UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
Corso di Laurea in: INFORMATICA MUSICALE

IMPLEMENTAZIONE DI UN PROCESSORE DI DINAMICA MULTI-BANDA PER L'OTTIMIZZAZIONE AUTOMATICA DI ELEMENTI DEL MIX

Studente:
Manuel Sandri
(matricola 815659)

Relatore:
Professor Luca Andrea Ludovico

Correlatore:
Dottor Giorgio Presti

Anno Accademico 2014/2015

Indice

Introduzione

Nell'elaborato in questione si è analizzato il processo di realizzazione di un processore di dinamica multi-banda, nel caso particolare di un compressore di dinamica multi-banda, il cui scopo fosse quello di automatizzare le azioni che l'utente finale dovrebbe compiere con l'utilizzo di questa tipologia di processori.
La realizzazione del software ha richiesto un precedente lavoro di analisi sui procedimenti svolti nella compressione multi-banda in ambito professionale di mixaggio per poter ricavare i dati relativi ad una serie di tracce audio su cui tale processo viene maggiormente applicato, in modo tale da raccogliere un insieme di dati che portassero ad implementare il processo di automatizzazione.
L'elaborato inizia con un'analisi dello stato dell'arte in cui si descrivono i processori di dinamica a banda larga, focalizzandosi sulla descrizione dei compressori di dinamica e di tutti i parametri in gioco in questa tipologia di processori. Si prosegue poi con la descrizione dei processori di dinamica multi-banda e dei parametri aggiuntivi che questi processori comportano. Si descrivono poi alcuni tra i principali processori di dinamica multi-banda presenti in commercio: C4 e C6 di casa Waves, Ozone 6 Dynamics di casa iZotope. Infine vengono illustrate le motivazioni che hanno portato alla realizzazione del processore di dinamica in esame e che hanno portato alla realizzazione di due versioni del software:

- versione con interfaccia semplificata, in cui è solamente possibile scegliere lo strumento musicale su cui si desidera operare, impostare il livello in ingresso del segnale ed il livello di make up gain di ogni banda, per evitare agli utenti meno esperti l'onere di dover agire su un numero molto grande di parametri che, se impostati su valori errati, potrebbero comportare risultati sgradevoli;
- versione con interfaccia completa, con la quale gli utenti più esperti, oltre ad avere un'automatizzazione del settaggio dei parametri, possono avere un controllo completo di tutti i parametri del processore, comprese la possibilità di scegliere la tipologia di rilevatore di inviluppo e la tipologia di funzione di trasferimento.

L'elaborato prosegue poi con la descrizione delle tecnologie utilizzate per la realizzazione del software in esame. Viene descritto il linguaggio di programmazione C++, facendo riferimento allo standard ISO/IEC 14882:1998 e successive versioni (realizzate nel 2003, nel 2011 e nel 2014) e descrivendo brevemente la storia del linguaggio. Vengono poi descritti il framework JUCE, usato per la configurazione del progetto e dell'interfaccia grafica del software, e l'ambiente di sviluppo Apple Xcode, usato per la scrittura del codice e per la compilazione del software. Vengono descritte le tecnologie di DSP (Digital Signal Processing) utilizzate per suddividere lo spettro del segnale

audio in quattro bande di frequenza: si descrivono i filtri ideali, proseguendo poi con la descrizione dei corrispettivi filtri analogici soffermandosi in particolare sui filtri di Butterworth, in quanto utilizzati per la realizzazione dell'applicativo in esame. Infine viene descritta la libreria utilizzata per l'implementazione dei filtri all'interno del plugin.

Nell'ultima parte dell'elaborato viene discusso tutto il processo implementativo, a partire dalla descrizione delle principali funzionalità del software in esame e proseguendo con la descrizione dei procedimenti svolti per l'implementazione di queste funzionalità, presentando e commentando per ognuna di esse il codice sorgente sviluppato. Viene inoltre descritta brevemente la realizzazione delle due interfacce grafiche.

Capitolo 1:
Analisi dello stato dell'arte e motivazioni per la realizzazione del software in esame

1.1 – Analisi sullo stato dell'arte dei processori di dinamica multi-banda

Nell'ambito dell'elaborazione audio digitale, in particolare in ambito di mixaggio e mastering, al giorno d'oggi, sono disponibili numerosi software, in particolare plug-in (VST, Audio Units, RTAS, AAX, ecc.) per digital audio workstation (DAW), che svolgono la funzione di processori di dinamica.

1.1.1 – Processori di dinamica a banda larga e multi-banda

Esistono diverse tipologie di processori di dinamica. La suddivisione principale può essere fatta tra processori a banda larga, i quali agiscono su tutto lo spettro del segnale audio, e processori di dinamica multi-banda, i quali suddividono lo spettro del segnale audio in ingresso in due o più bande e agiscono su ognuna di esse senza influenzare le bande rimanenti.

Vi sono quattro tipologie principali di processori di dinamica: compressori di dinamica, i quali riducono il range dinamico del segnale, diminuendo il livello del segnale dinamicamente quando esso supera un determinato da un valore di soglia, con una riduzione determinata dal valore dato dal rapporto di compressione (ratio); limiter, i quali fanno in modo che il segnale audio sia limitato al valore di soglia e che non superi tale valore; expander, i quali incrementano il range dinamico del segnale audio processato riducendo il livello del segnale che sta al di sotto del valore di soglia; gate, utilizzati per rimuovere il rumore di fondo che si trova tra suoni più forti, i quali riducono il livello del segnale che si trova al di sotto del valore di soglia, e, rispondendo molto rapidamente ad un cambiamento del livello di segnale, quando esso raggiunge e supera il valore di soglia smette immediatamente di decrementarne il livello. [1]

I plug-in che svolgono la funzione di processori di dinamica sono accomunati da alcuni parametri fondamentali su cui operano. Questi parametri sono threshold, make up gain, rapporto di compressione, tempo di attacco e tempo di rilascio. Alcuni permettono di operare anche sull'angolo della curva di risposta al processo (“knee”).

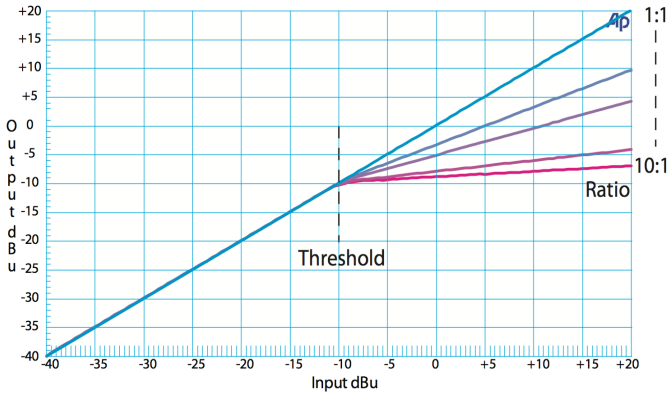
Poiché il software in esame svolge la funzione di compressore di dinamica, i parametri verranno ora descritti nell'ambito della compressione di dinamica.

La threshold (o “soglia”) indica il livello minimo del segnale in ingresso (generalmente espresso in decibel full scale) da cui il processore di dinamica può iniziare ad operare. Se il livello del segnale

in ingresso è inferiore al valore indicato dalla threshold il processore non si attiverà e il range dinamico del segnale audio non verrà ridotto.

Il make up gain, espresso in decibel full scale, permette di regolare il livello in uscita del segnale processato dal compressore. Solitamente questo parametro è compreso in un intervallo che va da -12 dB a +12 dB, anche se alcuni processori di dinamica possono avere un intervallo di make up gain maggiore.

Il rapporto di compressione (ratio) determina il rapporto ingresso/uscita per i segnali al di sopra della threshold. Ad esempio, con un rapporto di compressione 1:1, se il segnale in ingresso subisce una variazione di 2 dB anche il segnale in uscita subirà una variazione di 2 dB.[1] Se invece il rapporto di compressione fosse di 10:1, una variazione dell'input di 10 dB provocherebbe una variazione di 1 dB all'output.



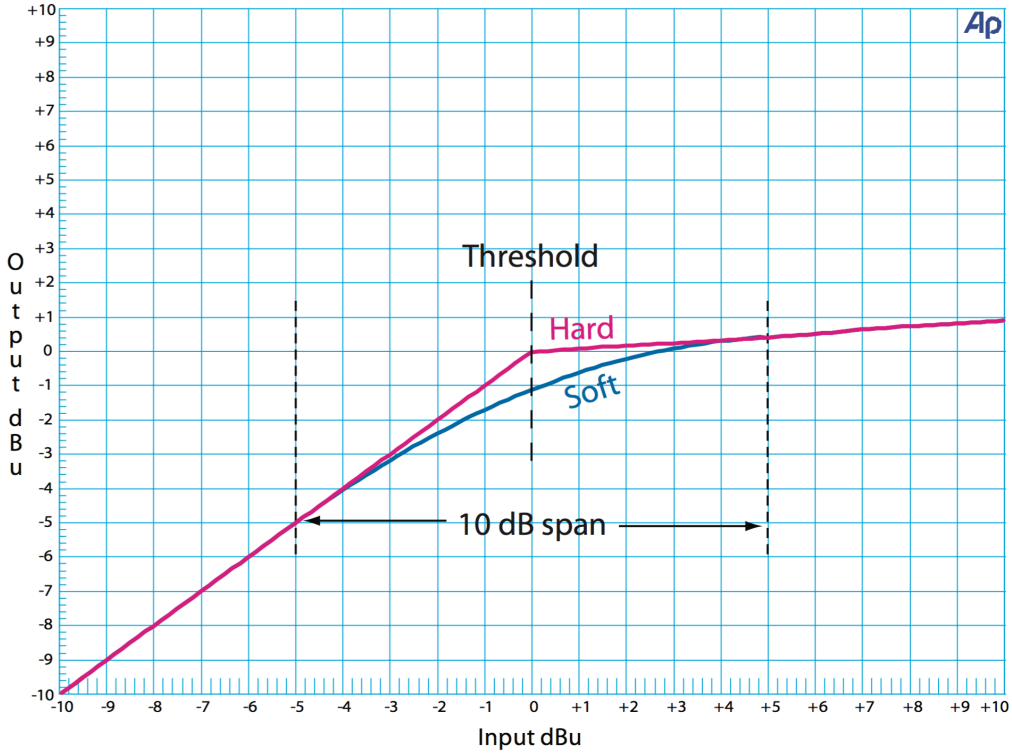
Rappresentazione di come agisce un compressore di dinamica alla variazione del rapporto di compressione [1]

Il tempo di attacco (attack) indica quanto rapidamente il compressore risponde ad un incremento del livello del segnale in ingresso al di sopra della threshold. Definisce infatti quanto rapidamente viene ridotto il guadagno del segnale. Espresso in millisecondi, il tempo di attacco generalmente può essere compreso in un intervallo che va dai 25 ms ai 500 ms.

Il tempo di rilascio (release) indica quanto rapidamente il compressore risponde ad un decremento del livello del segnale in ingresso al di sotto della threshold. Definisce infatti quanto rapidamente il guadagno del segnale viene riportato al livello precedente al processo di compressione. Espresso in millisecondi, il tempo di rilascio solitamente è compreso in un intervallo che va dai 25 ms ai 2000 ms (2 s).

Come accennato prima, talvolta è presente un parametro che regola l'angolo della curva di risposta alla compressione (knee). Questo parametro controlla l'azione del compressore di dinamica al livello della soglia (threshold). Si hanno due estremi per l'angolo della curva, entro i quali il parametro knee aussume i suoi valori: “soft knee” quando viene aumentato gradualmente il livello di compressione fino a raggiungere il valore prefissato; “hard knee” quando si giunge

immediatamente al livello di compressione prefissato. [1]



Rappresentazione delle due tipologie di knee. [1]

I processori di dinamica, per ottenere il livello del segnale da processare, possiedono al loro interno dei rilevatori di inviluppo. Questi rilevatori si dividono in due tipologie, che sono rilevatori di picco e rilevatori di RMS (Root Mean Square). I processori di dinamica solitamente possiedono un rilevatore di RMS, ma alcuni possiedono entrambe queste tipologie di rilevatori.

I rilevatori di picco, utilizzati per rilevare il valore di inviluppo di segnali audio a transienti molto rapidi, in genere calcolano il livello massimo all'interno di una finestra mobile del segnale. Questo calcolo viene effettuato per ogni transiente del segnale.

I rilevatori di RMS, utilizzati per rilevare il valore di inviluppo di segnali audio sostenuti, calcolano il valore efficace del segnale: calcolano la radice quadrata della media dei valori al quadrato che assume il segnale.

$$x_{\mathrm{rms}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

Formula di RMS per i segnali discreti.

Inoltre si possono utilizzare diverse funzioni di trasferimento per processare il segnale. Nel processore in esame sono state utilizzate due tipologie di funzioni di trasferimento, una di tipo esponenziale e una che utilizza funzioni lineari.

Un compressore di dinamica a banda larga può essere descritto dal seguente diagramma a blocchi:

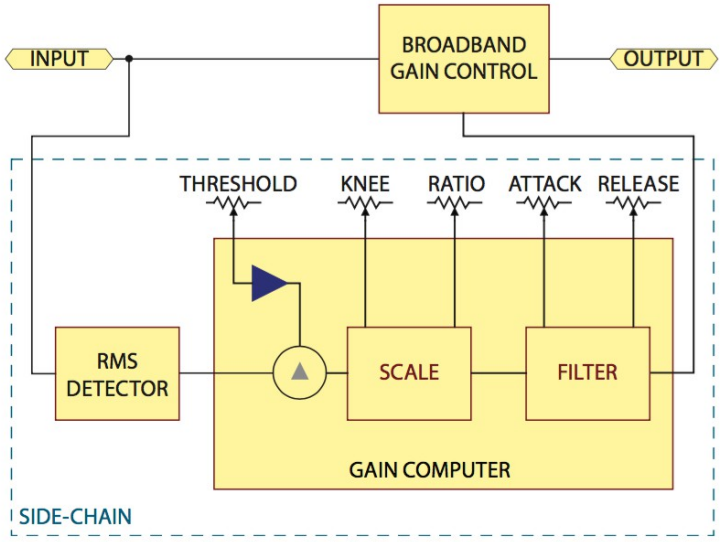


Diagramma a blocchi per un compressore di dinamica [1]

I processori di dinamica multi-banda, suddividono lo spettro del segnale audio in ingresso in due o più bande, ognuna delle quali ha un rilevatore di inviluppo, e su di esse viene applicato un processore che funziona in modo indipendente dagli altri: ciascun processore non va ad influenzare l'operato dei rimanenti. Dopodiché le bande vengono ricombinate in un unico segnale in uscita.

I processori applicati ad ogni banda possiedono i parametri considerati in precedenza per ognuno dei processori applicati alle bande. Poiché nei processori di dinamica multi-banda vi è un numero di processori di dinamica pari al numero di bande di frequenza presenti, tali parametri saranno ripetuti per ognuno dei compressori presenti. Il parametro “knee” può essere ripetuto per ogni compressore, oppure può essere unico e va ad incidere contemporaneamente sulla curva di ognuno dei compressori presenti nel software. Alcuni software trattano il valore di ratio come un range entro il quale vi è una riduzione o un aumento di livello, anziché come un rapporto vero e proprio.

Oltre ai parametri citati in precedenza, i processori di dinamica multi-banda permettono di scegliere le frequenze di crossover, cioè le frequenze in cui i filtri si intersecano tra loro.

Un compressore di dinamica multi-banda può essere rappresentato dal diagramma a blocchi presente nella pagina successiva.

