UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento Politecnico di Ingegneria e Architettura

Corso di Laurea in Ingegneria Gestionale

Percorso di Ingegneria dell'informazione

Tesi di laurea

Equalizzatore grafico IIR con filtri Shelving

Relatore:
Prof. Bernardini Riccardo

Laureando
Drius Andrea

Ultima revisione, post-Laurea 05/11/18

INDICE

INTRODUZIONE

Negli ultimi anni il sistema operativo *GNU/Linux* si è visto protagonista di svariati cambiamenti atti a migliorare l'esperienza degli utenti in ambito *desktop.* Importanti innovazioni sono state introdotte da *Red Hat,* in particolare dallo sviluppatore *Lennart Poettering* noto per aver scritto il sistema di Init *Systemd* che è stato adottato dalla maggior parte delle distribuzioni e il *sound system PulseAudio* che semplifica la gestione dell'audio in ambito *desktop.* I due progetti fanno parte della raccolta *freedesktop.org*[s1] che si pone l'obiettivo di rendere la gestione del sistema più versatile sia a livello professionale sia per quanto concerne l'utenza media.

PulseAudio svolge la funzione di *proxy* audio tra le varie applicazioni che elaborano input o output sonori, permettendo di regolare il volume delle singole applicazioni, applicare filtri di vario genere, equalizzare e mixare i flussi audio. Il comando *pactl* permette di controllare il demone di *PulseAudio* che lavora in background ed elabora gli stream. Esiste inoltre un front-end grafico, *pavucontrol*, che semplifica la gestione qualora non si volesse usufruire del terminale per controllare *PulseAudio.*

PulseAudio è fornito assieme a vari moduli che possono essere caricati per ampliarne le funzionalità, ci sono moduli per la gestione di protocolli, per la gestione di dispositivi *bluetooth*, moduli di compatibilità con altre utility, moduli che permettono di filtrare e mixare gli stream e molto altro. Il modulo *module-equalizer-sink* fornisce un equalizzatore grafico ma è stato deprecato e verrà presto rimosso causa instabilità, bug e altre problematiche tra cui la scarsa fedeltà del suono elaborato. Questa tesi si pone l'obiettivo di sviluppare un nuovo equalizzatore che possa essere usato sia a livello professionale sia per uso domestico, al fine di sviluppare una soluzione ottimale alla sostituzione dell'attuale modulo. Il modulo sviluppato in questa tesi prenderà il nome di *module-eqpro-sink* e salvo imprevisti verrà integrato nel *main tree* di *PulseAudio* a partire dalla versione *13.0* candidandosi a diventare un possibile standard su *desktop.*

È stato possibile lavorare in stretto contatto con gli sviluppatori e manutentori di *PulseAudio* mediante il protocollo di messaggistica *IRC (Internet Relay Chat)* scambiando informazioni sul canale *#pulseaudio* della rete *freenode*, punto di riferimento di vari progetti *Open Source.*

Per lo sviluppo dell'equalizzatore si è scelto di adottare filtri *IIR Shelving* che si prestano alla modellizzazione di equalizzatori grafici e parametrici, in quanto presentano un ridotto *ripple* in banda passante e favoriscono il progetto di filtri contigui e complementari. Si è partiti da un modello analogico che poi è stato discretizzato in modo da mantenere il più possibile le caratteristiche desiderate. I filtri *Shelving* presentano un'asimmetria tra amplificazione, caratterizzata da un *ripple* trascurabile, e attenuazione, caratterizzata da un *ripple* accentuato. Mediante la configurazione *cut-boost* il ripple in attenuazione è stato reso analogo a quello in amplificazione. Si è lavorato nel rispetto delle norme *ANSI* per le specifiche delle bande dei filtri degli equalizzatori audio[b3].

module-eqpro-sink è stato affiancato ad una interfaccia grafica, scritta in *C++* e *QML* mediante il framework *Qt*, che permette di controllare i parametri dell'equalizzatore comunicando con il modulo, grazie anche all'aiuto dello sviluppatore *Georg Chini*[s2] attivo su *PulseAudio* che ha sviluppato per l'occasione alcune *patch* atte a permettere la comunicazione tra moduli e applicazioni terze.

La tesi illustrerà tutti i passi che sono stati compiuti al fine di ottenere un equalizzatore professionale per *PulseAudio* soffermandosi in particolare sui modelli matematici che hanno permesso di ottenere l'alta fedeltà richiesta.

Capitolo 1

Filtro *Shelving* analogico

1.1 Definizione di filtro *Shelving*

I filtri *Shelving* sono particolarmente adatti per equalizzatori grafici e parametrici in quanto possiedono una banda passante piatta, con basso *ripple* che favorisce la modellizzazione di filtri contigui e complementari. In estrema sintesi, i filtri *Shelving* si prestano alla costruzione di un banco di filtri complementari che vengono generati a partire da una funzione di trasferimento *Butterworth* con guadagno g in banda passante e guadagno unitario in banda attenuata.

Per trattare i filtri *Shelving* è, pertanto, necessario considerare la funzione di trasferimento del prototipo *Butterworth* di tipo passa-basso a banda unitaria e di ordine M:

$$\alpha(m,M) := \pi\cdot\left(\frac{1}{2} - \frac{2\cdot m - 1}{2\cdot M}\right) \qquad H_{LP\_Butter}(s,M) := \prod_{m=1}^{M}\left(\frac{e^{i\cdot\alpha(m,M)}}{s + e^{i\cdot\alpha(m,M)}}\right)$$

Equazione 1.1: Funzione di trasferimento Butterworth [b1]

La funzione di trasferimento del filtro prototipo *Shelving* passa-basso con guadagno g in banda passante e unitario fuori banda è:

$$H_{Sh}(s,M,g) := \prod_{m=1}^{M}\frac{s + e^{j\cdot\alpha(m,M)}\cdot\sqrt[M]{g}}{s + e^{j\cdot\alpha(m,M)}}$$

Equazione 1.2: Shelving passa-basso [b2]

Si nota immediatamente come i filtri *Shelving* abbiano gli stessi poli dei *Butterworth*.

Una peculiarità dei filtri *Shelving* è che scegliendo il guadagno g maggiore o minore di 1 è possibile ottenere filtri che amplificano o attenuano la stessa banda, esattamente come ci si aspetta in un equalizzatore grafico. Per ottenere l'amplificazione opposta, in decibel, è sufficiente usare il reciproco di g, come mostrato dalle seguenti figure:

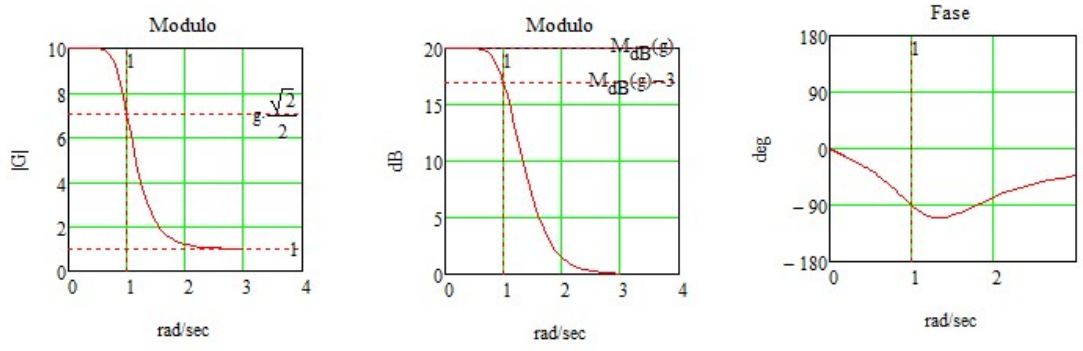


Figura 1.1: Grafico di un filtro Shelving con M=4 e g=10

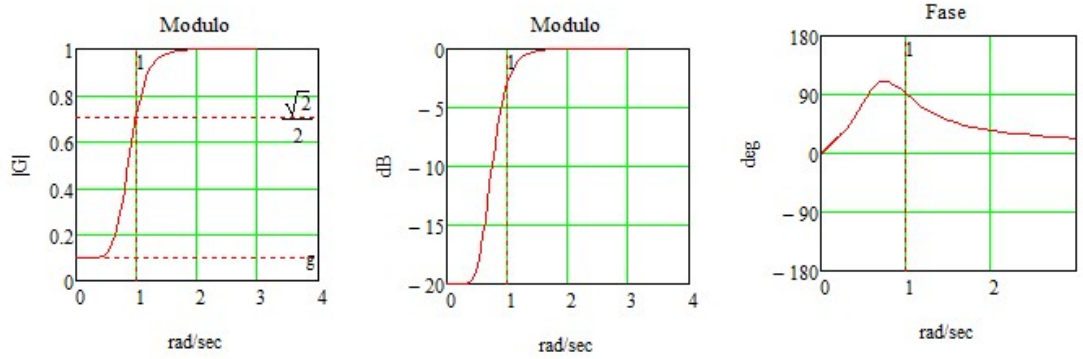


Figura 1.2: Grafico di un filtro Shelving con M=4 e g=1/10

È importante notare che sebbene si stia utilizzando la notazione *passa-** tipica dei filtri, gli *Shelving* non possiedono una banda attenuata come è definita formalmente, si limitano ad amplificare la banda che verrà denotata *banda passante* e lasciare inalterata la banda che verrà denotata *banda attenuata.*

Il modulo del filtro *Shelving* espresso come modulo della trasformata di *Fourier* è il seguente:

$$\left|H_{Sh}(j\,\omega)\right| = \sqrt{\frac{\omega^{2\cdot M} + g^2}{\omega^{2\cdot M} + 1}}$$

Equazione 1.3: Modulo di un filtro Shelving [b2]

1.2 *Shelving* a coefficienti reali

Sfruttando la notazione *di Eulero,* per l'esponenziale complesso, è possibile rendere i coefficienti della funzione di trasferimento reali nel caso in cui l'ordine sia pari:

$$H_{Sh.p}(s,M,g) := \prod_{m=1}^{\frac{M}{2}} \frac{s^2 + 2\cdot\cos(\alpha(m,M))\cdot\sqrt[M]{g}\cdot s + \left(\sqrt[M]{g}\right)^2}{s^2 + 2\cdot\cos(\alpha(m,M))\cdot s + 1}$$

Equazione 1.4: Shelving a coefficienti reali con M pari [b2]

Nel caso in cui l'ordine sia dispari è necessario passare per la scomposizione in frazioni parziali dell'Equazione 1.4:

$$c_m(m,M) := \cos(\alpha(m,M)) \qquad V(M,g) := \sqrt[M]{g} - 1$$

$$H_{Sh.p}(s,M,g) := \prod_{m=1}^{\frac{M}{2}} \left(1 + 2\cdot V(M,g)\cdot\frac{1 + c_m(m,M)\cdot s}{s^2 + 2\cdot c_m(m,M)\cdot s + 1} + \frac{V(M,g)^2}{s^2 + 2\cdot c_m(m,M)\cdot s + 1}\right)$$

Equazione 1.5 : Shelving con M pari scomposto in fratti semplici [b2]

Assumendo M dispari sarà presente un polo reale in $s = -1$ che portato fuori dalla produttoria permette di arrivare alla funzione di trasferimento nel caso di ordine dispari:

$$H_{Sh.d}(s,M,g) := \left(1 + V(M,g)\cdot\frac{1}{s+1}\right)\cdot\prod_{m=1}^{\frac{M-1}{2}} \left(1 + 2\cdot V(M,g)\cdot\frac{1 + c_m(m,M)\cdot s}{s^2 + 2\cdot c_m(m,M)\cdot s + 1} + \frac{V(M,g)^2}{s^2 + 2\cdot c_m(m,M)\cdot s + 1}\right)$$

Equazione 1.6: Shelving a coefficienti reali con M dispari [b2]

1.3 Trasformazione passa-banda

La trasformazione passa-banda nel dominio analogico è data da:

$$s_{BP}(s, \omega_{.o}, BW) := \frac{s^2 + \omega_{.o}^2}{s \cdot BW}$$

Equazione 1.7:Trasformazione passa-banda analogica [b1]

Dove ω_0 è la pulsazione di centro banda e BW è la larghezza della banda passante.

Si definisce dunque il filtro *Shelving* passa-banda come:

$$H_{BP}(s, M, g, \omega_o, BW) := H_{Sh}\left(\frac{s^2 + \omega_o^2}{s \cdot BW}, M, g\right)$$

Equazione 1.8: Shelving passa-banda

Dove H_{Sh} si riferisce all'Equazione 1.2.

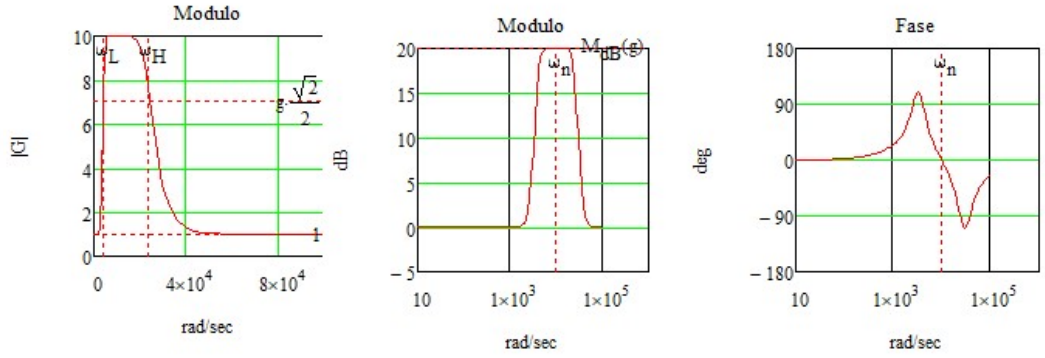


Figura 1.3: Shelving passa-banda con g=10

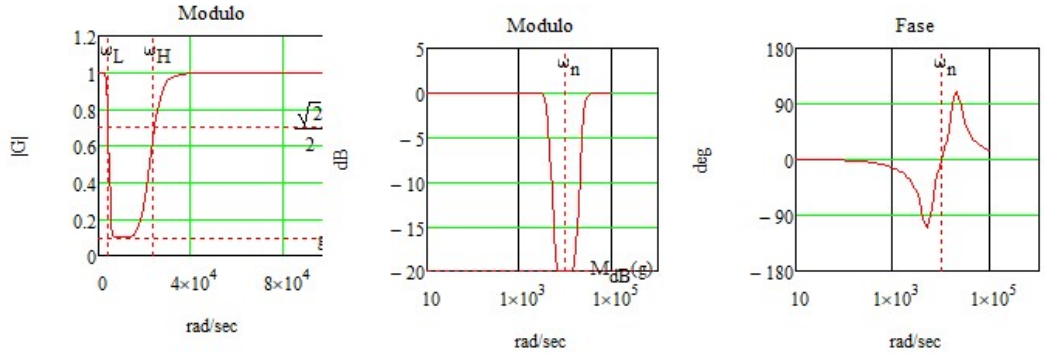


Figura 1.4: Shelving passa-banda con g=0.1

Capitolo 2

Modello di un equalizzatore grafico

2.1 Considerazioni generali

In un equalizzatore occorre posizionare i filtri in modo da renderli centrati sulle bande di interesse ed il più possibile complementari tra loro. Si noti che in un filtro passa-banda che taglia alle frequenze f_L e f_H, la frequenza centrale è data dalla media geometrica delle due frequenze o delle due pulsazioni:

$$\omega_o(\omega_L, \omega_H) := \sqrt{\omega_L \cdot \omega_H}$$

Equazione 2.1: Pulsazione di centro banda passante in funzione delle pulsazioni di taglio [b3]

La larghezza della banda passante è la differenza tra le due pulsazioni:

$$BW = \omega_H - \omega_L$$

Equazione 2.2: Larghezza della banda passante

Affinché le bande passanti siano complementari, deve valere il vincolo:

$$\omega_{L_{j+1}} = \omega_{H_j} \qquad j = 1,\ldots,\text{Numero di filtri}$$

Equazione 2.3: Pulsazioni di taglio complementari

Un equalizzatore grafico per applicazioni audio necessita che le bande siano distribuite in modo logaritmico, con un passo che dipende dalla risoluzione R scelta:

$$\omega_{L_{j+1}} = R \cdot \omega_{L_j} \qquad \omega_{o_{j+1}} = R \cdot \omega_{o_j} \qquad \text{con} \qquad R = 2 \quad \text{per equalizzatore ad ottave e} \qquad R = \sqrt[3]{2} \quad \text{per equalizzatore a terzi di ottava}$$

Equazione 2.4: Distanza in successione logaritmica tra le bande [b3]

Da cui derivano le specifiche delle pulsazioni di ogni filtro:

$$\omega_{H_j} = R \cdot \omega_{L_j} \qquad \omega_{H_j} = \sqrt{R} \cdot \omega_{o_j} \qquad \omega_{L_j} = \frac{1}{\sqrt{R}} \cdot \omega_{o_j}$$

Equazione 2.5: Specifiche delle pulsazioni del j-esimo filtro

Definendo il vettore ω_0 delle pulsazioni centrali delle bande e la risoluzione R si può, tramite l'Equazione 2.5, calcolare le specifiche di ogni filtro dell'equalizzatore:

$$BW = \left(\sqrt{R} - \frac{1}{\sqrt{R}}\right) \cdot \omega_o \qquad \omega_L = \frac{BW}{R - 1} \qquad \omega_H = \frac{BW}{1 - \frac{1}{R}}$$

Equazione 2.6: Vettori di specifiche dei filtri

Un ulteriore vincolo necessario a permettere la complementarietà dei filtri è che nei punti di incrocio i filtri adiacenti presentino lo stesso guadagno, come verrà illustrato nel successivo paragrafo.

2.2 Punti di incrocio nei filtri *Shelving*

I filtri *Shelving* lasciano inalterata la banda attenuata e amplificano la banda passante, ne consegue che il banco di filtri sarà costituito da una produttoria di filtri *Shelving* passa-banda. Per soddisfare il vincolo del guadagno uguale nei punti di incrocio sarà quindi sufficiente che il guadagno in tali punti sia uguale alla radice quadrata di g in modo che il prodotto sia g*:*

$$\left|H_{HP}\left(j\,\omega_{U_j}\right)\right| = \left|H_{HP}\left(j\,\omega_{L_j}\right)\right| = \sqrt{g}$$

$$\left|H_{HP}\left(j\,\omega_{U_{j+1}}\right)\right| = \left|H_{HP}\left(j\,\omega_{L_j}\right)\right| = \sqrt{g}$$

Equazione 2.7: Vincolo sul guadagno nei punti di incrocio [b2]

Chiamando ω_{cross} la pulsazione di taglio del filtro passa-basso normalizzato e inserendo il vincolo nell'Equazione 1.3 si ottiene:

$$\left|H_{HP}\left(j\,\omega_{cross}\right)\right| = \sqrt{g} \quad <=> \quad \frac{\omega_{cross}^{2\cdot M} + g^2}{\omega_{cross}^{2\cdot M} + 1} = g \quad <=> \quad g = \omega_{cross}^{2\cdot M}$$

Equazione 2.8: Guadagno nei punti di incrocio

$$M := 4 \qquad g := 10 \qquad \omega_{cross} := g^{\frac{1}{2 \cdot M}} = 1.334$$

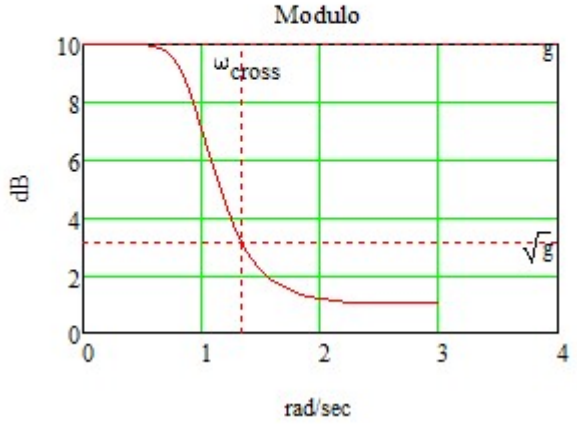


Figura 2.1: Punto di incrocio dei filtri Shelving

Dove si può notare che la ω_{cross} è stata calcolata mediante la funzione inversa dell'Equazione 2.8.

2.3 Esempio di cascata di due filtri *Shelving*

Una cascata di due filtri *Shelving* opportuni è un semplice modello di equalizzatore grafico a due bande.

Innanzitutto, è necessario scegliere il guadagno in banda passante e l'ordine dei filtri, che a fine esemplificativo vengono scelti rispettivamente pari a $g = 10$ e $M = 4$. Utilizzando una risoluzione ad un'ottava ($R = 2$) e pulsazione centrale della prima banda pari a $\omega_{0,1} = 1000\text{rad/sec}$, si può procedere al calcolo della seconda pulsazione centrale $\omega_{0,2}$ tramite l'Equazione 2.4 e delle larghezze delle bande tramite l'Equazione 2.6:

$$\omega_{o.1} := 1000 \qquad R := 2 \qquad \omega_{o.2} := R \cdot \omega_{o.1} = 2 \times 10^3$$

$$BW_n(M, g, R) := \left(\sqrt{R} - \frac{1}{\sqrt{R}}\right) \cdot \frac{1}{\omega_{cross}(g, M)}$$

$$BW_{\omega.1} := BW_n(M, g, R) \cdot \omega_{o.1} = 530.255$$

$$BW_{\omega.2} := BW_n(M, g, R) \cdot \omega_{o.2} = 1.061 \times 10^3$$

Equazione 2.9: Specifiche per un esempio di equalizzatore a due bande

È ora possibile applicare la trasformazione passa-banda e rappresentare graficamente i moduli dei due filtri, separatamente:

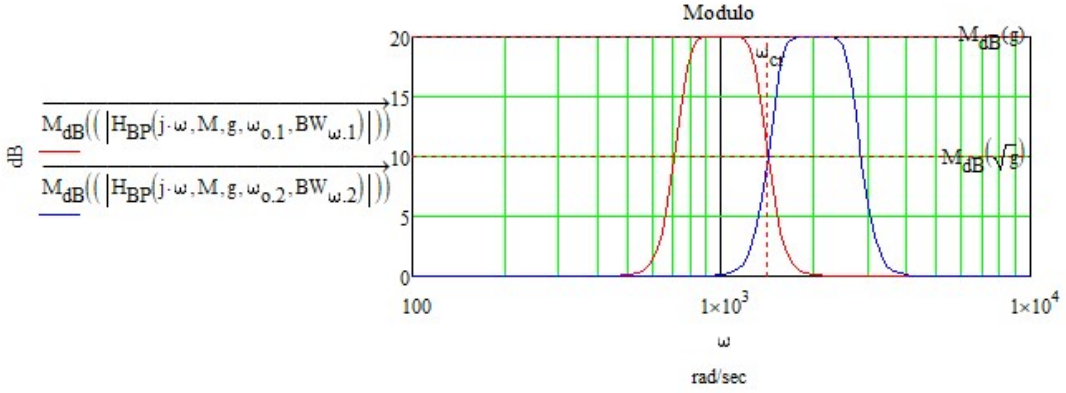


Figura 2.2: Moduli di un equalizzatore grafico Shelving a due bande in configurazione di massima amplificazione

Il punto di incrocio ha modulo $\sqrt{g}$ e pulsazione ωc pari alla media geometrica tra le due pulsazioni di centro banda, come da specifica.

2.4 Equalizzatore grafico analogico *Shelving* a 10 bande

2.4.1 Progetto dell'equalizzatore

Dalle precedenti considerazioni risulta intuitivo il modello di un equalizzatore grafico analogico *Shelving* a $N = 10$ bande. Scelta la risoluzione a un'ottava ($R = 2$) e la frequenza centrale della prima banda $f_{ini} = 30\text{Hz}$ è possibile determinare la successione delle frequenze centrali f_c in uno spazio logaritmico.

