UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA TRIENNALE IN INFORMATICA MUSICALE

MODELLAZIONE E IMPLEMENTAZIONE DEL DISPOSITIVO TALK BOX

Relatore: Prof. Luca Andrea Ludovico
Correlatore: Dott. Giorgio Presti

Tesi di Laurea di:
Daniele Bianchi
Matr. Nr. 841566

ANNO ACCADEMICO 2015-2016

Indice

Capitolo 1

Introduzione

1.1 Il Talk Box

Il Talk Box è un dispositivo analogico che permette di conferire intelligibilità vocale ad un qualsiasi suono (nella maggior parte dei casi proveniente da uno strumento musicale) attraverso l'apparato fonatorio del musicista. Ha cominciato a diffondersi e a guadagnare popolarità intorno agli anni Settanta (nonostante la comparsa di un prototipo chiamato "Singing Guitar" nel 1939 ad opera del chitarrista Alvino Rey) grazie al chitarrista Joe Walsh (celebre è il brano "Rocky Mountain Way" del 1973) e, successivamente, al chitarrista Peter Frampton, che comincia a farne uso all'interno dell'album "Frampton", datato 1975. [1]

L'inventore di questo dispositivo è oggetto di disputa, tuttavia Bob Heil viene generalmente accreditato come creatore della versione moderna del Talk Bok. [2] Dalla sua entrata in commercio, moltissimi sono gli artisti che ne hanno fatto uso oppure che hanno ottenuto notorietà grazie ad esso, a cominciare dallo stesso Frampton. In ordine sparso: Roger Troutman, leader del gruppo Zapp, nel brano "So Ruff, So Tuff" datato 1981; Daft Punk nel brano "Around The World" datato 1997; Bon Jovi all'interno del brano "Livin' On A Prayer" del 1986; Aerosmith in "Sweet Emotion" del 1975; Stevie Wonder nel brano "Girl Blue" (1972); P-Thugg, membro del duo Chromeo, in "Rage!" (2004); Alain Maratrat, chitarrista del gruppo Rockets, in "Future Woman" (1977).

Durante la realizzazione di questo lavoro è stato preso in considerazione il modello "HT-1" (Heil Talk Box) prodotto da Dunlop. È la versione moderna del modello originariamente creato da Bob Heil e prodotto da Heil Sound (fondata proprio da Heil nel 1966).

1.2 Come funziona il Talk Box

Il Talk Box, come si è detto, consente di modellare il suono proveniente da uno strumento musicale (tipicamente una chitarra o una tastiera) e di attribuirgli le caratteristiche del parlato. Questa operazione, tuttavia, non viene svolta dal dispositivo stesso, bensì dall'apparato fonatorio umano. La sua funzione, dunque, si riduce a mero vettore, poiché consente al suono proveniente dallo strumento musicale di giungere alla cavità orale del musicista, che agisce come un filtro smorzando alcune frequenze e amplificandone altre (le cosiddette formanti). [3]

L'anima di un Talk Box è costituita da un compression driver[1]. Esso amplifica il suono in ingresso e grazie alla sua risposta in frequenza gli conferisce la sonorità tipica e facilmente riconoscibile di un Talk Box. Alcuni compression driver prodotti da Electro Voice erano molto utilizzati. L'uscita del compression driver è direttamente collegata ad un tubo realizzato con cloruro di polivinile (PVC) di diametro e lunghezza non standard.

Esiste tuttavia una suddivisione tra i diversi modelli di Talk Box. Alcuni (come ad esempio il modello Banshee prodotto da Rocktron) possiedono già al proprio interno preamplificatori e amplificatori che operano sul segnale audio in ingresso, prima di indirizzarlo verso il compression driver, e necessitano quindi di essere semplicemente collegati alla presa elettrica per funzionare; altri, invece, ricevono il segnale già amplificato dalla testata di un amplificatore (e senza quest'ultimo non possono funzionare).

1.3 Talk Box vs. Vocoder

Sebbene l'output del Talk Box possa sembrare simile a quello prodotto attraverso il Vocoder, il risultato viene ottenuto in un modo totalmente diverso. La prima sostanziale differenza consiste nel modo in cui il musicista utilizza i due dispositivi: nel caso del Vocoder è necessario acquisire il segnale vocale attraverso un microfono, mentre nel caso del Talk Box è sufficiente che il musicista muova la bocca come se parlasse ma senza emettere fisicamente alcun suono. Le diverse posizioni della bocca selezionano un preciso set di formanti: l'algoritmo che sarà discusso nel Capitolo 3, inferisce le posizioni della bocca a partire da una traccia vocale.

La seconda differenza consiste nel modo in cui il segnale audio è processato. Diversamente dal Talk Box, che non esegue alcuna operazione sul segnale audio in ingresso, il Vocoder processa i segnali audio (ovvero la voce acquisita tramite microfono, chiamata modulante, e un segnale proveniente, ad esempio, da un sintetizzatore, chiamato portante) attraverso filtri passa-banda, rivelatori di inviluppo e amplificatori [4]. Dato

[1] Un particolare tipo di altoparlante

che sono le caratteristiche estratte dalla modulante a modellare la portante, si deduce che l'ampiezza del segnale in uscita dal Vocoder dipende in una certa misura dall'ampiezza della modulante. Questo fatto rappresenta un'ulteriore differenza rispetto al Talk Box, poiché in questo caso l'ampiezza del segnale finale dipende esclusivamente dall'ampiezza del segnale portante. È importante sottolineare che il Talk Box non ha bisogno di un segnale modulante, bensì è la bocca del musicista che modula fisicamente il suono.

Capitolo 2

Modellazione del Talk Box

2.1 Studio della risposta all'impulso

2.1.1 Il metodo Exponential Sine Sweep (ESS)

Sia S un sistema lineare e tempo invariante (LTI). La risposta all'impulso del sistema S è, per definizione, il segnale in uscita dal sistema quando esso riceve in ingresso l'impulso unitario $\delta(n)$. Conoscendo la risposta all'impulso di S (chiamata $h(n)$) è possibile calcolare il segnale in uscita da S per qualsiasi segnale in ingresso. Nel caso di sistemi che operano su segnali a tempo discreto valgono le seguenti equazioni:

$$h(n) = S(\delta(n)) \tag{1}$$

$$S(x(n)) = \sum_{j=-\infty}^{\infty} h(n-j)x(j) \tag{2}$$

Nel caso invece di sistemi che operano su segnali a tempo continuo valgono invece le equazioni:

$$h(t) = S(\delta(t)) \tag{3}$$

$$S(f(t)) = \int_{-\infty}^{\infty} h(t-x)f(x) \tag{4}$$

Dato che registrare l'output di un sistema che riceve in ingresso il segnale $\delta(n)$ è molto difficile perché altoparlanti e microfoni non sono in grado di riprodurre impulsi di Dirac ideali, è necessario ricorrere ad altre tecniche di misurazione. Lo studio della risposta all'impulso del Talk Box è stato effettuato attraverso il metodo "Exponential Sine Sweep" (ESS) che consente non solo di ricavare la risposta all'impulso, ma anche di separare quest'ultima dagli effetti prodotti dalla distorsione. [5]

È possibile esprimere il segnale in uscita da un qualsiasi sistema con la seguente equazione (si considerano segnali a tempo continuo):

$$S(x(t)) = y(t) = n(t) + f(x(t)) \tag{5}$$

Dove $n(t)$ rappresenta il rumore generato dal sistema e $f(x(t))$ è una funzione che dipende dal segnale in ingresso. Grazie all'assunzione di linearità e tempo-invarianza, $f(x(t))$ può essere espressa come convoluzione tra il segnale in ingresso $x(t)$ e la risposta all'impulso del sistema $h(t)$:

$$f(x(t)) = x(t) * h(t) \tag{6}$$

Il segnale in uscita può quindi essere riscritto come:

$$y(t) = n(t) + x(t) * h(t) \tag{7}$$

La rimozione di $n(t)$ può essere fatta considerando il segnale $\hat{y}(t)$ ottenuto attraverso la media di diversi segnali in uscita $y_1(t) \dots y_n(t)$, a patto che essi siano tutti sincronizzati [5]. La precedente equazione diventa quindi:

$$\hat{y}(t) = x(t) * h(t) \tag{8}$$

A questo punto si considera un filtro inverso $g(t)$ tale che la convoluzione tra $g(t)$ e il segnale in ingresso $x(t)$ dia come risultato l'impulso unitario $\delta(n)$:

$$x(t) * g(t) = \delta(n) \tag{9}$$

L'estrazione della risposta all'impulso del sistema si ottiene quindi eseguendo l'operazione di convoluzione tra l'uscita del sistema $\hat{y}(t)$ e il filtro inverso $g(t)$. Usando le precedenti equazioni si può quindi scrivere:

$$\hat{y}(t) * g(t) = (x(t) * h(t)) * g(t) \tag{10}$$

Sfruttando la proprietà associativa della convoluzione, la definizione di filtro inverso e la definizione di impulso unitario, si giunge al risultato:

$$\hat{y}(t) * g(t) = (x(t) * g(t)) * h(t) = h(t) \tag{11}$$

Il metodo ESS prevede che il segnale in ingresso $x(t)$ sia una sinusoide del tipo $x(t) = \sin f(t)$ la cui frequenza vari nel tempo seguendo un andamento esponenziale secondo la seguente equazione:

$$x(t) = \sin\left(\frac{\omega_1 \cdot T}{\ln \frac{\omega_2}{\omega_1}} \cdot \left(e^{\frac{t}{T} \cdot \ln \frac{\omega_2}{\omega_1}} - 1\right)\right) \tag{12}$$

dove ω_1, ω_2 sono, rispettivamente, la frequenza iniziale e la frequenza finale, T è la durata dello sweep.

Il filtro inverso $g(t)$ non è altro che il segnale $x(t)$ invertito rispetto all'asse temporale, ovvero $x(T-t)$. Si pensi all'impulso unitario $\delta(t)$. Lo spettro di tale segnale contiene, per definizione, tutte le frequenze in un unico istante di tempo. Il segnale $x(t)$ contiene anch'esso tutte le frequenze da ω_1 a ω_2, ma esse sono spalmate su un intervallo temporale di durata T. Ciò significa che all'istante t ($t \leq T$) sarà presente la frequenza ω_t, all'istante $t+1$ la frequenza ω_{t+1} e così via fino all'istante T, in cui è presente la corrispondente frequenza ω_T. Ogni frequenza, quindi, si presenta con un certo ritardo e per fare in modo che questo si annulli (in altre parole, per fare in modo che tutte abbiano lo stesso ritardo), è sufficiente che le frequenze che in $x(t)$ hanno il ritardo maggiore, si presentino prima rispetto alle frequenze con un ritardo minore. Ad esempio, all'instante $t=0$ deve essere presente la frequenza ω_T e al tempo $t=T$ la frequenza $\omega_{t=0}$.

