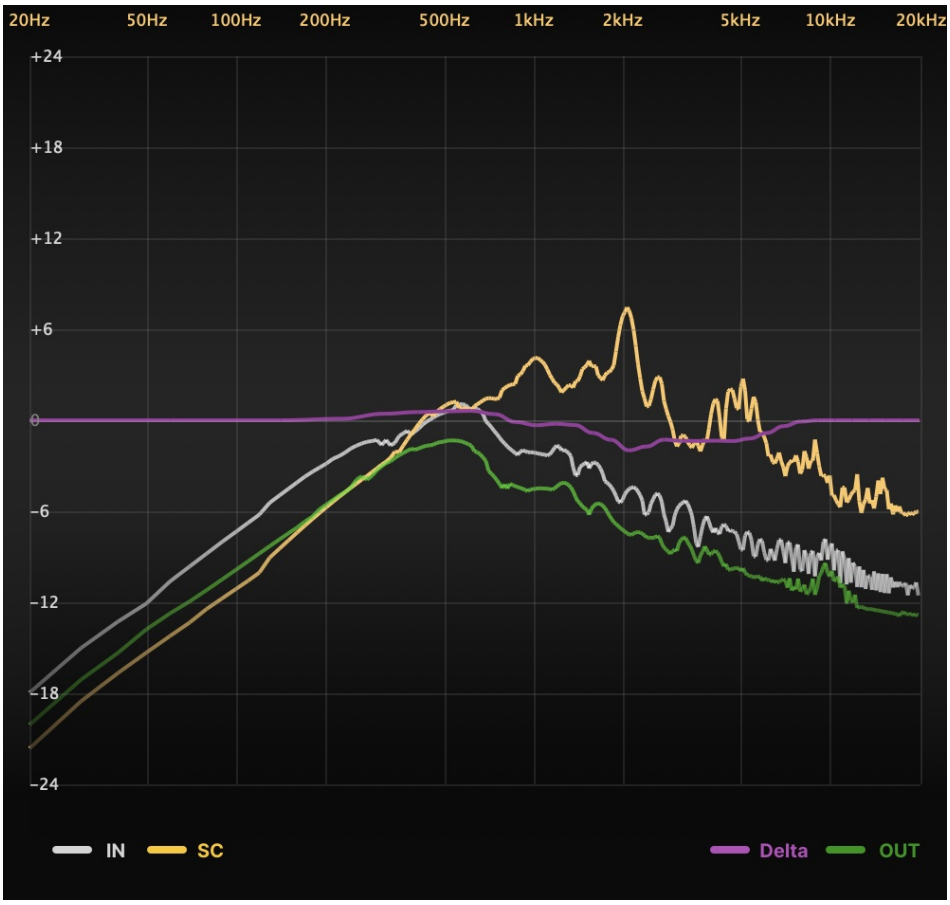


Figura 40: Spettri con relativa legenda

Esiste anche un'altra classe chiamata Plotter, che si occupa della creazione e gestione di tutti gli oggetti di classe Drawer, tra cui anche la possibilità di essere mostrati o nascosti tramite bottoni istanziati dentro PluginEditor.

6.3.5 Ridimensionamento

Come abbiamo anticipato, il plugin è ridimensionabile, quindi facilmente adattabile a diversi dimensioni e formati di schermi. I limiti di ridimensionamento sono: dalla metà dell'originale ai 3/2 dell'originale. Oltre questi limiti la visibilità del plugin risulta complessa o spiacevole. Per mantenere un coerente rapporto fra gli elementi grafici della UI, abbiamo inserito un oggetto, chiamato *Wrapper*, che si occupa di contenere e ridimensionare l'intero pluginEditor, senza dover aggiungere complessità nelle modifiche apportate e da apportare.

CHAPTER

7 Testing

7.1 Primi test e differenze rispetto al prototipo

Lo sviluppo in JUCE ha avuto una cura differente rispetto al prototipo in MATLAB. Se in fase di prototipazione l'obiettivo era avere un veloce riscontro dell'elaborazione pensata in fase di design e trovare un algoritmo le cui funzioni raggiungessero i risultati desiderati, in JUCE tali funzioni sono state integrate con un sistema di classi pensato per l'algoritmo sviluppato. Le principali differenze rispetto al prototipo sono state:

- La costruzione di un **UML** e una conseguente struttura del software basata su classi, ereditarietà, polimorfismo che in MATLAB si è ritenuto superfluo.
- L'**ottimizzazione** dell'algoritmo sul software JUCE, richiedente una bassa latenza (**real-time**) introdotta dall'elaborazione. Primo aspetto fra tutti, una migliore distinzione delle funzioni da chiamare nella *prepareToPlay()* e quelle da chiamare nella *processBlock()*.
- La gestione di numero di canali e buffersize variabili durante il runtime sono state dovute gestire tramite un controllo nella *processBlock()* del *PluginProcessor*.
- La presenza di un **interfaccia**, grafica e parametrica, assente in MATLAB, che lavorasse simultaneamente all'elaborazione, leggendo e sovrascrivendo il contenuto del back-end.

Di conseguenza la gestione di numero di canali e buffersize variabile, il sistema delle classi e la loro istanziazione, il sistema di visualizzazione e aggiornamento delle manopole e degli

slider così come le elaborazioni utili alla UI per il plotting dei segnali di spettro e per il delta sono state scritte ex novo in JUCE.

In aggiunta al diverso approccio utilizzato nello sviluppo dovuto a i punti citati sopra, dopo i primi test si sono subito notate alcuni difetti nell'inizializzazione e nell'aggiornamento degli oggetti e delle variabili contenute in essi.