Il primo passaggio è quello di trovare la frequenza centrale dell'ultima banda f_{max}, ricorrendo all'Equazione 2.4:

$$f_{max} := f_{min} \cdot R^{N-1} = 15360$$

Equazione 2.10: Calcolo della frequenza centrale dell'ultima banda

Potrebbe risultare utile la funzione inversa che ritorna il numero di bande in funzione di R, f_{min}, f_{max}:

$$N := \text{floor}\left(\frac{\log(f_{max}) - \log(f_{min})}{\log(R)} + 1\right) = 10$$

Equazione 2.11: Calcolo del numero di bande

Per questioni di comodità verrà ora definita la funzione *LnSpace* che prende in input gli estremi di un intervallo $[a, b]$ ed un numero intero positivo N e ritorna un vettore N punti equidistanti in uno spazio logaritmico:

$$\text{LnSpace}(a,b,N) := \left| \begin{array}{l} \text{for } i \in 0..N-1 \\ \quad x_i \leftarrow \exp\left(\ln(a) + \ln\left(\frac{b}{a}\right)\cdot\frac{i}{N-1}\right) \\ x \end{array} \right.$$

Algoritmo 2.1: Spazio logaritmico

La successione delle frequenze di centro banda f_c sarà data da:

$$f_c := \text{round}\left(\text{LnSpace}\left(f_{min}, f_{max}, N\right)\right) \qquad \omega_c := 2\cdot\pi\cdot f_c$$

$$f_c{}^T = (30 \;\; 60 \;\; 120 \;\; 240 \;\; 480 \;\; 960 \;\; 1920 \;\; 3840 \;\; 7680 \;\; 15360)$$

$$\omega_c{}^T = (188 \;\; 377 \;\; 754 \;\; 1508 \;\; 3016 \;\; 6032 \;\; 12064 \;\; 24127 \;\; 48255 \;\; 96510)$$

Equazione 2.12: Frequenza [Hz] e pulsazioni [rad/sec] di centro banda dell'equalizzatore

Prima di poter determinare le larghezze delle bande sarà necessario definire il guadagno, ovvero la massima amplificazione, nel centro banda, che verrà posto pari a 12dB per ottenere una buona complementarietà nei filtri:

$$g_{dB} := 12 \qquad g := 10^{\frac{g_{dB}}{20}} = 3.981$$

Equazione 2.13: Massima amplificazione delle bande

Per ottenere una buona simmetria la massima attenuazione viene posta pari a -12dB:

$$g_{cut} = 10^{\frac{-g_{dB}}{20}} \quad \text{da cui} \qquad g_{cut} := \frac{1}{g} \qquad g_{cut} = 0.251$$

Equazione 2.14: Massima attenuazione delle bande

Può risultare rappresentativo il calcolo delle larghezze di banda, che viene effettuato mediante l'Equazione 2.9:

$$BW_\omega := BW_n(M, g, R) \cdot \omega_c$$

$$BW_\omega^T = (112 \quad 224 \quad 449 \quad 897 \quad 1794 \quad 3589 \quad 7177 \quad 14355 \quad 28710 \quad 57419)$$

$$BW_\omega^T = (18 \quad 36 \quad 71 \quad 143 \quad 286 \quad 571 \quad 1142 \quad 2285 \quad 4569 \quad 9139) \cdot 2 \cdot \pi$$

Equazione 2.15: Larghezze di banda

Nell'ultima riga è stato raccolto $2/\pi$ per mettere in evidenza le bande in Hertz.

Come visto in precedenza, il banco di filtri *Shelving* è dato dalla produttoria di N filtri a cui è stata applicata la trasformazione passa-banda, la funzione di trasferimento dell'equalizzatore sarà dunque:

$$H_{eq}(s, M, \omega_c, BW_\omega, G) := \prod_{j=0}^{N-1} H_{BP}(s, M, G_j, \omega_{c_j}, BW_{\omega_j})$$

Equazione 2.16: Funzione di trasferimento di un equalizzatore a filtri Shelving

Dove l'elemento *j-esimo* di G è l'amplificazione scelta per il filtro *j-esimo.*

2.4.2 Verifica dei risultati

Per prima cosa verrà verificata la condizione di incrocio. Gli estremi delle bande passanti si possono calcolare come segue:

$$\omega_H := \overrightarrow{\left(\frac{1}{2} \cdot BW_\omega + \frac{1}{2} \cdot \sqrt{BW_\omega^2 + 4 \cdot \omega_c^2}\right)} \qquad f_H := \frac{\omega_H}{2 \cdot \pi}$$

$$\omega_L := \overrightarrow{\left(-\frac{1}{2} \cdot BW_\omega + \frac{1}{2} \cdot \sqrt{BW_\omega^2 + 4 \cdot \omega_c^2}\right)} \qquad f_L := \frac{\omega_L}{2 \cdot \pi}$$

$$f_L^T = (22 \quad 45 \quad 89 \quad 179 \quad 358 \quad 716 \quad 1432 \quad 2864 \quad 5728 \quad 11456)$$

$$f_H^T = (40 \quad 80 \quad 161 \quad 322 \quad 644 \quad 1287 \quad 2574 \quad 5149 \quad 10297 \quad 20594)$$

Equazione 2.17: Frequenze di taglio [Hz]

Denotando con sel_1 la configurazione con tutti i cursori *flat*, ad eccezione del primo che viene posto ad amplificazione massima, e con sel_{All} la configurazione con tutti i cursori ad amplificazione massima, si ha che il guadagno nel punto di incrocio tra la prima e la seconda banda è:

$$\left|H_{eq}\left(j\cdot\sqrt{\omega_{H_1}\cdot\omega_{L_2}},M,\omega_c,BW_\omega,sel_1\right)\right| = 1.995 \qquad \sqrt{g} = 1.995$$

$$\left|H_{eq}\left(j\cdot\sqrt{\omega_{H_1}\cdot\omega_{L_2}},M,\omega_c,BW_\omega,Sel_{All}\right)\right| = 3.982 \qquad g = 3.981$$

Equazione 2.18: Guadagno nel primo punto di incrocio

Si nota un leggero discostamento tra il valore effettivo del modulo nel punto di incrocio e il valore g desiderato, ciò è dovuto all'interferenza tra le code dei filtri più distanti e cresce all'aumentare del numero N di bande scelto. In ogni caso l'errore è trascurabile per applicazioni audio.

Nella seguente figura è rappresentato ogni filtro a sé stante, l'equalizzatore è il prodotto dei filtri.

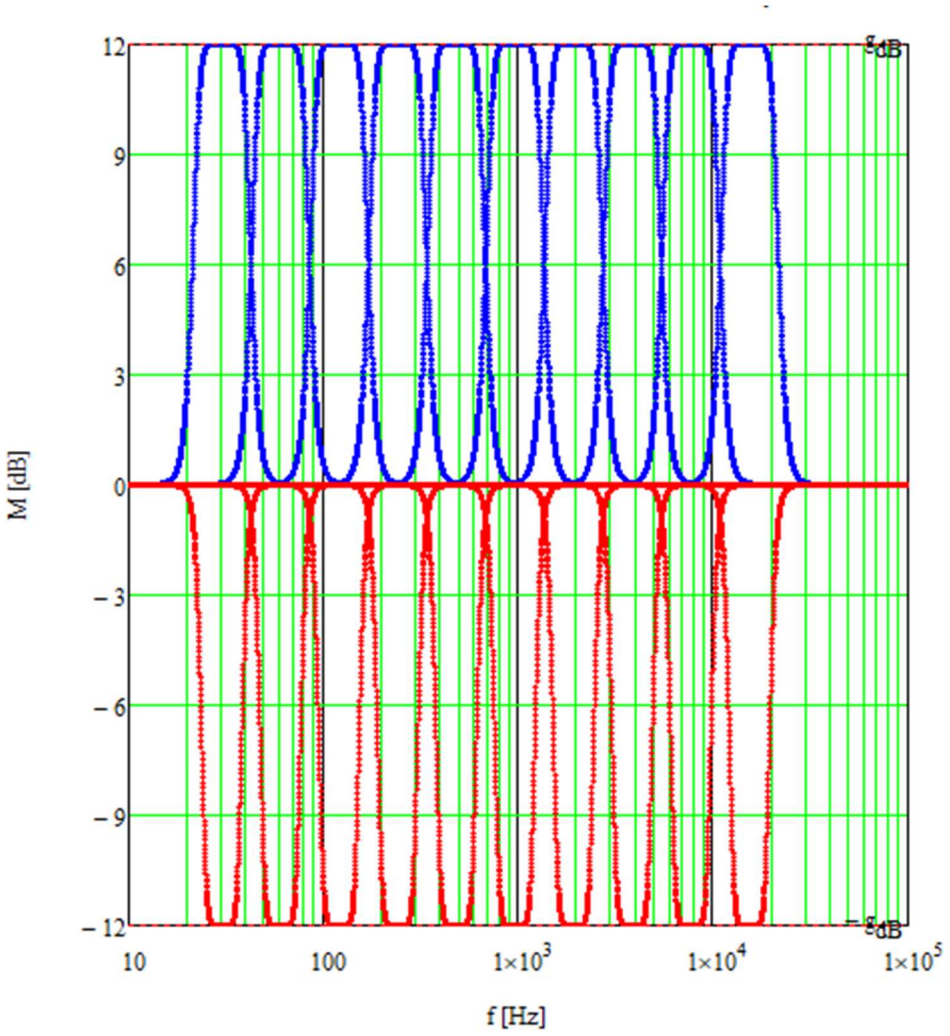


Figura 2.3: Configurazione a cursori ad amplificazione massima, in blu e cursori ad attenuazione massima, in rosso

Nelle successive figure viene rappresentata la funzione di trasferimento in varie configurazioni dei cursori dell'equalizzatore:

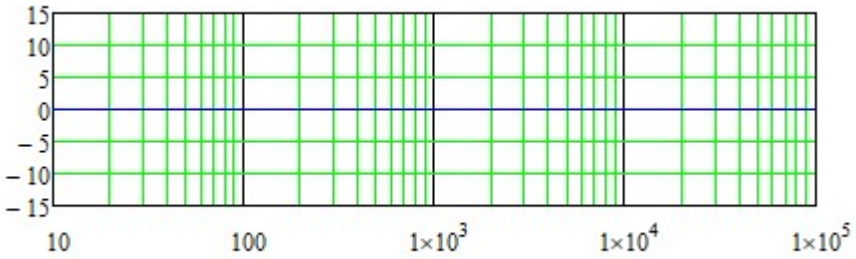


Figura 2.4: Configurazione a cursori flat

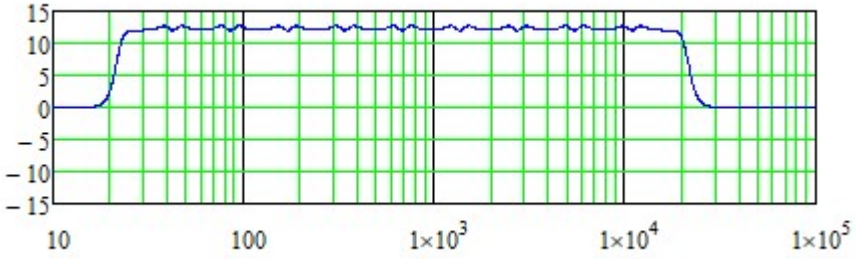


Figura 2.5: Configurazione a cursori alzati

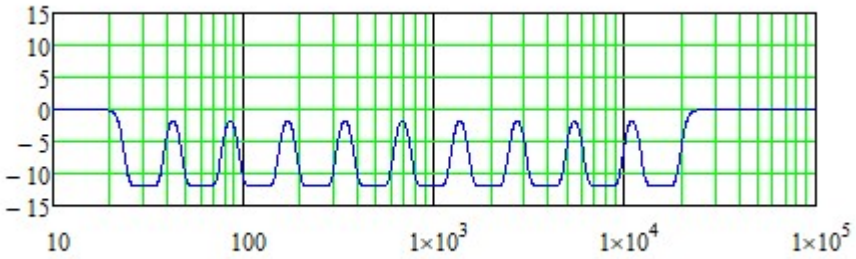


Figura 2.6: Configurazione a cursori abbassati

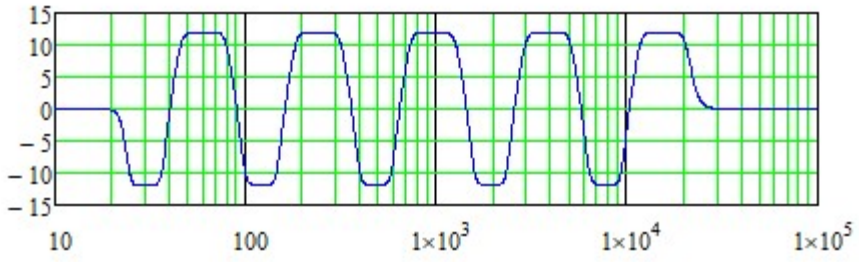


Figura 2.7: Configurazione a cursori alterni

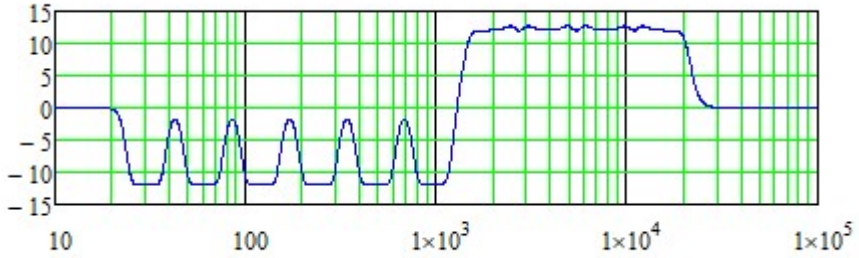


Figura 2.8: Configurazione a 5 cursori abbassati e 5 alzati

Si nota che i filtri *Shelving* presentano una certa asimmetria tra amplificazione e attenuazione: la risposta in frequenza dell'equalizzatore, in amplificazione, presenta un *ripple* trascurabile tuttavia, in attenuazione, è piuttosto marcato. Per ottenere la stessa qualità in amplificazione ed attenuazione verrà usata la configurazione *cut-boost* che verrà trattata nel relativo capitolo.

Capitolo 3

Discretizzazione del filtro *Shelving*

Per passare nel dominio discreto verrà utilizzata la *trasformazione bilineare*:

$$s(z,T) = \frac{2}{T}\cdot\left(\frac{z-1}{z+1}\right)$$

Equazione 3.1: Trasformazione bilineare [b1]

Tale trasformazione introduce un'inevitabile deformazione tra la maschera del filtro continuo e quella del filtro discreto che lo approssima. È possibile rendere esatta l'approssimazione in un punto della maschera adottando la tecnica del *prewarping*:

$$s\left(z,T_{pw}\right) = \frac{2}{T_{pw}}\cdot\left(\frac{z-1}{z+1}\right) \quad \text{con} \quad T_{pw}\left(\omega_{pw},T\right) := \frac{2}{\omega_{pw}}\cdot\tan\left(\frac{\omega_{pw}}{2}\cdot T\right)$$

Equazione 3.2: Trasformazione bilineare con prewarping [b1]

3.1 Scelta del prewarping per filtri *Shelving*

Il *prewarping* permette di eliminare esattamente la discrepanza tra funzione di trasferimento continua e discreta in un solo punto. Nella seguente figura è mostrato il confronto tra le funzioni di trasferimento H_s: la funzione di trasferimento di uno *Shelving passa-banda e* H_{z_bil}: la sua trasformazione bilineare ottenuta dall'Equazione 3.1, senza *prewarping,* con una frequenza di campionamento pari a 44100Hz*:*

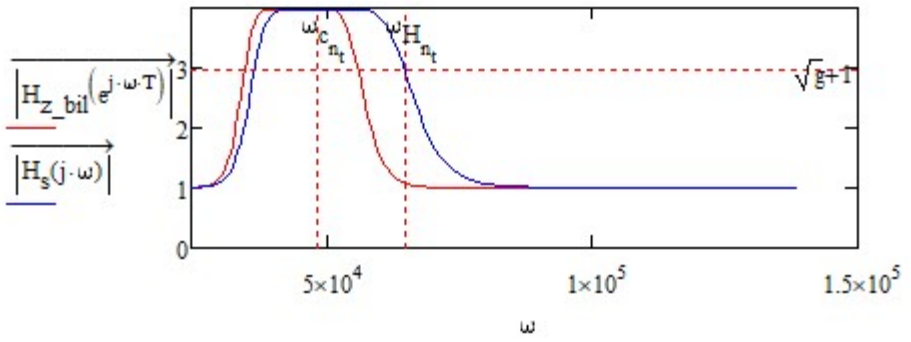


Figura 3.1: Confronto tra funzione di trasferimento analogica e la sua discretizzazione senza prewarping

Si nota che la banda passante discreta è asimmetrica rispetto al centro banda passante analogico, un primo miglioramento si ottiene correggendo il centro banda col *prewarping,* come si nota dalla figura:

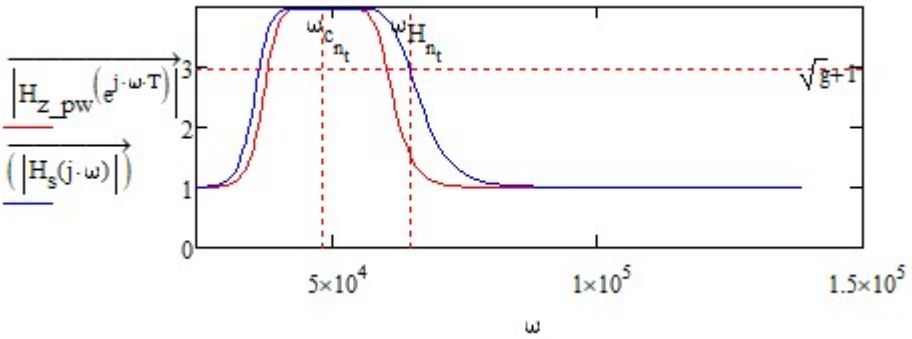


Figura 3.2: Confronto tra funzione di trasferimento analogica e la sua discretizzazione con prewarping

Dove H_{z_pw} è stata ottenuta utilizzando l'Equazione 3.2 calcolata nel punto centrale della banda passante analogica.

Utilizzando la distorsione dell'Equazione 3.2 è possibile correggere la discretizzazione in un solo punto, ma le ipotesi fatte in precedenza sul banco di filtri *Shelving* richiedono che le specifiche siano rispettate anche nei punti di incrocio oltre che al centro banda passante, ne consegue che neanche il *prewarping* dell'Equazione 3.2 è efficace. Per poter correggere l'errore nelle frequenze di taglio è necessaria una distorsione ad-hoc che tenga conto di tutti e tre i punti di interesse. Verrà quindi eseguita una mappatura di tutte le specifiche nel dominio analogico, in modo che la distorsione della trasformazione bilineare ritorni le frequenze desiderate, nel discreto.

Per ottenere la distorsione introdotta dalla trasformazione bilineare è necessario partire dal legame tra una funzione di trasferimento analogica H_s e la sua versione discretizzata H_z*:*

$$H_z\left(e^{j\cdot\omega\cdot T}\right) = H_s\left(\frac{2}{T}\cdot\frac{e^{j\cdot\omega\cdot T}-1}{e^{j\cdot\omega\cdot T}+1}\right) = H_s\left(\frac{2}{T}\cdot\frac{e^{j\cdot\omega\cdot\frac{T}{2}}-e^{-j\cdot\omega\cdot\frac{T}{2}}}{e^{j\cdot\omega\cdot\frac{T}{2}}+e^{-j\cdot\omega\cdot\frac{T}{2}}}\right) = H_s\left(\frac{2}{T}\cdot\frac{j\cdot\sin\left(\omega\cdot\frac{T}{2}\right)}{\cos\left(\omega\cdot\frac{T}{2}\right)}\right) = H_s\left(j\cdot\frac{2}{T}\cdot\tan\left(\omega\cdot\frac{T}{2}\right)\right)$$

In altri termini risulta che $H_z\left(e^{j\cdot\omega_z\cdot T}\right) = H_s\left(j\cdot\omega_s\right)$ con $\omega_s = \frac{2}{T}\cdot\tan\left(\frac{\omega_z}{2}\cdot T\right)$ $\omega_z = \frac{2}{T}\cdot\operatorname{atan}\left(\frac{T}{2}\cdot\omega_s\right)$

Equazione 3.3: Legame tra f.d.t. continua e discreta

Da cui si definiscono le funzioni di mappatura diretta e inversa delle pulsazioni, distorte dalla trasformazione bilineare:

$$\omega_{s2z}\left(\omega_s, T\right) := \frac{2}{T}\cdot\operatorname{atan}\left(\frac{\omega_s}{2}\cdot T\right) \quad \omega_{z2s}\left(\omega_z, T\right) := \frac{2}{T}\cdot\tan\left(\frac{\omega_z}{2}\cdot T\right)$$

Equazione 3.4: Mappatura delle pulsazioni continuo/discreto

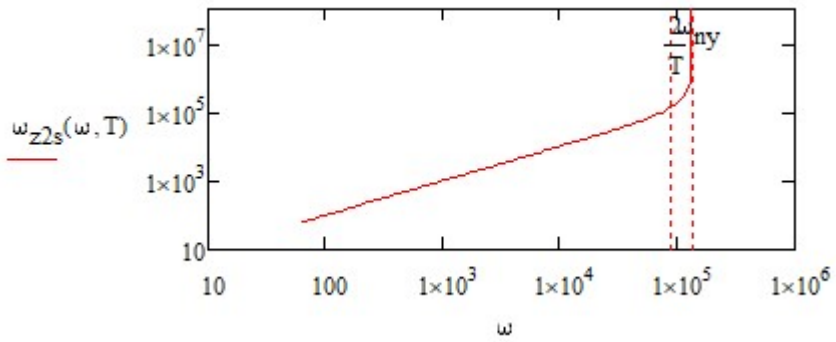


Figura 3.3: Rappresentazione grafica della mappatura

Avendo a disposizioni le funzioni dell'Equazione 3.4 è possibile pre-distorcere le tre specifiche del filtro *Shelving:*

$$\omega_{Lpw} := \overrightarrow{\omega_{z2s}\left(\omega_L, T\right)} \quad \omega_{Hpw} := \overrightarrow{\omega_{z2s}\left(\omega_H, T\right)} \quad \omega_{cpw} := \overrightarrow{\sqrt{\omega_{Hpw}\cdot\omega_{Lpw}}}$$

Equazione 3.5: Specifiche pre-distorte

Modificando le pulsazioni nella funzione di trasferimento analogica con quelle ottenute dalla funzione di mappatura, sarà sufficiente la trasformazione bilineare semplice per il rispetto totale delle specifiche nella funzione di trasferimento a tempo discreto:

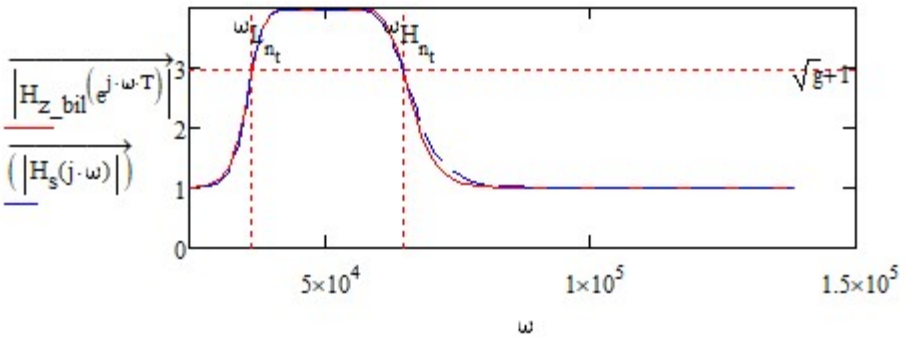


Figura 3.4: Confronto tra funzione di trasferimento analogica e la sua discretizzazione con prewarping di tutte e tre le specifiche

È possibile osservare che quest'ultima metodologia di discretizzazione porta a un risultato nettamente migliore delle due precedenti, da questo momento in poi quando ci si riferirà al filtro *Shelving* digitale o discreto verrà sottinteso questo tipo di trasformazione.

3.2 Distorsione in prossimità della frequenza di Nyquist

Come si nota dalla Figura 3.3, in prossimità della frequenza di Nyquist la distorsione diventa decisamente marcata e nell'equalizzatore la frequenza di taglio superiore dell'ultima banda si avvicina molto a tale frequenza. Utilizzando le funzioni di mappatura dell'Equazione 3.4 si osserva che le frequenze nell'intorno inferiore della frequenza di Nyquist vengono mappate al di sopra di essa, ma ciò non preclude il fatto di poter ottenere un filtro discreto che rispetti esattamente le frequenze che sono state pre-distorte.