Questo risultato può essere espresso anche in un modo più formale. La teoria dei segnali afferma che, data la funzione $x(t) = \sin(f(t))$, la derivata dell'argomento della funzione sin rappresenta la frequenza istantanea del segnale [5]. La derivata dell'equazione 12 è:

$$\frac{\omega_1 \cdot T}{\ln \frac{\omega_2}{\omega_1}} \cdot e^{\frac{t}{T} \cdot \ln \frac{\omega_2}{\omega_1}} \tag{13}$$

Se il filtro inverso è lo stesso segnale, ma ribaltato rispetto all'asse del tempo, significa che la frequenza iniziale diventa ω_2, mentre la frequenza finale è ω_1. La derivata dell'argomento del sin di questo segnale è uguale a quella precedente, ma con ω_1 e ω_2 invertiti:

$$\frac{\omega_2 \cdot T}{\ln \frac{\omega_1}{\omega_2}} \cdot e^{\frac{t}{T} \cdot \ln \frac{\omega_1}{\omega_2}} \tag{14}$$

La convoluzione tra i due segnali corrisponde, nel dominio delle frequenze, al prodotto delle stesse, quindi:

$$\begin{aligned}
&\frac{\omega_1 \cdot T}{\ln \frac{\omega_2}{\omega_1}} \cdot e^{\frac{t}{T} \cdot \ln \frac{\omega_2}{\omega_1}} \cdot \frac{\omega_2 \cdot T}{\ln \frac{\omega_1}{\omega_2}} \cdot e^{\frac{t}{T} \cdot \ln \frac{\omega_1}{\omega_2}} = \\
&\frac{\omega_1 \cdot \omega_2 \cdot T^2}{\ln \frac{\omega_2}{\omega_1} \cdot \ln \frac{\omega_1}{\omega_2}} \cdot e^{\frac{t}{T} \cdot \ln \frac{\omega_2}{\omega_1}} \cdot e^{\frac{t}{T} \cdot \ln \frac{\omega_1}{\omega_2}} = \\
&\frac{\omega_1 \cdot \omega_2 \cdot T^2}{\ln \frac{\omega_2}{\omega_1} \cdot \ln \frac{\omega_1}{\omega_2}} \cdot e^{\frac{t}{T} \cdot \left(\ln \frac{\omega_2}{\omega_1} - \ln \frac{\omega_2}{\omega_1}\right)} = \\
&\frac{\omega_1 \cdot \omega_2 \cdot T^2}{\ln \frac{\omega_2}{\omega_1} \cdot \ln \frac{\omega_1}{\omega_2}} = k
\end{aligned} \tag{15}$$

k, quindi, è una costante. Operando l'inverso della trasformata di Fourier (sapendo che $k = k \cdot 1$ e quindi $\mathfrak{F}^{-1}(k \cdot 1) = k \cdot \mathfrak{F}^{-1}(1)$) si ottiene $k \cdot \delta(t)$. Si noti che questa

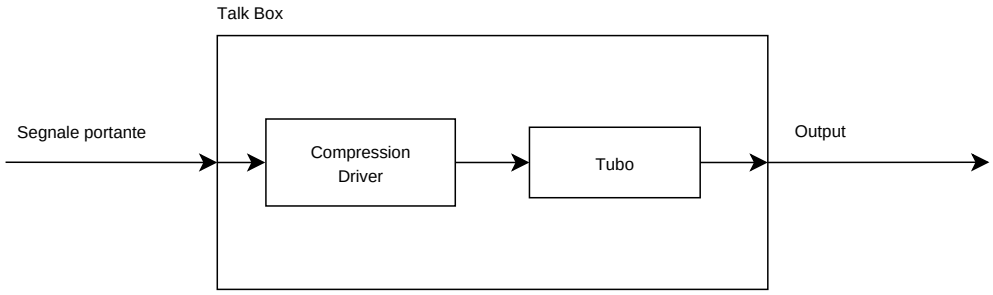


Figura 1: Il Talk Box visto come composizione di sistemi.

proprietà è tipica del segnale ottenuto tramite sweep e non vale necessariamente per qualsiasi segnale.

Il risultato della convoluzione tra $\hat{y}(t)$ e $g(t)$ è una serie di risposte all'impulso ben separate l'una dall'altra (ad una distanza proporzionale al logaritmo della frequenza), facilmente riconoscibili, con la possibilità di essere quindi estratte e studiate individualmente. Tra tutte le risposte, però, solo quella più a destra rappresenta la risposta lineare del sistema, mentre le altre sono prodotte dalle distorsioni armoniche.

2.1.2 Il metodo ESS applicato al Talk Box

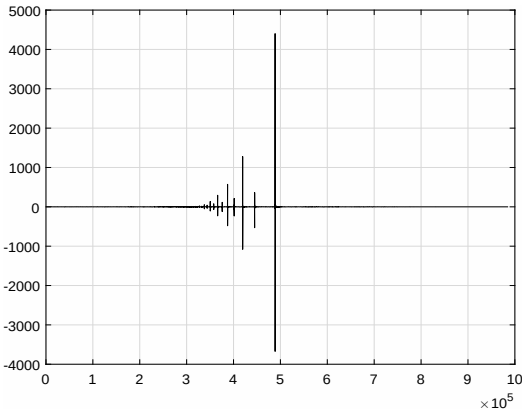
Per riassumere, il procedimento per ricavare la risposta all'impulso lineare di un sistema è composto da tre passaggi fondamentali:

- Sintesi del segnale di sweep secondo l'equazione 12;
- Acquisizione del segnale in uscita dal sistema avente in input il segnale appena sintetizzato;
- Convoluzione tra il segnale acquisito e il segnale originale invertito rispetto all'asse temporale. Il risultato della convoluzione contiene la risposta all'impulso cercata.

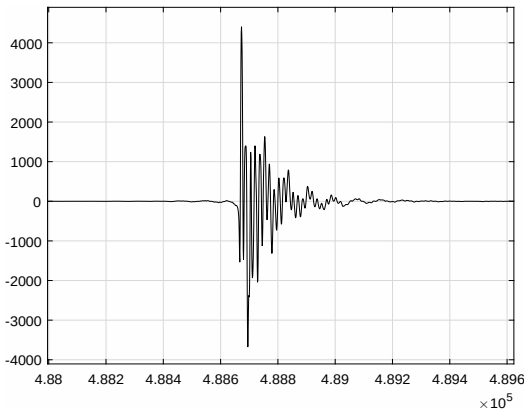
Il segnale originale è stato sintetizzato con l'aiuto del programma Sound Forge, con i seguenti parametri: la frequenza di campionamento è di $44100Hz$, la durata totale T è di $11s$ ($10s$ di sweep e $1s$ di silenzio all'inizio), ω_1 è pari a $20Hz$ mentre ω_2 è $22050Hz$. L'ampiezza del segnale è invece $-6.0dB$.

Il Talk Box può essere considerato come una scatola nera (un segnale entra e un segnale esce) oppure può essere visto come due sistemi in cascata rappresentati dal compression driver e dal tubo (Figura 1). Durante la modellazione, quindi, sono state studiate separatamente la risposta all'impulso del compression driver, del tubo e dell'intero Talk Box.

Per il processing dei segnali è stato utilizzato MATLAB.



(a) Si notino, sulla sinistra, le diverse risposte all'impulso causate dalle distorsioni armoniche.



(b) Particolare della risposta lineare del compression driver.

Figura 2: Risultato della convoluzione tra il segnale originale invertito e il nuovo segnale.

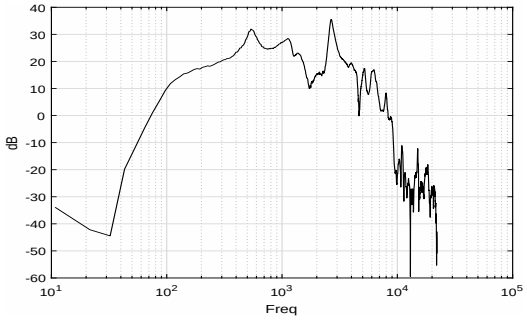
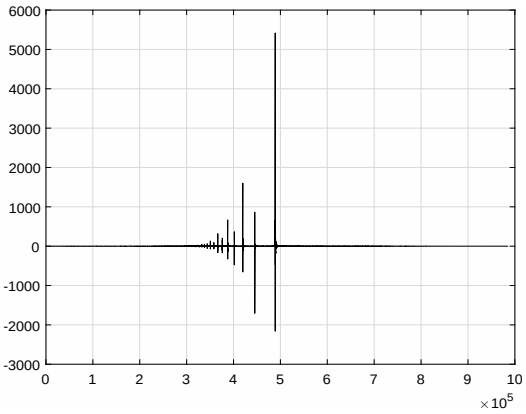


Figura 3: FFT della risposta all'impulso lineare.

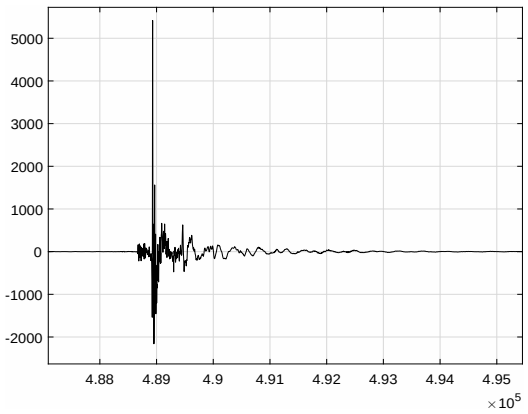
Risposta all'impulso del compression driver

Sono stati acquisiti quattro diversi segnali in uscita dal compression driver, ogni volta distanziando il microfono di $10cm$ dalla posizione precedente, partendo da una distanza iniziale dal driver di $10cm$. Prima di procedere con la convoluzione, i segnali sono stati allineati con le funzioni `xcorr` (per calcolare l'offset, in campioni) e `circshift` (per allinearli in base all'offset). Una volta allineati è possibile calcolare un nuovo segnale $\hat{y}$ come media tra i quattro segnali registrati, operazione che può essere fatta direttamente nel dominio del tempo.

Dopo queste operazioni preliminari è stata eseguita la convoluzione tra il segnale originale invertito rispetto all'asse del tempo e $\hat{y}$. Il risultato è mostrato in Figura 2a. Infine, è stata isolata la risposta più a destra.



(a) Si notino anche in questo caso le diverse risposte all'impulso causate dalle distorsioni armoniche.



(b) Particolare della risposta lineare del Talk Box.

Figura 4: Risultato della convoluzione tra il segnale originale invertito e il nuovo segnale.

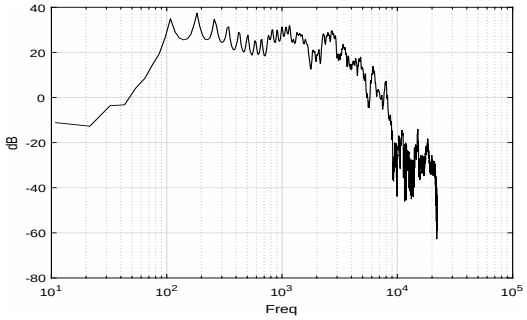


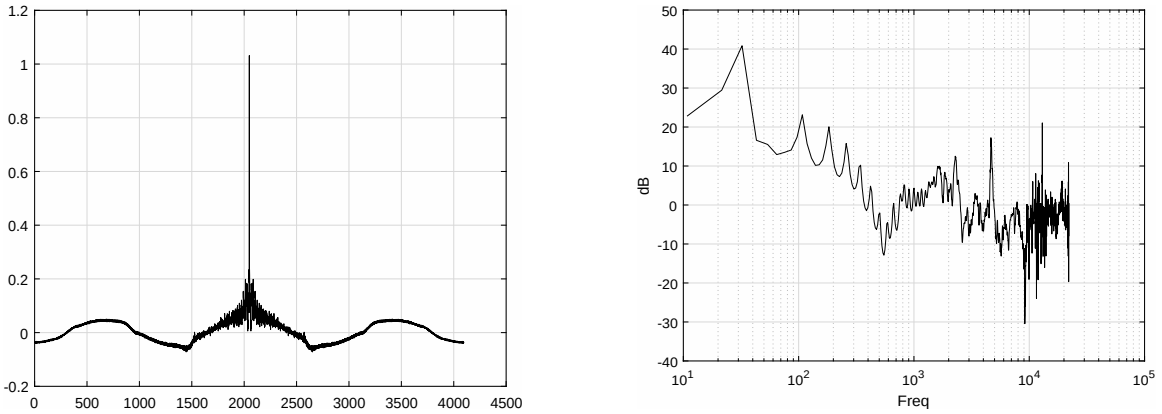
Figura 5: FFT della risposta all'impulso lineare.

Risposta all'impulso del Talk Box

Le medesime operazioni (comprese anche le misurazioni con il microfono) sono state eseguite per calcolare la risposta all'impulso del Talk Box. I risultati sono esposti in Figura 4b e in Figura 5.

Risposta all'impulso del tubo

Il primo approccio usato per calcolare la risposta all'impulso del tubo era basato, come i casi precedenti, sul metodo ESS, considerando il segnale in uscita dal compression driver come segnale originale. I risultati ottenuti, tuttavia, differivano dalla teoria. Non è stato possibile identificare la risposta all'impulso lineare poiché a destra della



(a) Risultato della $IFFT$. (b) Risultato del rapporto tra gli spettri.

Figura 6: Risposta all'impulso del tubo.

presunta risposta lineare erano presenti altre risposte all'impulso (la teoria ammette la presenza di una sorta di "coda" dopo la risposta all'impulso lineare, ma solo se il sistema introduce del riverbero [5], e non è il caso del Talk Box).