E' possibile notare nel diagramma a blocchi la presenza di alcuni filtri che permettono la divisione in bande: nel diagramma in esame sono presenti un passa-basso e un passa-alto, ma nel caso in cui il segnale in ingresso fosse diviso in quattro bande sarebbero presenti anche altri due filtri passa-banda posti tra il filtro passa-basso e il filtro passa-alto. Il segnale contenuto in ogni banda è poi processato da un compressore di dinamica, uno per ogni banda come si nota in figura, e successivamente le bande vengono sommate tra loro ricostruendo così un unico segnale di uscita.

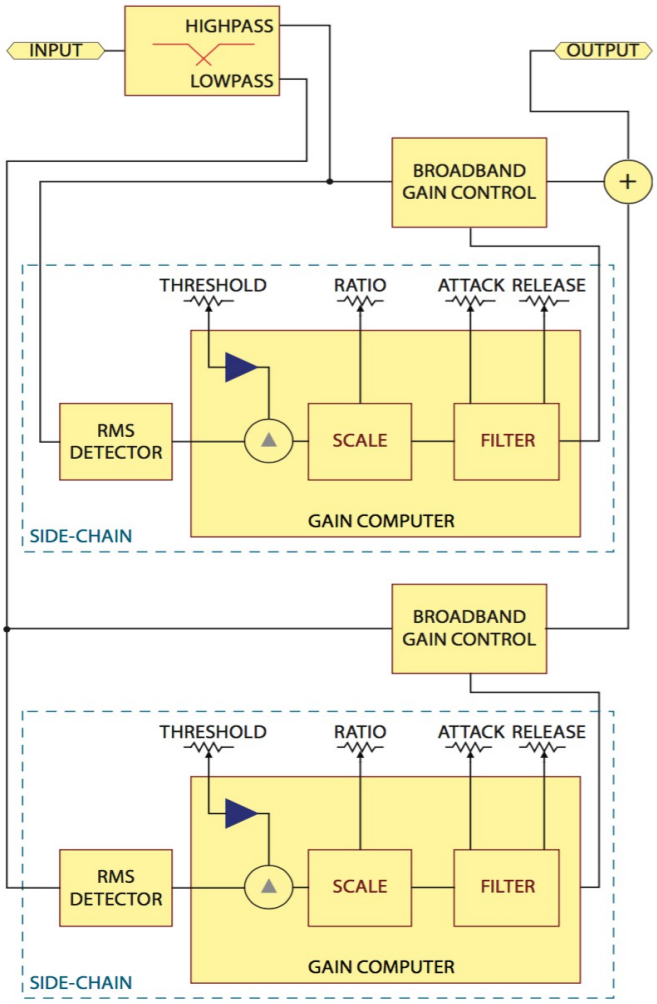


Diagramma a blocchi per un compressore di dinamica multi-banda a due bande [1]

1.1.2 – Alcuni processori di dinamica multi-banda presenti in commercio

In commercio sono presenti molti software che agiscono come dei processori di dinamica multi-banda. Tali software sono realizzati da varie software house. Alcuni tra i più conosciuti sono i seguenti:

- "C4", realizzato da Waves; si tratta di un compressore (funziona anche come expander) di dinamica multi-banda che divide lo spettro del segnale in quattro bande e applica una compressione su ognuna di esse in maniera indipendente. Questo plugin può essere utilizzato anche come equalizzatore a quattro bande. Può operare sia su tracce audio mono che su tracce audio stereo. Questo software invece del parametro ratio presenta un parametro *"range"* che determina un intervallo entro il quale il segnale può essere compresso (o espanso se usato come expander). Il parametro knee è unico e va ad influire su ognuno dei processori presenti.

Interfaccia grafica del plugin “C4” di casa Waves

- “C6”, realizzato da Waves, il quale, a differenza del C4, opera su sei bande di frequenza. Come il C4, anche il C6 funziona anche come expander e può essere utilizzato come equalizzatore. Inoltre questo plugin permette anche di lavorare in modalità “side chain”. Può operare sia su tracce audio mono che su tracce audio stereo.

Interfaccia grafica del plugin “C6” di casa Waves

- “Ozone Dynamics”, realizzato da iZotope e giunto ora alla sesta versione; si tratta di un compressore di dinamica multi-banda adatto principalmente ad essere utilizzato per il mastering. Oltre che come compressore di dinamica multi-banda (lavora su 4 bande), può essere utilizzato come limiter e come expander. Inoltre permette di utilizzare tre algoritmi di rilevazione di inviluppo, tramite picchi, RMS oppure tramite un processo di rilevazione di inviluppo simile al metodo RMS, ma più accurato. Può operare sia su tracce audio mono che su tracce audio stereo. In questo software il parametro knee può essere modificato per ognuno dei processori presenti.

Interfaccia grafica del plugin “Ozone 6 Dynamics” di casa iZotope

1.2 – Motivazioni per la realizzazione del processore di dinamica oggetto dell'elaborato

Come descritto nel paragrafo 1.1, i software che svolgono la funzione di compressori di dinamica multi-banda, sono costituiti da numerosi controlli, in quanto per ogni banda di frequenza sono solitamente ripetuti i parametri di regolazione di threshold, make up gain, tempo di attacco, tempo di rilascio e knee.

Tutto questo insieme di controlli rende ostico ad un utente poco esperto l'utilizzo di un software di compressione multi-banda del segnale audio. Infatti una regolazione errata di alcuni di questi parametri porterebbe a risultati finali indesiderati.

Da ciò nasce la necessità di ridurre il numero di controlli su cui deve agire l'utente.

Dopo un continuo utilizzo di software per la compressione di dinamica multi-banda, in particolare il C4 di Waves, analizzando i preset presenti all'interno dei software ed analizzando il lavoro di professionisti nell'ambito del sound engeneering, si è notato che per ogni strumento musicale solitamente si utilizzano delle regolazioni specifiche sui parametri analizzati nel paragrafo precedente. Prendendo in esame più casi di utilizzo di compressori di dinamica multi-banda sulla stessa tipologia di strumenti musicali, anche se suonati da differenti musicisti, si è notato che le

regolazioni per ogni parametro non si discostavano di molto tra loro.
Ciò ha portato quindi all'idea di automatizzare il processo di compressione multi-banda.
Si è quindi realizzato un software con un'interfaccia costituita dalla possibilità di controllare solo il make up gain delle bande di frequenza e il livello di tutto il segnale in uscita. Questo software determinaa internamente i valori relativi a threshold, tempo di attacco e tempo di rilascio in maniera differente sulla base dello strumento musicale scelto dall'utente finale; viene inoltre regolato in modo automatico anche il valore delle frequenze di crossover. Infine vengono decise automaticamente la tipologia di algoritmo utilizzato dal processore e la tipologia di rilevazione di inviluppo.
Per gli utenti più esperti è stata realizzata una versione del software in cui sono presenti tutti i parametri per la regolazione di ognuno dei compressori, i quali si settano automaticamente (come dei preset) sulla base dello strumento musicale su cui si sta lavorando in fase di mixaggio, ma possono essere comunque modificati dall'utente, come in un classico processore di dinamica multi-banda. Inoltre questa versione permette di scegliere se utilizzare un rilevatore di RMS oppure un rilevatore di picco, e permette anche di scegliere la tipologia di funzione di trasferimento. Questa è una caratteristica molto rara nei processori di dinamica presenti in commercio.

Capitolo 2:
Tecnologie utilizzate per lo sviluppo del software in esame

Per lo sviluppo del processore di dinamica multi-banda in esame, denominato *"Auto Multiband Comp"*, sono stati usati i seguenti strumenti di sviluppo: linguaggio di programmazione C++, il framework JUCE, l'ambiente di sviluppo Apple Xcode. Inoltre sono state utilizzate tecnologie in ambito DSP (Digital Signal Processing) per la suddivisione dello spettro del segnale audio in bande di frequenza.

2.1 – Linguaggio di programmazione C++

Il linguaggio di programmazione C++ è un linguaggio di programmazione ad alto livello, reso standard da ISO nel 1998 (ISO/IEC 14882:1998). Si tratta di un linguaggio versatile utilizzato per differenti scopi (general purpose). E' basato sul linguaggio C, descritto nello standard *"ISO/IEC 9899:1999 Programming Languages – C"*. Oltre alle caratteristiche fornite dal linguaggio C, C++ fornisce ulteriori tipologie di dati, di classi, di template, di eccezioni, di namespace, di overload degli operatori, di overload dei nomi delle funzioni, di riferimenti e di operatori per la gestione della memoria. Inoltre fornisce delle librerie aggiuntive rispetto al C. [2]

Il linguaggio C++ richiede che il codice venga compilato. E' fortemente tipizzato, in quanto, quando non si usano istruzioni di conversione specifica, pone restrizioni sulla conversione di variabili da un tipo ad un altro. Infatti richiede che le variabili vengano castate esplicitamente ad un altro tipo. Supporta il controllo di conversione delle variabili in run-time o in compile-time (durante la compilazione). Offre supporto per la programmazione procedurale, generica e in particolare per la programmazione orientata agli oggetti (OOP: Object Oriented Programming). E' uno dei linguaggi maggiormente usati al mondo e offre un vasto catalogo di compilatori che possono essere eseguiti su numerose piattaforme differenti. Inoltre è compatibile con il linguaggio C. [7]

Per la realizzazione del software in esame è stato scelto il linguaggio C++ in quanto supportato da JUCE, piattaforma per lo sviluppo di plugin audio che verrà descritta al paragrafo successivo. Inoltre le caratteristiche del linguaggio hanno reso agevole la programmazione di *"Auto Multiband Comp"*.

2.1.1 – Breve storia del linguaggio C++ [7] [8]

Il linguaggio C++ (denominato in origine "C con classi") fu sviluppato a partire dal 1979 da Bjarne Stroustrup. Nel 1985 venne pubblicato il manuale *"The C++ Programming Language"* scritto da

Stroustrup e nello stesso anno il linguaggio venne per la prima volta come prodotto commerciale. In quell'anno il linguaggio non era ancora stato standardizzato e ciò rese il manuale sopra citato un importante punto di riferimento.
Nel 1989 vi fu un ulteriore aggiornamento al linguaggio e nel 1990 fu pubblicato il manuale *"The Annotated C++ Reference Manual"*.
Nel 1998 un comitato che presentava membri di ANSI (American National Standards Institute) e di ISO (International Organization for Standardization) pubblicò il primo standard internazionale per il linguaggio C++: ISO/IEC 14882:1998. Questo standard era informalmente noto come C++98. Nel 2003 questo standard fu rivisitato per correggere alcune problematiche riscontrate, portando così allo standard C++03 (ISO/IEC 14882:2003). Nel 2005 un report tecnico (TR1) in cui vennero dettagliatamente elencate le caratteristiche presenti nel successivo standard per il linguaggio. A metà del 2011 il nuovo standard venne completato. Fu pubblicato quindi lo standard C++11 (ISO/IEC 14882:2011). Tra il 2013 e il 2014 fu pubblicato un nuovo standard, C++14 (ISO/IEC 14882:2014), definito da ISO uno standard minore in preparazione al nuovo standard C++17. [8]

2.2 – JUCE e Apple Xcode

Per l'implementazione del software di cui si parla in questo elaborato, è stato utilizzato il framework JUCE per tutte le operazioni riguardanti la realizzazione dell'interfaccia grafica e la creazione di file e moduli relativi ai formati del plug-in in esame, creati automaticamente da JUCE al momento della configurazione del progetto. Per la scrittura del codice sorgente relativo alle classi realizzate per il funzionamento del software, e per la compilazione del codice, è stato utilizzato l'ambiente di sviluppo Apple Xcode.

2.2.1 – JUCE (Jules' Utility Class Extensions)

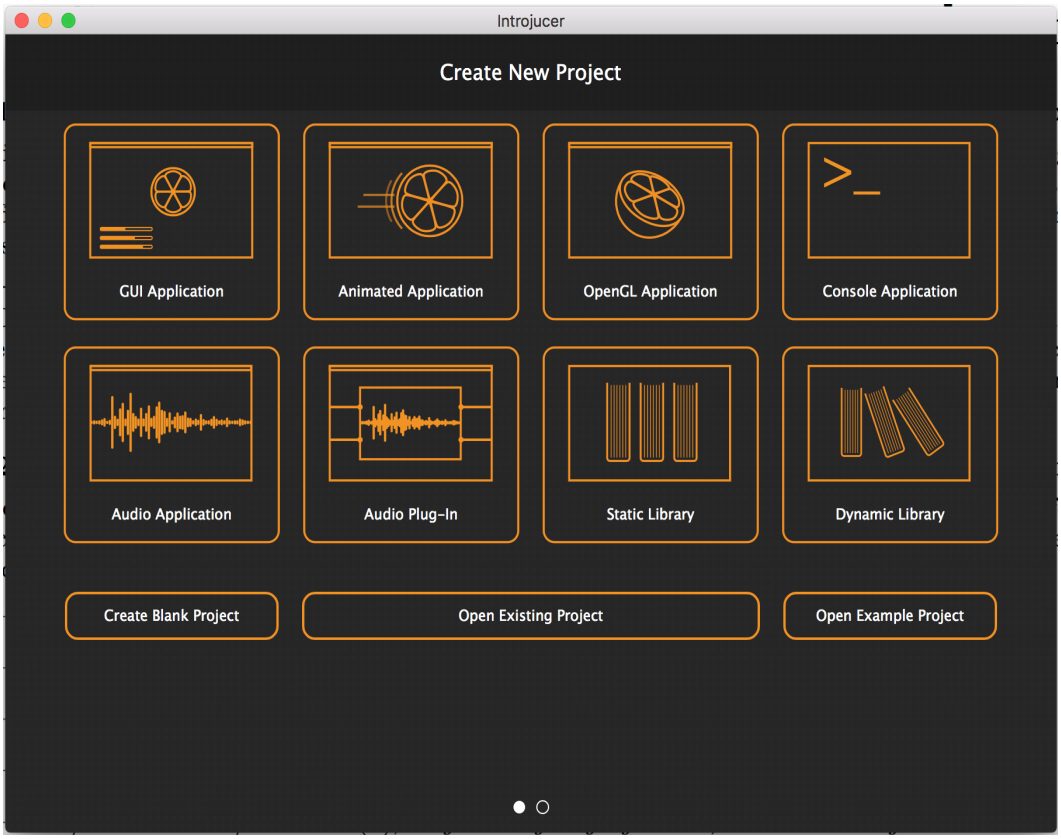
JUCE (Jules' Utility Class Extensions), ora giunto alla versione 4.0, è un framework multi-piattaforma open-source utilizzato per la realizzazione di applicazioni desktop e applicazioni per dispositivi mobili. E' usato in particolare per le sue librerie grafiche e per le sue librerie riguardanti i plug-in audio.
JUCE è stato creato da Julian Storer per Raw Materials Software ltd e successivamente, dopo che Raw Material Software venne acquisita da ROLI ltd nel 2014, anche JUCE divenne proprietà di quest'ultima compagnia.
JUCE facilità la realizzazione di applicazioni che possano funzionare su più piattaforme, ed infatti è supportato da Microsoft Windows, Apple Mac OS, Linux, Apple iOS e da Android. Supporta lo standard MIDI e permette la realizzazione di plug-in audio in ogni formato: VST, VST3, AU (Audio Units), RTAS e AAX.

Per la realizzazione di *"Auto Multiband Comp"* è stata utilizzata la versione 3.2 di JUCE.

JUCE è scaricabile dal sito internet *www.juce.com*, dopo essersi registrati al sito. Con il download verrà scaricato *"Introjucer"*, un IDE che permette di creare e gestire progetti scritti in C++.A differenza di altri IDE, però, non permette la compilazione dei progetti, ma permette di generare progetti compilabili (e modificabili) da IDE come Apple Xcode (usato per la realizzazione del software in esame), Microsoft Visual Studio, Code::Blocks ed altri.