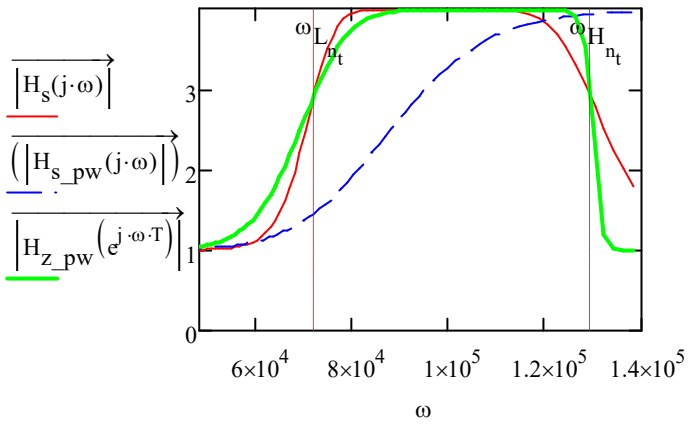


Figura 3.5: In rosso la risposta in frequenza target, analogica, in blu la risposta in frequenza analogica pre-distorta, in verde la sua discretizzazione vicino alla frequenza di Nyquist

Si nota che le specifiche vengono rispettate e che la distorsione ha contributo trascurabile.

3.3 Equalizzatore Shelving con *prewarping* delle bande

Con i filtri discretizzati in questo capitolo è già possibile modellizzare un equalizzatore grafico *Shelving*, tuttavia non presenterebbe la possibilità di cambiare i parametri *online*. Nel successivo capitolo sarà implementata la forma parametrica che permetterà di mettere in evidenza i parametri nel calcolo dei coefficienti portando ad una soluzione più generale che si presta sia all'implementazione di equalizzatori grafici sia parametrici. Non verranno per cui studiati e descritti i modelli implementativi dell'equalizzatore fatto con il semplice *prewarping* delle bande. Tale equalizzatore viene presentato al solo scopo di poter confrontare i risultati con quello parametrico descritto successivamente.

Si definisce la funzione di trasferimento discreta passa-banda dello *Shelving* pre-distorta come trasformazione bilineare della funzione di trasferimento passa-banda analogica, calcolata nelle pulsazioni pre-distorte e nelle larghezze di banda pre-distorte:

$$Hz_{BPpw}(z, M, g, \omega_{cpw}, BW_{\omega pw}) := H_{BP}\left[\frac{2}{T}\cdot\left(\frac{z-1}{z+1}\right), M, g, \omega_{cpw}, BW_{\omega pw}\right]$$

Equazione 3.6: Funzione di trasferimento passa-banda discreta con prewarping

Per il calcolo delle pulsazioni di centro banda e delle larghezze di banda pre-distorte si utilizza l'Equazione 3.5 per distorcere le pulsazioni di taglio e successivamente se ne calcola la loro media geometrica e la loro differenza.

L'equalizzatore discreto, come nel caso analogico, diventa la cascata dei filtri Hz_{BPpw}:

$$Hz_{eq.pw}(z, M, \omega_{cpw}, BW_{\omega pw}, G) := \prod_{j=0}^{rows(\omega_c)-1} Hz_{BPpw}(z, M, G_j, \omega_{cpw_j}, BW_{\omega pw_j})$$

Equazione 3.7: Equalizzatore discreto con prewarping

Nel successivo capitolo verrà sviluppata l'implementazione parametrica che porterà agli stessi risultati e verranno forniti esempi di risposte in frequenza con possibili configurazioni dei cursori.

Capitolo 4

Implementazione discreta parametrica

Un approccio più generale è dato dalla *forma parametrica* grazie alla quale è possibile ricavare la funzione di trasferimento dell'equalizzatore completo in funzione dei suoi parametri. Questa permetterà di semplificare l'implementazione e di costruire equalizzatori parametrici.

Per la costruzione di questo tipo di equalizzatore si lavorerà direttamente nel *digitale* usando le trasformazioni in frequenza discrete.

4.1 Cella del filtro *Shelving* unitario discreto in forma polinomiale fratta

Come visto in precedenza la trasformazione bilineare del filtro *Shelving* è data da una serie di $M/2$ celle del secondo ordine. Il filtro *Shelving* unitario discreto verrà utilizzato come prototipo per la costruzione dei filtri che compongono l'equalizzatore parametrico.

$$Hz_{Sh}(z, M, g, T) := \prod_{m=1}^{\frac{M}{2}} \left(Hz_{Sh\_m}\left(z, V(M, g), c_m(m, M), T\right)\right)$$

Equazione 4.1: Filtro Shelving discreto

Hz_{Sh_m} si ottiene applicando la trasformazione bilineare senza *prewarping* alla funzione di trasferimento continua della cella.

Esplicitando il risultato, ogni cella del secondo ordine discreta assume la forma:

$$Hz_{Sh\_m}\left(z, V, c_m, T\right) = 1 + 2\cdot V\cdot T\cdot\frac{\left(T + 2\cdot c_m\right) + 2\cdot T\cdot z^{-1} + \left(T - 2\cdot c_m\right)\cdot z^{-2}}{\left(4 + 4\cdot c_m\cdot T + T^2\right) + \left(2\cdot T^2 - 8\right)\cdot z^{-1} + \left(4 - 4\cdot c_m\cdot T + T^2\right)\cdot z^{-2}} \ldots$$
$$+ V^2\cdot T^2\cdot\frac{1 + 2\cdot z^{-1} + z^{-2}}{\left(4 + 4\cdot c_m\cdot T + T^2\right) + \left(2\cdot T^2 - 8\right)\cdot z^{-1} + \left(4 - 4\cdot c_m\cdot T + T^2\right)\cdot z^{-2}}$$

Equazione 4.2: Cella del secondo ordine discreta

Notando che il denominatore degli ultimi due addendi è lo stesso, è possibile separare numeratore e denominatore per l'implementazione del filtro in forma polinomiale fratta a variabile z:

$$\mathrm{den}(z, V, c_m, T) = \left(4 + 4 \cdot c_m \cdot T + T^2\right) + \left(2 \cdot T^2 - 8\right) \cdot z^{-1} + \left(4 - 4 \cdot c_m \cdot T + T^2\right) \cdot z^{-2}$$

$$\mathrm{num}(z, V, c_m, T) = 1 + T \cdot V \cdot \left[(V + 2) \cdot T - 4 \cdot c_m\right] \cdot z^{-2} + 2 \cdot T^2 \cdot V \cdot (V + 2) \cdot z^{-1} + T \cdot V \cdot \left[(V + 2) \cdot T + 4 \cdot c_m\right]$$

Equazione 4.3: Denominatore e numeratore cella Shelving discreta

I coefficienti del numeratore vengono chiamati b_i e i coefficienti del denominatore a_j, è inoltre presente un termine di resto pari a 1, il cui scopo è intuitivamente lasciar passare le altre frequenze della serie, a cui viene attribuito il nome d.

Riassumendo, si possono calcolare i coefficienti polinomiali mediante la seguente funzione:

$$\mathrm{Hz}_{\mathrm{Sh\_m.coeff}}(V, c_m, T) := \left| \begin{array}{l} a \leftarrow \begin{pmatrix} 4 + 4 \cdot c_m \cdot T + T^2 \\ 2 \cdot T^2 - 8 \\ 4 - 4 \cdot c_m \cdot T + T^2 \end{pmatrix} \\ b \leftarrow V \cdot T \cdot \begin{pmatrix} 2 \cdot T + 4 \cdot c_m + V \cdot T \\ 4 \cdot T + 2 \cdot V \cdot T \\ 2 \cdot T - 4 \cdot c_m + V \cdot T \end{pmatrix} \\ d \leftarrow 1 \\ (b \quad a \quad d) \end{array} \right.$$

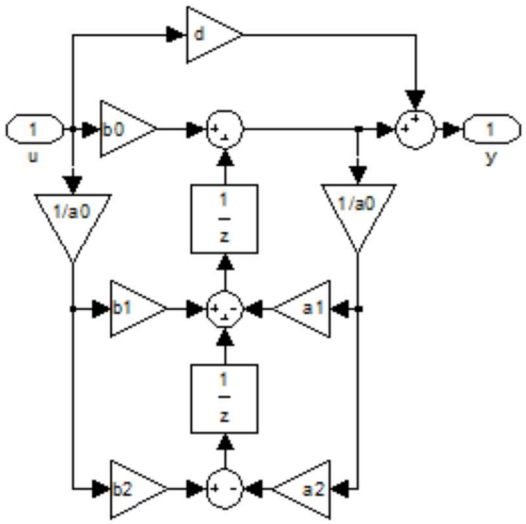
Algoritmo 4.1: Coefficienti di una cella del filtro Shelving unitario e discreto

La funzione di trasferimento della cella di secondo ordine è esprimibile come:

$$\mathrm{Hz}_{\mathrm{Sh\_m\_impl}}(z, V, c_m, T) := \left| \begin{array}{l} (b \quad a \quad d) \leftarrow \mathrm{Hz}_{\mathrm{Sh\_m.coeff}}(V, c_m, T) \\ H \leftarrow \dfrac{\sum_{i=0}^{\mathrm{rows}(b)-1} \left(b_i \cdot z^{-i}\right)}{\sum_{i=0}^{\mathrm{rows}(a)-1} \left(a_i \cdot z^{-i}\right)} + d \\ H \end{array} \right.$$

Algoritmo 4.2: Funzione di trasferimento della cella di secondo ordine discreta

Usando i soli coefficienti polinomiali una forma implementativa, per le celle de secondo ordine, è la *II trasposta* con l'aggiunta del termine di resto d:



Algoritmo 4.3: Forma II trasposta con resto [b1]

Per la discretizzazione dello *Shelving* unitario non si è utilizzato il *prewarping* perché si trova in bassa frequenza, dove la distorsione è bassissima, e alcuni test eseguiti hanno dimostrato che non porta migliorie apprezzabili.

4.2 Trasformazione delle frequenze discrete

L'equalizzatore parametrico necessita di spostare alle frequenze desiderate i filtri *Shelving* prototipi, per attuare questa trasformazione, nel discreto, una possibile soluzione è l'utilizzo della *trasformazione passa-banda discreta* che si trova in letteratura (O*ppenheim Schafer*):

$$z_{BP}(z, \omega_{ini}, \omega_1, \omega_2, T) := \left| \begin{array}{l} \alpha \leftarrow \dfrac{\cos\left(\dfrac{\omega_1 + \omega_2}{2} \cdot T\right)}{\cos\left(\dfrac{\omega_2 - \omega_1}{2} \cdot T\right)} \\ \beta \leftarrow \cot\left(\dfrac{\omega_2 - \omega_1}{2} \cdot T\right) \cdot \tan\left(\dfrac{\omega_{ini}}{2} \cdot T\right) \\ \left(-\dfrac{\dfrac{\beta - 1}{1 + \beta} - \dfrac{2 \cdot \alpha \cdot \beta}{1 + \beta} \cdot z^{-1} + z^{-2}}{1 - \dfrac{2 \cdot \alpha \cdot \beta}{1 + \beta} \cdot z^{-1} + \dfrac{\beta - 1}{1 + \beta} \cdot z^{-2}} \right)^{-1} \end{array} \right.$$

Algoritmo 4.4: Trasformazione passa-banda discreta [b1]

Dove ω_{ini} è la pulsazione di *taglio* del filtro prototipo, ω_1 e ω_2 le pulsazioni di taglio obiettivo.

La prima considerazione da fare a proposito dell'algoritmo di trasformazione passa-banda è che sposta nel dominio discreto le pulsazioni di taglio in modo esatto, come se fosse stato applicato un *prewarping* nel dominio analogico, motivo per cui se si usasse questa trasformazione per come è fornita non sarebbe necessario eseguire nessun tipo di *prewarping*. In secondo luogo, tale trasformazione prende in ingresso le frequenze di taglio, mentre in questo contesto si preferisce lavorare con i centri banda e le larghezze di banda. Viene ridefinito l'algoritmo di trasformazione calcolando ω_1 e ω_2 in funzione di ω_o e BW:

$$z_{BP1}(z, \omega_{ini}, \omega_o, BW, T) := \left| \begin{array}{l} \omega_1 \leftarrow \frac{\sqrt{BW^2 + 4 \cdot \omega_o^{\,2}}}{2} - \frac{BW}{2} \\ \omega_2 \leftarrow \omega_1 + BW \\ \alpha \leftarrow \frac{\cos\left(\frac{\omega_1 + \omega_2}{2} \cdot T\right)}{\cos\left(\frac{\omega_2 - \omega_1}{2} \cdot T\right)} \\ \beta \leftarrow \cot\left(\frac{\omega_2 - \omega_1}{2} \cdot T\right) \cdot \tan\left(\frac{\omega_{ini}}{2} \cdot T\right) \\ \left(- \frac{\frac{\beta - 1}{1 + \beta} - \frac{2 \cdot \alpha \cdot \beta}{1 + \beta} \cdot z^{-1} + z^{-2}}{1 - \frac{2 \cdot \alpha \cdot \beta}{1 + \beta} \cdot z^{-1} + \frac{\beta - 1}{1 + \beta} \cdot z^{-2}} \right)^{-1} \end{array} \right.$$

Algoritmo 4.5: Trasformazione passa-banda discreta in funzione di ω_o e di BW, con media geometrica

Al fine di semplificare la formula vengono fatte due considerazioni:

- ω_{ini} nel prototipo unitario è sempre pari a 1,
- non è necessario usare questa trasformazione per cambiare la larghezza di banda in quanto è possibile modificarla tramite una frequenza di campionamento *fittizia*.

Per quanto riguarda il secondo punto, scegliere una frequenza di campionamento *fittizia* porta ad allargare la maschera del filtro prototipo mediante ricampionamento e successivamente a traslare in frequenza la maschera già scalata. Intuitivamente per eseguire questa trasformazione basta moltiplicare il tempo di campionamento per la larghezza di banda desiderata:

$$T_{BW}(T, BW) := T \cdot BW$$

Equazione 4.4: Cambio scala del tempo di campionamento, per scegliere la banda passante

Tuttavia, questo cambiamento di scala porterebbe alla perdita della perfetta traslazione del *prewarping* implicito nella trasformazione passa-banda, è pertanto opportuno eseguire la pre-distorsione del tempo di campionamento usata nella trasformazione bilineare con

prewarping (Equazione 3.2), arrivando alla corretta trasformazione del tempo di campionamento:

$$T_{BWpw}(T, BW) := T_{pw}(BW, T) \cdot BW$$

Equazione 4.5: Cambio scala del tempo di campionamento con prewarping, per scegliere la banda passante

È ora possibile calcolare la trasformazione passa-banda unitaria, che si ottiene sostituendo $\omega_{ini} = 1, BW = 1$ nell'Algoritmo 4.5 e semplificando il risultato:

$$z_{BPu}(z, \omega_o, T) := \frac{z^{-1} \cdot \cos\left(\frac{T \cdot \sqrt{4 \cdot \omega_o{}^2 + 1}}{2}\right) - \cos\left(\frac{T}{2}\right)}{z^{-2} \cdot \cos\left(\frac{T}{2}\right) - z^{-1} \cdot \cos\left(\frac{T \cdot \sqrt{4 \cdot \omega_o{}^2 + 1}}{2}\right)}$$

Equazione 4.6: Trasformazione passa-banda unitaria

Per *sampling rate* alti alcuni test hanno dimostrato che è possibile utilizzare l'espansione in serie di *Taylor* troncata al primo ordine e approssimare $\cos(T/2) \approx 1$ ma per mantenere generalità non verrà eseguita tale approssimazione nella trattazione.

Applicando la trasformazione passa-banda unitaria e la trasformazione al tempo di campionamento al filtro prototipo si ricava la funzione di trasferimento dello *Shelving* passa-banda discreto, in funzione di tutti i parametri utili:

$$Hz_{BP}(z, M, g, T, \omega_o, BW) := Hz_{Sh}(z_{BPu}(z, \omega_o, T), M, g, T_{BW}(T, BW))$$

Equazione 4.7: Shelving passa-banda discreto

4.3 Equalizzatore *Shelving* discreto

L'equalizzatore parametrico discreto è realizzabile come cascata di filtri passa-banda. Un equalizzatore a N bande avrà dunque la funzione di trasferimento:

$$Hz_{eq}(z, \omega_c, BW, G, T) := \prod_{j=0}^{N-1} Hz_{BP}(z, M, G_j, T, \omega_{c_j}, BW_j)$$

Equazione 4.8: Equalizzatore parametrico Shelving discreto a N bande

Nella successiva sezione verrà illustrato come implementarlo in modo algoritmico

4.4 Parametrizzazione online delle celle e implementazione

La metodologia usata precedentemente per il progetto di filtri può essere usata direttamente in fase di filtraggio modificando la forma *II trasposta* in modo da traslare il filtro prototipo a piacimento senza dover riprogettare il banco di filtri. A tale scopo nello schema implementativo della cella del secondo ordine verrà sostituita alla variabile *z* l'Equazione 4.6:

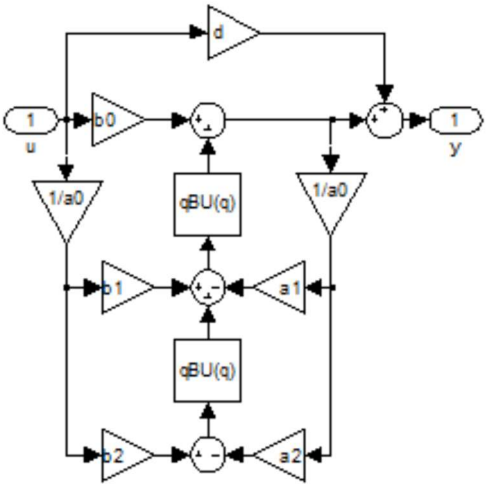


Figura 4.1: Forma II trasposta con traslazione della banda [b4]

Per semplificare la rappresentazione si è passati da z (anticipo unitario) a $q = 1/z$ (ritardo unitario) ottenendo la formula:

$$q_{BPu}(q, \omega_o, T) := \frac{q^2 \cdot \cos\left(\frac{T}{2}\right) - q \cdot \cos\left(\frac{T \cdot \sqrt{4 \cdot \omega_o{}^2 + 1}}{2}\right)}{q \cdot \cos\left(\frac{T \cdot \sqrt{4 \cdot \omega_o{}^2 + 1}}{2}\right) - \cos\left(\frac{T}{2}\right)}$$

Equazione 4.9: Trasformazione passa-banda unitaria in funzione di q=1/z

Chiamando c lo zero della trasformazione passa-banda in q, la trasformazione può essere scritta mettendo in evidenza zeri e poli:

$$c = \frac{\cos\left(\frac{T\cdot\sqrt{4\cdot\omega_o{}^2 + 1}}{2}\right)}{\cos\left(\frac{T}{2}\right)}$$

$$z_{BPu}(z) = \frac{c\cdot z - 1}{z\cdot(z - c)} \qquad q_{BPu}(q) = q\cdot\frac{c - q}{1 - c\cdot q}$$

Equazione 4.10: Zeri e poli della trasformazione passa-banda unitaria

È presente un polo in 0 ed uno in c che risulta essere minore di 1 per ω_0 inferiori alla pulsazione di *Nyquist,* il filtro è stabile e può essere implementato in forma *II trasposta.*

In definitiva, lo step di calcolo iterativo della Figura 4.1 può essere implementata mediante il seguente algoritmo, in cui si sono usati i coefficienti della cella del secondo ordine trovati in precedenza ed è presente inoltre l'implementazione del filtro z_{BPu}:

$$Hz_{Sh\_m\_step1}\left(x_a, x_b, u, \omega_o, V, c_m, T, BW\right) :=$$

$$T_{BW} \leftarrow T \cdot BW$$

$$a \leftarrow \begin{pmatrix} 4 + 4 \cdot c_m \cdot T_{BW} + T_{BW}^{2} \\ 2 \cdot T_{BW}^{2} - 8 \\ 4 - 4 \cdot c_m \cdot T_{BW} + T_{BW}^{2} \end{pmatrix}$$

$$b \leftarrow V \cdot T_{BW} \cdot \begin{pmatrix} 2 \cdot T_{BW} + 4 \cdot c_m + V \cdot T_{BW} \\ 4 \cdot T_{BW} + 2 \cdot V \cdot T_{BW} \\ 2 \cdot T_{BW} - 4 \cdot c_m + V \cdot T_{BW} \end{pmatrix}$$

$$b \leftarrow \frac{b}{a_0}$$

$$a \leftarrow \frac{a}{a_0}$$

$$c \leftarrow \frac{\cos\left(\frac{T \cdot \sqrt{4 \cdot \omega_o^{2} + 1}}{2}\right)}{\cos\left(\frac{T}{2}\right)}$$

$$y_a \leftarrow x_{a_0}$$

$$y_b \leftarrow x_{b_0}$$

$$y_c \leftarrow y_a + b_0 \cdot u$$

$$u_a \leftarrow -a_1 \cdot y_c + b_1 \cdot u + y_b$$

$$u_b \leftarrow -a_2 \cdot y_c + b_2 \cdot u$$

$$x_{a_0} \leftarrow c \cdot x_{a_0} - x_{a_1} + c \cdot u_a$$

$$x_{a_1} \leftarrow u_a$$

$$x_{b_0} \leftarrow c \cdot x_{b_0} - x_{b_1} + c \cdot u_b$$

$$x_{b_1} \leftarrow u_b$$

$$y \leftarrow y_c + u$$

$$\begin{pmatrix} x_a & x_b & y \end{pmatrix}$$

Algoritmo 4.6: Step di calcolo dell'implementazione parametrica

Si nota che non è stato utilizzato il *prewarping* del tempo di campionamento in quanto si intende passarlo alla funzione come argomento.

Nel caso di un equalizzatore grafico non sarà necessario calcolare i coefficienti del filtro b e a ad ogni iterazione in quanto sono costanti e possono essere calcolati in fase di progetto, *offline*.

Nel caso di equalizzatore parametrico, i coefficienti a e b si ricalcoleranno solo quando l'utente altererà le bande dei filtri.

4.5 Implementazione di un filtro *Shelving* passa-banda di ordine 2M in cascata

È stato illustrato come implementare le singole celle del filtro *Shelving* passa-banda e come il filtro *Shelving* sia costituito da una cascata di M/2 celle del secondo ordine. Scegliendo un valore di M con la forma dello *Shelving* a coefficienti reali di ordine pari si potrà implementare un filtro di ordine 2M usando l'implementazione parametrica.

Il seguente algoritmo implementa il progetto e il filtraggio con un filtro *Shelving* passa-banda parametrico di ordine 2M di un segnale u:

$\text{Filter}_{Sh1}(u, M, g, T, \omega c, BW) :=$

"Inizializzazione"

for $m \in 0..\frac{M}{2} - 1$

$x_{a_m} \leftarrow (0 \quad 0)^T$

$x_{b_m} \leftarrow (0 \quad 0)^T$

$c_m \leftarrow \cos\left[\pi \cdot \left[\frac{1}{2} - \frac{2\cdot(m+1) - 1}{2\cdot M}\right]\right]$

$V \leftarrow \sqrt[M]{g} - 1$

"Loop di calcoo della cascata di celle"

for $k \in 0..\text{rows}(u) - 1$

$y \leftarrow u_k$

for $m \in 0..\frac{M}{2} - 1$

$\left(x_{a_m} \quad x_{b_m} \quad y\right) \leftarrow \text{Hz}_{Sh\_m\_step1}\left(x_{a_m}, x_{b_m}, y, \omega c, V, c_m, T_{pw}(BW, T), BW\right)$

$y_{out_k} \leftarrow y$

y_{out}

Algoritmo 4.7: Implementazione parametrica di un filtro Shelving di ordine 2M

Per il calcolo dei coefficienti e l'aggiornamento dello stato viene usato $\text{Hz}_{Sh\_m\_step}$ (Algoritmo 4.6) a cui viene passato il tempo di campionamento opportunamente pre-distorto.

4.6 Filtraggio di un segnale di test

A titolo esemplificativo verrà usato l'Algoritmo 4.7 per filtrare un segnale *chirp*, è stato scelto un ordine del filtro pari a 8 ($M = 4$).