Sapendo che il Talk Box può essere considerato come un sistema a cascata, la sua risposta all'impulso nel dominio delle frequenze può essere espressa in questo modo:

$$H_{talkbox} = H_{driver} \cdot H_{tubo} \tag{16}$$

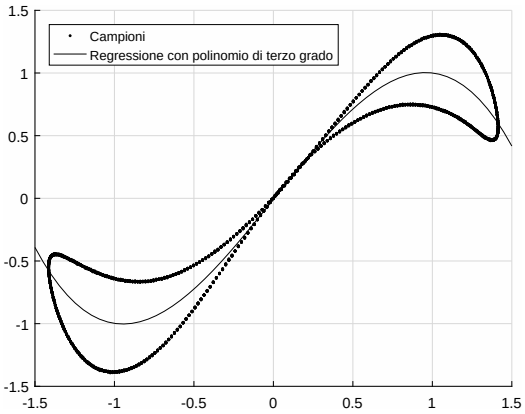
Risolvendo questa equazione si ottiene:

$$H_{tubo} = \frac{H_{talkbox}}{H_{driver}} \tag{17}$$

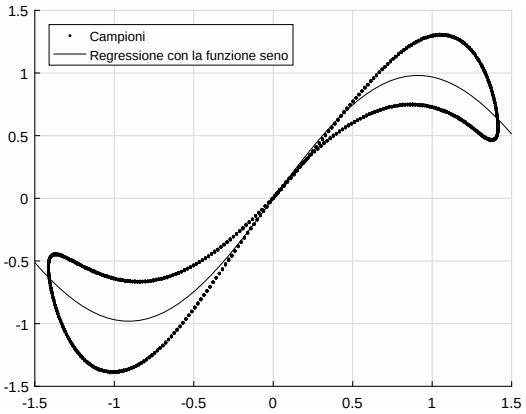
Quindi è sufficiente calcolare il rapporto tra i due spettri e successivamente la trasformata inversa di Fourier. I risultati si trovano in Figura 6

2.2 Calcolo della distorsione

Un sistema reale, a differenza di un sistema ideale, altera il contenuto frequenziale di un qualsiasi segnale, sia secondo la sua funzione di trasferimento, sia attraverso dei fenomeni non lineari. Ad esempio, se si dà in input ad un sistema una sinusoide con una frequenza di $1kHz$, lo spettrogramma del segnale registrato evidenzia la presenza di energia in corrispondenza dei multipli interi della frequenza del segnale in ingresso, quindi a $2kHz$, $3kHz$, $4kHz$ e così via. Questo fenomeno è noto come distorsione armonica.



(a) Risultato dello scatter tra la sinusoide da $1kHz$ e il segnale acquisito in output. La curva è il risultato della regressione polinomiale.



(b) Risultato della regressione non lineare attraverso la funzione sin.

Figura 7: Grafico di dispersione per lo studio della distorsione del driver.

Lo studio della distorsione del compression driver del Talk Box è stato effettuato registrando l'output del sistema avente in input un segnale sinusoidale di frequenza $1kHz$. Successivamente, il segnale originale e il segnale registrato sono stati confrontati attraverso un grafico di dispersione, posizionando il segnale da $1kHz$ sull'asse x. In Figura 7a è mostrato il risultato della sola operazione di scatter. Per capire il tipo di relazione (chiaramente non lineare) tra i campioni audio, sono state tentate due regressioni non lineari. La prima è stata effettuata attraverso un polinomio di terzo grado della forma $ax^3 + bx^2 + cx + d$, dove i coefficienti sono risultati essere:

$$\begin{aligned} a &= -0.583502199316281 \\ b &= 0.009294199220819 \\ c &= 1.582511488646361 \\ d &= -0.008098169340910 \end{aligned} \tag{18}$$

La seconda, invece, è stata effettuata mediante la funzione sin, nella forma $a\sin(2\pi(x+b)\cdot c)$. In questo caso i coefficienti sono:

$$\begin{aligned} a &= 0.980812691278513 \\ b &= -0.005834735575077 \\ c &= 0.274945609390718 \end{aligned} \tag{19}$$

I risultati della regressione sono mostrati in Figura 7b

A prescindere da quale delle due approssimi meglio i campioni, è stata preferita la seconda regressione poiché la funzione sin è sempre definita nell'intervallo $[-1, 1]$, che è lo stesso utilizzato nella rappresentazione dei campioni audio (sapendo inoltre che il coefficiente a dell'equazione 19 è strettamente minore di 1, si può avere la certezza che la distorsione non provocherà mai il fenomeno del clipping, fatto non garantito usando un polinomio di terzo grado). Non è stata studiata la distorsione del tubo in PVC poiché i tubi, in generale, sono considerati sistemi LTI.

2.3 Analisi della voce

2.3.1 La voce

La voce è il risultato del filtraggio da parte dell'apparato fonatorio umano di un'onda sonora [6] prodotta da una continua sequenza di apertura e chiusura delle corde vocali sollecitate dalla pressione dell'aria proveniente dai polmoni. Essa, a causa del movimento delle corde vocali, assume le stesse caratteristiche di un'onda a dente di sega [6]. Nel caso in cui il suono sia prodotto senza il movimento delle corde vocali, si parla di suono sordo.

La voce può essere modellata in termini di sorgente e di filtro [6]. L'apparato fonatorio, tuttavia, è un sistema tempo variante, poiché cambia continuamente la sua funzione di trasferimento (se così non fosse non si potrebbero pronunciare differenti vocali, si avrebbe a disposizione, ad esempio, un solo fonema). Tale sistema è, inoltre, non lineare. Nella fase di analisi sono stati trascurati i fenomeni non lineari e il segnale vocale è stato finestrato per poter sfruttare l'ipotesi di tempo-invarianza.

Nel dominio delle frequenze (sia ω la frequenza), la relazione tra voce ($P(\omega)$), sorgente sonora ($S(\omega)$) e filtro ($T(\omega)$) può essere espressa nel modo seguente (la fase viene tralasciata poiché non aggiunge alcuna informazione rilevante)[6]:

$$|P(\omega)| = |S(\omega)||T(\omega)| \tag{20}$$

I picchi presenti in $|P(\omega)|$ sono chiamati formanti mentre le frequenze che corrispondono ad un massimo locale in $|T(\omega)|$ sono dette frequenze di risonanza. Le formanti sono indicate con la dicitura $F1$,$F2$,$F3$,$F4\ldots$ in base all'ordine in cui compaiono in $|P(\omega)|$, mentre con la dicitura F_1,F_2,F_3,$F_4\ldots$ si indicano le frequenze corrispondenti alle formanti. Le diverse formanti si distinguono per l'ampiezza e per la loro frequenza. Fonemi diversi corrispondono a distribuzioni diverse delle formanti all'interno dello spettro [6]. Lo studio delle formanti è stato eseguito attraverso il metodo Linear Predictive Coding (LPC).

2.3.2 Il metodo Linear Predictive Coding (LPC)

Sia $x(n)$ un processo stocastico stazionario in senso stretto. Si supponga di stimare il campione $x(n)$ utilizzando una combinazione lineare degli N campioni più recenti. La stima assume la forma (per semplicità si considerano solo segnali reali) [7]:

$$x_N(n) = -\sum_{i=1}^{N} a_{N,i} x(n-i) \tag{21}$$

Dove N è l'ordine di predizione. L'errore sulla stima può essere espresso come [7]:

$$\begin{aligned} e_N(n) &= x(n) - \hat{x}_N(n) \\ &= x(n) + \sum_{i=1}^{N} a_{N,i} x(n-i) \end{aligned} \tag{22}$$

L'errore quadratico medio è definito come valore atteso del quadrato del valore assoluto di $e_N(n)$ [7]:

$$\epsilon_n = E[|e_N(n)|^2] \tag{23}$$

Essendo $x(n)$ un processo stocastico fortemente stazionario, $\epsilon(n)$ non dipende dal tempo. A questo punto, il problema consiste nel trovare un set di coefficienti a_1, a_2,...,a_N in modo tale da rendere minimo l'errore quadratico medio. La tecnica di predizione lineare converte quindi un segnale in N coefficienti con un errore $e_N(n)$.

I coefficienti a_1, a_2,...,a_N sono tali per cui l'errore $e_N(n)$ è ortogonale a $x(n-i)$ [7], ovvero:

$$E[e_N(n)x(n-i)] = 0, \quad 1 \leq i \leq N \tag{24}$$

Si considerino a questo punto due vettori colonna:

$$\hat{x} = \begin{bmatrix} x_{n-1} \\ x_{n-2} \\ \cdots \\ x_{n-N} \end{bmatrix} \quad a = \begin{bmatrix} a_{N,1} \\ a_{N,2} \\ \cdots \\ a_{N,N} \end{bmatrix}$$

L'equazione 22 può essere riscritta come:

$$e_N(n) = x(n) + a^T \hat{x} \tag{25}$$

Mentre l'equazione 24 come:

$$E[e_N(n)\hat{x}^T] \tag{26}$$

Sostituendo l'equazione 25 nell'equazione 26 si ottiene:

$$E[(x(n) + a^T\hat{x})\hat{x}^T] = E[x(n)\hat{x}^T] + a^T E[\hat{x}\hat{x}^T] \tag{27}$$

$R = E[\hat{x}\hat{x}^T]$ è definita come la matrice di correlazione del vettore $\hat{x}$ e assume la forma:

$$R = E[x(n-1-i)x(n-1-m)],\ 0 \leq i, m \leq N-1 \tag{28}$$

Chiamando r il vettore $E[\hat{x}x(n)]$ (è il vettore di cross-correlazione), l'equazione 27 diventa:

$$Ra = -r \tag{29}$$

Esprimendo questa relazione attraverso le matrici:

$$\underbrace{\begin{bmatrix} E[\,x_{n-1}^2] & E[x_{n-1}x_{n-2}] & \dots & E[x_{n-1}x_{n-N}] \\ E[x_{n-2}x_{n-1}] & E[\,x_{n-2}^2] & \dots & E[x_{n-2}x_{n-N}] \\ \vdots & \vdots & \ddots & \vdots \\ E[x_{n-N}x_{n-1}] & E[x_{n-N}x_{n-2}] & \dots & E[\,x_{n-N}^2] \end{bmatrix}}_{R} \underbrace{\begin{bmatrix} a_{N,1} \\ a_{N,2} \\ \vdots \\ a_{N,N} \end{bmatrix}}_{a} = \underbrace{-\begin{bmatrix} E[x_{n-1}x(n)] \\ E[x_{n-2}x(n)] \\ \vdots \\ E[x_{n-N}x(n)] \end{bmatrix}}_{-r} \tag{30}$$

Se la matrice R è non singolare, allora la soluzione dell'equazione 29 è:

$$a = -R^{-1}r \tag{31}$$

Prima di risolvere le equazioni per trovare i coefficienti ottimi, però, è necessario eseguire ancora alcuni passaggi matematici. Si definisce $R(k)$ l'autocorrelazione di $x(n)$:

$$R(k) = E[x(n)x(n-k)] \tag{32}$$

Le precedenti matrici diventano:

$$\begin{bmatrix} R(0) & R(1) & \dots & R(N-1) \\ R(1) & R(0) & \dots & R(N-2) \\ \vdots & \vdots & \ddots & \vdots \\ R(N-1) & R(N-2) & \dots & R(0) \end{bmatrix} \begin{bmatrix} a_{N,1} \\ a_{N,2} \\ \vdots \\ a_{N,N} \end{bmatrix} = -\begin{bmatrix} R(1) \\ R(2) \\ \vdots \\ R(N)] \end{bmatrix} \tag{33}$$

Queste equazioni prendono il nome di equazioni di Yule-Walker o *normal equations*. A questo punto, previa trasformazione di queste equazioni nelle cosiddette *augmented normal equations* (spostando a sinistra il vettore $-r$ e aggiungendo una nuova riga alle matrici che dia maggiori informazioni sull'errore di predizione), è possibile calcolare i coefficienti con la ricorsione di Levinson.

Con i coefficieni è possibile costruire il polinomio caratteristico di un filtro FIR:

$$A_N(z) = 1 + \sum_{i=1}^{N} a_{N,i} z^{-i} \tag{34}$$

Una importante proprietà di questo polinomio è la seguente: ogni radice del polinomio caratteristico è contenuta nella circonferenza unitaria, ovvero $|z_i| < 1$ [7]. Questo garantisce che il filtro IIR (filtro che verrà usato nell'algoritmo per l'implementazione del Talk Box) $\frac{1}{A_N}$ è stabile.