"Introjucer" inoltre permette di gestire i moduli di JUCE che l'utente necessita di utilizzare per i propri progetti, e, grazie al suo supporto interno per la realizzazione di plug-in audio, permette la realizzazione di progetti per plug-in audio creando automaticamente i file e le classi necessarie per ogni formato di plugin evitando al programmatore un ulteriore carico di lavoro.

Oltre ai plug-in audio, *"Introjucer"* permette di impostare anche altre tipologie di progetto, come applicazioni con interfaccia grafica o che funzionano da terminale, applicazioni OpenGL, applicazioni animate, applicazioni audio (DAW, registratori di suoni, ecc.), librerie statiche e librerie dinamiche. [9]

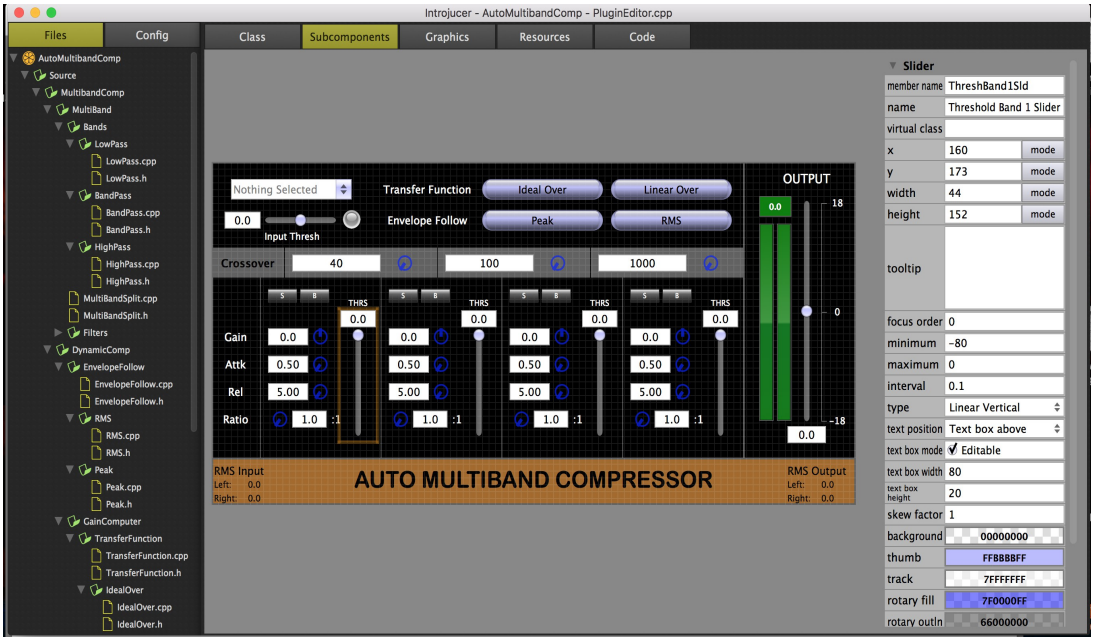


Schermata iniziale di *"Introjucer"* in cui l'utente ha la possibilità di scegliere tra le tipologie di progetti realizzabili con JUCE.

Per la realizzazione di plug-in audio è sufficiente selezionare la voce “Audio Plug-In”. Si aprirà una schermata in cui l'utente può indicare il nome del progetto, i moduli audio di JUCE da utilizzare e le piattaforme verso cui è indirizzato il progetto (con i relativi IDE).

Dopo aver indicato queste informazioni si aprirà all'utente una schermata per configurare le caratteristiche del plug-in: formato (VST, VST3, AU, RTAS, AAX), nome del plug-in, nome della casa produttrice (o del programmatore), tipologia di plug-in (si indica se il plug-in è o meno uno strumento virtuale), supporto MIDI (si indica se il plug-in necessità o meno di segnale MIDI in ingresso e in uscita), numero di canali in ingresso e in uscita, e altri dati opzionali.

Dopo aver inserito le informazioni necessarie, l'utente verrà rimandato all'interfaccia principale di *“Introjucer”*, in cui è possibile disegnare l'interfaccia grafica del proprio plug-in, per la quale verrà automaticamente generato il codice sorgente necessario alla visualizzazione di tutto ciò che sarà presente nell'interfaccia del plugin.



Schermata principale di *“Introjucer”*. Oltre all'editor grafico è possibile anche scrivere codice sorgente e gestire tutti i file sorgenti ed header del progetto.

Oltre al codice relativo all'interfaccia grafica (il plug-in editor), verrà generata parte del codice sorgente relativo al processore interno al plug-in (il plug-in processor) in cui saranno presenti le funzioni che l'utente dovrà poi modificare, così che il plug-in possa compiere le azioni a cui è destinato. L'utente potrà poi scrivere delle classi e dei sorgenti relativi ad ulteriori funzionalità del plug-in richiamabili poi all'interno del plug-in processor.

E' infine interessante notare come JUCE permetta di associare più plug-in editor ad un unico plug-in processor, e questo ha reso possibile la realizzazione delle due interfacce grafiche di cui si è accennato al capitolo precedente.

2.2.2 – Apple Xcode

Apple Xcode è un ambiente di sviluppo creato da Apple Inc. che permette di scrivere e compilare codice sorgente in più linguaggi di programmazione. Con questo IDE è possibile realizzare applicazioni funzionanti su dispositivi e sistemi operativi Apple. Fornisce inoltre strumenti per il test di applicazioni funzionanti su dispositivi mobili come iPhone, iPad, iWatch e iPod.

Oltre ai linguaggi di programmazione più comuni, con Apple Xcode è possibile programmare in linguaggio Swift, realizzato da Apple Inc., con il quale sono scritte la maggior parte delle applicazioni per i dispositivi Apple.

2.3 – Tecnologie di Digital Signal Processing (DSP): filtraggio del segnale audio per la divisione dello spettro in bande di frequenza

Alla base del software in esame vi sono alcune tecnologie di Elaborazione Digitale del Segnale (DSP: Digital Signal Processing): la compressione del segnale audio, di cui si è discusso nel capitolo precedente, ed il filtraggio del segnale audio per poter suddividere lo spettro del segnale audio in quattro bande, così da poter applicare un compressore di dinamica ad ogni banda di frequenza.

Per suddividere lo spettro del segnale in quattro bande di frequenza sono stati usati quattro filtri: un filtro passa-basso per ottenere la banda relativa alle frequenze basse; due filtri passa-banda per ottenere le bande relative alle frequenze medio-basse e alle frequenze medio-alte; infine è stato utilizzato un filtro passa-alto per ottenere la banda relativa alle frequenze alte.

In questo paragrafo si parlerà quindi del filtraggio del segnale audio partendo dai filtri ideali fino a giungere ai filtri di Butterworth, utilizzati per filtrare lo spettro del segnale audio, accennando poi a come si sono implementati nella realizzazione del software (l'implementazione verrà descritta più dettagliatamente nel prossimo capitolo).

2.3.1 – Filtri ideali [3] [4]

I filtri ideali sono una classe di sistemi lineari tempo-invarianti (LTI), sistemi le cui caratteristiche non cambiano nel tempo e le cui risposte a una combinazione lineare di segnali sono la combinazione lineare delle risposte dei singoli segnali. Prendendo in considerazione la risposta in frequenza, si osserva che lo spettro dell'uscita è il prodotto dello spettro dell'ingresso per lo spettro della risposta all'impulso, detta funzione di trasferimento. I sistemi LTI esibiscono quindi la capacità di filtrare componenti in frequenza, e per questo vengono chiamati anche filtri lineari.

I filtri ideali sono filtri caratterizzanti da funzioni di trasferimento a modulo costante in banda passante e nullo in banda proibita e aventi fase lineare. Tali filtri non sono causali, cioè non hanno

risposta nulla per valori negativi di t (tempo), e quindi possono solo essere approssimati da filtri fisicamente realizzabili.

I filtri ideali sono sistemi che annullano le componenti armoniche di un segnale in determinati intervalli di frequenza e che si comportano come sistemi senza distorsione sulle frequenze rimanenti. I sistemi senza distorsione sono sistemi che riproducono in uscita la stessa forma del segnale in ingresso, a meno di un eventuale fattore amplificativo e di un eventuale ritardo temporale. Un sistema senza distorsione può essere descritto dalla seguente trasformazione:

$$g(t) = Af(t - t_0)$$

Passando alle trasformate di Fourier e applicando la proprietà di traslazione temporale, si ha:

$$G(\nu) = Ae^{-2\pi i \nu t_0} F(\nu)$$

La funzione di trasferimento del sistema è quindi data da:

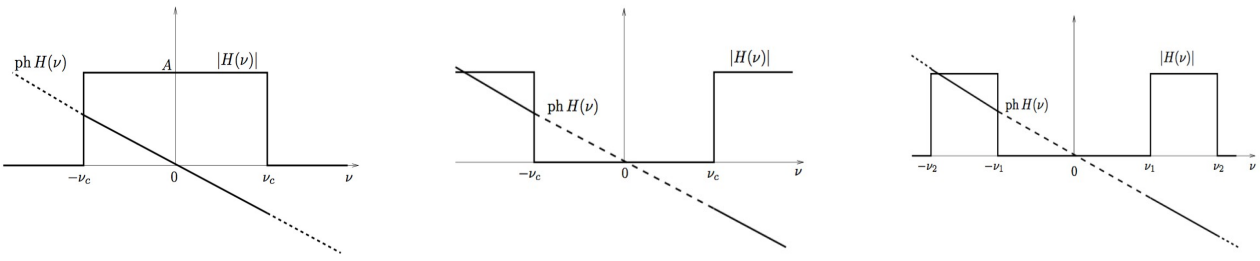
$$H(\nu) = \frac{G(\nu)}{F(\nu)} = Ae^{-2\pi i \nu t_0}$$

Il modulo della funzione di trasferimento è costante, vale cioè che $|H(\nu)| = A$, mentre la fase è lineare nella frequenza ν, cioè $\text{ph}H(\nu) = -2\pi\nu t_0$.

Un esempio di filtro ideale è il filtro *passa-basso*, che riproduce in uscita con guadagno A e fase lineare le componenti con frequenza non superiore a una cera frequenza ν_c detta frequenza di taglio, ed elimina quelle con frequenza superiore a questa soglia. Tale filtro ha la seguente funzione di trasferimento:

$$H(\nu) = Ae^{-2\pi i \nu t_0} rect(\frac{\nu}{\nu_c})$$

Un filtro *passa-alto*, viceversa, elimina le componenti in frequenza basse e passa quelle alte (superiori alla frequenza di taglio); il filtro *passa-banda*, infine, passa una banda (o intervallo) di componenti in frequenza ($\nu_a < \nu < \nu_b$) ed elimina quelle inferiori o superiori ad essa.



Risposta in frequenza di filtri passa-basso passa-alto e passa-banda ideali [3]

Le bande evidenziate nelle immagini sono due: la banda che interessa preservare effettivamente (*banda passante*) e la banda nella quale si richiede l'eleminazione (*banda proibita*).

2.3.2 – Caratteristiche di filtri analogici e famiglie di filtri causali [3] [4]

Nel sotto paragrafo precedente si è discusso dei filtri ideali. I filtri ideali sono caratterizzati da funzioni di trasferimento a modulo costante in banda passante, nullo in banda proibita e fase lineare. Come si è detto in precedenza, tali filtri non sono causali e quindi possono solo essere approssimati da filtri fisicamente realizzabili. Il problema della realizzazione di filtri per una data applicazione richiede almeno tre passi per la soluzione:

1. l'individuazione delle specifiche del filtro data la particolare applicazione;
2. la determinazione della funzione di trasferimento di un filtro soddisfacente le specifiche individuate;
3. la realizzazione fisica di un sistema la cui funzione di trasferimento coincide con quella determinata.

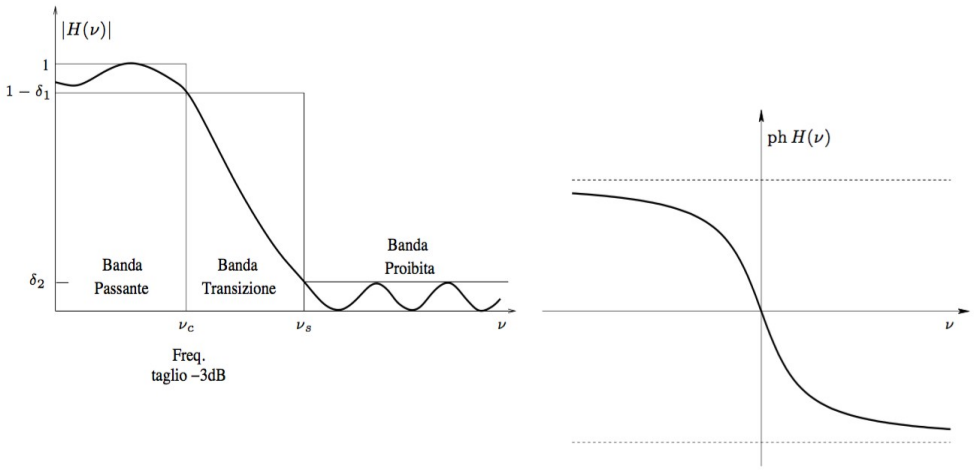
In questo sotto paragrafo, per descrivere le caratteristiche dei filtri analogici, si farà riferimento ai filtri passa-basso, in quanto le considerazioni che seguono possono essere estese alle altre tipologie di filtri.

Dalla funzione di trasferimento di un filtro passa-basso è possibile ottenere la sua risposta all'impulso tramite l'anti-trasformata di Fourier. Si avrà quindi

$$h(t) = \mathscr{F}^{-1}\left\{\mathrm{rect}(\frac{\nu}{2\nu_c})\right\} = 2\nu_c\,\mathrm{sinc}(2\nu_c t)$$

Si osserva che h(t) è in generale diversa da zero per t<0 quindi il filtro lineare è un sistema LTI non causale. Quindi tale filtro non può essere realizzato da un sistema fisico che verifichi il principio di causalità.

Indicando con H(ν) la funzione di trasferimento di un eventuale filtro passa-basso realizzabile, essa è completamente specificata dal suo modulo |H(ν)| e dalla sua fase phH(ν). Nella figura seguente vengono illustrate le forme tipiche di modulo e fase di un filtro passa-basso realizzabile.



Modulo e fase di un filtro passa-basso realizzabile [3]

Rispetto ad un filtro ideale si possono notare le seguenti differenze:

1) $|H(\nu)|$ non è costante nella banda passante e non è identicamente nullo nella banda proibita; si possono inoltre rilevare delle oscillazioni di ampiezza non trascurabile sia in banda passante sia in banda proibita. Un parametro importante è l'ampiezza della massima attenuazione in banda proibita δ_2, il cui logaritmo $-20\log_{10}\delta_2$ dB è detto *attenuazione*.
2) La banda passante e la banda proibita non confinano, ma sono separate da una banda detta *banda di transizione*. Tale banda inizia dalla frequenza di taglio ν_c e termina alla frequenza di stop ν_s; la dimensione della banda di transizione è data da ν_s-ν_c.
3) La fase $\text{ph}H(\nu)$ non risulta essere lineare.

Con frequenza di taglio si è soliti intendere la frequenza per la quale si ha un guadagno del 50% di quello in banda passante. Quindi se $|H(\nu)| \approx 1$ in banda passante, ν_c è la frequenza per cui

$$H^2(\nu_c) = \frac{1}{2}$$

La frequenza ν_c è detta anche *frequenza di taglio a 3 dB* poiché $10\log(1) \approx -3$ dB.

Usualmente la massima oscillazione δ_2 permessa in banda proibita viene espressa in dB attraverso l'attenuazione R_p, dove

$$R_p = -10\log(\delta_2^2)$$

La banda di proibita è data dall'insieme delle frequenze per le quali il quali il guadagno è inferiore ad un'opportuna soglia di attenuazione. Indicando con ν_s la frequenza di stop, le frequenze tra ν_s e ν_c costituiscono la banda di transizione.