7.1.1 Il nuovo ruolo del knob *cleanUp*

Inoltre, nel toccare con mano la parametrizzazione pensata, si è verificata l'inefficienza dell'effetto del parametro *cleanUp* riguardante l'ammontare di ATQ influente sul confronto con la soglia cosiddetta "relativa", ovvero quella dipendente dal segnale *sidechain* in entrata. Si ricorda che tale confronto consisteva in un'operazione di *max()*. L'effetto risultante era un aumento dell'effetto (la cui natura è definita dai knob di *maskedAmount* e *clearAmount*) negli estremi di banda, ovvero sotto i 150Hz e sopra i 7KHz.

Tale ammontare del delta (e dei guadagni sulle corrispondenti bande filtrate) produceva forti effetti di distorsione di fase dovuti ai diversi guadagni di bande (filtrate con filtri LR) adiacenti. Per questa ragione si è pensato a convertire tale knob, al contrario, in un attenuante dell'ammontare del delta in quelle bande. Mappando la curva dell'ATQ tra 1 e $(-1+cleanUp*1.5)$ e limitandone i valori al range $\{0\dots1\}$ e moltiplicando il risultato per il delta, si è costruito così un nuovo per tale knob. Esso diventa così un attenuante del delta delle bande in cui la curva dell'ATQ ha valori maggiori, quindi negli estremi di banda (alte e basse freq). Il range è stato scelto in modo che, grazie al clipping tra 0 e 1 del risultato, quando il knob è a 0 si ha la curva mappata tra tra 1 e 0.5, con una leggera attenuazione degli estremi di banda. Man mano che il parametro sale va ad azzerare il delta in un'area sempre più ampia delle basse e delle alte frequenze, attenuando gradualmente le bande centrali, secondo l'andamento graduale della curva.

In particolare tale curva è stata poi ritoccata per questo nuovo ruolo, assumendo i seguenti coefficienti e il conseguente andamento:

$$ATQ = 8.5 * \left(\frac{f}{1000}\right)^{-0.5} - 6.5 * \exp\left(-0.6 * \left(\frac{f}{1000} - 3.3\right)^2\right) + 0.57 * \left(\frac{f}{1000}\right)^{1.61} \quad (6)$$

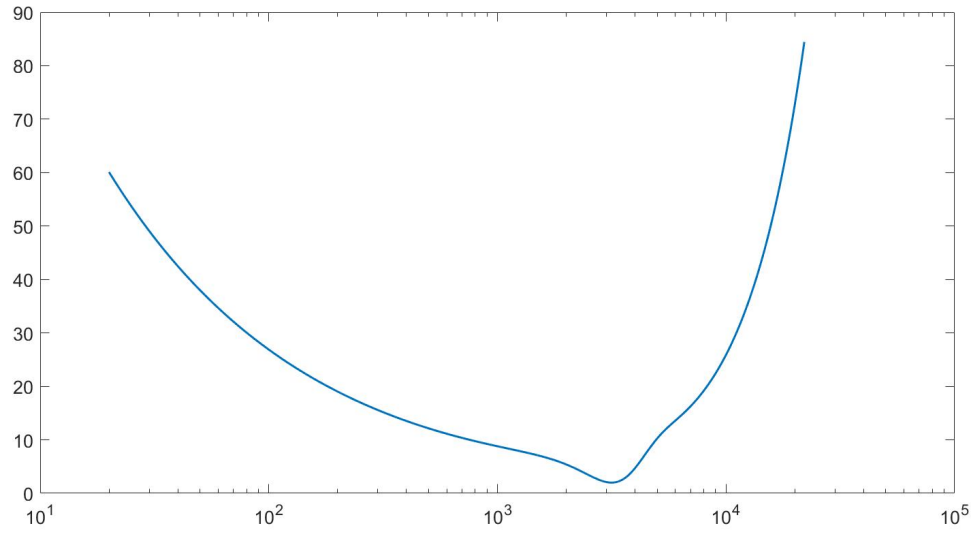


Figura 41: La ATQ con le modifiche designate per il nuovo utilizzo

7.1.2 Il gating del sidechain

Un ulteriore differenza rispetto al prototipo è la scelta effettuata dopo i primi test di effettuare un annullamento del delta anche nei casi in cui il segnale *sidechain* non superasse una certa soglia, come il segnale di *input* era già stato previsto e messo in pratica. Tramite la stessa funzione (e gli stessi parametri di *gateThreshold* - -40 dB - e *gateKnee* - 10 dB) si è andato quindi a effettuare un ”gating” del delta in caso di ”silenzio” nella traccia di *sidechain*. Si ricorda la funzione utilizzata per tale scopo:

$$SCclip[i] = \{1.0 + \tanh[(delta[i] - 40dB) * \frac{1}{10dB}]\} * 0.5;$$

7.1.3 L'array di aggiustamento del guadagno delle bande

Durante le prima fasi di testing si è notato come il plugin, anche con il knob *mixAmount* e i guadagni di *outputGain* e *inputGain* a 0, produceva una leggera equalizzazione, sopratutto alle altissime e alle bassissime frequenze. Questo è dovuto alla ricostruzione del segnale dopo il filtraggio in bande. Si è quindi cercato di porre rimedio con la costruzione di un array di float dal nome *gains_adjustments* dentro l'oggetto di classe *MultiBandMod*. I 32 coefficienti di tale array sono volti a essere poi moltiplicati per il rispettivo elemento del *delta* referenziato nell'oggetto. Dopo molteplici tentativi, si sono definiti i coefficienti delle prime e delle ultime bande in questo modo:

- gainAdjustments[0] = 0.1f;
- gainAdjustments[1] = 0.15f;
- gainAdjustments[2] = 0.5f;
- gainAdjustments[nfilts-4] = 0.85f; [1]
- gainAdjustments[nfilts-3] = 0.8f;
- gainAdjustments[nfilts-2] = 0.55f;
- gainAdjustments[nfilts-1] = 0.2f;

I restanti coefficienti sono unitari.