Capitolo 3

Implementazione del Talk Box

3.1 Implementazione in MATLAB

Per simulare il comportamento del Talk Box è stato sviluppato in MATLAB un algoritmo che lavora su segnali non in real time. L'algoritmo richiede in input due segnali audio: un segnale portante (proveniente ad, esempio, da un sintetizzatore) e un segnale modulante contenente la voce del musicista. Anche se nel Paragrafo 1.3 è stato affermato che, nella realtà, il Talk Box non ha bisogno di un segnale modulante, nel dominio digitale esso è necessario per inferire la posizione della bocca tramite analisi delle formanti. Nella Figura 8 è esposto lo schema generale dell'algoritmo. L'algoritmo si compone dei seguenti stadi:

1. Moltiplicazione del segnale portante per un valore di gain;
2. Distorsione del segnale portante come modellato nel Paragrafo; 2.2
3. Convoluzione del segnale portante con la risposta all'impulso del Talk Box calcolata in 2.1.2;
4. Divisione in frame del segnale portante;
5. Divisione in frame del segnale modulante;
6. Calcolo del valore di RMS di ogni frame del segnale portante;
7. Aggiunta di zero-padding all'inizio e alla fine di ogni frame del segnale portante;
8. Analisi tramite LPC di ogni frame del segnale modulante;
9. Filtraggio di ogni frame del segnale portante con un filtro IIR utilizzando i coefficienti estratti con LPC dai corrispondenti frame del segnale modulante;

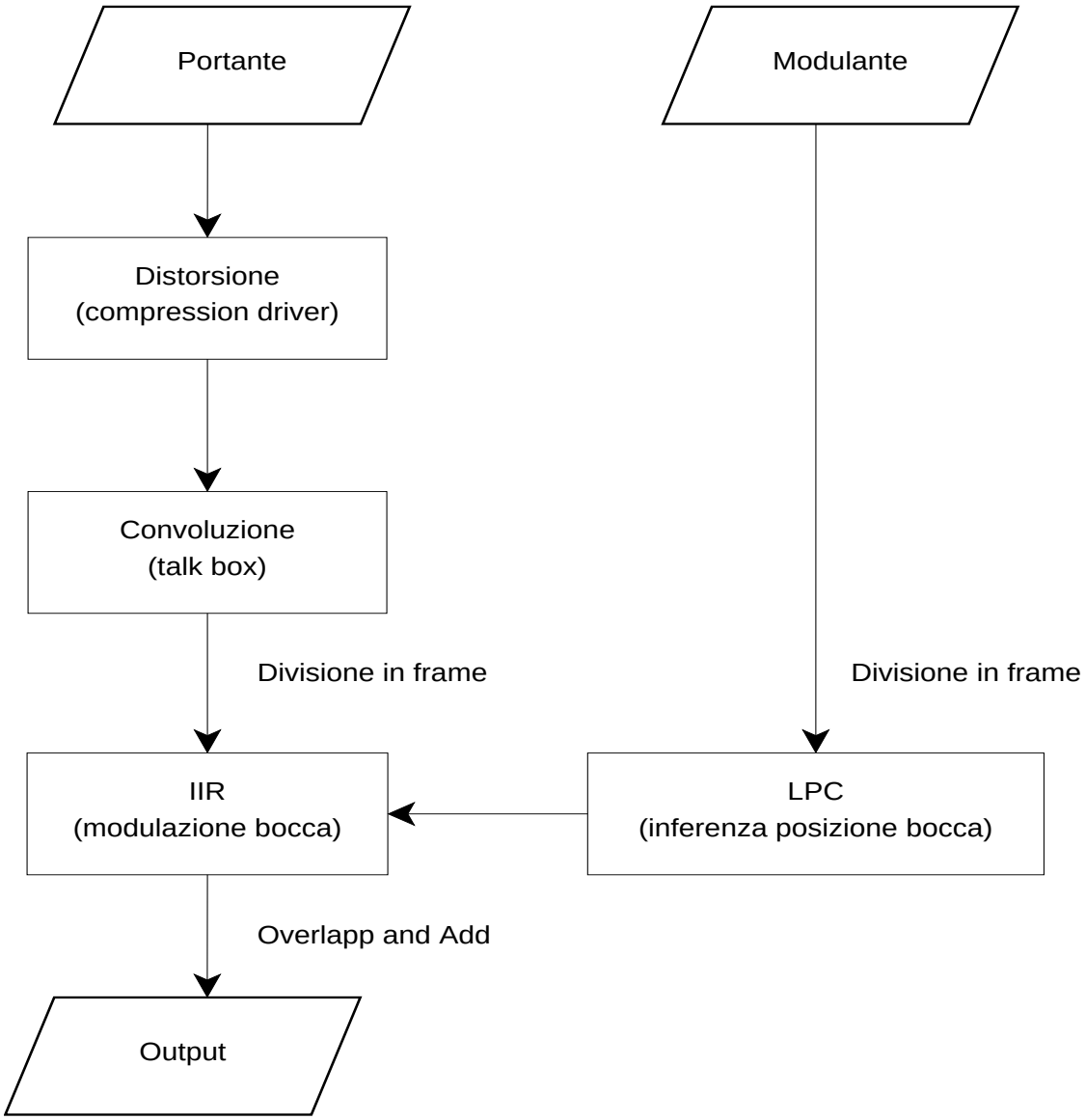


Figura 8: Schema a blocchi dell'algoritmo.

10. Rimozione dei campioni di padding da ogni frame del segnale portante;
11. Moltiplicazione di ogni campione di ogni frame del segnale portante per il rapporto tra il vecchio valore di RMS di ogni frame calcolato al punto 6 e il nuovo valore di RMS dello stesso frame;
12. Finestratura di ogni frame del segnale portante con la finestra di Hann;
13. Ricostruzione del segnale portante a partire dai frame (overlap and add, OLA).

Moltiplicazione del segnale portante per il gain

Questo è il primo stadio dell'algoritmo: serve a simulare la manopola del volume presente sugli strumenti musicali. Cambiando il valore di gain cambia anche l'effetto della distorsione sui campioni audio. Il gain è espresso in dB. Se si vuole applicare ad un campione x un valore di $gain$, è necessario applicare la formula $x = x \cdot 10^{\frac{gain}{20}}$.

Distorsione del segnale portante

In questa fase si applica la distorsione armonica provocata dal compression driver. Ogni campione x del segnale portante viene distorto secondo la formula $x = a \sin(2\pi(x + b) \cdot c)$, dove a, b, c sono i coefficienti ricavati nella regressione non lineare presenti in 19.

Convoluzione con la risposta all'impulso del Talk Box

Il segnale portante viene convoluto con la risposta all'impulso dell'intero Talk Box calcolata nel Paragrafo 2.1.2. Per la convoluzione è stata usata la funzione di MATLAB `conv`.

Divisione in frame del segnale portante

A questo punto il segnale viene diviso in frame di lunghezza 2048 campioni e overlap del 50% (ovvero i primi 1024 campioni di un frame corrispondono agli ultimi 1024 campioni del frame precedente). Il risultato di questa operazione è una matrice di 2048 righe e $\lceil L/(n-p) \rceil$ colonne (ogni colonna della matrice rappresenta un frame), dove L è la lunghezza in campioni della portante, n è la dimensione dei frame e p e la quantità, in campioni, di overlap. L'uso di frame con sovrapposizione evita la formazione di glitch (presenti invece in frame senza overlap). La divisione in frame è stata realizzata mediante la funzione `buffer`.

Divisione in frame del segnale modulante

La medesima operazione svolta sul segnale portante viene applicata al segnale modulante mantenendo inalterati i parametri.

Calcolo del valore di RMS di ogni frame del segnale modulante

Questo passo è necessario per tenere traccia del valore di RMS prima del filtraggio, che tende ad aumentare tale valore in modo significativo.

Zero-padding

I filtri IIR distorcono la fase di un segnale. Ciò significa che introducono un certo ritardo sul segnale in ingresso. Senza la sequenza di padding alla fine dei frame, una parte del segnale filtrato andrebbe persa. Per compensare questo ritardo, il segnale viene filtrato anche in senso opposto, quindi è necessario inserire il padding anche all'inizio dei frame. La dimensione di padding all'inizio e alla fine dei frame è di 2500 campioni, quindi il numero di righe della matrice è ora $2048 + 5000 = 7048$.

Analisi con LPC di ogni frame del segnale modulante

Ogni frame del segnale modulante viene convertito in un set di coefficienti $a_1,\ldots,a_N$, dove N è l'ordine di predizione (l'informazione relativa all'errore di predizione viene scartata). Durante l'implementazione in MATLAB è stato posto $N = 48$, seguendo la regola empirica che prescrive di utilizzare un ordine pari alla frequenza di campionamento espressa in kHz aumentata di 4 (ad esempio, nel caso di una frequenza di campionamento di $44100kHz$, l'ordine è di $44+4 = 48$). Per eseguire l'analisi tramite LPC, MATLAB mette a disposizione la funzione `lpc`.

Filtraggio dei frame della portante

I coefficienti estratti dai frame del segnale modulante vengono utilizzati per filtrare i campioni provenienti dai corrispondenti frame del segnale portante attraverso un filtro IIR di forma $\frac{1}{A_N}$ (A_N rappresenta il set di coefficienti, quindi $A_N = 1+\sum_1^N a_{i,N}$). Per compensare il ritardo provocato dal filtro IIR, il segnale viene filtrato in entrambi in sensi. Tutto ciò viene fatto automaticamente con la sola funzione `filtfilt`.

Rimozione del padding

A questo punto dell'algoritmo, il padding precedentemente aggiunto non è più necessario, quindi viene rimosso. Il numero di righe della matrice torna ad essere 2048.

Moltiplicazione dei campioni per il rapporto tra gli RMS prima e dopo il filtro

All'uscita dal filtro, il valore di RMS dei frame è notevolmente diverso rispetto al valore originale. Per riportarlo al valore iniziale è necessario calcolare il nuovo RMS, calcolare il rapporto tra quest'ultimo e quello originale ed eseguire una moltiplicazione tra i campioni di un frame e il suddetto rapporto. Questa operazione deve essere eseguita su tutti i frame del segnale portante.

Finestratura di ogni frame del segnale portante tramite finestra di Hann

Prima di ricostruire il segnale a partire dai frame, è necessario moltiplicarli per una funzione finestra per evitare la formazione di glitch nel segnale ricostruito. La funzione utilizzata è la finestra di Hann, definita come:

$$\omega(n) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{35}$$

Dove N è la lunghezza in campioni della finestra. MATLAB possiede la funzione `hann` che permette di calcolare la finestra di Hann.

Overlap and add

L'ultima fase dell'algoritmo prevede la ricostruzione a partire dai singoli frame. Si calcola la posizione di ogni campione nel segnale finale (un vettore lungo quanto il segnale originale prima che fosse diviso in frame) e lo si somma ad eventuali campioni già sommati in quella posizione. Da ciò si deduce che alla fine è necessario dividere il campione finale per il numero di volte per cui è stato sommato (ad esempio, nel caso di un overlap del 50%, ad esempio, è necessario dividere tutti i campioni per 2), in realtà in questa occasione non occorre, poiché una delle proprietà della finestra di Hann è la somma di finestre sovrapposte pari a 1.

3.2 Implementazione come plug-in audio (VST)

MATLAB, seppur indispensabile per modellare e sviluppare algoritmi e per analizzare i dati, risulta poco spendibile nella creazione di applicazioni stand-alone al di fuori dell'ambito accademico. Uno dei maggiori problemi dell'algoritmo esposto nel paragrafo precedente, ad esempio, è l'impossibilità di lavorare su segnali in real time.

Per questo motivo, accanto all'implementazione realizzata con MATLAB, che è stata comunque tenuta come riferimento, ne è stata sviluppata un'altra utilizzando strumenti software concepiti proprio per la creazione di applicazioni audio (nello

specifico, plug-in audio VST) utilizzabili nelle cosiddette Digital Audio Workstation (DAW). L'implementazione è stata realizzata attraverso il framework Jules' Utility Class Extensions (JUCE) in C++.

3.2.1 Cosa è un plug-in audio

Un plug-in audio è un programma che aggiunge funzionalità ad una DAW. Il termine programma può essere fuorviante: un plug-in non può essere eseguito in modo autonomo, ha bisogno del supporto della DAW (che viene chiamata "host") per poter essere in esecuzione.