Viene in primo luogo definita la funzione *chirp*:

$$\text{chirp}(t, t_{end}, \omega_{ini}, \omega_{end}) := \sin\left[\left[\omega_{ini} + \frac{1}{2}\cdot\left(\frac{\omega_{end} - \omega_{ini}}{t_{end}}\right)\cdot t\right]\cdot t\right]$$

Equazione 4.11: Funzione chirp

Che viene usata per inizializzare un segnale di ingresso u_{chirp} di 1000 campioni, campionato a 44100Hz con pulsazioni che vanno da 0rad/s alla pulsazione di *Nyquist:*

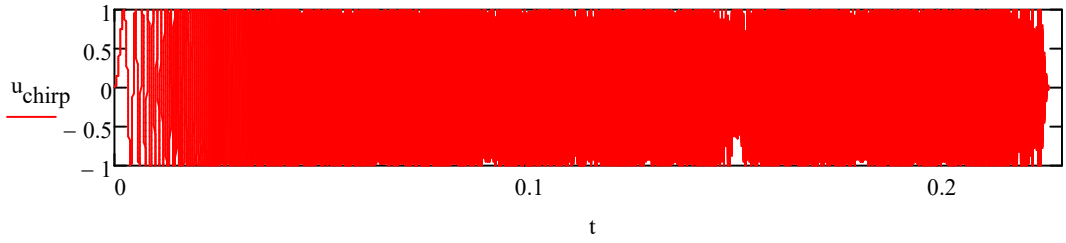


Figura 4.2: Rappresentazione di u_{chirp} nel tempo

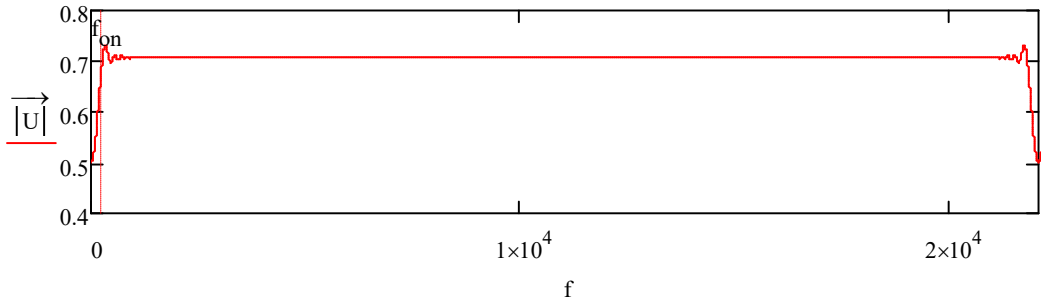


Figura 4.3: Rappresentazione del modulo di u_{chirp} in frequenza

Il segnale viene filtrato con $Filter_{Sh1}$ dove i parametri scelti sono $g_{dB} = 12\text{dB}, M = 4, T = \frac{1}{44100}\text{s}, \omega c = 2.487\text{E}4\frac{\text{rad}}{\text{s}}, BW = 2\text{E}4$ ottenendo il segnale y:

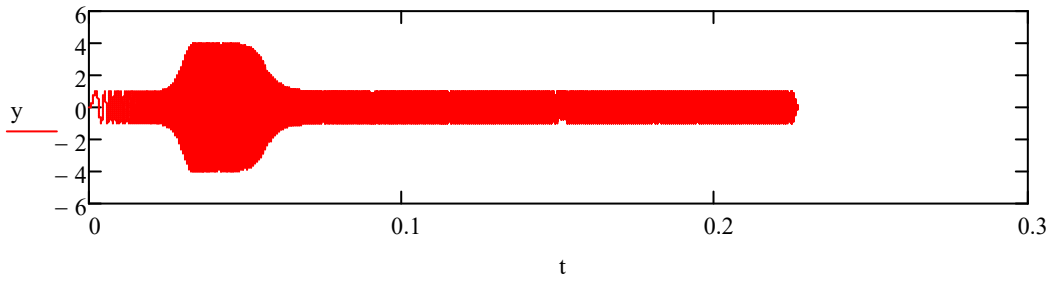


Figura 4.4: Rappresentazione di y nel tempo

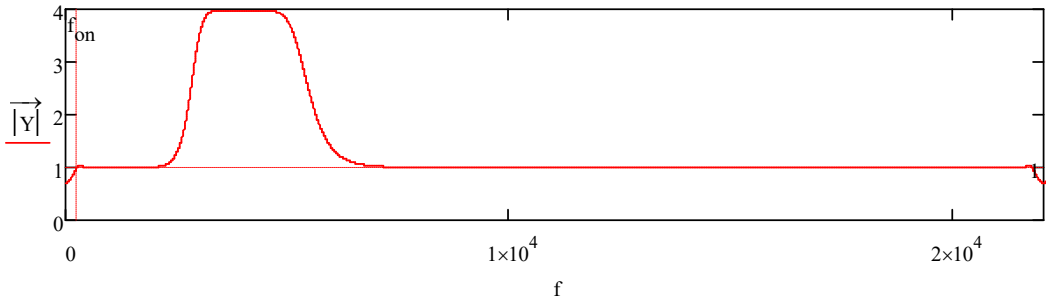


Figura 4.5: Rappresentazione del modulo di y in frequenza

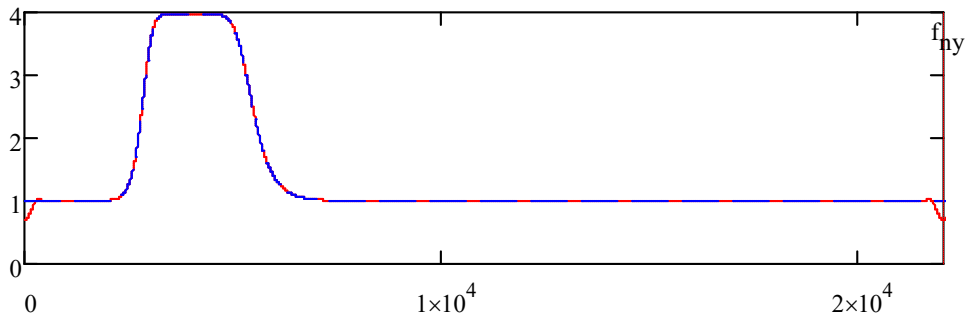


Figura 4.6: Confronto tra risposta in frequenza del filtro Hz_{BP} (in blu) e modulo di y in frequenza (in rosso)

Si nota che il *chirp* viene filtrato correttamente e che il modulo della trasformata di *Fourier* dell''uscita è congruente alla risposta in frequenza del filtro Hz_{BP} salvo per l'effetto ai bordi dovuto alla *DFT.*

4.7 Equalizzatore *Shelving* parametrico

L'equalizzatore, come visto in precedenza, è una cascata di filtri *Shelving* che sono a loro volta una cascata di celle del secondo ordine. Il seguente algoritmo implementa un loop di $Filter_{Sh1}$ mettendo in serie i filtri passa-banda al fine di equalizzare il segnale di ingresso u mediante il vettore di N cursori G:

$\text{Equalizer}_{Sh1}(u, G) :=$ "Inizializzazione"

$y_{out} \leftarrow u \cdot 0$

for $n \in 0..N-1$

for $m \in 0..\frac{M}{2}-1$

$x_{a_{n,m}} \leftarrow (0\ \ 0)^T$

$x_{b_{n,m}} \leftarrow (0\ \ 0)^T$

$c_m \leftarrow \cos\left[\pi \cdot \left[\frac{1}{2} - \frac{2 \cdot (m+1) - 1}{2 \cdot M}\right]\right]$

"Loop di calcolo della cascata di celle"

for $k \in 0..\text{rows}(u)-1$

$y \leftarrow u_k$

for $n \in 0..N-1$

$V \leftarrow \sqrt[M]{G_n} - 1$

for $m \in 0..\frac{M}{2}-1$

$\left(x_{a_{n,m}}\ \ x_{b_{n,m}}\ \ y\right) \leftarrow \text{Hz}_{Sh\_m\_step1}\left(x_{a_{n,m}}, x_{b_{n,m}}, y, \omega_{c_n}, V, c_m, T_{pw}\left(BW_{\omega_n}, T\right), BW_{\omega_n}\right)$

$y_{out_k} \leftarrow y$

y_{out}

Algoritmo 4.8: Implementazione di un equalizzatore Shelving parametrico

L'algoritmo chiama $\text{Hz}_{Sh\_m\_step1}$ (Algoritmo 4.6) passandogli iterativamente i valori delle matrici che contengono i parametri necessari, opportunamente definite a seconda del tipo di equalizzatore che si vuole ottenere.

Capitolo 5

Configurazione Cut-Boost

In precedenza, si è evidenziato che i filtri *Shelving* presentano una certa asimmetria tra l'amplificazione e l'attenuazione. In questo capitolo verrà implementata la configurazione *Cut-Boost* che permetterà di raggiungere la perfetta simmetria.

5.1 Funzione di trasferimento della configurazione Cut-Boost

La funzione di trasferimento della configurazione Cut-Boost è rappresentata nel seguente diagramma a blocchi:

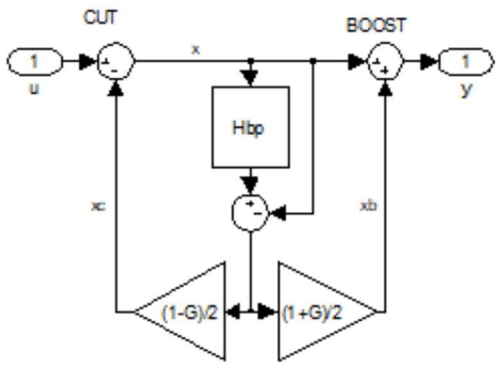


Figura 5.1: Configurazione Cut-Boost [b5]

Da cui sostituendo H_{bp} con la funzione di trasferimento del filtro passa-banda discreto *Shelving* Hz_{BP} si ottiene:

$$CB(z, G, M, g, T, \omega_c, BW) := \frac{(1 - G) + (1 + G)\cdot Hz_{BP}(z, M, g, T, \omega_c, BW)}{(1 + G) + (1 - G)\cdot Hz_{BP}(z, M, g, T, \omega_c, BW)}$$

Equazione 5.1: Funzione di trasferimento configurazione Cut-Boost [b5]

La configurazione *Cut-Boost* gode delle seguenti proprietà:

Boost	$\mathrm{CB}(z,1) = \mathrm{Hz}_{\mathrm{BP}}(z)$
Flat	$\mathrm{CB}(z,0) = 1$
Cut	$\mathrm{CB}(z,-1) = \frac{1}{\mathrm{Hz}_{\mathrm{BP}}(z)}$

Equazione 5.2: Proprietà Cut-Boost

sarà sufficiente scegliere valori di G compresi tra -1 e 1 per regolare l'amplificazione o l'attenuazione delle bande.

Questa configurazione, oltre a garantire la simmetricità dei filtri, semplifica l'utilizzo dell'equalizzatore.

La funzione di trasferimento di un equalizzatore *Shelving* in configurazione *Cut-Boost* a N bande è:

$$\mathrm{Hz}_{\mathrm{eq,CB}}(z, M, \omega_c, BW, G, T) := \prod_{j=0}^{N-1} \mathrm{CB}\left(z, G_j, M, g, T, \omega_{c_j}, BW_j\right)$$

Equazione 5.3: Funzione di trasferimento discreta di un equalizzatore Shelving in configurazione Cut-Boost

5.2 Forma implementativa per la configurazione Cut-Boost

Lo step di calcolo dell'equalizzatore in configurazione *Cut-Boost* è stato implementato in forma *ABCD (Rappresentazione di Stato)*, si parte quindi dalla funzione di calcolo dei coefficienti per la forma *ABCD*:

$$\text{MatCellSyntZ}\left(\omega_o, V, c_m, T, BW\right) :=$$

$$Tpw \leftarrow T_{pw}(BW, T)$$

$$T_{BW} \leftarrow Tpw \cdot BW$$

$$a0 \leftarrow 4 + 4 \cdot c_m \cdot T_{BW} + T_{BW}^{\ 2}$$

$$a \leftarrow \frac{1}{a0} \cdot \begin{pmatrix} 4 + 4 \cdot c_m \cdot T_{BW} + T_{BW}^{\ 2} \\ 2 \cdot T_{BW}^{\ 2} - 8 \\ 4 - 4 \cdot c_m \cdot T_{BW} + T_{BW}^{\ 2} \end{pmatrix}$$

$$b \leftarrow \frac{T_{BW} \cdot V}{a0} \cdot \left[\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \cdot T_{BW} \cdot V + \begin{pmatrix} 2 \cdot T_{BW} + 4 \cdot c_m \\ 4 \cdot T_{BW} \\ 2 \cdot T_{BW} - 4 \cdot c_m \end{pmatrix} \right]$$

$$d \leftarrow 1$$

$$c \leftarrow \frac{\cos\left(\frac{Tpw \cdot \sqrt{4 \cdot \omega_o^{\ 2} + 1}}{2} \right)}{\cos\left(\frac{T}{2} \right)}$$

$$Az \leftarrow \begin{pmatrix} -c \cdot a_1 + c & -1 & c & 0 \\ -a_1 & 0 & 1 & 0 \\ -c \cdot a_2 & 0 & c & -1 \\ -a_2 & 0 & 0 & 0 \end{pmatrix}$$

$$Bz \leftarrow \begin{bmatrix} \left(c \cdot b_1 - c \cdot a_1 \cdot b_0 \right) \\ \left(-a_1 \right) \cdot b_0 + b_1 \\ (-c) \cdot a_2 \cdot b_0 + c \cdot b_2 \\ \left(-a_2 \right) \cdot b_0 + b_2 \end{bmatrix}$$

$$Cz \leftarrow (1 \quad 0 \quad 0 \quad 0)$$

$$Dz \leftarrow b_0 + d$$

$$\begin{pmatrix} Az & Bz \\ Cz & Dz \end{pmatrix}$$

Algoritmo 5.1: Aggiornamento dello stato e calcolo dei coefficienti per la forma ABCD

In definitiva lo step di calcolo risulta essere:

$$\mathrm{CB\_step}(X, u, G, V, c_m, T, \omega_o, BW) := \left| \begin{array}{l} \begin{pmatrix} Az & Bz \\ C & D \end{pmatrix} \leftarrow \mathrm{MatCellSyntZ}(\omega_o, V, c_m, T, BW) \\ \text{"Calcolo dell'uscita"} \\ \Delta \leftarrow (1 - G)\cdot D + (G + 1) \\ x \leftarrow \dfrac{(G - 1)\cdot C\cdot X + 2\cdot u}{\Delta} \\ y \leftarrow \dfrac{2\cdot G\cdot C\cdot X + [(G + 1)\cdot D + (1 - G)]\cdot u}{\Delta} \\ \text{"Aggiornamento dello stato"} \\ X \leftarrow Az\cdot X + Bz\cdot x \\ (X \quad y) \end{array} \right.$$

Algoritmo 5.2: Step di calcolo per la configurazione Cut-Boost

5.3 Verifica dei risultati per la configurazione *Cut-Boost*

Nel seguente grafico si riportano i valori del modulo in Decibel delle singole bande in configurazione a cursori alzati, *post* e cursori abbassati, *cut*:

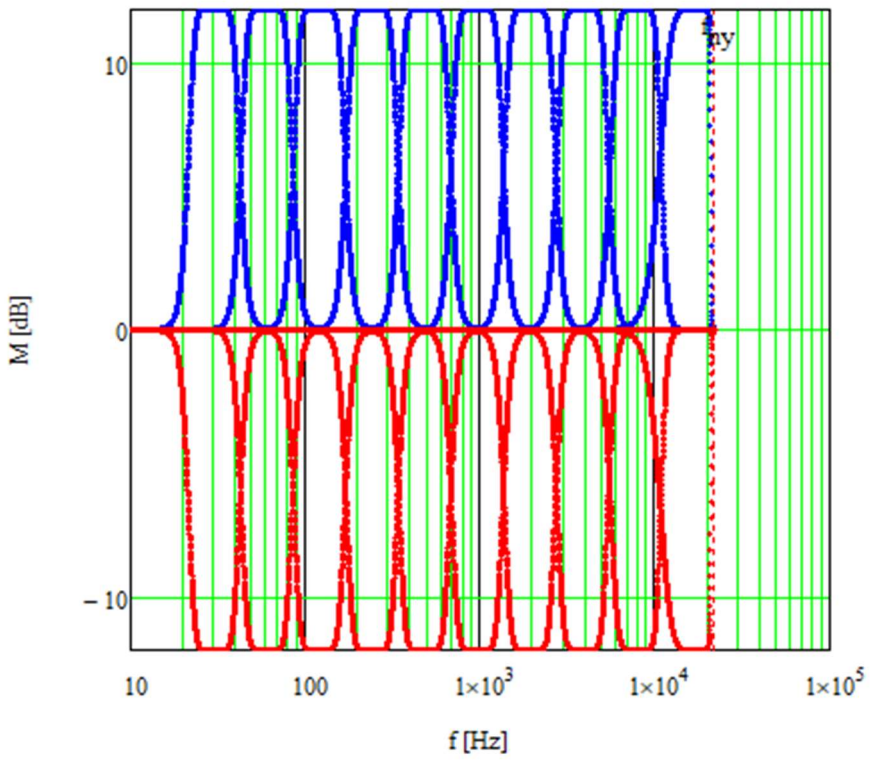


Figura 5.2: Rappresentazione delle singole bande in configurazione Cut-Boost

Di seguito invece vengono riportati il modulo della serie nelle medesime configurazioni:

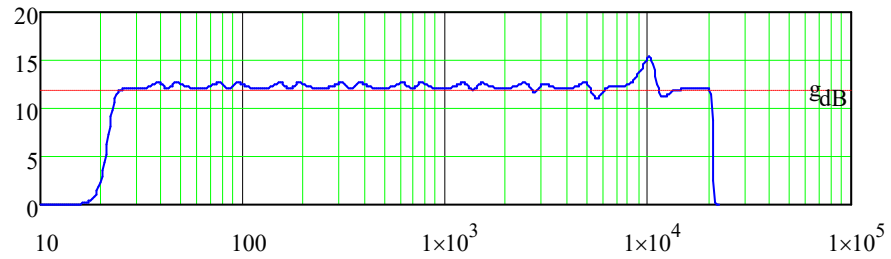


Figura 5.3: Modulo in decibel della configurazione boost

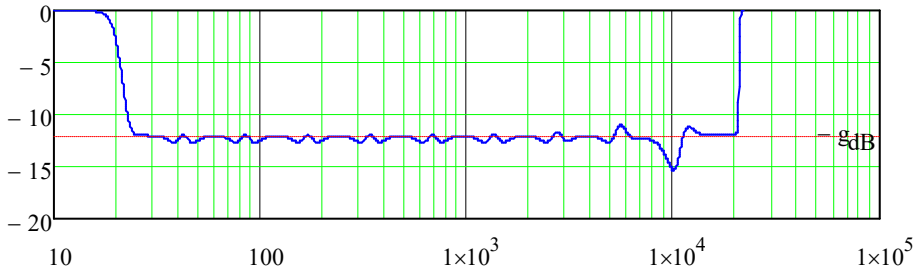


Figura 5.4: Modulo in decibel della configurazione cut

Si nota la perfetta simmetria, tuttavia, in prossimità della frequenza di *Nyquist* è presente un ripple dovuto alla deformazione delle maschere degli *Shelving* passa-banda che il *prewarping* non può migliorare ulteriormente visto che le specifiche di centro banda e larghezza di banda sono rispettate.

Sono stati considerati tre metodi per eliminare il *ripple ad alta frequenza*:

- Rimuovere il *prewarping*
- Alterare le specifiche dei filtri con algoritmi di ottimizzazione al fine di minimizzare il ripple *[b6]*
- *Oversampling*

Rimuovendo il *prewarping* il *ripple* si riduce in prossimità della frequenza di *Nyquist,* in quanto le specifiche analogiche tendono a contrarsi su frequenze sensibilmente minori, tuttavia il rispetto delle specifiche di centro banda e larghezza di banda è considerato di primaria priorità, per cui questo approccio risulta non conforme ai requisiti dell'equalizzatore.

Modificando le specifiche dei filtri con algoritmi di ottimizzazione si riesce a ridurre notevolmente il *ripple,* senza scostarsi sensibilmente dalle specifiche analogiche, tuttavia questo metodo risulta costoso dal punto di vista computazionale e non è adatto al design *real-time* in cui il valore di amplificazione delle bande, o le specifiche in caso di equalizzatore parametrico, possono variare rapidamente in ogni istante.

L'*Oversampling* riduce il *ripple* in modo efficace ma risulta costoso dal punto di vista computazionale, per cui è preferibile non usarlo quando il carico di lavoro è fatto su *CPU non specializzate per il Signal Processing e* la frequenza di campionamento è alta.

D'altra parte, per alte frequenze di campionamento (44100Hz o superiori) e per uso *real-time,* in ambito audio, il *ripple* non risulta essere un difetto percepibile dall'orecchio e può essere lasciato inalterato. Per altri tipi di applicazioni uno o più delle precedenti soluzioni possono essere prese in considerazione.

Capitolo 6

Implementazione ottimizzata per filtri di ordine 8

6.1 Pseudocodice

L'algoritmo della configurazione *Cut-Boost* è scritto in forma *ABCD* per cui sono presenti varie moltiplicazioni per 0, la forma matriciale risulta più complicata da implementare in molti linguaggi di programmazione, infine molte operazioni possono essere semplificate algebricamente o raccolte, riducendo il numero totale di operazioni da svolgere e quindi anche il costo computazionale.

Il seguente algoritmo è stato ottenuto semplificando il più possibile i calcoli svolti dagli algoritmi precedenti per un ordine dei filtri pari a 8 ovvero imponendo $M = 4$:

$$
\begin{aligned}
&\text{FilterSetup}_{CB4}(\text{Specs}) := \left|
\begin{array}{l}
\left(M \;\; g \;\; T \;\; \omega_c \;\; BW_\omega\right) \leftarrow \text{Specs}^T \\
N \leftarrow \text{rows}\left(\omega_c\right) \\
\text{"Precalcolo di tutti i filtri"} \\
V \leftarrow \sqrt[M]{g} - 1 \\
\text{for } n \in 0..N-1 \\
\quad \left|
\begin{array}{l}
Tpw \leftarrow T_{pw}\left(BW_{\omega_n}, T\right) \\
c\omega \leftarrow \dfrac{\cos\left[\dfrac{Tpw}{2}\cdot\sqrt{4\cdot\left(\omega_{c_n}\right)^2+1}\right]}{\cos\left(\dfrac{Tpw}{2}\right)} \\
Tbw \leftarrow Tpw\cdot BW_{\omega_n} \\
\text{for } m \in 0..\dfrac{M}{2}-1 \\
\quad \left|
\begin{array}{l}
c\_m \leftarrow \cos\left[\pi\cdot\left(0.5-\dfrac{m+0.5}{M}\right)\right] \\
a_0 \leftarrow 4+4\cdot c\_m\cdot Tbw+Tbw^2 \\
\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \leftarrow \dfrac{1}{a_0}\cdot\begin{pmatrix} 2\cdot Tbw^2-8 \\ a_0-8\cdot c\_m\cdot Tbw \end{pmatrix} \\
\begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \leftarrow \dfrac{Tbw\cdot V}{a_0}\cdot\begin{bmatrix} 2\cdot Tbw+4\cdot c\_m+Tbw\cdot V \\ 2\cdot Tbw\cdot(V+2) \\ 2\cdot Tbw-4\cdot c\_m+Tbw\cdot V \end{bmatrix} \\
d \leftarrow b_0+1 \\
c_m \leftarrow \begin{bmatrix} c\omega\cdot\left(1-a_1\right) \\ c\omega \\ c\omega\cdot\left(b_1-a_1\cdot b_0\right) \\ b_1-a_1\cdot b_0 \\ -a_1 \\ c\omega\cdot\left(b_2-a_2\cdot b_0\right) \\ -a_2\cdot c\omega \\ \left(b_2-a_2\cdot b_0\right) \\ -a_2 \\ d \end{bmatrix}
\end{array}
\right. \\
D_S \leftarrow \left(c_0\right)_9\cdot\left(c_1\right)_9 \\
\text{coeff}^{\langle n\rangle} \leftarrow \text{stack}\left(c_0, c_1, D_S\right)
\end{array}
\right. \\
\text{coeff}
\end{array}
\right.
\end{aligned}
$$

Algoritmo 6.1: Calcolo dei coefficienti ottimizzato per M=4

$\text{Step}_{CB4}(u_e, X_e, G_e, c_e) :=$

"Ingresso dell'equalizzatore"

$y \leftarrow u_e$

$N \leftarrow \text{cols}(X_e)$

for $n \in 0..N-1$

"Estrazione componenti del filtro n-esimo"

$X \leftarrow X_e^{\langle n \rangle}$

$G \leftarrow G_{e_n}$

$c \leftarrow c_e^{\langle n \rangle}$

"Gestione Cut Boost"

$GG1 \leftarrow G + 1$

$GG2 \leftarrow G - 1$

$\Delta \leftarrow GG1 - GG2 c_{20}$

$y1 \leftarrow X_4 + c_{19} \cdot X_0$

$u0 \leftarrow \frac{GG2 y1 + 2 \cdot y}{\Delta}$

$u1 \leftarrow X_0 + c_9 \cdot u0$

$y \leftarrow \frac{2 \cdot G \cdot y1 + (GG1 \cdot c_{20} - GG2) \cdot y}{\Delta}$

"Filtraggio BP"

$X2C1 \leftarrow X_2 \cdot c_1$

$X6C11 \leftarrow X_6 \cdot c_{11}$

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} \leftarrow \begin{pmatrix} X_0 \cdot c_0 - X_1 + X2C1 + c_2 \cdot u0 \\ X_0 \cdot c_4 + X_2 + c_3 \cdot u0 \\ X_0 \cdot c_6 + X2C1 - X_3 + u0 \cdot c_5 \\ X_0 \cdot c_8 + u0 \cdot c_7 \\ X_4 \cdot c_{10} - X_5 + X6C11 + c_{12} \cdot u1 \\ X_4 \cdot c_{14} + X_6 + c_{13} \cdot u1 \\ X_4 \cdot c_{16} + X6C11 - X_7 + c_{15} \cdot u1 \\ X_4 \cdot c_{18} + c_{17} \cdot u1 \end{pmatrix}$$

$X_e^{\langle n \rangle} \leftarrow X$

$(y \quad X_e)$

Algoritmo 6.2: Step di calcolo ottimizzato per M=4

6.1.1 Performance

Utilizzando l'algoritmo ottimizzato per $\mathrm{M} = 4$, il costo computazionale del calcolo dei coefficienti è trascurabile in quanto in un equalizzatore grafico sono calcolati una sola volta *offline,* in funzione del numero di bande N. È interessante notare che il numero di coefficienti totali è pari a $21 \cdot \mathrm{N}$.