7.1.4 Lo smoothing del delta

Un'ulteriore differenza con il prototipo è data dallo smoothing del *delta* nel tempo operato dalla classe *CustomSmoothingValue*. In particolare, attraverso i primi test è stato notato come i tempi "teorici" di azione del mascheramento erano un vincolo troppo rigido per l'azione dell'elaborazione del plugin, che andava a seguire l'andamento del delta troppo velocemente, snaturando il timbro del segnale. L'effetto risultante era poco gradevole: è stato quindi

[1] nfilts = 32

aumentato il tempo di attacco a 80 ms e quello di rilascio a 250 ms, cosa che ha permesso un andamento del *delta* più smooth e un effetto più gradevole a livello acustico.

7.2 Testing

Lo sviluppo nell'IDE ha permesso il debugging, il controllo della coerenza dell'effetto dei parametri rispetto allo scopo proposto e l'affinamento delle variabili macro definite una tantum in previsione della fase finale di ritocco. Per la fase di testing si è effettuato un controllo qualità sia sul fronte pratico - andando a testare la UX - che su quello teorico, verificando la qualità della ricostruzione del segnale tramite il calcolo dell'Signal to Noise Ratio (SNR) della distorsione introdotta dal filtraggio.

7.2.1 Il controllo del corretto filtraggio

Durante tutta la fase di sviluppo e di debug, nonchè come primi test del plugin, è stata preso in esame il filtraggio di un rumore bianco, sulla base di una sinusoide mandata come *sidechain* in entrata. La scelta del rumore bianco come *input* è dovuta alla semplicità nel visualizzare le differenze nello spettro (inizialmente piatto) con il segnale di *output*. Producendo invece una sinusoide mandata in *sidechain* con un Virtual Studio Technology Instrument (VSTi) esterno si ha avuto la possibilità di utilizzare il la tastiera Musical Instrument Digital Interface (MIDI) per farne variare la nota, e di conseguenza muoverla nello spettro verificando la reazione TheMasker. Dopo opportuni accorgimenti e ritocchi, l'obiettivo è stato raggiunto con successo. Ecco alcune immagini della reazione di TheMasker a questa configurazione:

Tenendo il parametro *clearAmount* a 0 e portando *maskedAmount* a un valore negativo, le frequenze dell'*input* (colore bianco) superate dal *sidechain* (colore giallo) verranno attenuate. La curva dei guadagni (contenuto del *delta* - nell'oggetto *gains_sm*) è tracciata in viola, mentre l'*output* filtrato in verde.

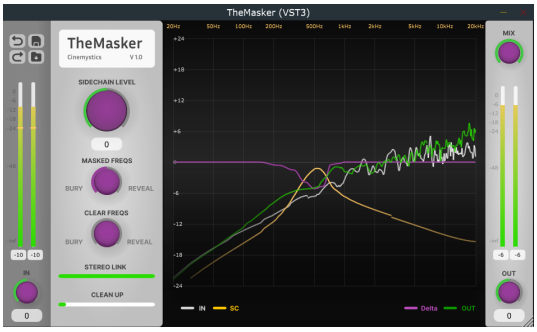


Figura 42: *maskedAmount* negativo

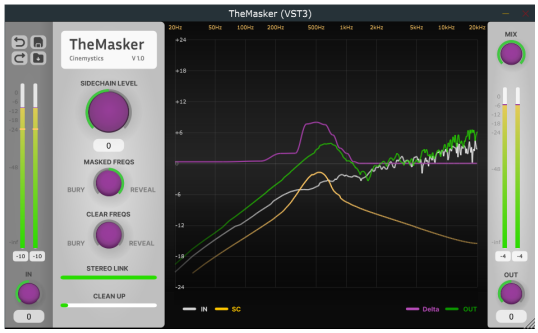


Figura 43: *maskedAmount* positivo

Viceversa, con il parametro *maskedAmount* a un valore positivo tali frequenze saranno enfatizzate.

Nelle immagini seguenti si nota invece come il parametro *clearAmount* gestisce le frequenze in cui è l'*input* a superare il *sidechain* - tenendo conto dell'offset di 10 dB che va ad abbassare ulteriormente il segnale di *sidechain* prima della sottrazione.

Figura 44: *clearAmount* negativo (sx) e positivo (dx)

Il knob del *sidechainGain* influisce abbondantemente sul confronto, come si apprezza dall'immagine a lato. In questo caso è stato alzato il segnale di *sidechain* di 9,2 dB. Il *delta* risultante è notevolmente più ampio, e va ad attenuare le frequenze mascherate (knob *maskedAmount* negativo).

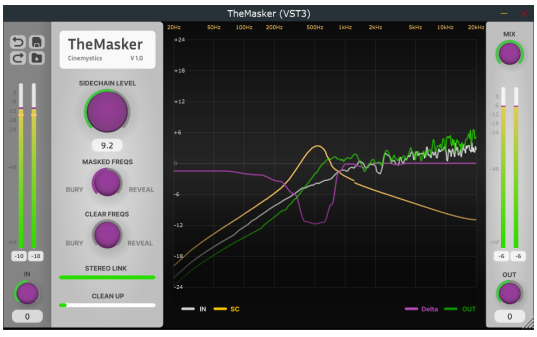


Figura 45: *maskedAmount* negativo

In ultimo, ma non per importanza, il knob del *cleanUp* agisce andando ad attenuare (e azzerare) il *delta* nelle alte e basse frequenze, seguendo l'andamento graduale tipico dell'ATQ.