I plug-in audio possono processare i segnali ricevuti dalla DAW (implementano quindi effetti o filtri, eventualmente emulando processori audio hardware) e/o generare segnali audio (fungono quindi da strumenti virtuali). Esistono diversi protocolli per la creazione di plug-in audio, ad esempio il protocollo Virtual Studio Technology (VST) di Steinberg oppure Audio Units (UA) di Apple. Il Talk Box è stato sviluppato attraverso il protocollo VST.

3.2.2 Jules' Utility Class Extensions (JUCE)

JUCE è un framework creato da Jules Storer per lo sviluppo di applicazioni multipiattaforma in C++ [8]. Include un vasto set di classi per risolvere alcuni problemi comuni durante lo sviluppo di software, come ad esempio la creazione di interfacce grafiche, la gestione dell'audio, l'interazione con l'utente...

JUCE è molto usato per la creazione di plug-in VST e applicazioni audio, anche se le sue possibilità non si limitano solo a questo ambito. È possibile sviluppare inoltre: applicazioni con o senza interfaccia grafica (di particolare interesse è l'editor WYSIWYG per gestire i componenti), applicazioni che generano animazioni, librerie statiche e dinamiche. JUCE supporta tutti i maggiori sistemi operativi: OSX, Windows, Linux, iOS e Android.

Il framework include Projucer, che consente di lavorare direttamente su un progetto (può essere considerato un IDE) oppure può essere usato congiuntamente ad un altro IDE (può esportare un progetto per un IDE a scelta tra quelli supportati, tra cui Visual Studio e XCode). In questo modo è possibile, ad esempio, sviluppare il codice con Visual Studio e lavorare sull'interfaccia grafica sfruttando l'editor di Projucer.

3.2.3 Implementazione in JUCE

JUCE vs. MATLAB

Durante lo sviluppo del plug-in VST, è stato seguito l'algoritmo presentato nel Paragrafo 3.1, mettendo a punto alcuni accorgimenti dovuti al fatto che il processing

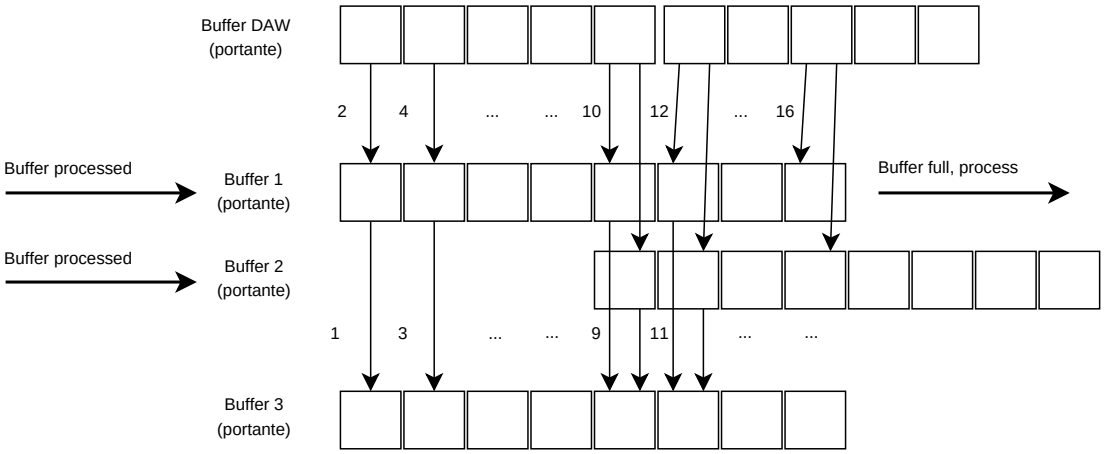


Figura 9: Illustrazione della buffer policy.

dei dati audio avviene in real-time e al modo in cui una DAW "passa" i campioni audio al plug-in. In merito a questo, il protocollo VST concede al programma host la libertà di scelta della dimensione del buffer dei campioni audio da passare al plug-in (il programma REAPER, ad esempio, ha un buffer lungo 1024 campioni). A priori, dunque, un plug-in audio non può sapere quale è il numero di campioni da processare. Dato che l'algoritmo proposto precedentemente necessita di un frame di 2048 campioni con overlap di 1024, è stato implementato un sistema di buffer (nello specifico, una classe buffer) per soddisfare questi requisiti. Prima di descrivere questo sistema, è necessario precisare che il plug-in acquisisce il segnale portante dal canale destro di una traccia audio, mentre il segnale modulante dal canale sinistro. L'audio processato viene poi copiato sia sul canale destro sia su quello sinistro (l'output è dual mono). Sono stati predisposti 3 buffer per il segnale portante e 3 buffer per quello modulante (tutti i buffer sono della stessa classe).

La Figura 9 illustra il funzionamento dei buffer. La prima operazione consiste nel salvataggio di un campione dal buffer appena processato verso un buffer di supporto. Successivamente viene prelevato un campione dal buffer proveniente dalla DAW e viene salvato nella stessa posizione del campione precedentemente spostato, e così via. Quando i due buffer cominciano a sovrapporsi, le operazioni si ripetono in modo identico coinvolgendo il secondo buffer: i due campioni appena processati vengono prelevati dai due buffer, sommati (per la proprietà della finestra di Hann) e il risultato salvato nel buffer di supporto. Il campione prelevato dal buffer del programma host viene quindi copiato in entrambi i buffer per permettere l'overlap. Quando uno dei due buffer è pieno, viene processato e il ciclo ricomincia. I campioni contenuti nel terzo buffer vengono poi copiati nel buffer della DAW. Le medesime operazioni sono svolte nella gestione del segnale modulante.

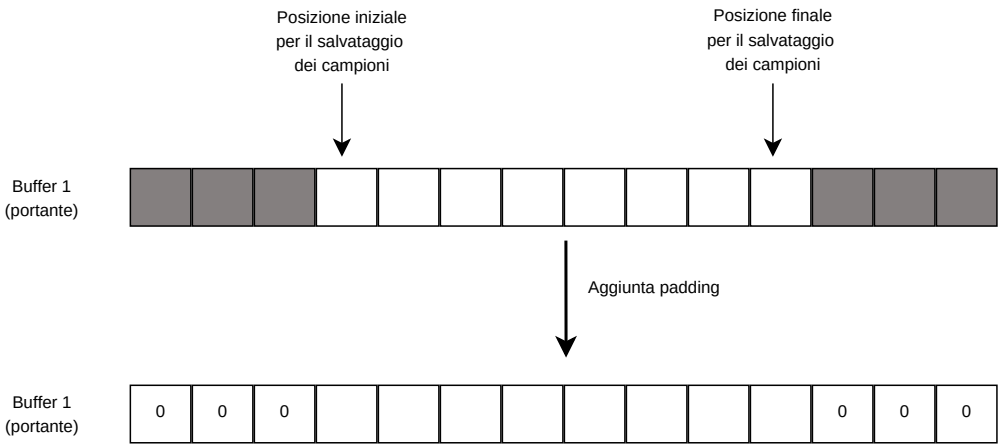


Figura 10: Sia ad esempio 3 la dimensione di padding. Quando uno dei due buffer, in questo caso il primo, è pieno, si procede ad assegnare alle prime e alle ultime 3 celle il valore 0.

Si ponga l'attenzione al fatto che, diversamente dall'implementazione in MATLAB, la fase di OLA (overlap and add) viene eseguita prima del processing a livello di codice, ma dopo dal punto di vista logico, perché non appena il plug-in va in esecuzione si assume che i buffer che gestiscono il segnale portante siano già stati processati (ovviamente all'inizio i buffer contengono solo silenzio). I primi 2048 campioni che il plug-in restituisce alla DAW, quindi, hanno tutti valore 0 (si può affermare che il plug-in introduce un ritardo di 2048 campioni, compensato grazie all'uso della funzione di JUCE `setLatencySamples()`, che informa il programma host dell'entità del ritardo).

Prima delle operazioni per la gestione dei buffer, i campioni vengono moltiplicati per un fattore di gain e distorti secondo i parametri descritti nell'equazione 19.

Quando uno dei due buffer è pieno, viene processato. Per prima cosa al buffer che contiene il segnale portante viene "aggiunto" il padding di zeri. Diversamente dall'implementazione in MATLAB, tuttavia, ciò non viene fatto dinamicamente durante il processing. Lo spazio per il padding è già compreso nella dimensione del buffer. Quando un buffer viene allocato per la prima volta, la sua dimensione è pari a $2048 + 2 \cdot 1000$ (per questioni di performance la dimensione di padding è stata ridotta a 1000 campioni) e non cambia mai durante l'esecuzione del plug-in. Questo perché, dato che un buffer è di tipo `vector`, le operazioni di ridimensionamento graverebbero pesantemente sulla performance. In questo contesto, dunque, l'aggiunta del padding si concretizza nell'assegnare alle prime e alle ultime 1000 celle del buffer il valore 0. In Figura 10 viene esemplificato questo procedimento.

Il passo successivo consiste nella convoluzione tra la risposta all'impulso del Talk

Box e il buffer che contiene il segnale portante. La teoria afferma che, dati due segnali discreti $x(n)$ e $y(n)$, la loro convoluzione è un segnale $z(n)$ la cui lunghezza è pari a $lunghezza(x(n)) + lunghezza(y(n)) - 1$. Considerate le lunghezze della risposta all'impulso (2192 campioni, ridotta rispetto alla versione usata in MATLAB per questioni di performance) e del buffer (2048, poiché si considerano solo i campioni "validi" prelevati dal buffer della DAW), la dimensione del segnale risultante è $2192 + 2048 - 1 = 4239$. Ciò pone una complicazione: 4239 è maggiore rispetto alla dimensione del buffer. Non è possibile scartare i campioni che non trovano spazio, sia per la formazione di glitch, sia perché sarebbe tutta informazione deliberatamente persa. Come soluzione è stata implementata una sorta di convoluzione "continua", che somma i campioni "precedenti" e in eccesso ai campioni risultanti dalla convoluzione "attuale". La Figura 11 dà un esempio di come funziona.

Analogamente all'implementazione in MATLAB, lo stadio successivo è rappresentato dall'analisi del segnale modulante mediante LPC. Una volta che il segnale è stato convertito in una serie di coefficienti, si filtra il buffer portante corrispondente con un filtro IIR che ha la caratteristica di filtrare i campioni in entrambe le direzioni per annullare la distorsione di fase introdotta dal filtro stesso (distorsione che creerebbe fenomeni di interferenza durante la fase di OLA). Questa operazione è spiegata in Figura 12. Dato che dopo questa fase il valore di RMS del buffer risulta alterato, per riportarlo al valore iniziale (calcolato non appena il buffer si riempie) occorre moltiplicare ogni campione "valido" per il rapporto tra gli RMS prima e dopo il filtraggio. Si noti che le operazioni di calcolo del valore di RMS non tengono conto dei campioni riservati per il padding (se così fosse, il valore di RMS risulterebbe sottostimato). L'ultima operazione consiste nel moltiplicare i campioni per la finestra di Hann descritta dall'equazione 35. Si veda la Figura 13 per maggiori dettagli.