Poiché la fase della funzione di trasferimento non è lineare, si generano ritardi differenti per le componenti a diversa frequenza, provocando così una distorsione complessiva del segnale. Per certe applicazioni tali distorsioni devono essere il più possibile eliminate, mentre in altre applicazioni la non linearità può essere usata per dar luogo ad effetti speciali.

Per garantire il principio di causalità, si è detto che i filtri ideali possono solo essere approssimati da filtri realizzabili fisicamente. Le principali famiglie di filtri causali sono i filtri di Butterworth, di Chebyshev, di Cauer (o ellittici) e di Bessel. Nella tabella seguente è mostrata una grossolana valutazione comparativa della bontà di questi filtri a parità di ordine. Si analizzerà più in dettaglio la classe dei filtri di Butterworth, in quanto utilizzati per filtrare il segnale audio nel processore di dinamica in esame.

Filtro	**Accuratezza approssimazione guadagno**	**Linearità fase**
Butterworth	media	media
Chebyshev	buona	cattiva
Ellittico	ottima	pessima
Bessel	cattiva	buona

2.3.3 – Filtri di Butterworth [3] [4]

I filtri di Butterworth costituiscono una famiglia di filtri che soddisfa bene i requisiti sul guadagno in banda passante e meno bene in banda di transizione. Non esibiscono una fase lineare in banda passante, però la loro approssimazione non è troppo cattiva e sono tra i filtri elettronici più semplici da realizzare.

Un filtro di Butterworth è caratterizzato da due parametri: l'ordine N e la frequenza di taglio ν_c. La forma generale del modulo della funzione di trasferimento di un filtro di Butterworth di ordine N e frequenza di taglio ν_c è:

$$\frac{1}{|B_N(i\frac{\nu}{\nu_c})|} = \frac{1}{\sqrt{1+\left(\frac{\nu}{\nu_c}\right)^{2N}}}$$

$B_N(s)$ è un opportuno polinomio detto N-esimo polinomio di Butterworth, mentre la scalatura ν/ν_c rispetto a ν denota la frequenza normalizzata rispetto alla frequenza di taglio.

Se N è pari, il polinomio $B_N(s)$ è dato dal prodotto di N/2 polinomi di secondo grado del tipo $s^2 + bs + s$ con $b > 0$, mentre se N è dispari allora è presente anche il fattore $s + 1$. Di seguito è indicata la fattorizzazione dei primi otto polinomi di Butterworth. Un'interessante proprietà è che tutti i polinomi hanno le radici che giacciono sul cerchio di raggio unitario. [4]

$B_1(s) = s + 1$

$B_2(s) = s^2 + 1{,}414s + 1$

$B_3(s) = (s + 1)(s^2 + s + 1)$

$B_4(s) = (s^2 + 0{,}765s + 1)(s^2 + 1{,}848s + 1)$

$B_5(s) = (s + 1)(s^2 + 0{,}618s + 1)(s^2 + 1{,}618s + 1)$

$B_6(s) = (s^2 + 0{,}518s + 1)(s^2 + 1{,}414s + 1)(s^2 + 1{,}932s + 1)$

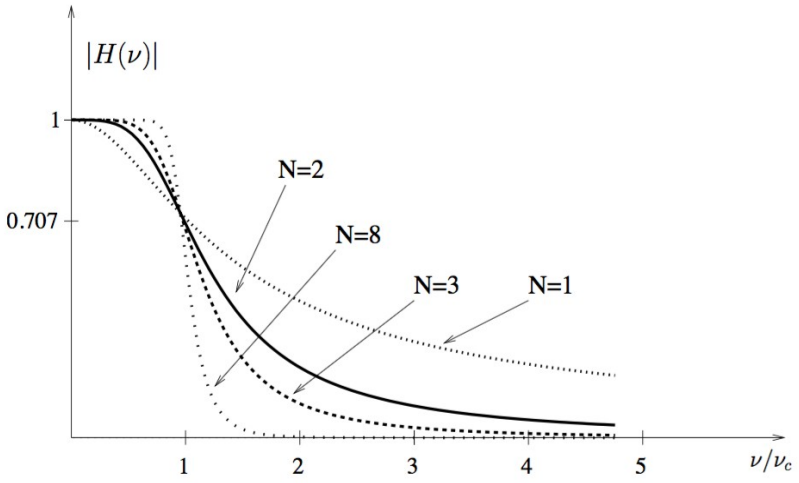
$B_7(s) = (s + 1)(s^2 + 0{,}445s + 1)(s^2 + 1{,}247s + 1)(s^2 + 1{,}802s + 1)$

$B_8(s) = (s^2 + 0{,}390s + 1)(s^2 + 1{,}111s + 1)(s^2 + 1{,}663s + 1)(s^2 + 1{,}962s + 1)$

Nella figura a pagina seguente è riportata la risposta in frequenza di alcuni filtri di Butterworth.

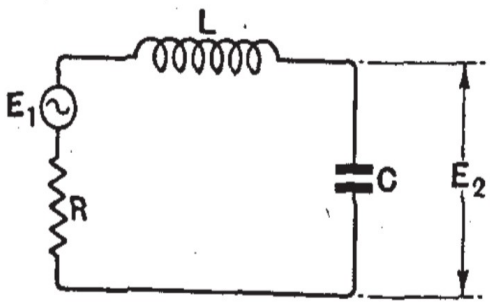
Dalla figura si possono effettuare le seguenti osservazioni:

- La frequenza di taglio a 3dB è ν_c, indipendentemente dall'ordine N del filtro.
- L'attenuazione nella banda proibita dipende da N in modo critico; risulta infatti $R_p \approx 20N \log(\frac{\nu_s}{\nu_c})$.
- Non sono presenti oscillazioni né in banda passante né in banda proibita: il filtro di Butterworth ha la caratteristica di essere piatto in banda passante.

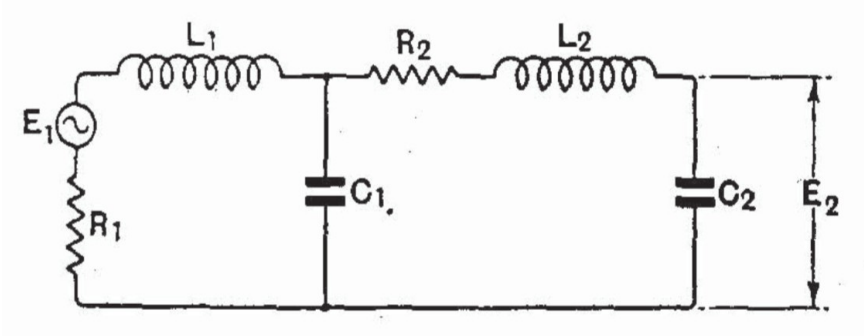


Risposta in frequenza di alcuni filtri di Butterworth di diverso ordine [3]

Nell'articolo "On the Theory of Filter Amplifiers" (pubblicato su Wireless Engineer vol.7 nell'ottobre del 1930) Butterworth descrisse la sua tipologia di filtro. Nella sezione relativa allo schema generale scrisse che vi era "perfetta libertà riguardo alle costanti elettriche degli elementi e che essi sono stati scelti con l'obbiettivo di ottenere la più vicina approssimazione della condizione di sensibilità uniforme in banda passante e alla condizione di sensibilità nulla nella banda proibita."[5] Le figure seguenti rappresentano gli schemi dei filtri di Butterworth a uno e a due elementi presenti nell'articolo:



Filtro di Butterworth a un elemento. [5]



Filtro di Butterworth a due elementi [5]

Ogni elemento è costituito da un induttore L, un condensatore C e una resistenza R. E_1 ed E_2 rappresentano rispettivamente la tensione in ingresso e la tensione in uscita.

2.3.4 – Accenni sull'implementazione dei filtri di Butterworth per l'applicativo in esame

Per realizzare la divisione in quattro bande di frequenza dello spettro del segnale audio, vi è stata la necessità di implementare un filtro passa-basso, un filtro passa-alto e due filtri passa-banda.

Si è scelto di utilizzare filtri di Butterworth del quarto ordine. La scelta sui filtri di Butterworth è stata determinata da una serie di test su differenti filtri, e i risultati più gradevoli sono stati ottenuti tramite l'utilizzo di questi filtri. La scelta dell'ordine del filtro è stata dettata dal fatto che non vi fosse un eccessivo carico computazionale sulla CPU. Inoltre il livello di attenuazione in banda proibita era tale da evitare eccessive sovrapposizioni di segnale tra le varie bande di frequenza presenti.

Per implementare tali filtri è stata utilizzata una libreria costituita da un insieme di classi in C++ per l'elaborazione digitale del segnale realizzata da Vincent Falco e resa disponibile nel 2009 sotto licenza MIT. Questa libreria è costituita da classi che permettono la realizzazione di filtri di ordine arbitrario e con varie proprietà. I filtri che possono essere realizzati sono filtri di Butterworth, di Chebyshev, di Bessel, filtri ellittici, filtri Legendre e filtri RBJ Biquad tratti da "Audio EQ Cookbook".

La libreria si focalizza su filtri IIR (a risposta infinita all'impulso), permette la realizzazione di varie tipologie di filtri (passa-basso, passa-alto, passa-banda, elimina-banda) e permette, per alcuni filtri, l'implementazione di low high e band shelf.

Tale libreria non richiede dipendenze da librerie esterne in quanto è stata scritta utilizzando la libreria standard del C++.[6]

Nel capitolo successivo si discuterà sull'implementazione dei filtri di Butterworth indicati all'inizio del sotto-paragrafo utilizzando la libreria appena descritta.

Capitolo 3:
Descrizione delle funzionalità principali del processore di dinamica multi-banda in esame e relativo processo implementativo

3.1 - Funzionalità principali del software in esame

Poiché l'applicativo in esame è un processore di dinamica multi-banda con impostazione automatica dei parametri, le sue principali funzionalità sono tre: suddivisione dello spettro del segnale audio in ingresso in quattro bande di frequenza; applicazione di un processore di dinamica, nel caso particolare di un compressore di dinamica, ad ognuna delle bande di frequenza e successiva ricomposizione del segnale processato; impostazione automatica dei parametri dei processori e scelta automatica delle bande di frequenza su cui operare sulla base dello strumento musicale selezionato dall'utente finale.

Ogni azione, ad eccezione dell'impostazione dei parametri, viene effettuata un campione alla volta per tutto il segnale su cui il software opera: i parametri assegnati in decibel (threshold e make up gain) vengono convertiti in valori di ampiezza, mentre i parametri temporali (tempo di attacco e rilascio, che sono in millisecondi) vengono convertiti in campioni, moltiplicandoli per un millesimo di frequenza di campionamento. Il valore del rapporto di compressione viene poi convertito nel suo reciproco poiché il software ha necessità di sapere di quanto il segnale in uscita dovrà essere ridotto rispetto al segnale in ingresso. Quindi tali parametri, se non indicato diversamente, nel proseguimento del capitolo saranno considerati già convertiti.

3.1.1 – Descrizione delle funzionalità principali: suddivisione dello spettro del segnale audio in quattro bande di frequenza

Per suddividere lo spettro del segnale audio in ingresso in quattro bande di frequenza l'applicativo sfrutta quattro filtri di Butterworth del quarto ordine: un filtro passa-basso, due filtri passa-banda e un filtro passa-alto.

I parametri considerati per l'inizializzazione dei filtri sono molteplici:

- Filtro passa-basso: il software prende in considerazione la frequenza di taglio, che varia con il variare della prima frequenza di crossover. Considera poi la frequenza di campionamento, la quale viene inviata direttamente dalla DAW al plug-in, e l'ordine del filtro, impostato a 4 all'interno del plug-in stesso.

- Filtri passa-banda: oltre all'ordine dei filtri e alla frequenza di campionamento, il software, per inizializzare i due filtri passa-banda, calcola il centro e l'ampiezza delle bande in esame. Il centro della banda considerata per il primo filtro è dato dalla media tra la prima e la seconda frequenza di crossover, mentre l'ampiezza è data dalla distanza tra queste due frequenze; il centro della banda considerata per il secondo filtro è dato invece dalla media tra la seconda e la terza frequenza di crossover, e la loro distanza determina l'ampiezza della banda.
- Filtro passa-alto: come per gli altri tre filtri, per inizializzare il filtro passa-alto il software considera la frequenza di campionamento e l'ordine del filtro; considera inoltre, come per il filtro passa-basso, la frequenza di taglio, che varia al variare della terza frequenza di crossover.

3.1.2 – Descrizione delle funzionalità principali: applicazione di un processore di dinamica ad ognuna delle bande di frequenza.

Dopo che lo spettro del segnale in ingresso è stato suddiviso in quattro bande, il software applica quattro processori di dinamica, uno per ognuna delle bande presenti. Il processori applicati sono dei compressori di dinamica.

I quattro compressori utilizzati nell'applicativo presentano le stesse caratteristiche: sono presenti le stesse tipologie di parametri (threshold, rapporto di compressione, tempo di attacco, tempo di rilascio, make up gain), possono sfruttare due tipologie di funzioni di trasferimento e possono sfruttare due tipologie di rilevatori di inviluppo. I parametri possono essere impostati differentemente nei vari compressori, mentre la funzione di trasferimento ed il rilevatore di inviluppo una volta selezionati sono identici per tutti i compressori.

Nel primo capitolo si è descritto come agisce un compressore di dinamica nel caso generale, e quindi è ora possibile descrivere più dettagliatamente come agisce un compressore all'interno del software in esame, applicato ad una delle bande di frequenza. Il comportamento sarà il medesimo anche per i tre compressori rimanenti, con risultati che variano in base ai valori assegnati ai loro parametri.

Una volta che sono stati impostati i parametri, se non è bypassato dal software (o eventualmente dall'utente nel caso della versione ad interfaccia a più parametri), il compressore di dinamica agisce in questo modo per il segnale filtrato dalla banda in esame:

- calcola l'inviluppo del segnale utilizzando la tipologia di rilevatore selezionato;
- tramite la funzione di trasferimento selezionata, il compressore determina la riduzione di guadagno (gain reduction) sulla base dell'inviluppo rilevato;
- il valore di gain reduction ricavato dalla funzione di trasferimento viene passato al filtro di

attacco e rilascio, il quale determina un fattore dipendente dal tempo di attacco o dal tempo di rilascio sulla base del confronto tra il valore ottenuto dalla funzione di trasferimento e il gain reduction del campione precedente, moltiplicando il gain reduction del campione precedente per questo fattore;

- infine il valore del segnale in ingresso viene moltiplicato per il valore di gain reduction ottenuto nel punto precedente e per il valore di make up gain assegnato al compressore.

Se il compressore di dinamica fosse stato bypassato, il segnale in ingresso verrebbe solamente moltiplicato per il valore di make up gain assegnato, e le tre azioni per il calcolo del gain reduction non verrebbero compiute.

L'inviluppo del segnale può essere calcolato tramite un rilevatore di RMS o tramite un rilevatore di picco: se è stato selezionato il rilevatore di RMS allora verrà calcolato il valore efficace del segnale tramite la formula indicata nel primo capitolo; se è stato selezionato il rilevatore di picco verrà calcolato il livello massimo all'interno di una finestra mobile del segnale, come indicato nel primo capitolo. Nel seguito del capitolo sarà illustrata l'implementazione di entrambi questi rilevatori.

Nel software in esame sono presenti, come già spiegato, due tipologie di funzioni di trasferimento per il calcolo del gain reduction, una di tipo esponenziale e una che utilizza funzioni lineari:

- la prima, eleva il rapporto tra threshold ed inviluppo rilevato al valore del rapporto di compressione sottratto a 1;
- la seconda invece calcola il prodotto tra il valore dato da rapporto di compressione meno uno e il minimo tra l'inviluppo rilevato e la threshold. Il risultato di questo prodotto viene infine sommato al prodotto tra inviluppo e rapporto di compressione.