Mostriamo a sx un caso di estremo boost sia di frequenze mascherate dal *sidechain* che di frequenze dell'*input* prevalenti sul segnale secondario. Nell'immagine di fianco andando ad alzare il *cleanUp* per evidenziare questo contenimento dell'effetto nelle suddette bande.

Figura 46: L'effetto del *cleanUp* a 0 (sx) e a 1 (dx)

7.2.2 Controllo qualità ricostruzione segnale (SNR)

Procedendo con un calcolo quantitativo, si è verificata la qualità della ricostruzione del segnale mediante il calcolo dell'SNR tra il segnale di *input* e quello di *output*. Questo è un classico controllo per verificare la qualità di codificatori audio e altri esempi di processing. Ricordando la formula dell'SNR tra un segnale di *input* e uno di *noise* (differenza tra *input* e *output*):

$$SNR = 10 * \log_{10}\left(\frac{inputPower}{noisePower}\right) \tag{7}$$

Dove la potenza del segnale e del rumore è data da:

$$signalPower = \sum_{i=0}^{dimension} |\ input\ |^2 \tag{8}$$

E dove *dimension* rappresenta una finestra in cui effettuare il calcolo, nel nostro caso fissata a 20000 samples.

Il risultato degli SNR, calcolati in real-time e dunque dal valore fluttuante repentinamente, è stato poi mediato su un'ulteriore dimensione di 400 elementi, andando a oscillare intorno al valore di **8,7154 dB**. A giudicare dall'ordine di grandezza il risultato si ritiene moderatamente

accettabile, dal momento in cui il valore dell'SNR riserva un'importanza minore in casi come questo, in cui è più la qualità percettiva ad avere priorità come discriminante della qualità del segnale a livello matematico.

7.2.3 User experience test

Per quanto riguarda il lato pratico si è proceduto a testare il software su una DAW, ovvero dove ne è previsto l'utilizzo. Alla comodità di gestione delle tracce (e delle mandate, coi rispettivi volumi) si aggiunge il vantaggio di poter monitorare la CPU con facilità (tantissime DAW hanno un visualizzatore delle risorse occupate incorporato nell'interfaccia). Inoltre, quello di verificare il corretto funzionamento della funzione *setLatencySamples()*, che comunica all'host (la DAW, in questo caso) con quanti samples di anticipo passare le tracce - in questo caso samples.

Si è individuato il seguente possibile use case di TheMasker: si hanno due tracce bus, una di batteria (nelle immagini "DRUM BUS") e una di tutti i restanti strumenti di un brano strumentale (nelle immagini "INSTRUMENTS BUS"). L'obiettivo è quello di far emergere maggiormente la batteria, rendendola più intellegibile.

La struttura dei parametri designati consente l'utilizzo di TheMasker in due modi: sulla traccia di batteria, andando a enfatizzare le frequenze che confliggono (knob *maskedAmount* con valori positivi) o sulla traccia di strumenti, attenuandole. In entrambi i casi è richiesto chiaramente l'altro segnale come *sidechain* in entrata. Scegliendo quest'ultimo caso si è proceduto con il testing qualitativo del plugin.

Dai test si è notato come il knob *clearAmount* si può invece utilizzare a piacimento: tendenzialmente scegliendo una direzione è concorde al *maskedAmount* si enfatizza l'effetto, estendendolo anche alle frequenze non in conflitto. Occorre però compensare con il knob *outputGain* di uscita. Utilizzandolo con direzione discorde si tende a una compensazione dell'effetto (in questo caso, un ri-bilanciamento delle frequenze), rendendo però molto evidente e a volte distruttivo l'intervento del plugin, che va a snaturare il timbro del segnale in entrata.

Il nuovo uso di *cleanUp* risulta estremamente funzionale per contenere l'effetto del *delta* sulle alte e basse frequenze, così come il suo mapping (dovuto anche alla modifica della

funzione che ne definisce la curva (6).

Dopo opportuni ritocchi, anche il parametro *mixAmount* risulta adeguato nel range e nella sensibilità al tocco, consentendo di attenuare l'effetto fino a renderlo molto morbido e "trasparente", termine che nel gergo del sound design indica un intervento quasi non percettibile.

Il knob di *stereoLinkedAmount* ha una utilità relativamente ridotta, in quanto ha un intervento meno evidente, anche se molto importante sopratutto per le fasi di mastering, e per ritoccare la naturalezza dell'intervento del filtraggio.

Una nota negativa è sicuramente la colorazione[2] data dal plugin anche con il *mixAmount* e i guadagni di *outputGain* e *inputGain* a 0. Nonostante le bande non vengano alterate nel guadagno, la loro ricostruzione provoca una risposta in frequenza non perfettamente piatta. Attivando e disattivano il plugin si nota infatti come si ha un leggero boost sia delle basse frequenze che delle alte frequenze.

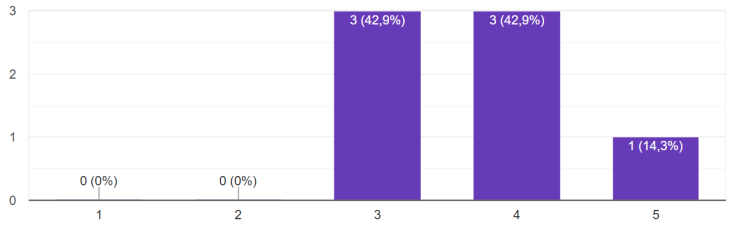
La risposta dei tester

Attraverso un modulo di 10 domande a scelta multipla più una risposta testuale lunga facoltativa si è raccolto il parere di 7 tester. Si tratta di giovani produttori musicali amatoriali o semi-professionisti, tutti con esperienza di innumerevoli plugin provati nel coltivare la passione della produzione musical, del sound design o del mixing, a seconda dei casi. Prima della compilazione e del testing è stato spiegato quale fosse lo scopo del plugin e quale il funzionamento di tutti i knob, agendo sull'esempio di su un possibile use case. Si riportano le domande che gli sono state sottoposte e le relative risposte, successive al testing effettuato su un use case a loro piacere.