Il codice

Si è già accennato al fatto che il plug-in è stato sviluppato in C++. Sono state scritte diverse classi per la gestione dei passi dell'algoritmo proposto:

1. `GainClass`: applicazione di un fattore di gain ai campioni audio (si vedano i codici sorgente A.1 e A.2);

2. `DistortionClass`: applicazione della distorsione ai campioni audio (codici sorgente A.3 e A.4);

3. `BufferClass`: gestione del sistema di buffer. Include metodi per il calcolo del valore di RMS, per l'aggiunta di padding, per la gestione circolare del buffer, per il prelievo di campioni dal buffer proveniente dalla DAW (codici sorgente A.5 e A.6);

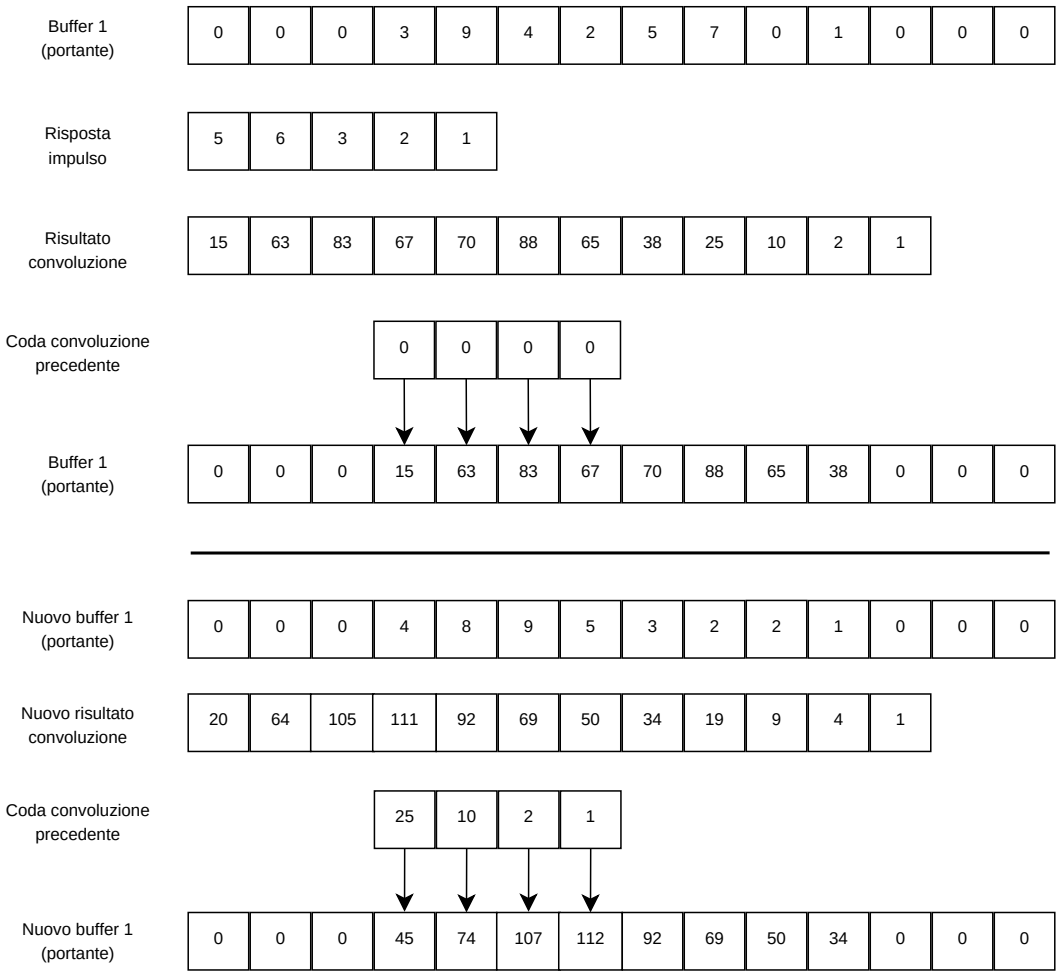


Figura 11: Esempio della convoluzione continua. In questo esempio si presuppone che la convoluzione sia calcolata per la prima volta, quindi la prima coda contiene solo valori pari a 0.

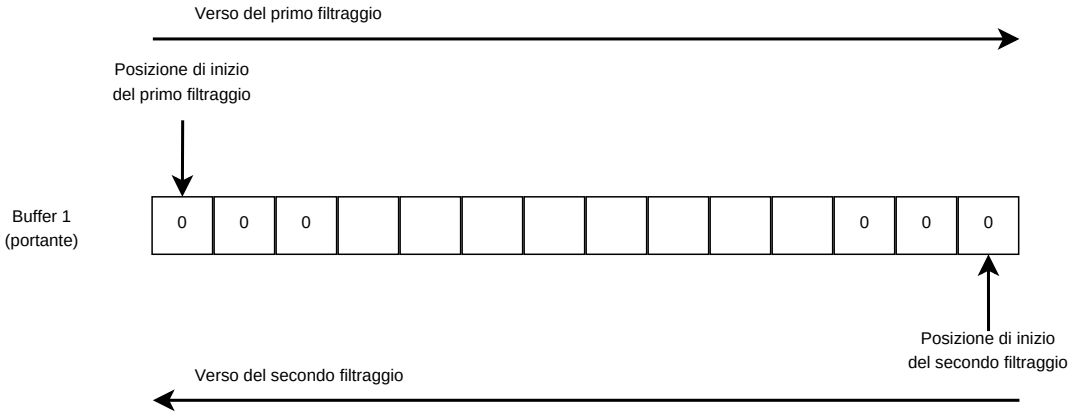


Figura 12: La figura illustra il funzionamento del filtro IIR implementato.

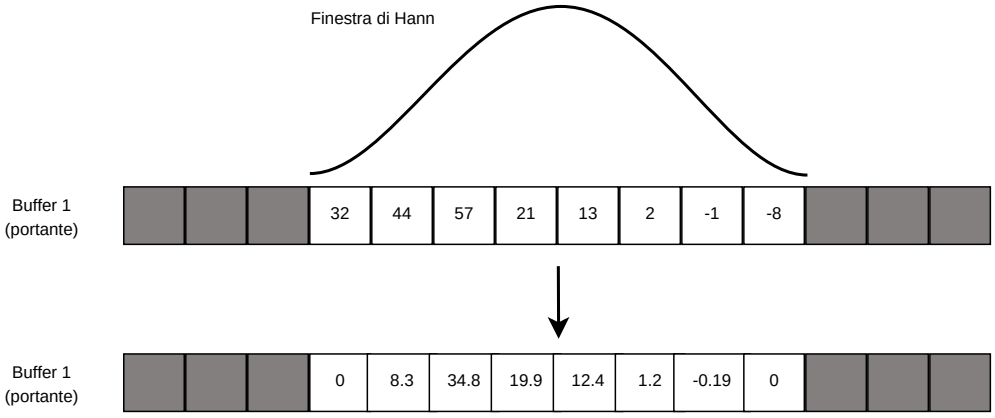


Figura 13: Come nei casi già esaminati, la funzione che applica la finestra di Hann non tiene conto dei campioni usati per il padding.

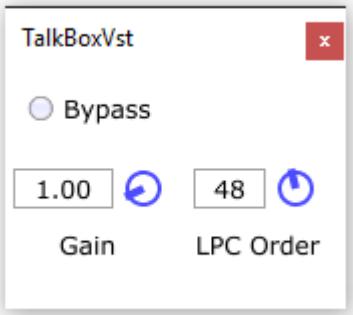


Figura 14: Interfaccia generata con l'editor di Projucer del plug-in non appena viene aperto.

4. `ConvolutionClass`: implementazione della convoluzione "continua" (codici sorgente A.7 e A.8);
5. `LPCClass`: conversione del segnale modulante in un set di coefficienti. Il metodo che implementa l'algoritmo di LPC è stato reperito in rete [9] (prima dell'utilizzo è stato testato e i risultati confrontati con quelli forniti dalla funzione `lpc` di MATLAB) (codici sorgente A.9 e A.10);
6. `IIRFilterClass`: implementazione del filtro IIR in entrambe le direzioni per annullare la distorsione di fase (codici sorgente A.11 e A.12);
7. `HannWindowClass`: calcolo della finestra di Hann e applicazione della finestra ai campioni (codici sorgente A.13 e A.14).

Il codice sorgente A.15 rappresenta il metodo `processBlock` in cui viene eseguito il processing dei campioni proveniente dalla DAW.

Il plugin

Una volta aperto in una qualsiasi DAW, il plugin assume l'aspetto rappresentato in Figura 14. L'utente può interagire con il plug-in disattivandolo (pulsante "Bypass"), cambiando il valore di gain da applicare ai campioni (il range è 0-10) oppure modificando l'ordine di predizione dell'algoritmo LPC (il range in questo caso è 5-100). I primi due controlli hanno una loro controparte nel mondo fisico: infatti, tutti i Talk Box in commercio possiedono un pulsante che permette di renderli passivi; mentre la manopola che regola il gain è presente su tutti gli strumenti musicali usati in coppia con un Talk Box.

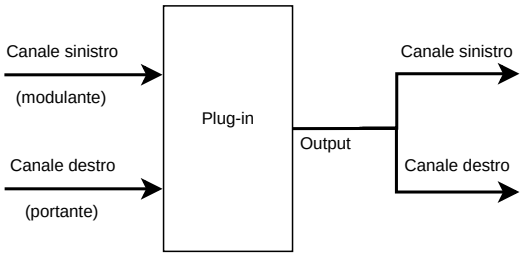


Figura 15: Routing dei segnali audio da/verso il plug-in.

La manopola che regola l'ordine di predizione è stata aggiunta per dare un tocco "artistico" al plug-in. Ovviamente, nel mondo fisico, la funzione di trasferimento della bocca, a parità di posizione, è sempre la stessa. La differenza tra i vari ordini di predizione comincia a essere percepibile man mano che l'ordine raggiunge il limite inferiore e il suono in output diventa sempre più simile ad una voce nasale.

I valori di default del plug-in sono: Bypass a 0 (il plug-in si attiva non appena viene lanciato), Gain a 1 (cambiando il valore di gain muta anche l'effetto distorsivo) e LPC Order a 48 (si segue la regola empirica enunciata nel Paragrafo 3.1).

Come già accennato precedentemente in questo paragrafo, il plug-in riceve i dati audio da un'unica traccia stereo. Esso è istruito sul modo in cui considerare i segnali audio presenti nei due canali. Il canale sinistro è preposto a contenere il segnale modulante e viene quindi sottoposto all'analisi tramite LPC. Il canale destro contiene invece il segnale portante e viene filtrato. Anche se durante lo sviluppo del plug-in all'interno del codice è stata dichiarata un'uscita stereo (si verificano malfunzionamenti se impostata su mono quando l'input è stereo), nella realtà il segnale in uscita è copiato tale e quale sia sul canale destro che su quello sinistro (l'uscita è quindi dual mono).

Capitolo 4

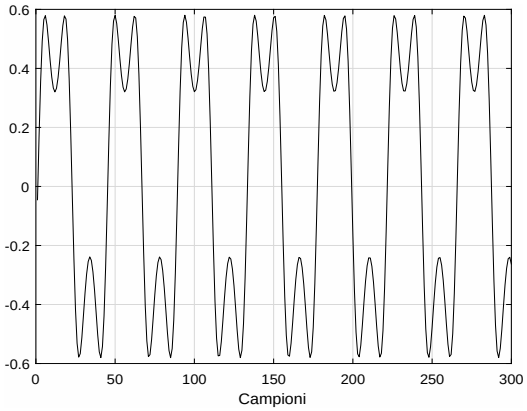
Validazione dei risultati

4.1 Test della distorsione

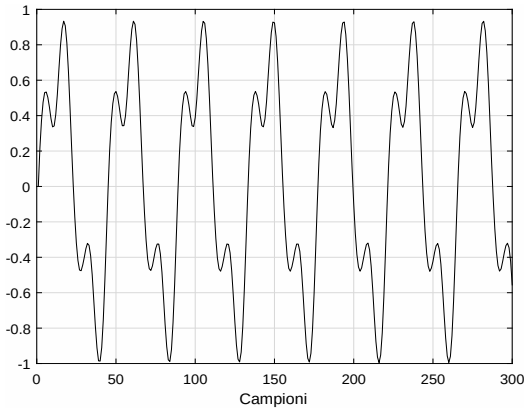
In questa fase sono state messe a confronto la distorsione prodotta dal Talk Box e la distorsione prodotta dalla funzione modellata attraverso la regressione non lineare effettuata nel Paragrafo 2.2. È stato somministrato al Talk Box un segnale sinusoidale con una frequenza di $1kHz$ e il suo output acquisito tramite microfono. Per quanto riguarda la distorsione introdotta dall'algoritmo, è stato preso nuovamente il segnale da $1kHz$ e ogni suo campione è stato distorto secondo la formula (i coefficienti, in questo caso arrotondati alla quarta cifra decimale, sono stati calcolati in 2.2):

$$y = 0.9801 \sin(2\pi(x - 0.0058) \cdot 0.2749) \tag{36}$$

I risultati sono mostrati in Figura 16 e in Figura 17. A prima vista, considerando il dominio temporale, i due segnali sembrerebbero essere differenti. Ciò è dovuto ad un problema di fase. Fissando l'attenzione invece sul dominio frequenziale, salta subito all'occhio che i profili dei due spettri sono molto simili. Guardando la Figura 17a si possono facilmente notare dei picchi in corrispondenza dei multipli interi della frequenza fondamentale di $1kHz$, in particolare a $2kHz$, $3kHz$, $4kHz$, $5kHz$, $6kHz$, $7kHz$. I primi sei picchi mantengono le stesse relazioni di ampiezza presenti nel segnale registrato (l'energia in corrispondenza dell'armonica fondamentale è maggiore rispetto all'energia presente a $2kHz$, che a sua volta è minore dell'energia a $3kHz$ e così via...). Fa eccezione la relazione tra le ampiezze tra la sesta e la settima armonica. Rispetto al segnale registrato, il segnale distorto dall'algoritmo non possiede picchi evidenti oltre i $7kHz$. In realtà, questo fatto dipende dal gain. Aumentando il gain da applicare al segnale di $1kHz$ trattato dall'algoritmo cominciano ad essere visibili dei picchi anche nella zona delle frequenze superiori a $7kHz$ (anche se, però, in questo caso il profilo frequenziale comincia a differire da quello del segnale registrato).