Il gain reduction così calcolato dalla funzione di trasferimento viene poi passato al filtro di attacco e rilascio. Questo filtro confronta il valore ottenuto dalla funzione di trasferimento con il valore di gain reduction ricavato dal campione precedente:

- se il valore attuale è minore del gain reduction del sample precedente, significa che la riduzione di range del segnale sta diminuendo e quindi il filtro calcola il fattore con cui moltiplicare il gain reduction precedente utilizzando il valore del tempo di attacco: il fattore viene calcolato elevando il rapporto tra valore attuale e il precedente gain reduction al reciproco del tempo di attacco.
- Se il valore attuale è invece maggiore del gain reduction del sample precedente, significa che la riduzione di range dinamico del segnale sta aumentando e quindi il fattore con cui moltiplicare il precedente gain reduction viene calcolato utilizzando il valore del tempo di rilascio: il fattore viene calcolato allo stesso modo del caso precedente, con la differenza che ad esponente vi è il reciproco del tempo di rilascio.

Dopo che il fattore è stato calcolato, lo si moltiplica per il valore di gain reduction del sample precedente.
Infine, come detto prima, il livello del segnale in ingresso del sample attuale viene moltiplicato per il valore di gain reduction ottenuto dal filtro di attacco e rilascio e per il livello di make up gain assegnato.

3.1.3 – Descrizione delle funzionalità principali: impostazione automatica dei parametri

L'utente finale ha la possibilità di selezionare, tramite il menù a tendina presente sull'interfaccia grafica del software, lo strumento musicale sul quale sta lavorando.
La scelta dello strumento aziona l'impostazione di tutti i parametri relativi al processore, ad esclusione del livello di ingresso e di uscita globale del segnale audio: saranno assegnati automaticamente i valori relativi alle frequenze di crossover e i valori relativi a make up gain, threshold, rapporto di compressione e tempo di attacco e rilascio di ogni compressore di dinamica; verranno impostati automaticamente anche il rilevatore di inviluppo e la funzione di trasferimento e i compressori non necessari per lo strumento da processare verranno bypassati automaticamente.
Questi parametri, nell'interfaccia semplificata, verranno impostati internamente al software e l'utente avrà la possibilità di controllare solo i make up gain delle bande, oltre al livello di ingresso e di uscita del segnale, e avrà la possibilità di ascoltare ogni singola banda isolata dalle altre cliccando sul pulsante “solo” presente su ognuna di esse. Nell'interfaccia a più parametri verranno visualizzati i nuovi valori per ognuno dei parametri e l'utente avrà la possibilità di modificarli, avendo così pieno controllo su tutto il processore.
L'automatizzazione si verifica in quanto ad ogni strumento è assegnata una serie di valori relativi a tutti i parametri, e la scelta dello strumento permetterà al software di assegnare i parametri necessari per poter processare adeguatamente la traccia audio su cui si sta lavorando.
Nel paragrafo successivo sarà descritta l'implementazione dell'assegnazione automatica dei parametri relativa a uno strumento musicale.

3.2 – Implementazione dell'applicativo in esame

Per implementare il software in esame, come descritto nel capitolo precedente, sono stati utilizzati il framework JUCE e l'ambiente di sviluppo Apple Xcode. JUCE è stato utilizzato per la configurazione del progetto relativo al software e per la realizzazione dell'interfaccia grafica. JUCE ha generato poi il codice sorgente in linguaggio C++ e i file necessari per poter proseguire con l'implementazione del software. Xcode è stato invece utilizzato per la scrittura del codice sorgente, in linguaggio C++, relativo all'implementazione di tutte le funzionalità dell'applicativo in esame.

Le fasi di implementazione del software sono state le seguenti:

- implementazione dell'interfaccia grafica per la versione a più parametri;
- implementazione dei parametri del processore interno al software, la quale comprende il settaggio dei parametri relativi ai quattro compressori e comprende l'impostazione relativa ai parametri utilizzati per i filtri;
- realizzazione della classe compressore, implementando poi le relative sotto-classi;
- realizzazione della classe relativa alla divisione in bande dello spettro, implementando poi le tre sotto-classi relative alle tre tipologie di filtri
- realizzazione di una classe che contenesse la classe compressore e la classe per la divisione in bande
- assegnazione dei parametri relativi ad ognuno degli strumenti musicali indicati nel plug-in
- collegamento delle varie classi con il processore principale
- implementazione dell'interfaccia semplificata.

Si può riassumere la struttura del software con il seguente diagramma:

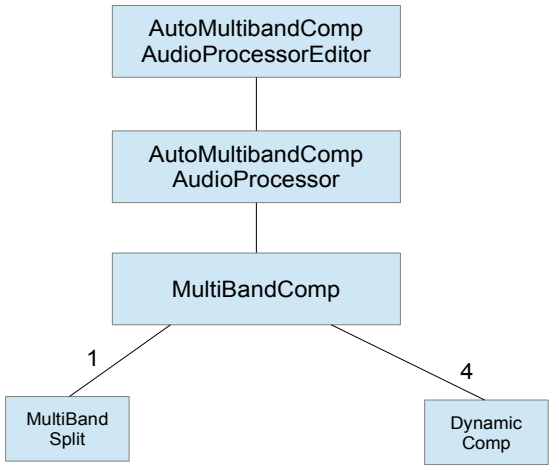


Diagramma rappresentante la macro-struttura del software in esame

Dal diagramma è possibile osservare come la classe *MultiBandComp* richiami dentro di se un'istanza di *MultiBandSplit* e quattro istanze di *DynamicComp*. Questo perché al software è necessario un unico oggetto che compia la funzione di suddividere in bande lo spettro del segnale audio, ma per ogni banda è necessario un compressore di dinamica, ed essendoci quattro bande di frequenza sono necessari quattro oggetti che compiano la funzione di compressori di dinamica.
La classe denominata nello schema *AutoMultibandCompAudioProcessorEditor* è la classe che riguarda l'interfaccia grafica, la quale per prima riceve i parametri in ingresso: livello di input, livello di output, strumento musicale selezionato e make up gain ed eventuale stato di “solo” di ogni

banda per la versione ad interfaccia semplificata; i parametri precedenti più tutti i parametri relativi ai compressori di dinamica nella versione ad interfaccia con più parametri.

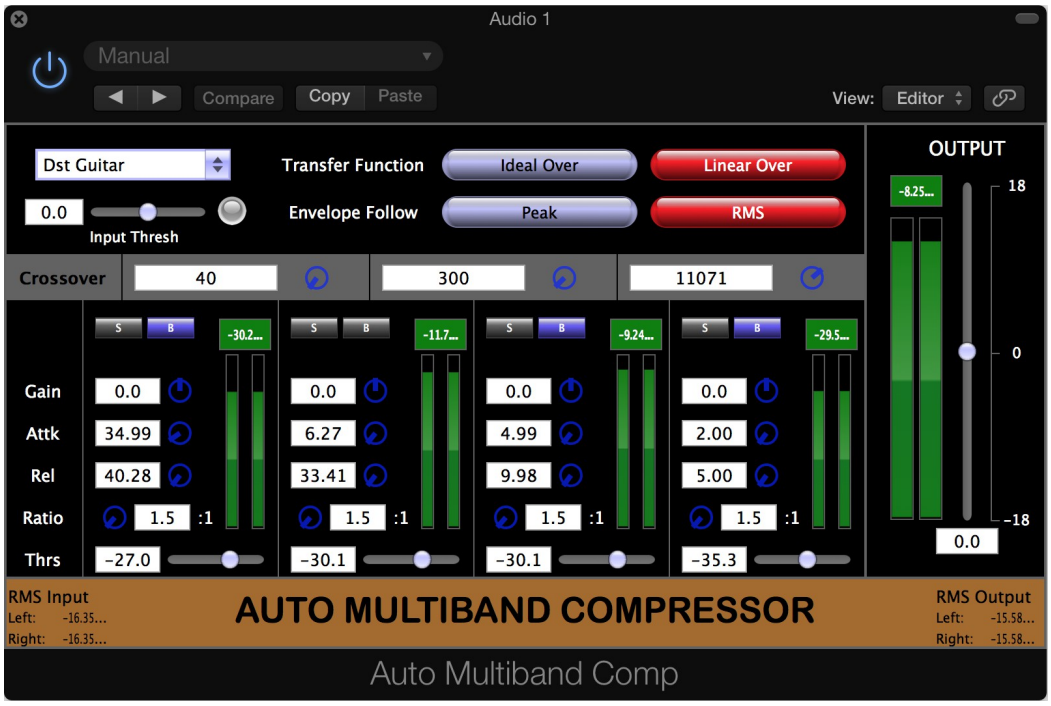
I parametri verranno poi inviati alla classe *AutoMultibandCompAudioProcessor*, la quale è il processore interno al software. Questa classe invia i parametri ricevuti alla classe *MultiBandComp*, la quale li invierà rispettivamente all'oggetto di tipo *MultiBandSplit* e ai quattro oggetti di tipo *DynamicComp*.

Le fasi di implementazione che verranno analizzate in seguito non seguiranno l'ordine effettivo della realizzazione, ma seguiranno un approccio top-down, partendo cioè dalla realizzazione delle due interfacce grafiche, arrivando fino alla descrizione della classe per la suddivisione in bande e della classe relativa al compressore di dinamica.

3.2.1 – Implementazione del software in esame: le interfacce grafiche

Le due interfacce grafiche sono state realizzate tramite il framework JUCE, il quale fornisce un editor grafico con cui è possibile inserire tutti gli elementi che costituiscono l'interfaccia dell'applicativo.

La prima interfaccia ad essere stata realizzata è stata l'interfaccia a più parametri. Sono stati inseriti tutti i cursori necessari ad inserire i valori, i pulsanti per bypassare i compressori e per mettere in "solo" le bande, i pulsanti per la scelta del rilevatore di inviluppo e per la scelta della funzione di trasferimento, il menu a tendina per la scelta dello strumento e infine sono stati inseriti i vari indicatori per il livello di uscita di ogni banda, per il livello di uscita globale e per il livello di RMS in ingresso e in uscita. Questa interfaccia permette di avere il controllo di tutti i parametri necessari ad un compressore di dinamica multi-banda.



Interfaccia a più parametri dell'applicativo

Successivamente alla realizzazione di tutte le funzionalità del software (suddivisione in bande, compressori di dinamica e automatizzazione dei parametri), è stata realizzata la seconda interfaccia, la quale consiste in una versione semplificata della precedente. Infatti è limitata a pochi controlli per l'utente finale: sono stati inseriti quattro cursori per regolare il make up gain delle quattro bande, un cursore per regolare il livello del segnale in ingresso, uno per il livello del segnale di uscita e quattro pulsanti per mettere in stato di "solo" le singole bande, permettendo così all'utente di isolare una determinata banda di frequenza ed ascoltare il suono riprodotto da quella banda. Sono presenti inoltre gli indicatori di livello per le singole bande, gli indicatori di livello per il segnale in uscita e gli indicatori di RMS in input e in output.

Interfaccia semplificata dell'applicativo

Il codice sorgente relativo alla visualizzazione degli elementi è stato generato automaticamente da JUCE. Per quanto riguarda le interfacce sono state scritte le istruzioni relative ai vari componenti che ne fanno parte, cioè le varie chiamate a *setParameter()* e a *getParameter()* del processore interno.

E' stata scritta una funzione relativa all'aggiornamento dell'interfaccia grafica (*timerCallback()*), la quale compie le proprie azioni dopo lo scadere di un timer pre-impostato a 200ms oppure successivamente all'interazione dell'utente con uno dei componenti dell'interfaccia. Ogni volta che si verificano queste condizioni viene richiamata una funzione del processore che imposta a *true* una variabile di controllo, e quando questa variabile è impostata a *true* la funzione *timerCallback()* aggiorna lo stato di ognuno dei componenti dell'interfaccia.

La funzione di *timerCallback()* è presente su entrambe le interfacce.

Per quanto riguarda l'impostazione automatica dei parametri, il codice è identico per entrambe le interfacce. Si tratta infatti di un costrutto *switch – case*: il programma controlla l'indice relativo allo strumento selezionato e in base al valore dell'indice vengono impostati tutti i parametri in modo da far lavorare il processore di dinamica multi-banda con le regolazioni necessarie. I parametri vengono impostati richiamando per ognuno la funzione *setParameter()* del processore interno. Dopodiché viene richiamata la funzione che aggiorna la variabile di controllo permettendo così l'aggiornamento dell'interfaccia.

Di seguito viene presentato il codice relativo ad un caso dello switch, quello relativo alla chitarra distorta.

```
1107 void AutoMultibandCompAudioProcessorEditor::comboBoxChanged (ComboBox* comboBoxThatHasChanged)
1108 {
1109     //[UsercomboBoxChanged_Pre]
1110     AutoMultibandCompAudioProcessor* ourProcessor = getProcessor();
1111     //[/UsercomboBoxChanged_Pre]
1112 
1113     if (comboBoxThatHasChanged == InstrBox)
1114     {
1115         //[UserComboBoxCode_InstrBox] -- add your combo box handling code here..
1116 
1117         switch (InstrBox -> getSelectedItemIndex())
1118         {
1119             case 0:
1120                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::InstrBox, (float)InstrBox -> getSelectedId());
1121                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::BypassBand1, 1.0f);
1122                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::BypassBand3, 1.0f);
1123                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::BypassBand4, 1.0f);
1124                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::RMSBtn, 1.0f);
1125                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::LinearOverBtn, 1.0f);
1126                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::MakeupGainBand2, 0.0f);
1127                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::AttkTimeBand2, 6.27f);
1128                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::RelTimeBand2, 33.42f);
1129                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::RatioBand2, 1.5f);
1130                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::ThreshBand2, -30.1f);
1131                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::CrossFreq1, 40.0f);
1132                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::CrossFreq2, 300.0f);
1133                 ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::CrossFreq3, 11071.0f);
1134 
1135                 if (ourProcessor -> getParameter(AutoMultibandCompAudioProcessor::BypassBand2) != 0.0f)
1136                     ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::BypassBand2, 0.0f);
1137 
1138                 if (ourProcessor -> getParameter(AutoMultibandCompAudioProcessor::SoloBand1) != 0.0f)
1139                     ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::SoloBand1, 0.0f);
1140 
1141                 if (ourProcessor -> getParameter(AutoMultibandCompAudioProcessor::SoloBand2) != 0.0f)
1142                     ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::SoloBand2, 0.0f);
1143 
1144                 if (ourProcessor -> getParameter(AutoMultibandCompAudioProcessor::SoloBand3) != 0.0f)
1145                     ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::SoloBand3, 0.0f);
1146 
1147                 if (ourProcessor -> getParameter(AutoMultibandCompAudioProcessor::SoloBand4) != 0.0f)
1148                     ourProcessor -> setParameter(AutoMultibandCompAudioProcessor::SoloBand4, 0.0f);
1149 
1150                 ourProcessor -> UIRequestUpdate();
1151                 break;
```

Codice relativo all'impostazione dei parametri dovuto alla scelta di uno strumento

Il codice in figura è contenuto all'interno della funzione *comboBoxChanged*. Viene dichiarato un puntatore ad *AutoMultibandCompAudioProcessor* inizializzato al valore ritornato dalla funzione *getProcessor()*, interna al processore, il quale permette di mettere in comunicazione l'interfaccia con il processore. La funzione *comboBoxChanged* riceve come parametro un puntatore alla combo box che viene modificata. La funzione controlla se la combo box modificata è *InstrBox*. In caso affermativo esegue un costrutto *switch-case* che controlla l'indice relativo allo strumento musicale selezionato. Nel caso in esame l'indice è 0, quindi lo strumento selezionato è la chitarra distorta. Il puntatore che mette in comunicazione interfaccia e processore punta alla funzione *setParameter*

relativa ad ogni parametro passando così al processore e ai vari parametri i valori assegnati. Dopodiché la funzione esegue un controllo per verificare se il compressore (o i compressori) necessario per processare lo strumento selezionato è bypassato o meno e in caso affermativo cambia il suo stato da bypassato a non bypassato assegnandogli il valore 0. Poi esegue quattro controlli per verificare se le bande sono in stato di "solo" e in caso affermativo cambia il loro stato a "non in solo" assegnando ad ognuno dei parametri di "solo" il valore 0.
Alla fine il puntatore ourProcessor punterà alla funzione *UIRequestUpdate()* interna al processore per impostare la variabile di controllo a 1 in modo tale da permettere alla funzione *timerCallback()* di aggiornare l'interfaccia grafica. Le stesse operazioni vengono svolte per gli altri strumenti musicali, cambiando solo i parametri da modificare.