[2] Colorazione: leggera equalizzazione

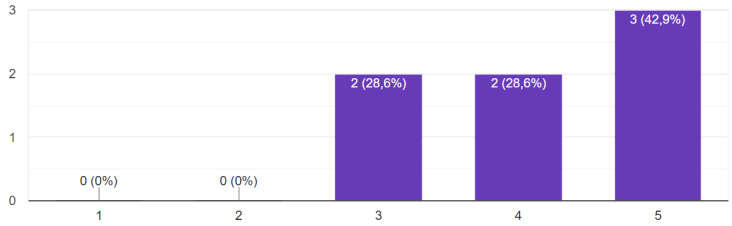
Come valuti l'**originalità** dell'idea dietro al plugin e del suo scopo principale? Copia

7 risposte



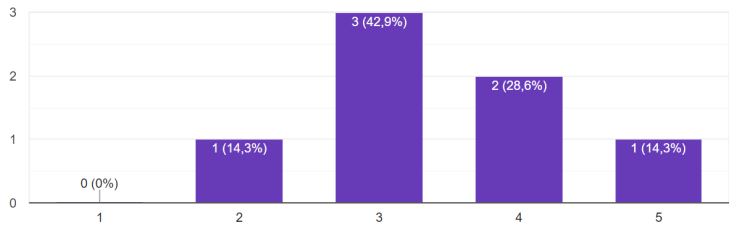
Come valuti l'**appropriatezza dei parametri** scelti rispetto agli obiettivi proposti? Copia

7 risposte



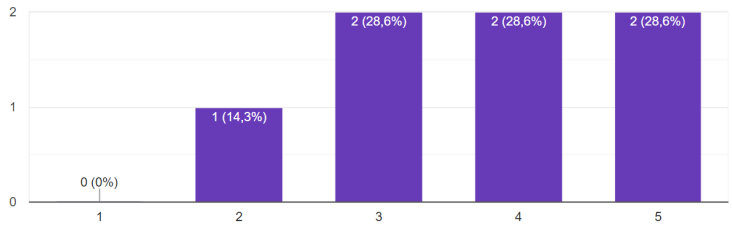
Come valuti la **gradevolezza del filtraggio** effettuato? Copia

7 risposte



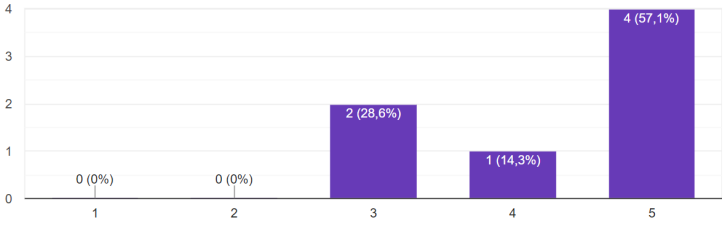
Come valuti la **qualità del segnale** di output? Copia

7 risposte



Come valuti il **peso** del plugin sulla **CPU**? Copia

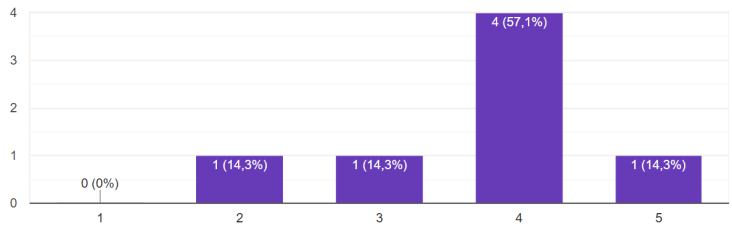
7 risposte



Come valuti la **robustezza** del plugin?

Copia

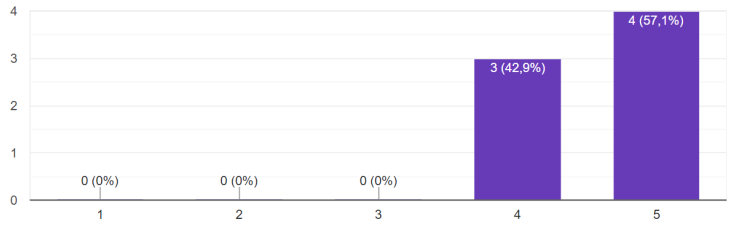
7 risposte



Come valuti l'**interfaccia** grafica a livello **visivo**?

Copia

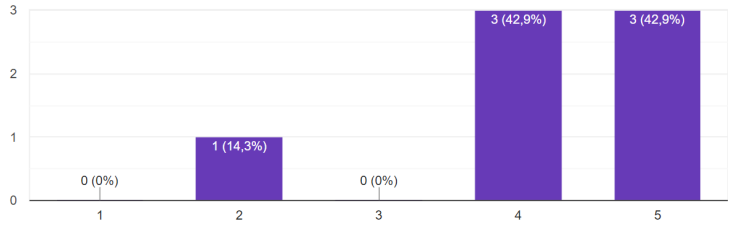
7 risposte



Come valuti l'interfaccia grafica a livello di **chiarezza**?