Per quanto riguarda lo *step* di calcolo che viene effettuato *online,* per ogni campione, si hanno $\mathrm{N} \cdot (30\ prodotti\ reali\ +\ 26\ addizioni\ reali)$, per cui:

$$\text{Costo computazionale} \ = \ \mathrm{N} \cdot 56\ \frac{operazioni\ reali}{campione}$$

Equazione 6.1: Costo computazionale per M=4

6.2 Implementazione in C90

L'algoritmo per $\mathrm{M} = 4$ è stato implementato in *C90* in un semplice programma esemplificativo che utilizza la libreria *web* scritta dal professor *Pier Luca Montessoro*[s4] per filtrare un file *wav* il cui nome viene passato come argomento al programma, si veda l'appendice per il codice sorgente.

La definizione *N* impone il numero di bande da usare e il vettore *par* i valori del guadagno di ciascuna banda, a valori compresi tra *-1* e *1,* che a scopo di test sono state impostate in una configurazione oscillante tra *-1* e *1* che corrispondono rispettivamente a $-12\mathrm{dB}$ e $+12\mathrm{dB}$ come da definizione di *DB.*

La funzione *preprocessing()* calcola i coefficienti e la funzione *filt()* viene chiamata per ogni campione sinistro e destro del file, passandogli gli stati del filtro che sono rispettivamente *xL* e *xR.*

F_MIN è la frequenza centrale della prima banda che è stata posta a $30\mathrm{Hz}$.

Capitolo 7

Modulo equalizzatore per *PulseAudio*

PulseAudio è un *sound system* per sistemi operativi *POSIX*, appartenente al progetto *freedesktop.org* (che si impegna a semplificare l'utilizzo di *Linux* sui sistemi desktop ed è mantenuto da *Red Hat*), funge da *proxy* per l'audio delle applicazioni e incorpora diversi moduli e funzionalità di vario tipo tra cui filtraggi, rimozione del rumore, mixaggi, …

Di recente è diventato il *sound system* di default nella maggior parte delle distribuzioni *Linux*, tuttavia, il suo modulo equalizzatore è stato deprecato a causa di diversi *bug* riscontrati dagli utenti e a causa del *design* del modello matematico e del software che è risultato essere poco robusto e accurato.

Discutendo con gli sviluppatori di *PulseAudio* si è evinto che l'equalizzatore trattato in questa tesi potrebbe, salvo imprevisti, diventare il nuovo equalizzatore di *PulseAudio* dalla versione *13.0.*

È stato implementato l'equalizzatore in un modulo per *PulseAudio* con l'aggiunta di un'interfaccia grafica scritta in *C++* e *QML* con l'ausilio del framework *Qt.* Per la scrittura dell'interfaccia grafica si è fatto ausilio dell'IDE *Qt Creator.*

Tutto il codice sorgente è *Open Source* ed è consultabile liberamente nel *repository GitHub* del tesista sottoscritto (https://github.com/andrea993/audioeqpro.git), anche mediante il *front-end* https://github.com/andrea993/audioeqpro. In appendice è riportato il codice sorgente più rilevante, omettendo alcuni file generati automaticamente da *Qt Creator.*

7.1 Eqpro: il modulo *PulseAudio*

Per la scrittura dei moduli *PulseAudio* ci si basa sul codice sorgente del modulo *virtual sink* che rappresenta uno scheletro comune a tutti i moduli ma che non compie alcun tipo di elaborazione, è stato quindi necessario aggiungere alcune funzioni e ridefinire alcune *callback* in modo da implementare l'elaborazione dell'equalizzatore grafico e permetterne la customizzazione.

Il modulo è stato chiamato *eqpro* ed è stato scritto in *C90,* va compilato assieme al codice sorgente di *PulseAudio*, inserendolo nell'apposita directory dell'albero del sorgente, assieme agli altri moduli, si veda il codice sorgente in appendice.

Il modulo è composto da un insieme di callback che *PulseAudio* chiama a seconda dello stato in cui si trova, di seguito verranno spiegate le callback che sono state ridefinite in parte o completamente rispetto al modulo *module-virutal-sink*[s3]. L'unica informazione che può essere condivisa tra le callback è quella contenuta nella *struct userdata* che in questo modulo

contiene le informazioni di base per i moduli *PulseAudio* e una sottostruttura *equalizerPar* che contiene le informazioni necessarie all'equalizzatore.

7.1.1 Callback *pa_init()*

La callback *pa_init()* viene chiamata al momento del caricamento del modulo, vengono letti e parsati gli eventuali argomenti passati al modulo, vengono allocate e inizializzate le risorse necessarie all'utilizzo del modulo e vengono registrate le callback gestite dal modulo.

Il modulo permette di essere inizializzato passandogli alcuni parametri, quelli standard presenti in ogni modulo e alcuni specifici per l'equalizzatore, quali:

- *db*
  Il guadagno massimo e minimo delle bande, ogni banda avrà guadagno personalizzabile nell'intervallo $[-\mathrm{db}, \mathrm{db}]$ espresso in decibel, di default è $12\mathrm{dB}$
- *fmin*
  La frequenza centrale della prima banda, di default è $30\mathrm{Hz}$
- *octave*
  Il numero di ottave tra bade adiacenti, di default è 1
- *Nbands*
  Il numero di bande, di default è calcolato in funzione dei parametri precedenti
- *par*
  I valori di amplificazione o attenuazione per ogni banda nell'intervallo compreso tra -1 e 1 che devono essere passati nella forma $(\mathrm{x}_1; \mathrm{x}_2; \ldots; \mathrm{x}_N)$, di default la configurazione è *flat*
- *K*
  Un valore di amplificazione o attenuazione lineare per prevenire la saturazione qualora l'amplificazione porti l'uscita a un livello superiore a quello consentito dalla quantizzazione, di default è posto pari a 1.

Non tutte le specifiche possono essere impostate contemporaneamente, alcune sono mutualmente esclusive e dipendenti. Non si può imporre contemporaneamente la frequenza centrale della prima banda, il numero di bande e il numero di ottave tra le bande perché sono specifiche dipendenti e da due di esse è possibile calcolare la terza. La funzione *calcArgs()* prende in input le specifiche $*\,\mathrm{fmin}$,$*\,\mathrm{Nbands}$,$*\,\mathrm{octave}$, passate come puntatore, la frequenza di *Nyquist* FN e due valori booleani isfmin, $\mathrm{isNbands}$ che comunicano alla funzione se le due specifiche sono state imposte dall'utente e calcola le specifiche rimanenti sulla base di quelle passate dall'utente ed eventualmente da quelle predefinite.

Per analizzare la stringa contente i parametri è stata definita la funzione *readParFromStr()* che prende in input la stringa e il numero di bande, controlla che il formato sia corretto e setta i valori delle bande, se viene passato un numero di valori minore rispetto al numero delle bande assume che gli ultimi siano 0.

I parametri dell'equalizzatore sono salvati nella struttura *equalizerPar* contenuta in *struct userdata* e sono:

- *N*
  Il numero di bande dell'equalizzatore
- *R*
  La risoluzione delle bande, ovvero il passo della successione logaritmica, calcolato

come *R=pow(FN / *fmin, 1.0 / *Nbands)* qualora l'utente abbia specificato *fmin* e *Nbands* oppure come *R=pow(2,*octave)* nel caso abbia specificato *octave* o nel caso predefinito

- *DB*
  Il guadagno delle bande in decibel
- *K*
  Il valore di amplificazione master
- *f_min*
  La frequenza centrale della prima banda
- ***c*
  Il puntatore alla matrice dei coefficienti
- **par*
  Il puntatore al vettore dei valori di amplificazione delle singole bande
- ****X*
  Il puntatore a una matrice a tre dimensioni in cui viene salvato lo stato dei filtri

Le risorse puntate in *equalizerPar* vengono allocate nella funzione *eq_init()* dove si allocano le matrici sopra elencate, le cui dimensioni dipendono dal numero di bande scelte e dal numero di canali audio. Successivamente le risorse vengono inizializzate nella funzione *eq_preprocessing()* che funziona in modo analogo a quella dell'*implementazione in C90* salvo il fatto che le dimensioni delle matrici non sono note a compile time.

7.1.2 Callback *pa_done()*

La callback *pa_done()* viene chiamata da *PulseAudio* qualora il modulo venga rilasciato o da *pa_init()* nel caso avvenga un errore in fase di inizializzazione, si occupa di liberare le risorse precedentemente allocate.

7.1.3 Callback *sink_input_pop_cb()*

La callback *sink_input_pop_cb()* viene chiamata da *PulseAudio* quando uno stream ha bisogno di essere elaborato, per prima cosa controlla che uno stream precedentemente elaborato non sia da scartare, in tal caso chiama la funzione *pa_sink_process_rewind()* per riportare il modulo ad uno stato precedente. In caso di *rewind* andrebbe fatto un *rollback* sullo stato dei filtri dell'equalizzatore, tuttavia questo significherebbe tenere molti stati passati in memoria in quanto non è noto su quanti campioni il *rewind* avrà luogo e questo comporterebbe l'ausilio di molte risorse. Un ulteriore modo di gestire il *rewind* è quello di fare una compressione *lossy* sulla variazione degli stati, ad esempio con interpolazione, in modo che ne risulti un'approssimazione adeguata degli stati precedenti, tuttavia, anche questo metodo comporterebbe un costo computazionale non trascurabile. Infine, un ultimo modo proposto è la possibilità di sfruttare la linearità dei filtri e calcolare una funzione inversa che prende come ingresso lo stream scartato e operi una deconvoluzione per calcolare lo stato precedente, se però i guadagni dei filtri sono stati variati durante lo stream è necessario invertire una matrice di dimensioni elevate e non è possibile farlo nel tempo di campionamento, a meno di non possedere una grossa potenza di calcolo. In definitiva è stato scelto di non gestire il *rewind* per quanto riguarda lo stato dei filtri poiché le situazioni in cui diventa necessario sono rare e costerebbe molte risorse che nella maggior parte dei casi sarebbero spese inutilmente. Da un punto di vista intuitivo, per quanto riguarda lo stato dei filtri, non gestire il *rewind* si traduce nel filtrare un segnale che nella peggiore delle ipotesi

diventa leggermente differente dallo stream effettivo, in quanto potrebbero esserci ripetizioni o aggiunte di piccoli intervalli.

Successivamente nella definizione della callback viene allocato un puntatore allo stream originale e uno all'area di memoria disponibile alla scrittura di quello elaborato che avrà la stessa dimensione, variabile ad ogni chiamata della callback. I campioni sono quantizzati in *float* e gli stream sono vettori in cui i campioni sono raggruppati per canale e per istante di campionamento. Per filtrare il segnale viene chiamata la funzione *eq_filter()* che si comporta in modo analogo a quella dell'*implementazione in C90* e si occupa di filtrare il campione attuale moltiplicato per il fattore di guadagno lineare *K* e infine di aggiornare lo stato dei filtri.

7.1.4 Callback *eqpro_message_handler()*

Lo sviluppatore di *PulseAudio Georg Chini* si è offerto di sviluppare un sistema di messaggi tra moduli e applicazioni terze allo scopo di permettere la semplice implementazione di un'interfaccia grafica per questo modulo e per poterlo controllare, inoltre, da linea di comando mediante l'utility *pactl* che serve a gestire *PulseAudio*. Per aggiungere questa funzionalità lo sviluppatore ha scritto otto patch per *PulseAudio* che verranno rilasciate parallelamente a questo modulo. Le patch permettono di registrare una callback e chiamarla da applicazioni terze passando argomenti come stringhe, le patch permettono anche di convertire in stringa e viceversa vari tipi di dato, tra cui interi e floating point.

Nel modulo sono stati gestiti i seguenti messaggi:

- *sliderchange*
  viene chiamato quando l'utente intende cambiare il valore di uno slider dell'equalizzatore grafico e necessita, come informazioni aggiuntive, l'indice dello slider e il nuovo valore che dovrà assumere. Se il cambiamento avviene con successo risponde all'applicazione terza con la stringa "OK"
- *dialchange*
  viene chiamato, con la variabile numerica aggiuntiva, quando si intende cambiare il valore dell'amplificazione lineare dell'equalizzatore, *K.* In caso di avvenuto cambiamento risponde con la stringa "OK".
- *getinfo*
  viene chiamato quando l'applicazione terza ha bisogno di conoscere lo stato attuale dell'equalizzatore e le viene risposto con il numero di bande, la frequenza centrale della prima banda, l'amplificazione in decibel, la risoluzione tra le bande, il coefficiente di amplificazione lineare e il valore attuale di amplificazione di ciascuna banda.

7.2 Interfaccia grafica

Allo scopo di gestire in modo semplice il modulo è stata scritta un'interfaccia grafica in *C++* e *Qt.* Per quanto concerne il framework *Qt* è stato utilizzato, in aggiunta, il linguaggio dichiarativo *QML* per definire i componenti della gui affiancato da alcune funzioni *Javascript*, il codice sorgente è presentato in appendice.

L'interfaccia grafica è composta da tre pagine:

- **Modulo non caricato**
  Questa pagine è visibile quando si avvia l'interfaccia grafica ma il modulo *eqpro* non stato caricato, è necessario caricarlo e premere *Retry*.

Figura 7.1: Pagina modulo non caricato

- **Scelta del modulo**
  Nel caso ci fossero più moduli *eqpro* caricati contemporaneamente l'interfaccia grafica richiede di scegliere quale modulo si intende controllare tramite l'istanza attuale della gui.

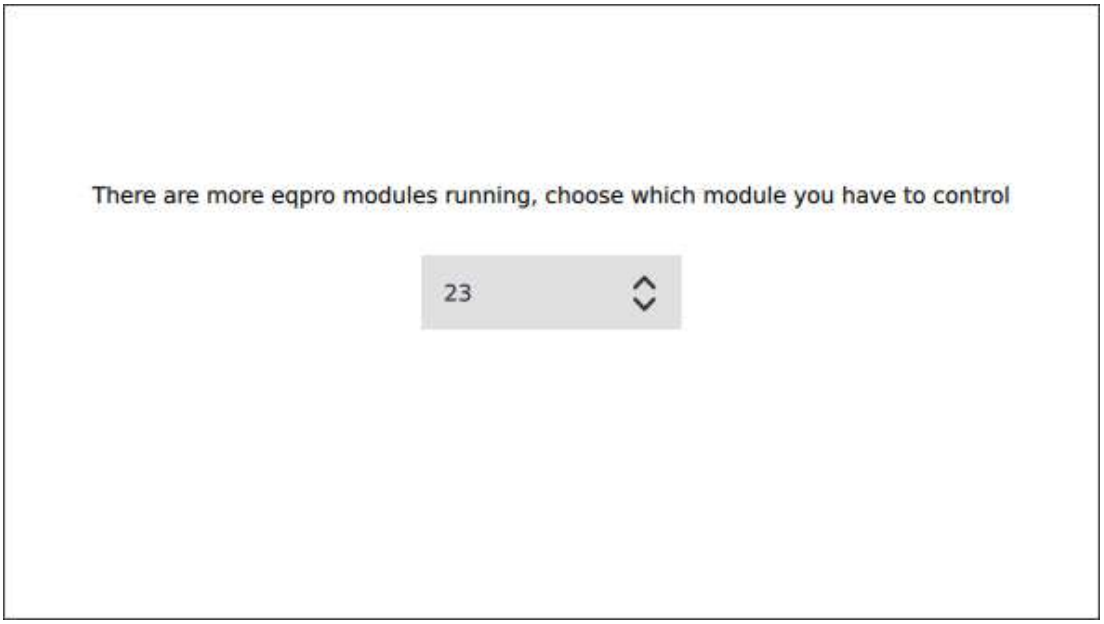


Figura 7.2: Figura 30: Pagina di scelta del modulo

- **Controllo del modulo**
  La pagina di controllo permette di avere accesso a tutti i parametri del modulo e permette di regolare l'equalizzatore in tempo reale, i valori degli slider possono essere cambiati sia mediante trascinamento sia scrivendo il valore decimale nella casella di testo sottostante. Il fattore di amplificazione lineare è regolabile mediante il *dial* in basso a destra, la manopola è stata graduata in modo che l'effetto sia logaritmico in modo da semplificare la gestione all'utente. Qualora il numero di bande sia elevato diventa possibile scorrere la sezione degli slider mediante trascinamento verso sinistra o destra.

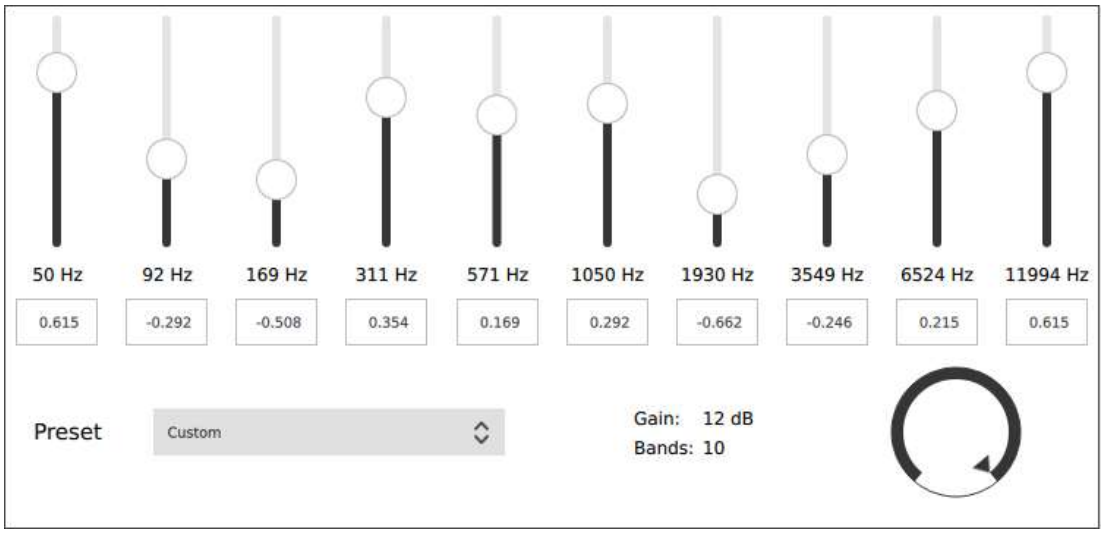


Figura 7.3: Pagina di controllo

Le tre pagine sono descritte nel linguaggio *QML* e interagiscono con il sorgente *C++*. Il sorgente *C++* si occupa di interagire con *PulseAudio* mediante le apposite *API* che grazie alle *patch* per la comunicazione permettono, inoltre, di inviare messaggi al modulo equalizzatore.

La parte *QML* si occupa di descrivere i vari *widget* usati dall'interfaccia grafica con l'ausilio di alcune funzioni dinamiche scritte in *Javascript* direttamente nei file *.qml,* come ad esempio la generazione di un numero di *slider* variabile, a seconda di quanti sono stati definiti in fase di caricamento del modulo o anche di simulare il funzionamento logaritmico per quanto concerne il *dial* che controlla il guadagno lineare.

La parte *C++* si occupa di comunicare con il modulo mediante la classe *PulseDriver*, aggiornare lo stato della *GUI,* gestire gli interventi dell'utente e cambiare la pagina corrente nella *GUI* a seconda dello stato del modulo mediante la classe *GuiManager.*

La classe *GuiManager* gestisce tre differenti stati della *GUI:*

- *connected*
  il modulo è caricato ed è risolta la disambiguazione su quale modulo l'istanza della *GUI* deve gestire, nel caso ne fossero caricati molteplici. Viene visualizzata e gestita la *Pagina di controllo.*

- *disconnected*
  nessun modulo equalizzatore risulta essere caricato. Viene visualizzata la *Pagina modulo non caricato.*
- *chooseconnection*
  non è mai stata risolta la disambiguazione su quale modulo equalizzatore controllare e ne è caricato più di uno, sarà necessario scegliere il l'*ID* del modulo corretto. Viene visualizzata la *Pagina di scelta del modulo.*

La funzione *EstimateState()* viene chiamato ogni volta che è possibile che lo stato sia variato e si occupa di comprendere qual è lo stato dell'interfaccia che descrive la situazione attuale e di impostarlo.

La funzione *SetState()* cambia lo stato attuale della *GUI* e viene in genere chiamata da *EstimateState()* quando risulta necessario modificare lo stato, in base allo stato precedente gestisce in modo adeguato il cambio di stato.

Quando la *GUI* si trova nello stato *connected* e la finestra viene attivata dall'utente, la *callback AppStateChanged()* richiede lo stato attuale del modulo poiché altre applicazioni terze potrebbero averlo cambiato mentre l'interfaccia è rimasta in *idle.*

Nel caso si verifichi un errore con la comunicazione con *PulseAudio* viene ristimato lo stato dell'interfaccia grafica in quanto l'errore potrebbe essere dovuto a una variazione esterna.

La classe *PulseDriver* si interfaccia con *PulseAudio* a cui fa alcune richieste. *PulseAudio* non risponde immediatamente alle richieste ma le inserisce in una propria coda e le gestisce quando lo ritiene opportuno. È pertanto necessario registrare alcune *callback* che *PulseAudio* si impegnerà a chiamare quando i compiti richiesti verranno assolti.

È stato definito il metodo *template MakeOperation()* che si occupa di interfacciarsi a *PulseAudio,* registrare le *callback,* istanziare una struttura *userdata* il cui puntatore verrà scambiato tra le *callback*, attendere che *PulseAudio* gestisca le richieste e infine deferenziale le *callback* e liberare le risorse per poi tornare il risultato desiderato. Grazie all'uso del *template* è stato possibile sfruttare metodo per operazioni di natura diversa, come ad esempio richiedere la lista dei moduli attualmente caricati, la lista dei generatori di *stream* (*sinks)* attualmente in uso e inviare messaggi al modulo equalizzatore selezionato.