3.2.2 – Implementazione del software in esame: il processore interno

Il processore interno al plug-in, *AutoMultibandCompAudioProcessor*, è il cuore di tutto il software. Infatti all'interno di esso vengono settati tutti i parametri, tramite la funzione *setParameter*, in cui vengono richiamati i setter della classe *MultiBandComp,* e, tramite la funzione *processBlock*, il processore richiama la funzione *ClockProcess* di *MultiBandComp*.
La funzione *setParameter* riceve in ingresso l'indice del parametro ed il valore da assegnare al parametro indicato dall'indice. Viene così assegnato un valore all'elemento collocato alla posizione indicata dall'indice nell'array *UserParams*. Dopodiché il valore modificato, se necessario per le operazioni di *MultiBandComp*, viene passato al relativo setter della classe.
Nel processore principale vi è anche una funzione chiamata *getParameter*, la quale riceve in ingresso un indice relativo ad ogni parametro e ritorna il valore contenuto nel vettore *UserParams* alla posizione indicata dall'indice.
La funzione *processBlock* è la funzione in cui il plug-in processa il segnale audio. Infatti al suo interno viene determinato il numero di canali della traccia audio, e in caso di audio mono o stereo la funzione agirà di conseguenza. Vengono dichiarati due puntatori a *float* inizializzati ad un puntatore a canale del buffer audio (canale destro e canale sinistro). In caso di audio mono entrambi vengono inizializzati allo stesso puntatore. I due puntatori conterranno così il segnale audio da processare.
All'interno della funzione *processBlock* viene eseguita la chiamata alla funzione *ClockProcess* di *MultiBandComp* alla quale viene passato il valore contenuto nei puntatori a float inizializzati in precedenza, alla posizione indicata dal numero del campione su cui si sta lavorando. Inoltre al metodo *ClockProcess* di *MultiBandComp* viene passato anche il valore 1, che sta a indicare che si sta operando su un unico campione.
All'interno di *processBlock* è stato eseguito il calcolo di RMS in ingresso e in uscita, e il calcolo dei livelli delle bande dopo la compressione e del segnale audio in uscita.

In figura viene presentata parte del codice relativo alla funzione *processBlock*. E' riportata la parte iniziale della funzione e il caso dell'audio in mono *(numOfChannel=1)*. L'unica differenza con il caso dell'audio in stereo è che il puntatore *RightData* è inizializzato a *buffer.getWritePointer(1)*.

```
void AutoMultibandCompAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
{
    if (isFirstRun) {
        sampleRate_HZ = getSampleRate();
        mMultiBandCompCtrl.setSampleRate(sampleRate_HZ);

        RMSInCheckLeft = new FastRMS(500*(sampleRate_HZ*0.001));
        RMSInCheckRight = new FastRMS(500*(sampleRate_HZ*0.001));

        RMSOutCheckLeft = new FastRMS(500*(sampleRate_HZ*0.001));
        RMSOutCheckRight = new FastRMS(500*(sampleRate_HZ*0.001));

        isFirstRun = false;
    }

    // In case we have more outputs than inputs, this code clears any output
    // channels that didn't contain input data, (because these aren't
    // guaranteed to be empty - they may contain garbage).
    // I've added this to avoid people getting screaming feedback
    // when they first compile the plugin, but obviously you don't need to
    // this code if your algorithm already fills all the output channels.
    for (int i = getNumInputChannels(); i < getNumOutputChannels(); ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    int numOfChannel = getNumInputChannels();

    if (numOfChannel == 1)      // audio mono
    {
        float *LeftData = buffer.getWritePointer(0);
        float *RightData = buffer.getWritePointer(0);

        int numSamplesMono = buffer.getNumSamples();

        for (int i = 0; i<numSamplesMono; ++i)
        {
            // Multiplicates audio with input gain
            LeftData[i] *= pow(10, UserParams[InputGain]/20);
            RightData[i] *= pow(10, UserParams[InputGain]/20);

            // Gets global RMS IN
            RMSInputLeft = 20*log10(RMSInCheckLeft -> ClockProcess(LeftData[i]));
            RMSInputRight = 20*log10(RMSInCheckRight -> ClockProcess(RightData[i]));

           // Calls MultiBandComp Clock Process
           mMultiBandCompCtrl.ClockProcess(&LeftData[i], &RightData[i], 1);

            // Gets output from every band
            Band1OutLeft = mMultiBandCompCtrl.getOutputBand1Left();
            Band1OutRight = mMultiBandCompCtrl.getOutputBand1Right();
            AvgBand1Out = (Band1OutLeft + Band1OutRight)/2;
            Band2OutLeft = mMultiBandCompCtrl.getOutputBand2Left();
            Band2OutRight = mMultiBandCompCtrl.getOutputBand2Right();
            AvgBand2Out = (Band2OutLeft + Band2OutRight)/2;
            Band3OutLeft = mMultiBandCompCtrl.getOutputBand3Left();
            Band3OutRight = mMultiBandCompCtrl.getOutputBand3Right();
            AvgBand3Out = (Band3OutLeft + Band3OutRight)/2;
            Band4OutLeft = mMultiBandCompCtrl.getOutputBand4Left();
            Band4OutRight = mMultiBandCompCtrl.getOutputBand4Right();
            AvgBand4Out = (Band4OutLeft + Band4OutRight)/2;

            // Get global RMS OUT in dB
            RMSOutputLeft = 20*log10(RMSOutCheckLeft -> ClockProcess(LeftData[i]));
            RMSOutputRight = 20*log10(RMSOutCheckRight -> ClockProcess(RightData[i]));

            // Get global RMS OUT in amplitude value in order to get output VU
            RMSOutputLeftAmp = RMSOutCheckLeft -> ClockProcess(LeftData[i]);
            RMSOutputRightAmp = RMSOutCheckRight -> ClockProcess(RightData[i]);

            // Get left out and average global output in db
            MainOutputLevelLeft = 10*log10((0.9f*(RMSOutputLeftAmp) + (1.0f-0.9f)*pow(LeftData[i],2)));
            MainOutputLevelRight = 10*log10((0.9f*(RMSOutputRightAmp) + (1.0f-0.9f)*pow(RightData[i],2)));
            AvgMainOutput = (MainOutputLevelLeft + MainOutputLevelRight)/2;
        }
    }
```

Porzione di codice relativo alla funzione *processBlock*

Nella prima parte della funzione il programma controlla la variabile booleana *isFirstRun*, inizializzata a *true* nel costruttore di *AutoMultibandCompAudioProcessorEditor*. Se è vera significa che il plug-in è stato avviato per la prima volta. La variabile s*ampleRate_HZ* viene inizializzata al valore ritornato da *getSampleRate()*, pari al valore della frequenza di campionamento che la DAW passa al plug-in. Viene richiamato il metodo *setSampleRate* dell'oggetto *mMultiBandCompCtrl*,

oggetto della classe *MultiBandComp*. Vengono re-inizializzati gli oggetti relativi al rilevamento del valore di RMS in input e in output. Infine alla variabile *isFirstRun* viene assegnato il valore *false.*
Non appena terminano le azioni dovute al primo utilizzo del plug-in, vi è un'istruzione, costituita da un ciclo *for*, generata da JUCE, la quale libera i canali in uscita che non contengono dati audio. Questa istruzione serve nel caso vi siano più canali in uscita che in ingresso.
Viene inizializzata la variabile *numOfChannel* al valore ritornato dalla funzione *getNumInputChannel()*, la quale ritorna il numero di canali dello streaming audio.
Viene eseguito un controllo sulla variabile e se il valore è 1 significa che si tratta di streaming mono, se è 2 significa che si tratta di streaming stereo. Nell'immagine è riportato il caso in cui *numOfChannel* sia pari a 1 e quindi si ha audio in mono. Le uniche differenze con il caso stereo sono l'inizializzazione del puntatore *RightData* a *buffer.getWritePointer(0)* invece che a *buffer.getWritePointer(1),* ed il nome della variabile *numSamplesMono*, che nel caso di streaming stereo sarà chiamata *numSamplesStereo*: le azioni compiute dalla funzione *processBlock* sono le stesse in entrambi i casi.
Come accennato prima vengono inizializzati i due puntatori a float *LeftData* e *RightData* al puntatore ritornato dal metodo *getWritePointer(0)* dell'oggetto *buffer*.
Viene poi inizializzata la variabile *numSamplesMono* al valore ritornato dal metodo *getNumSamples()* dell'oggetto *buffer*. Il valore ritornato è pari al numero di campioni dello streaming audio.
La funzione entra in un ciclo for che terminerà quando il contatore *i* avrà raggiunto un valore pari al numero di campioni dello streaming audio. All'interno del ciclo ogni singolo campione verrà processato. L'i-esimo campione sinistro e l'i-esimo campione destro vengono moltiplicati per il valore ricevuto dal parametro InputGain convertito da valore in dB a valore di ampiezza tramite la formula $10^{\text{UserParams[InputGain]}/20}$. Dopo di che viene calcolato e convertito in dB il valore di RMS in input per entrambi i canali moltiplicando per 20 il logaritmo in base 10 del valore ritornato dalla chiamata al metodo *ClockProcess* dei due oggetti inizializzati in precedenza per il calcolo del valore di RMS in input. Questo valore sarà ricevuto dall'interfaccia grafica tramite la chiamata alla funzione *getParameter* relativa ai parametri degli RMS in input del processore.
Vi è poi la chiamata al metodo *ClockProcess* dell'oggetto *mMultiBandCompCtrl*. Con questa chiamata il campione verrà suddiviso in bande e ogni banda verrà processata dai compressori di dinamica e infine il segnale audio contenuto nel campione verrà ricomposto.
Vengono poi calcolati i livelli delle singole bande per quel campione tramite la chiamata al relativo metodo dell'oggetto *mMultiBandCompCtrl*, e per ogni banda verrà calcolata la media dei livelli ricavati dal canale sinistro e dal canale destro. Questo valore sarà poi mostrato nell'interfaccia grafica per ognuna delle bande.

Vengono infine calcolati i valori di RMS in dB in uscita e il livello dell'output generale per ogni canale. La media dei due valori darà il valore del livello di uscita del segnale audio che verrà poi mostrato nell'interfaccia grafica.

3.2.3 - Implementazione del software in esame: la classe MultiBandComp

La classe MultiBandComp è la classe che si trova subito al di sotto del processore principale. I suoi metodi sono dei *setter* e dei *getter* dei parametri globali dell'applicativo, come la frequenza di campionamento, lo stato di "solo" delle bande, la tipologia di funzione di trasferimento, la tipologia di rilevazione di inviluppo e i valori delle frequenze di crossover.

Tale classe presenta anche un metodo chiamato *ClockProcess*, che riceve in ingresso il valore contenuto nei due canali audio e il numero di campioni da processare. Questo metodo è il metodo principale della classe, in quanto svolge le azioni principali per cui la classe è stata progettata.

Al suo interno vengono dichiarati quattro vettori costituiti da due elementi, e ad ogni vettore viene assegnato il contenuto del canale sinistro e del canale destro dello streaming audio. Questi vettori verranno poi filtrati e ogni vettore conterrà una delle quattro bande di frequenza. Per fare ciò il metodo *ClockProcess* esegue una chiamata al metodo omonimo dell'oggetto *splitter* di classe *MultiBandSplit*, all'interno del quale ogni vettore verrà filtrato e il contenuto verrà sostituito dal segnale audio filtrato.

Dopodiché ognuno dei due canali (destro e sinistro) delle quattro bande così ottenute vengono inviati al metodo *ClockProcess* di quattro oggetti di classe *DynamicComp*. I quattro oggetti sono infatti i quattro compressori di dinamica applicati alle bande di frequenza.

Per ogni banda così processata viene calcolato il valore di RMS e di livello, inviati poi al processore, il quale li invierà all'interfaccia grafica per mostrarli a video.

Il canale sinistro e il canale destro del segnale audio vengono poi ottenuti sommando tra loro gli elementi contenuti nelle bande alla posizione del relativo canale: alla posizione 0 di ogni vettore vi è il segnale per il canale sinistro e i quattro elementi in posizione 0 saranno sommati tra loro; alla posizione 1 vi è invece il segnale per il canale destro e i quattro elementi in posizione 1 saranno sommati tra loro.

Il risultato delle due somme diviso per il valore del livello in ingresso determinato dall'utente darà il segnale contenuto nei due canali.

Gli addendi delle due somme vengono moltiplicati per un elemento di un vettore, i cui quattro elementi possono valere o 0 o 1. Se l'elemento per cui la banda è moltiplicata vale 1 significa che la banda in questione è in stato di "solo" oppure significa che nessuna banda è in stato di "solo". Se contiene 0 significa che una o tutte le altre bande, eccetto quella considerata, sono in stato di "solo" e di conseguenza quella banda sarà annullata e non verrà riprodotta.

Tale array, denominato *bandsSoloState*, è un vettore di quattro elementi, uno per ogni banda. Il valore contenuto in ognuna delle quattro posizioni, come indicato prima, può essere 0 oppure 1 per le motivazioni indicate in precedenza. Il valore di ogni elemento viene determinato nei setter dello stato di "solo" di ogni banda. I setter ricevono in ingresso il valore inviato dalla *setParameter* del processore principale relativa al pulsante di solo delle bande. Poi eseguono un controllo su una funzione booleana chiamata *anySolo()*. Questa funzione dichiara al suo interno due variabili booleane chiamate rispettivamente *oneSolo* e *moreSolo*. *oneSolo* controlla se vi è una sola banda posta in "solo", verificando che la somma dei quattro parametri di solo sia pari a 1. Se questa condizione si verifica significa che un parametro è pari a 1 e i tre rimanenti sono pari a 0. *moreSolo* controlla se vi sono più bande poste in stato di solo, compiendo un operazione logica di OR tra le condizioni "*parametro di solo di ogni banda > 0,5"*. Se la variabile booleana ritorna *true* significa che una o più bande sono in stato di "solo", altrimenti significa che nessuna banda è in stato di "solo".