Copia

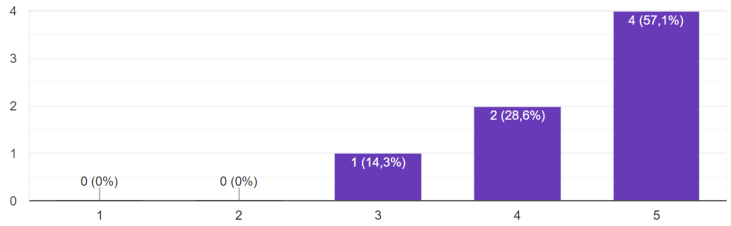
7 risposte



Come valuti l'esperienza di uso del plugin al **tocco**?

Copia

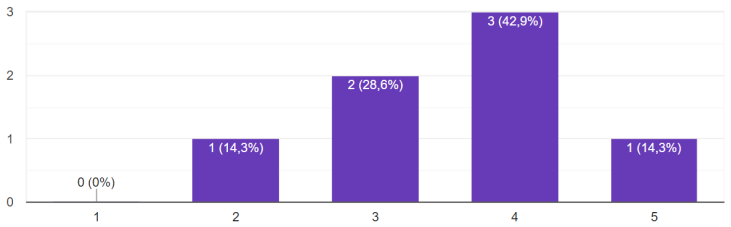
7 risposte



Come valuti l'**utilità** del plugin?

Copia

7 risposte



Nella tabella a fondo pagina si riportano media, mediana e varianza delle risposte, che si sono rivelate inaspettatamente discretamente positive in quasi tutti i 10 campi, ognuno relativo a un aspetto diverso del software e della UX. Gli aspetti in cui il test ha rilevato valutazioni relativamente più basse sono i seguenti:

- La gradevolezza - probabilmente a causa delle distorsioni di fase in casi di utilizzo spinto.
- L'utilità - comprensibile in quanto non tutti i tester utilizzano effetti dall'intervento leggero, pensati per un aspetto tecnico del mix più che per la fase creativa del sound design.

Leggermente più alta è la media dei valori di:

- La robustezza - probabilmente a causa di problemi nell'installazione e altri bug.
- L'originalità - probabilmente per la conoscenza dei comparable analizzati nel capitolo "Stato dell'arte"
- La qualità del segnale ricostruito - imprecisa nella risposta in frequenza, probabilmente a causa delle distorsioni di fase (e conseguente alterazione della risposta in frequenza).

Il risultato generale è complessivamente soddisfacente considerando l'esperienza e le competenze dei 7 tester, alcuni dei quali sono andati a confrontare l'effetto con Track-Spacer della Waves o Soothe2, notandone delle differenze e dei use case in cui TheMasker potrebbe sostituirli o addirittura avere un comportamento più gradito. Per canali informali, tra le altre cose, sono stati fatti numerosi apprezzamenti alla GUI, che anche dalle risposte risulta unanimamente gradita.

	MEDIA	MEDIANA	VARIANZA
Originalità	3,71	4	0,57
Scelta parametri	4,14	4	0,81
Gradevolezza effetto	3,43	3	0,95
Qualità segnale	3,71	4	1,24
Peso CPU	4,29	5	0,90
Robustezza	3,71	4	0,90
Interfaccia grafica	4,57	5	0,29
Comprensibilità	4,14	4	1,14
Esperienza tocco	4,43	5	0,62
Utilità	3,57	4	0,95

Figura 47: Sommario delle risposte al form

Conclusioni

Con TheMasker ci si è cimentati in quella che è stata la prima esperienza di sviluppo di un VST. L'idea, che richiedeva un sistema di analisi robusto e un modello matematico in grado di emulare il mascheramento psicoacustico, era particolarmente ambiziosa. I risultati però hanno superato ogni aspettativa, in quanto quello che era pensato come un prototipo di stampo accademico, come è emerso dai pareri dei tester e dall'esperienza utente, è effettivamente un software utilizzabile nella fase di missaggio o mastering di un brano, o in un generico progetto di sound design.

7.2.4 Obiettivi raggiunti

Ricordando gli obiettivi definiti inizialmente essi si distinguevano nelle tre macro aree di analisi psicoacustica, elaborazione del segnale e interfaccia utente, li andiamo ad analizzare per valutarne il raggiungimento.

Analisi psicoacustica

L'implementazione svolta porta a termine tutti tutti gli obiettivi prefissati per l'analisi psicoacustica, che si riportano qui sotto.

L'analisi psicoacustica deve essere in grado di fornire in output una curva:

- In tempi assimilabili al real-time (con latenza di massimo 80 ms)
- Costituita da un numero di punti in grado di rappresentare il segnale analizzato con sufficiente risoluzione [3]

[3] Risoluzione minima: il numero di bande critiche secondo la teoria posizionale sviluppata da Hermann Von Helmoltz (metà '800) [2] è 25

- In grado di rappresentare la soglia di mascheramento introdotto dal segnale accuratamente sia nei valori di magnitudine (asse y) che di frequenza (asse x)

L'unica causa di latenza si è rivelata la FFT, che introduce un ritardo di $\frac{1024}{sampleRate}$ samples. Con una *sampleRate* di 44100 Hz la latenza introdotta dalla richiesta di 1024 punti in anticipo a causa della FFT è di 23,2 ms. Con 23,2 ms circa di latenza, un numero di bande di 32 e una rappresentazione real-time del mascheramento abbastanza accurata - con le approssimazioni del caso - si ritiene di aver superato con successo tutti i test.