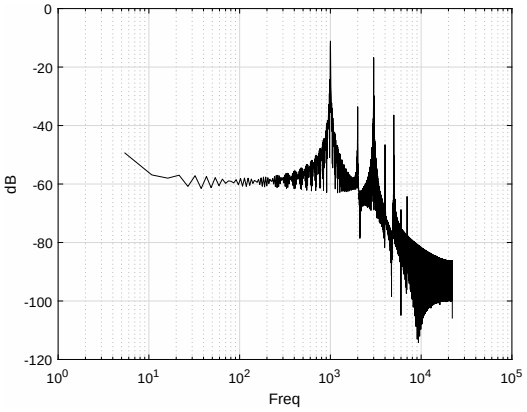


(a) Particolare dei primi 300 campioni estratti dal segnale distorto dall'algoritmo.

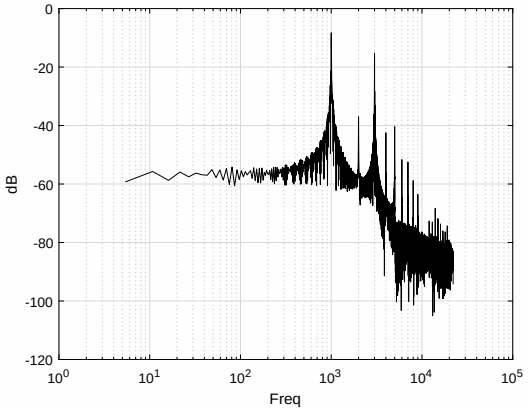


(b) Particolare dei primi 300 campioni estratti dal segnale distorto dal Talk Box.

Figura 16: Confronto tra le due distorsioni nel dominio del tempo. A sinistra il segnale distorto dall'algoritmo, a destra il segnale registrato dal Talk Box.



(a) FFT del segnale da $1Khz$ distorto secondo l'equazione 36.



(b) FFT del segnale in uscita dal Talk Box avente in input un segnale da $1kHz$.

Figura 17: Confronto tra le due distorsioni nel dominio della frequenza. A sinistra il segnale distorto dall'algoritmo, a destra il segnale registrato dal Talk Box.

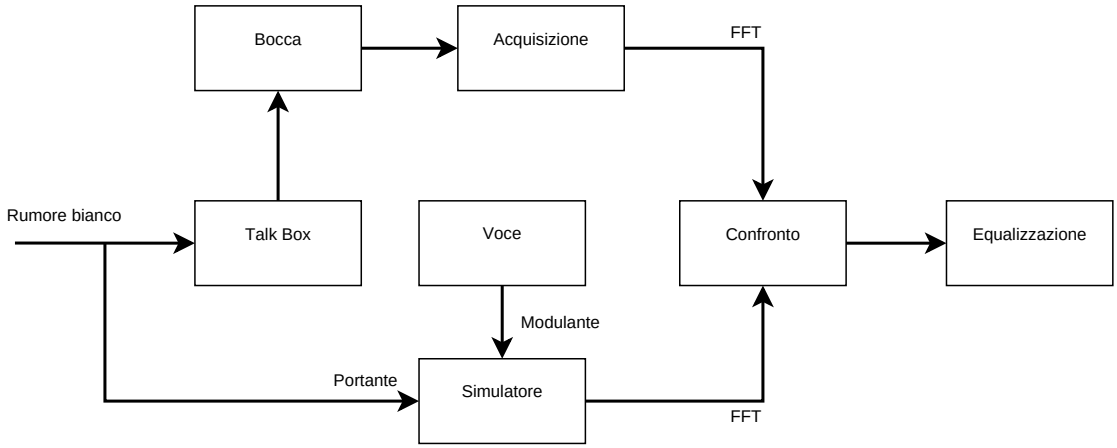


Figura 18: Schema a blocchi per il test del metodo LPC.

4.2 Test del metodo LPC

È bene precisare fin da subito che non è stato possibile effettuare la validazione del metodo LPC come estimatore della posizione della bocca a causa di problemi tecnici dovuti ad un guasto del Talk Box. Si rimanda la presentazione dei risultati in sede di discussione. Di seguito vengono invece proposte le operazioni da eseguire per il test. Esse sono schematizzate in Figura 18. Si seguono due percorsi diversi, ma paralleli. Entrambi i percorsi partono dalla generazione di rumore bianco. La prima diramazione prevede che il segnale appena sintetizzato venga dato in input al Talk Box che, a sua volta, provvede ad indirizzarlo verso la bocca. Il suono filtrato dalla bocca viene acquisito tramite microfono e convertito nel dominio frequenziale. Il secondo ramo prevede invece di dare in input il segnale precedentemente sintetizzato al simulatore sviluppato. Il segnale modulante proviene dalla registrazione della voce che esegue esattamente la modulazione performata quando è stato acquisito, nella prima fase di questo test, il segnale in uscita dalla bocca. Questo perché la voce è il risultato del filtraggio da parte dell'apparato fonatorio (Paragrafo 2.3.1) secondo le diverse forme e posizioni da esso assunte durante l'emissione dei suoni. Cambiando la posizione, cambia la distribuzione frequenziale dei suoni emessi (in particolare, come si è già detto, cambia la distribuzione e l'ampiezza delle formanti) e a parità di distribuzione corrisponde una posizione ben definita dell'apparato. Si instaura quindi una relazione biunivoca. Con questo test si cerca di capire se il metodo LPC è capace di mettere in luce questa relazione, perché il segnale acquisito nel ramo precedente contiene informazione solo in merito alla posizione dell'apparato (in particolare, della bocca), non essendoci produzione di suono da parte delle corde vocali; mentre in questo caso si cerca di desumere la posizione a partire dalle formanti presenti nel suono prodotto durante la fonazione.

Successivamente, anche in questo caso viene operata la trasformata di Fourier sul segnale in uscita dal simulatore. A questo punto i due rami si incontrano: viene eseguito il confronto tra i due spettri. In caso di errore sistematico, si provvede ad estrarre dal confronto un fattore di correzione che viene usato per equalizzare il segnale in uscita dal simulatore.

Capitolo 5

Conclusioni

5.1 Conclusioni e possibili miglioramenti

Non potendo avere il quadro completo dei risultati a causa del mancato test per il metodo LPC, non è possibile giungere a delle conclusioni esaustive (anche in questo si rimanda alla discussione). Tuttavia, dal punto di vista qualitativo, il plug-in genera in uscita un segnale intelligibile mantenendo l'intonazione del segnale portante. L'audio risulta però scuro e il timbro dello strumento usato non è facilmente riconoscibile. In definitiva si può affermare (almeno per ora) che la simulazione ha avuto esito positivo per quanto riguarda gli aspetti tecnici, ma risulta migliorabile per quanto riguarda la qualità del suono in uscita. Si spera che grazie ai risultati dei test quantitativi si possa risolvere il problema relativo all'assenza di alte frequenze.

Appendice A

Codice sorgente delle classi implementate

In questa sezione viene presentato il codice sviluppato durante l'implementazione con JUCE del Talk Box. Se non esplicitamente scritti, i metodi costruttori e distruttori delle classi non contengono istruzioni.

Listing A.1: Definizione della classe GainClass.

```
class GainClass
{
  public:
  GainClass();
  ~GainClass();
  float getGainValue();
  void setGainValue(float new_value);
  void ClockProcess(float *Sample);

  private:
  float GainValue;
};
```

Listing A.2: Metodi della classe GainClass.

```
float GainClass::getGainValue()
{
  return GainValue;
}

void GainClass::setGainValue(float new_value)
{
```

```
  GainValue = new_value;
}

void GainClass::ClockProcess(float *Sample)
{
  *Sample = (*Sample) * GainValue;
}
```

Listing A.3: Definizione della classe DistortionClass.

```
class DistortionClass {
  public:
  DistortionClass();
  ~DistortionClass();
  void ClockProcess(float *sample);

  private:
  float parametro1 = 0.980777015580225;
  float parametro2 = 0.274943806353101;
  float parametro3 = -0.005834735575077;
};
```

Listing A.4: Metodi della classe DistortionClass.

```
void DistortionClass::ClockProcess(float *sample)
{
  *sample = parametro1 * sin(2 * 3.1415926535 * (*sample +
      parametro3) * parametro2);
}
```

Listing A.5: Definizione della classe BufferClass.

```
class BufferClass
{
  public:
  BufferClass();
  ~BufferClass();
  void ClockProcess(float *audio_sample);
  void ClockProcess(float sample1, float sample2);
  float getSampleBuffered(int index);
  bool isFull();
  float computeRMS();
  void addPadding();
```

```
  void applyRMS();
  void setBufferLength(int new_value);
  int getBufferLength();
  int getInputIndex();
  std::vector<float>& getAudioBuffer();
  void setAudioBuffer(std::vector<float>& new_buffer);
  int getPadding();
  int getOverlap();

  private:
  int overlap = 1024;
  int padding = 1000;
  float old_rms, new_rms, ratio, sum;
  bool buffer_full = false;
  int buffer_input_index = padding;
  size_t buffer_length = 2048;
  std::vector<float> audio_buffer;
};
```

Listing A.6: Metodi della classe BufferClass.

```
BufferClass::BufferClass()
{
  audio_buffer.resize((size_t)(buffer_length + 2 * padding
     ), 0);
}

void BufferClass::ClockProcess(float *audio_sample)
{
  audio_buffer[buffer_input_index] = *audio_sample;
buffer_input_index++;
}

void BufferClass::ClockProcess(float sample1, float
   sample2)
{
  audio_buffer[buffer_input_index] = (sample1 + sample2) /
      2;
buffer_input_index++;
}
```

```
float BufferClass::getSampleBuffered(int index)
{
  return audio_buffer[index];
}

int BufferClass::getInputIndex()
{
  return buffer_input_index;
}

bool BufferClass::isFull()
{
  buffer_full = false;
  if (buffer_input_index == (padding + buffer_length))
  {
    buffer_input_index = padding;
    old_rms = computeRMS();
    return buffer_full = true;
  }
  return buffer_full;
}

float BufferClass::computeRMS()
{
  sum = 0;
  for (int i = padding; i < (padding + buffer_length); ++i
     )
  {
    sum = sum + audio_buffer[i] * audio_buffer[i];
  }
  return sqrt(sum / buffer_length);
}

void BufferClass::addPadding()
{
  for (int i = 0; i < (buffer_length + 2 * padding); ++i)
  {
    if (i < padding || i >= padding + buffer_length)
    {
      audio_buffer[i] = 0;
```

```
    }
  }
}

void BufferClass::applyRMS()
{
  new_rms = computeRMS();
  ratio = new_rms / old_rms;
  for (int i = padding; i < (padding + buffer_length); ++i
     )
  {
    audio_buffer[i] = audio_buffer[i] / ratio;
  }
}

std::vector<float>& BufferClass::getAudioBuffer()
{
  return audio_buffer;
}

void BufferClass::setBufferLength(int new_value)
{
  buffer_length = new_value;
}

int BufferClass::getBufferLength()
{
  return buffer_length;
}

int BufferClass::getPadding()
{
  return padding;
}

int BufferClass::getOverlap()
{
  return overlap;
}
```

```
void BufferClass::setAudioBuffer(std::vector<float>&
   new_buffer)
{
  audio_buffer = new_buffer;
}
```

Listing A.7: Definizione della classe ConvolutionClass.

```
class ConvolutionClass {
  public:
  ConvolutionClass();
  ~ConvolutionClass();
  std::vector<float> &process(std::vector<float> &x, int
     padding,int overlap, float *h, int size_h, float *y);

  private:
  float *tail = NULL;
  int n, index = 0;
};
```