CONCLUSIONI

I risultati proposti dagli articoli in bibliografia hanno spesso necessitato di pesanti modifiche e miglioramenti per poter arrivare al modello finale dell'equalizzatore, tutti i risultati presentati sono stati verificati e simulati in diverse situazioni prima di essere approvati e integrati nel modello.

In futuro il lavoro sarà sicuramente protagonista di ulteriori miglioramenti atti a mantenere una linea aggiornata per il modulo equalizzatore di *PulseAudio*. Ci sono inoltre altri ambiti in cui il modello si candida di essere implementato, primo tra questi un *plugin LADSPA (Linux Audio Developers Simple Plugin)*, compatibile con molti programmi che elaborano segnali audio.

Per quanto concerne il modulo equalizzatore, nella fase di approvazione da parte di *PulseAudio,* che avrà presto luogo, potranno esserci richieste non ancora previste da parte degli sviluppatori che potrebbero portare ad alcune modifiche del sorgente. Una volta che il modulo farà parte di *PulseAudio,* nel caso in cui future modifiche alla struttura del programma necessiteranno di aggiornare il modulo, sarà compito degli sviluppatori occuparsene. L'aggiunta di possibili nuove *feature* in *PulseAudio* sarà sicuramente motivo di ispirazione per il sottoscritto tesista di aggiornare il sorgente del modulo o il modello matematico per implementare nuove funzionalità e migliorie.

La speranza attuale è che l'equalizzatori diventi uno *standard* nel mondo del *free software*, l'integrazione in *PulseAudio* lo porterà ad essere ampliamente usato in ambito *consumer* e professionale. Lavorare a stretto contatto con gli sviluppatori di *PulseAudio* è risultato molto stimolante e ha portato a risultati estremamente soddisfacenti. Ci si augura che il contributo al miglioramento del progetto *freedesktop.org* possa essere in parte motivo di allargamento nell'utilizzo del sistema operativo *Linux* in ambito *consumer.*

Il *software* è stato fatto testare ad alcuni abituali ascoltatori di musica mediante impianti *hi-fi* che hanno immediatamente osservato come la qualità di questo equalizzatore sia al di sopra di quella di molte soluzioni che si trovano in rete o integrate anche in programmi di largo utilizzo, rimanendo stupefatti soprattutto per la capacita del modello di riuscire a isolare le singole bande senza che le code di quelle adiacenti interferiscano sensibilmente. La sensazione dei *tester* viene confermata anche sperimentalmente per mezzo della registrazione e dell'analisi di alcune risposte impulsive di vari programmi *consumer* e *professionali* provvisti di equalizzatori grafici audio.

APPENDICE

A. Implementazione in C90

```
/* Copyright (c) 2018 Andrea Drius
 * This file is under MIT License
 *
 * Read the LICENSE file for more information*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <memory.h>

#include "wav.h"

#define M 2
#define M2 4
#define N 10
#define SR 44100
#define DB 12
#define R 2
#define F_MIN 30

void preprocessing(double c[N][10]);
double filt(double u,double par[],double c[N][10],double x[N][M2]);

int main(int argc, char **argv)
{
    double c[N][10];
    double xL[N][M2]={{0}};
    double xR[N][M2]={{0}};
    double par[N]={-1,0,1,0,-1,0,1,0,-1,0};

    WAVE inp,out;
    FILE *finp, *fout;
    char outfname[100];
    unsigned i;

    if(argc != 2)
    {
        puts("filename of a valid stereo wav file is required, sampling
rate must be 44100Hz");
        return EXIT_FAILURE;
    }

    preprocessing(c); /*initialize filters*/

    finp=fopen(argv[1],"rb");
```

```
    sprintf(outfname,"%s_out.wav",argv[1]);
    fout=fopen(outfname,"wb");

    inp=ReadWave(finp);
    out=CreateEmptyCDaudioWave(inp.numofstereosamples);

    for(i=0;i<inp.numofstereosamples;i++)
    {
        SAMPLE(out,LEFT,i)=filt(SAMPLE(inp,LEFT,i),par,c,xL);
        SAMPLE(out,RIGHT,i)=filt(SAMPLE(inp,RIGHT,i),par,c,xR);
    }

    WriteWave(out,fout);
    fclose(fout);
    fclose(finp);

    ReleaseWaveData(&inp);
    ReleaseWaveData(&out);

    return EXIT_SUCCESS;
}

void preprocessing(double c[N][10])
{
    double v,g,cw,wcross,wc_n,fc_n,f_max,bw_n,T,tbw,c_m,d,Tpw;
    double a[3], b[3];
    int n;

    T=1.0/SR;

    g=pow(10,DB/20.0);
    wcross=pow(g,1.0/2.0/M);
    v=pow(g,1.0/M)-1;
    f_max=F_MIN*pow(R,N-1);

    for(n=0;n<N;n++)
    {
        fc_n=round(exp(log(F_MIN)+log(f_max/(double)F_MIN)*(n-
1)/(double)(N-1)));
        wc_n=2*M_PI*fc_n;
        bw_n=wc_n*(sqrt(R)-1.0/sqrt(R))/wcross;

        Tpw=2.0/bw_n*tan(bw_n/2.0*T);
        cw=cos(Tpw/2.0*sqrt(4*wc_n*wc_n+1))/cos(Tpw/2.0);
        tbw=Tpw*bw_n;
        c_m=cos(M_PI*(0.5-0.5/M));

        a[0]=4+4*c_m*tbw+tbw*tbw;
        a[1]=a[2]=1.0/a[0];
        a[1]*=2*tbw*tbw-8;
        a[2]*=a[0]-8*c_m*tbw;

        b[0]=b[1]=b[2]=tbw*v/a[0];
        b[0]*=2*tbw+4*c_m+tbw*v;
        b[1]*=2*tbw*(v+2);
```

```
        b[2]*=2*tbw-4*c_m+tbw*v;

        d=b[0]+1;
        c[n][0]=cw*(1-a[1]);
        c[n][1]=cw;
        c[n][2]=cw*(b[1]-a[1]*b[0]);
        c[n][3]=b[1]-a[1]*b[0];
        c[n][4]=-a[1];
        c[n][5]=cw*(b[2]-a[2]*b[0]);
        c[n][6]=-a[2]*cw;
        c[n][7]=b[2]-a[2]*b[0];
        c[n][8]=-a[2];
        c[n][9]=d;
    }

}

double filt(double u,double par[],double c[N][10],double x[N][M2])
{
    double gg,gg1,gg2,den,u0,xn2cn1;
    double xn[M2];
    int n,i;
    double y=u;

    for(n=0;n<N;n++)
    {
        gg=2*par[n];
        gg1=par[n]+1;
        gg2=par[n]-1;

        den=gg1-gg2*c[n][9];
        u0=(gg2*x[n][0]+2*y)/den;
        y=(gg*x[n][0]+(gg1*c[n][9]-gg2)*y)/den;

        xn2cn1=x[n][2]*c[n][1];

        xn[0]=x[n][0]*c[n][0]-x[n][1]+xn2cn1+c[n][2]*u0;
        xn[1]=x[n][0]*c[n][4]+x[n][2]+c[n][3]*u0;
        xn[2]=x[n][0]*c[n][6]+xn2cn1-x[n][3]+u0*c[n][5];
        xn[3]=x[n][0]*c[n][8]+u0*c[n][7];

        for (i=0;i<M2;i++)
            x[n][i]=xn[i];
    }
    return y;
}
```