La funzione *anySolo* esegue poi un controllo sulla variabile *oneSolo* e se risulta *true* riempie l'array *bandsSoloState* ponendo a 0 tutti gli elementi. Esegue poi un controllo sulla variabile *moreSolo* e se risulta *false* riempie l'array *bandsSoloState* ponendo a 1 tutti i suoi elementi. Infine la funzione *anySolo* ritorna il valore di *moreSolo*. I setter, come accennato prima, eseguono un controllo sul valore ritornato dalla funzione booleana *anySolo*: se ritorna *true* allora ad ogni elemento dell'array *bandsSoloState* viene assegnato il valore del parametro di solo di ogni banda: *bandsSoloState[0]=stato di solo prima banda; bandsSoloState[1]=stato di solo seconda banda;* ecc.

```
50 bool MultiBandComp::anySolo()
51 {
52     bool oneSolo = ((m_Solo1 + m_Solo2 + m_Solo3 + m_Solo4) == 1);
53     bool moreSolo = ((m_Solo1 > 0.5f)||(m_Solo2 > 0.5f)||(m_Solo3 > 0.5f)||(m_Solo4 > 0.5f));
54 
55     if(oneSolo)
56     {
57         for (int i = 0; i<4; ++i)
58             bandsSoloState[i] = 0;
59     }
60 
61     if(!moreSolo)
62     {
63         for (int i = 0; i<4; ++i)
64             bandsSoloState[i] = 1;
65     }
66 
67     return moreSolo;
68 }
```

Codice della funzione *anySolo()*

In figura è presente il codice sorgente relativo al metodo *anySolo()* della classe *MultiBandComp*. Nella pagina seguente è presente il codice dei setter dello stato di "solo" di ogni banda.

```
69
70  void MultiBandComp::setSolo1(float SoloBand1)
71  {
72      m_Solo1 = SoloBand1;
73
74      if(anySolo())
75      {
76          bandsSoloState[0] = m_Solo1;
77          bandsSoloState[1] = m_Solo2;
78          bandsSoloState[2] = m_Solo3;
79          bandsSoloState[3] = m_Solo4;
80      }
81  }
82
83  void MultiBandComp::setSolo2(float SoloBand2)
84  {
85      m_Solo2 = SoloBand2;
86
87      if(anySolo())
88      {
89          bandsSoloState[0] = m_Solo1;
90          bandsSoloState[1] = m_Solo2;
91          bandsSoloState[2] = m_Solo3;
92          bandsSoloState[3] = m_Solo4;
93      }
94  }
95
96  void MultiBandComp::setSolo3(float SoloBand3)
97  {
98      m_Solo3 = SoloBand3;
99
100     if(anySolo())
101     {
102         bandsSoloState[0] = m_Solo1;
103         bandsSoloState[1] = m_Solo2;
104         bandsSoloState[2] = m_Solo3;
105         bandsSoloState[3] = m_Solo4;
106     }
107 }
108
109 void MultiBandComp::setSolo4(float SoloBand4)
110 {
111     m_Solo4 = SoloBand4;
112
113     if(anySolo())
114     {
115         bandsSoloState[0] = m_Solo1;
116         bandsSoloState[1] = m_Solo2;
117         bandsSoloState[2] = m_Solo3;
118         bandsSoloState[3] = m_Solo4;
119     }
120 }
```

Codice relativo ai metodi di set degli stati di “solo” di ogni banda

3.2.4 - Implementazione del software in esame: la classe MultiBandSplit e le tre tipologie di filtro

La classe *MultiBandSplit* è costituita da tre setter, tre getter e dal metodo *ClockProcess.*

All'interno della classe vengono dichiarati quattro oggetti: uno di classe *LowPass*, due di classe *BandPass*, e uno di classe *HighPass*. Questi oggetti rappresentano i filtri da cui verrà processato il segnale audio.

Ogni setter della classe *MultiBandSplit* riceve in ingresso il valore di una frequenza di crossover e il valore della frequenza di campionamento. Questi valori vengono poi inviati al metodo *setFilterParameters* dei filtri:

- il setter che riceve in ingresso la prima frequenza di crossover invierà i parametri ricevuti al metodo *setFilterParameters* del filtro passa-basso e del primo passa-banda;
- il setter che riceve in ingresso la seconda frequenza di crossover invierà i parametri al metodo *setFilterParameters* del primo e del secondo passa-banda;
- il setter che riceve in ingresso la terza frequenza di crossover invierà i parametri al metodo *setFilterParameters* del secondo passa-banda e del filtro passa-alto.

Il metodo *ClockProcess* della classe *MultiBandSplit* richiamerà i metodi *ClockProcess* di ognuno dei filtri, alle quali passerà un puntatore a float che conterrà il segnale audio da filtrare e il numero di campioni da processare. Ogni puntatore, dopo essere stato processato dalla relativa *ClockProcess,* conterrà il segnale audio contenuto nel canale sinistro e nel canale destro di ogni banda.

```
void MultiBandSplit::setCrossoverLow(int CrossFreq1, double sr)
{
    m_x0 = CrossFreq1;

    LowFilt -> setFilterParameters(m_x0, sr);
    LowMidFilt -> setFilterParameters(m_x0, m_x1, sr);
}

void MultiBandSplit::setCrossoverMid(int CrossFreq2, double sr)
{
    m_x1 = CrossFreq2;

    LowMidFilt -> setFilterParameters(m_x0, m_x1, sr);
    HighMidFilt -> setFilterParameters(m_x1, m_x2, sr);
}

void MultiBandSplit::setCrossoverHigh(int CrossFreq3, double sr)
{
    m_x2 = CrossFreq3;

    HighMidFilt -> setFilterParameters(m_x1, m_x2, sr);
    HighFilt -> setFilterParameters(m_x2, sr);
}

void MultiBandSplit::ClockProcess(float *Band1, float *Band2, float *Band3, float *Band4, int numSamples)
{
    LowFilt->ClockProcess(Band1, numSamples);

    LowMidFilt->ClockProcess(Band2, numSamples);

    HighMidFilt->ClockProcess(Band3, numSamples);

    HighFilt->ClockProcess(Band4, numSamples);
}
```

Codice relativo ai setter e al metodo *ClockProcess* della classe *MultiBandSplit*

Verranno ora analizzate le tre classi di filtri implementate all'interno del progetto:

- *LowPass*: La classe *LowPass* contiene al suo interno due metodi principali, il metodo *setFilterParameters* e il metodo *ClockProcess*. Questi due metodi sono comuni per tutte e tre le classi di filtri. Il metodo *setFilterParameters* riceve in ingresso la prima frequenza di crossover e la frequenza di campionamento. Alla posizione 0 del vettore *L_Params*, dichiarato nell'header file della classe come variabile di tipo *Dsp::Params* (tipo contenuto nella libreria descritta al capitolo 2), è assegnato il valore della frequenza di campionamento: sarà il primo parametro considerato per la realizzazione del filtro. Alla posizione 1 di *L_Params* è assegnato il valore 4, che indica l'ordine del filtro. Infine in posizione 2 è assegnato il valore della prima frequenza di crossover, la quale sarà la frequenza di taglio del filtro passa-basso. Dopodichè viene richiamato il metodo *setParameters* dell'oggetto *LowPassFlt*, di classe *Dsp::Filter* (classe contenuta nella libreria descritta nel secondo capitolo) dichiarato nell'header e inizializzato nel costruttore come filtro passa-basso del di Butterworth del quarto ordine, al quale viene passato il vettore *L_Params*.

Nel metodo *ClockProcess* viene dichiarato un vettore di puntatori a float di due elementi e ad ogni elemento sono assegnati rispettivamente gli indirizzi del canale sinistro e del canale destro della prima banda. Il vettore di puntatori e il numero di campioni vengono poi passati al metodo *process* dell'oggetto *LowPassFlt*.

- *BandPass*: La classe *BandPass* è costituita dagli stessi metodi di *LowPass*. Il metodo *setFilterParameters* riceve i valori di due frequenze di crossover e della frequenza di campionamento. Nell'header file sono dichiarati il vettore di parametri *BP_Params* e l'oggetto, di tipo *Dsp::Filter, BandPassFlt* che nel costruttore sarà inizializzato come filtro passa-banda di Butterworth del quarto ordine. Poichè si tratta di un filtro passa-banda, sono tenuti in considerazione quattro parametri, cioè la frequenza di campionamento, l'ordine, il centro di banda e l'ampiezza di banda, assegnati rispettivamente agli elementi in posizione 0, 1, 2 e 3 del vettore *BP_Params*. Il centro di banda è stato determinato come la media tra le due frequenze di crossover ricevute in ingresso da *setFilterParameters*, mentre l'ampiezza di banda è stata determinata come il massimo tra il valore assoluto della seconda frequeza di crossover ricevuta meno la prima e 1 (max(abs(xover2-xover1), 1)). Questo per evitare che il filtro possa avere ampiezza di banda nulla diventando così instabile. Il metodo *ClockProcess* agisce allo stesso modo del metodo *ClockProcess* di *LowPass*.

- *HighPass*: La classe *HighPass* presenta gli stessi metodi delle due classi precedenti i quali agiscono allo stesso modo dei metodi della classe *LowPass*. La differenza sta nel fatto che l'oggetto di tipo *Dsp:Filter* viene inizializzato nel costruttore come un filtro di Butterworth del quarto ordine.

```
#ifndef LOWBAND_H_INCLUDED
#define LOWBAND_H_INCLUDED

#include "Dsp.h"

class LowPass
{
public:
    LowPass();
    ~LowPass();

    void setFilterParameters(int CrossFreq1, double sampleRate);

    void ClockProcess(float* Input, int numSamples);

private:

    double m_SampleRate;
    int m_CrossFreq1, m_LPFCutFrequency;
    Dsp::Filter *LowPassFlt;
    Dsp::Params L_Params;
};

#endif  // LOWBAND_H_INCLUDED
```

```
#include "LowPass.h"

LowPass::LowPass()
{
    LowPassFlt = new Dsp::SmoothedFilterDesign<Dsp::Butterworth::Design::LowPass<4>, 2> (1024);
}

LowPass::~LowPass()
{
    LowPassFlt = nullptr;
}

void LowPass::setFilterParameters(int CrossFreq1, double sampleRate)
{
    m_SampleRate = sampleRate;
    m_CrossFreq1 = CrossFreq1;

    L_Params[0] = m_SampleRate;                         // sample rate
    L_Params[1] = 4;                                    // order
    L_Params[2] = m_CrossFreq1;                         // cutoff frequency

    LowPassFlt->setParams(L_Params);
}

void LowPass::ClockProcess(float *Input, int numSamples)
{
    int m_samples = numSamples;
    float *audioData[2];

    audioData[0] = &Input[0];
    audioData[1] = &Input[1];

    LowPassFlt->process(m_samples, audioData);
}
```

Header file e codice sorgente della classe *LowPass*

E' possibile notare dall'immagine la chiamata nell'header file dell'header *Dsp.h*. In questo header

sono contenuti tutti i riferimenti alle classi di filtri e ai rispettivi metodi. Grazie a questa libreria è stato possibile implementare i filtri necessari per filtrare il segnale audio in modo molto semplice, inizializzando il filtro alla tipologia richiesta, come mostrato nel codice. I valori presenti nell'inizializzazione rappresentano rispettivamente l'ordine del filtro (4) il numero massimo di canali processabili dal filtro (2) e il numero di campioni oltre il quale verranno attenuate le variazioni dei parametri (1024).

3.2.5 - Implementazione del software in esame: il compressore di dinamica

Nel sotto-paragrafo relativo alla classe *MultiBandComp* si è detto che sono stati dichiarati quattro oggetti appartenenti alla classe DynamicComp, i quali svolgono la funzione di compressori di dinamica.

La classe DynamicComp è la classe che sta in cima alla struttura del compressore di dinamica. Infatti vi sono ulteriori classi che compiono diverse azioni e che stanno dietro al funzionamento del processore di dinamica. Si può schematizzare la struttura delle classi che compongono il compressore di dinamica con il seguente diagramma:

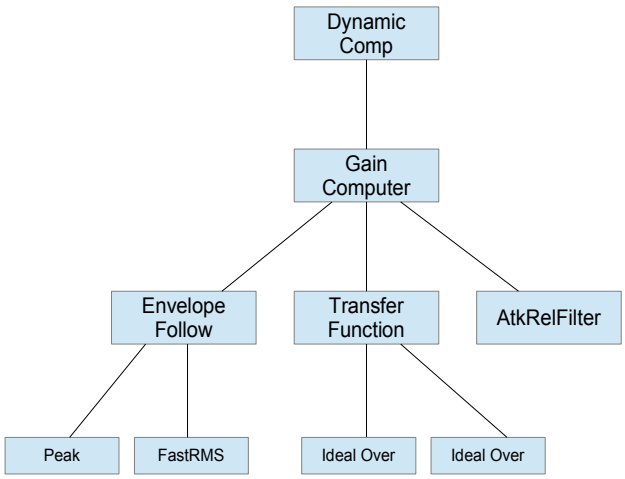


Diagramma relativo alla struttura del processore di dinamica

La classe *DynamicComp* è costituita principalmente da tutti i setter e i getter relativi ai parametri necessari al processore, i quali, una volta ricevuti in ingresso, vengono inviati ai relativi setter della classe *GainComputer*, richiamati tramite la chiamata ai metodi setter dell'oggetto *Compressor,* dichiarato nell'header file della classe, appartenente alla classe *GainComputer*. La classe *DinamicComp* presenta anche un metodo *ClockProcess*, il quale esegue un controllo sul parametro relativo allo stato di "bypass" del compressore: se il parametro vale 0 significa che il compressore non è bypassato e richiama il metodo *ClockProcess* dell'oggetto *Compressor* appartenente alla classe *GainComputer* al quale passa le informazioni contenute nel canale sinistro e nel canale destro del campione da processare. Se il parametro risulta essere diverso da 0 significa che il compressore è

bypassato, e quindi moltiplica il canale destro e il canale sinistro del campione per il livello di make up gain assegnato a quel determinato compressore.

La classe *GainComputer* può essere considerata come il cuore del compressore. Infatti essa riceve ed elabora i parametri del compressore tramite i suoi setter, e nel suo metodo *ClockProcess* richiama le funzioni *ClockProcess* delle classi relative al rilevatore di inviluppo (*EnvelopeFollow*) alla funzione di trasferimento (*TransferFunction*) e al filtro di attacco e rilascio (*AtkRelFilter*).

Prima di soffermarsi ad analizzare il comportamento del metodo *ClockProcess* della classe *GainComputer* è opportuno analizzare in che modo vengono impostati il rilevatore di inviluppo e la funzione di trasferimento.

Il setter relativo al rilevatore di inviluppo è il metodo *setFollow*, il quale riceve in ingresso una stringa in cui è indicato il tipo di rilevatore di inviluppo e riceve il tempo di rilascio relativo al compressore di dinamica. La classe *EnvelopeFollow* possiede due sotto-classi, *Peak* e *FastRMS*, e gli oggetti relativi alla rilevazione di inviluppo per il canale destro e sinistro, dichiarati come puntatori ad *EnvelopeFollower*, possono essere inizializzati nel metodo *setFollow* di *GainComputer* al costruttore di una o dell'altra sotto-classe sfruttando la proprietà del polimorfismo fornita dal linguaggio C++: se la stringa è uguale a “Peak”, gli oggetti dichiarati per la rilevazione di inviluppo dei due canali del campione saranno inizializzati al costruttore della classe *Peak*, mentre se la stringa è uguale a “RMS” gli oggetti dichiarati per la rilevazione di inviluppo saranno inizializzati al costruttore della classe *FastRMS* al quale verrà passato come argomento una dimensione iniziale per la finestra pari a (30*(frequenza di campionamento)/1000) campioni, dove 30 indica un valore temporale di 30ms.

Queste diverse inizializzazioni determinano un diverso comportamento dei rilevatori di inviluppo quando viene richiamato il loro metodo *ClockProcess*: infatti sulla base del costruttore della sotto-classe a cui sono inizializzati verrà richiamata la funzione *ClockProcess* di una o dell'altra sotto-classe.

Discorso analogo può essere fatto per il metodo *setBehaviour*, che tramite la proprietà del polimorfismo inizializzerà il puntatore a oggetto appartenente alla classe *TransferFunction,* per il calcolo della funzione di trasferimento del compressore, al costruttore di una delle sue due sotto-classi *IdealOver* e *LinearOver,* sulla base del valore contenuto nella stringa ricevuta dal metodo *setBehaviour*. Di conseguenza non appena verrà richiamato il metodo *ClockProcess* di questo oggetto, sulla base dell'inizializzazione presente nel setter verrà richiamato il metodo *ClockProcess* della classe *LinearOver* o della classe *IdealOver*.