Listing A.8: Metodi della classe ConvolutionClass.

```
ConvolutionClass::~ConvolutionClass()
{
  if (!tail)
  free(tail);
}

std::vector<float>  &ConvolutionClass::process(std::vector
   <float> &x, int padding, int overlap,float *h, int
   size_h, float *y)
{
  int size_x = (x.size() - 2 * padding)/2;

  if (!tail)
    tail = (float *)calloc(size_h - 1, sizeof(float));

  int total_size = size_x + size_h - 1;

  for (n = 0; n < total_size; ++n)
  {
    int kmin, kmax, k;
```

```
    y[n] = 0;

    kmin = (n >= size_h - 1) ? n - (size_h - 1) : 0;
    kmax = (n < size_x - 1) ? n : size_x - 1;

    for (k = kmin; k <= kmax; ++k)
    {
      y[n] += x[k + padding] * h[n - k];
    }
  }

  for (n = 0; n < size_x; ++n)
  {
    x[n + padding] = y[n] + tail[index];
    tail[index++] = 0;
    if (index > size_h - 2)
      index = 0;
  }

  for (n = size_x; n < total_size; ++n)
  {
    tail[index++] += y[n];
    if (index > size_h - 2)
      index = 0;
  }

  for (n = 0; n < total_size; ++n)
  {
    int kmin, kmax, k;

    y[n] = 0;

    kmin = (n >= size_h - 1) ? n - (size_h - 1) : 0;
    kmax = (n < size_x - 1) ? n : size_x - 1;

    for (k = kmin; k <= kmax; ++k)
    {
      y[n] += x[k + padding + overlap] * h[n - k];
    }
```

```
  }

  for (n = 0; n < size_x; ++n)
  {
    x[n + padding + overlap] = y[n] + tail[index];
    tail[index++] = 0;
    if (index > size_h - 2)
      index = 0;
  }

  for (n = size_x; n < total_size; ++n)
  {
    tail[index++] += y[n];
    if (index > size_h - 2)
      index = 0;
  }
  return x;
}
```

Listing A.9: Definizione della classe LPCClass.

```
class LPCClass {
  public:
  LPCClass();
  ~LPCClass();
  void ForwardLinearPrediction(const std::vector<float>& x
     );
  void setNumberCoeffs(int new_value);
  int getNumberCoeffs();
  std::vector<double> &getCoeffs();
  void setCoeffs(std::vector<double> &new_coeffs);
  private:

  std::vector<double> coeffs;
  int NumberCoeffs;
};
```

Listing A.10: Metodi della classe LPCClass.

```
void LPCClass::ForwardLinearPrediction(const std::vector<
   float>& x)
{
```

```
  size_t N = x.size() - 1;
  size_t m = coeffs.size() - 1;

  std::vector<double> R(m + 1, 0.0);
  for (size_t i = 0; i <= m; i++)
  {
    for (size_t j = 0; j <= N - i; j++)
    {
      R[i] += x[j] * x[j + i];
    }
  }

  std::vector<double> Ak(m + 1, 0.0);
  Ak[0] = 1.0;

  double Ek = R[0];

  for (size_t k = 0; k < m; k++)
  {
    double lambda = 0.0;
    for (size_t j = 0; j <= k; j++)
    {
      lambda -= Ak[j] * R[k + 1 - j];
    }
    lambda /= Ek;

    for (size_t n = 0; n <= (k + 1) / 2; n++)
    {
      double temp = Ak[k + 1 - n] + lambda * Ak[n];
      Ak[n] = Ak[n] + lambda * Ak[k + 1 - n];
      Ak[k + 1 - n] = temp;
    }

    Ek *= 1.0 - lambda * lambda;
  }

  coeffs.assign(Ak.begin(), Ak.end());
}
```

```
std::vector<double> &LPCClass::getCoeffs()
{
  return coeffs;
}

void LPCClass::setCoeffs(std::vector<double> &new_coeffs)
{
  coeffs = new_coeffs;
}

int LPCClass::getNumberCoeffs()
{
  return NumberCoeffs;
}

void LPCClass::setNumberCoeffs(int new_value)
{
  if (new_value != NumberCoeffs)
  {
    NumberCoeffs = new_value;
    coeffs.resize((size_t)(new_value + 1));
  }
}
```

Listing A.11: Definizione della classe IIRClass.

```
class IIRFilterClass
{
  public:
  IIRFilterClass();
  ~IIRFilterClass();
  std::vector<float> &process(std::vector<double> &
     NumCoeffs, std::vector<double> &DenCoeffs, std::
     vector<float> &Signal);

  private:
  std::vector<float> FilteredSignal,
     FilteredSignalReversed;
};
```

Listing A.12: Metodi della classe IIRClass.

```
IIRFilterClass::IIRFilterClass()
{
  BufferClass tmp;
  FilteredSignal.resize((size_t)(tmp.getBufferLength() + 2
      * tmp.getPadding()));
  FilteredSignalReversed.resize((size_t)(tmp.
     getBufferLength() + 2 * tmp.getPadding()));
}

std::vector<float> &IIRFilterClass::process(std::vector<
   double> &NumCoeffs, std::vector<double> &DenCoeffs, std
   ::vector<float> &Signal)
{
  int signal_size = Signal.size();
  int numcoeffs_size = NumCoeffs.size();
  int dencoeffs_size = DenCoeffs.size();

  for (int i = 0; i < signal_size; ++i)
  {
    FilteredSignal[i] = 0;

    for (int j = 0; j < numcoeffs_size && (i - j) >= 0; ++
       j)
    {
      FilteredSignal[i] += NumCoeffs[j] * Signal[i - j];
    }
    for (int j = 1; j < dencoeffs_size && (i - j) >= 0; ++
       j)
    {
      FilteredSignal[i] -= DenCoeffs[j] * FilteredSignal[i
          - j];
    }
  }

  for (int i = signal_size - 1; i >= 0; --i)
  {
    FilteredSignalReversed[i] = 0;

    for (int j = 0; j < numcoeffs_size && (i + j) <
       signal_size; ++j)
```

```
    {
      FilteredSignalReversed[i] += NumCoeffs[j] *
         FilteredSignal[i + j];
    }
    for (int j = 1; j < dencoeffs_size && (i + j) <
       signal_size; ++j)
    {
      FilteredSignalReversed[i] -= DenCoeffs[j] *
         FilteredSignalReversed[i + j];
    }
  }
  return FilteredSignalReversed;
}
```

Listing A.13: Definizione della classe HannWindowClass.

```
class HannWindowClass {
  public:
  HannWindowClass();
  ~HannWindowClass();
  std::vector<float>& process(std::vector<float>& buffer);
  void computeHann(int size);

  private:
  int begin;
  std::vector<float> hann_window;
};
```

Listing A.14: Metodi della classe HannWindowClass.

```
HannWindowClass::HannWindowClass()
{
  BufferClass tmp;
  begin = tmp.getPadding();
}

void HannWindowClass::computeHann(int size)
{
  hann_window.resize(size);
  for (int i = 0; i < size; ++i)
  {
```

```
    hann_window[i] = 0.5*(1 - cos((2 * 3.1415926535 * i) /
        (size - 1)));
  }
}

std::vector<float>& HannWindowClass::process(std::vector<
   float>& buffer)
{
  int window_size = hann_window.size();

  for (int i = 0; i < window_size; ++i)
  {
    buffer[i + begin] = buffer[i + begin] * hann_window[i
       ];
  }

  return buffer;
}
```

Listing A.15: Metodo processBlock.

```
void TalkBoxVstAudioProcessor::processBlock (
   AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
{
  int numSamples = buffer.getNumSamples();

  if (!y)
  {
    y = (float *)calloc(buffer_portante1.getBufferLength()
       /2 + ir_length -1, sizeof(float));
  }

  if (!UserParams[MasterBypass])
  {
    float *leftData = buffer.getWritePointer(0); //
       Modulante
    float *rigthData = buffer.getWritePointer(1); //
       Portante

    for (long i = 0; i < numSamples; ++i)
    {
```

```
GainCtrl.ClockProcess(&rigthData[i]);
distortion.ClockProcess(&rigthData[i]);

if (iteratore < buffer_portante1.getOverlap())
{
  buffer_portante_processed.ClockProcess(
     buffer_portante1.getSampleBuffered(
     buffer_portante1.getInputIndex()), 0);
  buffer_modulante_processed.ClockProcess(
     buffer_modulante1.getSampleBuffered(
     buffer_modulante1.getInputIndex()), 0);
  buffer_portante1.ClockProcess(&rigthData[i]);
  buffer_modulante1.ClockProcess(&leftData[i]);
  rigthData[i] = buffer_portante_processed.
     getSampleBuffered(buffer_portante_processed.
     getInputIndex() - 1);
  leftData[i] = rigthData[i];
  iteratore++;
}

else

{
  buffer_portante_processed.ClockProcess(
     buffer_portante1.getSampleBuffered(
     buffer_portante1.getInputIndex()),
     buffer_portante2.getSampleBuffered(
     buffer_portante2.getInputIndex()));
  buffer_modulante_processed.ClockProcess(
     buffer_modulante1.getSampleBuffered(
     buffer_modulante1.getInputIndex()),
     buffer_modulante2.getSampleBuffered(
     buffer_modulante2.getInputIndex()));
  buffer_portante1.ClockProcess(&rigthData[i]);
  buffer_modulante1.ClockProcess(&leftData[i]);
  buffer_portante2.ClockProcess(&rigthData[i]);
  buffer_modulante2.ClockProcess(&leftData[i]);
```

```
        rigthData[i] = buffer_portante_processed.
            getSampleBuffered(buffer_portante_processed.
            getInputIndex() - 1);
        leftData[i] = rigthData[i];
    }

    if (buffer_portante1.isFull() && buffer_modulante1.
        isFull())
    {
        buffer_portante1.addPadding();
        buffer_portante1.setAudioBuffer(
            convolution_buffer1.process(buffer_portante1.
            getAudioBuffer(), padding, buffer_portante1.
            getOverlap(), ir, ir_length, y));
        lpc.ForwardLinearPrediction(buffer_modulante1.
            getAudioBuffer());
        buffer_portante1.setAudioBuffer(filter.process(
            NumCoeffs, lpc.getCoeffs(), buffer_portante1.
            getAudioBuffer()));
        buffer_portante1.applyRMS();
        buffer_portante1.setAudioBuffer(window.process(
            buffer_portante1.getAudioBuffer()));
    }

    if (buffer_portante2.isFull() && buffer_modulante2.
        isFull())
    {
        buffer_portante2.addPadding();
        buffer_portante2.setAudioBuffer(
            convolution_buffer2.process(buffer_portante2.
            getAudioBuffer(), padding, buffer_portante1.
            getOverlap(),ir, ir_length, y));
        lpc.ForwardLinearPrediction(buffer_modulante2.
            getAudioBuffer());
        buffer_portante2.setAudioBuffer(filter.process(
            NumCoeffs, lpc.getCoeffs(), buffer_portante2.
            getAudioBuffer()));
        buffer_portante2.applyRMS();
        buffer_portante2.setAudioBuffer(window.process(
            buffer_portante2.getAudioBuffer()));
```

```
        }

        if (buffer_portante_processed.isFull() &&
           buffer_modulante_processed.isFull())
        {

        }
      }
   }
}
```

Bibliografia

[1] R. Hall, *Joe Walsh to Frampton: The Story of the Talk Box*, http://www.gibson.com/news-lifestyle/features/en-us/joe-walsh-to-frampton-talk-box-0107-2013.aspx, 2013.

[2] J. Diaz, *The Fate of Auto-Tune*, 2009.

[3] S. Fortner, *TALKBOX 101*, Keyboard, 2011.

[4] J. Aikin, *How Does a Vocoder Work?*, Keyboard, 2000.

[5] A. Farina, *Simultaneous Measurement of Impulse Response and Distortion with a Swept-Sine Technique*, 2000.

[6] G. Fant, *Acoustic Theory of Speech Production*, Mouton De Gruyter, 1970

[7] P. P. Vaidyanathan, *The Theory of Linear Prediction*, Morgan & Claypool, 2008

[8] M. Robinson, *Getting started with JUCE*, PACKT Publishing, 2013.

[9] C. Collomb, *Linear Prediction and Levinson-Durbin Algorithm*, http://www.emptyloop.com/technotes/a%20tutorial%20on%20linear%20prediction%20and%20levinson-durbin.pdf, 2009.