Elaborazione del segnale

Per quanto riguarda l'elaborazione del segnale i risultati sono complessivamente positivi ma non così eccellenti. Riportiamo gli obiettivi previsti:

Il processing del segnale dovrà:

- Essere abbastanza rapido da fornire un segnale in uscita in tempo reale (tempi inferiori ai 10 ms)
- Essere coerente con il settaggio dei parametri scelto dall'utente
- Garantire il più possibile la linearità di fase, il preservamento della qualità del suono e l'assenza di glitch, distorsioni e artifici digitali
- Fornire in uscita un segnale che presenti un'equalizzazione effettivamente utile per gestire il mascheramento causato del segnale *sidechain*.
- Avere un andamento nel tempo smussato e liscio, in modo da non avere variazioni repentine dell'effetto dell'elaborazione e garantirne la gradevolezza

I risultati sono positivi per quanto riguarda la latenza, misurata 2,01 ms (totali misurati $25,22ms - 23,21ms$ di FFT) con la DAW FL Studio 20 eseguita su un pc con CPU Intel Core i7-9750H (@ 2.60GHz) con 128 sample di buffersize e 44100Hz di sample rate. La percentuale di CPU utilizzata fluttua attorno all'11%, si ritiene dunque un plugin sostenibile dalla maggior parte dei processori.

La coerenza dei parametri è analogamente positiva, in quanto il mapping di questi ultimi è risultato soddisfacente anche dai pareri dei tester.

Per quanto riguarda la linearità di fase invece risulta una carenza sopratutto attorno alle frequenze di crossover dei filtri LR, specialmente nei momenti in cui le due bande adiacenti hanno valori di *delta* - dunque di guadagno - diversi da 0 e diversi tra loro. L'SNR misurato conferma tale tesi. Per quanto riguarda glitch e distorsioni l'obiettivo è raggiunto, in quanto la CPU non viene sovraccaricata e la risposta in frequenza è piatta.

L'utilità dell'effetto sull'intellegibilità del segnale è chiara nella teoria e si evince anche nella pratica, anche se non risulta facilmente apprezzabile. Sicuramente lo scopo prefissato era per un uso tecnico e non creativo, quindi l'effetto era previsto che non fosse apprezzabile da tutti.

La gradevolezza è garantita dai tempi di smoothing dell'andamento nel tempo dell'effetto, che come si è detto nel paragrafo "Lo smoothing del delta" sono stati aumentati rispetto ai limiti teorici in modo da venire incontro alla gradevolezza dell'esperienza utente piuttosto che alla correttezza teorica.

L'esperienza utente

Anche la UX ha raggiunto tutti gli obiettivi designati. Li riportiamo qui sotto:

> Essa [la UX] deve dunque:
>
> - Garantire la comprensibilità dell'analisi del segnale, dello stato dell'elaborazione e, in generale, del funzionamento del software
> - Seguire la dinamica dei segnali e dell'elaborazione nel tempo in maniera comprensibile
> - Permettere un controllo dei parametri consapevole, comodo e adeguato alla funzione proposta
> - Fornire un'esperienza visiva coerente e gradevole

Gli spettri e il plot del *delta* garantiscono la comprensibilità dell'analisi, dello stato dell'elaborazione e di ciò che fa il software, ovvero un filtraggio variante in tempo reale.

Essi sono anche implementati con andamento smooth - classe *CustomSmoothedValue* - per favorirne la comprensibilità. La possibilità di nascondere i singoli spettri cliccandone le rispettive indicazioni sulla legenda, e la presenza della stessa, garantisce il raggiungimento dell'obiettivo.

La consapevolezza del controllo dei parametri forse non è immediato e richiederebbe un tutorial; esso inoltre dipende dal grado di conoscenza degli utenti del fenomeno del mascheramento in frequenza. Tuttavia l'effetto che il tocco dei parametri produce sulla curva è estremamente chiaro e può essere intuito facilmente anche dopo un solo utilizzo.

La gradevolezza dell'esperienza visiva e della GUI è considerato un obiettivo raggiunto, confermato dalle risposte ai questionari, in cui la domanda relativa ammonta a un valore medio di 4 punti pieni su 5.

7.2.5 Known issues

La build del VST è stata lanciata con i formati .vst3 e .AU come formati di destinazione. In particolare l'estensione .AU è dedicata al solo sistema operativo macOSX, mentre .vst3 è supportato da Windows, macOSX e Linux. La build di quest'ultimo non ha problemi su nessun sistema operativo.

Il sistema operativo proprietario della Apple invece sembra dare problemi di diversa natura nel riconoscimento del plugin. A seconda del dispositivo, sono stati riscontrati i seguenti problemi per gli utenti macOS:

- La configurazione delle tracce di *input* mono e *sidechain* stereo (che prevede la duplicazione dell'*input* e un filtraggio dei due canali differente a seconda dei due diversi valori del delta dei due canali) produce un crash del plugin. Esso funziona dunque solo nei restanti casi: *input* mono e *sidechain* mono, *input* stereo e *sidechain* mono (il delta sarà uguale per entrambi i canali), *input* stereo e *sidechain* stereo.

- Sul software Logic Pro X, che supporta solo plugin .AU, TheMasker può non passare il test di validazione. Dal log di errore si evince un problema relativo al supporto delle diverse configurazioni del numero di canali, sicuramente legata al punto precedente, ma

che non si è riusciti a risolvere. Per fortuna non in tutte le macchine avviene questo problema.

- Anche sul software Reaper può capitare che sia il formato .vst3 che il .AU non passino i test di validazione, o che una volta effettuato il re-scan dei plugin non compaiano nella lista dei VST disponibili.

7.2.6 Possibili miglioramenti e sviluppi futuri

Essendo il progetto nato come lo sviluppo prototipo, esso necessita di alcuni fix e accorgimenti importanti al fine di diventare un prodotto pronto per il lancio nel mercato. Vediamo quali punti è necessario e quali auspicabile portare a termine al fine di un completamento del progetto e di futuri miglioramenti.