Codice sorgente 1: main.c

B. Modulo *PulseAudio*

```
/***
  This file is part of PulseAudio.

  Copyright 2018 Andrea Drius <andrea993 dot nokiastore at gmail dot com>

  PulseAudio is free software; you can redistribute it and/or modify
  it under the terms of the GNU Lesser General Public License as published
  by the Free Software Foundation; either version 2.1 of the License,
  or (at your option) any later version.

  PulseAudio is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU Lesser General Public License
  along with PulseAudio; if not, see <http://www.gnu.org/licenses/>.
 ***/

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <pulse/gccmacro.h>
#include <pulse/xmalloc.h>
#include <pulse/message-params.h>

#include <pulsecore/i18n.h>
#include <pulsecore/namereg.h>
#include <pulsecore/sink.h>
#include <pulsecore/module.h>
#include <pulsecore/core-util.h>
#include <pulsecore/modargs.h>
#include <pulsecore/log.h>
#include <pulsecore/rtpoll.h>
#include <pulsecore/sample-util.h>
#include <pulsecore/ltdl-helper.h>
#include <pulsecore/message-handler.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <float.h>

PA_MODULE_AUTHOR("Andrea Drius");
PA_MODULE_DESCRIPTION("Professional customizable equalizer");
PA_MODULE_VERSION("v0.00.1-beta");
PA_MODULE_LOAD_ONCE(false);
PA_MODULE_USAGE(
        _("sink_name=<name for the sink> "
          "sink_properties=<properties for the sink> "
          "master=<name of sink to filter> "
          "rate=<sample rate> "
          "channels=<number of channels> "
```

```
            "channel_map=<channel map> "
            "use_volume_sharing=<yes or no> "
            "force_flat_volume=<yes or no> "
            "db=<filter gain in decibel>"
            "fmin=<central frequecy of the first band> "
            "octave=<octaves between bands> "
            "Nbands=<number of bands>"
            "par=<equalizer levels in the form (x1;x2;...;xn) from -1 to 1>"
            "K=<amplitude, to prevent saturation>"
            ));

#define MEMBLOCKQ_MAXLENGTH (16*1024*1024)
#define M 2
#define M2 4
#define DEFAULT_FMIN 30.0
#define DEFAULT_OCT 1.0
#define DEFAULT_DB 12.0
#define DEFAULT_K 1.0

#define MODULE_MSG_PATH "/modules/eqpro"

typedef struct __equalizerPar
{
    int N;
    double R;
    double DB;
    double K;
    double f_min;
    double **c;
    double *par;
    double ***X;
}equalizerPar;

struct userdata {
    pa_module *module;

    /* FIXME: Uncomment this and take "autoloaded" as a modarg if this is
a filter */
    /* bool autoloaded; */

    pa_sink *sink;
    pa_sink_input *sink_input;

    pa_memblockq *memblockq;

    bool auto_desc;
    unsigned channels;
    equalizerPar eqp;
};

static const char* const valid_modargs[] = {
    "db",
    "fmin",
    "Nbands",
    "octave",
```

```
    "par",
    "sink_name",
    "sink_properties",
    "master",
    "rate",
    "channels",
    "channel_map",
    "use_volume_sharing",
    "force_flat_volume",
    NULL
};

/*equazlier functions */
void eq_preprocessing(equalizerPar *eqp, double SR);
void eq_init(equalizerPar *eqp, double db, double f_min, int nChans,
double oct, int N, double *par, double K);
double eq_filter(double u, double par[], double **c, double **x, double K,
int N);
void calcArgs(bool isfmin, bool isNbands, double *fmin, unsigned *Nbands,
double *octave, double FN);
int readParFromStr(char *str, unsigned N, double *out);
static int eqpro_message_handler(const char *object_path, const char
*message, char *message_parameters, char **response, void *userdata);

/* Called from I/O thread context */
static int sink_process_msg_cb(pa_msgobject *o, int code, void *data,
int64_t offset, pa_memchunk *chunk) {
    struct userdata *u = PA_SINK(o)->userdata;

    switch (code) {

    case PA_SINK_MESSAGE_GET_LATENCY:

        /* The sink is _put() before the sink input is, so let's
                         * make sure we don't access it in that time.
Also, the
                         * sink input is first shut down, the sink second.
*/
        if (!PA_SINK_IS_LINKED(u->sink->thread_info.state) ||
                !PA_SINK_INPUT_IS_LINKED(u->sink_input->thread_info.state)
||
                !u->sink_input->sink) {
            *((int64_t*) data) = 0;
            return 0;
        }

        *((int64_t*) data) =

                /* Get the latency of the master sink */
                pa_sink_get_latency_within_thread(u->sink_input->sink,
true) +

                /* Add the latency internal to our sink input on top */
                pa_bytes_to_usec(pa_memblockq_get_length(u->sink_input-
>thread_info.render_memblockq), &u->sink_input->sink->sample_spec);
```

```
        return 0;

    case PA_SINK_MESSAGE_SET_STATE: {
        pa_sink_state_t new_state = (pa_sink_state_t)
PA_PTR_TO_UINT(data);

        /* When set to running or idle for the first time, request a
rewind

* of the master sink to make sure we are heard immediately */
        if ((new_state == PA_SINK_IDLE || new_state == PA_SINK_RUNNING) &&
u->sink->thread_info.state == PA_SINK_INIT) {
            pa_log_debug("Requesting rewind due to state change.");
            pa_sink_input_request_rewind(u->sink_input, 0, false, true,
true);
        }
    }
    }

    return pa_sink_process_msg(o, code, data, offset, chunk);
}

/* Called from main context */
static int sink_set_state_in_main_thread_cb(pa_sink *s, pa_sink_state_t
state, pa_suspend_cause_t suspend_cause) {
    struct userdata *u;

    pa_sink_assert_ref(s);
    pa_assert_se(u = s->userdata);

    if (!PA_SINK_IS_LINKED(state) ||
            !PA_SINK_INPUT_IS_LINKED(pa_sink_input_get_state(u-
>sink_input)))
        return 0;

    pa_sink_input_cork(u->sink_input, state == PA_SINK_SUSPENDED);
    return 0;
}

/* Called from the IO thread. */
static int sink_set_state_in_io_thread_cb(pa_sink *s, pa_sink_state_t
new_state, pa_suspend_cause_t new_suspend_cause) {
    struct userdata *u;

    pa_assert(s);
    pa_assert_se(u = s->userdata);

    /* When set to running or idle for the first time, request a rewind
     * of the master sink to make sure we are heard immediately */
    if ((new_state == PA_SINK_IDLE || new_state == PA_SINK_RUNNING) && u-
>sink->thread_info.state == PA_SINK_INIT) {
        pa_log_debug("Requesting rewind due to state change.");
        pa_sink_input_request_rewind(u->sink_input, 0, false, true, true);
    }
```

```
    return 0;
}

/* Called from I/O thread context */
static void sink_request_rewind_cb(pa_sink *s) {
    struct userdata *u;

    pa_sink_assert_ref(s);
    pa_assert_se(u = s->userdata);

    if (!PA_SINK_IS_LINKED(u->sink->thread_info.state) ||
            !PA_SINK_INPUT_IS_LINKED(u->sink_input->thread_info.state))
        return;

    /* Just hand this one over to the master sink */
    pa_sink_input_request_rewind(u->sink_input,
                                 s->thread_info.rewind_nbytes +
                                 pa_memblockq_get_length(u->memblockq),
true, false, false);
}

/* Called from I/O thread context */
static void sink_update_requested_latency_cb(pa_sink *s) {
    struct userdata *u;

    pa_sink_assert_ref(s);
    pa_assert_se(u = s->userdata);

    if (!PA_SINK_IS_LINKED(u->sink->thread_info.state) ||
            !PA_SINK_INPUT_IS_LINKED(u->sink_input->thread_info.state))
        return;

    /* Just hand this one over to the master sink */
    pa_sink_input_set_requested_latency_within_thread(
                u->sink_input,
                pa_sink_get_requested_latency_within_thread(s));
}

/* Called from main context */
static void sink_set_volume_cb(pa_sink *s) {
    struct userdata *u;

    pa_sink_assert_ref(s);
    pa_assert_se(u = s->userdata);

    if (!PA_SINK_IS_LINKED(pa_sink_get_state(s)) ||
            !PA_SINK_INPUT_IS_LINKED(pa_sink_input_get_state(u-
>sink_input)))
        return;

    pa_sink_input_set_volume(u->sink_input, &s->real_volume, s-
>save_volume, true);
}
```

```
/* Called from main context */
static void sink_set_mute_cb(pa_sink *s) {
    struct userdata *u;

    pa_sink_assert_ref(s);
    pa_assert_se(u = s->userdata);

    if (!PA_SINK_IS_LINKED(pa_sink_get_state(s)) ||
            !PA_SINK_INPUT_IS_LINKED(pa_sink_input_get_state(u-
>sink_input)))
        return;

    pa_sink_input_set_mute(u->sink_input, s->muted, s->save_muted);
}

/* Called from I/O thread context */
static int sink_input_pop_cb(pa_sink_input *i, size_t nbytes, pa_memchunk
*chunk) {
    struct userdata *u;
    float *src, *dst;
    size_t fs;
    unsigned n, c;
    pa_memchunk tchunk;
    pa_usec_t current_latency PA_GCC_UNUSED;

    pa_sink_input_assert_ref(i);
    pa_assert(chunk);
    pa_assert_se(u = i->userdata);

    if (!PA_SINK_IS_LINKED(u->sink->thread_info.state))
        return -1;
    /* Hmm, process any rewind request that might be queued up */
    pa_sink_process_rewind(u->sink, 0);

    /* (1) IF YOU NEED A FIXED BLOCK SIZE USE
         * pa_memblockq_peek_fixed_size() HERE INSTEAD. NOTE THAT FILTERS
         * WHICH CAN DEAL WITH DYNAMIC BLOCK SIZES ARE HIGHLY
         * PREFERRED. */
    while (pa_memblockq_peek(u->memblockq, &tchunk) < 0) {
        pa_memchunk nchunk;

        pa_sink_render(u->sink, nbytes, &nchunk);
        pa_memblockq_push(u->memblockq, &nchunk);
        pa_memblock_unref(nchunk.memblock);
    }

    /* (2) IF YOU NEED A FIXED BLOCK SIZE, THIS NEXT LINE IS NOT
         * NECESSARY */
    tchunk.length = PA_MIN(nbytes, tchunk.length);
    pa_assert(tchunk.length > 0);

    fs = pa_frame_size(&i->sample_spec);
    n = (unsigned) (tchunk.length / fs);

    pa_assert(n > 0);
```

```
    chunk->index = 0;
    chunk->length = n*fs;
    chunk->memblock = pa_memblock_new(i->sink->core->mempool, chunk-
>length);

    pa_memblockq_drop(u->memblockq, chunk->length);

    src = (float*)pa_memblock_acquire_chunk(&tchunk);
    dst = (float*)pa_memblock_acquire(chunk->memblock);

    /* (3) PUT YOUR CODE HERE TO DO SOMETHING WITH THE DATA */

    while(n>0)
    {
        for(c=0; c<u->channels; c++)
        {
            *dst=(float)eq_filter(*src,u->eqp.par,u->eqp.c,u->eqp.X[c],u-
>eqp.K,u->eqp.N);
            //*dst=*src;
            src++;
            dst++;
        }
        n--;
    }

    pa_memblock_release(tchunk.memblock);
    pa_memblock_release(chunk->memblock);

    pa_memblock_unref(tchunk.memblock);

    /* (4) IF YOU NEED THE LATENCY FOR SOMETHING ACQUIRE IT LIKE THIS: */
    current_latency =
            /* Get the latency of the master sink */
            pa_sink_get_latency_within_thread(i->sink, false) +

            /* Add the latency internal to our sink input on top */
            pa_bytes_to_usec(pa_memblockq_get_length(i-
>thread_info.render_memblockq), &i->sink->sample_spec);

    return 0;
}

/* Called from I/O thread context */
static void sink_input_process_rewind_cb(pa_sink_input *i, size_t nbytes)
{
    struct userdata *u;
    size_t amount = 0;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    /* If the sink is not yet linked, there is nothing to rewind */
    if (!PA_SINK_IS_LINKED(u->sink->thread_info.state))
        return;
```

```
    if (u->sink->thread_info.rewind_nbytes > 0) {
        size_t max_rewrite;

        max_rewrite = nbytes + pa_memblockq_get_length(u->memblockq);
        amount = PA_MIN(u->sink->thread_info.rewind_nbytes, max_rewrite);
        u->sink->thread_info.rewind_nbytes = 0;

        if (amount > 0) {
            pa_memblockq_seek(u->memblockq, - (int64_t) amount,
PA_SEEK_RELATIVE, true);

            /* (5) PUT YOUR CODE HERE TO RESET YOUR FILTER  */
        }
    }

    pa_sink_process_rewind(u->sink, amount);
    pa_memblockq_rewind(u->memblockq, nbytes);
}

/* Called from I/O thread context */
static void sink_input_update_max_rewind_cb(pa_sink_input *i, size_t
nbytes) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    /* FIXME: Too small max_rewind:
         * https://bugs.freedesktop.org/show_bug.cgi?id=53709 */
    pa_memblockq_set_maxrewind(u->memblockq, nbytes);
    pa_sink_set_max_rewind_within_thread(u->sink, nbytes);
}

/* Called from I/O thread context */
static void sink_input_update_max_request_cb(pa_sink_input *i, size_t
nbytes) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    /* (6) IF YOU NEED A FIXED BLOCK SIZE ROUND nbytes UP TO MULTIPLES
         * OF IT HERE. THE PA_ROUND_UP MACRO IS USEFUL FOR THAT. */

    pa_sink_set_max_request_within_thread(u->sink, nbytes);
}

/* Called from I/O thread context */
static void sink_input_update_sink_latency_range_cb(pa_sink_input *i) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);
```

```
    pa_sink_set_latency_range_within_thread(u->sink, i->sink-
>thread_info.min_latency, i->sink->thread_info.max_latency);
}

/* Called from I/O thread context */
static void sink_input_update_sink_fixed_latency_cb(pa_sink_input *i) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    /* (7) IF YOU NEED A FIXED BLOCK SIZE ADD THE LATENCY FOR ONE
         * BLOCK MINUS ONE SAMPLE HERE. pa_usec_to_bytes_round_up() IS
         * USEFUL FOR THAT. */

    pa_sink_set_fixed_latency_within_thread(u->sink, i->sink-
>thread_info.fixed_latency);
}

/* Called from I/O thread context */
static void sink_input_detach_cb(pa_sink_input *i) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    if (PA_SINK_IS_LINKED(u->sink->thread_info.state))
        pa_sink_detach_within_thread(u->sink);

    pa_sink_set_rtpoll(u->sink, NULL);
}

/* Called from I/O thread context */
static void sink_input_attach_cb(pa_sink_input *i) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    pa_sink_set_rtpoll(u->sink, i->sink->thread_info.rtpoll);
    pa_sink_set_latency_range_within_thread(u->sink, i->sink-
>thread_info.min_latency, i->sink->thread_info.max_latency);

    /* (8.1) IF YOU NEED A FIXED BLOCK SIZE ADD THE LATENCY FOR ONE
         * BLOCK MINUS ONE SAMPLE HERE. SEE (7) */
    pa_sink_set_fixed_latency_within_thread(u->sink, i->sink-
>thread_info.fixed_latency);

    /* (8.2) IF YOU NEED A FIXED BLOCK SIZE ROUND
         * pa_sink_input_get_max_request(i) UP TO MULTIPLES OF IT
         * HERE. SEE (6) */
    pa_sink_set_max_request_within_thread(u->sink,
pa_sink_input_get_max_request(i));

    /* FIXME: Too small max_rewind:
```

```
         * https://bugs.freedesktop.org/show_bug.cgi?id=53709 */
    pa_sink_set_max_rewind_within_thread(u->sink,
pa_sink_input_get_max_rewind(i));

    if (PA_SINK_IS_LINKED(u->sink->thread_info.state))
        pa_sink_attach_within_thread(u->sink);

}

/* Called from main context */
static void sink_input_kill_cb(pa_sink_input *i) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    /* The order here matters! We first kill the sink so that streams
         * can properly be moved away while the sink input is still
conneted
         * to the master. */
    pa_sink_input_cork(u->sink_input, true);
    pa_sink_unlink(u->sink);
    pa_sink_input_unlink(u->sink_input);

    pa_sink_input_unref(u->sink_input);
    u->sink_input = NULL;

    pa_sink_unref(u->sink);
    u->sink = NULL;

    pa_module_unload_request(u->module, true);
}

/* Called from main context */
static void sink_input_moving_cb(pa_sink_input *i, pa_sink *dest) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    if (dest) {
        pa_sink_set_asyncmsgq(u->sink, dest->asyncmsgq);
        pa_sink_update_flags(u->sink,
PA_SINK_LATENCY|PA_SINK_DYNAMIC_LATENCY, dest->flags);
    } else
        pa_sink_set_asyncmsgq(u->sink, NULL);

    if (u->auto_desc && dest) {
        const char *z;
        pa_proplist *pl;

        pl = pa_proplist_new();
        z = pa_proplist_gets(dest->proplist, PA_PROP_DEVICE_DESCRIPTION);
        pa_proplist_setf(pl, PA_PROP_DEVICE_DESCRIPTION, "Virtual Sink %s
on %s",
```

```
                        pa_proplist_gets(u->sink->proplist,
"device.vsink.name"), z ? z : dest->name);

        pa_sink_update_proplist(u->sink, PA_UPDATE_REPLACE, pl);
        pa_proplist_free(pl);
    }
}

/* Called from main context */
static void sink_input_volume_changed_cb(pa_sink_input *i) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    pa_sink_volume_changed(u->sink, &i->volume);
}

/* Called from main context */
static void sink_input_mute_changed_cb(pa_sink_input *i) {
    struct userdata *u;

    pa_sink_input_assert_ref(i);
    pa_assert_se(u = i->userdata);

    pa_sink_mute_changed(u->sink, i->muted);
}

int pa__init(pa_module*m) {
    struct userdata *u;
    pa_sample_spec ss;
    pa_channel_map map;
    pa_modargs *ma;
    pa_sink *master=NULL;
    pa_sink_input_new_data sink_input_data;
    pa_sink_new_data sink_data;
    bool use_volume_sharing = true;
    bool force_flat_volume = false;
    pa_memchunk silence;
    double fmin=DEFAULT_FMIN, octave=DEFAULT_OCT, db=DEFAULT_DB,
K=DEFAULT_K;
    unsigned Nbands = 0;
    double* par=NULL;
    char *str=NULL, *fullpath_str=NULL;
    bool isfmin=false, isNbands=false, isoctave=false;
    int ret;

    pa_assert(m);

    if (!(ma = pa_modargs_new(m->argument, valid_modargs))) {
        pa_log("Failed to parse module arguments.");
        goto fail;
    }
```

```
    if (!(master = pa_namereg_get(m->core, pa_modargs_get_value(ma,
"master", NULL), PA_NAMEREG_SINK))) {
        pa_log("Master sink not found");
        goto fail;
    }

    pa_assert(master);

    ss = master->sample_spec;
    ss.format = PA_SAMPLE_FLOAT32;
    map = master->channel_map;
    if (pa_modargs_get_sample_spec_and_channel_map(ma, &ss, &map,
PA_CHANNEL_MAP_DEFAULT) < 0) {
        pa_log("Invalid sample format specification or channel map");
        goto fail;
    }

    if (pa_modargs_get_value_boolean(ma, "use_volume_sharing",
&use_volume_sharing) < 0) {
        pa_log("use_volume_sharing= expects a boolean argument");
        goto fail;
    }

    if (pa_modargs_get_value_boolean(ma, "force_flat_volume",
&force_flat_volume) < 0) {
        pa_log("force_flat_volume= expects a boolean argument");
        goto fail;
    }

    ret=pa_modargs_get_value_double(ma, "db", &db);
    if (ret < 0) {
        pa_log("db= expects a double argument");
        goto fail;
    }

    ret=pa_modargs_get_value_double(ma, "K", &K);
    if (ret < 0) {
        pa_log("K= expects a double argument");
        goto fail;
    }
    if (K<0) {
        pa_log("K= expects a positive double");
        goto fail;
    }

    ret=pa_modargs_get_value_double(ma, "fmin", &fmin);
    if (ret < 0) {
        pa_log("fmin= expects a double argument");
        goto fail;
    }
    if (fmin <= 0 || fmin>=ss.rate/2.0) {
        pa_log("fmin= expects a positive double less than (sampling
rate)/2");
        goto fail;
    }
```

```
    if (fabs(fmin - DEFAULT_FMIN) > DBL_EPSILON)
        isfmin=true;

    ret=pa_modargs_get_value_double(ma, "octave", &octave);
    if (ret < 0) {
        pa_log("octave= expects a double argument");
        goto fail;
    }
    if (octave <= 0) {
        pa_log("octave= expects a positive double");
        goto fail;
    }
    if (fabs(octave - DEFAULT_OCT) > DBL_EPSILON)
        isoctave=true;

    ret=pa_modargs_get_value_u32(ma, "Nbands", &Nbands);
    if (ret < 0) {
        pa_log("Nbands= expects an unsigned argument");
        goto fail;
    }
    if (Nbands > 0)
        isNbands=true;

    if (isoctave && isNbands && isfmin) {
        pa_log("You can choose up to two arguments between: octave,
Nbands, fmin");
        goto fail;
    }
    calcArgs(isfmin, isNbands, &fmin, &Nbands, &octave, ss.rate/2.0);

    par=pa_xmalloc0(Nbands*sizeof(double));
    if ((str=pa_xstrdup(pa_modargs_get_value(ma, "par", NULL)))) {
        if (readParFromStr(str, Nbands, par) < 0) {
            pa_log("par= expects an array of double in the format
(x1;x2,...;xn) from -1 to 1");
            goto fail;
        }
    }

    if (use_volume_sharing && force_flat_volume) {
        pa_log("Flat volume can't be forced when using volume sharing.");
        goto fail;
    }

    u = pa_xnew0(struct userdata, 1);
    u->module = m;
    m->userdata = u;
    u->channels = ss.channels;

    /* Create sink */
    pa_sink_new_data_init(&sink_data);
    sink_data.driver = __FILE__;
    sink_data.module = m;
    if (!(sink_data.name = pa_xstrdup(pa_modargs_get_value(ma,
"sink_name", NULL))))
```

```
        sink_data.name = pa_sprintf_malloc("%s.vsink", master->name);
    pa_sink_new_data_set_sample_spec(&sink_data, &ss);
    pa_sink_new_data_set_channel_map(&sink_data, &map);
    pa_proplist_sets(sink_data.proplist, PA_PROP_DEVICE_MASTER_DEVICE,
master->name);
    pa_proplist_sets(sink_data.proplist, PA_PROP_DEVICE_CLASS, "filter");
    pa_proplist_sets(sink_data.proplist, "device.vsink.eqpro",
sink_data.name);

    if (pa_modargs_get_proplist(ma, "sink_properties", sink_data.proplist,
PA_UPDATE_REPLACE) < 0) {
        pa_log("Invalid properties");
        pa_sink_new_data_done(&sink_data);
        goto fail;
    }

    if ((u->auto_desc = !pa_proplist_contains(sink_data.proplist,
PA_PROP_DEVICE_DESCRIPTION))) {
        const char *z;

        z = pa_proplist_gets(master->proplist,
PA_PROP_DEVICE_DESCRIPTION);
        pa_proplist_setf(sink_data.proplist, PA_PROP_DEVICE_DESCRIPTION,
"EqualizerPro Sink %s on %s", sink_data.name, z ? z : master->name);
    }

    u->sink = pa_sink_new(m->core, &sink_data, (master->flags &
(PA_SINK_LATENCY|PA_SINK_DYNAMIC_LATENCY))
                          | (use_volume_sharing ?
PA_SINK_SHARE_VOLUME_WITH_MASTER : 0));
    pa_sink_new_data_done(&sink_data);

    if (!u->sink) {
        pa_log("Failed to create sink.");
        goto fail;
    }

    u->sink->parent.process_msg = sink_process_msg_cb;
    u->sink->set_state_in_main_thread = sink_set_state_in_main_thread_cb;
    u->sink->set_state_in_io_thread = sink_set_state_in_io_thread_cb;
    u->sink->update_requested_latency = sink_update_requested_latency_cb;
    u->sink->request_rewind = sink_request_rewind_cb;
    pa_sink_set_set_mute_callback(u->sink, sink_set_mute_cb);
    if (!use_volume_sharing) {
        pa_sink_set_set_volume_callback(u->sink, sink_set_volume_cb);
        pa_sink_enable_decibel_volume(u->sink, true);
    }
    /* Normally this flag would be enabled automatically be we can force
it. */
    if (force_flat_volume)
        u->sink->flags |= PA_SINK_FLAT_VOLUME;
    u->sink->userdata = u;

    pa_sink_set_asyncmsgq(u->sink, master->asyncmsgq);
```

```
    /* Create sink input */
    pa_sink_input_new_data_init(&sink_input_data);
    sink_input_data.driver = __FILE__;
    sink_input_data.module = m;
    pa_sink_input_new_data_set_sink(&sink_input_data, master, false,
true);
    sink_input_data.origin_sink = u->sink;
    pa_proplist_setf(sink_input_data.proplist, PA_PROP_MEDIA_NAME,
"Virtual Sink Stream from %s", pa_proplist_gets(u->sink->proplist,
PA_PROP_DEVICE_DESCRIPTION));
    pa_proplist_sets(sink_input_data.proplist, PA_PROP_MEDIA_ROLE,
"filter");
    pa_sink_input_new_data_set_sample_spec(&sink_input_data, &ss);
    pa_sink_input_new_data_set_channel_map(&sink_input_data, &map);
    sink_input_data.flags |= PA_SINK_INPUT_START_CORKED;
    pa_sink_input_new(&u->sink_input, m->core, &sink_input_data);
    pa_sink_input_new_data_done(&sink_input_data);

    if (!u->sink_input)
        goto fail;

    u->sink_input->pop = sink_input_pop_cb;
    u->sink_input->process_rewind = sink_input_process_rewind_cb;
    u->sink_input->update_max_rewind = sink_input_update_max_rewind_cb;
    u->sink_input->update_max_request = sink_input_update_max_request_cb;
    u->sink_input->update_sink_latency_range =
sink_input_update_sink_latency_range_cb;
    u->sink_input->update_sink_fixed_latency =
sink_input_update_sink_fixed_latency_cb;
    u->sink_input->kill = sink_input_kill_cb;
    u->sink_input->attach = sink_input_attach_cb;
    u->sink_input->detach = sink_input_detach_cb;
    u->sink_input->moving = sink_input_moving_cb;
    u->sink_input->volume_changed = use_volume_sharing ? NULL :
sink_input_volume_changed_cb;
    u->sink_input->mute_changed = sink_input_mute_changed_cb;
    u->sink_input->userdata = u;

    u->sink->input_to_master = u->sink_input;

    pa_sink_input_get_silence(u->sink_input, &silence);
    u->memblockq = pa_memblockq_new("module-virtual-sink memblockq", 0,
MEMBLOCKQ_MAXLENGTH, 0, &ss, 1, 1, 0, &silence);
    pa_memblock_unref(silence.memblock);

    //init eq
    eq_init(&u->eqp,db,fmin,u->channels,octave,(int)Nbands,par, K);
    eq_preprocessing(&u->eqp,ss.rate);

    /* The order here is important. The input must be put first,
         * otherwise streams might attach to the sink before the sink
         * input is attached to the master. */

    pa_sink_input_put(u->sink_input);
    pa_sink_put(u->sink);
```

```
    pa_sink_input_cork(u->sink_input, false);

    pa_modargs_free(ma);
    if (str)
        pa_xfree(str);


    /* Messages handling */
    fullpath_str = pa_sprintf_malloc("%s/%d",MODULE_MSG_PATH, m->index);
    if (pa_hashmap_get(m->core->message_handlers, fullpath_str) != NULL) {
        pa_log("the communication path is in use");
        goto fail;
    }

    pa_message_handler_register(m->core, fullpath_str, "communication with
eqpro_gui", eqpro_message_handler, (void*)u);
    pa_xfree(fullpath_str);

    return 0;

fail:
    if (ma)
        pa_modargs_free(ma);
    if (par)
        pa_xfree(par);
    if (str)
        pa_xfree(str);

    pa_xfree(fullpath_str);

    pa__done(m);

    return -1;
}

int pa__get_n_used(pa_module *m) {
    struct userdata *u;

    pa_assert(m);
    pa_assert_se(u = m->userdata);

    return pa_sink_linked_by(u->sink);
}

void pa__done(pa_module*m) {
    struct userdata *u;
    int n, i;
    char *fullpath_str;

    pa_assert(m);

    if (!(u = m->userdata))
        return;

    fullpath_str = pa_sprintf_malloc("%s/%d",MODULE_MSG_PATH, m->index);
```

```
    if (pa_hashmap_get(m->core->message_handlers, fullpath_str)) {
        pa_message_handler_unregister(m->core, fullpath_str);
    }
    pa_xfree(fullpath_str);

    /* See comments in sink_input_kill_cb() above regarding
     * destruction order! */

    if (u->sink_input)
        pa_sink_input_cork(u->sink_input, true);

    if (u->sink)
        pa_sink_unlink(u->sink);

    if (u->sink_input)
        pa_sink_input_unlink(u->sink_input);

    if (u->sink_input)
        pa_sink_input_unref(u->sink_input);

    if (u->sink)
        pa_sink_unref(u->sink);

    if (u->memblockq)
        pa_memblockq_free(u->memblockq);

    /*free equalizerPar resources*/
    if (u->eqp.par)
        pa_xfree(u->eqp.par);

    for(n=0; n<u->eqp.N; n++) {
        pa_xfree(u->eqp.c[n]);
    }

    pa_xfree(u->eqp.c);

    for(n=0; n<(int)u->channels; n++) {
        for(i=0; i<u->eqp.N; i++) {
            pa_xfree(u->eqp.X[n][i]);
        }

        pa_xfree(u->eqp.X[n]);
    }

    pa_xfree(u->eqp.X);

    /*free userdata*/
    pa_xfree(u);
}

double eq_filter(double u, double par[], double **c, double **x, double K,
int N)
{
    double pn2,pn21,pn22,den,u0,xn2cn1;
    double xn[M2];
```

```
    int n,i;
    double y=u*K;

    for(n=0; n<N; n++)
    {
        pn2=2*par[n];
        pn21=par[n]+1;
        pn22=par[n]-1;

        den=pn21-pn22*c[n][9];
        u0=(pn22*x[n][0]+2*y)/den;
        y=(pn2*x[n][0]+(pn21*c[n][9]-pn22)*y)/den;

        xn2cn1=x[n][2]*c[n][1];

        xn[0]=x[n][0]*c[n][0]-x[n][1]+xn2cn1+c[n][2]*u0;
        xn[1]=x[n][0]*c[n][4]+x[n][2]+c[n][3]*u0;
        xn[2]=x[n][0]*c[n][6]+xn2cn1-x[n][3]+u0*c[n][5];
        xn[3]=x[n][0]*c[n][8]+u0*c[n][7];

        for (i=0; i<M2; i++)
            x[n][i]=xn[i];
    }
    return y;

}

void eq_init(equalizerPar *eqp, double db, double f_min, int nChans,
double oct, int N, double *par, double K)
{
    int n,i;
    eqp->f_min=f_min;
    eqp->DB=db;
    eqp->R=pow(2,oct);
    eqp->N=N;
    eqp->par=par;
    eqp->K=K;

    eqp->c=(double**)pa_xmalloc(eqp->N*sizeof(double*));
    for(n=0; n<eqp->N; n++)
        eqp->c[n]=(double*)pa_xmalloc(10*sizeof(double));

    eqp->X=(double***)pa_xmalloc(nChans*sizeof(double**));
    for(i=0; i<nChans; i++)
    {
        eqp->X[i]=(double**)pa_xmalloc(eqp->N*sizeof(double*));
        for(n=0; n<eqp->N; n++)
            eqp->X[i][n]=(double*)pa_xmalloc0(M2*sizeof(double));
    }

}

void eq_preprocessing(equalizerPar *eqp, double SR)
{
    double v,g,cw,wcross,wc_n,fc_n,f_max,bw_n,T,tbw,c_m,d,Tpw;
```

```
    double a[3], b[3];
    int n;

    T=1.0/SR;

    g=pow(10,eqp->DB/20.0);
    wcross=pow(g,1.0/2.0/M);
    v=pow(g,1.0/M)-1;
    f_max=eqp->f_min*pow(eqp->R,eqp->N-1);

    for(n=0; n<eqp->N; n++)
    {
        fc_n=round(exp(log(eqp->f_min)+log(f_max/(double)eqp->f_min)*(n-
1)/(double)(eqp->N-1)));
        wc_n=2*M_PI*fc_n;
        bw_n=wc_n*(sqrt(eqp->R)-1.0/sqrt(eqp->R))/wcross;


          Tpw=2.0/bw_n*tan(bw_n/2.0*T);
          cw=cos(Tpw/2.0*sqrt(4*wc_n*wc_n+1))/cos(Tpw/2.0);
          tbw=Tpw*bw_n;
        c_m=cos(M_PI*(0.5-0.5/M));

        a[0]=4+4*c_m*tbw+tbw*tbw;
        a[1]=a[2]=1.0/a[0];
        a[1]*=2*tbw*tbw-8;
        a[2]*=a[0]-8*c_m*tbw;

        b[0]=b[1]=b[2]=tbw*v/a[0];
        b[0]*=2*tbw+4*c_m+tbw*v;
        b[1]*=2*tbw*(v+2);
        b[2]*=2*tbw-4*c_m+tbw*v;

        d=b[0]+1;
        eqp->c[n][0]=cw*(1-a[1]);
        eqp->c[n][1]=cw;
        eqp->c[n][2]=cw*(b[1]-a[1]*b[0]);
        eqp->c[n][3]=b[1]-a[1]*b[0];
        eqp->c[n][4]=-a[1];
        eqp->c[n][5]=cw*(b[2]-a[2]*b[0]);
        eqp->c[n][6]=-a[2]*cw;
        eqp->c[n][7]=b[2]-a[2]*b[0];
        eqp->c[n][8]=-a[2];
        eqp->c[n][9]=d;
    }
}

void calcArgs(bool isfmin, bool isNbands, double *fmin, unsigned *Nbands,
double *octave, double FN)
{
    double R;

    if (isfmin && isNbands){
        R=pow(FN / *fmin, 1.0 / *Nbands);
        *octave=log(R)/log(2);
```

```
        return;
    }
    if (isNbands){
        R=pow(2,*octave);
        *fmin=FN/pow(R,*Nbands);
        return;
    }
    R=pow(2,*octave);
    *Nbands=floor(log(FN/ *fmin)/log(R));

}

int readParFromStr(char *str, unsigned N, double *out)
{
    unsigned i;
    char *s=str, *end=NULL;

    if (strlen(str) < 2)
        return -1;

    if (*s!='(')
        return -1;

    s++;
    i=0;
    while (*s!='\0' && i<N){
        out[i]=strtod(s, &end);
        if(!end || s==end)
            return -1;

        if (fabs(out[i])>1)
            return -1;

        s=end;
        if(*s=='\0' || (*s!=';' && *(s+1)!='\0') || (*s!=')' &&
*(s+1)=='\0'))
            return -1;

        s++;
        i++;
    }

    for (; i<N; i++)
        out[i]=0;

    return 0;

}

static int eqpro_message_handler(const char *object_path, const char
*message, char *message_parameters, char **response, void *ud) {

    struct userdata *u;

    void *state = NULL, *state2=NULL;
```

```
    char *startpos = NULL, *fullpath_str;
    double arg_d;
    int64_t arg_i;
    pa_message_param* param;
    int i;

    pa_assert(u = (struct userdata*)ud);
    pa_assert(message);
    pa_assert(response);

    fullpath_str = pa_sprintf_malloc("%s/%d",MODULE_MSG_PATH, u->module-
>index);
    pa_assert(pa_safe_streq(object_path, fullpath_str));
    pa_xfree(fullpath_str);

    if(pa_streq(message, "sliderchange")) {

        if(pa_message_param_split_list(message_parameters, &startpos,
NULL, &state) <= 0)
            return -PA_ERR_NOTIMPLEMENTED;

        if(pa_message_param_read_double(startpos, &arg_d, &state2) <= 0)
            return -PA_ERR_NOTIMPLEMENTED;

        if(pa_message_param_read_int64(startpos, &arg_i, &state2) <= 0)
            return -PA_ERR_NOTIMPLEMENTED;

        if(arg_i < 0 || arg_i >= u->eqp.N) {
            *response = pa_xstrdup("Cursor doesn't exists");
            return -PA_ERR_NOTIMPLEMENTED;
        }

        if(fabs(arg_d) > 1.0) {
            *response = pa_xstrdup("Cursor value out of range");
            return -PA_ERR_NOTIMPLEMENTED;
        }

        u->eqp.par[arg_i]=arg_d;

        //pa_log(pa_sprintf_malloc("Change par %d to %f",(int)arg_i,
arg_d));

        *response=pa_xstrdup("OK");

        return PA_OK;
    }

    if(pa_streq(message, "dialchange")) {

        if(pa_message_param_read_double(message_parameters, &arg_d,
&state) <= 0)
            return -PA_ERR_NOTIMPLEMENTED;

        if(arg_d < 0) {
            *response = pa_xstrdup("K value must be positive");
```

```
            return -PA_ERR_NOTIMPLEMENTED;
        }

        u->eqp.K=arg_d;

        *response=pa_xstrdup("OK");

        return PA_OK;

    }

    if(pa_streq(message, "getinfo")) {
        param = pa_message_param_new();
        pa_message_param_begin_list(param);
        pa_message_param_write_int64(param, u->eqp.N);
        pa_message_param_write_double(param, u->eqp.f_min,32);
        pa_message_param_write_double(param, u->eqp.DB,32);
        pa_message_param_write_double(param, u->eqp.R, 32);
        pa_message_param_write_double(param, u->eqp.K, 32);
        for (i=0; i<u->eqp.N; i++)
            pa_message_param_write_double(param, u->eqp.par[i],32);
        pa_message_param_end_list(param);

        *response=pa_message_param_to_string(param);

        return PA_OK;
    }


    return -PA_ERR_NOTIMPLEMENTED;
}
```

Codice sorgente 2: module-eqpro-sink.c

C. Codice sorgente dell'interfaccia grafica

```
QT += qml quick

CONFIG += c++11

SOURCES += main.cpp

RESOURCES += qml.qrc

# Additional import path used to resolve QML modules in Qt Creator's code
model
QML_IMPORT_PATH =

# Additional import path used to resolve QML modules just for Qt Quick
Designer
QML_DESIGNER_IMPORT_PATH =

# The following define makes your compiler emit warnings if you use
# any feature of Qt which as been marked deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain
version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the
APIs deprecated before Qt 6.0.0

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

LIBS += -lpulse

HEADERS += \
    guimanager.h \
    pulsedriver.h
```

Codice sorgente 3: eqpro_gui.pro

```
import QtQuick 2.7
import QtQuick.Window 2.3
import QtQuick.Controls 2.0
import QtQuick.Layouts 1.0

Window {
    id: mainWindow
    objectName: "mainWindow"
    visible: true
```

```
    width: 640
    height: 480
    title: qsTr("EqualizerPro")

    Loader {
        id: pageloader
        objectName: "pageloader"

        source: "Page1Form.qml"
        anchors.fill: parent
    }

}
```