Nella pagina successiva è presente il codice relativo a questo utilizzo del polimorfismo relativo ai metodi *setBehaviour* e *setFollow*.

```
/*
 ========================================================================

 Next function determines if behaviour will be IdealOver or LinearOver

 ========================================================================
 */

void GainComputer::setBehaviour(string Behaviour)
{
    m_Behaviour = Behaviour;

    if (m_Behaviour == "IdealOver")
    {
        tFunct = new IdealOver();
        tFunct -> setRatio(m_Ratio);            // sets ratio for transfer function
        tFunct -> setThreshold(m_Threshold);    // sets threshold for transfer function
    }
    else if(m_Behaviour == "LinearOver")
    {
        tFunct = new LinearOver();
        tFunct -> setRatio(m_Ratio);            // sets ratio for transfer function
        tFunct -> setThreshold(m_Threshold);    // sets threshold for transfer function
    }
}

/*
 ========================================================================

 Next function determines if Envelope will be followed as Peak or RMS

 ========================================================================
 */

void GainComputer::setFollow(string Follower, float Release)
{
    m_Follower = Follower;

    float m_RelForFollow = Release;

    if (m_Follower == "Peak")
    {
        EnvelopeLeft = new Peak();
        EnvelopeRight = new Peak();
        EnvelopeLeft -> setBuffer(m_RelForFollow);
        EnvelopeRight -> setBuffer(m_RelForFollow);
    }
    else if(m_Follower == "RMS")
    {
        EnvelopeLeft = new FastRMS(30*(0.001*m_SampleRate));
        EnvelopeRight = new FastRMS(30*(0.001*m_SampleRate));
        EnvelopeLeft -> setBuffer(m_RelForFollow);
        EnvelopeRight -> setBuffer(m_RelForFollow);
    }
}
```

Codice relativo ai metodo *setBehaviour* e *setFollow* della classe *GainComputer*

Il metodo *ClockProcess* della classe *GainComputer* compie quattro azioni, ognuna ripetuta per il canale destro del campione che si sta processando e per il canale sinistro:

- assegna alla variabile che tiene traccia del valore di inviluppo il valore ritornato dalla funzione *ClockProcess* di una delle due sotto-classi della classe *EnvelopeFollow,* al cui costruttore è stato inizializzato l'oggetto incaricato di rilevare l'inviluppo del segnale;
- assegna alla variabile che tiene traccia del gain reduction, ricavato dalla funzione di trasferimento, il valore ritornato dal metodo *ClockProcess* di una delle due sotto-classi di *TranferFunction,* al cui costruttore è stato inizializzato l'oggetto incaricato di calcolare il gain reduction tramite la funzione di trasferimento richiesta, a partire dal valore di inviluppo precedentemente calcolato;
- assegna alla variabile che tiene traccia del gain reduction processato tramite il filtro di attacco e rilascio il valore ritornato dal metodo *ClockProcess* dell'oggetto appartenente alla classe *AtkRelFilter*, incaricato di calcolare il gain reduction processato dal filtro di attacco e rilascio;
- moltiplica il segnale audio contenuto nel campione per il valore assegnato alla variabile che tiene traccia del valore di gain reduction processato e per il livello di make up gain assegnato al compressore.

```
void GainComputer::ClockProcess(float *InputL, float *InputR)
{
    EnvLeft = EnvelopeLeft -> ClockProcess(*InputL);        // Calculates envelope for left channel using Envelope Follow
    EnvRight = EnvelopeRight -> ClockProcess(*InputR);       // Calculates envelope for right channel using Envelope Follow

    GainRedLeft = tFunct -> ClockProcess(EnvLeft);     // Calculates gain reduction for left channel using Transfer Function
    GainRedRight = tFunct -> ClockProcess(EnvRight);   // Calculates gain reduction for right channel using Transfer Function

    GainRedLeftProc = GainRedAtkRelFilterLeft.ClockProcess(GainRedLeft);      // Attack Release filter for left channel
    GainRedRightProc = GainRedAtkRelFilterRight.ClockProcess(GainRedRight);     // Attack Release filter for right channel

    *InputL *= (GainRedLeftProc) * m_MkUpGain;        // Compression is result of input left * gain reduction left * makup gain
    *InputR *= (GainRedRightProc) * m_MkUpGain;       // Compression is result of input right * gain reduction right * makup gain
}
```

Il metodo *ClockProcess* della classe *GainComputer*

L'immagine mostra come sono state implementate le quattro azioni descritte in precedenza.

La classe *EnvelopeFollow* posside un setter e una *ClockProcess* virtuali. Il motivo per cui questi due metodi sono virtuali è dato dal fatto che, venendo sfruttata la proprietà del polimorfismo dell'oggetto dichiarato in *GainComputer* relativo alla rilevazione di inviluppo, in base a quale costruttore viene inizializzato questo oggetto il setter e la funzione *ClockProcess* svolgeranno le azioni definite dai metodi o della sotto-classe *Peak* o della sotto-classe *FastRMS*.

Nel costruttore della sotto-classe *Peak* la variabile di inviluppo che sarà ritornata dal metodo *ClockProcess* è inizializzata a 0. Il setter di *Peak* riceve in ingresso il valore del tempo di rilascio del compressore e al suo interno viene calcolato un fattore di decadimento pari a 0,75 elevato al reciproco della dimensione della finestra (il tempo di rilascio). Il metodo *ClockProcess* della sotto-classe *Peak* ritorna il massimo tra il valore assoluto del contenuto del segnale audio e l'inviluppo precedente moltiplicato per il fattore di decadimento calcolato nel setter.

```

#include "Peak.h"

// Constructor
Peak::Peak()
{
    Y = 0.0f;
}

// Destructor
Peak::~Peak()
{

}


// Params

void Peak::setBuffer(float Release)
{
    m_Time = (int)Release;
    decay = pow(0.75f, (1/(float)m_Time));
}


/*
 ================================================================================

 Next function provides an algorithm for a Peak follow

 ================================================================================
 */

float Peak::ClockProcess(float input)
{
    Y = max(abs(input),(Y*decay));
    return Y;
}
```

Implementazione del setter della sotto-classe *Peak* e del metodo *ClockProcess*

La sotto-classe *FastRMS* implementa un algoritmo per il calcolo del valore efficace del segnale audio. All'interno di una finestra costituita da un certo numero di campioni del segnale audio inserisce il quadrato del valore del segnale contenuto in ognuno dei campioni diviso il numero di campioni che può contenere la finestra. Dopodiché calcola la somma di tutti gli elementi contenuti nella finestra e di tale somma ritorna la radice quadrata. Ogni volta che è stata completata, fino al termine dello streaming audio, la finestra si sposta di un numero di campioni pari al numero di campioni che può contenere. Queste azioni sono state implementate come segue:

- al costruttore della sotto-classe viene passata la dimensione temporale di default e questa finestra viene poi passata al setter della sotto-classe; nel costruttore vengono poi inizializzati a 0 il contatore che terrà traccia della posizione all'interno della finestra e il valore della somma degli elementi;
- il setter riceve la dimensione della finestra data dal tempo di rilascio del compressore, inizializza un vettore finestra a un nuovo vettore di float della dimensione data dal valore ricevuto dal setter e assegna il valore 0 a tutti gli elementi della finestra;
- il metodo *ClockProcess* assegna ad una variabile il quadrato del contenuto del segnale diviso la dimensione della finestra; alla variabile che tiene traccia della somma degli elementi è sottratto l'elemento presente nell'attuale posizione della finestra e viene sommato il nuovo valore di segnale al quadrato fratto la dimensione della finestra; l'elemento presente nell'attuale posizione della finestra viene sostituito dal valore aggiunto alla variabile che tiene traccia della somma degli elementi; viene aggiornato l'indice che assume valore pari al resto del suo attuale valore più 1 diviso la dimensione della finestra, così che una volta raggiunto l'ultimo elemento della finestra l'indice riassuma valore 0; infine il metodo *ClockProcess* ritornerà la radice quadrata della variabile che tiene traccia della somma degli elementi.

```cpp
#include "RMS.h"

// Constructor

FastRMS::FastRMS(float defaultWindowSize)
{
    setBuffer(defaultWindowSize);
    i = 0;
    TotalSum = 0;
}


//Destructor

FastRMS::~FastRMS()
{
    Window = nullptr;
}


// Params

void FastRMS::setBuffer(float WindowSize)
{
    m_Time = (int)WindowSize;

    /*if (Window!=NULL)
        Window = nullptr;*/

    Window = new float[m_Time];

    for (int j = 0; j<m_Time; j++)
        Window[j] = 0;
}
```

```cpp
/*
 ========================================================================

 Next function provides an algorithm for a RMS follow

 ========================================================================
 */

float FastRMS::ClockProcess(float Input)
{
    m_Input = (Input * Input)/m_Time;
    TotalSum = TotalSum - Window[i] + m_Input;
    Window[i] = m_Input;
    i = (i+1)%m_Time;

    return sqrt(TotalSum);
}
```

Implementazione di costruttore, distruttore, setter e ClockProcess della sotto-classe FastRMS

La classe *TransferFunction* presenta al suo interno due setter, uno per la threshold e uno per il rapporto di compressione, e una funzione virtuale *ClockProcess*. Il motivo per cui la funzione *ClockProcess* è virtuale è lo stesso relativo alla classe *EnvelopeFollower*. Infatti, sulla base del costruttore a cui sarà inizializzato l'oggetto relativo al calcolo della funzione di trasferimento dichiarato nella classe *Gain Computer*, la funzione *ClockProcess* della classe *TransferFunction* assumerà il comportamento del metodo *ClockProcess* di *IdealOver* oppure di *LinearOver*.
Il metodo *ClockProcess* della sotto-classe *IdealOver* si serve di una funzione di trasferimento di tipo esponenziale per il calcolo del gain reduction: controlla se il valore di inviluppo è maggiore al valore della threshold, in caso affermativo eleva il rapporto tra threshold e inviluppo ad 1 meno il rapporto di compressione e ritorna il valore così ottenuto.

```
25 float IdealOver::ClockProcess(float Env)
26 {
27     m_Env = Env;
28
29     if (m_Env>m_Threshold)
30         GR = pow((m_Threshold/(m_Env)), (1.0f - m_Ratio));
31
32     return GR;
33 }
```

Implementazione del metodo *ClockProcess* della sotto-classe *IdealOver*

Il metodo *ClockProcess* della sotto-classe *IdealOver* si serve di una funzione di trasferimento che utilizza funzioni lineari per il calcolo del gain reduction: processa il valore di inviluppo moltiplicando il minimo tra valore di inviluppo e threshold per 1 meno il valore del rapporto di compressione e a questo prodotto somma il prodotto tra inviluppo e il rapporto di compressione.
Infine determina il gain reduction tramite il rapporto tra inviluppo processato e valore effettivo di inviluppo e ritorna il valore di gain reduction ottenuto.
Nell'immagine a pagina seguente è rappresentato il codice relativo all'implementazione del metodo *ClockProcess* della sotto-classe *LinearOver*.

```
25 float LinearOver::ClockProcess(float Env)
26 {
27     m_Env = Env;
28
29     pE = (min(m_Env, m_Threshold) * (1-m_Ratio)) + (m_Env * m_Ratio);
30
31     GR = pE/m_Env;
32
33     return GR;
34 }
```

Implementazione del metodo *ClockProcess* della sotto-classe *LinearOver*

L'ultima classe da descrivere è la classe *AtkRelFilter*, la quale implementa un algoritmo che confronta il gain reduction del sample precedente con il valore determinato dalla funzione di trasferimento e determina, quindi, l'effettivo gain reduction da moltiplicare all'attuale campione, moltiplicando il gain reduction del campione precedente per un fattore determinato o dal tempo di attacco oppure dal tempo di rilascio.

Questa classe è caratterizzata da un setter per il tempo di attacco, da un setter per il tempo di rilascio e dal metodo *ClockProcess* che riceve in ingresso il valore ricavato dalla funzione di trasferimento.
Il metodo *ClockProcess* confronta il valore del gain reduction del campione precedente con il valore determinato dalla funzione di trasferimento dell'attuale campione di segnale audio:

- se il valore del gain reduction del campione precedente è maggiore del valore determinato dalla funzione di trasferimento del campione attuale, significa che vi è stata una diminuzione di gain reduction, il segnale audio sta diventando più forte e quindi il fattore è determinato dal tempo di attacco: il fattore, chiamato *m* nella funzione, viene calcolato come il rapporto tra valore ricevuto dalla funzione di trasferimento e precedente gain reduction e tale rapporto viene elevato al reciproco del tempo di attacco;
- se il valore del gain reduction del campione precedente è minore del valore determinato dalla funzione di trasferimento del campione attuale, significa che vi è stato un aumento di gain reduction, il segnale audio sta diventando più debole e quindi il fattore è determinato dal tempo di rilascio: il fattore *m* viene calcolato come il rapporto tra valore ricevuto dalla funzione di trasferimento e precedente gain reduction e tale rapporto viene elevato al reciproco del tempo di rilascio.

Il metodo *ClockProcess* ritornerà il valore del precedente gain reduction moltiplicato per il fattore appena calcolato.

```
38 float AtkRelFilter::ClockProcess(float GR)
39 {
40     float m;
41 
42     m_GR = GR;
43 
44     if (pGR>GR)
45         m = pow((m_GR/pGR),(1/m_AtkTime));      // Gain reduction is decreasing (audio signal is louder) so attack time is used
46     else
47         m = pow((m_GR/pGR),(1/m_RelTime));      // Gain reduction is increasing (audio signal is quieter) so release time is used
48 
49     pGR = pGR*m;
50 
51     m_GR = pGR;
52 
53     return m_GR;
54 }
```

Implementazione del metodo *ClockProcess* della classe *AtkRelFilter*

Conclusioni finali

Al termine della trattazione del software in esame, è possibile concludere che esso potrebbe essere sfruttato in ambiti professionali relativi alla produzione audio, come studi di registrazione, di mixaggio e di mastering, velocizzando il lavoro di professionisti del settore grazie all'automatizzazione del settaggio dei parametri, sia in interfaccia semplificata che in interfaccia a più parametri. Inoltre è possibile sfruttare il software anche in ambito di home studio, per il quale l'interfaccia semplificata è molto indicata in quanto gli utenti meno esperti sarebbero facilitati nell'operazione di compressione multi-banda.

L'applicativo potrebbe essere migliorato in futuro implementando algoritmi per il riconoscimento automatico della timbrica degli strumenti musicali sia su segnali monofonici che polifonici, e per ogni strumento riconosciuto applicare l'automatizzazione prevista.

Bibliografia

[1] R. Jeffs, S. Holden, D. Bohn, “Dynamics Processors – Technology & Application Tips”, Rane Corporation, 2005

[2] ISO/IEC, “Working Draft, Standard for Programming Language C++”, document number N4296, 19-11-2014

[3] A. Bertoni, P. Campadelli, G. Grossi, “Introduzione all'elaborazione dei segnali”, Università degli Studi di Milano, 2010

[4] A. Bertoni, G. Grossi, “Dispensa del corso di Elaborazione Numerica dei Segnali”, Università degli Studi di Milano, 2009

[5] S. Butterworth, “On the Theory of Filter Amplifiers”, Wireless Engineer vol.7 pag. 536-541, ottobre 1930

[6] V. Falco, “A Collection of Useful C++ Classes for Digital Signal Processing”, documentation, 2009

Sitografia

[7] http://www.cplusplus.com/info/history/, 28-11-2015

[8] https://isocpp.org/std/status, 28-11-2015

[9] http://www.juce.com/features 03-12-2015