- Il primo fix necessario è quello relativo alla compatibilità con su macOSX del file .vst3 e/o del formato .AU. L'esclusione degli utenti - o di parte di essi - del sistema operativo proprietario della Apple consiste in un privare dell'uso del prodotto una fetta di mercato troppo consistente, sopratutto essendo il target quello dei sound designer, che non è raro che preferiscano il sistema operativo mac. Considerando che si tratta di un bug e che tutte le carte in regola per risolverlo ci siano, sarebbe il primo passo da compiere.

- Inoltre, un altro bug (probabilmente correlato - almeno per la validazione dell'.AU su Logic) da fixare quanto prima è quello della configurazione di canali *sidechain* stereo e *input* mono. Il crash che causa può risultare in una potenziale perdita di tempo, o alla peggio addirittura di un progetto, il che sarebbe inammissibile per qualsiasi utente.

- Una questione tecnica molto importante è inoltre il migliorare la linearità della risposta in frequenza che il plugin crea anche quando il mix è a 0. Essendo pensato per un uso anche solo minimale e di fino, il corrente non raggiungimento dell'obiettivo della perfetta linearità è molto importante che venga risolto. Probabilmente con un settaggio diverso degli elementi dell'array *gain_adjustments* si può correggere il guadagno eccessivo degli estremi di banda con rapidità ed efficienza.

- Per una migliore esperienza utente, sarebbe ottimale lo sviluppo di un sistema di tutorial iniziale (al primo utlizzo) che spieghi i pochi parametri esposti all'utente e auspicabilmente anche un sistema di popup di piccoli paragrafi all'hover del mouse sui di essi - anche negli utilizzi successivi al primo. Probabilmente anche un indicatore dei dB e degli Hz a cui corrisponde la posizione del mouse nello spettro sarebbe un integrazione comoda per alcuni utilizzi.

- Infine, come suggerito da uno dei tester, l'integrazione di un knob aggiuntivo che permetta di dosare il filtraggio sulle componenti del segnale stereo "mid" e "side": spesso l'utilizzo potrebbe essere necessario per la sola parte centrale dell'immagine stereo, o solo per quella laterale,

In conclusione, nonostante l'inesperienza e le difficoltà del caso si può dire di aver raggiunto con successo l'obiettivo proposto, pur essendo ambizioso. L'idea di TheMasker è stata portata a termine con successo e non resta che pensare a come, se si è nell'interesse, rendere disponibile al pubblico il prodotto.

Bibliografia

Bibliografia

[2] Hermann L. F. von Helmholtz. ≪Die Lehre von den Tonempfindungen, als Physiologische Grundlage für die Theorie der Musik≫. In: Vieweg, Braunschweig, 1863.

[7] Wwe-Whei Chang Chi-Min Liu. ≪CHAPTER 4 - Audio Coding Standards≫. In: *Multimedia Communications* (2001), pp. 45–60. DOI: https://doi.org/10.1016/B978-012282160-8/50005-0.

[9] A. Valle V. Lombardo. ≪Audio e multimedia≫. In: Apogeo Education, 2014. Cap. 2.3.

Sitografia

[1] Wikipedia. *Audio Ducking*. [Online; Accessed 02 February 2023]. 2023. URL: https://en.wikipedia.org/wiki/Ducking.

[3] Miro. *Miro*. 2011. URL: https://miro.com/it/.

[4] Mathworks. *MATLAB*. 1984. URL: https://it.mathworks.com/products/matlab.html.

[5] J. Storer. *JUCE*. 2004. URL: https://juce.com/.

[6] A. Fresia N. Degiorgi. *Schema a blocchi Miro*. 2022. URL: https://miro.com/app/board/uXjVPR9-ThY=/?share_link_id=322010253442.

[8] Wikipedia. *Auditory Masking*. [Online; Accessed 31 January 2023]. 2023. URL: https://en.wikipedia.org/wiki/Auditory_masking#:~:text=onset%5C%20attenuation%5C%20lasting%5C%20approximately%5C%2020%5C%20ms%5C%20and%5C%20the%5C%20offset%5C%20attenuation%5C%20lasting%5C%20approximately%5C%20100%5C%20ms.

[10] Wikipedia. *Critical band*. [Online; Accessed 10 March 2023]. 2023. URL: https://en.wikipedia.org/wiki/Critical_band.

[11] R. Profeta G. Schuller. *Audio Coding - psychoAcousticsModels*. [Online; Accessed 17 June 2022]. 2020. URL: https://colab.research.google.com/github/GuitarsAI/AudioCodingTutorials/blob/master/AC_05_psychoAcousticsModels.ipynb.

[12] Fabien Petitcolas. *MPEG for Matlab*. [Online; Accessed 13 February 2022]. 2003. URL: https://www.petitcolas.net/fabien/software/mpeg/index.html.

[13] Mathworks. *Butter filter*. 2023. URL: https://it.mathworks.com/help/signal/ref/butter.html.

[14] Mathworks. *Zero-pole-gain filter parameters to second-order sections form conversion function*. 2023. URL: https://it.mathworks.com/help/signal/ref/zp2sos.html.

[15] Mathworks. *Zero-phase digital filtering*. 2023. URL: https://it.mathworks.com/help/signal/ref/filtfilt.html.

[16] Figma. *Figma*. 2016. URL: https://www.figma.com/.

Acronimi

DAW Digital Audio Workstation

VST Virtual Studio Technology

UX User Experience

FFT Fast Fourier Transform

ATQ Absolute Threshold in Quiet

GUI Graphical User Interface

UML Unified Modeling Language

IDE Integrated Development Environment

UI User Interface

RMS Root Mean Square

LR Filtri Linkwitz-Riley

IIR Infinite Impulse Response

SNR Signal to Noise Ratio

VSTi Virtual Studio Technology Instrument

MIDI Musical Instrument Digital Interface

LnF Look And Feel