Codice sorgente 4: main.qml

```
import QtQuick 2.7
import QtQuick.Controls 2.0
import QtQuick.Layouts 1.0

Item {

    id: page1
    objectName: "page1"

    signal dialChange(double val);

    property bool presetIsChanging: false
    property alias dialvalue: dial.value

    ColumnLayout {
        anchors.fill: parent

        Flickable {

            id: flickable
            Layout.fillHeight: true
            Layout.fillWidth: true
            focus: true

            contentHeight: height
            contentWidth: width*slidersRow.nBands/10

            ScrollBar.horizontal: ScrollBar {
                parent: flickable.parent
                anchors.left: flickable.left
                anchors.right:  flickable.right
                anchors.bottom: flickable.bottom

            }
            flickableDirection: Flickable.HorizontalFlick

            Layout.alignment: Qt.AlignHCenter | Qt.AlignVCenter
            RowLayout {
```

```
                objectName: "slidersRow"
                id: slidersRow

                height: parent.height
                width: page1.width/10*nBands

                property int nBands: 10
                property double fmin: 1000.0
                property double dB: 12
                property double r: 0.25;

                signal sliderChange(double val, int idx)

                Component.onCompleted: {
                    //redrawSlider();
                }

                function inSliderChanged(val,idx) {
                    sliderChange(val, idx)

                    if(!presetIsChanging)
                        comboBoxPresets.currentIndex=0;
                }

                function redrawSlider(par) {
                    var i;

                    for (i=0; i<slidersRow.children.length; i++) {
                       slidersRow.children[i].destroy();

                    }

                    var sli=Qt.createComponent("Eqpro_slider.qml");
                    var fmax=fmin*Math.pow(r,nBands-1);
                    for (i=0; i<nBands; i++) {
                        var
f=Math.round(Math.exp(Math.log(fmin)+Math.log(fmax / fmin)*i/(nBands-1)))
                        var sli_i=sli.createObject(slidersRow,{
                                                   "id": "slider_"+i,
                                                   "objectName": "slider_"+i,
                                                   "Layout.alignment":
Qt.AlignHCenter | Qt.AlignVCenter,
                                                   "freq": f,
                                                   "sliderIdx": i,
                                                   "val": par[i]
                                               });

sli_i.inSliderChange.connect(slidersRow.inSliderChanged);
                    }

                }

            }

        }
```

```
RowLayout {
    id: rowLayout1
    width: 100
    height: 100
    Layout.columnSpan: 1
    spacing: 5
    Layout.alignment: Qt.AlignHCenter | Qt.AlignVCenter
    Layout.rowSpan: 1
    Layout.fillWidth: true
    Layout.maximumHeight: 100
    Layout.maximumWidth: 65535

    Text {
        text: qsTr("Preset")
        fontSizeMode: Text.FixedSize
        scale: 1
        Layout.fillWidth: false
        Layout.alignment: Qt.AlignRight | Qt.AlignVCenter
        horizontalAlignment: Text.AlignHCenter
        font.pixelSize: 15
        Layout.leftMargin: 10
    }

    ComboBox {
        id: comboBoxPresets
        scale: 0.8
        Layout.minimumWidth: 300
        Layout.fillWidth: false

        model: ListModel {
            id: presets
            ListElement { text: "Custom" }
            ListElement { text: "Flat" }
        }

        onCurrentIndexChanged: {
            var preset = presets.get(currentIndex).text

            presetIsChanging = true;
            if (preset === "Flat") {
                for (var i=0; i<slidersRow.nBands; i++)
                    slidersRow.children[i].val=0;
            }

            presetIsChanging = false;
        }

    }
    GridLayout {
        id: gridLayout
        width: 100
        height: 100
        transformOrigin: Item.Center
        Layout.alignment: Qt.AlignHCenter | Qt.AlignVCenter
```

```
                columns: 2
                Layout.fillWidth: true

                Text {
                    id: text1
                    text: qsTr("Gain:")
                    font.pixelSize: 12
                }

                TextInput {
                    id: textInput
                    width: 80
                    height: 20
                    text: qsTr("%1 dB".arg(slidersRow.dB))
                    Layout.alignment: Qt.AlignRight | Qt.AlignTop
                    font.pixelSize: 12
                }

                Text {
                    text: qsTr("Bands:")
                }

                Text {
                    id: text3
                    text: slidersRow.nBands.toString()
                    font.pixelSize: 12
                }
            }

            Dial {
                id: dial
                value: 0
                from: 0
                to: 1
                Layout.margins: 5
                Layout.alignment: Qt.AlignRight | Qt.AlignVCenter
                Layout.fillHeight: true
                Layout.fillWidth: false

                onValueChanged: {
                    dialChange((Math.pow(10,dial.value/20.0)-
1)/0.12201845430196343559103 89) //(10^(1/20)-1)

                }
            }


        }

    }

}
```

Codice sorgente 5: Page1Form.qml

```
import QtQuick 2.7
import QtQuick.Controls 2.0
import QtQuick.Layouts 1.0

Item {
    width: 400
    height: 400

    ColumnLayout {
        anchors.fill: parent

        Text {
            Layout.alignment: Qt.AlignBottom
            anchors.horizontalCenter: parent.horizontalCenter
            text: qsTr("Pulse module eqpro seems to be not running")
            font.pixelSize: 12
            Layout.margins: 10
        }

        Button {
            objectName: "retryButton"
            Layout.alignment: Qt.AlignTop
            anchors.horizontalCenter: parent.horizontalCenter
            id: button
            text: qsTr("Retry")
            Layout.margins: 10
        }
    }
}
```

Codice sorgente 6: Page0CheckModuleForm.qml

```
import QtQuick 2.7
import QtQuick.Controls 2.0
import QtQuick.Layouts 1.0

Item {
    width: 400
    height: 400

    ColumnLayout {
        anchors.fill: parent

        Text {
            Layout.alignment: Qt.AlignBottom
            anchors.horizontalCenter: parent.horizontalCenter
            text: qsTr("There are more eqpro modules running, choose which
module you have to control")
            font.pixelSize: 12
            Layout.margins: 10
        }

        ComboBox {
            id: comboModules
```

```
            objectName: "comboModules"
            Layout.alignment: Qt.AlignTop
            anchors.horizontalCenter: parent.horizontalCenter
            Layout.margins: 10

            function selectedMod(index) {
                return parseInt(comboModules.model[index]);

            }
        }
    }
}
```

Codice sorgente 7: Page05SelectConnection.qml

```
import QtQuick 2.7
import QtQuick.Controls 2.0
import QtQuick.Layouts 1.0

Item {

    Layout.fillHeight: true
    Layout.fillWidth: true

    property double freq: 0.0
    property int sliderIdx: 0
    property alias val: slider.value

    signal inSliderChange(double val, int idx);

        ColumnLayout {
            id: columnLayout
            anchors.fill: parent
            //property int freq

            Slider {
                id: slider
                clip: false
                from: -1
                to: 1
                Layout.fillHeight: true
                Layout.fillWidth: true
                orientation: Qt.Vertical
                value: 0

                onValueChanged: {
                     textF.text=value.toFixed(3).toString();
                     inSliderChange(value, sliderIdx);
                }

            }

            Text {
                id: freqstr
```

```
                text: "%1 Hz".arg(freq)
                Layout.alignment: Qt.AlignHCenter | Qt.AlignVCenter
                horizontalAlignment: Text.AlignHCenter
                font.pixelSize: 12
            }

            TextField {
                id: textF
                text: qsTr("0.0")
                Layout.maximumWidth: 70
                scale: 0.8
                horizontalAlignment: Text.AlignHCenter
                font.letterSpacing: 0
                Layout.alignment: Qt.AlignHCenter | Qt.AlignVCenter
                inputMethodHints: Qt.ImhFormattedNumbersOnly
                onTextChanged: {
                    slider.value=parseFloat(text);
                }
            }

            property alias value: slider.value

        }

}
```

Codice sorgente 8: Eqpro_slider.qml

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>

#include "pulsedriver.h"
#include "guimanager.h"

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.load(QUrl(QLatin1String("qrc:/main.qml")));

    GuiManager gm(&engine);
    QObject::connect(&app,
SIGNAL(applicationStateChanged(Qt::ApplicationState)),
            &gm, SLOT(AppStateChanged(Qt::ApplicationState)));

    return app.exec();
}
```

Codice sorgente 9: main.cpp

```
#ifndef GUIMANAGER_H
#define GUIMANAGER_H

#include <QObject>
#include <QtDebug>
#include <QQmlApplicationEngine>

#include "pulsedriver.h"

#define EQGUI_DEBUG

class GuiManager : public QObject
{
    Q_OBJECT

private:
    enum class GUIstate
    {
        connected,
        disconnected,
        chooseconnection
    };

    QQmlApplicationEngine* engine;
    PulseDriver pd;

    GUIstate state;
    QObject *pageloader;

    QObject* sliderRow;
    QObject *retryButton;

    void SetState(GUIstate s)
    {
        if(state==s)
            return;

        QObject* root =engine->rootObjects().first();
        QVector<int> mds;
        QObject* combomod;
        QStringList model;

        switch (s)
        {
        case GUIstate::disconnected:
            pageloader->setProperty("source",
QUrl(QLatin1String("qrc:/Page0CheckModuleForm.qml")));

            retryButton=root->findChild<QObject*>("retryButton");
            connect(retryButton,
SIGNAL(clicked()),this,SLOT(RetryButtonPressed()));

            break;

        case GUIstate::chooseconnection:
```

```
            pageloader->setProperty("source",
QUrl(QLatin1String("qrc:/Page05SelectConnection.qml")));

            combomod=root->findChild<QObject*>("comboModules");

            mds=(pd.GetEqproModules());
            foreach (auto m, mds)
                model<<QString::number(m);

            combomod->setProperty("model", QVariant::fromValue(model));


connect(combomod,SIGNAL(activated(int)),this,SLOT(EqproModuleChosen(int)))
;

            break;

        case GUIstate::connected:
             pageloader->setProperty("source",
QUrl(QLatin1String("qrc:/Page1Form.qml")));

             QObject* page1=root->findChild<QObject*>("page1");

             PulseDriver::Eqinfo info = pd.RequireEqInfo();
             sliderRow=root->findChild<QObject*>("slidersRow");

             sliderRow->setProperty("nBands",info.nBands);
             sliderRow->setProperty("fmin",info.fmin);
             sliderRow->setProperty("r",info.r);
             sliderRow->setProperty("dB",info.dB);

             page1->setProperty("dialvalue",info.K);

             QVariant ret_x;
             QMetaObject::invokeMethod(sliderRow, "redrawSlider",
Q_RETURN_ARG(QVariant, ret_x), Q_ARG(QVariant,
QVariant::fromValue(info.par)));

             //pd.RegisterModuleDiconnectCallback();

             connect(sliderRow, SIGNAL(sliderChange(double,int)),this,
SLOT(sliderChanged(double,int)));

connect(page1,SIGNAL(dialChange(double)),this,SLOT(dialChanged(double)));
             break;

        }
        state=s;
    }

    void EstimateState()
    {
        QVector<int> mds=pd.GetEqproModules();

        if (mds.length() == 1)
```

```
        {
            if (pd.getModuleNum() == -1 || state != GUIstate::connected)
            {
                pd.setModuleNum(mds.first());
                SetState(GUIstate::connected);
            }
            else if(pd.getModuleNum() == mds.first() && state !=
GUIstate::connected)
                SetState(GUIstate::connected);
            else
                SetState(GUIstate::disconnected);
        }
        else if (mds.length() == 0)
            SetState(GUIstate::disconnected);
        else
            SetState(GUIstate::chooseconnection);
    }

public:
    explicit GuiManager(QQmlApplicationEngine* engine):
        engine(engine)
    {
        QObject* root =engine->rootObjects().first();
        pageloader=root->findChild<QObject*>("pageloader");

        EstimateState();


connect(&pd,SIGNAL(module_seems_disconnected()),this,SLOT(ModuleSeemsDisco
nnect()));

    }

    virtual ~GuiManager() {}

signals:

public slots:
    void sliderChanged(double val, int idx)
    {
        pd.SendSliderChangeMsg(val,idx);
    }

    void dialChanged(double val)
    {
        pd.SendDialChangeMsg(val);
    }

    void RetryButtonPressed()
    {
        EstimateState();
    }

    void EqproModuleChosen(int n)
    {
```

```
        QObject* root =engine->rootObjects().first();
        QObject* combomod=root->findChild<QObject*>("comboModules");

        QVariant ret_x;
        QMetaObject::invokeMethod(combomod, "selectedMod",
Q_RETURN_ARG(QVariant, ret_x), Q_ARG(QVariant, QVariant::fromValue(n)));

        try {
            pd.setModuleNum(ret_x.toInt());
        } catch (std::logic_error){
            SetState(GUIstate::disconnected);
            return;
        }

        SetState(GUIstate::connected);
    }

    void ModuleSeemsDisconnect()
    {
        EstimateState();
    }

    void AppStateChanged(Qt::ApplicationState as)
    {
        qDebug()<<"a state changed " << as;
        if (as ==Qt::ApplicationActive && state == GUIstate::connected)
        {
            QObject* root =engine->rootObjects().first();
            PulseDriver::Eqinfo info = pd.RequireEqInfo();

            sliderRow->setProperty("nBands",info.nBands);
            sliderRow->setProperty("fmin",info.fmin);
            sliderRow->setProperty("r",info.r);
            sliderRow->setProperty("dB",info.dB);

            QObject* page1=root->findChild<QObject*>("page1");
            page1->setProperty("dialvalue",info.K);

            QVariant ret_x;
            QMetaObject::invokeMethod(sliderRow, "redrawSlider",
Q_RETURN_ARG(QVariant, ret_x), Q_ARG(QVariant,
QVariant::fromValue(info.par)));
        }

    }


};

#endif // GUIMANAGER_H
```

Codice sorgente 10: guimanager.h

```
#ifndef PULSEDRIVER_H
#define PULSEDRIVER_H


extern "C"
{
#include <pulse/pulseaudio.h>
#include <pulse/message-params.h>
}

#include <iostream>
#include <stdexcept>
#include <string>
#include <QList>
#include <QtDebug>
#include <QObject>
#include <QStringList>

class PulseDriver;

enum class Actions{
    getSinks,
    getModules,
    sendMsg,
    subscrive //fixme
};

struct ud_t
{
    void *rawout;
    PulseDriver* pd_context;
    pa_mainloop *m;
    Actions a;
    pa_context* c;
    pa_mainloop_api *mapi;
    pa_operation* pa_op;
};

class PulseDriver : public QObject
{
    Q_OBJECT

public:

    struct Eqinfo
    {
        int nBands;
        double fmin;
        double dB;
        double r;
        double K;
        QVector<double> par;
    };

    PulseDriver(QObject* parent=0):
```

```
        QObject(parent), modulenum(-1)
    {}

    ~PulseDriver()
    {}

    void SendSliderChangeMsg(double val, int idx)
    {
        pa_message_param *param = pa_message_param_new();
        pa_message_param_begin_list(param);
        pa_message_param_write_double(param,val,32);
        pa_message_param_write_int64(param,idx);
        pa_message_param_end_list(param);

        Message
mx("sliderchange",pa_message_param_to_string(param),MessageDest());
        SendMessage(mx);

        if(mx.response != "OK")
            emit module_seems_disconnected();
    }

    void SendDialChangeMsg(double val)
    {
        pa_message_param *param = pa_message_param_new();
        pa_message_param_write_double(param,val,32);

        Message
mx("dialchange",pa_message_param_to_string(param),MessageDest());
        SendMessage(mx);

        if(mx.response != "OK")
            emit module_seems_disconnected();

    }

    Eqinfo RequireEqInfo()
    {
        Eqinfo info;

        Message mx( "getinfo", "", MessageDest());
        SendMessage(mx);

        void *state=NULL, *state2=NULL;
        //pa_message_param* param;
        char* startpos=NULL;

        QByteArray resp = mx.response.toLocal8Bit();
        pa_message_param_split_list(resp.data(), &startpos, NULL, &state);

        int64_t n;
        pa_message_param_read_int64(startpos, &n,&state2);
        info.nBands=n;
        pa_message_param_read_double(startpos, &info.fmin, &state2);
        pa_message_param_read_double(startpos, &info.dB, &state2);
```

```
        pa_message_param_read_double(startpos, &info.r, &state2);
        pa_message_param_read_double(startpos, &info.K, &state2);
        for (int i=0; i<info.nBands; i++)
        {
            double x;
            pa_message_param_read_double(startpos, &x, &state2);
            info.par.append(x);
        }

        return info;
    }

    QVector<int> GetEqproModules()
    {
        QList<Module> modules=GetModule();
        QVector<int> out;

        foreach (auto s, modules)
        {
            if(s.name == DRIVERNAME)
                out.append(s.idx);
        }

        return out;
    }

    // doesn't work
    void RegisterModuleDiconnectCallback()
    {
        MakeOperation<void*>(Actions::subscrive);
    }

    int getModuleNum() const { return modulenum; }
    void setModuleNum(unsigned n)
    {
        QList<Module> ml=GetModule();
        foreach (auto m, ml)
        {
            if(m.idx == int(n) && m.name == DRIVERNAME)
            {
                modulenum = n;
                return;
            }
        }
        throw std::logic_error("Module number is not correct");
    }

private:
    const QString DRIVERNAME="module-eqpro-sink";
    const QString MSGDEST="/modules/eqpro";

    int modulenum;

    struct Sink
    {
```

```
        Sink(const QString &name="", const QString &driver="", int idx=0):
            name(name), driver(driver), idx(idx)
        {}

        QString name;
        QString driver;
        int idx;
    };

    struct Message
    {
        Message(const QString& message="", const QString& param="", const
QString& to="", const QString& response=""):
            to(to), message(message), param(param), response(response)
        {}

        QString to;
        QString message;
        QString param;
        QString response;
    };

    struct Module
    {
        Module(const QString &name="", int idx=0):
            name(name), idx(idx)
        {}

        QString name;
        int idx;
    };

    QString MessageDest()
    {
        if (modulenum<0)
            throw std::logic_error("No module selected");

        return MSGDEST + QString("/") + QString::number(modulenum);
    }

    template<typename T>
    T MakeOperation(Actions a, T x=T())
    {
        pa_context *context = nullptr;
        pa_mainloop *m = nullptr;
        pa_mainloop_api *mapi = nullptr;

        if (!(m = pa_mainloop_new()))
            throw std::runtime_error("pa_mainloop_new() failed.");

        mapi = pa_mainloop_get_api(m);

        if (!(context = pa_context_new(mapi,"eqpro_gui")))
            throw std::runtime_error("pa_context_new() failed.");
```

```
        ud_t ud={(void*)&x, this, m, a, context, mapi, nullptr};

        pa_context_set_state_callback(context, context_state_callback,
(void*)&ud);
        pa_context_connect(context, NULL, (pa_context_flags)0, NULL);

        pa_mainloop_run(m, NULL);

        pa_context_unref(context);
        pa_mainloop_free(m);

        return x;
    }

    QList<Sink> GetSinks()
    {
        return MakeOperation<QList<Sink>>(Actions::getSinks);
    }

    QList<Module> GetModule()
    {
        return MakeOperation<QList<Module>>(Actions::getModules);
    }

    void SendMessage(Message& message)
    {
        message=MakeOperation<Message>(Actions::sendMsg,message);
    }

    static void get_sink_info_callback(pa_context *c, const pa_sink_info
*l, int is_last, void *userdata)
    {
        qDebug()<<"sink callback";
        ud_t* ud=(ud_t*)userdata;

        QList<Sink>* list=(QList<Sink>*)ud->rawout;

        if(is_last > 0)
            return;

        list->append(Sink(l->name,l->driver,l->index));
    }

    static void get_module_info_callback(pa_context *c, const
pa_module_info *l, int is_last, void *userdata)
    {
        qDebug()<<"module callback";
        ud_t* ud=(ud_t*)userdata;

        QList<Module>* list=(QList<Module>*)ud->rawout;

        if(is_last > 0)
            return;

        list->append(Module(l->name,l->index));
```

```
    }

    static void get_content_string_callback(pa_context *c, int success,
char *response, void *userdata)
    {
         qDebug()<<"get_content_string_callback";

         ud_t* ud=(ud_t*)userdata;
         Message* mx =(Message*)ud->rawout;

         if(!success)
         {
             qDebug()<<"not success, send emit";
             ud->mapi->quit(ud->mapi,0);
             emit ud->pd_context->module_seems_disconnected();
             return;
         }

         if(response)
         {
             mx->response=response;
             qDebug()<<mx->response;
         }
    }

    static void context_subscribe_callback(pa_context *c,
pa_subscription_event_type_t t, uint32_t idx, void *userdata)
    {
        qDebug()<<"context_subscribe_callback";

        ud_t* ud=(ud_t*)userdata;

        if (idx == (int)ud->pd_context->getModuleNum())
        {
            switch (t & PA_SUBSCRIPTION_EVENT_TYPE_MASK)
            {
            case PA_SUBSCRIPTION_EVENT_NEW:
            case PA_SUBSCRIPTION_EVENT_CHANGE:
                break;
            case PA_SUBSCRIPTION_EVENT_REMOVE:
                emit ud->pd_context->module_seems_disconnected();
            }

        }

    }

    static void context_state_callback(pa_context *c, void *userdata)
    {
        assert(c);

        ud_t* ud=(ud_t*)userdata;
        Message* mx;
        qDebug()<<"context state";
```

```
        switch (pa_context_get_state(c))
        {
        case PA_CONTEXT_CONNECTING:
        case PA_CONTEXT_AUTHORIZING:
        case PA_CONTEXT_SETTING_NAME:
            break;

        case PA_CONTEXT_READY:
            qDebug()<<"Ready";
            switch(ud->a)
            {
            case Actions::getSinks:
                ud->pa_op = pa_context_get_sink_info_list(c,
get_sink_info_callback, userdata);
                break;
            case Actions::getModules:
                ud->pa_op = pa_context_get_module_info_list(c,
get_module_info_callback, userdata);
                break;
            case Actions::sendMsg:
                mx=(Message*)ud->rawout;
                ud->pa_op = pa_context_send_message_to_object(c,
                                                              mx-
>to.toLocal8Bit().constData(),
                                                              mx-
>message.toLocal8Bit().constData(),
                                                              mx-
>param.toLocal8Bit().constData(),

get_content_string_callback,
                                                              userdata);
                break;
            case Actions::subscrive:
                pa_context_set_subscribe_callback(c,
context_subscribe_callback, userdata);
                ud->pa_op = pa_context_subscribe(c,

(pa_subscription_mask)(PA_SUBSCRIPTION_MASK_SINK|

PA_SUBSCRIPTION_MASK_SOURCE|

PA_SUBSCRIPTION_MASK_SINK_INPUT|

PA_SUBSCRIPTION_MASK_SOURCE_OUTPUT|

PA_SUBSCRIPTION_MASK_MODULE|

PA_SUBSCRIPTION_MASK_CLIENT|

PA_SUBSCRIPTION_MASK_SAMPLE_CACHE|

PA_SUBSCRIPTION_MASK_SERVER|

PA_SUBSCRIPTION_MASK_CARD),
                                                    NULL,userdata);
```

```
                break;
            }

            pa_operation_set_state_callback(ud->pa_op,
get_operation_state, userdata);
            break;

        case PA_CONTEXT_TERMINATED:
            qDebug()<<"terminated";
            break;

        case PA_CONTEXT_FAILED:
            ud->mapi->quit(ud->mapi,0); //stop mainloop
            emit ud->pd_context->module_seems_disconnected();
            break;
        default:
            throw std::runtime_error("Connection failure: " +
std::string(pa_strerror(pa_context_errno(c))));

        }

    }

    static void get_operation_state(pa_operation* pa_op, void* userdata)
    {
        ud_t* ud=(ud_t*)userdata;

        qDebug()<<"op state "<< (int)ud->a;
        switch(pa_operation_get_state(pa_op))
        {
        case PA_OPERATION_RUNNING:
            qDebug()<<"Op running";
            break;

        case PA_OPERATION_CANCELLED:
            qDebug()<<"Op canc";
        case PA_OPERATION_DONE:
            qDebug()<<"Op done canc";
            pa_operation_unref(ud->pa_op);
            ud->mapi->quit(ud->mapi,0); //stop mainloop
            break;

        }
    }

private slots:

signals:
    void module_seems_disconnected();

};

#endif // PULSEDRIVER_H
```

Codice sorgente 11: pulsedriver.h

BIBLIOGRAFIA

1. ALAN V. OPPENHEIM, RONALD W. SCHAFER, JOHN R. BUCK
   Discrete-time Signal Processing, 1999
2. MARTIN HOLTERS, UDO ZÖLZER
   Graphic Equalizer Design Using Higher-Order Recursive Filters, 2006
3. AMERICAN NATIONAL STANDARDS INSTITUTE
   ANSI S1.11: Specification for Octave, Half-Octave, and Ðhird Octave Band Filter Sets, 2009
4. PHILLIP A. REGALIA, SANJIT K. MITRA, P. P. VAIDYANATHAN
   The Digital All-Pass Filter: A Versatile Signal Processing Building Block, 1988
5. FEDERICO FONTANA, MATTI KARJALAINEN
   A Digital Bandpass/Bandstop Complementary Equalization Filter with Independent Tuning Characteristics, 2003
6. JUSSI RÄMÖ, VESA VÄLIMÄKI, BALÁZS BANK
   High-Precision Parallel Graphic Equalizer, 2014

SITOGRAFIA

1. freedesktop.org
   https://www.freedesktop.org/wiki/
2. Georg Chini
   https://patchwork.freedesktop.org/project/pulseaudio/list/?submitter=15697
3. module-virtual-sink
   https://github.com/pulseaudio/pulseaudio/blob/master/src/modules/module-virtual-sink.c
4. Prof. Pier Luca Montessoro
   http://www.montessoro.it/